NUS Orbital 2023

Study Stash

Toh Ming Chun, Marques Tye

Artemis

Motivation

Whenever midterm/exam season rolls around, students always scramble to source for past year practice papers, solutions and cheatsheets. Typically, these resources are located across multiple platforms, such as github repositories, google drives or from seniors' personal files. More often than not, these papers would also lack proper solutions and explanations.

Aim

We hope to make the stressful process of preparing for quizzes, midterms and finals easier by offering a centralised platform for students to access and share resources, including past year papers, solutions, cheatsheets and notes. (Cheatsheets, notes and solutions would be generated by the students)

User stories

1. As a student who is revising for exams, I want to be able to easily find the resources that I need, especially solutions for difficult questions. It would also be nice if I could be able to identify at a glance the relative difficulty of the papers and validity of the solutions.
2. As a student revising for exams, I would like to find users studying the same module and work on problems with them, and discuss the optimal solutions.
3. As a student who has satisfactorily completed the module, I would like to help the coming batches of students by uploading my compiled cheatsheets, notes and proposed solutions.

Tech Stack

Frontend

- React
  - React-hot-toast
  - React-select

    - Jotai (State Management Library)
- TailwindCSS (CSS Framework)
- Framer Motion (Animation Library)
- shadcnUI (UI Library Built upon Radix UI)
- Lucide icons & React-icons (Icon Libraries)

Meta framework

- NextJS

Database

- Planetscale (Using MySQL)
- Prisma (Object Relational Mapper)
- AWS S3 (PDF storage)

Tools

- Typescript
- Jest & React Testing Library (Unit Testing)
- Cypress (E2E Testing)
- ESLint (Linting)
- Prettier (Code Formatting)
- Google OAuth and NextAuth (Authentication)
- Vercel (Hosting and Deployment)

Design philosophy

Our aim was to create a web app that provides an experience similar to a desktop app, instead of a web page. To create that feeling, we set the main page to be non scrollable, and instead have specific scrollable components when needed.

We also wanted to make the app as accessible as possible without logging in, and thus instead of requiring authentication app-wide, we instead opted to have a fine-grained control over the features that require authentication, and the features that do not.

Features

For each feature, include description, implementation (how the code works, UML diagrams), challenges faced (e.g. jotai for atomic state management for the rating in bg issue) (e.g. for sort and filter functionalities, we lifted state into the URL for ease of sharing. challenge: preserve state on navigation)

Authentication

Description

As StudyStash is a personalised revision platform, we require user authentication to provide unique experiences for each user.

For authentication, we decided to use Google's OAuth to make it as easy as possible for users to sign up and log in. We also decided to use NextAuth to handle the authentication flow over implementing it ourself, as we highly valued security and wanted an established library to handle it for our app. It also provides a simple API to handle authentication, and also provide JSON Web Tokens (JWTs) which are .

Implementation

When a user logs in through Google OAuth, their name, email and picture is retrieved from Google and stored in the SQL database if the user is not already stored. A JWT is also generated and stored in the browser's cookies, and is used to authenticate the user for subsequent requests. The JWT is crytographically encrypted (via JWE) and is stored in server-readable-only cookies to prevent XSS attacks.

(insert authentication flow diagram)

Challenges

NUS Authentication?

File upload

Description

Users are able to upload resources, such as past year papers, cheatsheets and notes to the platform, with relevant metadata such as the module code, exam

type, year and semester. Users are also able to upload solutions to the resources uploaded by other users.

Implementation

Users are able to upload files by dragging and dropping the files into the dropzone, or by clicking on the dropzone to open the file explorer. The metadata is selected through many different dropdown menus, and the browser then checks that all fields are filled in and the file size and type is correct, before making several API calls to upload the file to AWS S3 and store the metadata in the SQL database. The PDFs are displayed in the browser using iframes from the AWS S3 link, but we have plans to implement a PDF viewer in the future.

(add UML diagram)

Challenges

Initially, the file names of the uploaded files were used to uniquely identify the files, but we later realised that this would cause issues if users were to upload files with the same name, causing the files to be overwritten. There were also a few invalid characters that AWS S3 was unable to accept. To solve this, we decided to generate a random CUID to be used as the file identifier for AWS S3, which is stored in the database along with the original file name.

Dark mode

Description

Users are able to toggle between light and dark mode, and the preference is persisted across sessions.

Implementation

The user's preference is stored in the browser's local storage in order to persist the theme across sessions. The theme is applied using TailwindCSS's dark mode feature, which allows us to easily apply different styles to different themes.

Challenges

As we are using Server Side Rendering, we set the initial theme to be dark mode, and then use a useEffect hook to apply the correct theme after the page has loaded. However, this results in a flash of dark mode before the theme is applied (if the user's preference is light mode). To solve this, we could choose to display the webpage only after the theme has been applied, but this would result in a blank page for a few seconds, which is not ideal.

Resource search

Description

Users are able to search and filter for resources by module code, exam type, year and semester.

Implementation

The data for the module code search was fetched using NUSMods API, and is searchable with and without module code prefix. Depending on the filters in place, different SQL queries would be made to fetch the relevant resources and its metadata. A tab UI element was used to allow users to switch between the different resource types (notes, question papers, cheatsheets). A sheet was used to open up each individual resource instead of using a new page, to provide a more seamless experience for the user to quickly browse through the various resources, and to do so without affecting the filters.

