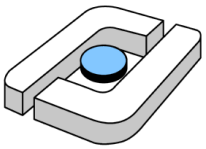

# Hochschule Osnabrück

University of Applied Sciences

## Fakultät
## Ingenieurwissenschaften und Informatik

Schriftliche Ausarbeitung zum Thema:

## Erstellung einer Rest-API mit Quarkus

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


| | |
|---|---|
| Autor: | Benno Steinkamp |
| Matr.-Nr.: | 855624 |
| E-Mail: | benno.steinkamp@hs-osn-mabrueck.de |

| | |
|---|---|
| Autor: | Christoph Freimuth |
| Matr.-Nr.: | 693714 |
| E-Mail: | christoph.freimuth@hs-osnab-rueck.de |

Themensteller: Prof. Dr. Rainer Roosmann


Abgabedatum: 07.08.2021

# Inhaltsverzeichnis

| Kapitel | Verantwortlicher |
|---|---|
| 1 | Benno Steinkamp, Christoph Freimuth |
| 2 + 2.1 | Christoph Freimuth |
| 2.2 | Christoph Freimuth |
| 2.3 | Christoph Freimuth |
| 2.4 | Benno Steinkamp |
| 2.5 | Benno Steinkamp |
| 2.6 | Benno Steinkamp |
| 2.7 | Christoph Freimuth |
| 2.8 | Benno Steinkamp |
| 2.9 | Christoph Freimuth |
| 3 + 3.1 | Christoph Freimuth |
| 3.2 | Christoph Freimuth |
| 3.3 | Christoph Freimuth |
| 3.4 | Benno Steinkamp |
| 3.5 | Benno Steinkamp |
| 3.6 | Benno Steinkamp |
| 3.7 | Benno Steinkamp, Christoph Freimuth |
| 3.8 | Benno Steinkamp |
| 4 | Benno Steinkamp, Christoph Freimuth |

# Abbildungsverzeichnis

# Tabellenverzeichnis

## Source-Code Verzeichnis

## Abkürzungsverzeichnis

| | |
|---|---|
| CDI | Context and Dependency Injection |
| DDD | Domain Driven Design |
| DI | Dependency Injection |
| DTO | Data-Transfer-Object |
| ECB | Entity-Controller-Boundary Pattern |
| ERD | Entity-Relationship Modell |
| Java EE | Java Enterprise Edition, in der Version 7 |
| JDBC | Java Database Connectivity |
| JPA | Java Persistence API |
| JPQL | Java Persistence Query Language |
| JSR380 | Java Bean Validation 2.0 |
| JVM | Java Virtual Machines |
| ORM | Object-Relational Mapping |
| SWA | Software Architektur |
| DBMS | Database Management System |
| UI | User-Interface |

# 1 Einleitung

Wirft man einen Blick in das Internet, findet man viele Angebote für kostenlose und kostenpflichtige APIs für verschiedenste Zwecke. Viele dieser APIs sind nicht gut strukturiert und für Entwickler daher schwer zu nutzen. Daher ist es für die Entwicklung einer nutzerfreundlichen und leicht erweiterbaren API wichtig, das Design einer API auf standardisierte Vorgehensweisen und Patterns aufzubauen. Durch diese Projektarbeit soll untersucht werden, ob diese Anforderungen mit Hilfe des Framesworks Quarkus umsetzbar sind.

## 1.1 Vorstellung des Themas

Die Aufgabe dieses Projektes ist es, eine Room-Planner-Applikation zu entwickeln, die es einem Internetnutzer ermöglicht, Grundrisse für einzelne Zimmer, sowie Templates für Möbel zu erstellen. Der Nutzer soll in der Lage sein, diese erstellten Pläne und Templates über ein UI einfach und schnell verwalten zu können. Die Verwaltung beinhaltet Standardfunktionalitäten wie das Editieren, Löschen und Erstellen von Plänen und Möbel. Des Weiteren soll der User über verschiedene Filtermöglichkeiten nach bestimmten Plänen und Möbel suchen können.

Um eine gewisse Sicherheit zu gewährleisten, kann die Applikation nur von angemeldeten Usern genutzt werden. Dafür soll ein simples Login- und Registrierungsystem über einen Nutzernamen und Passwort bereitgestellt werden.

## 1.2 Ziel der Ausarbeitung

Das Ziel dieser Hausarbeit ist es, eine funktionsfähige und gut designte Rest-API mit einer Datenbankanbindung und einem Front-End für Nutzer bereitzustellen. Das Design der Applikation soll auf den in der Vorlesung *Software Architektur – Konzepte und Anwendungen* besprochenen Konzepten DDD und ECB-Pattern basieren. Die Authentifizierung soll über die *Quarkus Security Architecture* realisiert werden. Das UI soll mithilfe der Templating-Engine *Qute* umgesetzt werden, damit sichergestellt ist, dass ein Nutzer auch visuell mit der API interagieren kann. Die Persistierung der Daten soll unter Verwendung von JPA über eine relationale Datenbank realisiert werden. Abschließend soll die Anwendung über verschiedene Möglichkeiten getestet und dokumentiert werden. Als Testumgebungen sollen JUnit und Swagger-UI genutzt werden.

## 1.3 Aufbau der Hausarbeit

Zu Beginn dieser Ausarbeitung werden unter anderem alle genutzten und grundlegenden Konzepte, Vorgehensweisen und Frameworks genauer erläutert. Zum Start werden die zugrunde liegenden Design-Patterns DDD und das ECB-Pattern genauer untersucht und erklärt. Im Anschluss wird das verwendete Framework Quarkus und verschiedene Konzepte, wie CDI, JPA und Java Bean Validation näher beschrieben. Überdies wird

dann sowohl die verwendete Datenbank h2, als auch die Authentifizierung über Quarkus Security erläutert. Am Schluss des Kapitels Grundlagen werden die genutzten Testumgebungen und die Templating-Engine *Qute*, die für die Realisierung des UIs verwendet wurde, ausgeführt.

Im Hauptteil der Ausarbeitung werden dann die erwähnten Grundlagen auf die Applikation angewendet. Anfangs wird die Designphase und der Aufbau der Anwendung fokussiert und genauer dargestellt. Basierend darauf werden dann die Module im Einzelnen beschrieben und gezeigt, wie Designkonzepte umgesetzt wurden. Anschließend wird unter Bezugnahme auf JPA, h2 und Java Bean Validation sowohl die Persistierung und Validierung von Daten als auch die Konfiguration der Datenbank in der Anwendung erläutert.

Zum Schluss der Ausarbeitung wird das Projekt noch einmal zusammengefasst und ein abschließendes Fazit formuliert.

# 2 Darstellung der Grundlagen

## 2.1 Grundlegende Design-Pattern

Das grundlegende Design dieses Projektes basiert auf einer Kombination des ECB-Patterns und des DDD. Der Fokus beim DDD liegt vor allem auf den Komponenten des Model-Driven Design. Dieses Vorgehen ist laut [1] eine gute Methode, um komplexe Software in eine übersichtliche Struktur zu bringen und damit das Design möglichst klar zu gestalten.

### 2.1.1 Layered Architecture

Komplexe Software besteht häufig aus vielen Komponenten wie UI, Datenbanken, Steuerungselementen usw. mit vielen Zeilen Code. Aus Zeitersparnis werden diese Komponenten oft in wenige Klassen aufgeteilt, um so in kurzer Zeit eine lauffähige Software zu erhalten. Dadurch entsteht sehr schnell eine unübersichtliche Struktur, in der z.B. eine Änderung an der UI mögliche Auswirkungen auf die Business-Logik der Software hat. Desweitern müssen Änderungen oft an mehreren Stellen durchgeführt werden, damit die Software lauffähig bleibt.

Um diese Problematiken zu umgehen, wird im DDD das *Layered-Architecture-Pattern* genutzt. Ziel dieses Patterns ist es, die Abhängigkeiten zwischen den Komponenten so zu isolieren, dass Komponenten nur noch für spezielle Aufgabebereiche zuständig sind. Dafür wird die Software in verschiedene Schichten unterteilt. Das Prinzip dieser Aufteilung ist es, dass alle Komponenten innerhalb einer Schicht kohärent sein müssen und diese nur von der genau darunterliegenden Schicht abhängig sein dürfen [1]. Jede dieser Schichten bildet nun einen Aufgabenbereich der Software ab. Zusammen bilden diese Schichten die Grundstruktur eines Softwaremoduls.

### 2.1.2 Modules

Module realisieren ein Layering auf Basis einer fachlichen Aufteilung der Software. Durch diese Aufteilung erreicht man eine hohe Kohäsion innerhalb eines Moduls und gleichzeitig eine lose Kopplung zwischen einzelnen Modulen [1]. Dadurch ist es möglich, früh in der Designphase eines Projektes in Kombination mit der *Layered Architecture* eine klare und greifbare Struktur der Software zu erstellen.

### 2.1.3 Entities

Unter Entitäten versteht man im DDD Objekte, die Dinge der Realität mit einem Zustand und einem Verhalten modellieren. Damit jede Entität eindeutig bestimmt werden kann, besitzt jede eine eindeutige Identität. Dadurch ist sichergestellt, dass Entitäten nicht über ihre Attribute bestimmt werden, da diese auch oftmals gleich sein können [1]. Als praktisches Beispiel kann die Identifikation von Studierenden genutzt werden. Zwei Studierende können den gleichen Namen besitzen, aber werden dennoch eindeutig über ihre Matrikelnummer identifiziert.

### 2.1.4 Service

Services werden dann gebraucht, wenn wichtige Funktionalitäten z.B. keiner Entität zugeordnet werden können. Tritt dieser Fall ein, wird ein Service-Objekt eingeführt, das verschiedene Funktionen für bestimmte Aufgaben anbietet [1].

### 2.1.5 Repository

Sobald eine Software große Mengen an Daten verarbeiten muss, werden Datenbanken für die Persistenz benötigt. Im DDD bildet das Repository die Schnittstelle zwischen der Software und der Datenbank ab. Diese Schnittstelle kapselt den kompletten Datenbankverkehr mit der Software und kann Daten einer Datenbank lesen, schreiben, ändern oder löschen. Mit Hilfe von Filtern ist es auch möglich, Objekte in einer Datenbank zu suchen und diese zurückzugeben [1].

### 2.1.6 ECB-Pattern

Das ECB-Pattern ist eine Abwandlung des *Model-View-Controller Patterns* und eine Spezifikation des *Layered-Architecture Patterns*. Es besteht aus vier Ebenen, welche in der folgenden Abbildung dargestellt werden [2].

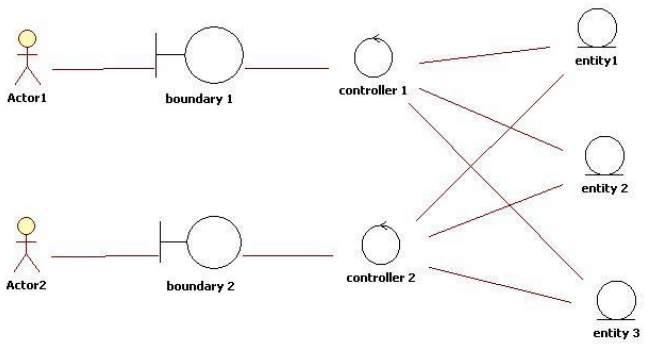


Abbildung 1: Ebenen des ECB-Patterns

Die in Abbildung 1 dargestellten vier Ebenen können als einzelne Layer der *Layered-Architecture* interpretiert werden. Neben den einzelnen Ebenen werden auch ihre Abhängigkeiten angezeigt. Der Actor verkörpert den Nutzer einer Software. Die Boundary stellt den Eintrittspunkt für den Nutzer dar und ist in den meisten Fällen eine Benutzeroberfläche, mit der der Nutzer interagiert. Durch diese Boundary werden Eingaben an die Control-Ebene der Software weitergegeben und Informationen aus der Software an den Nutzer zurückgegeben. Die Controller sind das Bindeglied zwischen den Boundaries und den Entitäten. Sie steuern die gesamte Applikation und verarbeiten Anfragen, die aus den Boundaries kommen. Entitäten sind, ähnlich wie beim allgemeinen DDD, Objekte, die eine eindeutige Identität und verschiedene Attribute besitzen und in den meisten Fällen persistiert werden [2].

