

# Arbeitsdokumentation des Backendteam

Team Backend

28. April 2019

Die Umsetzung der in der API-Spec definierten Funktionalität erfolgt im Backend. Dies ist eine Java-Applikation, welche auf einem Tomcat-Applikationsserver läuft. Die Applikation übernimmt unter anderem folgende Aufgaben:

- Bereitstellen der Routen
- Businesslogik
- Authentifizierung und Authorisierung
- Speichern in einer Datenbank

## 1 Einstieg in Spring Boot

Die Entwicklung erfolgt mithilfe des Frameworks [Spring Boot](#). Das Framework bietet verschiedene Features, die die Entwicklung erleichtern oder strukturieren. Es ist zum Beispiel möglich, einen [objektrelationalen Mapper](#) zu nutzen, der den Datenbankzugriff stark abstrahiert. Auch das Bereitstellen des Web-Services wird durch Spring Boot vereinfacht. Zusätzlich hat das Framework eine gewisse *Meinung*, mit welchen Mustern entwickelt werden soll.

Zum Einstieg in das Backend-Projekt betrachten wir die Flugzeugverwaltung. Die findet im Paket Plane statt, in dem zwei Klassen und ein Interface definiert werden (vgl. [Abbildung 1](#)).

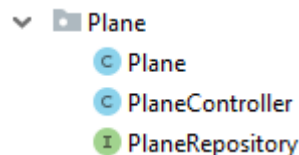


Abbildung 1: Inhalt des Package Plane

Dabei ist die Klasse Plane das Business-Objekt, welches ein Flugzeug repräsentiert. Im PlaneController werden die Webservice-Routen und die Business-Logik definiert und das PlaneRepository abstrahiert das Speichern von Plane-Objekten in der Datenbank.

Schauen wir uns nun zunächst einen Ausschnitt aus der Plane-Klasse an:

```
@Entity
public class Plane {

    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Integer id;
    private String number;
    private String name;
    ...
}
```

Die @Entity-Annotation gibt an, dass die Klasse in der Datenbank gespeichert werden soll, wobei die id der Primärschlüssel hierfür ist und vom Framework generiert werden soll.

Das PlaneRepository verwaltet dann die Datenbank-Interaktion und stellt dafür unter anderem die Methoden aus Abbildung 2 bereit. Diese Methoden erbt es von einem Interface, welches aus dem Spring-Framework kommt. Daher wird der Datenbankzugriff in der Entwicklung stark vereinfacht. Es ist an keiner Stelle nötig, SQL-Befehle zu formulieren. Auch Schemata werden automatisch generiert. Dies ist ein besonders großer Vorteil, da so eine Änderung in der Java-Klasse automatisch im Datenbankschema übernommen wird.

Abbildung 3 zeigt die MySQL-Tabelle, die vom objekt-relationalen Mapper generiert wurde, um die Plane-Objekte zu persistieren. Diese Tabelle wird automatisch durch die Entity-Annotation erstellt. Wenn sich die Klasse verändert, wird auch beim nächsten Start der Applikation auch die Tabelle geändert. Durch diese Abstraktion wird ein hypothetisches Datenbank-Team in der Entwicklung unnötig, da nur noch zu Kontroll- und Test-Zwecken direkt auf die Datenbank zugegriffen wird.

Der PlaneController enthält nun die Business-Logik:

```
@RestController
@RequestMapping(path = "planes")
public class PlaneController {
    ...
    @GetMapping(path =("/{id}")
    public Plane detail(@PathVariable int id) {
```

All Methods	Instance Methods	Abstract Methods
Modifier and Type		Method and Description
long		<b>count()</b> Returns the number of entities available.
void		<b>delete(T entity)</b> Deletes a given entity.
void		<b>deleteAll()</b> Deletes all entities managed by the repository.
void		<b>deleteAll(Iterable&lt;? extends T&gt; entities)</b> Deletes the given entities.
void		<b>deleteById(ID id)</b> Deletes the entity with the given id.
boolean		<b>existsById(ID id)</b> Returns whether an entity with the given id exists.
Iterable<T>		<b>findAll()</b> Returns all instances of the type.
Iterable<T>		<b>findAllById(Iterable&lt;ID&gt; ids)</b> Returns all instances of the type with the given IDs.
Optional<T>		<b>findById(ID id)</b> Retrieves an entity by its id.
<S extends T> S		<b>save(S entity)</b> Saves a given entity.
<S extends T> Iterable<S>		<b>saveAll(Iterable&lt;S&gt; entities)</b> Saves all given entities.

