Author
Adrian Vinojčić
11904250

Submission
Institute for System Software

Thesis Supervisor
DI **Lukas Makor**, BSc.

January 2025

MOODLE MANAGER

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Statutory Declaration

Abstract

Lecturers are busy people who have to juggle their research work and teaching at university. Instead of correcting home exercise submissions themselves, they hire students to correct the submissions and call them tutors. Distributing submissions to these tutors is a weekly hassle though. Therefore, the idea of a Moodle Manager came to be. The Moodle Manager logs in to Moodle, downloads the assignment submissions, and distributes them via Discord to the tutors. The result of this thesis is a program that eases the life of lecturers by taking away a part of their busy work schedule. Thanks to the Moodle Manager, lecturers have more time for their research or to answer students' questions.

Kurzfassung

Lehrende sind vielbeschäftigte Menschen, welche zwischen ihrer Forschung und dem Lehren hin und her jonglieren müssen. Anstatt die Hausübungsabgaben selbst zu korrigieren, stellen sie Studenten an, welche die Abgaben für sie korrigieren und Tutoren genannt werden. Das Verteilen von Abgaben an diese Tutoren ist dennoch ein wöchentlicher Mehraufwand. Deshalb entstand die Idee des Moodle Managers. Der Moodle Manager meldet sich in Moodle an, lädt die Hausaufgabenabgaben herunter und verteilt diese über Discord an die Tutoren. Das Resultat dieser Bachelorarbeit ist ein Programm, welches das Leben der Lehrenden vereinfacht, indem es einen Teil ihres stressigen Arbeitsalltages erleichtert. Dank des Moodle Managers haben Lehrende mehr Zeit für ihre Forschung oder für Fragen der Studierenden.

Table of Content

Contents

Abstract ii

Kurzfassung ii

1 Introduction 1

2 Background 2
- 2.1 TypeScript 2
- 2.2 Docker 2
- 2.3 Discord Webhooks 2
- 2.4 SQLite 2

3 Implementation 3
- 3.1 Architecture 3
- 3.2 Moodle Parsing 4
  - 3.2.1 Web Requests 4
  - 3.2.2 Moodle Login 5
  - 3.2.3 Retrieving the list of courses 6
  - 3.2.4 Getting the assignments for a course 7
  - 3.2.5 Downloading Submissions of an assignment 8
- 3.3 Distribution 8
  - 3.3.1 Calculating submissions per tutor 8
  - 3.3.2 Splitting into files 8
  - 3.3.3 Distribution via Discord 9
- 3.4 Website 10
  - 3.4.1 Cookie management 10
  - 3.4.2 Endpoint design 11
  - 3.4.3 Rendering HTML pages 12
- 3.5 Database and Logging 12
  - 3.5.1 Database 12
  - 3.5.2 Logging 13
  - 3.5.3 Logging in the Database 13

4 Usage 14
- 4.1 Moodle Requirements 14
- 4.2 Installation 15
- 4.3 Setup 15
- 4.4 User Interface 16
  - 4.4.1 Login 16
  - 4.4.2 Dashboard 16
  - 4.4.3 Enabling the Moodle-Management feature 19

5 Lessons Learned **20**
5.1 ChatGPT Driven Coding 20

6 Future Work **21**
6.1 Feedback visualization 21
6.2 Warnings to students 21
6.3 Tutor view 21

7 Conclusion **22**

Literature **24**

1 Introduction

Especially in technical faculties at the Johannes Kepler University (JKU), it is common for students to deliver home exercises regularly. These regular exercises are not only consuming students' time but the lecturer's time as well. Lecturers must create fitting exercises for students and answer many questions related to the assignment. Before each semester starts they hire tutors, which correct the submissions of the home exercises, to save some of the lecturers time. Consequently, the lecturers have to provide solutions for the tutors as well as download and distribute the submissions to them. All that while focusing on research, which is the central part of their job.

To have an idea of which tasks are automated by the Moodle Manager, we first have to take a look at the processes and tools used by the individuals involved. First, one of the lecturers at the Institute for System Software (SSW) distributes home exercises via Moodle. For about a week, students submit their finished homework on the same page by uploading the necessary files. When the deadline arrives, a lecturer logs into Moodle again and downloads the submissions. These submissions are then divided based on the tutors' working hours. Finally, the lecturers upload the split submissions to the correct channel in Discord, where the tutors can download them.

The goal of this project is to ease the life of lecturers by reducing their workload by a certain degree. While it is hard to automate the process of creating exercises, distributing submissions to tutors automatically saves the lecturers a lot of time. The new system automates more than half of the tasks that previously needed to be done by lecturers on the Moodle platform. The Moodle Manager (MM) also increases the quality of life for tutors since they can start correcting the submissions right away, and they do not need to worry about the lecturers forgetting to distribute the submissions.

2 Background

As the application is programmed using TypeScript, executed inside a docker container, relies on webhooks for its operation and uses an SQLite database these concepts and technologies are briefly presented in the following sections.

2.1 TypeScript

TypeScript is a language which adds a typed layer to JavaScript, which means that every variable needs to have its immutable type. TypeScript works asynchronously using the class `Promise`. Promises represent an ongoing task, which will return a value eventually. They can be resolved with a value or rejected with an error. Promises can then be awaited to extract the resolved value enclosed when the Promise is returned in an `async` method. The `async` keyword stands for asynchronous and enables the `await` keyword which allows extraction of values from promises like explained already. Another concept are callbacks, which are used all over JavaScript. Developers add callbacks, which themselves are methods, to another method. At runtime, the execution of this callback happens somewhere in the method body. The rationale for using Typescript is the intention to provide web interface features. As JavaScript is used modern browsers and node.js provides an scalable server runtime, Typescript provides a common language for both frontend and backend.

2.2 Docker

Docker is a tool responsible to package and run applications inside a container. A container is a virtualized operating system which is isolated from the rest of the system. Containers can be delivered as a unit that includes everything needed for an application to run without the need to install something else on the host system. This allows developers to have their production software run in the same environment as in development.

2.3 Discord Webhooks

Discord webhooks allow applications to send messages to Discord channels automatically. Webhooks are Uniform Resource Locator (URL)s and can be used to send an HTTP POST request to Discord. These requests contain messages which should be displayed in the channel corresponding to the webhook. In our case this will be the name ranges of the students, which are interesting for each tutor, as well as the compressed zip files containing the actual submissions.

2.4 SQLite

One of the requirements was to have a small footprint. Furthermore, a relational database was another requirement. These two reasons were enough to use *SQLite* [5] as database engine. SQLite provides an SQL based Database engine. It has a file format sharable among many platforms. The syntax of SQLite is similar to most SQL-like query-languages but provides only a trimmed down version of the functionality, hence the small image size.

3 Implementation

This chapter describes the implementation of the MM. First, we explain the architecture in Section 3.1. Section 3.2 explains how the parsing of the Moodle pages works. Section 3.3 contains the algorithm for splitting and distributing submissions via Discord. We detail the internals of the web interface in Section 3.4 and the usage in Section 4. *Database* and Logging are described in the Sections 3.5.1 and 3.5.2.

3.1 Architecture

Before explaining the concrete code, we have to define a few terms first. As shown in Figure 1, the system consists of six parts, which will be called *module*s from now on. There exists one additional hidden module, the *Logger*. Every other module accesses it to produce log messages.

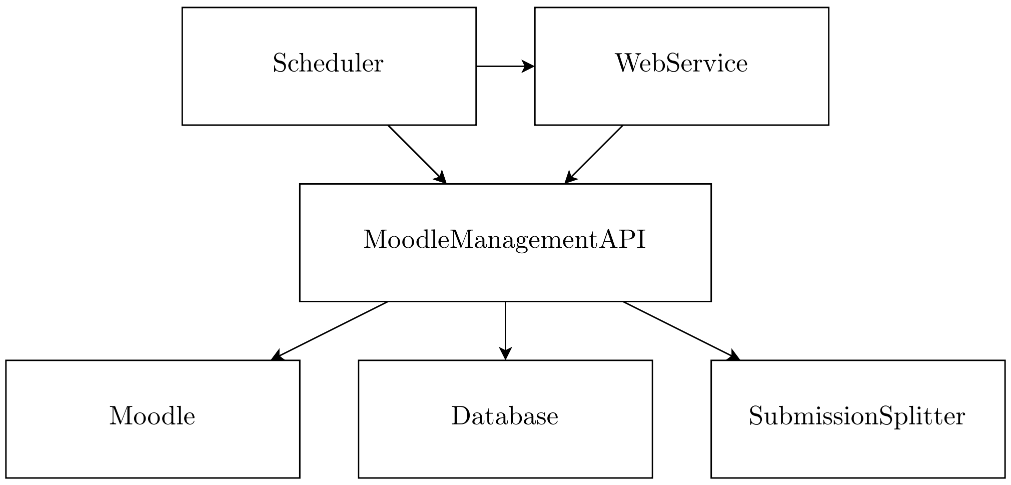


Figure 1: Top level architecture of the MM

The top level module is the *Scheduler*. At first, it ensures that all required files and directories are in place. To provide a web interface, it starts a *WebService*, responsible for handling all User Interface (UI) requests. To keep the system reliable, it is important to work with up to date data. Since Moodle does not send updates when lecturers change something in Moodle, the MM has to take care about fetching data itself. Therefore third step for the `Scheduler` is to refresh the database at startup and start a job which refreshes said database at 3 a.m. every day.

The `Scheduler` and the `WebService` use functionalities from the `MoodleManagementAPI`, which itself combines functionalities from the low level modules `Moodle`, `Database` and `SubmissionSplitter`. The `Moodle` module handles all calls to the Moodle page. The `Database` module handles all queries from the database. The `SubmissionSplitter` module splits the submissions of an assignment into parts, according to the passed tutors, and distributes the assignments via Discord.

3.2 Moodle Parsing

Unfortunately, using Moodle's Application Programming Interface (API) is not an option for the MM. Hence, we have to automate the interaction with Moodle by using the web interface which is intended for the browser. The `Moodle` module is responsible for everything related to interactions with Moodle website. The code of this module consists of a class where every object holds an instance to a connection in the form of a cookie file. The capabilities of the `Moodle` module are

- static validation of credentials.
- static creation of a `Moodle` instance by logging in to Moodle to access additional functionalities.
- to get the *course* list of the lecturer.
- to get *assignments* for a course.
- to get the due date of an assignment.
- to download the *Submission*s of an assignment.

The idea for the `Moodle` module is to retrieve the cookie for the Moodle session, store it in a file and use it for subsequent requests to the Moodle website.

3.2.1 Web Requests

There are a few possibilities to interact with websites (e. g. Selenium, which is often used for automated website testing), but in the end, everything boils down to sending HTTP requests to the desired website. A common tool for requesting resources from the web is "Client for URLs" (cURL). The command line tool cURL is available on almost every operating system. For example, a cURL request for the Moodle start page would be

```
curl 'https://moodle.jku.at/jku/'
```

These cURL requests can be extended with optional parameters. In Listing 1, these parameters are used to create a method to generate cURL requests dynamically for later use. Every command starts with the `-X POST` parameter, which defines that the request is sent using the HTTP POST request method. The `-L` parameter tells cURL to follow redirects. The parameter `-b` specifies the file from where cURL should get the cookies for the request, and `-c` sets the file where the cookies should be stored when the response arrives. In this case, both are `this.cookies`, which is a field that stores the name of the cookie file for the current `Moodle` instance. After the cookies, optionally, writing data in the cURL request is also possible. For that, the content type is set to `application/x-www-form-urlencoded` using the `-H` parameter. The function takes a string-string map and converts it to URL encoded data and adds it to the command with the `-d` parameter. Finally, we add the URL itself to the command.

```
private postCurl(
    url: string, data: {[key: string]: string} = null
): string {
    const contentType = "Content-Type:␣application/x-www-form-urlencoded"
    const values = new URLSearchParams(data).toString()
    let command = `curl␣-X␣POST␣-L␣-b␣${this.cookies}␣-c␣${this.cookies}`
    if(data) command += `␣-H␣"${contentType}"␣-d␣'${values}'`
    command += `␣'${url}'`
    return command
}
```

Listing 1: Function to create curl post commands with optional URL encoded data

3.2.2 Moodle Login

When a user logs into Moodle successfully, a browser cookie is set. All subsequent requests of this session utilize this cookie to gain access to the different Moodle pages. Therefore, we need to retrieve such a session cookie using cURL and utilize it for all follow-up requests (as explained in section 3.2.1). Logging into Moodle requires more than just sending the credentials to the server. These extra steps are due to Shibboleth, which is a system for authorization and authentication. At JKU, Shibboleth is used to implement a single sign-on system that safeguards multiple resources, such as JKU's study system or JKU's Moodle." Therefore, logging in to Moodle requires a eight-step process with four requests from the MM, as depicted in Figure 2.

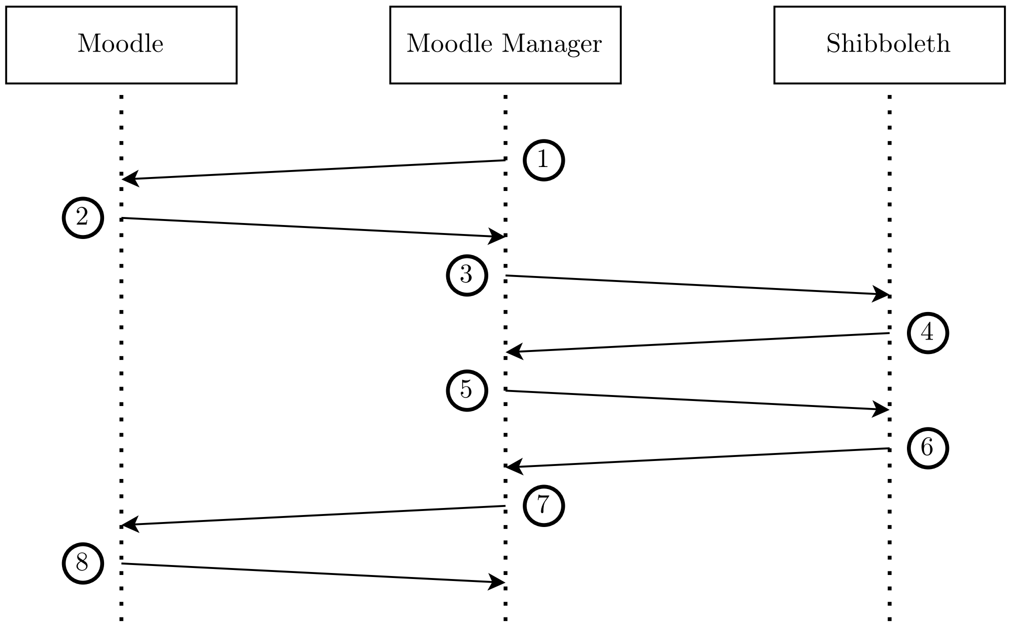


Figure 2: Data flow when logging in to Moodle

In step ①, the MM tries to access a Moodle page with the following GET request:

```
curl -L -X GET -b cookie.txt -c cookie.txt
    'https://moodle.jku.at/jku/auth/shibboleth/index.php'
```

It is important to provide the cookie file because the -L parameter is set, which means that the request is following redirects. In this redirect some cookies are set, which are required for furter login steps.

Now, one of two scenarios will occur. In the first scenario, the MM has a valid cookie for the user and sends it with the request. This cookie would allow Moodle to verify the request as valid. Furthermore, the request provides the requested resources, skipping steps ② to ⑦, continuing with ⑧. In the second scenario, no valid cookie is available, and Moodle starts a login process.

The request from step ① returns a form that would be submitted automatically by a JavaScript function once loaded in a browser. Without a browser, the MM has to send a new cURL request. The MM parses the form from step ②, and generates a request with the function from Section 3.2.1 similarly in step ③:

```
postCurl("https://shibboleth.im.jku.at[form-action]", {"form-data": "..."})
```

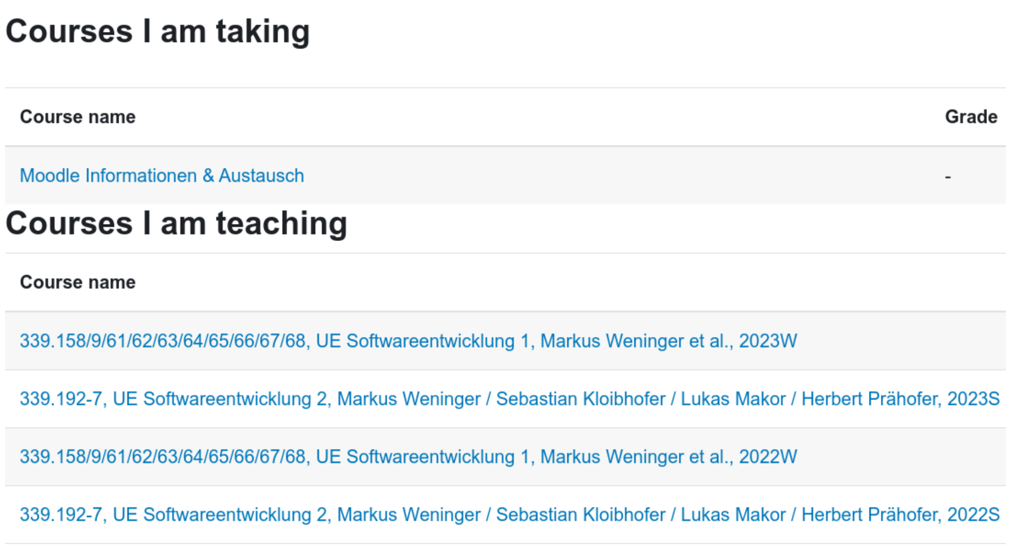
This request returns the login form from Shibboleth in step ④, which users usually see when logging in. Just as before, the MM parses the form and extracts the form action and form data. For step ⑤, the MM injects the username and password (which are provided by the user as shown in Section 4.4.1) into the parsed form data and sends the request the same way as in step ③. Now, in step ⑥ Shibboleth provides us with the wanted cookie in the headers as well as an Hypertext Markup Language (HTML) response which contains a form where the content would be submitted automatically with JavaScript, meaning that a similar procedure to that from step ③ is repeated for step ⑦. The difference to step ③ is that the action already contains the entire URL. Finally, in step ⑧, Moodle returns the initially requested page. Once the MM has extracted the cookie, the MM can use the cookie for further requests to Moodle.

3.2.3 Retrieving the list of courses

The first action that gains a lot of information is to get a list of all courses of a user. Figure 3 shows the Moodle page, available under https://moodle.jku.at/jku/grade/report/overview/index.php which can be found by selecting *Grades* in the top-right dropdown after clicking on the user icon. This URL yields a list of all courses which can be parsed into a data structure. For the MM, just the courses taught are of interest, so only parsing the second table from Figure 3 is necessary. After selecting the table with a Cascading Style Sheets (CSS) selector, the next step is to iterate over and store the elements such as the one in Listing 2. The essential details include the course `title` and the `semester`. Furthermore, the unique `courseID` is extracted and stored.

```html
<a href="https://moodle.jku.at/jku/grade/report/index.php?id=12345">
    987.654, VL Sample Course, Sam Sample / John Doe, 2023W
</a>
```

Listing 2: HTML of a course



Courses I am taking

Course name	Grade
Moodle Informationen & Austausch	-

Courses I am teaching

Course name
339.158/9/61/62/63/64/65/66/67/68, UE Softwareentwicklung 1, Markus Weninger et al., 2023W
339.192-7, UE Softwareentwicklung 2, Markus Weninger / Sebastian Kloibhofer / Lukas Makor / Herbert Prähofer, 2023S
339.158/9/61/62/63/64/65/66/67/68, UE Softwareentwicklung 1, Markus Weninger et al., 2022W
339.192-7, UE Softwareentwicklung 2, Markus Weninger / Sebastian Kloibhofer / Lukas Makor / Herbert Prähofer, 2022S

Figure 3: A snippet of the grades page in Moodle

3.2.4 Getting the assignments for a course

The SSW provides and collects all home exercises for a course on the same Moodle page. Furthermore Moodle has a sub page for every assignment which is linked in the corresponding course page. As described in Section 3.2.3, Moodle courses have unique IDs. Appending said IDs to the link `https://moodle.jku.at/jku/course/view.php?id=` results in the desired course page in Moodle when requested with a `GET` request.

These course pages usually contain links to feedbacks, assignments, learning material and other external resources. Assignments in particular stand out, because all of them contain the keyword *assign* in the URL. To find all Moodle assignments, the MM could parse the HTML to get every link instance and filter them to contain the keyword "assign". However, introducing mini exams during the semester resulted in a small problem. The exams are of the type *assign* as well, which makes this method unusable. Assignments need a sort of tag in Moodle to enable reliable detection. Since it was already a practice to put the assignment number into the section header, we now make this practise a requirement, in order to make an assignment detectable.

Using *cheerio*, the MM extracts the elements using CSS selectors. First the MM searches the element with id `region-main` which contains the main page of Moodle, then it searches and filters all list items from the *topics* list. The MM uses the Strings "UE" and "Übung" for filtering the topics. After filtering the sections, the MM extracts a data structure by parsing the assignment ID and the assignment number. Another important information for an assignment is its due date. However, as the due date is not listed on the course page, the MM has to extract the due date directly from the respective assignment page. Furthermore, the MM also stores the course ID in the data structure for easier access later in section 3.5.1.

3.2.5 Downloading Submissions of an assignment

The goal of the entire `Moodle` module is to download assignment submissions from students. We can download the submissions with the parsed assignment ID, using cURL by providing a specific parameter for the output file. The parameter `-o [destination]` pipes the output from the curl command into a file for later use. The MM uses this feature to store the assignment submissions. In the end, the command,

```
curl -L -X GET -b cookie.txt -c cookie.txt -o [destination]
    'moodle.jku.at/jku/mod/assign/view.php?id=[assignmentID]&action=downloadall'
```

downloads the assignment submissions into the destination file. In the case of the MM, this is `HOMEPATH/.moodle-manager/downloads`, where `HOMEPATH` stands for the individual user directory provided by the operating system.

3.3 Distribution

After downloading an assignment submission in Section 3.2, the next goal is to split it according to a tutor table using the `zip` library. The inputs for the following procedure is a zip file containing all submissions and a list of tutors as demonstrated in Table 1. The weight of the tutors corresponds to their working hours. This list of tutors will be following us throughout this entire section as a sample. For the MM, we divided the splitting into two simpler tasks. The tasks are the *calculation of the submission count per tutor* and *splitting the submissions into multiple files*. For the distribution to the tutors, a webhook to the correct Discord channel, in which the resulting assignment submissions are sent, is necessary as well. More on that in Section 3.3.3.

Name	**Weight**
Tom	4
Sarah	4
Maria	8
John	4

Table 1: Sample tutors

3.3.1 Calculating submissions per tutor

The first step is to shuffle the list of tutors and their respective weight, so that the tutors do not have to correct the same students every time. Next, the MM sums up the weights of the tutors and store the value into `cumulatedWeight`. Then, the accurate submissions are calculated by $submissionCount * \frac{tutor.weight}{cumulatedWeight}$. Since these accurate submissions are real numbers instead of integers, the MM takes the floor values as basis values for the corrections. We use the decimal part of the accurate submissions as the priority for the correction. We also calculate the remaining submissions by subtracting the accumulated basis from the submission count. These remaining submissions are always below the number of tutors. Therefore, the few tutors with the highest priorities (decimal part) get an additional submission to correct. Table 2 contains an example with 137 submissions and the tutors mentioned above from Table 1.

3.3.2 Splitting into files

My supervisor, Lukas Makor, has developed the splitting algorithm. Unlike the algorithm in Section 3.3.1, the following one has almost not been modified. At first, we define the constant `MAX_ZIP_SIZE = 24_500_000` because Discord only allows files of at most 25MiB to be uploaded. Then, for each tutor, the `startIndex` and the `endIndex` of the entry file are calculated. Then,

Name	**Weight**		**Accurate**	**Base**	**Priority**		**Effective**
John	4	**Submissios**	27,4	27	0,4	**Remainder**	28
Tom	4	137	27,4	27	0,4	3	28
Maria	8	**Cumulated**	54,8	54	0,8		55
Sarah	4	20	27,4	27	0,4		27

Table 2: Calculation of the submission count for the tutors

the MM iterates the entry file between those indices and write the elements to the new tutor file. If the current element exceeds the `MAX_ZIP_SIZE`, the algorithm closes the writer, stores the data to the data structure, and starts a new tutor file. After writing all submissions in the tutors' files, the names of the first and last submissions are extracted and written to a file as information for the tutor as *name range* into the zip. When all submissions are mapped to tutors, the name ranges of each tutor are added to data structure as an additional field to display a message later. This data structure is returned and used in Section 3.3.3. Table 3 contains an example of the calculation for the sample tutors. Assuming that every submission has exactly 1 MB, column *number of zips* contains the number of zips that would be produced for the respective tutor. Since the sizes of the individual submissions usually vary a lot, this is an improbable scenario.

Name	**Submissions**	**Start**	**End**	**Number of zips**
John	28	0	28	2
Tom	28	28	56	2
Maria	55	56	111	3
Sarah	27	111	138	2

Table 3: The splittings of the tutors

3.3.3 Distribution via Discord

The distribution of the submissions to the tutors happens in the same module as the splitting and uses the data structure returned from splitting the submission, as explained in Section 3.3.2. To distribute the split submissions of an assignment, a webhook from Discord is necessary.

The user needs to be logged in to Discord to obtain a webhook. *Note: Obtaining a webhook can only be done if the user is an administrator of the server.* The webhook can be obtained by clicking on the cog icon (Edit Channel) and selecting *Integrations* in the next window. Integrations in the current version show a *Webhooks* and a *Channels Followed* option. When selecting *Webhooks*, another page is displayed where creating a new webhook is possible. Once a user creates the webhook, its name may be changed, which is unnecessary for the MM because it sets the name itself. Furthermore, the webhook URL can be copied for later use in Section 4.

Once we obtain the webhook, any request according to the Discord documentation [1] can be sent using the webhook. For the MM, only the executing webhook request [2] is of essence. For that purpose, the MM uses the *fetch* library. First, the MM sends one request with the name ranges of each tutor as a message. Because of the file size limit, the MM sends one request at a time for each zip file. Additionally, when sending a file, the webhook does not accept the standard `application/json` content type but requires `multipart/form-data` content type.

```
const data = new FormData()
data.append("username", "SSW␣Submissions")
data.append("avatar_url", avatarUrl)
data.append("file", zips[filename], filename)
```

Listing 3: Data for the webhook

```
fetch(webhook, {
    method: 'POST',
    body: data
})
```

Listing 4: Example fetch request

Listing 3 packs the zip files into form data objects. In Listing 4, the form data is sent to the specified webhook. This code gets executed in a loop for every submission zip to send all files in the corresponding channel to the tutors.

3.4 Website

After reading about the core functionality of the MM in Sections 3.2 and 3.3, this Section will describe the internals of the web interface. The reason for this module is to provide a web interface that enables lecturers to

- provide credentials for Moodle,
- set up tutor lists for courses,
- provide Discord webhooks,
- turn distribution of homework on and off,

in a web interface for lecturers. The usage of these functionalities is detailed in Section 4.

The WebService module uses the *express* and *crypto* libraries as node modules. The express library is used to run an express server in the background with a few endpoints, which are callable with button clicks in the UI. Furthermore, cookies are generated and decrypted again by this module.

3.4.1 Cookie management

The MM must not be accessible by everyone. That is why only logged in users can access the website. For remembering whether the user is logged in, the MM sends a cookie to the user on login, which the browser then stores. Once the user clicks on *Log out*, the cookie gets deleted, and further requests fail.

To generate a cookie, the MM encrypts the username and password, as well as the current timestamp. Furthermore, the *configuration file* of the MM contains a customizable *encryption salt*. First, the MM writes the salt, the username, the password, and the timestamp into a single string delimited by newlines. Then, the MM generates a random initialization vector, and the string is encrypted with the crypto library using the algorithm *aes-256-cbc*, with a further secret from the configuration file. The encrypted cookie is then returned as a base64 in the format `initializationVector:cipherText`.

The initialization vector is split from the encrypted text to decrypt the cookie again. After decrypting the cookie, the values are split into fields again, and the MM performs a few checks.

If the cookie contains four values, and the salt is correct, then the cookie is considered valid and returned as such for further processing. The MM does not perform a password check because a valid cookie is only issued if the `MoodleManagementAPI` deems some credentials valid.

3.4.2 Endpoint design

Express provides a `setRoutes` method, where one has to define endpoints. Listing 5 shows the login as an example of an endpoint definition. When clicking the *login* button in the login form, this endpoint is called. This form contains a field for username and password. When clicked, this form sends these fields as `application/x-www-form-urlencoded` data in the request body.

```
this.express.post('/login', async (req, res) => {
    const username = req.body.username as string
    const password = req.body.password as string

    //cookie validation ...

    res.cookie("auth", this.generateCookie(username, password, Date.now()))
    res.redirect('/dashboard')
})
```

Listing 5: Login endpoint in express

We can extract resources such as username and password from the `req.body` field. After successful validation of the provided credentials, the MM generates a cookie with these credentials, as shown in Section 3.4.1. The MM sends this cookie with the name `auth` as a redirect to the `/dashboard` page to the user. A redirect is an HTTP response type that tells the browser to perform a new request to the provided resource. The dashboard is the main page of the MM.

Another example with other functionalities can be found in Listing 6. The UI calls this endpoint automatically once it requires the rendered tutors of a course. In contrast to the endpoint in Listing 5, a POST request, the following is a GET request. We can specify the request type by calling the corresponding method in express. The remaining behavior we can specify in a callback similar to the `/login` endpoint.

```
this.express.get('/tutors/:courseID', async (req, res) => {
    const cookie = this.decryptCookie(req.cookies.auth as string)

    if(cookie.valid) {
        const courseID = req.params.courseID
        const tutors = //get tutors
        res.render('tutors.ejs', { tutors, courseID })
    } else {
        res.render('login')
    }
})
```

Listing 6: Tutor list endpoint in express

Furthermore, we can specify URL parameters in the endpoint by prepending a colon to the wanted field. In this case, `/tutors/:courseID` specifies that any request with `/tutor` followed by a course ID will be matched. In the example, the cookie is decrypted, and depending on its validity, different actions are taken. When the cookie is valid, the MM extracts the course ID from `req.params` and render a corresponding HTML according to the Embedded JavaScript (EJS) template. More on that in Section 3.4.3. When the cookie is invalid, the MM prompts the user again with the login page.

3.4.3 Rendering HTML pages

We can configure express to render EJS files as HTML. Listing 7 demonstrates such a page part. In the example, we pass a `tutors` array and a `courseID` to the engine. In the engine, we can insert JavaScript code with specific tags such as `<% code %>` and `<%= variable %>`. The first tag allows to insert code, while the second tag injects variables into the resulting HTML.

```
<% for(const tutor of tutors) { %>
    <tr>
        <td><%=tutor.name%></td>
        <td><%=tutor.weight%></td>
        <td><button onclick="...">Delete</button></td>
    </tr>
<% } %>
```

Listing 7: Rendering tutors with EJS

This extract of the tutor rendering visualizes how we can generate HTML by iterating the `tutors` array. For every tutor, a table row is generated with a name cell, a weight cell, and a delete button cell to delete the tutor from the MM.

3.5 Database and Logging

The last chapter of the implementation section explains how the `Database` works and how we implement the logging. In Section 3.5.3, we present a short logging sample in a *select* method.

3.5.1 Database

The MM must store important data in a file. While the course information could be fetched from Moodle whenever needed, the tutors and credentials would need to be stored anyway. Furthermore, accessing information from a database is faster than requesting it from the network.

The `Database` module contains all code related to the database. For that, all necessary database queries are wrapped into methods, which the `MoodleManagementAPI` then accesses. To work with SQLite in Typescript, an SQLite string has to be passed to and executed by the `Database` module from the *sqlite3* library. For example a simplified table for the tutors can be created with `db.exec("CREATE TABLE tutor(name TEXT,weight INT)")`. Furthermore, non-nullability is achievable with the `NOT NULL` modifier, primary and foreign keys can be specified, and table creation can be ignored if the table already exists.

The run method has to be used in SQLite to insert, update, and delete elements from the database. The selection of elements has to be done with `db.get(...)` to retrieve a single element or `db.all(...)` to retrieve all rows from a query. Queries are the usual SQL syntax, with tuple notation to sanitize input. For example `SELECT * FROM course WHERE (courseID) = (?)` would select all `Courses` with the passed `courseID`.

3.5.2 Logging

For logging, we wrapped the *pino* library into a new class. It supports four different levels, i. e., *info*, *warn*, *error* and *debug*. Each log level has an annotation in the shape of a symbol, which is prepended to the message to demonstrate the log level further. Every day, the MM creates a new file in the specified directory, to which the `Logger` writes.

3.5.3 Logging in the Database

Listings 8 and 9 demonstrate the usage of the `Logger` using a method from the `Database`. The `select<T> method` reduces code duplication. As an example, the `getCourseByID` method from Listing 8 will be used to demonstrate the `select<T>` method.

The method first initializes the selection clause in SQLite syntax as a string, and then uses the select method, which for now only returns the course with the wanted id. As the method returns a `Promise<Course>` we await the value before returning it and select the first element, because the ids in the system are unique.

The parameters for the select method are the query string, and the required values for the query as well as a string which will be displayed in the log. At first, we call the `Logger` to log the course query from the database, with the passed log message, which customizes the log to include the required data. As already mentioned the query string and values are required to return the actual value. If the database query throws an error, the `Logger` gets called again to tell the administrator that retrieval of the `Course` was not possible.

```
public async getCourseByID(courseID: string): Promise<Course> {
  const query = "SELECT␣*␣FROM␣course␣WHERE␣(courseID)␣=␣(?)"
  return (await this.select<Course>(query, [courseID], ‘course␣${courseID}‘))[0]
}
```

Listing 8: Calling the select function

```
private logger = Logger.instance
private select<T>(query: string, values: any[], log: string): Promise<T[]> {
  logger.debug(‘Getting␣${log}␣from␣database‘)
  return new Promise((resolve, reject) => {
    this.database.all<T>(query, values, (err, rows) => {
      if(err) {
        logger.error(‘Could␣not␣retrieve␣${log}␣from␣databaseS␣${err.message}‘)
        reject(err)
      }
      resolve(rows)
    })
  })
}
```

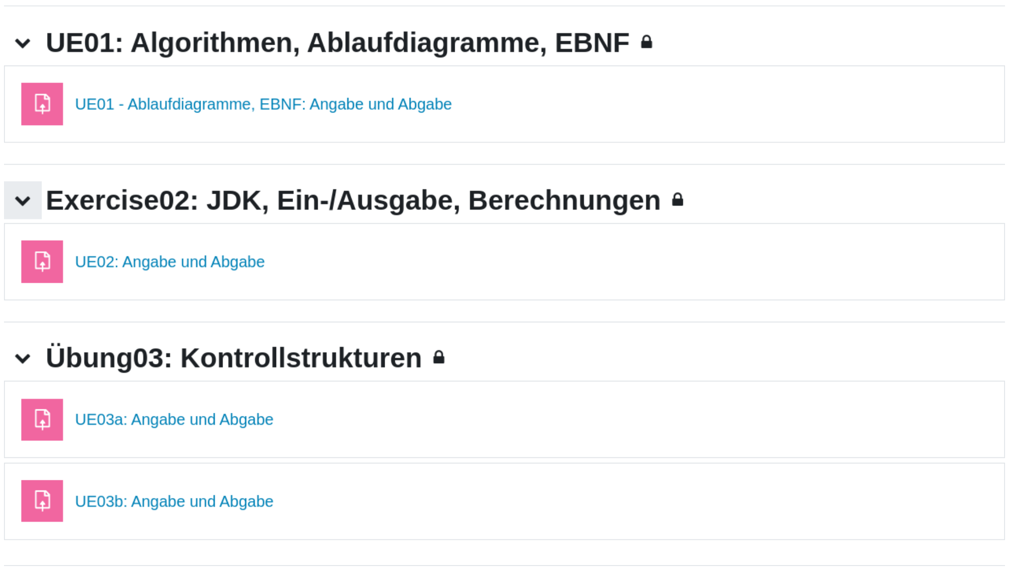
Listing 9: Logging in the Database module

4 Usage

In this section, we explain the usage of the MM. In section 4.1, the requirements for the MM are elucidated. Section 4.2 will cover the correct installation of the MM. The setup of the MM is explained in section 4.3. Finally, the correct UI usage of the MM is explained step by step in section 4.4. An important note is that access to the private GitHub project [4] is necessary in order to get access to the MM.

4.1 Moodle Requirements

Before installing the MM, lecturers must configure all Moodle courses correctly that the MM should manage. In order to parse the assignments from Moodle, as described in section 3.2.4, the sections have to be set up correctly. The MM can only find an exercise if there is a heading that starts either with "Übung" or "UE". Important note is that if there are multiple assignments in one chapter, then the MM will take the first of them.



UE01: Algorithmen, Ablaufdiagramme, EBNF

UE01 - Ablaufdiagramme, EBNF: Angabe und Abgabe

Exercise02: JDK, Ein-/Ausgabe, Berechnungen

UE02: Angabe und Abgabe

Übung03: Kontrollstrukturen

UE03a: Angabe und Abgabe

UE03b: Angabe und Abgabe

Figure 4: Example of course headings

In Figure 4, three exercises are shown. In this scenario, all of them are set up differently regarding their headings and the amount of exercises per section. A heading is a title with an arrow on the left, where users can hide the section. *UE01* would be detected correctly because the heading starts with "UE". On the other hand, *Exercise02* can not be detected because the heading neither starts with "UE" nor with "Übung". For *Übung03* only *UE03a* will get registered in the MM because the heading starts with "Übung", but the MM is only capable of detecting one exercise per section. The MM expects the keywords in the title to be followed by a number, which the MM takes into account for enumerating the assignments in the system. This mapping is the reason why only one exercise per section gets detected.

4.2 Installation

The recommended way to setup the MM is to use Docker [3]. Hence, docker needs to be installed by following the steps on `www.docker.com/products/docker-desktop/`. Once Docker is installed, the next step is to clone the GitHub repository into the file system using the following command:

```
git clone https://github.com/rechen-werk/Moodle-Manager.git
```

Once the repository is on the system, we need to change the current working directory to the location of the MM. When the preconditions are met, executing `docker build -t moodle-manager .` (note the dot in the end) builds the MM. Docker now has an image of the MM, which we can start in Docker Desktop or in the command line. It is important to provide a port for the UI and a folder for the MM, where it can store the files. In the command line, this works as follows:

```
docker run
    --name moodle-manager                               //1
    -p [desired external port]:8080                     //2
    -v [/path/to/directory]:/home/ssw/.moodle-manager   //3
    moodle-manager                                      //4
```

The lines are only delimitered, in order to explain the parameters. The first parameter sets the name of the container. The parameter `-p` specifies the port that the docker container uses. Providing an arbitrary port makes the web interface accessible on this exact port. The third parameter, `-v`, specifies the directory on the system, which the MM uses. Here, we can find the log files, the configuration file, etc. Line 4 specifies which image should run, and this is the image built in the last step.

4.3 Setup

Once we start the Docker container, the MM is theoretically usable. In practice, nothing will happen because, by default, no one has access to the MM. The admin can change this by adding users to the whitelist. In order to make this possible, the MM creates its files in the specified location, which then can be edited by the administrator. One of these files is the configuration file `config.json`, which looks similar to listing 10.

```json
{
  "whitelist": [],
  "webservice": {
    "encryption_key": "...",
    "encryption_salt": "Heimdall"
  },
  "distributionDelay": 180
}
```

Listing 10: config.json

The `encryption_key` should be a random string with 32 characters. The `encryption_salt` may be changed to increase security further. The MM verifies that the received cookies to contain this salt in the first line. The `distributionDelay` can be customized to delay the distribution by the specified minutes. The idea is to grant students a grace period to submit assignments after the deadline. The most important setting is the `whitelist`. Here, the admin can specify lecturers who may use the MM. The whitelist is a list of strings, representing the staff numbers of the lecturers that are allowed to use the MM. An important note is that changes in the `webservice` object will only take effect upon restart of the container by performing the command `docker restart moodle-manager`. The other fields will take effect during run time.

4.4 User Interface

When the MM is set up by following the sections 4.1 to 4.3, the first step is opening the UI at the specified port in the browser.

4.4.1 Login

Figure 5 shows a login prompt when accessing the MM in the browser. It is necessary to enter the Moodle credentials (i. e. staff number and password) to log in. When logging in for the first time, the courses are loaded. Thus, the first login might take some time. The assignments of the current semester are loaded immediately, while the courses of earlier semesters are loaded asynchronously, which means they might not be visible immediately.

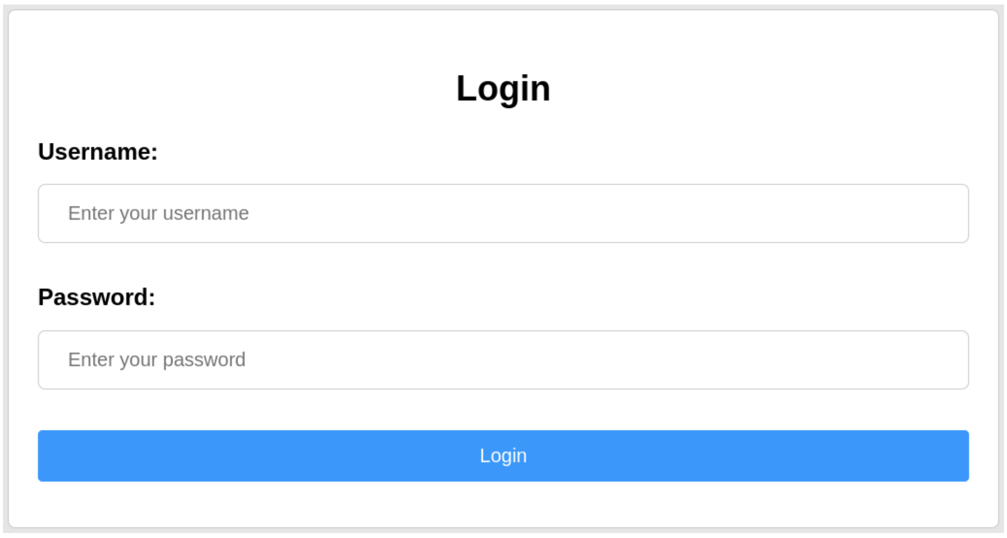


Figure 5: Login page of the Moodle Manager

4.4.2 Dashboard

Once logged in, the dashboard greets the user with their staff number below the *Moodle Manager* caption. Figure 6 shows the main components of the dashboard. Element (1) is a logout button, which deletes the authentication cookie and leads the user back to the login page when clicked.

The filter at element ② is currently set to the *current* semester. Further settings are to show *all* courses available, and each semester is selectable with this filter. When a view has only one

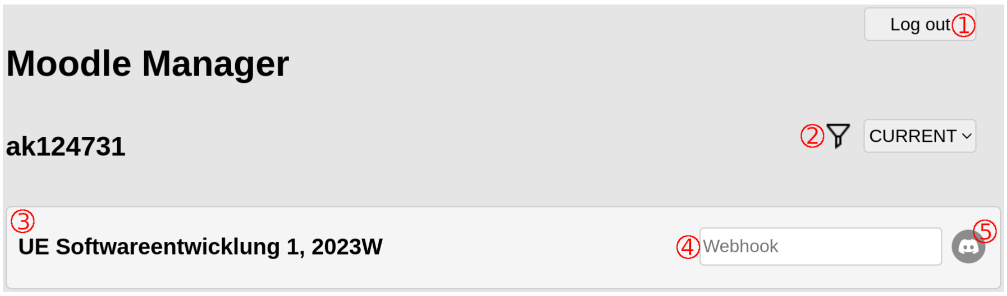
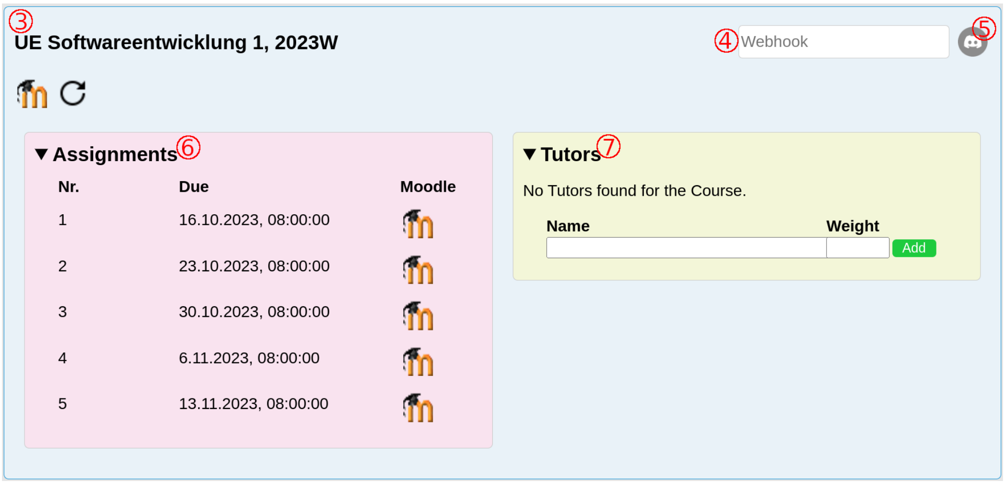


Figure 6: First look at the dashboard

course, then the course is extended by default. Otherwise, it is collapsed like in Figure 6. We can click element ③ to expand the course to the extent as visible in Figure 7. Element ④, this is a text field where the Discord webhook for the course can be pasted in. An important note is that users may only paste the webhook (and not type it character by character), because the MM checks it on every event, and if the provided value is not a valid webhook, the field is erased. Button ⑤, the toggle button for enabling the Moodle management feature for the course, will be explained in section 4.4.3. Elements ⑥ and ⑦ can be expanded and collapsed similarly to element ③. Element ⑥ is a list of the assignments of this course. The view either shows the list, such as in Figure 7, or denotes "No assignments found for this course.". In element ⑦ tutors can be added and removed. Similarly to the assignments, when no tutor exists, the MM tells the user that no tutors exist for the course.



Nr.	Due	Moodle
1	16.10.2023, 08:00:00	
2	23.10.2023, 08:00:00	
3	30.10.2023, 08:00:00	
4	6.11.2023, 08:00:00	
5	13.11.2023, 08:00:00	

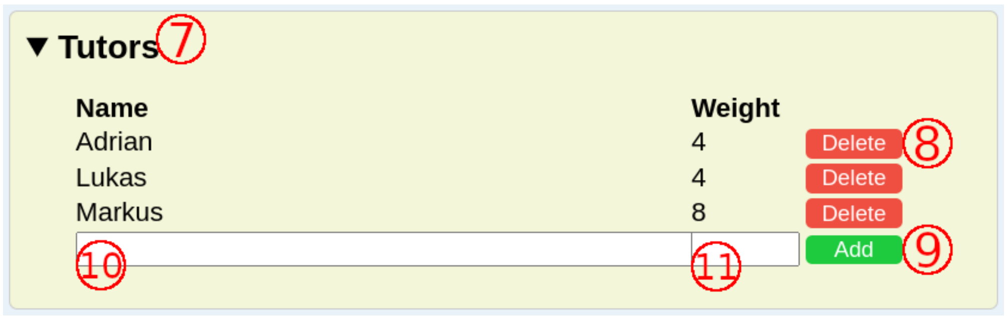
No Tutors found for the Course.

Name	Weight

Figure 7: Extended course in the dashboard

In contrast to the assignments, though, lecturers can manage tutors in the panel. In Figure 8, already three tutors are registered for the course. The delete buttons, indicated with ⑧, allow deletion of the tutor in the corresponding row. Deleting will immediately affect the database without warning but can be corrected by adding the tutor again.

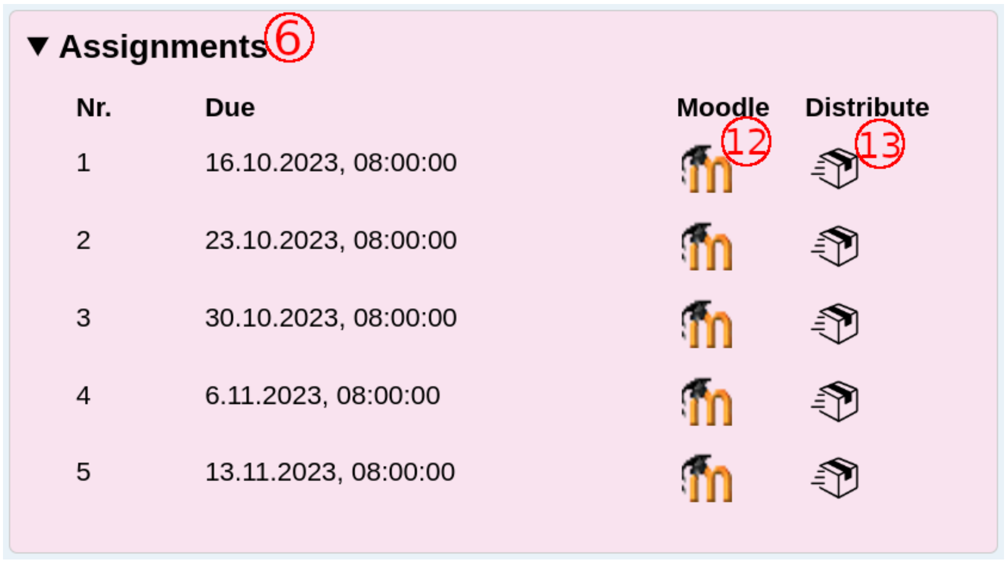
Adding can be done by providing a name for the tutor in field ⑩ and a weight in field ⑪. To confirm the input, clicking element ⑨, the add button, will send the data to the MM. When a lecturer provides a name already in the tutor list, then the weight of the tutor is only adjusted, and the MM creates no further tutor.



Name	Weight	
Adrian	4	Delete
Lukas	4	Delete
Markus	8	Delete
		Add

Figure 8: The tutor panel for a course

The assignment panel is a list of assignments for the selected course. It displays the assignments sorted by number. A table row renders the assignment number, the due date, a button ⑫ which leads to the Moodle page of the assignment, and a button ⑬ which can be used to manually trigger the assignment distribution. Furthermore, this distribution button is only displayed when the distribution feature is activated for this course as explained in Section 4.4.3.



Nr.	Due	Moodle	Distribute
1	16.10.2023, 08:00:00		
2	23.10.2023, 08:00:00		
3	30.10.2023, 08:00:00		
4	6.11.2023, 08:00:00		
5	13.11.2023, 08:00:00		

Figure 9: The assignment panel for a course

4.4.3 Enabling the Moodle-Management feature

The entire MM builds up on distributing assignments automatically. Element ⑤ in Figures 6 and 7 shows a Discord icon which is grey. This element indicates the status of the automatic distribution for this course. Figure 10 shows a state diagram of the different distribution states.

A grey icon means that the preconditions for Discord distribution have not been met for this course. A red icon tells the user that Discord distribution is possible, but disabled for this course. Lastly, the green icon means that Discord distribution is enabled for this course.

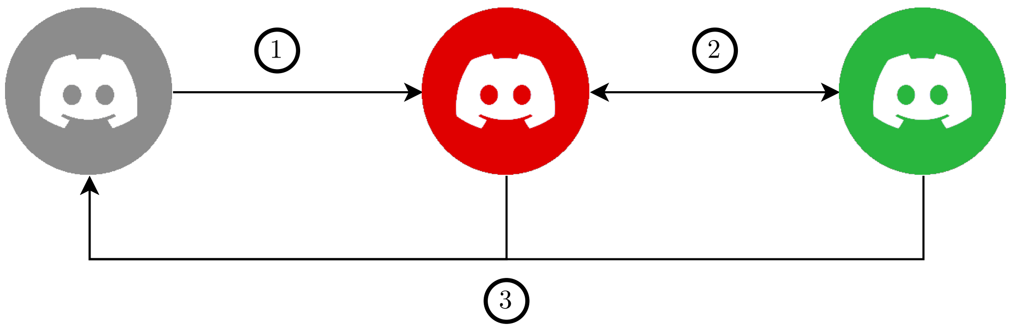


Figure 10: Discord distribution state diagram

The diagram furthermore has five possible transitions. Transition ① is from grey to red and happens when the preconditions for Discord distributions are met. I. e. there is at least one tutor in the list (⑦) for this course, and a lecturer provided a valid Discord webhook (in field ④) for this course. ② denotes two transitions between red and green, which are the enabling and disabling of the Discord distribution by a button click on the icon. Finally, ③ denotes two transitions again from both red and green to grey, which happens when a user removes the webhook or if the tutor list gets emptied. A transition from grey to green is impossible because Discord distribution has, by design, to be enabled manually.

5 Lessons Learned

One of my discoveries while working on this thesis is the TypeScript language. Although I had a better time working with this language than working with JavaScript, I do not think that I will use it in a future project again. The random quirks of JavaScript are still unbearable.

My joyous discovery is the tool ChatGPT, which I did hear about but did not facilitate earlier. My supervisor told me that using this AI is a reasonable thing to do, so I created an account for it and started coding.

5.1 ChatGPT Driven Coding

ChatGPT is a text-based AI that answers every question thrown at it. Since I was quite a novice at TypeScript, my questions initially related to language concepts. First, I asked how to write classes and make them accessible from other files, which is not quite intuitive if you never worked with TypeScript. Later, questions were about callbacks and Promises. ChatGPT suggested all frameworks used in this thesis since I could ask them questions on the framework when I could not find anything in the documentation. Furthermore, ChatGPT provides a list of similar frameworks and compares them for you, which makes the choice easier. This support makes life easier if one is new to a language. This holds true especially for TypeScript/JavaScript, where grasping all the available frameworks and libraries is the real challenge.

In conclusion, I am astonished at how ChatGPT provides good points to start coding. It provides examples of how to use some frameworks, which often need to be corrected. It is essential to understand the code provided to correct these mistakes. The user can point out the corrections to ChatGPT, which corrects the mistakes for you.

6 Future Work

The first version of the MM has only the functionality to distribute assignments automatically. For further versions, additional features are possible. Especially the feature in section 6.1, the visualization of feedback, is of interest, as it can help the lecturers get better insights into the feedback of students.

6.1 Feedback visualization

The lecturers at SSW are very interested in the feedback of their students. That is why they provide a field for feedback in Moodle for every homework. Moodle has some rather unusable visualization tools. The idea is to download the feedback data and visualize it in a new course view in the MM. The lecturers at SSW can adapt these visualizations fully to their needs. Additionally, students may be listed and compared against others by the lecturers. We could also predict how many (and which) students may fail the course with the following homework based on statistics.

6.2 Warnings to students

Another idea is to warn students about their submissions. The MM could download the submissions regularly and do simple checks, such as a check for an empty submission. Furthermore, the MM could ensure correct file structure. Even a warning via E-Mail before the submission deadline is an idea.

The problem with this idea is that this could result in spam to students. Students need to be able to unsubscribe from these warnings using an button in the E-Mail. Furthermore, the return on investment for this feature is questionable.

6.3 Tutor view

One last idea is to remove Discord from the stack, making the MM more independent and versatile. The main reason why this would be a good idea is because Discord only allows file sizes up to 25MiB, as explained in section 3.3.2. Having a page for the tutors to download (and upload) the splittings would make the distribution easier. Furthermore, Discord has yet to get an uptime of 100 percent.

7 Conclusion

The idea for the Moodle Manager came from lecturers which lost too much time with small repetitive tasks. One of these tasks was to download the submissions from Moodle, split them according to their tutors working hours and distribute them via Discord. This weekly task during the semester added further workload to the lecturers while they had other important tasks to do.

To achieve an automation of this task, Moodle has been thoroughly analyzed to parse and download data from there. Now the Moodle Manager downloads the assignment submissions once the deadline is due. Once the assignment submissions are downloaded, they are automatically split into smaller batches according to the the list of tutors for this course. The split submissions then are sent in to the according channel on discord where the tutors can download them. All this automation is configurable in a Web User Interface by the lecturers. There lecturers can set the Discord webhook for the correct channel and define a list of tutors. Furthermore they can enable and disable the automatic distribution by the Moodle Manager per course.

The Moodle Manager lifted the burden of downloading, splitting and distributing the assignment submissions to the tutors every week from the lecturers. Therefore, the lecturers can use the time saved for other tasks.

API	Application Programming Interface
CSS	Cascading Style Sheets
cURL	"Client for URLs"
EJS	Embedded JavaScript
JKU	Johannes Kepler University
MM	Moodle Manager
SSW	Institute for System Software
UI	User Interface
URL	Uniform Resource Locator
HTML	Hypertext Markup Language

List of Tables

1 Sample tutors 8
2 Calculation of the submission count for the tutors 9
3 The splittings of the tutors 9

List of Figures

1 Top level architecture of the Moodle Manager 3
2 Data flow when logging in to Moodle 5
3 A snippet of the grades page in Moodle 7
4 Example of course headings 14
5 Login page of the Moodle Manager 16
6 First look at the dashboard 17
7 Extended course in the dashboard 17
8 The tutor panel for a course 18
9 The assignment panel for a course 18
10 Discord distribution state diagram 19

Listings

1 Function to create curl post commands with optional URL encoded data 5
2 HTML of a course 6
3 Data for the webhook 10
4 Example fetch request 10
5 Login endpoint in express 11
6 Tutor list endpoint in express 11
7 Rendering tutors with EJS 12
8 Calling the select function 13
9 Logging in the Database module 13
10 config.json 15

References

[1] Discord webhook documentation. Available at `https://discord.com/developers/docs/resources/webhook`.

[2] Discord webhook documentation. Available at `https://discord.com/developers/docs/resources/webhook#execute-webhook`.

[3] docker.com. Available at `https://www.docker.com/`.

[4] Moodle manager on github. Available at `https://github.com/rechen-werk/Moodle-Manager`.

[5] Sqlite. Available at `https://www.sqlite.org/index.html`.