Der Aufbau jedes fachlichen Moduls dieses Projektes setzt dieses ECB-Pattern um.

## 2.2 Quarkus

„Quarkus ist ein Kubernetes-natives Java Framework für Java Virtual Machines (JVMs) […] Es wurde speziell für beliebte Java-Standards, Frameworks und -Bibliotheken wie Eclipse MicroProfile und Spring […] sowie für Apache Kafka, RESTEasy (JAX-RS), Hibernate ORM (JPA), Spring, Infinispan, Camel und viele mehr konzipiert.“ [3]

Das Framework bietet ein hohes Maß an Benutzerfreundlichkeit und unterschiedliche Zusatzfunktionen, die speziell für Entwickler konzipiert wurden [3]. Zusätzlich wurde Quarkus nach dem Container-First-Prinzip entwickelt. Dadurch können Anwendungen, die mit Quarkus realisiert wurden, effizient durch schnelle Startzeiten und einen niedrigen Speicherverbrauch gehostet werden [3].

## 2.3 CDI – Context and Dependency Injection

*Context and Dependency Injection* ist ein Standard in Java, um Komponenten über eine typsichere, aber dennoch lose Kopplung miteinander zu verknüpfen [4]. Dieser Standard ist seit Java EE 6 ein fester Bestandteil der Programmiersprache und ermöglicht es den Entwicklern über simple Annotationen Context und Dependency Injection zu nutzen [5]. Es gibt drei Arten von Dependency Injection, die man unterscheiden muss:

- Field-Injection: Bei dieser Art von Injection wird der Injection-Point auf der Ebene von Felder bzw. Attributen einer Klasse gesetzt [6]. Diese Injection-Methode wird häufig genutzt, um Services in andere Komponenten zu injizieren.

- Constructor-Injection: Im Gegensatz zu der Field-Injection wird hier der Injection-Point direkt auf Ebene des Konstruktors einer Klasse gesetzt. Der CDI-Container

holt sich dann alle nötigen Referenzen und injiziert diese dann in den Konstruktor der Klasse. Wichtig hierbei ist, dass pro Klasse nur ein Injection-Point auf Ebene des Konstruktors existieren darf [6].

- Method-Injection: Hierbei handelt es sich um eine ähnliche Injection-Methode wie bei der Constructor-Injection. Der Injection-Point wird hier auf Ebene einer Methode gesetzt.

Die zweite Art der Injection ist die Context-Injection. Dabei handelt es sich um die Verwaltung von Lebenszyklen einzelner Komponenten über Kontexte bzw. Scopes. CDI definiert verschiedene Scopes, um Context-Injection zu ermöglichen.

- *@ApplicationScoped*: Während der gesamten Laufzeit des Programms existiert genau eine Instanz der CDI-Bean [6].
- *@SessionScoped*: Der CDI-Bean existiert genau für die Dauer einer Http-Session [6].
- *@RequestScoped*: Der CDI-Bean existiert genau für die Dauer eines Http-Requests [6].
- *@ConversationScoped*: Der CDI-Bean lebt für die Dauer einer vorher definierten Zeit in einer Applikation [6].

Außerdem stellt das CDI-Framework sogenannte Pseudo-Scopes zur Verfügung.

- *@Dependent*: Bei diesem Scope wird immer ein neues Objekt erzeugt und injiziert. Das erzeugte Objekt übernimmt dann den Scope des Hosts [6].
- *@Singleton*: Ähnlich wie bei *@ApplicationScoped* existiert immer nur eine Instanz der Singleton-Klasse. Der Unterschied liegt darin, dass ein Singleton immer direkt verwendet und nicht über Proxy-Objekte referenziert wird, wie es bei den oberen vier Scopes der Fall ist [6].

### 2.3.1 Dependency Injection in Quarkus

Wie bereits besprochen geht es bei der Dependency Injection darum, Abhängigkeiten einzelner Komponenten so zu gestalten, dass diese typsicher und so lose wie möglich gekoppelt sind. Quarkus bietet eine Teilmenge an Funktionen des CDI-Standards an, um diese Injections möglich zu machen.

```java
@ApplicationScoped
public class CounterBean {

    @Inject
    CounterService counterService; 1

    void onMessage(@Observes Event msg) { 2
    }
}
```

Snippet 1: Beispiel Field-Injection in Quarkus [7]

Dieser kleine Code-Ausschnitt zeigt eine simple Umsetzung einer Field-Injection in Quarkus. Auffällig ist, dass jede Klasse, die CDI nutzen will, mit einem Kontext bzw. Scope annotiert werden muss. In diesem Fall wird die Klasse *CounterBean* mit dem Kontext *@ApplicationScoped* annotiert. Mit der 1 gekennzeichnet, sieht man die eigentliche Field-Injection in Quarkus. Hierbei wird einfach eine Variable der Klasse, in diesem Fall *CounterService,* mit einem *@Inject* annotiert. Bei dieser Methode ist es außerdem wichtig darauf zu achten, dass alle Attribute, die so injiziert werden, package-private sein müssen. Dieser Schritt ist erforderlich, da Quarkus-DI nur über die Nutzung von Reflection auf private Attribute zugreifen kann und Reflection in Quarkus auf ein Minimum reduziert werden soll. Analog zu dieser Lösung funktionieren auch die *Constructor-Injection* und die *Method-Injection* in Quarkus. Im weiteren Verlauf dieses Projektes wird diese Injection-Methode am häufigsten verwendet, um Services in verschiedenen Komponenten einzubinden und Nutzen zu können.

### 2.3.2 Context Injection in Quarkus

In der Einführung dieses Kapitels wurde die Grundfunktionalität der Context-Injection bereits erläutert. In Quarkus wird diese From der Injection, wie in Snippet 1 bereits dargestellt, mithilfe von Scope-Annotationen an den Klassen realisiert. Quarkus unterstützt die bereits bekannten Built-In-Scopes von CDI bis auf *@ConversationScoped*. Weiterhin ünterstützt Quarkus auch Pseudo-Scopes wie z.B. *@Singleton*.

In Snippet 13 ist eine Umsetzung der Context-Injection mit normalen Scopes und Pseudo-Scopes in Quarkus dargestellt. Der *AmazingService* wird als Singleton annotiert und damit direkt in der Klasse *PingResource* unter Einsatz der *@Inject*-Annotation instanziiert. Im Gegensatz dazu wird die Klasse *CoolService* mit einem normalen Scope annotiert. Dies hat zur Folge, dass dieser Service in der *PingResource*-Klasse über einen Client-Proxy injiziert und nicht, wie beim Singleton, direkt instanziiert wird. Die

Instanziierung von *CoolService* erfolgt erst bei dem ersten Aufruf des injizierten Proxy unter Punkt 3 der Abbildung.

## 2.4 JPA und Hibernate ORM

JPA ist eine Spezifikation, welche es ermöglicht Daten mit Hilfe von Annotationen in einer Relationalen Datenbank zu speichern. Eine Implementation dieser Spezifikation ist des Hibernate ORM. Die Folgenden Annotationen sind in dieser Anwendung besonders zu beachten.

- *@Entity*: Mit dieser Annotation wird eine Klasse als persistierbar markiert, das heißt sie kann in eine Tabelle der Datenbank geschrieben werden.

- *@Id*: Mit dieser Annotation wird ein Member der Entität als identifizierend gekennzeichnet. Er muss innerhalb der Entität für jede Instanz einzigartig sein und die zugehörige Spalte wird in der Datenbank mit einem Primary-Key-Constraint versehen.

- *@Table*: Mit dieser Annotation kann kontrolliert werden in welche Tabelle der Datenbank eine Entität übertragen wird.

- *@GeneratedValue*: Mit dieser Annotation wird gekennzeichnet, dass der markierte Member durch die Datenbank generiert werden soll.

- *@ManyToOne*: Mit dieser Annotation wird eine Collection von Objekten durch eine entsprechende Many-To-One-Relation in der Datenbank gespeichert. Zu beachten ist hierbei der *mappedBy*-Wert der Annotation, welcher angibt durch welches Attribut in der Entität auf der Many-Seite der Relation die Zugehörigkeit zu dieser Entität gekennzeichnet wird. Der hier eingetragene Member der anderen Entität muss mit @OneToMany gekennzeichnet sein.

- *@OneToMany*: Diese Annotation stellt das Gegenstück zu *@ManyToOne* Annotation auf der Many-Seite der Relation dar. Dies führt in der Datenbank dazu, dass der Primärschlüssel der Entität auf der Many-Seite der Relation in die Tabelle der Entität auf der One-Seite als Fremdschlüssel eingefügt wird.

- *@Embeddable*: Auch Mit dieser Annotation wird eine Klasse als persistierbar markiert, in diesem Fall kann sie aber nicht allein persistiert werden, sondern nur als Member einer Klasse, welche mit *@Entity* oder *@Embeddable* markiert ist. Dieser Member sollte außerdem mit *@Embeded* markiert werden. Das führt in

der Datenbank dazu, dass alle Spalten des eingebetteten Objektes in die Tabelle der einbettenden Entität übernommen werden.

- *@Immutable*: Diese Annotation stammt nicht aus der ursprünglichen JPA Spezifikation, sondern wird durch das Hibernate ORM bereitgestellt. Sie verhindert, dass Änderungen an einem Objekt in die Datenbank persistiert werden.

- *@NamedQuery*: Diese Annotation erlaubt das Erstellen von statischen Queries in JPQL. Diese Annotation muss vor Klassen geschrieben werden, welche mit *@Entity* gekennzeichnet sind.

## 2.5 Java Bean Validation

Java Bean Validation (auch bekannt als JSR380) ist eine Spezifikation zur Validierung von Java-Klassen mithilfe von Annotationen. Die Referenzimplementierung dieser Spezifikation ist der Hibernate Validator [8]. Diese Implementation wird in dieser Anwendung verwendet.

Die Konfiguration der Validierung geschieht über so genannte Constraints. Constraints sind Annotationen, welche den Wertbereich einer Membervariable in einer Klasse oder eines Übergabeparameters einschränken. Folgende Constraints sind in dieser Applikation besonders zu beachten:

- *@NotNull*: Mit dieser Annotation wird festgelegt, dass die markierte Variable keine null-Werte mehr enthalten darf. Sie muss also zum Überprüfungszeitpunkt eine Objektreferenz halten.

- *@NotBlank*: Diese Annotation ist für Strings. Sie deklariert, dass die markierte Variable nicht leer (also länge 0) oder null sein darf. Die Annotation kann auch für Collections von Strings verwendet werden. Bei Collections wird die Annotation dann auf jeden enthaltenen String angewendet.

- *@Positive*: Diese Annotation deklariert, dass eine markierte float-, double-, int-, long- oder char-Variable > 0 ist.

- *@Valid*: Diese Annotation deklariert ein Feld als valid. Das heißt, dass bei der Validierung der Klasse die Annotationen auf der Klasse des Feldes verwendet werden, um dieses zu Validieren.

Diese Constraints können über verschiedene Mechanismen validiert werden. Die Erste, für diese Anwendung relevante Validierungsstelle, ist zunächst die Validierung der Übergabeparameter beim Aufruf von Funktionen auf durch CDI injizierte Objekte [9]. Die Zweite relevante Stelle ist die Validierung von Entitätsklassen, wenn diese durch das Hibernate ORM in die Datenbank geschrieben werden.