Challenges

We wanted the filters to be able to be shared via URL and persist across navigation. However, the state was controlled by React hooks, and thus would be lost on navigation. To solve this, we decided to lift the state of the filters and the sheet showing individual resources into the URL, and use the URL to determine the filters to be applied and the resource currently open.

Rating system (upvote/downvote)

Description

Users are able to upvote or downvote resources, and the ratings are displayed alongside the resources. The ratings can also used to determine the order of the resources when sorting by rating.

Implementation

Each individual rating is stored as a record in the SQL database, unique for every combination of user and resource.

Challenges

The state of the rating is handled by each individual rating component. However, when looking at each individual resource in the resource sheet, we can also see the rating of the resource from the resource list in the background. We could bubble the state up to the parent, but this means that the top level component would be rerendered every time state changes given how React handles state changes, which is not ideal as it would rerender the resource sheet. To solve this, we decided to use Jotai, an atomic state management library, to have fine

grained control over the rerenders and only rerender the rating components in the background and in the resource sheet when the rating changes.

Additional information

(ADD TABLE OF FEATURES ACCESSIBLE TO AUTHENTICATED VS NON AUTHENTICATED USERS)

(Insert list of API endpoints, and mention that all API calls are wrapped in try catch blocks)

Overall navigation flow

show diagram of routing

Timeline and Development Plan

Wireframe

Sketches were created using Excalidraw.

SEO and metadata

Open Graph Protocol

Twitter Cards

Sitemap

React Component tree

(Talk about how we created and utilised reusable UI components such as selects and buttons)

As we are using NextJS 13, we are able to utilise React Server Components, which allows us to create components that are rendered on the server, allowing us to fetch data directly from the database instead of having to call an API endpoint to fetch the data form the client. This also allows us to ship less javascript to the client, improving performance.

Model Entity Relationship Diagram

show database relationships

Type validation

Typescript was used to provide static typing to reduce bugs, and more importantly to utilise the Typescript Language Server Protocol to provide autocomplete and type checking in the IDE for a better developer experience.

Prisma

Prisma not only serves as an object relational mapper for us to interact with the database, but also extends the type definitions of the database to the application, allowing us to perform type checking on the database queries.

Zod

At the API level, we used Zod to validate the request body at runtime to ensure that the data sent by the user is of the correct type, as well as to extend the benefits of Typescript to the API level.

User experience

Interactive UI elements

All interactive UI elements demonstrate their interactivity through hover effects, animations and cursor changes, to intuitively express their interactivity to the user.

Instant feedback

To provide instant feedback to the user, we designed our app to be as responsive as possible, with loading states and error messages to inform the user of the status of their request.

Transitions and animations

Page transitions and animations were used to provide a more seamless experience for the user.

Testing

Unit Testing

We used Jest and React Testing Library to conduct unit tests. Unit tests were conducted to ensure that individual components of the application were working as intended.

End to End Testing

We intend to use Cypress to conduct E2E tests.

User Testing

We intend to engage users to conduct AB testing to evaluate design choices, and conduct usability testing to identify any pain points in the user experience.

Software Engineering Practices

Version control

The remote master branch is a protected branch containing contains the code used in production, and code cannot be pushed into it directly, instead requiring a pull request to be made before merging, as well as approval by the partner as the code reviewer, as well as passing status checks such as GitGuardian (see Security Measures)

For code review, we come together and discuss the changes to ensure everyone is on the same page.

Github Issues

Github Issues was used to keep track of any bugs in the application and features we wanted to implement. Labels were included to more easily identify the type of issues, be it bugs, features or something we would like to revisit.

Github Projects

We used Github Projects, which was highly integrated with Github Issues to have a spreadsheet like field of all the issues (and features) and their statuses, making it easy to keep track of the progress of the project.

Two-week Sprints

Security Measures

Updating dependencies

Dependabot was used to automatically create pull requests to update dependencies to the latest version, allowing us to keep our dependencies up to date to use the latest features, and more importantly to keep our app secure and free from security vulnerabiltiies.

Concealing secret keys

Secret keys, such as database (SQL and AWS S3) and authentication (NextAuth and Google OAuth) related information was stored in a .env file and excluded from version control using a gitignore file.

GitGuardian

GitGuardian was used to scan for any leaked secret keys in the repository, and would automatically create an issue in the repository if any were found.

Protect API routes with server side validation and authentication

To protect our API routes, we would also validate the authenticity of the request sent by the user using JSON Web Token authentication.

Linting and code formatting

ESLint was used to perform static analysis of the code to identify stylistic errors and potential bugs, while Prettier was used to format the code on save to boost productivity and ensure consistent code formatting. Config files for both were included in the repository to ensure consistency across all contributors.

CI/CD

Github Actions and Vercel was utilised to perform Continuous Integration and Continuous Deployment. The CI workflow was triggered on every push to the repository, and would run the production build, followed by unit tests and linting. The CD workflow was triggered on every push to the main branch, and would deploy the application to Vercel.

Limitations

No API rate limiting

Our APIs are not rate-limited, and malicious users could potentially spam our APIs, causing our database to be overloaded.

No rollbacks to database

In the event of a malicious attack against the database, there are no rollback mechanisms implemented to recover the database to a previous state.

Difficult to invalidate JWTs

JWT authentication was used instead of session-based to avoid needing to make a database query for authentication, but this means that in the event that an

unauthorised user gains access to another person's account, it is difficult for us to invalidate JWTs, as it is stored in the client instead of sessions in the server. However, as our application deals with non-sensitive data, this is not a major concern.

Challenges

Edge cases

1. long names causing layout shifts, solution: overflow