

HOCHSCHULE OSNABRÜCK

UNIVERSITY OF APPLIED SCIENCES

Fakultät Ingenieurwissenschaften und Informatik

Schriftliche Ausarbeitung zum Thema:

Entwicklung eines Themenforums mit hexagonaler Softwarearchitektur

im Rahmen des Moduls
Software-Architektur – Konzepte und Anwendungen
des Studiengangs Informatik – Medieninformatik (B.Sc.)

Verfasser: Lorenzo Battiston (LB)
Matrikelnummer: 919355
E-Mail: lorenzo.battiston@hs-osnabueck.de

Oliver Schlueter (OS)
Matrikelnummer: 914726
E-Mail: oliver.schlueter@hs-osnabueck.de

Dozent: Prof. Dr. Rainer Roosmann

Abgabedatum: 17.02.2023

Inhaltsverzeichnis

1 Einleitung – LB, OS.....	1
1.1 Vorstellung des Themas – LB, OS.....	1
1.2 Ziel der Ausarbeitung – LB, OS	1
1.3 Aufbau der Hausarbeit – LB, OS.....	2
2 Technische Grundlagen – LB	3
3 Konzeption der Softwarearchitektur – LB, OS	4
3.1 Zielsetzung der Architektur für das Themenforum – OS.....	4
3.2 Verwendete Design-Prinzipien und Architektur-Patterns – OS	4
3.2.1 SOLID-Prinzipien – OS	4
3.2.2 Hexagonale Architektur – OS.....	5
3.2.3 Domain Driven Design – LB, OS.....	6
3.3 Aufbau der Themenforum Applikation – LB, OS	6
4 Umsetzung des Domänenmodells – LB.....	7
4.1 Domain Model – LB.....	7
4.2 Domain Services – LB.....	8
4.3 Domain Factories – LB.....	9
5 Umsetzung der Applikationslogik – LB, OS	11
5.1 Use Cases des Themenforums – OS	11
5.2 Definition der Input Ports – OS	12
5.2.1 Input Port Validierung – OS	14
5.2.2 Generische <i>ApplicationResult</i> -Klasse – LB, OS	16
5.3 Definition der Output Ports – LB, OS	17
5.3.1 Repositories – LB, OS.....	17
5.3.2 Authorization-Gateway – OS.....	18
5.4 Implementierung der Application Services – LB	19
6 Implementierung der Infrastruktur-Adapter – OS	21
6.1 Persistente Datenspeicherung – OS.....	21

6.2	Zugriffskontrolle – OS	25
7	Implementierung der Interaktions-Adapter – LB	27
7.1	Umsetzung der REST-Schnittstelle – LB	27
7.1.1	Modellierung der REST-Endpunkte – LB	27
7.1.2	Implementierung der REST-Endpunkte – LB	28
7.1.3	Dokumentation der REST-Endpunkte – LB	29
7.1.4	Testen der REST-Endpunkte – LB.....	30
7.2	Quarkus Qute Webfrontend – LB.....	30
7.2.1	Implementierung der User Interface Endpunkte – LB	31
8	Zusammenfassung und Fazit – LB, OS	33
9	Referenzen.....	35

Eidesstattliche Erklärung

Abbildungsverzeichnis

Abbildung 1: Das DIP bei hexagonaler Architektur [Homb19, S.17]	5
Abbildung 2: Architektur der Themenforum-Applikation (Eigene Darstellung)	6
Abbildung 3 Domain Model - UML Diagramm (Eigene Darstellung)	7
Abbildung 4: Use Cases des Themenforums (Eigene Darstellung)	11

Source-Code Verzeichnis

Snippet 1: <i>VoteService</i> als Domain-Service	9
Snippet 2: <i>SortByDate</i> als Domain-Service	9
Snippet 3: Nutzung des <i>SortByUpvote</i> Services	9
Snippet 4: <i>TopicFactory</i> -Klasse.....	10
Snippet 5: <i>CommentPostUseCase</i>	12
Snippet 6: Command DTO für <i>CommentPostUseCase</i>	13
Snippet 7: <i>GetFilteredPostsUseCase</i>	13
Snippet 8: Query DTO für <i>GetFilteredPostsUseCase</i>	13
Snippet 9: <i>PostFilterParam</i> -Enum	14
Snippet 10: <i>ValidPostFilterParams</i> -Annotation	14
Snippet 11: <i>PostFilterParamsValidator</i>	15
Snippet 12: <i>EnumValueValidator</i>	15
Snippet 13: <i>ApplicationResult</i> -Klasse (Ausschnitt).....	16
Snippet 14: <i>PostRepository</i>	17
Snippet 15: Status-Codes der <i>RepositoryResult</i> -Klasse	17
Snippet 16: <i>AuthorizationGateway</i> (Ausschnitt)	18
Snippet 17: Status Codes der <i>AuthorizationResult</i> -Klasse	18
Snippet 18: <i>GetPostById</i> -Service (Teil 1).....	19
Snippet 19: <i>GetPostById</i> -Service (Teil 2).....	19
Snippet 20: <i>VoteEntityService</i> (Ausschnitt)	20
Snippet 21: <i>DeleteVoteService</i> (Ausschnitt).....	20
Snippet 22: Application Properties zur Hibernate-Konfiguration (Ausschnitt)	21
Snippet 23: <i>UserPersistenceModel</i> (Ausschnitt)	21
Snippet 24: Converter im <i>UserPersistenceModel</i>	22
Snippet 25: Relationen im <i>PostPersistenceModel</i>	22
Snippet 26: ManyToOne Relation im <i>VotePersistenceModel</i>	22
Snippet 27: Bidirektionale Parent-Child Beziehung im <i>CommentPersistenceModel</i>	23

Snippet 28: <i>TopicPersistenceView</i>	23
Snippet 29: Subquery zur Ermittlung der Anzahl von Posts zu einem Topic	23
Snippet 30: Mapping der <i>TopicPersistenceView</i>	24
Snippet 31: <i>TopicPersistenceAdapter</i> (Ausschnitt)	24
Snippet 32: Die Methode <i>searchTopic</i> im <i>TopicPersistenceAdapter</i> (Ausschnitt)	24
Snippet 33: <i>AuthUser</i> -Entity im Authorization-Adapter.....	25
Snippet 34: <i>OwnerOf</i> -Entity im Authorization-Adapter	25
Snippet 35: Die Methode <i>addOwnership</i> im Authorization-Adapter	26
Snippet 36: Die Methode <i>canDeletePost</i> im Authorization-Adapter	26
Snippet 37: JAX-RS Annotationen der <i>PostRessource</i>	28
Snippet 38: Die Methodensignatur <i>createPost</i> der <i>PostRessource</i>	28
Snippet 39: Die Methode <i>createPost</i> (Ausschnitt).....	29
Snippet 40: Überprüfung des <i>ApplicationResult</i> in der <i>createPost</i> -Methode	29
Snippet 41: Rückgabe einer <i>ErrorResponse</i>	29
Snippet 42: REST-Assured Testing (Ausschnitt):.....	30
Snippet 43: <i>GetTopics</i> -Methode des <i>TopicEndpoint</i> (Ausschnitt).....	31
Snippet 44: <i>Topics.html</i> -Template (Ausschnitt).....	32
Snippet 45: Template Extension <i>loggedInUserCanDownvote</i>	32
Snippet 46: Konfiguration der Authentifizierung in den Application Properties	32

Abkürzungsverzeichnis

API	A pplication P rogramming I nterface
DIP	D ependency I nversion P rinciple
DTO	D ata- T ransfer- O bject
HTTP	H yper T ext T ransfer P rotocol
ISP	I nterface S egregation P rinciple
JAX-RS	J akarta R ESTful W eb S ervices
JDBC	J ava DC onnectivity
JPA	J ava P ersistence API
JTA	J ava T ransactional API
LSP	L iskov S ubstitution P rinciple
OCP	O pen- C losed P rinciple
ORM	o bject- r elational M apping
REST	R epresentational S tate T ransfer
ROA	R essource O riented A rchitecture
SOC	S eparation o f C oncerns
SQL	S tructured Q uery L anguage
SRP	S ingle R esponsibility P rinciple
SSR	S erver S ide R endered
UI	U ser I nterface
URI	U niform R essource I dentifier

1 Einleitung – LB, OS

REST-APIs (Representational State Transfer) sind ein wichtiger Teil der Kommunikation im Internet. Sie ermöglichen es Anwendungen, über ein Netzwerk miteinander zu kommunizieren und Daten auszutauschen. Durch ihre Einfachheit, Flexibilität und Übertragbarkeit sind REST-APIs ein unverzichtbares Instrument für die Entwicklung von Microservices, Single Page Applications und Webanwendungen. Ein weiterer Vorteil von REST-APIs ist, dass sie verständlich und leicht zu verwenden sind, da sie auf einer einfachen HTTP-basierten (Hyper Text Transfer Protocol) Schnittstelle beruhen. Die Endpunkte und Ressourcen, auf die über eine REST-API zugegriffen werden kann, sind in einer logischen und nachvollziehbaren Struktur organisiert, die es Entwicklern ermöglicht, die API schnell zu verstehen und effektiv zu nutzen.

In diesem Zusammenhang spielt die Wahl eines richtigen Architekturnusters eine entscheidende Rolle für die Performance, Skalierbarkeit und Wartbarkeit einer REST-API. Deshalb soll im Rahmen dieser Arbeit eine solide Softwarearchitektur erarbeitet werden, auf Basis derer anschließend eine Anwendung implementiert wird.

1.1 Vorstellung des Themas – LB, OS

Als Anwendung soll eine Themenforum umgesetzt werden, welches eine Plattform bietet, Beiträge zu bestimmten Themen zu erstellen. Dazu können sich Internetnutzer auf der Plattform registrieren, um Mitglied zu werden. Die Beiträge im Forum können von Mitgliedern erstellt, kommentiert und bewertet werden. Zudem können registrierte Nutzer auf Kommentare antworten und diese ebenfalls bewerten.

Mitglieder haben die Möglichkeit, das Forum nach bestimmten Themen zu durchsuchen und Posts sowie Kommentare nach Aktualität und Beliebtheit sortiert aufzulisten. Auf einer privaten Profilübersicht werden die veröffentlichten Inhalte eines Mitglieds aufgelistet. Nicht-registrierte Nutzer können sich lediglich Beiträge und Themen ansehen, jedoch nicht mit ihnen interagieren.

1.2 Ziel der Ausarbeitung – LB, OS

Ziel der Ausarbeitung ist es das zuvor beschriebene Themenforum als REST-API umzusetzen. Die Musskriterien aus dem Projektvorschlag (siehe Anhang) dienen dabei als Pflichtenheft und sollen nach Möglichkeit alle umgesetzt werden. Außerdem sollen die Best-Practices und Richtlinien in Bezug auf die Entwicklung von REST-APIs berücksichtigt werden. Ein besonderer Fokus liegt auf der Erarbeitung einer soliden Anwendungsarchitektur. Dazu werden zunächst die theoretischen Grundlagen beschrieben. Anschließend wird anhand dieser Softwarearchitektur die Anwendung implementiert. Dabei wird konkret gezeigt, wie die einzelnen Architekturprinzipien und -Muster bei der Programmierung der Anwendung zum tragen kommen.

Außerdem wird neben der REST-API auch ein Server Side Rendered UI (User Interface) für die Anwendung implementiert. Das Forum kann dadurch in einem sinnvollen Kontext genutzt und getestet werden.

1.3 Aufbau der Hausarbeit – LB, OS

Zu Beginn werden die technischen Grundlagen für die Entwicklung einer REST-API beschrieben. Dazu wird die Funktion und Struktur des Java-Frameworks *Quarkus* erklärt.

Im Hauptteil des Projektberichts wird die Implementierung des Themenforums beschrieben. In *Kapitel 3* wird dafür zunächst eine geeignete Softwarearchitektur erarbeitet. In den darauffolgenden Kapiteln wird die Umsetzung der einzelnen Softwarekomponenten näher betrachtet. Begonnen wird in den *Kapiteln 4* und *5* mit der Implementierung der Domain- und Applikationslogik. Anschließend wird in *Kapitel 6* die Umsetzung der Datenhaltung und Zugriffskontrolle näher beschrieben. In *Kapitel 7* wird schließlich die Umsetzung der REST-API und des Webfrondens erklärt.

Abschließend erfolgt eine kritische Betrachtung der entwickelten Anwendung. Dabei soll analysiert werden, ob die gesetzten Ziele erreicht wurden. Insbesondere soll hier betrachtet werden, wie sich die gewählte Softwarearchitektur auf die Umsetzung der Anwendung ausgewirkt hat. Weiterhin werden offengebliebene Fragen und Verbesserungsansätze formuliert.

2 Technische Grundlagen – LB

Im Folgenden werden die technischen Grundlagen erläutert, die für die Umsetzung der Anwendung genutzt wurden. Dazu wird zunächst das Quarkus-Framework näher beschrieben.

Quarkus ist ein full-stack Java-Framework für die Entwicklung leichtgewichtiger Anwendungen. Es zeichnet sich durch seinen geringen Speicherverbrauch und schnelle Startzeit aus und unterstützt eine einfache Containerisierung der Anwendungen durch Docker und Kubernetes. Dadurch können Quarkus-Anwendungen sehr einfach in einer Cloud-Umgebung eingesetzt werden. Neben der klassischen imperativen Programmierung unterstützt Quarkus auch die reaktive Programmierung.

Als Build- und Dependency-Management-Tool wird Apache Maven 3 verwendet. Mit diesem Tool lassen sich einfach Java-Bibliotheken und Plugins in das Quarkusprojekt integrieren. Über Scopes lässt sich im Maven-Projekt konfigurieren, welche Dependencies wann genutzt werden. So können beispielsweise bestimmte Dependencies nur beim Testen der Anwendung eingefügt werden.

Quarkus stützt die Implementierung der integrierten Bibliotheken auf etablierte Standards in der Entwicklung von Java-Anwendungen. So kann zum Beispiel der Dependency Injection Standard CDI (Context Dependency Injection) problemlos in Quarkusanwendung genutzt werden.

Ebenfalls sind die Annotationen der Java Persistence API (JPA) und der Java Transactional API (JTA) verfügbar, um die Datenpersistierung zu erleichtern. Hier wird als konkrete Implementierung dieser Standards das Hibernate-Framework genutzt. Dabei handelt es sich um einen Objekt-Relationalen Mapper (ORM), welcher das Persistieren von Java-Objekten in einer Relationalen Datenbank erleichtert. Ergänzend wird hier das Blaze-Persistence-Framework genutzt, um das Formulieren von Datenbankabfragen zu erleichtern. [Quar23a][Pete22]

Für die Implementierung von REST-Endpunkten, wird die Jakarta RESTful Web Services (JAX-RS) Spezifikation als Basis genutzt und von der Quarkus-Resteasy Extension implementiert. JSON-B übernimmt dabei die Serialisierung und Deserialisierung von JSON-Dateien in Java-Objekte. [Quar23a]

Mit Hilfe der QuTe-Templating-Engine können Server Side Rendered (SSR) Webfrontends umgesetzt werden. Dieses Frontend-Framework ist speziell für Quarkus-Anwendungen konzipiert und bietet die Möglichkeit Java-Code in HTML-Seiten einfließen zu lassen. Dazu wird eine einfache Syntax verwendet, die Kontrollflüsse, wie Schleifen und Konditionsabfragen bereitstellt. [Quar23b]

Zum Testen der REST-Endpunkte wird „Rest-Assured“ verwendet. Die Dokumentation der REST-API erfolgt über den OpenAPI-Standard, der ebenfalls von Quarkus unterstützt wird. Mit Hilfe von Annotationen und Konfigurationsparametern in der Quarkus-Anwendung können über das Swagger-UI die Endpunkte strukturiert dargestellt und von API-Nutzern getestet werden.