Es ist auch möglich, eigene Constraints zu definieren. Dazu muss zunächst das @interface deklariert werden, mit welchem eine Variable für die Validierung mit diesem Constraint markiert werden kann. Dieses @inteface muss mit der *@Constraint*-Annotation markiert werden. Außerdem muss dann eine Validierungsklasse erzeugt werden, welche das ConstraintValidator-Interface spezifiziert implementiert. Die Template-Parameter werden dabei durch das zu Validierende @interface und den Typ der zu validierenden Klassen ersetzt. Im @interface muss außerdem noch der „validatedBy“-Wert der *@Constraint*-Annotation auf die Validierungsklasse gesetzt sein.

## 2.6 Datenbank h2

Als Datenbank wurde die H2 Database Engine [10] verwendet. Diese Zeichnet sich durch einen besonders geringen Speicherverbrauch aus (siehe Tabelle 1). Der entsprechende JDBC-Treiber wird über die Quarkus-Extension „jdbc-h2“ installiert.

| | H2 | Derby | HSQLDB | MySQL | PostgreSQL |
|---|---|---|---|---|---|
| Pure Java | Yes | Yes | Yes | No | No |
| Memory Mode | Yes | Yes | Yes | No | No |
| Encrypted Database | Yes | Yes | Yes | No | No |
| ODBC Driver | Yes | No | No | Yes | Yes |
| Fulltext Search | Yes | No | No | Yes | Yes |
| Multi Version Concurrency | Yes | No | Yes | Yes | Yes |
| Footprint (embedded) | ~2 MB | ~3 MB | ~1.5 MB | — | — |
| Footprint (client) | ~500 KB | ~600 KB | ~1.5 MB | ~1 MB | ~700 KB |

Tabelle 1: Speicherverbrauch verschiedener Datenbanken [8]

Die Datenbank kann im In-Memory, Embedded oder Server Mode verwendet werden. Der Modus wird entsprechend der JDBC-Url gewählt [11]. Mit der Einstellung quarkus.datasource.jdbc.url kann die verwendete JDBC-Url konfiguriert werden [12].

## 2.7 Autorisierung und Authentifizierung via Basic-/Form Based Authentication

Als Autorisierungs- und Authentifizierungssystem wird in diesem Projekt die *Quarkus Security Architecture* genutzt. Quarkus unterstützt eine Vielzahl von Erweiterungen, um Informationen zur Autorisierung und Authentifizierung zu verarbeiten [13]. Im Folgenden werden die in diesem Projekt genutzten Authentifizierungsmechanismen *Basic*- und *Form Authentication* näher erläutert.

```
quarkus.http.auth.form.enabled=true
quarkus.http.auth.form.landing-page=/plan
quarkus.http.auth.form.login-page=/user/login
quarkus.http.auth.form.error-page=/user/login/?error=true
```

Snippet 2: Konfiguration der From Authentication (vgl. resources.application.properties)

Um die *Basic Authentication* in Quarkus nutzen zu können, muss diese in der Konfigurationsdatei aktiviert werden. Wie in Snippet 2 zu sehen, kann man noch zusätzliche Konfigurationen tätigen. Man hat unter anderem die Möglichkeiten die Startseite der Applikation, sowie die Login-Seite als auch eine Error-Seite festzulegen. Zusätzlich wird auch ein sogenannter *IdentityProvider* benötigt [14]. Dieser hat die Aufgabe, die Nutzerdaten zu verifizieren und auf eine *SecurityIdentity* abzubilden. Diese *SecurityIdentity* enthält dann sowohl den Benutzernamen und das Passwort als auch Rollen und die originalen Authentisierunginformationen des Nutzers [13]. In dieser Hausarbeit wird der JPA IdentityProvider genutzt, auf den später noch weiter eingegangen wird.

Für den Login und die Registrierung wird dann die *From Based Authentification* eingesetzt. Im Gegensatz zu traditionellen Form-Authentifizierungen wird der Nutzer nicht in einer HTTP Session gespeichert, sondern in einem verschlüsselten Cookie im Browser. Das liegt daran, dass Quarkus keine geclusterten HTTP Sessions unterstützt. Jedes Cookie enthält unter anderem ein Ablaufdatum. Nach Intervallen von einer Minute wird ein neues Cookie mit einem aktualisierten Ablaufdatum generiert, solange die Session aktiv ist [14].

Wie bereits angesprochen wird nun weiter auf den *JPA-IdentityProvider* eingegangen. Dieser IdentityProvider sorgt dafür, dass Nutzerinformationen wie der Nutzername, Passwort und die zugewiesenen Rollen verifiziert, aber auch in einer Datenbank persistiert werden.

```
@Path("api/v1/plan")
@RequestScoped
@Produces(MediaType.APPLICATION_JSON)
@RolesAllowed("user")
public class PlanResource {
```

Snippet 3: Rollenspezifischer Zugriff

(vgl. de.hsos.roomplanner.plan.boundary.rest.PlanResource.class)

Dieser Ausschnitt aus der PlanResource des Projektes zeigt, wie man das *Role-Based-Access-Control* auf einzelne Ressourcen realisiert. Mit Hilfe der Annotation *@RolesAllowed* wird festgelegt, welche vordefinierte Rolle Zugriff auf einen bestimmten Endpunkt hat. In diesem Fall haben Nutzer mit der Rolle *user* Zugriff auf den Plan-Endpunkt der Anwendung [15].

```
@Entity
@UserDefinition
@NamedQuery(name = "U-
ser.find", query = "SELECT u FROM User u WHERE u.username = :username")
public class User {

    @Username
    @Id
    @NotBlank
    private String username;
    @NotBlank
    @Password(value = PasswordType.MCF)
    private String password;
    @Roles
    @ElementCollection
    private Set<String> roles;
```

Snippet 4: User Entity (vgl. de.hsos.roomplanner.user.entity.user.class)

Für die Korrekte Initialisierung des *IdentityProviders* sind vier grundlegende Annotationen nötig. Die *@UserDefinition* Annotation muss bei einer Entität genutzt werden, um dem Provider klarzumachen, dass diese Entität den Nutzer der Applikation abbildet. Zusätzlich dazu wird die Annotation *@Username* genutzt, um ein Attribut der Entität als Nutzernamen zu kennzeichnen. Die *@Password* Annotation wird genutzt, um ein Attribut als Nutzerpasswort festzulegen. Standardmäßig wird für die Verschlüsselung *bcrypt hashed* Passwörter genutzt. Zuletzt wird noch *@Roles* verwendet, um einem Nutzer eine bestimmte Rolle zuzuordnen [15]. Diese Rollen kommen dann, wie schon in Snippet 3 zu sehen war, bei den verschiedenen Endpunkten zum Einsatz.

## 2.8 Unit Tests

### 2.8.1 JUnit 5

JUnit ist ein Testframework für Unittests. Eine entsprechende Erweiterung für das Quarkus-Framework ermöglicht es, Unittests für Quarkus-Applikationen zu definieren [16]. Die Tests werden in separaten Klassen definiert. Jede Test-Klasse muss dabei mit der *@QuarkusTest* Annotation versehen werden. Jeder Testfall muss mit *@Test* annotiert werden. Ein Test kann mit der *fail*-Methode als fehlgeschlagen markiert werden, dies kann aber auch durch die Verwendung einer der durch das Framework gegebenen Vergleichsmethoden passieren.

### 2.8.2 Rest Assured

Rest Assured ist ein Framework, welches in Kombination mit JUnit dafür verwendet werden kann, Rest-Schnittstellen zu testen. Speziell steht hier die Möglichkeit im Vordergrund Antworten von Schnittstellen im JSON-Format zu testen.

## 2.9 Quarkus Qute – Templating Engine

Neben vielen anderen Funktionalitäten bringt Quarkus auch eine eigene Templating Engine mit dem Namen Qute mit. Diese Templating Engine wurde speziell für Quarkus entwickelt. Wie in anderen Templating Engines ist es möglich über erstellte Template-Dateien das *Server-Side-Rendering* zu realisieren. Hierfür werden im Developer-Modus alle Dateien, die im Pfad *src/main/resources/templates* liegen, genutzt und bei jeder Änderung sofort aktualisiert [17]. Templates können verschiedene Dateitypen sein. In diesem Projekt nutzen wir als Templates HTML-Dateien. Der Aufbau so einer Datei ist sehr ähnlich zu dem Aufbau einer normalen HTML-Datei. Der einzige Unterschied ist, dass bei einem Qute-Template sogenannte *value expressions* genutzt werden. Das sind Platzhalter, die einerseits für Objektdaten und andererseits für die Verwendung von Schleifen und/oder If-Bedingungen genutzt werden können. Diese Platzhalter werden, während das Template gerendert wird, mit Daten aus der Applikation ausgetauscht, dazu später mehr. Diese Daten müssen dem Template vorher übermittelt werden.

```
/**
 * Templates used in the templating-engine.
 */
@CheckedTemplate
public static class Templates {

    […]
```

```
        public static native TemplateInstance plan(PlanDtoDetail plan);
    }
```

Snippet 5: Genutzte Templates einer Resource (vgl. de.hsos.roomplanner.plan.boundary.http.PlanResource.class)

Zuallererst müssen die erstellten Template-Dateien dem Endpunkt, der diese Templates nutzen will, bekanntgemacht werden. Dies geschieht, wie in Snippet 5 zu sehen, über die Annotation *@CheckedTemplate* und einer darauffolgenden öffentlichen statischen Klasse. Durch die Methode *plan* innerhalb dieser Klasse wird ein Template mit dem Pfad *templates/PlanResource/plan* deklariert. Hierbei ist wichtig, dass sowohl die Template-Datei als auch der Methodenname gleich sind. Zusätzlich können noch verschiedene Übergabeparameter mitgegeben werden [17].

```
@GET
    @Path("{id}")
    public Response getPlan(@PathParam("id") long id) {
        try {

            return Response
                    .ok(Templates.plan(this.planService.find-
Plan(this.securityIdentity.getPrincipal().getName(), id)))
                    .build();
        } catch (EntityDoesNotExistException ex) {
            return Response.status(Sta-
tus.NOT_FOUND).entity(ex.getMessage()).build();
        } catch (UserNotFoundException ex) {
            return Response.status(Sta-
tus.INTERNAL_SERVER_ERROR).build();
        }
    }
```

Snippet 6: Get Request aus PlanResource (vgl. de.hsos.roomplanner.plan.boundary.http.PlanResource.class)

Gibt es dann beispielsweise einen GET-Request kann, wie in Snippet 6 dargestellt, in der Response die Template-Klasse mit ihren Methoden genutzt werden, um eine neue Instanz des Templates mit sämtlichen Daten zurückzugeben. Zu erwähnen ist, dass das Template hierbei nicht gerendert wird. Dieser Vorgang passiert automatisch über eine spezielle Implementation eines *ContainerResponseFilters* [17].

# 3 Anwendung

Gleich am Anfang des Projektes wurde in einer ausführlichen Designphase die Grundstruktur der Applikation erstellt. Hierzu wurden in Visual Paradigm verschiedene Diagramme erstellt, um einen Überblick über das Projekt zu bekommen. Es wurde sowohl ein ausführliches Klassendiagramm als auch ein ERD und ein ORM-Diagramm erstellt, um die Datenbankstruktur zu visualisieren.

