TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Karen Ofljan 164912 IAPB

WEB APPLICATION FOR MANAGING EXERCISES AND TEST FILES FOR THEM

Bachelor's thesis

Supervisor: Ago Luberg
MSc

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Karen Ofljan 164912 IAPB

VEEBIRAKENDUS HARJUTUSTE JA TESTIFAILIDE HALDAMISEKS

Bakalaurusetöö

Juhendaja: Ago Luberg
MSc

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Karen Ofljan

12.08.2019

Abstract

The goal of the thesis is to provide an overview of the process of designing and deploying a web application, which purpose is to ease management of programming exercises in the future. This thesis is written in English and is 50 pages long, including 5 chapters, 17 figures and 2 tables.

In the first chapter, general goals and work plan are introduced. Second chapter introduces use cases of the application to the reader, including comprehensive flow charts. Third chapter provides explanations for using certain tools in the development process. Fourth chapter describes the development process in high detail, acquainting the reader with various steps and decisions during the development process. Fifth and final chapter provides an overview of the results and sums up the thesis, pointing out, what was achieved and what was not.

Annotatsioon

Antud töö sihiks on anda ülevaadet veebirakenduse disainimisest ja paigaldamisest. Rakenduse eesmärk on lihtsustada programmeerimisülesannete ja testifailide haldamist. Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 50 leheküljel, 5 peatükki, 17 joonist, 2 tabelit.

Esimene peatükk tutvustab lugejat antud töö eesmärkide ja esialgse arenduse plaaniga. Teine peatükk toob välja rakenduse kasutajalugusid, mis on illustreeritud nn *voodiagrammide* abil. Kolmas peatükk põhjendab tööristade valikut, mida kasutati rakenduse arenduses. Neljas peatükk detailselt kirjeldab arenduse protsessi, tutvustades lugejat erinevate sammude ja ja otsustega, mis olid tehtud arendusprotsessi jooksul. Viies ja viimane peatükk annab ülevaade tulemustest ja selgitab, mis oli tehtud ja mis mitte.

List of abbreviations and terms

TUT	Tallinn University of Technology
OOP	Object-Oriented Programming
API	Application Programming Interface
ORM	Object-Related Mapping
DAO	Data Access Object
DTO	Data Transfer Object
UI	User Interface
UX	User Experience
PK	Primary Key
FK	Foreign Key
CRUD	Create, Read, Update, Delete (database operation)
DI	Dependency Injection
CI	Continuous Integration
CD	Continuous Deployment
UML	Unified Modeling Language
SOLID	Single responsibility, Open/closed, Liskov substitution, Interface segregation Dependency inversion

Table of contents

List of figures

List of tables

1 Introduction

Each subject in TUT which is focused on programming usually involves regular practical assignments. Once composed, those practical assignments are being collected during each course and sometimes are reused in the following courses. At this point TUT does not have any centralized system for storing the exercise information, so the process of searching for specific exercise/categorizing them is overly complicated. This paper gives a detailed overview about building a web application which would solve those issues.

The web application must provide functionality for easy manipulation of the exercise data, which includes exercise name, year and statistics (students passed, students failed), test files for the exercise and tags. Test file is either a file with actual tests (*TestComparator.java* etc) or an input/output file for specific exercise. Author defines an input/output file as a file, which contains necessary data for generating automatic tests from it: for example, consider a function named *compare* which takes in an integer and returns "*Even*" if given integer is even number and "*Odd*" if provided integer is odd. Input/output file will contain different inputs, e.g 2, 3, 12, 456 etc, and expected output for each input: 'Even', 'Odd', 'Even'... For an example of such file see Appendix 1.

In addition, some exercises are being reused for courses, where the programming language taught to the students differs from the original language in which the exercise and tests were written. It was proposed to introduce a simple generation mechanism, which, given an input/output file mentioned above will generate an actual test file in a language specified by user. So, for example, from single source file user can potentially generate both Java and Python tests.

To sum up, the goal of this paper is

1) Design and build a web application (database, API and frontend) for managing exercises;
2) Provide functionality of generating actual tests from input/output files.

2 USE CASES AND FUNCTIONALITY

The application's main goal is to simplify the process of managing exercises and related data. The storage for exercise related data should be centralized. By term 'data managing' author means basic CRUD for exercises, tags and test files, as well as possibility to conveniently search for specific exercises among those in the database. In addition, the application should provide functionality for automatic upload of the exercise test files into the database from custom Git repository after corresponding push event.

- *Add test file from Git*

<u>Prerequisites</u>: API must be running on external server and API endpoint must be added to Git repository as a webhook and must be configured to listen for push events. Example for Gitlab with API running on a box with IP of *15.188.3.43*:

Settings => Integrations => URL: 15.188.3.43/Git_callback => mark 'Push Events => unmark 'SSL verification' => Add webhook.

<u>Flow description</u>: Teacher wants to add new test file for exercise. Test file must be in *json* format and file name must match the following pattern: *<any text>**EX**<any text>**_test_data.json***. Teacher commits and pushes the file. API receives a Git callback with commit data. API parses the callback, clones the repo if it was not previously cloned or pulls it in the opposite case, then looks if modified/added section of callback contains strings of required pattern. If so, API reads the files by one, tries to fetch exercise and file by name. If both exercise and file exist in the database, API updates the file contents. If only exercise is present, API creates the file and attaches it to the exercise. If none exist, API creates both. The cycle continues until all files in the commit are parsed. Figure 1 gives an illustration of the flow.

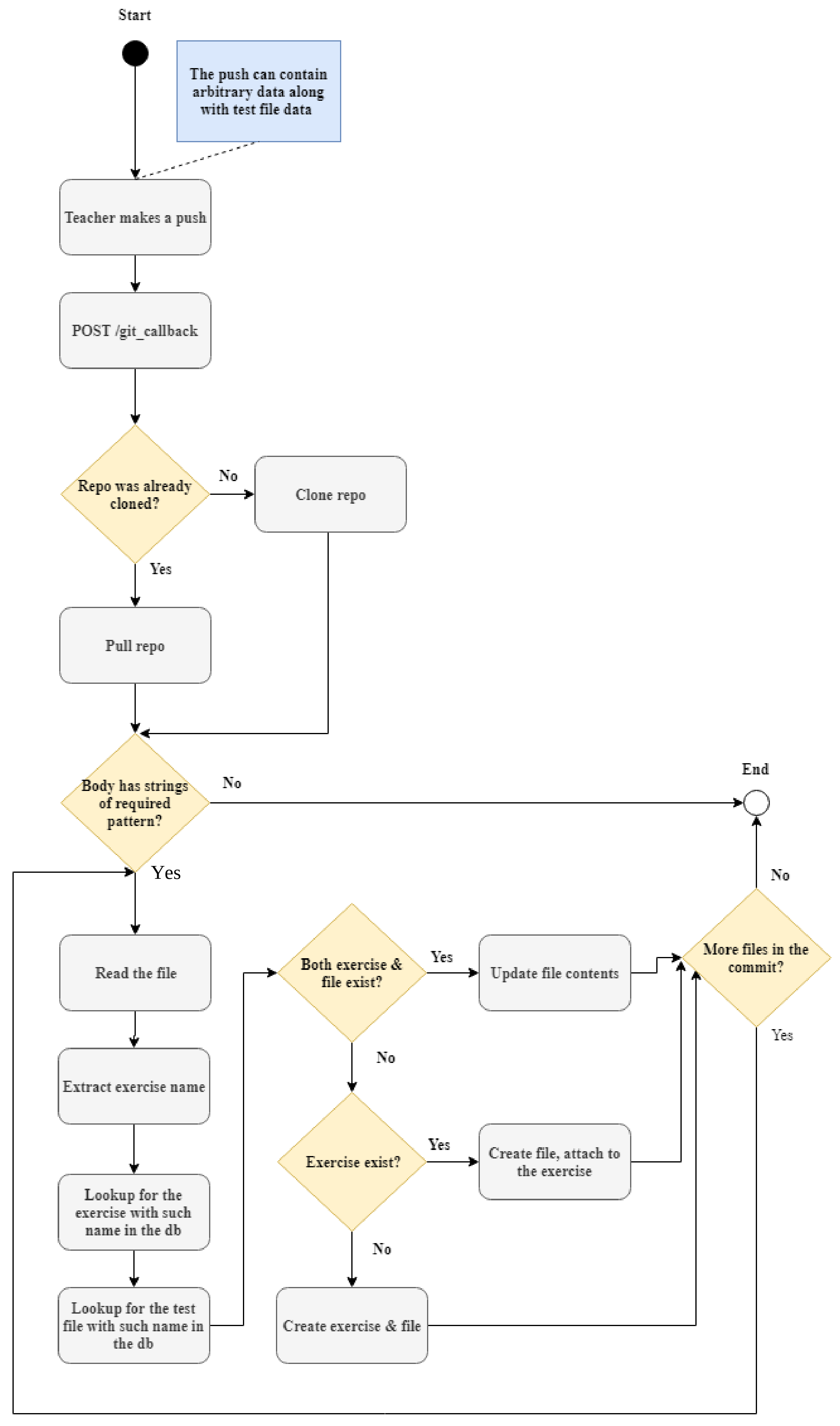

Start
The push can contain arbitrary data along with test file data
Teacher makes a push
POST /git_callback
Repo was already cloned?
No
Clone repo
Yes
Pull repo
Body has strings of required pattern?
No
End
Yes
Read the file
Extract exercise name
Lookup for the exercise with such name in the db
Lookup for the test file with such name in the db
Both exercise & file exist?
Yes
Update file contents
No
Exercise exist?
Yes
Create file, attach to the exercise
No
Create exercise & file
More files in the commit?
No
Yes

Figure 1. Add test file from Git repository flow.

