ETH zürich

Department of Computer Science
Algorithms and Didactics

 [1]

WebTigerPython Classroom

Nicolas Kupper

Master's Thesis

2024

Supervisors: Prof. Dr. Dennis Komm
Alexandra Maximova

[1] Generated with Recraft

Abstract

This thesis presents Classroom, a modern learning platform that meets modern security standards and allows teachers and students to use state of the art technologies to teach and learn. The Center for Computer Science Education of the ETH Zurich (ABZ) developed numerous teaching materials for a wide range of student profiles as well as programming environments, one of which is WebTigerPython. WebTigerPython is a programming environment which supports Turtle Graphics and various hardware devices in order to teach programming. In a previous work, a small frontend-only prototype was built which combines the integration of WebTigerPython with the ability to write numerous types of exercises, such as multiple choice questions using Markdown. In this thesis, the new productive learning platform Classroom was developed from scratch and deployed at https://classroom.ethz.ch. Classroom fully integrates WebTigerPython which avoids the necessity of using external programming platforms when creating or solving coding exercises. This allows for a seamless integration of the learning platform and the programming environment for a better user experience. Classroom extends the rich text editor Quill, allowing teachers to easily create various types of exercises, including coding exercises. By combining multiple frameworks, such as the Conflict-Free Replicated Data Type (CRDT) framework Yjs, the rich text editor used in Classroom has been made collaborative. This allows teachers to easily work on content simultaneously which enhances collaborative team work.

Acknowledgments

First of all, I want to thank my supervisor Alexandra Maximova for the great discussions, inputs, the honest feedback and her guidance during this thesis. The working environment she created was very pleasant and allowed for constructive discussions. I also want to thank Andre Macejko for coordinating tasks where I was dependent on the internal information technology team at ETH Zurich. In this context, I also want to thank Matthias Gabathuler from the information technology team at ETH Zurich who took the time to explain to me the Kubernetes [15] cluster and how to set up the fully automated continuous integration and continuous delivery pipeline. I also want to thank my brother Dario Kupper for proofreading this thesis. Furthermore, I want to thank Dennis Komm for allowing me to write this thesis in his group.

Contents

Chapter 1

Introduction

In this chapter, I outline the importance of this thesis, define the requirements, talk about related work and state what my main contributions are.

1.1 Motivation

In today's digital world we move further and further away from in person teaching. The importance of digital learning environments and remote possibilities has been particularly highlighted during the global COVID-19 pandemic where the vast majority of all people around the world were forced to work and study from home. The fact that programming has become an increasingly important skill in many fields has further increased the need for modern learning environments. However, currently, most learning platforms still lack the support for it and those that support programming are often lacking in other areas such as user management and the ability to create exercises, making them unsuitable as a complete learning platform for teachers. To redress this shortage, the goal of this thesis is to develop a new learning platform, named 'Classroom'.

1.2 Requirements

A modern complete learning platform should fulfill the following requirements:

- The frontend of the learning platform should be kept simple to create an easy to use platform that is readily accessible for students and teachers.
- It should distinguish between students and teachers to take into account the fact that teachers are the ones creating and managing content and students are the ones consuming it.
- Teachers should be able to freely create their own content. This content should include graded exercises, like multiple choice or fill in the gaps exercises, but also more advanced exercises like coding exercises. The creation of this content should not require any special skills but should be intuitively and easily doable for most teachers.
- The learning platform should provide the frame for teachers to meaningfully structure and manage their content.

- Teachers should be able to access insights about the performance of their students, such as averages per chapter, the achieved points per course or performance by exercise.
- The learning platform should have integrated programming support. Teachers should not be dependent on an additional external platform to create their content, nor should they have to use external platforms to grade student submissions.
- The learning platform should support and encourage teachers to work in teams when creating content. This means that the content editor should be collaborative.
- The learning platform should enable teachers all around the world to openly share quality content.
- Teachers should have full control over their content and decide with whom they share it.
- The learning platform should meet modern security standards to prevent unauthorized access.
- The learning platform should be designed in such a way that it is easily extendable by future undergraduate or graduate students. It should enforce modern CI/CD approaches to ensure an easy development workflow.
- The learning platform should be multilingual and should allow for the easy addition of additional languages.

1.3 Related Work

Moodle

Moodle [19] is a open source learning platform initially released in 2002. Moodle is very widely used and constructed for large institutions with many users. While Moodle provides a useful frame for many things a learning platform wants - like user management, permission management, exercises, and grading - it lacks in other aspects. The first of which is the ability to create advanced content. The ability to create content on Moodle is very limited and many teachers practically only use it as structured data storage by linking the PDF files that contain the actual content. The second aspect that makes Moodle a subpar learning platform is its overloaded user interface which decreases the user experience. While Moodle supports a wide variety of exercises, creating more advanced exercises is extremely time intensive. To give a short example on the inflexibility of Moodle: When creating an exercise, teachers have to upload images in the exact size they want them to be displayed, there is no option for resizing which forces teachers to resize their images before uploading them to Moodle. Another shortcoming of Moodle is that teachers rely on external programming environments. Moodle lacks native support for a programming environment. The way Moodle is used to solve coding exercises at ETH Zurich is by embedding an external coding platform like CodeExpert [9] as a full screen

exercise in Moodle. This is very unpractical for students, since they then have to constantly maximize and minimize the programming environment to switch exercises. Moodle fails several of the requirements stated for a modern learning platform as it fails to provide a powerful content creation editor that allows teachers to nicely create content, has a confusing user interface, and does not have native support for a programming environment.

WebTigerJython-Classroom

In 2024, Remo Frei [10] developed a frontend only prototype called 'WebTigerJython-Classroom'. The idea of this prototype was to extend the Markdown language with the help of NuxtContent [20] to support a new custom Markdown-like format which can then be translated to Vue [31] components in a Markdown file. In this prototype, exercises of the type 'single choice', 'multiple choice', 'fill in the blanks', 'sorting', and 'coding' were implemented.

````
::fill-in-the-blanks---
id: b27134a5-d140-435c-b96b-c67b20571b77
blanks:
  - index: 0
    solutions: ["4"]
---
Complete the code so that the turtle draws a square.

#code
```python
from gturtle import *
makeTurtle()

repeat §§0§§:
    forward(100)
  left(90)
  ```
::
````

Listing 1.1. Fill in the gaps coding exercise from WebTigerJython-Classroom

While the idea and concept of custom exercises to declare special parts of your content that are then translated to Vue components for a more powerful editor is promising, this prototype is lacking in many aspects. The first aspect, in my opinion, is the choice of Markdown. Markdown is too weak to be suitable as the content creation base: It lacks many features teachers want, such as basic formatting options like indenting without making a list, coloring or background coloring text, and also does not support more advanced functionalities like embedding videos. Furthermore, WebTigerJython-Classroom forces teachers to know Markdown which is a major limitation when it comes to the target audience. Not all teachers are willing to learn Markdown in order to be able to create content. Even if they know Markdown, Markdown is heavily dependent on the Markdown processor used and has no global standard. The next major shortcoming of WebTigerJython-Classroom is the custom pseudo Markdown which is used to describe the exercises. This forces teachers to write special non-standardized parsers whenever they want to export their content. Exercises should be kept in a standardized format that can be easily parsed, like JSON [13]. Lastly, as mentioned earlier, WebTigerJython-Classroom is a frontend

only prototype. WebTigerJython-Classroom fails several of the requirements stated for a modern learning platform, as it lacks the concept of a teacher and student role, has no user and permission management, no content editor to write new content in, and no possibility for teachers to grade exercises.

Nilsdoc

In 2023, Nils Leuzinger developed a new new Markup-like language called Nilsdoc [17]. The purpose of this new framework language is to give teachers more freedom in creating teaching content than a rich text editor offers. Nilsdoc implemented certain elements of Markdown syntax, like using * for lists and using # for titles, but did not fully adopt it. This means that not all aspects of common Markdown syntax are supported. Furthermore, Nilsdoc addressed many of Markdown's shortcomings like the missing functionalities of coloring, making tab structured content, native support for LaTeX [16] math formulas and drawing graph trees. While I absolutely see the need for teachers to have a more powerful editor than a basic rich text editor, Nilsdoc introduces new problems which, in my view, make it unrealistic that it will be adopted by teachers broadly. The first problem is that Nilsdoc requires teachers to learn a completely new language. While this language partly builds on Markdown, it is not a superset of Markdown, meaning that not even teachers who know Markdown can fully transfer their knowledge to Nilsdoc. The common teacher does not have the time nor the desire to learn a whole new language - especially not a custom language - in order to create their content. While Nils Leuzinger implemented a simple web framework that integrates Nilsdoc in a learning environment that has teacher and student roles, this learning environment is not yet production ready. It does not meet crucial requirements for a learning environment as described in this thesis, such as the integration of a programming environment, the support of the concept of exercises, a concept for grading student answers, and the possibility to work collaboratively on content.

