

Studienarbeit

Testen von Microservices in Spring Boot

Dennis Brysiuk
Hochschule für Angewandte Wissenschaften Hof

Testverfahren für komplexe Software-Systeme
Prof. Dr. Matthias Meiner
Sommersemester 2019

Inhaltsverzeichnis

1. Einführung	3
2. Microservice	3
2.1 Datenpersistenz	4
2.1.1 Persistence Entity und Objektrelationale Metadaten	4
2.1.2 Persistence Query Language	4
2.2 Business Logik	5
2.2.1 Serviceschicht	6
2.2.2 Utils Bibliotheken	6
2.3 REST API	6
3. Testverfahren	7
3.1 Datenschicht Test	7
3.2 Serviceschicht Test	9
3.3 Utils Tests	12
3.4 Controller Schicht Tests	13
4. Fazit	14
Abbildungsverzeichnis	15
Begriffsverzeichnis	15
Literaturverzeichnis	16

1. Einführung

In der vorliegenden Arbeit wird eine Einführung des Spring Boot Frameworks, der Architektur der Microservices und dem Testen mithilfe des Spring Boot Test Frameworks behandelt.

Das Spring Framework wurde 2002 erstmals als Idee vorgestellt und ein Jahr später als quelloffenes Projekt veröffentlicht [1]. Es ist also ein Open Source Java-Framework, das im Kern einen Dependency-Injection-Mechanismus, aber auch viele weitere Funktionalitäten bereitstellt [2]. Unter anderem lassen sich damit sehr gut und effektiv Microservices erstellen.

Microservices dienen zur Modularisierung von Software [1]. Mit Microservices lässt sich eine komplexe Anwendung aus unabhängigen Prozessen zusammenstellen, die untereinander mit sprachunabhängigen Schnittstellen kommunizieren [3].

Für die Einfügung des Spring Boot Microservices wurde ein vereinfachter Mensa Microservice mit Spring Boot geschrieben. Der Service liest die Webseite des [Studentenwerks Oberfranken](#) zeitgesteuert aus, extrahiert den Inhalt des Speiseplans für alle Tage und legt diesen in eine Datenbank ab. Die wichtigste Funktionalität ist das personalisierte bzw. filtrierte abrufen des gespeicherten Speiseplans über eine Schnittstelle. Genau diese Funktionalität des Mensa Microservices wird in dieser Arbeit getestet.

2. Microservice

Das Ziel eines Microservices ist eine vereinfachte Architektur und eine erhöhte Sichtbarkeit von Interaktionen. Er setzt sich auf mehrschichtige, hierarchische Systeme ("Layered System") auf. Jede Komponente des Systems kann ausschließlich die jeweils direkt angrenzenden Schichten sehen [4].

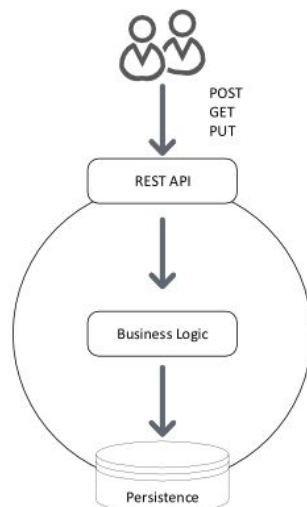


Abbildung 1: Microservice Schichten Architektur

Damit der Mensa Microservice getestet werden kann, muss man zuerst verstehen, wie so ein Service aufgebaut ist und welche Aufgaben die einzelne Schichten übernehmen. In den folgenden Kapiteln werden die Schichten aus *Abbildung 1* von unten nach oben abgearbeitet.

2.1 Datenpersistenz

Die Datenpersistenz ist ein Mittel, mit dem eine Anwendung Informationen in einem nicht flüchtigen Speichersystem persistent speichern und aus diesem abrufen kann [5]. Die Datenpersistenz in Microservice Architekturen ist bekannt als Datenschicht, Persistenzschicht, Repository, JPA oder als DAO.

