Asgeir Hunshamar

A Flashcard Based Web Application for Collective Learning and Peer Review Based Evaluation of Students

May 2021

Master's thesis
2021
Asgeir Hunshamar

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

A Flashcard Based Web Application for Collective Learning and Peer Review Based Evaluation of Students

Asgeir Hunshamar

Cybernetics and Robotics
Submission date: May 2021
Supervisor: Sverre Hendseth

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem statement

A flashcard based learning tool is to be developed for use in the Real-Time Programming course (TTK4145) at NTNU. This flashcard system will handle the creation, peer review and studying of student submitted flashcards.

The following tasks will be performed in order to create a maintainable, working web application to realize the flashcard system.

- Specify a list of functional and non-functional requirements for the system
- Develop the client-server system, consisting of a single-page application and a REST-API connected to a SQL database, satisfying the system requirements.
- Test, deploy and host the system on a server
- Beta test the system on students to test functionality, usability and scalability of the application.

Abstract

This thesis describes the implementation of a flashcard-based educational application developed for the course TTK4145 - Real-Time Programming at NTNU. The system was developed with the goal of providing a new, unique type of exercise for students to submit, whilst simultaneously providing a learning tool for students to study the course material and to prepare for the final exam. The goal of the application is to facilitate a collective effort of student contributions to create a set of quality flashcards, covering the curriculum of the course. This collective set of flashcards can be used by students to effectively study the course material. A peer-review process will filter out the highest quality flashcards, whilst also providing a basis for evaluation of students.

The implemented web application was beta tested as a voluntary exercise by students attending the Real-Time Programming Course. The students submitted flashcards that were rated and evaluated in a peer review process by fellow students. A selection of unique, highly rated flashcards were made available for studying. The flashcard application was actively used by 105 students. A majority of students surveyed found that the flashcards studied had a high degree of quality and relevance to the course material and that the flashcard application helped them prepare for the exam.

The effectiveness and ease of use of this application has been demonstrated through beta testing and user surveys, but further work remains to be implemented for maximizing the learning effect and potential of the application.

The application is available at http://ttk4145flashcards.no alternative link

Sammendrag

Denne oppgaven beskriver implementeringen av en flashcard-basert pedagogisk applikasjon utviklet for emnet TTK4145 - Sanntidsprogrammering ved NTNU. Systemet ble utviklet med formål om å tilby en ny, unik type øving for studentene å levere, samtidig som det gir et læringsverktøy for studenter til å studere emnematerialet og forberede seg til avsluttende eksamen. Målet med applikasjonen er å legge til rette for en samlet innsats av studentbidrag for å lage et sett med flashcard av høy kvalitet som dekker pensum i emnet. Dette kollektive settet med flashcards kan brukes av studenter til å effektivt studere emnematerialet. En fagfellevurderingsprosess vil filtrere ut flashcards av høyeste kvalitet, samtidig som det gir grunnlag for evaluering av studentene.

Den implementerte webapplikasjonen ble betatestet som en frivillig øvelse av studenter som var meldt opp i sanntidsprogrammeringsemnet. Studentene leverte flashcards som ble vurdert og evaluert i en fagfellevurdering av medstudenter. Et utvalg av unike, høyt rangerte flashcards ble gjort tilgjengelig for å studere. Flashcard-applikasjonen ble aktivt brukt av 105 studenter. Et flertall av spurte studenter svarte at flashcard som ble studert i applikasjonen hadde en høy grad av kvalitet og relevans for emnematerialet, og at applikasjonen hjalp dem med å forberede seg til eksamen.

Effektiviteten og brukervennligheten til denne applikasjonen har blitt demonstrert gjennom betatesting og brukerundersøkelser, men ytterligere arbeid gjenstår å implementeres for å maksimere læringseffekten og potensialet til applikasjonen.

Applikasjonen kan testes på http://ttk4145flashcards.no alternativ link

Table of Contents

Chapter 1

Introduction

1.1 Motivation and Project Description

Compulsory activities, usually in the form of obligatory exercises or more creative projects, are commonly used at Norwegian universities and higher education. The purpose of these exercises is usually to either motivate the students to work continuously through the semester, deterring them from procrastinating the entire workload of a subject to the weeks leading up to the final exam, or to provide an alternative mode of evaluation of students from the written examination. Or both.

The problem with this approach, especially with the more generic obligatory exercises consisting of a set of tasks from the curriculum with rather specific answers, is the widespread plagiarism and sharing of answers, also known as *koking*, a term coined by students to describe the act of copying answers either from an answer sheet or from other students [1]. An interview with PhD Candidate Tir Aksel Heirung revealed that of 50 submitted assignments in a subject, nine were clearly copied from an answer sheet that could be found online. [1]

Another problem with the current approach is the lack of repetition and long-term review of the course material. Students are expected to retain the information learned throughout the semester, and preferably throughout their education and career. Without any attempt to retain information learned, through conscious review of the material, humans tend to forget most of the newly acquired knowledge in a matter of days. This phenomenon is known as *the forgetting curve* and was first described by German psychologist Hermann Ebbinghaus in 1985 [2]. Many different strategies and techniques are commonly used by students to resist this phenomenon. These strategies include well organized notes, flashcards, re-watching lectures and redoing assignments, but without a systematic tool for retention, a lot of the learned material will usually quickly be forgotten.

This thesis will describe the development of a new and unique learning tool, which

is to be used in the subject TTK415 - Real Time Programming at NTNU. This tool will be developed with the goal of encouraging students to gain an understanding of the curriculum early as well as a tool for students to review the curriculum and retain the material throughout the semester. The tool will allow students to deliver mandatory assignments consisting of a set number of questions and answers for each section of the curriculum after it has been lectured. Through systematic peer review of the assignments, the highest-rated questions and answers will be made available as flashcards for students to study. The cards will be rated on quality and difficulty, and questions too similar can be marked as duplicates, discouraging the students from *koking* questions from each other or from older exams. The flashcard study review process is voluntary, but encouraged. The tool will be developed with the goal to utilize spaced repetition for flashcard studying, a technique for active recall with increasing intervals, taking advantage of the forgetting curve, optimizing the retention of information.

1.2 Similar Applications

Multiple flashcard based learning applications already exists. A popular choice is *Anki*, a flashcard program supporting the creation of Flashcards using HTML and using spaced repetition for flashcard reviews. Anki is especially popular among medical students, with a 2015 study finding that 31% of medical students answering a survey reported using Anki. It was also found that the use of Anki flashcards resulted in higher scores in the United States Medical Licensing Examination [3]. Whilst it is possible to share flashcard decks in Anki, collaborative decks with user ratings are not a feature.

Other popular flashcard applications include Brainscape, Quizlet, Cram and IDoRecall [4], but none of these, or any other researched applications, include any similar collective, peer review based rating and quality assurance features of the flashcard application described in this thesis.

Chapter 2

Theory

2.1 Flashcards

Studying flashcards is a commonly used study technique by students to retain information. Flashcards, either digital or physical, consists of a question on one side and an answer on the other. The studying of flashcards provides an alternative mode of studying from passively reading the material, whilst also taking advantage of the *testing effect*, a well-established learning method that increases the effectiveness of reviewing during learning by answering questions about the learning content rather than restudying the material [5]. A study concluded that university students answering short-answer questions after a lecture provided greater retention of learning than students who read summarizing statements about the core lecture content [5].

The efficiency of flashcards as a study aid for higher education has also been demonstrated through a study on the use of flashcards for studying for exams. The study found that students in an Introduction to Psychology class who used flashcards for studying had significantly higher exam scores overall than those students who did not use flashcards. [6]

2.2 Spaced Repetition and the Forgetting Curve

The information retention effect of flashcards can be further optimized using a *spaced repetition* approach. Spaced repetition is a learning technique designed to exploit the psychological *spacing effect*, a phenomenon that suggests that active recall with increasing time intervals increases the probability of remembering information. [7]. For most people, a single exposure to a fact or concept is usually inadequate for long-term retention. Flashcards can provide a great resource to be frequently exposed to the material, but with a sporadic approach to the repetition of flashcards, the result will usually be a lot of redundant studying of already

familiar and learned material. The idea behind spaced repetition is to create a system to increase the frequency of reviews for hard and unfamiliar flashcards, whilst the familiar flashcards are de-prioritized by gradually increasing the time interval between their exposure to the student. Spaced repetition will also space out the flashcard reviews, preventing cramming and increasing long-term retention of the material. [8]

The *forgetting curve*, illustrated in figure 2.1, discovered by Hermann Ebbinghaus in 1885, representing the spacing effect, shows how information is forgotten over time when there is no attempt to recall it, and how each repetition of the material increases the interval before the next repetition is needed. [8]. Initially, the material may need to be repeated within days, but as the repetition interval increases, using spaced repetition, the material can be remembered for months or years before requiring new repetitions.

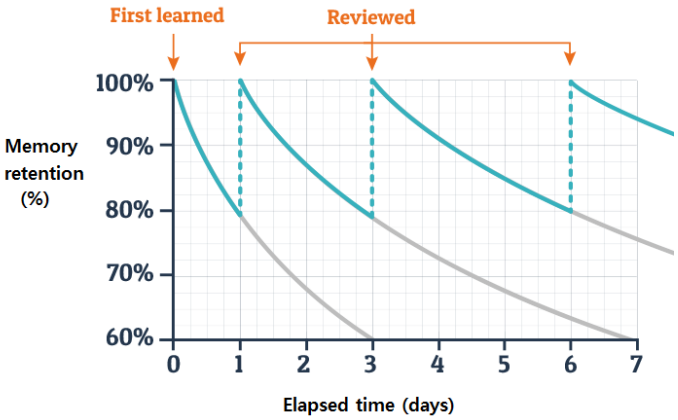


Figure 2.1: Ebbinghaus' Forgetting Curve [9]

2.3 Peer Review and Feedback

Peer reviewing is a reciprocal process, where student produce and receive feedback on each others' work. Peer reviews are familiar to most students at NTNU and is also utilized in the Real Time Programming subject, where students are assigned to review each others code for the course lab assignments. A study conducted at the University of Strathclyde found that *producing feedback reviews engages students in multiple acts of evaluative judgement, both about the work of peers, and, through a reflective process, about their own work; that it involves them in both invoking and applying criteria to explain those judgements; and that it shifts control of feedback processes into students' hands, a shift that can reduce their need for external feedback.* [10]. In addition the use of peer reviews significantly reduces the workload of academic staff and by collecting feedback from multiple peers, a larger quantity and variety of feedback is collected.

The use of peer review feedback in this application ensures that minimal additional workload is given to the student assistants or academic staff of the course as well as giving the students the learning benefit gained by evaluating and reflecting each others' work.

Chapter 3

Tools and Methods

3.1 Client-Server Model

The application is built using the client-server model, a *two-tier* architecture model, providing a presentation layer, often referred to as the *front end*, and a data access layer, referred to as the *back end*. [11]

The main principle behind the client-server architecture is a separation of concerns. The user interface concern is separated from the data storage concern, improving portability of the user interface across multiple platforms and improving the scalability of the system by simplifying the server component. This allows for the components to evolve independently. [11]

The technology stack used to implement the client-server application is summarized in figure 3.1

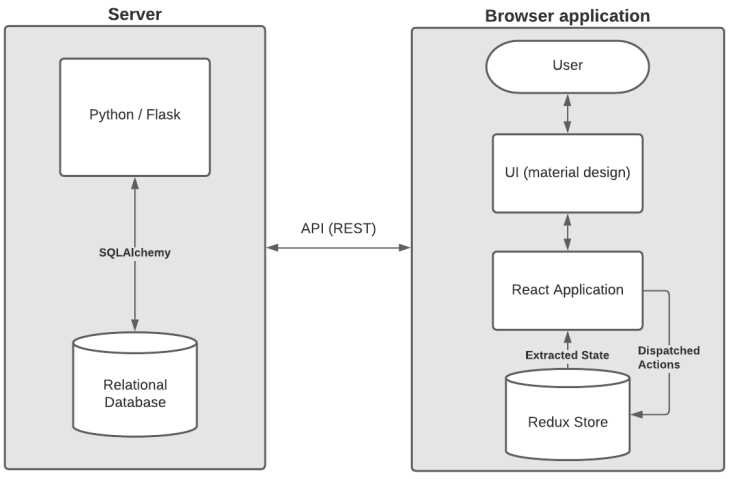


Figure 3.1: Application Architecture

3.1.1 REST API

To pass data between the client-side and the server-side, the server application was built as a *Representational state transfer application programming interface* (REST API). The REST API defined a set of endpoints, or routes, that can be requested to fetch or manipulate data on the server [11]. Five types of requests are used in REST APIs

- **GET:** Used to fetch data
- **POST:** Used to send data
- **PUT:** Used to replace existing data
- **PATCH:** Used to modify existing data
- **DELETE:** Used to delete existing data

3.1.2 Relational Database

To store the data on the server, a relational database is utilized. Relational databases store data in tables representing entity types. Each row in each table has a unique key and data in one table can be linked to data from other tables by using *foreign keys*, referencing the unique key of another table row. [12]. The use of structured query language (SQL) to manage data in a relational database allows for retrieving data using queries, requesting information based on the fields in the table.

3.1.3 Single Page Application

In order to provide a better user experience as well as a high level of speed and performance on the web page, the client application is built as a *single page application* (SPA) with React. SPA is a type of web application or website that dynamically reloads selected page elements in line with user interactions in order to avoid fetching entire new pages from a server, reducing the amount of data needed to be fetched and increasing the speed of the application. [13].

3.2 Python with Flask

The Python programming language was chosen to implement the server. Python was chosen because of its emphasis on code readability, simplicity, and its vast selection of libraries and packages. Python is becoming a more and more popular language, having *a solid claim to being the fastest-growing major programming language* [14].

Python is a language familiar to most NTNU students, which in conjunction with the benefits listed above allows for the creation of a maintainable and readable server application that can be handed over to future students at NTNU for further development.

There exists multiple web frameworks for handling the REST API implementation in Python. The two most popular frameworks are, according to the 2020 JetBrains Python Developer Survery, by far Flask and Django [15]. Whilst Django is a more broad, monolithic web framework with a steep learning curve, Flask is a micro web framework, meaning it aims to keep its core simple, but extensible. Whilst Django makes a lot of decisions for you when it comes to database management and security, Flask supports a wide range of third-party extensions, leaving the decisions up to the developer. [16]. For its flexibility and simplicity Flask was chosen as a web framework.

Creating a REST API endpoint in Flask is as simple as importing the framework, creating an instance of the Flask class and defining a function with the `route()` decorator to specify what URL to trigger the function as well as which HTTP methods are allowed on the route. An example is shown below in listing 3.1, with the `jsonify()` method creating a JSON representation of the data being returned by the API, in this case, a python dictionary.

Code listing 3.1: simple api route in Flask

```
1 from flask import Flask, jsonify
2 app = Flask(__name__)
3
4 @app.route("/api/hello", methods=["GET"])
5 def hello_world():
6    return jsonify({"status":"Hello,World!"})
```

The API can be tested using *curl*, a tool to transfer data from a server. Running the `curl http://localhost:5000/api/hello` command would make a GET request to this REST API running on localhost:5000, returning `{"status":"Hello,World!"}`

3.2.1 Features of Flask

Blueprints and Modularization

Blueprints are a built-in feature of Flask, allowing organizing related code, modularizing the application into different components, simplifying how large applications work and their structure. [17]. An example of a blueprint, used for the user module, can be created simply by writing

```
user_blueprint = Blueprint("user", __name__)
```

Routes are added to the `user_blueprint` by writing

```
user_blueprint.route("/api/users", methods=["GET"])
```

followed by the rest of the route information as shown in listing 3.1

This allows for all the routes, logic and methods of the modules to be separated from the main application into different files and folders. These blueprints are simply imported to the main `app.py` file using the `register_blueprint()` method as shown below

```
from blueprints.user import user_blueprint

app.register_blueprint(user_blueprint)
```

Session

Flask sessions is a built-in feature in Flask that provides a method to store data on the server-side. Sessions store information specific to a user from one request to another, allowing data to be shared between routes in the API. To use sessions, users must be allowed to make authenticated requests, which can be achieved by modifying the CORS policy of the application, further detailed in section 3.2.6, to support credentials by setting `CORS(app, supports_credentials=True)`. In addition, API requests to the URLs using session must include the property `withCredentials: true` to send cross-origin cookies required by *session* [18].