1.4 Main Contribution

The following section deals with the main contributions of this thesis and the components it builds on. Classroom, the learning environment developed for this thesis, meets the requirements for a modern learning environment outlined above by extending Quill [25] with a custom exercise format and making it collaborative. Thereby, the different type of exercises that were developed for WebTigerJython-Classroom have been integrated into an accessible platform that is, however, not limited to pseudo Markdown and can be easily parsed. Lastly, Classroom fully integrates the programming environment WebTigerPython [2] but extends it with the useful feature of having code history.

Information Technology Architecture

An important part of building a new platform is the process of system design decisions. This includes the base decision of which frameworks are used to build the platform, the definition of models corresponding to a database schema, but also more granular

decisions, such as how content is stored. A big part of this thesis was therefore the decision making and design of this newly created learning platform.

Content Creation Editor

As part of this thesis, a rich text editor had to be evaluated which is used as the editor for teachers to create content. This rich text editor was extended with a custom exercise format to allow teachers to create graded exercises. Furthermore, to make the creation of these exercises intuitive and doable by most teachers, a user interface which is integrated into the rich text editor was developed that gives teachers without a technical background the possibility to easily create exercises. To meet our requirements of collaborative editing, a further contribution of this thesis is to make this content editor collaborative with the help of the CRDT framework Yjs [35].

Software Engineering

While system design decisions are crucial for a platform to fulfill its requirements, the goal of this thesis was not to hypothetically talk about how a modern complete learning environment could look like but to actually build one. Hence, another crucial part of this thesis was to implement a complete learning environment with all the needed components to meet our requirements. In doing so, this thesis builds on the following previous work:

- **The exercises**: The goal of this thesis was not to create new exercise types but rather to create an environment that is easily extendable with any kind of exercises. To have a starting foundation, the type of exercises currently supported are mostly based on the previous work on WebTigerJython-Classroom. This does not include the Markdown approach on how to store these exercises but rather the Vue components used to define the frontend functionalities of the exercises.
- **The programming environment**: Developing a programming environment was not part of this thesis. Rather, the goal was to use a well-established and powerful existing programming environment. Therefore, Classroom fully integrates the existing programming environment WebTigerPython as an iframe. Thereby, Classroom also supports more advanced features of WebTigerPython, like using hardware devices. Furthermore, it adds the new feature of keeping track of the code history from users using WebTigerPython in Classroom.

Chapter 2

System Architecture

In this chapter I describe the system design decisions made when building Classroom and describe how the core components of Classroom are working together.

2.1 Overview

To meet the requirements outlined in Section 1.2 several components had to be implemented and connected. The Keycloak [14] service is used as a central user management service, allowing multiple platforms to use the same user base. The Redis [27] service acts as a message broker to enable communication between multiple teachers when collaborating on content. Postgres [22] is used as the database to store all Classroom related data, while the PostgreSQL cluster acts as a mirrored database cluster enabling failover in case of a single database instance failure. The Django [4] backend communicates with all these services and exposes an API endpoint to fetch data from, as well as a websocket endpoint to connect teachers in a digital room to work on the same chapter. The Vue frontend provides an intuitive user interface for Classroom and communicates with the backend whenever it needs data. In the following, the components of Classroom depicted in Figure 2.1 are explained in more detail.

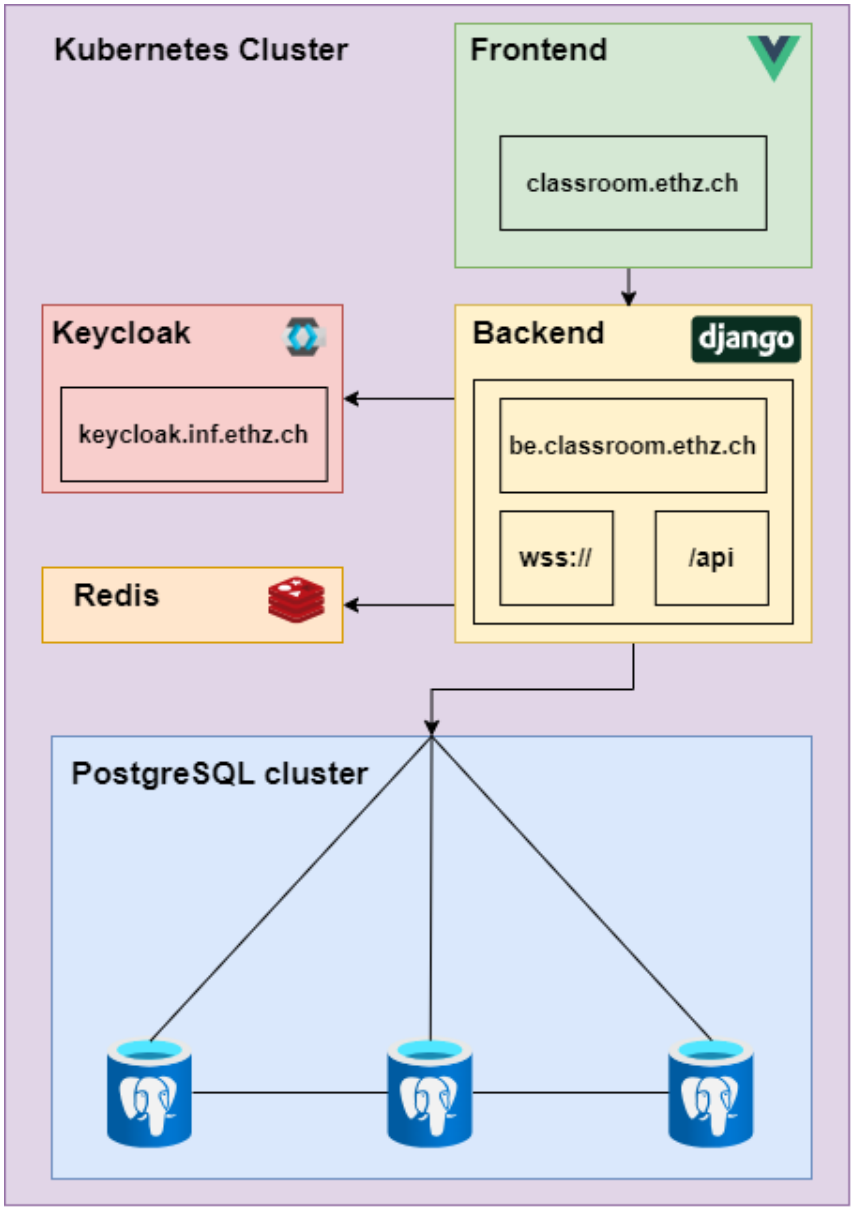


Figure 2.1. High level overview of the system architecture

Backend

When choosing a backend framework, it is important to choose one that fits the requirements of the project. In this thesis, I needed a framework which allowed me to develop quickly while meeting modern security standards. Furthermore, I combined multiple components like the Keycloak server, Postgres databases and a Redis broker and therefore needed a framework that supports a wide range of technologies. My decision for the backend framework Django was motivated by the fact that Django is one of the most popular open source web frameworks today. It has existed for almost twenty years and has an active community by which it is maintained. It also supports all the technologies needed to build Classroom. Furthermore, it builds on top of Python [24] which currently is one of the most popular programming

languages, and is known for its simplicity and for allowing developers to quickly write code. As stated in our requirements, Classroom will continue to be developed by future undergraduate and graduate students. It was therefore important to have a framework that can be quickly learned as the time frame for student projects is always very limited. Classroom's complete backend consists of the following elements:

- A Django [4] application exposing two endpoints:
  - https://be.classroom.ethz.ch/api/ - The REST API used for Classroom API calls.
  - wss://be.classroom.ethz.ch/ - The Django Channels [5] consumer endpoint used for collaborative editing Section 3.1.
- A Postgres [22] cluster with three Postgres database instances used for persisting data. Django supports a variety of databases and the decision for Postgres was motivated by the rigorous constraint checking Postgres enforces, as well as the support of the PostgresSQL cluster by the ETH Zurich IT.
- A Redis [27] instance used as the message broker for Django Channels. Django Channels uses Redis as the backing store and the choice of using Redis was therefore inherited from the choice of using Django Channels.

Frontend

Just like the backend framework, the frontend framework is another major decision when building a platform. However, the choice of Vue was due to ABZ lab's decision to define a technology stack over multiple projects. This makes a lot of sense for maintainability, as it enables employees of the ABZ lab who are familiar with Vue to mentor multiple projects. Vue, together with React [26] and Angular [1], leads the list of the most popular frontend frameworks in the world. The frontend therefore consists of a Vue application which makes use of the following components:

- Vuetify [32] as the UI library. The goal of choosing Vuetify was to enable quicker development by using pre-built UI components rather than styling all components myself.
- Pinia [21] as the data store. Pinia is the standard data store of Vue.
- Quill as the rich text editor for content creation. The choice of Quill was motivated by several factors. Firstly, Quill is an API-first open source editor which is known to be easily extendable. This was very important for my requirement to incorporate custom exercises into the rich text editor. Secondly, Quill is extremely popular, with more than 1.1 million weekly downloads. The popularity of an open source project is always crucial, as it means that there are more contributors who develop new features for the project. Lastly, Quill is supported by the state of the art open source CRDT framework Yjs which allowed me to fulfill my requirement for collaborative editing, as explained in Section 3.1.

