Educado - Web application for content management

Frederik Bode Thorbensen, Hans Erik Heje, Jakob Elias Gylling Saadbye, Markus Emil Hye-Knudsen, Ming Hui Sun, Sture Skar Svensson, Abdallah Ziad

April 11, 2023

AALBORG UNIVERSITY
DENMARK

Department of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Educado - Web application for content management

Theme:
Complex Back-end Developing

Project Period:
5. Semester

Project Group:
Group 2

Participant(s):
Frederik Bode Thorbensen
Hans Erik Heje
Jakob Elias Gylling Saadbye
Markus Emil Hye-Knudsen
Ming Hui Sun
Sture Skar Svensson
Abdallah Ziad

Supervisor:
Andres R. Masegosa

Copies: 1

Page Numbers: 77

Date of Completion:
April 11, 2023

Abstract:

This project aims to improve the standards of living for waste pickers in Brazil through education. This report describes how we improved the existing Educado platform while applying agile methods to improve our development processes and how to develop a larger product across multiple teams. Since the entire class of software 5 worked on the same project, all of the teams involved had to iteratively work towards an improved Educado platform. To streamline the process, we employed various methods from the Scrum framework, including sprint planning, sprint reviews and working around user stories to continuously provide valuable increments. Taking over from an existing project, we identified a lot of issues that had to be tackled. Through extensive refactoring guided by principles from clean architecture in both the front-end and back-end of the application, we have managed to create a more maintainable codebase for future developers to come. Using static code analysis tools, indicated improvements in the code health on all metrics compared with the initial codebase. Thus allowing for a maintainable and flexible codebase that allows for easier development for next year's semester students.

Preface

We would like to thank our supervisor Andres Masegosa for his inspiration to the project and knowledge about coordinating with teams.

The people involved in this project

Before we start to describe the project in all its details, we wanted to give an overview of the people and teams involved in this project. Since some of the people/teams will be mentioned or referenced later in the report, this serves as a point of reference and for the readers understanding.

Stakeholders

The stakeholders in this project, are students from University of Brasilia and the original developers of the Educado platform:

- Jacob Vejlin - Bachelor graduate from AAU
- Daniel Britze - Bachelor graduate from AAU

Product Owner

- Mateus Halbe Torres - Research Assistant, department of electronic systems AAU

Development Teams

Each of the development team, consists of 6-7 software students (5. semester, AAU)

- Group 1: Team Half full stack, Complex Front-end & Back-end
- Group 2: Team Sharp Deluxe, Complex Back-end
- Group 3: Team Cows, Complex Front-end

Contents

1 Introduction **1**

2 Limitations **2**

3 Background **3**

3.1 The platform in large . . . 3

3.2 Initial state of the web-application . . . 4

3.2.1 A look at the initial front-end . . . 4

3.2.2 A look at the initial back-end . . . 5

3.3 Important concepts in this project . . . 7

3.3.1 Roles in Scrum . . . 8

3.3.2 General Concepts . . . 8

3.3.3 Events in Scrum . . . 9

4 Sprints **10**

4.1 Prior to the sprints . . . 10

4.2 Sprint 1 . . . 12

4.2.1 Sprint 1 Planning . . . 12

4.2.2 Daily Scrum . . . 13

4.2.3 Cross-team coordination . . . 13

4.2.4 Increments . . . 13

4.2.5 Sprint 1 Review . . . 14

4.2.6 Sprint 1 Retrospective . . . 14

4.3 Sprint 2 . . . 15

4.3.1 Sprint 2 Planning . . . 15

4.3.2 Daily Scrum . . . 16

4.3.3 Cross-team coordination . . . 16

4.3.4 Increments . . . 16

4.3.5 Sprint 2 Review . . . 18

4.3.6 Sprint 2 Retrospective . . . 18

4.4 Sprint 3 . . . 19

4.4.1 Sprint 3 Planning . . . 19

4.4.2 Daily Scrum . . . 21

4.4.3 Cross-team coordination . . . 21

4.4.4 Increments . . . 21

4.4.5 Sprint 3 Review . . . 21

4.4.6 Sprint 3 Retrospective . . . 22

4.5 Sprint 4 . . . 22

4.5.1 Sprint 4 Planning . . . 23

4.5.2 Daily Scrum . . . 23

4.5.3 Increments . . . 24

4.5.4 Sprint 4 Review . . . 25

4.5.5 Sprint 4 retrospective . . . 25

4.5.6 Collaborative retrospective . . . 26
4.6 Sprint 5 . . . 27
4.6.1 Sprint 5 Planning . . . 27
4.6.2 Daily Scrum . . . 28
4.6.3 Cross-team coordination . . . 29
4.6.4 Increments . . . 29
4.6.5 Sprint 5 review . . . 29
4.6.6 Sprint 5 Retrospective . . . 29
4.7 Sprint 6 . . . 30
4.7.1 Sprint 6 Planning . . . 30
4.7.2 Daily Scrum . . . 30
4.7.3 Cross-team coordination . . . 30
4.7.4 Increment . . . 31
4.7.5 Sprint 6 Review . . . 31
4.7.6 Sprint 6 Retrospective . . . 31
4.8 Combined user story diagram across groups . . . 33

5 Front-end Implementation 35
5.1 Migration from JavaScript to TypeScript . . . 35
5.2 Unified Data Fetching Strategies . . . 35
5.3 Utility First Style Libraries . . . 35
5.4 Global State management . . . 36

6 Back-end architecture and implementation 36
6.1 Overview of the back-end components . . . 36
6.2 Following a clean architecture . . . 38
6.3 Dependency injection . . . 38
6.4 Clean folder structure . . . 40
6.5 Call flow . . . 43
6.6 Back-end implementation . . . 44
6.6.1 Content creator application . . . 44
6.6.2 Courses . . . 48
6.7 Data models . . . 49
6.7.1 Authentication . . . 50
6.7.2 Role-based security . . . 53

7 Code quality 56
7.1 Static code analysis . . . 56
7.1.1 CodeScene . . . 56
7.1.2 SonarQube . . . 61
7.2 Testing . . . 64

8 Discussion 66
8.1 Agile learning . . . 66
8.2 Current state . . . 66
8.3 Future works . . . 66

8.3.1 Reasons for each feature . . . 67
8.4 Working with scrum . . . 67
8.4.1 Product owner and stakeholders . . . 67
8.4.2 Scrum in practice . . . 68
8.4.3 Scrum master & daily scrums . . . 68
8.4.4 Sprint backlog . . . 68
8.4.5 External Collaboration . . . 68

9 Conclusion **69**

10 Appendices **71**

A Group contract **75**

1 Introduction

Written in collaboration with all groups

One of the world's largest landfills lies in Brazil [1]. The landfill was closed in 2018 which is a positive development in regard to the current climate crisis and the general health of the people working and living in or near the landfill [2].
When the landfills were closed, the waste pickers who lived and worked there were instead provided with jobs at the newly created recycling centers to sort the waste. This has led to a lower income for the waste pickers in general, even though they were hired at the recycling facilities. Furthermore, the payment went from a daily to a weekly salary, which meant that the waste pickers had a harder time managing the money that was available to them. The waster pickers do not have many other options as many of them do not have any education so they can get another better paying job [3, p. 3].

This problem identification has led to the creation of the *Educado* project, a digital learning platform for waste pickers in Brazil. It began in 2019 when students at Aalborg University collaborated with students at the University of Brasilia to create a mobile application that can be used to educate waste pickers.
In order to complete this, an app was created that apart from providing basic education also gives certifications for completed courses. This is meant to help these people improve their standards of living. The app should be intuitive and contain audio and video clips instead of text when possible, to make it more available for people who cannot read very well.
A front-end for the Educado platform was also created, allowing "Content Creators" to build, modify and publish courses all from a web browser [4].

This year, three groups at Aalborg University are collaborating to continue this project and its vision. The project's goal is to clean up and refactor the existing code and database, as well as redesigning the mobile, the back-end and the web application while adding new functionality.

End of collaboration

2 Limitations

Written in collaboration with all groups

In this section, we describe the limitations that dictate the scope of the project. The project description has set a baseline for the project limitations:

> *Improve the Educado platform and transform it into a great functional Mobile Education solution for Waste Pickers to be tested in the field by the end of the semester in Brazil [5].*

Working from an existing project

Since this project is based on an existing solution, we are limited to working with the same underlying technologies as the original solution. This means that all further implementation will be built with:

- MongoDB
- Express (Node)
- React
- React Native

The only differentiation in the technology stack that has been made, is adding type support for the React front-end with Typescript, as well as some minor changes concerning the styling of the user interface in the mobile application.

Continuous learning

Our course on agile software development ran concurrently with our project, so some sprints will introduce agile development methods that do not appear in previous sprints. As such this project involved a steep learning curve regarding many new concepts and methods of agile development for multiple groups.

Deployment

Since the project is already an existing working platform, we are not going to work with the deployment of the system, and as such, further discussion about the current deployment choices will not appear in the report.

Dependencies

As described in the introduction, all of Software 5 is going to work on the same project this semester. Therefore, the improvement of the existing platform is divided between the three teams, where each group has focus on a specific domain of the project. Group 1 will work on both back-end and front-end of the mobile application and assist with creating a link of information flow between the other two groups' implementations. Group 2 will focus on both back-end and front-end for the Educado platform for content creators of educational courses for the waste pickers in Brasilia. Group 3 will work on creating an engaging mobile application for the waste pickers, and will primarily be in charge of the front-end of the mobile application. Therefore, a main concern of ours in this project is to balance the work between fulfilling stakeholder wishes and resolving cross-team dependencies. This means that having a functionally aligned platform in every team is more important than a single group providing new features for a single aspect of the platform.

End of collaboration

3 Background

This section explores the initial state of the Educado platform as we started this project. Its structure and any issues we identified are discussed. Additionally, we provide the reader with a list of concepts of relevance to the semester.

3.1 The platform in large

Written in collaboration with all groups

To get a sense of the entire system and how the components are laid out, we have come up with the following architectural diagram which illustrates the system at a high-level (see figure 3.1). The diagram is coloured according to each group's main responsibility in the project. The part of the diagram that is coloured orange has been our group's responsibility. At a high level, the Educado platform consists of three parts: From the left, the **Creator Studio** is a web front-end for creators to create content for the platform. In the middle, the **Content platform** is a common back-end used for the creator studio and mobile app to make them functional. The back-end can be seen as having two main logical parts. One part deals with everything related to content-creation and the mobile part is concerned with the mobile-app users such as the courses they have been enrolled in and their progression in them. The final part is the **Mobile-App** which is used by the waste pickers to consume the content. The three main parts reside in three different code repositories communicating via restful HTTP calls.

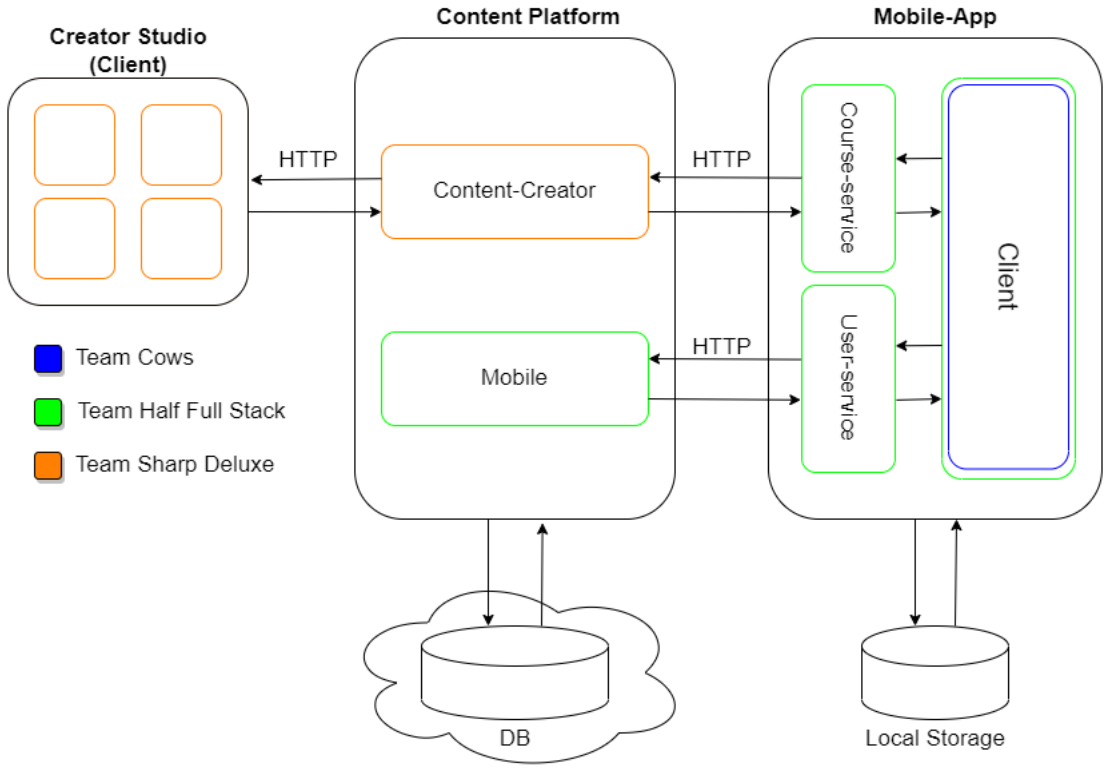


Figure 3.1: A high-level architectural diagram of the Educado system

End of collaboration

3.2 Initial state of the web-application

Before diving into our project and how it unfolded, it is a good idea to get a good grip on the existing work to get an idea of where the project is now and where to go from here. In the general sense, the web-application part is separated into two parts: a front-end for displaying and creating content, and a back-end for handling all the logic and persisting data to the database.

3.2.1 A look at the initial front-end

As discussed with the product owner, the initial front-end application had not had any updates in a fair amount of time before the SW5 project started. That naturally led to the fact that many of the underlying packages and project dependencies were either outdated or unmaintained. Therefore, one of the first topics we discussed when we got access to the initial front-end application, besides the mono-repository structure in GitLab, was the use of deprecated libraries.

The front-end library used for the project: React, was running version 16.13, compared to the latest stable version (LTS), React 18.2. As discussed with the team, continuing our work from a React version so outdated was not a good idea. Especially since one of the most influential changes to React was introduced in v. 16.8, i.e., going from class-based to function-based components (Hooks). When we looked at the initial application, we noticed that the previous developers had been using both design patterns within the same project, which is frowned upon in the community. Having chosen to continue our development of the front-end application with the recommended hook-based approach, this meant that a couple of the project dependencies that were only available for class-based projects had to be replaced, such as:

- **react-beautiful-dnd**: Drag and drop package that is only implementable with the class-based approach.
- **Redux**: class-based global state management package.

Another topic of discussion was the use of inconsistent code-splitting practices in the project. Since React is very unopinionated about the project structure, it is highly advantageous for development teams to adopt stricter code-splitting and project structure practices, such as having dedicated directories for pages, components, hooks etc. A similar structure was found in the initial project, but components could be found in various locations outside the components folder.

Besides the development practices, dependencies and code smell around the application, the application was also lacking some of the expected functionality, such as the option to register as a new user to access the parts of the application, accessible only to authenticated users. To gain access to the application you had to create a user directly in the database and only with a specific user id, to be able to sign in to the application.

To summarize the initial state of the front-end application, it is fair to say that the application requires some work in order to scale which can be seen at figure 3.2 and figure 3.3. As mentioned above, the Achilles heel of this application as we took over was the lack of standardization in an unopinionated framework. That being said, the underlying implementation is of fair quality and when we apply some standardized patterns and update the underlying packages, the application will provide an adequate starting point for our team.

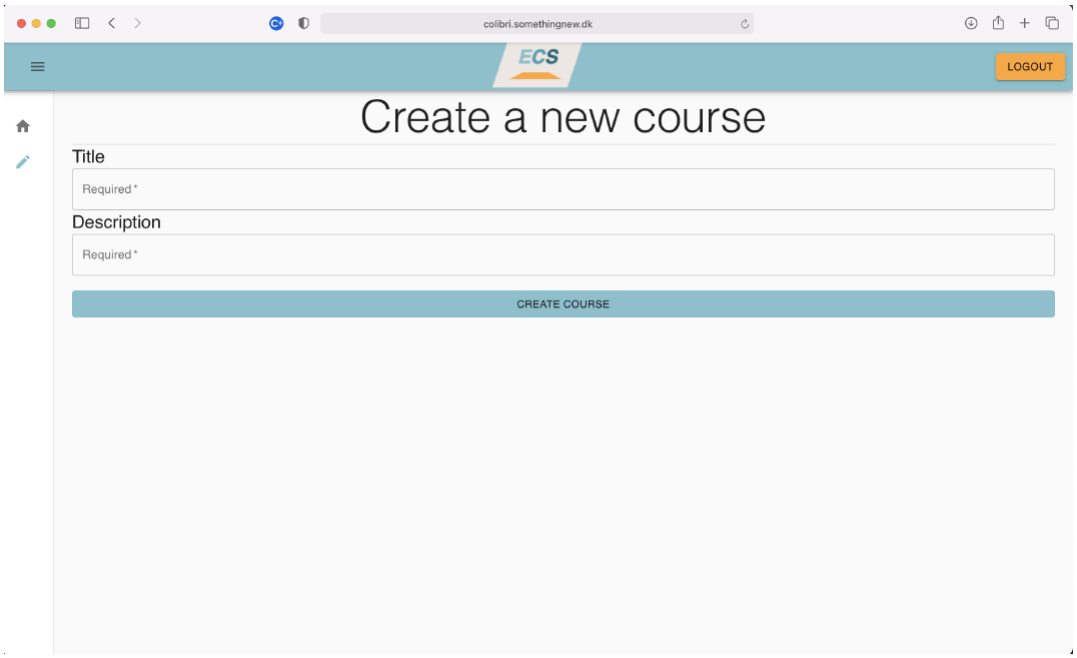


Figure 3.2: Create course in the previous state of the front-end

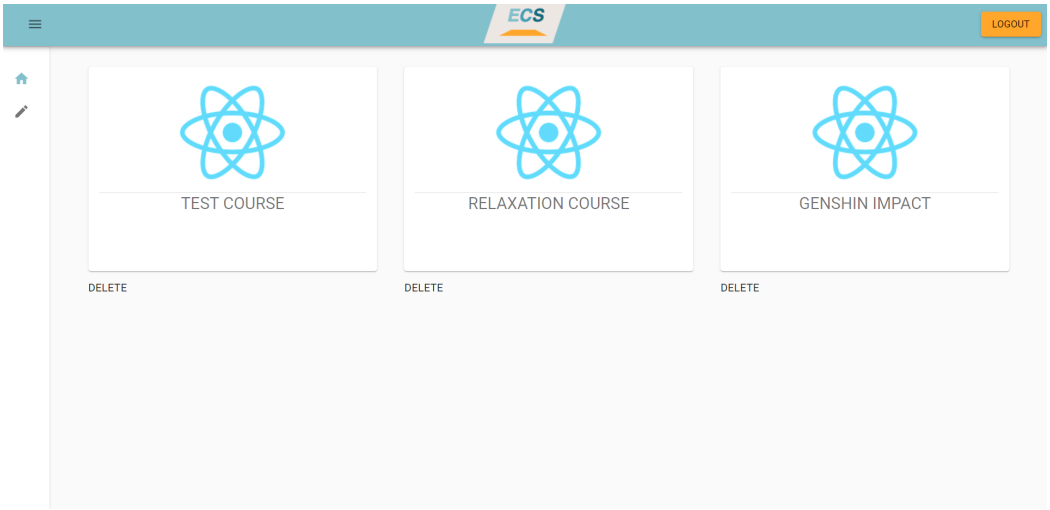


Figure 3.3: Dashboard in the previous state of the front-end

3.2.2 A look at the initial back-end

To get a sense of the initial state of the back-end, let us look at how it is structured internally. The three main parts that make it up is found within the routes folder in three files. In figure 3.4 shows the overall structure of files in the back-end. The first file "authRoutes" hosts all the logic for authenticating users through OAuth with google. It makes heavy use of a third-party library for actually dealing with authentication. Inside the "bucketRoutes" file is mainly integration code for saving images and videos to an AWS storage bucket. The last file "courseRoutes" hosts all the remaining logic for manipulating the courses.

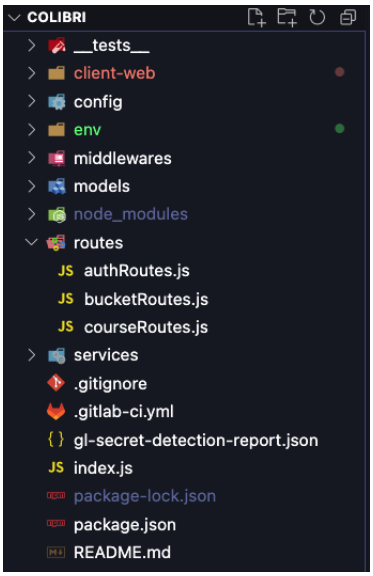


Figure 3.4: Initial folder structure

One of the apparent things, looking at the folder structure, is that it reveals very little of what the system is actually about. This is a common mistake when starting out a project, that the project structure is guarded by the terminology of framework providers. In the book *Clean Architecture* by Martin Roberts [6], he talks about having a screaming architecture, one which clearly reflects the intent of the system. The key to achieving a screaming architecture is to use the terminology of the domain rather than technical terms when naming folders and files. The benefit of this is that it becomes way easier to navigate within the project when it is structured according to different responsibilities of the system.

Another issue with the back-end is that there is very little reuse of code. This is due to the fact that all logic that needs to be carried out when hitting a single endpoint is very coupled to that specific endpoint, which leaves very little room for reusability. Another issue when stuffing all logic close to the endpoints is that it becomes very hard to test individual parts in isolation. As it is, the only way to test a bit of logic is to go through the front-end interface, login with google to get behind authentication, and then hit the API endpoints in debug mode. This is a very slow way of testing and will only lead to slower development time when bugs begin to show up. The devil in all this is the very tight coupling. The tight coupling is apparent everywhere in the codebase and it would take a lot of time to refactor it all at once. An incremental approach, slowly making the codebase less coupled would be our best option if we are to also make new features on the platform.

Without criticizing the existing codebase too much, another point to make is the lack of standardization and consistency on the API-endpoints. The most commonly used HTTP methods include: `GET` for retrieving resources, `POST` for creating a new resource, `PUT` for updating an existing resource and `DELETE` for deleting a resource. The standard is to use these methods to express the action on the resource. So instead of suffixing the existing endpoints with create, update and delete, it should instead use the corresponding HTTP

```
app.post("/api/course/create", ...)
app.post("/api/course/update", ...)
app.post("/api/course/delete", ...)

app.get("/api/course/getall", ...)
app.get("/api/course/eml/getall", ...)

app.post("/api/course/update/title", ...)
app.post("/api/course/update/description", ...)
app.post("/api/course/update/category", ...)
app.post("/api/course/update/published", ...)
```

Figure 3.5: Initial API endpoints

method. In this case the more correct methods would
be, in order, post, put, delete. Instead of suffixing the endpoint with "getall" a better approach is to pluralize the resources so that `"/api/course/getall"` becomes `"/api/courses"`. With the `GET` method in-front, it is easy to get the intent that this endpoint will retrieve all courses. To retrieve a single resource it is common to have a request parameter that identifies the exact resource. So the endpoint for retrieving a single course would look like this `"/api/courses/:id"` where the `:id` part is replaced with some identifier for the course. The last part to touch on is the existing way of updating resources. Almost every field on the course has a specific endpoint to change just that one field. Adding additional fields that should be updatable, would mean creating new endpoints for each one. This is obviously going to make an explosion of new endpoints to be managed from any client code hitting them. The better way would be to send a collection of related changes inside the request body to a single endpoint.

The existing data model that makes up the entire back-end data is pretty straightforward. It consists of four schemas: A **User** for storing account details of a content creator, a **Course** with details such as title, description and cover image, a **Section** for sub-dividing content with info as description and title and lastly a **Component** schema that stores either text, audio or video wherein the case of audio and video it stores a key to locate a resource within Amazon's storage bucket. Looking at the relationships between them, the user that is connected to multiple courses is the author of the course. Each course has many sections and each section can have many components.

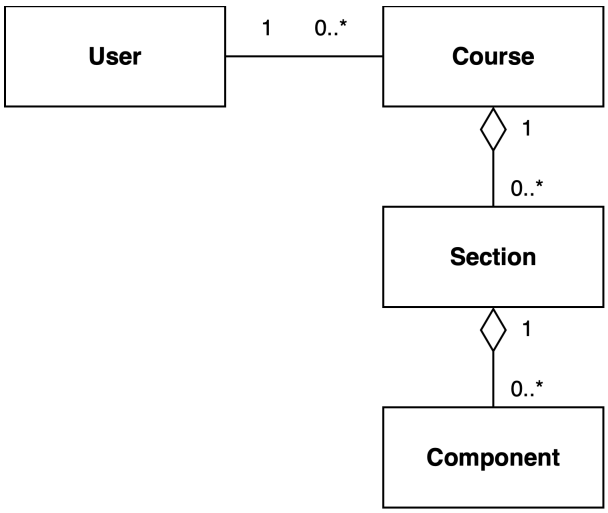


Figure 3.6: Initial data model

To summarize the initial state of the back-end, there are quite some structural changes to be tackled in order to decouple the existing codebase and make it more testable. Then some effort needs to be made in order to make the API-endpoints more standardized and consistent so that future developers, hopefully using the API, are not scratching their heads over confusing endpoints. The existing data model is quite simple and understandable, so it should not be an issue to add new schemas to it when the need comes.

Written in collaboration with all groups

3.3 Important concepts in this project

In this project, we are working as members of a development team following Agile practices while learning about them. This section briefly describes key concepts of the Agile framework called Scrum that we are meant

to work within. For in-depth explanations of these concepts, explore original resources on Agile [7] and Scrum [8].

- The **Agile Manifesto**, through its twelve principles, serves as a guideline for software development.
- **Scrum** is a specific Agile framework that provides structure to the development process. This structure is centered around specific roles [9] and events described below.

3.3.1 Roles in Scrum

Key actors and their roles in Scrum.

- The **Scrum Master** is the head of a Scrum team. First among his peers of developers, he is responsible for the effectiveness of the team.
- The **Product Owner** (PO) ensures that the team produces a valuable product by being the link between developers and stakeholders.
- The **Developers** are the members of the scrum team creating the product.
- A **Scrum Team** consists of a Scrum Master, a PO and the Developers.
- The **Stakeholders** are people outside of the scrum team, who have knowledge of or an interest in the product. They are represented by the PO and should play an active role during Sprint Reviews.

3.3.2 General Concepts

The following concepts fall into a more general category.

- The **Product Backlog** is a list of all the items or work that needs to be done to achieve the desired state of the product.
- A **Sprint Backlog** is a subset of the Product Backlog containing the work required to fulfill a sprint's goal.
- An **Increment** represents the finished valuable work the Developers have completed during a sprint.
- A **Definition of Done** formally describes the state an Increment has when its quality is what is required for the product. It is about the activities needed to ensure a certain level of quality of the work being done on the backlog items before they can be considered an Increment.
- The **User Story** is not described in the Scrum Glossary, because it is not a mandatory part of Scrum [10]. However, perfectly compatible with Scrum, a user story aims to express requirements from a user's perspective. It is what the user needs from the system. In Scrum it is the PO's responsibility to relay this information to the developers [11]. In this project, we, the developers, have made the user stories based on our understanding of the system and then confirmed them in talks with the PO.
- The concept of **Scaling** becomes relevant when a developer team or software product grows in size and complexity, possibly becoming unmanageable. A scaling framework attempts to set up the rules or structure for managing these challenges. One such framework, Nexus [12], minimally builds upon Scrum to enable three to nine teams to work together on a single product.

3.3.3 Events in Scrum

Defining activities in Scrum that have been central to this project.

- The **Sprint** is a time-restricted period during which all the other events take place.
- **Sprint Planning** marks the start of a sprint, where the scrum team chooses from the Product Backlog what is most valuable to work on during the coming sprint.
- **Daily Scrum** is a daily 15 minute event to inspect previous work and lay out the work plans for the following day.
- At **Sprint Review** the Increment is inspected by the Scrum Team and Stakeholders, its value assessed and the Product Backlog updated.
- During **Sprint Retrospective** the Scrum Team evaluates the ending sprint and tries to find ways to improve future sprints.

End of collaboration

4 Sprints

The purpose of this chapter is to introduce the sprints that play a central role in this project, where the main focus area was learning to work within the Agile guidelines and the Scrum framework. Each sprint had a two-week duration. We describe how each sprint unfolded and discuss the conditions that were interesting to address.
Additionally, the sprint sections will deal with working with other groups including meetings and managing dependencies.

We describe our sprints in the following format:

- Overall goal: Introducing the theme of the sprint or the overall sprint goal for all groups
- Planning: What happened during the sprint planning
- User stories: The user stories that were approved at planning and worked on during the sprint, and the reason for their importance
- Daily scrum: How these meetings went and how it affected our workflow
- Our increment: A high-level description of what we managed to implement or worked on
- Review: What happened at the review, approved or declined increments
- Retrospective: Reflection on the sprint and what we did to improve

4.1 Prior to the sprints

Each group in our team had to choose from a list of project proposals which part of the Educado platform to work on during the project. We chose to focus on the web-application that handles content creation, which, to us, meant working on both front-end and back-end. As we signed up for this project to work with complex back-end, our Implementation section will focus on the back-end, leaving any front-end development as a bonus.

First off we were introduced to the product owner, a research assistant from the department of electronic systems, who would relay the requirements from the stakeholders to us and the rest of the groups.
The stakeholders that initially started this project made a short run-through of the existing codebase to get us all up to speed on all the different components within the codebase. After that, roughly two weeks were spent on understanding the existing codebase to get familiar with what was already done.

First meeting with the product owner

To get an initial understanding of the problem, we met with the product owner to discuss and align our ideas for the system.
The main thing that we discussed was how new content creators should join the platform. The existing solution for this was to ask the two stakeholders of the Educado project which are Daniel Britze and Jacob Vejlin Jensen to create a new user which clearly would not suffice if the project was to have many content creators signing up. The idea we agreed on, was to have users apply for being content creator on the platform, and then let an administrator handle the approval and rejection of these. This way we could still control who would join the platform. For reducing the amount of administrative work, we came up with the idea of letting content creators sign-up through an institution, that way, if the content creator has an email that matches the domain

of the institution's email, they would automatically get rights to the platform without the need for sending an application first.

The figure below illustrates the flow of signing up as a content creator:

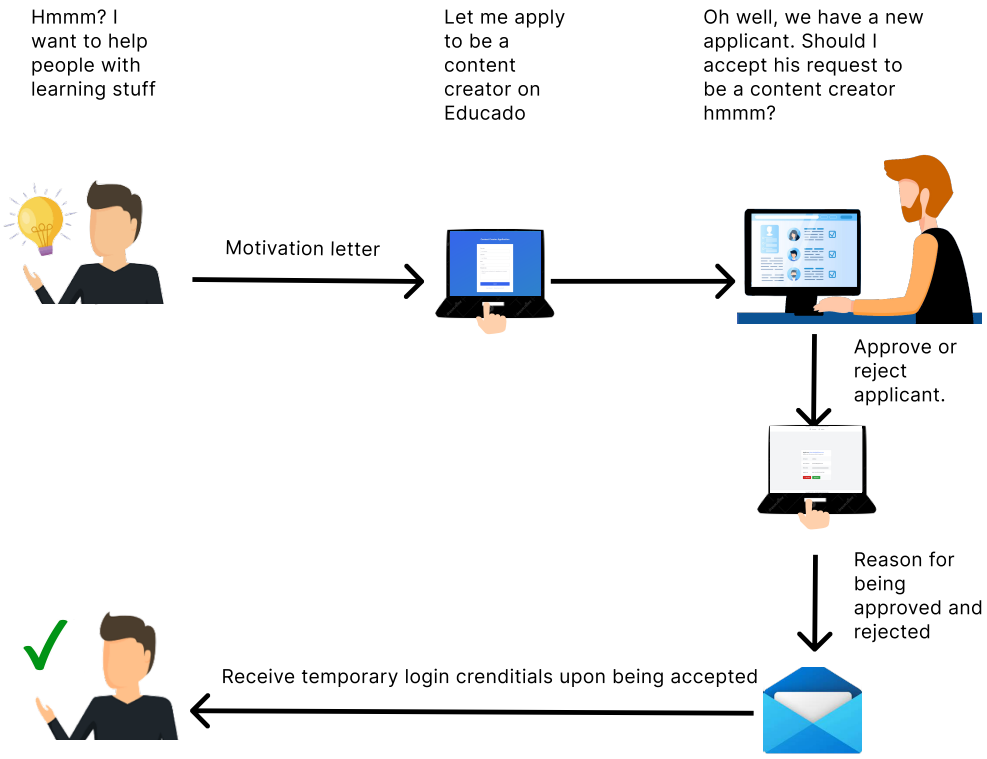


Figure 4.1: Content creator onboarding process

In conclusion for the meeting, the focus would be to first get the basic sign-up flow ready, before moving to the more advanced feature of signing up through an institution. This led us to come up with the first user-stories for the web-application.

Identifying the actors of the system

In addition to the first meeting with the product owner, we thought of the end users of the web-application and what use-cases they might carry out. For that we identified the following actors to be:

- Educado admin: The Educado admin role would be to administer the platforms content and approve applications from potential content creators.
- Content creator (CC): The content creators role is just as the name implies, to create content for the platform.
- Editor: Editors would be a sub-role of content creators who can only edit the courses that he is invited to collaborate on.

4.2 Sprint 1

The first sprint started on 22 September 2022 and focused on getting the project started. Researching agile development techniques, understanding how the workflow of a larger collaborative project between groups is structured, getting an understanding of what Educado is and learning about the problems it faces as a whole were all critical steps in the process of getting started. Additionally, we were instructed by our semester coordinator, Daniel Russo, to appoint a group member to the role of Scrum Master. At the start of each following sprint, we would appoint a new Sprint Master.

4.2.1 Sprint 1 Planning

During the first sprint planning, we went over all of our user stories with the product owner and discussed each one in turn. In collaboration with the product owner, we then selected the user stories that we felt were the most important and had to be worked on. These user stories were added to our sprint backlog. We agreed that the group goal for this sprint was to implement the functionalities for signing-up new content creators. The following subsections will cover the user stories and their importance to the project.

User story 1

As a Content Creator I want to sign up for the Educado platform using my name, email and a motivation application, so I can either join an institution and make content or make content as an individual.
A sign up platform for a Content Creator (CC) was heavily needed in order to allow new users to become part of the Educado platform. A sign up form was not implemented in the project we received, so we had to come up with an original design and requirements for the application that should satisfy the needs of a content creator, as well align with the product owner's expectations.
The Educado platform will house two different types of content creators. The first one is an individual who is not affiliated with an organization or institution, the other one is affiliated with an institution. To prioritize and efficiently code the content creator application form, we will start with the implementation of the sign up for an individual. Allowing sign ups using a name, an email and a motivation will provide a clear indication of who the person is, how to contact them and why they want to be a content creator.

User story 2

As a content creator I want to sign into the platform using my email and password
Once an application process is established, the approved CCs will need a way to sign-in. The initial system only allowed a pre-registered user to sign in using Google OAuth, which would not suffice in a larger system, where user data is stored in a database.

User story 3

As an Educado admin I want to receive and review content creator applications to control onboarding of individual content creators.
Content creator application form will be received and reviewed by an Educado admin, who are the employees that oversee the Educado platform. Receiving an application should be through another page that is exclusive to the Educado admin where they can approve or reject the applicant. The system will allow the admin to send an email to notify the applicant whether their application was approved or denied. This way of on-boarding content creators allows them to create an account that has no permissions to access the site, while awaiting approval from the Educado admin.

User story 4

As a content creator I want to have a dashboard overview of courses, so it is easier to access them.
Previously, the original dashboard for the web application was a list of boxed courses which were correctly placed but the design was not consistent which can be seen at figure 3.3. Therefore we want to refactor and add new features to the courses page in order for a better design which provides easier access to them. After content creators have successfully logged into the web application they should have a dashboard view with a list of courses. The view is going to be different depending on which user is signed in.

4.2.2 Daily Scrum

At this point, we had not yet been introduced to the concept of a daily scrum so we went with sticking to the ways of having general meetings as we had done in previous projects.

4.2.3 Cross-team coordination

At this point in time there was none or very little communication between the groups. Any talks happened spontaneously when the need arose.

4.2.4 Increments

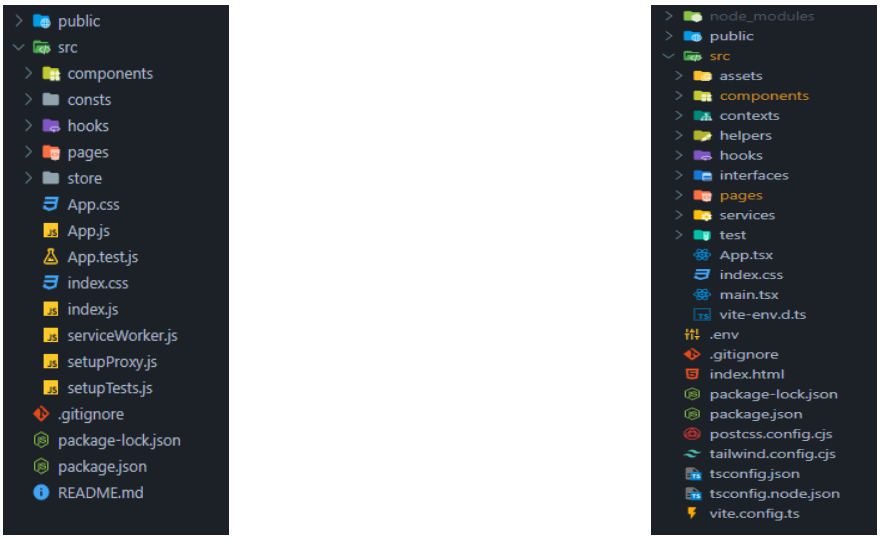
After having identified a lot of issues with the initial web-application 3.2 we started tackling some of the structural issues first, as they would lay the groundwork for the future development of the project.

Splitting up the repository into two smaller applications

The initial web-application had both the front-end and the back-end in the same repository. Having the front-end and the back-end application inside the same repository is an issue to be tackled if we were going to scale up this project. It would be better if these were split into two different repositories so that we could work on them isolated from each other. Before making the change we had to have the stakeholders create the repository for us, as we did not have the permissions to do so ourselves. This did not pose the biggest of hassles but non the less a slight inconvenience that involved some back and forth communication. Having the front-end and back-end clearly separated meant that it was really easy to divide our work internally in the group.

Refactoring the front-end

As discussed in Section 7.1.1 about the initial state of the front-end application, several areas needed some attention. Therefore with the creation of dedicated repositories for the front-end and back-end, we decided to address some of these areas. So during the first sprint, we added TypeScript support to the application and wrote the interfaces required for the initial application to run with the stricter type system introduced. Once we had the application in a steady state, we started to refactor the project structure to adhere to several React project management guidelines, such as project structure, and better code-splitting practices and rewrote most of the components from class-based to function-based. This increment resulted in a project, better suited for the team's development efforts and a cleaner codebase to continue our work.



(a) Initial (b) Refactored

Figure 4.2: Project structure before & after refactor

Refactoring the back-end

The initial state of the back-end was an unmaintainable monolith that didn't follow any standards, as is discussed in section 3.2.2.
We realized the issues that this carried while attempting to implement the user stories. Therefore it was of vital significance that the back-end would be refactored into something maintainable and flexible. This encompasses developing clean architecture, something that will be delved into in section 6.2.

4.2.5 Sprint 1 Review

During our first sprint review, each group presented their user stories and the increments they were able to deliver based on them. In terms of our user stories two of our four tasks were approved as increments, namely user story two regarding sign-ups and four regarding the content creator dashboard overview. The remaining two user stories were moved into the sprint backlog for the second sprint. We were unable to complete the last two user stories due to the workload involved with refactoring the repositories. During the sprint review we had a discussion with the product owner regarding some of the user stories related to creating courses and pushing them out to the app. These will have some dependencies which involve the other two groups, and should be prioritized in sprints where the other groups work on related user stories.

4.2.6 Sprint 1 Retrospective

After receiving feedback from the product owner, we need to reflect on what went well, what did not go well and what we can improve upon in the next sprint. We discussed this within our own group, and continued to sprint planning, bringing with us any insights gained and starting the sprint cycle anew.

What went well

During the sprint, the group was able to accumulate a good understanding of the work ahead. A lot of knowledge was gained through researching the codebase of the initial product. Furthermore, we were able to add value to the product by refactoring both the front-end and back-end.

What did not go well

Towards the sprint review, we had to understand the code structure we inherited which took more time than expected. We could not finish all four user stories which resulted in user stories 1 and 3 being incomplete and delayed.

While discussing each user story in planning and deciding whether or not they would be applicable to the first sprint, too much time was spent discussing each story irrespective of their size and complexity. This forced us to defer discussing acceptance criteria until a meeting with the PO the next day. During this meeting we discussed user stories again, but we did not specifically decide upon how to implement acceptance criteria for the user stories.

What should we improve towards the next sprint

We felt that we maybe had put too many user-stories on our plate for the first sprint, without really considering the time it would take to implement each one. The reason for this overestimation could be that we had not accounted for the fact that, although the sprints last for two weeks, the actual time available for implementing features during a sprint is much less. For future sprints, we want to be more realistic when considering what is reasonable to implement, in the actual time-span that we have. This means we will have to take a more conservative approach when discussing tasks with the PO.

4.3 Sprint 2

In our second sprint, we were still new to Agile and the Scrum framework. We simply continued working on our sprint backlog without having a clear idea of what the other groups were working on or agreeing with them on an overall sprint goal (since we were not yet aware of this as a Scrum concept).

4.3.1 Sprint 2 Planning

The transition from sprint one review and sprint one retrospective to the second sprint planning was an unstructured and informal way of doing the scrum framework process because only two out of three groups were able to get feedback on their user stories while the remaining one could not. Thereafter, user stories were agreed upon between the group and the product owner. Ultimately four user stories were chosen for the current sprint where each of them has at least one relation to each other. Choosing stories which are in close relation infers the constitution of an overall sprint goal, where each single user story can be considered part of a larger whole.

User story 1

As a content creator I want to sign up and into the web application using my name, email and motivation application, so I can either join an institution and make content or make the content as an individual.

This user story was described in Sprint 1, Section 4.2.1

User story 2

As an Educado admin I want to receive and review content creator applications to control onboarding of individual content creators.

This user story was described in Sprint 1, Section 4.2.1

User story 3

As a content creator I want the ability to create new courses, so I can expand my portfolio of courses on the platform.

Having courses created by content creators is one of the most essential functionalities of the web application, since if no courses exist there is no content for the mobile app. Creating courses should require a title and a description. As a course is created, an edit option should be available where a content creator can edit several things, such as adding a cover image or giving the course a specific category.

User story 4

As a content creator I want to be able to add additional sections to an existing course and edit them, so that related educational material is grouped in a larger course.

For content creators to structure larger courses on a given topic, they will want to divide the content of a course into sections addressing a particular area within the overall course topic. These sections will need to contain exercises that test the users' understanding of the course material.

4.3.2 Daily Scrum

During this sprint, we were introduced to the concept of Daily Scrum in our course on Agile Software Development. We tried to hold daily scrum meetings but were not always able to stay within the 15 minutes time slot. Therefore, not all members of our group got to explain how their work was proceeding. As a result, we did not use daily scrum to its full potential and did not have a clear picture of whether our increment would be ready for the sprint review.
Some of our work days were only a few hours long, so on these days, we felt that a scrum meeting would take away time from the actual work that needed to be done.

4.3.3 Cross-team coordination

As we were introduced by daily scrum, we did not entirely focus on other groups because we had to refactor more from the initial codebase in the back-end and front-end. Cross-team coordination was not a massive focal point for our group during this sprint, as there were no clear dependencies between the groups at this point.

4.3.4 Increments

During this sprint, we managed to deliver two valuable features as increments based on the chosen user stories. These new features are onboarding the prospective content creators and receiving and reviewing their applications from them. The focus for sprint two was to prioritize the user stories that were started, but not finished during sprint one. This is why our increment is limited to user stories one and two.

Onboarding content creators

Onboarding content creators was a work in progress from sprint one, where we did not manage to meet the definition of done for the user story. In this sprint, however, we finalized the onboarding process.

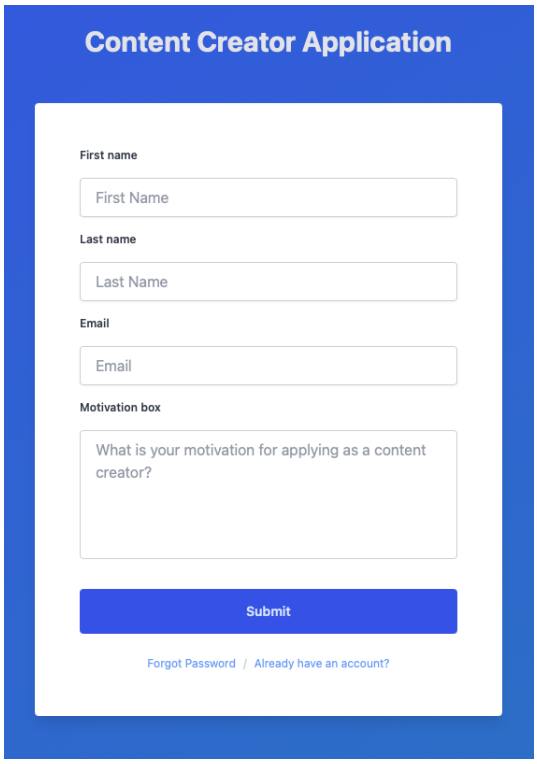


Figure 4.3: Sign-up form for individual creators

Receiving and reviewing CC applications

This was implemented by storing content creator applications in the database. We designed an admin page on the front-end to display all pending applications. An Educado admin can then see the details of an application and then either approve or reject it. If approved, the back-end will send an email to the applicant containing a congratulatory message and a temporary password (hashed in the back-end) with which they can log into the web app and update their password. If rejected, the back-end sends an email with a reason for the rejection. To send the email we use a middleware mail system provider called SendGrid[13], to whom we just provide the email of the applicant and the message we want to give them. Receiving and reviewing applications can be viewed at figure 4.4 and figure 4.5

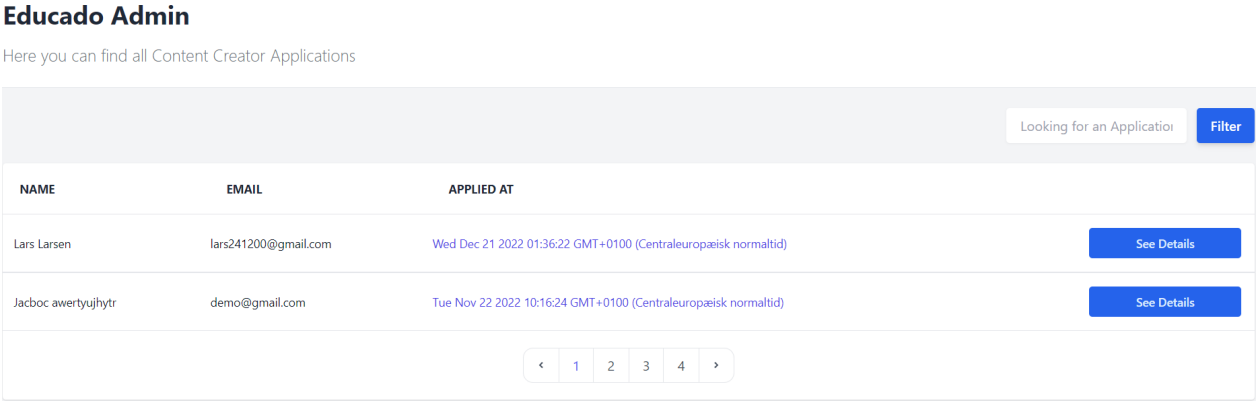


Figure 4.4: Dashboard of content creator applicants

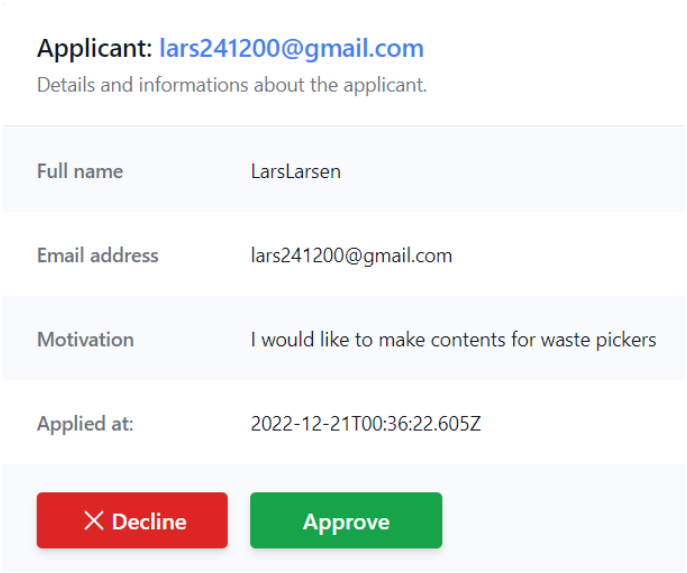


Figure 4.5: Content creator application detail

4.3.5 Sprint 2 Review

During the second sprint review, we had a facilitator named Christian, who is a certified scrum master with many years of experience. He provided valuable insight into how a sprint review can work effectively and be less time-consuming. At the start of the review, everyone presented their user stories and their increment like we had done in the previous review. The product owner approved user stories one and two, though each with a minor caveat. The product owner pointed out that we needed to have a meaningful button design for approving and rejecting a content creator application. This was added as a minor user story to the backlog of the next sprint. In addition, when an applicant returns to the login page after submitting a content creator application, the user will not for example receive an *application sent* message, which could possibly confuse them, even though the application was otherwise handled correctly by the system. In the end the two user stories which were approved needed some minor adjustments in order to function more smoothly, but the PO would rather we focus on other tasks.
Another point from the product owner was to focus on some of the user stories related to creating courses and pushing them to the mobile app will, which would involve both group 1 and group 3. In order to avoid overlaps, the product owner suggested both groups meet up to organize good code structure and naming conventions for variables.

4.3.6 Sprint 2 Retrospective

The second sprint retrospective gave us an insight into how we should start merging all the relevant code that has been made across the groups. The facilitator Christian gave us a lot of ideas on how we should work together as groups, and what a scrum master should do for each individual group, in order to promote teamwork. One main point was that the whole team should agree upon a common sprint goal during sprint planning to help each group decide which user stories to work on, such that the team as a whole can make progress towards producing increments that play well together and focus on bringing the product forward. This is in contrast with the way we made user stories in the first two sprints, where we devised them solely based on our group's understanding of what might be needed to move the project forward and adjusting those ideas through discussions with the PO. Christian also had a suggestion for making sprint reviews execute faster. The idea was to deliver increments to the PO during the sprint, as soon they met the definition of done, instead of

saving them for the sprint review. In principle this is a great idea, as the user stories would be accepted prior to the sprint review, shortening the process. It would however be difficult in practice, at least for our group, as we worked with all the user stories in parallel, finishing them up close to the review.

Collaboration between the groups

Unfortunately, we lacked proper collaboration between the groups as we had yet to realize the importance of such. As we saw it as another individual project instead of what it really was: A larger collaboratory project.

What went well

Apart from having two user stories finished and approved, it is hard to point out an area in which we did really well. The product owner was still happy that some implementations have met the deadline and most refactoring of the back-end is steadily moving towards the end goal. Although there is not much of an accomplishment when only half of the user stories were approved, but this does not mean things went badly for us, as work on the remaining user stories proceeded, but had taken longer than anticipated.

What did not go well

Early during the sprint, the issue of coordination between groups was discussed with our supervisor. We agreed that it was important that the responsibilities of each group were agreed upon and clear to all groups. We drew a diagram of the project highlighting each group's area of responsibility, as shown in Figure 3.1 in Section 3. However, we did not decide how this would be achieved in practice, and the diagram was not discussed with the other groups in a timely fashion.

Towards the end of the second sprint, it became apparent that our coordination with another group working on the back-end was lacking. Our groups were working on related functionality but in different areas of the project without communicating. Specifically, both groups were working on different branches of the GitLab repository. This meant that we had no idea whether people were working against each other, possibly producing conflicting or redundant code in the back-end.

What should we improve towards the next sprint

To address these issues of lacking communication, we met with representatives from the other groups and decided to set up a meeting with the other groups to align our work. This would involve agreeing on the structure of the GitLab repository of the project, and agreeing on which part of the project each group should be responsible for. We decided to have these alignment meetings on a fortnightly basis following each sprint planning meeting. The aim was to make sure each group's user stories would not be in conflict and to discuss any dependencies between them.

4.4 Sprint 3

Taking into account what we learned from the previous review and retrospective, we decided on an overall sprint goal, which was to merge the codebase among the groups.

4.4.1 Sprint 3 Planning

The previous sprint review covered four user stories where half of which were unfinished and moved back to the sprint backlog. For the current sprint, we plan on focusing on the two unfinished user stories and some

additional user stories. The plan will also entail using the scrum framework often enough which would benefit the group and having a merged back-end code between the groups.

Scrum framework and merging code

In our group, we have been using the scrum framework whenever the product owner was in attendance, but when he was not present the motivation for implementing code was more the priority than speaking with the other groups and, for example, doing daily scrum or having meetups with other groups. A scrum master is essential for having an overview of the developing group. As such if all in the group have tried being a scrum master, it will also benefit each person to know the working process where the role will change each time a sprint has ended.

The last focus point for sprint three is to merge code with the other back-end team, as working in the same branch should result in less merge conflicts and a more unified product. By agreeing to merge the two back-end codebases we will have fewer branches to deal with.

User story 1

As an Educado admin I want to be able to add a reason when rejecting content creator applications, such that the content creators have an explanation of why they were rejected.

Applicants need to know the reason why they were rejected, so they can improve their application before trying again.

User story 2

As a content creator I want the ability to create new courses, so I can expand my portfolio of courses on the platform.

This user story was described in Sprint 2, Section 4.3.1

User story 3

As a Content creator I want to be able to create and edit sections for a course I have created.

This user story was described in Sprint 2, Section 4.3.1

User story 4

As an Educado admin I want to have meaningful buttons, so I know which button to choose when approving or rejecting a content creator application.

The two buttons available to an Educado admin reviewing an application represent opposing actions and should therefore have visibly different colours. This should make the distinction between the buttons more clear, minimising the amount of errors in the approval process.

4.4.2 Daily Scrum

Daily scrum was not our strong suit in the previous sprint, and this resulted in group members working on the same thing or working on features that were not as necessary. In this sprint, we have made an effort to conduct the daily scrum meetings, as a tool for increasing productivity and coordination between the group members.

4.4.3 Cross-team coordination

During this particular sprint, meetings across the three groups were not used sufficiently. This was due to the fact that all groups thought that we should do the increments first, while dependencies between the groups were of secondary importance. This caused complications during the sprint, as there was no communication, but a great deal of implementation, which in turn caused the project to go in three different directions at once.

4.4.4 Increments

We did not manage to produce and deliver a working increment this sprint. We were able to create courses, but it was not polished and contained some bugs, which ultimately meant it was not yet a valuable improvement. While development on the other user stories was a work in progress, they were not well enough integrated, which meant we could not showcase any complete features.

4.4.5 Sprint 3 Review

During the sprint review, we showed incomplete features and explained what we needed to make them functional. The increments we showed were not able to satisfy the product owner's requirements, which resulted in the user stories getting transferred to the next sprint. The reason was that the front-end tasks were not finished or working optimally when making a course since there were errors along the way. We spent time discussing and explaining the difficulties of accessing AWS S3 bucket where we store pictures and videos related to the courses.

For the back-end we worked on merging branches between group 1 and our group. There were still missing links with the communication between the mobile app and the web application back-end, as group 1 could not yet fetch one demo course that had been created. There was quite a bit of dummy data that needed to be removed as soon as possible so real courses could be fetched in the mobile app. We did not find it useful to present the results of the code changes, since the purpose of the review is mostly to show of the product without delving too deeply into the technical details.

Speaking to Jacob and Daniel, we agreed that the overall sprint goal for the next sprint was to have a functional web app such that a content creator can upload a complete course with sections, exercises and answers. The whole process with a complete course needs therefore to be working and tied together. We also had to solve some authentication issues and provide APIs for group 1 and 3 to retrieve courses from the database. The aim was to have the entire flow from creating a course to displaying it on the mobile app working. The stakeholders also believe that functionalities should always take precedence over design, since without matching components to specific places in the front end design, the design will be empty, without purpose.

4.4.6 Sprint 3 Retrospective

We talked about a better way to structure our group with regard to work tasks. This meant that we decided to have two subgroups of 3 members each with one member as a leader, one for our front-end work and one for the back-end. Each leader would work on a task independently while the other 2 members of the subgroup would work together and assign their tasks for code review by the leader. Our scrum master would move between each subgroup to help with tasks as needed and help resolve dependencies between each subgroup.

Collaboration between the groups

Since our stakeholders emphasised a need for a functional product, we had to increase attention to communicating with the other groups. On the back-end side, we had to make sure the APIs we made available to group 1 provided the information they needed since they were responsible for providing that information to group 3, whose job it was to display the information in the mobile app. On the front-end side of the web app, we needed to make sure the content being created matched the needs of group 3 with respect to how they wanted to display courses in the mobile app.

What went well

After having a meeting with the other group that also had an influence on the back-end, we decided that it would be a good idea to have more control over how we merged changes into the shared development branch. We agreed to have changes merged through merge requests needing a separate person to review the code before it got merged. This forced us to review each other's changes which meant that we had to read the works of others increasing our understanding and communication with the other group.
We had a good talk with group 2, who had been working on a design for the data model for courses. This lead to a couple of modifications to the model and ensured that our future work on courses and their contents would be more closely aligned between the web-app and the mobile app.

What did not go well

We worked on too many user stories with each member working on their own task. We experienced difficulties coordinating coding tasks between multiple people because the tasks were closely related and interdependent. This meant we spent too much time idling while other tasks were being completed. There were some difficulties getting used to daily scrum even though it had already been introduced in the second sprint. Many of our daily scrums were either cancelled due to some people not showing up at the right time or some went more than 15 minutes which is not ideal for daily scrum.

What should we improve towards the next sprint

We should focus more on in-group communication about tasks, and explain what we need from other group members in order to complete our tasks. This could mean devoting time to agree on a Definition of Done for our increment at the start of the sprint. For the next sprint, the focus should also be on ensuring that the daily scrum meetings are more efficient so that more time can be allocated to implementing code and designing the web application's architecture of the new features.

4.5 Sprint 4

In this sprint, our goal was to complete the work on the user stories we had been working in the last sprint. Keeping in mind that delivering valuable increments to the product is an essential part of agile software engi-

neering, thus we decided to focus on the features we knew were realistic to decisively complete. Furthermore, our weekly team coordination meetings revealed that the mobile-app group was desperately in need of real courses delivered from the back-end, which is why this became the overall sprint goal.

4.5.1 Sprint 4 Planning

With the overall goal of delivering meaningful content to the mobile app in mind, we included several user stories which were unfinished from earlier sprints. In collaboration with the PO and in coordination with the mobile app team, we picked the following user stories, which we felt would improve the product the most at the current stage of development. Another focus point for this sprint was the continuous attention to dependencies between the three groups.

User story 1

As a content creator I want to add an exercise with a description text within a section
As our end product is an educational platform, the system will have to support the content creators' need to ensure that the learning goals of their educational material are met by the content consumers. In order to achieve this we will need to implement functionality that couples an exercise to a section within the web application.

User story 2

As a content creator I want to be able to add additional sections to an existing course, so that related educational material is grouped in a larger course
This user story is described in sprint 2, section 4.3.1. It is included in this sprint as well, as we did not fully complete the increment earlier, and it conforms to the overall goal of the sprint.

User story 3

As a content creator I want to be able to see and edit my own courses after I have signed in, such that the content I provide for the platform is only accessible to me
This is another remnant of an unsuccessful increment from a previous sprint, however still incredibly relevant. As part of the content delivery system, which includes courses, sections, and exercises, courses constitute the foundation of the educational platform. Restricting access to courses based on user credentials increases security, and allowing content creators to edit their courses provides business value.

User story 4

As an Educado admin I want to be able to add a reason when rejecting content creator applications
This user story is described in sprint 2, section 4.3.1

4.5.2 Daily Scrum

During this sprint our scrum leader made an effort to carry out the daily scrum meetings, to keep the development team involved in each other's work. However, it proved difficult for our group to keep the meetings short and on point. One reason behind the difficulties of these meetings is based on the fact that the group was rarely gathered in its entirety. This meant that there was no natural point in time, where we could stand up and reflect on our process.

4.5.3 Increments

This sprint yielded two increments to the content creator platform, with the acceptance of user stories three and four.

Content creators can see and edit their own private courses:

This user story was planned in the previous sprint, but the implementation of features needed proved to be more time-consuming than expected. Therefore, together with the PO, we planned to extend this user story into this sprint. We successfully implemented the necessary features both in the front-end and back-end so that a user is able to fulfil the user story. With this increment in place, our content creators are able to view courses, update the title, description, and cover image of courses, as well as create new sections within the courses.

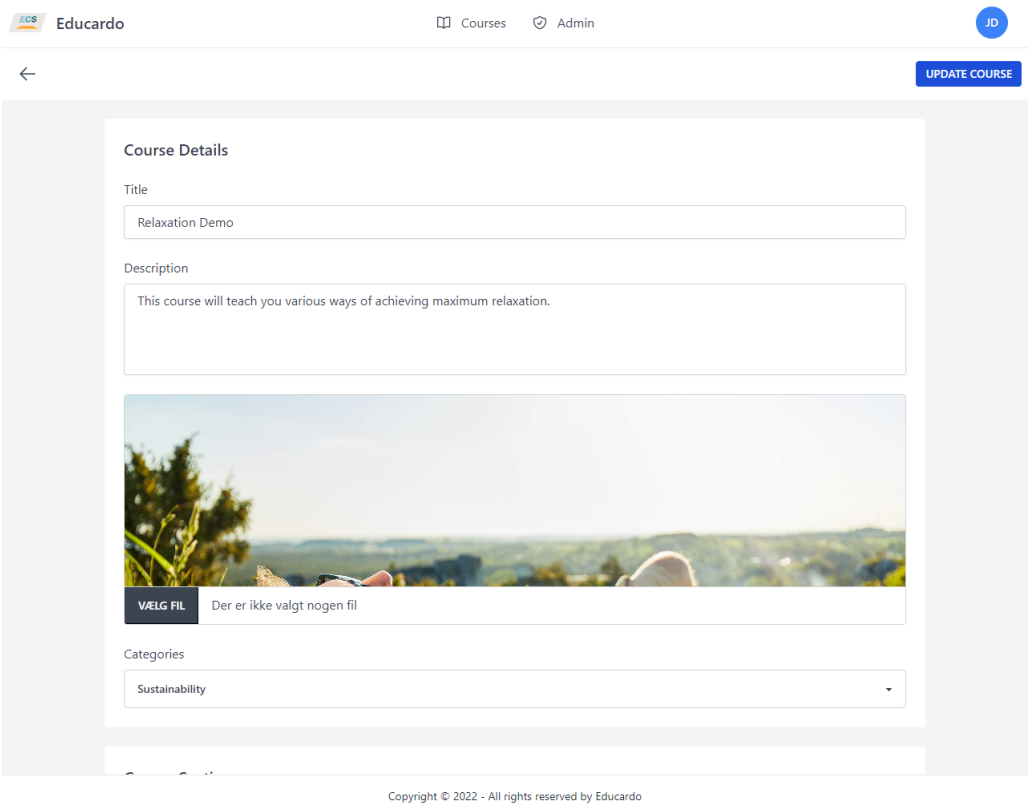


Figure 4.6: Editing a course

Educado Admins able to provide reasons for application rejections:

This user story was related to the onboarding process and initial control of content creator registration. The new features for this user story are an improvement of the content application feature provided to the project in previous sprints. The PO stated that it would be practical for the Educado admins to be able to elaborate rejection reasons to aspiring content creators. This increment required an extra form field in the application form, updates to the content in the post request, and alterations to the API route in the back-end.

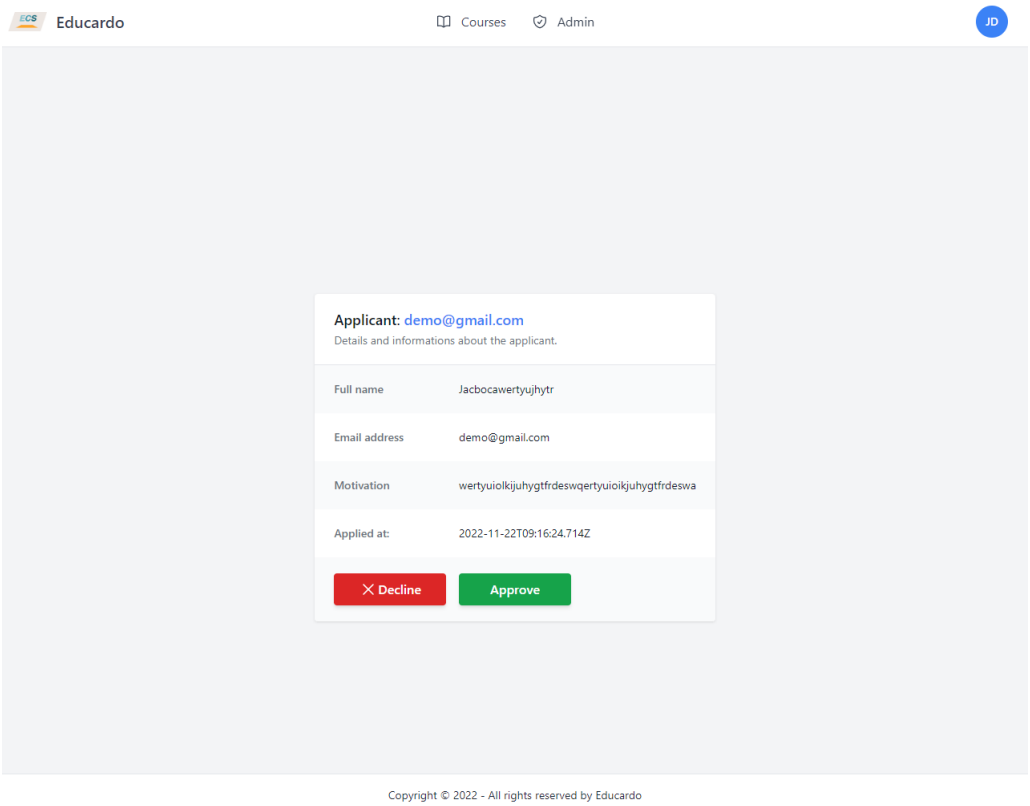


Figure 4.7: Approve or decline a content creator application

4.5.4 Sprint 4 Review

During the review, one of our stakeholders Daniel was present instead of our PO. The to increments we delivered during this sprint review left the group with two unaccepted user stories, which were extended into the next sprint, as they were close to being accepted by Daniel. After showcasing our increment we discussed the feature of marking a course as published / unpublished. Daniel mentioned the implementation of a review functionality, when publishing a course. After briefly discussing the pro and cons of this approach with the PO. We ultimately ended up agreeing that, the platform would be better suited with a "trust first, punish later"-approach. This means that content creators would be able to create courses without the manual review, but their courses could be flagged for review and lead to potential exclusion from the platform.

4.5.5 Sprint 4 retrospective

Having Daniel present during the sprint review proved to be very beneficial to the project. He was able to provide technical insight into some of the more embedded aspects of the project, such as the S3 bucket on AWS. The new group structure we devised in sprint retrospective 3 worked really well. Dividing the group into two smaller teams increased productivity for both the back-end team and front-end team. However, we were not able to conduct the desired code reviews.

Collaboration between the groups

With the ever-increasing need for integrating all the parts of the product, we in the last part of this sprint finally had a cross-team meeting. The meeting was mainly about how to improve the communication between all of us and we agreed that it would be a good idea to have a weekly meeting between the different scrum masters

within the different teams going forwards. Having held this first meeting it finally felt that we were on a path to better communication.

What went well

Since the beginning of this project, the group has improved its ability to pick the optimal workload. The optimal workload is a combination of an appropriate amount of user stories and the extent of the chosen stories. In this fourth sprint, we came closer to achieving this. Having fewer user stories meant that we could give each of them more attention throughout the sprint. This also meant that the team was less stressed, due to the burden of unfinished work being lighter.

What did not go well

Overall the group is disappointed we were not able to conduct the code reviews we had planned to do during our sprint 3 retrospectives. Another point, where we did not manage to meet our own expectations was with branching and merging. Prior to the start of this sprint, we had planned to merge into a dev branch with pull requests, to avoid merge conflicts and code that had not been reviewed in the main branch. As all three groups are engaged in their own work and providing meaningful increments for the system, it was difficult to find the time to review the code of others, before it was pushed to development.

What should we improve towards the next sprint

For the next sprint, the team hopes to become even better at communicating internally. While this sprint offered more opportunities for the group members to communicate, it should be supported by improved daily scrum meetings. Furthermore, the cross-team coordination and dependencies between groups are becoming more discernible, and as such, we would like to improve the weekly scrum master meetings.

4.5.6 Collaborative retrospective

Written in collaboration with all groups

During the sprint retrospective, at least one member from each group formed a team and discussed what they felt did not go well in this sprint and how we could improve it. A summarized list of some of these discussions has been collected below:

- **Sprint review:** One of the things discussed was the flow of the sprint review. It has happened that during one group's review they have been interrupted with questions or comments. The groups should be allowed to finish their review before questions are asked, and ideally, further discussions about future approaches should be done after the review.
  There has also been a noticeable lack of engagement from other teams during the presentations. Everybody should follow along during the review.
- **Communication:** There is still insufficient communication between the groups. There has been a slight increase in meetings since it was discussed in earlier sprints, but it has mostly either happened spontaneously, or only two of the three groups have been partaking in it.

  As per request, the number of meetings between groups has increased slightly. However, the meetings have been quite informal as they are usually not scheduled ahead of time and occur in a more spontaneous manner. There has only been held one official meeting in which all three groups partook.
  In order to get a better communication flow, a fixed meeting between all Scrum Masters of each group has been set. For the rest of the project, the groups will have at least one meeting every Friday at 10:00.

- **Backlog:** A topic that has been discussed several times during this project is having a shared backlog between all groups so that every group always know what the other groups are working on. The PO initially promised to create this with the Trello boards that were shared with him, but we have not yet received them. A person from each group should get together to set up a backlog. Ideally, the Scrum Masters will do it at their weekly meeting.
- **Alignment:** Each group has been focusing on their respective user stories and goals. As a result of this, the common end goal has not received a lot of attention. This also means that everyone has made their own design choices for both the code structure, as well as the front-end design.
  This has led to extra work, as the design patterns should ideally be consistent in a repository so future developers can easily understand the flow of the code.

  There also needs to be better alignment between front-end and back-end developers. If changes need to be made to one of the routes in the back-end repository, a front-end developer has to be informed about this, since it can affect their code as well. Another contingency that will be used from now on is to have shared documentation for all REST APIs, so front-end developers can easily check existing APIs and how they are to be utilized.
- **Definition of done:** One way to fix our alignment issues would be using a common definition of done (DoD). There has not been any agreed-upon set of rules for this yet. One DoD that will be added is that each repository should have a branch that will mock the master branch of the project. Only the working code should be pushed to this branch. In order to maintain this, each group should appoint a code review master. Their job will be to code review and accept the other group's code before it will be pushed to the master branch.
  When the code is reviewed it should also be checked to see if it follows the newly agreed upon design pattern, so our codebase is aligned.

End of collaboration

4.6 Sprint 5

In this sprint our overall sprint goal was to make everything work. This meant finishing the user stories we had extended from the previous sprints, and ensuring that all of the currently implemented features worked correctly.

4.6.1 Sprint 5 Planning

As this is the second to last sprint, the main objective and overall sprint goal for this sprint is to make a functional and unified product. This sprint should ideally end with a sprint review where we follow a course through from beginning to the end. This meant the creation of a course in the web app, displaying it in the mobile app, as well as being able to download it in the mobile app. In the previous sprints, it was not possible for us to fully implement the functionality for editing sections and adding exercises to them, which is why that is our top priority this sprint. Furthermore, we need to be particularly conscious of the dependencies between the three groups, as the interconnectivity of each groups' work is going to be important for the final product.

User story 1

As a content creator I want to add an exercise with a description text within a section

Extended into this sprint from sprint 4, Section 4.5.1

User story 2

As a content creator I want to be able to add additional sections to an existing course, so that related educational material is grouped in a larger course

Extended into this sprint from sprint 4, Section 4.5.1

User story 3

As a content creator I want to add, remove and edit questions to exercises to ensure learning goals for a section is achieved

This functionality is crucial to the product. The exercises contained within a section is the content creators' only chance to ensure that the waste pickers actually attain the desired knowledge. By providing correct and incorrect questions, the waste pickers can receive feedback based on their choice.

User story 4

As a content creator I want to upload an explanation video on wrong questions so that users get an explanation for why their answer is wrong

Providing the feature for content creators to upload an explanatory video to an exercise further supports their aspiration to ensure learning goals.

User story 5

As a content creator I want to choose which questions are correct/incorrect in an exercise, so I can provide valuable feedback to the content consumer

The ability for a content creator to set correct/incorrect answers is essential for the exercise to work in the desired fashion. This should allow the explanatory video associated with the exercise to play, should the content consumer choose an incorrect answer.

User story 6

As a content creator I want to upload videos to exercises, so that content I produce has both a visual and auditory consumption option

Since the mobile-app users have a high degree of illiteracy, the primary type of content in courses should be in either video or audio format.

4.6.2 Daily Scrum

In this sprint we tried to have daily scrum meetings in the group. Some of the meetings had the preferred duration of approximately 15 minutes, some meetings became much longer and therefore can no longer be considered a correct daily scrum.

4.6.3 Cross-team coordination

To assist the groups delivering value to the product through the increments, we had our cross-team meetings once a week to get a sense of where each group was towards the sprint goal. These meetings were intended to discuss and settle on what we should focus our efforts on to achieve the shared goal. We began the process of defining a shared project structure in a document that would include all the necessary sections and common materials for the group reports.

4.6.4 Increments

We completed work on user story 1 and 2, and although they were approved, we still faced some rendering issues when creating new sections and exercises where they would not show up immediately. By refreshing the web page we were able to show that the correct information was being saved in the database. The completed user stories can be seen at figure .1 and figure .2 in the appendix.
On the other hand, we still lacked some additional features that are an extension of the approved user story. These were not fully completed, and therefore we had to take them into account in the next sprint.

4.6.5 Sprint 5 review

During our fifth sprint review, we attempted to showcase the full flow of content creation, but we experienced re-rendering issues along the way. To our frustration, we had to reload the page a few times until changes would appear. We explained the minor details of what was missing, and how we intended to fix them. However, the increment we were able to show as a whole was enough to satisfy the product owner.

4.6.6 Sprint 5 Retrospective

During this sprint, we were challenged by the fact that other coursework demanded our attention due to their exam structure. This left us with less time to allocate to the project than had been the case up until this point.

What went well

The weekly cross-team meetings proved to be very valuable. Since they were fixed to a specific day they provided structure for our collaboration and allowed for regular resolution of dependencies.

What did not go well

During our sprint, we encountered several challenges that affected our progress and efficiency. A slew of external factors meant that the total amount of time when all group members were available was less than a day per week. As a result, we had limited collaborative development time, which reduced the efficiency of our user story implementations. Additionally, we inherited multiple stories from the previous sprint, which contributed to a disorganized and stressful development period.

