Arbeitsdokumentation des Backendteam

Team Backend
2. Mai 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 API-Routen
- Abbilden der Businesslogik
- Speichern in einer Datenbank
- Authentifizierung und Authorisierung

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 der API-Routen wird durch Spring Boot vereinfacht. Zusätzlich hat das Framework eine gewissen *Meinung*, mit welchen Mustern entwickelt werden soll, welches sehr hilfreich für uns unerfahrene Entwickler war.

Zum Einstieg in das Backend-Projekt und dem Kennenlernen der Spring Boot-Basics betrachten wir die Flugzeugverwaltung. Die findet im Paket `Plane` statt, in dem zwei Klassen und ein Interface definiert werden (vgl. Abbildung 1).

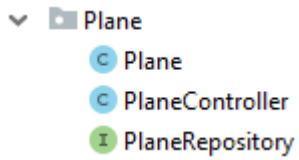

Abbildung 1: Inhalt des Package `Plane`

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:

Listing 1: Entity-Klasse

```
@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 persitieren. 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. In Listing 2 wird die Definition der unter dem Pfad `<server>:<port>/planes/{id}` anzufindengen Methode gezeigt. Es wird in der URL also als 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 Bibliothek *Jackson*, das *Marshalling* und *Unmarshalling*, also das Umwandeln von JSON zu einfachen Java-Objekten und umgekehrt.

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 2: Ausschnitt der Methoden, die das PlaneRepository bereitstellt

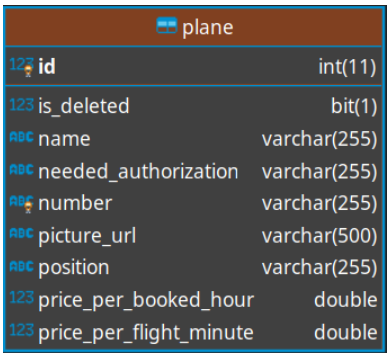

Abbildung 3: Generierte Tabelle, die `Plane`-Objekte speichert

Listing 2: Ausschnitt eines REST-Controllers

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

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

Dieser Dreiklang aus Business-Objekt, dazugehörigem Controller und Repository ist relativ typisch für die meisten Business-Objekte im Projekt.

2 Die Business-Objekte und -Logik

2.1 Relationen mit *Hibernate*

Eine Übersicht über die Business-Objekte des Projekts kann das Diagramm in Abbildung 4 geben. Es wurde aus den Datenbanktabellen generiert, welche wiederum mit *Hibernate*, dem objekt-relationalen Mapper, generiert wurden.

Relationen zwischen Objekten/Entitäten werden im Quellcode über Annotationen wie `@OneToOne`, `@OneToManty` und `@ManyToMany` zwischen den Objekten abgebildet. Listing 3 zeigt einen Ausschnitt der Member-Klasse, die das zentrale Business-Objekt ist, welches ein Vereinsmitglied abbildet. Ein Vereinsmitglied hat unter anderem ein Logbuch, abgebildet als `pilotLog`, für seine Flüge und eine Reihe von Berechtigungen und Lizenzen, wie zum Beispiel die Privatpilotenlizenz, abgebildet als eine Liste von `flightAuthorization`-Objekten.

Alle diese Objekte sind *Hibernate*-Entitäten. Dadurch werden entsprechende Tabellen in der Datenbank angelegt. Durch die Annotationen werden nun die Relationen charakterisiert. Der Parameter `cascade = CascadeType.ALL` gibt an, dass wenn ein `Member` über das `MemberRepository`, welches sich analog dem `PlaneRepository` verhält, aus der Datenbank geladen werden soll, auch gleich das Objekt, mit dem der Member in Relation steht, mitgeladen werden soll.

Listing 3: Verknüpfung zu anderen Entitäten in der Member-Klasse

```
@OneToMany(cascade = CascadeType.ALL)
private List<FlightAuthorization> flightAuthorization = new
    ArrayList<>();