- *Show list of available exercises*

Prerequisites: none

Flow description: Teacher wants to see all exercises which are stored in the database. Example of the flow:

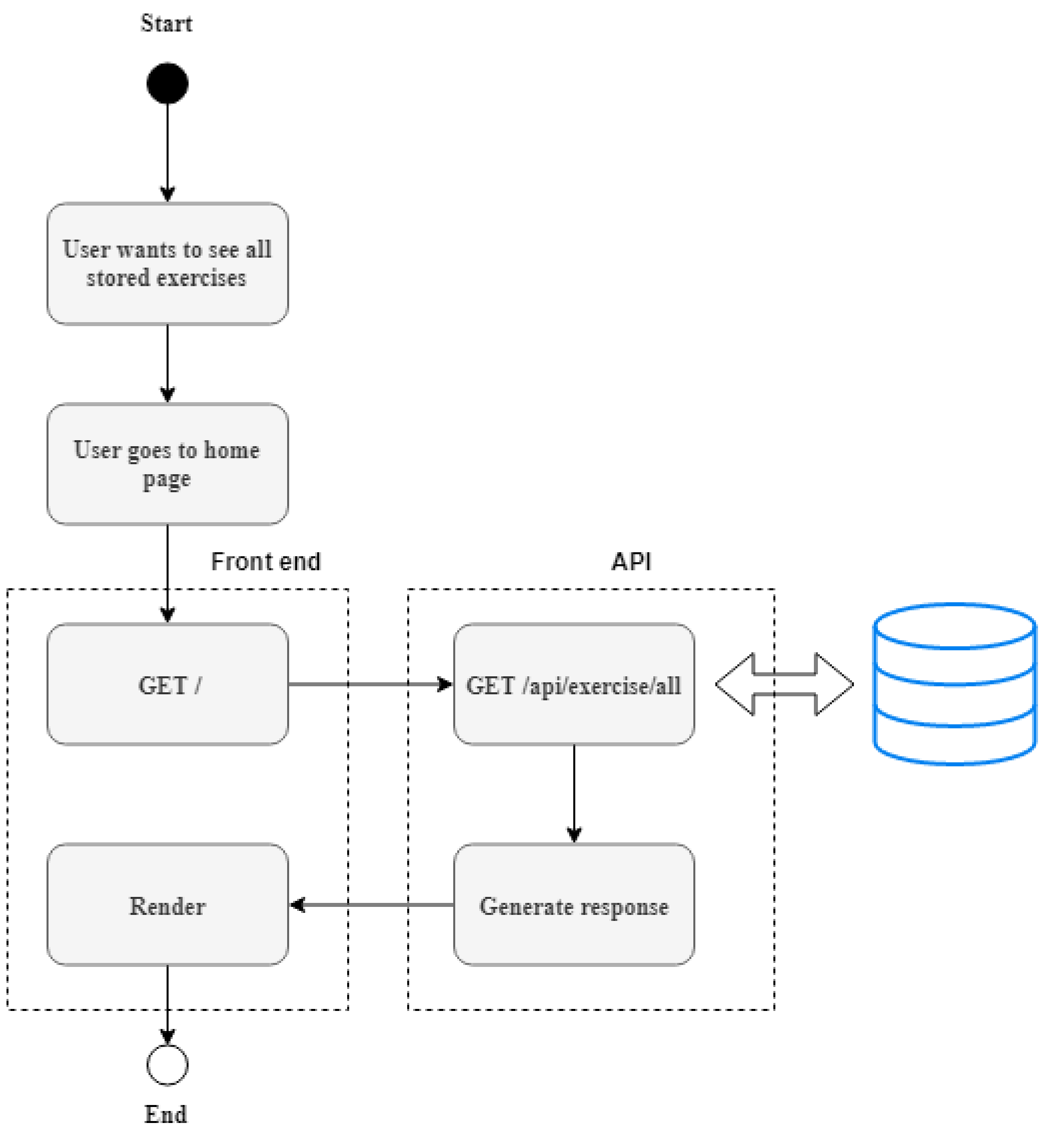


Figure 2. See all exercises flow.

- *Add exercise and related data manually*

Prerequisites: none

Flow description: Teacher wants to add new exercise. For the example of the flow, please refer to figure 3:

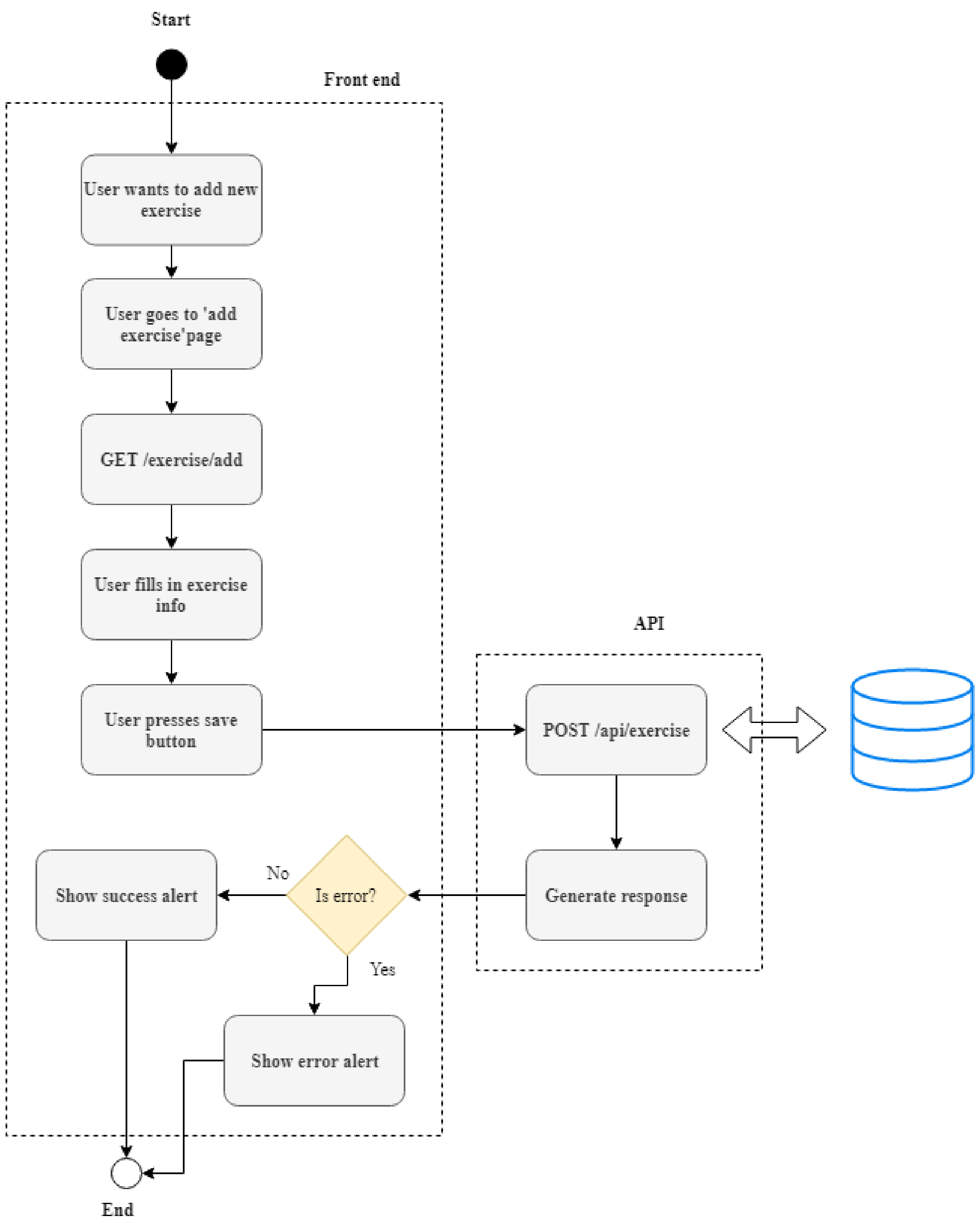


Figure 3. Add new exercise flow.

- *Show list of available tags*

Prerequisites: none

Flow description: Teacher wants to see all tags in the database. The flow is illustrated on figure 4.

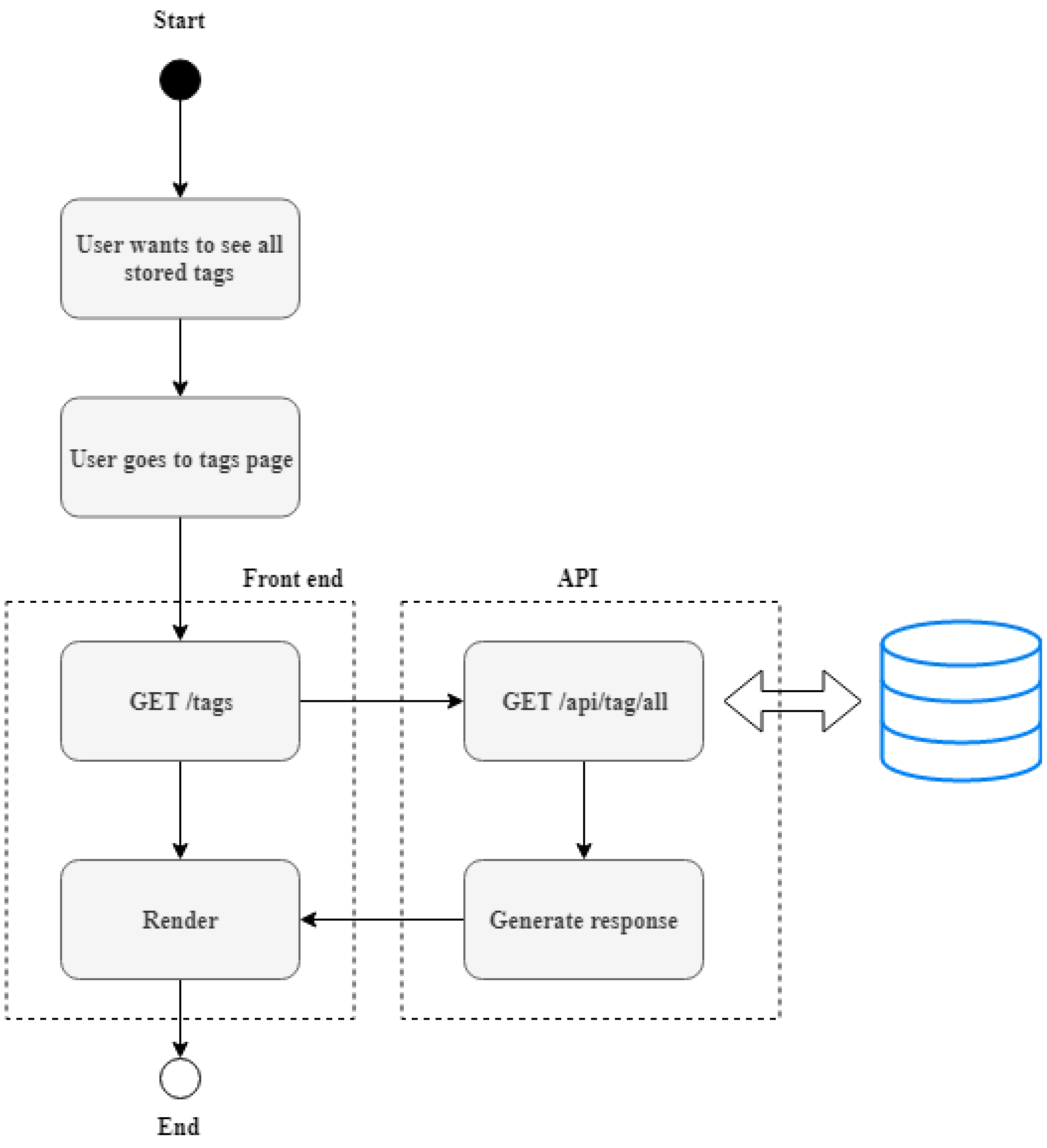


Figure 4. Show all tags flow.

