

Arbeitsdokumentation des Backendteam

Team Backend

15. April 2019

Zusammenfassung

Das MdWI-Projekt der Semester fünf und sechs wird als agiles Softwareprojekt durchgeführt. Für die Umsetzung wurde die Entscheidung getroffen, den Kurs in drei Teams aufzuteilen: *Marketing*, *Frontend* und *Backend*. Hier wird die Motivation hinter dieser Trennung, verbunden mit der technischen Architektur, beschrieben.

1 Ausgangssituation

Zu Beginn gab für die Umsetzung des Projekts zwei zentrale Fragen:

- Wie soll die technische Architektur aussehen?
- Wie organisieren wir uns als Kurs?

Als Rahmenparameter war gegeben, dass das Projekt per Scrum organisiert werden soll. Dies bringt einige Implikationen für die Entwicklung mit sich, da Scrum interdisziplinäre Entwicklerteams fordert, im Gegensatz zu zum Beispiel einer Aufteilung in Entwicklungs-, Datenbank-, und Test-Team.

Scrum fordert außerdem Teams von rund sieben Mitarbeitern. Den Kurs in drei Teams aufzuteilen, schien deshalb sinnvoll. Daraus ergab sich die Frage nach einer geeigneten technischen Architektur, bei der mehrere Entwicklerteams parallel an einem Produkt arbeiten können. Da eine Webapplikation entwickelt werden sollte, wurden zwei Optionen gesehen:

- Die Umsetzung als Monolith.
- Die Umsetzung als Single-Page-Anwendung mit getrenntem Front- und Backend.

Eine Umsetzung als *monolithische* Applikation hätte zum Beispiel so aussehen können, dass zum Beispiel mit PHP und HTML oder einem Webframework mit HTML-Template-Engine von mehreren Teams eine einzige Applikation entwickelt wird.

Die Alternative wäre, eine *Single-Page-Anwendung* mit Webservice-Backend zu entwickeln. Hier entwickelt ein Team eine Webanwendung, welche vom

| | Monolith | Single-Page-Application |
|-----------|--|---|
| Vorteile | Alle arbeiten mit einer Sprache, einem Framework | Entkopplung in zwei Anwendungen: Frontend und Backend |
| Nachteile | eventuell überbordende Komplexität | Herausforderung der Absprache zw. Front- und Backend |

Tabelle 1: Vor- und Nachteile der techn. Ansätze

Endnutzer als ein einziges HTML-Dokument heruntergeladen wird, welches daraufhin anzuzeigende Inhalte dynamisch von einem Backend-Server abfragt, der von einem anderen Team entwickelt wird. Hier ist die Website eine art *Fat-Client*, da die Website, die der Nutzer herunterlädt, viel mehr Logik enthält als die verschiedenen Webseiten, die eine monolitische Applikation generiert.

Es wurde die Ansätze abgewogen (vgl. Tabelle 1) und sich gegen den monolitischen Ansatz entschieden. Daraus resultierte nun endgültig die Teilung in *Frontend* und *Backend* (und *Marketing*).

2 Single-Page-Anwendung, Frontend, Backend

Abbildung 1 zeigt die Funktionsweise einer Single-Page-Anwendung. Im ersten hervorgehobenen Rechteck wird das Laden der Anwendung dargestellt. Ein Anwender ruft zum Beispiel die Webseite der Projekt-Anwendung auf. Sein Browser schickt dann eine GET-Request nach der `index.html`, der zentralen HTML-Seite. Diese wird vom Webserver beantwortet. Nun hat der Anwender die Anwendung in den Speicher seines Browsers geladen.

Wenn nun der Anwender zum Beispiel seine Mitgliedsdaten sehen möchte, klickt er im Browser auf den entsprechenden Link. Es wird jedoch keine neue HTML-Seite aufgerufen und heruntergeladen, sondern die Webanwendung schickt stattdessen eine Anfrage an einen Webservice, das *Backend*. Dieser prüft dann zum Beispiel, ob die Anfrage berechtigt ist und greift auf die Datenbank zu um die angeforderten Daten zu lesen.

Daraufhin schickt er die Daten an die Webanwendung im Browser des Nutzers zurück, welche dann entscheidet, wie diese Daten dargestellt werden sollen und dann das HTML, also den Aufbau der Seite, dynamisch verändert.