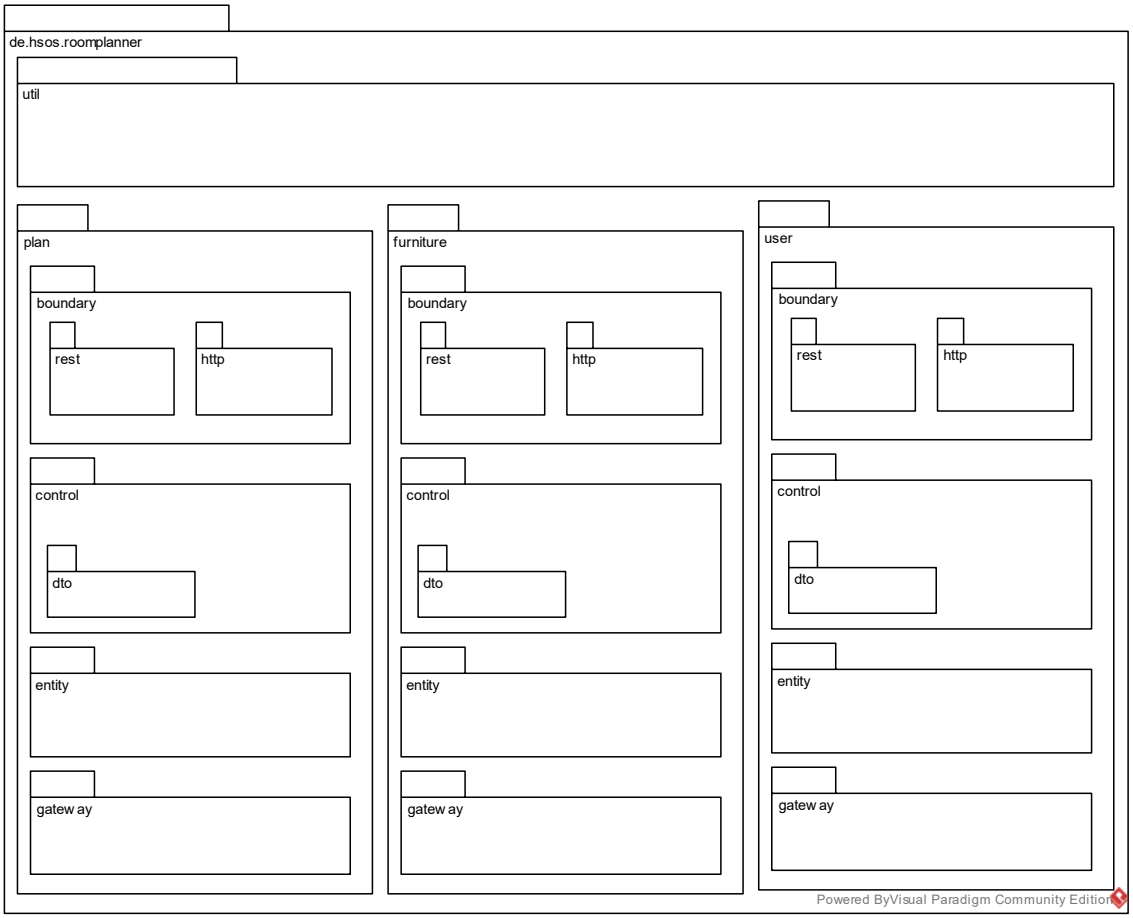


Abbildung 2: Strukturdiagramm

Die grobe Struktur unserer Applikation wird in Abbildung 2 dargestellt. Es ist deutlich zu sehen, dass die Grundlage der Struktur auf dem DDD und dem ECB-Pattern basiert. Die Software wurde in drei große fachliche Module aufgeteilt. Jedes dieser Module wurde unter Anwendung des ECB-Patterns gleich strukturiert, um eine durchgehende Konsistenz zu erhalten. Die Module bestehen aus einem Boundary-, Control, Entity- und Gateway-Layer. Innerhalb des Boundary-Layers gibt es eine weitere Unterteilung in ein Rest-Paket, welches als Rest-Schnittstelle dient und einem http-Paket, welches für das *Server-Side-Rendering* genutzt wird. In diesen Paketen befinden sich unsere Endpunkte der Software. Im Control-Layer befindet sich unsere Applikations-Logik in Form von verschiedenen Services, welche bereitgestellt werden. Zusätzlich gibt es noch ein weiteres

Paket, in denen sich alle benötigten DTO´s und deren Konverter befinden. Die Entitäten der einzelnen Module befinden sich im Entity-Layer. Der Gateway-Layer und damit niedrigste Layer eines Moduls bildet den Persistence-Layer der *Layered-Architecture* ab. Hier befinden sich die Repositories eines Moduls, welches für sämtlichen Datenbankverkehr zuständig sind. Das komplette Klassendiagramm kann unter Abbildung 11 gefunden werden.

## 3.1 Modul – Plan

Das Plan-Modul hat die Aufgabe Pläne zu verwalten. Dazu gehört unter anderem das Erstellen, Verändern, Suchen und Löschen von Plänen, sowie das Hinzufügen, Verändern und Löschen eines platzierten Möbel in einem Plan.

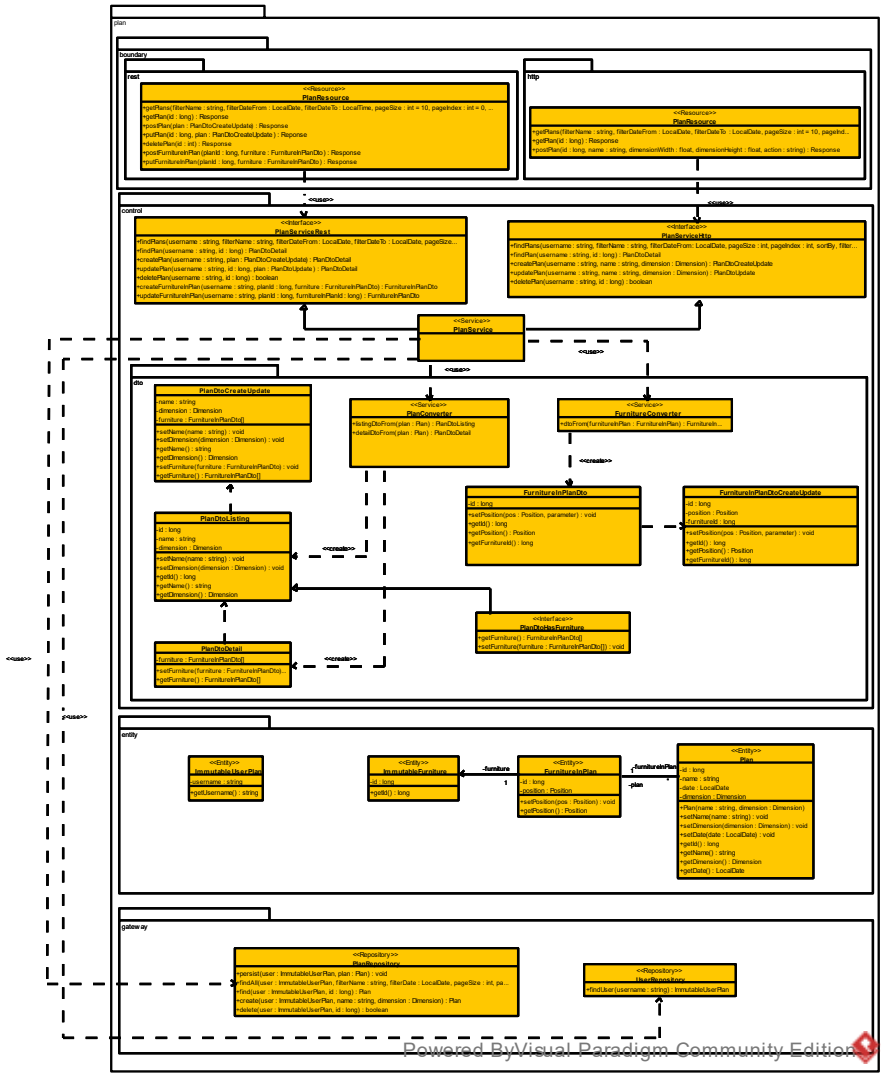


Abbildung 3: Plan-Modul

Wie bereits in der Einleitung zu diesem Kapitel erwähnt, ist dieses Modul auf Basis des ECB-Patterns in verschiedene Ebenen eingeteilt.

In der Boundary-Ebene befinden sich beide Endpunkte des Moduls. Der Endpunkt im Rest-Paket bildet den Entry-Point für sämtliche Anfragen an die Rest-Schnittstelle des Moduls. Im Gegensatz dazu befindet sich im http-Paket der Entry-Point für das *Server-Side-Rendering*. Im Folgenden werden die beiden Endpunkte des Moduls näher beschrieben.

| **/api/v1/plan** | |
|---|---|
| GET | Gibt eine Auflistung von Plänen eines Users zurück |
| GET/p_id | Gibt einen bestimmten Plan eines Users zurück |
| POST | Erzeugt einen neuen Plan für einen User |
| PUT/p_id | Ändert einen bestimmten Plan eines Users |
| DELETE | Löscht einen bestimmten Plan eines Users |
| POST/p_id/furniture | Erzeugt ein Möbel in einem Plan eines Users |
| PUT/p_id/furniture/f_id | Verändert ein Möbel in einem Plan eines Users |
| DELETE/p_id/furniture/f_id | Löscht ein Möbel in einem Plan eines Users |

Tabelle 2: PlanResource Rest

Der Aufbau des Endpunktes setzt, wie in der obenstehenden Tabelle 1 dargestellt, im Wesentlichen die bekannten CRUD-Funktionalitäten Create, Read, Update, Delete um. Des Weiteren ist es möglich, über weitere POST, PUT, und DELETE Anfragen ein Möbel in einem Plan zu platzieren, zu verändern oder zu löschen. Durch die Annotation *@RolesAllowed(„user“)* an der Klasse haben nur Nutzer mit der Rolle *user* die Möglichkeit diesen Endpunkt anzufragen.

| **/plan** | |
|---|---|
| GET | Gibt ein HTML-Dokument mit einer Liste von Plänen eines Users zurück |

| | |
|---|---|
| GET/id | Gibt ein HTML-Dokument mit einem bestimmten Plan eines Users zurück |
| POST | Erzeugt einen neuen Plan für einen User und leitet auf das Listing der Pläne weiter |
| POST/id | Löscht oder verändert einen bestimmten Plan eines Users und leitet auf das Listing der Pläne weiter |

Tabelle 3: PlanResource http

Ähnlich wie beim Rest-Endpunkt werden Methoden zum Erstellen, Löschen, Verändern und Lesen von Plänen bereitgestellt. Auffällig ist, dass nur GET und POST Anfragen verarbeitet werden. Das liegt daran, dass für Http-Forms nur GET und POST Anfragen existieren. Ein weiterer Unterschied ist, dass alle Responses dieses Endpunktes Templates in Form von HTML-Dokumenten zurückgeben, um das *Server-Side-Redndering* zu realisieren. Genau wie beim Rest-Endpunkt können nur Nutzer mit der *user*-Rolle auf diesen Endpunkt zugreifen.

Wie in Abbildung 3 zu sehen ist, wird die Applikations-Logik in der Control-Ebene über zwei Interfaces und einem Service, der diese implementiert, realisiert.

```
@RequestScoped
public class PlanService implements PlanServiceHttp, PlanServiceRest {
    […]
    @Override
    @Transactional
    public PlanDtoDetail createPlan(String username, @Valid PlanDtoCre-
ateUpdate plan) throws UserNotFoundException {
        return this.createPlan(username, plan.getName(), plan.getDimen-
sions());
    }
```

Snippet 7: PlanService (vgl. de.hsos.roomplanner.plan.control.PlanService.class)

Der *PlanService* ist mit *@RequestScoped* annotiert und die einzelnen Methoden dieser Klasse mit der Annotation *@Transactional*. Damit wird gekennzeichnet, dass auf dieser Ebene die Datenbanktransaktionen gestartet werden. Zusätzlich werden auf dieser Ebene *Data-Transfer-Objects* definiert. Zweck dieser DTO´s ist es einerseits, die Entitäten von der Boundary-Ebene zu trennen und andererseits nur für die Anfragen relevante Informationen zurückzugeben. Beispielsweise werden im *PlanDtoListing* nur die ID und das Erstellungsdatum gehalten. Das *PlanDtoDetail* hingegen hält zusätzlich noch die

Liste der Möbel in einem Plan. Durch diese Aufteilung ist sichergestellt, dass nur benötigte Informationen an den Nutzer zurückgegeben werden.