Abbildung 2: Ausschnitt der Methoden, die das PlaneRepository bereitstellt


plane		
	<b>id</b>	int
	name	varchar(255)
	<b>needed_authorization</b>	varchar(255)
	number	varchar(255)
	picture_url	varchar(255)
	position	varchar(255)
	<b>price_per_booked_hour</b>	double
	<b>price_per_flight_minute</b>	double

Abbildung 3: Generierte Tabelle, die Plane-Objekte speichert

```

        return planeRepository.findById(id)
            .orElseThrow(() -> new
                NoSuchElementException());
    }
    ...
}

```

Hier ist die Definition der unter dem Pfad `<server>:<port>/planes/{id}` anzufindenden Methode gezeigt. Es wird in der URL also Parameter angegeben, welches Flugzeug angezeigt werden soll, woraufhin das Flugzeug in der Datenbank gesucht wird. Wird es gefunden, wird es zurückgegeben, wird kein Flugzeug gefunden, wird eine Exception geworfen. Hierbei übernimmt Spring Boot, oder genauer, die Jackson-Bibliothek das *Marshalling* und *Unmarshalling*, also das Umwandeln von JSON zu einfachen Java-Objekten und umgekehrt. Außerdem gibt es eine Klasse, welche Exceptions fängt und daraufhin an den Client passende Fehlermeldungen zurücksendet.

Dieser Dreiklang aus Business-Objekt, dazugehörigem Controller und Repository ist relativ typisch für das Projekt.

## 2 Die Business-Objekte

Eine Übersicht über die Business-Objekte des Projekts kann das Diagramm in Abbildung 4 geben. Es ist dabei zu beachten, dass alle diese Tabellen vom objekt-relationalen Mapper generiert wurden. Assoziationen werden im Quellcode über Annotationen wie `@OneToOne`, `@OneToMany` und `@ManyToMany` zwischen den Objekten hergestellt.

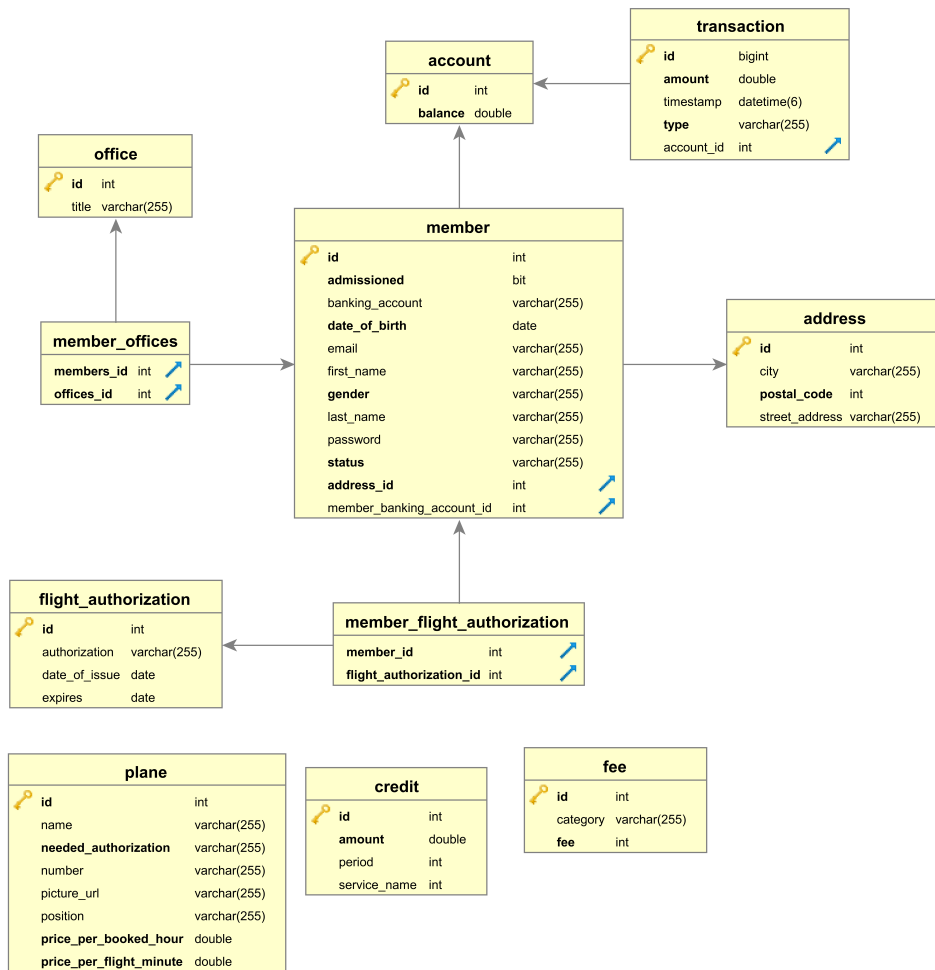


Abbildung 4: Diagramm des Datenbanktabellen