3 Konzeption der Softwarearchitektur – LB, OS

Vor der Umsetzung der REST-Applikation soll zunächst ein geeignetes Softwaredesign erarbeitet werden. Dazu wird in *Abschnitt 3.1* zunächst auf die Zielsetzung in Bezug auf die Architektur eingegangen. In *Abschnitt 3.2* werden verschiedene Designprinzipien und Architekturmuster beschrieben, die in die Umsetzung der Anwendung einfließen. Abschließend wird in *Abschnitt 3.3* die der Umsetzung zugrundeliegende Architektur dargestellt.

3.1 Zielsetzung der Architektur für das Themenforum – OS

Ziel der Architektur für das Themenforum ist, die Softwarekomponenten so zu isolieren, dass die Abhängigkeiten untereinander möglichst gering sind. Die Kommunikation zwischen den einzelnen Komponenten soll dadurch für Entwickler einfach nachzuvollziehen sein.

Dadurch soll gewährleistet werden, dass die Applikation gut test- und wartbar ist. Darüber hinaus soll sie auch flexibel gegenüber Anpassungswünschen sein. Änderungen können hierbei der Austausch von Infrastruktur-Technologien, wie zum Beispiel des Datenbanksystems sein. Aber auch zusätzlichen Schnittstelle zur Interaktion sollen einfach hinzugefügt werden können.

Damit Wartbarkeit und Erweiterbarkeit gewährleistet werden, sollte die Domänen- und Applikationslogik des Themenforums im Fokus stehen. Sie soll die geringsten Abhängigkeiten zu anderen Komponenten haben und möglichst unabhängig von den technischen Details eingesetzter Frameworks und Bibliotheken sein.

Nicht selten wird Software technologiegetrieben konzipiert und umgesetzt. Oft wird dabei eine klassische Drei-Schichten-Architektur verwendet, die in Präsentations-, Applikations- und Datenzugriffsschicht aufgeteilt ist. Häufig kommt es bei dieser Schichtenarchitektur dazu, dass der Datenbank eine übergeordnete Rolle gegeben wird. Dadurch wird die Applikationslogik sehr stark an die Datenzugriffsschicht gekoppelt oder die Datenbank setzt sogar Teile der Geschäftslogik um. Dadurch können sich indirekte Abhängigkeiten bis in die Präsentationsschicht bilden. [Homb19, S.2]

Dies führt dazu, dass die Applikationslogik im gesamten System verteilt wird. Testen, Warten oder Austauschen einzelner Komponenten ist dann mit steigender Komplexität der Anwendung immer aufwändiger. Damit dies bereits zu Beginn der Entwicklung verhindert wird, lassen sich einige Designprinzipien und Entwurfsmuster anwenden.

3.2 Verwendete Design-Prinzipien und Architektur-Patterns – OS

Um die beschriebene Zielsetzung zu erreichen, werden einige Designprinzipien und Patterns näher untersucht, die das Entwickeln von Software mit möglichst robuster Trennung der fachlichen Details von technologischen Details vereinfachen.

3.2.1 SOLID-Prinzipien – OS

Um eine gute Softwarearchitektur zu entwickeln, lassen sich die fünf SOLID-Prinzipien von Robert C. Martin als Richtlinie heranziehen [Mart18]:

- Das **SRP** (Single Responsibility Principle) besagt, dass Klassen sauber nach ihren Zuständigkeiten getrennt werden sollten.
- Das **OCP** (Open-Closed Principle) empfiehlt sinnvolle Abstraktionsebenen in Form von Interfaces oder abstrakten Klassen, um Klassen bei Änderungen von Anforderungen nicht modifizieren zu müssen.
- Das **LSP** (Liskov Substitution Principle) verlangt einen sinnvollen Einsatz von Polymorphie in der objektorientierten Programmierung.
- Das **ISP** (Interface Segregation Principle) fordert spezifisch definierte Schnittstellen, damit die abhängigen Klassen nicht durch unnötige Komplexität belastet werden.
- Das **DIP** (Dependency Inversion Principle) betont, dass Abhängigkeiten innerhalb der Software in Richtung höherer Abstraktion bewegt werden sollten.

Diese Prinzipien werden an unterschiedlichen Stellen der Konzeption und Umsetzung des Themenforums aufgegriffen.

3.2.2 Hexagonale Architektur – OS

Bei dem hexagonalen Architekturansatz stellt die Anwendungslogik die zentrale Schnittstelle dar. Diese wird als "Hexagon" bezeichnet. Alistair Cockburn verdeutlicht in seinem Blog-Artikel zum "Ports & Adapters"-Pattern, dass die Applikationslogik nicht durch ihre Akteure oder die Infrastruktur definiert sein sollte: „*Create your application to work without either a UI or a database so you can [...] work when the database becomes unavailable, and link applications together without any user involvement.*“ [vgl. Cock08]

Die Definition einer API um den Kern der Anwendung garantiert, dass die Anwendungslogik von den mit ihr verbundenen Systemen unabhängig bleibt. Das Application-Hexagon regelt nur die Interaktion mit externen Systemen und kann sowohl als steuernde als auch als gesteuerte Komponente agieren. Die API der Applikationslogik wird durch Interfaces – den sogenannten „Ports“ – definiert. Diese werden in Input- und Output-Ports unterteilt. Die Anbindung externer Systeme über Ports wird durch „Adapter“ realisiert. Die Hexagonale Architektur unterscheidet zwischen primären Adapters, welche die Anwendung steuern – sogenannte *driving Adapters* – und sekundären Adapters, welche durch die Anwendung gesteuert werden, den *driven Adapters*. [Cock08][Hexa23]

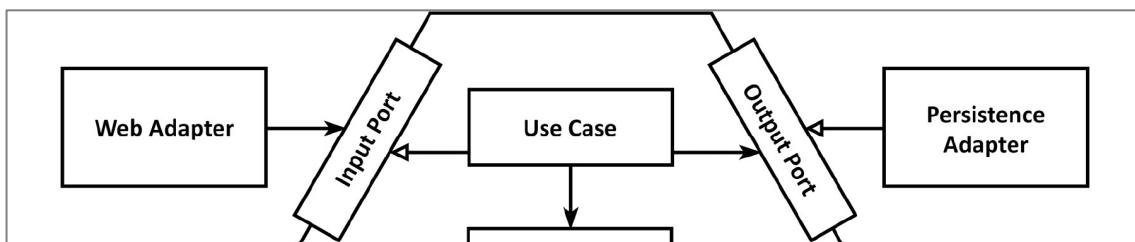


Abbildung 1: Das DIP bei hexagonaler Architektur [Homb19, S.17]

Die Input Ports werden von den Primäradaptoren, beispielsweise einem REST-Endpunkt oder einem User Interface, verwendet und durch die Applikation implementiert. Die Output Ports wiederum werden gemäß dem DIP von der Applikation verwendet und durch die Sekundäradapter implementiert, beispielsweise einem ORM zur Anbindung einer Datenbank. Dadurch wird sichergestellt, dass alle Quellcodeabhängigkeiten in Richtung der Applikationslogik zeigen.

3.2.3 Domain Driven Design – LB, OS

Die Hexagonale Architektur lässt offen, wie das Application-Hexagon konkret implementiert ist. Das Domain Driven Design (DDD) ist ein geeignetes Prinzip, um die Businesslogik an dieser Stelle zu modellieren. Domain-Driven Design ist ein Ansatz zur Softwareentwicklung, bei dem die Businesslogik einer Domäne im Mittelpunkt steht. Mit Hilfe des DDD lässt sich ein innerer Layer im Application Hexagon bilden: das Domain Model.

Das Domain-Model repräsentiert domänenspezifische Regeln und Prozesse. Application Services implementieren je einen Use Case des Application-Hexagons und steuern den Ablauf der Applikation. Dazu nutzen sie Domain Services, Entitäten und andere Elemente aus dem Domänen-Modell, um die Businesslogik umzusetzen. Außerdem steuern sie die Output Ports der Anwendung, um Prozesse in externen Adapters anzustoßen. [Mart18, S.204][Vern13, S.521]

3.3 Aufbau der Themenforum Applikation – LB, OS

Die folgende Abbildung 2 zeigt den Aufbau der Architektur für das Themenforum. Als Basis wird die zuvor beschriebene hexagonale Architektur verwendet. Die Input Ports repräsentieren je einen Anwendungsfall (Use Case) für das Themenforum. Sie werden von verschiedenen REST-Ressource-Klassen und UI-Endpunkten verwendet. Als Output Ports werden Repositories für die verschiedenen Entitäten der Applikation und ein Authorization-Gateway zur Zugriffssteuerung definiert.

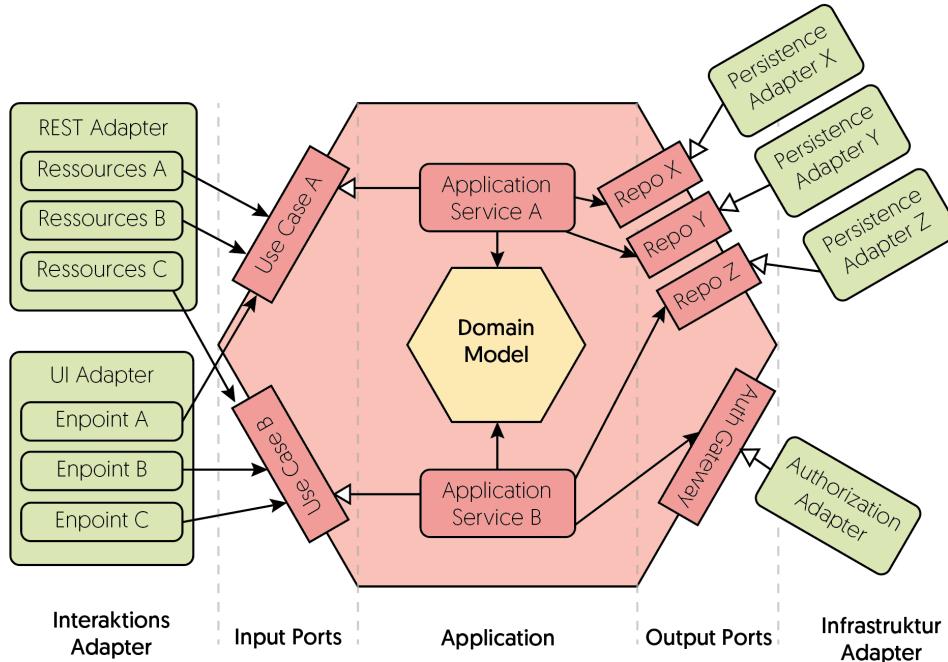


Abbildung 2: Architektur der Themenforum-Applikation (Eigene Darstellung)

4 Umsetzung des Domänenmodells – LB

Im Folgenden wird die Modellierung der Themenforum-Domäne beschrieben. Dabei wird erläutert, welche Elemente aus dem Domain Driven Design eingesetzt wurden, und wie diese programmatisch umgesetzt sind. Zu Beginn wird ein Überblick über das Domain Model gegeben und die Modellierung der Domain Entities beschrieben. Anschließend wird die Implementierung der Domain Services erklärt. Abschließend wird der Einsatz von Domain Factories in der Applikation erläutert.

4.1 Domain Model – LB

Zunächst werden die Entitäten der Domäne identifiziert und als Java-Klassen angelegt. Entitäten im DDD zeichnen sich dadurch aus, dass sie entweder durch ein bestimmtes Attribut oder die Kombination von Attributen eindeutig identifizierbar sind. Entitäten mit der gleichen Identität müssen vom System als dieselbe Entität behandelt werden. [AvMa06]

Bei der Implementierung des Domain Models wurden ID-Attribute genutzt, um dies sicherzustellen. Hier wurde bewusst darauf verzichtet, die von der Datenbank generierten IDs zu verwenden, um eine Abhängigkeit zu dieser zu verhindern. Deshalb werden die IDs von der Domäne selbst generiert. Nun lassen sich die einzelnen Entitäten des Themenforums implementieren.

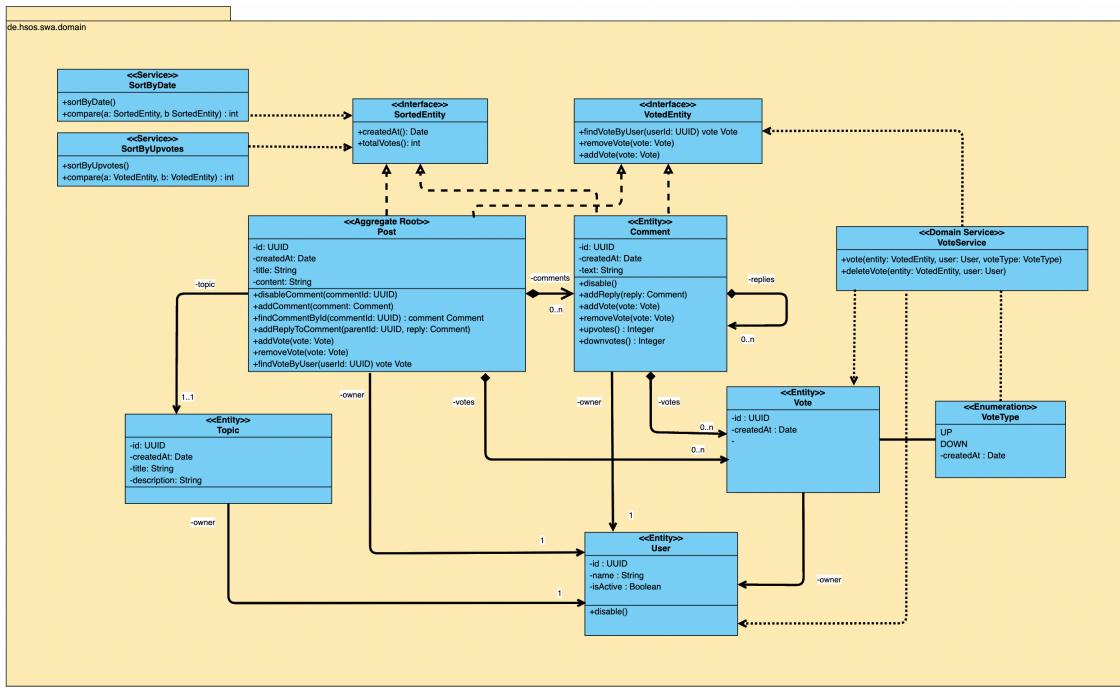


Abbildung 3 Domain Model - UML Diagramm (Eigene Darstellung)

Die `User`-Klasse repräsentiert einen Nutzer des Forums und erhält neben einer ID einen Username. Die Klasse bietet weiterhin die Möglichkeit den User-Status auf inaktiv zu setzen.