Die Datenschicht wird als eine Schnittstelle implementiert, die die Zuordnung und die Übertragung von Objekten zu Datenbankeinträgen vereinfacht. Die Datenschicht besteht aus folgenden Komponenten: Persistence Entity, Objektrelationale Metadaten und Java Persistence Query Language.

2.1.1 Persistence Entity und Objektrelationale Metadaten

Eine Entity ist eine einfache Java-Klasse, die normalerweise auf eine einzelne Tabelle in der relationalen Datenbank abgebildet wird. Instanzen dieser Klasse entsprechen hierbei den Zeilen der Tabelle [6].

Objektrelationale Metadaten stellen die Beziehungen zwischen den einzelnen Tabellen und werden meist als Java-Annotationen angelegt [7]. Dadurch geschieht die Verarbeitung der meisten Datenbankoperationen ohne dass JDBC- oder SQL-Code für die Verwaltung der Persistenz geschrieben werden muss.

Im Mensa Microservice heißt die verwendete Entity *DishDO* und ist folgendermaßen aufgebaut:

```
@Entity
@Table(name="dishes")
public class DishDO {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String date;
    private String name;
    private String price;
    private String info;
    @Enumerated(EnumType.STRING)
    private TypeUtil type;
    // standart getters and setters, constructors
}
```

Die Annotation *@Entity* definiert die Java-Klasse als Persistence Entity und in *@Table* wird der Name der Tabelle angegeben, unter dem die Entity abgelegt werden soll. Weitere Annotationen *@Id*, *@GeneratedValue* und *@Enumerated* definieren die objektrelationalen Metadaten.

2.1.2 Persistence Query Language

Die JPQL wird genutzt, um Anfragen bezüglich der in der Datenbank gespeicherten Entitäten durchzuführen. Diese Anfragen ähneln syntaktisch SQL-Abfragen, beziehen sich aber auf Entitäten statt auf Datenbanktabellen. Die JPA-Implementierung überführt die in JPQL formulierten Abfragen zur Laufzeit in SQL-Statements, die vom Ziel-Datenbanksystem verstanden werden [\[7\]](#).

Der Mensa Microservice verwendet dafür die Klasse *MensaDao*:

```
public interface MensaDao extends CrudRepository<DishDO, Integer> {
    List<DishDO> findByTypeIn(List<TypeUtil>types);
    List<DishDO> findByDate(String date);
    List<DishDO> findByDateGreaterThanOrEqualAndDateLessThanOrEqual(String
        date, String dateUntil);
}
```

Das Spring Boot Framework enthält eine Schnittstelle namens *CrudRepository*. Über die Schnittstelle wird der Datenzugriff deutlich erleichtert [\[8\]](#). Es sind bereits die Basis Methoden für CRUD-Operationen vorhanden und müssen nicht mehr separat in der Schnittstelle *MensaDao* definiert werden.

```
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S var1);
    <S extends T> Iterable<S> saveAll(Iterable<S> var1);
    Optional<T> findById(ID var1);
    boolean existsById(ID var1);
    Iterable<T> findAll();
    Iterable<T> findAllById(Iterable<ID> var1);
    long count();
    void deleteById(ID var1);
    void delete(T var1);
    void deleteAll(Iterable<? extends T> var1);
    void deleteAll();
}
```

Das Interface *MensaDao* erweitert lediglich die Schnittstelle *CrudRepository* mit den Typen-Spezifikationen der Entity Klasse *DishDO* und dem Datentyp *Integer* des Id Parameters.

Jedoch benötigt der Mensa Microservice weitere Methoden für Datenzugriff in der Datenbank. Diese lassen sich sehr einfach dank der JPA erstellen. JPA kann aus Methodennamen die SQL-Befehle ableiten.

2.2 Business Logik

Business Logik erlaubt es, das Verhalten der Anwendung anzupassen. Diverse Anfragen werden bearbeitet, geparkt und weitergeleitet, sowie Verbindungen aufgebaut, Daten angefragt und spezifische Funktionen aktiviert. Im Fokus steht der Datenaustausch, die Optimierung und die Validierung der Daten. In der Microservice Architektur wird diese Schicht als Serviceschicht bezeichnet.

2.2.1 Serviceschicht

Die Serviceschicht besteht aus einer Schnittstelle, in der alle Methoden enthalten sind, die für die Applikation notwendig sind, sowie einer Java-Klasse, in der die Methoden aus der Schnittstelle die Logik enthalten.

In dem Mensa Microservice heißt die Serviceschicht Schnittstelle *MensaService* und die Implementierungs-Klasse *MensaServiceImpl*.

```
@Service
public class MensaServiceImpl implements MensaService{
    @Autowired
    private MensaDao mensaDao;
    //Methods ...
}
```

Mit der Annotation *@Service* wird markiert, dass die Klasse die Business Logik der Anwendung enthält. Zu beachten ist auch, dass die Datenschicht *MensaDao* in die Serviceschicht eingebunden wird. Weil die Serviceschicht über der Datenschicht steht, behandelt sie die erhaltenen Anfragen und leitet diese in einem akzeptablen Format an die Datenschicht weiter.

Die benötigten Methoden für die Serviceschicht im vereinfachten Mensa Microservice sind in der Schnittstelle *MensaService* dokumentiert.

Des öfteren nutzt die Serviceschicht Utils Bibliotheken. Dies ermöglicht die Komplexität der Business Logik zu vereinfachen und einheitlich zu trennen.

2.2.2 Utils Bibliotheken

Utils Bibliotheken sind Hilfsklassen die bereits in Java enthalten sind oder speziell für die Anwendung entwickelt wurden. Meistens übernehmen die Utils Funktionalitäten, die an mehreren Stellen in der Anwendung benötigt werden.

In der Serviceschicht des Mensa Microservice werden die Hilfsklassen *DayDateUtil* und *TypeUtil* verwendet. *DayDateUtil* wandelt ein Wochentag in ein Datum um und *TypeUtil* enthält alle gültigen Kategorien, die im Speiseplan zur Verfügung stehen. Weitere Utils werden in dieser Arbeit aus komplexitäts Gründen nicht behandelt.

2.3 REST API

REST steht für REpresentational State Transfer, API für Application Programming Interface [\[4\]](#). Die Schnittstelle versetzt das System in die Lage, Daten und Aufgaben auf unterschiedliche Server zu verteilen oder mithilfe eines HTTP-Requests anzufordern [\[9\]](#). In einer Microservice Architektur wird diese Schicht als Controller Schicht bezeichnet.

Im Großen und Ganzen ist der Controller für die Navigation zwischen verschiedenen Komponenten und Views zuständig und erleichtert somit die Wartbarkeit der API.

```

@RestController
@RequestMapping("/api")
public class MensaController {
    @Autowired
    MensaService mensaService;
    //Methods ...
}

```

`@RestController` ist eine Kombination aus `@Controller` und `@ResponseBody`. `ResponseBody` bewirkt, dass keine View, sondern stattdessen der Rückgabewert der Methode als JSON serialisiert zurückgeliefert wird. Welcher Request durch welche Methode bearbeitet wird, wird mit `@RequestMapping` konfiguriert. Durch die `@RestController`-Annotation weiß das Spring Framework, dass es sich bei dieser Klasse um einen Controller handelt, also dass HTTP Requests hierher geroutet werden können [2]. Außerdem ist der Controller von der Serviceschicht abhängig. Deswegen wird die Schnittstelle `MensaService` injiziert.

3. Testverfahren

Nachdem die Architektur und die Funktionalitäten der einzelnen Komponenten des Microservices klar sind, kann das Testen des Microservices beginnen. Um Microservices in Spring zu Testen reichen zwei Arten von Tests, Integrationstests und Unittests. Integrationstests dienen dazu, verschiedene voneinander abhängige Komponenten eines komplexen Systems im Zusammenspiel zu testen [10]. Wichtig ist, dass manche Komponenten integriert werden müssen, also werden fehlende Module simuliert und andere Komponenten werden gleichzeitig zusammengefügt und in das zu testende System integriert. In Unit Tests werden die Einzelteile des Systems auf ihre korrekte Funktionalität geprüft [11].