What should we improve towards the next sprint

The increased work pressure we experienced in this sprint showed us that we might benefit from giving more attention to task management to ensure that we are able to complete our tasks. This means adhering more closely to the Scrum framework and making sure we actually have the Daily Scrum in the way it is intended.
Additionally, we could have prioritized the allocation of time and resources to the sprint and planned our workload more effectively. We could also have improved our communication and collaboration within the team to ensure that we were all working towards the same goals.

4.7 Sprint 6

In this, our 6th and final sprint, the overall goal was to make all the parts of the product work together as a complete product. This meant no introduction of new features, but rather a focus on code cleanup, integrating each group's functionality and fixing bugs in already existing functionality.

4.7.1 Sprint 6 Planning

It was clear we had to focus on the remaining user stories, and the PO was content if we could make everything work smoothly.

User story 1

As a content creator I want to upload an explanation video on wrong questions so that users get an explanation for why their answer is wrong

Transferred from sprint 5, Section 4.6.1

User story 2

As a content creator I want to choose which questions are correct/incorrect in an exercise, so I can provide valuable feedback to the content consumer

Transferred from sprint 5, Section 4.6.1

User story 3

As a content creator I want to upload videos to exercises, so that content I produce has both a visual and auditory consumption option

Transferred from sprint 5, Section 4.6.1

User story 4

As a content creator I want to be able to see and edit my own courses after I have signed in, such that the content I provide for the platform is only accessible to me

This user story is described in Sprint 4, Section 4.5.1

4.7.2 Daily Scrum

Only two people worked on user stories 1, 2 and 3, closely communicating about the front-end and back-end work being done. The bug-fixing was likewise done in pair, but not deemed necessary to be discussed in detail. Other group members worked on designing the role based security, which was not to be implemented, and therefore Daily Scrum was not held often in this sprint.

4.7.3 Cross-team coordination

There was a lot of spontaneous communication between the groups and a willingness to listen to their requests. Since we were close to tying all of the parts all the groups had been working on together, we felt quite enthu-

siastic at the prospect of making a great presentation at our final sprint review. We felt a greater sense of team spirit towards the conclusion of the project.

4.7.4 Increment

We implemented user stories 1, 2, 3 and 4, and managed to fix several bugs pertaining to how course content was saved incorrectly if there was a mismatch between what the put request sent from the web-app contained and what the back-end expected to receive. The completed user stories can be seen at figure .3, figure .4, figure .5 and figure .6 in the appendix.

4.7.5 Sprint 6 Review

We presented the increments by creating a new course, section and exercise, to which we uploaded both content and feedback video to show how the web application can serve the needs of the mobile application in its current state of development. Some minor caching issues interrupted the presentation a bit, but the PO was still satisfied that the basis for content creation was working as agreed upon.

Group 1 took over and presented the mobile app. They showed how the app was able to display the courses and related content that had just been uploaded through the web-application, thus illustrating that the full flow of content creation to consumption was functional.

4.7.6 Sprint 6 Retrospective

Written in collaboration with all groups

We began the sprint retrospective with an internal group discussion about sprint 6. Afterwards, each group presented their thoughts on this sprint. This included discussions about what did not go well, but also about improvements from previous sprints.

- **Sprint review:** In the sprint review, it felt more like one product compared to previous reviews. We had a flow where not all groups presented. Instead, we began showcasing the web application for the content creators, and afterwards, we presented the shared work of the two mobile app groups together. Here we could also see the new course created during the web presentation.

- **Workflow:** It has been a very stressful sprint. Not only did we want to try and get a working MVP before usability testing in Brazil, but we also had to try and fix the issues that were presented from the Static Code Analysis. We did have technical debt from previous sprints, which ideally should have been continuously fixed, but at the beginning of the project, the focus was on implementing new features and learning to work together in order to ensure a good product.

- **Communication:** The communication in this sprint was a lot better compared to previous sprints, but there is still room for improvement.

Final Retrospective

After we had the retrospective for sprint 6 we completed the development process with a final retrospective that covered the whole project from beginning to end.

- **Progress:** One thing we discussed was how we as students had evolved since a similar project in the 3rd semester. Here we had to work as a single group that solved a real-life problem for a company.

  In the 3rd semester, we had to spend a lot of time trying to figure out how to use the skills we learned in courses such as system development. Whereas in this semester these skills were simpler to apply, which was a great indication of our own development.

- **Sprint 0:** It was discussed that one way to start with a stronger foundation for common goals and better communication would be with a sprint 0. This was not something we were aware could be done before the end of sprint 1. In sprint 0 we should not touch any of the code, the focus should be on communicating with the other groups and all agree on a common set of requirements based on the project we received and the wishes of the stakeholders and PO.

- **Communication:** One of the common topics throughout the development process was the lack of communication. This includes communication among the three groups, but also communication internally. Each group's sprint backlogs were not shared with the other two groups. This caused issues as we did not always know what the other groups were working on. Especially in the beginning when we had yet to establish a good communication flow.
  The lack of communication lead to the risk of groups working on similar tasks, but there were also examples of different coding approaches in the same repository. In order to make it easier for future developers, the naming conventions and approaches should have been discussed before we began the development.
  But, in the last two sprints, the groups started to work more as one unit with one common goal. People from different groups would start sitting together to fix common issues. Also, the communication between front-end and back-end became clearer. If there were something front-end or back-end needed from each other they would talk together about it to try and find a good solution.

- **Scaling:** In continuation of 'Communication'. We did not agree on the use of any good tools that could help ease the process of working together across teams. This is of course one of the learning goals for this project, and we did in the last 3 sprints have a weekly meeting with the Scrum Masters from each team. One thing that was discussed was that we had hoped there would have been more information earlier, about how we could have used a scaling framework to assist us.

- **PO:** When the project first started, it felt as if the PO was learning his role along with us. Especially in the first few sprints, we were still not clear about a lot of fundamental knowledge from agile, so we had hoped that the PO could be of assistance. At the end of the 3rd sprint, we had a guest lecturer joining us for the sprint review. The guest lecture came with a lot of useful feedback, that helped us, but also was of assistance to the PO. After this, the PO seemed more confident and aware of his role. This along with our expanded agile knowledge made sprint reviews and sprint planning easier to complete.

Another thing that was discussed in the retrospective was that we would have had great benefit from the PO writing the user stories or setting clear sprint goals for each sprint. We wrote our own user stories and for the first few sprints, there was no agreed-upon goal for what should be achieved. This is one of the reasons why we started to accumulate technical debt.
It was sometimes not easy to get proper approval for the user stories we had written for each sprint and at times it felt as if the PO was not aware of what was in the sprint backlog.

- **Sprint review:** Just as with the scaling framework, it was discussed that it would have been beneficial to have learned how an actual sprint review is normally handled. There was confusion about how to actually proceed with the sprint review, and it was not until the final sprint that it felt as if the review was about one product and not 3 different products.

- **Pipelines:** It was not until the last two sprints that we learned about pipelines and Continuous Integration. It is normally considered good practice to open a pull request instead of merging code directly into the main branch of the project. It was discussed that it would have been very beneficial for us if we were provided with an option and guidelines on how to set up a pipeline required us to open pull requests.

Overall there have been a lot of frustrations during the project for every group. There was a very steep learning curve as we not only had to take over an existing project but also had to work together across teams.
In the end, our way of working together still has room for improvement and the product may still have some issues, but through the frustrations and failures, we experienced throughout the project we also learned valuable skills that we could slowly start applying while working on the project.

4.8 Combined user story diagram across groups

In an effort to better understand what happened during each sprint in the different groups, the following diagram was made to give the reader a better understanding of what has happened in the project as a whole. Figure 4.8 gives an overview of the user stories that have been completed during the six sprints. The blue ’Team COWS’ have been contributing to the front-end app. The green ’Team Half Full Stack’ has been contributing to both the front app and back-end capabilities inside the app, while also handling the back-end specific for the app. The orange ’Team Sharp Deluxe’ main focus was content creation and they have worked on both back-end and front-end capabilities.

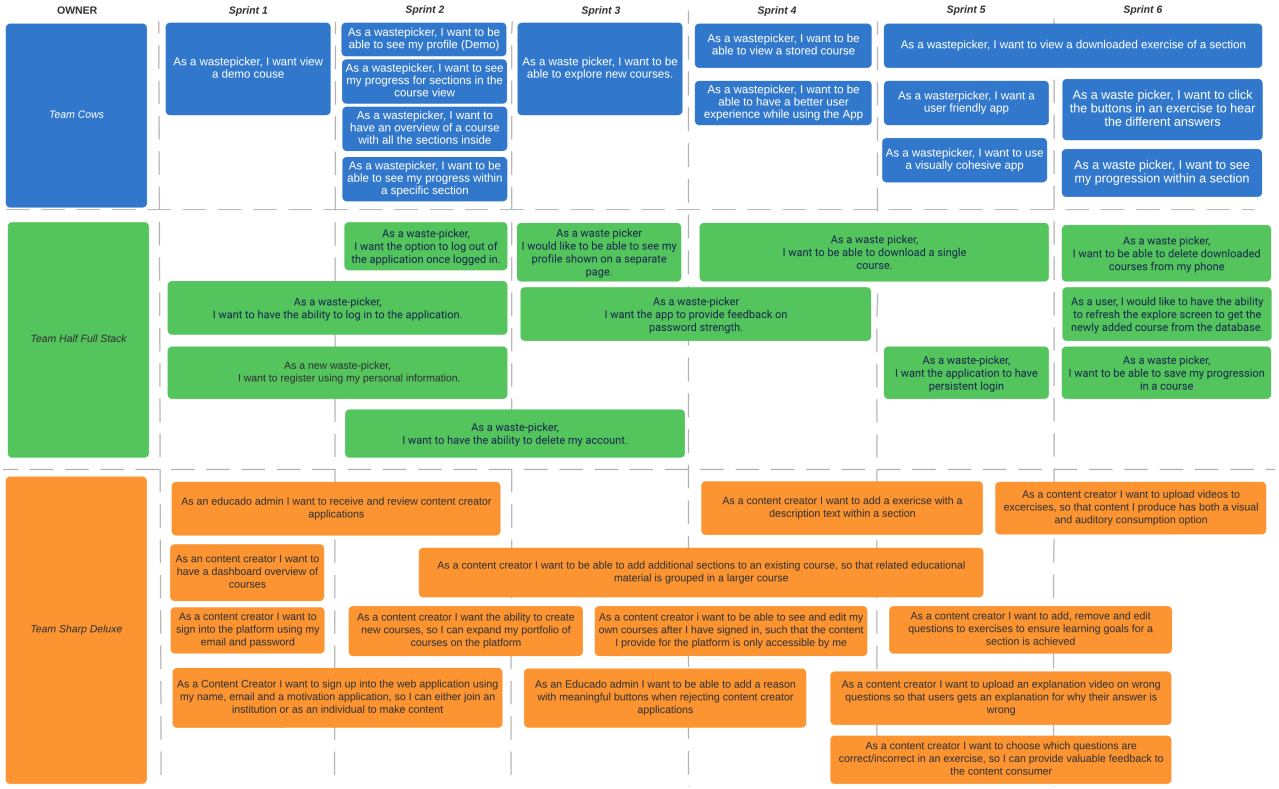


Figure 4.8: Combined User stories diagram

End of collaboration

5 Front-end Implementation

Since this team is working within the web application scope of the platform, we also want to discuss some of the most important changes, that have been made regarding the front-end of the web application. Even though this team primarily focused on the back-end implementation, there are certain changes to the front-end that we would like to discuss further.

5.1 Migration from JavaScript to TypeScript

One of the first major changes we made to the existing React application was to change the programming language from JavaScript to TypeScript. Due to the nature of React applications and the expectations for the growth of the entire platform, we decided that the extra type safety provided by TypeScript which would improve both the quality and scalability of the application. When developing React applications with JavaScript, it is often more difficult to validate whether you pass a parameter to a component and could potentially break the product. Most of these errors can only be caught at runtime when a user renders the page with a broken component and crash the application. Since TypeScript is a strongly typed language, it requires the developers to think about the types of variables and parameters during the development of the application. Besides aiding developers in writing higher-quality code, it also improves the self-documentation of the codebase and improves the cooperation between developers.

5.2 Unified Data Fetching Strategies

Data fetching strategies in React have always been a topic of discussion amongst developers in the community. Since React does not provide its data fetching library like Angular, it is unopinionated in the ways you load data into your application there exists a multitude of ways to solve this. But when more developers work on the same project, individual choices on how to solve this problem often lead to an inconsistent codebase. Therefore we chose to integrate the data-fetching library *useSWR*, created by the team behind the popular React framework NextJS. *useSWR* provides multiple features such as caching, error handling, loading indication, auto revalidation, etc. Based on the HTTP cache invalidation strategy, *stale-while-revalidating*, meaning that the hook returns the cached data (stale) while fetching new data (revalidating). For the users, this means less time waiting for data to load while providing a semi-real-time experience without using WebSockets.

5.3 Utility First Style Libraries

In the initial development of the content creator platform, the development team decided on using Google's Component design library Material UI (MUI) for the application styling. While MUI has been immensely popular over the last decade, there has been a paradigm change in the responsibility of the style libraries. Today, a vast majority of development teams have shifted towards utility-first frameworks, like the one we chose to use: tailwindCSS.

Control over design: Since tailwindCSS is a utility-first library, it does not vendor-lock developers to a specific design or opinionated way of doing things. Instead of providing predefined components such as cards, modals etc. utility-first libraries provide you with an API for a design system. This means that while still maintaining freedom of choice, it is easy for developers to create a consistent user interface design. This means that you do not have to fight against your design system if you want to make changes.
Smaller bundle sizes through build optimizations: Another benefit to using tailwindCSS over component design libraries is smaller production bundle sizes. When a new production build is created, the installed

tailwind package will automatically scan the codebase for every used tailwind class and only include the CSS needed in the production CSS files *(Tree Shaking)*, which decreases the page load impact. Compared to popular component libraries which often include all the CSS needed for every component.

5.4 Global State management

The final change we made to the existing web application in terms of front-end technologies was to move away from Redux. Redux was at one time the industry standard for managing global states, but the main issue with Redux is that it was designed for class-based react. A lot of the problems Redux helped developers with, have been alleviated by the introduction of *Hooks (functional components)* in React 16.8. The setup and amount of code required to work with Redux have been replaced with simpler solutions such as the Context API. Simple as Reacts Context API might be there exists a plethora of solutions that make it easier to manage global-state in our application, such as Recoil by Facebook, Jotai, and Zustand, just to mention a few. We chose to work with Zustand because of the ease of use and the simplistic approach. Zustand allows you to create a store, similar to how a class is created in OOP languages like Java. You tell it which values to hold and write some getters and setters for these values. This allows you to access and update the information in the store everywhere, without passing it down through the component tree.

6 Back-end architecture and implementation

The Educado platform has a very long expected lifetime, as its supposed objectives are to be deployed for many years and continually developed upon each year by new students who have never seen the project before. Considering this, it is therefore paramount that the codebase has a strict structure in order to ensure ease of learning for new developers. It should be maintainable and consequently be self-documenting. It should be very adaptable to allow for new changes, therefore it should be library agnostic such that a change in the use of a library is easily accomplished without changing the entire codebase.
This requires a structure that utilizes separation of concern, which in turn allows each file to have the minimal amount of concern possible, and therefore the minimum amount of code in each file. For this, we took inspiration from many of the concepts of clean architecture [6]:

6.1 Overview of the back-end components

Written in collaboration with all groups

To get an idea of the structure of the back-end, lets look at the components that make it up. The back-end is made up of a few components that each have their own set of responsibilities within the system. Each of these components is sub-divided into several layers. These layers help split up the concerns by focusing on a single aspect of the application. The following is a short description of the purpose of each layer.

- **Routes**: The routing layer is responsible for the outermost interaction with the web and therefore contains all the valid endpoints of the API.
- **Controllers**: The controllers are responsible for validating the data of incoming requests and passing that data to the corresponding use-cases or services.
- **Use-cases**: The use-cases have the responsibility of carrying out the actual steps in an operation. For example, a use-case might be `approveMotivation(motivationId)` in the case of successfully signing up as a new content creator. The use-case should find the existing motivation record with the given id, then

update the status to approved and finally send an email containing a one-time password to that applicant. The use-case would contain the for carrying out the steps.

- **Domain**: The domain layer is where the main entities in the system reside. For the case of content creation, examples of an entity might be the Course, Section, Category and so on. The entities are themselves responsible for doing the state changes that always ensure that they are in a valid state. For example, an applicant's motivation should not be able to go from an accepted state to being a rejected state as this would rarely be the case in the real world. The motivation should also always include the first name and last name of the person applying. These types of rules would be enforced inside the motivation entity itself.
- **Gateways**: The gateways have the responsibility of transforming and persisting data to the database while providing a simple interface to client code.

The below diagram shows how the components of the back-end each store these layers as well as the direction of dependencies between them.

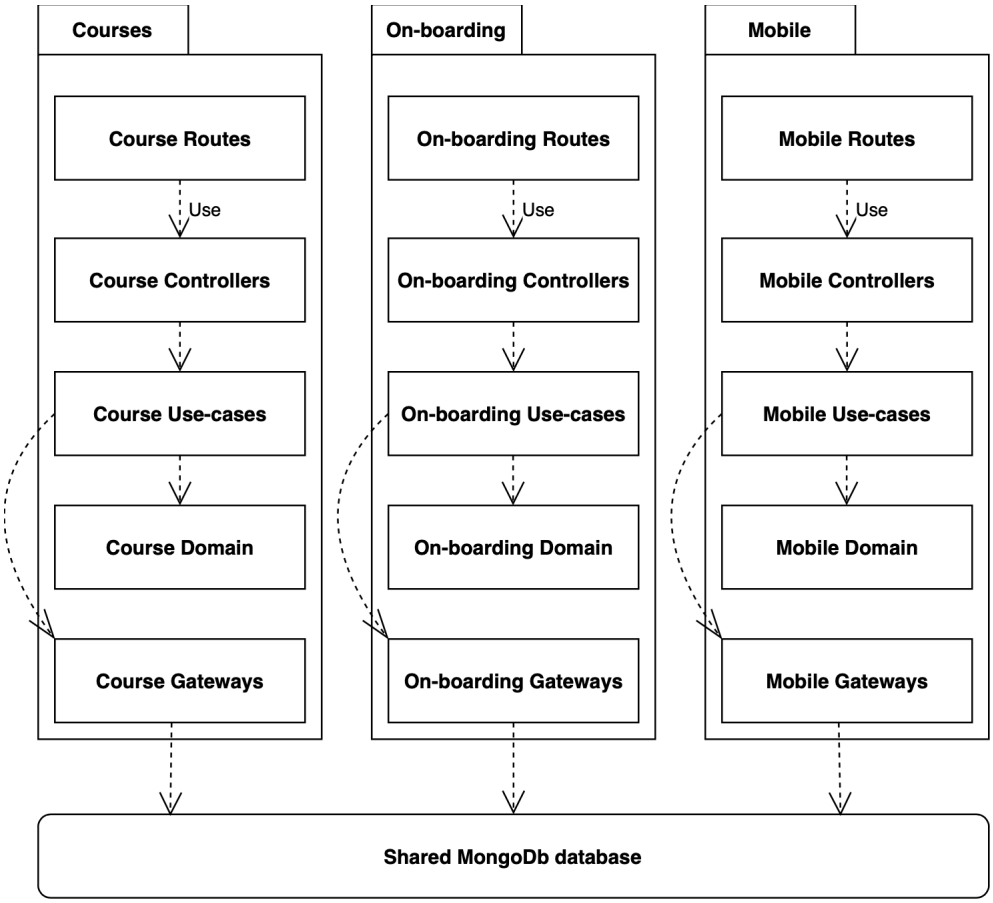


Figure 6.1: Component diagram of the back-end

End of collaboration

6.2 Following a clean architecture

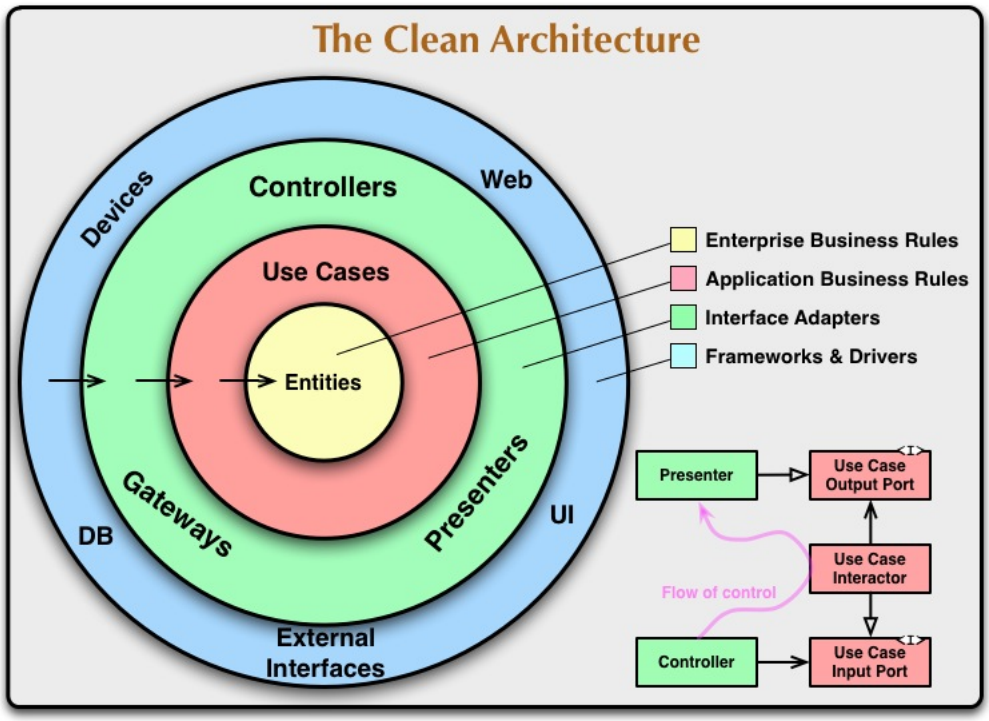


Figure 6.2: clean architecture[6]

Even though we are not strictly following clean architecture, we took its core principles and tried to apply them whenever it made sense. The further in the circle, the less likely a functionality is to change. Libraries and frameworks are quite susceptible to change so we want to isolate these as much as possible so they have minimal impact on some of the core logic. As an example, we do not want a change of email provider to affect how creator applications are approved or declined.
Generally, this type of clean architecture produces the following results:

- Framework agnostic: The architecture does not depend on frameworks, but more so uses them as tools that can be applied when necessary and easily changed if deemed necessary.
- Testable: The business logic can be tested without requiring the direct use of packages.
- Independent of external agencies: The business rules know nothing about the outside world. Each ring inside knows nothing about the rings on the outside, therefore each ring on the outside can not impact a ring further inward.

These features make clean architecture very resilient to unforeseen changes in lower-level code. To achieve the fact that logic in an inner ring is not directly dependent on logic in an outer ring, we need a mechanism to reverse the dependencies.

6.3 Dependency injection

An important aspect of the clean architecture we employ is that dependencies coming from layers further out are not imported in layers further in but instead injected using dependency injection.

The advantage of this, is that even if the package is later changed or the version updated, it is not necessary to update all functions using the package but only the original dependency import and its subsequent custom function utilizing it. This makes the inner layers package agnostic. As an example, it allows one to create custom select functions for SQL, such that a new SQL database may later be used, while only requiring refactoring of the custom functions. This also allows us to easily be able to change between a test database or live database by only changing the code in one place instead of multiple places.
An example of how it is used can be seen in the email sender:

```
1  "/helpers/email.js"
2
3  const sgMail = require('@sendgrid/mail')
4
5  module.exports = Object.freeze({
6    isValid,
7    send: sendMail
8  })
9
10 function isValid(email) {...}
11
12 async function sendMail({...}) {
13
14   ...
15
16   await sgMail.send({
17     from,
18     to,
19     subject,
20     text,
21     html
22   })
23 }
```

First the external library is imported and the library is encapsulated in a function called sendMail(). Here the logic for how to send mail is specified. Then the functions sendMail() and its utility function isValid() are exported as a final object.
This Email object is subsequently imported (line 1) in the injector commonly just the "index.js" file, and passed down to the subsequent factory function that makes use of it in this case the `makeAddCCApplication`.

```
1  "use-cases/index.js"
2
3  const Email = require('../../helpers/email')
4
5  const { contentCreatorApplicationList } = require('../gateways')
6
7  const makeAddCCApplication = require('./addCCApplication')
8
9  const addCCApplication = makeAddCCApplication({ contentCreatorApplicationList, Email })
10
11 module.exports = {
12     addCCApplication,
13 }
```

Here the function `makeAddCCApplication` gets 2 dependency injections, where one of them is the custom email object.

```
"addContentCreatorApplication.js"

module.exports = function makeAddCCApplication({ contentCreatorApplicationList, Email }) {
    return async function addCCApplication(applicationInfo) {
        ...

        sendConfirmationEmail(application)

        ...
    }

    function sendConfirmationEmail(applicant) {

        Email.send(...)
    }
}
```

Now the email function can be utilized in the specific use-case called addCCApplication, and even if we in the future wanted to change how emails were sent, it only needs to be changed in the email adapter. When maintaining code, libraries can get outdated, have security risks or become too expensive. Therefore it is important that libraries are easily changeable, such that a system is not dependent on hard to change outdated libraries.