The use of sessions in the server allows for example for one route to receive user-data from an external login api, storing it in the session, allowing the information to be accessible from another *login callback* route. A simplified example, not very different from the one implemented in the final application, is shown in listing 3.2

Code listing 3.2: Login example using sessions

```
user_blueprint.route("api/login/userdata")
def user_data():
   userdata = request.values.get("userdata")

   # loads - serialize json data to python dict
   session["userdata"] = json.loads(userdata)

   .... # return redirect to login page

user_blueprint.route("api/login/callback")
def login_callback():
   userdata = session.pop("userdata")

   .... # return login token
```

3.2.2 Package Managing System - Pip

The following is a description of packages used in conjunction with Flask to implement the server application in Python. All packages were installed with pip, a package management system for python, in a virtual environment. A virtual environment is a self-contained directory tree of packages used to isolate the dependencies of the project, ensuring the correct versions of the packages are utilized. [19]. All python dependencies, with the related version, are summarized in a `requirements.txt` file. All dependencies are installed by running the `pip install -r requirements.txt` command inside the virtual environment. The `requirements.txt` file of the server application contains the following dependencies, shown in listing 3.3.

Code listing 3.3: requirements.txt

```
Flask_JWT_Extended==3.25.0
Flask==1.1.2
requests==2.25.1
Flask_Script==2.0.6
alembic==1.5.8
Flask_Migrate==2.7.0
Flask_Cors==3.0.10
PyMySQL==1.0.2
Flask_SQLAlchemy==2.4.4
SQLAlchemy==1.3.23
python-dotenv==0.17.0
```

3.2.3 Database ORM - SQLAlchemy

The communication between python and the relational database was facilitated using the SQLAlchemy extension for Flask. SQLAlchemy functions as an object relational mapper (ORM) which translates python classes to tables on relational

databases. SQLAlchemy converts python function code into database queries, allowing database objects to be treated as python objects, allowing the creation of database-agnostic code for the application [20]. SQLAlchemy is compatible with different types of relational databases, including PostgreSQL, MySQL, and SQLite. All three of these were tested and used with the application through development, with little modification to the python code necessary.

A minimal example of the SQLAlchemy implementation of the User table is shown in listing 3.4. With the exception of the ORM mapping process, which here converts the class into a database table with the columns *id* and *username*, with a creator relationship to the *Flashcard* class, the class mostly remains a normal Python class, that can be given ordinary attributes and methods. Here a `to_dict()` is implemented to return a dictionary presentation of the class attributes, useful for formatting the outgoing data of the REST API.

Code listing 3.4: Python example

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)

class User(db.Model):
   __tablename__ = "user"
   id = db.Column(db.Integer, primary_key=True)
   username = db.Column(db.String(24), unique=True)

   flashcards = db.relationship("Flashcard", backref="creator")

   def to_dict(self):
      return {
         "id": self.id,
         "username": self.username,
         "flashcards": [f.to_dict() for f in self.flashcards]
      }
```

3.2.4 Database Migration - Flask-Migrate

The Flask-Migrate extension expands the Flask-SQLAlchemy application with database migration features using *alembic*, a lightweight database migration tool for SQLAlchemy [21]. Database migration refers to controlled sets of changes used to modify the structure of the objects in a relational database. Migrations help transition database schemas from their current state to a new desired state, whether that involves adding tables and columns, removing elements, splitting fields, or changing types and constraints [22].

Flask Migrate is used in conjunction with the *Manager* from the *Flask-Script* library to create an external script that can be run in the python shell to handle migrations

in the database in a well organized way [23] . A minimal implementation of a migration script for the flask app is shown in listing 3.5

Code listing 3.5: Implementation of Database Migration for SQLAlchemy with Flask-Migrate and Flask-Script

```
1
2  from flask_script import Manager
3  from flask_migrate import Migrate, MigrateCommand
4
5  migrate = Migrate(app, db)
6
7  manager = Manager(app)
8  manager.add_command('db', MigrateCommand)
9
10 if __name__ == '__main__':
11    manager.run()
```

3.2.5 Tokens - Flask-JWT-Extended

Flask-JWT-Extended provides support for using JSON Web Tokens (JWT) in the flask application. JWT is a compact and self-contained way for transmitting information between parties as JSON objects securely. [24] JWT is used for authorization. Access tokens are generated on the server-side from each unique user id and are sent and stored in local storage on the client-side. Each JSON request from the client-side will include the access token in its header, allowing the server to verify the user credentials of each request. Access tokens have a short lifecycle, and in Flask-JWT-Extended they have an expiration date of 15 minutes by default [25]. To remedy this, refresh tokens are utilized, a type of token that can be used to retrieve new access tokens.

Flask-JWT-Extended allows for a a simple and robust implementation of JSON Web Tokens. The `create_access_token(user_identification)` method creates a unique JWT whilst the python decorator `@jwt_required` and the `get_jwt_identity()` function allows for route protection and user identification.

3.2.6 Other Libraries

Flask-CORS

Flask-CORS is a Flask extension for handling Cross-Origin Resource Sharing [18]. Flask-CORS allows the server to handle requests from different urls. CORS is useful in the application if the API is to be accessed externally, running cross-origin requests, which is mainly used in the development environment for this specific application, running the server and client on different localhost ports. CORS are easily enabled for the Flask application with the following code, shown in listing 3.6

Code listing 3.6: Enabling CORS for Flask application

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app)
```

Requests

Requests is a simple Python library for HTTP. It allows the server application to send HTTP requests, which in this application is used to handle external API calls to the FEIDE login API, as shown in listing 3.7

Code listing 3.7: Request `GET` example

```
import requests

feide_token = requests.get(
   "https://www.itk.ntnu.no/api/feide_token.php?apiKey="+api_key)
```

3.3 Javascript with React

For implementing the client-side of the application, the JavaScript library *React* was chosen. React is a JavaScript library for building user interfaces developed by Facebook. As an alternative to vanilla JavaScript, React allows the creation of reusable entities named components which can be rendered to the Document Object Model (DOM), the programming interface for HTML document in the browser [26].

React is used and installed from NodeJS, with NodeJS's default package manager, *npm*, explained in section 3.3.1. NodeJS is a runtime environment that executes JavaScript code outside the web browser, which also allows for running, compiling and optimizing the react applications to pure JavaScript, HTML and CSS code.

React version 17.0.1 was used, supporting *react hooks*, a new feature added in version 16.8 allowing us to create react components as simple JavaScript functions [27] using *hooks*, special functions allowing the creation of local states in JavaScript function components.

The two most commonly used hooks are `useState` and `useReducer`, the former being used to control the state and the latter to control the side effects of the component. The `useState` hook takes one argument, the initial state, and returns two values, the current state and a function to update it. The useEffect hook takes a function as an argument, referred to as the *effect*, and an array of dependencies as an optional second argument. Every time one of the dependencies change, the function, or effect, is executed.

A simple example of a react component named UserInfo, created using both hooks described above, is shown in listing 3.8. Here the useEffect hook is used to trigger a fetch request to an external API and update the UserInfo state every time the `id` state is changed. The return value of the function component is Javascript XML (JSX), a modified version of HTML that allows us to write HTML elements in JavaScript and place them directly in the DOM. [28]

Code listing 3.8: Simple React Hooks Example

```
1  import React, { useEffect, useState } from "react";
2
3  const UserInfo = ({ style }) => {
4    const [id, setId] = useState(null);
5    const [userInfo, setUserInfo] = useState({});
6
7    useEffect(() => {
8      if (id) {
9        fetch("https://jsonplaceholder.typicode.com/users/" + id)
10         .then((response) => response.json())
11         .then((json) => {
```

```
12          setUserInfo(json);
13        });
14    }
15  }, [id]);
16
17  const setRandomId = () => {
18    let newRandomId = Math.floor(Math.random() * 10) + 1;
19    setId(newRandomId);
20  };
21
22  return ( // jsx
23    <div style={style}>
24      <h1>userinfo: </h1>
25      <pre>{JSON.stringify(userInfo, null, 2)}</pre>
26      <button onClick={setRandomId}>Generate random id</button>
27      <div>Current id: {id ? id : "undefined"} </div>
28    </div>
29  );
30};
```

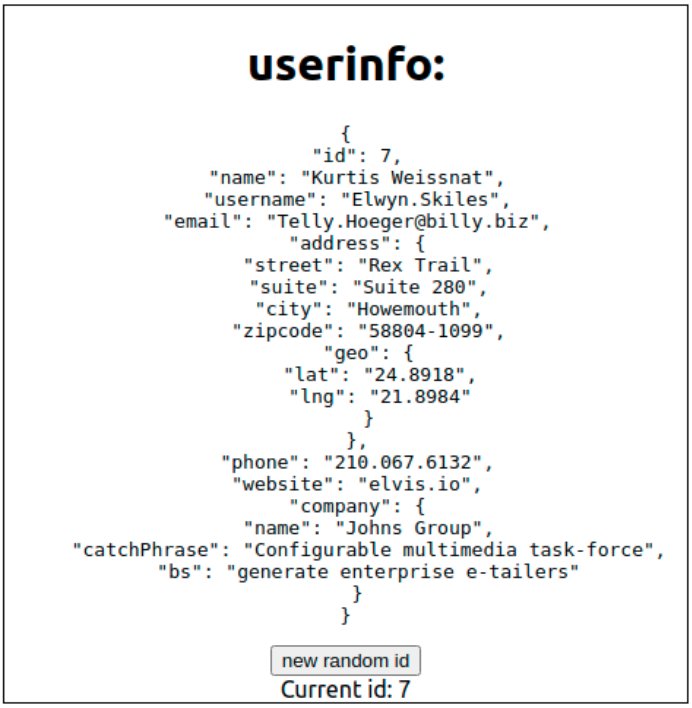


Figure 3.2: Rendered HTML from the implemented UserInfo component

The id state is changed when the setRandomId() function is executed, on every click of the *Generate random id* button. The component, rendered to the DOM is shown in figure 3.2. React components are rendered from the reactDOM.render() function of the react application by creating component tags. In this example, a `<UserInfo style= border: "1px solid black" />` tag is added to the react-

DOM.render() function to render the UserInfo component.

3.3.1 Package manager - npm

Node Package Manager (npm) is a package manager for JavaScript, and the default package manager for NodeJS. A `package.json` file in the project directory allows for the correct version of all dependencies of a project to be installed. New packages can simply be installed with the `npm install package_name` command from the terminal with NodeJS [29]. The package.json file for this application is shown in listing 3.9

Code listing 3.9: package.json

```
"dependencies": {
  "@date-io/date-fns": "^1.3.13",
  "@material-ui/core": "^4.11.2",
  "@material-ui/data-grid": "^4.0.0-alpha.20",
  "@material-ui/icons": "^4.11.2",
  "@material-ui/lab": "^4.0.0-alpha.57",
  "@material-ui/pickers": "^3.2.10",
  "@testing-library/jest-dom": "^5.11.9",
  "@testing-library/react": "^11.2.3",
  "@testing-library/user-event": "^12.6.0",
  "axios": "^0.21.1",
  "date-fns": "^2.0.0-beta.5",
  "eslint": "^7.24.0",
  "react": "^17.0.1",
  "react-device-detect": "^1.17.0",
  "react-dom": "^17.0.1",
  "react-markdown": "^5.0.3",
  "react-redux": "^7.2.2",
  "react-router-dom": "^5.2.0",
  "react-scripts": "4.0.1",
  "redux": "^4.0.5",
  "redux-thunk": "^2.3.0",
  "web-vitals": "^0.2.4"
},
```

3.3.2 Internal state management - Redux

While React handles the rendering of components to the DOM and their local component states, React is not very suitable for managing the state of the application as a whole. As an alternative to React handling states between components as props in *Parent-Child* relationships, which would quickly become a mess, Redux allows the creation of a global *store* in the application for managing the application state. The *store* is accessible across the entire application from all components, making the application state centralized, resulting in cleaner code while also making the architecture of the application more flexible, modular, and maintainable. [30]

The *Redux store*, which contains the application's global state, should never be modified directly, but is updated by *reducer* functions dispatched by *actions*. Actions, which typically sends fetched data from the UI, are triggered from the React UI components, which are rendered based on the extracted state from the Redux store. This one-way data flow of the Redux setup is illustrated in figure 3.3

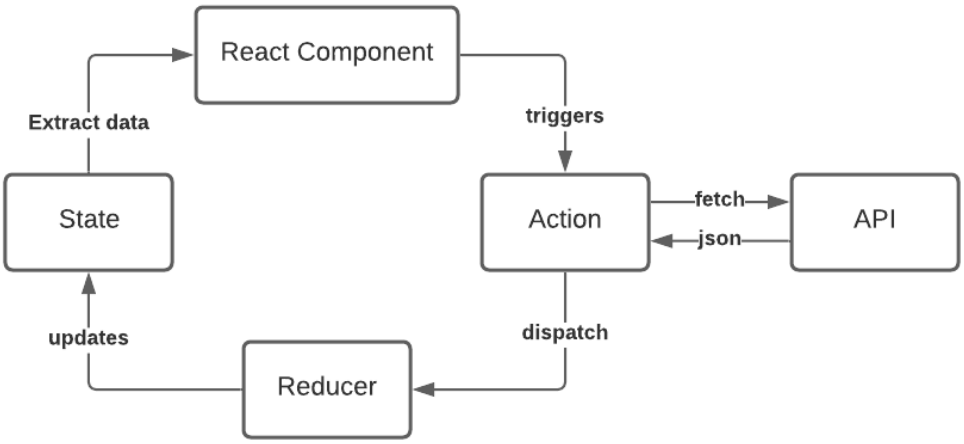


Figure 3.3: Redux data flow

Using the *Thunk* middleware, officially supported by Redux, allows us to write middleware functions with async logic that can dispatch actions to the store [30]. This is necessary for handling asynchronous actions, such as API requests. An example of a *thunk* function, expanding the UserInfo example from listing 3.8 with redux, is shown in listing 3.10. The data is fetched and an action, consisting of a type and a payload, is dispatched to be received by a reducer.

Code listing 3.10: Redux action example

```
export const fetchUser = (id) => async (dispatch) => {
  await fetch("https://jsonplaceholder.typicode.com/users/" + id)
    .then((response) => response.json())
    .then((json) => {
      dispatch({ type: "LOAD_USER", payload: json });
    });
};
```

The *reducer* takes the current state and receives actions to return a new, modified state. With Redux, the application state is always kept in plain JavaScript objects. To modify the state in the *UserInfo* example, an example of a reducer with an initial state is shown in listing 3.11. The received action's type is handled by the switch statement, updating the application state based on the payload received.