Die Test-Frameworks für Integrationstests und Unittests stellt Spring-Boot mit dem Framework `spring-boot-starter-test` zu Verfügung. Dieses muss lediglich über eine Maven-Dependency in das Projekt eingefügt werden. Außerdem muss noch eine H2-Datenbank als Maven-Dependency eingefügt werden. Eine H2-Datenbank ist eine In-Memory Datenbank. Diese wird für das Testen des Datenschicht benötigt.

3.1 Datenschicht Test

In [Kapitel 2.1](#) wurde die Funktionalität der Datenschicht ausführlich erklärt. Jetzt kann die Funktionalität unabhängig von anderen Schichten getestet werden. Das Testen der Datenschicht für den Mensa Microservice erfolgt in der Klasse `MensaRepositoryTest`.

```

@RunWith(SpringRunner.class)
@DataJpaTest
public class MensaRepositoryTest {
    @Autowired
    private TestEntityManager entityManager;
    @Autowired
    private MensaDao mensaDao;
    //Tests
}

```

Die Annotation `@RunWith(SpringRunner.class)` wird benötigt, um eine Verbindung zwischen den Spring-Boot-Testfunktionen und JUnit zu bilden. Desweiteren wird noch die Annotation `@DataJpaTest` benötigt. Diese bietet einige Standardeinstellungen, die zum Testen der Datenschicht dem Konfigurieren der H2-Datenbank, der Einstellungen für Hibernate, Spring Data und DataSource sowie dem Loggen von SQL-Statements notwendig ist [\[12\]](#).

Um einen Datenbank-Vorgang auszuführen, müssen einige Datensätze bereits in der H2-Datenbank eingerichtet sein. Der von Spring Boot bereitgestellte `TestEntityManager` ist eine Alternative zum `Standart-JPA-EntityManager`, der Methoden bereitstellt, die beim Schreiben von Tests verwendet werden können [\[12\]](#). Die Initialisierung von H2 wird in der `setUp()` Methode durchgeführt. Die Methode ist mit `@Before` annotiert. Dies bedeutet, dass vor den Test Methoden erst die `setUp()`-Methode aufgerufen werden muss.

```
@Before
public void setUp () {
    List<DishDO> items = new ArrayList<>(...);
    entityManager.persistAndFlush(items);
}
```

Die Klasse `MensaDao` ist der tatsächliche Komponent, der getestet werden soll. Die Annotation `@Autowired` ermöglicht es, die Spring-Beans zu deklarieren und die Bean-Referenzen zu injizieren.

Folgende Kriterien muss die Datenschicht im Mensa Microservice erfüllen:

Bedingung	Erwartetes Ergebnis
findAll()	
Methode wird aufgerufen	Es werden alle Einträge zurückgegeben
findByTypeIn(List<TypeUtil>types)	
Methode wird mit leerer liste aufgerufen	Es werden alle Einträge zurückgegeben
Methode wird mit einen Typ aufgerufen	Es werden alle Einträge für nur diesen Typ zurückgegeben
Methode wird mit mehreren Typen aufgerufen	Es werden alle Einträge für nur diese Typen zurückgegeben
findByDate(String date)	
Methode wird mit falsch formatierten Datumswert aufgerufen	Es werden keine Einträge zurückgegeben
Methode wird mit richtig formatierten Datumswert aufgerufen	Es werden alle Einträge nur für dieses Datum zurückgegeben
Methode wird mit Datum aufgerufen welches nicht in Datenbank vorhanden ist	Es werden keine Einträge zurückgegeben

findByDateGreaterThanEqualAndDateLessThanEqual(String from, String until)	
Methode wird mit Datumswerten aufgerufen welche in Datenbank vorhanden sind	Es werden alle Einträge für dieses Zeitintervall sowie inklusiv die Einträge mit jeweils übergebenen Datum zurückgegeben
Methode wird mit Datumswerten aufgerufen das in Datenbank nicht vorhanden sind	Es werden Einträge nur dann zurückgegeben, falls diese in den Zeitintervall liegen
Methode wird mit ersten Datum in der Zukunft und zweiten Datum in der Vergangenheit aufgerufen.	Es werden keine Einträge zurückgegeben
Methode wird mit falsch formatierten Datumswerten aufgerufen	Es werden alle Einträge zurückgegeben
Methode wird nur mit ersten falsch formatierten Datumswert aufgerufen (zweiter Datum ist korrekt formatiert)	Es werden keine Einträge zurückgegeben
Methode wird nur mit zweiten falsch formatierten Datumswert aufgerufen (erster Datum ist korrekt formatiert)	Es werden alle Einträge ab dem ersten Datum zurückgegeben