The big disadvantage of this system is that it makes the initial development time a lot more cumbersome and time-consuming. On the flip side, the code becomes a lot more flexible and maintainable in the long run.

6.4 Clean folder structure

To help create a clean overview, each distinguishable feature/component of the back-end has its own folder, where all the rules and all the layers of our version of clean architecture apply.

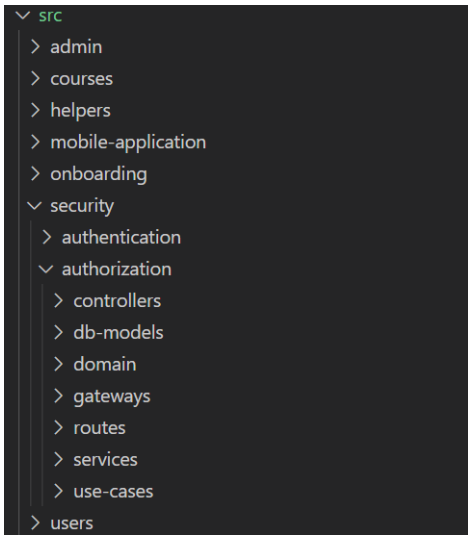


Figure 6.3: back-end folder structure, see figure 3.4 for initial folder structure

One of the initial problems with the back-end was the over-use of framework terminology as folder structure

that said very little about what the application was about. As can be seen now, the major folders or packages reflect concepts for the learning platform instead of being hidden away in code.
We use the following encapsulations to subdivide the different responsibilities in each package in a consistent manner:

- **domain**: The domain contains the entities. Entities are objects that know everything about what it means to be an entity, including validating that the data is correct. Generally, this is where the enterprise-wide business logic is.
- **use-cases**: use-cases contain concrete use-cases of an entity. More generally, it contains application-specific business logic.
- **services**: Services are a bundle of use-cases, used when the flow of information makes more sense to have together instead of in separate use-case files.
- **gateways**: Functions act as the gateway between the rest of the application and the database.
- **controllers**: Handles the high-level logic and call flow for the various routes.
- **db-models**: DB-models contain the Database schemas that specify how data should exist in the database.
- **routes**: Exposes the API-endpoints that can be called.
- **utils**: Utility functions that serves functionality for the package but not other for packages
- **helpers**: Utility functions that may have general functionalities across multiple packages

Each of these encapsulations has an index.js file where external dependencies are imported to allow for dependency injection into the concrete files in the folder.
Not all packages may use all of these forms of encapsulation, as not all packages may necessarily, as an example, have a database model, but these are the ones that we have used to create consistency in the codebase.

The interaction between these folders is as follows: If a package has a dependency deeper within, then it can be imported. In all other circumstances, it must be injected, including if it is on the same layer. The following is an illustration of the general architecture where this applies:

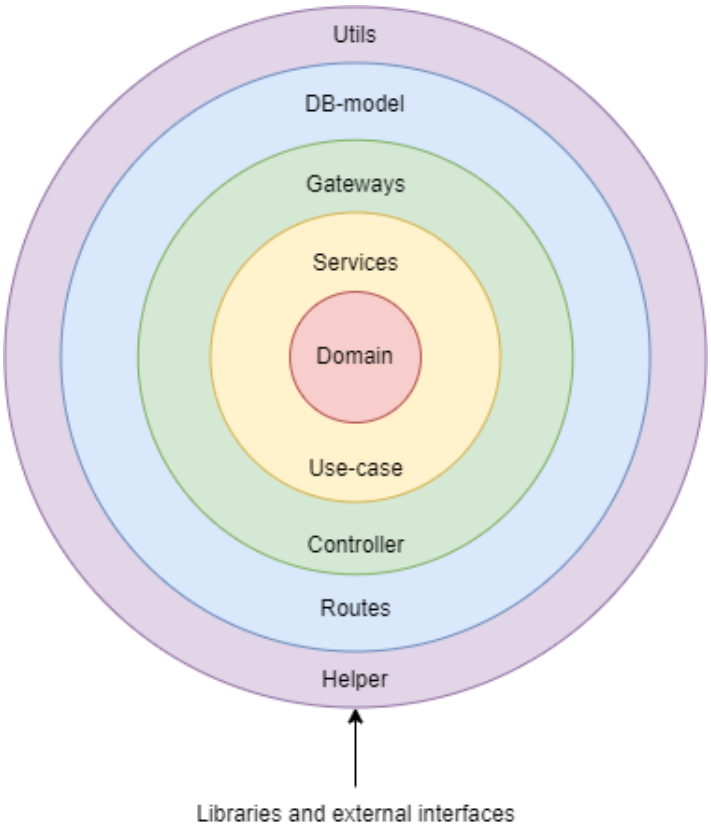


Figure 6.4: Our general architecture

Unfortunately, it is not always possible to adhere to the rules of dependency injection, as there are always circumstances where breaking the rule is beneficial. For example, sometimes due to programming language constraints, it may not be possible to inject dependencies without creating messy convoluted code. Therefore in rare circumstances, a direct import between the layers may be used even though it in principle should have been injected.

In general, our architecture is followed unless something with justifiable advantages breaks it. The benefits should be constantly weighted to maintain the system's structure.

6.5 Call flow

The current call flow between the different parts of our architecture can generally be described as follows:

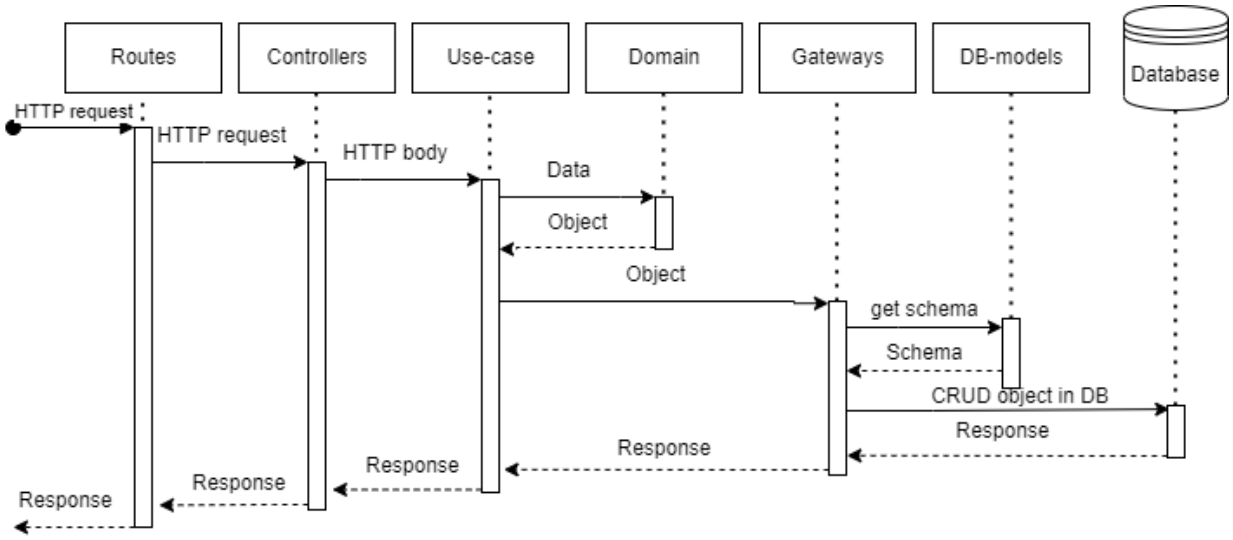


Figure 6.5: Typical call flow of a post request

Where **utils** and **helpers** may be called at any point during the process.
Generally the steps taking are as follows:

1. An HTTP request is created at a valid exposed endpoint.
2. If the endpoint exists, the HTTP request will be forwarded to the controller
3. The controller takes care of the request. Handling any input validation and error handling of the HTTP request. Depending on the input, a certain use case will be called with any necessary data from the HTTP request. If required, such as in instances of a put request, the gateway is typically called to retrieve an object from the database before the use-case can be called.
4. The use-case handles the flow of function calls necessary to achieve the specific use-case. Typically first requesting a domain object be made based upon the HTTP request and its body.
5. The domain knows what it means to be a certain object. Validating the input data and throwing an error if anything is not as it should be. It then returns a newly created object if everything is ok.
6. The object is then passed to the gateway to be performed together with a relevant CRUD command on the database.
7. The gateway takes the object and maps it to the database schema, then performing the CRUD command on the database.
8. The result of this action is returned all the way back to the controller, which then interprets the result and returns a relevant response to the routes.
9. The routes finally formats the response that goes back to the original caller.

As is evident, a lot of validation is done along the way, to slowly check that the request fills all the requirements necessary, thereby ensuring input validation and proper error handling.

6.6 Back-end implementation

In this chapter we will be going through in a little more detail the implementation on some of the parts that we have worked on during this project.

6.6.1 Content creator application

In this section we will be going through an example of how we implemented the feature for onboarding new content creators on the platform.

The use-case to be demonstrated is rejecting a content creator application.
The first thing to be created is some endpoints to handle the new functionality. This is done inside the routes folder.

```
const router = require("express").Router();

const { makeExpressCallback } = require('../../helpers/express')
const { contentCreatorApplicationController } = require('../controllers')

router.get("/applications", makeExpressCallback(contentCreatorApplicationController))
router.get("/applications/:id", makeExpressCallback(contentCreatorApplicationController))
router.post("/applications", makeExpressCallback(contentCreatorApplicationController))
router.put("/applications/:id", makeExpressCallback(contentCreatorApplicationController))

module.exports = router;
```

Here the custom express wrapper is used to adapt the request and response object from express into a slightly modified HTTP request object that can be understood by the controller.
Assuming that an application in the database has already been created, the act of changing the content creators application status from 'awaiting' to 'rejected' utilizes the put request. This put request is then handled in the controller:

```
module.exports = function makeContentCreatorApplicationController({
    contentCreatorApplicationList, Params, Id }) {

    return async function handle(httpRequest) {

        switch (httpRequest.method) {
            case 'GET':
                return await getContentCreatorApplication(httpRequest)
            case 'POST':
                return await postContentCreatorApplication(httpRequest)
            case 'PUT':
                return await putContentCreatorApplication(httpRequest)
            default:
                throw new HttpMethodNotAllowedError(httpRequest.method)
        }
    ...
```

Note, that the use-cases, which will later be used, are imported, as use-cases stem from a layer deeper than the controller while the gateway `contentCreatorApplicationList` is dependency injected through the function

parameter as it stems from the same layer as the controller.

The controller calls the relevant function by switching on the corresponding HTTP method in this case PUT.

```
1  async function putContentCreatorApplication(httpRequest) {
2      const id = httpRequest.params.id
3      const rejectionReason = httpRequest.body.reason
4
5      const allowedActionsSchema = {
6          type: 'object',
7          properties: { 'action': { enum: ['approve', 'reject'] } },
8          required: ['action']
9      }
10
11     const { action } = Params.validate({
12         schema: allowedActionsSchema,
13         data: httpRequest.queryParams,
14         throwOnFail: true
15     })
16
17     if (!Id.isValid(id)) throw new ValidationError("Invalid or missing id")
18
19     const existing = await contentCreatorApplicationList.findById(id)
20     if (!existing) throw new ValidationError(`No content creator application with id '${id
   }' was found`)
21
22     let updated
23     if (action === 'approve') {
24         updated = await approveCCApplication(existing)
25     }
26     else {
27         updated = await rejectCCApplication({
28             applicationInfo: existing,
29             reason: rejectionReason
30         })
31     }
32
33     return {
34         success: true,
35         status: 200,
36         data: updated
37     }
38 }
```

The parameters from the HTTP request are first extracted, as shown on line 2-3.
The important parameter 'action' is validated by checking that it exists and that it is either "approve" or "reject", otherwise an error is thrown as shown in lines 5-15.
Next, the other parameters are validated, and if the parameters are valid, a content creator application from the database is retrieved through the use of the gateway, as shown in lines 15-20. An error is thrown if the parameters are not valid.

At last the use-case 'rejectCCApplication' is called with the rejection reason and the existing application in the database. If successful, the updated application will be returned back to the controller.

```
1  const { makeContentCreatorApplication } = require('../domain')
2
3  module.exports = function makeRejectCCApplication({ contentCreatorApplicationList, Email }) {
4
5      return async function rejectCCApplication({ applicationInfo, reason }) {
6
7          const application = makeContentCreatorApplication(applicationInfo)
8
9          if (reason) {
10             application.reject({ reason })
11         }
12         else application.reject()
13
14         const updated = await contentCreatorApplicationList.update({
15             id: application.getId(),
16             approved: application.isApproved(),
17             rejectReason: application.getRejectReason(),
18             isRejected: application.isRejected(),
19             modifiedAt: application.getModifiedAt()
20         })
21
22         sendRejectMail(application)
23
24         return updated
```

Note, how only the domain is imported on line 1, while once again the gateway is injected, as only the domain is from a deeper layer than the concrete use-case.
First, the contentCreatorApplication object is created through a call to the domain (The domain object creation will be explored later) using the pre-existing contentCreatorApplication found in the database.
Next, the domain object's method `reject()` is called to change its status value to 'rejected'. A rejection reason will be taken as input depending on if a rejection reason was given. Otherwise, a default rejection reason will be given.
Next, the gateway, which functions as a wrapper to communicate with the database, is called. It takes an object of changes and uses the mongoDb driver to update the given application. The result from the update is the type of a mongoose document, which gets converted back to a traditional JavaScript object to not propagate the frameworks type higher up in the application.

```
1      async function update({ id: _id, ...changes }) {
2          const result = await dbModel.findOneAndUpdate({ _id }, { ...changes }, { new: true })
3
4          return result.toObject()
5      }
```

The result of which is ultimately returned back up the call stack.
At last a rejection email is sent to the user using the custom helper function `sendRejectMail()`, informing them of why they were rejected.

The domain object that handles the rejection reason has the following code:

```
1          if (!firstName) throw new ValidationError('A firstname must be provided in the
       application')
```

```
2         if (!lastName) throw new ValidationError('A lastname must be provided in the
      application')
3         if (!email) throw new ValidationError('An email must be provided in the application')
4
5         let rejectReason = ''
6
7         return Object.freeze({
8             getId: () => id,
9             getFirstName: () => firstName,
10            getLastName: () => lastName,
11            getEmail: () => email,
12            getMotivation: () => motivation,
13            isApproved: () => approved,
14            isRejected: () => rejectReason.length > 0,
15            getRejectReason: () => rejectReason,
16            getCreatedAt: () => createdAt,
17            getModifiedAt: () => modifiedAt,
18            fullname: () => `${firstName} ${lastName}`,
19            approve: () => approved = true,
20            reject: ({ reason = 'No reason given' } = {}) => {
21                approved = false,
22                rejectReason = reason
23            }
24        })
```

The domain object will check that the information necessary is available through multiple validation steps. If the information is either incorrect or not given, then an error will be thrown.
The domain object is then created alongside any methods necessary, such as `reject()` which holds a default rejection reason. Finally, the object is returned up the call stack, ultimately ending up in the controller where it will be used to send back a HTTP response.

6.6.2 Courses

The **Courses** folder was by far the largest contributor to individual use-cases, totalling 10 use-cases as shown below:

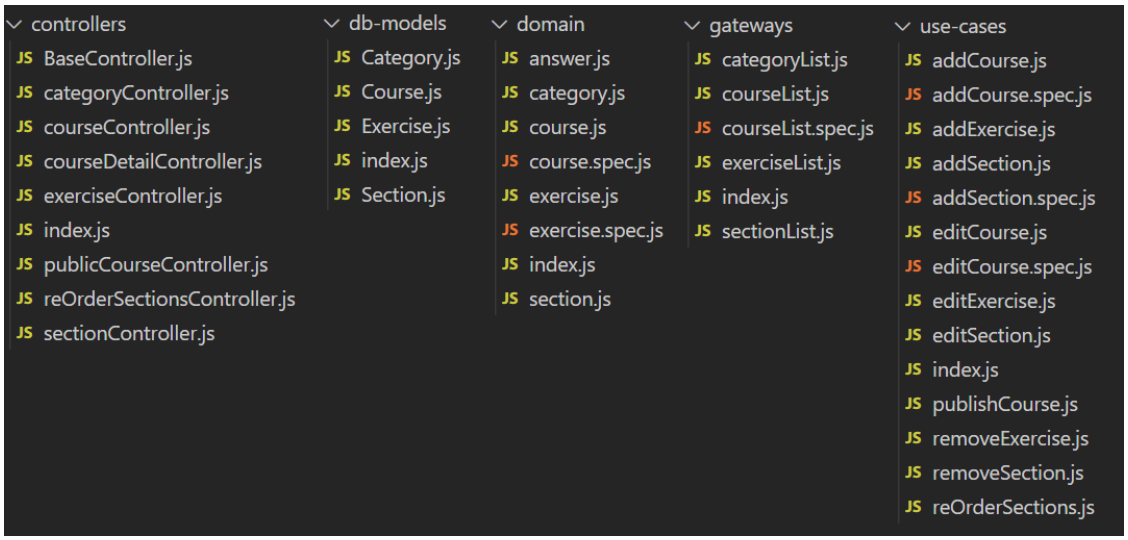


Figure 6.6: Files apart of the Courses folder

Note, spec.js files are testing files

This in turn resulted in the following API endpoints:
These endpoints were some of the endpoints primarily used by the other groups, as they dealt with the caching and visual aspects of the courses. This API was therefore of the highest priority when developing the application.

```
/* Courses */
router.get('/public/courses', makeExpressCallback(publicCourseController))
router.get('/public/courses/:id', makeExpressCallback(publicCourseController))
router.get('/courses', restricted, makeExpressCallback(courseController))
router.delete('/courses/:id', restricted, makeExpressCallback(courseController))
router.post('/courses', restricted, makeExpressCallback(courseController))
router.get('/courses/:id', restricted, makeExpressCallback(courseController))
router.put('/courses/:id', restricted, makeExpressCallback(courseController))
/* Categories */
router.get('/public/categories', makeExpressCallback(categoryController))
router.get('/categories', makeExpressCallback(categoryController))
/* Sections */
router.get('/sections/:sid', restricted, makeExpressCallback(sectionController))
router.put('/sections/:sid', restricted, makeExpressCallback(sectionController))
router.delete('/sections/:sid', restricted, makeExpressCallback(sectionController))
router.get('/courses/:cid/sections', restricted, makeExpressCallback(sectionController))
router.get('/courses/:cid/sections/:sid', restricted, makeExpressCallback(sectionController))
router.post('/courses/:cid/sections', restricted, makeExpressCallback(sectionController))
router.put('/sections/reorder', restricted, makeExpressCallback(reorderSectionsController))
/* Exercises */
router.get('/exercises/:eid', restricted, makeExpressCallback(exerciseController))
router.put('/exercises/:eid', restricted, makeExpressCallback(exerciseController))
router.delete('/exercises/:eid', restricted, makeExpressCallback(exerciseController))
router.post('/sections/:sid/exercises', restricted, makeExpressCallback(exerciseController))
```

```
25 router.get('/courses/:cid/sections/:sid/exercises', restricted, makeExpressCallback(
       exerciseController))
26 router.get('/courses/:cid/sections/:sid/exercises/:eid', restricted, makeExpressCallback(
       exerciseController))
```

The general structure and interaction of the code follow that of the content creator application, and therefore won't be discussed in further detail.

6.7 Data models

Written in collaboration with all groups

MongoDB is neither an object database nor a relational database [14]. This means the methods we have learned for Entity Relationship Diagrams do not apply here as the database is not relational. Below is a best-effort attempt to model the database using a UML class diagram.

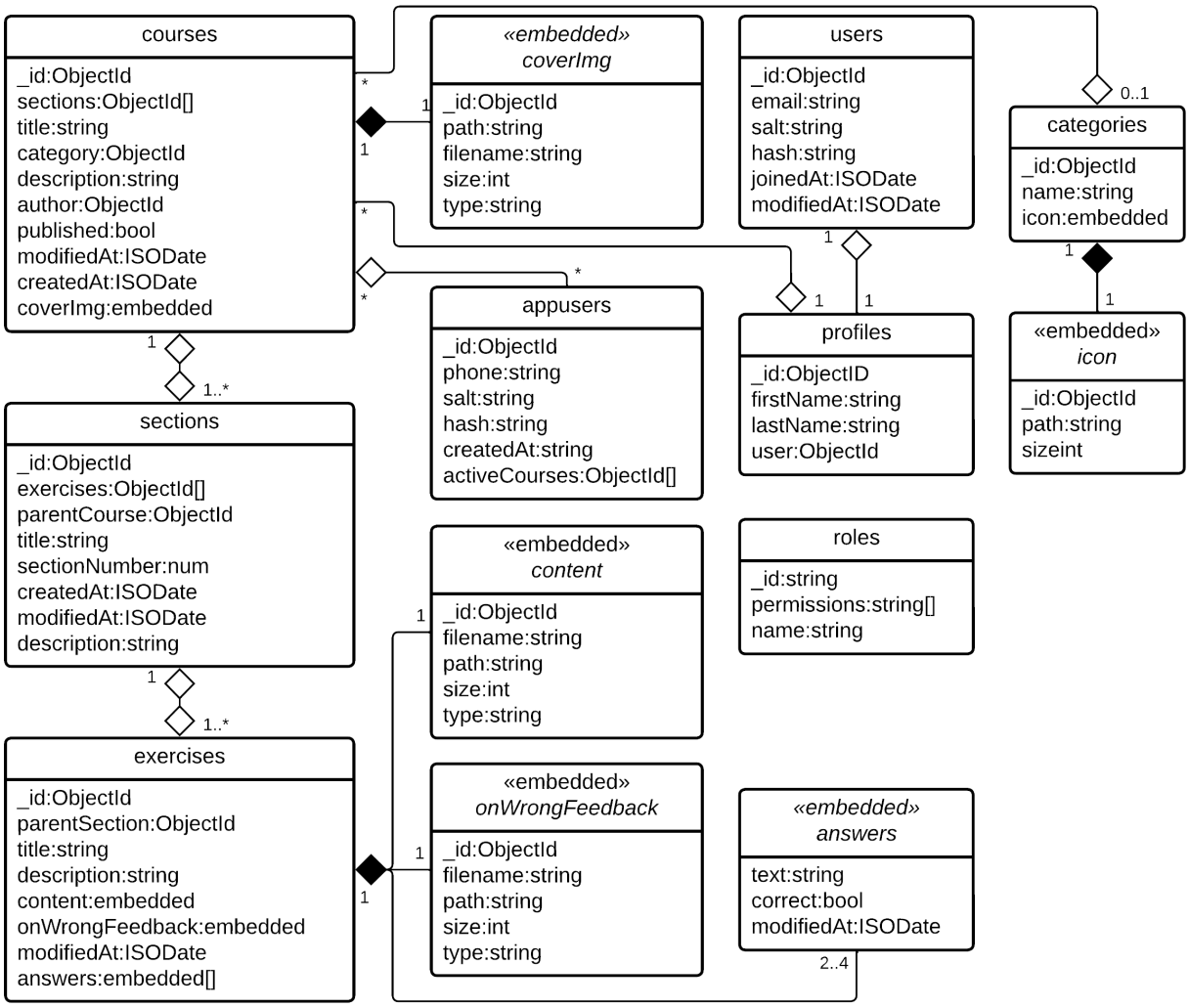


Figure 6.7: Class Diagram depicting relationships in MongoDB collections.

Note the non-standard usage of composition and aggregation. Composition (black diamond) denotes objects embedded inside other objects in a collection, and aggregation (white diamond) denotes referenced objects from another collection. Class fields are separated by a colon where the right-hand side denotes the type. Any field other than _id with type ObjectId references another object by id. ObjectId[] type denotes an array of referenced ids, similar rules apply to embedded and embedded[] with the exception that those objects are embedded in the parent object and not part of a collection.
Class names in figure 6.7 refer to the collection name in MongoDB, the collection contains objects, or documents to be exact, with a structure as depicted in the figure.

End of collaboration

At the initial state, there were 4 schemas, as shown in figure 3.6. The rest shown in figure 6.7 has been developed since then.
These are the current collections (schemas) in the database:

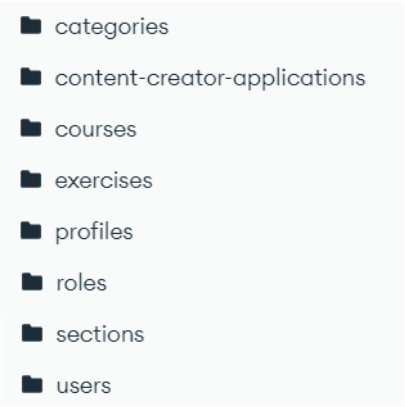


Figure 6.8: Database collections

The **content-creator-applications** are the motivations sent by content creators.
The **user** contains the elementary information required for a content creator user to login.
This is separated from the profile which contains all none essential information about a user such as preferences and what not.
The **appusers** serves the same purpose as profile and users combined, but is utilized by the app development group.

The **courses** have a specific category and are made up of **sections**, which in turn contain specific **exercises**. These exercises contain content, which may be a video, and 4 answers with at least 1 being right.

6.7.1 Authentication

As described earlier in the user story in sprint 1, the team behind the Educado project wanted their users to be able to sign in with just email and password, in addition to the OAuth2 authentication already set up. Since the existing OAuth2 authentication was implemented using the popular authentication library Passport, we found it logical to also implement the email/password authentication with their JWT strategy.

JSON web tokens with Passport:
As stated we chose to implement JWT authentication using the Passport Package[15]. Passport is a middleware

built for express js applications and provides developers with a modular framework to work with many different strategies. A strategy in terms of Passport is prebuilt methods of authenticating users E.g. OAuth, OpenID, JWT etc. Now that we found a strategy that matched the authentication needs, we needed to configure the strategy, with the following options for initialization:

```
// JWT Strategy options
const options = {
    secretOrKey: config.TOKEN_SECRET,
    algorithms: ['HS256'],
    ignoreExpiration: false,
    jwtFromRequest: ExtractJwt.fromExtractors([
        ExtractJwt.fromAuthHeaderAsBearerToken(),
    ])
}

const jwtStrategy = new JwtStrategy(options, (payload, done) => {
    userList.findById(payload.user)
        .then(user => done(null, user))
        .catch(err => done(err, false))
})
```

JWT strategy options detailed

- *secretOrKey:* String or PEM-encoded public key to verify token signatures, we chose to use a randomly generated hash/string (secret)
- *algorithms:* List of strings with the names of the allowed algorithms, we chose to use the one shown in the documentation examples: HS256
- *ignoreExpiration:* Option to ignore JWT expiration, we wanted to validate token expirations, so we set this to false.
- *jwtFromRequest:* Function that accepts a request as the only parameter and returns either the JWT as a string or null. The package has multiple extractor functions, and we chose to go with: *fromAuthHeaderAsBearerToken*, as it seemed the most straightforward.

JWT in action:
When using JSON web tokens, the developers need to be able to handle both authentication and re-authentication. This is due to the design principles of JSON web tokens, and the inherent security in short-lived tokens, which becomes useless after expiration. In the coming section, we will describe how we successfully handled both scenarios in our back-end application.[16]

authentication: When the user initially tries to authenticate, the user sends an email/password combination to the back-end. If these credentials are valid, the back-end will return with an access and refresh token pair. Once the user is authenticated, each subsequent request will include the access token, allowing the user to access routes, services, and resources.

```
    async function loginUser(httpRequest) {
        user = httpRequest.body
        const response = await authService.authenticate(user)
```

```
        return {
            success: true,
            status: 200,
            data: response
        }
    }
```

```
    async function authenticate(user) {
        const foundUser = await userList.findByEmail(user.email)
        if (!foundUser) { throw new AuthenticationError("Authentication: Access denied") }
        const isAuthenticated = Password.isValid({
            password: user.password,
            salt: foundUser.salt,
            hash: foundUser.hash
        })

        if (!isAuthenticated) { throw new AuthenticationError("Authentication: Access denied")
     }
        return JWT.generateTokenPair({ user: foundUser.id })
    }
```

re-authentication: Since JSON web tokens by design are intended to expire after some given time frame, a core concept of the authentication strategy is to issue new token pairs, once a user's access token is expired. For users to obtain new valid token pairs, we implemented an endpoint that receives the refresh token and returns a new token pair if the token is valid.

```
    async function refreshLogin(httpRequest) {
        const token = JWT.extractFromRequest(httpRequest)
        const { user } = JWT.verify(token)

        return {
            success: true,
            status: 200,
            data: JWT.generateTokenPair({ user })
        }
    }
```

```
    function generateTokenPair(payload = {}) {
        return {
            accessToken: signAccessToken(payload),
            refreshToken: signRefreshToken(payload)
        }
    }

    function signAccessToken(payload = {}) {
        return jwt.sign(payload, config.TOKEN_SECRET, { expiresIn: config.ACCESS_TOKEN_MAX_AGE
     })
    }

    function signRefreshToken(payload = {}) {
```

```
13          return jwt.sign(payload, config.TOKEN_SECRET, { expiresIn: config.
        REFRESH_TOKEN_MAX_AGE })
14      }
```

6.7.2 Role-based security

More and more user stories required special rights to be given depending on the profile and the courses associated. Instead of conditionally rendering pages on the front-end, we instead opted to try to design a role-based system where individual permissions could be given to profiles. This we did by designing the relations in the database to create a clear image of how different models were related. The advantage of having role-based security is that it allows for great flexibility when new actors or roles are introduced in the system.
The initial **Role-Based security** diagram of the system was designed as follows:

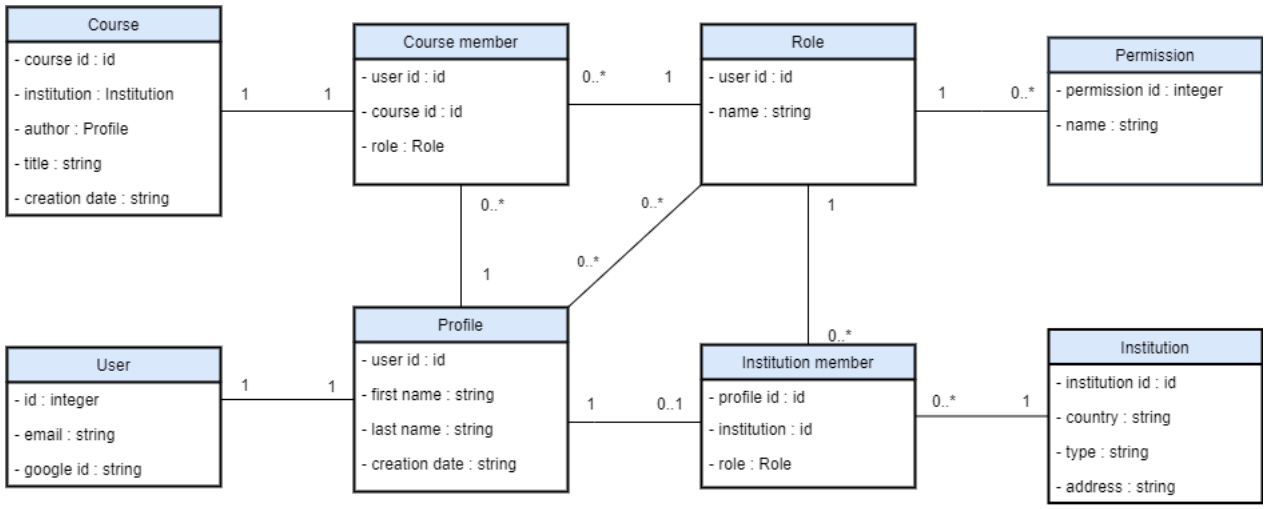


Figure 6.9: Class diagram of role-Based security

The idea is that each role is a collection of permissions and each role is associated with a profile either directly or indirectly through course members or institution member depending on the type of role. Course member and institution member would be new schemas that serve the purpose to denote the relation between a profile and a course or institution and the role associated with this membership.
As example, the profile might have an admin role associated, which would be a general role for the entire Educado platform. Another profile might have a role as an editor, but only on a specific course, thereby having a role indirectly through the 'course member'.
To give an idea of how the data will be laid in the database we came up with the following JSON schema primarily focusing on the extensions for the profile. Designing database models in a non-relational database like mongoDb, means that we can utilize a few tricks to make the implementation a bit shorter than otherwise would be in a traditional relational database. As an example, we can have arrays of records inside a single document to store a many-to-one relationship on the many side which would otherwise have to be done on the one side. As a result we can have the following in the profile, where we have `courseMember` as a list representing the courses that the profile have access to. This makes it very easy when we later have to check if the given profile has the appropriate permissions for some operation. The `groups` represents some overall permission to some profile, which is different from the more membership kind of role that is individual for each course or ins titution.

```
Profile: {
    ...  // Other fields omitted in this example

  groups: [
    EducadoAdmin, // Reference Role ids
    ContentCreator
  ]
  courseMember: [{
    course: 123,
    role: {
      id: 123,
      name: 'Course Editor'
    }
  }]
  institutionMember: {
    institution: {
            id: 123,
            name: "Aalborg University"
        }
    role: {
      id: 123,
      name: 'Institution member'
    }
  }
}
```

```
Role: {
  id: 123
  name: "Institution Owner",
  permissions: [
    "VIEW_INSTITUTION" : "View Institutions",
    "EDIT_INSTITUTION" : "Edit institution details",
        ...
  ]
}
```

Implementation To implement this, it became apparent that it was not necessary to have the permissions in the database. Instead, we opted to store the permissions as code, as we potentially need to reference them quite often in the code, and having CRUD requests each time was deemed as too cumbersome.

Their permissions could then be added to individual role classes. E.G. of an institution owner role:

```
module.exports =  InstitutionOwnerPermissions = {
  key: "ROLE_INSTITUTION_OWNER",
  name: "Institution owner",
  permissions: [
        Permissions.VIEW_INSTITUTION,
        Permissions.EDIT_INSTITUTION,
    Permissions.ASSIGN_ADMIN,
    Permissions.RESIGN_ADMIN
    ]
}
```

A problem that quickly arose, was that storing the designs locally instead of in the database created a problem regarding updating the roles and their permissions.
To circumvent this, the role collection in the database was cleared and repopulated each time the back-end was started. This made sure that any changes would also apply to the database.
By keeping the primary key, called key in the above code, the same value each time, any relation to the role would not be affected.

This resulted in the collection called roles:

Figure 6.10: Example of 2 items in the role collection

Unfortunately, role-based security was never fully implemented, due to time constraints.
What was missing was having a check on each API endpoint to check if the current user has the correct permission, through a role, to perform a request on the API endpoint. Additionally, the relation between the profile, course member, institution member and the role was never fully developed.
For future development, this would definitely be a priority.

7 Code quality

Maintaining good code quality ensures that future developers can familiarize themselves with the code and continuously develop valuable increments for the product. Code quality impacts a variety of areas, from user experience to application security. This is usually a challenge in assessing code quality across multiple development teams and improving it at scale in the current project. Increments that accumulate defects can decrease the overall value of the product. The right tools or processes can mitigate this by helping to ensure high code quality. It is crucial to prevent code quality issues at the source before developers introduce them.
In the Agile software engineering course, we learned that one of the definitions of software quality is:

> *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it[17, slide. 6]*

Ensuring code quality is important because it helps to deliver a useful product that provides measurable value for all parties involved, including the end user. In our project, code quality is particularly crucial because we are delivering the platform for content creators, who will be the main users of the web application. Furthermore, there will be future developers who will be responsible for further developing and maintaining the web application. One way to achieve great code quality is through testing, which is used to identify problems, improve usability, and validate the correctness of results. Our testing method is covered in Section 7.2.

7.1 Static code analysis

In this section, we discuss the use of *CodeScene*[18] and *SonarQube*[19], both of which are licensed code analysis tools. These tools conduct a static code analysis for any software product based on several factors. These factors help provide general feedback on the overall code by identifying hidden risks in the code itself.

Static code analysis is usually performed as part of a Code Review during implementation. It involves the use of static code analysis tools to identify potential vulnerabilities in non-running source code [20]. This can help to improve the likelihood of discovering security flaws with high confidence. In addition to security risks, static code analysis tools can be helpful for developers in avoiding unnecessary complexity in their code, making it more readable and understandable. They can also help developers to identify specific code sections related to code health and track how much of the original code has changed compared to the current code.

Using static code analysis tools during the development process is a powerful way to detect problems and provide immediate feedback to developers. This allows vulnerabilities to be discovered earlier in the development process, which is very useful. In addition to helping to identify security risks, static code analysis can also help improve the overall quality and maintainability of the code.

7.1.1 CodeScene

CodeScene is a static code analysis tool that was introduced in the agile software engineering course. It has various functions to analyze a codebase and help developers visualize, understand, and improve their software. When CodeScene detects bad code, it can also suggest ways to rewrite it to improve it.

Our expectations for codeScene are to get the results of two categories which can give us the overall code health score and knowledge distribution. Unfortunately, we were introduced to CodeScene late in the project, so we

were unable to benefit from their refactoring suggestions to improve our code iteratively alongside development.

The metrics we use to assess our project will be based on multiple factors that are scanned from the source code. Maintenance costs and defect risks are correlated with code health factors. According to CodeScene, a healthy codebase takes 124% less time to develop on average, enabling a faster time-to-market[21]. The number of defects in healthy code is 15 times lower than in unhealthy code, and healthy code also helps create new features twice as fast in development and reduces task completion uncertainties by a factor of nine [21]. Additionally, red code (code with poor health) is more vulnerable than white code (code with good health) when several code health factors are taken into account.

The knowledge distribution indicates how much of the code is written by currently active developers. A low number indicates that a significant amount of code is written by former contributors who have since left the organization. By obtaining both a code health score and a knowledge distribution, we can determine whether we have improved the web application back-end while refactoring and adding new features. In the following subsections, we compare the initial back-end, middle-stage back-end, and final back-end.

Initial codebase

At the end of the project, we conducted an analysis of the initial codebase to obtain a health score and a knowledge distribution of the project as we received it. However, it was not possible to get a detailed static code analysis because the initial codebase only consists of about 3819 lines of code in the back-end and front-end combined, making it difficult to determine if it is good code.

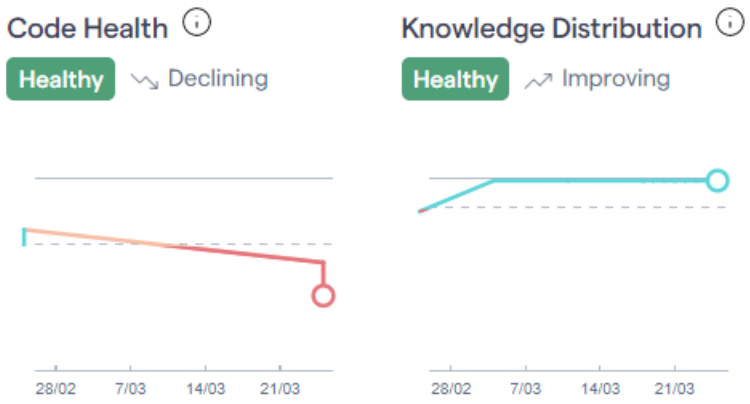


Figure 7.1: Graph overview of initial codebase

As shown in Figure 7.1, the graph shows the code health declining over time. Although this is what the analysis showed, it is still arguable that this codebase is very poor in both quality and quantity according to the standards of what an e-learning site should have in terms of possible functionalities and design patterns (see reference [22]). For example, the initial codebase does not provide a user-friendly experience or any course interaction through the uploading of exercises, since it does not support the creation of exercises with answers in the first place. This results in a non-functional e-platform site.

Figure 7.2: Code health scores - initial codebase

The scores for the code health of the initial codebase range from 1 to 10, with 1 being the worst and 10 being the best. As shown in Figure 7.2, the "worst performer" area has a score of 8.6, which is in the orange range. This means that there are some issues with the code in this area that need to be addressed, as they could potentially have a negative impact on the overall code health of the codebase. However, the other areas of the codebase have scores that are higher than 8.6, indicating that they are in better shape. In particular, the hotspot code health has a score of 9.5, which is considered to be good. This suggests that the initial codebase has only a small amount of technical debt, or additional rework needed in the code, compared to other areas.

Halfway codebase

Halfway through the project, we had made significant changes to the majority of the back-end codebase, including refactoring over half of the code and adding new features. We also started merging our code with the other back-end group (group 1) at the end of sprint 3. In CodeScene, we used the knowledge distribution feature to identify any old code from the initial stage of the product (see section 7.1.1) that was still being actively used compared to the current code. As shown in Figure .9, we can see that some parts of the codebase are still from the original stakeholders, Jacob Vejlin Jensen and Daniel Britz, but compared to Figure .8, which shows the entire codebase, there is a significant difference. CodeScene has determined that almost all of the code is made by the current team, which includes groups 1 and 2, indicating that the former code made by the stakeholders Jacob Vejlin Jensen and Daniel Britz is either refactored or deprecated.

Figure 7.3: Code health scores - halfway web application back-end

As another point of reference, we decided to find a middle ground and check whether the code health improved or worsened over time by conducting a code health check in CodeScene. To do this we backtracked through our commits to find a halfway mark of the project. As shown in Figure 7.3, one area, the *worst performer*, has improved, with a higher score indicating better code health. The other areas showed less of a difference, but the hotspot code health has a perfect score, indicating that we have also improved in this area.

Final codebase

At the end of the final sprint, we had made a lot of changes compared to the initial product (see section 7.1.1). One of the main changes was that we refactored the initial code, which slightly improved the code health and

quality. However, the amount of code that was analyzed in this final stage was about 4530 lines of code, which is double the amount of code compared to the initial stage.

Even though the difference in code health is a minimal change in the slightly positive direction, we can still argue that our refactoring and newly added code have significantly improved the quality of the original product. For example, the initial stage barely fulfilled any requirements of what an e-learning platform should be. It had no quantity or quality and was missing many functionalities and had outdated libraries in the back-end. In conclusion, the code of the initial stage did not even have any substance except an idea of what it was intended to be. compared to the current code, which has better scores as shown in Figure 7.4. This is even though we completely refactored the code and added some small additional features, such as improved login and the ability to add exercises to sections.

Figure 7.4: Code health scores - final web application back-end

Comparison between the final back-end, final front-end and final mobile application

By comparing our code to other repositories of the Educado suite, we can assess the quality of our code. In addition to the code health score mentioned earlier, the following figures provide a general overview of all the folders in bubbles. We will compare the final web application (both front-end and back-end) to the mobile application.
The results were outstanding, in the figures below 7.5, 7.6 and 7.7 we can see a map of the different parts of the code in each of the respective repositories.

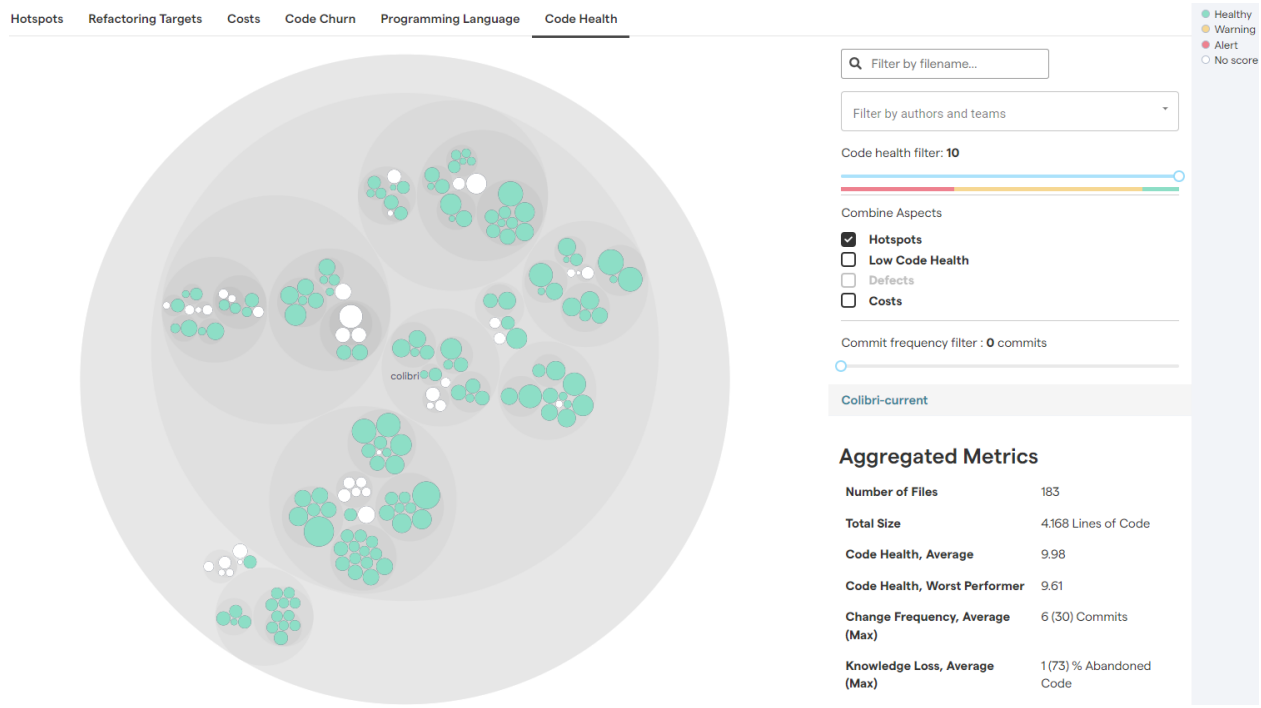


Figure 7.5: Code health of the whole colibri back-end

Figure 7.5 shows us that we have developed 4168 lines of code and achieved an average of 9.98 code health which is very close to 10 as described in the current colibri paragraph at 7.1.1. Whereas the worst performer is the course.js file located in the domain folder related to courses with a score of 9.61, it still has a rather high score. In any case, this provides us with important information about which files may cause us problems in the future. Although this part of the code health is the lowest, we are still generally satisfied.

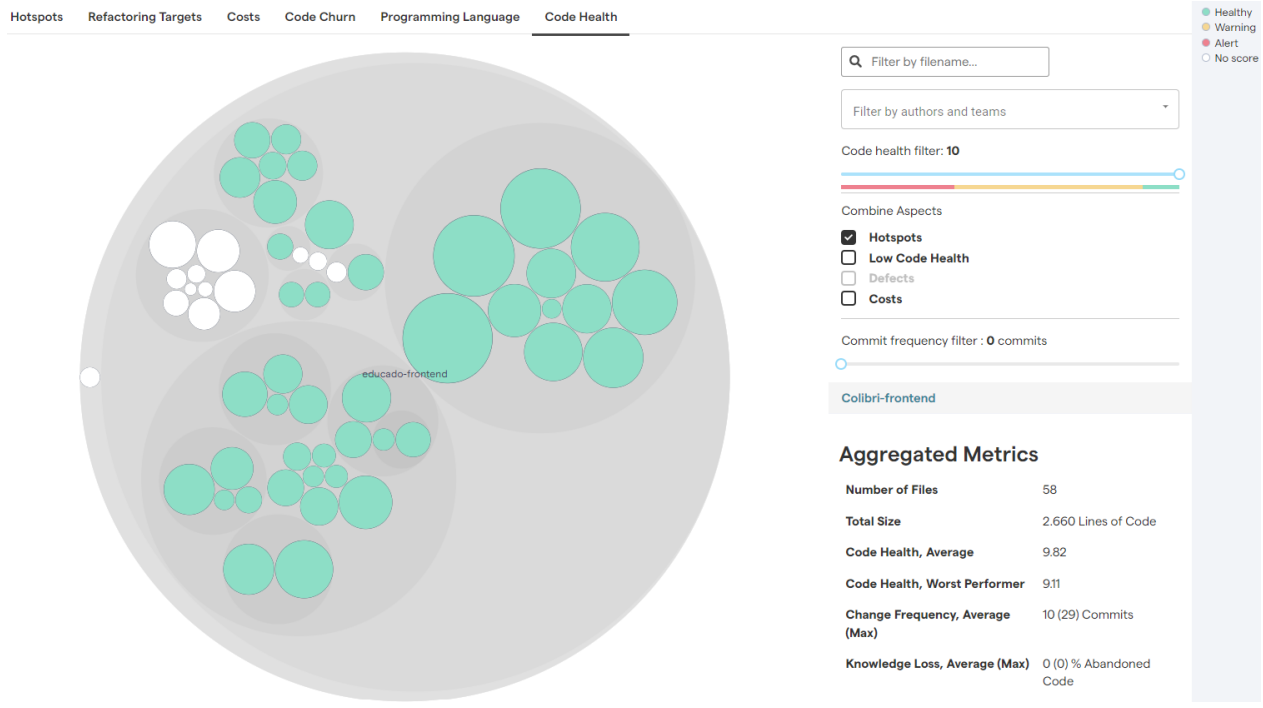


Figure 7.6: Code health of the web application front-end

Figure 7.6 shows us that we developed 2660 lines of code and an average code health of well over nine, it still a high score. The worst performer score does fall a little bit short of figure 7.5's worst performer score, but we are still very satisfied with the result. Even though we have two repositories which are split, they still have a high score while working together. Aggregating the results from both the web application back-end and front-end, the code health average is 9.9 and worst performer is 9.36.

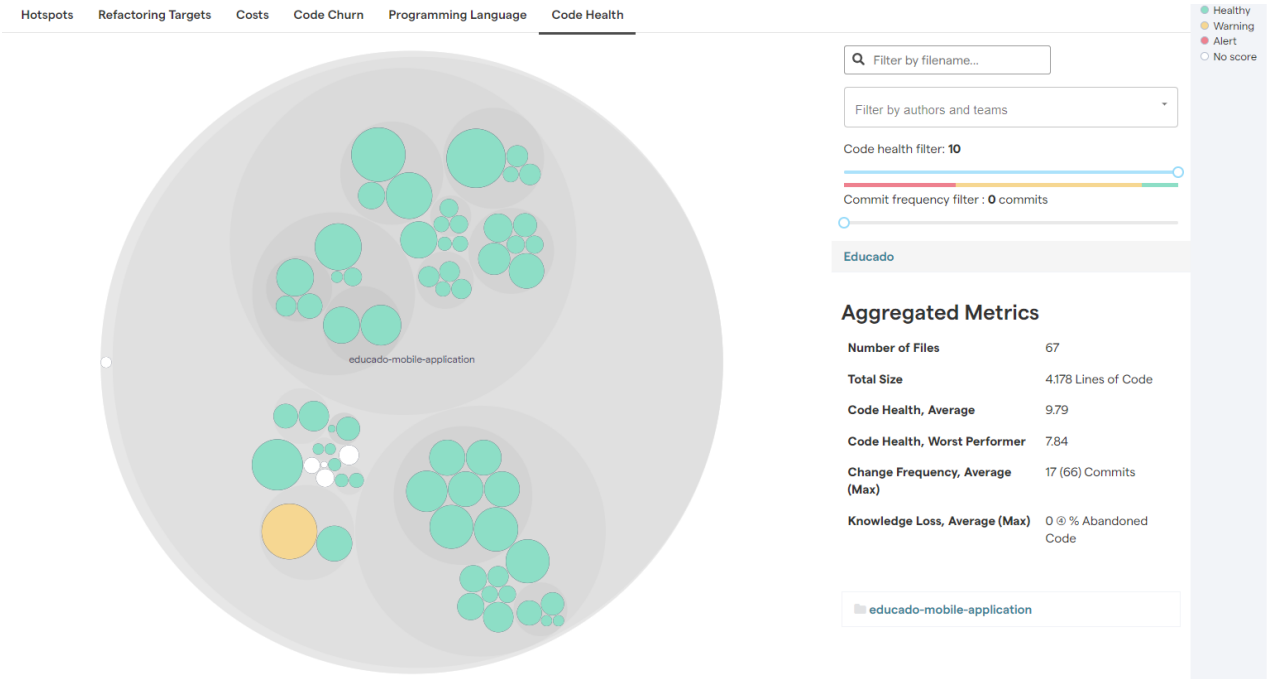


Figure 7.7: Code health of the whole mobile application

Lastly figure 7.7 shows the mobile application that group 1 and 3 has been working on. They have implemented 4178 lines of code with an average score of 9.79 code health. Whereas the worst performer out of all the files is a back-end related file, which could indicate that this file did not get as much attention as the other files. However, we now have some information as to which files may present us with future problems. Despite this code health score, we are generally satisfied with it.

As it stands now by accumulating the code health scores across all code that have been implemented by different groups, the web application repositories do get a higher score than the mobile application repository.

7.1.2 SonarQube

The second and last static code analysis tool that we were introduced to during the Agile Software Development course is SonarQube. SonarQube is an open-source automatic code review tool which aims to aid software developers to deliver clean code. It works by scanning a repository and provides feedback in four measurable categories: Reliability, Security, Security hotspots and Maintainability. In combination, these metrics help identify security hotspots, measure technical debt, and pinpoint problem areas that may contain bugs and code smells. Figure 7.8 below shows the overview of an analyzed repository, in this case, it is the initial state of the Educado platform, containing both the original front-end and back-end.

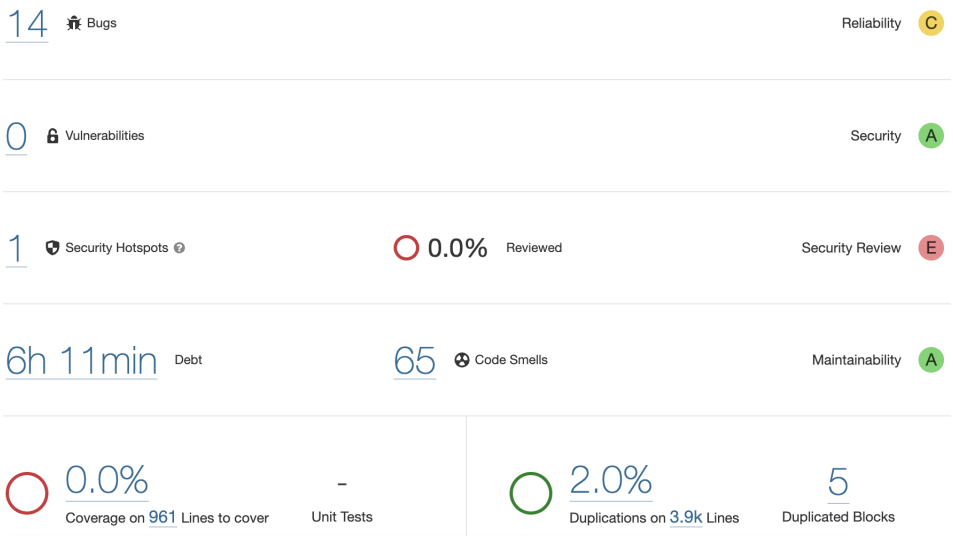


Figure 7.8: Overview of the initial project

SonarQube boasts a slew of features, from measuring reliability, security, and maintainability on more than 25 languages, to decorating pull requests, only allowing code of a customizable quality standard to be pushed. Not only does it identify possible problems in all these areas, it also provides important information as to where the issue persists, what risk is involved in the code as it is, and offers recommendations on how to resolve it. Figure 7.9 below shows the assessment of a potential security flaw detected by SonarQube, reviewed through their admin panel.

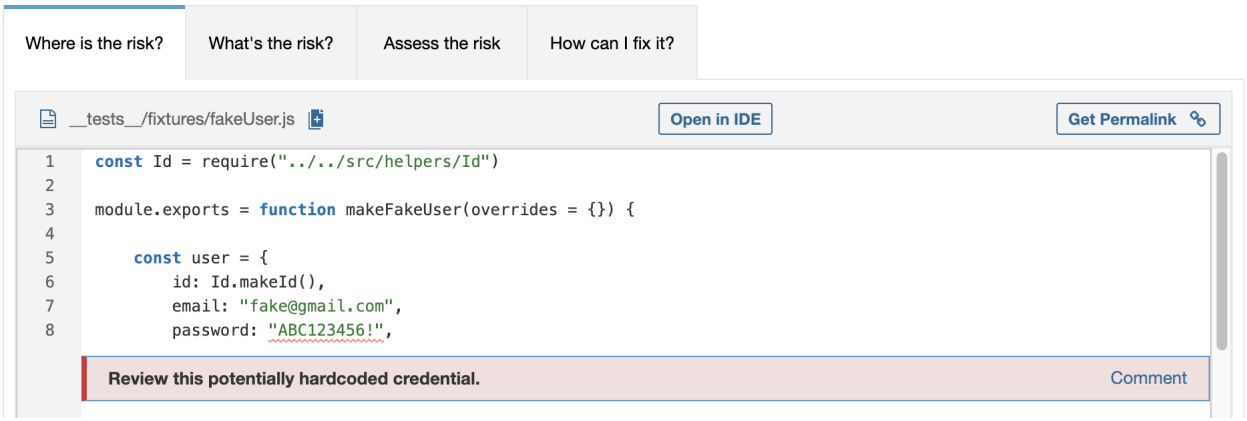


Figure 7.9: Assessment of potential security flaw with SonarQube

Unfortunately, as with CodeScene described in Section 7.1.1, SonarQube was introduced to us very late in the project. Therefore, we were not able to benefit from the many features it offers, towards improvements in code quality, during the development of the system. We can, however, use static code analysis to obtain some valuable insights regarding our current code quality and compare it to the quality of the initial project.

To get an overview of how the codebase has progressed in terms of code quality, technical debt, and security issues, we analyzed our current front-end, current back-end and the original Educado project containing both the initial front-end and back-end.

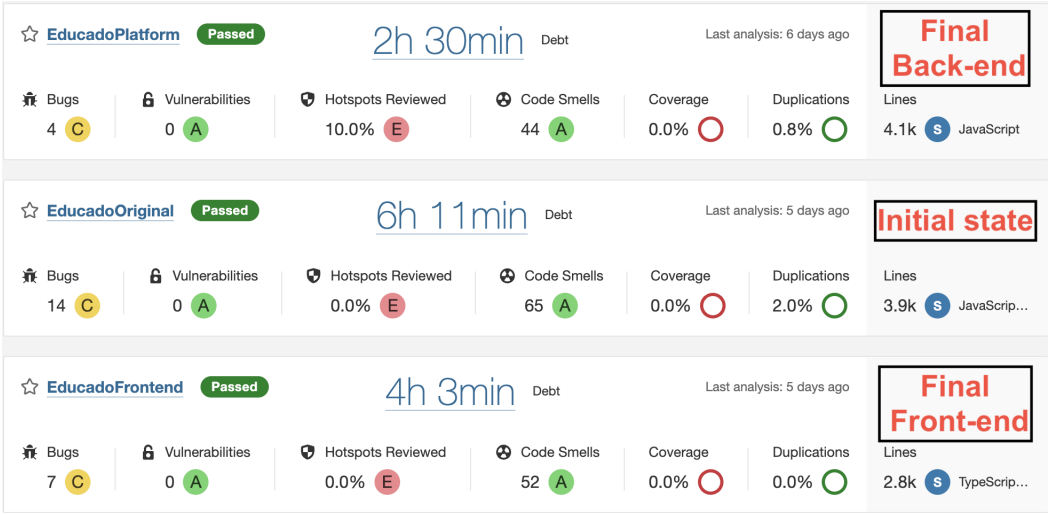


Figure 7.10: Comparison between analyzed repositories

Figure 7.10 above shows that although we increased the size of the project by approximately 3000 lines of code overall, we managed to maintain a reasonable level of technical debt. Our main focus was the back-end for the Educado platform which, as this figure also shows, has a minimal amount of technical debt, as well as very few bugs.

Diving into details with the technical debt, SonarQube is able to provide an overview of problem areas. This gives us a visual representation of where we can improve our code quality with refactoring. Figure 7.11 below shows the technical debt of the final back-end. On the x-axis we have lines of code for each file and on the y-axis we have SonarQube's estimated technical debt in minutes based on the severity of the code smells. The size of the circles express the amount of code smells contained within a file. As can be seen, the biggest culprits in terms of code smells are the files called index.js and appUser.js, having three code smells each.

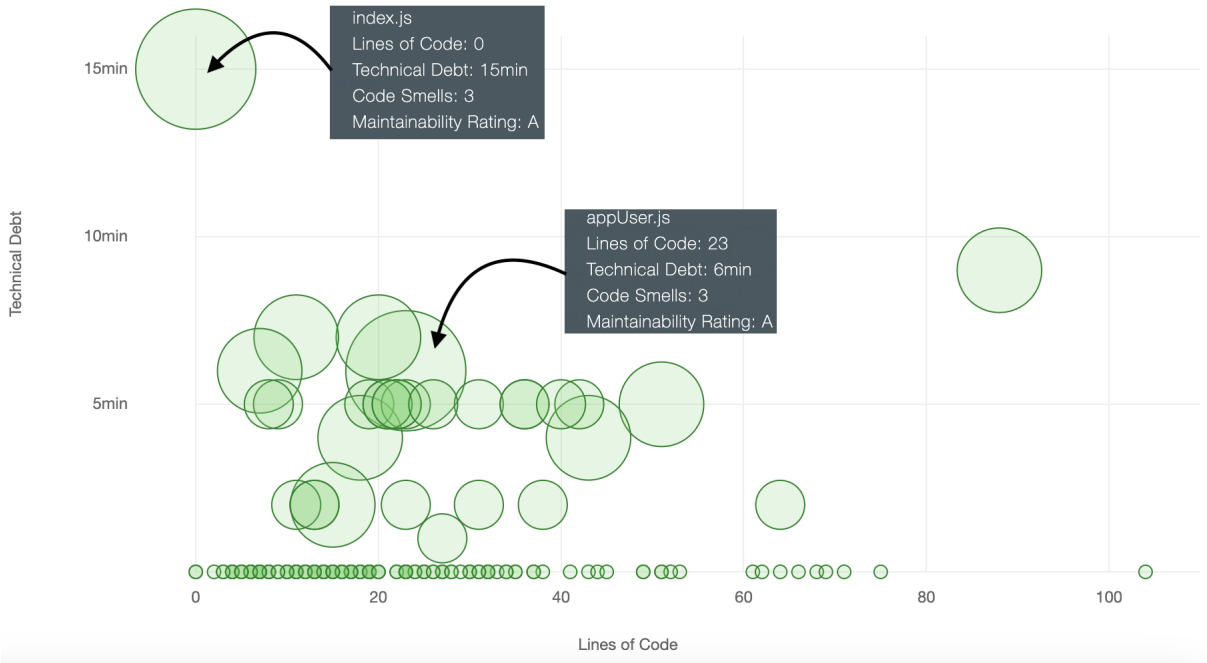


Figure 7.11: Technical debt of the final back-end

In conclusion, SonarQube is an extremely powerful tool if applied correctly on a project. The group members are novices in our usage of the tool at best, but would unquestionably have been able to deliver a codebase of higher quality, had we been introduced to the tool at an earlier stage in the development process. The efficiency of the tool increases alongside the scope of the project, with larger projects benefiting even more from the analysis, and will as such prove even more valuable when applied to future developments.

7.2 Testing

We can not mention code-quality without bringing up testing. Testing is really an integral part of software quality be it manual or automated testing. The importance of testing only grows with time as a software system evolves and more code is added, which means more places where things might go wrong. Hence, with a growing project such as this one where many people might introduce failures in the system its important to have tests that can quickly give feedback on the impact of new changes.

Working to make the code more testable

Working from an existing codebase where testing clearly was not in the mind of the creators, we had to refactor quite a bit in-order to get to a point where we could do unit-testing. Having refactored to a clean architecture besides the benefits of isolating external libraries also meant that things were much simpler to test. We could now test things all the way down to individual rules of entities within the domain up to integration level tests at the request/response level. The process of writing tests would be a mixture of sometimes writing them before writing the actual code in a TDD [23] fashion and other times after writing the actual code to verify an aspect of the code. When trying to write the tests before the code, it sort of guides the design of the actual code by having to think of how it is going to be used. The real benefit is that the test code almost gives a step-by-step guide on what to write to fulfil the need in the system. By writing how we imagined the code to be used and then making the effort on making that reality made for a very good workflow which felt like the real benefit of writing the tests first. Having the automated test afterwards that can be run to verify an aspect of the code felt like a secondary consequence of doing it this way.

Example of a test

To get an idea of how the tests for the back-end is written, let's have a look at one of the tests to see how it is done. To give some context, the tests are written with the Jest testing framework that allows creating a test using the `it` keyword. The tests make use of fake data to make test data more reusable between tests but also allows overrides if the tests need to focus on some specific part of the data. The tests follow the structural principle of given X, when doing Y, expect the result to be Z. This makes every test easy to follow. The test below checks that sections successfully get added to a course in the database. In order to test the functionality, an existing course needs to be in the database. For this, the function `setupCourse` does the job of setting up all the resources that a course needs to have in the database. Consequently, the test ends with ripping down all the resources related to the course. Other than that the test is pretty straightforward: two fake sections get created and added to the course via the use-case `addSection`, then it finds the course and asserts that indeed the amount of sections it have is two.

```
7   describe('Add section', () => {
8
9       it('successfully adds sections to a course', async () => {
10          const fakeCourse = makeFakeCourse()
11          await setupCourse(fakeCourse)
12
13          const section1 = makeFakeSection({ title: "Section 1" })
14          const section2 = makeFakeSection({ title: "Section 2" })
15
16          await addSection({ info: section1, toCourse: fakeCourse.id })
17          await addSection({ info: section2, toCourse: fakeCourse.id })
18
19          const course = await courseList.findById(fakeCourse.id)
20
21          expect(course.sections.length).toBe(2)
22
23          await teardownCourse(fakeCourse)
24      })
25  })
```

Figure 7.12: Example of an integration level test

Issues moving to integration testing

The transition to testing at a higher level than unit tests significantly increased the complexity of the test code. In order to have accurate and useful tests, we needed a way for the tests to interact with the database. Connecting the tests directly to the actual database was not an option because we didn't want the tests to interfere with the real data in the database. From previous projects we have learnt that testing with an actual database over the network just leads to slow tests and slow tests means not running and writing tests at all. As an alternative, we decided to use an in-memory database that mimicked all the interactions of the actual MongoDB database. This allowed us to keep the code for interacting with the actual database unchanged and identical. Using the in-memory database significantly sped up development time by reducing the time it took for tests to run from seconds to milliseconds.
One problem we encountered during testing was inconsistent test runs. Sometimes a test would fail when run together with other tests but would succeed when run individually. The issue was that each test by default would be run in parallel and letting each test share the same in-memory database meant, that it would often happen that two tests, one adding a resource and another test cleaning up resources, would break the assertions they were making. After experimenting with various complex solutions to keep each test separate, we ultimately decided to simply not run tests in parallel in order to have more control over the resources that each test had access to. This fixed the inconsistent tests, but it took a significant amount of time to investigate the issue.

Takeaways from testing

Despite the wonders of having tests, they do come with a downsize of having to be maintained just like any other code. If not written at a sufficiently high-level the tests became fragile and susceptible to breaking whenever the production code it was meant to test changed which in cases ended up becoming a hurdle to refactoring.
Setting up the environment to be able to test more complex interactions also was hard and required a lot of thought and time to work properly. Hopefully the effort is well paid for future developers working to test their code not having to go through the struggles of setting it up.

8 Discussion

In this section, we evaluate and discuss the overall success of our project and the shortcomings along the way, as well as what we could have done if we had more time to implement more functionality to the content creator platform. The focus points will be about how we have handled teamwork internally in the group and externally with the other groups, product owner, and stakeholders, and finally how we have handled working with the agile scrum framework which was a big part of the project.

8.1 Agile learning

A point that was emphasised more than once by our semester coordinator, was that the purpose of this project, for us students, was to learn, more than it was to produce a product. It gave rise to complications, but at the same time, it also provided a better learning experience, since we had ample opportunity to learn from our mistakes. In light of this, the fact that we were introduced to scrum concepts steadily through the project, which meant we did not understand the importance of the scrum master role or daily scrum, nor how to work with two other groups from the start, now makes sense.

8.2 Current state

We took over the project assuming our only role would be as back-end developers, as per our course description. It soon dawned on us that it was also necessary to develop further on the front-end application, as requested by the product owner and the semester coordinator. This heavily divided our focus into multiple areas, even if our primary focus should have been the back-end. Compounded by the new agile development method and more meetings than we could count, our progress was slowed. What manpower we could dedicate to back-end development was focused on refactoring the two codebases into something sustainable and maintainable in the future, as we deemed this to be more important than higher velocity. This heavily moved our focus from developing code to developing system architecture. This unfortunately meant that we did not get to implement all the features we had hoped for and left a larger backlog than we had initially expected, although a much more robust architecture.

8.3 Future works

This section describes the areas of focus that, in our opinion, could improve the quality of the platform, and therefore should matter to the next generation of developers.
In this project we had a large backlog of features. Although important to the platform as a whole, many were not implemented, because other features took priority. Our top priority was to focus our development efforts on features related to course creation, because we had to align and integrate our work according to the dependencies of the other groups.
Both front-end and back-end underwent extensive refactoring, from the initial application, we inherited from the original developers. On the front-end we mainly implemented the basic course creation functionality for content creators plus some 'quality of life' features. The list of features from our product backlog, that still needs to be implemented consists of the following items:

- Unpublishing a course
- Role-based access control
- Course editor role

- Improved course management
- Restriction rules for the Educado admin page
- Allowing users to sign into the web application with both Google OAuth and JWT-based authentication
- Live preview of the course in the course editor

8.3.1 Reasons for each feature

The unpublishing feature is needed in case a course needs to be temporarily taken down for maintenance purposes or reasons pertaining to rules violations.

Role-based access control is needed to restrict access to various parts of the web app to certain roles such as Educado- and institution administrators, course owners- and editors.

The role of course editor is needed to assist course owners create and maintain content but with fewer privileges.

To improve course management, the ability to delete entire courses or individual sections and exercises is required in order to avoid cluttering the system with outdated or inappropriate information.

In order to improve the content creator platform, future developers could look into restricting the Educado admin page for base users. This could have been easily implemented but the current endpoint for user information fetching, did not include information about the user's admin status.

While we are discussing the improvements to the user and authentication, it is of note that the initial solution had Google (OAuth) authentication setup up for the platform, whereas the final solution relies on JSON web tokens (JWT) for authentication in the front-end. The back-end supports both authentication strategies, but after a meeting with the product owner and the stakeholders, we found out that it was preferable that content creators are able to sign up without having a Google Account. Since both strategies are currently implemented in the back-end, the platform would only improve in user experience if both options were available to the content creators, because we would be able to better accommodate the authentication preferences for new users.

8.4 Working with scrum

Working with Scrum was definitely a challenge for us in the beginning, since it was the first time we were working with agile concepts in practice. The challenges were mainly situated in the fact that Scrum requires a significant shift in both mindset and approach to how work is organized, prioritized and carried out, when compared to more traditional methods like the waterfall model, where we usually plan out all the requirements in advance, then follow a more linear and procedural plan. In contrast, Scrum is an agile framework that emphasizes flexibility, collaboration, and continuous improvement.

8.4.1 Product owner and stakeholders

The product owner was initially not very effective at their role, but they improved significantly as they gained more experience and gained a better understanding of the project and its goals. Unfortunately, the stakeholders were not very engaged in our work and mostly communicated with us via email. They showed up during the sprint reviews twice within the 6 month period, which made it difficult for us to get the support and guidance that we needed.

This taught us an important lesson: The absence of effective communication and collaboration between the stakeholder, product owner, and development teams, can severely impact the efficiency and velocity of the entire team. One reason for the lack of clear communication could be contributed to our inexperience with the framework itself. In order to avoid this particular issue in future projects, we need to have the stakeholders who are interested, present and engaged in the sprint reviews. This would not only allow us to present our increments to them, but also provide us insight into their wishes and wants for the end product. Working closely with stakeholders would ensure that the work being done aligns with the overall goals of the project.

8.4.2 Scrum in practice

As mentioned earlier, one of the hardships we encountered in the duration of this project was our own lack of experience with agile methods, specifically the scrum framework. We learned about the agile methodology in parallel with this semester's project. This meant that at times, we were facing challenges, whose solutions we learned about later in our Agile Software Engineering course. But these types of situations also made it possible to learn from experience rather than just learning the theory in our lectures. This type of applied learning, meant that we got better with time, thus ensuring that we became more proficient in applying agile methods in real scenarios.

8.4.3 Scrum master & daily scrums

During this project, each group member had the opportunity to try out the role of scrum master. Although we rotated the positions, being a scrum master in this project was predominantly about the learning experiences and the different responsibilities of this role. Due to our lack of knowledge within the scrum framework, our ability to be effective as scrum masters only stretched as far as our current understanding of the role. One of the responsibilities of the scrum master is to promote the key activities of the scrum framework, one of these is to conduct daily scrum meetings. Our team's daily scrum meetings were something we struggled with due to the short 15-minute time-box allocated to the meetings. Our daily scrums often turned to ordinary 60-90 minute discussions. If our scrum masters where better at moderating these discussions, we might have had more success with the daily scrum method.

8.4.4 Sprint backlog

When it comes to the management of the product backlog for this project, we developers were often in charge of coming up with our own user stories and user story prioritization. Under ordinary circumstances, the Product owner should have been responsible for creating the product backlog and working with the scrum teams in deciding which items to put into their sprint backlog. With an understanding of the premises for this project, a more controlled and less chaotic setup would probably have been beneficial for the learning experience. We have learned that our user stories were often too many and too complex. More than often we ended up with a user story with a completion level shy of the definition of done. A final note for our sprint backlog management is that it would have been beneficial for us to create tags for our backlog items. These tags would have been useful for the team when evaluating our backlog and stories.

8.4.5 External Collaboration

That our group did not work on-site at AAU on the same days as the two other groups was an issue, because it made it more difficult to resolve dependencies or clarify features they were working on. While some things could be resolved over Discord, there is no substitution for face-to-face communication.

9 Conclusion

We signed up for this project to work with complex back-end, guided by the following problem statement: *Improve the Educado platform and transform it into a great functional Mobile Education solution for Waste Pickers to be tested in the field by the end of the semester in Brazil.* More specifically, we worked on the back-end of the content creator web application of the Educado platform. The end goal being the codebase should serve as the groundwork for future semester students to develop upon.

We faced challenges as we were new to Agile, especially considering the trouble that followed trying to create a cohesive product across multiple teams. A challenge at the start was that we operated as three individual teams working on three projects instead of one cohesive entity. From facing issues with integration, we learned the importance of effective communication and collaboration when prioritizing work and finishing a cohesive product.

The heavy refactoring of the back-end structure into a clean architecture was a major investment that took a lot of effort to have in place, but the result was the isolation of different concerns, while also making it library agnostic, within the codebase increasing the testability and maintainability of the back-end.

Even though the front-end of the content creation platform was not our team's primary focus, we have managed to provide a highly scalable application. The application is now designed around best practices and provides improved quality of life for future developers with TypeScript support, unified data fetching strategies and a better self-documenting codebase.

One of our contributions to the platform has been the ability to have an onboarding process for new content creators. The new on-boarding process has made it easier for new creators to join and thus improve the scalability of the platform.

To have some useful content for the waste-pickers we developed in parts with the other groups a way to create new short-formed learning videos that include follow-up questions to test their knowledge.

Near the end of the project, we employed static code analysis to determine the code health of our current code base versus the initial code base: Our code health has improved on all 3 metrics: average code health, worst performer and in hot spots, resulting in a project with a healthier codebase than the one we overtook initially. The ability of static analysis tools to pinpoint weak spots in the code could have helped massively in the development and maintenance process to create consistently good code. Something we, due to its late introduction, did not utilize, but for future reference should have.

In order to control the different access levels of resources on the platform, an initial design for role-based security has been devised. Although not nearly implemented, we see it as an essential step to provide course collaboration features to the platform.

In conclusion, we obtained a thorough understanding of agile processes, which helped lead us to develop an improved overall architecture of the codebase. Thus increasing the maintainability and ease of development for the next semester's students overtaking the project, while providing valuable new features in close collaboration with multiple other teams.

References

1. Accessed: 05/12/2022, 2022, (https://www.statista.com/statistics/530481/largest-dump-sites-worldwide/).
2. K. Vasarhelyi, *Environmental Center*, Accessed: 05/12/2022, (https://www.colorado.edu/ecenter/2021/04/15/hidden-damage-landfills) (Apr. 2021).
3. D. Britze, R. N. Nielsen, "Mobile Education Platform - Smart Caching Learning Materials", AAU Student Report.
4. D. Britze, J. V. Jensen, "Digital learning platform for waste-pickers in Brazil", Bachelor Thesis (Aalborg University, May 2021).
5. P. A. Nielsen, *software 5 introduction*, Dec. 2022, (2022; https://www.moodle.aau.dk/pluginfile.php/2728021/mod_resource/content/1/Introduktion%5C%20SW5%5C%20E22.pdf).
6. *Clean Coder Blog*, (2022; https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html).
7. *The Agile Manifesto*, (2021; https://agilemanifesto.org/principles.html).
8. *What is Scrum?*, (2021; https://www.scrum.org/resources/what-is-scrum/).
9. *Scrum Glossary*, (2021; https://www.scrum.org/resources/scrum-glossary).
10. *[Myth Busting] What Is A User Story?*, (2021; https://www.scrum.org/resources/blog/myth-busting-what-user-story).
11. *User Stories are Needs Described from the Business Perspective*, (2021; https://www.scrum.org/resources/blog/user-stories-are-needs-described-business-perspective).
12. *Scaling Scrum with Nexus*, (2021; https://www.scrum.org/resources/scaling-scrum).
13. *Email Delivery, API, Marketing Service*, (2022; https://sendgrid.com/).
14. Philipp, *Answer to "MongoDB schema diagram"*, Dec. 2015, (2022; https://stackoverflow.com/a/34369991).
15. *passport-jwt*, en, (2022; https://www.passportjs.org/packages/passport-jwt/).
16. auth0.com, *JWT.IO - JSON Web Tokens Introduction*, en, (2022; http://jwt.io/).
17. D. Russo, *Software Quality*, Dec. 2022, (2022; https://www.moodle.aau.dk/pluginfile.php/2762819/mod_resource/content/1/ASE-7.pdf).
18. *Software Engineering Intelligence - CodeScene*, en, (2022; https://codescene.com).
19. *Code Quality and Code Security | SonarQube*, en, (2022; https://www.sonarqube.org/).
20. *Static Code Analysis | OWASP Foundation*, en, (2022; https://owasp.org/www-community/controls/Static_Code_Analysis).
21. *How It Works*, en, (2022; https://codescene.com/how-it-works).
22. *5 Characteristics Of A Successful eLearning Course*, (2021; https://elearningindustry.com/successful-elearning-course-characteristics).
23. K. Beck, *Test Driven Development. By Example (Addison-Wesley Signature)* (Addison-Wesley Longman, Amsterdam, 2002), ISBN: 0321146530.

10 Appendices

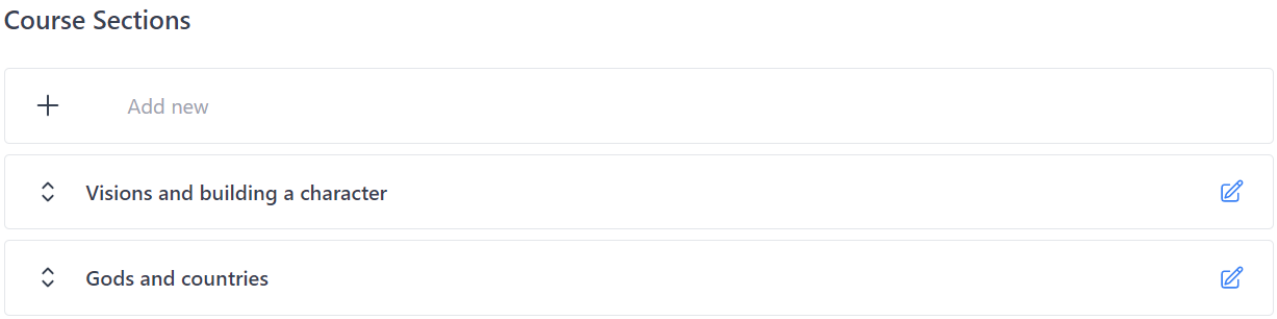


Figure .1: Add sections in current Educado

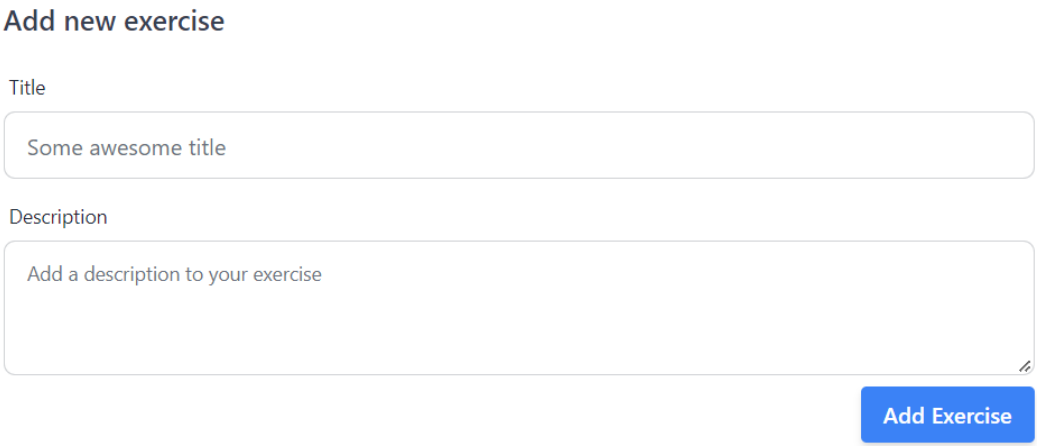


Figure .2: Add exercises in current Educado

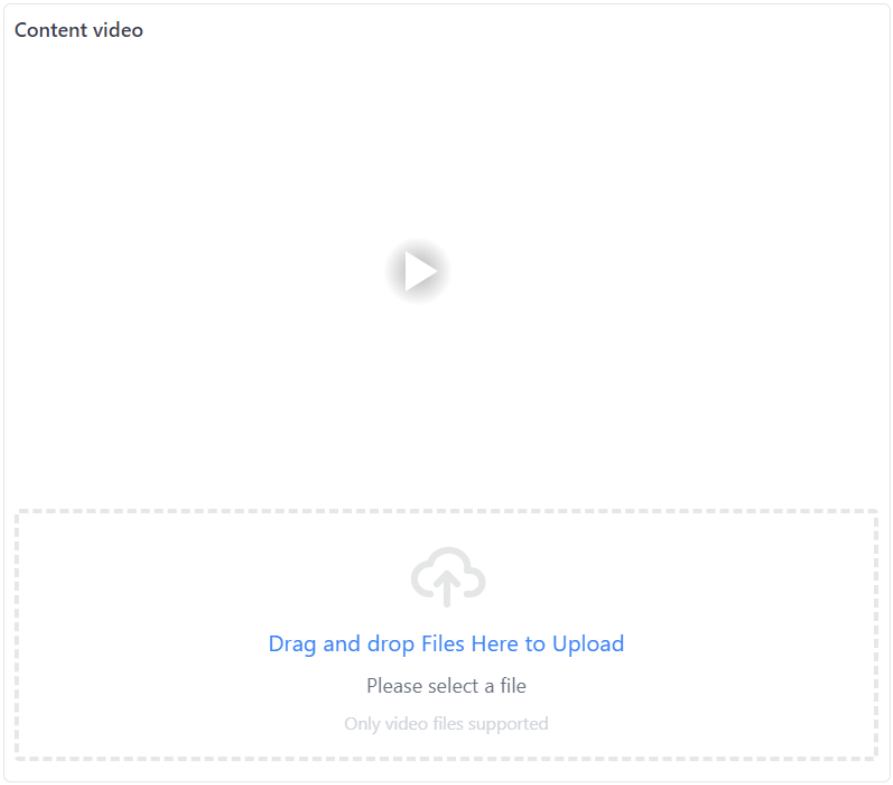


Figure .3: Upload content video in current Educado

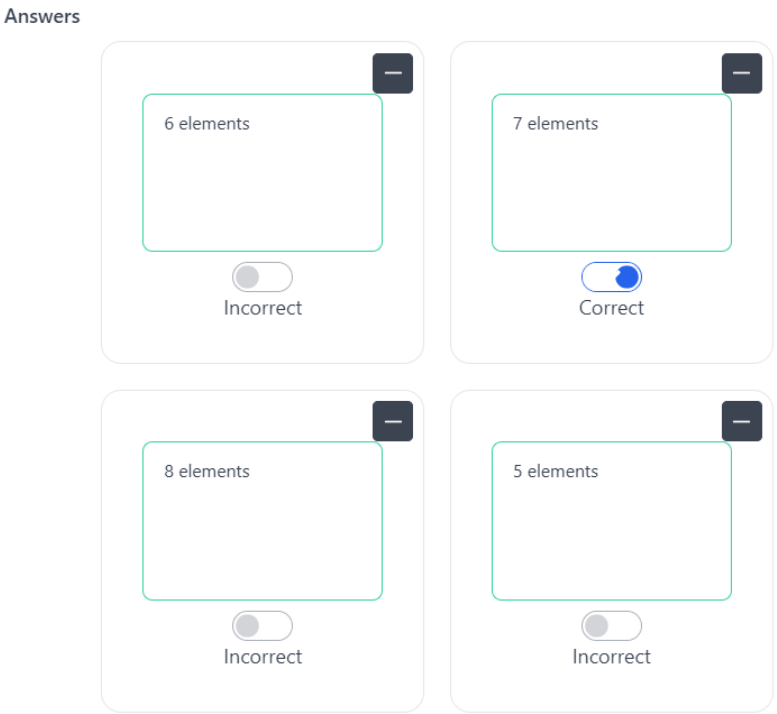


Figure .4: Choose correct and incorrect answers in current Educado

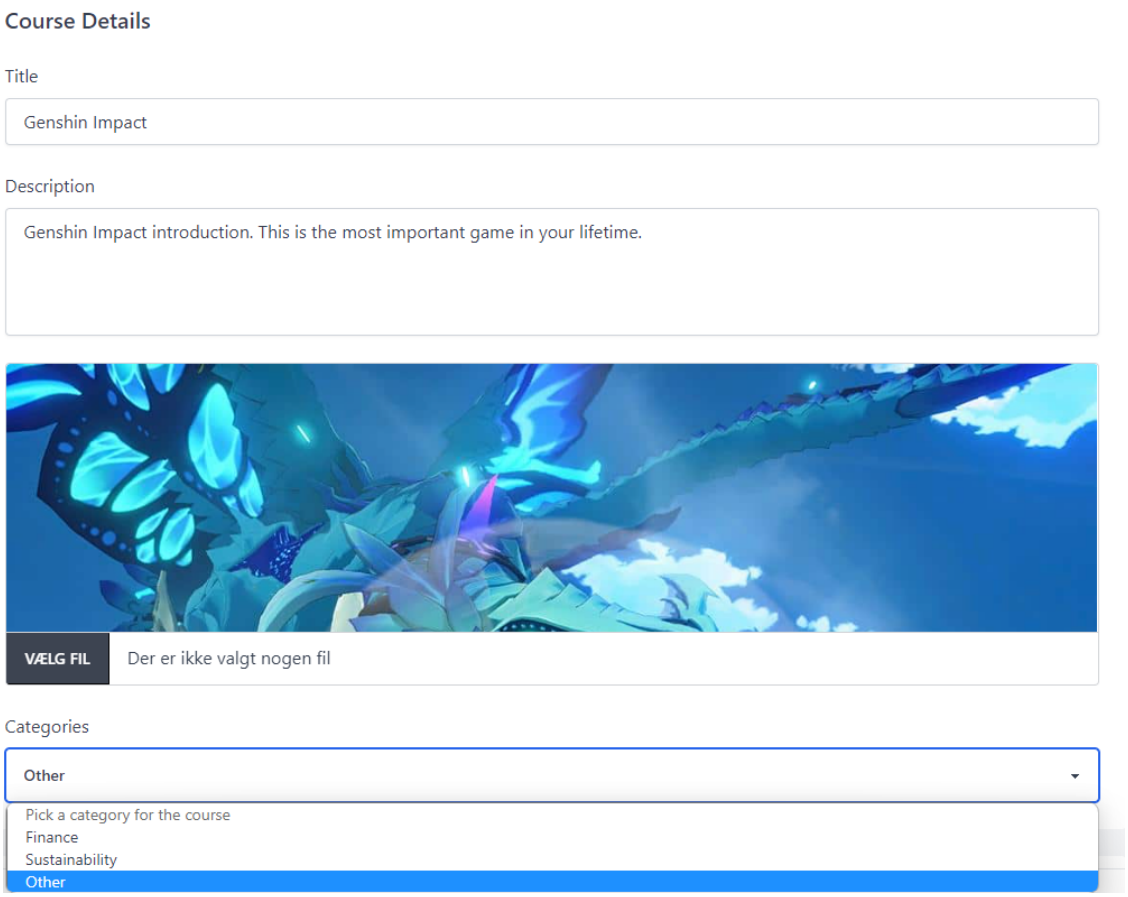


Figure .5: Edit course in current Educado

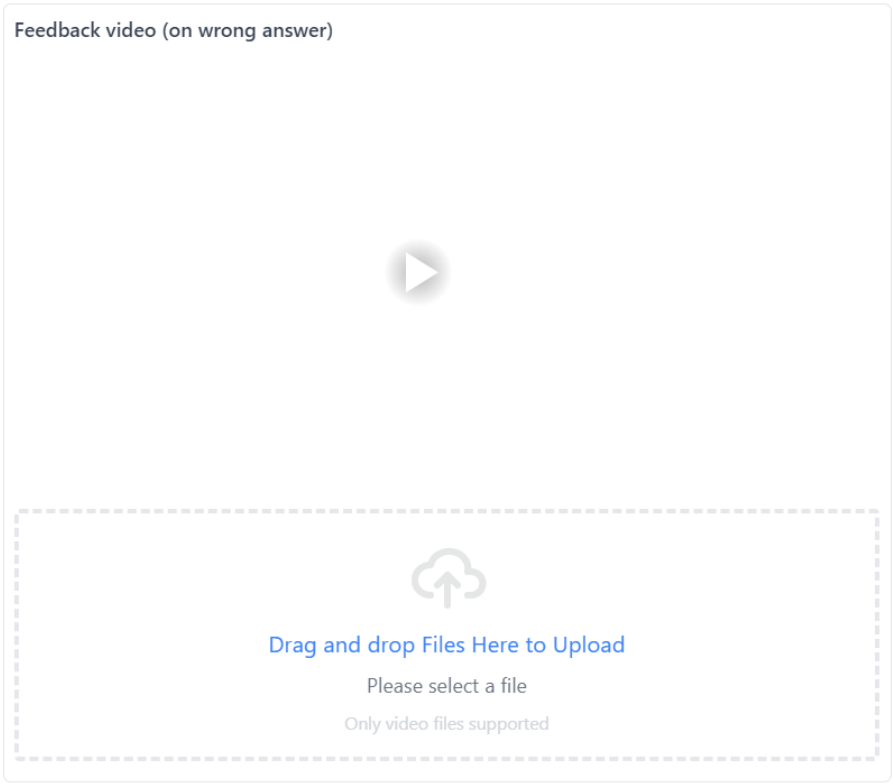


Figure .6: Upload wrong answer video in current Educado

Figure .7: Knowledge distribution of the middle version code base of Colibri

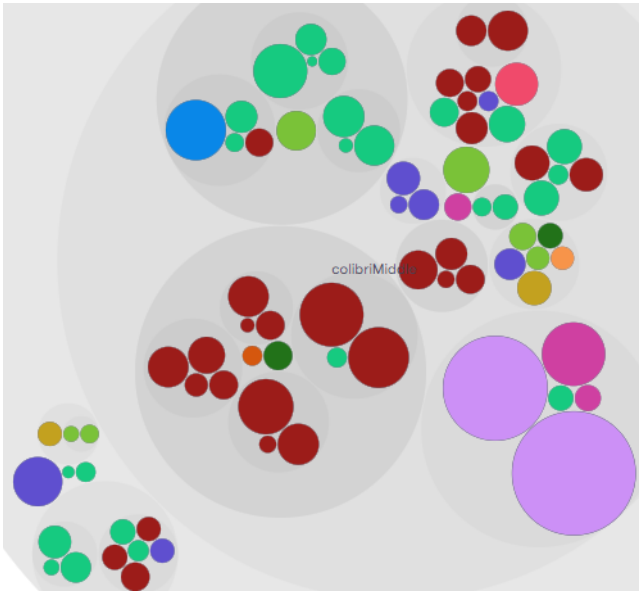


Figure .8: Colibri middle main authors

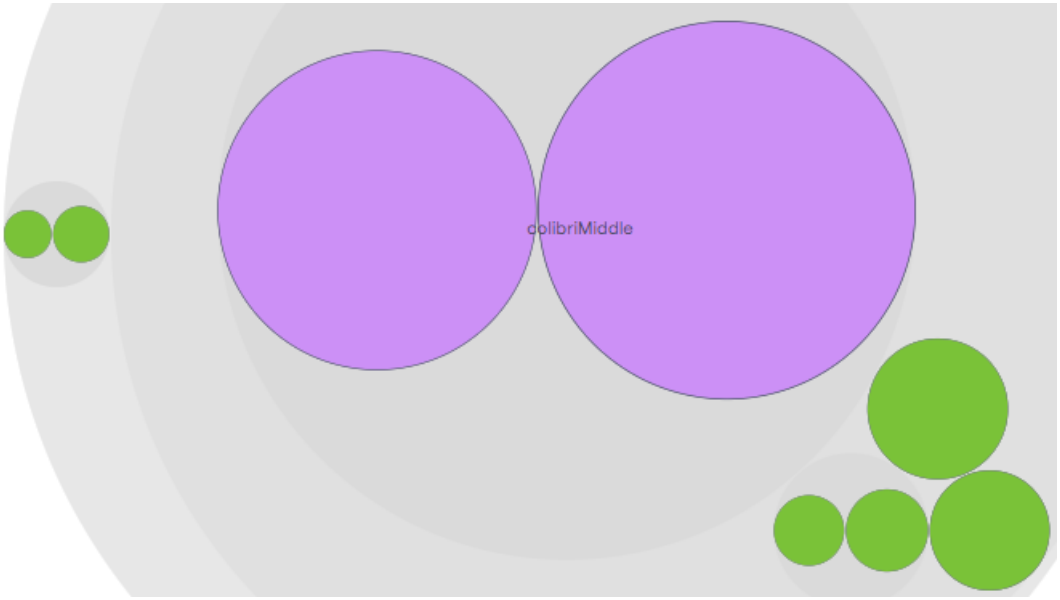


Figure .9: Colibri middle main authors filtered

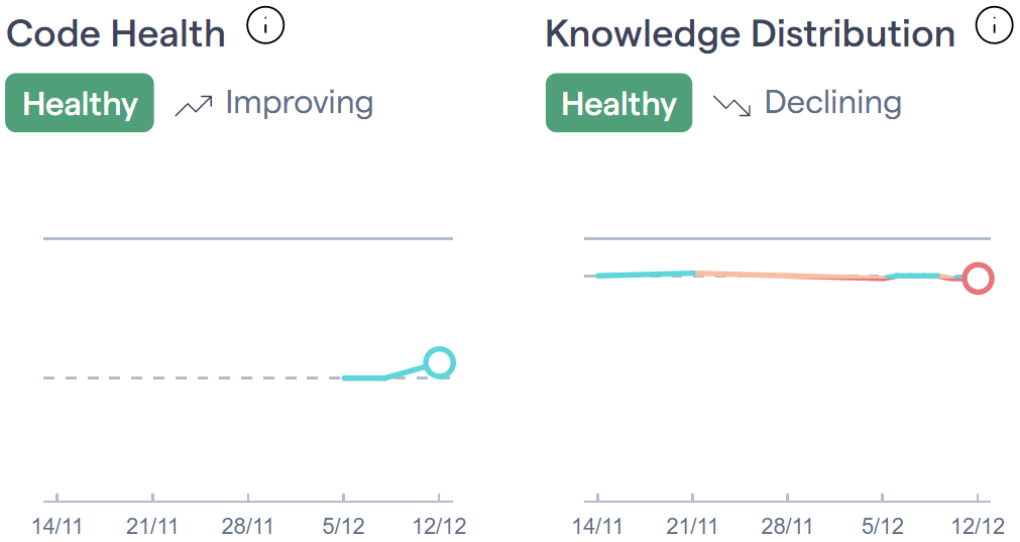


Figure .10: Colibri Final stage- knowledge loss

A Group contract

Generelt

§1. Der skal være mulighed for at revidere gruppekontrakten hvis flertallet af gruppen stemmer for.

Arbejdsdagen

§1. Hver arbejdsdag startes med scrum-statusmøde og en dagsorden med mindre andet aftalt.

§2. Projektmål og -struktur revideres ugentligt.

§3. Alle deltager i alle samtaler om projektrelevante emner.

§4. Små opgaver kan uddelegeres til enkelte medlemmer, men alle skal have det fulde overblik.

§5. Visuelle hjælpemidler benyttes i videst mulige omfang - eksempelvis benyttes whiteboardet ved alle gruppemøder og -diskussioner.

Arbejdsfordeling

§1. Vi holder scrum møde for teamet hvor der diskuteres arbejdsopgaver for alle sub-teams.

§2. Ugentligt holdes Status for hvordan teams har arbejdet med opgaverne.

§3. Der skal være plads til at vi skifter sub-teams efter behov, så alle for chance for at arbejde med alle.

Regler for status møde

§1. Status mødets længde skal helst ikke være meget længere end 30 min

§2. Der skal opsamle hvad der er blevet lavet i hvert team

Vejledersamarbejde

§1. Der skal aftales en dagsorden for vejledningsmøde forud for mødet.

§2. Ved første vejledning, aftales der på forhånd hvad der forventes af samarbejdet.

§3. 2 personer vælges til at skrive noter til hvert vejledningsmøde.

Tidsplanlægning

§1. Alle arbejdsdage starter klokken 9 (medmindre andet er aftalt på forhånd).

§2. Al information omkring vigtige mødetidspunkter skal være lagt ind på Google Kalenderen, så alle er klar over hvornår vi i gruppen mødes.

Programmerings skik og arbejdsform

JavaDoc-format:
/** * Prints out "Hello World"
* and the command line arguments.
* @param arg A string array containing
* @return No return value. */

§1. Der skal laves en funktionsbeskrivelse der forklarer hvad funktionen tager af input og hvad den gør. (JavaDoc-format)

§2. Tænk over gode logiske variabel, deskriptive og funktionsnavne.

§3. Vi benytter den korrekte casing af text i hh. til oracle :

- (https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html)

§4. Konstanter defineres i screaming snake_case.

§5. Variabler defineres i pascalCase

§6. Classes and interfaces defineres i CamelCase

§7. Lav små meningsfyldte commits, med en beskrivelse af hvad der er blevet ændret.

§8. Sørg for at køre koden inden koden bliver committet for at undgå fejl.

§9. Rapportens dele skrives på engelsk.

Faglige forventninger

§1. Alle medlemmer skal vide, hvad gruppen laver.

§2. Faglige diskussioner holdes indenfor emnet.

§3. Visuelle forklaringer benyttes om muligt.

§4. Alle skal deltage nogenlunde ligeligt i såvel programmering som rapportskrivning.

§5. Aftalt hjemmearbejde skal overholdes.

§6. Produktet skal gennemgås og afleveres gennemarbejdet.

§7. I udgangspunktet forventes timer brugt svarende til normeringen (ca. 20 timer om ugen for P5), og denne forventning kan om nødvendigt øges nær deadline.

Sociale forventninger

§1. Gruppens primære kontakt foregår gennem Messenger/Discord.

§2. Alle skal tjekke Messenger (mindst) dagligt (før klokken 22:00, hvis alle skal reagere på det).

§3. Gruppen mødes på campus alle hverdage, med mindre andet er aftalt.

§4. Alle møder på det aftalte tidspunkt, og der gives besked, hvis man er mere end 5 min forsinket.

§5. Alle lytter til og respekterer hinandens person og meninger.

§6. Konflikter og uenighed løses ved demokratisk afstemning i gruppen.

Fravær

§1. Alle skal i udgangspunktet møde til alle forelæsninger og møder (arbejde/andet er gyldigt, i et vist omfang).

§2. Bliver man syg, eller har man anden god grund til fravær, gives der besked i rimelig tid.

Konsekvenser

§1. Gruppen kan når som helst, efter brud på denne kontrakts bestemmelser, stemme om, hvorvidt et gruppemedlem skal tildeles en advarsel.

§2. Uddeling af advarsel kræver almindeligt flertal blandt gruppens øvrige medlemmer.

§3. Er et gruppemedlem tildelt en advarsel ved mail, kan gruppen efter yderligere overtrædelser når som helst stemme om, hvorvidt gruppemedlemmet skal ekskluderes.

§4. Ekskludering kræver enstemmighed blandt gruppens øvrige medlemmer.