Keycloak

As the ABZ lab has multiple applications needing user management, it was decided during this thesis that a central Keycloak [14] instance would be set up as a user management tool. The Keycloak instance was set up by the ETH Zurich IT. The ETH Zurich IT already had several other Keycloak instances running for other labs and it was therefore logical to use Keycloak. Keycloak is one of the most popular open source identity and access management solutions in the world. The Keycloak instance was then configured by me for Classroom to meet the project's needs. In particular, the configurations needed for Classroom are:

- Two custom user attributes 'isClassroomTeacher' and 'language'. The first one is used to determine whether a user is a teacher or a student and the second one is used to determine the users interface language.
- An OpenID Connect client that allows users to connect from Classroom to Keycloak.
- A user attribute mapper mapping the custom Classroom attributes 'isClassroomTeacher' and 'language' into the access token issued by the OpenID Connect client.
- A user named 'classroom-admin' used to perform privileged actions like creating and updating users and triggering verification and reset password emails.

2.2 Managing Content

A major decision when developing a learning platform is how to manage content. This includes creating, storing and loading content.

The Life-Cycle of a Chapter

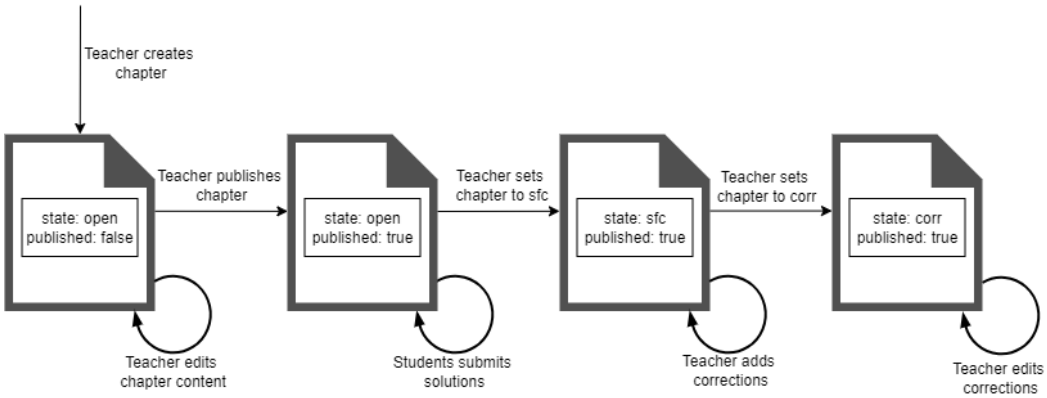


Figure 2.2. The life-cycle of a chapter

During the life-cycle of a chapter, a chapter runs through the following phases:

- A chapter starts as open and unpublished. Here, teachers can create the chapter content. The students cannot see the chapter.

- After the teacher publishes the chapter, they can no longer edit the chapter content. Students can now view the chapter and add submissions for all exercises present.
- When the teacher sets the chapter to 'set for correction' (sfc), students can no longer make submissions for their exercises. However, they can still view their answers. At this point, all exercises which can be automatically corrected are corrected. For the coding exercises, the teacher can provide a sample solution and add corrections for each submission made by a student. Furthermore, teachers can overwrite existing solutions, for example, if they marked the wrong multiple choice answer.
- After the teacher sets the chapter to 'corrected' (corr), students can view their achieved points and the correct answers. The teacher still has the possibility to overwrite corrections made and existing solutions.

The 'state' attribute on a chapter is a permission attribute which is used to decide which actions are allowed on a chapter while the 'published' attribute was originally intended to be a pure visibility attribute, in the sense that when a chapter is published, students can see it. However, in the current state of Classroom, it was decided to not allow teachers to edit chapter content once a chapter has been published. This was a conscious decision in favor of students to create a fair environment where each student always has access to the same information. If one allows teachers to edit chapter content post-publishing, it might happen that students get additional information for exercises they have already solved when this information was not present. While this restriction increases fairness, it of course constrains teachers, as it means that when publishing a chapter, they have to be certain that all the information needed is present. Also, it does not account for typos made, as they cannot be fixed in a published chapter.

The Chapter Editor

In many learning platforms, when writing content, teachers are forced to use clunky user interfaces. For example, this can mean that when creating a multiple choice exercise, teachers have to add each option in a user interface and then press a button to add the option. Then, when switching an exercise type, they have to create a new exercise, copy each answer manually, and then insert them. In Classroom, I wanted to give teachers the full-on experience of a rich text editor while still keeping practicality. Each chapter corresponds to exactly one Quill document. In this document, teachers can use all the built-in features of Quill, like writing and formatting text, inserting images, embedding iframes, and writing latex math formulas. However, Quill does not support embedded Vue components which are needed for custom exercises. Therefore, I extended Quill with a new custom format exercise. For Quill, the exercise format is just text (JSON) which then, at export time to HTML, gets tagged as 'exercise' and has the JSON content as data value. This allows for dynamic Vue components which then map each JSON exercise to the corresponding Vue component. To still support teachers' need for user interfaces - which, despite their tendency for being overloaded with superfluous elements, make sense for certain exercise types like coding exercises - I extended the Quill toolbar with a custom exercise creator dialog. When creating

an exercise in this dialog, the exercise gets inserted as JSON text in the Quill editor. Double-clicking an exercise in the editor brings it back up in the user interface. This provides teachers with the best of both worlds, as they can make changes to exercises directly in the editor by simply changing the text but can also use the editor for more advanced tasks like writing code. Furthermore, copying or moving an exercise is now as easy as just copying the JSON text and inserting it.

```
{
  "type": "sc",
  "points": "1",
  "task": "Choose the mountain in
    Switzerland",
  "answers": [
    "Pilatus",
    "Mount Everest",
    "Himalaya"
  ],
  "correctAnswers": [
    "Pilatus"
  ]
}
```

Listing 2.1. Single choice exercise example

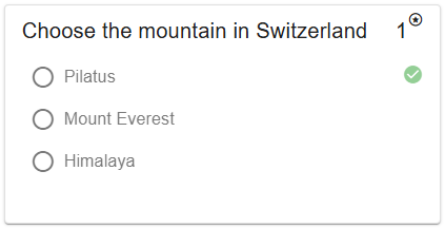


Figure 2.3. Single choice exercise component

Storing Content

As described in Section 2.2, the content of a chapter can consist of several exercises combined with other built-in Quill content. When teachers write a chapter's content, students cannot submit answers. Therefore, it is sufficient to simply store the whole content of a chapter in the database and treat exercises as text. This way, it is not necessary to keep track of exercises in the content and sync them with exercises in the database. As we use Yjs to support collaborative editing, as described in Section 3.1, it makes sense to store the content as a YDoc binary blob.

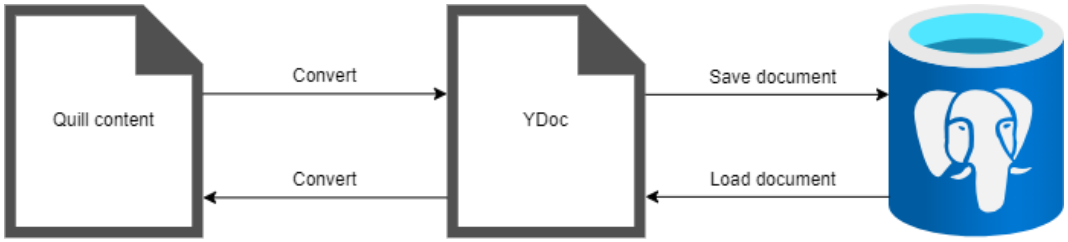


Figure 2.4. Saving and loading a chapter

This method of storing content posed a challenge, however, since, as soon as students need access to exercises, storing them in their plain JSON form is insufficient. Not only would it mean that when loading the document, students could potentially see the correct answers, as they are part of the JSON, but it would also mean that we could not link student answers to exercises. Therefore, when a chapter gets published, for each exercise in the content, an exercise is created in the database and the JSON

content of the exercise in the content is replaced with the id of the database exercise. This way, the content visible for students now contains no sensitive data and we can use our permission classes to control access to exercise solutions, as mentioned in Section 4.2. Often, teachers want to copy old exercises from previously published chapters. To maintain this functionality, when a teacher opens a published chapter, Classroom converts the exercises in the content back to the original JSON format. This way, the teacher always has access to all exercises and they remain the same as they were initially written by the teacher. This leads to an improved user experience and has no effects on the way content is stored in the database.

```
{
  "id": "Exercise UUID"
}
```

Listing 2.2. Exercise in the content after publishing

2.3 Models

In Django, instead of directly defining an SQL schema, one defines models. Out of these models, Django then generates migration files which can then be used to initialize and update the database. This way Django, supports multiple databases that can easily be switched out. In this section, the most important models created for Classroom are described. The following attributes do not constitute an exhaustive list of all attributes available on the listed models but, rather, were chosen as examples to illustrate how Classroom works. For a full list of all attributes and models, please refer to the database schema, shown in Figure 2.6.