Die Entity-Ebene bildet die Business-Logik der Anwendung. Sie enthält alle Klassen, die für Business-Transaktionen benötig werden.

```
@Entity
[…]
public class Plan {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    […]
    private LocalDate date;
    private Dimension dimension;
    […]
    private List<FurnitureInPlan> furnitureInPlan;
    […]
    private ImmutableUserPlan user;
```

Snippet 8: Plan-Entity (vgl. de.hsos.roomplanner.plan.entity.Plan.class)

Wie in Snippet 8 beispielhaft dargestellt, sind alle Klassen auf der Entity-Ebene mit *@Entity* annotiert und halten alle benötigten Informationen. Außerdem besitzt jede dieser Klassen eine eindeutige Identifikation, welche mit *@Id* gekennzeichnet wird.

Die letzte Ebene in diesem Modul ist die Gateway-Ebene. Hier befinden sich alle Repository-Klassen, die für die Kommunikation mit der Datenbank benötigt werden.

```
@ApplicationScoped
public class PlanRepository {

    @Inject
    EntityManager em;
```

Snippet 9: PlanRepository (vgl. de.hsos.roomplanner.plan.gateway.PlanRepository.class)

Jedes Repository ist mit *@ApplicationsScoped* annotiert. Zusätzlich wird jedem Repository ein *EntityManager* der Java-Persistance-API injiziert. Dieser *EntityManager* wird genutzt, um Entitäten in der Datenbank zu persistieren, zu löschen, zu finden oder um Queries zu erstellen [18].

## 3.2 Modul – Furniture

Das Furniture-Modul hat die Aufgabe Möbel in der Applikation zu verwalten. Genau wie beim Plan-Modul können hier Pläne erstellt, verändert, abgefragt und gelöscht werden.

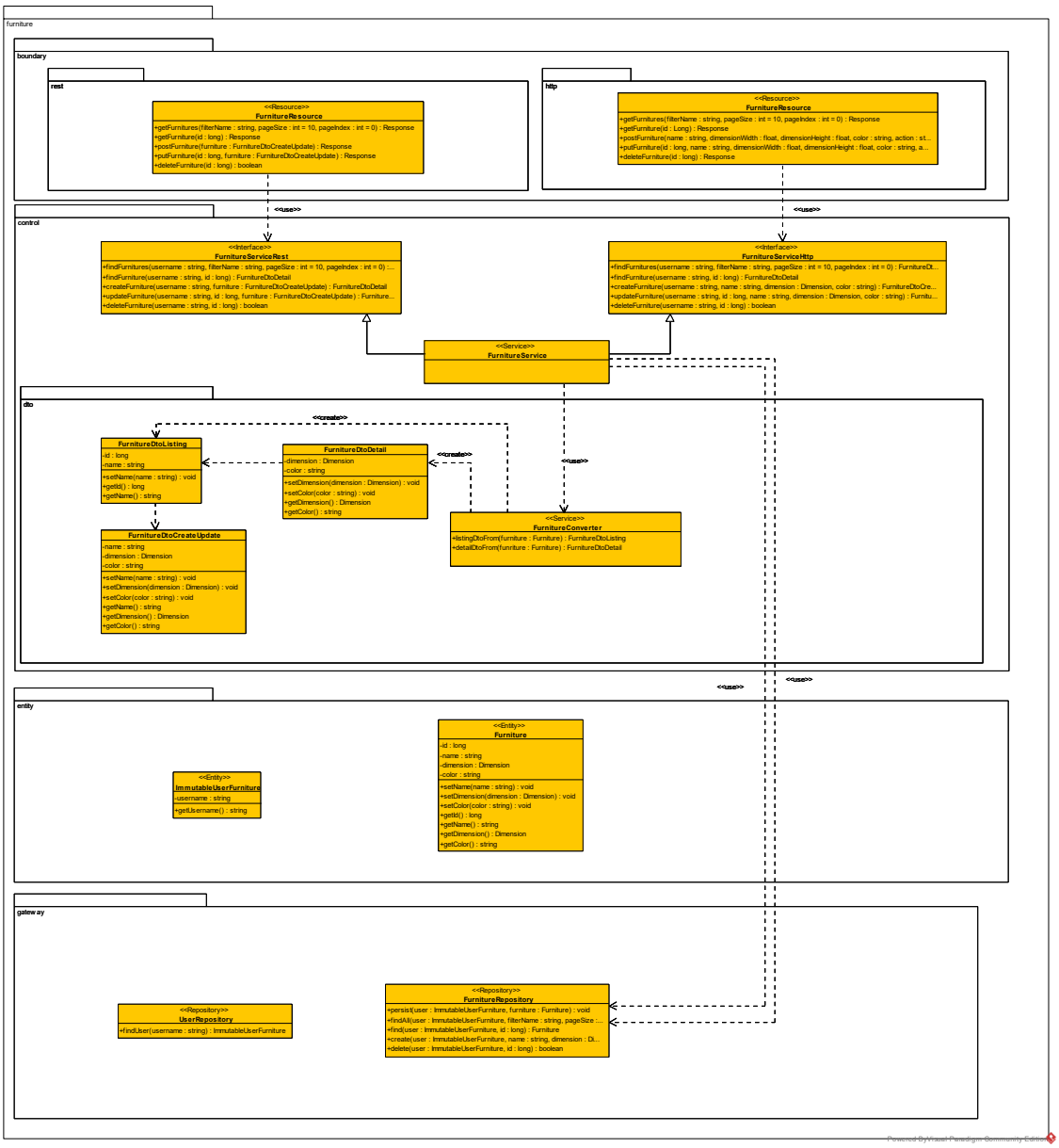


Abbildung 4: Furniture-Modul

Identisch zum Plan-Modul besitzt das Furniture-Modul auf der Boundary-Ebene insgesamt zwei gleichnamige Endpunkte für die Rest-Schnittstelle und das *Server-Side-Rendering*. Diese beiden Endpunkte werden im Folgenden näher beschrieben.

| **/api/v1/furniture** | |
|---|---|
| GET | Gibt eine Auflistung von Möbel eines Users zurück |
| GET/id | Gibt ein bestimmtes Möbel eines Users zurück |
| POST | Erzeugt ein neues Möbel für einen User |
| PUT/id | Verändert ein bestimmtes Möbel eines Users |
| DELETE/id | Löscht ein bestimmtes Möbel eines Users |

Tabelle 4: FurnitureResource rest

Der Rest-Endpunkt bietet die Möglichkeit über eine GET-Anfrage eine Auflistung von Möbel zu erhalten oder bei zusätzlicher Möbel-Id ein bestimmtes Möbel zurückzugeben. Weiterhin können Möbel erzeugt, verändert und gelöscht werden. Analog zu der Plan-Ressource können auch nur Nutzer mit der Rolle *user* diesen Endpunkt nutzen.

| **/furniture** | |
|---|---|
| GET | Gibt eine HTML-Dokument mit einer Liste von Möbel eines Users zurück |
| GET/id | Gibt ein HTML-Dokument mit einem bestimmten Möbel eines Users zurück |
| POST | Erzeugt ein neues Möbel für einen User und leitet auf das Listing der Möbel weiter |
| POST/id | Löscht oder ändert ein Möbel eines Users und leitet auf das Listing der Möbel weiter |

Tabelle 5: FurnitureResource http

Genau wie der Http-Endpunkt des Plan-Moduls verarbeitet diese Ressource nur GET und POST Anfragen. Die Methoden decken die typischen Funktionalitäten eines Endpunktes ab und geben dem Nutzer mit Hilfe von *Qute*, wie in Kapitel 2.9 beschrieben, ein visuelles Feedback.

Da sich die Control-, Entity- und Gateway-Ebene dieses Moduls nur in der Spezifikation der einzelnen DTO´s zum vorherigen Modul unterscheidet, wird dieses Modul nicht weiter erläutert.

## 3.3 Modul – User

Das User-Modul stellt die Nutzerverwaltung dieser Applikation dar. Die Verwaltung ist sehr einfach gehalten, daher ist es nur möglich, neue Nutzer über eine Registrierung mit Username und Passwort anzulegen.

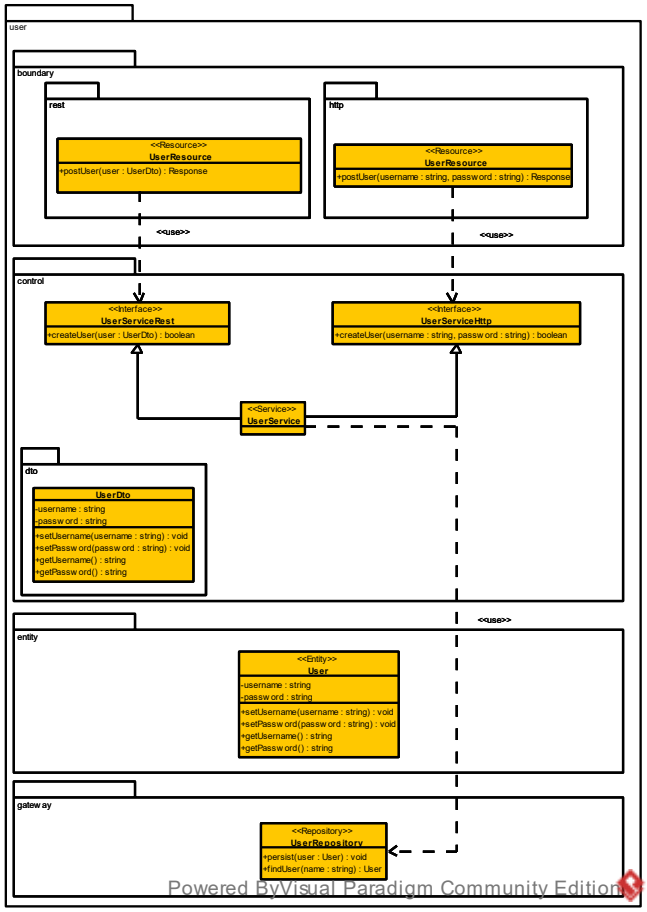


Abbildung 5: User-Modul

Dieses Modul ist sehr übersichtlich gehalten und unterstützt nur die notwendigsten Funktionalitäten, um Nutzer zu registrieren und anzulegen. Dennoch werden auch hier zwei Endpunkte bereitgestellt.

| **/api/v1/user** | |
|---|---|
| POST | Erzeugt einen neuen User |

Tabelle 6: UserResource rest

Der Rest-Endpunkt, der in Tabelle 6 dargestellt ist, kann lediglich einen neuen Nutzer hinzufügen.

| **/user** | |
|---|---|
| GET/login | Gibt ein HTML-Dokument für das Login zurück |
| GET/register | Gibt ein HTML-Dokument für das Registrieren zurück |
| POST/register | Erzeugt einen neuen Nutzer und leitet auf die Login-Seite weiter |

Tabelle 7: UserResource http

Über die Anfrage GET/login wird der Nutzer auf die Login-Seite der Applikation weitergeleitet. Wenn ein Nutzer nicht angemeldet ist, oder noch keinen Account besitzt, wird immer die Login-Seite zurückgegeben. Über GET/register wird der Nutzer auf die Registrierungs-Seite weitergeleitet und hat dann die Möglichkeit über die Eingabe eines Usernames und Passwort einen Account anzulegen. Dies erfolgt über die POST/register Anfrage.

Aus den im Furniture-Modul genannten Gründen werden die nachfolgenden Ebenen nicht weiter erläutert.