Code listing 3.11: Redux reducer example

```
const initState = {
  user: "undefined",
};

const userReducer = (state = initState, action) => {
  switch (action.type) {
    case "LOAD_USER":
      return {
        user: action.payload,
      };
    default:
      return state;
  }
};

export default userReducer;
```

Updated code from the *UserInfo* component from listing 3.8 is shown in listing 3.12. As illustrated, the API calls and state handling is abstracted away from the UI Component itself, and handled in the redux store. The `fetchUser()` function is dispatched from the component using the `useDispatch()` hook, whilst the state of the user is read with the `useSelector()` hook, which allows data to be extracted from the Redux store. A second `useEffect()` hook is used to handle the change of the userInfo state.

Code listing 3.12: Updated function component with Redux

```
  const [id, setId] = useState(null);
  const [userInfo, setUserInfo] = useState({});

  const user = useSelector((state) => state.userReducer.user);

  const dispatch = useDispatch();
  useEffect(() => {
    if (id) {
      dispatch(fetchUser(id));
    }
  }, [id]);

  useEffect(() => {
    setUserInfo(user);
  }, [user]);
```

This basic setup is used across the application, scaled to handle hundreds of API requests and allows for the applications state to be accessible from all components. Due to the size and complexity of the application, multiple reducers are created separate files, each managing independent parts of the state. Multiple reducers

are combined into one *rootReducer*, with the `combineReducer()` function from the Redux API [30]

3.3.3 User Interface - Material UI

As the project is time-restricted and building a unique design was not a main priority, designing and building UI components from scratch was not an option. As the focus on the user interface is to create a functional intuitive experience, not a visually unique solution, a UI Component library was used. The library *Material UI* was chosen due to its flexibility and comprehensive documentation. Material UI is a React Framework that implements Google's Material Design, a design system based on the physical world with features such as shadows, visual feedback and animations, providing great usability and versatility [31].

Material UI implements Material Design as React components that can be styled, modified and combined in the React application. Material UI contains components such as buttons, grid, input, cards and tables. The library also includes a react hook function, `makeStyles()`, which is used for creating themes and styles for the application components.

3.3.4 Other Libraries

Axios

As an alternative to using the built in `fetch()` method, demonstrated in the example in listing 3.8, the third-party library *axios* is used to handle HTTP request to the API. Axios has wider browser support than fetch, supports features such as request cancellation and automatic transformation of JSON data [32]. Since all data of our REST API is transferred using JSON, Axios is the obvious choice for handling requests, resulting in simplified and cleaner code.

React-Markdown

For supporting user input in Markdown for flashcard, supporting the creation of formatted text in the flashcards, the npm library *React-Markdown* was used. Unlike other Markdown libraries for React, *React-Markdown* does not rely on `dangerouslySetInnerHTML` for setting the inner HTML of components, preventing *Cross Site Scripting Attacks* [33], which are more detailed in section 4.1.1

React-Router-Dom

As React is only concerned with state management and rendering the state to the DOM, the application requires an additional library for routing, *React-Router-Dom*. This library gives the application the capacity to show different pages on different urls. [34]

Chapter 4

Requirements specification

In order to understand the demand and specifications of the system before implementation, an overview of requirements was made, prioritized based on their necessity in the system and the time constraints of the development of the system.

4.1 Functional Requirements

The following is an overview and discussion of functional requirements for different parts of the application. The functional requirements describe what the system **should do**, specifying the behavior and functionality of the system. [35]

4.1.1 General and Home Page

The system requires a secure method for user login. As all the users of the system are students at NTNU and FEIDE is the solution for secure identification for *Innsida* and other NTNU services, FEIDE was chosen as a login solution for the application (FR1) [36]. A distinction must be made between the regular users of the system, and the course staff overseeing the use of the system, hence separate admin and user roles must be implemented (FR2).

To make the page welcoming and user-friendly, the user should be presented with an *about* section on the home page after login. This *about* page will give the user information about the use and functionality of the system, hopefully removing confusion and minimizing questions from students (FR3). The home page should also have functionality for submission of anonymous feedback on the system, to receive bug reports and suggestions for improvements (FR4).

As an added bonus, there could exist a *user mode* for admins to view the web site as a regular user (FR5).

The *General and Home Page* functional requirements are summarized in table 4.1 below.

ID	Priority	Description
FR1	High	The user should be able to log in through feide
FR2	High	The page should have separate admin and user privileges
FR3	Medium	The home page should provide the user with a adequate text containing information and suggestions for using the system.
FR4	Medium	The user should be able to submit anonymous feedback, bugs and suggestions from the home page
FR5	Low	The admin should be able to view the page in a *user mode*, removing all admin functionality temporary until disabled.

Table 4.1: Caption

4.1.2 Flashcards and Flashcard Groups

After a chapter or part of the syllabus is finished in a subject, the students will be able to create flashcards covering the material. The admin users of the system will be able to create flashcard groups corresponding to the part of the syllabus, giving it a name, due date, and a set number of flashcards for each user to submit (FR6). These attributes should be possible to edit and delete later, as mistakes can be made by the admin users, and due dates could be useful to postpone (FR7). The same logic applies to the user-submitted flashcards, which should be able to be created, deleted, and edited before the due date has passed (FR8). To get an idea of the usability and function of the flashcard, the user should be able to preview the flashcard that was created as if the user was studying it (FR9). This *preview flashcard* feature is especially useful if the text of the flashcard is formattable using a markup language, such as HTML, Markdown, or LaTeX, that also supports images, lists and other features, allowing for more engaging flashcards (FR10).

ID	Priority	Description
FR6	High	An admin user should be able to create a flashcard group, giving it a name, due date and a set number of flashcards for each user to submit
FR7	High	An admin user should be able to edit and delete the flashcard group
FR8	High	The user should be able to create, delete and edit flashcards in the flashcard group before the due date of the flashcard group is passed
FR9	High	The user should be able to preview the flashcard
FR10	Medium	The User should be able to format the text of the flashcard, add images and lists using a markup language.

Table 4.2: Caption

4.1.3 Peer Review

To filter out the best flashcards from the user-submitted ones, and discard flashcards that are too similar, a flashcard rating process is necessary. As explored in section 2.3, the use of *peer reviews*, letting students rate each others flashcards, provide both a learning benefit to the students and minimal workload to the student assistants. This peer review system of the application will be automated as much as possible.

Peer review sessions for each student will be created by admin users, choosing a flashcard group, number of flashcards for each student to rate, and a due date for submission (FR11). These sessions should also be able to be edited and deleted by the admin user, in case of the need to postpone deadlines or make other changes. (FR12).

The peer review process consists of a user being presented with a random set of flashcards to rate on difficulty and quality. The quality rating is a rating of how relevant the flashcard is to the course curriculum and the overall quality of the flashcard as a study tool. The difficulty rating is useful for letting the user decide the difficulty range of the flashcards when studying them. In order to get an overview over which flashcards are too similar to each other, a *mark as duplicate* functionality is necessary, making it possible to select one or more flashcards. (FR13). To remove confusion and increase the ease of use of this feature, the process should be bidirectional, meaning that if one flashcard is selected to be similar to another, they are both marked. (FR14). As this application is meant to be used to evaluate students based on the quality of their submitted flashcards, all flashcards should receive approximately the same amount of ratings, meaning the *random selection*, must be weighed (FR15).

ID	Priority	Description
FR11	High	An admin user should be able to create peer review sessions for students, choosing a flashcard group which due date has passed, choosing number of flashcards for each student to rate and a due date for submitting the peer review
FR12	High	An admin user should be able to edit and delete the peer review
FR13	High	The user should be presented with a random set of flashcards to review, rating each flashcards' difficulty and quality as well as marking similar flashcards as duplicates
FR14	High	Marking flashcards as duplicate should be bidirectional
FR15	Medium	All flashcards should receive approximately the same amount of ratings, making the *random selection* weighed

Table 4.3: Caption

4.1.4 Flashcard Study

Due to the relatively short period between the completion of the system and the end of the semester, the spaced repetition flashcard review process was deprioritized in favor of an alternative solution. As spaced repetition spaces the introduction and review of flashcards over weeks or months, depending on the number of flashcards and the algorithm implemented, it was not a useful feature for the short usage time of the application. As an alternative, studying a random selection of flashcards that have passed the peer review process was prioritized to be implemented (FR16). Using a form, the user can choose difficulty range, flashcard groups, and how many flashcards to study (FR17). Based on the concept of spaced repetition, incorrectly answered flashcards should be re-asked, but correctly answered flashcards discarded (FR18). After each review process, a summary of the users' study performance should be presented (FR20). The lower priority features of flashcard study, related to spaced repetition, were not implemented (FR20, FR21, FR22), but added as *futher work* in section 7.2

ID	Priority	Description
FR16	High	The user should be able to a random selection of flashcards that have passed the peer review process.
FR17	High	The user should be able to choose a difficulty range and which flashcard groups to select n flashcards from for the random review
FR18	Medium	The random flashcard review should discard flashcards that are successfully answered and repeat flashcards that are failed
FR19	Medium	The user should receive a detailed summary and statistics of their flashcard review performance
FR20	Low	The user should be able to review the flashcard in a spaced repetition manner
FR21	Low	The spaced repetition review should be customizable for the user, i.e. review intervals and being able to discard flashcards from their spaced repetition flashcard set
FR22	Low	The user should be able to add custom flashcards to their own spaced repeition set, bypassing the peer review progress.

Table 4.4: Caption

4.1.5 Admin Page

The main function of the admin page is to provide an overview of the users on the application and their activity. The necessary data to present is a list of all users, the delivery status of the flashcard groups and peer reviews, as well as a list of all flashcards and their ratings from the peer review (FR23). To make the admin page user-friendly and make it easy to look up specific data, all lists should be filterable and searchable (FFR24). Further details on the admin page requirements are detailed in table 4.5 below, allowing for a full overview and necessary information presented to the administrator users.

ID	Priority	Description
FR23	High	The admin page should contain lists of all users, delivery status for all flashcard groups, delivery status for peer reviews and a list of all flashcards and their ratings.
FR24	High	All lists should be filterable and searchable
FR25	High	User list should contain a list of all users, their credentials and role
FR26	High	The admin should be able to promote a user to admin status from the user list
FR27	High	The *flashcard delivery status* page should contain the delivery status of each user for each flashcard group
FR28	High	The *all flashcards* page should contain all flashcards of all users from each flashcard group, their ratings and duplicates
FR29	Medium	Clicking a flashcard from the *all flashcards* page should give a preview of the flashcard as well as a list of all ratings associated with the flashcard
FR30	High	The *all flashcards* page should have a functionality for automatically filtering out the highest quality flashcards, removing duplicates, based on their ratings, so they can be added to the *collective deck* to be used by students for studying.
FR31	High	The *peer review delivery status* page should contain the delivery status of each user's peer review submission for each cardgroup

Table 4.5: Caption

4.2 Non-functional requirements

Non-functional requirements describe not how the system functions, but how it performs it functions and the constraints on the functions in the system as a whole. [35]. The following non-functional requirements were decided. The non-functional requirements for this application mainly revolve around usability and security.

Because this project is most likely to be handed over to future students for development, the maintainability of the project is highly prioritized (nFR3). This is reflected both in the choice of technology used for implementation and the implementation and code itself.

Because the application is to be used to collect data and evaluation from students, security is a high priority. Both in the secure transfer between the server and the client (nFR4), but also from cross-site scripting (XSS) attacks (nRF5), which can be a vulnerability if the user data is carelessly rendered to the browser by the application, allowing users to inject client-side scripts [37].

The main priority of the user interface should be simplicity (nFR1) and intuitively (nFR2). The web application should be usable on mobile screens and tablets, but the main priority will be development for a desktop browser (nFR6).

ID	Priority	Description
nFR1	High	The system should have a minimalistic interface consisting of simple elements
nFR2	High	The interface of the system should be responsive, inviting the user to intuitive actions
nFR3	High	The system should have a high degree of maintainability
nFR4	High	The system should have secure transfer between the server and the client
nFR5	High	The system should be secure from Cross-Site Scripting attacks
nFR6	Low	The interface of the system should be responsive for adapting to mobile screens and tablets.

Table 4.6: Caption

Chapter 5

Implementation and Application Architecture

In this chapter the implementation and architecture of the server and client applications for the system will be presented, satisfying the application requirements set in chapter 4

5.1 Server

The server, which is built as a REST API in Python with Flask, contains all the logic for manipulating and accessing data in the systems client-server relationship. The independent server application is connected to a MySQL relational database using SQLAlchemy and uses JWT Tokens for secure, authenticated transmission of data, as explained in chapter 3.

5.1.1 Structure and modularization

Using the Blueprint feature of Flask, discussed in section 3.2.1, the different parts of the server application were modularized into separate folders. In each module folder, separate files for the module routes and the module logic were created. For example, the *cardgroup.py* file in the *cardgroup* module contains the Cardgroup class, which is correlated to a `cardgroup` table in the SQL database using SQLAlchemy, as discussed in section 3.2.3, and functions and methods for interacting with this class. This includes functions for creating class instances, editing, deleting, and modifying the class as well as other logic required by the module, for example for calculating the average rating of flashcards.

The `routes.py` file of each module is only concerned with handling the HTTP routes of the REST API. Each module's `routes.py` file contains a blueprint for that module and a set of API route functions that are triggered by HTTP calls to specified routes as explained in chapter 3.1.1. These route functions utilize

the functions from the module logic file to manipulate the database and return the correct data for each HTTP request. Each module is imported into the main `app.py` file using the `app.register_blueprint(...)` function as explained in section 3.2.1. The HTTP routes for each module, their input parameters, and return values, are listed in appendix B.

This modularity and separation of concerns of the server application allows the creation of code with a high level of maintainability and readability, aiding in the fulfillment of the nFR3 maintainability requirement from section 4.2 for the server. The file structure, containing all the modules of the server application, was implemented as follows.

```
server
├── app.py
├── db.py
└── blueprints
    ├── cardgroup
    │   ├── cardgroup.py
    │   └── routes.py
    ├── cardrating
    │   ├── cardrating.py
    │   └── routes.py
    ├── collective_deck
    │   ├── collective_deck.py
    │   └── routes.py
    ├── flashcard
    │   ├── flashcard.py
    │   └── routes.py
    ├── peerreview
    │   ├── peerreview.py
    │   └── routes.py
    ├── user
    │   ├── user.py
    │   └── routes.py
    └── user_flashcard_deck
        ├── user_flashcard.py
        └── routes.py
```

5.1.2 SQL Database Tables and Entity Relationships

An *Entity Relationship Diagram* (ER diagram) was created to show the relationship between tables in the database, and their columns. By using the visual database design tool *mySQL Workbench*, built-in data modeling tools can be used to model an ER diagram of the database [38]. This generated ER diagram is shown in appendix A. A similar, but simplified, ER diagram created using *Lucidchart* is shown in figure 5.1. This simplified diagram models the pure entity relationship tables (association tables) with a more descriptive, unique syntax using rectangles. In addition, the structural constraints between classes are represented as numbers, (min,max), as an alternative to the less readable differently styled lines used by

the MySQL Workbench generated ER diagram. For example the (1,1) to (0,1) relationship between the User table and Flashcard table, illustrates that each flashcard can only be related to one user, whilst one user can be the creator of zero to multiple flashcards. Some columns, such as foreign keys, are omitted from the simplified diagram, as they are represented by the relationship lines in the model.