@OneToOne(cascade = CascadeType.ALL)
private PilotLog pilotLog;
```

2.2 Vereinskonto

Während der Member ein schönes Beispiel ist, wie relativ typische Sachverhalte mit *Hibernate* umgesetzt werden können, ist das *Vereinskonto* ein Beispiel für eine Stelle , an der wir etwas kreativer werden mussten im Umgang mit dieser Technologie.

Die Mitglieder besitzen ein Mitgliedskonto, in dem ihre Transaktionen, *zum Beispiel das Zahlen der Mitgliedsgebühr und ihr Kontostand*, verwaltet werden. Auch der Verein hat so ein Konto. Hier wird es nun schwierig: Die Mitgliederkonten sind Objekte, die Mitgliedern zugeordnet sind. Wie also das Vereinskonto gestalten? Es wurden zwei Vorschläge diskutiert:

- Das Vereinskonto ist von einer anderen Klasse als das Mitgliedskonto und ein *Singleton*.
- ein *unsichtbares* Mitglied repäsentiert den Verein und dessen Konto ist das Vereinskonto.

Die Umsetzung des Vereinskontos als Singleton, eine Klasse, von der immer maximal eine Instanz existiert, scheint logisch. Der zweite Vorschlag wurde formuliert, falls sich der erste als nicht umsetzbar erweist, mangelt aber der Eleganz des ersten.

Bei der Implementierung des Vereinskontos als Singleton gab es Hürden: Die Methode, die statt dem Kunstruktor implementiert wird, um eine Referenz auf das Singleton zu erhalten (`getInstance`-Methode), muss nun auch dafür Sorge tragen, dass die Instanz aus der Datenbank aktualisiert wird.

Listing 4: `getInstance`-Methode des Vereinskonto-Singletons

```
public static VereinsAccount getInstance(AccountRepository
    accountRepository) {
    if (instance == null) {
        instance = new VereinsAccount();
        instance.setBalance(25000.0);
        accountRepository.save(instance);
    }
```

Abbildung 4: Diagramm des Datenbanktabellen

```
    return (VereinsAccount)
        accountRepository.findById(instance.getId())
            .orElseThrow(() -> new
                IllegalStateException("VereinsAccount nicht
                richtig gespeichert"));
}
```

Das ganze lies sich implementieren, wenn auch mit einigem Aufwand und einem zwischenzeitlichen versteckten Bug, der eine halbe Woche später gefunden wurde und über einige Tage aufwändig analysiert werden musste. Listing 4 zeigt die inzwischen fehlerfreie Methode. Der Code ist deutlich komplexer als eine *typische* `getInstance`-Methode. Dies liegt aber weniger an *Hibernate* als an der grundlegenden Herausforderung, ein Singleton zu persistieren. Bei der Implementierung hat sich gezeigt, dass es manchmal durchaus nötig ist, die von *Hibernate* verborgene Komplexität zu kennen und zu verstehen. Dies kann einige Zeit kosten. Es herrscht aber die Einschätzung, dass es in unserem Fall trotzdem definitiv eine gute Entscheidung war, *Hibernate* intensiv zu nutzen. *Hibernate* erspart in den häufig *Standardfällen* viel Mühe und ist in den ein, zwei komplexeren Fällen durchschaubar genug, um auch diese mit der Technologie abzubilden.

2.3 Transactions

In den Konten werden verschiedene Transaktionen gespeichert. Grundsätzlich gibt es aus Vereinssicht zwei Typen von Transaktionen:

- Einem Mitgliedskonto wird Geld hinzugefügt oder abgezogen. *(externe Transaktion)*
- Es wird Geld zwischen dem Vereinskonto und einem Mitgliedskonto überwiese. *(interne Transaktion)*

Externe Transaktionen werden als solche bezeichnet, weil Geld *von außen kommt* respektive *abfließt*, während bei *internen Transaktionen* Geld innerhalb des Vereins bewegt wird. Es ist nicht vorgesehen, dass direkt zwischen zwei Mitgliedern Geld überwiesen werden kann.

Auch hier wurden zwei verschiedene Implementierungen vorgeschlagen:

In der Ursprünglichen ist eine Transaktion einem Konto zugeordnet und beschreibt eine positive oder negative Wertänderung. Dies wurde zwischenzeitlich kritisch gesehen und es wurde überlegt, stattdessen eine Transaktion immer als Geldfluß zwischen zwei Konten abzubilden. Es wurde der Vorteil hierbei gesehen, dass *interne Transaktionen* so leichter korrekt abbildbar sind. Tatsächlich erwies sich der Ansatz als schwierig zu implementieren, daher wurde auf den ersten zurückgewechselt und dieser

erweitert. *Externe Transaktionen* werden als eine Transaktion im Mitgliedskonto implementiert, bei *internen Transaktionen* werden zwei Transaktionen gespeichert: eine im Mitgliedskonto und eine Vereinskonto-Transaktion im Vereinskonto. Diese enthält zusätzlich die ID des Mitglieds, welches an der Transaktion beteiligt ist. Abbildung 5 zeigt ausschnittsweise eine *interne Transaktion* in der Datenbank. Die beiden gezeigten Einträge bilden diese ab. Die Fremdschlüssel der Einträge sind in Abbildung 6 zu sehen. Die *normale* Transaction ist so dem Mitglied zugeordnet. Die Vereinskonto-Transaktion hingegen hat zwei Fremdschlüssel, so dass nachvollzogen werden kann, zwischen welchem Mitgliedskonto und dem Vereinskonto überwiesen wurde.

	dtype	id	amount	text	timestamp
1	Transaction	1	-146.6700000000000000	Mitgliedsbeitrag Neueintritt 9999. Hansen	2019-05-02 14:38:22
2	VereinsKontoTransaction	2	146.6700000000000000	Mitgliedsbeitrag Neueintritt 9999. Hansen	2019-05-02 14:38:22

Abbildung 5: Ausschnitt der Tabelle der Transaktionen.

	dtype	id			timestamp		member_acc_id	account_id
1	Transaction	1	46.6	Mitg	2019-05-02 14:38:22	MIT	[NULL]	15,389
2	VereinsKontoTransaction	2	5.67	Mitg	2019-05-02 14:38:22	MIT	15,389	19,304

Abbildung 6: Ausschnitt der Tabelle der Transaktionen, hier mit den Konto-Fremdschlüsseln.

2.4 Events

Besonders bei den Business-Transaktionen ist, dass hierbei *Events* genutzt werden. Diese Funktionalität wird ebenfalls vom Framework bereitgestellt, und ermöglicht es, die Businesslogik geschickt zu koppeln. Bei einer Implementierung der Businesslogik per Events gibt es drei zentrale Elemente:

Event Objekt, das alle notwendigen Informationen bündelt.
Eventpublisher *veröffentlicht* Events, generisch
Eventlistener verarbeitet Events, enthält Großteil der Logik

Betrachten wir dies am Beispiel der externen Transaktionen:

Listing 5 zeigt einen Teil des `AccountController`. Wird im Frontend befohlen einem Mitglied Geld zum Konto hinzuzufügen, schickt dieses das entstehende Transaktionsobjekt und die Id des betroffenen Kontos. Daraufhin wird ein Event veröffentlichlicht.

Dieses Event wird im Eventlistener verarbeitet (Listing 6).

Listing 5: Ausschnitt aus dem `AccountController`, in dem ein Event veröffentlicht wird