Die `Post`-Klasse wird als Aggregate Root modelliert. Dies ist ein Konzept aus dem DDD, bei dem eine Entität weitere Entitäten und Value Objects aggregiert. Der Zugriff auf die aggregierten Objekte erfolgt dann über diese Aggregate Root. Ein Post setzt sich aus einem Titel, Inhalt und Erstelldatum zusammen. Weiterhin wird in dieser Klasse eine

Beziehung zur User-Entität und einem Topic hergestellt. Kommentare und Bewertungen (Up-/Downvotes) zu einem Post werden in einer Liste von Comment- beziehungsweise Vote-Entitäten aggregiert. Wie zuvor beschrieben erfolgt das Hinzufügen und Entfernen von Kommentaren und Bewertungen durch Methoden der Post-Entität.

Bei den Kommentaren muss ein rekursives Verhalten modelliert werden, da im Forum auf Kommentare geantwortet werden kann. Dazu speichert ein Kommentar seinen übergeordneten Kommentar als *parentComment* und die Antworten in einer Liste von weiteren Kommentaren. Neben diesen Informationen werden wieder das Erstelldatum und der zugehörige User, sowie die Bewertung gespeichert. Die *Comment*-Klasse bietet Methoden zur Interaktion an, die – gemäß dem DDD – nur von der Post-Aggregate Root verwendet werden.

Die Vote-Entität speichert abgegebene Bewertungen im Forum. Dazu wird neben dem Nutzer, der die Bewertung abgegeben hat, auch das Erstelldatum und der Typ der Bewertung gespeichert. Dieser wird als Enumeration implementiert und kann die Werte „UP“ und „DOWN“ annehmen.

Das Topic setzt sich aus einem Titel und einer kurzen Beschreibung zusammen. Außerdem wird hier der Nutzer referenziert, der das Thema erstellt hat.

4.2 Domain Services – LB

Das Verhalten des Domänenmodells im Domain Driven Design wird über die öffentlichen Methoden der jeweiligen Entitäten abgebildet. Jedoch gibt es einige Verhaltensweisen in der Businesslogik, die sich nicht einer konkreten Entität zuordnen lassen. Um nicht das Single-Responsibility-Prinzip zu verletzen, werden dazu Domain Services implementiert. Diese Services bilden verschiedenen Verhaltensweisen und Funktionen der Domäne ab und können dazu auf verschiedene Entitäten zurückgreifen. Im Modell des Themenforums wird ein Vote-Service angelegt, der die nötige Businesslogik zum Bewerten eines Posts oder Kommentars beinhaltet.

```
@RequestScoped
public class VoteService {
    public Optional<Vote> vote(VotedEntity entity, User user,
        VoteType voteType) {
        // User kann seine eigenen Posts/Comments nicht voten
        if (entity.getUser().getId().equals(user.getId())) {
            return Optional.empty();
        }
        // Fall 1: Neuer Vote
        Optional<Vote> optionalVote =
            entity.findVoteByUser(user.getId());
        if (optionalVote.isEmpty()) {
            Vote vote = new Vote(user, voteType);
            entity.addVote(vote);
            return Optional.of(vote);
        }
        // Fall 2: Änderung eines bestehenden Votes
        Vote oldVote = optionalVote.get();
        if (voteType != oldVote.getVoteType()) {
            entity.removeVote(oldVote);
            Vote newVote = new Vote(user, voteType);
            entity.addVote(newVote);
            return Optional.of(newVote);
        }
    }
}
```

```

        return Optional.empty();
    }

    public Optional<Vote> deleteVote(VotedEntity entity, User user) {
        Optional<Vote> optionalVote = entity.findVote-
ByUser(user.getId());
        if (optionalVote.isEmpty()) {
            return Optional.empty();
        }
        entity.removeVote(optionalVote.get());
        return optionalVote;
    }
}

```

Snippet 1: VoteService als Domain-Service

An dieser Stelle kann gemäß LSP sinnvoll Polymorphie eingesetzt werden. Die *Post*- und *Comment*-Klassen implementieren dazu das Interface *VotedEntity*. Dieses Interface deklariert Methoden, die für die Bewertung einer Entität nötig sind. Der Vote-Service nimmt als Parameter ein Objekt dieses Interface entgegen und führt die nötigen Schritte zur Bewertung der entsprechenden Entität aus, ohne sich auf eine konkrete Klasse zu beschränken.

Ein ähnlicher Ansatz wird beim Sortieren der Entitäten verwendet. Hier wird wieder ein Interface *SortedEntity* angelegt. Im Themenforum sollen Beiträge und Kommentare nach Aktualität und der Bewertung sortiert werden. Deshalb gibt das *SortedEntity*-Interface zwei Getter-Methoden für diese Parameter vor.

```

public class SortByDate<T extends SortedEntity> implements
Comparator<T> {
    public int compare(T a, T b) {
        int result = b.getCreatedAt().compareTo(a.getCreatedAt());
        if(result == 0) {
            result = b.getTotalVotes().compareTo(a.getTotalVotes());
        }
        return result;
    }
}

```

Snippet 2: SortByDate als Domain-Service

Die *SortByDate*- und *SortByUpvote*-Services werden mit einem generischen Typ instanziert, der das *SortedEntity*-Interface erweitert. Außerdem wird das funktionale *Comparator*-Interface der Java-Utils implementiert. Nun kann die compare-Methode des Interfaces mit dem generischen Typ implementiert werden. Dabei kann der Service auf die beiden Getter-Methoden des *SortedEntity*-Interfaces zurückgreifen. Wird nun eine sortierte Liste der Posts oder Kommentare benötigt, kann diese mit den beiden Sorting-Services sortiert werden.

```
posts.sort(new SortByUpvotes<Post>());
```

Snippet 3: Nutzung des SortByUpvote Services

4.3 Domain Factories – LB

Entitäten im Domain Driven Design können schnell eine hohe Komplexität erreichen. Möchte man in der Anwendung eine solche Entität erstellen, benötigt man Zugriff auf den Konstruktor und Wissen darüber, wie das Objekt konkret erstellt wird. Dadurch wird die interne Struktur der Entität für andere Softwarekomponenten sichtbar gemacht. Bei

Änderungen in der Instanziierung, müssen alle Aufrufe des Konstruktors angepasst werden.

Um diese Abhängigkeiten aufzulösen, und das Erstellen komplexer Objekte nach außen zu kapseln, wird im DDD das Factory-Pattern verwendet. Dies wird mit Hilfe einer Klasse umgesetzt, die den komplexen Erstellprozess eines Objekts vornimmt und über eine Methode für andere Komponenten bereitstellt. In Java wird dies über eine statische Methode implementiert, welche die benötigten Parameter entgegennimmt, um das Objekt zu erstellen. Intern wird nun der Konstruktor des Objekts aufgerufen. Fehler bei der Erstellung können nun ebenfalls in der Factory behandelt werden. Für das Domänenmodell des Themenforums wurden für *Posts*, *Comments*, *Users* und *Topics* Factories erstellt. Die Kapselung des Konstruktor Aufrufs erscheint hier wenig vorteilhaft, da die Methodenparameter intern an den Konstruktor übergeben werden. Allerdings hat die Factory den weiteren Vorteil, dass die Entitäten direkt beim Erstellen validiert werden und dadurch sichergestellt wird, dass nur gültige Objekte im Domänenmodell existieren.

```
public class TopicFactory {  
    public static Topic createTopic(String title,  
                                    String description, User user) {  
        ValidatorFactory factory =  
            Validation.buildDefaultValidatorFactory();  
        Validator validator = factory.getValidator();  
        LocalDateTime createdAt = LocalDateTime.now();  
        Set<ConstraintViolation<Topic>> violations  
        = validator.validate(new Topic(title, description, createdAt, user));  
  
        if (!violations.isEmpty()) {  
            throw new ConstraintViolationException(violations);  
        }  
        return new Topic(title, description, createdAt, user);  
    }  
}
```

Snippet 4: *TopicFactory*-Klasse

5 Umsetzung der Applikationslogik – LB, OS

In diesem Kapitel wird in *Abschnitt 5.1* zunächst ein Überblick über die verschiedenen Anwendungsfälle des Themenforums gegeben. Anschließend wird in *Abschnitt 5.2* die Definition der Use Cases als Interface durch Beispiele genauer betrachtet. In diesem Zusammenhang werden auch auf die Input-Validierung und die Verwendung einer generischen Result-Klasse eingegangen. In *Abschnitt 5.3* werden die Output Ports der Applikation genauer betrachtet. Zum Ende dieses Kapitels wird in *Abschnitt 5.40* schließlich die Implementierung der Use Cases durch Application Services beschrieben.

Die Paketstruktur in denen die Komponenten des Application-Hexagons untergebracht werden, besteht aus einem *input*-, *output*- und *service*- Package. Neben diesen gibt es ein *annotations*-Package in denen Marker-Annotationen für technische Begriffe der hexagonalen Architektur definiert sind. Diese dienen der besseren Orientierung im Quellcode. Somit ergibt sich neben dem domain-Paket aus dem vorherigen Kapitel das zweite Paket auf erster Ebene: *application*.

5.1 Use Cases des Themenforums – OS

Die verschiedenen Anwendungsfälle des Themenforums werden in Abbildung 4: Use Cases des Themenforums *Abbildung 4* dargestellt. Diese leiten sich aus den funktionalen Anforderungen an das Projekt ab (siehe Anhang).

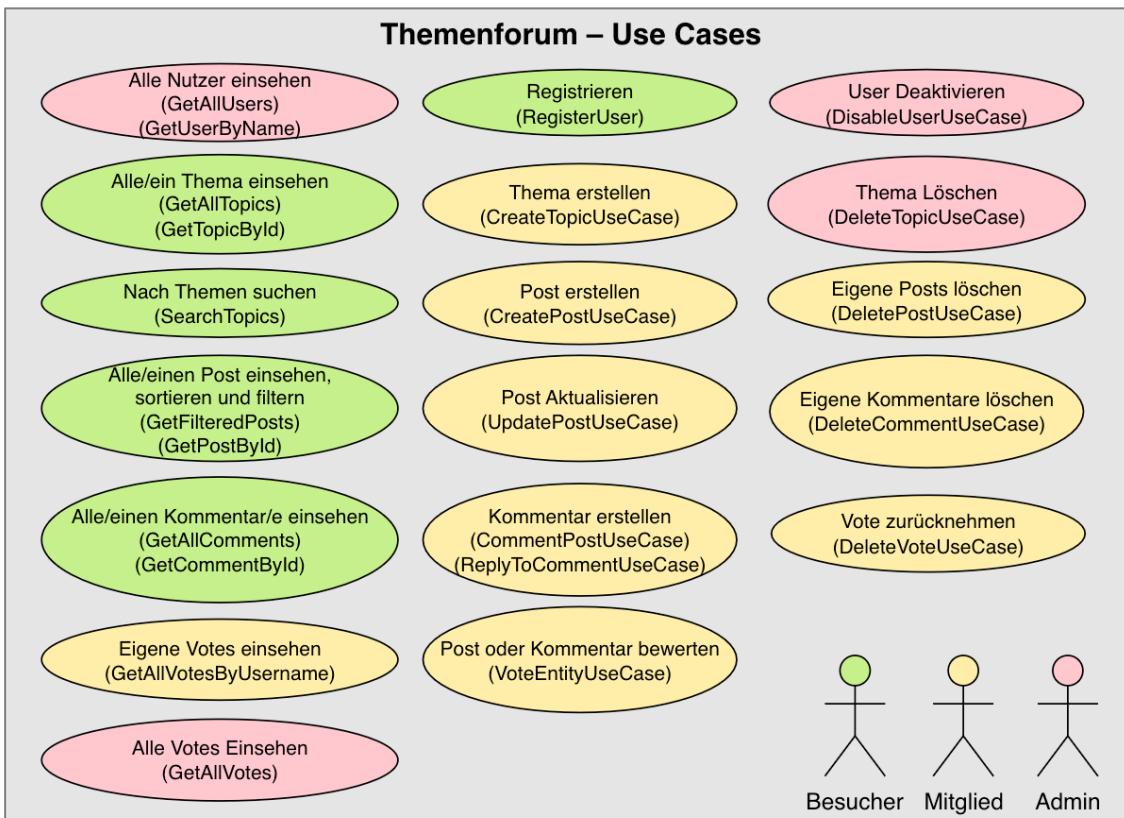


Abbildung 4: Use Cases des Themenforums (Eigene Darstellung)

Das Diagramm stellt die Use Cases für einen Themenforum-Nutzer in den Rollen Besucher, Mitglied und Admin dar. Um das Diagramm übersichtlicher zu gestalten, wurde alternativ zu den Verbindungskanten von den Akteuren zu den einzelnen Use Cases mit Farben gearbeitet, welche das Berechtigungslevel darstellen. Zum Beispiel darf der Admin die Votes aller Mitglieder abrufen, während ein Mitglied ausschließlich seine eigenen Votes einsehen kann. Die Anmeldung eines Nutzers wird hier nicht als Use Case aufgeführt, da die Authentifizierung des Nutzers nicht Teil der Business-Logik ist, sondern durch die primären Adapter umgesetzt werden soll.

Für jeden ermittelten Use Case wird im Sinne des Interface Segregation Principle ein Input Port als Schnittstelle definiert. Dadurch können in den Interactionsadapters genau die benötigten Use Cases injiziert werden. Damit werden unnötige Abhängigkeiten der Adapter zu nicht benötigten Use Cases verhindert. Im Folgenden wird der Aufbau der Use Cases genauer betrachtet.

5.2 Definition der Input Ports – OS

Ein Use Case ist dafür zuständig, die Anfrage aus dem Adapter entgegenzunehmen und auf Basis der Business Logik und unter Verwendung der Output Ports zu verarbeiten, sowie ein Ergebnis an den anfragenden Adapter zurückzugeben.

Ebenfalls wird im Input Port die Validierung der Eingabe mit Hilfe der Java Bean Validation durchgeführt. Die Verantwortlichkeit der Validierung liegt damit bei den Input Ports. Dadurch wird sichergestellt, dass ausschließlich gültige Eingaben in den Application Services, die die einzelnen Use Cases implementieren, verarbeitet werden. Es ist notwendig, dass die Validierung im Application-Hexagon erfolgt, da nicht vorausgesetzt werden kann, dass diese bereits in externen Adapter durchgeführt wurde.

Jeder Use Case ist nach einem einheitlichen Schema aufgebaut. In dem folgenden *Snippet 5* ist das *CommentPostUseCase*-Interface dargestellt.

```
package de.hsos.swa.application.input.command;
...
@InputPort
public interface CommentPostUseCase {
    ApplicationResult<Comment> commentPost(
        @Valid CommentPostCommand command, String requestingUser);
}
```

Snippet 5: CommentPostUseCase

Das Interface stellt eine Methode zur Verfügung, die mit der entsprechenden Aktion des Use Cases benannt ist. Als Parameter für diese Methode ist ein zugehöriges Data Transfer Object (DTO) definiert, welches die Daten der Anfrage kapselt. Diese DTOs sind mit dem Namen des Use Cases und dem Postfix „Command“ gekennzeichnet. Durch die Annotation `@Valid` wird sichergestellt, dass bei Aufruf des Use Cases die Bean Validation angestoßen wird und entsprechende `ConstraintViolationExceptions` geworfen werden. Die Verantwortlichkeit, diese abzufangen und in passende Error-Messages zu mappen liegt bei dem anfragenden Adapter.