Es ist also ein bisschen so, als ob der Nutzer beim Öffnen der Seite ein Programm herunterlädt und mit diesem interagiert. Das Programm wiederum greift auf einen Server zu, schickt und holt Daten. Schließt der An-

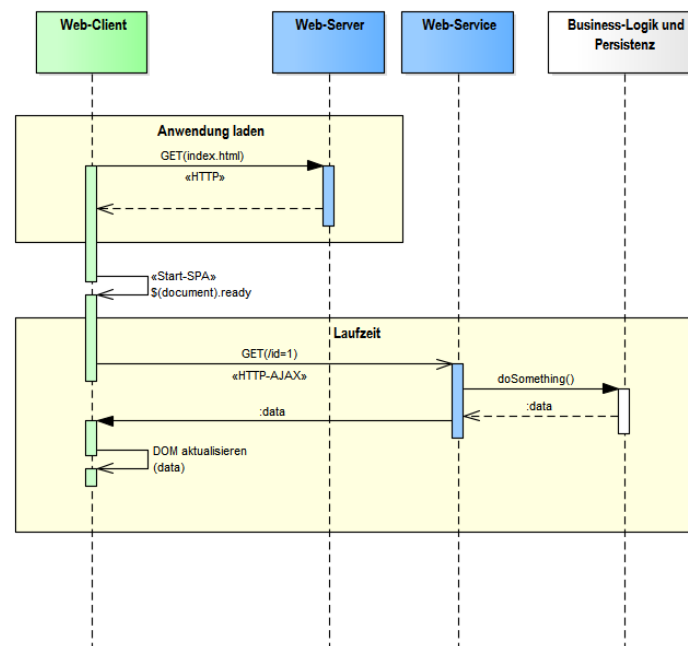


Abbildung 1: Skizzierung des Ablaufs einer Single-Page-Webanwendung

wender das Browserfenster löscht/deinstalliert er das Programm sozusagen.

Die Kommunikation zwischen der Anwendung im Browser und dem Webservice erfolgt über eine Webschnittstelle. Die Anwendung ruft also Webseiten auf, wobei diese aber nicht für Menschen gedacht sind, sondern einfach nur Daten in einer relativ rohen Form enthalten.

3 Die API-Spec

Im Projekt erfolgt die Definition und Dokumentation dieser Web-Schnittstelle über die sogenannte [API-Spec](#). Hier werden die möglichen Routen, die die Frontend-Anwendung aufrufen kann, definiert. Genauso werden die eventuell notwendigen Parameter der Anfragen und die möglichen Ergebnisse beschrieben.

Abbildung 2 zeigt einen Auszug aus der API-Speck. Hier wurden unter der Überschrift *Accounts* drei verschiedene Routen zugegriffen, auf die jeweils mit verschiedenen HTTP-Methoden zugegriffen wird. Rechts steht eine Beschreibung, was mit dem Aufruf geschieht. Dies ist jedoch nur die Kompakt-Ansicht, die sich expandieren lässt.

Ein Ausschnitt der expandierten Darstellung ist in Abbildung 3 zu se-

| Accounts | |
|----------|--|
| GET | /accounts Paginated list of accounts without transactions. |
| GET | /accounts/{id} Info for a specific account with his transactions |
| POST | /accounts/{id}/transactions Add a single transaction to an account |

Abbildung 2: Übersicht der möglichen Routen zu Accounts

hen. Hier wird beschrieben, welche Parameter für die Anfrage notwendig sind. Das ist zum einen ein Parameter in der URL des Aufrufs, um anderen muss beim POST eine Art Anhang mitgeschickt werden, nämlich das Objekt im *Request Body*, welches den Betrag und den Transaktionstyp enthält.

4 Das Backend

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 folgende Aufgaben:

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

4.1 Einstieg in Spring Boot

Die Entwicklung erfolgt mithilfe des Frameworks [Spring Boot](#). Das Framework bietet verschiedene Features, die die Entwicklung teilweise erleichtern. 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.

Zum Einstieg in das Backend-Projekt betrachten wir die Flugzeugverwaltung. Die findet im Paket *Plane* statt, wofür zwei Klassen und ein Interface definiert werden (vgl. [Abbildung 4](#)).

POST

/accounts/{id}/transactions

Add a single transaction to an account

Parameters

Try it out

| Name | Description |
|---|-------------------------------|
| id * required integer (path) | The id of the related account |

Request body required

application/json

Transaction to add to the account

Example Value

Model

```

{
  "amount": 0,
  "type": "MITGLIEDSBEITRAG"
}

```

Abbildung 3: Notwendige Parameter für das Hinzufügen einer Transaktion zu einem Mitgliedskonto

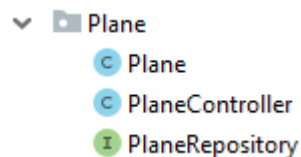


Abbildung 4: Inhalt des Package Plane

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 5 bereit. Diese Methoden erbt es von einem Interface, welches aus dem Spring-Framework kommt. Daher wird der Datenbankzugriff in der Entwicklung stark vereinfacht.

Abbildung 6 zeigt die MySQL-Tabelle, die vom objekt-relationalen Mapper erstellt 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) {

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

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

Abbildung 5: Ausschnitt der Methoden, die das PlaneRepository bereitstellt


| plane | | |
|---|--------------------------------|--------------|
|  | id | int |
| | name | varchar(255) |
| | needed_authorization | varchar(255) |
| | number | varchar(255) |
| | picture_url | varchar(255) |
| | position | varchar(255) |
| | price_per_booked_hour | double |
| | price_per_flight_minute | double |

Abbildung 6: Generierte Tabelle, die Plane-Objekte speichert

```
}  
...  
}
```

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.

4.2 Die Business-Objekte

Eine Übersicht über die Business-Objekte des Projekts kann das Diagramm in Abbildung 7 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.

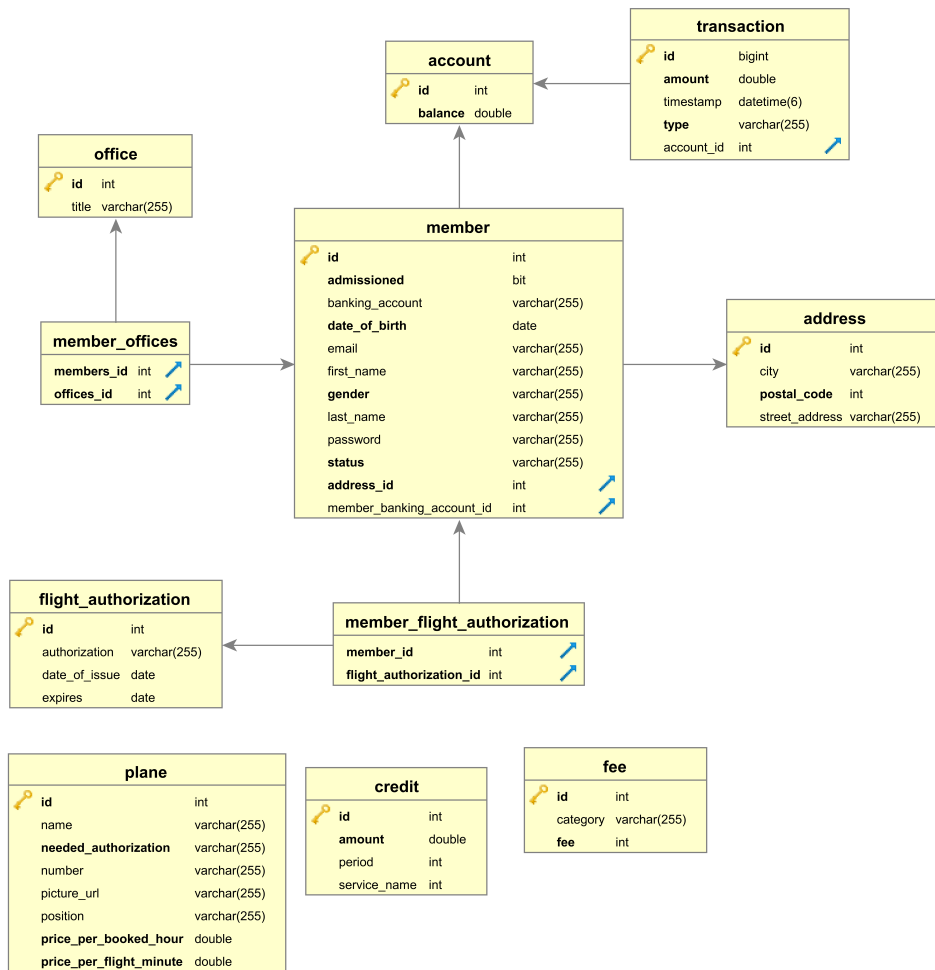


Abbildung 7: Diagramm des Datenbanktabellen