

Master Project

March 17, 2020

# ACCESS

Assistive Course Creation and Evaluation of  
Student Submissions Tool

**Alexander Hofmann,** 11-916-863  
**Philip Hofmann,** 14-710-842  
**Mervin Cheok,** 07-194-392  
**Luka Lapanashvili** 13-934-062

**supervised by**

Prof. Dr. Harald C. Gall  
Dr.-Ing. Sebastian Proksch



University of  
Zurich<sup>UZH</sup>



software evolution & architecture lab



Master Project

---

# ACCESS

Assistive Course Creation and Evaluation of  
Student Submissions Tool

**Alexander Hofmann,** 11-916-863  
**Philip Hofmann,** 14-710-842  
**Mervin Cheok,** 07-194-392  
**Luka Lapanashvili** 13-934-062



University of  
Zurich<sup>UZH</sup>



**Master Project****Author:**

Philip Hofmann, 14-710-842

Mervin Cheok, 07-194-392

Luka Lapanashvili 13-934-062

, {alexander.hofmann, philip.hofmann, mervin.cheok, luka.lapanashvili}@uzh.ch

**Project period:**

Software Evolution & Architecture Lab

Department of Informatics, University of Zurich

Alexander Hofmann, 11-916-863

18.03.2019 - 17.03.2020

---

# Abstract

ACCESS is an auto-grading tool developed at the University of Zurich (UZH) as part of a master's project. The developers had been tasked with finding a solution to the scalability problem of the *Informatics 1* course. This course received a continual increase in student registrations over the years and is expected to continue to grow in participants' size. The registered students receive weekly coding exercises. In the previous year, 12 Tutors had been hired, which corrected each week around 30 to 35 student submissions per tutor, which led to long waiting times for the students to receive feedback on their submissions.

This practice is not scalable, and the mentioned problem will only increase in magnitude. This paper presents the solution that had been designed, implemented, and consequently been used for the *Informatics 1* course at university, using ACCESS as a scalable solution in regards to registered students. The elicited requirements, chosen approach, and related design decisions and challenges are laid out in the following pages, concluding with remarks and suggestions on future features of this system.



---

# Zusammenfassung

ACCESS wurde im Auftrag der Universität Zürich (UZH) entwickelt und implementiert, um den teilnehmenden Studenten des 'Informatik 1' Kurses eine Onlineplattform für die Bearbeitung und die automatische Bewertung ihrer Programmieraufgaben zu bieten. Bis anhin wurden diese Programmierabgaben der Studenten von Tutoren von Hand korrigiert. Letztes Jahr wurden 12 Tutoren angestellt, welche jede Woche 30 bis 35 Abgaben korrigierten, was für die Studenten zu langen Wartezeiten bezüglich des Feedbacks führte.

Diese Arbeit beschreibt ein System, welches die Entwickler als Lösung zu dem dargestellten Problem konzipiert hatten. Es werden die Erarbeitung der Anforderungen an ein solches System, die Vorgehensweise, erarbeitete Konzepte und mögliche zukünftige Funktionen auf den folgenden Seiten dargelegt. Dieses System wurde anschliessend ein Semester lang von 450 Studenten benutzt, welche erfolgreich ihre wöchentlichen Programmieraufgaben durchführten.





---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Chapter Overview . . . . .	1
1.2	Thesis Context . . . . .	1
<b>2</b>	<b>Related Work</b>	<b>3</b>
2.1	Autograding Platforms . . . . .	3
2.1.1	OKpy . . . . .	3
2.1.2	Code Expert . . . . .	3
2.2	Massive Open Online Courses (MOOCs) . . . . .	3
2.2.1	Khan Academy . . . . .	4
2.2.2	edX . . . . .	4
2.2.3	Exercism . . . . .	4
2.2.4	Conclusion . . . . .	5
<b>3</b>	<b>Requirements</b>	<b>7</b>
3.1	Requirements Elicitation . . . . .	7
3.2	Functional Requirements . . . . .	8
3.3	Non-Functional Requirements . . . . .	10
<b>4</b>	<b>Approach</b>	<b>13</b>
4.1	Agile Methodology . . . . .	13
4.2	Project plan . . . . .	13
<b>5</b>	<b>Design</b>	<b>15</b>
5.1	System Context . . . . .	15
5.2	ACCESS Container Diagram . . . . .	16
5.3	Identity Provider . . . . .	18
5.3.1	Authentication . . . . .	19
5.3.2	Authorization . . . . .	19
5.3.3	User Onboarding and Course Enrollment . . . . .	19
5.4	Frontend . . . . .	20
5.4.1	Architecture . . . . .	21
5.4.2	Visual Design . . . . .	22
5.5	Backend . . . . .	24
5.5.1	Course Component . . . . .	25
5.5.2	Submission Component . . . . .	27
5.6	Code Runner . . . . .	30

---

<b>6</b>	<b>Challenges</b>	<b>31</b>
6.1	File System as Database . . . . .	31
6.1.1	File Indexing . . . . .	31
6.1.2	Change Detection . . . . .	31
6.1.3	Virtual Files . . . . .	32
6.2	Submission Versioning . . . . .	33
6.3	Security . . . . .	34
6.3.1	Remote Code Execution . . . . .	35
6.3.2	Denial of Service: Infinite Loops & RAM Usage . . . . .	35
6.3.3	Grading Test Suite Leakage . . . . .	36
6.4	Onboarding - User without UZH Email Address . . . . .	37
<b>7</b>	<b>Evaluation</b>	<b>39</b>
7.1	Decreased Workload . . . . .	39
7.2	Performance Testing . . . . .	40
7.3	User Testing . . . . .	40
7.4	Usability Testing . . . . .	41
<b>8</b>	<b>Conclusions</b>	<b>43</b>
<b>9</b>	<b>Future Work</b>	<b>45</b>
<b>A</b>	<b>Appendinx</b>	<b>47</b>
A.1	Feature Table . . . . .	48

# Introduction

## 1.1 Chapter Overview

This is a report on the design and implementation of the ACCESS tool. This Chapter provides an overview of the context and motivation for this work.

Chapter 2 displays our findings on tools that address a similar problem domain and our findings concerning our requirements.

Chapter 3 presents the requirements that we gathered concerning this tool, specifically in the context of a solution of the aforementioned *Informatics 1* course problem.

Chapter 4 briefly explains our approach to this project.

Chapter 5 contains content on the concepts and designs for this system that we worked out.

In Chapter 6, we list some specific challenges we solved in more detail as these might be of particular interest to the reader.

Chapter 7 contains information on how we tested the usability and performance of this system.

Furthermore, Chapter 8 consists of our conclusions of this project.

## 1.2 Thesis Context

**Problem** The Department of Informatics at the University of Zürich (UZH) observed a rapid increase in students for the introductory course *Informatics 1* over the last years. This lecture involves weekly exercises for each student that get manually corrected by tutors that the department hires each semester. In 2018, the department hired 12 tutors to cope with the increasing demands, which resulted in a workload of 30 to 35 exercises to be corrected by each tutor per week. This system of tutors manually correcting the exercises is not scalable, and there are no signs that the number of participants of this course will stabilize nor decrease.

This is not an isolated case; in fact, many universities have been facing the same problem<sup>12</sup>, many of them using software to grade student exercises automatically. The majority of these systems provide a web interface for course authoring. We make the case that this might result in a sub-par experience for course administrators.

**Solution** In this thesis, we envision an automated system as a solution to the mentioned problem, which automatically corrects and grades the student exercises for this specific university-lecture, while allowing course authors to edit content with their tools of choice. Furthermore,

---

<sup>1</sup><https://www.csail.mit.edu/news/auto-grader>

<sup>2</sup><https://www.cs.cmu.edu/link/meeting-demand>

each edit done by the content author will be versioned for transparency. The system allows tutors to switch attention from correcting programming exercises to assisting the students. In contrast to the current situation, the system is scalable and could easily handle a further increase in student numbers. Furthermore, our envisioned system could provide the students with instant feedback, thus enhancing the learning experience. In this thesis, we come up with a design and a system that could solve the mentioned problems. We subsequently got hired by the university to deploy and put in operation our prototype for the semester, while maintaining and updating and extending the live-in-production running prototype with several features needed by the course authors of the *Informatics 1* lecture to adapt the system to the requirements of their respective exercises. The system had then been used by those 450 students to solve their weekly exercises, and we received mostly positive feedback concerning our system.

# Related Work

## 2.1 Autograding Platforms

Autograding systems are increasingly used to grade code submissions automatically [19]. This is mostly being done by running the respective submitted code against internal unit tests. Those unit tests are specific to a particular programming language and tailored towards the objectives of a distinct assignment. The purpose is to provide a system which evaluates if the subject is understood and if the objectives of the assignment are reached. Those systems are being used on educational platforms, as demonstrated in this section.

### 2.1.1 OKpy

OKpy [10] is an automated grading system. It was developed at UC Berkeley for the introductory programming course and is similar to our tool. OKpy consists of a server, a client, and an autograder component. The server collects submissions from the clients; it is deployed with Flask and uses `virtualenv` to create isolated python environments. The client is needed to submit to OK Server and for the user to test their solution locally. The server side autograder executes the grading code in a sandboxed container on a remote server.

### 2.1.2 Code Expert

To enable scalable programming education, ETH Zürich developed *Code Expert* [3], an online integrated development environment (IDE) that allows students to work on assignments. The system provides immediate feedback to students by running the code against a test suite or a static analyzer. Students can submit assignments and projects that can either be automatically graded as well as be manually reviewed by tutors. Course maintainers can edit exercises directly online on the web application. The tool is in use since 2018 and has been used in various courses for exercises as well as exams. Unfortunately, it is offered exclusively to ETH members.

## 2.2 Massive Open Online Courses (MOOCs)

MOOCs are accessible online courses designed with a large number of participants in mind. These are often maintained or created by large institutions like companies or universities. These courses

educate the learner on a specific topic and can be enhanced by videos, exercises, discussion forums, and interaction with the course teachers or other learners through chat rooms. Some platforms also allow the user to receive a certificate of completion once the course has been completed.

A compact comparison of the checked MOOCs is in the attachment A.1 Feature Table.

### 2.2.1 Khan Academy

Khan Academy [8] offers free online education for everyone. The platform covers topics from Math, Computing, Arts and Humanities to Economics and Finance. Started in 2008 as one person tutoring, Khan Academy has grown to a nonprofit organization engaging over 150 persons.

The Khan Academy platform structures its content into a field of interests > course > lesson > chapter. A chapter presents information in text or video form and asks a user to do related exercises. Exercises can vary in scope and can be rated or unrated. More extensive exercises are presented in a guided step by step fashion, allowing the user to proceed only if the current question is answered correctly. The platform provides feedback if the user struggles to answer a question correctly. Other features provided by the platform are: gamification, exercise commenting, and the support for individual coaching.

Khan Academy allows creating custom courses by selecting and combining lessons of different courses. The platform does not allow us to create and upload custom content. The provided courses cover most basic knowledge up to high school level [8].

### 2.2.2 edX

edX [6] is a non-profit organization that provides a widely used learning platform. The main software of the platform uses Open edX. This open-source software provides all core components of a learning platform, including student delivering course-ware and course authoring components. It was founded and is maintained by Harvard and MIT Universities. The platform provides courses in a variety of topics ranging from Computer science, Languages, Data Science, Business and Management, Engineering, and Humanities. Ninety partner universities and other facilities create the courses. According to the latest impact report, edX has 24 million unique users and provides over 3000 courses.

Out of the box, edX supports checkbox, drop-down, multiple-choice, numerical and text input type problems. Although programming tasks are possible to create, the platform is mainly built for online learning lectures and therefore has no dedicated programming task interface. Courses use Codeboard [4], py4e [11] or other external editors for programming tasks and rely on the functionality that those editors provide.

### 2.2.3 Exercism

Exercism [7] is an online open-source platform and not-for-profit organization aimed to help people with previous programming experience who want to learn a new language or deepen their knowledge. It provides exercises in various programming languages and a mentoring system in which volunteers review each submission and give personalized feedback. In order to progress through a language track, a mentor needs to approve the student's submission. Exercises can be downloaded and submitted using Exercism's CLI tool and solved locally using any editor or IDE. The platform in itself does not teach any programming concepts but is rather a collection of exercise combined with a mentoring system. It has a community approach that allows students to publish their solutions and comment on other people's submissions. Additionally, exercises

are not executed or graded online. Due to its approach and target- audience, it is not beginner-friendly.

From the course maintainer's perspective, Exercism takes a similar approach to ACCESS, each course is stored in a git repository, and it defines its own exercises configuration using a metadata file.

## 2.2.4 Conclusion

Of all the tools we found, Okpy was the closest to fulfilling all requirements; unfortunately, it requires that the actual assignments are stored separately, which made our traceability requirements hard to implement (see Chapter 3). On OKpy, tutors are required to leave manually comments on each submission, which means that tutors still have to correct each submission. Additionally, it relies on a separate custom-made CLI client to test and submit solutions, which also has to be provided to the students together with each exercise template, which might be confusing for first-semester students.

From our findings reviewing similar software, we designed a custom solution based around the course *Informatics 1*. Our target audience are first-semester students, with little to no programming experience. We aimed to provide a simple environment that students could use to solve exercises. Thus, ACCESS provides an online platform on which students can solve programming exercises, without having to worry about setting up a development environment. By designing an auto-grading mechanism, the number of students can increase without having to hire more tutors.

Additionally, ACCESS provides other types of exercises, such as single-/multiple-choice and free-text, in order to support courses outside of computer science. Finally, to increase the learning effect, ACCESS also provides an automatic hint-system for students that struggle with the exercises.





# Requirements

This chapter describes the main requirements of ACCESS and the applied approach for the requirements engineering process. The most vital requirements for ACCESS are:

- Online provisioning of coding exercises
- Automated evaluation and grading of programming exercises
- Immediate feedback and assistance for failing exercises
- Support of the existing *Informatics 1* course exercise structure and workflow
- Traceability of exercises
- Handling the workload for 500 students

## 3.1 Requirements Elicitation

The acquisition of the ACCESS requirements used the requirements elicitation process as described in Zowghi and Coulin 2005 [28]. This process uses a set of activities such as communication, collaboration, prioritization and negotiation with relevant stakeholders. These activities are further categorized into five types: *(i)* understanding the application domain, *(ii)* identifying the sources of requirements, *(iii)* analyzing the stakeholders, *(iv)* selecting the techniques and tools to use, *(v)* eliciting the requirements from stakeholder and other sources [28]. This section describes the five types and the used activities in the ACCESS project.

**Understanding the application domain** It is essential to analyze the real environment with the existing processes in which the system will be used. The gained information helps to understand critical points and existing problems [28].

This work uses a survey of MOOC's as a first step to understanding the application domain. The survey provides insights about operational models, how lecture material is presented, what question types are used, and how coding exercises are supported. The next step is the analysis of the previous *Informatics 1* course held in 2018. The focus is set on screening the processes and interaction of the course between course authorities, teaching assistants, and the students, as well as the used Python exercises with the corresponding test suites. The third step combined the previously gained knowledge to answer how and which of the MOOC's provided functionality supports the *Informatics 1* course processes.

**Identify the sources of requirements** The primary source for the ACCESS requirements are the stakeholders involved in the *Informatics 1* course, especially the roles of product owner, teaching assistant, and students. Further requirements sources are the analysis of existing MOOC's and learning platforms and the use of IDE's. UZH regulations are requirement sources as well. Most requirements had to be written down after speaking to stakeholders. Only a few requirements came from existing documents.

Person/Group	Role
Prof. Dr. Harald C. Gall	Client
Dr.-Ing. Sebastian Proksch	Product owner
ACCESS development team	Developers
Teaching assistants year 2018 & 2019	Power users
<i>Informatics 1</i> participants 2019	End users

**Table 3.1:** ACCESS Stakeholders

**Analyzing the stakeholders** Stakeholders are people that are involved or affected by the system [28]. Table 3.1 lists the stakeholders identified for ACCESS. Prof. Dr. Harald Gall is the contractee and sponsor of this project. Dr. Sebastian Proksch is responsible for the *Informatics 1* course operations and holds the role of the ACCESS product owner. Former and current *Informatics 1* teaching assistants and the first-year students that have to participate in the *Informatics 1* course are identified as users.

**Selecting the techniques and tools to use** This project makes use of multiple requirement elicitation techniques. To understand the application domain, the project uses task and domain analysis to get an overview of the essential functionalities of a MOOC and to gain insight into the existing *Informatics 1* course procedures. Strong collaboration and frequent brainstorming sessions with the product owner and other stakeholders helped to explore and refine requirements.

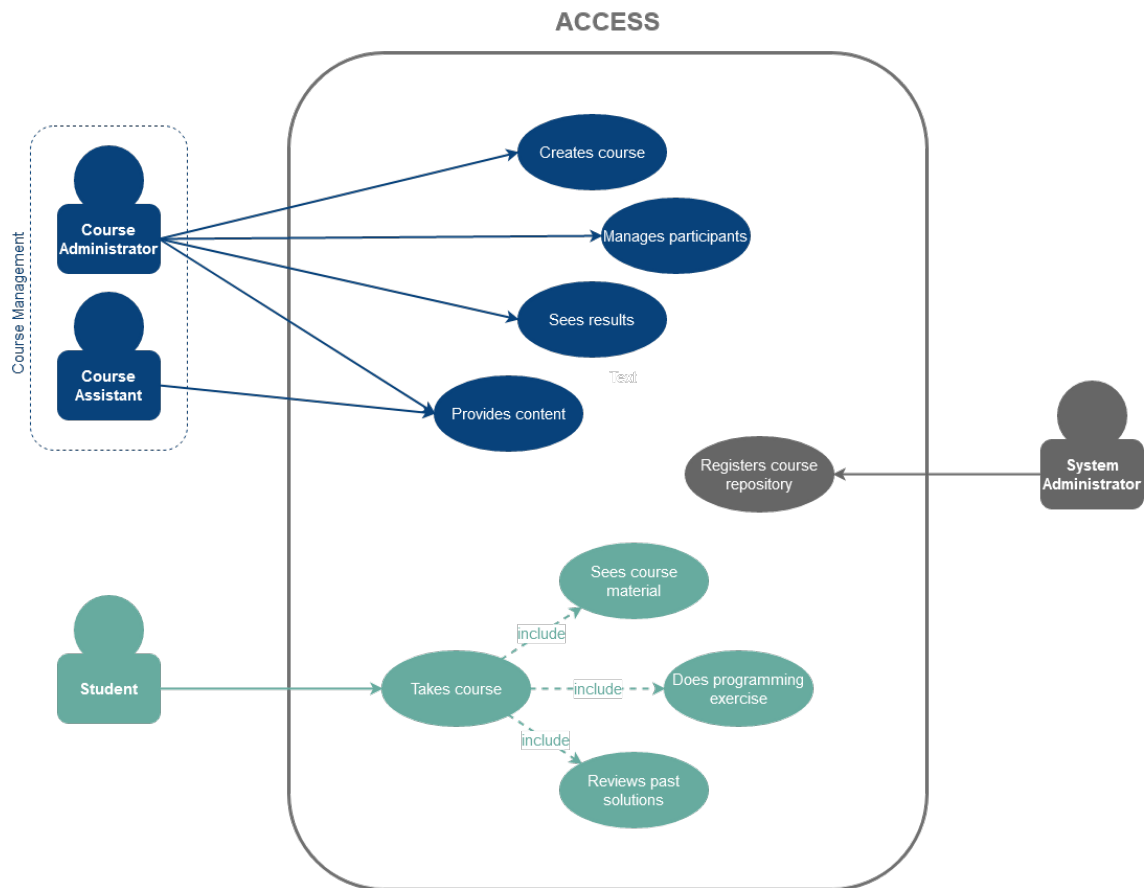
This project uses GitHub project management and collaboration tools. Project members write gathered information into the GitHub wiki and make the information available to everyone, to foster shared understanding in the team. We manage agreed Goals, use cases, and tasks in GitHub's issue tracker.

**Eliciting the requirements form stakeholder and other sources** The requirements elicitation process continues over the full project time. Although more time for elicitation is invested at the beginning of the project, to acquire a scope and the base requirements, requirements can be introduced, changed, re-prioritized, or canceled during the full project span. The fixed short intervals of the project status meeting are used to check the validity of the current requirements.

## 3.2 Functional Requirements

This section describes the main functional requirements. The functionalities are arranged in the three areas of responsibility: course management, student and system administration. The systems actors course admin, course assistant, student and system administrator are assigned to one

of these areas. Figure 3.1 shows the UML use case diagram visualizing the involved actors, their area of responsibility and the use cases.



**Figure 3.1:** ACCESS Use Case Diagram: Provides an overview of the main functionality of the system and shows the three areas of responsibility: course management in blue, student in green and system administration in gray.

All system users have to authenticate themselves and get their assigned permissions that unlock the different functionalities of the system.

**Course Management** This area addresses the task of administrating (create, run, finish and close) a course as well as the creation of lecture and exercise materials.

The course admin actor can import a student list from OLAT into the system, which triggers an automatic course enrollment for all of these students. After a course, the administrator can export a student list with all the results to all exercises. This list must be imported back into OLAT. A course admin can further check the classes exercises result. He can see an overview of all exercises for all student and their results. Besides, the course admin can also see the submissions of the students.

A course assistant actor is responsible for the course content management. His responsibility

focuses on providing the lecture material and exercises. For the *Informatics 1* course, this means describing the exercise task in Markdown and creating Python programming and ‘git’ exercises and the corresponding unit test suite used for the evaluation. To support this task, the teaching assistant can use his favorite tooling e.g. IDE, and registers the exercise using git.

To describe a task, the system must support different media types: image, video, audio. A course and its tasks can have publication and due dates that regulate the visibility and access type to the exercises. A course admin or assistant can then choose between different answer types:

- **Programming:** A bigger code programming exercise containing multiple files and folder structures
- **Code snippet:** A simple code exercise that fits in a single file
- **Text:** Free text
- **Single/multiple choice:** The answers to choose from are given

A course admin or assistant can also provide hints that are shown to the students if they struggle to solve an exercise correctly. Further, a course admin or assistant can register the solution to exercises, which is then available for the students after the exercises due date has passed.

**Student** A student can log in to the system and can see an overview of his/her registered courses. Opening a course leads to an overview of assignments, where an assignment contains none to multiple exercises. An exercise presents a task description as well as a suitable answer type input possibility. The assignment and exercise overview shows the progress of a student and the gained points accordingly.

A student can do the programming exercise online or can choose to download an exercise to work offline on his computer and favored IDE. The exercise submission is done over the web page and returns an immediate score. The system allows, depending on the configuration, to submit an exercise multiple times.

If a student encounters difficulties solving an exercise correctly, the system provides situational hints helping with the current problem. A hint offers limited guidance without revealing the solution so that a student can autonomously overcome the problem.

**System Administrator** The system admin registers new courses in the system. If needed, he provides a new git repository for a new course and links the repository with the system.

A complete catalog of the elicited functional requirements is accessible in the ACCESS GitHub issue tracker (<https://github.com/mp-access/Backend/issues>).

## 3.3 Non-Functional Requirements

This section lists the non-functional requirements for the ACCESS system.

**Technology Stack** The used technologies for frontend and backend have to be matured and widespread. They have to be mature to ensure a stable and secure productive operation. They have to be widespread and accepted by software engineers to make it easy to find suitable developers to guarantee further development and support after the project. The tool and language characteristics have to provide the functionality to address the ACCESS problem domain.

The technologies have to be approved by the product owner.

**Extendability and Configurability** The system architecture allows growth subsequently in multiple dimensions, preferably by configuration. Possible extension examples are:

- Additional courses: other than an introduction to python programming class
- Additional question and answer types
- Additional programming languages
- Additional evaluation tools: for example static code analysis

**Data Sovereignty** UZH regulations prohibit storing student-related data such as personal information or test results on external cloud providers. Thus, ACCESS has to run on-premise on the infrastructure of the Institute for Informatics (IfI).

**Identity and Access Management** Future versions of the system will be integrated into the UZH identity and access management infrastructure. ACCESS must work with external identity providers (IdP's).

**Scalability and Availability** The system handles different usage scenarios. (i) During the semester there will be sporadically requests by students doing their homework (ii) with peaks of increased request shortly before homework due dates. (iii) During an exam, all registered students will be using the system simultaneously.

ACCESS must be able to handle submissions for at least 500 students. Sporadic downtime during the daily semester routine is manageable. However, during an exam, the system must function so that students can hand-in their exercises.

**Due Date** The implementation and evaluation of this project have to be finished until the end of August 2019. This proof of concept will be the basis used for the *Informatics 1* course system used in the autumn semester 2019.



# Approach

This project uses an agile approach with incremental short-termed sprints.

## 4.1 Agile Methodology

The initial situation encountered looks as following: The product owner has a clear goal and vision in mind; detailed information, however, is not defined or changes continuously. The product owner requests strong collaboration and regular planning meetings. All involved persons are at the same institute, and collocated work is possible.

This scenario suggests an agile project methodology. Agile software development methodologies focus on delivering small software deliverables continuously, where every delivery contributes to customer value. Short increments allow for early requirements and design verification. Every increment starts with requirement prioritization and selecting work packages for implementation. An increment ends with testing to verify that the implementation meets the requirements [25].

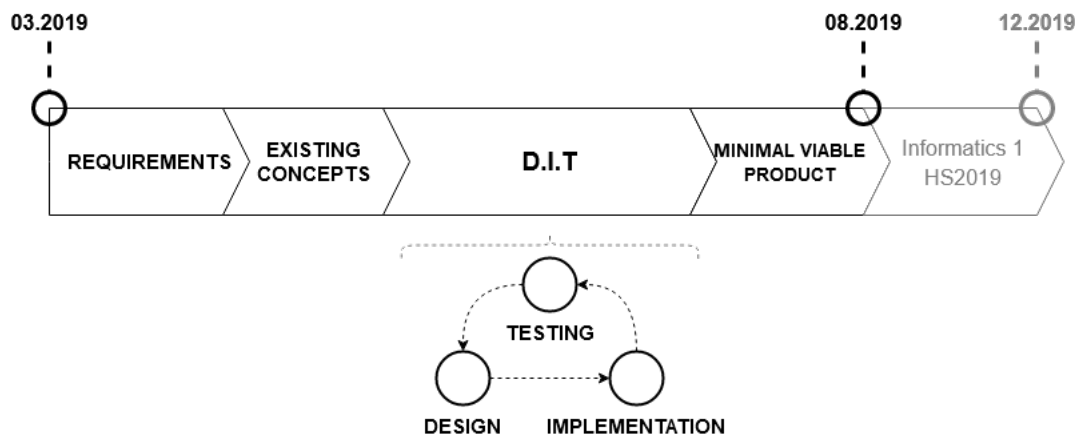
## 4.2 Project plan

The project starts in March 2019 and has to provide a minimal viable product (MVP) at the end of August 2019 before the start of the Autumn semester 2019. The MVP provides the base for the productive ACCESS system used in the *Informatics 1* lecture.

The project uses weekly sprints with one sprint meeting every Wednesday. Figure 4.1 Project plan shows the schedule and the thematic focus of the sprints.

**Requirements Engineering** The project starts with a requirement engineering phase. As mentioned before in Chapter 3.1 Requirements Elicitation, no complete requirements catalog exists. The requirements need to be identified gathered and prioritized.

**Existing Concepts Research** After the main requirements are defined, and a shared understanding is achieved, the focus changes to the question of what it takes to implement the requirements. Domain-specific systems are analyzed, and experiments with related technical concepts are made to gain the experience to decide on a feasible roadmap and a possible design solution.



**Figure 4.1:** Project plan

**Design, Implementation, and Testing** With a high-level design concept at hand, the design, implementation, and testing phase tackles one issue after another. For every issue, a detailed design concept and acceptance criteria are defined. The implemented concept is then tested and verified by the acceptance criteria.

**Minimal Viable Product** The last phase concentrates on preparing the implemented solution for productive usage and verifying that the implementation meets the main requirements. The deployment and configuration for the target infrastructure get set up. The setup undergoes load and performance tests as well as usability tests. The MVP has to prove that it is capable of being deployed in the productive *Informatics 1* lecture.

**HS2019** The feature implementation for the system continues during the Autumn 2019 semester. This activity is not part of the master project scope.



# Design

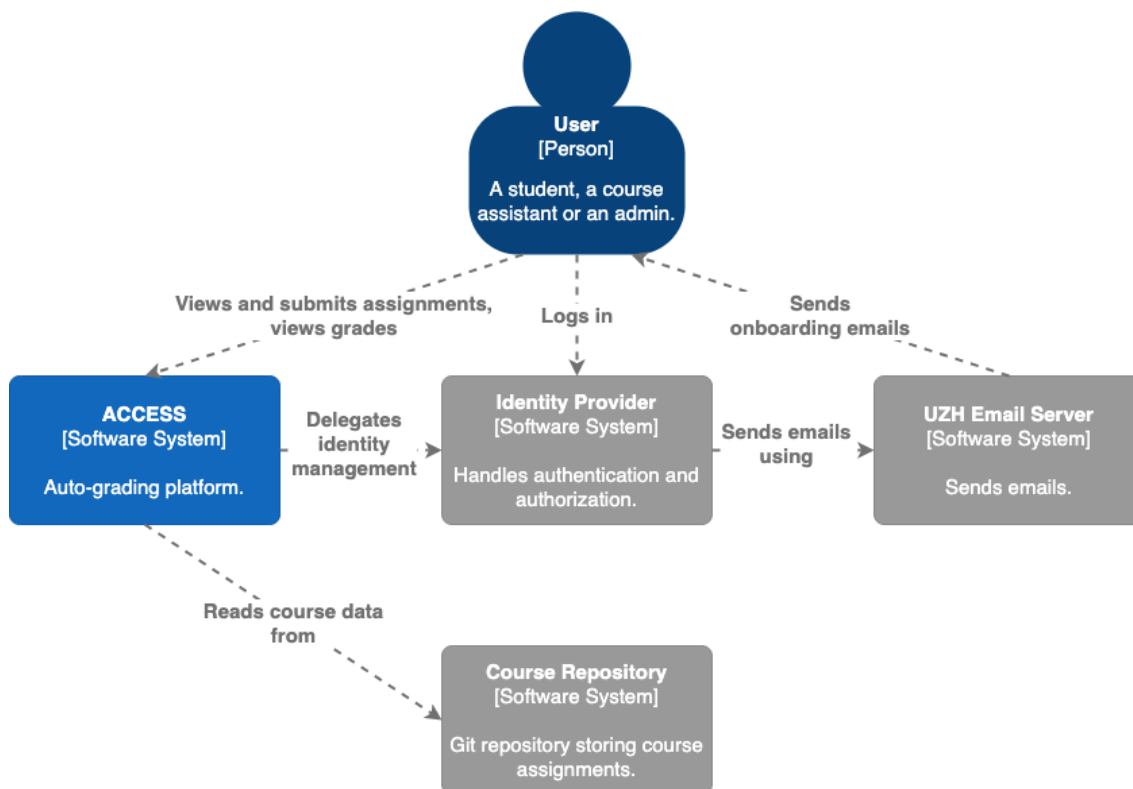
This section presents the design considerations, that went into the implementation of ACCESS. First, the underlying architecture is described, such as how the application is divided in its components and how these interact. Afterwards, we will look at how each component was designed, and its architecture. For more details, Chapter 6 presents further challenges in the design. We follow a top-down approach in our explanation, each section expanding more in detail following the C4 model [17]. Thus, this chapter is divided as follows:

- In Section 5.1, the context in which ACCESS is situated is presented and with whom and how ACCESS communicates with external services.
- In Section 5.2, the component of ACCESS are discussed.
- Section 5.3 presents how ACCESS handles the identity management, user onboarding, authentication and access control.
- In Section 5.4, the frontend and the various features provided to the users are presented.
- Section 5.5 presents how the backend is designed and its sub-components, in particular, how we represent courses, and how a student code submission is stored, executed and eventually graded.
- In Section 5.6 the actual environment in which student code is executed is described.

## 5.1 System Context

ACCESS, seen in blue in Figure 5.1, communicates with a number of external services. All course assignments and course participants are stored on external Git repositories. At the moment the system supports GitHub and GitLab. ACCESS downloads the contents on startup or upon receiving a signed request to a specific endpoint. Based on the list of participants, ACCESS provisions user accounts for each participant and enrolls existing accounts in a course user group by calling the identity provider's API. Authentication and authorization is delegated to the external identity provider. The identity provider is further connected to the UZH email server in order send registration emails to students.

In the remainder of this chapter each subsystem is described in detail.



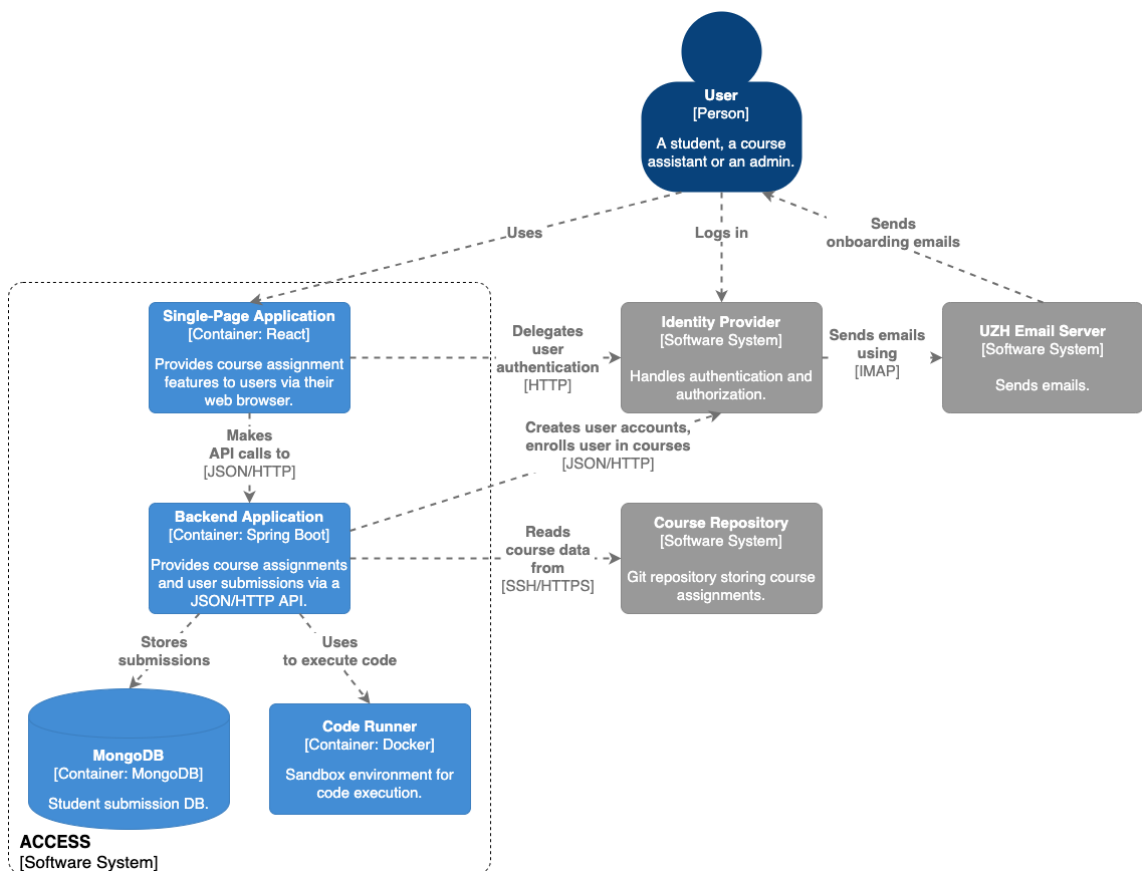
**Figure 5.1: ACCESS System Context diagram:** ACCESS (highlighted in blue) reads course assignments and configuration from a Git repository. It communicates with an external Identity Provider (IdP) which handles authentication, authorization and user account management. The IdP sends emails through the UZH email server.

## 5.2 ACCESS Container Diagram

ACCESS can be further split into four components:

- Single-Page Application (SPA)
- Backend Application
- MongoDB database
- Code Runner

**Single-Page Application (SPA)** The SPA is the only component which the user interacts with in ACCESS. It provides a way for users to view submissions as well as edit and submit exercise solutions. At a high level it is composed of a react SPA served from a reverse proxy which forwards all requests to the rest of ACCESS. Thus, the SPA acts as a single point of entry. This has the advantage of reducing the apparent complexity of the system, while also reducing the possible attack surface, as all traffic has to go through the reverse proxy. If the user has not yet logged in,



**Figure 5.2:** ACCESS container diagram: The Single-Page Application (SPA) is the single point of entry for the user and consumes the API provided by the backend, where the business logic is implemented. The backend stores submissions on a MongoDB database, and all code is executed on a separate Code Runner machine.

the SPA will redirect the user to the IdP for authentication. After successful user authentication, the SPA will receive a JSON Web Token (JWT) which can be used to make further API calls to the backend.

**Backend** The backend serves all the assignments and user submissions to the frontend, which simply presents them to the user. Furthermore the backend implements the business logic to store, execute submissions, and grade them. The Spring (Boot) Framework provides the base for this backend. Spring supports the construction and configuration of enterprise applications and allows developers to focus on the implementation of business logic [13].

All requests to the backend have to be authenticated, using the JWT.

**MongoDB** All student submissions are stored on a database. For this project, a NoSQL database, MongoDB, was chosen as the nature of the information stored fits in nicely with the concept of document stores: all submission metadata is stored in a non-normalized format together with the submission itself, whenever a submission is retrieved, all associated information is also needed.

If this were stored in a classical relational database, this data would have to be normalized in multiple tables, so for each read operation multiple joins would be required. Alternatively, all information could be embedded in a single table row, which would defeat the purpose of a RDB. A NoSQL database is better suited for the kind of hierarchical data: a code submission is a natural aggregate, spreading the information stored on multiple tables just for the sake of it, would only serve to increase the impedance mismatch [20].

In practice however, a classic SQL database could also have been chosen, and the way the persistence layer was implemented, means that it should be possible to replace the NoSQL DB with an SQL one with minimal effort, if needed.

**Code Runner** All user submissions are executed on a sandboxed environment which runs on a separate machine. This is done for 3 reasons:

- *Resource starvation*: In order to guarantee the availability of the server, all long running and cpu/memory intensive operations are done on a separate machine.
- *Scalability*: In order to allow more users to execute code, it is possible to scale the Code Runner horizontally and vertically. It is possible to increase the resources on the machine, as well as to add multiple Code Runner nodes. In addition, the distributed design makes it easier to scale the runners and backend servers separately.
- *Security*: The instructions flow strictly from the backend to the Code Runner. In case a user is able to escape the sandbox and run a process on the host machine, the integrity of the server and the submission store is guaranteed, since the runner has no way to initiate a communication with, or start a process on the backend server. In addition, in case a user were to, accidentally or maliciously, crash the host where the sandboxes run, the backend would still be available, albeit without the possibility to run code.

## 5.3 Identity Provider

ACCESS delegates identity management to an external provider. This has three advantages over implementing the necessary functionality ourselves: (i) it was possible to rapidly add authentication and authorization into ACCESS allowing us to concentrate on the core business logic; (ii) many identity providers allow to simply configure additional sources of identity, *e.g.*, adding a SAML/Shibboleth [27] (SwitchAAI, the identity provider at UZH, uses Shibboleth) or additional OpenID Connect [26] provider for identity federation, which would make it possible in future to allow users to authenticate using their UZH credentials with low implementation effort; and (iii) identity management is complex and implementing our own mechanism in ACCESS would likely be more error prone and insecure than using an established system.

ACCESS is agnostic of the provider used, the only requirement being that (i) that it provides an API which can be used to create new user accounts and manage users groups and roles (further details on user management can be found in Section 5.5); and (ii) that it can be hosted at UZH, since the provider will hold student information, it was important that this information not leave the university. For development and deployment we chose to use Keycloak [24] which fulfills the above requirements, with the added benefit of being an open-source project, maintained by Red Hat, a reputable company with a proven track record.

### 5.3.1 Authentication

When accessing ACCESS the user is redirected to Keycloak for authentication. The identity provider generates a signed JSON Web Token [22] storing the username along with authorization information. The client sends the JWT with each request in the header to the protected backend API. The API has access to the public key of the IdP which allows it to verify that the claims stored on the token (authentication and authorizations) have not been tampered with or were forged.

### 5.3.2 Authorization

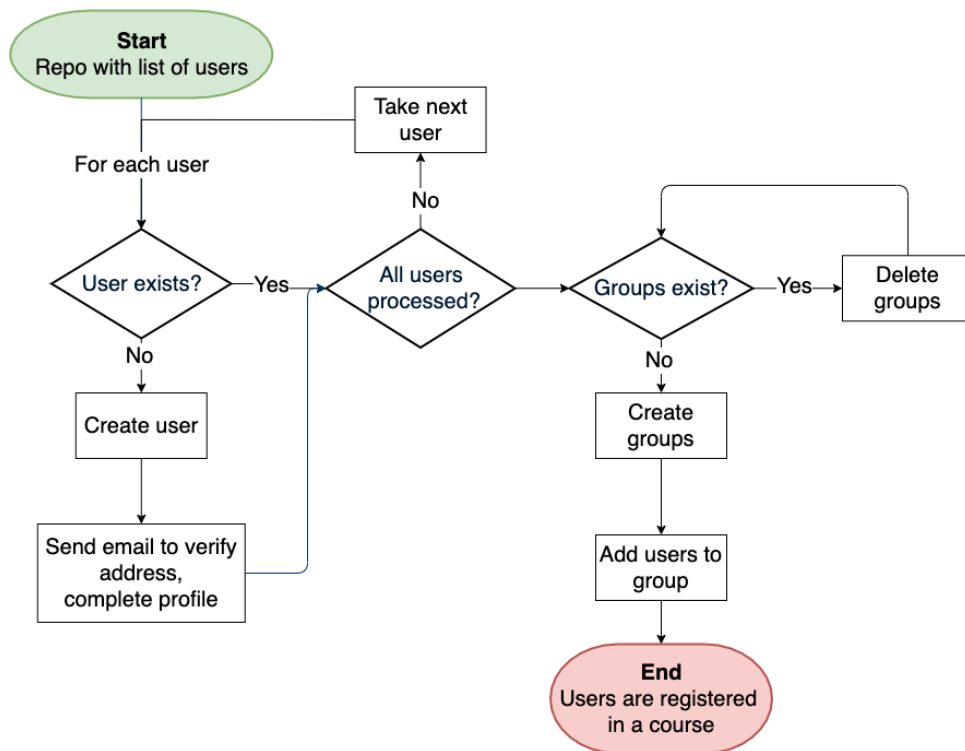
Users are organized around groups. A group is a collection of users with a given set of permissions assigned to the group (and transitively, to the users) and they are organized hierarchically: a group can be part of another group. A group is created for a course, and then for each course ACCESS defines three user groups: *Student*, *Assistant*, and *Admins*. Being part of a course-group grants various degrees of permissions related to the course depending on the subgroup.

A user can be assigned to one or multiple groups, within the same course, as well as being assigned to multiple courses. For example, within a single course a user can be assigned to one or multiple groups, e.g., *Student* and/or *Assistant*. Additionally, a user can have multiple group associations for various courses, for example a user can be a *student* in course A and an *assistant* in course B. This gives ACCESS a permission system with which it can model the various types of users which can be found in a university. Each group builds upon the permissions of the group beneath it:

- A *student* may view assignments once published and upload submissions within the configured window of time. A student may also view the solutions to an exercise, after its due date.
- An *assistant*, in addition to the permissions of a *student*, can view assignments and solutions before they are published for all users.
- And finally, an *admin* is an assistant which can also view submissions by any other course participants, as well as view and export the grades and trigger a manual re-grading of all submissions for an assignment.

### 5.3.3 User Onboarding and Course Enrollment

User enrollment is the process through which ACCESS provisions new accounts for new users, as well as, assigns existing accounts the correct permissions through group assignments. To ensure a single source of truth we decided to include the list of participants in the course repository together with the course's configuration. The process of user enrollment is represented in Figure 5.3 as a simple flowchart. When ACCESS parses a course configuration, it also reads a list of emails of course participants. There are three separate lists, one for each user type specified previously: *student*, *assistant*, and *admins*. The process for each group is the same. With the list of email addresses ACCESS asks the IdP, for each address, if an account already exists. If no account is found, ACCESS creates new accounts via the IdP's API. On account creation, Keycloak will send an email via the UZH email server, to ask the user to complete their profile. Once all accounts have been provisioned, a user group for the course and one subgroup for each user type is created. If a course group already exists, it is removed along with its subgroups and created anew. This is a simple approach to enroll users, without having to implement checks for: (i) "is the user already enrolled?"; and (ii) "has the user been removed from the participants list since the last update?".



**Figure 5.3:** Participant enrollment.

While the removal from the group could in theory cause a disruption for the user, in the case that the user sends a request in the precise moment that a group has been removed, in practice this entire process only lasts a few hundred milliseconds at best and should not be noticeable.

This process is repeated whenever a course is updated and on startup to ensure that only the correct participants have access to a course.

An additional challenge was how to handle cases in which a user receives a university email addresses only a few weeks after the start of the semester. For further details, refer to Chapter 6.

## 5.4 Frontend

The design-goals for the frontend were twofold. On one hand, a strict separation from the back-end through a clean API was necessary to allow for modularity, such that other frontend applications, which may follow in the future would be able to communicate with the backend given a proper authentication. Furthermore, it was essential to ensure clean visual design and low response-times on every action of the frontend, such that it would be feasible for students to work on their assignments inside the given framework, rather than using the tool as a solution upload terminal. In the following sections, we will discuss what technology stack was used as well as the rational behind the visual design.

### 5.4.1 Architecture

The frontend was built on top of ReactJS, which is a popular framework for creating frontend applications. Although uncommon, it is possible to build a frontend application without the use of an existing framework. This would however require manual management of state, routing and the implementation of a templating engine. Therefore it is common to use one of the three most popular frontend frameworks, such as React [12], Angular [2] or Vue [15]. For ACCESS it was decided to use React. Partly because of its popularity, robustness as it is developed and used by Facebook and finally due to our team members already having previous experience with the framework.

**Code Editor** The aim of the frontend of ACCESS was to deliver an integrated development environment which allows for online code authoring and execution. Here again an independent implementation or the native text area would have supplied the fundamental requirements of code authoring. However many developers are used to the ergonomic features of modern integrated development environments such as code completion, syntax highlighting and formatting. The vast collection of JavaScript libraries and frameworks offered again preexisting solutions. The three most popular code editors being: Monaco Editor [9], CodeMirror [5] and Ace [1]. Being actively developed by Microsoft and used inside Visual Studio Code, it was decided that the Monaco Editor was the best fit for our requirements.

**Media Viewer** To provide a flexible interface for the question authoring of a given exercise, it was decided to use Markdown as the underlying markup language. In combination with the react-markdown library, the frontend is able to parse and display markdown code. This ultimately allows the course author to do basic text formatting such as headings and lists as well as to include images, urls, code snippets and tables. Mathematical expressions can also be visualized with the utilization of the MathJax plugin.

Aside from displaying code and markdown, the course author is allowed to provide any kind of resource file. Commonly these files were complementary to the main programming exercise such as .txt and .csv files containing large data sets, image files containing diagrams and illustrations and .sh files containing shell code. To support a wide range of mime types, four general cases were introduced:

1. Markdown (.md)
2. Text based media such as plain-text and code (.txt, .py, .js, .c, .cpp, .html, ...)
3. Image types (.png, .jpg, .gif, .svg)
4. Unknown

For the first case, the previously mentioned library is used to parse and render markdown. For the second case, the Monaco editor is setup with the proper syntax highlighting and linting if available, presenting the file-contents as code/text. For the third case, an `img` tag is injected with the source URL of the image resource. Finally, if none of the previous cases applies to the file at hand, such is the case for .zip files, a message is displayed stating that the file format is not supported and an option for download is presented.

**Performance** Being an interface direct to the end-user, it is crucial for frontend applications to have low response times to ensure good user experience. This responsiveness can be achieved in two ways: first by ensuring that all interaction-points with the system are handled asynchronously whilst temporary status displays are immediately shown. Secondly by optimizing

the code to handle requests as fast as possible to lower waiting times.

We followed this principals by displaying a loading icon for every action that performs a request to the backend. Furthermore we immediately display empty component mocks where possible and asynchronously fill out the data as they arrive. This way the application can react on user-input at all times and feels responsive.

Special care had to be taken with the integration of the Monaco editor. During the development of ACCESS we encountered an issue with state handling, which lead to performance degradation. Due to the nature of react's state handling, a component will be redrawn, as soon as its state changes. The implementation of the Monaco editor required state changes on every keystroke, which lead to that change bubbling up to parent components, ultimately leading to redraws of many elements on every keystroke. This issue was remedied by selectively updating the Monaco state and ensuring that no unnecessary state bubbling occurred.

### 5.4.2 Visual Design

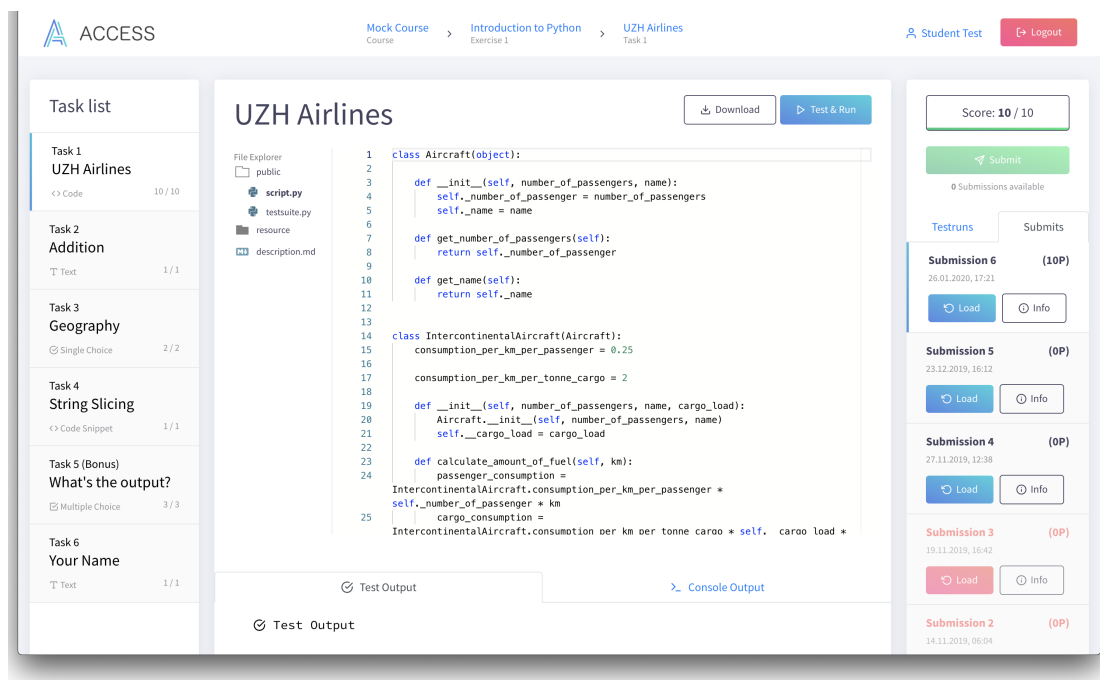
**Entry point** The frontend allows the user to log in with the account that had been created by the system by using the course creator's students list in the course config. After logging in, the user is then greeted with an overview of all courses that this account is connected to.

**Course selection** When choosing the course, the navigation from then on resembles the hierarchical structure of exercises that are common at universities, more specifically there are assignments that have to be done within a specific timeframe having a specified deadline, and each assignment consists of one or more tasks. The next view therefore shows to the user the assignments that have been published in the corresponding course. We also note here that there are variations on the view dependent on if the account has user or assistant or admin privileges. The normal user accounts will see the assignments after the corresponding publish date as defined in the config of this assignment, while privileged accounts will also be able to see and submit solutions to yet unpublished assignments. This distinction was particularly useful on our staging server, where assistants and tutors of the *Informatics 1* course would solve the new unpublished assignments in order to provide feedback to the content creator and to be better able to help students as they had previous understanding in solving this task. At the same time they were able to test new features before the release to the production server.

**Assignments Overview** The assignments overview includes the due-dates, where we had to resolve the challenge concerning local time zones because of users accessing the course with different system or browser configurations and time settings and different localities. When going into the assignment, the task view will be presented. This view is the most complex and consists the most components. We will shortly describe the components separately and included a screenshot for visualization purposes. Refer to Figure 5.4 for the following paragraphs about the tasks overview, where the user spends most time in our app by solving the tasks and utilizing the tools provided.

**Task Switching Panel** On the left side of the view we have the tasks-component. It simply lists, and therefore, provides a way to switch between the different tasks of the exercise. Additionally it displays the current points reached of the respective tasks which had been added later due to the user needing to actively navigate into each task to see the respective reached score. This component is the only one which content does not change when actively navigating the tasks, the remaining components both change when a different task is clicked.





**Figure 5.4:** The tasks overview

**Master Panel** The middle component includes multiple parts. The first part is a file tree explorer view, where the files that belong to this task can be navigated. Clicking on a different file will result in display changes, more specifically when `description.md` is clicked there will be a static presentation of the content of the markdown, generally used as a textual description into the task to be solved as guidance to the user.

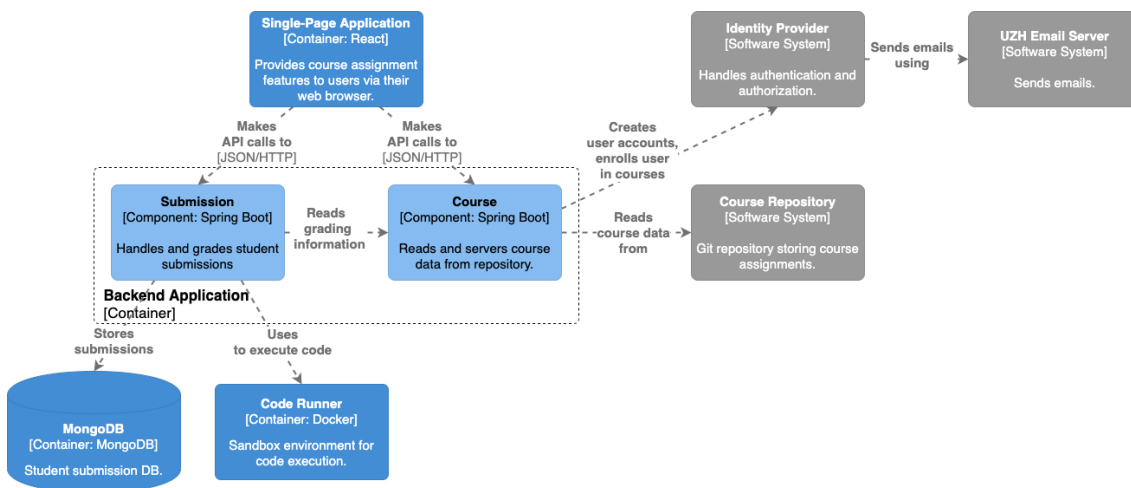
When a code file is clicked, the plugged in monaco editor will display the code and the user can then actively edit the code within the browser.

Above it are two buttons, one is to download the contents of the task to allow the user to solve the exercise on their local system. This empowers the user to use any IDE and locally test and run the code. Users can then simply paste the finished code into the frontend editor to submit it. The test&run button sends the code to our server, where we will spin up a new Docker container and run the code against the provided public test-suite. This is for the user to ensure the compatibility of the code with the grading test-suite and to store a version of the code on our servers before making a final submission. We also later on implemented the catching and handling of users pressing control+s to save the code, since in our interviews we had frequently observed this specific behavior which we believe stems from working on local files where one wants to save changes frequently.

Underneath we have two tabs, one is the console output that will be returned from running the code against the test-suite, namely it will display the user information on which unit-tests in the public test suite passed or failed. The second tab displays any standard output generated by executing the code like the console output in an IDE when using print statements.

**Submissions Panel** The right component displays the current score reached by the user for the respective task, together with a submit button for final submissions that will be executed against the private test suite used for grading, together with a display of the current number of remaining final submissions the user can execute. There are two tabs, the first one presents a versioned list of test-runs, which allows the user to restore specific save points of the code and also to view the generated console output with this version of the code. The second tab then displays a versioned list of final submissions together with the corresponding grade. The user can restore the code of a final submission. The info button, when clicked, simply displays a hint on what the user can improve in his code to reach a higher grade. This hint is of course provided by the course author and its usefulness depends on the externally provided content. Hints are extracted from the first failed unit tests assertion message.

## 5.5 Backend



**Figure 5.5:** ACCESS component diagram: inside the Backend Application, the *Submission* and *Course* components communicate to serve static course contents and student solutions to the Frontend Application. The *Course* component reads course data from the Git repositories as well as communicates with the IdP to provision accounts. The *Submission* component handles the storing, execution and autograding of student code, based on the grading information read from the *Course* component.

In the beginning it became immediately clear, that the backend would be composed of two main component: (i) one component to handle and serve the static contents parsed from the Git repositories, and (ii) one component to handle the user-generated contents, the student solutions to the exercises. For this reason, the backend application was designed as a monolith with two main subcomponents with clearly defined responsibilities, which can be seen in Figure 5.5 inside the Backend Application container. Eventually, the monolith could be broken up into two separate services, following a microservice pattern if needed.

## 5.5.1 Course Component

The *Course* component is responsible of fetching the static course contents from the Git repositories and then serving them to the user. Based on the participant lists stored in the course configuration, this component will provision new user accounts as needed and enroll users into their respective courses.

### Course Model

The ACCESS course as well as the Assignment and Exercise models are divided into two parts: configuration and metadata. This division is represented by the inheritance of, for example the `CourseConfig` and `Course` in the Figure 5.6. The configuration part of the model contains all relevant data that we extract from the `config.json` file created by the course author. This separation simplifies the extraction process, since the JSON can be directly deserialized into our Java class model. However, since additional metadata is required about the models such as the `id`, references to the child elements, such as assignments or exercises and other model specific data, an additional class was created, which inherits from the configuration class and encodes all additional attributes.

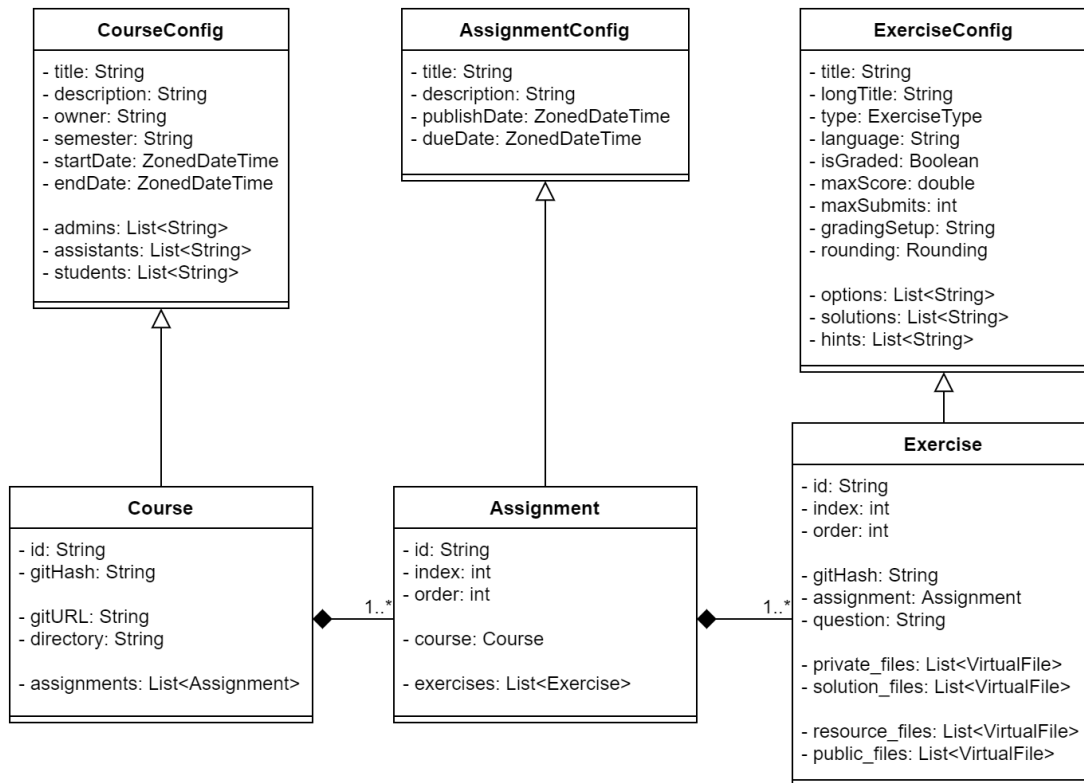
Choosing inheritance instead of composition may seem unintuitive at first glance, but the decision was made to simplify the API. With inheritance, all attributes can be directly called through the model object itself, whilst packing the configuration inside the model would hide certain attributes behind a `config` member. This arbitrary division seemed unintuitive, hence the decision to use inheritance in this case.

**Additional Model Data** Apart from the configuration parsed from the JSON configuration, the exercise model holds additional data-points such as lists to the private, public, solution and resource files as well as a question variable. This data is derived from the file-structure and therefore does not need to be explicitly encoded in the JSON. Two unusual data members that can be found inside the `Exercise` as well as the `Assignment` models are the `index` and `order`. The `order` value is parsed from the exercise, respectively assignment folder name. This order is a combination of a unique identifier and a logical ordering index. As will be discussed in chapter 6 concerning the challenges, this order helps to uniquely identify the assignment, respectively exercise between updates and also serves as a chronological ordering unit. On the other hand, the attribute `index` encodes the actual index that is displayed in the frontend starting at one increasing linearly.

### Functionality

The API exposes the course meta-model to the frontend through an HTTP API. Through the API it is only possible to view courses, assignments and exercises, but not edit them. To work on exercises a separate model and API is provided which is discussed in Section 5.5.2. All requests need to include authentication and authorization data stored in a JWT as explained in Section 5.3. Access control, the process of granting selective access to resources, is implemented via Aspect Oriented Programming (AOP) [23], to avoid mixing core business logic with authorization logic. For example in Listing 5.1, access is restricted through the `PreAuthorized` annotation provided by the framework, to check if the user is allowed to access a course before executing the main method logic. The `coursePermissionEvaluator` object, implements the authorization logic, and the `getCourseById` method can simply fetch the course.

```
1 @PreAuthorize("@coursePermissionEvaluator.hasAccessToCourse(authentication, #id)")
```



**Figure 5.6:** The ACCESS course model inherits from the `CourseConfig` class, which specifies all user defined configurations. The course model also contains a list of assignments, which internally contain a list of exercises

```

2 @GetMapping(path = "{id}")
3 public Course getCourseById(@PathVariable("id") String id) {
4     return courseService
5         .getCourseById(id)
6         .orElseThrow(ResourceNotFoundException::new);
7 }

```

**Listing 5.1:** Using AOP to implement access control: if the method `hasAccessToCourse` returns `false`, the API will return an HTTP 403 Forbidden code to the caller and the body of the method will not be executed.

The API exposes endpoints which are not supposed to be called by normal users, but rather from the Git repositories through webhooks [21]. The course repository admin can configure webhooks to trigger on any commit which will result in ACCESS pulling the remote repository and updating its meta-model accordingly. To avoid normal users triggering the update functionality, all webhooks need to be authenticated, either by sending a shared secret in the payload (Gitlab) or by computing a message authentication code on the JSON payload, via HMAC [16], again based on a shared secret (Github). Further details regarding course updates can be found in Chapter 6.

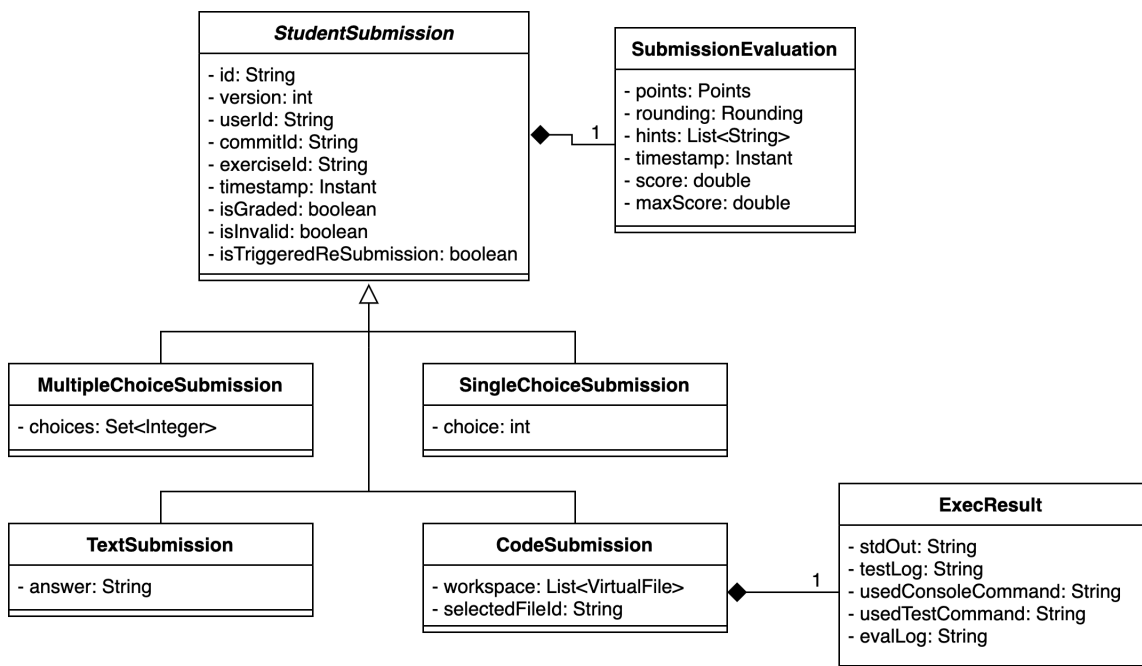
## IdP interface

Creating accounts, enrolling or removing user accounts into courses is a lot of manual, error-prone work, which needs to be done anytime a user is added or removed to a course. To mitigate this, we set the requirement that the IdP should provide an API which ACCESS can consume to automate this work. In addition to fulfilling this requirement, Keycloak also provides a java client library which was used to develop a small API, which is part of the *Course* component, to ease the manipulation of user accounts and groups. Using this subcomponent the processes described in 5.3 were implemented.

### 5.5.2 Submission Component

The *Submission Component* handles all matters regarding user submissions and their evaluation. Code submissions also need to be executed to be graded. In this case, the component also handles the orchestration of the necessary environment and the evaluation of the outputs. Thus, the *Submission Component* connects to both the database, to store submissions, as well as the *Code Runner*, described in Section 5.6, to execute user code. Additionally, it connects to the *Course Component* to fetch exercise points and solutions, and test suites needed to grade a student solution.

### Submission Model



**Figure 5.7:** Submission model. A submission holds a reference to its evaluation and for code types, its execution output. Each exercise type has a corresponding submission subtype.

ACCESS defines four main types of exercises: (i) code, (ii) free text, (iii) single- and (iv)

multiple-choice exercises. Metadata about the submission such as the timestamp, whether the submission is graded or was invalidated and versioning are common to all exercise types. For this reason the root of the model is the abstract `StudentSubmission` class. To add a new exercise type, a new implementation can be defined, only adding the fields necessary to hold the student answer. Usually, one of the main arguments against inheritance is the potential for subclass explosion, but we expect the types of exercises to remain mostly stable, making it a non-issue.

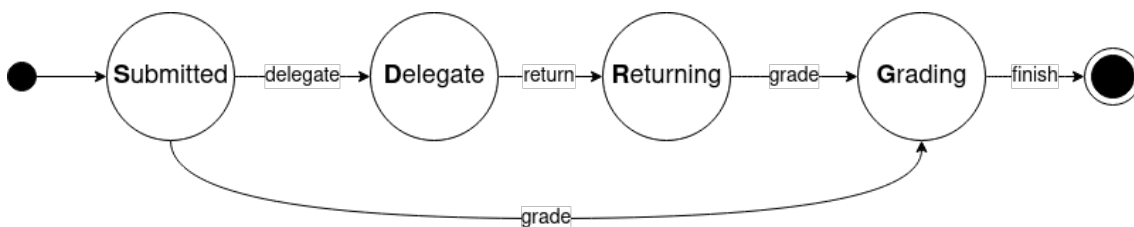
The ACCESS submission model can also be divided logically in two parts: pre-evaluation and post-evaluation. In the pre-evaluation stage, only the specific subclass, holding the user entered data, exists, and the association to `ExecResult` and `SubmissionEvaluation` are set to `null`.

Once the code has been executed, the `ExecResult` is set holding the execution logs and the command used. In case the submission needs to be graded, the `ExecResult` object is then used to compute the `SubmissionEvaluation` during grading, as explained in the following sections.

## Submission Workflow

The Submission Workflow distributes the submissions to the corresponding evaluation component.

An incoming untreated submission is stored in its raw form to the backend. From there, the Submission Workflow forwards the submission to a suitable evaluation component; If needed, the workflow can delegate submissions to external asynchronous tasks and evaluates the result of the task. Multiple-/single choice submissions, as well as text submissions from the Submission Model, are handled as synchronous tasks and are executed immediately. The two code submission types are treated as asynchronous tasks, and their submission is delegated to the Code Runner component. The Code Runner returns an `ExecResult` object that is then forwarded for grading.



**Figure 5.8:** State Machine for the Submission Workflow.

The workflow is modeled as a finite state machine visualized in Figure 5.8. There exist four states (plus the finished machine): 1) *Submitted* is the stored raw submission, 2) *Delegate* is a submission ready to be passed to an external task, 3) *Returning* is a parked submission waiting for the result of an external task, 4) *Grading* is a submission that is passed to the evaluation component. After the evaluation, a submission workflow is finished. The state machine allows the following actions. The *Submitted* state allows the actions of *delegate* and *grade*. *Grade* forwards to the *Grading* state and *delegate* to the *Delegate* state. From there the *return* forwards a submission to *Returning* which then sends *grade* to the *Grading* state.

ACCESS requires flexibility of the submission workflow in perspective to process modeling and application in different deployment scenarios.

The modeling flexibility is addressed by implementing the ACCESS state machine using the Spring Statemachine Framework [14]. This framework is easily integrable into the Spring en-

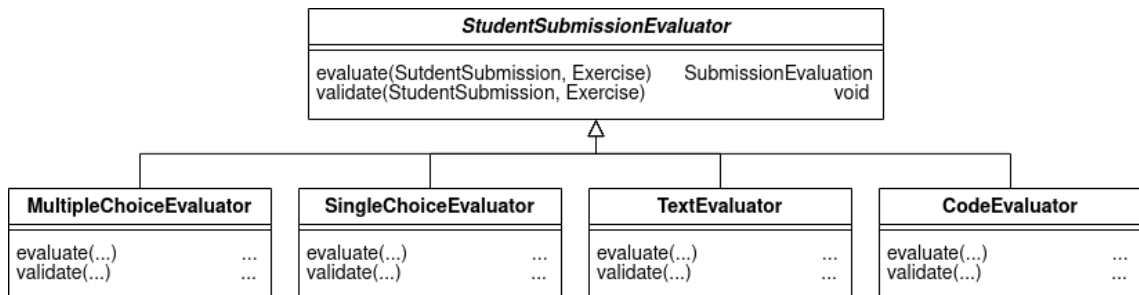
vironment. The framework handles the process execution and allows for a clear separation of business logic. The state machine can be configured programmatically or by providing a Papyrus Model in the UML Diagram Language. The Eclipse Papyrus framework provides a UI Modelling tool. The low complexity of the ACCESS state machine model allows a programmatic configuration. If the model complexity rises, it is possible to switch to the modeling tool.

The extendable architecture allows the adjustment for different deployment scenarios in the future. Every submission creates a new dedicated state machine. State changes in a machine are persisted in the backend, the MongoDB. Halted state machines can be resumed by loading from the database. Multiple ACCESS backends can exchange information over a shared MongoDB, e.g., by using MongoDB's publish-subscribe mechanism for collections. The chosen architecture also allows for a switch to a message queuing service, with minor code changes.

## Grading

The *Grading component* calculates the resulting points (score) for the handed in student submission.

The component provides evaluation strategies that can be linked to a submission type described in Section 5.5.2. For every submission type, a corresponding evaluation type exists, as shown in Figure 5.9. An evaluator accepts a student submission and the corresponding exercise description, which contains information such as a maximum allowed score and rounding strategy defined by the course managers configuration. The system allows ROUND, CEILING and FLOOR rounding strategies. An evaluator returns a `SubmissionEvaluation` that is stored in the student submission.



**Figure 5.9:** Evaluator model used for grading.

Text and choice-based evaluators are straightforward, using list comparison or regular expressions. The code evaluator utilizes the `ExecResult` returned by the Code Runner that contains the output of a private unit test suite. The private test suite is part of the exercise definition provided by a course author. A code evaluator then extracts the number of existing, passed and failed test cases and calculates a score considering the maximum allowed score. The ratio between passed test cases and total cases is then the score the student achieved. To compute the final grade, the score is multiplied by the maximum score, stored in the exercise definition, and rounded using one of the strategies mentioned before.

## Hint System

The *Hint* system provides students with situational information if they are stuck on an exercise and cannot solve it correctly. The hints shall guide the student and allow him to overcome the problem autonomously. The *Hint* system is hooked into the *Grading component*, the area that convenes all needed information to provide useful hints. ACCESS supports three hint mechanisms.

**Static hints** A static hint can be defined directly in the exercise configuration file. This allows for simple hint creation during the exercise configuration and is useful for simple submission types, such as multiple choice. The system shows this static hint if the student submits any wrong answer.

**Assertions** The second hint type is based on the unit tests assertion messages. The unit test output is scanned for predefined patterns to identify assertion messages that are meant to be used as a hint. This mechanism allows for creating hints while developing the unit test suite. Context information of the failed test case and involved attributes can be included in the hint, allowing to create precise and individual clues. An aspect to consider is that this mechanism returns information originating from untrusted source code (from the student). This mechanism needs careful handling, Chapter 6.3.3 presents more details that address this issue.

**Code Runner Limits** The third hint type considers the state of *Code Runner* executions. A *Code Runner* job can return with failures because of invalid shipped source code. A job can also be forcefully aborted when it exceeds limits such as runtime or memory consumption. In such cases, the error message of the *Code Runner* job is forwarded to the user. Chapter 6.3.2 provides reasoning for the used limits.

## 5.6 Code Runner

The *Code Runner* is responsible for the actual execution of code. As explained before, the *Submission component* prepares a Docker container which the Docker daemon, running on the *Code Runner* executes. This component is completely ignorant of how ACCESS works, it just takes the containers, runs them, and returns the output to the *Submission component*. All communication happens over HTTPS, through the API that the Docker daemon provides by default.

**Development Environments** Docker makes it very easy to prepare development environment, without having to worry about conflicting versions of tools and dependencies. For example, a course might need Python 3.7 while another uses Python 2.7. With Docker, there is no need to worry about these clashing with each other. Environments can be prepared and tested locally, and can then be easily deployed to any other machine, knowing that they will work. Furthermore, it makes it trivial, to prepare new environment for new programming languages.

**Code Runner Separation** This component was designed right from the beginning with the idea that it would run on a separate machine from the rest of ACCESS. This simple design decision, makes ACCESS more resilient, as any issue that may occur, will remain confined within the *Code Runner* component. In particular, if the *Code Runner* were to go offline, the main backend server would remain unaffected and available. More details regarding security can be found in Section 6. Additionally, it would be easy to add multiple such *Code Runners* and orchestrate them using a tool such as *Kubernetes* or *Docker Swarm* to accommodate more users.



# Challenges

## 6.1 File System as Database

Due to the requirement of git versioning of the course contents and therefore the lack of serialization of the course data in a database, a robust file system implementation was necessary. The system had to be able to index a folder structure and be able to distinguish between course configuration information and actual course content files. Furthermore, after indexing the folder structure, the differences to the older, already indexed course model had to be detected to identify possible changes. In this chapter we discuss the challenges associated with this requirement.

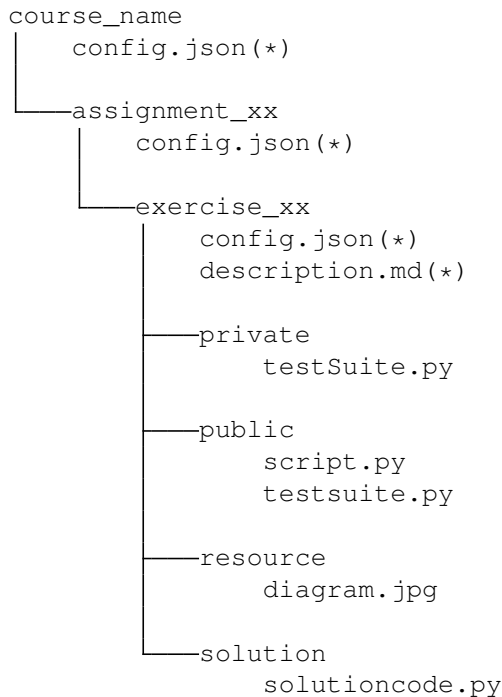
### 6.1.1 File Indexing

Usually in web application, data such as information about a course are stored in a database. An administration panel then allows access to those information and makes modifications on the database. In our case, the requirements specified the course definition to be provided through a file system. The main benefit being a straight forward integration with a git versioning system as well as collaboration possibilities.

To generate an in-memory model of the current course, we first download the course files from the specified repository. Then through a visitor pattern, iterate recursively over the layers in the folder structure and match the expected files inside those folders. For example: We expect the root folder to be representative of the course. By our specification, the course layer should contain one `config.json` file and any amount of folders with the prefix "assignment\_xx", where "xx" denotes the index of the assignment. Inside the assignment folders, we expect also a `config.json` file, which this time refers to the specification of the assignment. The assignment layer can furthermore contain any number of folders prefixed with "exercise\_xx". The exercise layer is the lowest one. Any further folder nesting will be interpreted as an actual folder structure with resource files for the current exercise.

### 6.1.2 Change Detection

One key requirement for our system was to detect changes in the course definition and react according to it. For example, an error in the course grading code was detected. The course owner makes adjustments to the code and commits the change. The system should detect that this change is a "breaking change", meaning the solutions submitted with the old grading schema could now result in a different score with the new grading code. To detect changes, we index the new course structure as described previously and check every value against the old one. Exercise



**Figure 6.1:** ACCESS Course Structure Template. (\*) Denotes required files.

settings such as: `type`, `language`, `options`, `solutions` as well as `executionLimits` will trigger a breaking change if different values are detected. Special care was taken when checking against exercises and assignments. Insertions such as for example an insert of a new exercise between two existing ones are hard to detect. Without any special consideration, the indexer would interpret an insertion as a normal addition at the end, which would result in all the exercises following after the inserted one to trigger breaking changes, as they would now be off by one. For this reason, we introduce an indexing strategy. The exercises as well as the assignments are enumerated by the aforementioned number inside the folder name, following the appropriate prefix. This way, the exercises and assignments are assigned a unique index, which allows us to detect not only additions, but also insertions, and deletions in the list. It is noteworthy, that this index does not have to be constructed by strictly increasing numbers incremented by one. An increment of 10 or 100 will give much more margin for inserting new elements at a later stage. The final index presented in the frontend is then automatically generated from the resulting sequence.

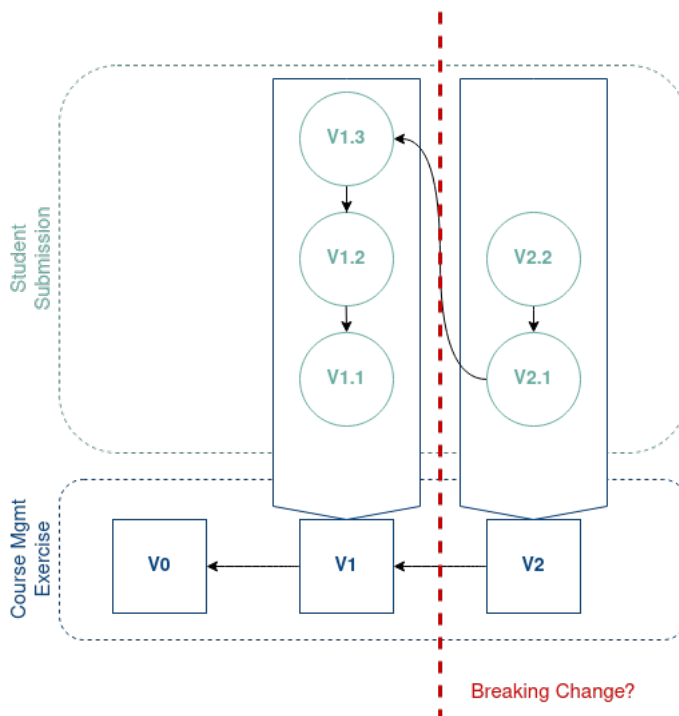
### 6.1.3 Virtual Files

Aside from the configuration files provided in the course file structure, the exercise layer can contain an actual folder structure with undefined recursion. This is necessary to provide all the required files, folders and libraries so that a code exercise can run. For this reason, we index files and folders that are located inside the "public", "private", "solution" and "resource" folder as a flattened list of Virtual files. The Virtual File contains all the usual path, name, extension, media type as well as the actual content of the file.

## 6.2 Submission Versioning

One big hurdle for the ACCESS project was the tractability of submissions, fairness of evaluation and the complexity of multidimensional versioning (see Figure 6.2). It was requested early on that the system should not only allow changes to given exercises but also handle all possible consequences. Given a change to an already published and active exercise, the following questions were considered:

- How severe is the change to the exercise?
- How should the user be notified?
- Can a submission be traced to its original question version?
- What should happen to preexisting submissions?
- How should students be evaluated, if they did not resubmit?
- What should happen if a breaking change is introduced close to the deadline?



**Figure 6.2:** Multidimensional versioning. A student has multiple submissions, three submissions for exercise version one and two submissions for exercise version two.

**How severe is the change to the exercise?** The severity of a changes needs to be decided first before taking any action. As described in Section 6.1.2 a binary classification was introduced describing the change. Any changes classified as "Non-breaking-change" will result in no special action other than overwriting all changed properties and files describing an exercise. However,

if identified as "Breaking-change", a chain of actions are to be triggered, which will be described in detail in the next paragraphs. The idea behind this classification was, to easily allow for silent changes such as adjustments to the exercise title, which don't affect the performance of students, whilst giving everyone an equal playing field when adjustments to grading or the question formulations are made, by notifying the students about the change.

**How should the user be notified?** As discussed, it is important to notify students about breaking changes in the active exercises. initially it was planned to send out notifications about such events via the registered email addresses of the students. However, since OLAT was still the official communication channel between the students and the course coordinators, the decision was made to use the OLAT forum as the main interaction platform for every breaking change that was published. Nevertheless, the ACCESS frontend also needed to highlight such events. Therefore, a prominent red banner as well as other warning labels were introduced to inform the students about the status of an exercise.

**Can a submission be traced to its original question formulation?** To insure against cases of dispute, where a submission could be argued of being made on the basis of an outdated exercise version, the git commit hash of the current course version is stored for every submission. This way every submission is linked to its corresponding version of the question, which provides traceability. This results in multidimensional versioning, where multiple submissions can be made on the basis of one version of the exercise.

**What should happen to preexisting submissions?** Together with the stakeholders and the given requirements, we derived a policy to only consider the last, non-outdated submission for evaluation. In case of a breaking change, all previous submissions get marked as outdated. This consequently means that after a breaking-change event, none of the students has a valid submission. Furthermore, in most cases, the exercise define a maximum amount of submissions that are granted. To ensure that all students can re-submit their solution regardless of the remaining amount of submissions, the submission count is reset.

**How should students be evaluated, if they did not resubmit?** Considering the following scenario, where a student submits his solution early on in the assignment period and is not able to perform a re-submission after a breaking change occurred. Due to the previously mentioned policy, the student would not receive any points. However, to ensure fairness, a system was put in place, which allows the course administrator to automatically re-submit the last outdated submission, in cases, where no valid submissions were detected.

**What should happen if a breaking change is introduced close to the deadline?** It was decided to leave it to the discretion of the course administrator to decide on the appropriate actions in such cases. Possible options that the system allows would be to transform the exercise into a bonus exercise which allows the students to work on the exercise without the grade of the exercise counting towards their final score. A simpler solution would be to extend the deadline.

## 6.3 Security

Due to the nature of ACCESS, it being a web application, in which users largely determine what is shown, and where users can execute arbitrary code, the application is prone to weaknesses. In the

following, we illustrate some security issues we encountered and how we went about mitigating these risks.

### 6.3.1 Remote Code Execution

*Remote Code Execution (RCE)* is a vulnerability which allows an attacker to craft code which, when evaluated by the software, will alter the intended control flow. Such an alteration can lead to arbitrary code execution and in some cases to complete system compromise. ACCESS is by design prone to RCE, as the main requirement is to take foreign code and execute it. Possible effects of running code without mitigation could be:

- Denial of Service (DoS), an attacker give its own processes priority, kill other processes, or even shutdown the system.
- Exfiltration and manipulation of the official grading test suite and/or other student code, before, during and after execution, allowing the attacker to change its own grading, as well as that of others.
- Use of the compromised server to mount an attack from the inside on the backend server, database or identity provider.

A sandbox is a typical security mechanism which can be used to execute untrusted code without risking harm to the host machine or operating system [18]. We decided to use Docker containers as the sandboxed environment. At runtime student code is copied inside a container and executed. Network and host file system access is deactivated to achieve isolation between student containers and between the containers and the host system.

In case a user were still able to escape the sandbox, the sandboxes are physically run on a separate machine from the remaining backend services, to limit the effects of the escape.

### 6.3.2 Denial of Service: Infinite Loops & RAM Usage

Often we tend to think of attacks as very sophisticated exploits, however the code that students learn in their first few weeks of programming can sometimes be enough to bring down an entire system. Listing 6.1 shows an example of a typical beginner error, in which a variable is not incremented on each iteration, resulting in an endless loop. Code like this, if left unchecked will hog the CPU, potentially starving other processes on the same machine. Additionally, since the code is endlessly adding items to a list, the RAM usage will increase until either, the interpreter, or the OS forcefully stops the process, or the system crashes.

```
1 a, b, l = 0, 10, []  
2 while a < b:  
3     l.append(a)
```

**Listing 6.1:** Example of buggy student code: the iterating variable should be incremented on each iteration. This results in an infinite loop and eventually an out of memory error, as the list increases in size.

While static analysis can sometimes be used to detect infinite loops, the problem remains, for finite, but long running processes and for processes which require a lot of RAM.

Clearly, a solution is needed, which is robust enough to catch any time- or memory-consuming processes. For this the chosen sandbox, Docker, provides all the tools necessary to set limits on the resources at disposal per execution. At execution time we pass the values stored in the exercise `config.json`, to the docker daemon, which set limits on the docker container. Listing 6.2 lists

all possible factors. It is possible to limit the amount of memory and CPU cores at disposal, the maximum runtime of the container, and whether the container has networking enabled. Any violation on the memory, CPU and runtime limits, results in immediate termination of the container with an error flag, which we then use to advise the user about what happened. Default limits are in place in case an administrator forgot to set any in the configuration.

```

1 {
2     "type": "code",
3     ...
4     "executionLimits": {
5         "memory": "64",
6         "cpuCores": "1",
7         "timeout": "10000",
8         "networking": false
9     }
10 }
```

**Listing 6.2:** Execution limits configuration: the container may at most use 64 MiB of memory, 1 CPU core, has a maximum runtime of 10 seconds (expressed in milliseconds) and has no networking.

### 6.3.3 Grading Test Suite Leakage

A further challenge to the design of ACCESS and which is still an open point, is how to guarantee the confidentiality of the grading test suite. As explained in Chapter 5, the course administrator prepares a grading test suite, which tests the code submitted by a user. This grading test suite remains secret until after the deadline, since knowledge of the test cases, while not exactly a solution, would greatly simplify getting the maximum score on an exercise.

In order to execute the test suite, both the student code and the tests need to be available in the container during runtime. A student could then use simple calls, for example `open()` in Python, to read the contents of any file on the file system, including the tests, and print them on the console. Operating system level solutions, such as a *chroot Jail* or blocking any Unix `read(2)` calls would not work since both the test suite and the student code has to be available to the user executing the code within the container. Shadowing or mocking function calls in Python is not effective as the probability to find a non-shadowed or non-mocked version of the same function is very high in the global namespace. Additionally, the cost of maintaining a black- or white-list of functions, other than being very error-prone, is prohibitively high. As an example, the following convoluted one-liner reads a file from the file system. Such an example would easily pass by any filtering and even through most static analyzers:

```

1 getattr(().__class__.__bases__[0].__subclasses__()[40](r'/etc/passwd'),
2     'read')()
```

**Listing 6.3:** Accessing an arbitrary file without explicitly importing any packages, nor explicitly calling the built-in `open()` or `read()`. Both the filename and the “read” string could be easily obfuscated.

Since it is not possible to guarantee the confidentiality of the test suite, we reduce the surface, or the window of time, where this is available. The grading test suite is not accessible to the code while testing but only during a graded execution. Since the test suite is vulnerable during this time, all test and console output is suppressed for graded runs.

Unfortunately, because of the hint system, it is hard to further protect the suite. Indeed, this can be used to extract information from the running container. As was shown in Section 5, the hint system relies on specially demarked text printed by an `AssertionError` to present information to the user regarding what went wrong with their submission. Unfortunately, a user can raise their own `AssertionError` using the same markers, to control what the hint will display. Static analysis, would likely not be effective, considering the possibilities obfuscation offers, as in the example above. Even dynamic analysis would not work, considering the stacktrace can be freely manipulated, it would be hard to determine where exactly an exception was raised.

A posteriori analysis of displayed hints would allow to find users abusing the system, at which point however the test suite was already leaked. A simple approach could be to collect all hints displayed and compute the distance of a hint from the majority of hints. If a hint is markedly different from the majority for a certain exercise, it would raise a warning, which an assistant or lecturer would have to then more closely examine. Another way, would be to prepare a white-list of hints which the system is allowed to display. However, depending on the number of tests and hints and weekly exercise cadence, this system could be too time-consuming and error-prone to be actually implemented.

## 6.4 Onboarding - User without UZH Email Address

In some cases, students receive their UZH credentials after the semester has started. This is especially the case, for students that transfer from other universities. Waiting for students to receive their university email address would not be a solution, as they would not be able to work on the first few assignments. In order to allow students to work on assignments from the start, an account is created using their private email address, but once they receive their new UZH email address, a new account would have to be provisioned, effectively resulting in two separate accounts for the same user. A simple approach to merge user accounts was devised, but was only implemented after the master project and in the maintenance phase, due to time constraints.

- The two accounts, *account 1* and *account 2*, to merge are selected.
- All submissions are migrated from *account 1* to *account 2*. A submission stores the user Id of the submitter as well as a version number. As such, migrating submissions is effectively just the process of replacing the user Id to that of account 2. A bit more complicated is how to handle the version number. This is a simple counter which starts from 0 for the first submission, and increments by one for each new submission. To avoid having duplicate version numbers, *account 1*'s submissions are prepended to those of *account 2*. For example if before there were two submissions *s1* and *s2* with version 0 and 1, after the migration *s1* would have version -2 and *s2* version -1. This way the versioning remains consistent, since *s1* has the lowest version number, and there are no conflicts with other submissions that might have occurred with the new account.

In future this process can be improved by interleaving the submissions instead of prepending the two submission histories.





# Evaluation

This project set out with the goal of designing and implementing a system to automatically grade students code submissions, allowing future programming courses at UZH to scale to even higher number of participants. We took the *Informatics 1* course as a target in terms of participants (number and type of users) and exercise types. Thus, for ACCESS we had the following goals:

- Lighten the workload for tutors.
- Allow the *Informatics 1* course to scale.
- The typical *Informatics 1* student has little to no programming experience. Thus the system has to be easy to use to a first year programming student.

In the following sections, we show that the design system achieves all the above requirements.

## 7.1 Decreased Workload

As mentioned previously, in 2018, the department hired 12 tutors to handle the increasing amount of students. Each tutor had a class of 30 to 35 students, which handed-in weekly exercises. By talking with tutors from the previous years, we found out that the average tutor required about 15 hours a week to grade their share of assignments. In 2018, the students had to hand-in 12 exercises over the course of the semester. With these numbers, we can calculate that tutors took:

$$\begin{aligned} 15 \frac{\text{hrs}}{\text{week tutor}} \times 12 \text{ week} &= 180 \frac{\text{hrs}}{\text{tutor}} \\ 180 \frac{\text{hrs}}{\text{tutor}} \times 12 \text{ tutor} &= 2160 \text{ hrs} \end{aligned} \tag{7.1}$$

This makes it the equivalent of 1.09 man-years in Switzerland (assuming 42 hours a week and 5 weeks vacation time which is equal to 2160 hours). This is purely the time needed to grade student code, without accounting tutoring sessions. We consider that a small team of tutors (2 or 3) together with ACCESS would be enough to handle the load of students for the future iterations of *Informatics 1*. Indeed, tutors would only be needed for tutoring sessions and to answer students' questions. Thus in practice 3 tutors, each working 3 hours a week (2 hours tutoring and 1 hour answering students' questions) would take:

$$3 \text{ tutor} \times 3 \frac{\text{hrs}}{\text{week tutor}} \times 12 \text{ weeks} = 108 \text{ hrs} \tag{7.2}$$

Which is a reduction of 95% in the amount of man-hours.

The amount of hours needed to prepare the exercises are not included in the above calculations. For ACCESS this is largely dependent on the hint system. If no hint are needed, then the amount of work is the same as in 2018. If hints are needed, than the total amount would increase depending on the desired granularity of the hints provided. The more precise the hints are, the more time a course author has to spend on preparing the test-suite for grading.

## 7.2 Performance Testing

When we had our system running on a DigitalOcean server, we also performed a stress-test to verify that our system is able to handle the amount of users we were expecting for *Informatics 1*. Specifically, we tested code submission executions and grading, since it is the most intensive process in the system, since each submission spins up a new Docker container. We prepared a test scenario with the Gatling framework in which we simulated up to 850 submissions within a one second time-frame where our system successfully handled over 700. Tests with 500 users were also successful. This is an extreme scenario, as even in an exam setting, the probability of 500 users submitting in the span of a single second is extremely unlikely. Additionally, other scenarios were simulated where the number of users increases from a 50 at a time all the way to 1000 to test how far the system could keep up. The system was able to keep up to about 800 users concurrently submitting in this other scenario. This scenario is more realistic, as usually, users do not appear all at once, but rather might come and go over the course of a period of time. All performance tests ran against a setup of a quad-core Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz and 8 GB of RAM. This led us to the conclusion that such a setup would be well equipped to successfully handle the described context of our app.

## 7.3 User Testing

After successfully implementing the prototype, the system was tested by observing volunteer participants. These participants were instructed in the context in which this application will be used, the target audience of the application since our participants were more advanced in their studies than our target audience. We gave them an account and tasked them with solving the exercises in the given assignment. We observed them as they were navigating the application, and using the components to solve the respective exercises. We were sitting next to the participant and gave answer if the participant had a question. We then asked the participant questions like:

- What do you like about the application?
- Can you explain how the navigation is organized?
- Can you explain what the template reset button does?
- Can you explain the difference between test runs and submits?
- Can you explain what the console output?
- Can you explain the test output are?
- Can you explain the difference between code and code snippets exercises?
- Can you explain how to save your progress without submitting it?
- Can you explain how many submissions are left?

- Can you explain why some submissions are marked red?
- Can you explain how many points you received for the first coding exercise?
- Can you explain how many points you received for the assignment?
- Can you imagine solving the Info 1 exercises for one semester with this tool?
- What did you find confusing?

Overall the general feedback or conclusion we got from these prototype tests were that they found their way around in our user interface, they had no problem coding and running the code and submitting their solution. The participants were confident that first semester students would be able to use this application and that this system would improve this course from a students perspective. Some of the observations we made and some of the remarks of the participants also led to changes concerning the user design, like button-coloring scheme, improvements in the content or presentation of the hint system and also changes in the presentation of a breaking change occurrence. We also found cases that we did not think about like participants using hotkeys like control+s which they are used to from local code editors which triggered some default browser behavior which we later on caught and made into a saving function or breadcrumbs for navigation.

## 7.4 Usability Testing

The biggest part of the evaluation of the system was when it was actively being used by those 450 students to solve their weekly exercises, feedback concerning our system came through emails to the course author, or in the form of comments in an OLAT forum of the course, a platform being used by the university to distribute course content like slide-sets. These comments and feedback led to bug fixes and other code improvements.



# Conclusions

This project started with the ambitious goal to gather requirements, design, and implement an appropriate auto grading system within six months, that would be used for the upcoming introductory course to programming. Tutors were still hired by the Software Engineering Institution to grade the exercises manually. We are proud to say that we were able to have this system running on UZH servers even before the semester started. The Tutors were then used for tutoring sessions and to test out the exercises on our staging server on DigitalOcean before we pushed the updates to the live server.

A big motivational factor of this project was the aim to build a system on UZH servers that would be used in practice by university students to solve their exercises.

We found that much time got invested into coming up with different concepts and solutions and design decisions, and the requirements gathering took longer than expected. Time was also an issue, each one of us taking lectures and having a student job of 40 to 60 percent. We tried keeping track of development progress and issues and requests through GitHub issues, and in our weekly meetings, prioritized features into sprints. There were specific challenges that we listed explicitly in this approach, like having multidimensional versioning for mapping exercise-versions, when the content of an exercise gets updates by the course creator, to student code submissions and the inherent consequences like what happens to a student submission that refers to an old version of an updated exercise. Since there were different stakeholders, it was sometimes demanding to resolve different notions on subjects like what content the console output will provide to the end-user.

Other things we learned were that questions and requests from users and supervisors took away time from the actual development of new features. To remedy this, we established more controlled communication patterns between the developers and the supervisors, like limiting the number of requests sent to the developers.

From a technological standpoint, we were free to choose and had the opportunity to familiarize ourselves with technologies and libraries like Docker, React, and Keycloak, which most of us had no experience. Integrating the components, like Keycloak, databases, and Docker containers into a running system, also added to our gained knowledge. Observing the system running on live servers in a stable state and being heavily used by students was a reassuring and satisfying experience. The observed side effects of students not making use of the local development setups and using the power of local IDE's but doing most of their work in ACCESS online code editor served as a testament of the high quality of our system.



# Future Work

**System** We see this system being continually used by the university for the *Informatics 1* course for which it had been originally designed, and we already had been asked to provide further maintenance as this system is being used for the next course. At the same time, we could envision the system to be used for other courses with other programming languages. Upcoming students will probably have the chance to continue working on this project in their respective master projects and could, therefore, extend this system to support other specific exercise types and programming languages used by other courses or even other institutions or universities or even integrated with online courses.

**Features** For our system itself, next to providing support for other programming languages, we would have envisioned the following features and might even develop them in our spare time as a team:

**Exercise Commit Message** When saving a version of an exercise, the submission is assigned an incrementing version number. If the user could provide a short commit message, we could display the version of the exercise together with its commit message, where the user could hint towards the content of this version. The user would then not have to click through the exercise submissions to know the changes the user made, and can more simply restore a specific previous submission.

**Exercise Submission Search** Together with the commit message, a search functionality could be included, to allow the user to search for a specific submission by commit messages. Currently, only the last ten submissions are displayed in the frontend to avoid cluttering the interface. This way, the user could search and restore an old submission that was even further back than ten submissions.

**Admin Statistics Dashboard** A prototype for a 'statistics dashboard' application has been implemented. This application provides a separate frontend and fetches data from the server to run statistical calculations. It includes a dashboard for the administrator where he can get detailed information, for example, on a task-level, how many students have already solved this task. It can give the course author insights into the level of difficulty of a task, or which task the students have more problems with or take longer to solve. This metrics might indicate a missing understanding of students and suggest to adjust the corresponding learning material. Additionally, this new data can be used for research on the habits and behavior of new programmers for programming or the use of ACCESS features.

**Course Setup** The course setup at the moment is laborious; it requires a repository owner to configure public and private keys manually to allow ACCESS to clone the repository and webhooks to be notified of updates. Many tools, such as for CI/CD and static analysis, can set up this kind of connections automatically, by only requiring the user to login to the application using the repository accounts. Such a system would make the initial course setup much more accessible.

**Repository Integration** Deeper integration with GitHub or GitLab, would allow to use ACCESS as a sort of CI system, where each commit to the repository is treated as a submission: the code is automatically checked-out and run against the test-suite. This would allow students to get even more insight in how developers usually work in a professional environment. Final submissions would still happen on ACCESS as a form of confirmation to avoid any possible issues. Additionally, two-way sync could also be added, where any submission on ACCESS commits the code to the student's repository.

This could pave the way for future collaborative features, for example a group project “exercise” type, where students work together on an assignment in a repository, using GitHub and ACCESS is used for grading, thus giving students a more realistic experience of how professional developers would normally work.

**Course Content Seed** Whenever testing any feature that required us to change a course repository, we would have to commit and push the changes and then start the server, which would clone the repository. However, some changes are not backward compatible, so people working on a different branch might find themselves not being able to work since the changes might crash the server on startup. A solution at the moment would be to fork the repository and test any changes there. However, it would be useful if there was a way to work directly on the file system and import the repository directly into ACCESS, without needing to clone from a remote repository.

**Hint System Hardening** As mentioned in Chapter 6, the hint system at the moment can be used to leak the grading test-suite. Thus, time should be invested to better protect the test suite.



## Appendix A

---

# Appendinx

## A.1 Feature Table

MOOC's Comparison Table						
Features	Khan Academy	edX	Exercism	Code Expert	Okpy	
Online Code Editing/Execution	Yes (snippets)	Yes	No	Yes	No	
Grading Method	Automatic	Manual & Automatic	Manual	Manual & Automatic	Yes	
Content Authoring	Online course customized using existing predefined content	edX Studio	Git-based local filesystem	Online editor	Okpy client	
Grading vs Learning/Mentoring	Grading + mentoring	Grading & Mentoring	(Crowd-sourced) Mentoring	Grading & Mentoring	Grading & Mentoring	
Question Types	Programming, multiple choice, text answers	Checkbox, drop-down, multiple choice, numerical, text input	Programming	Programming	Programming, multiple choice	
Present Content	Website	Website	No	No	Website	
Media Support	Text, images, videos	Yes	Any supported by markdown	Any supported by markdown	Images	
Hint System	Yes	Yes	Mentors can leave hints	Tutors can leave hints	Mentors can leave manual comments	
Open vs Closed Source	Closed	Open	Open	Closed	Open	
Managed vs Self-Hosted	Managed	Managed	Managed (code execution local)	Managed	Managed or self-hosted	
Pricing Model	Free (takes donations)	Free	Free	No	Free	



---

# Bibliography

- [1] Ace - high performance code editor for the web. <https://ace.c9.io/>.
- [2] Angular. <https://angularjs.org/>.
- [3] Code expert logo. <https://code-expert.net/>.
- [4] Codeboard - a web-based ide to teach programming in the classroom. <https://codeboard.io/>.
- [5] Codemirror is a versatile text editor implemented in javascript for the browser. <https://codemirror.net/>.
- [6] edx - edx is the trusted platform for education and learning. <https://www.edx.org/>.
- [7] exercism - code practice and mentorship for everyone. <https://exercism.io/>.
- [8] Khan Academy. <https://www.khanacademy.org/>. Accessed: 2020-01-27.
- [9] The monaco editor is the code editor that powers vs code. <https://microsoft.github.io/monaco-editor/>.
- [10] Ok - automate grading personalize feedback. <https://okpy.org/>.
- [11] py4e - python for everybody. <https://www.py4e.com/>.
- [12] React - a javascript library for building user interfaces. <https://reactjs.org/>.
- [13] Spring framework. <https://spring.io/projects/spring-framework>.
- [14] Spring statemachine framework. <https://projects.spring.io/spring-statemachine>.
- [15] Vue - the progressive javascript framework. <https://vuejs.org/>.
- [16] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. pages 1–15. Springer-Verlag, 1996.
- [17] S. Brown. Software architecture for developers. *Coding the Architecture*, 2013.
- [18] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer, et al. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, volume 6, pages 1–1, 1996.

- [19] G. Haldeman, A. Tjang, M. Babeundefined-Vroman, S. Bartos, J. Shah, D. Yucht, and T. D. Nguyen. Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education, SIGCSE '18*, page 278–283, New York, NY, USA, 2018. Association for Computing Machinery.
- [20] C. Ireland, D. Bowers, M. Newton, and K. Waugh. A classification of object-relational impedance mismatch. In *2009 First International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 36–43. IEEE, 2009.
- [21] Jeff Lindsay. Web hooks to revolutionize the web. <https://web.archive.org/web/20180630220036/http://progrum.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/>.
- [22] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). RFC 7519, RFC Editor, May 2015. <http://www.rfc-editor.org/rfc/rfc7519.txt>.
- [23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.
- [24] Red Hat. Keycloak - open source identity and access management.
- [25] P. Rempel and P. Mäder. Estimating the implementation risk of requirements in agile software development projects with traceability metrics. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 81–97. Springer, 2015.
- [26] F. N. Sakimura, J. Bradley, P. Identity, M. Jones, B. de Medeiros, and C. Mortimore. Openid connect core 1.0.
- [27] I. Young, L. Johansson, and S. Cantor. The entity category security assertion markup language (saml) attribute types. RFC 8409, RFC Editor, August 2018.
- [28] D. Zowghi and C. Coulin. Requirements elicitation: A survey of techniques, approaches, and tools. In *Engineering and managing software requirements*, pages 19–46. Springer, 2005.