User

Django's base user model was extended with the boolean attribute 'isTeacher' which determines whether a user is a teacher or a student. These roles were introduced to limit viewing privileges to the respective target audience. For example, a student has no need of being able to create courses or chapters, they only need to able to solve them. Removing unused components from the views of students makes the pages clearer and easier to use which helps achieving our requirement of making a simple to understand frontend. A teacher is treated as a superset of a student. This way, a teacher can still act as a student in courses which was implemented in this way to address the fact that users who act as teachers in some courses may be still be students in other courses. Furthermore, it allows teachers to get the exact view students get which will help them assist students in case of misunderstandings.

ClassGroup

A class group is used to group a list of users. This model was introduced to allow teachers to reuse the same user base on multiple courses. This prevents a teacher from needing to add each user by hand in every course they teach. Furthermore, Classroom allows teachers to paste comma separated lists of email addresses when adding students to courses or class groups, making it easy to import existing user

bases. Each class group is owned by a teacher and the teacher has full control over the members of the class group.

Course

A course is the top level entity of Classroom's data hierarchy. Each course consists of a list of chapters, with a defined order, and is owned by a teacher. A teacher in a course can add other teachers to the course as collaborators. Collaborating teachers have the same permissions as the teacher who owns the course, except that they cannot delete the course. Each course has a list of all its members which includes direct students as well as students in class groups. A member of a class group in a course shares the same permissions as direct students of the course, meaning that both have access to the course. A course can be public, meaning that any student can access the course without being listed as a member of the course. This is also the case for users who are not logged in. I added this feature to meet the requirement to allow teachers all around the world to openly share their content. This concept of a public course was introduced as one of the declared goals of Classroom is to let teachers share great courses with a broader audience. Furthermore, public courses can also act as new inputs for other teachers on how to improve their own courses. The way public courses are implemented in Classroom is that every teacher has the ability to make their courses public, resulting in a fair learning environment where all teachers are equal.

Chapter

A chapter is a sub entity of a course. The content of a chapter is stored in a binary field and can include exercises, with a defined order, as described in Section 2.2. Each chapter has two boolean attributes: 'webTPEnabled' and 'webTPUiEnabled'. These attributes are used to set the visibility of WebTigerPython when students solve a chapter and to decide whether the canvas field of WebTigerPython should be displayed.

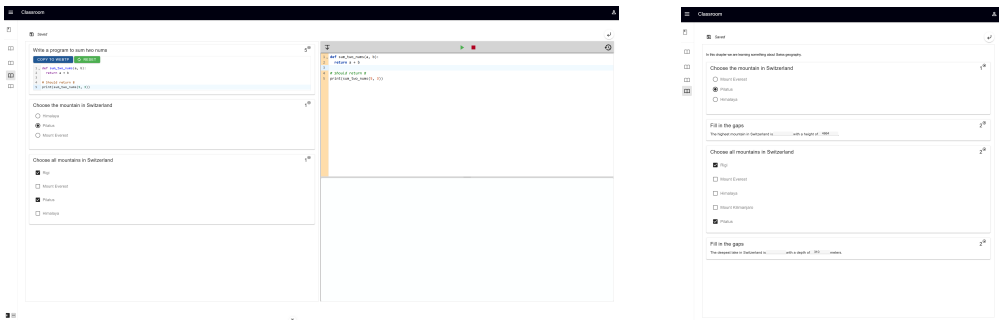


Figure 2.5. Classroom's chapter solve page with and without WebTigerPython

Exercise

An exercise is a sub entity of a chapter and belongs to exactly one chapter. Each exercise has the attribute points which correspond to the maximum points this exercise can give. Furthermore, each exercise has a type and additional attributes

based on that type. For example, a coding exercise has the attribute ‘codeSkeleton’ which defines the code skeleton a student gets when opening this exercise the first time.

CourseCode

Course codes are codes that can be generated on a course level. With a course code, a student can join a course as a direct student without being added by a teacher. The purpose of course codes is that instead of teachers needing to add all students to the course they can simply share a course code and then students can join themselves. A further benefit of course codes is that teachers do not have to wait for students to create a Classroom account to add them to a course, since they can simply share the corresponding course code with their class and students can individually join with the code once they have created their account. This is also handy in the case a student is absent or sick, as the student can do it later without any action required by the teacher. Each course code has an expiration date, only valid course codes can be used to join a course. With course codes, students have multiple ways of joining or being added to a course which gives teachers a lot of flexibility on deciding how to manage their students.

Database Schema

With the help of Django Extensions [6], one can generate a database schema graph based on the defined models. This schema shows all relations and attributes used by Classroom as depicted in Figure 2.6.

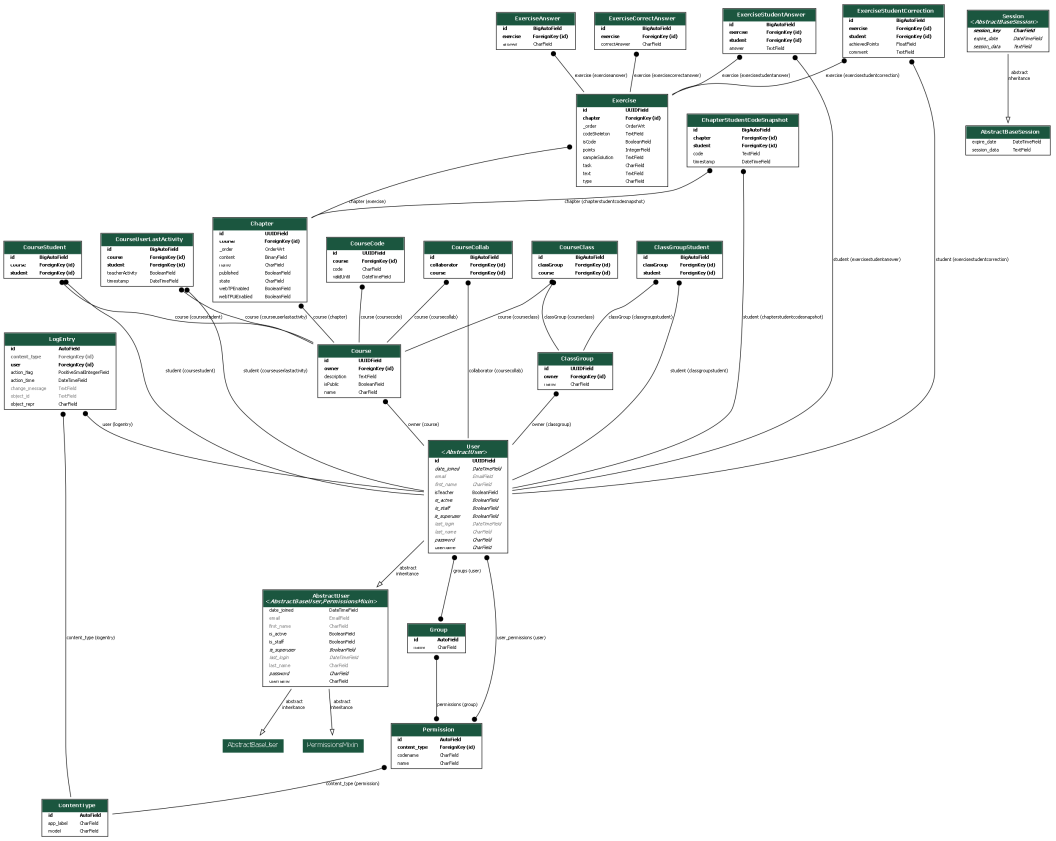


Figure 2.6. Full database schema describing Classroom's models

Chapter 3

Special Features

While Classroom implements a multitude of features, the following sections highlight those that had a significant positive impact on user experience.

3.1 Collaborative Editing

The need of enabling multiple users to work on the same document at the same time has been a hot topic for a long time. At the latest with the establishment of Google Docs and Microsoft365 as popular collaboration platforms, people have accustomed themselves to having this feature present. Therefore, I decided to implement this feature in Classroom, in order to allow multiple teachers to work on the same chapter simultaneously. However, having real time collaborative editing is a highly complex problem for which many things have to be taken into account. Currently there are two approaches to achieve collaborative editing, Operational Transformation (OT) and Conflict-Free Replicated Data Type (CRDT).