- *Add tag*

Prerequisites: none or existing exercise

Flow description: Teacher wants to add new tag. The flowchart is available below:

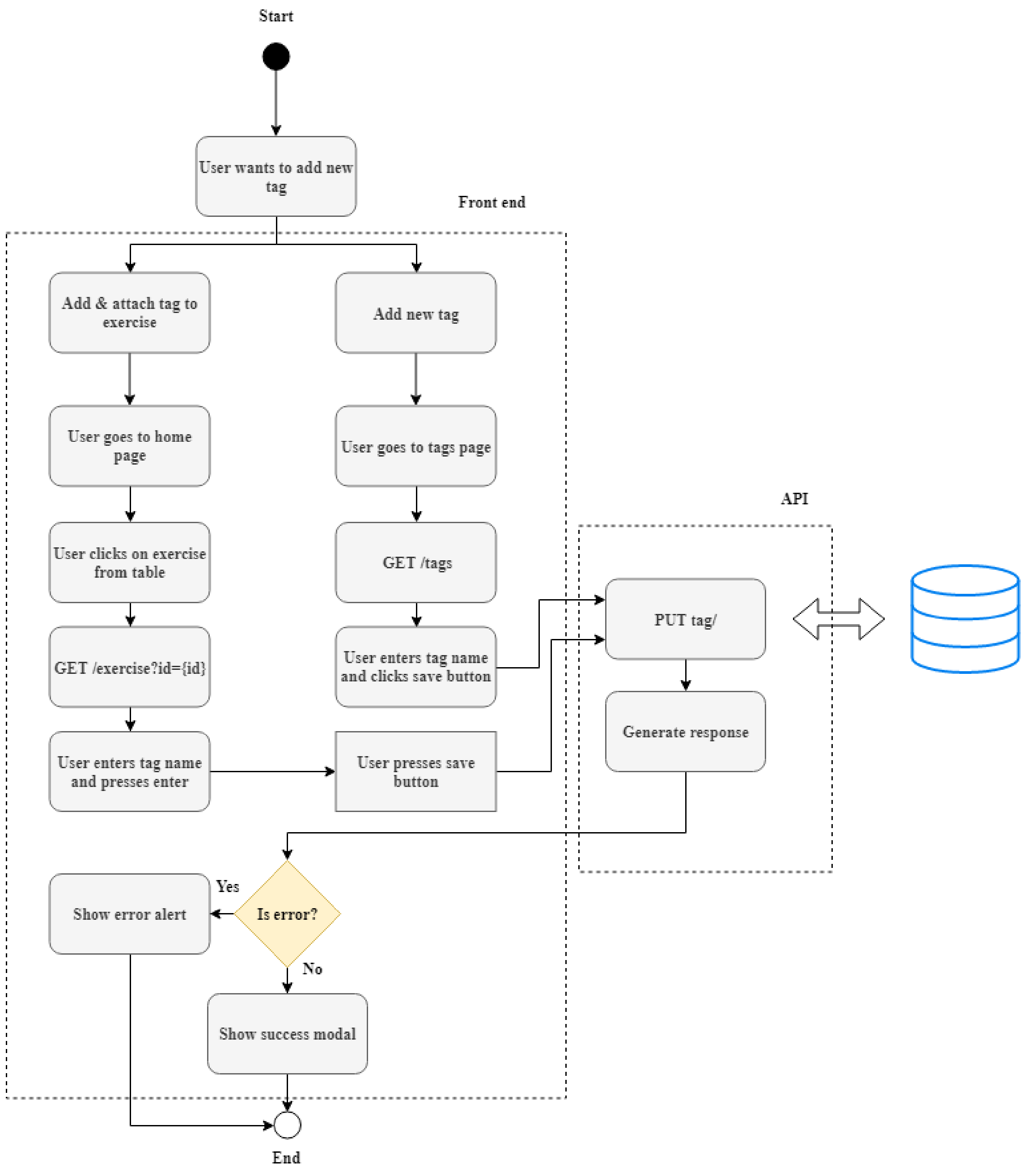


Figure 5. Add new tag flow.

- *Search for exercise*

Prerequisites: exercises exist in the database

Flow description: Teacher wants to find a specific exercise. Search flow illustrating chart is shown on figure 6:

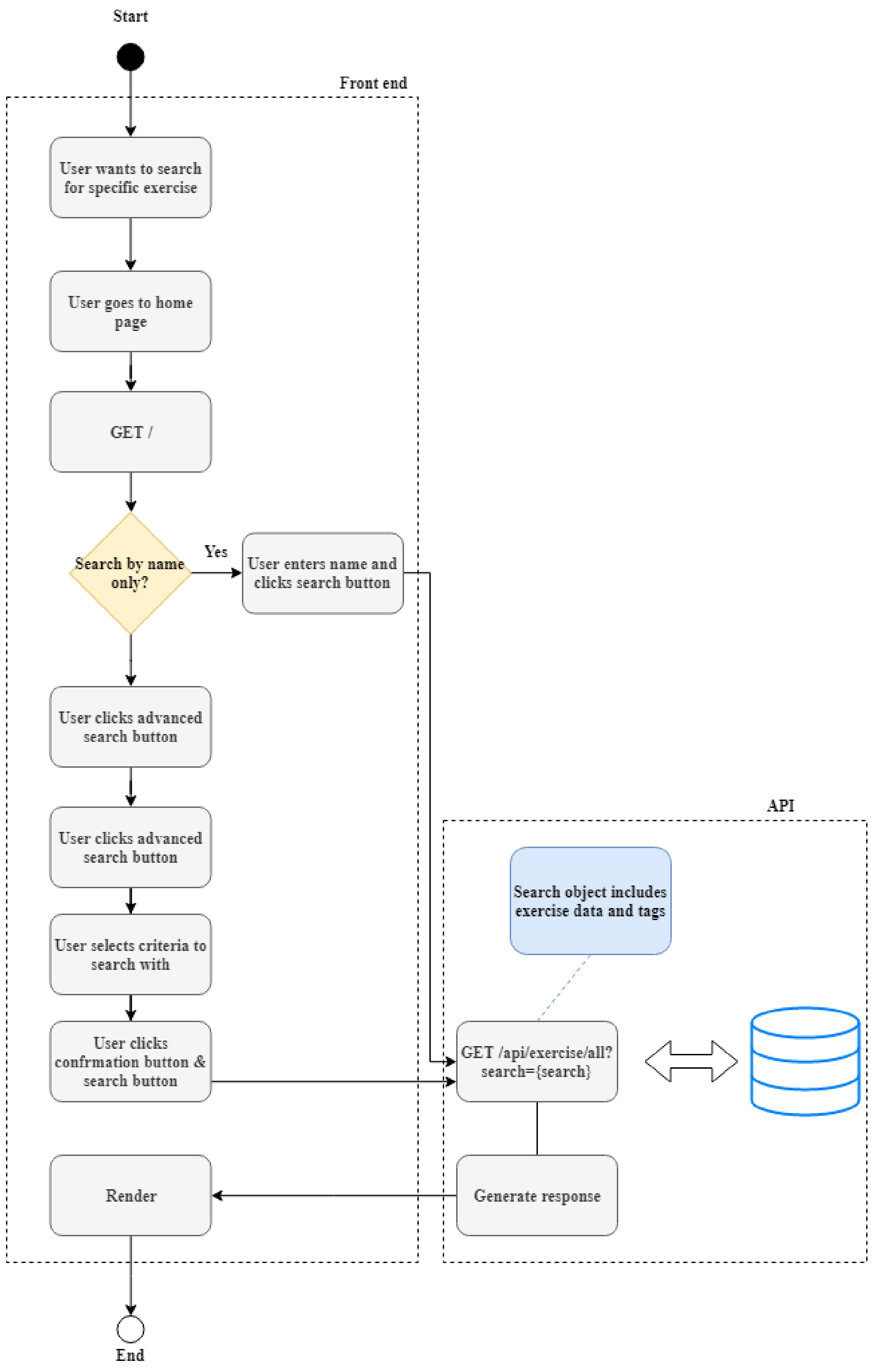

Start
Front end
User wants to search for specific exercise
User goes to home page
GET /
Search by name only?
Yes
User enters name and clicks search button
User clicks advanced search button
User clicks advanced search button
User selects criteria to search with
User clicks confrmation button & search button
Render
End
API
Search object includes exercise data and tags
GET /api/exercise/all?search={search}
Generate response

Figure 6. Search for exercise flow.

- *Edit exercise*

<u>Prerequisites:</u> Exercise exist in the database

<u>Flow description:</u> Teacher wants to edit an exercise (figure 7).

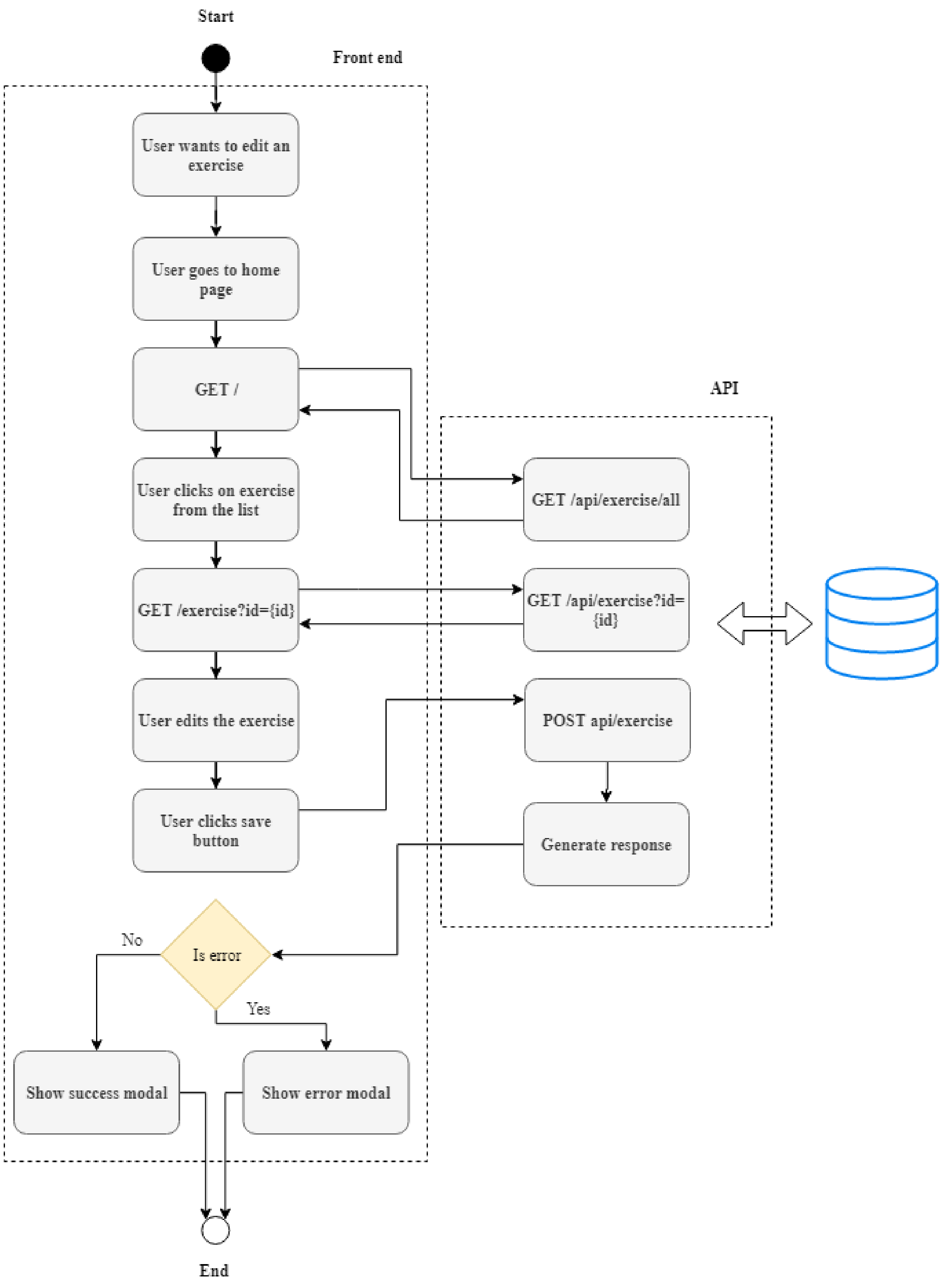


Figure 7. Edit exercise flow.