```
@PostMapping(path = "/{id}/transactions")
public Transaction addTransaction(@RequestBody Transaction
    transaction, @PathVariable int id) {
    Account acc = accountRepository.findById(id)
            .orElseThrow(() -> new
                NoSuchElementException("Account with the id "
                + id + " does not exist"));
    publisher.publishEvent(new ExtTransactionEvent(acc,
        transaction));
    return transaction;
}
```

Listing 6: Eventlistener für externe Transaktionen

```
@EventListener
public void makeExternalTransaction(final ExtTransactionEvent
    transactionEvent) {

    Transaction tr = transactionEvent.getTransaction();
    Account account = transactionEvent.getAccount();

    tr.setType(tr.getAmount() > 0 ?
        Transaction.FeeType.EINZAHLUNG :
        Transaction.FeeType.AUSZAHLUNG);

    checkIfBalanceGetsLow(account, tr);

    account.addTransaction(tr);
    accountRepository.save(account);
}
```

Die Business-Logik findet hier an zwei Stellen statt:

- in der Methode selbst wird geprüft, ob das Guthaben des Mitglieds durch die Transaktion unter 200 € sinkt. Falls dies passiert, wird ein weiteres Event veröffentlicht, welches das Versenden einer Mail an das Mitglied veranlasst.
- Außerdem wird beim Speichern des Accounts/Kontos die Transaktion validiert. Dies geschieht mittels Java-Validation-Annotationen.

2.5 Validation

Zur Validierung von Objekten bietet Java die Möglichkeit, dies per Annotationen zu tun. Listing 7 zeigt einige davon. Diese definieren allgemein, welche Bedingungen für *valide* Instanzen gelten. In diesem Fall darf z. B. der Text nicht kürzer als 4 und nicht länger als 50 Zeichen sein.

Listing 7: Validierung von Business-Objekten per Annotation

```
@NotNull
private FeeType type;
@NotBlank
@Pattern(regexp = ".{4,50}")
private String text;
```

Diese Validierungsregeln können nun an verschiedenen Stellen geprüft werden. Dies geschieht automatisch vor dem Speichern in der Datenbank. Dies ist der Regelfall der Prüfung im Projekt. Eine weitere Methode ist, direkt zu Beginn im Controller die Validität der mit der Request gesendeten Objekte zu kontrollieren (vgl. Listing 8).

Listing 8: Beispiel für Validierung von Request-Parametern

```
public List addPlaneLogEntry(@Validated @RequestBody
    PlaneLogEntry entry, @PathVariable int id)
```

2.6 Emails

In einigen Fällen sollen Mails an Mitglieder des Vereins geschickt werden. Dies geschieht wieder per Events. Es gibt ein `EmailNotificationEvent` und einen dazugehörigen Eventlistener. Versendet wird die Mail durch eine vom Spring-Framework bereitgestellte Mail-Sender-Klasse. Das HTML, was den Mailinhalt ausmacht, wird vorher mit der Templating-Engine *Thymeleaf* generiert. Diese hätte theoretisch auch genutzt werden können, um HTML für die Vereinswebseite zu generieren. Dann gäbe es kein separates Frontend sondern das Backend würde das HTML dynamisch generieren.

Da der Mailversand mindestens mehrere Sekunden dauert, was verhältnismäßig sehr lange ist, ist das `EmailNotificationEvent` *asynchron*. Dies bedeutet, dass die Bearbeitung der Events in einem anderen Thread geschieht. Diese Events wurden asynchron implementiert, da sonst eine spürbare Verzögerung z. B. beim Erstellen eines neuen Mitglieds auftritt. Transaktions-Events wurden jedoch bewusst synchron implementiert, da dies Vorteile in der Fehlerbehandlung bietet.

Ein Problem mit dem Mailversand ist, dass der Nutzer keine direkte Möglichkeit hat, zu sehen, ob die Mail wirklich versand wurde. Der `JavaMailSender` schickt die Mail an einen SMTP-Server, der dann versucht, diese an den Empfänger zuzustellen. Gelingt dies nicht, erfährt dies der `JavaMailSender` nicht.

2.7 ExceptionHandling

Im Gegensatz zum *unsichtbaren* Fehler der nicht versendeten Mail werden normalerweise Fehler explizit behandelt. Java bietet dazu Exceptions, die im Fehlerfall geworfen werden und dann an anderer Stelle behandelt werden können.

Im Projekt ist der generelle Ansatz, dass bei einem Fehler die passende Exception mit erklärendem Text geworfen wird und dann in einem für die Applikation zentralen Ort behandelt werden. Dieser Ort ist der *ControllerAdvice*. Hier ist definiert, was bei welcher Exception geschehen soll.

Da das Backend per HTTP mit dem Frontend kommuniziert, müssen Fehler über dieses Protokoll ans Frontend vermittelt werden. Dazu gibt es unter anderem die HTTP-Statuscodes, von denen der Code `404 - Not found` wohl am bekanntesten ist. Der ControllerAdvice nimmt nun den relevanten Text aus den Exceptions und sendet ihn mit dem passenden HTTP-Statuscode ans Frontend zurück. Listing 9 zeigt den Teil des ControllerAdvice, in dem die `NoSuchElementException` gefangen wird und ihr Inhalt mit dem 404-Status ans Frontend zurückgeschickt wird.

Listing 9: Globales Handling der NoSuchElementException

```
@ExceptionHandler(NoSuchElementException.class)
public ResponseEntity<String> handleNoEntryFound(Exception ex)
    {
     return new ResponseEntity<>(ex.getLocalizedMessage(),
         HttpStatus.NOT_FOUND);
}
```

3 Security

Ein integraler Bestandteil der Funktionalität der Applikation ist die *Security*, genauer; die Authentifizierung und Authorisierung der Nutzer. Bei der Authentifizierung wird sichergestellt, dass ein Nutzer tatsächlich der ist, für den er sich ausgibt, bei der Authorisierung wird geprüft, ob es einem Nutzer erlaubt ist, das zu tun, was er möchte.

3.1 Authentifizierung

Da HTTP ein *zustandsloses* Protokoll ist, muss jede einzelne Request vom Frontend ans Backend authentifiziert werden. Es gibt für die Sicherung von REST-Services verschiedene Möglichkeiten, zum Beispiel *JSON Web Tokens (JWT)*, welches eine in der Praxis weit verbreitete Methode ist.

Es wurde sich jedoch gegen ein *komplexeres* Verfahren entschieden. Stattdessen wurde die die Authentifizierung so simpel wie gerade noch sicher gestaltet. Als Authentifizierungsmethode wird die *HTTP-Basic*-Authentifizierung genutzt, bei der Nutzername und Passwort *Base64*-kodiert, also fast im Klartext, mit jeder Request im Authentifizierungsheader der Request übertragen werden. Diese Methode ist nur sicher, solange mit dem Server über HTTPS kommuniziert wird, damit ein Dritter nicht einfach die Zugangsdaten abhören kann, wenn er den Netzwerkverkehr überwacht. Der Server, auf dem die Applikation über den Verlauf des Projekts gehostet wurde, hat das für HTTPS nötige TLS-Zertifikat und erlaubt nur Kommunikation über HTTPS, daher ist die Methode als sicher anzusehen.

Die Implementierung der Authentifizierung erfolgte mittels *Spring Security*. Hier wird wieder viel abstrahiert. Spring Security stellt das Interface `UserDetailService` bereit, welches die meisten Implementierungsdetails verbirgt. Im Projekt wird nur eine Methode überschrieben, und zwar die, die angibt, wie man die Daten eines Mitglieds abhängig vom Username lädt.

In einer zentralen Konfigurations-Klasse ist dann definiert, dass alle Requests authorisiert sein müssen und dass der Algorithmus *BCrypt* zum hashen der Passwörter verwendet wird.

3.2 Authorisierung

Bei der Authorisierung wird ebenfalls viel von Spring Security erledigt. Es wird eine Methode überschrieben, die definiert, wie die Rollen, die ein Mitglied hat, ermittelt werden. Listing 10 zeigt einen Ausschnitt der Methode, bei der anhand der *Offices* (Ämter) des Mitglieds die Berechtigungen (Rollen) bestimmt werden.

Listing 10: Bestimmung der Rollen anhand der Ämter

```
Collection<SimpleGrantedAuthority> authorities = getOffices()
        .stream()
        .map(off -> new SimpleGrantedAuthority("ROLE_" +
            off.toString()))
        .collect(Collectors.toList());
```

In den Controllern wird dann geprüft, ob ein Mitglied die benötigten Rechte für den Aufruf der Route hat. In Listing 11 wird geprüft, ob das Mitglied ein aktives Mitglied ist (`hasRole('ACTIVE')` und ob es auch sein eigenes Logbuch bearbeitet. `#memberId` ist der Parameter in der Request, der angibt, wessen Logbuch bearbeitet werden soll, `principal.id` ist die Id des Mitglieds, welches die Request durchführt. Diese Bedingungen werden in der `@PreAuthorize`-Annotation oberhalb der Methodendefinition für die zu schützende Route angegeben.

Listing 11: Beispiel Berechtigungsprüfung

```
@PreAuthorize("hasRole('ACTIVE') and #memberId ==
    principal.id")
@PostMapping(path = "/{memberId}/pilotlogentry")
public PilotLogEntry addPilotLogEntry(@RequestBody
    PilotLogEntry pilotLogEntry, @PathVariable int memberId) {
    ... }
```

4 Unit-Testing

Um die Codequalität zu gewährleisten, wurde über den Verlauf des Projekts stark auf Unit-Tests gesetzt. Es wurden insgesamt 53 Tests definiert, wobei die meisten die Controllerfunktionalitäten prüfen.

Der Test in Listing 12 zeigt einen typischen Unit-Test. Hier wird geprüft, dass es nur einmal möglich ist, einen jährlichen Dienst zu speichern. Es soll möglich sein, zu speichern, dass ein Mitglied im aktuellen Jahr als Fluglehrer tätig war. Es soll aber nicht möglich sein, dies noch ein zweites mal für das selbe Jahr zu speichern.

Mit dem `mockMvc` werden Requests durchgeführt und es wird geprüft, dass der richtige HTTP-Statuscode zurückgegeben wird. Beim ersten Mal wird ein erfolgreiches Speichern erwartet, welches durch den Code `204 - No Content` symbolisiert wird. Bei der zweiten Anfrage wird erwartet, dass das Backend die fehlerhafte Anfrage vom Client mit dem Statuscode `400 - Bad Request` abweist.

Es gibt auch andere Tests, in denen zum Beispiel kontrolliert wird, dass die korrekte Anzahl an Einträgen in die Datenbank vorgenommen wurde.

Zu Beachten in dem Listing ist die Methode `TestUtil.marshal()`. Hier wird der Inhalt der Request von einem Java-Objekt zu JSON gemarshalt. Leider funktioniert das Marshalling in den Unit-Tests nicht *einfach so* wie in der eigentliche Applikation, stattdessen musste diese Marshalling-Methode implementiert werden.

Besonders nervig war, dass Datums-Objekte explizit definierte Serialisierungs-Klassen benötigten. Diese geben z. B. an, dass ein Datumsobjekt in das Format `YYYY-MM-DD hh:mm:ss` umgewandelt werden soll. Dies war relativ Aufwändig, hat aber auf der anderen Seite Einblicke in die Funktionsweise der Jackson-Bibliothek, die Spring für JSON-Serialisierung nutzt, gegeben.

Listing 12: Beispielhafter Unit-Test

```
@Test
@WithMockUser(roles = {"SYSTEMADMINISTRATOR"})
public void testJaehrlicherServiceDoppelt() throws Exception {

    Member mem = TestUtil.saveAndGetMember(memberRepository,
        officeRepository, enc, "wasGeht1");
    Service ys = new Service(ServiceName.J_FLUGLEHRER,
        getNextBillingDate().minusYears(1),
            getNextBillingDate().minusDays(1), 123);

    mockMvc.perform(post("/services/" + mem.getId())
            .contentType(MediaType.APPLICATION_JSON)
            .content(TestUtil.marshal(ys)))
            .andExpect(status().isNoContent());

    mockMvc.perform(post("/services/" + mem.getId())
            .contentType(MediaType.APPLICATION_JSON)
            .content(TestUtil.marshal(ys)))
            .andExpect(status().isBadRequest());
}
```

5 Tooling

Neben dem *Werk*, dem Quellcode, sei auch ein Blick auf die *Werkzeuge* geworfen. Im Folgenden soll kurz beschrieben werden, welche Entwicklungs- und Kollaborationstools hilfreich waren und welche Rolle diese spielten.

5.1 Git

Eine der zentralen Herausforderungen beim Programmieren ist die Versionsverwaltung. Die Software *Git* ist hier die Standardlösung. Auch im Projekt wurde Git von den Entwicklern genutzt, um den lokalen Stand der Entwicklung zu verwalten.

Git hat eine sehr wichtige Rolle gespielt und es sollte selbstverständlich sein, dass ein Softwareprojekt ein Versionsverwaltungssystem nutzt. Eventuell kritisch zu sehen ist der Umstand, dass keiner der Entwickler sehr tiefe Kenntnisse in der Software hat. So wurde zum Beispiel das *Rebasing* kaum eingesetzt, obwohl an einigen Stellen definitiv sinnvoll wäre. Abbildung 7 zeigt den teilweise etwas chaotischen Verlauf der Entwicklungszweige (Branches).

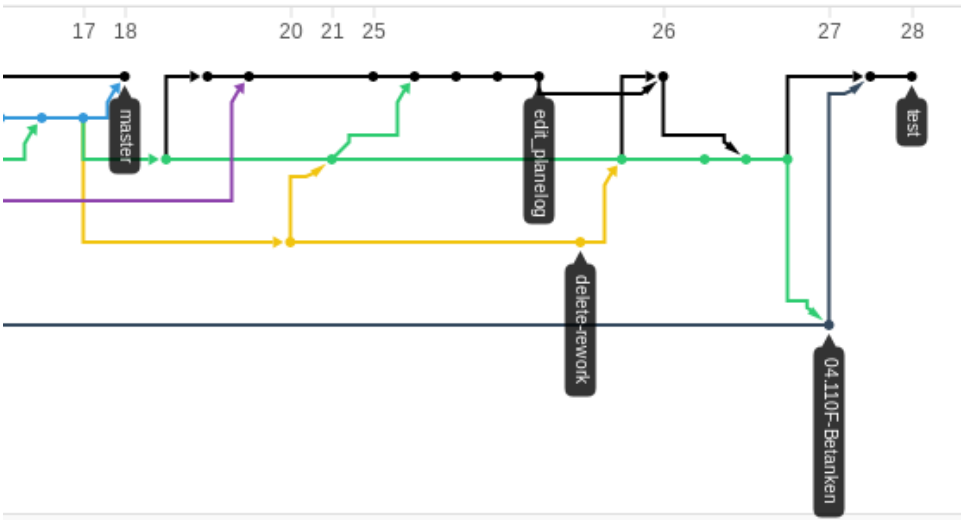

Abbildung 7: Netzwerk-Graph der Git-Branches

5.2 GitHub

Die Funktionalität von Git wird erweitert durch Github. Dies bietet die Möglichkeit, ein zentralen Punkt der Entwicklung zu haben und bringt einige wichtige *soziale Funktionen* mit sich.

Die wichtigste hierbei ist wohl die *Pull Request*. Dies ist die Bitte an andere, Änderungen aus dem eigenen Branch in einen anderen Branch zu ziehen. Ein Beispiel dafür kann sein, dass ein Entwickler ein Feature in einem Branch entwickelt hat und nun diesen Branch in den Hauptbranch mergen will. Dazu erstellt er eine Pull Request, und bittet dabei andere Entwickler, seinen Code zu reviewen. Im Backend wurden insgesamt knapp 20 Pull Requests geöffnet.

Neben Pull Requests zur Diskussion über Quellcode gibt es die Möglichkeit, über *Issues* Fehler zu dokumentieren. die erwies sich insbesondere in den Kommunikation zwischen den Teams als nützlich.

Ebenfalls sehr praktisch war der Discord-Bot, der in verschiedenen Kanälen alle nennenswerten Ereignisse im GitHub gepostet hat. So war es motivierten Teilnehmern einfach möglich, die Arbeit der anderen zu verfolgen.

5.3 Discord

Discord wurde als zentrale Chat-Plattform genutzt, zum einen, da es die Möglichkeit gibt, auf einem Server verschiedene Räume für verschiedene Gruppen zu gestalten und zum anderen, weil einige Teilnehmer Discord bereits privat verwenden.

Die Kommunikation über Discord ist insgesamt als positiv zu bewerten, auch wenn einige Teilnehmer schlecht erreichbar waren oder manche Nachrichten überlesen wurden.

5.4 IntelliJ IDEA

Als *integrierte Entwicklungsumgebung* wurde IntelliJ IDEA genutzt. IntelliJ bietet etwas mehr Features als Eclipse und ist generell besser *benutzbar*, da z. B. schneller.

Besonders hilfreich war die eingebaute Git-Unterstützung, die bei Merge-Konflikten sehr hilfreich war. Auch die Möglichkeit, sehr einfach Dateien zu *diffen*, also die Unterschiedlichen Versionen zu vergleichen, war sehr angenehm.

5.5 Datenbank

Bei der Datenbank für die lokale Entwicklung gab es verschiedene Ansätze. Einige Entwickler haben die von Spring automatisch konfigurierte H2-Datenbank genutzt, die den geringsten (keinen) Einrichtungsaufwand hat. Die Entwickler mit etwas größerem Interesse an den Ereignissen innerhalb der Datenbank nutzten lokale MySQL/MariaDB-Instanzen, auf die per HeidiSQL, DBeaver oder per Kommandozeile zugegriffen wurde.

5.6 REST-Client

Zum händischen Testen der definierten Routen wurden *Postman* und *HTTPie* genutzt. Es war wichtig, die Routen unabhängig vom Frontend selbst testen zu können, da die Entwicklung so weniger abhängig vom Fortschritt der Frontendentwicklung war. Dazu wurde insbesondere Postman genutzt. Mit diesem Tool können HTTP-Anfragen einfach über eine grafische Oberfläche durchgeführt werden.

Einige nutzten hier stattdessen HTTPie, ein kommandozeilenbasiertes Tool, welches für schnelle unkomplizierte Abfragen etwas schlanker zu benutzen war.

5.7 ripgrep

Eine kurzer aber liebevoller Blick sei noch auf *ripgrep* geworfen, ein kommandozeilenbasiertes Tool zum durchsuchen von Textdateien per regulären Ausdrücken *(Regex)*, dass dem UNIX-Programm *grep* ähnelt. Das Tool wurde zwar nur von einem Entwickler intensiv genutzt, erwies sich aber als wertvolle Hilfe, um z. B. alle Vorkommen einer Methode oder einer Annotation im Projekt zu suchen.