## 3.4 Konfiguration ORM

Um die Anforderung der Persistenz der Daten dieser Anwendung umzusetzen, wurden die Entitätsklassen mithilfe von JPA und Hibernate ORM in die Datenbank gespeichert.

Damit die Entitäten in die Datenbank gespeichert werden können müssen diese, wie in Kapitel 2.4 JPA und Hibernate ORM beschrieben, mit den entsprechenden JPA Annotationen versehen werden. Dabei ist die Verwendung folgender Annotationen in dieser Anwendung besonders zu beachten.

- *@ManyToOne*: Für alle Relationen wird der „cascade“-Wert der Annotation auf ALL gesetzt damit, wenn ein Plan-Objekt oder Furniture-Objekt gelöscht wird, auch alle entsprechenden FurnitureInPlan-Objekte gelöscht werden. Für die Relation zwischen Plan und FurnitureInPlan in Plan wird außerdem der *orphanRe-*

*moval*-Wert der Annotation auf *true* gesetzt. Dies führt dazu, dass, falls ein *FurnitureInPlan*-Objekt aus der Furniture-Collection eines Plans entfernt wird, dieses Objekt gelöscht wird.

- *@Embeddable*: Mit dieser Annotation sind die Klassen Dimension (Snippet 15), Color (Snippet 14) und Position (Snippet 16) gekennzeichnet. Hierbei handelt es sich um klassische Objekt-Values.

- *@Embeded*: Mit dieser Annotation werden Membervariablen der Typen Dimension, Color und Position gekennzeichnet. Dies geschieht in den Klassen Plan, FurnitureInPlan und Furniture.

- *@Table*: Mit dieser Annotation wurden die Klassen ImmutableFurniture, ImmutableUserPlan, ImmutableUser versehen, welche auf Tabellen von Entitäten eines anderen Moduls verweisen. Der Grund hierfür wird später im Kapitel erläutert.

- *@Immutable*: Mit dieser Annotation wurden auch die Klassen ImmutableFurniture, ImmutableUserPlan, ImmutableUser versehen, damit diese unveränderbar sind. Auch der Grund hierfür wird später im Kapitel erläutert.

- *@NamedQuery*: Mit dieser Annotation werden die Queries definiert mit denen die Entitäten abgefragt werden können.

Das dadurch entstandene ERM befindet sich in Abbildung 9. Dort ist auch zu erkennen, dass die strikte Trennung der Module nach DDD auf der Datenbankebene nicht mehr existiert. Es wurde sich bewusst dazu entschieden die Kohärenz der Daten zwischen den Modulen mithilfe der Constraints innerhalb der Datenbank zu gewährleisten. Die Trennung der Module innerhalb des Programmes soll aber trotzdem gewährleistet werden. Um dies zu erreichen, werden die jeweiligen Referenzen nicht auf die tatsächliche Klasse gemacht, sondern auf neue Entitäten, welche auf dieselbe Tabelle in der Datenbank gespeichert werden. Dies kann aber zu einigen Problemen führen:

Da der Entity Manager die beiden Entitäten nun als unabhängig ansieht, werden Änderungen auf dem einen Element nicht sofort auf das andere übertragen. Dieses Problem wird dadurch gelöst, dass die neu erstellte Entität immutable ist, sich also nicht verändern lässt. Dies wird einerseits dadurch erreicht, dass die einzige Membervariable der Entitäten immer privat ist und für sie nur ein Getter existiert und andererseits dadurch, dass die Entitäten mit der Annotation *@Immutable* gekennzeichnet ist.

Offene Änderungen an einer Entität werden nicht in die Datenbank geschrieben, wenn die andere Entität durch *EntityManager.find* geladen wird. Dieses Problem wird durch die Nutzung von JPQL Queries umgangen.

Wenn in einer Transaktion sowohl die originale Entität als auch die Referenz geändert wird, wirft Hibernate eine *OptimisticLockException*. Dies wird einerseits dadurch umgangen, dass die Referenz unveränderbar ist. Aber auch dadurch, dass neue Transaktionen immer in der Control-Schicht der Module geöffnet und geschlossen werden und es zwischen den einzelnen Modulen keine Abhängigkeiten gibt ist es unmöglich das in einer Transaktion sowohl auf das Original als auch die Referenz zugegriffen wird.

Die Speicherung mehrerer Klassen in derselben Tabelle der Datenbank wird, wie in Snippet 17, Snippet 18 und Snippet 19 zu sehen ist, mithilfe der `@Table` Annotation erreicht. Hierbei wird der „name“-Wert der Annotation auf den jeweiligen Tabellennamen der Originalklasse gesetzt. Also bei *ImmutableUserPlan* und *ImmutableUserFurniture* auf User und für *ImmutableFurniture* auf *Furniture*.

In Abbildung 10 sind die jeweiligen Mappings zu den Entitäten der Datenbank zu erkennen. Hier zu sehen ist auch, dass alle modulübergreifenden Abhängigkeiten sich auf Datenbank-Ebene befinden und nicht auf Entity-Ebene. (Auch zu beachten ist, dass in diesem Diagramm die doppelten Abbildungen auf die User-Tabelle und die Furniture-Tabelle nicht dargestellt sind.)

Die Abfrage einzelner Entitäten erfolgt über Named-Queries. Diese sind in JPQL geschrieben und befinden sich jeweils in den Annotationen der jeweiligen Entitätsklasse, welche durch diese Query gefunden oder bearbeitet wird.

## 3.5 Konfiguration Datenbank

Als Datenbank wurde die in 2.5 vorgestellte Datenbank h2 verwendet. Die dafür Vorgenommenen Konfiguration ist in Snippet 10 zu sehen.

```
quarkus.datasource.jdbc.url=jdbc:h2:./roomPlannerDb
```

Snippet 10: JDBC-URL Property vgl. application.properties

Die JDBC-URL konfiguriert gleich mehrere Dinge.

1. „jdbc“: Dies ist eine JDBC-URL
2. „h2“: Der JDBC-Treiber für die h2 Datenbank wird verwendet

3. „./roomPlannerDb“: Die h2-Datenbank läuft im Embedded Mode und die Datenbank wird in der Datei roomPlannerDb.mv.db im selben Ordner wie die Anwendung gespeichert.

Der Embedded Mode bedeutet, dass der JDBC-Treiber keine Verbindung zu einem externen DBMS aufbaut, sondern selbst das DBMS darstellt. Die Datenbank läuft also im selben Prozess wie die restliche Anwendung.

Jedes Plan-Objekt enthält in der Membervariable „date“ sein Erstellungsdatum. Dieses wird mit dem Java-Typ *java.time.LocalDate* abgebildet. *LocalDate* kann in Java Daten von 1.1.-999999999 bis 31.12.999999999 abbilden. In der Datenbank werden diese auf den internen typ Date gemappt, welcher mit *LocalDate* kompatibel sein sollte und durch die Dokumentation empfohlen wird [19]. Bei der Speicherung von Daten, welche sehr weit in der Vergangenheit liegen, oder Daten, welche sehr weit in der Zukunft liegen, kommt es in der Datenbank jedoch zu einem Überlauf der Datumswerte und diese werden nicht korrekt gespeichert. Um dieses Problem zu lösen, wurden die für diese Anwendung verwendeten Daten auf den Zeitraum zwischen 1.1.1900 und 31.12.9999 begrenzt.

## 3.6 Nutzung der Java Bean Validation

In der Anwendung wird die Java Bean Validation an zwei Stellen aktiv verwendet, um die Korrektheit von nutzerkontrollierten Daten zu überprüfen. Die erste Stelle ist in jedem Modul der Übergang von der Boundary-Ebene in die Control-Ebene. In den Funktionen auf Control-Ebene sind nutzerkontrollierte Parameter und DTOs mit Constraints eingeschränkt. Die Validierung geschieht dann beim Aufruf dieser Funktionen Automatisch [9]. Die zweite Stelle ist die Übertragung der Entitäten in die Datenbank. Hierfür sind alle Entitätsklassen mit passenden Constraints versehen.

Auch die in Konfiguration Datenbank 3.5 beschriebene Beschränkung der Datumseinträge auf den Bereich zwischen 1.1.1900 und 31.12.9999 (beides inklusive) geschieht über Constraints. Hierzu wurden zwei neue Constraints erzeugt @DateMin und @DateMax. Diese ermöglichen es, für eine Variable des Typen *LocalDate* ein inklusives Minimum bzw. ein Maximum anzugeben. Der Wert ist dabei standardmäßig die in *de.hsos.roomplanner.util.date.DateUtil* definierten Werte für das maximale und minimale Datum, welches erlaubt ist. Die Annotation werden jeweils durch *de.hsos.roomplanner.util.date.DateMinValidator* und *de.hsos.roomplanner.util.date.DateMaxValidator* validiert.

```java
public class ColorStringValidator implements ConstraintValidator<…> {

    private static final String PATTERN = "\\A#[0-9a-fA-F]{6}\\Z";

    @Override
    public boolean isValid(String value, …) {
        if (value == null) {
            return false;
        }
        return value.matches(PATTERN);
    }
}
```

Snippet 11: ColorStringValidator-Klasse

(vgl. de.hsos.roomplanner.util.color.ColorStringValidator)

Ebenso wurde noch ein weiteres Constraint erzeugt. Mit *@ColorString* kann eine String-Variable als html-Color-String gekennzeichnet werden. Das heißt, dass der String mit einem „#“ beginnt und dann die Rot-, Grün- und Blauwerte der Farbe als Hexadezimalzahl folgen (z.B. #ff0000 für Rot oder #000000 für Schwarz). Dieses Format wird von der Klasse *de.hsos.roomplanner.util.color.ColorStringValidator,* wie in Snippet 11 zu sehen ist, mithilfe eines regulären Ausdruckes überprüft.

## 3.7 User Interface

Im Folgenden wird der Aufbau des UI unter Verwendung der Templating Engine Qute dargestellt. Das UI besteht aus insgesamt sechs Template-Komponenten, wobei die Templates für Pläne und Möbel eine identische Struktur besitzen. Aus diesem Grund wird nur ein spezifischer Blick auf die beiden Templates der Plan-Ressource geworfen.

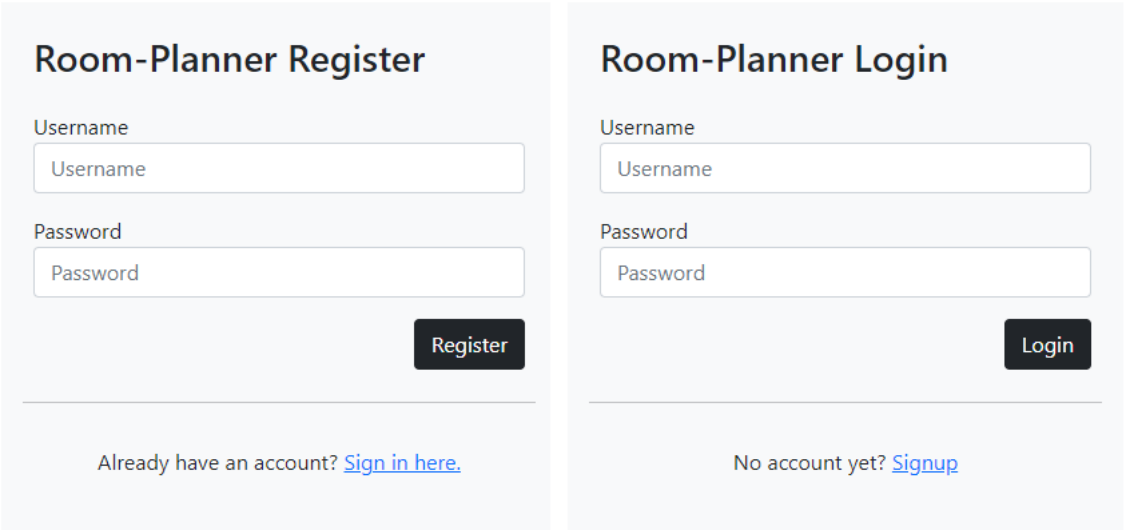


Abbildung 6: Login & Register Template

Abbildung 6 zeigt die gerenderten Darstellungen des Login- und Register-Templates. Ein Nutzer hat hier die Möglichkeit, einen Usernamen und ein Passwort für seinen persönlichen Account zu bestimmen. Der Account wird beim Login über die *Quarkus Security Architecture* authentifiziert.