- *Delete exercise/tag*

Prerequisites: exercise or tag exist in the database

Flow description: Teacher wants to delete exercise or tag (figure 8).

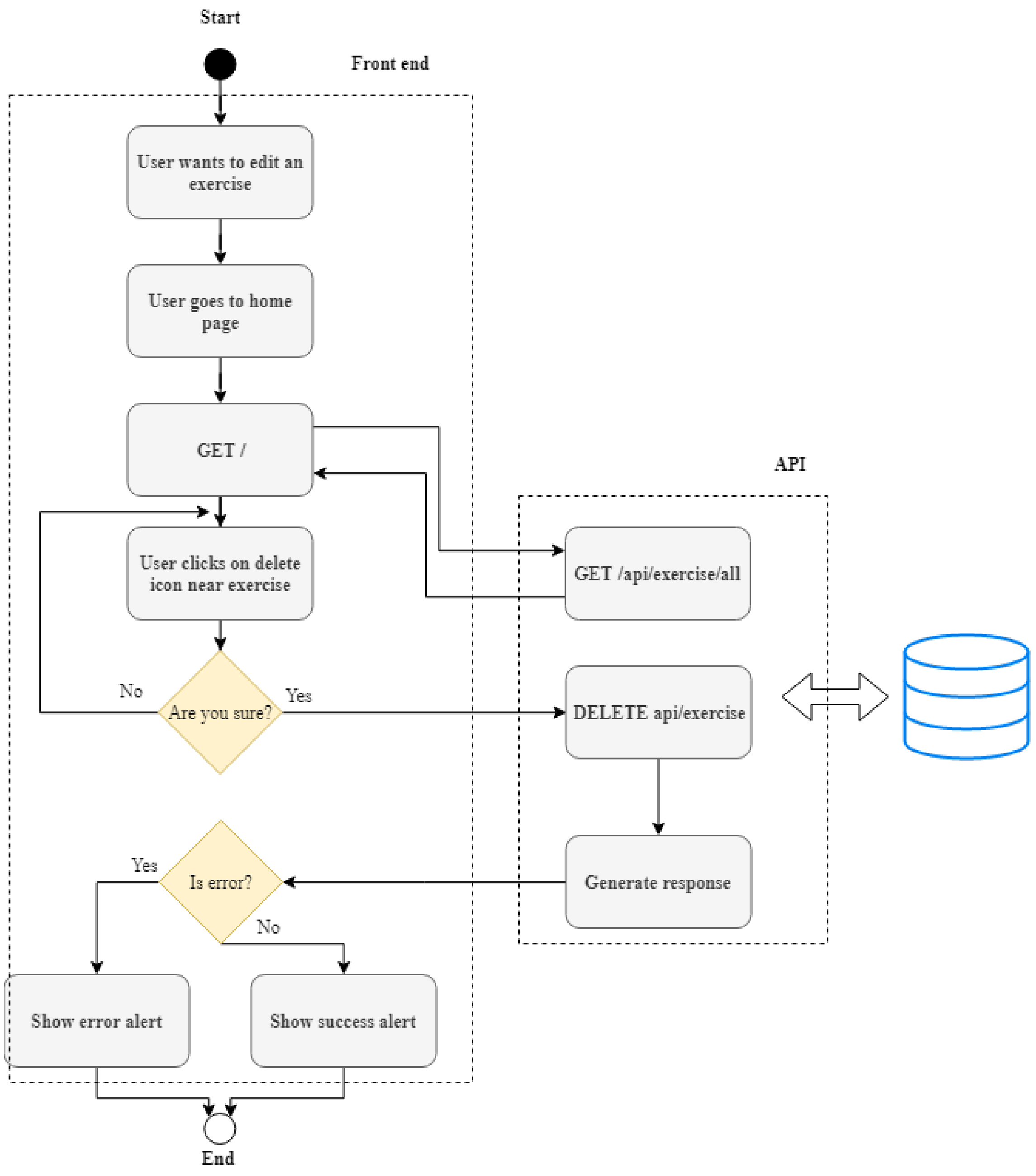


Figure 8. Delete exercise flow.

The same flow goes for tag, only URLs are different. For example, instead of calling *DELETE api/exercise* front end will call *DELETE api/tag*.

- *Detach tag from exercise / edit the tag*

Prerequisites: Exercise exist in the database, user is on the exercise view page (*/exercise?id={id}*).

Flow description: Teacher wants to detach certain tag from the exercise or edit certain tag (figure 9).

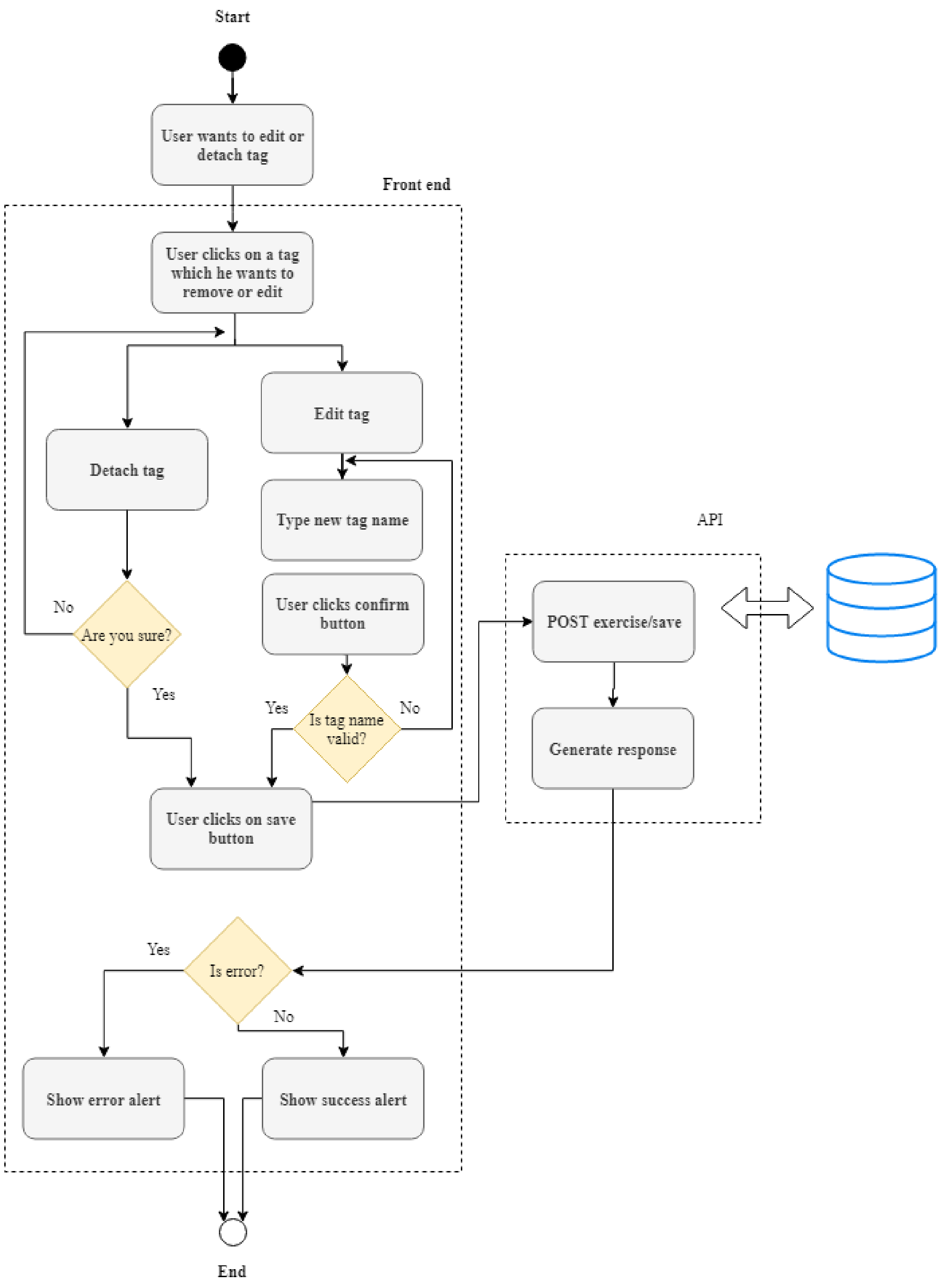


Figure 9. Detach/edit tag flow.

- *Generate test file from input/output file/download file*

Prerequisites: Exercise exist in the database, user is on the exercise view page (*/exercise?id={id}*).

Flow description: Teacher wants to either download test file or to generate test file from I/O file (figure 10).

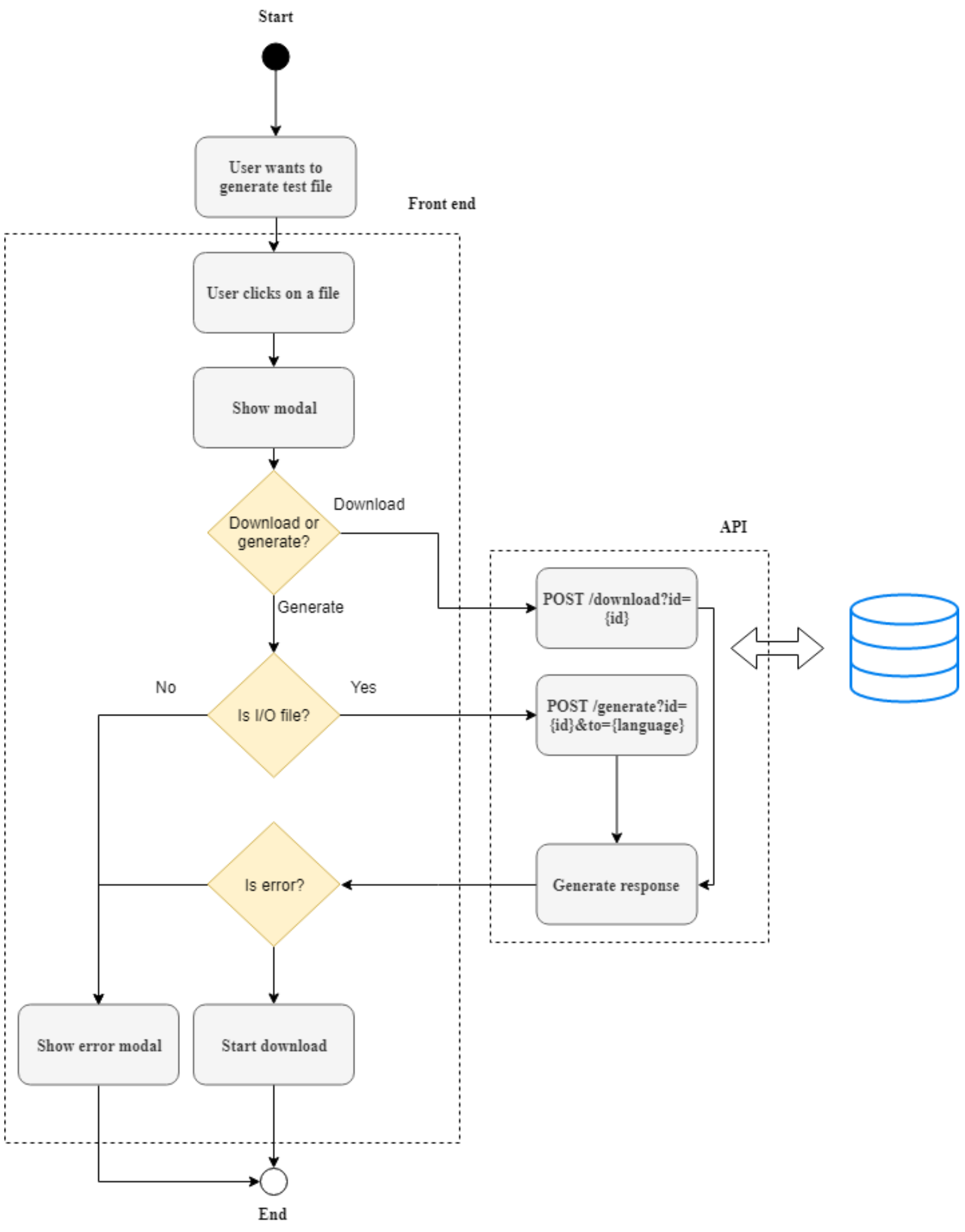


Figure 10. Download/generate file.