Die entstandenen Tests sind in der Klasse *MensaRepositoryTest* zu finden. Ob die Kriterien erfüllt wurden, wird mit Hilfe des AssertJ Framework überprüft. Dieses Framework ist im *spring-boot-starter-test* Framework enthalten.

3.2 Serviceschicht Test

Durch die Beschreibung der Serviceschicht aus dem [Kapitel 2.2.1](#) lässt sich herausnehmen, dass die Serviceschicht von der Datenschicht stark abhängig ist. Jedoch soll die Serviceschicht getestet werden ohne zu wissen, wie die Datenschicht implementiert ist. Um dies zu ermöglichen wird der Dienst von “mocking” verwendet, welcher bereits in *spring-boot-starter-test* Framework zur Verfügung steht.

```
@RunWith(SpringRunner.class)
public class MensaServiceImplTest {
    @TestConfiguration
    static class MensaServiceImplTestContextConfiguration {
        @Bean
        public MensaService mensaService () {
            return new MensaServiceImpl();
        }
    }
    @Autowired
    private MensaService mensaService;
    @MockBean
    private MensaDao mensaDao;
    // Tests
}
```

Um die Service-Klasse testen zu können, muss eine Instanz der Service-Klasse erstellt werden und als Bean zur Verfügung stehen, damit die Service-Klasse mit `@Autowired` in die Testklasse injiziert werden kann. Diese Konfiguration wird mithilfe der Annotation `@TestConfiguration` erreicht. Bei Erstellung der Microservices scannt Spring Boot nach allen Komponenten und Konfiguration, damit diese in den Microservice geladen und initialisiert werden können. Damit aber die Komponente oder Konfigurationen nicht aus den Testklassen in den tatsächlichen Service geladen werden, bietet Spring Boot die Annotation `@TestConfiguration an`, welche es verhindert, beim Scan nicht erfasst zu werden [\[12\]](#).

Eine weitere interessante Sache ist die Verwendung von `@MockBean`. Es wird ein **Mock** für die Datenschicht `MensaDao` erstellt, der mit dem Aufruf an die eigentliche Datenschicht umgehen kann, denn das Ziel ist, wie bereits erwähnt, nur die Serviceschicht zu testen - unabhängig von anderen Komponenten [\[12\]](#).

Folgende Kriterien muss die Serviceschicht im Mensa Microservice erfüllen:

Bedingung	Erwartetes Ergebnis
getMenu(List<String>categories)	
Methode wird aufgerufen mit Kategorien von Datentyp null	Die Kategorien werden richtig geparkt und an die richtige Methode in der Datenschicht übergeben welches alle Einträge zurückgibt.
Methode wird aufgerufen mit nicht vorhandenen bzw. falschen Kategorien	Die Kategorien werden richtig geparkt und an die richtige Methode in der Datenschicht übergeben welche keine Einträge zurückgibt.
Methode wird aufgerufen mit ein paar nicht vorhandenen bzw. falschen Kategorien und mit vorhanden bzw richtigen Kategorien	Die Kategorien werden richtig geparkt und an die richtige Methode in der Datenschicht übergeben welches alle Einträge zurückgibt für vorhandene bzw richtigen Kategorien.
Methode wird aufgerufen nur mit vorhanden bzw richtigen Kategorien	Die Kategorien werden richtig geparkt und an die richtige Methode in der Datenschicht übergeben welches alle Einträge zurückgibt für vorhandene bzw richtigen Kategorien.
Methode wird aufgerufen mit Kategorielliste die leer ist	Die Kategorielliste wird richtig geparkt und an die richtige Methode in der Datenschicht übergeben welches keine Einträge zurückgibt.
getMenuToday()	
Methode wird aufgerufen	Es werden nur Einträge für das heutiges Datum zurückgegeben.