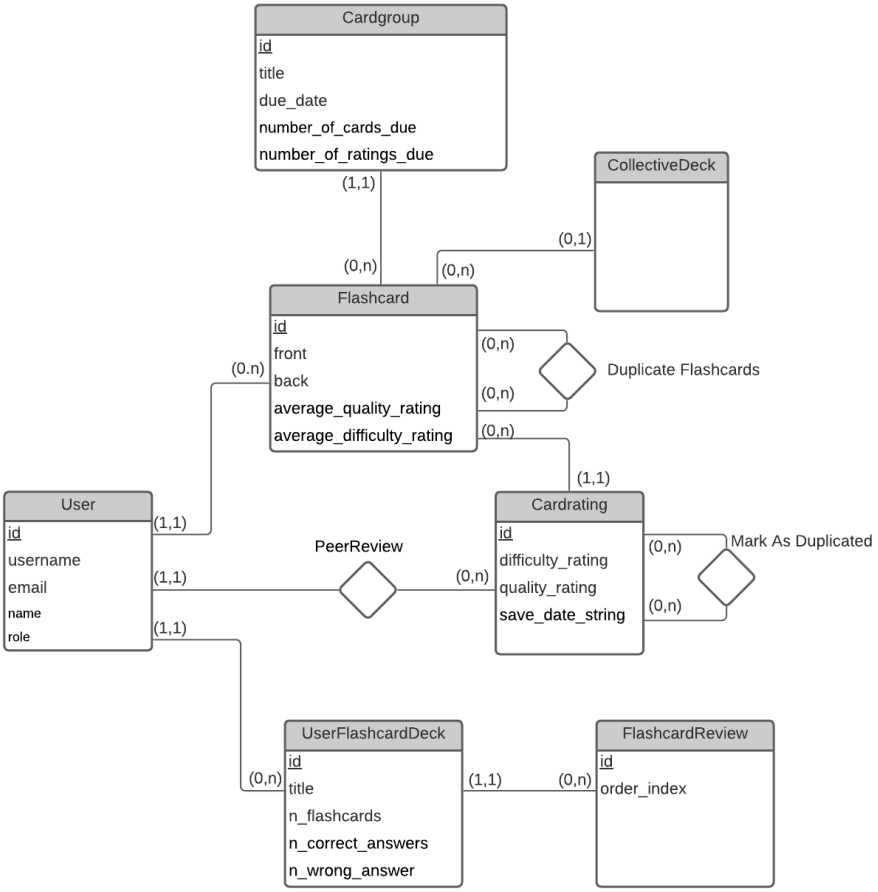


Figure 5.1: Application Architecture

As each module class is tied to a relational database class in SQLAlchemy, the ER diagram of the database of the server application highly correlates to the file structure tree in section 5.1.1, with a couple of exceptions. Despite the *PeerReview* class being a pure entity-relationship class, it was still given its own module and class in the system, as it contains a large amount of separate logic and functions for handling the peer review functionality. By contrast, the `duplicate_ratings` and `duplicate_flashcards` associative tables were implemented as Self-Referential Many-to-Many Association Tables in the *Cardrating* and *Flahcard* modules, respectively, to create a bidirectional sibling relationship. This type of implemented sibling relationships was based on the SQLAlchemy documentation [39].

In addition the *FlashcardReview* class was implemented as part of the *UserFlash-*

cardDeck module, as the *FlashcardReview* class did not require any separate URL routes or self contained logic, and is therefore not given its own module folder in the file structure.

5.1.3 JWT Tokens

To satisfy the non-functional requirement of a secure connection between the client and the server (nFR4), JSON Web Tokens implemented with Flask-JWT-Extended, as explained in section 3.2.5, were implemented in the server application to facilitate the secure transmission of data.

In addition to the built in `@jwt_required` wrapper, the `get_jwt_identity()` function allows for customized wrappers to protect routes and was used for checking admin privileges. This function is used to create the `@admin_only` decorator, shown in listing 5.1, protecting admin routes from being accessed by normal users.

Code listing 5.1: Decorator for admin routes with JWT

```
def admin_only(f):
   @wraps(f)
   def wrapper(*args, **kwds):
      uid = get_jwt_identity()
      user = get_user(uid)

      if user.is_admin():
         return f(*args, **kwds) # continue
      else:
         raise Exception("Not admin")
   return wrapper
```

To prevent tokens from being reused for authentication purposes, the refresh token and access tokens of a user are invalidated on logout. This is implemented by implementing a callback function in the `@jwt.token_in_blacklist_loader` wrapper, that checks if the token is invalid, as specified by the Flask-JWT-Extended documentation [40]. To implement this function, a database table for blacklisted tokens is created with SQLAlchemy as shown in listing 5.2. Blacklisted token are saved in the database on logout using the `save()` function, and the implemented `@jwt.token_in_blacklist_loader` wrapper callback function utilizes the `is_invalid()` function to check for validity.

Code listing 5.2: Invalid Token Database Table

```
class InvalidToken(db.Model):
   __tablename__ = "invalid_tokens"
   id = db.Column(db.Integer, primary_key=True)
   jti = db.Column(db.String(128))

```

```
6     def save(self):
7         db.session.add(self)
8         db.session.commit()
9
10    def is_invalid(self, jti):
11        q = self.query.filter_by(jti=jti).first()
12        return bool(q)
```

5.1.4 Users

Login

An external login API created by Åsmund Stavdahl, an engineer at the department of cybernetics at NTNU, was used to facilitate user login with FEIDE, as specified in requirement FR1 in section 4.1.1

The login solution works by simply redirecting the user to an external web service for login. To access this web page, a unique, one-time API token, which is generated by a separate API using a secret API KEY, is required. The external web service for FEIDE login posts a GET request to a requested API URL with the user information, as specified in table B.1 in appendix B. To secure the login GET request, a login token encrypted with a secret API key, the user information, and a unique one-time API token, using SHA-1, a cryptographic hash function, is included in the request. This token is validated on the server application to ensure the login is secure. Three API routes are used to securely login a user, below are a summary of each routes' task. Details about these routes can be found in table B.1 in appendix B.

- `/api/login/url`
    - get feide API key from external API using API KEY
    - Return valid login url to client
- `/api/login/userdata`
    - receive userdata and sha1 encrypted login token from login api
    - if token valid, store userdata in Flask Session
    - Return redirect to flashcard application
- `/api/login/callback`
    - check session for userdata
    - if user with userdata does not exist, register user
    - Return generated jwt access and refresh tokens

Roles

By default all users are given a ”User” role when signing in to the application. The ”Admin” role can either be designated by other admins, through the POST

and DELETE admin routes, detailed in appendix B, or through running the flask terminal of the application by running a `make_admin(user_id)` function from the server application, which is useful if no admins already exists on the system.

Admin users have all the privileges of normal users, with the additional permission of being able to make HTTP requests to admin protected routes in the API. All admin protected routes begin with `/api/admin/...` and are protected by the `@admin_only` wrapper explained in section 5.1.3

This separation of admin and user privileges satisfy the FR2 requirement from section 4.1.1

5.1.5 Flashcards and Flashcard Groups

Flashcards can be submitted by any user to a flashcard group. Flashcard groups can only be created by administrators, who specify the due date and the number of flashcards for each user to submit. Both the flashcards and flashcard groups exist as relational tables in the system and are related by foreign keys as shown in figure 5.1. The flashcards also have values for average quality rating and difficulty rating, which are automatically calculated on the server side after the peer review of the flashcard group has ended. The API routes of the server application to create, edit and delete flashcard groups. as well as create, edit and delete flashcards in the groups are detailed in table B.2 appendix B, fulfilling requirements FR6, FR7 and FR8 in section 4.1.2.

5.1.6 Peer Review and Card Ratings

After a flashcard groups' due date has elapsed, the admin is able to create peer review sessions for each student.

Each student is presented with a number of flashcards to rate on quality and difficulty. To ensure all flashcards receive approximately the same amount of ratings, an algorithm for picking out flashcards for all users was developed. The code for this algorithm is shown in listing 5.3

Code listing 5.3: Login example using sessions

```
# ensures all cards are picked the same amount of times
def pick_random_cards(cardids, number_of_users, ratings_per_student):

   users_cards = []
   for n in range(number_of_users):
      users_cards.append([]) # Create a 3d list of flashcards

   cards = cardids.copy() # shallow copy of cardid parameter
   random.shuffle(cards)
   for u in range(number_of_users):
      # not enough cards left to draw from ?
      if (len(cards) < ratings_per_student):
```

```
13
14          # draw all remaining cards
15          users_cards[u] += cards
16
17          # add new random cards
18          cards = cardids.copy()
19          random.shuffle(cards)
20
21          # Go through new random cards and find unique cards
22          for n in range(ratings_per_student - len(users_cards[u])):
23             for i, c in enumerate(cards):
24                if c not in users_cards[u]:
25                   users_cards[u].append(cards.pop(i))
26                   break
27
28       else: # draw first cards and remove them from list
29          users_cards[u] = cards[0:ratings_per_student]
30          cards = cards[ratings_per_student:]
31
32    return users_cards
```

Assuming all users complete their peer review, this algorithm ensures all flashcards receive approximately the same number of ratings, a deviation of one flashcard rating is expected, as the number of flashcards is rarely divisible by the number of students.

The algorithm simply returns a two-dimensional array, with one list of n flashcard ids for each student, where n is equal to the `ratings_per_student` parameter. The `cardids` parameter is a list of ids for all flashcards that belong to the flashcard group, which is shuffled and drawn to student lists. When there are too few cards left to draw, the remaining cards are drawn and the input `cardids` list is shuffled and drawn from again. To ensure a flashcard is not drawn multiple times, a duplicate check is done on line 24 of listing 5.3. This *weighed random selection* fulfills requirement FR15 of section 4.1.3.

The bidirectional *mark as duplicate* requirement, FR14, described in section 4.1.3 is fulfilled on the server side by a simple `add_duplicate()` function, shown in listing 5.4, that uses the *duplicate_rating* self-referential many-to-many association table to create a sibling relationship between the two ratings, as explained in section 5.1.2.

Code listing 5.4: Login example using sessions

```
1  def add_duplicate(rating, duplicate_rating):
2     rating.duplicates.append(duplicate_rating)
3     duplicate_rating.duplicates.append(rating)
```

The API routes of the server application to create, edit and delete peer reviews as well as rate flashcards in each peer review are detailed in table B.4 in appendix

B, which fulfill requirements FR11, FR12 and FR13 explained in section 4.1.2.

5.1.7 Collective Deck and User Flaskcard Decks

Flashcards can be added to and removed from the collective deck by administrator users using the API routes detailed in table B.6 in annexed B. After each peer review, the administrator has access to the average ratings and duplicate status of each flashcard in a flashcard group, giving the administrator the final responsibility to decide which flashcards pass the peer review process.

the `api/admin/cardgroup/<cgid>/flashcards` route detailed in table B.3 in appendix B, allows for filtering out flashcards below a certain rating and only showing the highest rated flashcard if duplicates exists. This provides a list of highly-rated, not duplicate flashcards for the admin to add to the collective deck, but the admin will still have the flexibility of being able to manually add and remove flashcards. Details about the client-side functionality of this process is explained in section 5.2.8.

As an alternative to spaced repetition, which is a lower priority requirement due to time constraints, as explained in section 4.1.4, the collective deck is studied through *user flashcard decks*, which are created by the user, choosing difficulty range of flashcards, flashcard groups to pick cards from and a name for their flashcard deck. Users receive a selection of random flashcards to study, and submit feedback after every flashcard study attempt on whether they are successful or not in answering the question. The number of correct and wrong answers are stored in a column in the `user_flashcard_deck` table created for each deck, allowing the user to receive feedback on their performance after each finished deck.

To allow for wrongly answered flashcards to be moved to the back of the flashcard deck, an `order_index` attribute of the `FlashcardReview` object associated with each flashcard in the user flashcard deck was used for ordering the flashcards in the deck. If a flashcard study attempt was failed, the `order_index` of that flashcard review is simply changed to be the largest one the deck.

The routes for studying flashcards from the collective deck, using *user flashcard decks*, are detailed in table B.7, fulfilling the FR16, FR17, FR18, FR19, and FR20 requirements for Flashcard Study, but not the lower prioritized requirements related to spaced repetition study.

5.2 Client

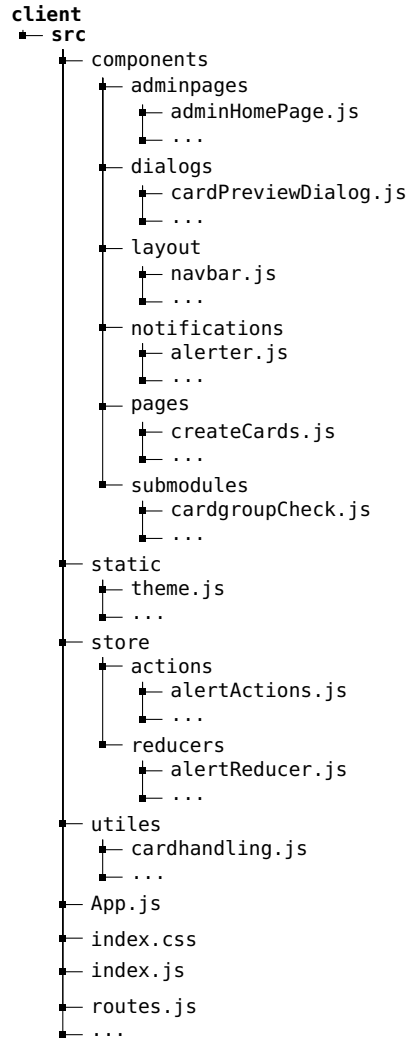
The client application, a single-page user interface for interacting with the REST API, is written in JavaScript using the library React. The client application interacts with the REST API through HTTP requests as explained in section 3.1.1. The design of the user interface was implemented using Material UI, as explained in section 3.3.3, with the goal of creating a minimalist, intuitive application.

5.2.1 Structure

A react-project was created using the `npx create-react-app` command in Node. With this command the foundation for a new single-page application in react is created with a `src` folder to place all components.

The file structure of this source folder is shown in the figure below. The *redux store* explained in section 3.3.2 is given a separate folder, containing the state of the applications, as well as actions and reducers to modify this state. The *static* folder contains files for themes and styling, while the *utilities* folder contains some simple, pure JavaScript functions that are shared between components.

The component folder contains all components that make up the application, separated into distinct folders related to their purpose. For example all components in the *page* folder are used for separate pages in the application, and utilize a common `PageWrapper` component from the static folder.



5.2.2 Design Patterns and Component Structure

A shared code style for all components was incorporated to ensure consistency and readability of the code, resulting in a higher degree of maintainability, nFR3 in table 4.2. A basic outline of this structure is shown in listing 5.5

Code listing 5.5: Component structure

```
1
2  ... // imports
3
4  const useStyles = makeStyles({
5     /* styles */
6     wrapper: {
7        .. // css
8     }
9    ...
10 });
11
12 const ExampleComponent = ({
13    ... // component props
14 }) => {
15    const classes = useStyles();
16
17    /* redux states */
18    const reduxState = useSelector(state => state.reducer.value)
19    ...
20
21    /* component states */
22    const [state, setState] = useState(initState)
23    ...
24
25    dispatch = useDispatch()
26    useEffect(() => {
27       dispatch(...) // dispatch redux actions
28    }. [])
29
30    /* component methods */
31    const handleChange = event => {
32       ... // change
33    }
34    ...
35
36    return (
37       <div className={classes.wrapper}>
38          // JSX for component
39       </div>
40    )
41 };
```

Most of the CSS styling of components is handled outside of the component itself,

and placed in a `useStyles` component using the `makeStyles` function from Material UI, explained in section 3.3.3. The CSS styling can be accessed by components using the `className` attribute as shown on line 37 of listing 5.5

Following the declaration of styling classes of the component, the Redux states are accessed using the `useSelector` hook explained in section 3.3.2. The component states follows, using the `setState` hook explained in section 3.3.