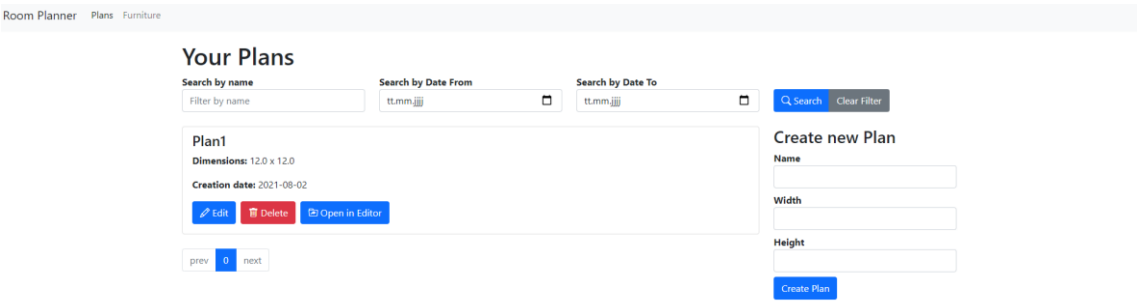


Abbildung 7: Plan-Listing-Page

Nachdem sich ein Nutzer eingeloggt hat, wird er auf in Abbildung 7 dargestellte Landing-Page weitergeleitet. Zu sehen sind vier verschiedene Bereiche. Ein Nutzer kann über die am oberen Bildrand platzierte Navigation zwischen Plänen und Möbel navigieren. Darunter befinden sich drei Input-Felder, über die ein Nutzer die Möglichkeit hat, nach bestimmten Plänen zu suchen. Das kann einerseits über den bestimmten Namen eines Plans oder über die Eingabe eines Zeitraums erfolgen. Auf der rechten Seite befinden sich die Eingabefelder, um einen neuen Plan anzulegen. Diese werden über einen Namen und zwei Parameter, Höhe und Breite, angelegt. In der Mitte des Bildschirms werden dann alle erstellten Pläne eines Nutzers untereinander aufgelistet. Hier besteht für den Nutzer die Möglichkeit, über das Klicken auf einen der Buttons eines Plans diesen zu editieren, zu löschen oder in einem Editor anzuzeigen.

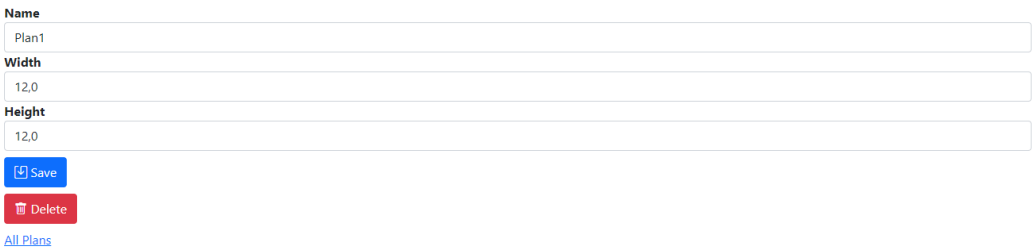


Abbildung 8: Plan-Edit-Page

Sobald ein Nutzer einen Plan ändern möchte, wird er auf die in Abbildung 8 dargestellte Seite weitergeleitet. Durch Eingabe in die drei Eingabefelder kann ein User den Namen, die Höhe und die Breite eines Plans verändern und durch einen Klick auf den Save-

Button speichern. Über den Link unter dem Delete-Button gelangt ein Nutzer auf die bereits bekannte Listing-Seite der Pläne.

## 3.8 Einbindung der Webanwendung

Zur Bearbeitung der Pläne wird eine Webapp verwendet, welche das Ergebnis der Hausarbeit im Modul Webanwendungen darstellt. Diese Webapp wird durch Louis Schraeder und Benno Steinkamp entwickelt. Zu Beginn der Planungsphase wurde deshalb die Rest-Schnittstelle zusammen mit diesem Projekt erarbeitet und spezifiziert. Dies garantiert beim Deploy der Webapp die Verträglichkeit mit diesem Backend. Der Zeitplan des Projektes der Webapp sieht eine Fertigstellung bis zum 26.09.2021 vor. Aus diesem Grund enthält dieses Projekt noch nicht die abgeschlossene Webapp, sondern lediglich einen Platzhalter. Da die Rahmenbedingungen, welche für die Integration der Webapp nötig sind, aber bereits stehen, ist dieses Projekt mit der Erfüllung dieser Rahmenbedingungen aber trotzdem abgeschlossen und vollständig.

```
public class Routes {


    @Route(methods = Route.HttpMethod.GET, regex =
        "^/roomplanner-editor-wa/.+(?<!\\.(js|css|ico|png)(\\.map)?)$"
    )
    void angularRoutes(RoutingContext ctx){
        ctx.reroute("/roomplanner-editor-wa/");
    }

}
```

Snippet 12: Routes Klasse (vgl. de.hsos.Routes)

Die Webapp basiert auf dem Framework Angular und wird daher für die Produktion zu statischen Ressourcen kompiliert. Dies Ressourcen werden unter *src\main\resources\META-INF\resources\roomplanner-editor-wa* abgelegt. Damit das Angular-Routing funktioniert, müssen Anfragen, welche nicht auf Ressourcen der Webapp zugreifen, auf das Root-Dokument der Webapp umgeleitet werden. Umgesetzt wird dies mit Hilfe von Reactive Routes, zu sehen in Snippet 12.

# 4 Zusammenfassung und Fazit

## 4.1 Zusammenfassung

Rückblickend lässt sich sagen, dass man unter Verwendung von Quarkus schnell eine gut strukturierte und nutzbare API entwickeln kann. Das liegt einerseits an der guten Dokumentation und der Fülle an Beispielen und Guides, die Quarkus zur Verfügung stellt und andererseits an der Umsetzung vieler Patterns und Standards direkt in Quarkus. So bietet Quarkus optimale Bausteine in Form von Extensions an, um eine nutzerfreundliche API entwickeln zu können. Trotz dieser Hilfsmittel, die Quarkus bietet, ist es für größere Projekte, wie dieses, notwendig, bestimmte Vorgehensweisen im Software-Design zu verinnerlichen. Durch die Verbindung des bspw. DDD in der Designphase und Quarkus in der Entwicklungsphase ist es möglich, von Beginn an eine klare und konsistente Struktur zu realisieren und beizubehalten.

## 4.2 Fazit

Wie in der Zusammenfassung bereits erwähnt, war die Nutzung des Frameworks Quarkus und dessen Extensions für die Entwicklung einer Rest-API optimal. Durch die Einarbeitungsphase innerhalb des Praktikums hatte man bereits gute Kenntnisse im Bereich Quarkus, was dem Projekt sehr geholfen hat. Durch die Wahl des Authentifizierungssystems und der Templating-Engine, welches entweder direkt in Quarkus integriert ist oder für Quarkus speziell entwickelt wurde, verliefen beide Themengebiete im Projekt reibungslos.

Abschließend lässt sich festhalten, dass man durch dieses Projekt die Wichtigkeit einer konsistenten Struktur und eines klaren Designs in einem großen Projekt, an dem auch mehrere Beteiligte arbeiten, erkannt hat.

# 5 Referenzen


[1] E. Evans, Domain Driven Design Tackling Complexity in the Heart of Software, Boston: Addison-Wesley, 2004.

[2] „The Entity-Control-Boundary Pattern," [Online]. Available: http://www.cs.sjsu.edu/~pearce/modules/patterns/enterprise/ecb/ecb.htm. [Zugriff am 28 7 2021].

[3] „Was ist Quarkus?," [Online]. Available: https://www.redhat.com/de/topics/cloud-native-apps/what-is-quarkus.

[4] „The Java EE 6 Tutorial," [Online]. Available: https://docs.oracle.com/javaee/6/tutorial/doc/giwhb.html.

[5] „An Introduction to CDI (Contexts and Dependency Injection) in Java," [Online]. Available: https://www.baeldung.com/java-ee-cdi.

[6] R. Roosmann, „Vorlesungsfolien CDI - Software Architektur - Konzepte und Anwendungen (SoSe 2021)," [Online]. Available: https://osca.hs-osnabrueck.de/lms/377663621920136/dat/Veranstaltungsunterlagen%20f%C3%BCr%20Studierende/Folien/05_SWA_CDI.pdf.

[7] „Quarkus - Context and Dependency Injection," [Online]. Available: https://quarkus.io/guides/cdi-reference.

[8] „The Bean Validation reference implementation. - Hibernate Validator," [Online]. Available: https://hibernate.org/validator/. [Zugriff am 30 Juli 2021].

[9] „Quarkus - Validation with Hibernate Validator," [Online]. Available: https://quarkus.io/guides/validation#service-method-validation. [Zugriff am 30 Juli 2021].

[10] „H2 Database Engine," [Online]. Available: http://www.h2database.com/html/main.html. [Zugriff am 29 07 2021].

[11] „H2 Database Engine Cheat Sheet," [Online]. Available: http://www.h2database.com/html/cheatSheet.html. [Zugriff am 1 Juli 2021].

[12] „Quarkus - Datasources," [Online]. Available: https://quarkus.io/guides/datasource. [Zugriff am 29 07 2021].

[13] „Quarkus - Security Architecture and guides," [Online]. Available: https://quarkus.io/guides/security.

[14] „Quarkus - Built-In Authentication Support," [Online]. Available: https://quarkus.io/guides/security-built-in-authentication.

[15] „Quarkus - Using Security With JPA," [Online]. Available: https://quarkus.io/guides/security-jpa.

[16] „Quarkus - Testing Your Application," [Online]. Available: https://quarkus.io/guides/getting-started-testing. [Zugriff am 30 Juli 2021].

[17] „Qute Templating Engine,“ [Online]. Available: https://quarkus.io/guides/qute.

[18] „Interface EntityManager,“ [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html. [Zugriff am 30 Juli 2021].

[19] „Data Types,“ [Online]. Available: http://www.h2database.com/html/datatypes.html#date_type. [Zugriff am 30 Juli 2021].

# 6 Anhang

## 6.1 Snippets

```
@Singleton // => pseudo-scope
class AmazingService {
  String ping() {
    return "amazing";
  }
}

@ApplicationScoped // => normal scope
class CoolService {
  String ping() {
    return "cool";
  }
}

@Path("/ping")
public class PingResource {

  @Inject
  AmazingService s1; 1

  @Inject
  CoolService s2; 2

  @GET
  public String ping() {
    return s1.ping() + s2.ping(); 3
  }
}
```

Snippet 13: Beispiel Context-Injection Quarkus [7]

```
@Embeddable
public class Color {
    @JsonValue
    @NotBlank
    @ColorString
    private String value;
}
```

Snippet 14: Color-Klasse vgl. de.hsos.roomplanner.util.color.Color

```
@Embeddable
public class Dimension {

    @Positive
```

```
    private float width;
    @Positive
    private float height;
    …
}
```

Snippet 15: Dimension-Klasse vgl. de.hsos.util.Dimension

```
@Embeddable
public class Position {…}
```

Snippet 16: Position-Klasse vgl. de.hsos.util.Position

```
@Entity
@Table(name = "User")
@Immutable
public class ImmutableUserFurniture {

    @Id
    private String username;
    …
}
```

Snippet 17: ImmutableUserFurniture-Klasse
vgl. de.hsos.roomplanner.furniture.entity.ImmutableUserFurniture

```
@Entity
@Table(name = "User")
@Immutable
public class ImmutableUserPlan {

    @Id
    private String username;
    …
}
```

Snippet 18: ImmutableUserPlan-Klasse
vgl. de.hsos.roomplanner.plan.entity.ImmutableUserPlan

```
@Entity
@Immutable
@Table(name = "Furniture")
public class ImmutableFurniture {

    @Id
    private long id;
    …
}
```

Snippet 19: ImmutableFurnitureKlasse
vgl. de.hsos.roomplanner.plan.entity.ImmutableFurniture

## 6.2 Abbildungen

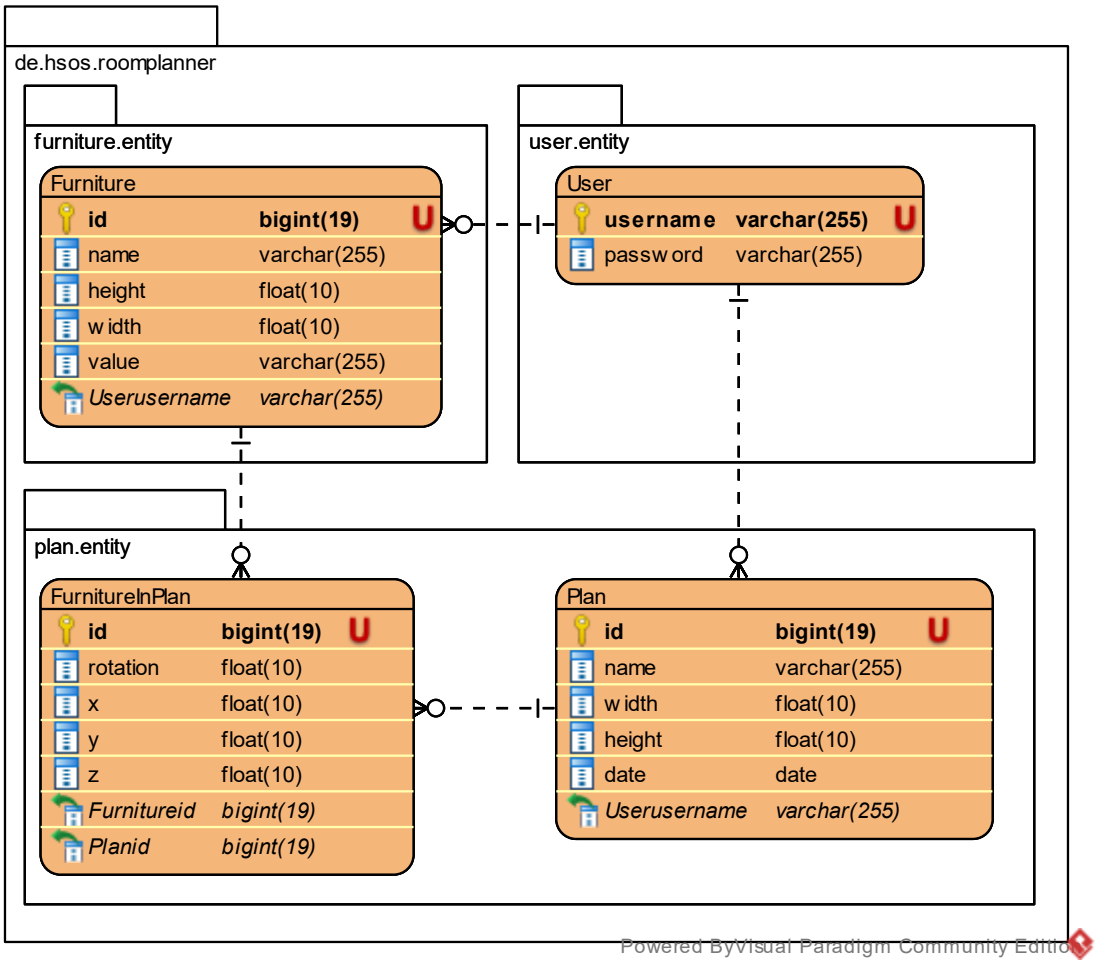


Abbildung 9: EMR Diagramm

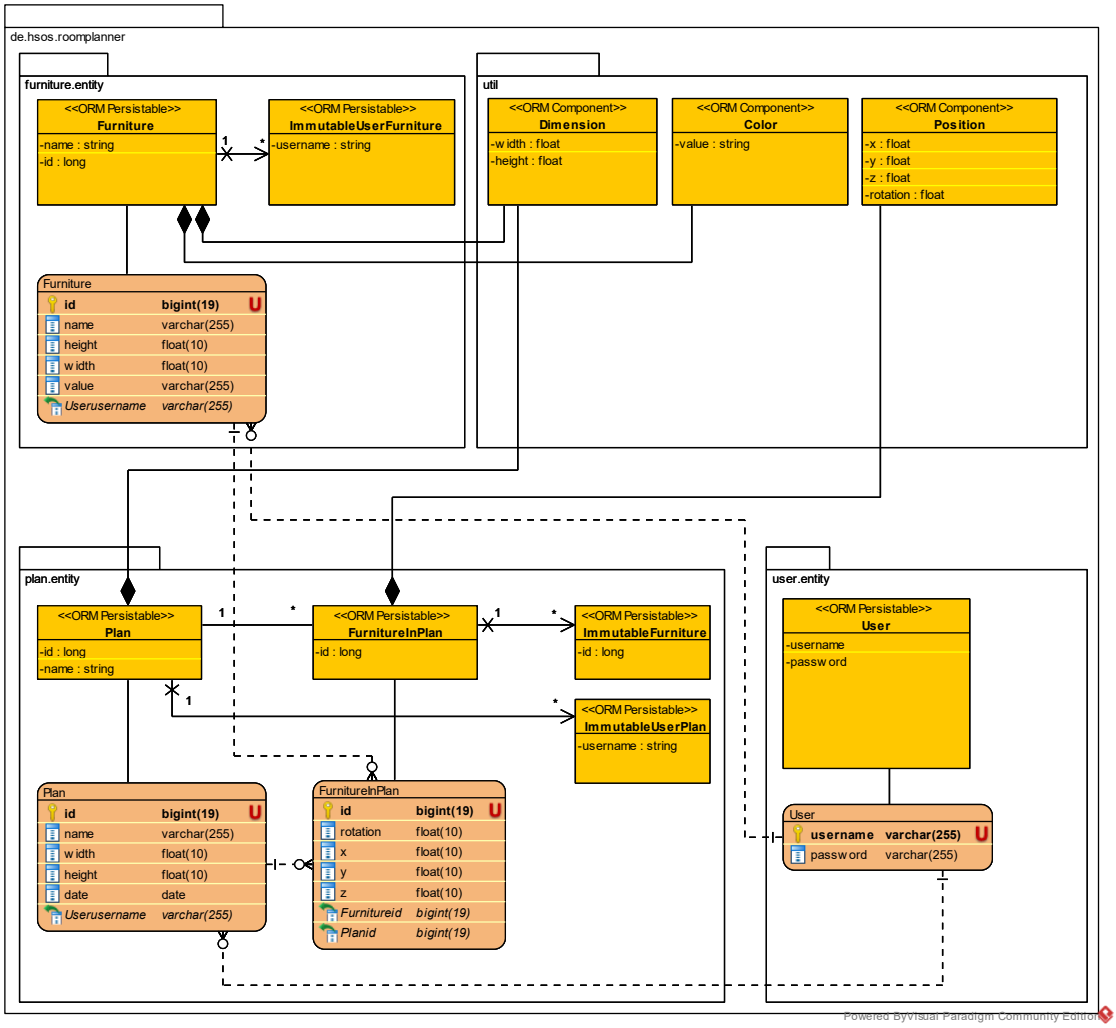


Abbildung 10: ORM Diagramm

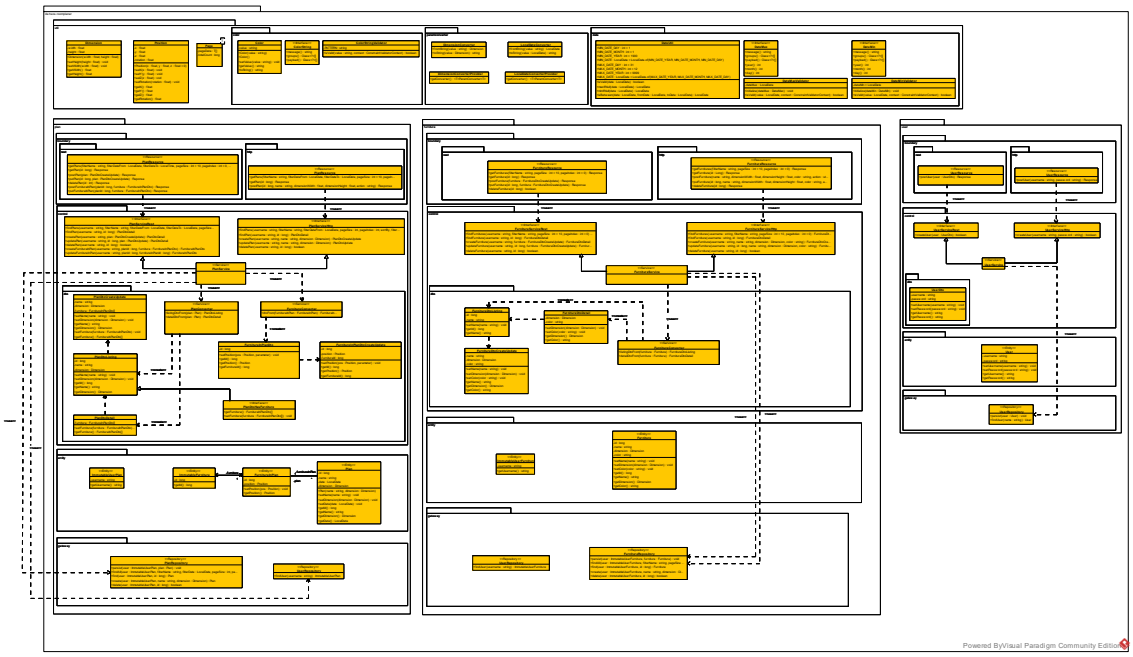

Abbildung 11: Klassendiagramm

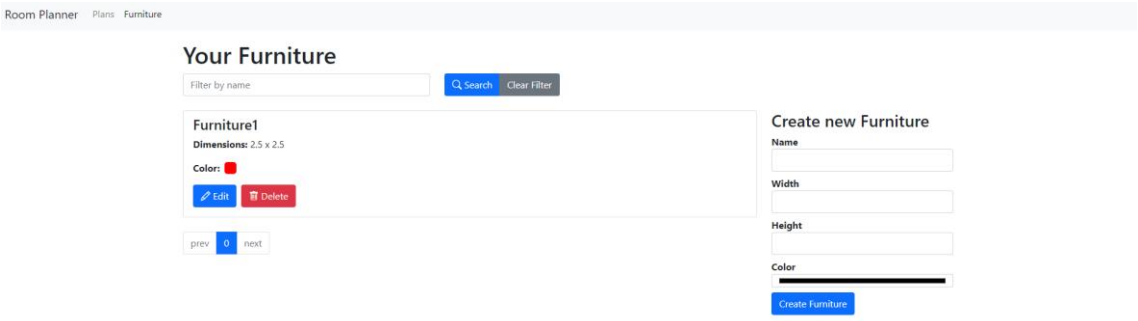


Abbildung 12: Fruniture-Listing-Page


Abbildung 13: Furniture-Edit-Page