getMenuByDate(Date date)	
Methode wird aufgerufen mit beliebigen Datum	Beim übergebenen Datum entstehen keine parsing fehler und es werden nur Einträge für das Datum zurückgegeben bzw keine Einträge falls für das Datum keine Einträge existieren.
Methode wird aufgerufen mit falschen Datumsformat	Beim übergebenen Datum entstehen keine parsing fehler und es wird eine Exception zurückgegeben
getMenuIntervalByDate(Date from, Date until)	
Methode wird aufgerufen mit gültigen Datumsintervall	Beim übergebenen Datum entstehen keine parsing fehler und es werden nur Einträge für das Datumsintervall zurückgegeben bzw keine Einträge falls für das Datumsintervall keine Einträge existieren.
Methode wird aufgerufen mit ungültigen Datumsintervall	Beim übergebenen Datum entstehen keine parsing fehler und es wird eine Exception zurückgegeben
Methode wird aufgerufen mit einen gültigen Datumswert und einen ungültigen	Beim übergebenen Datum entstehen keine parsing fehler und es wird eine Exception zurückgegeben
Methode wird aufgerufen mit ersten Datum größer als der zweites.	Beim übergebenen Datum entstehen keine parsing fehler und es wird eine Exception zurückgegeben
Methode wird aufgerufen mit Datumswerten die gleich sind	Beim übergebenen Datum entstehen keine parsing fehler und es werden nur Einträge für das Datum zurückgegeben bzw keine Einträge falls für das Datum keine Einträge existieren.
getMenuActualWeek()	
Methode wird aufgerufen	Es werden nur Einträge für die aktuelle Woche zurückgegeben.
getMenuByWeek(String week)	
Methode wird aufgerufen mit gültigen parameter Woche	Beim übergebene Woche entstehen keine parsing fehler und es werden nur Einträge für die übergebenen parameter Woche zurückgegeben.
Methode wird aufgerufen mit ungültigen parameter Woche	Beim übergebene Woche entstehen keine parsing fehler und es wird eine Exception zurückgegeben

getMenuByDay(String week, String day)	
Methode wird aufgerufen mit gültigen parameter Woche und Wochentag	Beim übergebene übergebenen Parametern entstehen keine passing fehler und es werden nur Einträge für die übergebenen parameter Woche und Wochentag zurückgegeben.
Methode wird aufgerufen mit gültigen parameter Woche und ungültigen Wochentag	Beim übergebenen Parametern entstehen keine passing fehler und es wird eine Exception zurückgegeben.
Methode wird aufgerufen mit ungültigen parameter Woche und gültigen Wochentag	Beim übergebenen Parametern entstehen keine passing fehler und es wird eine Exception zurückgegeben.
getMenuIntervalByDay(String week, String from, String until)	
Methode wird aufgerufen mit gültigen parameter Woche und Wochentag Intervall	Beim übergebenen Parametern entstehen keine passing fehler und es werden nur Einträge für die übergebenen parametern zurückgegeben.
Methode wird aufgerufen mit gültigen parameter Woche und ungültigen Wochentag Intervall	Beim übergebenen Parametern entstehen keine passing fehler und es wird eine Exception zurückgegeben.
Methode wird aufgerufen mit gültigen parameter Woche und einen ungültigen Wochentag	Beim übergebenen Parametern entstehen keine passing fehler und es wird eine Exception zurückgegeben.
Methode wird aufgerufen mit gültigen parameter Woche und gleichen Wochentagen	Beim übergebenen Parametern entstehen keine passing fehler und es werden nur Einträge für das Wochentag zurückgegeben bzw keine Einträge falls für das Wochentag keine Einträge existieren.
Methode wird aufgerufen mit ungültigen parameter Woche und gültigen Wochentag Intervall	Beim übergebenen Parametern entstehen keine passing fehler und es wird eine Exception zurückgegeben.