Der zweite Parameter `requestingUser` beinhaltet den Namen des anfragenden Nutzers. Dieser wird im betrachteten Use Case (*Snippet 5*) durch den Application Service an das Authorisierungs-Gatetway weitergegeben, um die Zugehörigkeit des Kommentars zum Nutzer für spätere Anfragen festzuhalten.

Die Rückgabe an den Adapter wird ebenfalls vereinheitlicht und erfolgt durch die generische Klasse *ApplicationResult*. Diese wird im *Unterabschnitt 5.2.2* näher betrachtet.

Im Folgenden *Snippet 6* ist das für den Use Case definierte DTO zu sehen.

```
package de.hsos.swa.application.input.dto.in;
...
@InputPortRequest
public record CommentPostCommand(
    @ValidId String postId,
    @ValidCommentText String commentText
) {}
```

Snippet 6: Command DTO für *CommentPostUseCase*

Alle DTOs der Input Ports sind als *Record* definiert. Die Variablen des DTOs werden mit passenden Annotationen versehen und von spezifischen *Validator*-Klassen validiert. Diese werden im nächsten Abschnitt erklärt.

Für eine übersichtliche Struktur werden die Use Cases in *commands* und *queries* unterteilt. Bei den Queries handelt es sich um „Read-Only“-Use Cases, welche den Application-State nicht verändern. Die Application Services, die diese Queries implementieren, sind ähnlich aufgebaut und nutzen die Repositories, um die angefragten Ressourcen an die Adapter weiterzureichen.

```
package de.hsos.swa.application.input.query;
...
@InputPort
public interface GetFilteredPostsUseCase {
    ApplicationResult<List<Post>> getFilteredPosts(
        @Valid GetFilteredPostQuery query);
}
```

Snippet 7: *GetFilteredPostsUseCase*

Snippet 7 zeigt die Definition der *GetFilteredPosts*-Query. Die Methode *getFilteredPosts* benötigt neben dem Query-DTO keinen Nutzernamen für die Anfrage, da die Posts auch von nicht-registrierten Besuchern des Themenforums abgerufen werden können. Dementsprechend ist keine Autorisierung der Abfrage im Use Case nötig.

```
package de.hsos.swa.application.input.dto.in;
...
@InputPortRequest
public record GetFilteredPostQuery(
    @ValidPostFilterParams
    Map<PostFilterParams, Object> filterParams,
    @NotNull
    Boolean includeComments,
    @ValidEnumValue(enumClass = SortingParams.class)
    String sortingParams,
    @ValidEnumValue(enumClass = OrderParams.class)
    String orderParams
) {}
```

Snippet 8: Query DTO für *GetFilteredPostsUseCase*

Im *Snippet 8* ist das DTO für die Abfrage der gefilterten und sortierten Posts mit den Annotationen zur Validierung zu sehen. Die Implementierung der Validierungslogik wird im folgenden Unterabschnitt näher betrachtet.

5.2.1 Input Port Validierung – OS

Das betrachtete DTO nimmt eine Map von Filter-Parametern entgegen. Die zugelassenen Parameter zur Filterung der Posts werden durch einen Enumeration-Typ vorgegeben (siehe *Snippet 9*).

```
package de.hsoc.swa.application.service.query.params;
...
public enum PostFilterParams {
    USERNAME(String.class),
    USERID(UUID.class),
    DATE_FROM(LocalDateTime.class),
    DATE_TO(LocalDateTime.class),
    TOPIC(String.class),
    TOPICID(UUID.class);

    private final Class<?> paramType;

    PostFilterParams(Class<?> paramType) {
        this.paramType = paramType;
    }

    public Class<?> getParamType() {
        return paramType;
    }
}
```

Snippet 9: PostFilterParam-Enum

Jeder Konstante der Enumeration ist eine Klasse als Typ zugeordnet. Das Enum enthält eine Methode, um die zugeordnete Klasse der Enum-Konstante zurückzugeben. Auf diese wird innerhalb der Validierung zurückgegriffen.

Zur Validierung der Map wird die in *Snippet 10* abgebildete Annotation *ValidPostFilterParams* definiert.

```
package de.hsoc.swa.application.input.validation.constraints;
...
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PostFilterParamsValidator.class)
public @interface ValidPostFilterParams {
    String message() default "Invalid PostFilterParams";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Snippet 10: ValidPostFilterParams-Annotation

Die *@Target*-Annotation legt fest, dass diese nur auf Felder angewendet werden kann. Durch die *@Retention*-Annotation wird festgelegt, dass die Annotation zur Ausführungszeit erhalten bleibt. In der *@Constraint*-Annotation wird angegeben, dass die Überprüfung der Constraints durch die Klasse *PostFilterParamsValidator* durchgeführt werden soll. In dem Annotation-Interface *ValidPostFilterParams* wird zudem die bei Verletzung des Constraints standardmäßig hinterlegte Nachricht „*Invalid PostFilterParams*“ festgelegt. [Fere22]

In *Snippet 11* ist die Implementierung der validierenden Klasse abgebildet. In dieser wird die Map überprüft.

```
package de.hsoc.swa.application.input.validation.validator;
/...
public class PostFilterParamsValidator implements
ConstraintValidator<ValidPostFilterParams, Map<PostFilterParams, Object>> {
    private String message;

    @Override
    public void initialize(ValidPostFilterParams constraintAnnotation) {
        message = constraintAnnotation.message();
    }

    @Override
    public boolean isValid(Map<PostFilterParams, Object> params,
                          ConstraintValidatorContext context) {

        for (Map.Entry<PostFilterParams, Object> fp : params.entrySet()) {
            if (!fp.getValue().getClass().equals(fp.getKey().getParamType())) {
                return constraintValidation(context,
                    fp.getValue() + " does not match type:" +
                    fp.getKey().getParamType().getName());
            }
        }
        return true;
    }

    private boolean constraintValidation(ConstraintValidatorContext context,
String defaultMessage) {
        context.disableDefaultConstraintViolation();
        context.buildConstraintViolationWithTemplate(defaultMessage)
            .addConstraintViolation();
        return false;
    }
}
```

Snippet 11: *PostFilterParamsValidator*

Die Klasse implementiert das *ConstraintValidator*-Interface und enthält als Member-Variablen die Nachricht, welche optional bei Verwendung der Annotation übergeben werden kann. Die Initialisierung der Nachricht erfolgt in der Methode *initialize*.

Die überschriebene Methode *isValid* durchläuft jedes Key-Value Paar der Map. Sie prüft, ob die Value der Map der Klasse entspricht, die im Key durch die Enum-Konstante definiert ist. Falls dies für einen der Werte nicht der Fall ist, wird die private Methode *constraintValidation* aufgerufen, die eine Fehlermeldung an den Kontext sendet und *false* zurückgibt. Entsprechen alle Werte den erwarteten Klassen, wird *true* zurückgegeben.

Im Snippet 12 ist ein Ausschnitt der Validierung der Enumerationen *Sorting*- und *OrderParams* zu sehen.

```
@Override
public void initialize(ValidEnumValue annotation) {
    acceptedValues = Stream.of(annotation.enumClass().getEnumConstants())
        .map(Enum::name)
        .collect(Collectors.toList());
}

@Override
public boolean isValid(String value, ConstraintValidatorContext context) {
    /...Zurvor: Null Check...
    if(!acceptedValues.contains(value)) {
        return constraintValidation(context, value + " is no valid Type");
    }
    return true;
}
```

Snippet 12: *EnumValueValidator*

Falls sich der übergebene String nicht auf den in der Annotation definierten Enum-Typ (siehe Snippet 8) mappen lässt, wird eine Fehlermeldung an den ValidatorContext gesendet. [Wing19]

Neben den beiden zuvor beschriebenen Validatoren, wurden weitere implementiert, die beispielsweise zur Überprüfung des Nutzernamen oder der Post- und Kommentarinhalte dienen. Hier werden die übergebenen Parameter unter anderem auf eine bestimmte Länge überprüft, und mit Hilfe von Regular Expressions (regex) validiert.

Alle Annotationen und die zugehörigen Implementierungen sind im Package validation abgelegt.

5.2.2 Generische *ApplicationResult*-Klasse – LB, OS

Als Rückgabe der Use Cases ist eine generische Result-Klasse definiert. Diese enthält einen Status, welcher als Enumeration definiert ist sowie ein Feld *data*, dass die Rückgabedaten des erfolgreich durchgeföhrten Use Cases beinhaltet. Kommt es bei der Durchführung des Use Cases zu Fehlern, kann zusätzlich eine Fehlermeldung im Feld *errorMessage* hinterlegt werden.

```
package de.hsoc.swa.application.input.dto.out;
...
public class ApplicationResult<T> {

    public enum Status {           // Korrespondierende HTTP Response
        OK,                      // 200 bzw. 201
        NO_CONTENT,               // 204
        NOT_AUTHORIZED,          // 401
        NO_ACCESS,                // 403
        NOT_FOUND,                // 404
        NOT_VALID,                // 400
        NO_PERMISSION,            // 550
        EXCEPTION,                // 500
    }
    private final Status status;
    private T data;
    private String errorMessage = "";
    ...
    Konstruktoren und statische Methoden zum erzeugen der Results
    und öffentliche Methoden zum Abfragen des Status ausgelassen/
}
```

Snippet 13: *ApplicationResult*-Klasse (Ausschnitt)

Durch diese Klasse wird die Rückgabe an die Adapter vereinheitlicht. Im Application-Hexagon auftretende Exceptions werden dabei nicht an die Adapter weitergereicht, sondern in Status Codes übertragen, die im Adapterkontext relevant sind. Dadurch wird sichergestellt, dass interne Informationen des Application-Hexagons nicht an externe Adapter weitergereicht werden. In einem REST-Adapter können diese Status Codes beispielsweise in passende HTTP-Responses gemappt.

Bei den zurückgegebenen Daten werden die Entitäten aus dem Domänenmodell eingesetzt. Das hat zwar den Nachteil, dass die Adapter direkt auf die Domain-Entitäten zugreifen, erspart aber einen weiteres Mapping auf lokale DTOs. Diese Entscheidung ist vertretbar, da die Abhängigkeit trotzdem in Richtung des Domänenmodells zeigt und die Entitäten an den Systemgrenzen der Adapter auf DTOs gemappt werden. [Fowl04]

5.3 Definition der Output Ports – LB, OS

Die Definition der Output Ports im Application-Hexagon erfolgt ähnlich wie bei den Input Ports. Es werden Interfaces definiert, welche die vom Application-Hexagon benötigten Funktionen bereitstellen. Anders als bei den Input Ports erfolgt hierbei eine Gruppierung der Methoden in Repositories.

5.3.1 Repositories – LB, OS

Insgesamt werden fünf Repository-Interfaces definiert, die sich jeweils auf eine Entität des Domänenmodell beziehen. Der implementierende Adapter ist verantwortlich für die benötigte CRUD-Funktionalität des Entitätstyps. Diese Gruppierung ist sinnvoll, da vorgesehen ist, dass die Datenbankenzugriffe einer Entität vom selben Adapter implementiert werden.

```
public interface PostRepository {  
  
    // COMMANDS  
    RepositoryResult<Post> savePost(Post post);  
    RepositoryResult<Post> updatePost(Post post);  
    RepositoryResult<Post> deletePost(UUID postId);  
  
    // QUERIES  
    RepositoryResult<List<Post>> getFilteredPosts(Map<PostFil-  
terParams, Object> filterParams, boolean includeComments);  
    RepositoryResult<Post> getPostById(UUID postId, boolean in-  
cludeComments);  
    RepositoryResult<Post> getPostByCommentId(UUID commentId);  
}
```

Snippet 14: PostRepository

Bei der Definition der Repositories ist zu beachten, dass schreibende Zugriffe im Domain Driven Design ausschließlich über den Post als Root-Aggregate erfolgen dürfen. Demnach ist eine save-, update- und delete-Methode nur im Post-Repository (Snippet 14) zu finden. Da Votes und Kommentare über das Post-Aggregate persistiert werden, sind für diese Entitäten ausschließlich lesende Methoden im jeweiligen Repository-Interface definiert. Die Entitäten User und Topic existieren wiederum unabhängig vom Post. Deshalb werden für diese Repository-Interfaces mit CRUD-Funktionalität erstellt.

Für die Rückgabe der Daten aus den Repositories wird erneut eine generische Result-Klasse implementiert.

```
public enum Status {  
    OK,  
    ENTITY_NOT_FOUND,  
    EXCEPTION  
}
```

Snippet 15: Status-Codes der RepositoryResult-Klasse

In dieser werden Status Codes vorgegeben, die das Application Hexagon vom verantwortlichen Adapter erwartet (Snippet 15). Demnach sollen Exceptions, die bei der Persistierung auftreten, nicht in die Applikation weitergereicht, sondern vom jeweiligen Adapter behandelt werden.

5.3.2 Authorization-Gateway – OS

Neben den Repositories wird ein *AuthorizationGateway* definiert. Dieses stellt Methoden bereit, die für die Zugriffskontrolle benötigt werden. In *Snippet 16* ist ein Ausschnitt dieser Methoden zu sehen.

```
package de.hsos.swa.application.output.auth;

public interface AuthorizationGateway {
    AuthorizationResult<Void> registerUser(RegisterUserCommand command);

    AuthorizationResult<Void> addOwnership(String owningUser, UUID resourceId);

    // READ PERMISSIONS
    AuthorizationResult<Boolean> canAccessVotes(String accessingUser);

    // DELETE PERMISSIONS
    AuthorizationResult<Boolean> canDeleteVote(String accessingUser, UUID VoteId);

    // UPDATE PERMISSIONS
    AuthorizationResult<Boolean> canUpdatePost(String accessingUser, UUID commentId);
}
```

Snippet 16: AuthorizationGateway (Ausschnitt)

Für eine Registrierungsanfrage stellt das Gateway dem Use Cases die Methode *registerUser* zur Verfügung, der die vom Nutzer gewählten Zugangsdaten übergeben werden.

Beim Erzeugen einer Ressource – beispielsweise einem Post oder Vote – durch ein Mitglied des Themenforums, greift der steuernde Application Service auf die *addOwnership*-Methode des Authorization-Gateways zurück. Der implementierende Adapter des Gateways ist dafür verantwortlich die Zugehörigkeit der Ressourcen zu einem Autor festzuhalten. Über die Methoden *canDeleteVote* oder *canUpdatePost* können die Berechtigungen eines Nutzers bezüglich der übergebenen Ressource abgefragt werden. Application Services können auf diese und weitere Methoden zugreifen, um die Berechtigungen zu prüfen.

Für das Authorization-Gateway wird ebenfalls eine Result-Klasse vorgegeben, welche die notwendigen Informationen für das Application Hexagon kapselt. Die Status Codes, die in *Snippet 17* zu sehen sind, werden in den Use Cases mit einer Mapper-Klasse in passende AuthorizationResult-Objekte gemappt.

```
public enum Status {
    OK,
    USER_NOT_AUTHENTICATED,
    ACCESS_DENIED,
    NO_RESOURCE,
    ERROR
}
```

Snippet 17: Status Codes der AuthorizationResult-Klasse

5.4 Implementierung der Application Services – LB

Jeder Use Case wird nun von einem Application Service implementiert. Dabei ist zu beachten, dass ein Use Case nicht von anderen Use Cases verwendet werden darf. Dadurch sollen unerwünschte Seiteneffekte reduziert und eine klare Struktur für Entwickler geschaffen werden.

Die Application Services sind mit der JTA-Annotation `@Transactional` versehen. Der `Transactional`-Typ `REQUIRES` spezifiziert, dass für jeden Aufruf des Services, eine Transaktion benötigt wird. Alle Aktionen des Use Cases finden innerhalb einer Transaktion statt, sodass sichergestellt wird, dass bei Fehlern ein Rollback der Daten stattfinden kann.

Außerdem sind alle Services der Anwendung als `RequestScoped` definiert. Diese Services, existieren nur so lange, bis eine Anfrage erfüllt ist. Dadurch ist sichergestellt, dass alle Clients den gleichen Kontext des Services erhalten und kein State über mehrere Anfragen gespeichert wird.

Im Folgenden werden exemplarisch Application Services beschrieben, anhand derer sich die Implementierungen der anderen Use Cases nachvollziehen lässt.

Der `GetPostById`-Service nimmt die validierte `GetPostByIdQuery`-Request als Parameter entgegen. In dieser Input Port Request ist die ID des angefragten Posts, sowie die Parameter zum Sortieren der zugehörigen Kommentare enthalten.

Der Service ruft auf dem injizierten Post-Repository die Methode `getPostById` auf. Das Ergebnis des Aufrufs wird in der `RepositoryResult`-Klasse gekapselt zurückgegeben und vom Service ausgewertet. Konnte das Repository den Post mit der übergebenen ID nicht finden oder ist es zu einem anderen Fehler gekommen, kann der Service eine eigene Fehlermeldung an den aufrufenden Adapter zurückgeben.

```
@Override
public ApplicationResult<Post> getPostById(GetPostByIdQuery query) {
    RepositoryResult<Post> postResult = postRepository
        .getPostById(UUID.fromString(query.id()));

    if(postResult.status().equals(RepositoryResult.Status.ENTITY_NOT_FOUND))
        return ApplicationResult.notFound(query.id() + "not found");
    if(postResult.error())
        return ApplicationResult.exception("Cannot get Post");
```

Snippet 18: `GetPostById`-Service (Teil 1)

Wurde der Post gefunden, werden die Parameter zur Sortierung ausgewertet. Der Passende Sorting-Service wird gewählt und der Post sortiert seine Kommentare dementsprechend. Abschließen wir der Post in der `ApplicationResult`-Klasse gekapselt und an den aufrufenden Adapter zurückgegeben.

```
Comparator<Comment> sortComparator = new SortByDate<>();
if (SortingParams.valueOf(query.sortingParams()) == SortingParams.VOTES)
    sortComparator = new SortByUpvotes<>();

boolean desc = OrderParams.valueOf(query.orderParams()) == OrderParams.DESC;
postResult.get().sortComments(desc sortComparator);

return ApplicationResult.ok(postResult.get());
```

Snippet 19: `GetPostById`-Service (Teil 2)

Der *VoteEntityService* steuert die Bewertung eines Posts oder Kommentars. Zu Beginn wird der anfragende Nutzer aus dem User-Repository geladen. Danach bezieht der Use Case den Post aus dem Repository. Handelt es sich bei der bewerteten Entität um einen Kommentar, wird er anhand der ID im Post gesucht. Anschließend werden Post oder Kommentar dem Vote-Service des Domain-Models übergeben. Hier ist zu beachten, dass die beiden Entitäten das *VotedEntity*-Interface implementieren. Im Application Service muss deshalb keine Unterscheidung beim Aufruf des Domain-Services gemacht werden.

```
Optional<Vote> optionalVote = voteService.  
    vote(entityToVote, user, command.voteType());
```

Snippet 20: *VoteEntityService* (Ausschnitt)

Im Domain-Service wird das Voting auf Basis der Businesslogik ausgeführt. Das Ergebnis wird im Application-Service ausgewertet und an den Adapter zurückgegeben. Besonders ist, dass das Speichern des Votes immer über die Post-Entität als Aggregate-Root erfolgen muss, da das Aktualisieren des Kommentars nur über den zugehörigen Post zulässig ist. Anschließend wird durch das *AuthorizationGateway* der erstellte Vote dem User zugeordnet.

Wird eine Anfrage zum Löschen des Votes gestellt, kann der *DeleteVoteService* auf das Gateway zurückgreifen und die Berechtigung abfragen.

```
AuthorizationResult<Boolean> permission = authorizationGateway  
    .canDeleteVote(requestingUser, UUID.fromString(command.voteId()));  
  
if(permission.denied())  
    return AuthorizationResultMapper.handleRejection(permission.status());
```

Snippet 21: *DeleteVoteService* (Ausschnitt)

6 Implementierung der Infrastruktur-Adapter – OS

In diesem Kapitel wird in *Abschnitt 6.1* zunächst auf die Umsetzung der Datenbank-Adapter eingegangen, welche je einen Repository Output Port implementieren. In *Abschnitt 6.2* wird die Implementierung des Authorization Gateways für die Zugriffskontrolle näher betrachtet.

6.1 Persistente Datenspeicherung – OS

Die Hauptaufgabe der Adapter zur persistenten Datenspeicherung ist es, die objektorientierten Domain-Entitäten aus dem Application-Hexagon in ein relationales Datenmodell für die Datenbank zu übertragen. Zur Persistierung der Daten wird eine PostgreSQL-Datenbank genutzt. Die Interaktion mit der Datenbank erfolgt mit Hilfe des Hibernate Frameworks, welches den JPA-Standard unterstützt. Zur Verwendung dieser Technologien innerhalb der Quarks-Applikation müssen die benötigten Dependencies für den JDBC-Treiber und das ORM-Framework in der pom.xml hinterlegt werden.

Neben der Datenbank und den Devservices wird Hibernate in den application.properties konfiguriert (siehe *Snippet 22*).

```
quarkus.hibernate-orm.database.generation=create  
quarkus.hibernate-orm.database.generation.create-schemas=true  
quarkus.hibernate-orm.sql-load-script=themeforum_import.sql
```

Snippet 22: Application Properties zur Hibernate-Konfiguration (Ausschnitt)

Hibernate wird angewiesen zum Start der Applikation ein SQL-Skript mit Testdaten zu laden. Zusätzlich wird das Erzeugen von Datenbankschemas aktiviert, um die im Authorization-Adapter verwalteten Daten in einem eigenen Schema zu persistieren.

Zu jeder Domain-Entität wird ein Datenmodell zur persistenten Speicherung definiert. Das folgende *Snippet 23* zeigt die Definition der *UserPersistenceModel*-Klasse.

```
@Entity(name = "User")  
@Table(name = "user_table")  
public class UserPersistenceModel {  
    @Id UUID id;  
    @Column(name = "user_name", unique = true) String name;  
    @Basic boolean active;  
    /**Konstruktoren und statische Converter Klasse**/  
}
```

Snippet 23: UserPersistenceModel (Ausschnitt)

Die *@Entity*-Annotation markiert die Klasse als JPA-Entity und gibt dieser den Namen *User*. Die *@Table*-Annotation definiert den Namen der Tabelle im Datenbankschema.

Als Primärschlüssel wird keine ID durch die Datenbank generiert, sondern die UUID der Domain-Entity verwendet. Die Spalte für den Namen des Nutzers wird auf *unique = true* gesetzt, was gewährleistet, dass dieses Attribut einzigartig ist. Weiterhin wird ein Attribut *active* definiert, welches anzeigt, ob der Nutzer gelöscht wurde.

Jedes Datenmodell enthält eine Klasse zum Mapping des Persistenz-Modells in eine Domain-Entität, sowie in die entgegengesetzte Richtung (siehe *Snippet 24*).

```
public static class Converter {  
    public static User toDomainEntity(UserPersistenceModel pm) {  
        return new User(pm.id, pm.name, pm.active);  
  
    public static UserPersistenceModel toPersistenceModel(User u) {  
}
```

```

    return new UserPersistenceModel(u.getId(), u.getUsername(), u.isActive());
}

```

Snippet 24: Converter im *UserPersistenceModel*

Die JPA-Entity zur Persistierung eines Posts enthält die Attribute *title*, *content* und *createdAt*, welche ebenfalls in der Domain-Entity des Posts definiert sind. Das folgende Snippet 25 zeigt die Relationen der Klasse *PostPersistenceModel*.

```

@ManyToOne
@JoinColumn(name = "topic_id")
TopicPersistenceModel topicPersistenceModel;
@ManyToOne
@JoinColumn(name = "user_id")
UserPersistenceModel userPersistenceModel;
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
List<CommentPersistenceModel> comments = new ArrayList<>();
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@JoinTable(name = "post_vote",
           joinColumns = @JoinColumn(name = "post_id"),
           inverseJoinColumns = @JoinColumn(name = "vote_id"))
List<VotePersistenceModel> votes = new ArrayList<>();

```

Snippet 25: Relationen im *PostPersistenceModel*

Die Beziehung des Posts zu einem Thema, sowie die Beziehung zu seinem Autor sind jeweils mit einer @ManyToOne-Relation versehen, da mehrere Posts zum selben Thema beziehungsweise zum selben Nutzer gehören können. Die Annotation bewirkt, dass die Tabelle für den Post mit einer Fremdschlüsselbeziehung zum User, sowie zum Topic versehen wird.

Die Kommentare werden mit @OneToMany annotiert. Dadurch wird eine gesonderte Tabelle *post_table_comment_table* im Datenbankschema definiert, welche die Beziehung des Posts zu seinen Kommentaren anzeigt. Die Spezifizierung *CascadeType.ALL* führt dazu, dass alle Änderungen an einem Post auch auf die referenzierten Kommentare angewendet werden. Beim Löschen eines Posts wird so beispielsweise gewährleistet, dass ebenfalls die Kommentare gelöscht werden. Durch den Parameter *orphanRemoval = true* wird sichergestellt, dass bei Aktualisierung eines bestehenden Posts in der Datenbank, die nicht mehr referenzierten Kommentare gelöscht werden. Dadurch ist gewährleistet, dass die Kommentare auch in der Datenbank nicht ohne den Post, welcher das Aggregate-Root darstellt, existieren können.

Für die Votes eines Posts wird zusätzlich die Annotation @Join-Table verwendet. Durch diese wird erreicht, dass die 1:N-Beziehung in einer separaten Tabelle *post_vote* festgehalten wird. Im Gegensatz zum Domain-Model wird diese Beziehung auf Adapter-Ebene ebenfalls in die Rückrichtung modelliert. In Snippet 26 ist die entgegengesetzte Beziehung innerhalb der *VotePersistenceModel*-Klasse abgebildet.

```

@ManyToOne
@JoinTable(name = "post_vote",
           joinColumns = @JoinColumn(name = "vote_id"),
           inverseJoinColumns = @JoinColumn(name = "post_id"))
PostPersistenceModel postVote;

```

Snippet 26: ManyToOne Relation im *VotePersistenceModel*

Durch die bidirektionale Beziehung zwischen den JPA-Entities *VotePersistenceModel* und *PostPersistenceModel* wird ermöglicht, dass beim Abfragen von Votes einfach auf die dazugehörigen Posts zugegriffen werden kann. Dadurch entfällt ein manuelles Mapping der Zugehörigkeit im Use Case des Application-Hexagons. Dies wurde ebenfalls für die Votes der Kommentare umgesetzt.

Im Snippet 27 ist die bidirektionale Beziehung zwischen einem Kommentar und seinen Antwortkommentaren dargestellt.

```
@ManyToOne(cascade = CascadeType.PERSIST)
@JoinColumn(name = "parent_comment_id")
CommentPersistenceModel parentComment;

@OneToMany(mappedBy = "parentComment", cascade = CascadeType.ALL)
List<CommentPersistenceModel> replies = new ArrayList<>();
```

Snippet 27: Bidirektionale Parent-Child Beziehung im *CommentPersistenceModel*

Durch die Annotationen wird die rekursive Struktur der Kommentare definiert. In der @OneToOne-Annotation wird durch *mappedBy* = „parentComment“ festgelegt, dass die Beziehung im erzeugten Schema durch die Spalte *parent_comment_id* abgebildet wird. Die erzeugte Tabelle *comment_table* in der Datenbank enthält demnach eine Fremdschlüsselbeziehung auf den übergeordneten Kommentar. Handelt es sich um einen Kommentar auf erster Ebene, welcher direkt vom Post referenziert wird, ist der Eintrag für diese Spalte *null*. In der Converter-Klasse des Kommentars wurde die rekursive Datenstruktur beim Mapping berücksichtigt.

In den Persistence-Adaptoren werden für einige Datenbank-Abfragen *EntityViews* verwendet. Diese Möglichkeit der Erweiterung von JPA-Entities ist Bestandteil der Blaze Persistence Bibliothek. Bei einer *EntityView* handelt es sich um eine alternative Darstellung einer JPA-Entität. Dadurch ist es möglich, eine Untergruppe von Feldern auszuwählen, oder weitere Felder für eine spezifische Sicht auf ein *Persistence-Modell* zu realisieren. [Beik21a]

```
@EntityView(TopicPersistenceModel.class)
public record TopicPersistenceView(
    @IdMapping UUID id,
    String title,
    String description,
    LocalDateTime createdAt,
    @Mapping("userPersistenceModel") UserPersistenceView owner,
    @MappingSubquery(PostCountSubqueryProvider.class) Long posts
) {}
```

Snippet 28: *TopicPersistenceView*

In Snippet 28 ist die Definition einer *EntityView* am Beispiel des *TopicPersistenceModel* dargestellt. Über die Annotation *@EntityView* wird die Entity-Klasse übergeben, auf die sich die View bezieht. In dem Beispiel wird durch die Annotation *@MappingSubquery* eine Unterabfrage eingebettet. Die angegebene Klasse stellt eine Abfrage bereit, welche die Anzahl der Posts zu einem bestimmten Topic zählt. Dadurch kann die Anzahl der Posts als zusätzliches Feld hinzugefügt werden. In Snippet 29 ist die dafür implementierte Unterabfrage in der Datenbank dargestellt.

```
public class PostCountSubqueryProvider implements SubqueryProvider {
    @Override
    public <T> T createSubquery(SubqueryInitiator<T> subqueryBuilder) {
        return subqueryBuilder
            .from(PostPersistenceModel.class, "topicPosts")
            .select("COUNT(*)")
            .where("topicPosts.topicPersistenceModel.id")
                .eqExpression("EMBEDDING_VIEW(id)")
            .end();
    }
}
```

Snippet 29: Subquery zur Ermittlung der Anzahl von Posts zu einem Topic

In der Subquery werden alle Posts durchsucht. Falls das referenzierte Topic eines Posts der ID der Topic-View entspricht, wird die Anzahl inkrementiert. [Beik21b]

In der *TopicPersistenceView*-Klasse wird die Abfrage in ein DTO gemappt (siehe *Snippet 30*), welches die Topic-Entity kapselt und die Anzahl der Posts in einem separaten Feld bereitstellt.

```
public static TopicWithPostCountDto
    toOutputPortDto(TopicPersistenceView view) {

    Topic topic = new Topic(view.id, view.title, view.description,
        view.createdAt, UserPersistenceView.toDomainEntity(view.owner));

    return new TopicWithPostCountDto(topic, view.posts());
}
```

Snippet 30: Mapping der *TopicPersistenceView*

Für jedes Repository wird schließlich eine Adapterklasse implementiert.

