Set-up Guide
Beigabe zum Bachelorprojekt 2020/2021

TECHNISCHE UNIVERSITÄT DARMSTADT

Marcel Mittenbühler, Justin Peschke, Justin Karkossa, Amelie Oberkirch, Linda Pöhlmann

17.03.2021 **Gruppe 18**

Teil 1: Programme und Installation

0.1 Allgemein

Um die Website zu bearbeiten, wird eine Entwicklungsumgebung benötigt. Wir haben mit Visual Studio gearbeitet, es sind aber auch andere IDEs möglich (eclipse o.Ä.).

Front- und Backend sind getrennt, der Code für beides ist auf GitLab zu finden.

0.2 Frontend

Für die Entwicklung im Frontend wird React benötigt. Für die Nutzung von React muss außerdem npm installiert werden. In der Entwicklungsumgebung kann dann der Ordner "kreuzen-frontend"geöffnet werden, um auf den Quellcode zuzugreifen.

Für die Entwicklung im Frontend ist die Einarbeitung in die Sprache von 'React', 'Html', 'Bootstrap', 'styled-components' und 'Typescript' empfehlenswert.

1 Deployment Guide

Dieses Dokument beschreibt den Prozess, um Kreuzen auf einem Root-Server zu deployen. Aufgrund der Architektur ist auch ein Deployment in Cloud-Umgebungen wie Kubernetes denkbar, wird jedoch hier nicht weiter beschrieben.

1.1 Benötigte Software

Folgende Software muss installiert werden, um Kreuzen bereitzustellen:

- PostgreSQL (>= 10) (`https://www.postgresql.org/`)
- NGINX(`https://nginx.org/`)
- Java 11 (`https://openjdk.java.net/`)

Lokal muss außerdem noch NodeJS (`https://nodejs.org/`) und npm (https://www.npmjs.com/) installiert werden, um das Frontend zu kompilieren. Auf Ubuntu kann die benötigte Software beispielsweise mit folgenden Befehlen installiert werden:

```
# Postgres
## Create the file repository configuration:
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg main" >
/etc/apt/sources.list.d/pgdg.list'
## Import the repository signing key:
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -
## Update the package lists:
sudo apt-get update
## Install the latest version of PostgreSQL.
```

```
sudo apt-get -y install postgresql

# NGINX
sudo apt-get -y install nginx

# Java 11
sudo apt-get -y install openjdk-11-jdk
```

1.2 NGINX Setup

NGINX wird als reverse Proxy verwendet, um Traffic zu dem Front- und Backend zu lenken. Außerdem kann NGINX genutzt werden, um Verbindungen zu Clients mit SSL zu verschlüsseln.

Um in Nginx einen Serverblock als Reverse Proxy einzurichten, kann z.B. der default(‘/etc/nginx/sites-available/default‘) ersetzt werden. Wie ein neuer hinzugefügt wird, kann der NGINX Doku entnommen werden. Folgende Konfiguration kann verwendet werden:

```
    server {
        listen 80 default_server;
        listen [::]:80 default_server;

        root /home/kreuzen/frontend;

        index index.html;

        server_name _;

        # Frontend Server
        location / {
                try_files $uri $uri/ /index.html =404;
        }

        # Reverse Proxy for Backend
        location /api/ {
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_pass http://localhost:8080/;
        }
}
```

Im Servername kann statt des defaults auch der Domainname verwendet werden. Auch das Root-Directory kann nach belieben anders gewählt werden. Außerdem kann mit Certbot und Let’s Encrypt automatisch SSL eingerichtet werden.

Damit NGINX keinen "Forbidden"Fehler zurückgibt, benötigt NGINX lese Berechtigungen auf alle bereitgestellten Files, sowie Ausführungsberechtigungen in allen Parent Verzeichnissen.

```
    chmod 755 /home/kreuzen
    chmod 755 /home/kreuzen/frontend
    chmod 664 /home/kreuzen/frontend/*
```

1.3 Postgres Setup

Zuerst muss eine neue Rolle angelegt werden. In den meisten Fällen sollte der Name des Nutzers verwendet werden, unter dem Kreuzen laufen soll. Danach kann eine Datenbank für diesen Nutzer angelegt werden. Damit der Login über SSH ohne Passwort möglich ist, muss der Name der Rolle für die Datenbank verwendet werden.

```
    createuser kreuzen
    createdb kreuzen
```

Soll auf die Datenbank von extern (z.B. zur Entwicklung) müssen folgende Zeilen in die ‘pg_hba.conf‘ Datei hinzugefügt werden (Ubuntu mit PG 13 in: ‘/etc/postgresql/13/main/pg_hba.conf‘):

```
host    all             all             0.0.0.0/0               md5
host    all             all             ::/0                    md5
```

Danach sollte die DB restartet werden. Unter Ubuntu geht dies mit ‘sudo service postgres restart‘.

Damit das Backend sich mit der Datenbank verbinden kann, muss nun Passwort Login eingerichtet werden. Hierfür muss erstmal eine Shell Verbindung zur Datenbank mit ‘psql‘ aufgebaut werden.

Das Passwort kann dann mit ‘ALTER USER kreuzen WITH ENCRYPTED PASSWORD ’passwort hier einfügen’;‘ gesetzt werden.

Das Schema kann erstellt werden, indem alle Statements aus der ‘schema.sql‘ ausgeführt werden.

Da das Frontend momentan keine Möglichkeiten bietet, um Univsersitäten, Studiengänge und Studienabschnitte zu verwalten müssen diese auch manuell angelegt werden. Dies kann beispielsweise so erreicht werden:

```
INSERT INTO defi_university (name, allowed_mail_domains)
VALUES ('TU Darmstadt', '{"stud.tu-darmstadt.de"}');
INSERT INTO defi_major (university_id, name) VALUES (1, 'Informatik');
INSERT INTO defi_major_section (major_id, name) VALUES (1, 'Bachelor');
INSERT INTO defi_major_section (major_id, name) VALUES (1, 'Master');
```

1.4 Frontend Setup

Bevor das Frontend kompiliert werden kann, muss es konfiguriert werden. Hierzu muss in der ‘.env‘ Datei die Variable ‘REACT_APP_BACKEND_URL‘ auf die Url des Backends gesetzt werden. So muss dort z.B. ‘REACT_APP_BACKEND_URL=https://kreuzenonline.de/api stehen.

Das Frontend kann lokal mit npm folgendermaßen kompiliert werden:

```
npm install
npm run build
```

Hierdurch wird ein Build im ‘build‘ Order erstellt. Der Inhalt des ‘build‘ Ordners muss nun auf den Webserver, in unserem Fall ‘/home/kreuzen/frontend‘, kopiert werden.

Dabei werden schon ein Großteil der benötigten Libraries installiert. Allerdings gibt es einige Libraries, die zusätzlich hinzugefügt werden müssen. Diese werden mithilfe des Befehls

```
npm install library-name
```

installiert. Dabei ist zu beachten: Einzelne Wörter mit Bindestricht getrennt und keine Leerzeichen. Die Befehle stehen zumeist auf den Websites der Library-AutorInnen.

1.5 Backend Setup

Das Backend kann lokal oder auf dem Server mit Maven gebaut werden. Davor müssen jedoch noch die Konfigurationsdateien ausgefüllt werden. Hierfür muss im Backend Ordner die ‘src/main/resources/application.properties‘ folgendermaßen ausgefüllt werden.

```
spring.datasource.url=jdbc:postgresql://IP_DES_SERVERS:5432/kreuzen
spring.datasource.username=kreuzen
spring.datasource.password=POSTGRES_PASSWORT
jwt.key=EIN_SECRET_KEY
app.base-url=https://DOMAIN
app.smtp.password=EMAIL_SERVER_PASSWORT
app.smtp.username=EMAIL_SERVER_USERNAME@DOMAIN
app.locale=de_DE
spring.jackson.serialization.write-dates-as-timestamps=false
```

Danach kann mit ‘./mvnw clean package‘ unter Linux oder ‘mvnw.cmd clean package‘ unter Windows ein Build erstellt werden. Das Build wird unter ‘target/kreuzen.jar‘ gespeichert und kann mit ‘java -jar target/kreuzen.jar‘ ausgeführt werden.

Nun kann das Build auf den Server kopiert werden und dort ausgeführt werden.

Für ein Produktivsystem bietet es sich an die Ausführung mit einem Daemon einzurichten, damit das Backend im Hintergrund läuft und automatisch mit dem Server startet. Folgendermaßen kann ein solcher Daemon unter Ubuntu eingerichtet werden:

Der Service wird definiert durch Erstellen von ‘/etc/systemd/system/kreuzen-backend.service‘ mit folgendem Inhalt:

```
[Unit]
Description=Kreuzen Backend

[Service]
WorkingDirectory=/home/kreuzen/backend
ExecStart=/usr/bin/java -jar /home/kreuzen/backend/kreuzen.jar
User=kreuzen
Type=simple
Restart=on-failure
RestartSec=10

[Install]
WantedBy=multi-user.target
```

In dieser Konfiguration wird die ‘kreuzen.jar‘ in dem Verzeichniss ‘/home/kreuzen/backend‘ erwartet.
Um den Service zu aktivieren, muss noch folgendes ausgeführt werden:

```
# Start Service
systemctl start kreuzen-backend
# Enable Start on Boot
systemctl enable kreuzen-backend
```

Der Log kann mit ‘sudo journalctl -u kreuzen-backend.service‘ ausgelesen werden.

Teil 2: Entwicklung im Frontend

1.1 Überblick

Wichtig ist vor Allem alles im Ordner 'src', in dem sich der meiste selbstgeschriebene Code befindet.
Die Struktur des src-Ordners ist wie folgt:

1. **Ordner 'api':** In diesem Ordner befinden sich verschiedene Files, die eine Verbindung zu der Datenbank und dem Backend herstellen. So gibt es beispielsweise in der Datei 'semester.ts' mehrere Funktionen, die einen Zugriff auf die gespeicherten Semester(-daten) in der Datenbank ermöglichen. So können zum Beispiel alle (oder ausgewählte) Semster abgefragt, gelöscht oder bearbeitet werden.

2. **Ordner 'components':** Alle Komponenten, die ausgelagert wurden, wie zum Beispiel Bestätigungsfelder, Formulare oder Modals befinden sich im Ordner 'components'. Die Dateien sind auch hier noch einmal nach der Art der Seite, auf der sie verwendet werden, gruppiert. Verwendet werden sie im Ordner 'pages'.

3. **Ordner 'contexts':** Stellt den 'AuthContext' und 'AuthProvider'. Muss wahrscheinlich nicht erweitert werden. Funktionalität: Inhalt der Website kann an benutzerspezifische Einstellungen, auswahlen und generell an die Daten eines Nutzers angepasst werden.

4. **Ordner 'grafiken':** Grafiken werden hier abgelegt (aber nicht verwendet!).

5. **Ordner 'layouts':** Beeinhaltet die Layouts, die auf verschiedenen Seiten genutzt werden. In 'MainLayout' können Header und Footer hinzugefügt werden und außerdem wird der AuthContext auf der gesamten Website so zur Verfügung gestellt.

6. **Ordner 'pages':** Beeinhaltet im Wesentlichen den Code für die einzelnen Seiten, meistens aber eher strukturell. Einzelne ausgelagerte Komponenten dieser Seiten befinden sich im Ordner 'components'.
   Der Ordner sind sortiert (Admin, Verwaltung, Registrierung, Nutzer).

7. **Ordner 'scss':** Hier befindet sich die custom CSS. Allerdings lassen sich Design-Elemente besser über 'theme.ts', 'Header.tsx', 'GlobalStyle.tsx' und weitere Dateien bearbeiten.

8. **Weitere Dateien:** Wichtig sind vor Allem 'Routes.tsx' (für die Verlinkung der Seiten) und 'App.tsx' (Sehr grobe Struktur der Seiten). Über 'setupTests.ts' kann getestet werden.

9. **Design-Einstellungen:** Files, in denen das (globale) Design angepasst werden kann, sind 'theme.ts'(src/theme.ts), 'GlobalStyle.tsx' (src/components/General/GlobalStyle.tsx), 'MainLayout.tsx' (src/layouts/MainLayout.tsx) und 'custom.scss' (src/scss/custom.scss).

1.2 Eine neue Seite hinzufügen

1. **Ort:** Eine neue Seite sollte im Ordner 'pages' (.../kreuzen-frontend/src/pages) erstellt werden und dort in den richtigen Ordner integriert werden.

2. **Benennung:** 'Seitenname.tsx' zur Erstellung eines Typescript-Dokuments. (Anmerkung: Einige wenige Files benötigen auch den Dateityp .ts, hier am Besten einfach an verwandten Files orientieren.)

3. **Rahmen:** Neue Seiten haben automatisch grundlegende Eigenschaften der Website; zum Beispiel die Navigationsleiste inklusive Benutzer-Kontext und sollten, sofern möglich, eine Card enthalten, damit das Design zum Rest der Website passt.

4. **Verlinkung:** In Routes gibt es 4 Abschnitte für die verschiednen Rollen. Je nachdem, ob die Rolle zugriff auf die Seite haben soll, wird ein Objekt der jeweiligen Seite übergeben. Das sieht folgendermaßen aus:

   ```
   <Route exact path="route/des/links">
       <Seitenname />
   </Route>
   ```

   Die Seite muss außerdem importiert sein.

1.3 Allgemeines Design

- **Farben:** Farben sollten möglichst nicht einzeln festgelegt werden (Beispiel: Eine Farbe für einen Button) sondern für alle Komponenten der gleichen Ebene gleichzeitig. Verändert werden können Farbgruppen in den unterschiedlichen Design-Files.
- **Überschriften:** Überschriften haben wir mit den html-Befehlen <h1/>, <h2/>, etc. gekennzeichnet und dann allgemein unter 'kreuzen-frontend/src/components/General/GlobalStyle.tsx' die Eigenschaften bearbeitet. Dies ist die effizienteste und unkomplizierteste Lösung.
- **Bilder einfügen:** Bilder können mithilfe des Befehls

```
import bild_name from '../quelle';
```

  importiert werden. Danach sieht die Einbindung des Bildes an der richtigen Stelle wie folgt aus:

```
<div style={{ width: 'auto', height: 'auto' }}>
       <img src={bild_name} alt="alternativer_Text" width="breite" height="höhe" />
</div>
```

1.4 Design bestimmter Komponenten

- **Allgemeine Modifikation:** Einzelne Komponenten können global mit 'styled-components' geändert werden. (Siehe https://styled-components.com)
- **Modifikation von html-Elementen:** Viele html-Komponenten haben unterschiedliche Möglichkeiten zur Veränderung des Designs. Diese können leicht durch css-Files bearbeitet werden. Gut dafür eignen sich 'GlobalStyle.tsx' (src/components/General/GlobalStyle.tsx) und 'custom.scss' (src/scss/custom.scss).
  Dort sind auch schon Beispiele implementiert, die ebenfalls für die weitere Entwicklung genutzt werden können.
- **Den Header anpassen** Der Header wurde mithilfe der Komponente 'NavBar' von React-Bootstrap erstellt. Anpassung können in 'src/components/General/Header.tsx' vorgenommen werden.
  Dabei sollte beachtet werden, dass die gewünschten Header-Elemente nur den richtigen Personen angezeigt werden (Normale User, Moderatoren, Admins, Superadmins)

Teil 3: Entwicklung im Backend

1.1 Globale Dateien

1.1.1 application.properties

Unter src/main/resources/ befindet sich die application.properties, die grundlegende Informationen wie die URL des Postgres-Servers oder die BaseURL von Kreuzen speichert. Außerdem werden hier mit dem Parameter app.locale die Spracheinstellungen festgelegt.

1.1.2 schema.sql

In /schema.sql befindet sich das aktuelle Datenbank-Schema. Änderungen an der Datenbank sollten also unverzüglich in der schema.sql eingetragen werden. Die Datei ist nicht für die Funktionalität verantwortlich, kann aber als Backup genutzt werden, falls die Datenbank neu aufgesetzt werden muss.

1.1.3 Fehlermeldungen

Im Verzeichnis src/main/resources/messages werden die Fehlermeldungen des Backends gespeichert. de_DE.properties enthält die Fehlermeldungen in deutscher Sprache. Zukünftig könnten hier also für andere Sprachen neue Files angelegt werden, bspw. en_GB.properties für Fehlermeldungen in britischem Englisch. Jede Fehlermeldung wird mit Schlüssel der Fehlermeldung=Text der Fehlermeldung abgespeichert. Da die Schlüssel in den jeweiligen Controller- und Service-Klassen verwendet werden, müssen die Schlüssel in allen Sprachdateien gleich lauten und nur die Texte der Fehlermeldungen angepasst werden.

1.1.4 Tests

Im Verzeichnis src/test/java/de/kreuzen/ findet sich je Package eine Tests-Datei, die den jeweiligen Controller mit JUnit-Tests auf dessen Funktionalität testet. Zur Aufrechterhaltung der Qualitätssicherung sollte eine line coverage von 80 % im jeweiligen Package angestrebt werden.

1.2 Grundlegender Aufbau von Paketen

Der Aufbau der Pakete folgt einem festen Schema, der nachfolgend am fiktiven Package object gezeigt wird. object kann beispielsweise durch course, module, university.. ersetzt werden. Der Aufbau kann bei Bedarf (wird später an den entsprechenden Beispielen gezeigt) erweitert werden.

```
object
├── requests
│   ├── CreateObjectRequest.java
│   └── UpdateObjectRequest.java
├── responses
│   └── ObjectResponse.java
├── Object.java
├── ObjectController.java
├── ObjectRepo.java
├── ObjectService.java
└── ObjectServiceImpl.java
```

1.2.1 CreateObjectRequest und UpdateObjectRequest

Der CreateObjectRequest wird dem Controller übergeben und enthält die Informationen, die das zu speichernde Objekt beschreiben. Analog wird der UpdateObjectRequest genutzt, um nachträglich die Eigenschaften eines Objekts zu verändern. Die Annotationen

@Data, @AllArgsConstructor und @NoArgsConstructor werden genutzt, um Setter, Getter und Konstruktoren automatisch generieren zu lassen.

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
```

Des Weiteren werden JavaX Validation Constraints verwendet, um bereits an dieser Stelle Requests herauszufiltern, die nicht den Anforderungen entsprechen. Die verwendete Fehlermeldung muss in der angesprochenen de_DE.properties hinterlegt sein.

```
import javax.validation.constraints.NotNull;
@NotNull(message = "CreateCourseRequest-semesterId-not-null")
private Integer semesterId;
```

Für den UpdateObjectRequest gelten die Ausführungen analog.

1.2.2 ObjectResponse

Die ObjectResponse gibt ein in der Datenbank gespeichertes Objekt zurück. Der Controller gibt bei entsprechenden Anfragen nicht das Objekt selbst, sondern die Response zurück, weshalb dafür ein Konstruktor definiert werden muss:

```
package de.kreuzenonline.kreuzen.course.responses;

import de.kreuzenonline.kreuzen.course.Course;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class CourseResponse {

    private Integer id;
    private Integer moduleId;
    private Integer semesterId;
    private String name;

    public CourseResponse(Course course) {
        this.id = course.getId();
        this.moduleId = course.getModuleId();
        this.semesterId = course.getSemesterId();
        this.name = course.getName();
    }
}
```

1.2.3 Object

Die Object-Klasse enthält das fertige Objekt. Die Parameter müssen mit dem Aufbau der dazugehörigen Datenbank-Table übereinstimmen. Zu beachten ist hierbei, dass ein Unterstrich in der Spaltenbenennung des Datenbank-Tables in Spring zu einem Großbuchstaben führt. Heißt also die Spalte in der Datenbank-Table is_active, muss der Boolean in der Objekt-Klasse isActive heißen.

Zusätzlich zu @Data, @AllArgsConstructor und @NoArgsConstructor muss die Annotation @Table("Name der Table") über der Klassendefinition sowie innerhalb der Klasse die Annotation @Id hinzugefügt werden.

1.2.4 ObjectController

Der Controller ist die Klasse, die Anfragen entgegennimmt und verarbeitet. Darüber hinaus wird hier die Authorisierung überprüft und die Validierung eingehender Requests angestoßen. Wie beschrieben, gibt der Controller keine Objekte, sondern ausschließlich Response-Klassen zurück. Der Beginn der Controller-Klasse sollte so aussehen:

```
@RestController
@Api(tags = "Object")
public class ObjectController {

    private final ObjectService objectService;
    private final ResourceBundle resourceBundle;

    public Object Controller(ObjectService ObjectService, ResourceBundle resourceBundle) {
        this.objectService = objectService;
        this.resourceBundle = resourceBundle;
}
```

Die einzelnen Methoden im Controller bestehen aus folgenden Elementen:

- Angabe der Anfrage-Operation sowie der dazugehörigen URL, bspw. @PostMapping(/object"), um ein neues Objekt zu speichern.
- Beschreibung der Funktion, wird für die Anzeige in Swagger benötigt, mithilfe von @ApiOperation
- Methodenkopf mit annotierten Parametern, bspw. @PathVariable für IDs, die direkt aus der URL abgeleitet werden oder @Valid @RequestBody, um bei einem übergebenen Request die Validierung anzustoßen, bevor der Request weiterverarbeitet wird. Außerdem ist @AuthenticationPrincipal CustomUserDetails userDetails zu übergeben, damit die Authorisierung überprüft werden kann.

Authorisierung: - Immer zu nutzen, damit nur eingeloggte Benutzer Zugriff bekommen können:

```
if (userDetails == null) {
        throw new ForbiddenException(resourceBundle.getString("unauthorized"));
```

Optional. um den Zugriff auf eine Funktion auf bestimmte Personengruppen (hier: ADMIN, MOD und SUDO) zu beschränken:

```
boolean isAdmin = userDetails.getAuthorities().contains(new SimpleGrantedAuthority(Roles.ADMIN.getI
      || userDetails.getAuthorities().contains(new SimpleGrantedAuthority(Roles.MODERATOR.getId()))
      || userDetails.getAuthorities().contains(new SimpleGrantedAuthority(Roles.SUDO.getId()));
// Only Admins, Mods and Sudo can create a course.
if (!isAdmin) {
    throw new ForbiddenException(resourceBundle.getString("create-course-forbidden"));
}
```

1.2.5 ObjectRepo

Die Repo-Klasse sieht grundsätzlich wie folgt aus, durch die Annotation @Repository werden viele grundlegende Funktionen wie das Speichern und Löschen von Objekten aus der Datenbank (in Abhängigkeit von der Id) automatisch implementiert.