Die entstandenen Tests sind in der Klasse *MensaServiceTest* zu finden.

3.3 Utils Tests

Von der Utils Bibliothek wird nur die Klasse *DayDateUtil* getestet. Die anderen Klassen werden in dieser Arbeit nicht getestet, entweder, weil sich keine sinnvolle Tests dazu erstellen lassen, oder, weil die Testfälle über den Rahmen dieser Arbeit hinausgehen.

Folgende Kriterien muss die Util Klasse *DayDateUtil* erfüllen:

Bedingung	Erwartetes Ergebnis
getCurrentWeekDay(DayOfWeek day)	
Methode wird aufgerufen mit einem beliebigen Wochentag	Es wird ein Datumswert für diesen Wochentag der aktuelle Woche zurückgegeben
getNextWeekDay(DayOfWeek day)	
Methode wird aufgerufen mit einem beliebigen Wochentag	Es wird ein Datumswert für diesen Wochentag der nächste Woche zurückgegeben

Die entstandenen Tests sind in der Klasse *DayDateUtilTest* zu finden.

3.4 Controller Schicht Tests

Aus dem Kapitel 2.3 kann herausgenommen werden, dass der Controller lediglich für das Routen von Anfragen zuständig ist. Wie in anderen Schichten ist es das Ziel, den Controller unabhängig von anderen Komponenten zu testen. Deswegen wird die Serviceschicht vorgetäuscht (**mocked**).

@MockBean bietet eine **mock** Implementation für benötigte Abhängigkeiten.

```
@RunWith(SpringRunner.class)
@WebMvcTest(MensaController.class)
public class MensaControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private MensaService mensaService;
}
```

Zum Testen von den Controller wird *@WebMvcTest* verwendet. Es konfiguriert automatisch die Spring MVC Infrastruktur für die Unittests sowie den **MockMvc**, welcher die Möglichkeit bietet, den MVC-Controller einfach und schnell zu testen, ohne einen HTTP-Server vollständig zu starten [12].

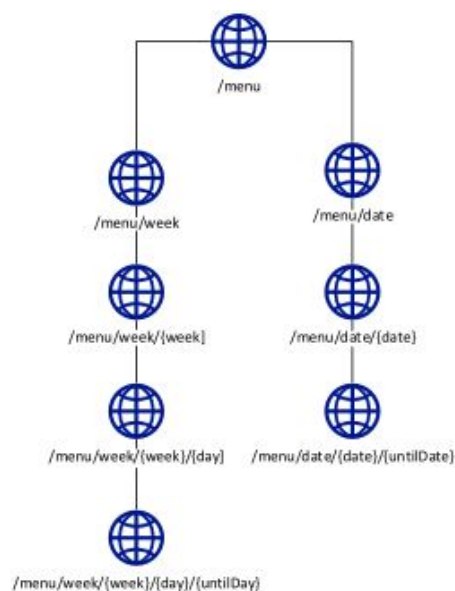


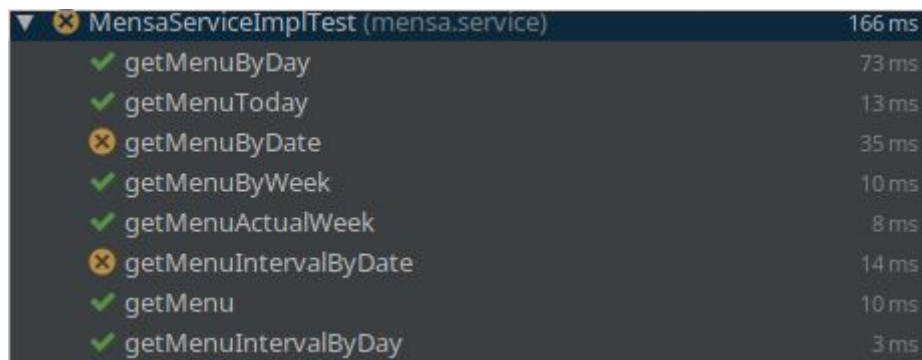
Abbildung 2: Wireframe Mensa Microservice REST API

Mithilfe von **MockMvc** können alle Methodenarten und Pfade getestet werden. Das Ziel ist die Abdeckung der REST API URLs aus der *Abbildung 2*.

Die entstandenen Tests sind in der Klasse *MensaControllerTest* zu finden.

4. Fazit

Testen ist ein wesentlicher Bestandteil der Entwicklung von Unternehmenssoftware. Beim Testen vom Mensa Microservice konnten einige Parse-Fehler in der Klasse *MensaServiceImpl* entdeckt werden.

A screenshot of a test runner interface showing the results of a test suite named 'MensaServiceImplTest (mensa.service)'. The suite has a total execution time of 166 ms and contains 9 individual tests. Most tests passed, indicated by green checkmarks, but two tests failed, indicated by orange 'X' icons: 'getMenuByDate' and 'getMenuIntervalByDate'. The execution times for each test are listed on the right side of the table.

Test Name	Execution Time
getMenuByDay	73 ms
getMenuToday	13 ms
getMenuByDate	35 ms
getMenuByWeek	10 ms
getMenuActualWeek	8 ms
getMenuIntervalByDate	14 ms
getMenu	10 ms
getMenuIntervalByDay	3 ms

Abbildung 3: Screenshot der Ausgabe des Ergebnisses von *MensaServiceImplTest*

Diese Parse-Fehler haben die Funktionalität nicht beeinflusst. Dennoch hätten diese Fehler entweder mutmaßlich ausgenutzt oder zum Absturz der Anwendung führen können. Deswegen ist es immer sinnvoll die Anwendung zu Testen bevor diese veröffentlicht wird, im Minimum den Komponenten, der für die Logik der Anwendung zuständig ist.

Abbildungsverzeichnis

Abbildung 1	Microservice Schichten Architektur	
Abbildung 2	Wireframe Mensa Microservice REST API	
Abbildung 3	Screenshot der Ausgabe des Ergebnisses von MensaServiceImplTest	

Begriffsverzeichnis

Dependency-Injection-Mechanismus	Abhängigkeiten eines Objekts zur Laufzeit initialisieren
JPA	Java Persistence API
DAO	Data Access Object
JDBC	Java Database Connectivity
SQL	Structured Query Language
DO	Data Object
JPQL	Java Persistence Query Language
CRUD	Create, Read, Update and Delete Operations
Maven	Build-Management-Tool
Hibernate	Open-Source-Persistenz Framework for Java
Spring Data	Bereitstellung von Models für den Datenzugriff
DataSource	Interceptor, request routing and access to data source (database)

Literaturverzeichnis

[1]	https://www.informatik-aktuell.de/entwicklung/programmiersprachen/4-jahre-spring-boot.html (abgerufen: 10.07.2019)
[2]	https://blog.mayflower.de/6950-spring-boot-rest-api.html (abgerufen: 10.07.2019)
[3]	https://de.wikipedia.org/wiki/Microservices (abgerufen: 11.07.2019)
[4]	https://www.cloudcomputing-insider.de/was-ist-eine-rest-api-a-611116/ (abgerufen: 11.07.2019)
[5]	https://www.ibm.com/support/knowledgecenter/de/SS7K4U_liberty/com.ibm.websphere.wlp.zseries.doc/ae/cwlp_jpa.html (abgerufen: 10.07.2019)
[6]	https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html (abgerufen: 09.07.2019)
[7]	https://de.wikipedia.org/wiki/Java_Persistence_API (abgerufen: 10.07.2019)
[8]	https://yourinsight.digital/2018/03/30/crudrepository/ (abgerufen: 10.07.2019)
[9]	https://www.datacenter-insider.de/was-ist-rest-api-a-714434/ (abgerufen: 11.07.2019)
[10]	https://de.wikipedia.org/wiki/Integrationstest (abgerufen: 11.07.2019)
[11]	https://de.wikipedia.org/wiki/Modultest (abgerufen: 11.07.2019)
[12]	https://www.baeldung.com/spring-boot-testing (abgerufen: 22.05.2019)