

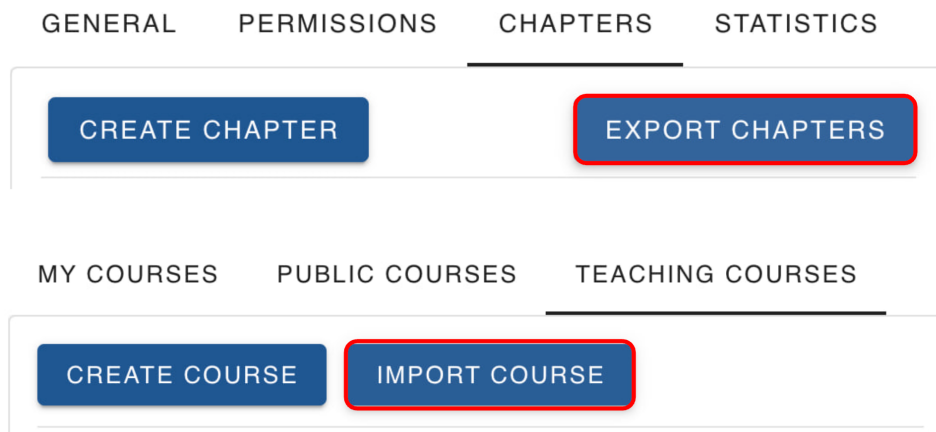


# Optimising the Classroom Platform: Import/Export Functionality

Daria Denk

Bachelor's Thesis

01.07.2025



**Supervisors:** Prof. Dr. Dennis Komm  
Alexandra Maximova



# Abstract

This thesis extends the Classroom learning platform by implementing an import/export functionality for course content. Motivation is the need for teachers to retain control over their material. The solution supports published and unpublished content, maintains formatting and exercises, and ensures consistency using a single backend transaction. Extensive testing including edge cases, malicious inputs as well as a teacher test were performed to validate functionality.



# Acknowledgment

I would like to sincerely thank my supervisor Alexandra Maximova. First of all, for selecting this suitable topic for me. Also, I am thankful for the open communication and for the support throughout this thesis. She gave me the freedom to explore my own ideas while guiding me with thoughtful questions and valuable input.

I would like to thank my supervisor, Prof. Dr. Dennis Komm, for giving me the opportunity to join a conference about WebTigerPython. It showed me that the work I was doing was important and meaningful for teachers.

I owe special thanks to Nico Kupper, who supported me far beyond what could be expected. Although his own work was already finished, he took the time to help me with technical questions and gave detailed and thoughtful feedback. His support made a real difference and I am very grateful for his help in this thesis and for his acro-yoga tips.

I also want to thank Domink for taking the time to test the new feature and some other parts of Classroom.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Main Contribution . . . . .	1
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Introduction to Classroom . . . . .	3
2.2	Core Data Model: Courses, Chapters, Exercises . . . . .	3
2.3	Chapter Content Pipeline . . . . .	4
	Quill Editor . . . . .	4
	Delta Format . . . . .	5
	Yjs and base64 Storage . . . . .	5
	Published Chapters . . . . .	5
2.4	Copy Chapters . . . . .	6
2.5	Frontend Architecture . . . . .	6
	Vue.js . . . . .	6
	Vuetify . . . . .	7
	Download.js . . . . .	7
2.6	Validation Rules . . . . .	7
2.7	Backend Architecture . . . . .	7
	URLs and Views . . . . .	8
	Django Serializer . . . . .	8
<b>3</b>	<b>Export</b>	<b>9</b>
3.1	Design . . . . .	9
	Export Formats . . . . .	9
	Button Placement . . . . .	12
	Database Fields . . . . .	12
	Error Handling . . . . .	13
3.2	Exporting JSON . . . . .	14
	Handling Published and Unpublished Chapters . . . . .	14
	Django Serializer . . . . .	14
3.3	Generating Markdown . . . . .	15
	Format . . . . .	15
	Parsing Quill Delta . . . . .	16
3.4	Testing . . . . .	16
	Edge Cases . . . . .	16
	Teacher Test . . . . .	17

<b>4</b>	<b>Import</b>	<b>19</b>
4.1	Design . . . . .	19
4.2	Implementation . . . . .	20
4.3	Validation . . . . .	21
	Frontend . . . . .	21
	Backend . . . . .	22
4.4	Testing . . . . .	22
	False Uploads . . . . .	22
	Teacher Test . . . . .	23
<b>5</b>	<b>Conclusion</b>	<b>25</b>
5.1	Limitations . . . . .	25
5.2	Future Work . . . . .	25



# Chapter 1

## Introduction

This thesis presents the design, implementation and testing of an import/export feature for the Classroom learning platform. The aim is to enable teachers to save, reuse, and transfer course content independently of the system. Although the platform already supports course creation, editing, and publishing, it previously lacked functionality for structured data export and re-import. The work involved both frontend and backend development.

### 1.1 Motivation

For Classroom [3] to be accepted and widely used by teachers, one feature is particularly crucial: content sovereignty. Teachers want to retain full control over their teaching materials. They are often hesitant to rely solely on a platform to store their content, preferring instead to keep ownership and maintain the ability to transfer their materials to other accounts or platforms if necessary. To support this need, Classroom should offer a straightforward and intuitive export functionality that allows teachers to manage their content independently and securely.

Teachers also want to export their content to store it themselves, without needing to trust the platform used. They want to be able to import their content back into the platform.

### 1.2 Main Contribution

My first main contribution is that I created a course export feature that supports several export formats. The user can choose between a structured JSON file, which can be imported, a human readable version in markdown (with or without solutions), and a lossy but human readable text file, which contains JSON. The export supports both published and unpublished chapters. For the markdown export a parser was needed that I also wrote.

The second contribution is that I implemented a course import feature via JSON upload. A JSON file can be uploaded. The course and chapter data are then parsed. The uploaded data are validated, and the platform provides feedback and error messages in case of problems.

I made sure to use existing features, such as the Quill editor or the `convertExercises` logic. I used and extended the state management with Pinia. And used the same

style for dialogs with the existing Vuetify components. I made the architectural decision to implement the export logic in the frontend and used the backend for validating and storing imported data. I stayed consistent with the existing serializer and Quill/Yjs setup. Teachers can now export their content and store it independently from Classroom.

## Chapter 2

# Background

This chapter gives the necessary background to understand the implementation of the import and export features. It introduces Classroom, explains the underlying data model and content processing pipeline, and describes important parts of the frontend and backend architecture.

### 2.1 Introduction to Classroom

Classroom is a web-based learning platform for creating and managing programming exercises. It uses WebTigerPython. WebTigerPython is a minimalistic, web-based Python environment designed for beginners. Students can enter and run code directly in the browser. Classroom uses this application and adds functionality, such as creating classes, courses, and assigning roles and rights. Teachers can use it to create, distribute, collect and grade exercises. Students can solve them directly in the environment.

### 2.2 Core Data Model: Courses, Chapters, Exercises

Classroom enables teachers to create courses. A course has an owner, a name, a description, and can contain chapters. Each chapter itself has a name and some attributes, like "is public", "contains WebTigerPython" or "is published". Each chapter can also have content.

This content is stored depending on whether the chapter is published or unpublished. For unpublished chapters, it is stored as a Quill Delta within a Yjs document, encoded as a base64 string. It may include embedded exercises. As soon as a chapter gets published exercises are recognized and replaced with an ID, which can be seen in Figure 2.1. The exercise itself gets its own entry in the database, what enables the student solutions to be assigned to the correct exercise.

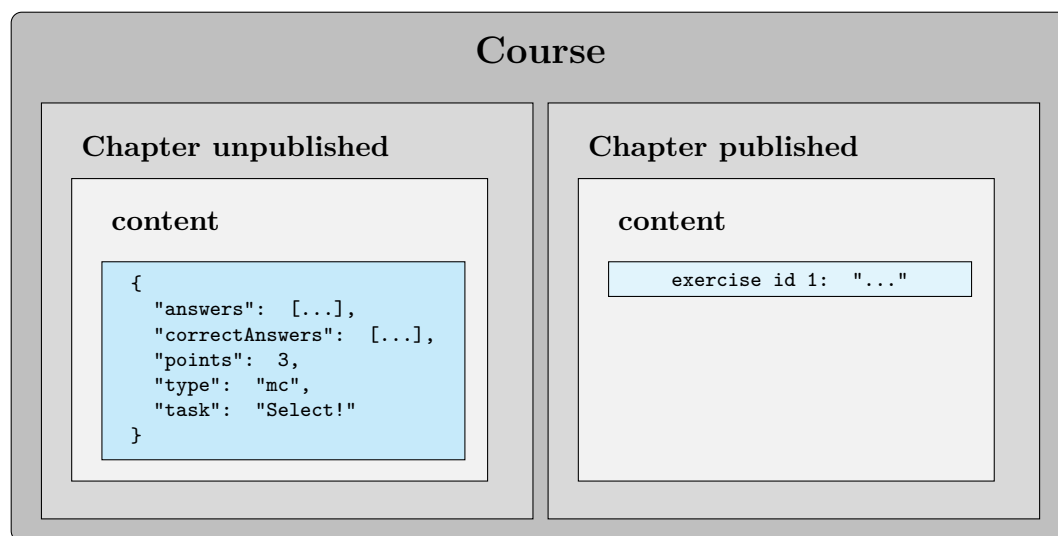


Figure 2.1. Published versus unpublished chapter

## 2.3 Chapter Content Pipeline

For the import and export of chapters, it is crucial to understand how chapters are stored, converted and how they are transferred from the editor into the database.

### Quill Editor

After creating a course and chapter, the user can start entering the content of a chapter. This is done in the Quill editor. While a chapter is unpublished the content is stored as one string in the content field in the database. But getting there is quite a journey.

First, the user sees the interface that can be seen in Figure 2.2.

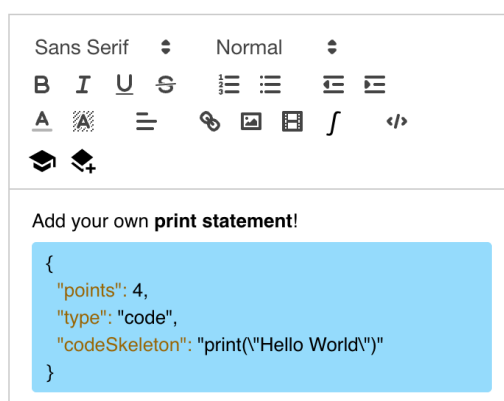


Figure 2.2. Quill Editor

One can notice that this is not only plain text. There is some bold text and a blue box around the exercise. This formatting has to be encoded. What is the Quill Editor's responsibility. The chapter content starts now as follows:

```
"ops": [ "insert": "Add your own " , "attributes":
  "bold": true , "insert": "print statement" , "insert": "!\n"
, "attributes": "exercise-token": true , "insert": "{" ,
"attributes": "exercise-block": "json" , "insert": "\n" ...
```

One can see that the bold text is stored as an attribute and that there is an exercise block for our coding exercise. Classroom has collaborative editing. To support this, the content is synced through an Yjs document (yDoc). There it has a similar representation to the one we have seen so far, but wraps its functionality around it. We can still access the content in the seen format, with `yDoc.getText('quill').toDelta()`.

## Delta Format

After wrapping the quill delta format into a yDoc, the result is stored as an int-Array and looks like this:

```
[6, 2, 163, 231, 250, 203, 11, 0, 198, 173, 160, 174, 142, 3, 77, 173,
160, 174, 142, 3, 78, 4, 98, 111, 108, 100, 4, 116, 114, 117, 101, 198,
173, 160, 174, 142, 3, 94, 173, 160, 174, 142, 3, 95, 4, 98, 111, 108,
100, 4,...]
```

## Yjs and base64 Storage

This binary representation is then encoded in base64 before being stored in the database. This is how we send our chapter content to the backend to store it.

```
BgKj5/rLCwDGraCujgNNraCujgNOBGJvbGQEdHJ1ZcatoK60A16toK60A18EYm9sZARudW
xsA574/cUHAMatoK60AwCwls7dBRE0ZXhlcNpc2UtdG9rZW4EdHJ1ZcatoK60AwGwls7d
BSU0ZXhlcNpc2UtdG9rZW4EdHJ1ZcGA0P/DBh0wls7dBVgBFIDQ/8MGAEawls7dBQA0ZX
hlcNpc2UtdG9rZW4EdHJ1Zcawls7dBQGWls7dBQIOZXhlcNpc2UtdG9rZW4EbnVsbMaw
ls7dBQWwls7dBQY0ZXhlcNpc2UtdG9rZW4EdHJ1Zcawls7dBQ6wls7dBQ80ZXhlcN...
```

## Published Chapters

As soon as a chapter is published, Classroom adds some functionality to the created exercises. Students can, for example, solve them and their solutions are stored. That would not be possible with just the above described format, as there is no way to connect student solutions to the corresponding exercises.

When a chapter is published, the content is analyzed and exercises are detected. Each exercise is extracted from the content string and replaced by an ID. The exercise itself is then stored under that ID in the database. In the delta format we no longer see the exercises, but their IDs:

```
[ "insert": "Add your own " , "insert": "print statement",
"attributes": "bold": true , "insert": "!\n{" , "insert":
"\n", "attributes": "exercise-block": "json" , "insert": "
  "id": "02e9ebf1-be58-44db-93c1-c8aa3ee20cea" " ... ]
```

When users look at the chapter in the editor, those exercises are inserted back at the corresponding position so that they can still view the text they originally entered. But the underlying structure has changed.

## 2.4 Copy Chapters

When copying a chapter, only the chapter's content is duplicated. For example, if a chapter already has student solutions, these are not copied.

As soon as a chapter is published, it can not be unpublished anymore. If a user needs to change an exercise, the only way to do that is to copy the chapter and edit it in the new unpublished version.

In what follows, we describe the copy chapter functionality for published chapters. Unpublished chapters are just copied and assigned a new chapter id. That's it. But for published chapters, we need to restore the original content without the ids, but the actual exercises. This is done with the help of a function `convertExercises()`. It injects the exercises back into the content of a chapter. Later, this functionality will be used to reassemble the published chapters.

## 2.5 Frontend Architecture

The frontend architecture is built using Vue.js and leverages Vuetify for UI components and Pinia for state management.

### Vue.js

The application uses Vue.js, a framework for building web user interfaces [5]. It provides several functionalities, like logic, data-flow or reactivity. The data flow is based on a so called store.

### Pinia Course Store

We expect the website to function well under different conditions. For example, we want it to work if several tabs are open. We want certain dialogs to show up at the right time and close again. This is managed inside a so-called store. Its job is to centrally manage the state and logic for course-related data and UI dialogs. Classroom has several stores like `chapterCorrection.ts` or `coursesStore.ts`. We will take a closer look at `coursesStore.ts` because this is where the windows are managed that are shown during the export.

This store is quite basic and stores only the following attributes:

- `teachingCourses`
- `studentCourses`
- `publicCourses`
- `showCourseCreator`
- `showCourseDelete`

In `teachingCourses` all the courses are saved, that are displayed at the Teaching-Courses page. The same applies to student and public courses. The `showCourseCreator`

and `showCourseDelete` are Booleans. If set to `true`, the windows are shown to the user. These properties can be updated during a session and are accessible to other components. As we also show a dialog, we needed to add it to the store.

## Vuetify

Vuetify [2] is a library for the user interface, built specifically for Vue. It provides pre-built UI components (buttons, dialogs, cards, loading spinners, progress bars etc.). To be consistent with the existing design, I used Vuetify components as well.

## Download.js

For file downloads, the lightweight JavaScript library Download.js is already used in Classroom. It generates and triggers the download of the created file (such as JSON or PDF). This is done directly in the browser using the following function call:

```
download(new Blob([content], type: "..."), filename);
```

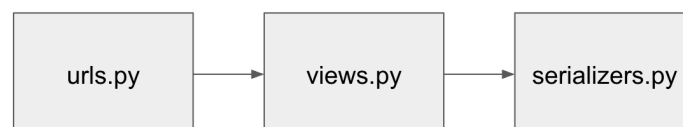
So far, it has been used to download data, such as the course code history of students, in a JSON file. This same mechanism is reused in the export feature I added.

## 2.6 Validation Rules

So far the user can enter free text when naming a chapter, writing a description, or entering text in the Quill editor itself. This data gets validated. For example, the length of the entered text in a title is limited. The frontend has the purpose to inform the user about false input. This is done with several rules that are checked when input is entered. As the backend cannot trust that it gets only validated data, because an attacker can bypass the frontend, it rejects false requests with the help of Django serializers.

## 2.7 Backend Architecture

The frontend communicates with the backend via HTTP API calls. The entry points are defined as URL patterns in `urls.py`. The path defines the URL structure and can be chosen freely, as long as it matches the requests made by the frontend.



Each URL is linked to an associated view. All view functions are defined in `views.py`, and the one corresponding to the matched URL is executed. A view is a

function that may call a serializer. These are all standard constructs provided by the Django framework.

## URLs and Views

As entry points to the backend, two new URLs were added. Since their structure and naming follow existing conventions, I'll explain their composition here:

```
path("course/export/chapters/<str:course_id>/", views.export_chapters)
```

The `<str:course_id>` is a parameter that is passed to the view function. The `views.export_chapters` refers to a function that will be executed when the backend receives a request matching the specified URL. Typically, it returns a JSON response.

## Django Serializer

For the communication between the front-end and the backend, the data need to be converted to datatypes that are most suitable for each specific use. The serializers in the Rest Framework in Django [1] provide a solution to this. We take a look at an existing serializer in our backend and walk through the structure.

```
class CourseCodeHistoriesSerializer(serializers.ModelSerializer):
    courseId = serializers.SerializerMethodField()
    chapters = serializers.SerializerMethodField()
```

It has a name, and in the beginning custom serializer fields can be assigned. In this case, we have the `courseId` for renaming purposes only and the `chapters` with a filter. For all custom fields, one has to define a function that returns what we want in our field.

```
class Meta:
    model = Course
    fields = ['courseId', 'name', 'chapters']
```

Then we need to define what `model` the serializer is based on. A Django model is a class that defines the structure of the database tables. It describes how data is represented in Classroom. So in this case the class `Course` has a field `name`. As we have not declared that on top, it must already be in `Course`. Now, the functions for the additional fields are defined.

```
def get_courseId(self, obj):
    return obj.id
```

We have a field `CourseId`, but in a `Course` the desired field is called `id`. So, it is renamed here. A serializer can have more sophisticated functions as well. This is needed, if for example, we only want some chapters or call another serializer. So what the serializer does is getting data base64-decoded from the database and returning it in JSON format. A similar serializer was added that is specialized for the export functionality.



## Chapter 3

# Export

A chapter export feature was implemented in the backend of the Classroom platform. Its purpose is to give teachers control over their content, allowing them to remain independent of the platform they use. The supported export formats are JSON, plain text, and Markdown with and without solutions. The export works for both published and unpublished exercises and is designed to support future re-import into the platform.

### 3.1 Design

Several design decisions had to be made to implement the export feature. The first was determining where users should trigger the export action. A button was added and placed where teachers would naturally expect to find it. Next, the export formats were chosen based on relevant use cases, like the potential for re-import, human readability, ease of editing, machine processability and familiarity. Finally, it was necessary to define the scope of the exported data. For example, whether to include only the chapter content or also student related data.

#### Export Formats

There are different requirements for the export feature, some of which are mutually exclusive. The most relevant ones are the following:

- **Re-importable:** The file can be imported into the Classroom platform without any loss. The import result is identical to the export.
- **Human-readable:** A human can easily read the export without further processing. Mostly, it contains the content written by the teacher.
- **Styling supported:** Text formatting is preserved in the exported format. For example, bold text remains visibly bold.
- **Easy to Edit:** Teachers can edit the exported format directly. The content is easy to modify or extend.
- **Machine-processable:** Usable by teachers who want to write simple scripts. The format is easily accessible without requiring knowledge of the Classroom system's internal architecture.
- **Familiar to Teachers:** How recognizable is the format for teachers.

The following table shows the formats that I considered. The yellow marked lines are the formats that I decided to implement.

**Table 3.1.** Export Format Comparison

Format	Re-importable	Human-readable	Styling Supported	Easy to Edit	Machine-processable	Familiar to Teachers
JSON	Yes	Hard	No	No	Hard	No
Yjs (Base64)	Yes	No	No	No	No	No
Markdown	No	Yes	Own Styling	Yes	No	Likely
Quill Plain Text	No	Partly	No	Yes	Yes	Yes
PDF	No	Yes	Yes	No	No	Yes
HTML	No	Yes	Yes	Possible	No	Maybe

Unfortunately, there is no format that satisfies all needs. Therefore we decided to export the most human-readable, but not lossy format in a JSON file. Compared to Yjs (Base64), it has the benefit that teachers can still open it and find their content. They can ensure themselves that the export worked and therefore develop trust that they are in control of their content. This format looks like this:

```
{
  "name": "All Exercises For Export",
  "description": "this course includes all exercises",
  "chapters": [
    {
      "name": "All exercises",
      "published": false,
      "webTPEnabled": false,
      "webTPUiEnabled": false,
      "content": {
        "ops": [
          {
            "insert": "Single Choice: \n"...
```

For teachers who want to further process their content, Quill Plain Text is the most relevant format. It is well-suited for those with basic programming skills who wish to write simple scripts to analyze or manipulate their exported content. The format looks like this:

```

    Single Choice:
  {
    "answers": [
      "Answer 1",
      "Answer 2"
    ],
    "correctAnswers": [
      "Answer 1"
    ],
    "points": 2,
    "type": "sc",
    "task": "This is a single choice exercise"
  }...

```

Those two formats already satisfy some requirements. But missing is a readable format that does not include any additional structure. As some teachers use editors that are based on markdown, I created two different markdown files. One that includes the correct solutions and one that does not. This is for teachers who want the least additional effort to be able to use their content elsewhere. An example with the correct solution marked looks like this:

```

# All Exercises For Export

this course includes all exercises

## All exercises

Single Choice:

**Task:** This is a single choice exercise (2 points)

- [x] Answer 1
- [ ] Answer 2

```

The requirement to preserve the exact styling from the Classroom editor was not met. However, I considered this the least relevant requirement, as teachers can always view the original content with styling directly within the Classroom interface.

In addition, the editor format is highly interactive, so a print-like representation would offer limited benefit. If needed, teachers can still export each chapter individually using the browser's built-in print functions to access a styled version.

### Chapter Export

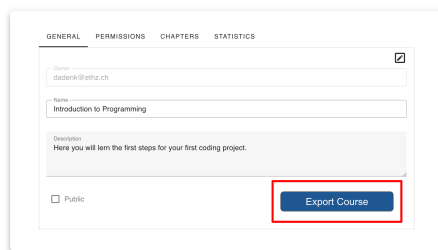
- ☒ JSON
- ☐ Text
- ☐ Markdown with answers
- ☐ Markdown without answers

**Figure 3.1.** User Interface for Export Format

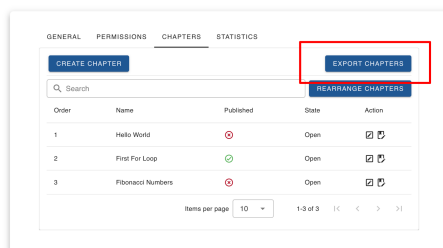
The user now has four different export formats to choose from in the interface shown in Figure 3.1. Each option serves a specific use case, allowing teachers to select the format that best fits their goal like re-importing, further editing, sharing, or simple reading.

## Button Placement

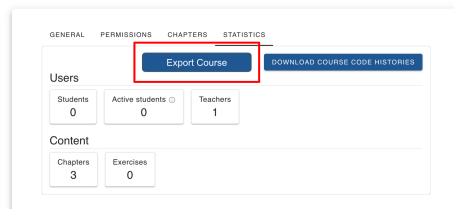
When adding a button to an existing interface, it is important to consider user expectations and existing structures. The button should be placed where users naturally look for related actions and help to conclude about its functionality.



The first possible location is the **GENERAL** page, where users can view and edit the course’s title and description. This is likely the first place a user would expect to find an export option.



The second option is the **CHAPTERS** page, which lists all chapters in the course and allows users to add new ones. Placing the export button here emphasizes that the export only includes chapter content—not student solutions or code history.



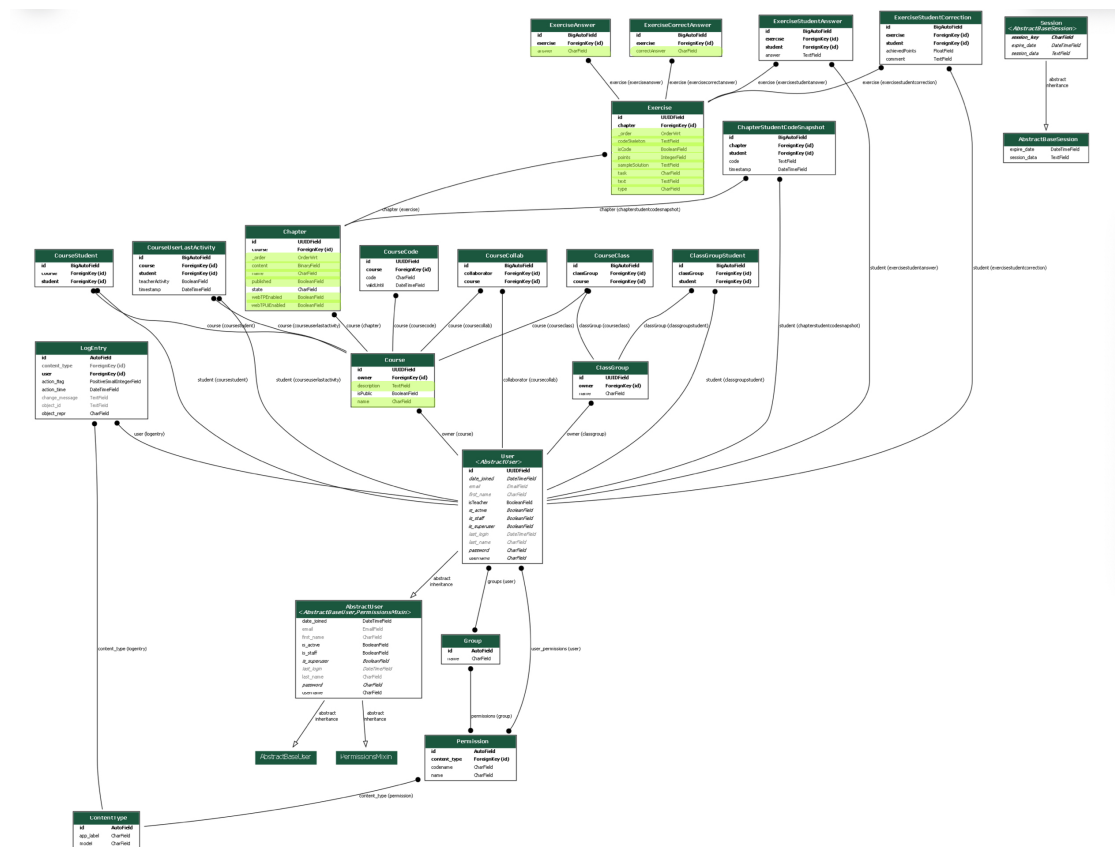
The third possible location is the **STATISTICS** page, which already contains a “Download course code histories” button. Adding the export option here would group all download-related actions in one place, providing a consistent user experience.

**Figure 3.2.** Different positions for the export button

I placed the button on the **CHAPTERS** page, because users will be familiar with this page as they use it regularly to create or edit their content. I found the risk of placing it on the **GENERAL** page or the **STATISTICS** page too big. Users could believe, that it might export the whole course and then lose data if they would not check the content of the export before deleting a whole course. On the **CHAPTERS** page it is clear that only the content of the chapters will be exported.

## Database Fields

It was necessary to determine which fields of the database are relevant for the export. Since the goal is to export only course content and not student data, many fields were excluded from the export by design. Figure 3.3 illustrates the database schema,



**Figure 3.3.** Database Overview - Relevant Fields for Export

adapted from the master’s thesis by Nicolas Kupper’s [4], with the fields included in the export highlighted.

The points of discussion were the following fields:

- **User ID:** Not included, as the exported file may be passed to another user, who should then import it into their own account.
- **Course ID:** Not included, since the course will be duplicated. Upon import, a new ID should be assigned.
- **isPublic:** Not relevant for the user. It must be reset during import, as the user could have manually modified the file.
- **Chapter state:** Only relevant if linked to student data and therefore not included.
- **published:** Exported, as it may be relevant to teachers. However, it will be reset upon import to allow for editing of all chapters.

## Error Handling

The export process does not produce any error messages, as any course that can be created in Classroom can also be exported. If a course contains an error, as in Figure 3.4, it will be shown in the editor, but the export will be possible. Required fields that are empty are simply exported as empty. I found no reason to exclude such courses from export. If a course can exist in Classroom, it can be exported.

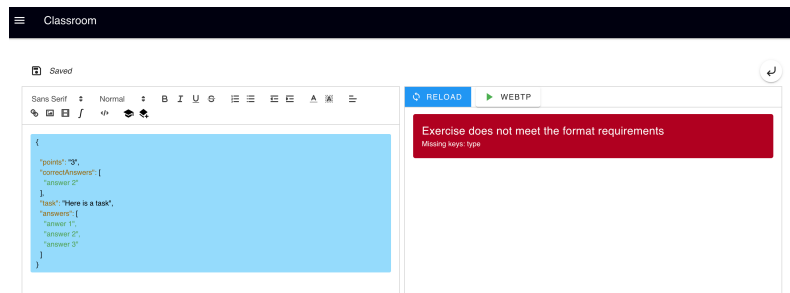


Figure 3.4. Chapter Error Example

## 3.2 Exporting JSON

One export that can be re-imported, without any loss is needed. The only wanted change after re-importing is that all chapters are set to unpublished so a teacher can edit them. As in the other versions, only chapter content is exported, not student-related data.

### Handling Published and Unpublished Chapters

For the user, the difference how published and unpublished chapters are stored is not relevant. It is a matter of organizing the database. Therefore, it does not make sense to export a chapter differently depending on whether it is published or unpublished.

I decided to treat all chapters as unpublished ones as that gives the possibility to edit them after re-importing them. Therefore, I needed to transform published chapters to unpublished ones. The Copy Chapter functionality as described above in Chapter 2 does something similar, so I reused that logic. At the positions in the content where the exercise IDs are, the exercises are injected back into their place.

### Django Serializer

The initial implementation relied on an existing serializer to retrieve chapter content. However, this approach required a separate backend call for each chapter, resulting in poor performance. To address this, a custom serializer was developed. This serializer receives a course ID, queries the relevant data from the database, and returns the selected fields, which are shown in Figure 3.5, to the frontend.

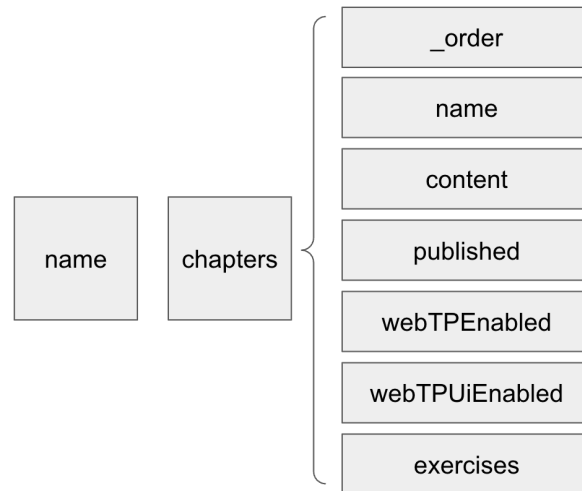


Figure 3.5. Serializer Fields

### 3.3 Generating Markdown

For generating the Markdown file, the main challenge stems from the fact that Classroom uses custom formats for exercises that are not supported by existing Quill-to-Markdown parsers. As a result, I developed a custom parser to handle the conversion of each supported exercise type into a readable Markdown structure.

#### Format

There are currently five different exercise types in Classroom:

- Single Choice
- Multiple Choice
- Coding
- Sorting
- Fill in the gaps

For each one, I decided how I want it to look like in Markdown. For example, the multiple choice exercises should look as follows:

```
**Task:** Multiple Choice Exercise (5 points)

- [ ] Answer 1
- [x] Answer 2
- [x] Answer 3
- [ ] Answer 4
```

Each task begins with a description, followed by the number of achievable points in parentheses. The possible answer options are then listed, with the correct ones marked by an “x”.

## Parsing Quill Delta

The parser is in the `CourseExport.ts` file. By case distinction, the different exercise types are detected and parsed individually.

For example, the Multiple Choice parser looks like this:

```
export function parseMultipleChoice
(exercise: ExerciseObj, solutions: boolean): string {
  let out = '**Task:** ${exercise.task ?? ""}
    (${exercise.points ?? 0} points)\n\n';
  for (const answer of exercise.answers ?? []) {
    const mark = solutions &&
      exercise.correctAnswers?.includes(answer) ? "x" : " ";
    out += '- [${mark}] ${answer}\n';
  }
  return out + '\n';
}
```

In yellow are the structures that I added to style this particular exercise in markdown. The blue parts are the content added from the exercise the teacher wrote. I did that for each exercise type and in case new exercises are added a parser has to be added for each new exercise as well.

## 3.4 Testing

To ensure the functionality of the export feature, several tests were performed. I verified that each exercise type was correctly exported in both published and unpublished chapters. In addition, various edge cases were tested. As a final test, I asked a teacher to try the new feature.

### Edge Cases

The edge cases that I tested are listed in Figure 3.6. The website responds to all of them as expected.



Order	Name	Published
1	Unpublished plain Text	✗
2	Unpublished with rich formatting	✗
3	Published chapter with mc exercise	✓
4	Published chapter with multiple exercises	✓
5	Unpublished chapter with exercises	✗
6	Empty unpublished chapter	✗
7	Empty published chapter	✓
8	Chapter with non-ASCII characters	✗
9	Chapter with same name	✗
10	Chapter with same name	✗
11	Chapter with JSON Content	✓

Figure 3.6. List of Test Cases

## Teacher Test

I asked a math teacher to test the export feature. He didn't use Classroom before and tested it for 30 min. I tried not to explain too much to him and gave him the task of creating a chapter and exporting and importing it.

He was able to do so and found that it works. He didn't like the button placement. He would prefer to have the import and export buttons next to each other on the course overview page and suggested adding a warning that student data are not exported. He preferred an additional feature that one could then choose which data is exported in a dialog and also include the student data.

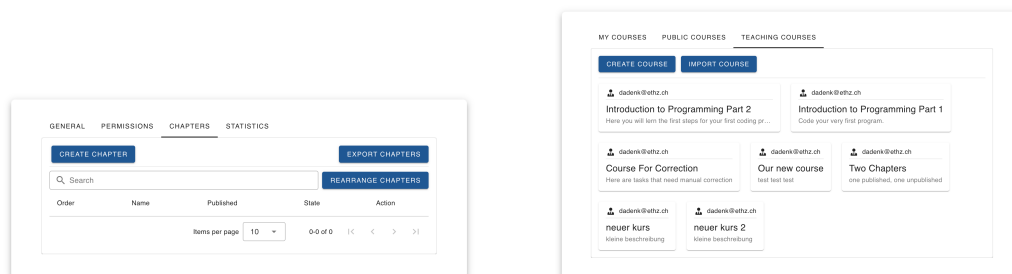
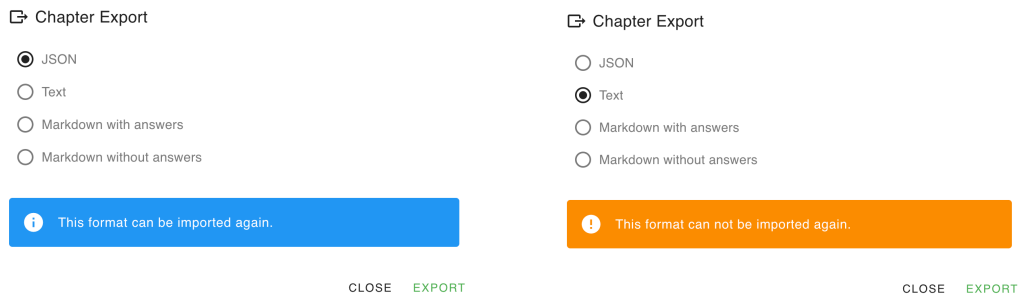


Figure 3.7. Teacher feedback on button placement

He appreciated the export interface and found the indication of whether a format

is lossy or lossless to be very clear. The colors helped him to notice the differences without problems. He clicked through the options and found the correct one for his task (to re-import the course) easily. The import worked without problems or complaints.



**Figure 3.8.** Teacher feedback on export dialog

# Chapter 4







## Import

This chapter describes the implementation of the import functionality in Classroom. It is about the design decisions, technical flow, validation steps, and testing performed to ensure that imported data are handled well.

### 4.1 Design

The goal of the import feature is to allow re-importing a course that was previously exported from Classroom. The imported course should retain original structure, with the exception that all chapters are set unpublished, making them editable again.

I initially considered allowing teachers to decide during import whether each chapter should be marked as published or unpublished. As shown in Figure 4.1, the interface would have included an icon to set the published state, which would change when it is clicked.

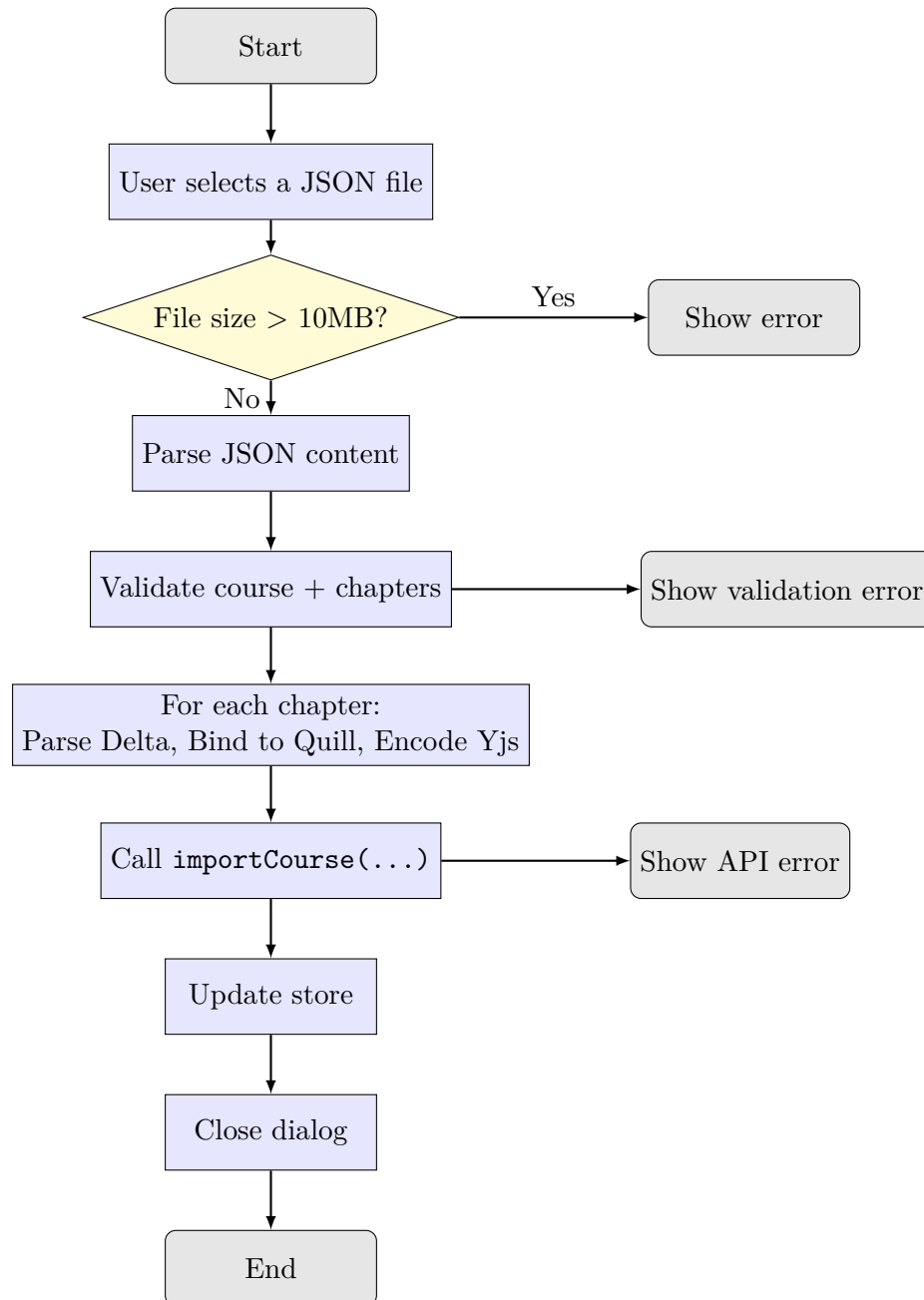
Order	Name	Published
1	Unpublished plain Text	
2	Unpublished with rich formatting	
3	Published chapter with mc exercise	
4	Published chapter with multiple exercises	
5	Unpublished chapter with exercises	
6	Empty unpublished chapter	

**Figure 4.1.** Discarded Import Dialog

I decided against including this feature, as it could lead to confusion. In the Chapter Editor, teachers are not allowed to change a chapter's published state once it has been published. This is enforced as a strict rule in Classroom. Published chapters cannot be unpublished again. Introducing a toggle during import would create inconsistent behavior and might mislead teachers into thinking that they could later revert the publish state in the editor, which is not possible.

## 4.2 Implementation

The import process follows the steps shown in Figure 4.2. Note that all chapters are sent to the backend in a single request. This reduces the number of API calls and improves performance. It also ensures database consistency. If the user starts an upload and the process is interrupted (either intentionally or accidentally), the import will succeed completely or fail entirely. This guarantees that no partial or inconsistent course is created.



**Figure 4.2.** Flowchart of the import process

The `importCourse()` call sends a request to the backend. A new backend view

was implemented using Django and the Django REST Framework. This view, accessible via the route `course/importChapters/`, enables the upload of the file.

The backend accepts a POST request containing the course's `name`, `description`, and an array of `chapters`. Each chapter must include a `name` and two fields `webTPEnabled` and `webTPUiEnabled`. Chapters can include `content`, which is expected to be a base64-encoded Yjs document.

When a request is received, the backend creates a new Course. It then iterates over the chapter array, checking for required fields. Each chapter is saved in an unpublished state. If content is present, it is decoded from base64 into binary format and stored in the database. The chapters are then linked to the course, and their order is preserved.

## 4.3 Validation

Validation of imported data is important, as users can modify the imported file. They can delete or add data or just create their own files.

One step in validation is to inform the user about false inputs with error messages, which can be handled in the frontend. The second step of validation has to happen in the backend, since users could still send invalid requests to the backend.

### Frontend

For validating the user input and giving immediate feedback, I set up rules. I only explain some rules that differ from the ones already described for another field.

#### General Rules

- `selectedFile.value && selectedFile.value.size > MAX_FILE_SIZE`  
One can only start the upload if a file is selected and the file size does not exceed a maximum that I set to 10000000 bytes.
- `accept=".json"`  
Only JSON files are accepted for upload.

#### `courseFieldsRules`

- `typeof course.name !== "string" || course.name.trim() === ""`  
An error message is shown if the name of the course is not a string or if the trimmed course name (removed white spaces at the start and end) is empty.
- `course.name.length > 60`  
The length of the name must be shorter than 60. This is analogous to the existing rule, when creating a new course.

#### `chaptersArrayRules`

- `Array.isArray(chapters)`: Chapters are in an array, but the array can also be empty.

**chapterFieldsRules**

- `typeof chapter.webTPEnabled !== "boolean"`: Check whether the field for WebTigerPython contains a boolean or not.

**Backend**

The current implementation includes validation for the required course and chapter fields. If a chapter is missing one of the expected fields or has invalid content, the process is aborted and an error message is returned. Errors are returned via `HttpResponse` to the frontend and there displayed to the user.

The following errors can be generated by the backend:

- Failed to decode content in chapter
- Missing required fields in chapter
- Import failed:

All other (and more specific) errors are generated by the frontend and shown before the user sends the request to the backend.

**4.4 Testing**

For testing the import I made several manual tests with flawed files and performed a teacher test.

**False Uploads**

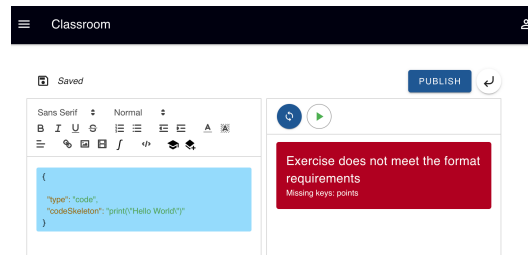
I tested uploading several flawed files to evaluate the robustness of the import functionality. These test cases include missing fields, corrupted content, and invalid values. An overview of the tested cases is shown in Table 4.1. All tests were successful (in the final test).

Testname	Description of Issue	Expected Behavior
text	File contains plain text, no JSON	Reject
name_twice	Duplicate "name" field in JSON file	Select one
Empty_Course_import	Course with no chapters	Accept
missing_webTP	Missing <code>webTPEnabled</code> field	Reject
corrupted_content	Delta format corrupted	Accept
strange_name	Only special characters in course name	Accept
#and.in_7&)(am\$	Special characters in filename	Accept
no_description	Missing course description field	Reject
name_empty	Course name is missing	Reject
empty	Empty file	Reject

Table 4.1. Upload Files

A file with `corrupted_content`, which can be parsed and is not rejected immediately, will be treated like it was written that way in the editor. The editor shows errors while editing, if the content is not correct. The corrupted content that is

imported will be treated in the same way. For example, such an error would look like follows:



**Figure 4.3.** Example for corrupted content

## Teacher Test

The teacher test for the import worked without complaints. The teacher was able to import the exported file and found it easily in the Course overview. He noticed that he could have two courses with the same name, which he liked.





## Chapter 5

# Conclusion

The import and export features have been successfully integrated into the Classroom platform, enhancing the user's options to have control over their content. These additions enable the teacher to manage and reuse their content.

### 5.1 Limitations

The teacher test revealed that the button placement may not be ideal and should be re-evaluated. Additionally, feedback from a broader group of teachers is needed to assess the usefulness and practicality of the available export formats.

### 5.2 Future Work

Additional export formats could be implemented in the future based on teacher feedback. For example, there is currently no styled export in PDF format that could be used directly as a printable worksheet.

It may also be valuable to support imports from popular platforms from which teachers are migrating. Providing compatibility with such formats could streamline onboarding and increase adoption of the Classroom platform.

Being able to export and import student data may also be important for teachers.



# Bibliography

- [1] Serializers. <https://www.django-rest-framework.org/api-guide/serializers/>. Accessed 2025-06-07.
- [2] Vuetify. <https://vuetifyjs.com/en/>, 2016. Accessed 2025-06-07.
- [3] Nicolas Kupper. Classroom. <https://classroom.ethz.ch/>, 2025.
- [4] Nicolas Kupper. WebTigerPython Classroom. Master's thesis, ETH Zurich, 2025.
- [5] Evan You et al. Vue.js. <https://vuejs.org>, 2014. Accessed 2025-06-07.

**Eigenständigkeitserklärung**

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden drei Optionen ist in Absprache mit der verantwortlichen Betreuungsperson verbindlich auszuwählen:

- ☐ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz<sup>1</sup> verwendet.
- ☐ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz<sup>2</sup> verwendet und gekennzeichnet.
- ☒ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz<sup>3</sup> verwendet. Der Einsatz wurde, in Absprache mit der Betreuungsperson, nicht gekennzeichnet.

**Titel der Arbeit:**

Optimising the Classroom Platform: Import/ Export Functionality
---

**Verfasst von:**

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

**Name(n):**

Denk

**Vorname(n):**

Daria

Ich bestätige mit meiner Unterschrift:

- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

**Ort, Datum**

Zürich, 28.6.2025

**Unterschrift(en)**

D. Denk

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

<sup>1</sup> z. B. ChatGPT, DALL E 2, Google Bard

<sup>2</sup> z. B. ChatGPT, DALL E 2, Google Bard

<sup>3</sup> z. B. ChatGPT, DALL E 2, Google Bard