The Redux actions are dispatched using the `useDisptach` and `useEffect` hook, as explained in section 3.3.2.

Lastly, before the return JSX of the component, the component methods are written. For example for handling changes to the state.

5.2.3 Login and User Authentication

The login page is created as a simple React Component, with a Material UI button that when pressed, sends a GET request to the server for a URL to open an external FEIDE login page. The simple login page is shown in figure 5.2. *Forgotten password* for FEIDE is handled by an external page supplied by *idporten*.

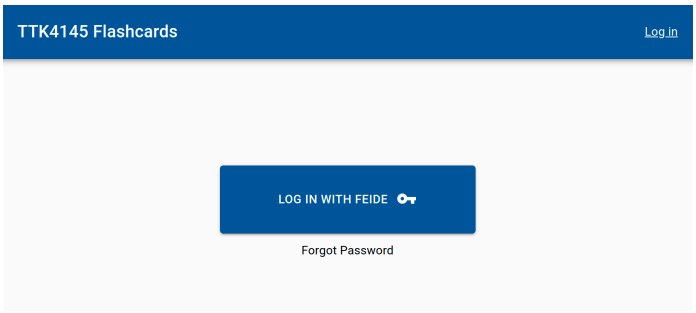


Figure 5.2: Login Page

The client side interacts the login functionality explained in section 5.1.4. The interaction between the client, server and FEIDE API to facilitate secure login of users is illustrated in figure 5.3

The unique user token and refresh tokens generated by the server application are stored locally in the users browser local storage. On logout they are deleted. This ensures the user is not logged out every time the browser is closed.

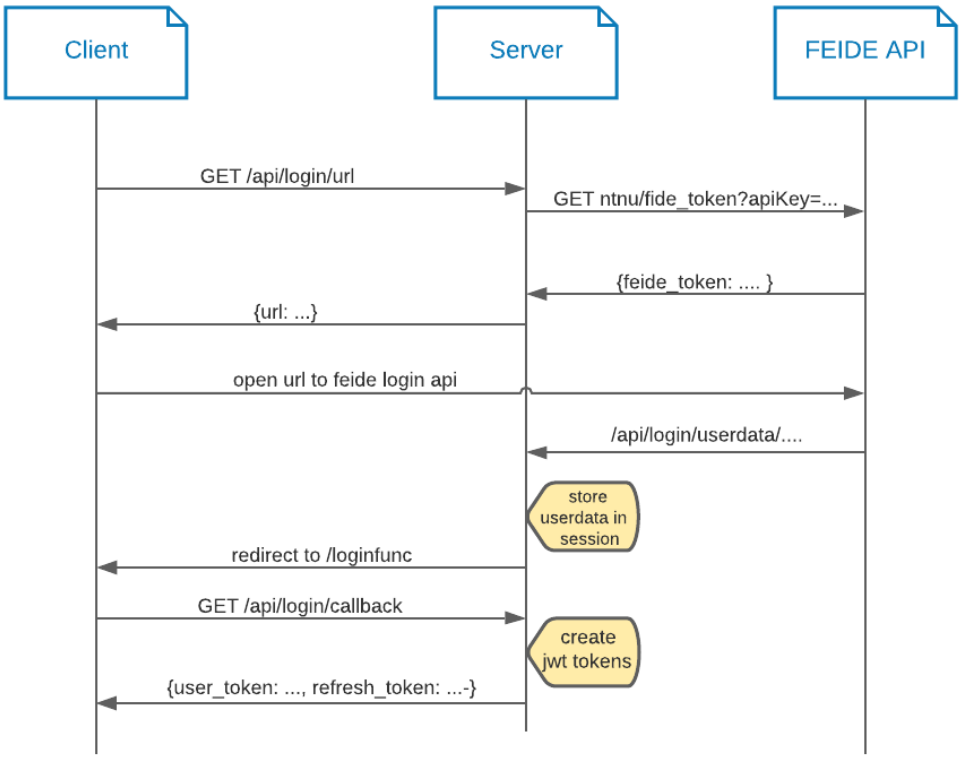


Figure 5.3: Client, Server, FEIDE API interaction for login

5.2.4 Home Page and Navigation

After login in, the user is redirected to the home page of the application, shown in figure 5.4. The home page contains text containing information about the application, its purpose and suggestions for using it, satisfying FR3 in section 4.1.1. FR4 is satisfied by adding a *SEND FEEDBACK* button to the home page, which opens a Google Form, a survey administration software [41] , allowing for anonymous feedback to be collected by users.

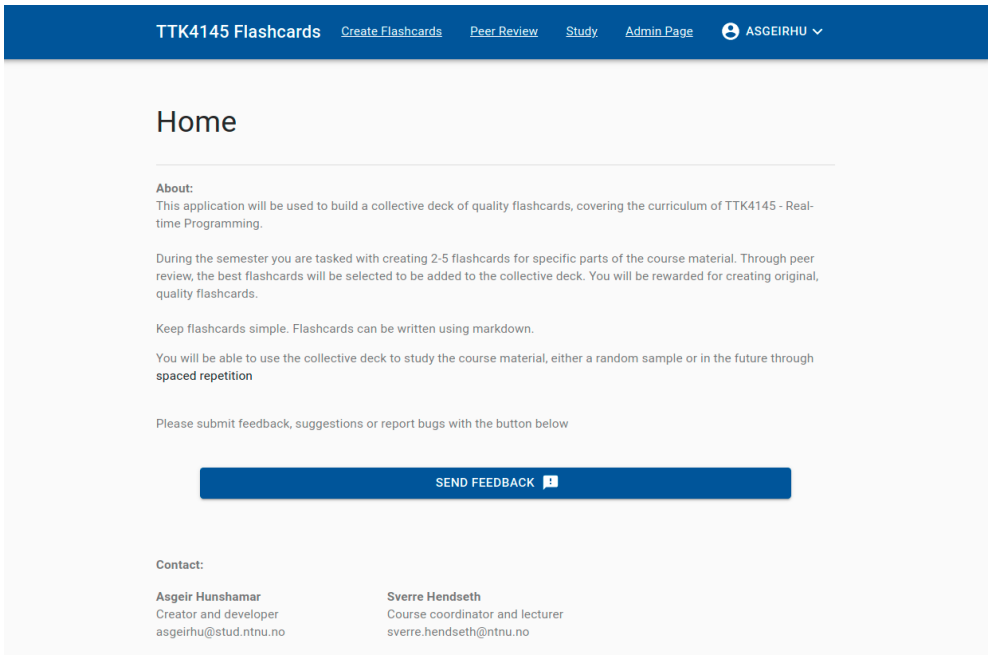


Figure 5.4: Caption

Overheading all pages of the application is a navigation bar, as shown in figure 5.4. This is a separate component which consists of a list of links to different pages of the application as well as information about the logged in user, and an related drop down menu, shown in figure 5.5. From this menu the user can access their profile, change theme of the page and log out. In addition, admin users are able to view the page in *User Mode* by toggling a button on the menu, satisfying requirement FR5 in section 4.1.1.

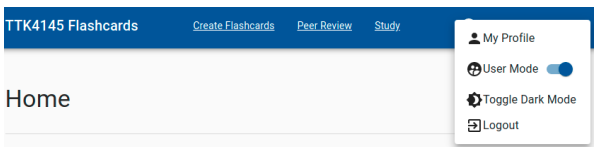


Figure 5.5: User drop down menu

5.2.5 Flascard Groups and Flashcard Creation

Flashcard Groups are created by administators using the form opened by clicking the *CREATE GROUP* button on the *Create Flashcard* page shown in figure 5.6, sending a POST request to the API as detailed in section 5.1.5. By clicking a flashcard group the user is redirected to a separate page for creating, editing and deleting flashcards.

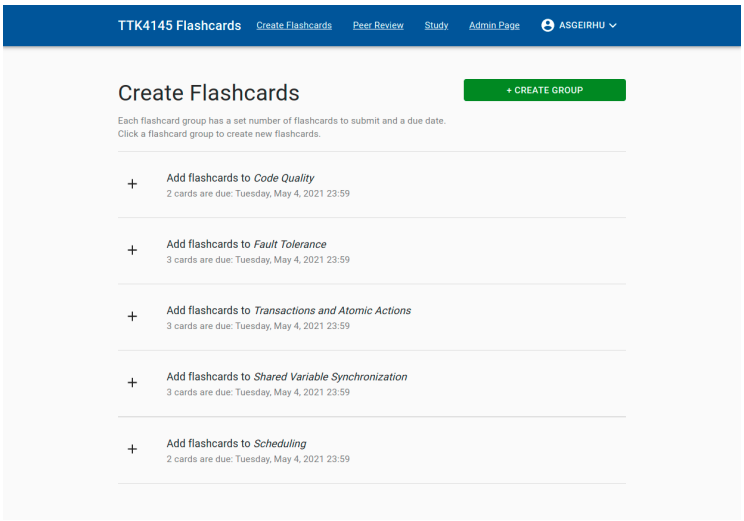


Figure 5.6: Caption

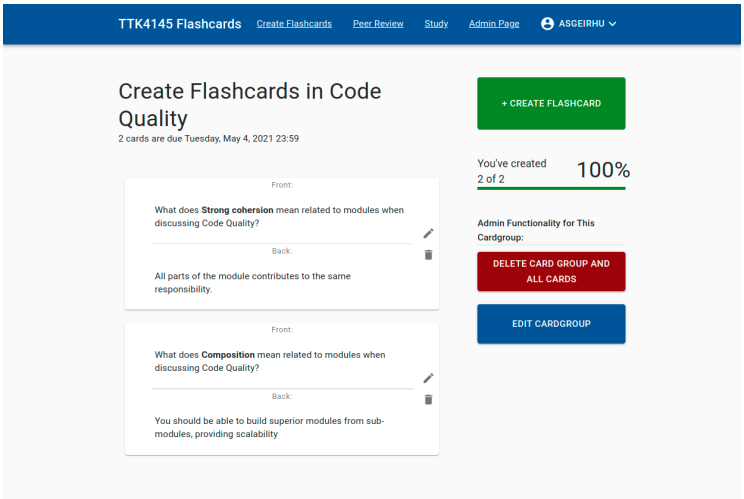


Figure 5.7: Caption

The separate page for flashcard actions is shown in figure 5.7. On this page buttons for editing and deleting the flashcard group appears for admin users. For normal users, preview of all their created flashcards are shown, with buttons for editing and deleting flashcards. The *+Create Flashcard* button opens a dialog with a form

to submit new flashcards.

The dialog for submitting new flashcards is shown in figure 5.8. Text fields for the front and back of the flashcard appear, and the user is able to preview the flashcards by clicking a *SHOW PREVIEW* button in the dialog, which is shown opened in figure 5.8, fulfilling requirement FR9 in section 4.1.2. In this, newest version of the application, Markdown is used for text formatting, fulfilling requirement FR10 in section 4.1.2. HTML was also tested. As discussed in section 3.3.4, the *React-Markdown* library was used to support Markdown formatting, a formatting library secure from cross site scripting attacks (nRF5). More details about the choice to use Markdown is discussed in section 6.2.

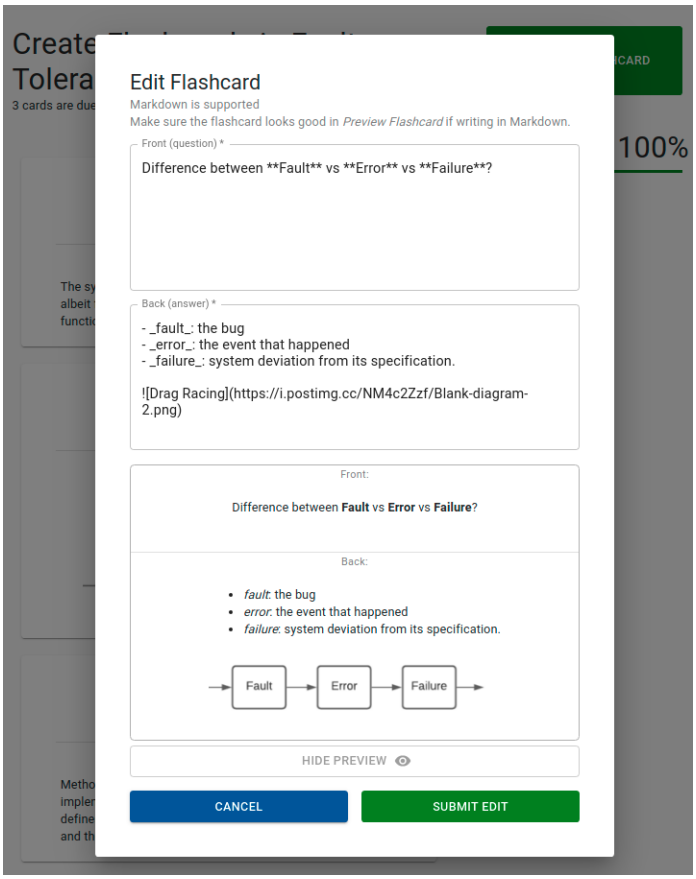


Figure 5.8: Caption

5.2.6 Peer Review of User Flashcards

Peer Review sessions are created by administrators using the form opened by clicking the *ADD PEER REVIEW SESSION* button on the *Peer Review* page shown in figure 5.9, sending a POST request to the API as detailed in section 5.1.6. Peer Review sessions have buttons for admins, for deleting and editing. By clicking a peer review session the user is redirected to a separate page for rating flashcards.

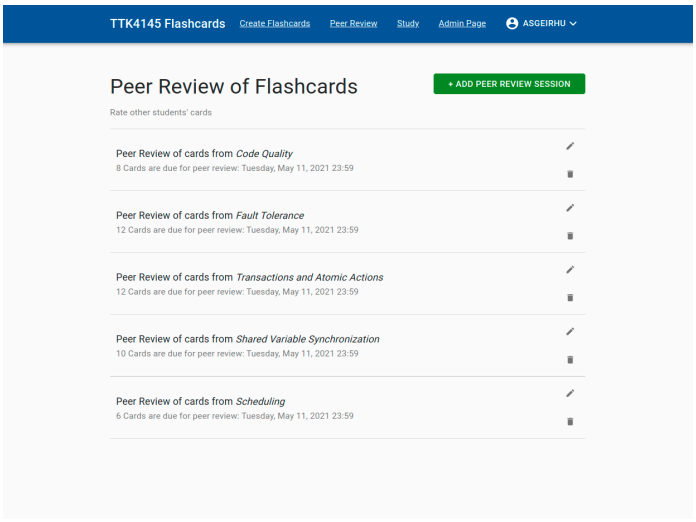


Figure 5.9: Caption

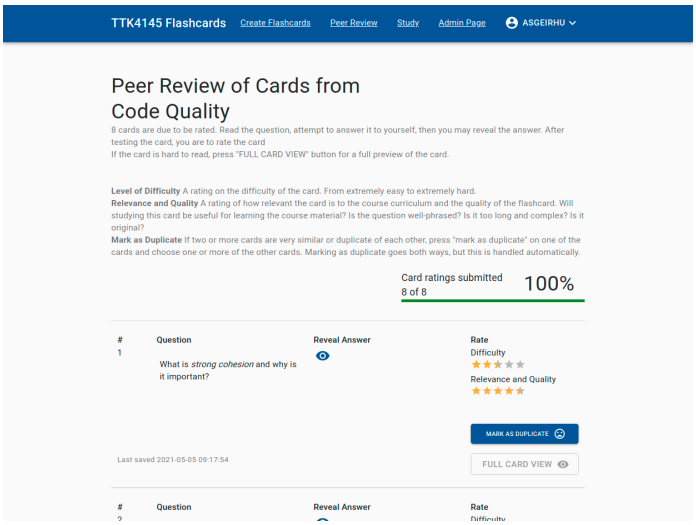


Figure 5.10: Caption

The separate page for rating flashcards is shown in figure 5.10. On this page, a random selection of flashcards is shown for rating. The details about which flashcards are shown to the user is explained in section 5.1.6. Pressing *Mark As*

Duplicated opens a dialog for checking flashcards on the page as duplicates.

5.2.7 Study - User Flashcard Decks

To allow students to study flashcards that have passed the peer review process, an alternative method for studying was implemented, discussed in section 4.1.4. Functionality for creating *user flashcard decks* was implemented, where students were able to study a random selection of flashcards from the *collective deck*. The page for user flashcard decks is shown in figure 5.11. The *+ NEW FLASHCARD DECK* button opens a form to create a new user deck of flashcards to study.

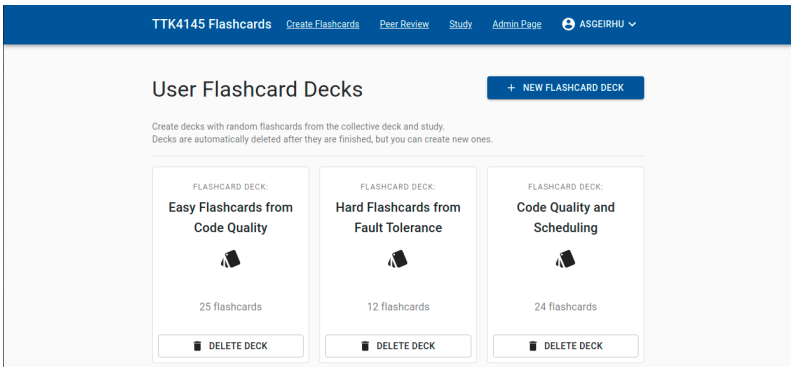


Figure 5.11: Caption

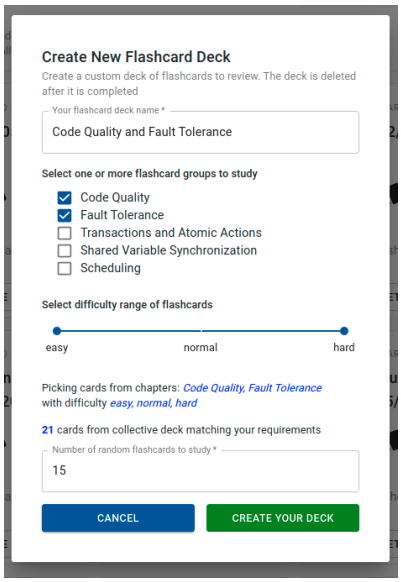


Figure 5.12: Caption

The dialog form to create a new user deck is shown in figure 5.12. Here users are able to choose which flashcard groups to study, difficulty of the flashcards and amount of flashcards to study.

The study process of flashcards is shown in figure 5.13. The user is presented with the flashcards in a random order, where only the front of the flashcard is initially presented. The user can check the back of the flashcard and is presented with options to retry the card, moving it to the end of the study sequence, or remove it, depending if they were successful or unsuccessful in correctly answering the question presented. At the end of the study sequence, the user is presented with how many correct and wrong answers they gave.

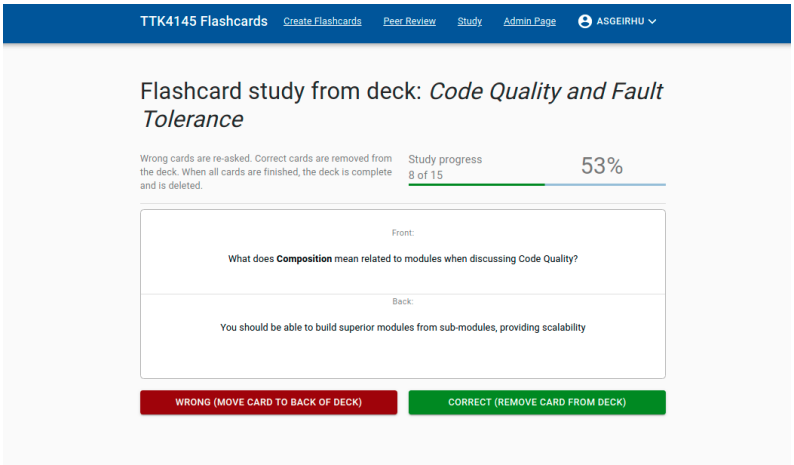


Figure 5.13: Caption

5.2.8 Admin page

An admin page was created to provide information about users, their delivery status, the peer review process, and all flashcards with their ratings, as discussed in section 4.1.5. The admin page utilizes many of the `/admin` routes presented in appendix B, to access information not accessible for normal users. The admin page fulfills all requirements in section 4.1.5. One of the pages, showing all flashcards and their ratings, is shown in figure 5.14. A toggle button was implemented that sends a GET request to the `/api/admin/cardgroups/<cgid>/flashcards` route detailed in table B.3 in appendix B. This route allows for filtering based on the minimum average rating of flashcards, as well as an option to remove duplicate flashcards, as explained in section 5.1.7. This functionality makes it simple for the administrator to filter out the highest quality flashcards, adding them to the collective deck for studying. By clicking a flashcard row, a dialog with a preview of the flashcard, as well as information about all ratings of the flashcard appears.

To organize the data on the admin page the *DataGrid* component of Material UI was used [42], allowing for client-side searching, filtering and ordering of columns, as well as pagination with dynamic row size.

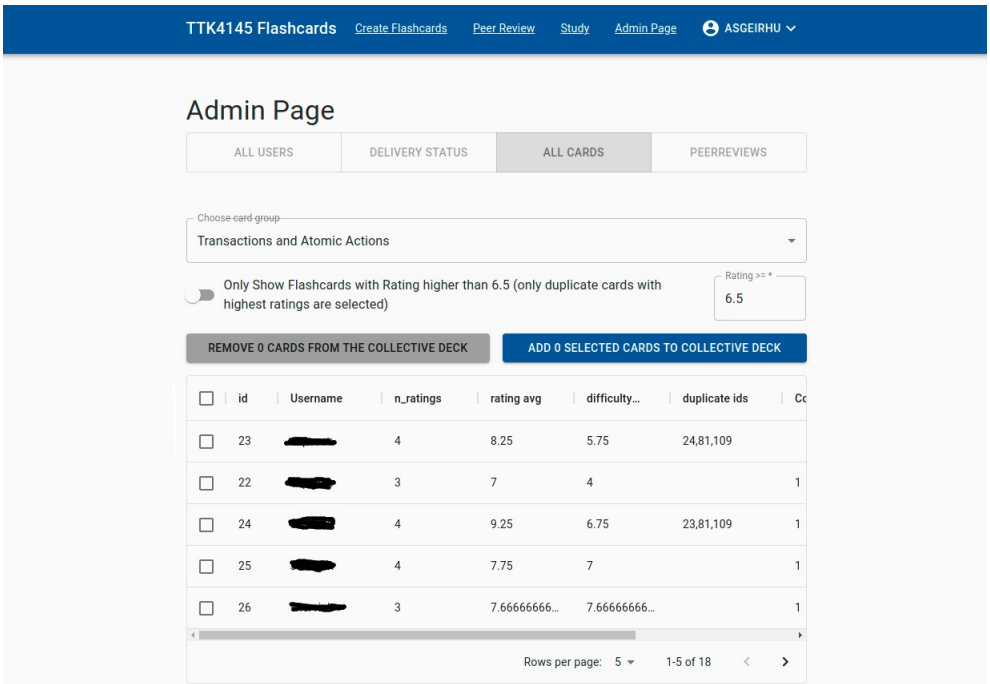


Figure 5.14: Caption

Chapter 6

Deployment and Beta Testing

6.1 Deployment With Heroku

Due to a very slow process of accessing a virtual server on NTNU to host the application, an alternative hosting service was used, named Heroku, running the Gunicorn web server and PostgreSQL.

6.1.1 Gunicorn

As an alternative to the built-in web server of Flask, the Python HTTP server Gunicorn was utilized. As the built-in web server of Flask only processes a single request at a time, resulting in an unresponsive and slow application, Gunicorn was chosen as it supports concurrently running multiple Python processes. [43]. Gunicorn is installed with `pip` and added to the `requirements.txt` file explained in section 3.2.2.

6.1.2 PostgreSQL

As Heroku natively uses PostgreSQL, this was used as an alliterative to MySQL when hosting on Heroku. As the python object-relational mapper SQLAlchemy was used, switching to an alternative SQL database management system required little configuration.

6.1.3 Temporary Domain

As the time constraints did not make it possible to host the system on NTNU's systems for beta testing an alternative, a temporary domain was acquired and connected to the Heroku page.

The web domain http://ttk4145flashcards.no was chosen and used during the beta test.

6.2 Closed Beta Test With Student Assistants

The choice of using Markdown for formatting text on flashcards was made after a short, closed beta test involving the student assistants and the course research assistant, only testing the *create flashcard* feature of the application. Initially, HTML was used as a markup language, but due to its lack of familiarity for the students and general more complicated syntax than Markdown, it was replaced after user feedback. Another benefit of Markdown is its readability and intuitively over HTML, for example line breaks are achieved by simply creating a new paragraph, whilst HTML requires the use of the less intuitive `<br/>` tag.

Markdown supports many of the same features as HTML, such as images, lists, and formatted text, but one drawback is that it does not support styling, for example for coloring text or resizing images. Adding support for both markup languages was considered, but as this could seem overwhelming for users and the choice was made to only support Markdown.

6.3 Open Beta Test With Students

With the completion of the application and having successfully hosted it on a server and conducted a closed beta test, the system was rolled out to be used by students in the Real-Time Programming course as a voluntary exercise. 234 students were asked to participate. A timeline of the beta test is shown below

- **April 21st 2021:** Create flashcards due Tuesday 4th of may
- **May 5th 2021:** Peer review flashcards due Tuesday 11th of may
- **May 12th 2021:** Study flashcards
- **May 19th 2021:** End of beta

6.3.1 Flashcard Creation

The following five flashcard groups for submission were created, all with the deadline of may the 4th. These flashcard groups were decided in collaboration with the course lecturer based on the structure of the course. The students were tasked with creating 2 or 3 flashcards for each flashcard group.

- Code Quality (2 flashcards)
- Fault Tolerance (3 flashcards)
- Transactions and Atomic Actions (3 flashcards)
- Shared Variable Synchronization (3 flashcards)
- Scheduling (2 flashcards)

By the due date, May the 4th, 73 flashcards had been submitted by 6 different students.

6.3.2 Peer Review

Following the flashcard creation process, the students were tasked with rating each other's flashcards. Based on the number of flashcards received in each flashcard group and the number of participating students, the following number of flashcards to rate per student for each flashcard group was decided.

- Code Quality (8 flashcard to rate)
- Fault Tolerance (12 flashcard to rate)
- Transactions and Atomic Actions (12 flashcard to rate)
- Shared Variable Synchronization (10 flashcard to rate)
- Scheduling (6 flashcard to rate)

247 flashcard ratings were collected from 10 different users, providing a sufficient number of ratings, with all flashcards receiving between 1 and 6 ratings. As not all users participated in the peer review process, the flashcards did not receive approximately the same amount of ratings. The algorithm for selecting which flashcards users are to rate did successfully hand out all flashcards approximately the same amount of times to be rated. But for each flashcard to receive an equal amount of ratings, the system depends on every user to complete the peer review process.

6.3.3 Flashcard Study

Based on the peer review process, the highest quality flashcards were selected to be added to the collective deck for studying. All flashcards with an average quality rating of 6.5 or above were selected. For duplicate flashcards, only the flashcard with the highest average rating was selected. The admin functionality for adding flashcards to the collective deck, explained in section 5.2.8, required very little effort, and yielded a set of highly rated, unique flashcards.

The studying of flashcards turned out to be an extremely popular feature of the application. While only 6 users participated in creating flashcards, and 10 users participated in the peer review process, 105 users used the *flashcard study* feature. Before the flashcard study feature of the application was rolled out, but students had been asked to participate in creating and rating each other flashcards, only 53 users had logged into the application. This number increased to 172 from the time the study feature was rolled out and leading up to the exam. A total of 6271 flashcard reviews were initiated in 251 user flashcard decks from 105 users, despite only 58 flashcards being available for studying.

The total flashcard reviews for each day leading up to the exam is shown in figure 6.1, with the most popular days being the launch of the study feature and the day before the exam.

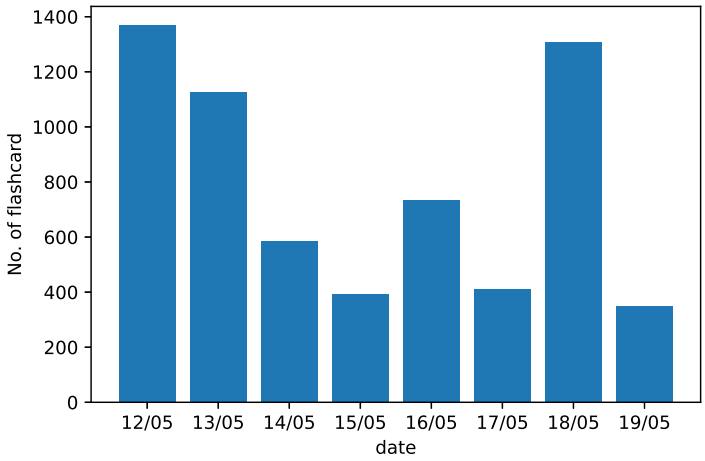


Figure 6.1: Number of Flashcard Reviews each day of the open beta test

6.4 User Survey

A voluntary user survey was sent out to all students after the beta test ended, with highly encouraging results. The survey consisted of a series of questions where the user would answer with a score of 1 to 5, from strongly disagree to strongly agree. 15 students answered the survey. The questions and results from the survey are shown in appendix D.

Whilst the flashcard creation process was not very popular as a voluntary exercise, the students who participated in creating flashcards found the creation of flashcards rewarding and educational and the creation process intuitive and simple to use. The users who did not create flashcards highly reported time constraints as the main issue, with a lot of other, mandatory exercises to hand in parallel to the beta test of this application.

The peer review process was also found to be intuitive and simple, as well as educational and rewarding for the students participating, based on the survey results. The users who did not contribute with peer reviews reported time constraints and lack of knowledge as the main issue.

The study process was also well received, with 38.5% of students rating the flashcard study process 4/5 and 61.5% rating the process 5/5 on intuitively and simplicity of use. Most students surveyed responded that studying flashcards in the application helped them prepare for the exam and found the flashcards studied had a high level of quality and relevance to the course material. The students surveyed agreed that they wished there were more flashcards to study, with all students responding between 3/5 and 5/5 on this question.

A majority of the students surveyed strongly agreed that the application user

friendly and intuitive, and a vast majority of 75% responded with *strongly agreee* on whether the web application was fast and responsive.

On the question of whether the use of the flashcard application to submit and rate flashcards should be a compulsory exercise in ttk4145 - Real-Time Programming, the answers ranged from *strongly disagree* to *strongly agree*, with the majority disagreeing with the suggestion. Based on the written feedback in the survey, it seemed that most students found the amount of compulsory work in the subject was already too much and did not think additional compulsory exercises were a good idea. However, some users reported that it depended on whether or not the other workload in the course was reduced or not.

Other feedback from users consisted of some confusion on the concept of *user flashcard decks* and suggestions of implementing *spaced repetition* for flashcard studying. This is further discussed in the further work section, chapter 7.2.

Chapter 7

Discussion and Further Work

7.1 Discussion

From the beta test, the functionality, usability, and scalability of the application have been demonstrated to perform with a high degree of success. No bugs were reported and there were no occurrences of crashes or malfunctions. The goal of creating a responsive, intuitive, and user-friendly application is also reflected in the survey results from the beta test, with all survey respondents agreeing all features of the application were intuitive and simple to use.

As testing the application as a voluntary exercise in the course at the end of the semester yielded few flashcards, efforts should be made to encourage students to participate in the flashcard creation process. To maximize the learning potential of the application the submission and peer review of flashcards could be made a mandatory exercise in courses, where students are evaluated based on the ratings of their flashcards and possibly on how well they rate other students' and their own flashcards. This type of exercise would provide a unique way to encourage students to get a solid understanding of the course material after it has been lectured, and effectively retain the information learned through the semester by the use of flashcard study.

Despite a low number of participants in the *create flashcard* and *peer review* process of the beta test, a majority of those who did participate and answered the survey responded that both the creation and peer review of flashcards was by itself rewarding and educational, demonstrating the potential learning benefits of these features of the system, as theorized in chapter 2

The functional and non-functional requirements were successfully implemented, with few compromises necessary. Still, some features remain to be implemented. Further use of this application in future semesters should come with the added feature of spaced repetition study of flashcards, maximizing the information retention effect of flashcard studying. Because the studying of flashcards will always

be a voluntary part of the application, the flashcard review method implemented for the beta test of this system should also be kept as a feature of the application, making it possible for students who have not utilized the spaced repetition system to cram random flashcards before the exam.

With the popularity of the *study* feature of the application and a majority of students surveyed responding positively to the application, reporting that studying the flashcards helped them prepare for the exam, the potential for this application as a learning resource and exercise system for future semesters has been demonstrated. There is potential for further expansion with spaced repetition and other features, and the system could also be used for other courses at NTNU with few modifications necessary.

7.2 Further Work

7.2.1 Hosting on NTNU Virtual Server

The application should be hosted on a NTNU Virtual Server to handle the data traffic and administration of the system. A virtual machine at http://ttk4145.it.ntnu.no/, supporting MySQL and automatic backups, was set up in collaboration with *Orakeltjenesten*, but hosting the application on this server remains to be done.

7.2.2 Implementation of Spaced Repetition

The foundations are set for expanding the application with a Spaced Repetition study feature. Different algorithms for spaced repetition exist, but the SuperMemo (SM-2) algorithm, used by Anki and other flashcard applications, [44] was implemented and tested on the back-end. However, due to time constraints and priorities, it was scrapped in favor of an alternative solution, as explained in chapter 4.1.4.

7.2.3 Further Data Analysis

The existing system collects data for each user, their flashcards and their ratings, but there is much more data that can be collected an analyzed. Examples include

- Study habits of students
- Which flashcards are most difficulty based on flashcard study reviews
- Evaluating the students' peer review performance - How well does the student rate flashcards?
- Which parts of the curriculum are most difficult
- Correlation between questions, if question A is successfully answered - is the student more likely to correctly answer Question B.

7.2.4 Responsive Design

Some work on the design and usability of the web application remains to be implemented. Mainly the application is not very responsive on smaller screen sizes in portrait mode. React libraries for the conditional rendering of separate or modified components to differently sized screens exists, but the existing components can also be made more responsive by further utilizing the Material UI Grid component or other flex based features of Material UI and React.

7.2.5 Mobile App for Flashcard Review

As the REST API on the back-end exists as a completely separate entity from the client, alternative clients can be used to interact with the server. The daily task of doing spaced repetition flashcard reviews could benefit from a simple mobile application for flashcard study. Building the application in React Native would be the obvious solution. React Native allows for the creation of native iOS and Android apps using React, but using native components instead of web components [45].

Bibliography

[1] A. Ebrahimi. (2011). 'Alle 'koker" oppgaver,' [Online]. Available: https://www.nrk.no/trondelag/_-alle-_koker_-oppgaver-1.7634773. (accessed: 27.02.2021).

[2] N. Sonnad. (2018). 'You probably won't remember this, but the "forgetting curve" theory explains why learning is hard,' [Online]. Available: https://qz.com/1213768/the-forgetting-curve-explains-why-humans-struggle-to-memorize/#:~:text=Hermann%5C%20Ebbinghaus'%5C%20memory%5C%20experiments,is%5C%20over%5C%20100%5C%20years%5C%20old.&text=Ebbinghaus%5C%20discovered%5C%20that%5C%20his%5C%20memory%5C%20of%5C%20them%5C%20quickly%5C%20decayed.. (accessed: 27.02.2021).

[3] J. A. G. Francis Deng, A. T. Douglas P. Larsen * and C. Breslina. (2015). 'Student-directed retrieval practice is a predictor of medical licensing examination performance,' [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4673073/. (accessed: 16.05.2021).

[4] R. Patterson. (2021). 'These flashcard apps will help you study better in 2021,' [Online]. Available: https://collegeinfogeek.com/flashcard-apps/. (accessed: 16.05.2021).

[5] S. Greving and T. Richter. (2018). 'Examining the testing effect in university teaching: Retrievability and question format matter,' [Online]. Available: https://www.frontiersin.org/articles/10.3389/fpsyg.2018.02412/full. (accessed: 15.05.2021).

[6] N. E. W. Jonathan M. Golding1 and B. Fletcher1. (2012). 'Examining the testing effect in university teaching: Retrievability and question format matter,' [Online]. Available: https://journals.sagepub.com/doi/pdf/10.1177/0098628312450436#:~:text=Almost%5C%2070%5C%25%5C%20of%5C%20the%5C%20class,on%5C%20one%5C%20or%5C%20two%5C%20exams.. (accessed: 15.05.2021).

[7] S. H. K. Kang. (2016). 'Spaced repetition promotes efficient and effective learning: Policy implications for instruction,' [Online]. Available: https://www.researchgate.net/publication/290511665_Spaced_Repetition_

Promotes_Efficient_and_Effective_Learning_Policy_Implications_for_Instruction. (accessed: 03.02.2021).

[8] S. Tamm. (2021). 'Spaced repetition: A guide to the technique,' [Online]. Available: https://e-student.org/spaced-repetition/. (accessed: 15.05.2021).

[9] B. A. Chun. (2021). 'The effect of flipped learning on academic performance as an innovative method for overcoming ebbinghaus' forgetting curve,' [Online]. Available: https://www.researchgate.net/figure/Ebbinghaus-forgetting-curve-and-review-cycle_fig1_324816198. (accessed: 26.05.2021).

[10] A. T. David Nicola * and C. Breslina. (2014). 'Rethinking feedback practices in higher education: A peer review perspective,' [Online]. Available: http://l2l.ie/documents/feedback.pdf. (accessed: 15.05.2021).

[11] R. T. Fielding. (2000). 'Chapter 5 representational state transfer (rest)?' [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api. (accessed: 7.05.2021).

[12] Oracle. (2021). 'What is a relational database?' [Online]. Available: https://www.oracle.com/database/what-is-a-relational-database/. (accessed: 21.05.2021).

[13] K. Ismail. (2018). 'What is a single page application?' [Online]. Available: https://www.cmswire.com/digital-experience/what-is-a-single-page-application/. (accessed: 7.05.2021).

[14] D. Robinson. (2017). 'The incredible growth of python,' [Online]. Available: https://stackoverflow.blog/2017/09/06/incredible-growth-python/. (accessed: 19.04.2021).

[15] jetbrains. (2020). 'Python developers survey 2020 results,' [Online]. Available: https://www.jetbrains.com/lp/python-developers-survey-2020/. (accessed: 19.04.2021).

[16] Flask. (2020). 'Python developers survey 2020 results,' [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/foreword/. (accessed: 19.04.2021).

[17] Pallets. (2010). 'Modular applications with blueprints,' [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/blueprints/. (accessed: 5.05.2021).

[18] C. Dolphin. (2013). 'Flask-cors,' [Online]. Available: https://flask-cors.readthedocs.io/en/latest/. (accessed: 19.04.2021).

[19] (2021). 'Python documentation - virtual environments and packages,' [Online]. Available: https://docs.python.org/3/tutorial/venv.html. (accessed: 16.04.2021).

[20] B. Krebs. (2017). 'Sqlalchemy orm tutorial for python developers,' [Online]. Available: https://auth0.com/blog/sqlalchemy-orm-tutorial-for-python-developers/. (accessed: 16.04.2021).

[21] M. Grinberg. (2019). 'Flask-migrate,' [Online]. Available: https://flask-migrate.readthedocs.io/en/latest/. (accessed: 19.04.2021).

[22] Prisma. (2020). 'What are database migrations?' [Online]. Available: https://www.prisma.io/dataguide/types/relational/what-are-database-migrations. (accessed: 19.04.2021).

[23] Smurfix. (2020). 'Flask-script?' [Online]. Available: https://flask-script.readthedocs.io/en/latest/. (accessed: 19.04.2021).

[24] Auth0. (2020). 'Introduction to json web tokens,' [Online]. Available: https://jwt.io/introduction/. (accessed: 19.04.2021).

[25] Auth0. (2020). 'Introduction to json web tokens,' [Online]. Available: https://flask-jwt-extended.readthedocs.io/en/stable/options/#configuration-options. (accessed: 19.04.2021).

[26] F. Inc. (2021). 'Python developers survey 2020 results,' [Online]. Available: https://reactjs.org/. (accessed: 30.04.2021).

[27] F. Inc. (2021). 'Introducing hooks,' [Online]. Available: https://reactjs.org/docs/hooks-intro.html. (accessed: 3.05.2021).

[28] F. Inc. (2021). 'Introducing jsx,' [Online]. Available: https://reactjs.org/docs/introducing-jsx.html. (accessed: 3.05.2021).

[29] npm. (2021). 'About npm,' [Online]. Available: https://docs.npmjs.com/about-npm. (accessed: 5.05.2021).

[30] D. Abramov. (2015). 'Introducing jsx,' [Online]. Available: https://react-redux.js.org/introduction/why-use-react-redux. (accessed: 3.05.2021).

[31] google. (2021). 'Material design - introduction,' [Online]. Available: https://material.io/design/introduction#components. (accessed: 4.05.2021).

[32] axios. (2021). 'Axios - getting started,' [Online]. Available: https://axios-http.com/docs/intro. (accessed: 7.05.2021).

[33] E. H. MIT. (2021). 'React-markdown,' [Online]. Available: https://github.com/remarkjs/react-markdown. (accessed: 7.05.2021).

[34] R. Florence. (2021). 'React-router-dom,' [Online]. Available: https://reactrouter.com/. (accessed: 7.05.2021).

[35] U. Eriksson. (2012). 'Why is the difference between functional and non-functional requirements important?' [Online]. Available: https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/. (accessed: 21.05.2021).

[36] NTNU. (2021). 'Feide innlogging,' [Online]. Available: https://innsida.ntnu.no/wiki/-/wiki/Norsk/FEIDE+innlogging. (accessed: 09.05.2021).

[37] KirstenS. (2021). 'Cross site scripting (xss),' [Online]. Available: https://owasp.org/www-community/attacks/xss/. (accessed: 21.05.2021).

[38] Z. Liew. (2021). 'Create er diagram of a database in mysql workbench,' [Online]. Available: https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/. (accessed: 24.05.2021).

[39] MIT. (2021). 'Sqlalchemy - self-referential many-to-many relationship,' [Online]. Available: https://docs.sqlalchemy.org/en/14/orm/join_conditions.html#self-referential-many-to-many. (accessed: 12.05.2021).

[40] Auth0. (2020). 'Jwt revoking / blocklist,' [Online]. Available: https://flask-jwt-extended.readthedocs.io/en/stable/blocklist_and_token_revoking/#database. (accessed: 09.05.2021).

[41] google. (2021). 'Google form - about,' [Online]. Available: https://www.google.com/forms/about/. (accessed: 14.05.2021).

[42] google. (2021). 'Material ui datagrid,' [Online]. Available: https://material-ui.com/api/data-grid/. (accessed: 15.05.2021).

[43] heroku. (2021). 'Deploying python applications with gunicorn,' [Online]. Available: https://devcenter.heroku.com/articles/python-gunicorn. (accessed: 15.05.2021).

[44] P. Wozniak. (1990). 'Application of a computer to improve the results obtained in working with the supermemo method,' [Online]. Available: https://www.supermemo.com/en/archives1990-2015/english/ol/sm2. (accessed: 21.05.2021).

[45] Facebook. (2021). 'React native,' [Online]. Available: https://reactnative.dev/. (accessed: 21.05.2021).

Appendix A

ER Diagram

Entity Relationship Diagram of the applications relational database. Generated with MySQL Workbench.

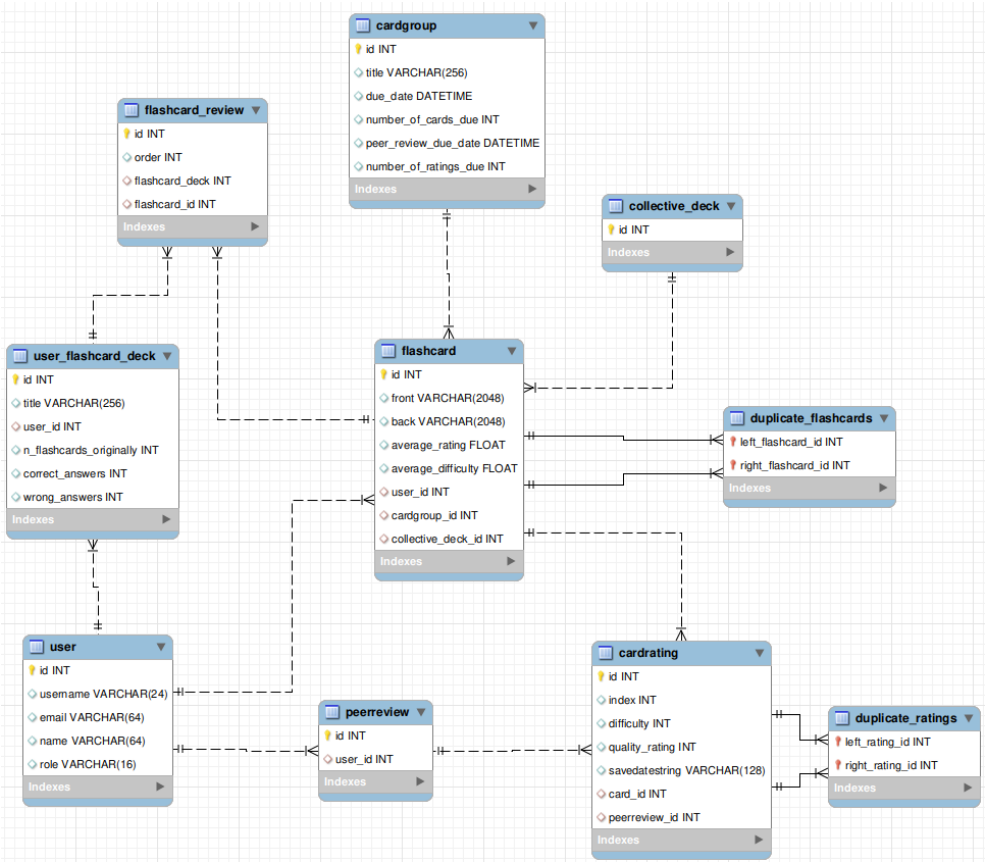


Figure A.1: MySQL Workbench generated ER diagram

Appendix B

API Routes

API Routes for the REST API

url	**method**	**input**	**returns**
/api/admin/<uid>	POST	none	created admin
/api/admin/<uid>	DELETE	none	removed admin
/api/currentuser/user	GET	none	current user
/api/admin/users/all	GET	none	all users
/api/admin/users/role=<role>	GET	none	all users with role
/api/login/url	GET	none	login url
/api/login/userdata*	GET	none	jwt tokens
/api/login/callback	GET	userdata	login redirect
/api/token/expired	POST	none	token status
/api/token/refresh	POST	none	refresh token
/api/logout/access	POST	none	logout status
/api/logout/refreshtoken	POST	none	logout status

Table B.1: user Routes
∗ *get request url params: userdata=[name, email,username, sha1] (sha1 is login token)*

url	method	input	returns
/api/admin/flashcards	GET	none	all flashcards
/api/admin/flashcards/<cid>	GET	none	flashcard with id=cid
/api/admin/flashcards/cardratings	GET	none	ratings of flashcard
/api/currentuser/flashcards	POST	flashcard*	created flashcard
/api/currentuser/flashcards/<cid>	PUT	flashcard**	edited flashcard
”/<cid>	DELETE	none	deleted flashcard
”/cardgroupid=<cdid>	GET	none	user flashcards from group

Table B.2: flashcard routes
∗ *flashcard: {front, back, cardgroupid}*
∗∗ *flashcard: {front, back}*

url	method	input	returns
/api/cardgroups	GET	none	all cardgroups
/api/cardgroups/<cgid>	GET	none	cardgroup with id=cgid
/api/admin/cardgroups	POST	cardgroup*	created cardgroup
/api/admin/cardgroups/<cgid>	PUT	cardgroup*	edited cardgroup
/api/admin/cardgroups/<cgid>	DELETE	none	deleted cardgroup
/api/admin/cardgroups/<cgid>/flashcards**	GET	none	all cardgroup flashcards

Table B.3: cardgroup routes
∗ *cardgroup: {title, numberOfCardsDue, dueDate}*
∗∗*: filters: minrating, removeduplicates*

url	method	input	returns
/api/admin/peerreviews	POST	peerreview*	success status
/api/admin/peerreviews	PUT	peerreview**	success status
/api/currentuser/peerreviews	GET	none	user peerreviews
/api/currentuser/peerreview/<prid>	GET	none	peerreview id=prid
/api/currentuser/peerreview/<prid>/cardratings	GET	none	peerreview ratings
/api/admin/cardgroup/<cgid>/peerreview	GET	none	cardgroup peerreviews
/api/admin/cardgroup/<cgid>/peerreview	DELETE	none	succcess status

Table B.4: peerreview routes
∗ *peerreview: {groupId, dueDate, numberOfReviews}*
∗∗ *peerreview: {groupId, dueDate}*

url	method	input	returns
/api/currentuser/cardrating/<rid>/difficulty	PATCH	difficulty	patched rating
/api/currentuser/cardrating/<rid>/quality	PATCH	quality	patched rating
/api/currentuser/cardrating/<rid>/duplicates	PATCH	duplicates	patched rating

Table B.5: cardrating routes

url	**method**	**input**	**returns**
/api/admin/collective-deck/flashcards	POST	none	collective deck flashcards
/api/admin/collective-deck/flashcards	DELETE	none	collective deck flashcards
/api/collective-deck/flashcards*	GET	none	collective deck flashcards
/api/collective-deck/cardgroups	GET	none	cardgroups with cards in collective deck

Table B.6: collective_deck routes
∗: filters: difficulty-min, difficulty-max, cardgroup-id, ncards, id-only

url	**method**	**input**	**returns**
/api/currentuser/user-flashcard-decks	GET	none	all user flashcard decks
/api/currentuser/user-flashcard-decks	POST	none	created user flashcard deck
”/<ufdid>	DELETE	none	deleted user flashcard deck
”/<ufdid>/flashcards	GET	none	flashcards from deck
”/<ufdid>/flashcard/<cid>/answer	POST	study answer*	flashcards from deck

Table B.7: user_flashcard_deck routes
∗ answer: { correct: true/false }

Appendix C

Code

Github link for the projects code and guides for setting up and running the application.

https://github.com/hunshamar/TTK4145-Flashcards

Appendix D

Beta Test Survey Results

Survey responses from 15 students conducted after the beta test of the system.

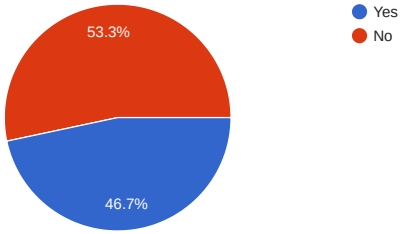
TTK4145flashcards - feedback

15 responses

Publish analytics

Did you use the "create flashcard" feature of the application

15 responses



If no, why not?

8 responses

Didnt have time

Time, Shitloads of things to do late (way to late) in the semester, including exercise 8 & 9, code review, post mortem.

Fint for å få inn kunnskap om sentrale begreper

This course takes too much time allready

Did not have time to contribute.

Prioritized elevator project and other projects

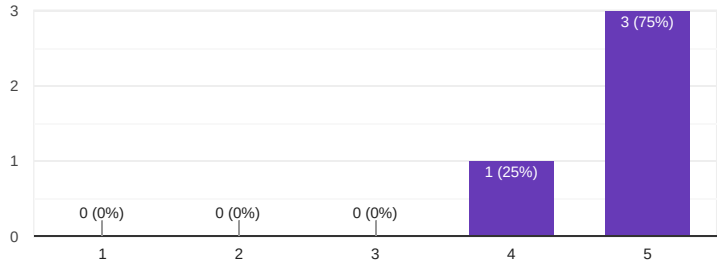
Because I had a lot do do and forgot unfortunately :(

I don't feel competent enough to know the right answer

Flashcard Submission

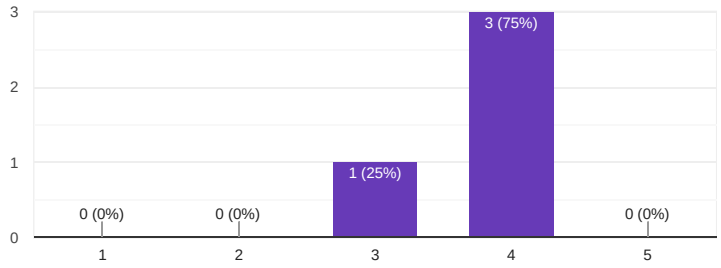
The flashcard creation process was intuitive and simple to use

4 responses



Creating flashcards was rewarding and educational

4 responses



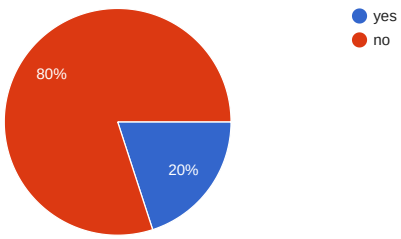
Additional feedback for flashcard submission

0 responses

No responses yet for this question.

Did you use the application to rate and "peer review" other students' flashcards?

15 responses



If no, why not?

10 responses

Didnt have time

I forgot it, since I was so busy with studying for other exams.

Didn't have enough knowledge on the subjects, so it felt pointless

Visste ikke funksjonen fantes

This course takes too much time allready

Hadde ikke nok kunnskap

Did not find others flashcards at the time I looked at the feature

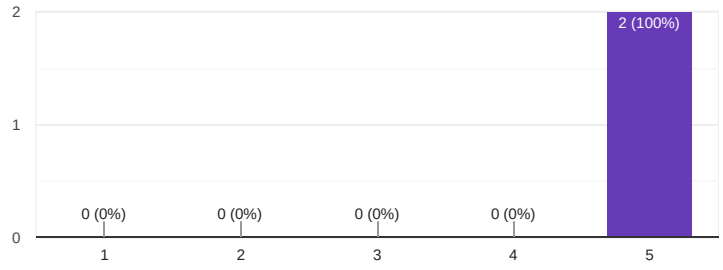
Didn't know when this happened, and didn't add cards either

My skills were so lacking that I couldn't really say if it was relevant to the course or not.

Peer Review

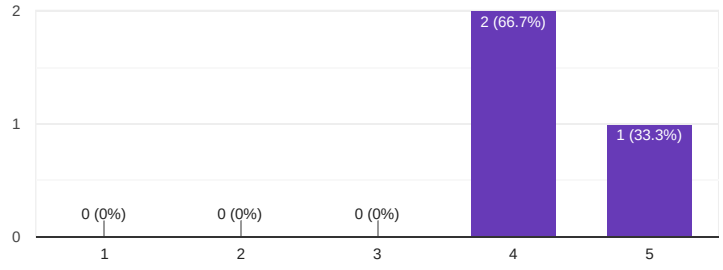
The "peer review" process was intuitive and simple to use

2 responses



Rating other students' flashcards was rewarding and educational

3 responses



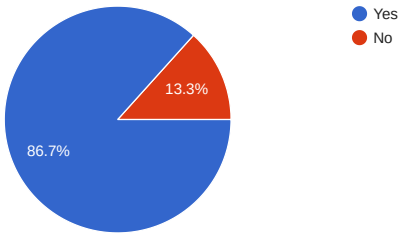
Additional Feedback for the peer review process

0 responses

No responses yet for this question.

Did you use the application to study flashcards

15 responses



If no, why not?

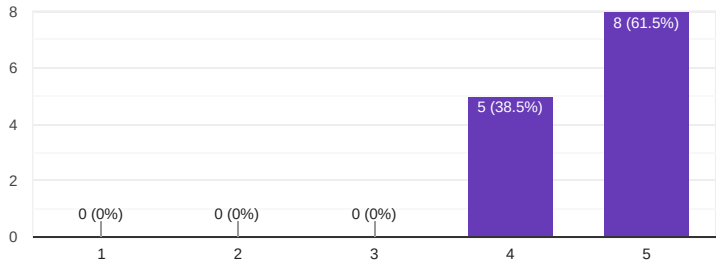
1 response

Did not find any

Flashcard Study

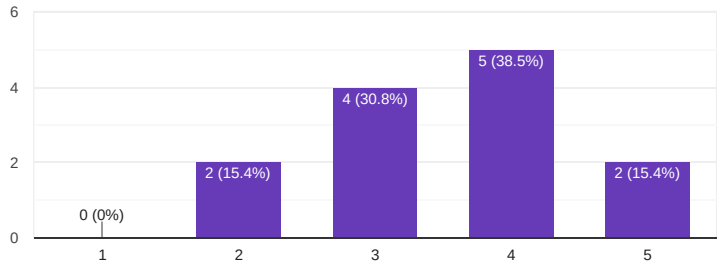
The "flashcard study" process was intuitive and simple to use

13 responses



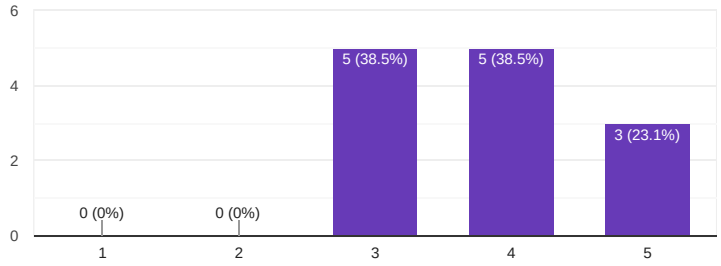
Studying flashcards in the application helped me prepare for the exam

13 responses



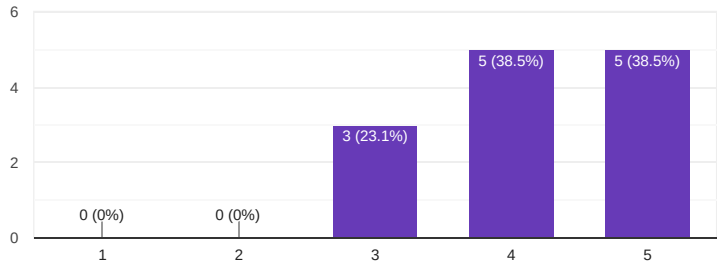
The flashcards studied had a high level of quality and relevance to the course material.

13 responses



I wish there were more flashcards to study

13 responses



Additional feedback for the flashcard study process

4 responses

Felt in some subject there could be a little more cards. Not that I can critizise the amount since I didnt add some myself. Other than that i find it irritating that the decks were deleted after using them. Was an uneccesary feature imo.

I liked it. But I wondered if the staff had looked through them and approved that they were all correct? I was a bit worried that there might be some mistakes in them, since they were made by students.

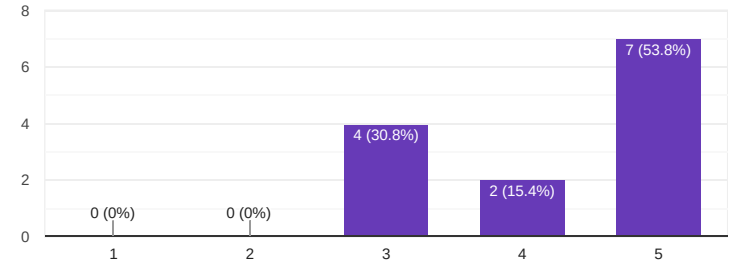
Some report or edit function for the cards with typos and/or too little information

I think it's useful, it would be even more useful if we could have done it from the start.

General

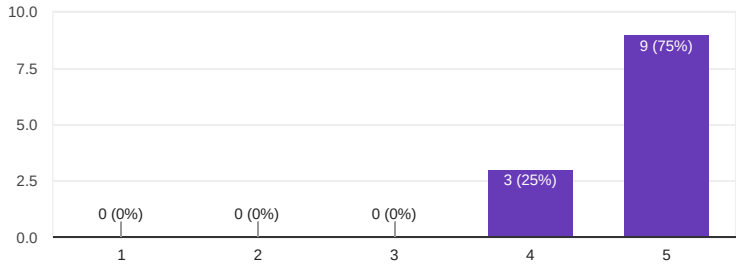
The design of the web application was user friendly and intuitive

13 responses



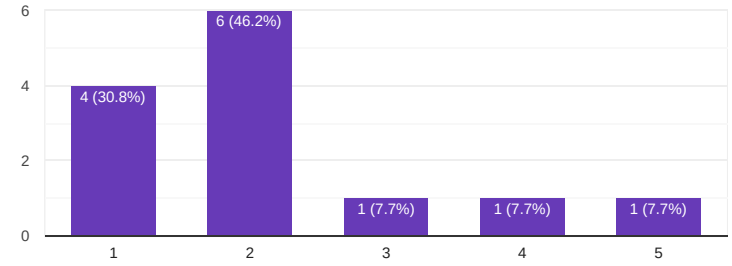
The web application was fast and responsive

12 responses



The use of the flashcard application to submit and rate flashcards should be a compulsory exercise in ttk4145 - Real Time Programming

13 responses



Additional feedback / suggestions for the flashcard application in general

6 responses

Could maybe, but that requires the students know the curriculum good

I do NOT think it should be compulsory, with all the other tasks we had to do, especially since we were given so many tasks at the very end. However if this is not the case, then it could be compulsory.

Ikke helt lett å skjønne hvordan jeg skulle lage et nytt deck første gang fordi jeg trodde det betydde å skape nye kort

Dis not understand how to start studying the cards (making a New board)

I think that whether it should be compulsary or not depend on how much else in the subject that is compulsory. I am glad that it was not compulsary this time.

I really like the way Anki have done their flashcards. Maybe this could provide some inspiration for additional features: https://apps.ankiweb.net/

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms