Flashcards

Part of the Examination Assessment of ITPE3200 Web Applications
at OsloMet

31.10.2023

Documentation

1 Introduction

Our project is a flashcard app solution for learning any kind of subject. The project is based on .Net 6.0 and the Model-View-Controller framework.

The major functionality is registration and login, CRUD operations on subjects, decks and cards, and the ability to practice a subject by starting a practice session. During a practice session the user is presented with flashcards from the selected deck and should think about

their answer before flipping to the side with the answer. If their answer was correct, they may mark the card as answered correctly. If it was wrong, they may choose to put the card back in the deck. This is repeated until all cards are answered correctly, or the user ends the session.

Users are able to create private subjects that only belong to them, and are not visible to other users. Subjects can also be made public and visible to others. The decks or cards of public subjects can not be edited by other users, but may be used in practice sessions. Two of the public decks are generated using ChatGPT (OpenAI, 2023).

2 Architecture

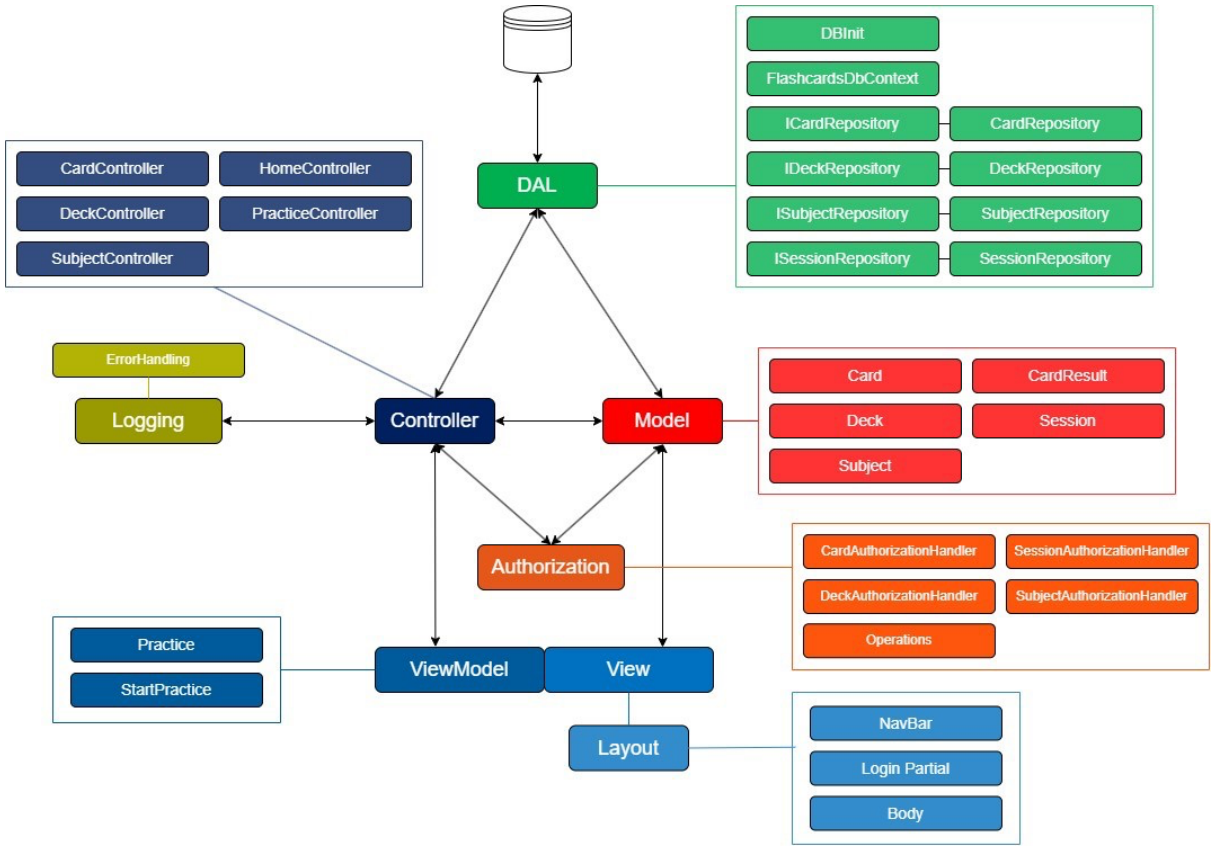

Figure 1. Description of the architecture.

The application architecture is shown in figure 1. It consists of the following groups.

2.1 DAL

The classes within *DAL* are interfaces that facilitate the communication with the database.

2.2 Controller

Our system follows the *Model-View-Controller (MVC)* pattern, with several controllers.

- CardController
- DeckController
- SubjectController
- HomeController
- PracticeController

2.3 Model

Model consists of the primary entities used in the application.

- Card
- CardResult
- Subject
- Deck ● Session

2.4 View

View presents data passed from the controller to the user.

The *Layout* partial wraps the views with top-level properties. This is where the navigation bar and login partial is placed.

2.5 ViewModel

ViewModel interacts with View. The ViewModel gives data in a suitable format for display.

2.6 Logging

Logging ensures that the returned error messages are formatted as strings that contain controller name, action name and the error message defined in controllers.

2.7 Authorization

Authorization ensures that users have the necessary permissions to perform certain tasks or access given resources.

3 Database

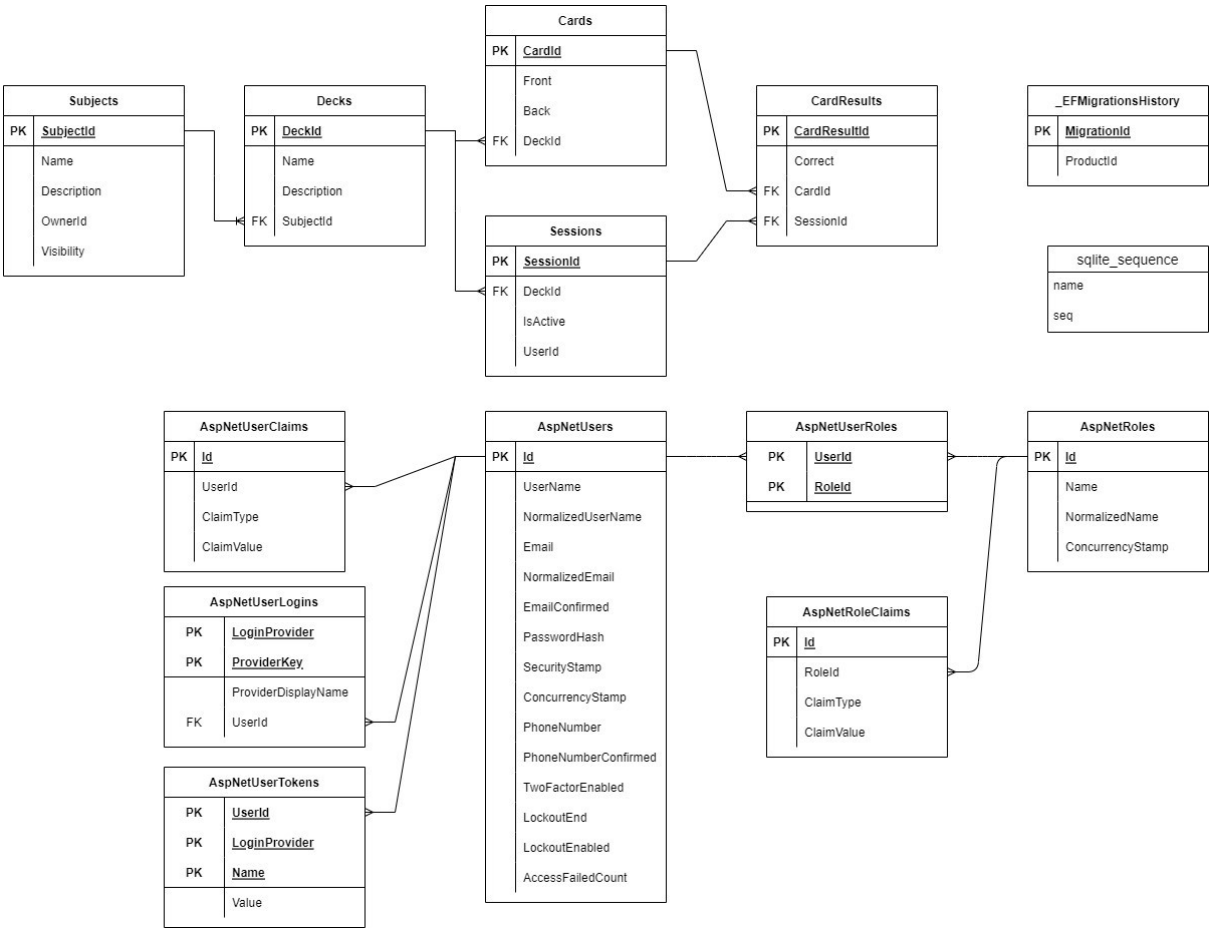

Figure 2. ER diagram of the SQLite database.

The web application uses a SQLite database that contains 14 tables. Seven of these relate to security, one holds the migration data, and the rest relate to the flashcards. The last table, *sqlite_sequence* , is generated automatically and shows the highest primary key of sessions, subjects, decks, and cards in the database.

We added authorization and authentication by using the packages *Microsoft.AspNetCore.Identity.EntityFrameworkCore* and *Microsoft.AspNetCore.Identity.UI* . Authentication is facilitated with registration and login, while authorization ensures that certain actions are only allowed by authorized users. For instance, only logged in users can access and modify their private cards, decks and views.

3.1 Migration History

The table *_EFMigrationsHistory* keeps track of the migration versions of the database. It contained the columns *MigrationId* and *ProductVersion* .

3.2 Security

The following tables in the database facilitate authentication and authorization: *AspNetUsers* , *AspNetUserClaims* , *AspNetUserLogins* , *AspNetUserTokens* , *AspNetRoles* , *AspNetRoleClaims* , and *AspNetUserRoles* .

3.3 Flashcards

The following tables are used for the flashcards: *Subjects* , *Decks* , *Cards* , *Sessions* and *CardResults* .

The *Subjects* table contains the field *SubjectId* (PK), *Name*, *Description*, *OwnerId* and *Visibility* . *OwnerId* is used to identify the ownership of subjects, decks and cards for logged-in users. For instance, cards that a logged-in user creates would get the *OwnerId* of that user stored in the database. The *Decks* table has the columns *DeckId* (PK), *Name*, *Description* and *SubjectId* . Subjects can have multiple decks. Likewise, decks can have many cards. The *Cards* table contains the fields *CardId* (PK), *Front* , *Back* and *DeckId* . Each deck can have multiple sessions. The *Sessions* table is used for the practice function, and consists of the columns *SessionId* (PK), *DeckId* , *IsActive* and *UserId* . The *UserId* belongs to the logged-in user that did a practice session. The table *CardResults* contains the columns *CardResultId* (PK), *Correct* , *CardId* and *SessionId* . *CardResults* stores the result for a particular card during a session.

4 Details

4.1 Design

The group decided to use a simple and intuitive design to make the experience reflect the simplicity of learning with flashcards. The design was in most part styled with bootstrap 5 classes (*Bootstrap* , n.d.). The web application uses colors, icons (*Bootstrap Icons* , n.d.), and hover states to provide the user with sufficient feedback.

The final look was inspired by initial sketches we made in Figma.

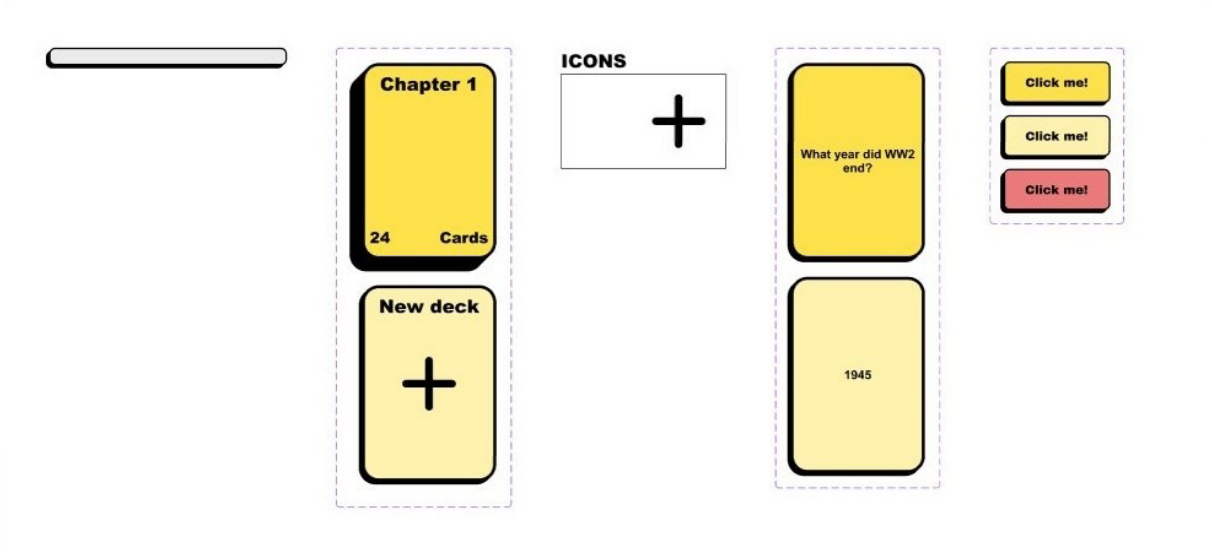

Figure 3. Figma sketch of a card and a deck.

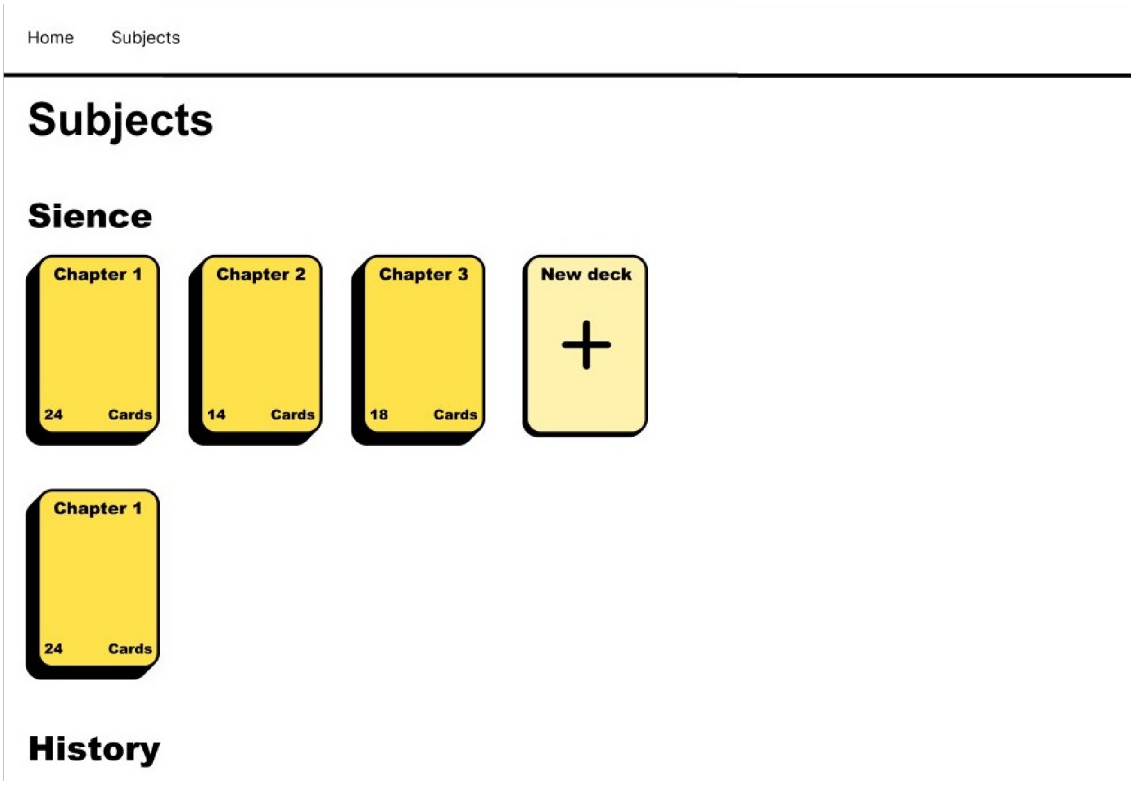

Figure 4. Figma sketch of the Subjects page with multiple subjects and decks.

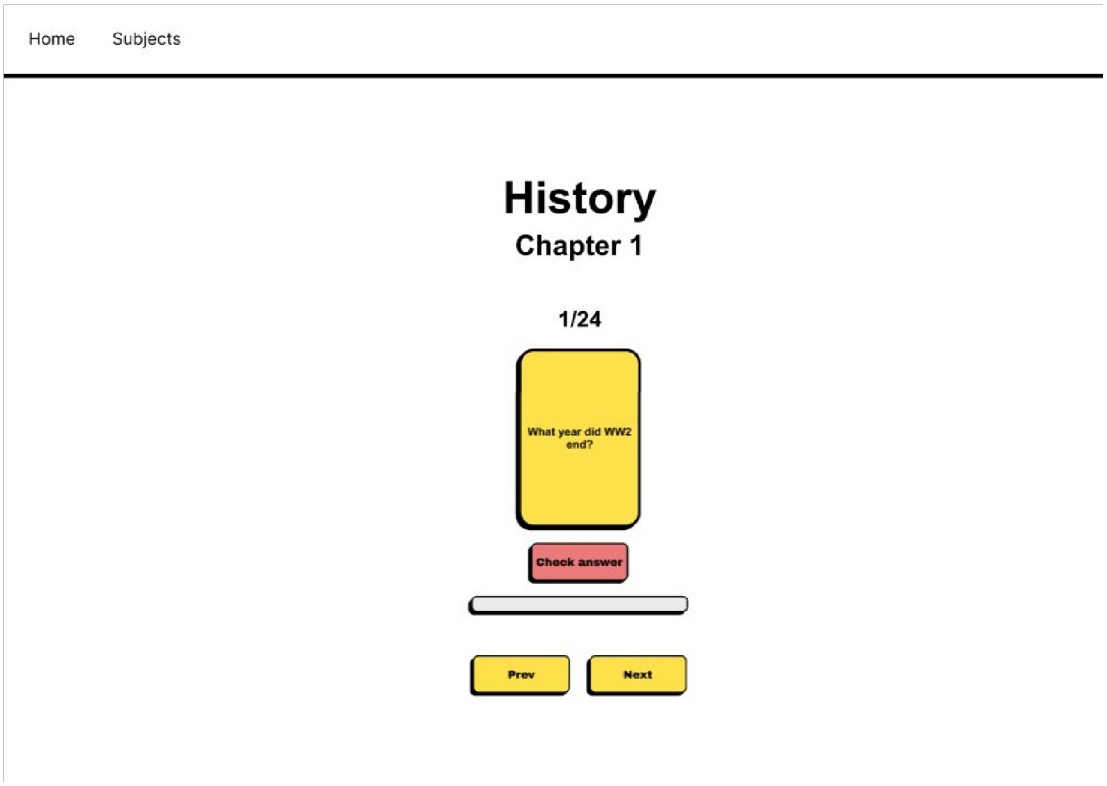

Figure 5. Figma sketch of the practice view.

The web application supports both light and dark mode. Bootstrap 5 comes with a dark theme that is disabled by default. We implemented a dark theme toggle since many users find a dark theme less taxing on their vision. The theme can be changed by pressing the button with the sun or moon icons next to the account buttons in the main navigation bar. If the user toggles the theme, their preference is stored in localstorage client side.

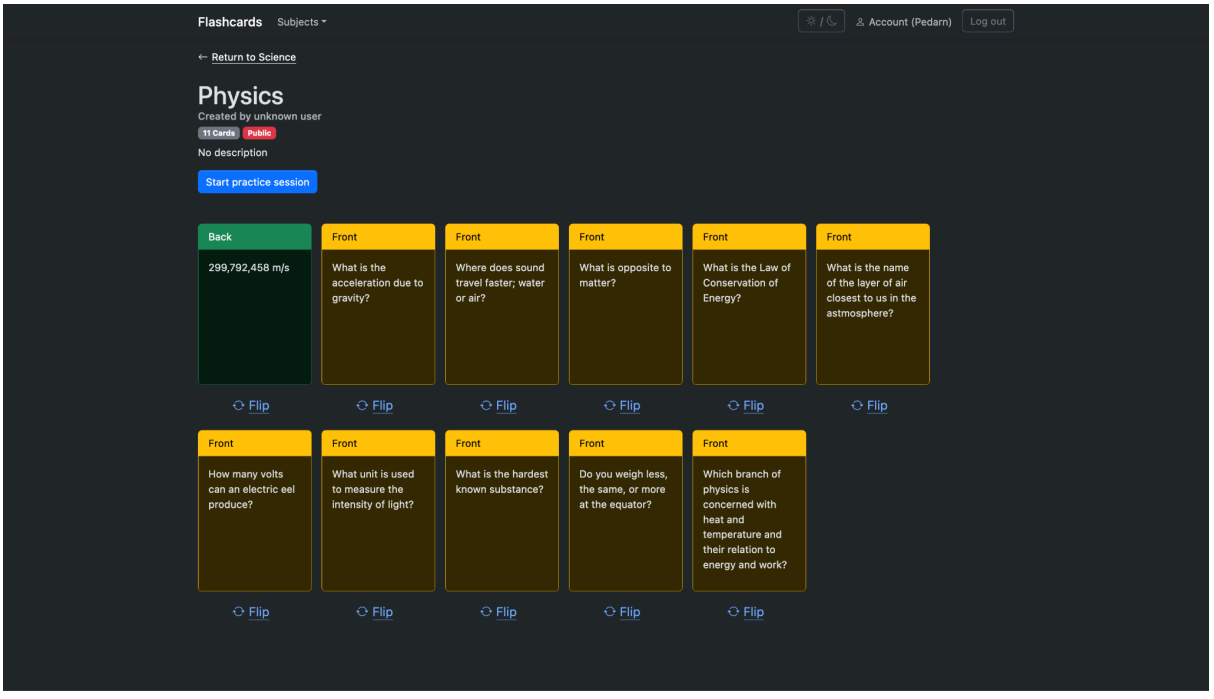

Figure 6. View of the deck Physics in the dark theme.

The landing page was inspired by hero sections, and utilized a bootstrap carousel with dot indicators. The caption is placed in the middle of the first image and has a strong color to catch the attention of the visitor. The caption and the call-to-action button have contrasting colors.

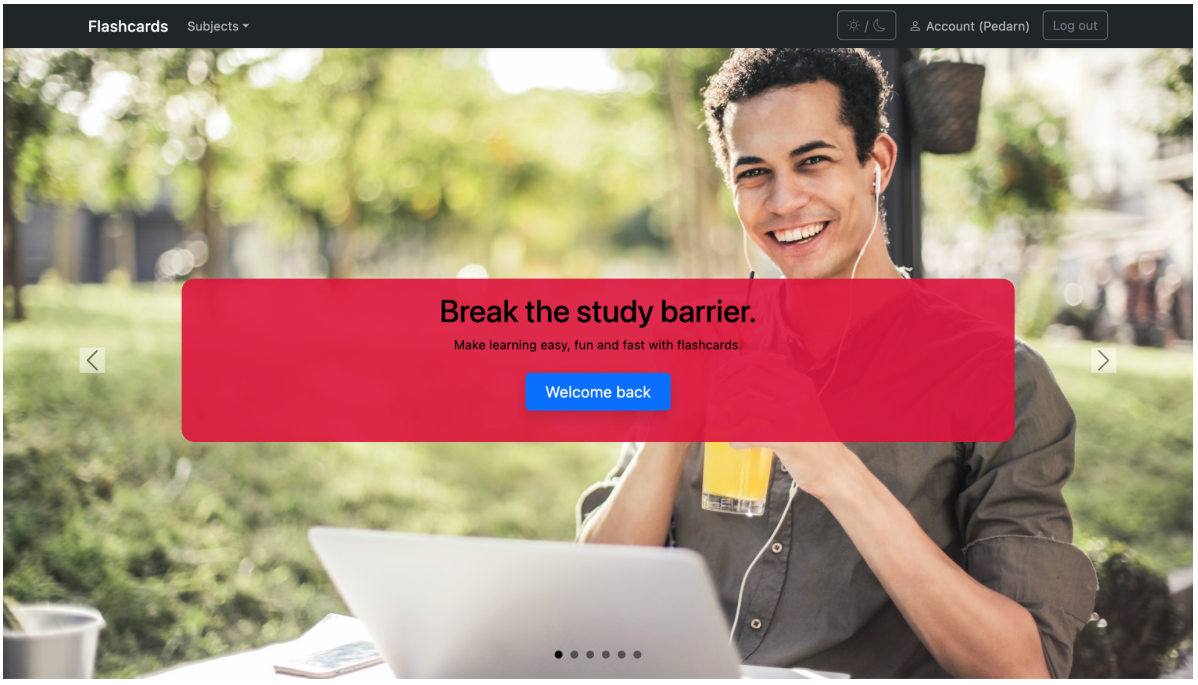

Figure 7. Landing page in the dark theme.

The login and registration pages are basic and have their components centered in a mobilefirst style. Login and registration pages have an important functional purpose and these pages should be predictable to users.

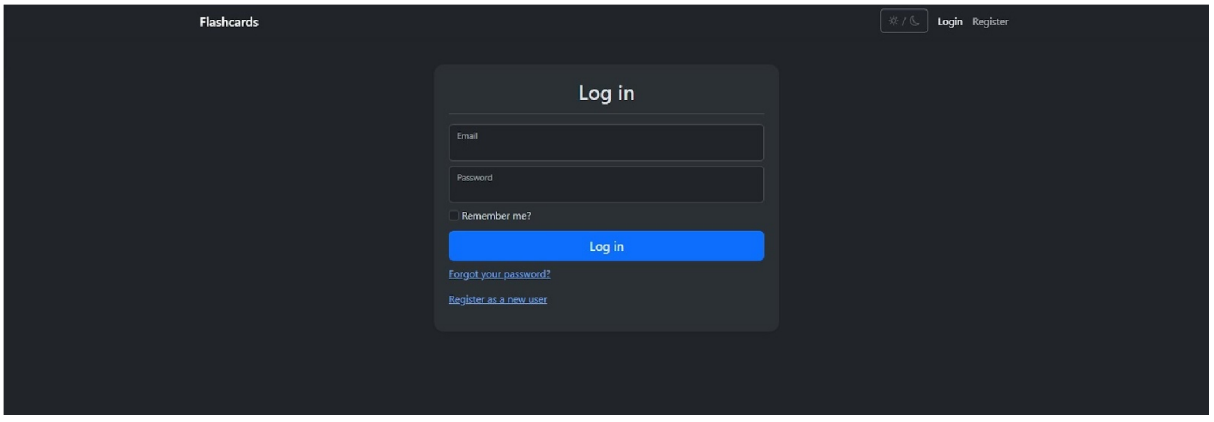

Figure 8. The login page in the dark theme.

4.2 Models

The application's data is defined in models. This project contains five data classes in the models which structure the data and these are *Subject* , *Deck* , *Card* , *CardResult* and *Session* .

Each class stores different properties requiring specific data types. We use the get and set method to get data and set it in the class. The controller will help models to display it to the view. We also have included navigation properties to all the classes so they can be connected between the other model classes. The models also contain input validations with various annotations to guide the user to give suitable inputs.

4.3 Views

The views are necessary for user interaction and views will present data passed from the controller to the user. In our project we have multiple layouts. There is one main layout wrapping all views, and this is where the head and navigation bar is placed. This is important so that all scripts and files are imported in all views. In addition to the main layout we created secondary layouts whose main purpose is to provide a visual container for the view. This makes it so all views sharing the same layout have similar styling. The layout files are placed in a shared folder under Views, and can be used by any view. The NavBar partial is present on all views and makes it possible for friendly navigation for the user.

Other than the default bootstrap classes we wrote some custom css. This was mainly to style the home page, and implement the flipping of cards. The card flipping was inspired by this swap animation (*Tailwind Swap Component — Tailwind CSS Components* , n.d.) but adapted for our project. Bootstrap relies on the css preprocessor Sass (*Sass* , n.d.) which is a compiled language. To facilitate using Sass in our project we installed the Sass compiler locally using Node Package Manager (*NpmJS* , n.d.) and compiled the .scss file before committing our changes.

4.4 Controllers

Our project consists of five different controllers where each controller has its own functionality. Across all controllers there have been implemented error logging and error views are returned, based on the status code.

CardController is responsible for handling user requests related to cards for a given deck. We have implemented multiple action methods in our CardController that handles CRUD operations for cards by the users. Depending on which CRUD operation the user intends to perform, a view is returned related to that operation.

DeckController also handles CRUD operations that the user performs and returns a view, based on user actions in the deck view.

HomeController is made of one Index action method. Index returns a view for our landing page which is the entry point for our application. Our landing page contains a brief introduction to our application and encourages users to create an account.

PracticeController manages requests by the user and processes them based on user interactions on the practice part in our web application. Depending on which interaction, the correct views are returned such as summary view or practice view, where cards from the current session deck are displayed.

In *SubjectController* we have implemented CRUD for handling Private and Public subjects related operations in our application, and returning correct views depending on user requests.

4.5 Data Access Layer

The Data Access Layer (DAL) in Flashcards application is the primary link between data models and the database.

4.5.1 Structure

Classes represent the data structure in the application and are known as entity models, where each entity has a corresponding table in the database. We simplify the complexity of the database operations with Entity Framework , and make the coupling between the database and application less tight. Instead of depending on implementation, the controllers only depend on the interfaces provided by the data access layer.

4.5.2 CRUD

DAL affords CRUD operations (create, read, update, delete) to the controllers. Entities such as Subject, Deck, and Card gives users the ability to add new topics, decks, or cards. Users can also view their existing cards, modify them, and delete if necessary.

4.5.3 FlashcardsDbContext

FlashcardsDbContext is a leading class within the Entity Framework. It establishes a connection with the database that enables CRUD operations on the entities. At runtime, it handles entity objects by loading them with data from the database, tracking changes, and saving data back to the database.

4.5.4 Repository Pattern

Distinct repositories are used for different entities. Each repository has methods dedicated to specific data operations, encapsulating the necessary logic for data retrieval and modification. By applying the repository pattern, DAL isolates the data access logic, improves system maintainability, and makes unit testing more accessible.

4.5.5 Validation

Our application has a built-in-server-side validation in DAL, to preserve data integrity and offer a high-quality user experience. Entity models use data annotations and validation attributes, such as [RegularExpression] and [StringLength], to verify data integrity before any database transaction.

4.6 Authorization

We started by implementing basic authorization using the built in Identity framework in ASP.NET Core, and scaffolding the required pages using aspnet-codegenerator.

To enforce authorization, instead of manually adding the [Authorize]

attribute to every controller, we used the AddAuthorization method in program.cs to add a fallback policy that requires users to be authenticated on every controller.
This has the benefit of ensuring every page is secured by default, and only pages that explicitly allow anonymous access will be accessible without authentication. We then added

the [AllowAnonymous] decorator to the home, login and register pages to allow users to perform authentication.

This was a good starting point but we quickly realized that we wanted more granular control over authorization. After looking over the different authorization strategies presented in the ASP.NET documentation, we decided to implement resource based authorization. Following the official guide (*Resource-Based Authorization in ASP.NET Core* , 2022) we implemented authorization with different permissions based on different operations. In general only the owner of a resource can edit or delete it, while any authenticated user can view it as long as the resource is marked as public.

After registering the authorization handlers, they are automatically injected when AuthorizationService needs a IAuthorizationHandler, and the right handler is chosen based on what model is passed. This makes it very simple to use authorization in the controllers. Simply call AuthorizeAsync with the User, Model and the requested operation (Create, Read, Update, Delete), and you get back an AuthorizationResult that either failed or succeeded.

4.7 Logging

In our application we have implemented a solid infrastructure for error handling and logging for data-related problems and security sections. For each method in our controllers, we have a try-catch block to detect and handle any exceptions during operations. When an exception occurs in a try-catch block, status code 500 is returned, which is an internal server error, and the exception message is formatted and written to the logging file for the specific controller and action where the error took place.

Inside the try-catch block in controllers, we have different checks for errors and null statements, depending on the controller actions. We handle those errors by returning a custom view for the specific given error code, log where the error happened, error message and set the HTTP response code.

The custom view for status codes contains user-friendly error messages to understand what has gone wrong and returns correct HTTP status code.

This robust system ensures that potential problems during data transactions and basic security logging, are correctly logged and communicated while maintaining the application's stability.

5 Conclusion

We explored the utilization of a Model-View-Controller framework for a learning platform with flashcards. The framework facilitated CRUD operations between the database and the user interface, as well as account management. We have experienced working as a group, managing a project using Github projects. It was challenging but rewarding to learn about and use Bootstrap for the design.

In conclusion, the framework was suitable for the development of a flashcards application in .NET.

6 Reference List

Bootstrap . (n.d.). Bootstrap · The most popular HTML, CSS, and JS library in the world. Retrieved October 31, 2023, from https://getbootstrap.com/

Bootstrap Icons . (n.d.). Bootstrap Icons · Official open source SVG icon library for Bootstrap. Retrieved October 31, 2023, from https://icons.getbootstrap.com/

npmJS . (n.d.). npm | Home. Retrieved October 31, 2023, from https://www.npmjs.com/

OpenAI. (2023). *ChatGPT* (GPT-3.5) [Large Language Model]. https://chat.openai.com/

Resource-based authorization in ASP.NET Core . (2022, June 3). Microsoft Learn. Retrieved October 31, 2023, from https://learn.microsoft.com/en-us/aspnet/core/security/authorization/resourcebased?view=aspnetcore-6.0

Sass . (n.d.). Sass: Syntactically Awesome Style Sheets. Retrieved October 31, 2023, from https://sass-lang.com/

Tailwind Swap Component — Tailwind CSS Components . (n.d.). daisyUI. Retrieved October 31 , 2023, from https://daisyui.com/components/swap/#swap-icons-with-flip-effect