```
@Repository
public interface ObjectRepo extends CrudRepository<Object, Integer> {

}
```

Soll ein Objekt in der Datenbank über einen anderen Parameter aufgerufen werden, kann Spring dies ebenfalls automatisch verarbeiten, bspw. können über findAllByModuleId(Integer moduleId) alle Kurse aufgerufen werden, bei denen die übergebene moduleId in der Spalte module_id steht. Für spezielle Anfragen können mit der @Query-Annotation vor dem Funktionsaufruf benutzerdefinierte Querys genutzt werden.

1.2.6 ObjectService

Der ObjectService wird zunächst als Interface definiert und enthält hier auch die Kommentare - Funktion und sämtliche Parameter sowie der Rückgabewert sollten hier beschrieben werden.

1.2.7 ObjectServiceImpl

Die ObjectServiceImpl-Klasse implementiert das Service-Interface und wird als @Service-Klasse definiert. Hier wird die tatsächliche Funktionalität sowie die Zugriffe auf die jeweiligen Repos definiert.

1.3 Das Question-Package

Einen Sonderfall stellt das Question-Package dar, da hier mit Code Generation gearbeitet wird. Jede Frage besteht aus einer Base- und einer Type-Question. Dabei enthält die Base-Question allgemeine Informationen wie die Id, die mögliche Punktzahl und ob die Frage bereits freigegeben wurde. In der Type-Question befinden sich die Fragentypspezifischen Informationen. Eine single-choice-Frage benötigt mehrere Antwortmöglichkeiten, von denen genau eine als korrekt markiert werden muss. Eine multiple-choice-Frage hat ebenfalls mehrere Antwortmöglichkeiten, von denen aber in der Regel mehr als eine als korrekt markiert werden soll.

1.3.1 ./origin

Das Sub-Package Origin wird benötigt, um unterschiedliche Fragenursprünge als Information einer Base-Question hinzuzufügen. Zunächst besteht ein QuestionOrigin lediglich aus deren Kurz-Schreibweise sowie dem angezeigten Langnamen, bspw. „ORIG“ = „Original-Frage“ oder "GEPR- "Gedächtnisprotokoll-Frage". Ziel ist es also, dass nur die festgelegten Kategorien bei der Anlage und Bearbeitung von Fragen genutzt werden können. Die weitere Funktionalität, durch die später Administratoren diese Kategorien bearbeiten können, ist bislang nicht implementiert.

1.3.2 ./requests und ./responses

Auf oberster Ebene sind Requests und Responses alleinig auf BaseQuestions, also allgemeine Informationen von Fragen, angelegt. Im Gegensatz zu anderen Packages kann die Validierung nicht im BaseQuestion-Controller angestoßen werden und muss somit auch nicht in den Requests festgelegt werden. Die Validierung findet im question-Package abweichend im Service statt.

1.3.3 ./types: Fragentypen

Im types-Subpackage befindet sich die Funktionalität für die einzelnen Fragetypen. Jedes dieser Sub-Packages ist nach der im gesamten Projekt genutzten Ordnerstruktur aufgebaut, allerdings mit einigen Eigenheiten. Bspw. gibt es keine Fragentyp-spezifischen Controller-Klassen, sondern nur eine einzige Controller-Klasse, die sämtliche Fragentypen verarbeitet. Des Weiteren müssen beim Paketaufbau Fragentyp-spezifische Besonderheiten bedacht werden.

Beispielsweise sieht die Struktur für single-choice-Fragen wie folgt aus:

```
.question/types/singleChoice
├─ requests
│  ├─ CreateSingleChoiceRequest.java
│  └─ UpdateSingleChoiceRequest.java
├─ responses
│  ├─ SingleChoiceAnswerResponse.java
│  └─ SingleChoiceQuestionResponse.java
├─ SingleChoiceAnswer.java
├─ SingleChoiceAnswerRepo.java
├─ SingleChoiceQuestion.java
├─ SingleChoiceQuestionEntry.java
├─ SingleChoiceQuestionRepo.java
├─ SingleChoiceService.java
└─ SingleChoiceServiceImpl.java
```

Es muss also zwischen Frage und Antwortmöglichkeiten differenziert werden. Des Weiteren muss die Unterscheidung zwischen tatsächlicher Frage (SingleChoiceQuestion) und dem Eintrag in der Datenbank (SingleChoiceQuestionEntry) beachtet werden. Die SingleChoiceQuestion erweitert die BaseQuestion und enthält somit alle Informationen, die zu der Frage gehören. Der SingleChoiceQuestionEntry hingegen enthält nur die Informationen, die über die BaseQuestion hinaus gehen. Diese Zusatz-Informationen werden in der Datenbank in einer eigenen Tabelle für Single-Choice-Fragen gespeichert.

1.3.4 ./types: Abstrakte Klassen

Die Klasse QuestionTypeMapperService.java wird vom QuestionController angesprochen und startet dann den jeweiligen Fragentyp-Service. Damit diese Funktion korrekt arbeitet, muss in der jeweiligen QuestionTypeServiceImpl-Klasse der Typ definiert werden:

```
public static final String TYPE = "multiple-choice";
```

QuestionTypeService.java enthält die abstrakten Funktionen, die von den jeweiligen Services implementiert werden. Grundlegende Funktionalitäten wie getById, create und update müssen für jeden Fragentyp vorhanden sein, aber mit jeweils passenden Requests arbeiten.

1.3.5 BaseQuestionServiceImpl.java

Wie im vorherigen Kapitel beschrieben, findet die Validierung eingehender Requests in diesem Package abweichend im Service statt. Anstelle der Annotationen wird mit einfachen If-Statements überprüft, ob inhaltliche Voraussetzungen durch einen Request verletzt werden würden:

```
if (type == null) {
  throw new ConflictException(resourceBundle.getString("CreateQuestionRequest-type-not-null"));
}
```

1.3.6 QuestionController.java

Im question-Package existiert ausschließlich ein QuestionController für die Kommunikation mit dem Frontend und nicht bspw. für jeden Fragentyp ein eigener Controller. Die Funktionen zur Erstellung und Bearbeitung von Fragen bekommen also keinen CreateTypeQuestionRequest, sondern einen allgemeinen Http-Request. Aus dem im Request enthaltenen Fragetyp erkennt der Controller, welcher Service für die weitere Verarbeitung des jeweiligen Fragetyps angesprochen werden muss.

Beim Löschen einer Frage wird die BaseQuestion angesprochen und gelöscht. Die Kaskadierung und damit das Löschen der dazugehörigen Typen-Frage übernimmt die Datenbank automatisch.

Soll ein neuer Fragentyp in das System eingefügt werden, muss zusätzlich zur tatsächlichen Fragetyp-Implementierung folgender Schritt abgeschlossen werden: In question/types/QuestionTypeMapperServiceImpl.java muss der neue Fragentyp per @Autowired hinzugefügt und auch der getServiceByType-Funktion analog zu den bereits bestehenden Typen hinzugefügt werden.

1.4 Das Session-Package

Einen weiteren Sonderfall stellen die Sessions dar. Eine Session ist eine Auswahl an Fragen, die vom Nutzer zusammengestellt und zu Lernzwecken beantwortet werden kann, d.h. eine Session besitzt einen "Head", der grundlegende Informationen wie einen Session-Namen oder einen Wahrheitswert, ob die Fragen in zufälliger Reihenfolge angezeigt werden sollen oder nicht, speichert. Darüber hinaus beinhaltet die Session wie beschrieben eine Auswahl an Fragen, die jeden unterschiedlichen Fragentyp annehmen dürfen. Die Besonderheit der verschiedenen Fragentypen kommt also auch hier zum Tragen.

1.4.1 ./requests

Neben den Requests zum Erstellen und Bearbeiten des Session-Heads gibt es folgende weitere Requests:

- SetSelectionRequest: Dieser Request beinhaltet Informationen darüber, wie ein Nutzer eine Frage beantwortet hat. Aktuell wird ein SetSelectionRequest für alle Fragentypen genutzt, wobei der Fragentyp im Parameter type angegeben werden muss. Des Weiteren beinhaltet der Request die Möglichkeit, angekreuzte (checked) und durchgestrichene (crossed) Antworten zu markieren. Für die durchgestrichenen Antworten können beide Fragentypen das Array crossedLocalAnswerIds nutzen. Bei single-choice-Fragen soll der Integer checkedLocalAnswerId für die eine als richtig markierte Antwort genutzt werden, bei multiple-choice-Fragen das entsprechende Array. Zukünftig sollte hier analog zu den Fragen mit abstrakten Klassen und fragentyp-spezifischen Requests gearbeitet werden, um dann auch die Einhaltung der Restriktionen besser prüfen zu können.
- SetTimeRequest: Solange eine Frage innerhalb einer Session beantwortet wird, wird jede Sekunde durch einen PUT-Request die kumulierte Zeit an das Backend gesendet und in der Datenbank gespeichert. Dadurch wird die Antwortzeit der Frage getrackt. Sobald die Frage beantwortet wurde und auf submitted gesetzt wurde, wird der PUT-Request für die time nicht mehr durchgeführt.

1.4.2 ./responses

- SessionResponse: gibt den "Head"der Session zurück
- SessionQuestionResponse: gibt die Informationen über den Fragen-"Headëiner einzelnen Frage innerhalb der Session zurück (bspw. Antwortzeit und Punktzahl)
- <QuestionType>SelectionResponse: gibt die Antwortenauswahl für eine einzelne Frage innerhalb einer Session zurück, benötigt wird eine Response je Fragentyp
- QuestionResultResponse: gibt das Ergebnis einer einzelnen Frage innerhalb einer Session mit der dazugehörigen Frage zurück
- QuestionCountResponse: gibt die Anzahl an Fragen zurück, die einer bestimmten Session zugeordnet sind.

1.4.3 ./selections

Im Subpackage selections besteht für jeden Fragentyp eine gesonderte Selection-Klasse, da diese auch jeweils in einer eigenen Datenbank-Table gespeichert werden. Die Selection wird für jede Antwort einer Frage innerhalb einer Session gespeichert und beinhaltet Informationen darüber, ob die jeweilige Antwort angekreuzt oder durchgestrichen wurde. Um hierfür mit der Datenbank zu interagiern, gibt es dementsprechend auch eigene <QuestionType>Repo-Klassen.

1.4.4 SessionController und SessionService

Es gibt im SessionController mehrere Funktionen, die aktuell hard-coded zwischen den unterschiedlichen Fragentypen unterscheiden. Um diese Funktionen ausführen zu können, wurden bislang auch Fragentyp-spezifische Funktionen im Service definiert. In Zukunft sollte hier eine generische Lösung, analog zum QuestionController angestrebt werden. Die betroffenen Funktionen im SessionController sind:

- getSelection
- getResult
- addSelection

1.5 Backend-only-Funktionen

Folgende Funktionalitäten sind im Backend bereits vorhanden, konnten im Frontend aber nicht innerhalb des Projekts implementiert werden:

- Zuordnungsfragen: finden sich in question/types/assignmentQuestion
- Bearbeiten und Bestätigen von Fragen durch Administratoren/Moderatoren, ist im QuestionController enthalten
- Beanstandung von Fragen (Nutzer können bei jeder gespeicherten Frage Fehler melden und neben einer Fehlerbeschreibung auch eine dazugehörige Quelle angeben. Das entsprechende Package ist /error. Wenn ein Fehler gemeldet wird, wird dazu im Kommentar-Bereich der Frage eine automatisch generierte Nachricht angezeigt. Wenn die Beanstandung behoben / abgelehnt wurde, wird dem Nutzer außerdem eine automatische E-Mail geschickt.
- Bearbeiten von Sessions, bspw. Änderungen der Session-Bezeichnungen sind bislang nur im Backend implementiert
- Tipp des Tages: befindet sich im Package /hint und beinhaltet Tipps, von denen jeweils ein zufälliger auf der Startseite angezeigt wird. Die Funktionalität zum Anlegen und Bearbeiten der Tipps ist bislang nur im Backend vorhanden.

1.6 Migration

Zu der Migration der DEFI Daten wurde ein Migrationsskript in Python entwickelt, das sich mit der alten MySQL Datenbank, sowie der neuen PostgreSQL verbindet und dann die Daten überträgt.
Das Schema sowie die Queries im "Migration.sql"File müssen vor ausführung des Pythonskriptes ausgeführt worden sein. ACHTUNG: Das Skript löscht alle Daten aus der Postgres Datenbank!
In dem Skript kann zur Entlastung der Datenbank eine PageSize gesetzt werden, typischerweiße ist ein Wert von 100 angebracht.

Momentan fehlt dem Skript noch, dass es Bilder, zu denen nur die URL existiert runterlädt und dann in die DB schreibt. Dies kann in der Funktion "process_file_dataëingebaut werden.

TODO: swagger