Operational Transformation

OT was pioneered by C. Ellis and S. Gibbs [8] in the GROVE (GRoup Outline Viewing Edit) system in 1989. There has been extensive research on this topic and Google Docs, as one of the leading online word processors, uses OT [12] for its collaborative editing. However, the source code is not open source and it is therefore not visible how they implemented their OT algorithm. The idea of OT is that when one has an operation b that has happened at some point in time and is received after an operation a has already been applied, then one has to transform the operation b with a transform function in such a way that it behaves in the way it was intended to. Quill Delta, the underlying structure Quill uses to store its content, has a transform method which applies such an OT. Initially, I tried to implement collaborative editing using Quill Delta's OT approach. Due to the issues encountered while working with Quill Delta, this approach was abandoned. The following example demonstrates the support Quill provides for OT. However, also why this alone is not enough to achieve proper collaborative editing.

```
const a = new Delta().insert('a');
const b = new Delta().insert('b').retain(1).insert('c');

a.transform(b, true);  // new Delta().retain(1).insert('b').retain(1).
    insert('c');
```

Listing 3.1. An example of Quill Delta OT

In the above example, if these deltas were applied without transforming b one would get following output:

bac

This, of course, is not the intended output. After the transformation, one gets the intended output of:

ab c

One might assume that having the OT function solves all problems for collaborative editing which is, however, not true. Firstly, it is entirely possible to have conflicting operations where a simple transformation is not enough to solve them. Secondly, Quill does not keep track of states and simply sends delta operations. This means that in case of delayed or mixed communication, for example due to a network outage or multiple users typing at the same time, it is not possible to simply recreate the correct state. Another problem Quill faces is that the change event one can catch to send one's changes to the websocket happens after the operation has already been applied to the document. This means that several users can have different local documents at any given time and Quill provides no solution for synchronizing them. For the above reasons, it was decided to not further pursue this approach.

Conflict-Free Replicated Data Type

CRDT is the more modern approach and was pioneered in 2011 [29]. The idea of the concept is to have a data structure which can be replicated across multiple computers in a network and has the following properties:

1. Replicas can be updated independently, concurrently, and without coordination with other replicas.

2. Inconsistencies between replicas are automatically resolved by an algorithm which is part of the data type.

3. Replicas are guaranteed to eventually converge, meaning that although they might have different states at some point in time, they are eventually synchronized to the same state.

The development of this concept was initially motivated by collaborative text editing and mobile computing but has since found use in other areas. One of the state of the art open source frameworks implementing such a data structure is Yjs. It has found use in many applications and has bindings for several rich-text editors, such as ProseMirror [23], TipTap [30], Monaco [18], Quill, CodeMirror [3], and Remirror [28]. A 'binding' describes the conversion of the content of an editor to the CRDT.

Implementation

I integrated Yjs into Classroom making use of the packages y-quill [33] for the binding, y-websocket [34] as the provider for the connection to our backend websocket and ypy-websocket [36] for the task of synchronizing different YDocs. This was implemented in Classroom by assigning teachers working on the same chapter to the same room where they share one YDoc per room. Additionally, each teacher keeps a local YDoc replica which is always synced with the YDoc in the room if the teacher is connected to the websocket. The local replica can become out-of-sync when editing while being offline. On connection, when no YDoc is present in the room, the YDoc from the room gets initialized with data from the database as shown in Listing 3.2.

```
async def make_ydoc(self) -> Y.YDoc:
    y_doc = Y.YDoc()
    # Loads content from the database
    content = await self.get_chapter_content()

    if content:
        Y.apply_update(y_doc, content)

    return y_doc
```

Listing 3.2. Code that initializes room YDoc with database data on user connection

Whenever a change is received on the websocket, a sync operation is initialized which sends the sync event to all active users as shown in Listing 3.3.

```
async def receive(self, text_data=None, bytes_data=None):
    if bytes_data is None:
        return
    await self.group_send_message(bytes_data)
    if bytes_data[0] != YMessageType.SYNC:
        return
    await process_sync_message(bytes_data[1:], self.ydoc, self.
    _websocket_shim, logger)
```

Listing 3.3. Code that sends synchronization event on receiving a change

It is not enough to initialize the local replicas in the frontend with the chapter content on initial page load. To understand why this is the case and why it is therefore necessary to load the content in the backend as well when no YDoc is present in the room, is shown with the following example:

Step	Action	YDoc states	Active users in the room
1	User1 opens the chapter in the frontend	YUser1: state db1, Yroom: None	-
2	User1 gets connected to the websocket	YUser1: state db1, Yroom: state db1	User1
3	User2 opens the chapter in the frontend and gets connected to the websocket	YUser1: state db1, YUser2: state db1, Yroom: state db1	User1, User2
4	User1 loses internet connection and keeps on editing locally	YUser1: state user1, YUser2: state db1, Yroom: state db1	User2
5	User2 makes changes to the document	YUser1: state user1, YUser2: state user2, Yroom: state db1	User 2
6	Changes are synced	YUser1: state user1, YUser2: state user2, Yroom: state user2	User 2
7	Changes are saved to the database	YUser1: state user1, YUser2: state db2, Yroom: state db2	User 2
8	User2 disconnects	YUser1: state user1, Yroom: state db2	-
9	No active users - room gets destroyed	YUser1: state user1, Yroom: None	-
10	User1 reconnects	YUser1: state user1, Yroom: state db2	User1
11	Changes are synced	YUser1: sync state db2user1, Yroom: sync state db2user1	User1
12	Changes are saved to the database	YUser1: state db3, Yroom: state db3	User1

Table 3.1. Exemplary case of collaborative editing in Classroom

Step 10 is the step where content for the newly created room YDoc is loaded from the database in the backend. Not loading the data from the database as described in Listing 3.2 would have the following consequences:

1. The YDoc in the room would be initialized with the state of the YDoc replica from user1.
2. The sync command would do nothing, since the local replica of user1 and the YDoc in the room are in sync.
3. The state of the YDoc of user1 would be saved to the database.
4. All changes made by user2 would be lost.

When reading this carefully, one might ask why the content is even set in the frontend at all and whether it would not be enough to just set it in the backend. From a functional standpoint it would absolutely be enough to just set it in the backend. The reason the content is also set in the frontend on initial load of the chapter editor component is to improve user experience. Before entering the route the chapter data is fetched from the backend. Besides obtaining the chapter data, this also acts as a check whether a user has permission to access the chapter, as described in Section 4.3. Only when a user enters the chapter editor component they establish a websocket connection. This means that if the content were only set in the backend, the user would be left with either an empty document or a second loading animation would have to be implemented, even though the data needed from the backend has already been fetched. In reality, the establishment of this websocket connection and the loading of the data from the backend takes a few seconds at most but there is no downside in just setting the data in the frontend as CRDT is designed to handle multiple replicas which then get synced. This leads to an improvement in user experience.

Quill Sources

While allowing multiple people to work on the same document simultaneously brings added value to the user experience, implementing it raises new questions: What happens with the history of a document when another user makes a change? Does the fact that Classroom has an autosave feature, as described in Section 3.3, mean that a change from any user in collaborative mode triggers each active user to resave the changes made to the document? Quill has an elegant solution to these questions, namely a source attribute which is sent whenever a change is made. Valid sources which can be set are:

- 'user': A change was made by the user.
- 'api': A change was made from other sources.
- 'silent': The change event is not emitted.

This can be utilized to achieve the desired functionality to only save on the instance where a user made a change, as shown in Listing 3.4:

```
quill.on('text-change', (delta, oldDelta, source) => {
if (source !== 'user') return;

chapterEditor.setChapterUnsaved();
})
```

Listing 3.4. Implementation of Quill only triggering the save event for changes made from the user

Furthermore, to only keep track of each user's own history, Quill has a predefined flag that can be set as shown in Listing 3.5:

```
history: {
    userOnly: true
},
```

Listing 3.5. Flag for Quill only keeping track of each user's individual history

In conclusion, this implementation with Yjs resulted in Classroom having a fully collaborative chapter editor, allowing any numbers of teachers to collaborate in real time. Furthermore, it allows offline editing which is another added benefit, as it allows working in Classroom without a stable internet connection, such as during the commute to work.

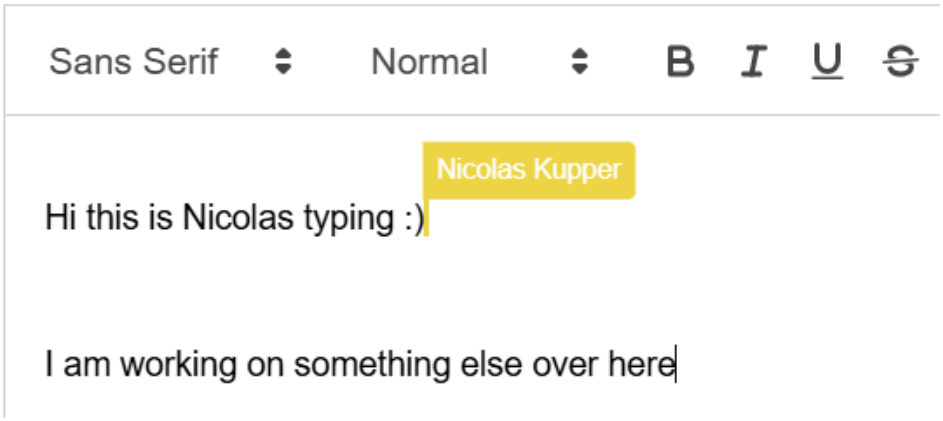


Figure 3.1. Classroom's collaborative chapter editor displaying another active teacher

3.2 Code History

When solving coding exercises, it may happen that students want to revert changes they have made when trying out different approaches to solving the task. Similarly, teachers may want to see the intermediate results a student had when a submission does not correspond to the exact solution, in order to gauge how well the student has understood the assignment, and consider this in the grading. Lastly, when doing research, it can be productive to analyze a student's path in reaching a solution. For these reasons, I decided to implement a code history in Classroom, as it leads to added user benefits both for students and teachers. However, WebTigerPython uses CodeMirror [3] to implement its editor which does not provide a code history. The only related functionality is letting users undo and redo. However, the local history CodeMirror uses is lost when refreshing the page. Also, this implementation only allows users to sequentially go back in time but not to jump to certain points in the past directly. Since WebTigerPython has no user management, the only way it would be able to keep track of the code history is session-based, meaning that whenever starting a new session, one does not have access to one's past code. Classroom addresses this issue by implementing a fully functional backend with user management which allows the creation of a chapter-based code history.

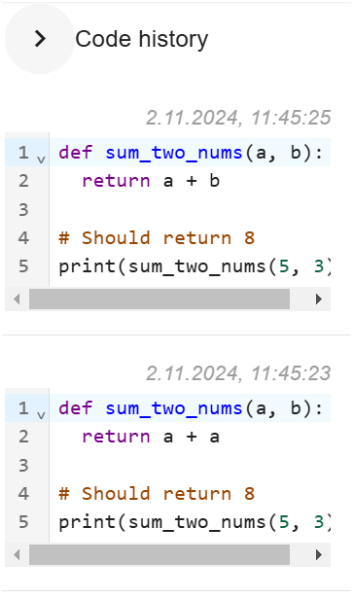


Figure 3.2. A code history in Classroom with two code snapshots

Code Snapshot Model

As described in Section 2.3, a chapter can consist of several exercises. When a student solves a chapter, they run code in the integrated WebTigerPython iframe. It is therefore natural to create the following model for a code snapshot:

```
class ChapterStudentCodeSnapshot(models.Model):
    chapter = models.ForeignKey(Chapter, on_delete=models.CASCADE)
    student = models.ForeignKey(User, on_delete=models.CASCADE)
    code = models.TextField()
    timestamp = models.DateTimeField()
```

Listing 3.6. Code snapshot model

Each code snapshot is assigned to a student as well as a chapter and consists of the code and a timestamp. Therefore, the timestamp not only gives an order of the snapshots but serves as an indicator for teachers for the time a student spent on a given chapter. This can be interesting when analyzing a student's path to a solution. The following conditions have to be met to trigger saving a new code snapshot:

- The user is logged in.
- The chapter is open.
- The user executes code in WebTigerPython.
- The latest code snapshot is different from the code being executed.

The reason why Classroom stops keeping track of the code history once a chapter is no longer open is that the code history also acts as a means for teachers to grade students' exercises. Moreover, while being able to follow a student's path to their solution is desirable from a research perspective, it does not make sense to keep track of code snapshots after an exercise has been corrected because at this point the student has access to the sample solution which means that their inputs no longer necessarily reflect their own thought processes.

Code History Access

Classroom gives users access to the code history in the following cases:

- A student has access to their code history when solving a chapter and can always jump back in time to access past snapshots.
- A teacher has access to each student's code history when correcting coding exercises and can copy each code snapshot directly to WebTigerPython by clicking on it.

Code History Export

To use the data collected from code histories, Classroom offers teachers the possibility to export code histories. Teachers can export code histories either by chapter or for the whole course. The export is in JSON format and is structured in the following way:

```
{
    "courseId": "Course UUID",
    "name": "Sample course",
    "chapters": [
        {
            "chapterId": "Chapter UUID",
            "name": "Chapter 1",
            "exercises": [
                {
                    "id": "Exercise UUID",
                    "points": 5,
                    "task": "Write a program to sum two nums",
                    "codeSkeleton": "def sum_two_nums(a, b):"
                },
            ],
            "codeHistories": [
                {
                    "studentId": "Student UUID",
                    "codeSnapshots": [
                        {
                            "code": "def sum_two_nums(a, b):",
                            "timestamp": 1728314127.332
                        },
                    ],
                    "results": [
                        {
                            "exercise": "Exercise UUID",
                            "achievedPoints": 5
                        }
                    ]
                }
          ]
        },
    ]
}
```

Listing 3.7. Code history export structure

In a real use case, all array elements shown above can contain multiple entries, which is here simplified for reasons of space. With Classroom's code history export, a

teacher has the ability to export all chapters associated with a course. For each chapter they then get all the coding exercises and the code histories of all students who have entered submissions as well as the points they have achieved. The export contains no student names but students can be identified using the UUID. This means that this data is only semi-anonymized, since students can be tracked over multiple chapters and even courses. However, this is intended for research purposes to analyze a student's performance and improvement over the span of multiple courses.

3.3 Autosave

To save users the trouble of having to manually save their work, Classroom implements an autosave feature on both the chapter solve and the chapter editor pages which sends changes to the backend every two seconds. These changes are only sent under certain conditions to avoid unnecessary API calls to the backend. The following conditions have to be met to trigger a save event:

- **Chapter solve**: Exercise answers have been changed or new code history snapshots are available.
- **Chapter editor**:
  - Changes have been made to the chapter content.
  - An active websocket connection exists.
  - The sync status is synced, see Section 3.1 for more information.

Chapter 4

Security

Security is a crucial concern for a web application, even more so when it comes to applications like Classroom which are publicly available and can be accessed by anyone. To fulfill the stated requirement of having a secure platform, for both the backend and the frontend, measures have been taken to secure them which are outlined in this chapter.

4.1 HTTPS on Deployed Instances

Both the deployed instances 'classroom' and 'classroom-test', see Section 5.1 for more details, are HTTPS-only, using valid certificates from ETH Zurich. Therefore, all communication between the user and Classroom is encrypted.

4.2 Backend

The backend application manages all the content of Classroom. It therefore contains sensitive data like user information and chapter contents which have to be secured. The backend consists of multiple components, as described in Section 2.1, each of which posed its own challenges in terms of security. For the Django application, this meant restricting the origins allowed to make requests, restricting the HTTP methods allowed on its views as well as enforcing permission checks when accessing a view and securing the websocket implementation. Furthermore, secrets used in the Django application, such as the database password, had to be encapsulated. For the user management using the Keycloak server, the origins allowed to make requests to the server were also restricted and Keycloak was set to be the only source of truth for token verification. The following sections outline these implementations in more detail.

Permission Classes and HTTP Method Restrictions

As described in Section 2.2, a student is not allowed to view the content of a chapter before it is published. If a student could access the chapter editor with the content of a chapter when knowing the chapter id, it would mean that this student would be able to access all chapter solutions and could cheat on the respective exercises. Not having permissions classes would also allow teachers to access content created

by other teachers without their permission. Each teacher should be in full control of permitting and denying access to whomever they choose when it comes to their content. In Classroom, these requirements were met by using permission classes from Django Rest Framework [7] which can be applied to views. This means that before the view is entered, the permission class is checked and if the permission check fails, the view returns either a '403 Forbidden' or a '401 Unauthorized' response. A '401 Unauthorized' response is returned when the user is not properly authenticated and a '403 Forbidden' response is returned when the user is properly authenticated but does not pass the permission check. Django Rest Framework also supports restricting views by HTTP method. This means that when a view is called with an HTTP method not present in the array of defined HTTP methods, the view returns a '405 Method Not Allowed' response and neither checks the permission class, nor enters the view.

Overview of the Classroom Permission Classes

For Classroom several, custom permission classes have been created and the built-in permission class 'IsAuthenticated' which checks if a user is authenticated has been used.

Name	Description	Inherits from
TeacherPermission	User has the attribute 'isTeacher' set to True	IsAuthenticated
TeacherCoursePermission	User is a teacher and is either the owner or a collaborator of the course	TeacherPermission
TeacherCourseOwnerPermission	User is a teacher and is the owner of the course	TeacherPermission
TeacherClassOwnerPermission	User is a teacher and is the owner of the class	TeacherPermission
TeacherClassCourseCollaboratorPermission	User is a teacher and the class is present in a course the user is a collaborator in	TeacherPermission
StudentChapterPermission	Student is a member of the course, as described in Section 2.3	IsAuthenticated
ChapterPresent	Chapter is provided and exists	-
ChapterPublished	Chapter is 'published'	ChapterPresent
ChapterOpen	Chapter is 'open'	ChapterPresent
ChapterSFCorCorrected	Chapter is 'set for correction' or 'corrected'	ChapterPresent
ChapterBelongsToPublicCourse	Chapter belongs to a course which is 'public'	ChapterPresent
ChapterPublishedAndOpen	Chapter is 'published' and 'open'	ChapterPublished, ChapterOpen

Table 4.1. Overview of Classroom's permission classes

Sample Usage Permission Classes and HTTP Method Restrictions

Both permission classes and HTTP method restrictions can be enforced on a view using annotations. Permission classes can be logically combined when used on views, meaning that one can, for example, use logical $\vee$ and $\wedge$, allowing the combination of multiple permission classes, as shown in Listing 4.1.

```
@api_view(['GET'])
@permission_classes([(StudentChapterPermission |
    ChapterBelongsToPublicCourse) & ChapterPublished])
def get_chapter_content_student(request, chapter_id):
    chapter = Chapter.objects.get(pk=chapter_id)

    serializer = ChapterContentStudentSerializer(chapter, context={'
    request': request}, many=False)

    return Response(serializer.data)
```

Listing 4.1. Permission classes and HTTP method restriction example

In the above example, the HTTP methods allowed is restricted to the 'GET' method and the permission check which has to be fulfilled is: (StudentChapterPermission $\vee$ ChapterBelongsToPublicCourse) $\wedge$ ChapterPublished. This means that in order to get access, a student either has to be part of a course or the course is to 'public', and the chapter is set to 'published'.

Cross-Origin Resource Sharing Restrictions

For security reasons, the allowed origins for Classroom as well as the Keycloak server had to be restricted. Each request made contains HTTP headers which include the origin of the request. The same-origin policy enforces that requests made to a resource can only come from the same origin as the resource. While only allowing the same origin is not feasible for a web backend, which usually gets requests from other origins, it is still necessary to restrict the origins. I therefore enforced the following rules:

Name	Allowed origins
Django backend	https://classroom.ethz.ch, https://be.classroom.ethz.ch, https://test.classroom.ethz.ch, https://test.be.classroom.ethz.ch
Keycloak abz-prod	https://be.classroom.ethz.ch
Keycloak abz-test	*

Table 4.2. Classroom's and Keycloak instances allowed origins

The same-origin policy is extended for the backend to also receive requests from the frontend to maintain the functionality of Classroom. Similarly, the same-origin policy of the Keycloak production instance is extended to also receive requests from the backend in order to allow user authentication. The Keycloak test instance allows all

origins, as this instance is also used when developing locally, meaning that requests can come from any origin.

Websocket Hardening

The websocket connection manages who is allowed to join the rooms in which teachers share the content of the chapters they are currently working on. Therefore, it is also necessary that the websocket connection is secured. The websocket is secured in two ways: Firstly, the CORS header checks are enforced equivalently to the ones enforced when accessing the API, as described in Section 4.2, and, secondly, a user authentication is enforced. For user authentication purposes, a small authentication middleware was written. This middleware takes the accessToken provided during the handshake phase, checks with Keycloak whether the token is valid, and populates the user to the websocket scope. If the token is valid, the middleware populates the corresponding user, otherwise it will populate the Django object 'AnonymousUser' which represents a non-authenticated user.

```
application = ProtocolTypeRouter({
    ...
    "websocket": AllowedHostsOriginValidator(
        TokenAuthMiddleware(URLRouter(websocket_urlpatterns))
    ),
})
```

Listing 4.2. Securing the websocket with the help of middlewares

As shown in Listing 4.2, the websocket is wrapped by the 'TokenAuthMiddleware' and the 'AllowedHostsOriginValidator' middleware, which fulfills the purpose described above of securing the websocket.

```
async def connect(self):
        user = self.scope["user"]

        # User has no permission, deny access
        if user.is_anonymous or not await self.
    user_has_chapter_permission(user, self.chapter_id):
            await self.close()
            return
        ...
        await self.channel_layer.group_add(self.room_name, self.
    channel_name)
        await self.accept('Authorization')
```

Listing 4.3. Websocket connection phase user check

With the population of the user, it is now possible to check whether the user is authenticated and has permission to access the chapter at any point. For Classroom this check is conducted in the connection phase of the websocket, as shown in Listing 4.3. If the user fails this check, the connection is rejected. If the user passes the check, they are added to the room and the connection is accepted.

Keycloak Token Verification

As described in Section 2.1, Keycloak is responsible for user management. The backend application can therefore not validate a token by itself. Therefore, whenever the

backend application receives a request, it checks for the 'HTTP_AUTHORIZATION' header. If no header is present, the request is clearly not from an authenticated user. If the header is present, the backend application forwards the provided JWT token to the Keycloak server in order to validate it. Once the token has been validated, the backend knows that the token is valid and also receives the corresponding user information from the Keycloak server. This way, the authentication process is securely handled by the Keycloak server and the backend application only acts as a broker.

Environment Variables Encapsulation

The backend requires sensitive data, like secret keys used to connect to the Keycloak server or a password used to connect to the database. Having this information inside the source code would not only mean that every developer can access it, but would also mean that the source code could never be published as open source. In Classroom, this data is therefore loaded from the environment. For local development this means that a developer creates a .env file which is listed in the .gitignore file so that it never gets added to the source code. As the local setup also uses a local database over a Docker container, the developer can set the database password arbitrarily and has no need of knowing the productive or test database password. For Classroom's productive and test instances running in the Kubernetes cluster, these secrets and passwords are set up as GitLab [11] CI/CD masked variables and are only injected when deploying the containers. This means that these environment variables are not available at build time and are not part of the Docker image, which is important because it allows for publishing the Docker images without the risk of leaking sensitive data. The masked property ensures that the values of the variables are not displayed in GitLab job logs. If a variable would be displayed, it is replaced with [masked] by GitLab, which also avoids leaking sensitive data.

4.3 Frontend

The frontend itself contains no sensitive data which would have to be secured independently of the backend. However, it is still important from a user experience perspective to regulate access to individual pages and give meaningful responses in cases where users are trying to access pages for which they have no permissions. Otherwise, a user might, for example, be able to access the chapter page even though they have no access to the chapter content and would receive a broken chapter page with an empty chapter. Therefore, Classroom implements navigation guards which are applied before a route is entered.

Guard name	**Action**
requireLoggedIn	Redirect the user to the login page if the user is not logged in
requireTeacher	Require the user to be a teacher, otherwise return 'redirectAccessDenied'
redirectAccessDenied	Return the 'access denied' page

Table 4.3. Classroom's navigation guards

```
        {
            path: '/account',
            name: 'account',
            props: {signup: false},
            component: () => import('../views/AccountView.Vue'),
            beforeEnter: (to) => {
                return requireLoggedIn(to);
            }
        },
```

Listing 4.4. Navigation guard before a route is entered

Listing 4.4 shows an example of how a guard is applied on the account page, since accessing the account page only makes sense if the user is logged in.

```
        {
            path: '/chapter/:chapterId',
            name: 'chapter',
            props: true,
            component: () => import('../views/EditorView.Vue'),
            beforeEnter: async (to) => {
                if (!isLoggedIn()) return redirectLogin(to);

                // Awaits the response from the backend
                const {data, error} = await getChapterContent(String(
    to.params.chapterId));

                // In case the backend returned an error
                if (error) return redirectAccessDenied(to);

                const chapterEditor = useChapterEditor();
                chapterEditor.setChapter(data);
            }
        },
```

Listing 4.5. More complex navigation guard with backend communication

When a page which requires certain user permissions is entered, the frontend cannot know whether the user has these permissions. As illustrated in Listing 4.5, the frontend receives the information whether the user has permission from the backend when it tries to fetch the data, as described in Section 4.2. The frontend therefore can check whether the backend returns an error and if it does, return the 'access denied' page. If no error occurs, the chapter is displayed.

Chapter 5

Continuous Integration and Continuous Delivery

As described in the project requirements, Classroom will be further developed mainly by undergraduate and graduate students from ETH Zurich. The curriculum of ETH Zurich is highly research focused and many of these students have no hands-on experience with software engineering. Therefore, it is possible and realistic that Classroom will be their first point of contact with software engineering. In order to make development smoother, a fully automated CI/CD pipeline was set up, which means that no additional system administration knowhow is required for deploying new developments to Classroom. This pipeline ensures that the newest version of the code is available at all times on the deployed instances of Classroom and consists of following stages:

- **Build**: Builds the Docker container and pushes it to the ETH Zurich Inf Docker registry.
- **Deploy**: Deploys the Docker container to the ETH Zurich Inf Kubernetes cluster.