```
@RequestScoped
@Transactional(value = Transactional.TxType.MANDATORY)
public class TopicPersistenceAdapter implements TopicRepository {
    @Inject
    EntityManager entityManager;

    @Inject
    CriteriaBuilderFactory criteriaBuilderFactory;

    @Inject
    EntityViewManager entityViewManager;
}
```

Snippet 31: *TopicPersistenceAdapter* (Ausschnitt)

In *Snippet 31* ist die Definition des *TopicPersistenceAdapters* zu sehen. Zum Verwalten der Entitäten wird in allen Adapter der *EntityManager* injiziert. Weiterhin wird eine *CriteriaBuilderFactory* der Blaze Persistence Bibliothek verwendet. Der *CriteriaBuilder* bietet eine API, um SQL-Abfragen in einer einfach lesbaren Form zu definieren. Der *EntityViewManager* ist ebenfalls Bestandteil von Blaze Persistence. Dieser ermöglicht die Nutzung der zuvor beschriebenen EntityViews.

In *Snippet 32* ist die Nutzung der Criteria-API von Blaze Persistence innerhalb der Funktion *searchTopic* zu sehen. Zunächst wird mit Hilfe der Factory ein *CriteriaBuilder* für das Filtern der Topic-Abfrage erzeugt. Anschließend werden die Kriterien zur Filterung der Themen mithilfe der API definiert.

```
CriteriaBuilder<TopicPersistenceModel> cb = criteriaBuilderFactory
    .create(entityManager, TopicPersistenceModel.class);
cb.whereOr()
    .where("title").like().value("%" + searchString + "%").noEscape()
    .where("description").like().value("%" + searchString + "%").noEscape()
    .where("userPersistenceModel.name").eq().value(searchString)
.endOr();

CriteriaBuilder<TopicPersistenceView> criteriaBuilderView =
    entityViewManager.applySetting(EntityViewSetting
        .create(TopicPersistenceView.class), criteriaBuilder);

List<TopicPersistenceView> topicList = criteriaBuilderView.getResultList();
return RepositoryResult.ok(topicList.stream().map(TopicPersistenceView::toOutputPortDto).toList());
```

Snippet 32: Die Methode *searchTopic* im *TopicPersistenceAdapter* (Ausschnitt)

In einem weiteren Schritt wird auf Basis der Kriterien und über den *EntityViewManager* ein *CriteriaBuilder* für die *TopicEntityView* erzeugt. Auf diesem wird *getResultSet* aufgerufen. Die resultierenden Topics der Anfrage werden auf ein DTO gemappt und innerhalb eines *RepositoryResult* an den Application-Service zurückgegeben.

6.2 Zugriffskontrolle – OS

Neben den Persistence-Adaptoren wird ein Authorization-Adapter zur Zugriffskontrolle implementiert. Dieser übernimmt zwei Aufgaben. Zum einen werden die Zugangsdaten zur Authentifizierung und rollenbasierten Zugriffskontrolle mittels Basic-Authentication am REST- und UI-Endpunkt bereitgestellt. Darüber hinaus wird in diesem Adapter die Logik für die Zugriffskontrolle auf einzelne Ressourcen umgesetzt.

Um die Authentifizierung, sowie die rollenbasierte Zugriffskontrolle zu ermöglichen, wird der *IdentityProvider* von JPA verwendet. Dafür muss zunächst die *quarkus-security-jpa* Bibliothek in der *pom.xml* eingebunden werden.[Quar22] Die Bereitstellung der Zugangsdaten erfolgt über die Definition der Klasse *AuthUser* (siehe Snippet 33).

```
@Entity
@Table(name = "auth_user_table", schema = "auth")
@UserDefinition
public class AuthUser {
    @Id @GeneratedValue() UUID id;
    @Username @Column(unique = true) String username;
    @Password String password;
    @Roles String role;
    @Column(name = "user_id") UUID userId;
    @OneToMany(mappedBy = "owner", cascade = CascadeType.REMOVE)
    List<OwnerOf> ownedRessources;
```

Snippet 33: AuthUser-Entity im Authorization-Adapter

Die Nutzerdaten zur Autorisierung werden getrennt von den restlichen Applikationsdaten gespeichert. Hierfür wird in der *@Table*-Annotation definiert, dass diese im getrennten Schema *auth* persistiert werden sollen.

Der *AuthUser* wird mit *@UserDefinition* annotiert. Die Klasse stellt damit die *SecurityIdentity* bereit und benötigt hierfür neben dem Nutzernamen und Passwort ebenfalls eine Rolle. Die ID des *AuthUsers* wird automatisch generiert. Der *AuthUser* enthält weiterhin eine *OneToMany*-Relation zu den IDs der Ressourcen, die von dem Nutzer erstellt worden sind. Diese werden in der *OwnerOf*-Entity modelliert (siehe Snippet 34).

```
@Entity
@Table(name = "owner_of_table", schema = "auth")
public class OwnerOf {
    @Id
    @SequenceGenerator(name = "ownerOfIdSequence", initialValue = 100)
    @GeneratedValue(generator = "ownerOfIdSequence")
    Long id;

    @ManyToOne()
    @JoinColumn(name = "owner_id")
    AuthUser owner;

    @Column(unique = true, name = "ressource_id")
    UUID resourceId;
```

Snippet 34: OwnerOf-Entity im Authorization-Adapter

Die *OwnerOf*-Entity hält fest, welche Ressourcen-IDs zu welchem Nutzer gehören. Bei Erzeugung einer Ressource innerhalb eines Use Cases wird hierfür die Methode

addOwnership des Authorization-Adapers verwendet. Der übergebene Nutzernname *owningUser* wird zuvor am REST-Adapter über den *SecurityContext* ermittelt.

```
public AuthorizationResult<Void> addOwnership(String owningUser, UUID resourceId) {
    Optional<AuthUser> optionalAuthUser = this.loadUser(owningUser);
    if(optionalAuthUser.isEmpty())
        return AuthorizationResult.notAuthenticated();
    AuthUser owner = optionalAuthUser.get();
    try{
        OwnerOf ownerOf = new OwnerOf(owner, resourceId);
        entityManager.persist(ownerOf);
        return AuthorizationResult.ok();
    } catch (PersistenceException e) {
        return AuthorizationResult.exception();
    }
}
```

Snippet 35: Die Methode *addOwnerhip* im Authorization-Adapter

Innerhalb der Methode wird zunächst der Nutzer aus der *AuthUser*-Tabelle geladen. Anschließend wird die *OwnerOf*-Beziehung mit Hilfe des EntityManagers persistiert.

Für die verschiedenen Ressourcentypen werden innerhalb des Adapters einzelne Methoden definiert, welche die Zugriffe des anfragenden Nutzers auf einzelne Ressourcen autorisieren. In *Snippet 36* ist dies am Beispiel der *canDeletePost*-Methode zu sehen.

```
@Override
public AuthorizationResult<Boolean> canDeletePost(String accessingUser, UUID postId) {
    // Admins dürfen alle Posts löschen
    if(isActiveAdmin(accessingUser))
        return AuthorizationResult.ok();
    // Member dürfen ihre eigenen Posts löschen
    if(isActiveRessourceOwner(accessingUser, postId))
        return AuthorizationResult.ok();
    return AuthorizationResult.notAllowed();
}
```

Snippet 36: Die Methode *canDeletePost* im Authorization-Adapter

Diese gibt das Löschen von Posts für Administratoren frei. Handelt es sich bei dem anfragenden Nutzer um einen Member, wird geprüft, ob der Post ihm zugeordnet ist.

In vielen Use Cases wird eine Zugriffskontrolle benötigt. Damit diese einheitlich erfolgt und zentral angepasst werden kann, ist es sinnvoll diese in den Authorization-Adapter auszulagern.

7 Implementierung der Interaktions-Adapter – LB

Da nun die Applikationslogik und die Adapter zur persistenten Speicherung der Daten implementiert sind, können die Interaktions-Adapter umgesetzt werden. Diese nehmen beispielsweise direkte Nutzereingaben eines User Interfaces entgegen oder bieten eine Machine-to-Machine-Kommunikation in Form einer REST-Schnittstelle an. Sie nutzen dazu die Use Cases des Application-Hexagons und führen teilweise eigene Logik zur Zugriffskontrolle und Validierung aus.

Für das Themenforum werden zwei Adapter zur Interaktion mit der Anwendung implementiert. Zum einen eine REST-API und zum anderen ein Server Side Rendered Web-frontend. Diese werden in den folgenden Abschnitten beschrieben.

7.1 Umsetzung der REST-Schnittstelle – LB

Representational State Transfer wurde im Jahre 2000 von Roy Fielding erarbeitet und beschreibt ein Architekturkonzept für Webservices. Ursprünglich war der Ansatz protokollunabhängig modelliert. Er wird aber nahezu immer mit dem HTTP-Protokoll und den dazugehörigen Verben implementiert.

Ziel einer REST-API ist es, eine standardisierte Kommunikation mit Client-Anwendungen zu ermöglichen. Dadurch wird verhindert, dass Client-Anwendungen an eine spezifische Implementierung der API gebunden sind. Um dies zu erreichen, gibt REST einige Standards in der Modellierung einer API vor. Diese Standards können aber aufgrund der technologischen Offenheit nicht erzwungen werden und sind deshalb eher als Richtlinien zu betrachten.

REST-APIs sind auf Basis von Ressourcen designt (ROA). Dies können Objekte, Daten oder Dienste sein, die für den Client zugänglich gemacht werden sollen. Jede Ressource ist eindeutig über eine URI (Uniform Ressource Identifier) identifizierbar. Weiterhin sind alle Requests an eine REST-API zustandslos. Das bedeutet, dass jede Anfrage eines Clients alle benötigten Informationen beinhaltet, die für die Ausführung der Anfrage notwendig sind. Der Server speichert keinen Client-State über mehrere Requests. Dadurch sind REST-APIs sehr gut skalierbar. [Fiel08][Micr22]

7.1.1 Modellierung der REST-Endpunkte – LB

Bei der Modellierung der REST-Endpunkte für das Themenforum wurden zunächst die verschiedenen Ressourcentypen identifiziert. Diese decken sich weitestgehend mit den Entitäten, die im Domänen-Modell vorhanden sind. Allerdings muss hier an einigen Stellen die Beziehung der Entitäten aufgelöst werden, um den REST-Standard zu erfüllen. Dieser lässt strenggenommen keine verschachtelten URIs zu über die beispielsweise eine Hierarchie der Ressourcen abgebildet wird. Denn dadurch wird die Implementierung der API an den aufrufenden Client gebunden.

Kommentare, die im Domänenmodell nur im Zusammenhang mit einem Post existieren können, sollen durch die REST-API auch unabhängig von Posts zugänglich sein. Gleiches gilt beispielsweise auch für die Bewertungen eines Posts oder Kommentars. Die Tabelle im Anhang zeigt die modellierten REST-Endpunkte mit den Zugehörigen HTTP-Verben und einer kurzen Beschreibung der Aktion.

7.1.2 Implementierung der REST-Endpunkte – LB

Auf Basis dieser Modellierung können die Endpunkte nun als Adapter in der Anwendung implementiert werden. Dazu wurden Klassen für die Ressourcen *Comments*, *Posts*, *Topics*, *Users* und *Votes* angelegt. Diese Klassen können nun über Annotationen des JAX-RS-Standards konfiguriert werden.

```
@RequestScoped
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@Path("api/v1/posts")
@Transactional(value = Transactional.TxType.REQUIRES_NEW)
@Adapter
public class PostsRessource {
```

Snippet 37: JAX-RS Annotationen der PostRessource

Das Snippet 37 zeigt die Annotationen der *PostRessource*-Klasse. Über *Produces* und *Consumes*, kann festgelegt werden, welcher Medientyp von dem Endpunkt verarbeitet und zurückgeben wird. Die REST-Endpunkte der Anwendung verarbeiten alle JSON. Weiterhin wird ein Basispfad für die entsprechende Ressource angegeben. Außerdem wird eingestellt, dass für jede Anfrage an den Endpunkt eine neue Transaktion erstellt wird. Dies ist zwar bereits in den Input Ports der Applikation erfolgt, jedoch kann sich dort nicht darauf verlassen werden, dass jeder externe Adapter auch eine Transaktion gestartet hat, um die Anfrage zu bearbeiten.

Über Annotationen der CDI werden nun die Use Cases in den Endpunkt injiziert, die für die Bearbeitung der Anfragen benötigt werden.

Eine Methode der Ressource-Klasse kann nun mit einem HTTP-Verb versehen und so mit zugänglich für HTTP-Aufrufe gemacht werden. Weiterhin kann ein Path-Parameter hinzugefügt werden, der bei einem Aufruf aus der URI ausgelesen und an die Methode gegeben wird. Dies wird genutzt, um die ID einer Ressource aus dem Pfad auszulesen, wenn der Nutzer diese Ressource anfragt. Queryparameter sind optionale Felder der URI, die ebenfalls an die Methode übergeben werden. Über Diese werden beispielsweise Filterung oder Sortierung der Ressourcen realisiert. Handelt es sich um eine POST-, PATCH- oder PUT-Methode kann ebenfalls ein Requestbody an die Methode übergeben werden. Eine rollenbasierte Zugriffskontrolle wird von dem JAX-RS Standard ebenfalls unterstützt und kann über Annotationen auf Methoden-Level konfiguriert werden. Im Themenforum wird zwischen den Rollen „Member“ und „Admin“ unterschieden.

```
public Response createPost()
    CreatePostRequestBody request,
    @Context SecurityContext securityContext)
```

Snippet 38: Die Methodensignatur createPost der PostRessource

Das Snippet 38 zeigt die Signatur der Methode zum Erstellen eines neuen Beitrags im Themenforum (Annotationen weggelassen). Die Klasse *CreatePostRequestBody* dient der Serialisierung des JSON-Body, der in der HTTP-Request gesendet wird, auf ein Java-Objekt. Demnach beinhaltet die Klasse alle Properties, die für das Erstellen eines neuen Posts benötigt werden. Außerdem wird über die Bean-Validation eine einfache Validierung der Anfrage durchgeführt. Hier wird nur auf Vollständigkeit der Daten und nicht auf Businesslogik geprüft.

Erhält der Endpunkt eine POST-Request, wird die Methode aufgerufen und durch einen Validation-Service die zuvor beschriebene Validierung angestoßen. Ist diese gültig, wird

der Username aus dem *SecurityContext* der Request ausgelesen. Anschließend wird der Requestbody mit Hilfe einer internen Mapper-Klasse auf den korrespondierende Input Port Command konvertiert. Dieses Objekt kann nun zusammen mit dem anfragenden Nutzer und den Use Case gegeben werden (siehe *Snippet 39*).

```
validationService.validate(request);
String username = securityContext.getUserPrincipal().getName();
CreatePostCommand command = CreatePostRequestBody.Converter
    .toInputPortCommand(request);
ApplicationResult<Post> result = this.createPostUseCase
    .createPost(command, username);
```

Snippet 39: Die Methode *createPost* (Ausschnitt)

Nun führt der Use Case auf Applikationsebene die Businesslogik aus und gibt ein Ergebnis in Form der *ApplicationResult*-Klasse an die Methode zurück. Das Ergebnis wird im Restadapter ausgewertet. War die Anfrage erfolgreich und ein neuer Post wurde erstellt, wird der erstellte Post auf ein DTO gemappt und als HTTP-Response zurückgegeben. Dadurch wird sichergestellt, dass keine Domain-Entitäten an externe Systeme gesendet werden.

```
if (result.ok()) {
    PostDto response = PostDto.Converter.fromEntity(result.data());
    return Response.status(Response.Status.OK).entity(response).build();
}
```

Snippet 40: Überprüfung des *ApplicationResult* in der *createPost*-Methode

Kam es auf Applikationsebene zu Fehlern, werden diese mit Hilfe der *ErrorResponse*-Klasse auf einen passenden HTTP-Statuscode gemappt. Dabei kann neben dem Status des *ApplicationResults* auch auf die spezifische Fehlermeldung zurückgegriffen werden. Ist bereits bei der Validierung des Requestbody oder des Input Port Commands ein Fehler aufgetreten, wird eine *ErrorResponse* mit den entsprechenden *ConstraintViolations* zurückgegeben.

```
return ErrorResponse
    .fromApplicationResult(result.status(), result.message());
} catch (ConstraintViolationException e) {
    return ErrorResponse.
        fromConstraintViolation(e.getConstraintViolations());
}
```

Snippet 41: Rückgabe einer *ErrorResponse*

Dieser Ablauf findet sich in den Methoden der anderen REST-Endpunkte in ähnlicher Form wieder. Lediglich die Use Cases und Input Port Commands müssen angepasst werden, um die benötigten Funktionen zu implementieren. Durch die *ErrorResponse*-Klasse ist ebenfalls eine einheitliche Fehlerbehandlung für alle Endpoints gegeben.

7.1.3 Dokumentation der REST-Endpunkte – LB

Eine gute REST-Schnittstelle zeichnet sich auch durch eine angemessene Dokumentation aus. Dazu bietet Quarkus eine Implementierung der OpenAPI-Specification an. Diese ermöglicht es, REST-APIs einheitlich zu dokumentieren. Tools, wie zum Beispiel Swagger, können auf diese Dokumentation zurückgreifen und eine grafische Oberfläche zum Testen der API bereitstellen.

Die Dokumentation mit OpenAPI wird zunächst in den application.properties konfiguriert. Hier werden allgemeine Informationen, wie zum Beispiel Titel und Versionsnummer der

API angegeben. Nun können die Endpunkte auf Methoden-Level mit verschiedenen Annotationen dokumentiert werden. Dabei werden neben einem Titel und einer Beschreibung der Methode auch die erwarteten Response-Codes und Bodies beschrieben. Die verwendeten DTO-Klassen werden an die Annotation übergeben. RequestBodies können ebenfalls der Dokumentation hinzugefügt werden.

7.1.4 Testen der REST-Endpunkte – LB

Mit dem Rest-Assured-Framework können die Endpunkte getestet werden. Dazu wird zunächst in den application.properties ein Port für die *devservices* eingestellt und die form-based-authentication deaktiviert. Testdaten können ebenfalls über die application.properties in Form einer import.sql eingefügt werden. Properties, die mit dem Präfix „%test“ versehen sind, werden nur beim Testen der Anwendung berücksichtigt.

Für jeden Endpunkt wird eine Testklasse im test-Package angelegt. Über die @TestHT-TPEndpoint-Annotation kann die zu testende Ressourcen-Klasse referenziert werden. Die Methoden werden mit @Test annotiert. Nun können die einzelnen Http-Requests konstruiert und an den Endpunkt gesendet werden. Für jeden Endpunkt werden auf diese Weise die Response-Codes überprüft.

Um neben den Response-Codes auch Zugriff auf den Responsebody zu bekommen, bietet Rest-Assured die Klasse *ValidatedResponse* an. Diese beinhaltet die HTTP-Response der Anfrage und kann auf verschiedene Parameter analysiert werden. So wird zum Beispiel beim Erstellen eines neuen Posts darauf geprüft, ob alle Felder richtig übergeben werden. Ebenfalls werden diverse Kombinationen falscher Eingaben getestet. Die Response kann daraufhin neben dem Response-Code auch auf die richtige Fehlermeldung überprüft werden. Auch die Authentifizierung und Autorisierung kann mit Rest-Assured getestet werden. Beim Erstellen einer Request muss dazu lediglich die basic-authentication eingestellt werden.

```
ValidatableResponse response = given()
    .header("Content-Type", "application/json")
    .contentType("application/json")
    .auth().basic("lbattist", "lbattist")
    .body(postData)
    .post()
    .then()
    .statusCode(400);

JsonPath jsonPath = response.extract().jsonPath();
List<Map<String, String>> errors = jsonPath.getList("errors");

Assertions.assertEquals(errors.get(0).get("message"),
    "Invalid request");
Assertions.assertEquals(errors.get(0).get("detail"),
    "title is blank");
}
```

Snippet 42: REST-Assured Testing (Ausschnitt):

7.2 Quarkus Qute Webfrontend – LB

Nun wird das Webfrontend für das Themenforum implementiert. Dieses wird mit der Quarkus QuTe-Templating Engine umgesetzt und stellt neben der Restschnittstelle den zweiten Interaktionsadapter der Anwendung dar. QuTe verfolgt dazu ein Konzept, bei

dem HTML-Dateien mit Hilfe von Java-Code generiert und dann serverseitig gerendert werden. Dazu bietet QuTe sogenannte *TemplateInstances* an, denen Parameter übergeben werden können. Auf diese Parameter kann man im HTML-Code zurückgreifen, um die Seite mit Informationen anzureichern.

7.2.1 Implementierung der User Interface Endpunkte – LB

Es werden zunächst vier Endpunkte angelegt, über die das UI erreichbar ist. Der *PublicEndpoint* ist für alle Internetnutzer zugänglich und stellt eine Index-Seite, sowie Templates zum Einloggen und Registrieren auf dem Themenforum bereit. Der AdminEndpoint hingegen ist nur für Administratoren der Anwendung zugänglich. Hier können alle Inhalte und Nutzer des Themenforums verwaltet werden. Die Endpunkte für Topics und Posts können für nicht registrierte Nutzer nur mit eingeschränkter Funktionalität aufgerufen werden. So ist beispielsweise das Kommentieren von Posts nur für angemeldete Nutzer möglich.

Die Implementierung der Endpunkte erfolgt ähnlich, wie die der Restschnittstelle. Für jeden der vier Endpunkte wird eine Pfad deklariert, über den dieser erreichbar ist. Weiterhin wird rollenbasierte Zugriffskontrolle wieder mit Annotationen der JAX-RS-Spezifikation realisiert. Anders als in den Restendpunkten, sind die Methoden der UI-Endpoints nur mit der @GET-Annotation versehen. Ebenso werden keine HTTP-Responses, sondern die zuvor beschriebenen *TemplateInstances* an den Client zurückgegeben. Deshalb wird der produzierte Mediatype der Methoden auf TEXT_HTML eingestellt.

Um die angefragten Seiten zu rendern, können die Methoden auf die Use Cases der Applikation zurückgreifen. Dadurch können sie zum Beispiel, die Liste aller Topics aus der Applikation beziehen, und diese an das Template weiterreichen. Basierend auf diesen Daten wird der HTML-Code generiert und zurückgegeben.

```
ApplicationResult<List<TopicWithPostCountDto>> allTopics
    = getAllTopicsUseCase.getAllTopics();
List<TopicDto> topicDtos = new ArrayList<>();
if (allTopics.ok()) {
    topicDtos = allTopics.data().stream()
        .map(TopicDto.Converter::fromInputPortDto).toList();
}
return Templates.topics(topicDtos, isLoggedIn, username);
```

Snippet 43: GetTopics-Methode des TopicEndpoint (Ausschnitt)

Das Snippet 44 zeigt einen Ausschnitt aus der topics.html-Seite. Mit Hilfe des for-Loops von QuTe kann über die übergebenen TopicDTOs iteriert und HTML-Code generiert werden. Dabei kann QuTe entweder auf die Getter oder die public-Felder einer Java-Klasse zurückgreifen.

```
{#for entry in allTopics}
<div class="content-box">
    <div>
        <h5 style="cursor: pointer" onclick="window.o-
pen('/ui/posts?topic={entry.title}', '_self')"
            class="title-info">{entry.title}</h5>
        <span class="sub-info">{entry.description}</span><br>
        <span class="sub-info">
            {entry.parsedCreated At Date} |
            by <strong>{entry.owner}</strong> |
        </span>
```

```

        {entry.numberOfPosts} posts |
    </span>
    <a href="/ui/posts?topic={entry.title}">goto</a>
</div>
</div>
{ /}

```

Snippet 44: Topics.html-Template (Ausschnitt)

An manchen Stellen wird eine Formatierung oder zusätzliche Berechnung von Daten für das Frontend benötigt. So müssen beispielsweise die Java-Time Stamps in ein angemessenes Datumsformat übertragen werden oder die Anzahl an Kommentaren unter einem Post dargestellt werden. Dazu bietet QuTe sogenannte *TemplateExtensions* an. Dies sind statische Methoden, die die Funktionalität eines Objektes erweitern können. Dies ist sinnvoll, da man das Domänenmodell nicht mit Methoden zur Präsentationslogik belasten will. Dadurch kann das Datumsformat von einer Methode der TemplateExtensions angepasst werden. Aber auch komplexere Berechnungen sind möglich. So wird zum Beispiel geprüft, ob der angemeldete Nutzer den aktuellen Post bereits mit einem Up- oder Downvote bewertet hat. Basierend auf dieser Information wird das UI angepasst, sodass die Buttons zum Bewerten ausgegraut sind.

```

public static boolean loggedInUserCanDownvote(Post post, String
username) {
    Vote vote = loggedInUserVote(post, username);
    return !post.getUser().getName().equals(username) &&
           (vote == null || vote.getVoteType() != VoteType.DOWN);
}

```

Snippet 45: Template Extension *loggedInUserCanDownvote*

Um die Anmeldung von Nutzern im Webfrontend zu gewährleisten, wird neben der Basic-Authentication zusätzlich die formularbasierte Authentifizierung konfiguriert.

```

quarkus.http.auth.basic=true
quarkus.http.auth.form.enabled=true
%test.quarkus.http.auth.form.enabled=false
quarkus.http.auth.form.login-page=/ui/login
quarkus.http.auth.form.error-page=/ui/error
quarkus.http.auth.form.landing-page=/ui/topics

```

Snippet 46: Konfiguration der Authentifizierung in den Application Properties

Dazu wird in den Application Properties ein Pfad zum Login-Screen definiert. Das Template enthält ein HTML-Formular mit der Action *j_security_check*. Bei erfolgreicher Anmeldung wird im Browser des Clients ein *quarkus-credential* Cookie gesetzt. Dieses wird bei allen Anfragen mitgesendet und ermöglicht die Authentifizierung des Nutzers. Für das Testen der Quarkus-Applikation wird die Formular-Basierte Authentifizierung deaktiviert, da beim Testen ansonsten automatisch auf die Login-Page umgeleitet wird.

8 Zusammenfassung und Fazit – LB, OS

Das Themenforum wurde erfolgreich umgesetzt. Die Muss-Kriterien des Projektvorschlags wurden alle erfüllt. Die Anwendung verfügt über die geforderten Schnittstellen zur Mensch-Maschine-Interaktion über ein Webfrontend und zur Maschine-Maschine-Interaktion über eine REST-API.

Durch die Hexagonale Architektur und das „Ports & Adapters“ Pattern war eine klare Aufteilung der Software-Komponenten möglich. Dadurch konnte zunächst die Applikationslogik unabhängig von den Adapters umgesetzt werden.

Das Domain Model im Kern der Applikation wurde so umgesetzt, dass es einen Großteil der Businesslogik des Themenforums kapselt. Hierbei konnten Elemente des taktischen Domain Driven Designs sinnvoll angewendet werden. Allerdings gibt es Aspekte, die bei der Modellierung nicht beachtet wurden. So stellen zum Beispiel die Autorisierung und Zugriffskontrolle ebenfalls einen Teil des Domain Models dar, der aber nicht mit in die Konzeption aufgenommen wurden. Ein zusätzlicher Fokus auf strategisches Design hätte dazu beitragen können, die Zugriffskontrolle als weiteren Systemkontext zu identifizieren.

Das Domain Model konnte dennoch sinnvoll in den Use Cases verwendet werden. Hier haben sich die Command-DTOs an den eingehenden Ports als hilfreich herausgestellt, da diese zentral auf Applikationsebene validiert werden können. Durch die strenge Zuordnung eines Command-DTOs zu einem Use Case ist die Änderung bestehender Use Cases ohne Seiteneffekte möglich. Ebenfalls können neue Use Cases nach dem bestehenden Schema einfach hinzugefügt werden. Auf den Einsatz lokaler DTOs als Rückgabe aus der Applikation wurde verzichtet, um zusätzliches Mapping zu vermeiden. In einem weiteren Schritt könnten diese eingeführt werden, um zu verhindern, dass Domain-Entities an die Adapter gegeben werden.

Durch die Hexagonale Architektur wurde erreicht, dass die Infrastruktur-Adapter den Use Cases zuarbeiten. Sie selbst enthalten wenig Businesslogik. Durch die Umkehrung der Abhängigkeiten sind diese einfach austauschbar und man erhält ein System mit hoher Kohäsion und geringer Kopplung.

Der REST-Standard und die ROA unterscheiden sich von den objektorientierten Prinzipien im Application-Hexagon und dem relationalen Modell zur Persistierung. Hier musste an einigen Stellen umgedacht werden, um die Beziehungen passend aufzulösen. Dennoch wurden die REST-Prinzipien eingehalten. Die Endpunkte konnten durch den klaren Aufbau und die einheitliche Fehlerbehandlung einfach mit Rest-Assured getestet werden. Die Integration von Fault Tolerance Mechanismen wurde im Rahmen der Umsetzung nicht weiter untersucht, da keine externen Systeme in die Anwendung eingebunden sind.

Das SSR-Webfrondend konnte ebenfalls erfolgreich mit Quarkus QuTe umgesetzt werden. Es veranschaulicht auf der einen Seite die Nutzung des Themenforums. Auf der anderen Seite konnte es auch im Entwicklungsprozess für das Testen der Funktionalität – wie etwa dem Voting von Posts und Kommentaren – zur Hilfe genommen werden.

Abschließend betrachtet hat sich die Verwendung einer Hexagonalen Architektur für das Themenforum als hilfreich herausgestellt. Durch den gewählten Architekturansatz mussten viele zusätzliche Klassen, Interfaces und Mapper definiert werden. Dies erschien zu

Beginn der Entwicklung als Mehraufwand. Allerdings wurde dadurch erreicht, dass die Applikationslogik in eine API gekapselt ist. Dadurch wurde eine gute Developer Experience erreicht, welche mit zunehmendem Projektfortschritt die Zusammenarbeit erleichterte.

9 Referenzen

- [AvMa06] AVRAM, ABEL; MARINESCU, FLOYD: Domain-Driven Design Quickly: A Summary of Eric Evans' Domain-Driven Design, Enterprise Software Development Series. C4Media, 2006 — ISBN 978-1-4116-0925-9
- [Beik21a] BEIKOV, CHRISTIAN: *Blaze Persistence - Entity View Module*. URL https://persistence.blazebit.com/documentation/1.6/entity-view/manual/en_US/. - abgerufen am 2023-02-11
- [Beik21b] BEIKOV, CHRISTIAN: *Blaze Persistence - Subquery-Mappings*. URL https://persistence.blazebit.com/documentation/1.6/entity-view/manual/en_US/index.html#anchor-subquery-mappings. - abgerufen am 2023-02-11
- [Cock08] COCKBURN, ALISTAIR: *Hexagonal architecture*. URL <https://alistair.cockburn.us/hexagonal-architecture/>. - abgerufen am 2023-02-11. — Alistair Cockburn
- [Fere22] FERENTSCHIK, HARDY: *Hibernate Validator 8.0.0.Final - Jakarta Bean Validation Reference Implementation: Reference Guide*. URL https://docs.jboss.org/hibernate/validator/8.0/reference/en-US/html_single/#validator-customconstraints-constraintannotation. - abgerufen am 2023-02-11
- [Fiel08] FIELDING, ROY T.: *On software architecture » Untangled*. URL <https://roy.gbiv.com/untangled/2008/on-software-architecture>. - abgerufen am 2023-02-11
- [Fowl04] FOWLER, MARTIN: *bliki: LocalDTO*. URL <https://martinfowler.com/bliki/LocalDTO.html>. - abgerufen am 2023-02-11. — martinfowler.com
- [Hexa23] *Hexagonal Architecture - What Is It? Why Should You Use It?* URL <https://www.happycoders.eu/software-craftsmanship/hexagonal-architecture/#hexagonal-architecture-vs-layered-architecture>. - abgerufen am 2023-02-11
- [Homb19] HOMBERGS, TOM: *Get Your Hands Dirty on Clean Architecture*. 1st edition. Birmingham, UK : Packt Publishing, 2019 — ISBN 978-1-83921-196-6
- [Mart18] MARTIN, ROBERT C.: *Clean architecture: a craftsman's guide to software structure and design, Robert C. Martin series*. London, England : Prentice Hall, 2018 — ISBN 978-0-13-449416-6
- [Micr22] MICROSOFT: *Web API design best practices - Azure Architecture Center*. URL <https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>. - abgerufen am 2023-02-11
- [Pete22] PETERLIĆ, ANA: *Getting Started with Blaze Persistence | Baeldung*. URL <https://www.baeldung.com/blaze-persistence-tutorial>. - abgerufen am 2023-02-11

- [Quar22] QUARKUS: *Secure a Quarkus application with Basic authentication.* URL <https://quarkus.io/guides/security-basic-authentication-tutorial>. - abgerufen am 2023-02-11
- [Quar23a] QUARKUS: *Quarkus Standards.* URL <https://quarkus.io/standards/>. - abgerufen am 2023-02-11
- [Quar23b] QUARKUS: *Qute Templating Engine.* URL <https://quarkus.io/guides/qute>. - abgerufen am 2023-02-11
- [Vern13] VERNON, VAUGHN: *Implementing domain-driven design.* Upper Saddle River, NJ : Addison-Wesley, 2013 — ISBN 978-0-321-83457-7
- [Wing19] WINGERDEN, MARTIN VAN: *Validations for Enum Types | Baeldung.* URL <https://www.baeldung.com/javax-validations-enums>. - abgerufen am 2023-02-11

Anhang

Anhang 1: Projektvorschlag

SWA (MI || TI), WS 2022, Prof. Dr. R. Roosmann

Projektarbeit, Gr. 16

Themenforum

Modul: Software-Architektur – Konzepte und Anwendungen

Studiengang: Informatik-Medieninformatik

Semester: Wintersemester 2022/2023

Dozent: Prof. Dr. R. Roosmann

Studierende:

[1] Lorenzo Battiston, Matnr. 919355, <mailto:lorenzo.battiston@hs-osnabrueck.de>, Kennung: lbattist

[2] Oliver Schlueter, Matnr. 914726, <mailto:oliver.schlueter@hs-osnabrueck.de>, Kennung: oschluet

Die Vorgaben der über Ilias veröffentlichten Datei "SWA_Infos.pdf" und der als Anlage 1 beigefügten Rahmenanforderungen sind umzusetzen.

1. Zielbestimmung

Das Themenforum bietet eine Plattform, auf der Nutzer Beiträge posten und kommentieren können. Posts und Kommentare können mit Up- und Downvotes von den Nutzern bewertet werden. Die geposteten Beiträge sind immer einem Thema zugeordnet. Diese Oberthemen können ebenfalls von den Nutzer erstellt werden, und sind dann für alle anderen Nutzer zugänglich.

Als nicht registrierter Nutzer kann man alle Beiträge und Kommentare sehen. Möchte man selbst einen Post erstellen oder kommentieren, benötigt man einen Account.

Das Projekt orientiert sich im Wesentlichen an der Internetplattform "Reddit.com".

1.1. Musskriterien

Besucher = nicht registrierter Benutzer; Mitglied = registrierter und angemeldeter Benutzer
Administrator = Mitglied mit speziellen Rechten zur Administration des Systems

- Besucher können nach Themen suchen
- Besucher können sich zu einem Thema die zugehörigen Posts ansehen
- Besucher können Kommentare zu einem Post ansehen
- Besucher können Up-/Downvotes eines Posts/Kommentars sehen
- Besucher können Beiträge nach Upvotes und Datum sortieren
- Besucher der Plattform können sich registrieren, um Mitglied zu werden
- Besucher können sich als Mitglied registrieren
- Mitglieder können sich am System anmelden
- Mitglieder können neue Themen erstellen
- Mitglieder können Posts zu einem Thema erstellen
- Mitglieder können eigene Posts bearbeiten und andere Posts kommentieren
- Mitglieder können auf Kommentare antworten
- Mitglieder können Posts und Kommentare Up-/Downvoten
- Mitglieder haben eine private Profilübersicht, die Posts, Kommentare und Votes anzeigt
- Der Administrator kann all Themen, Posts, Kommentare und Mitglieder verwalten

1.2. Wunschkriterien

- Mitglieder haben ein öffentliches Profil
- Posts sollen in einem Rich-Text-Format gespeichert und angezeigt werden (Bold, Kursiv, Absätze möglich)

1.3. Abgrenzungskriterien

- Qualität der Umsetzung und die Erreichung definierter Qualitätsmerkmale ist höher zu gewichten, als die Quantität angebotener Funktionalität
- als Qualitätsmerkmale sollen berücksichtigt werden: a) Benutzer- und Entwicklerfreundliche Schnittstellen, b) Korrektheit der Funktionalität, c) angemessene IT-Security und d) Resilienz (Fault-Tolerance und Metriken zum Laufzeitmonitorings)
- Es werden zwei Schnittstellen zur Interaktion bereitgestellt, konkret:
 - Mensch-Maschine Interaktion über eine Web-App,
 - Maschine-Maschine Interaktion über eine REST-API
- Es werden keine externen APIs eingebunden und kein Rich-Media unterstützt
- Es gibt keine Moderatoren für einzelne Themen auf der Plattform

2. Produktfunktionen

2.1. Nutzerverwaltung

- /F0011/ Ein beliebiger Internetnutzer kann sich mit einem Benutzernamen und Passwort registrieren
- /F0012/ Bei erfolgreicher Registrierung kann sich der Nutzer mit seinem Benutzernamen und Passwort anmelden
- /F0013/ Ein Nutzer wird eindeutig über seinen gewählten Benutzernamen identifiziert.
- /F0014/ Nicht-registrierte Nutzer können Inhalte nur ansehen

2.2. Inhalte verwalten

Inhalte = Posts, Themen, Kommentare

- /F0021/ Ein angemeldeter Nutzer kann über die bereitgestellten Schnittstellen Posts auf der Plattform erstellen
- /F0022/ Ein angemeldeter Nutzer kann Posts anderer Nutzer kommentieren
- /F0023/ Ein angemeldeter Nutzer kann auf andere Kommentare antworten
- /F0024/ Ein angemeldeter Nutzer kann neue Themen auf der Plattform veröffentlichen
- /F0025/ Ein angemeldeter Nutzer kann zu allen Themen auf der Plattform Posts erstellen
- /F0026/ Ein angemeldeter Nutzer kann ausschließlich seine eigenen Inhalte löschen

2.3. Bewertung der Inhalte

- /F0031/ Ein angemeldeter Nutzer kann Posts bewerten (Up-/Downvotes)
- /F0032/ Ein angemeldeter Nutzer kann Kommentare bewerten (Up-/Downvotes)

2.4. Darstellung der Inhalte

- /F0041/ Posts und Kommentare werden standardmäßig nach Anzahl der Upvotes absteigend sortiert (Abzüglich der Downvotes)
- /F0042/ Posts können alternativ nach Erstelldatum (neueste Zuerst) sortiert werden
- /F0043/ Ein beliebiger Internetnutzer kann alle Posts (nach den in /F0041/, /F0042/ beschriebenen Sortierungen) auf der Plattform ansehen
- /F0044/ Ein beliebiger Internetnutzer kann alle Posts (nach den in /F0041/, /F0042/ beschriebenen Sortierungen) eines bestimmten Themas ansehen
- /F0045/ Ein angemeldeter Nutzer kann alle seine geposteten Inhalte auf seiner private Profilübersicht einsehen und löschen

3. Benutzungsoberfläche - Mockup

/ui/posts/

The page title is "Alle Beiträge". It features a "Join" button and a "[Browse Themen]" link. Below this, there are two tabs: "Popular" and "New". The "Popular" tab is selected. A teal arrow points from the "Join" button towards the "Post Comment" area of the second mockup.

	Post Title	Date	Author	Category	Comments
12	"Lorem Ipsum dolor sit amet"	13.12.2022	by olli	SMA	3 Comments
8	"Lorem Ipsum dolor sit amet"	3.1.2023	by olli	Autos	12 Comments
5	"Lorem Ipsum dolor sit amet"	20.5.2021	by olli	Reisen	3 Comments
1	"Frohes neues Jahr"	1.1.2023	by olli	Politik	2 Comments
1	"Lorem Ipsum dolor sit amet"	30.11.2022	by olli	Reisen	3 Comments

/ui/posts/{id}

The page title is "[Browse Posts]". It shows a single post with the title "Frohes neues Jahr". Below the post, there is a "write a comment" input field and a "Post Comment" button. A teal arrow points from the "Post Comment" button towards the "Reply" button of a comment in the list.

Post Content:

1.1.2023 | by olli | Politik | 2 Comments
 Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptum. At vero eos et accusam et justo duo dolores et ea rebum. Stet etiam kant gubernem no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam.

Comment Section:

- 1. "Frohes neues Jahr!" (by olli | 1.1.2023) - reply
- 2. "Cooles Beitrag!" (by olli | 3.1.2023) - reply
- 3. "Hallo, ich antworte auf dein Comment" (by olli | 3.1.2023) - reply

/ui/register/

Registrieren

olli

Submit Registration

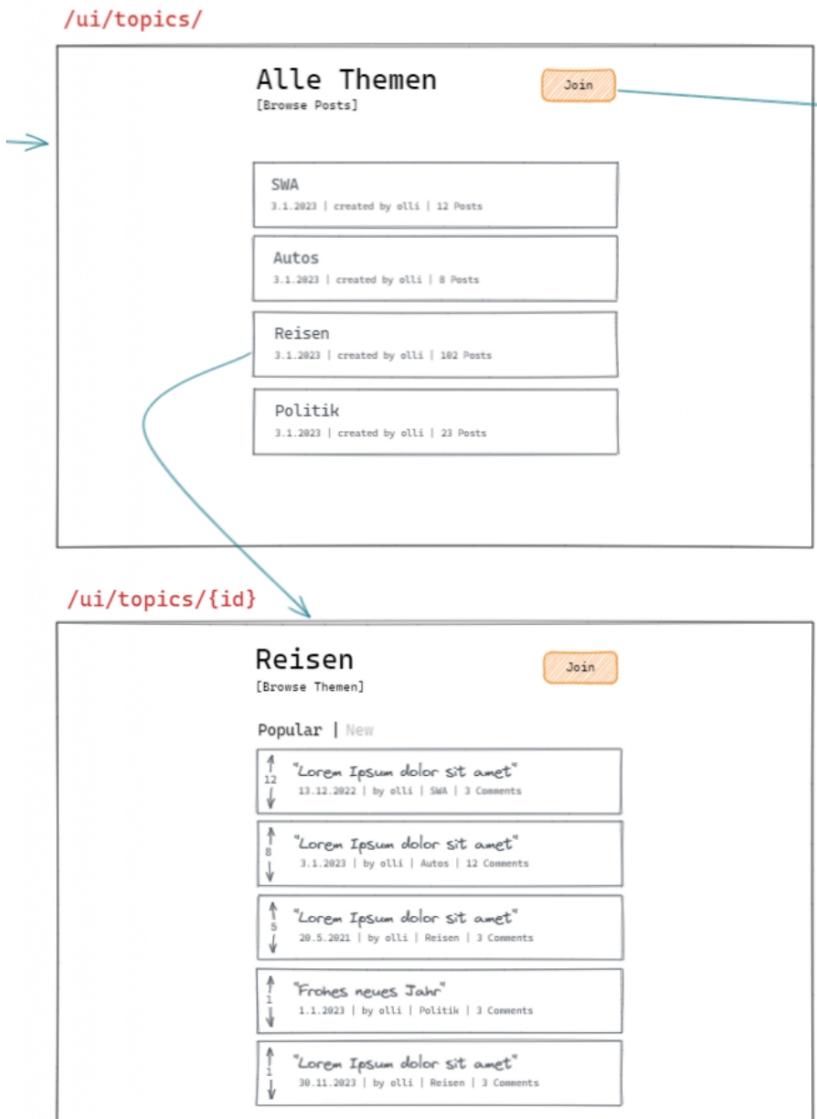
/ui/profile/me

Olli

Posts | Comments | Upvoted | Downvoted

↑ "Frohes neues Jahr"
↓ 1 1.1.2023 | by olli | Politik | 3 Comments

↑ "Lorem ipsum"
↓ 89 1.1.2023 | by olli | Politik | 16 Comments



Anhang 2: Entwurf der REST-Endpunkte

Ressource/ Verben	Pfad	Beschreibung
Comments		
GET	/comments	get all Comments
GET	/comments/{id}	get Comment by Id
POST	/comments	comment on Post
POST	/comments/{id}	reply to Command Id
DELETE	/comments/{id}	delete Command Id
Posts		
GET	/posts	get all Posts
GET	/posts/{id}	get post by Id
POST	/posts	create Post
PATCH	/posts/{id}	update Post Id
DELETE	/posts/{id}	delete Post id
Topics		
GET	/topics	get all Topics
GET	/topics/{id}	get Topic by Id
POST	/topics	create Topic
DELETE	/topics/{id}	delete Topic Id
Users		
GET	/useres/{id}	get User by Id
POST	/useres	create User
DELETE	/useres/{id}	delete User Id
Votes		
GET	/votes	get all Votes
POST	/votes	create Vote
DELETE	/votes/{id}	delete Vote Id

Eidesstattliche Erklärung

Hiermit erklären wir an Eides statt, dass wir die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt 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, 17.02.2023

Ort, Datum



Lorenzo Battiston

Osnabrück, 17.02.2023

Ort, Datum



Oliver Schlüter