3 SELECTION OF THE TOOLS

3.1 Back end

3.1.1 API

Since the project is going to be rather small, it was decided to avoid enterprise-oriented languages and frameworks like Java (Spring) or C# (.NET), and look up for open source solutions. In that case, what is needed is a small API, maybe with some simple ORM framework. API should be written in language which is flexible, easy enough to develop with and at the same time has community support of considerable size. These criteria leave us with two possible options (taking the author's past experience into account) – Python (Django) and JavaScript (Node.js). After some thinking and weighing pros and cons of both approaches, it was decided to move along with Node.js, since it has some advantages over Django:

1) First and most important – TypeScript support, which was created with the goal to provide strong typing for JavaScript, as well as add simplified syntax for advanced OOP concepts (annotations, generics, interfaces, abstract classes...) [1], which is hard to achieve with pure JS;

2) Using one language for a whole application is appealing, since it is easier to develop in that case (no need to worry about serialization compatibilities etc);

3) Python for web is not the best choice, since, based on author's previous experience, the code is hard to structure and debug (no strong typing + weak OOP);

Since using pure Node.js (even with Express) will mean reinventing the wheel in most of the use cases of the application, it was decided to choose a framework with most of the desired features supported. Those features included, but were not limited to:

- Full TypeScript support;
- Possibility of annotation-based endpoints;
- Easy Swagger integration;

- ORM tool(s) support;
- Easy middleware integration (for example, custom logging or exception filtering);
- Possibility of annotation-based validation of incoming payloads;

Considering all the required features, the NestJs seemed like a best choice: it is written entirely in TypeScript, hence has full support of it and perfectly meets all the rest of listed requirements.

3.1.2 Database

The structure of the application implies that at least part of the data fits best into the relational type database for several reasons: entities used will have relations with each other, our application is not expected to process a lot of traffic and data amount which we are going to store is limited. [2] Whereas NoSQL tools provide flexible solutions for certain cases where large volumes of different data need to be stored and processed, this specific application cannot gain any advantage from it. In addition, considering that at the beginning it was decided to put aside enterprise-oriented tools, author had excluded Oracle, DB2 and MsSQL from the list of available options. At this point, the list included MySQL and PostgreSQL, as those are the most popular open source relational database engines. Author’s previous experience with MySQL was rather dramatic; MySQL has shown itself very simple to use, but, on the other hand, the exposed functionality was limited. For instance, MySQL does not allow subsequent updates for table with unique alternative key [3] and, moreover, has some architectural drawbacks. As a result, PostgreSQL was the way to go. As for the server-side GUI for database representation, phpPgAdmin was chosen. The whole list of database engines taken into consideration is represented by table 1:

Table 1. Database engines taken into consideration.

Engine	Open-source	Free to use	Relational
PostgreSQL	Yes	Yes	Yes
MySQL	Yes	Yes	Yes
DB2	No	No	Yes

MsSQL	No	No	Yes
Oracle	No	No	Yes
Elasticsearch	Partly	Yes	No
MongoDB	Yes	Yes	No

3.2 Front end

The most challenging thing to choose was the tool for the front end side of the application. The reason for this is mostly based on author’s previous experience with front end technologies, which is very limited. The initial list of possible options included Angular, Vue, React. These frameworks are the most popular and used at the moment of writing the paper, hence they were selected as potential candidates. [4] [5] [6]

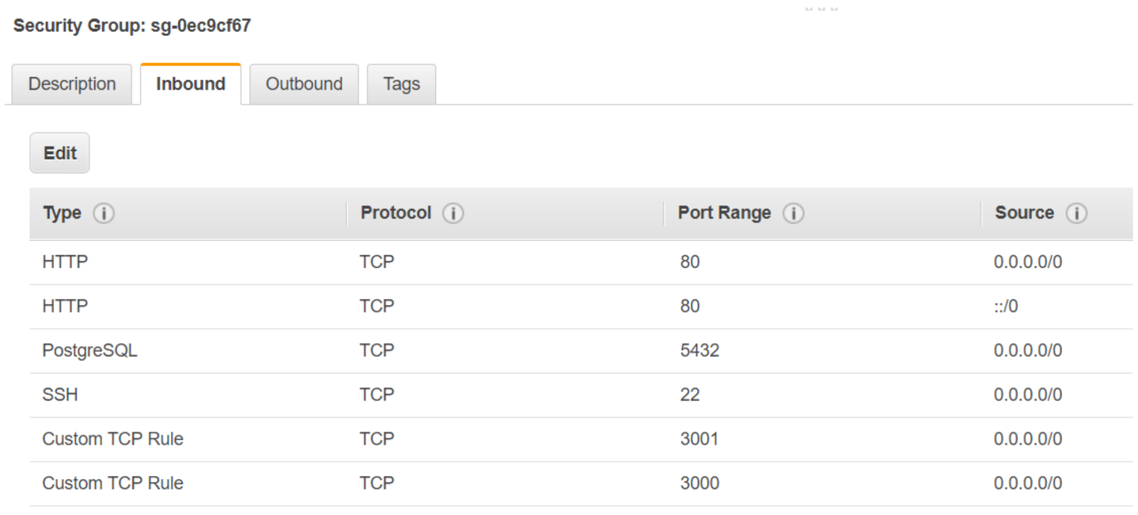
Vue was excluded from the list first, since, although it is considered to be more flexible and easier to use than its alternatives, it is also harder to maintain clean code with it. Vue is also the newest framework from the three and has least of contributors, which could mean potential instability and bugs. [7]

Angular is the oldest from the three and author had some experience with its predecessor, AngularJS, so it was decided to move along with React – to learn something completely new and check how the most popular JS framework [4] [5] [6] feels like. As for styling frameworks, bootstrap and sass were chosen.

3.3 Setting up development environment

Since the application will make use of actual database, it is required to set up a running server with selected database engine and simple GUI installed. For this to happen it was decided to take advantage of Amazon AWS EC2 instances. EC2 stands for Elastic Compute Cloud or virtual computing environments. Simply put, these are just servers which anybody can use for a fixed price. [8]

The choice of operating system is obvious: Ubuntu, since, considering author's previous experience, it is the easiest to use (also, there are not so much options provided for the micro instances). After the decision has been made, it was needed to install required packages and configure Apache to allow connections from public Internet. Happily, most of the options can be configured using Amazon AWS security groups (figure 11).



Type	Protocol	Port Range	Source
HTTP	TCP	80	0.0.0.0/0
HTTP	TCP	80	::/0
PostgreSQL	TCP	5432	0.0.0.0/0
SSH	TCP	22	0.0.0.0/0
Custom TCP Rule	TCP	3001	0.0.0.0/0
Custom TCP Rule	TCP	3000	0.0.0.0/0

Figure 11. Firewall configuration via Amazon AWS security group.

I that case, the configuration allows HTTP connections to the instance at ports 80, 3000 and 3001, as well as SSH connections at port 22 and database connections at port 5432. In that case we do not necessarily need access to ports 3000 and 3001, they were marked accessible in case we will want to deploy and test the application on the instance.

The whole work of setting the database environment is to install Apache and PHP, postgresql, phpPgAdmin, allow connections to the database from remote hosts, set user info for phpPgAdmin and allow remote authentications with username and password. [9] [10]

4 DEVELOPMENT FLOW

4.1 Whole application design

Whole application design is simple – user manipulates the data stored in the external database through API, which accepts requests from front end. User does not interfere with API nor database directly, but only through front end application layer. The diagram of the application's architecture is shown below:

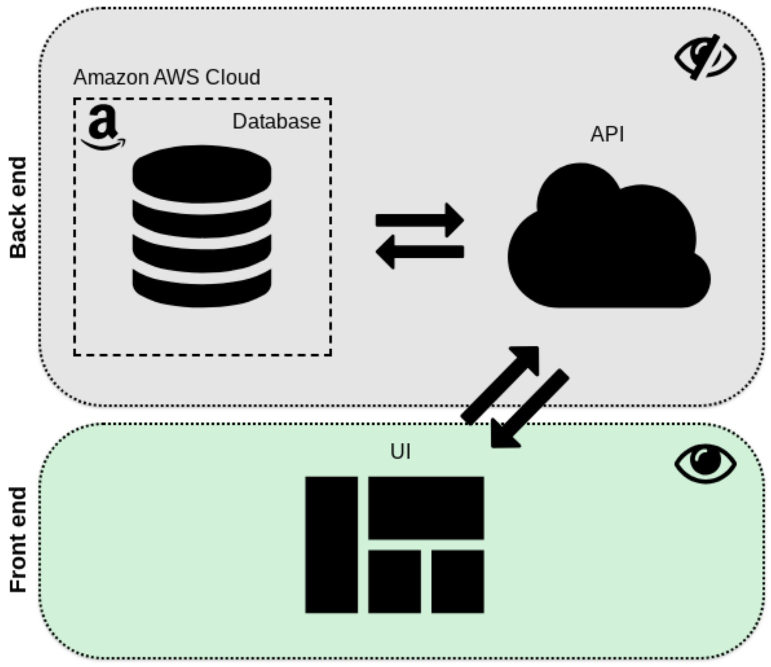


Figure 12. Application's architecure.

As you can see, back end is totally hidden from the user and does its job under the hood.

4.2 Back end

4.2.1 Database schema design

To start with, we must decide what database schema the API will work with. Considering that the application main goal is to manage exercises, the schema should definitely contain a table called **exercise**. Exercise should have **id** *(int PK*), **exercise_name** (*varchar NOT NULL UNIQUE*), **exercise_description** (*TEXT NULL*) **year** (*int NOT NULL <= current year*) of creation, students **passed** (*int NOT NULL NOT NEGATIVE*) and **failed** (*int NOT NULL NOT NEGATIVE*). For the table information to be consistent, we must introduce constraints: thus, each row should have unique exercise name, year must not be in the future, exercise name must not be empty string, passed and failed must not hold negative values and neither of the columns can be null.

Next, each exercise can have one or more test files – to accomplish this, we should introduce a new table, **test_file**, which will hold information about each test file and its relation to a specific exercise. To sum up, each row must have: id(int), file_name (*varchar UNIQUE NOT NULL*), **file_content** (*bytearray*), foreign key to **exercise** table as **exercise_id** (*int NOT NULL*) and additional column called **is_raw_code** (boolean NOT NULL). The last column's purpose is to indicate whether the file is the actual test file with the code (e.g *Test.java*), or it is a file with input/output values for a function or exercise (*testfile.json* or *testfile.txt*). There is no doubt that some constraints are to be introduced for this table as well. We would definitely want file name to be unique, but it is important not to add such constraint for raw file content, since that can considerably slow down the database performance. We will generate hash from the content instead and check uniqueness using the hash, calculated with built-in md5 function [11]. The last two constraints we must add will check if the file content is not empty and does not exceed 5 megabytes, which is more than enough for text files.

In addition, we want to introduce tags for each exercise. Each tag will represent some abstract attribute of the exercise: for instance, we want to categorize exercises by complexity, and in order to achieve that, we must introduce tags like *easy*, *intermediate*, *hard*. One exercise can have multiple tags and same tag can be attached to multiple exercises. This kind of relationship is called many-to-many. To avoid redundant data and keep our database performance at its peak, instead of introducing just one table, *tag*, we will introduce two: *tag* and *exercise_tags*. First table will hold info about each tag and an unique identifier: **id** (*int PK*), **tag_name** (*varchar NOT NULL UNIQUE*), whereas the second table will serve as an intermediate table, which purpose is to join exercise and tag together. Thus, the table **exercise_tags** will consist of id (*int PK*), **exercise_id** *(int FK to exercise*), **tag_id** (*int FK to tag*) and will have single unique check: the combination of **exercise_id** and **tag_id** must be unique, since we do not want repetitive data in our database.

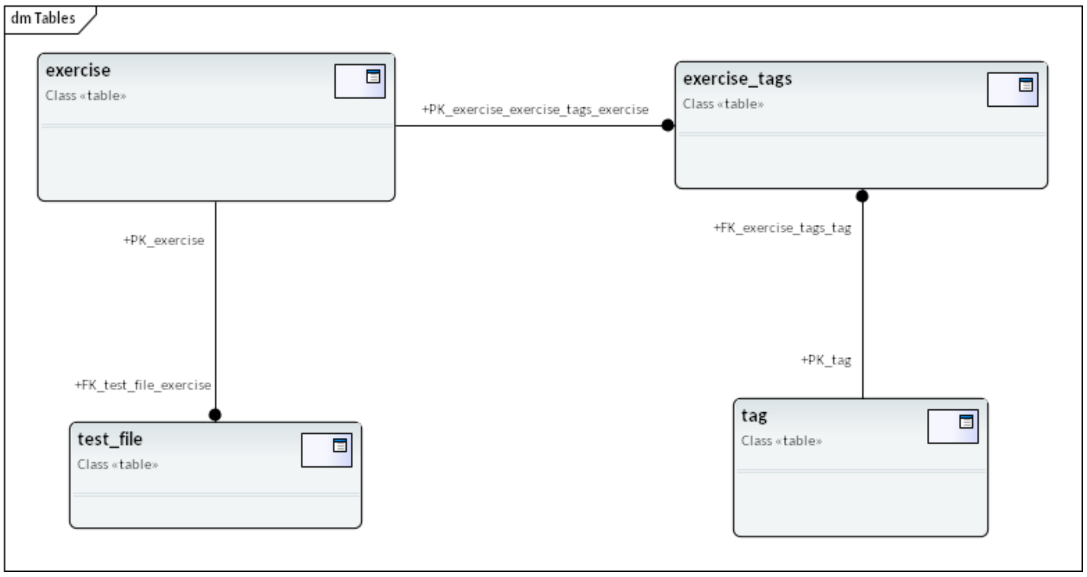


Figure 13. Schema representation in UML.

To compose UML representation of the schema and generate SQL from the diagram author used the tool called *Enterprise Architect*. For complete schema source code please refer to Appendix 2.

4.2.2 API design

Before writing first lines of code it is important to think through the architecture of the application, regardless if it is just a small API like this or more ambitious project. It is considered a good practice to follow SOLID principles when designing an application. In short, this means several things: each class/module should be responsible for single part of the application, code should be reused as much as possible, dependencies should be correctly organized [12].

The commonly used design pattern for such applications is called MVC. MVC, or model-view-controller, is a design pattern which was introduced to solve the problem of designing an application with rather trivial goal: handle user input and display the results [13]. Good API design commonly makes use of the following concepts:

- *Entity* for reflecting database structure in the code;
- *Dao* or Data Access Object for manipulating Entities (basic CRUD as well as more complex operations);

- *Dto* or Data Transfer Object for mapping and validation of the received data from the client;

- *Service* for additional manipulations with the results queried from the database/received from client;

- *Controller* for actual endpoint declarations;

This design pattern provides excellent logic separation, hence, makes it easier both to develop and maintain the code in the future. The flowchart of properly structured back end should look like this:

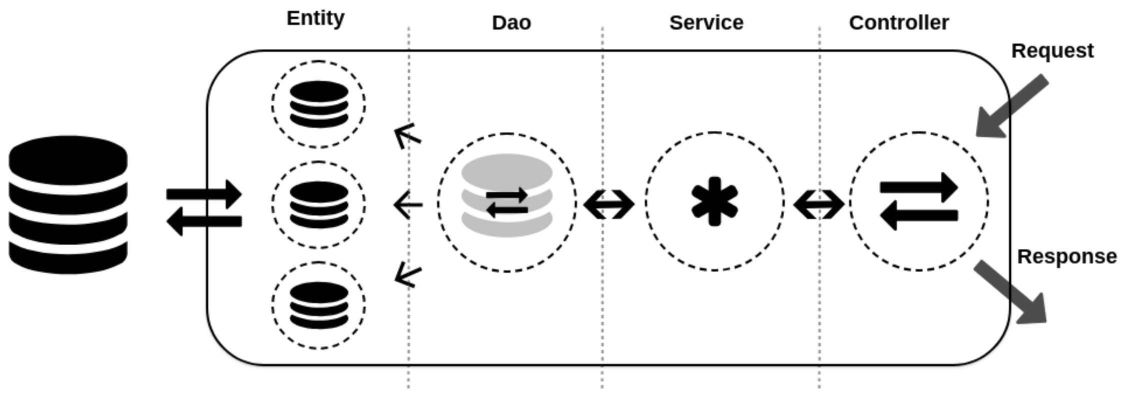


Figure 14. Back end structure and flow.

Ideally, each part of the application knows only about its predecessor: so, for instance, service knows about dao but does not know about the controller and dao knows about entity(s) but does not know about the service. Consider the following example: given a table called *user* with columns *id*, *email* and *password*, we want to set up an API which will allow basic CRUD for the table. Using the pattern described above, the pseudocode for such API will look like this:

```
@Entity
public class User {
	@Column
	private int id;
	@Column
	private String email;
	@Column
	private String password;
}

@Repository
public class UserDao<User> {
	// SELECT * FROM `user` WHERE `id` = “id”;
	public User findOne(Integer id) {...}
	// INSERT INTO `user` (...) VALUES (...);
	public void put(User u) {...}
	// DELETE FROM `user` WHERE `id` = “id”;
	public void delete(Integer id) {...}
}

@Service
public class UserService {
	private UserDao userDao;

	@Autowired
	public UserService(UserDao userDao) {
		this.userDao = userDao;
	}
	public User findOne(Integer id) {
		User u = this.dao.findOne(id);
		delete u.password;
		return u;
	}
	public void delete(Integer id) {...}
	public void put(User u) {...}
}
```

```
public class UserDto {
	@Min(0)
	private Integer id;
	@NotEmpty
	@IsEmail
	private String email;
	@NotEmpty
	@Min(7)
	private String password;
	public User getUser() { return new User(id, email, password); }
}

@Controller
@Path(“/user”)
public class UserController {
	private UserService userService;

	@Autowired
	public UserService(UserService userService) {
		this. userService = userService;
	}

	@Get(“/”)
	public User getUserById(@QueryParam(“id”) Integer id) {
		return this.userService.findOne(id);
	}

	@Put(“/”)
	public void put(@Valid UserDto dto) {
		return this.userService.put(dto.getUser());
	}

	@Delete(“/”)
	public void delete(@QueryParam(“id”) Integer id) {...}
}
```

As you can see from the example, each layer is responsible for its own part of the application and each next layer depends on the previous one. In this example, our API has three endpoints (table 2):

Table 2. Endpoints and actions.

Http method	**Example URL**	**Action**
GET	user?id=15	Fetches user info from the database
PUT	user/	Inserts new user to the database
DELETE	user?id=15	Deletes the user from the database

In order to avoid repetitive code and make the application more flexible, the special technique was used in this example. The technique is called dependency injection, or DI, which allows to create implementations of dependencies automatically, thus reducing the amount of manually written code. [14] In the example, the DI is marked with annotation *@Autowired* (commonly used in Java). So, to clarify, instead of writing

```
private UserDao userDao = new UserDao();
```

we can create the object instance implicitly with a help of DI:

```
@Autowired
private UserDao userDao;

Or

private UserDao userDao;
@Autowired
public UserDao(UserDao userDao) { this.userDao = userdao; }
```

The application needs to provide possibility of environment separation. Considering the fact that no sane person would prefer to manually build code and restart the server every time changes are made, it was decided to make use of *nodemon* library, which will reload the server when changes in the code are detected. Sensitive information like database authorization credentials etc must be separated from the logic of the application. Moreover, this information must be passed with environment variables, as this makes the application more flexible and secure.

It is more convenient to develop the application when all endpoints are exposed and easily accessible from browser, without need of third-party libraries like Postman. To fulfil this goal, Swagger should be installed and configured.

In addition, nobody wants to write queries by hand, especially trivial ones (e. g. CRUD). To avoid unnecessary code, ORM framework is required. It is also important security-wise, since escaping raw values is safer when it is done automatically. Fortunately, NestJs is shipped with such framework already included. The library is called *typeorm* and it provides all the basic functionality we need to generate secure SQL queries.

4.3 CI/CD integration

CI/CD is a group of techniques, which goal is to simplify the process of building and deploying the project [15]. To ease the deployment process and make first steps in the direction of fully automated builds and deploys, it was decided to use docker-compose + Gitlab runner as a CI/CD tool. Why is it important? At this moment the application consists of two parts – back end and front end, each requires manual installation of dependencies, building and copying files from one folder to another. Furthermore, the application deployed this way is not isolated from the server and its dependencies, which has potential security drawbacks and simply less clean [16] [17].

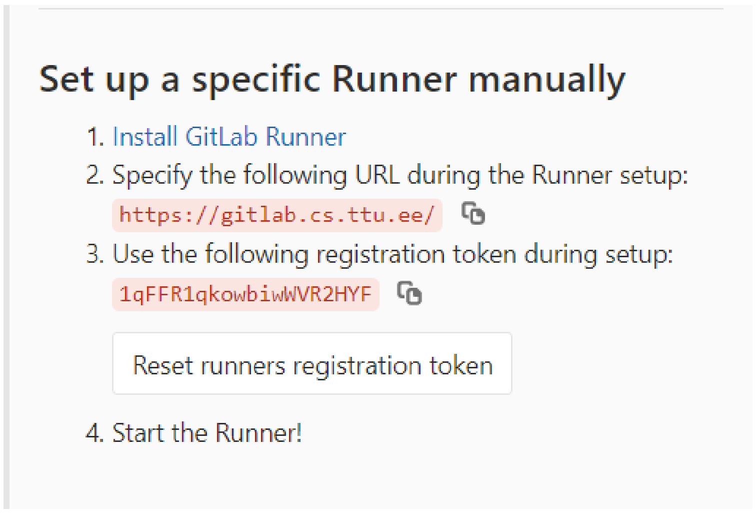
4.3.1 Docker and docker-compose

Docker and docker-compose provide functionality of building and deploying the whole app with a single command, moreover, in a containerized form. To start with, Dockerfile should be created. Dockerfile is a special type of file, which serves as an instruction for building a single container. [17] For an example of Dockerfile used for the application please refer to Appendix 3. Docker-compose is a higher-level tool, which is designed to manipulate multiple docker containers. Taking statement above into consideration, one may think that docker-compose is not necessary in case of our application, since for now it uses only one container – it was decided not to deploy database using docker, since it is considered bad practise to deploy a database to production inside a container. [18] Although this may be true now, docker-compose will ease the production rollouts in the future as the application grows from one container to multiple.

4.3.2 Setting up a Gitlab runner

As proper CI/CD is usually triggered by push events to target branches, we need to configure a Gitlab runner and attach it to the repository our project is hosted in. Gitlab runner is an open-source project, which allows to run jobs and send feedback to Gitlab. [19]

A new server is needed to set up the runner. As before, it was decided to take advantage of Amazon AWS EC2 instance. After it is created and configured (configuration is basically the same as in case of database server) we need to install the runner with a few simple commands. Installation is followed by registration process, which will attach the runner to desired Gitlab repository. During registration process, it is required to specify Gitlab's domain (Gitllab.cs.ttu.ee), secret token of the repository (in that case – *1qFFR1qkowbiwWVR2HYF*). The information is available at: *Settings* → *CI/CD* →*Runners* → *Set up Runner manually* (figure 15).



Set up a specific Runner manually

1. Install GitLab Runner
2. Specify the following URL during the Runner setup:
   `https://gitlab.cs.ttu.ee/`
3. Use the following registration token during setup:
   `1qFFR1qkowbiwWVR2HYF`

   Reset runners registration token
4. Start the Runner!

Figure 15. Instruction for setting up a Gitlab Runner.

The last thing to specify is the *Runner Executor*. For simplicity it was decided to use *shell* type [20], since the only thing is expected from the runner is to execute a single command:

```
$ docker-compose up -d --build
```

Clarification: run Dockerfile script (*docker-compose up*), rebuild every time (--*build*) run in detached mode (-*d*).

If everything went as it should, attached runner will appear under "Active runners" section:

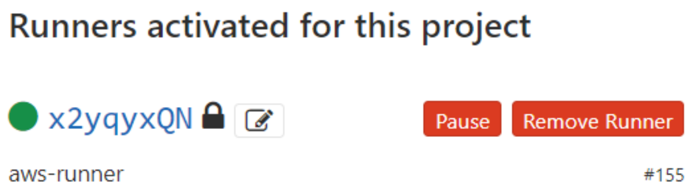
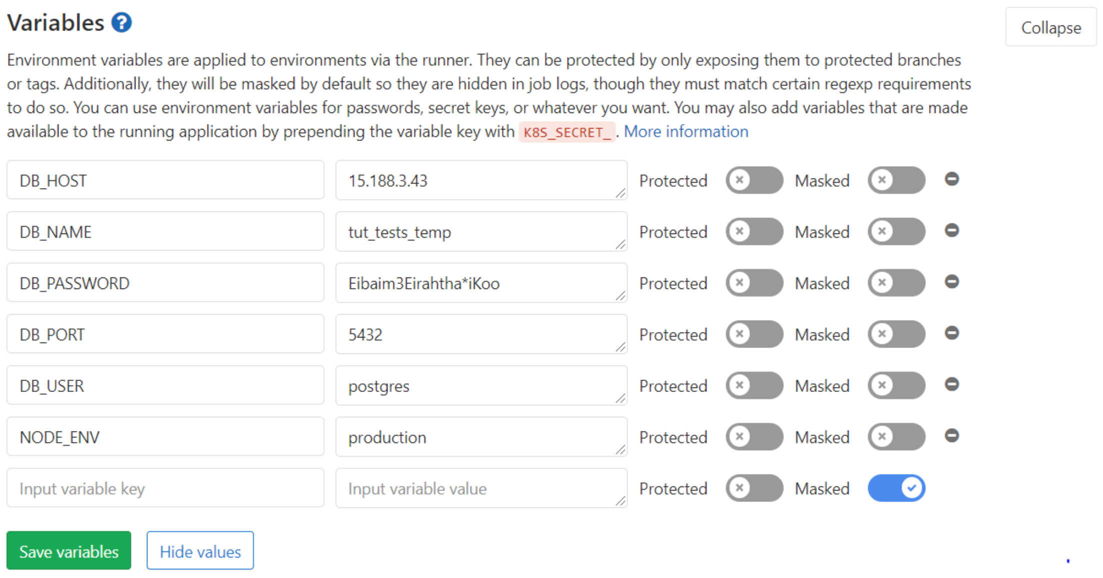


Figure 16. Attached runner.

The last thing to do is to set up a special file, called *Gilab-ci.yml*, which will serve s an instruction to the Runner and initialize environment variables which docker-compose will inject. For an example of *gilab-ci.yml* file with clarification notes please see Appendix 4. Initialization of environment variables can be done at **Settings → CI/CD → Variables** (figure 17).



Variables

Environment variables are applied to environments via the runner. They can be protected by only exposing them to protected branches or tags. Additionally, they will be masked by default so they are hidden in job logs, though they must match certain regexp requirements to do so. You can use environment variables for passwords, secret keys, or whatever you want. You may also add variables that are made available to the running application by prepending the variable key with `K8S_SECRET_`. More information

DB_HOST	15.188.3.43	Protected	Masked
DB_NAME	tut_tests_temp	Protected	Masked
DB_PASSWORD	Eibaim3Eirahtha*iKoo	Protected	Masked
DB_PORT	5432	Protected	Masked
DB_USER	postgres	Protected	Masked
NODE_ENV	production	Protected	Masked
Input variable key	Input variable value	Protected	Masked ✓

Figure 17. Injecting environment variables.

4.4 Front end

4.4.1 UI and UX design

Proper UI and UX is crucial, since it is the only layer of the application user directly interacts with. Even with most marvellous back end and brilliant CI/CD, an application will never have any success if it does not provide comprehensive UI.

General layout of the application will be as follows: the application should have a sidebar with links to available pages which could be toggled on and off as well as a top bar.

The application will consist of four pages:

1. Front page

Front page, or home page, will display data of all exercises in a table form. User will be able to delete exercises or go into more detailed view by clicking on specific one.

2. Exercise page

Exercise page will display all data of a specific exercise (including tags and test files). Using that page user will be able to modify exercise data, delete the exercise completely or add a new one.

3. Exercise tags page

Exercise tags page will display all the tags available in a table form. User will be able to add a new tag or delete existing ones.

4. Login page (Optional)

Simple login page.

As for the UI, it was decided to choose dark and minimalistic colours.

4.4.2 Code design

Same as with the API, frontend part of the application requires design as well. React provides its own pattern of logic separation using components. A component is usually just a JSX file, which is responsible for the small part of the whole application. As stated in React docs [21]:

Components let you split the UI into independent, reusable pieces,
and think about each piece in isolation.

In context of this application, it is convenient to separate logic by entities: exercise tag, and test file. Each entity will have its own table, where its properties will be displayed; thus, it was decided to separate components as follows:

```
components/
        ├───errors/
        ├───exercise/
        ├───layout/
        ├───minor/
        ├───router/
        ├───table/
        └───tag/
```

Where *layout* folder includes general layout components, like sidebar.tsx and top-bar.tsx, *router* is for routing configurations and *minor* is for utility components, for example, spinner animation component. Other names are self-explanatory.

5 Summary

The purpose of the paper is to give an overview of the work done, as well as to acquaint the reader with its results. The work process included designing and deploying a web application for managing and categorizing programming exercises. In addition, database schema was designed, and CI/CD was configured. As a result, the process of managing exercises in the future will be more optimized than before.

Although major part of the work is done, there is still space for improvements. For instance, test file generation functionality was left undone. In addition, the application in its current form does not have proper authentication, which is crucial (ideally, authentication should be perform with Uni-ID and be role-based). Thirdly, although invalid user inputs are properly handled with both API and database, front end does not provide much feedback and error messages, which is important as well. Another major part of every application is automated tests, which were skipped in this case due to the lack of time.

To sum up, although there are some things left to do, the application in its current form is ready to be used to fulfil its purpose.

References

[1] "TypeScript Design Goals," 2014. [Online]. Available: https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals. [Accessed 10 08 2019].

[2] M. Shahin, "WHEN TO USE SQL VS. NOSQL," 08 11 2018. [Online]. Available: https://www.integrant.com/when-to-use-sql-vs-nosql/. [Accessed 11 08 2019].

[3] "How to swap values of two rows in MySQL without violating unique constraint?," [Online]. Available: https://stackoverflow.com/questions/11207574/how-to-swap-values-of-two-rows-in-mysql-without-violating-unique-constraint. [Accessed 11 08 2019].

[4] S. A, "The Best JS Frameworks for Front End," 19 01 2019. [Online]. Available: https://rubygarage.org/blog/best-javascript-frameworks-for-front-end. [Accessed 13 08 2019].

[5] "Front end frameworks popularity," 12 2018. [Online]. Available: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190.

[6] "9 Popular JavaScript Frameworks for 2019," 01 2019. [Online]. Available: https://raygun.com/blog/popular-javascript-frameworks/. [Accessed 2019 08 13].

[7] "Angular vs Vue vs React," 13 06 2019. [Online]. Available: https://www.codeinwp.com/blog/angular-vs-vue-vs-react/. [Accessed 14 08 2019].

[8] "What is Amazon EC2?," [Online]. Available: https://docs.aws.amazon.com/en_us/AWSEC2/latest/UserGuide/concepts.html. [Accessed 2019 08 14].

[9] "Install PostgreSQL with phpPgAdmin on Ubuntu 16.04," 24 08 2016. [Online]. Available: https://www.rosehosting.com/blog/install-postgresql-with-phppgadmin-on-ubuntu/. [Accessed 14 08 2019].

[10] “How to allow remote connections to PostgreSQL database server,” [Online]. Available: https://bosnadev.com/2015/12/15/allow-remote-connections-postgresql-database-server/. [Accessed 16 08 2019].

[11] “PostgreSQL MD5 Function,” [Online]. Available: http://www.postgresqltutorial.com/postgresql-md5/. [Accessed 10 08 2019].

[12] S. LH, “SOLID Principles: Explanation and examples,” [Online]. Available: https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4. [Accessed 17 08 2019].

[13] “The MVC pattern in theory and practice,” [Online]. Available: http://warp.povusers.org/programming/mvc.html. [Accessed 14 08 2019].

[14] J. Shore, “Dependency Injection Demystified,” 22 03 2006. [Online]. Available: https://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html. [Accessed 17 08 2019].

[15] S. PITTET, “Continuous integration vs. continuous delivery vs. continuous deployment,” [Online]. Available: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. [Accessed 10 08 2019].

[16] “Top 7 benefits of using containers,” 03 04 2017. [Online]. Available: https://blog.kumina.nl/2017/04/the-benefits-of-containers-and-container-technology/. [Accessed 15 08 2019].

[17] “Docker documentation,” [Online]. Available: https://docs.docker.com/. [Accessed 15 08 2019].

[18] “Should You Run Your Database in Docker?,” [Online]. Available: https://vsupalov.com/database-in-docker/. [Accessed 16 08 2019].

[19] “GitLab Runner Docs,” [Online]. Available: https://docs.gitlab.com/runner/. [Accessed 16 08 2019].

[20] “Runner Executor - Shell,” [Online]. Available: https://docs.gitlab.com/runner/executors/shell.html. [Accessed 20 08 2019].

[21] “React Docmentation,” [Online]. Available: https://reactjs.org/docs. [Accessed 20 08 2019].

[22] D. Radulov, “Continuous Integration with Docker Compose,” 13 09 2017. [Online]. Available: https://semaphoreci.com/blog/2017/09/13/continuous-integration-with-docker-compose.html. [Accessed 10 08 2019].

[23] “UNIQUE constraint on large VARCHARs - PostgreSQL,” [Online]. Available: https://dba.stackexchange.com/questions/94205/unique-constraint-on-large-varchars-postgresql. [Accessed 15 08 2019].

Appendix 1 – Example of input/output file

```
{
    "year": 2017,
    "author": "Karen",
    "exercise_name": "comparator",
    "data": [
        {
            "function_name": "comparator",
            "arguments": [
                "int"
            ],
            "output_type": "string",
            "test_data": [
                {
                    "input": [
                        93
                    ],
                    "output": "The input number is bigger than 5!"
                },
                {
                    "input": [
                        5
                    ],
                    "output": "The input number is 5!"
                },
                {
                    "input": [
                        2
                    ],
                    "output": "The input number is smaller than 5!"
                }
            ]
        }
    ]
}
```

Appendix 2 – SQL code of database schema

```
/* ---------------------------------------------------- */
/*  Generated by Enterprise Architect Version 13.0          */
/*  Created On : 13-Jul-2019 2:03:04 PM                     */
/*  DBMS       : PostgreSQL                                 */
/* ---------------------------------------------------- */

/* Drop Sequences for Autonumber Columns */

DROP SEQUENCE IF EXISTS exercise_id_seq
;

DROP SEQUENCE IF EXISTS exercise_tags_id_seq
;

DROP SEQUENCE IF EXISTS tag_id_seq
;

DROP SEQUENCE IF EXISTS test_file_id_seq
;

/* Drop Tables */

DROP TABLE IF EXISTS exercise CASCADE
;

DROP TABLE IF EXISTS exercise_tags CASCADE
;

DROP TABLE IF EXISTS tag CASCADE
;

DROP TABLE IF EXISTS test_file CASCADE
;

/* Create Functions */

CREATE OR REPLACE FUNCTION f_check_year_not_in_the_future()
  RETURNS trigger AS
$BODY$
BEGIN
    IF NEW.year > DATE_PART('year', current_date) THEN
            RAISE EXCEPTION 'Year must not be in the future'
                  USING HINT = 'Change the year';
            END IF;

   RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql
;

/* Create Tables */

CREATE TABLE exercise
(
      id       integer       NOT       NULL                      DEFAULT
NEXTVAL(('"exercise_id_seq"'::text)::regclass),
	exercise_name varchar(50) NOT NULL,
	exercise_description text NULL,
	passed integer NOT NULL DEFAULT 0,
	failed integer NOT NULL DEFAULT 0,
	year integer NOT NULL
)
;

CREATE TABLE exercise_tags
(
	id	integer	NOT	NULL			DEFAULT
NEXTVAL(('"exercise_tags_id_seq"'::text)::regclass),
	exercise_id integer NOT NULL,
	tag_id integer NOT NULL
)
;

CREATE TABLE tag
(
	id	integer	NOT	NULL			DEFAULT
NEXTVAL(('"tag_id_seq"'::text)::regclass),
	tag_name varchar(255) NOT NULL
)
;

CREATE TABLE test_file
(
	id	integer	NOT	NULL			DEFAULT
NEXTVAL(('"test_file_id_seq"'::text)::regclass),
	exercise_id integer NOT NULL,
	file_name varchar(150) NOT NULL,
	file_content bytea NOT NULL,
	is_raw_code boolean NOT NULL DEFAULT false
)
;

/* Create Primary Keys, Indexes, Uniques, Checks */

ALTER TABLE exercise ADD CONSTRAINT PK_exercise
	PRIMARY KEY (id)
;

ALTER TABLE exercise
  ADD CONSTRAINT AK_exercise_exercise_name_uniq UNIQUE (exercise_name)
;

ALTER TABLE exercise ADD CONSTRAINT CHK_exercise_passed_is_not_negative
CHECK (passed >= 0)
;

ALTER TABLE exercise ADD CONSTRAINT CHK_exercise_failed_is_not_negative
CHECK (failed >= 0)
;

ALTER TABLE exercise ADD CONSTRAINT CHK_exercise_exercise_name_not_empty
CHECK (LENGTH(TRIM(exercise_name)) > 0)
;

CREATE TRIGGER TRG_exercise_year_not_in_the_future
BEFORE INSERT OR UPDATE ON exercise
FOR EACH ROW
```

```
      EXECUTE PROCEDURE f_check_year_not_in_the_future();
;

ALTER TABLE exercise_tags ADD CONSTRAINT PK_exercise_tags
      PRIMARY KEY (id)
;

ALTER TABLE exercise_tags
  ADD     CONSTRAINT     AK_exercise_tags_exercise_tag_uniq     UNIQUE
(exercise_id,tag_id)
;

CREATE INDEX IXFK_exercise_tags_exercise ON exercise_tags (exercise_id
ASC)
;

CREATE INDEX IXFK_exercise_tags_tag ON exercise_tags (tag_id ASC)
;

ALTER TABLE tag ADD CONSTRAINT PK_tag
      PRIMARY KEY (id)
;

ALTER TABLE tag
  ADD CONSTRAINT AK_tag_tag_name_uniq UNIQUE (tag_name)
;

ALTER  TABLE  tag  ADD  CONSTRAINT  CHK_tag_tag_name_not_empty  CHECK
(LENGTH(TRIM(tag_name)) > 0)
;

ALTER TABLE test_file ADD CONSTRAINT PK_test_file
      PRIMARY KEY (id)
;

ALTER TABLE test_file
  ADD CONSTRAINT AK_test_file_file_name_uniq UNIQUE (file_name)
;

ALTER          TABLE          test_file          ADD          CONSTRAINT
CHK_test_file_file_content_not_empty CHECK (LENGTH(file_content) > 0)
;

ALTER          TABLE          test_file          ADD          CONSTRAINT
CHK_test_file_file_content_does_not_exceed_5_mb                  CHECK
(LENGTH(file_content) <= 5 * pow(10, 6))
;

CREATE UNIQUE INDEX IXAK_test_file_file_content_hash_uniq ON test_file
(md5(file_content))
;

CREATE INDEX IXFK_test_file_exercise ON test_file (exercise_id ASC)
;

/* Create Foreign Key Constraints */

ALTER TABLE exercise_tags ADD CONSTRAINT FK_exercise_tags_exercise
      FOREIGN  KEY  (exercise_id)  REFERENCES  exercise  (id)  ON  DELETE
Cascade ON UPDATE No Action
;
```

```
ALTER TABLE exercise_tags ADD CONSTRAINT FK_exercise_tags_tag
      FOREIGN KEY (tag_id) REFERENCES tag (id) ON DELETE Cascade ON
UPDATE No Action
;

ALTER TABLE test_file ADD CONSTRAINT FK_test_file_exercise
      FOREIGN KEY (exercise_id) REFERENCES exercise (id) ON DELETE
Cascade ON UPDATE No Action
;

/* Create Table Comments, Sequences for Autonumber Columns */

CREATE SEQUENCE exercise_id_seq INCREMENT 1 START 1
;

CREATE SEQUENCE exercise_tags_id_seq INCREMENT 1 START 1
;

CREATE SEQUENCE tag_id_seq INCREMENT 1 START 1
;

CREATE SEQUENCE test_file_id_seq INCREMENT 1 START 1
;
```

Appendix 3 – Dockerfile and docker-compose.yml

Dockerfile:

```
### Use image based on alpine linux (most lightweight) which has nodejs installed by default ###
FROM node:12-alpine

WORKDIR /usr/app/

RUN apk update && apk upgrade
RUN mkdir tut-tests-api && mkdir tut-tests-ui

###  Caching node modules for faster subsequent builds ###
COPY ./tut-tests-api/package.json /usr/app/tut-tests-api/package.json
RUN cd tut-tests-api/ && npm i
COPY ./tut-tests-ui/package.json /usr/app/tut-tests-ui/package.json
RUN cd tut-tests-ui/ && npm i
###  ================================================ ###

COPY . /usr/app
RUN cd ./tut-tests-api && npm run build
RUN cd ./tut-tests-ui && npm run build && cp -a build/* ../tut-tests-api/build/public
### Clean up ###
RUN rm -rf tut-tests-ui/ && cd tut-tests-api/ && ls | grep -v -E "build|node_modules" | xargs rm -rf

WORKDIR /usr/app/tut-tests-api/build/

EXPOSE 3000

CMD ["node", "src/main.js"]
```

docker-compose.yml:

```
version: '3'

services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - "DB_USER=${DB_USER}"
      - "DB_PASSWORD=${DB_PASSWORD}"
      - "DB_NAME=${DB_NAME}"
      - "DB_HOST=${DB_HOST}"
      - "DB_PORT=${DB_PORT}"
      - "NODE_ENV=${NODE_ENV}"
```

Appendix 4 – Gilab-ci.yml

```
image: docker

services:
    - docker:dind

# before_script:
#     - sudo apt update && sudo apt upgrade

##### Maybe later it will be better to do everything here, without docker-
compose. But now is now. #####
stages:
#  - build
  - test
  - deploy

run_tests:
    stage: test
    script: echo "No tests yet :("

deploy_to_server:
    stage: deploy
    ### Available only on push to master branch ###
    only:
        - master
    environment: production
    script:
        - sudo apt install python-pip -y
        - pip install docker-compose
        - docker-compose up -d --build
    ### User must trigger the job manually ###
    when: manual
```

Appendix 4 – Repository link

Repository https://gitlab.cs.ttu.ee/Karen.Ofljan/tut_test_generator