The repositories involved in the pipeline are:

- **Frontend repository**: Responsible for building the Vue Docker container.
- **Backend repository**: Responsible for building the Django Docker container.
- **Deployment repository**: Responsible for deploying the previously built containers to either the test or production environment.

5.1 Test Environment

To allow teachers to test newly developed features, a test environment was set up. This environment has exactly the same specifications as the production environment but has its own database and uses a different Keycloak realm 'abz-test'. The purpose of this environment is that when new features get developed, they first get deployed to the test environment and then, upon further testing and feedback from teachers, to the production environment which allows for higher code quality as bugs are found before they are released in the production environment.

5.2 Automated Deployment

To fully automate the 'build' and 'deploy' processes, a multi-project trigger pipeline was set up. The trigger is configured on the backend and the frontend repositories and triggers on the following conditions:

- A commit is made on:
  - **'develop' branch**: This triggers the test environment deployment.
  - **'main' branch**: This triggers the production environment deployment.

```
1 trigger_deployment:
2   stage: trigger_deployment
3   only: [ 'main', 'develop' ]
4   variables:
5     UPSTREAM_BRANCH: ${CI_COMMIT_BRANCH}
6   trigger:
7     project: prv-dkomm/projects/webtp-classroom/webtp-classroom-
    deployment
8     branch: main
```

Listing 5.1. Trigger deployment stage

The variable 'UPSTREAM_BRANCH' is used to pass the branch which triggers the deployment. This can be either 'main' or 'develop', as the stage is limited to only these two, as shown in Listing 5.1. This ensures that when following the best practice Git workflow, only finished features are deployed. The variable is then used on the deployment repository to either deploy the test or the production environment.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

With the development of Classroom, this thesis has resulted in a new modern learning platform that meets modern security standards, allows teachers and students to use state of the art technologies to teach and learn, and, with the collaboration feature, further enables teachers to work as a team. As outlined in this thesis, Classroom meets all the requirements set for this project. I have laid the foundation and have taken the necessary software architecture decisions for Classroom to be easily further developed by future undergraduate and graduate students. During this thesis, I faced various challenges which proved to be a valuable learning opportunity in the area of software engineering. I am very satisfied with the final product of this thesis and am excited to see what the future will bring for the further development of Classroom.

6.2 Limitations

While Classroom is productive and fully usable, the current state of the project still entails certain limitations which are discussed in the following sections.

No Content Moderation

The most pressing concern in the current implementation, in my view, is that currently, each teacher can create unmoderated public courses. While it is unlikely that actual teachers will use Classroom in a malicious manner, the fact that anyone can identify as a teacher on Classroom could cause unwanted issues. The content creation possibilities with Classroom are very powerful: Besides the exercises, they range from integrating other websites as iframes to uploading images and links to external resources. This could be misused to upload and display inappropriate content or content linked to criminal activities. Furthermore, the iframe possibility could be used to display advertisement from other pages and generate clicks via Classroom. Therefore, it may make sense to establish a moderator role for the platform who checks the content of courses before they are publicly released. While this comes with the drawback that it causes extra work for the moderators and results in a slower content sharing environment, it would create a safer learning environment.

The Power of a Teacher

In the current implementation of Classroom, the role of 'teacher' comes with a notable amount of power that could potentially be misused. When a teacher adds a student to their course, the student cannot leave the course. Giving students the possibility to leave courses and delete the data linked to their course activity would mean that students who performed badly could use this to get rid of their bad results, which, of course, would be unwanted. While teachers therefore require a certain degree of power over students, this functionality of adding students to courses without their consent could also be misused to display unwanted content to students. A possible solution to this issue would be to create a consent request upon adding a student to a course. A student would then only get added to a course once they accept the request or if they join the course themselves using a course code.

6.3 Future Work

There are multiple areas where Classroom could be productively expanded upon in the future. Together with my supervisor Alexandra Maximova, I have gathered possible directions for this future development in GitLab and outline some of the most promising approaches in the following section.

Import/Export Feature

As a teacher, it never feels good to be bound to a platform. Therefore, teachers should have the possibility to easily export the content of their courses in a human-readable, parsable format. In the same way, it should be possible to import courses into Classroom from the same format.

Exam Mode

Currently, a chapter is visible to students as soon as it is published and it remains visible forever. However, when writing a closed-book exam, students should not have access to all the course material. Therefore, an exam chapter which has a start and an end time and can only be accessed by students after providing a password could be introduced. During an active exam, students would not be able to access any course material with this feature.

Giving More Power to Teachers after Publishing Content

As described in Section 2.2, out of fairness for students, teachers cannot change their chapter content after publishing it. During an early test run of Classroom, a few teachers have stated that this is a deal breaker for them. Classroom is intended to be a platform that teachers love and should it turn out that the majority of teachers need the ability to change content at any given time, Classroom should allow them to. However, it is important to understand that this can come with new problems that have to be accounted for. For example, a teacher might change answers on an existing exercise, resulting in phantom answers for students who have already added submissions with the original answer options.

More Advanced Exercise Types

Beside the coding exercises, the exercises present in Classroom at the moment are rather basic. However, the way Classroom is built allows for the creation of any kind of custom exercise. Therefore, the addition of more advanced exercises would be desirable and a welcome opportunity to demonstrate Classroom's capabilities as a learning platform. An example for a more advanced exercise would be a task where students have to draw a shape displayed in an image using the turtle from WebTigerPython and get visual feedback on whether the shape was correctly drawn. Another possible addition would be tree graph exercises where students have to construct a tree based on a given task.

Bibliography

[1] Angular. https://angular.dev/. Accessed: 07.11.2024.

[2] Clemens Bachmann. WebTigerJython 3 A Web-Based Python IDE Supporting Educational Robotics. Master's thesis, ETH Zurich, 2023.

[3] Codemirror. https://codemirror.net/. Accessed: 16.10.2024.

[4] Django. https://www.djangoproject.com/. Accessed: 22.10.2024.

[5] Django channels. https://channels.readthedocs.io/en/latest/. Accessed: 22.10.2024.

[6] Django-extensions. https://django-extensions.readthedocs.io/en/latest/. Accessed: 25.10.2024.

[7] Django REST framework. https://www.django-rest-framework.org/. Accessed: 16.10.2024.

[8] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989.

[9] ETH Code Expert. https://expert.ethz.ch/info/. Accessed: 06.11.2024.

[10] Remo Frei. WebTigerJython-Classroom. Gym-Inf thesis, Universität Freiburg, ETH Zurich, 2024.

[11] Gitlab. https://about.gitlab.com/. Accessed: 06.11.2024.

[12] What's different about the new google docs: Conflict resolution. https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_22.html. Accessed: 07.11.2024.

[13] JSON. https://www.json.org/json-en.html. Accessed: 07.11.2024.

[14] Keycloak. https://www.keycloak.org/. Accessed: 16.10.2024.

[15] Kubernetes. https://kubernetes.io/. Accessed: 07.11.2024.

[16] LaTeX. https://www.latex-project.org/. Accessed: 07.11.2024.

[17] Nils Leuzinger. A file format and browser-based framework for creating digital dynamic learning content. Master's thesis, ETH Zurich, 2023.

[18] Monaco. https://microsoft.github.io/monaco-editor/. Accessed: 16.10.2024.

[19] Moodle. https://moodle.com/. Accessed: 06.11.2024.

[20] Nuxt content. https://content.nuxt.com/. Accessed: 06.11.2024.

[21] Pinia. https://pinia.vuejs.org/. Accessed: 22.10.2024.

[22] Postgres. https://www.postgresql.org/. Accessed: 22.10.2024.

[23] Prosemirror. https://prosemirror.net/. Accessed: 16.10.2024.

[24] Python. https://www.python.org/. Accessed: 07.11.2024.

[25] Quill. https://quilljs.com/. Accessed: 16.10.2024.

[26] React. https://react.dev/. Accessed: 07.11.2024.

[27] Redis. https://redis.io/. Accessed: 22.10.2024.

[28] Remirror. https://remirror.io/. Accessed: 16.10.2024.

[29] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types.

[30] Tiptap. https://tiptap.dev/. Accessed: 16.10.2024.

[31] Vue. https://vuejs.org/. Accessed: 22.10.2024.

[32] Vuetify. https://vuetifyjs.com/en/. Accessed: 22.10.2024.

[33] y-quill. https://github.com/yjs/y-quill. Accessed: 16.10.2024.

[34] Y-websocket. https://github.com/yjs/y-websocket/. Accessed: 16.10.2024.

[35] Yjs. https://yjs.dev/. Accessed: 16.10.2024.

[36] Ypy-websocket. https://github.com/y-crdt/ypy-websocket/. Accessed: 16.10.2024.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

◉ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

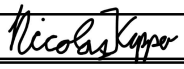
WebTigerPython Classroom

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Kupper	Nicolas

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Bern, 07.11.2024	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard