

Author
Alexander Kainzinger,
Bsc
k11802873

Submission
Institute for System
Software

Thesis Supervisor
Dipl.-Ing Dr. **Markus**
Weninger, Bsc

October 2024

Workflow Enhancements for the Online Examination Tool Xaminer



Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

Abstract

The online exam system Xaminer has undergone several functional expansions. However, its design and technological stack have not kept up with the times, making maintenance and use more difficult. This thesis addresses these issues by modernizing the frontend stack, decoupling frontend and backend components, and implementing user-side quality-of-life (QOL) improvements. Modernizing the frontend from Vue.js 2 to Vue.js 3 ensures compatibility with current standards, enhancing performance and maintainability. Several architectural enhancements were made, such as decoupling the frontend and backend using Docker and Docker Compose to improve system resilience. Establishing a CI pipeline with GitHub Actions streamlined updates. User-side improvements include single PDF exports of exam results, support for preliminary submissions, subquestion formatting, and real-time updates on the exam administration page. These enhancements improve the user experience for students and lecturers. The thesis achieves a modernized, decoupled architecture with streamlined deployment and notable improvements in user functionality, contributing to the advancement of the online examination system Xaminer.

Kurzfassung

Das Online-Prüfungssystem Xaminer hat diverse funktionale Erweiterungen erfahren, jedoch konnte sein Design und technologischer Stack in dieser Zeit nicht auf dem neuesten Stand gehalten werden, was die Wartbarkeit und Nutzung erschwerte. Diese Arbeit adressiert diese Probleme durch die Modernisierung des Frontend-Stacks, die Entkopplung von Frontend- und Backend-Komponenten und die Implementierung von benutzerseitigen Quality-of-Life (QOL) Verbesserungen. Die Modernisierung des Frontends von Vue.js 2 auf Vue.js 3 stellt die Kompatibilität mit aktuellen Standards sicher und verbessert die Performance und Wartbarkeit. Des Weiteren wurden mehrere architektonische Verbesserungen vorgenommen. Die Entkopplung von Frontend und Backend mit Docker und Docker Compose verbessert die Systemresilienz des Systems. Durch die Einrichtung einer CI-Pipeline mit GitHub Actions wurde der Update Prozess zusätzlich vereinfacht. Zu den Verbesserungen auf der Benutzerseite für Studierende und Dozenten gehören der PDF-Export von Prüfungsergebnissen, die Unterstützung für vorläufige Einreichungen, die Formatierung von Unterfragen und Echtzeit-Updates auf der Prüfungsverwaltungsseite. Diese Erweiterungen verbessern die Benutzerfreundlichkeit für Studierende und Dozenten. Die Dissertation erzielt eine modernisierte, entkoppelte Architektur mit optimierter Bereitstellung und bedeutenden Verbesserungen der Benutzerfunktionalität und trägt damit zur Weiterentwicklung des Online-Prüfungssystems Xaminer bei. Das Benutzererlebnis wird durch diese Erweiterungen der Funktionen für Studenten und Dozenten verbessert.

Contents

1	Introduction	1
1.1	Outline	2
2	Background	3
2.1	Technoloy Stack	3
2.1.1	Backend - Spring Boot & Kotlin	4
2.1.2	Frontend - Vue & TypeScript	6
2.1.3	Application Programming Interface - API	7
2.2	Deployment and Development Tools	7
2.2.1	Docker	8
2.2.2	Version control (Git) & GitHub	9
2.2.3	Frontend	9
2.2.4	Backend	10
2.3	Xaminer Overview	11
2.3.1	Architecture	11
2.3.2	Deployment & Hosting	12
3	Modernize Frontend Stack	14
3.1	Vue 3 Upgrade	14
3.2	Bundling Tool	15
3.3	Migration Path	17
3.3.1	Dynamic Route Generation	17
3.4	Testing Tool	19
4	Deployment and Hosting	21
4.1	Containerization	21
4.2	Docker Compose	22
4.2.1	Backend & Database	24
4.2.2	Frontend	24
4.2.3	Reverse Proxy	24
4.3	Automated Builds	26
4.3.1	GitHub Actions	26

Contents

5	Bidirectional Communication	31
5.1	Technology Evaluation	31
5.2	Socket.IO	33
5.2.1	Connection Lifecycle	33
5.2.2	Namespaces	33
5.3	Use Cases	34
6	Question with Sub-questions	37
6.1	Motivation	37
6.2	Implementation	38
6.2.1	Recursiveness in the Backend	38
6.2.2	Recursiveness in the Frontend	39
7	Question Search	41
7.1	Motivation & Concept	41
7.2	Implementation	43
7.2.1	Recursive SQL Join	44
8	Preliminary Submission	46
8.1	Motivation & Status Quo	46
8.2	Concept & Security Considerations	47
8.3	Implementation	49
9	Single PDF Export	53
9.1	Motivation & Concept	53
9.2	Implementation	54
10	Usage and Evaluation	56
10.1	Docker & Docker-Compose	56
10.2	Questions with Sub-Questions	58
10.3	Question Search	59
10.4	Single PDF Export	60
10.5	Bi-directional Communication	60
10.6	Developer Experience Improvements	61
10.6.1	Frontend Tooling	61
10.6.2	Automated Builds	62
11	Conclusion and Outlook	65

Master's Thesis

Workflow enhancements for the online examination tool Xaminer

Dipl.-Ing. Dr. Markus Weninger, BSc

Institute for System Software

T +43-732-2468-4361

markus.weninger@jku.at

Student: Alexander Kainzinger, BSc

Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc

Start date: October 2023

The coronavirus outbreak in 2020 strongly changed the way how teaching is performed at universities. Most classes switched to an online teaching mode – including digital exams. While teaching mostly went back to teaching in presence, a number of exams are still performed digitally. For this, the online examination tool Xaminer has been developed at the Institute for System Software at the JKU. Since its first version, the tool has been expanded into various directions but still has potential to provide more quality-of-life features, for users as well as for the developers.

Thus, the goal of this thesis is to target these quality-of-life features as well as the system's architecture to increase its easy-of-use as well as its simplicity of hosting the system. Overall, the enhancements can be divided in the following categories, which should be tackled in the given order:

- **Modernize Frontend Stack:** Even though Xaminer constantly received new feature, only little time has been spent on keeping its tech stack up-to-date. For example, Xaminer was built using the frontend framework Vue.js 2. Yet, since September 2020 Vue.js 3 is available. Thus, old libraries and frameworks should be brought up-to-date and should be checked for compatibility.
- **Deployment and Hosting:** Since Xaminer was first released, its frontend and backend have been strongly coupled. It is thus not possible to restart the one without the other. Part of this thesis is to de-couple both entities, setting up Docker and Docker Compose settings for both, and to run automatic tests using GitHub Actions upon repository commits („CI pipeline“). A setup guide has to be written that describes how to host Xaminer on a server and how to restart it if necessary.
- **User-side QOL improvements:** Over the years, various „nice-to-have“ feature requests have accumulated which should be targeted in this thesis. These include:
 - **PDF export in a single file:** Currently, exam results can be exported in a .zip file containing one .pdf file per student submission. In the future, it should also be possible to export all submission in a single .pdf file that can be easily printed double-sided (one sheet of paper should never contain the submission of more than one student).
 - **Preliminary exam submissions:** Currently, students can only submit the exam once. To prevent data loss, it should be possible to also save exams preliminary.
 - **Questions with subquestions:** One of the most requested features by lecturers is to support questions with sub-questions, e.g., multiple questions sharing the same descriptor and summing up their points to a common end score. This involves a nice editor to define such questions as well as nice formatting during exam participation.
- **Bi-directional server-client communication:** Currently, certain pages have to be manually refreshed to show the most current data, first and foremost the exam admin page. This page is used to monitor which students are still working on their exam and which students have already submitted. In this thesis, this page should be reworked to automatically update when a new submission has been performed. The exam supervisor should also be informed with a small popup / notification that a new submission has been performed.

Modalities:

The progress of the project should be discussed at least every three to four weeks with the advisor. A time schedule and a milestone plan must be set up within the first two weeks and discussed with the advisor and the supervisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis is expected to be finished before the 31.10.2024.

1 Introduction

The coronavirus pandemic necessitated a rapid shift to online education, disrupting traditional teaching and examination methods at universities. This sudden transition highlighted the limitations of existing online learning platforms like Moodle, particularly for coding exams. In response, the Institute for System Software (SSW) at Johannes Kepler University (JKU) developed Xaminer, an online examination tool designed to address these shortcomings. Xaminer allows various questions, including text, single- or multiple-choice, and coding questions with syntax highlighting and in-browser compilation.

Despite its initial success and continuous updates, Xaminer requires significant enhancements to improve its technology stack, deployment process, and user experience. This thesis aims to modernize Xaminer by updating its frontend from Vue.js 2 to Vue.js 3, improving performance and compatibility with modern libraries. Decoupling the frontend and backend, implementing Docker configurations, and setting up a continuous integration (CI) pipeline will streamline deployment and maintenance.

User experience improvements are also crucial. Key enhancements include consolidating exam results into a single PDF file, allowing preliminary exam submissions to prevent data loss, and supporting complex question structures with sub-questions. Additionally, implementing bi-directional server-client communication will enable real-time updates and notifications, enhancing exam monitoring capabilities.

Moreover, these updates will foster a more efficient and engaging experience for lecturers and students. Lecturers will benefit from enhanced exam creation tools with better visual dependencies and live notifications of submissions. Students will have a more reliable platform that safeguards their work through continuous server backups, ensuring no data loss, even in the event of a browser crash. By modernizing the development infrastructure with tools like Docker and GitHub Actions, the process of adding new

1 Introduction

features and maintaining the system will become more streamlined, ultimately supporting the sustained evolution of Xaminer.

By addressing these specific areas, this thesis aims to significantly enhance the functionality, usability, and maintainability of Xaminer, ensuring that it remains a valuable tool for digital examinations for many semesters to come.

1.1 Outline

The further structure of this thesis is as follows: In Chapter 2, we introduce the technologies used and mention sources of truth relied upon during the implementation while also describing the general idea and architecture of Xaminer. Following that, each chapter covers an individual important component that has changed or is newly introduced to the software. Chapter 3 explains how the frontend application has been modernized, drastically improving the development experience and future-proofing Xaminer. In Chapter 4, we will elaborate on the second part of modernizing and future-proofing the application by creating modern shippable containers to run the software and automating the build steps required. Chapter 5 introduces new bidirectional communication enhancements, providing new capabilities of sending real-time updates to supervisors of specific student actions such as submitting the exam. Chapter 6, 7, 8, and 9 introduce the four new quality-of-life improvements for supervisors and students such as sub-questions, a question search for creating exams, or creating a single PDF of all submissions ready to be sent to the printer. For documentation of the newly introduced improvements to Xaminer, Chapter 10 will give a short evaluation and a glance at how each feature can be leveraged. Finally, we provide a brief conclusion and outlook on future work in Chapter 11.

2 Background

In this section, we go over the technological stack, concepts of the architecture, which is split into *backend* and *frontend*, and resources relied on to plan, implement, and verify all parts of the system. Lastly, we will provide a short overview of the current system.

2.1 Technology Stack

The foundation on which the system is built can be divided into two main concepts that are connected using a standardized interface. Understanding each part of the system is crucial to preparing the changes accordingly so that no existing functionality is broken. However, to achieve the desired improvements, a deep knowledge of the newly introduced components and software is also needed.

A solid understanding of object-oriented programming and software architecture is essential for effectively working with the backend system. Proficiency in Java or Kotlin, along with practical experience with the *Spring Boot framework* (Section 2.1.1), is particularly valuable. The source code for this part of the application can be mainly written using IntelliJ IDEA and the latest stable Java LTS-version 17. JetBrains offers the Ultimate edition for students at no cost, allowing for better IDE integration when working with the Spring Boot framework, such as intelligent code insight, inspections, instant code navigation, and many more.

Secondly, deep knowledge and understanding of HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS), which are the main languages used to create a *frontend* (Section 2.1.2) should be known or acquired through personal education. A learning-by-doing can be performed when working with the framework *Vue* as Xaminer is using this framework for building the user interface. Implementation and test code for

2 Background

the second part of the application can be written using Visual Studio Code with many different open-source plugins to improve productivity, code insights, and inspections.

Defining the interface for connecting those two concepts using an *Application Programming Interface (API)* (Section 2.1.3), the introduction of automated testing and building of the application stack using GitHub Action, and the change to the hosting setup to make use of Docker instead of directly hosting the application on a server, in addition to basic Linux and Bash understanding, should also be well known to gain a good understanding of each of the following sections.

Finally, many internet resources like the Spring Boot, Vue, Cypress, GitHub Action, and Docker documentation, or tutorial sites like Baeldung¹ provided a comprehensive and helpful elaboration about certain features of the framework, using practical code-samples and detailed explanations.

2.1.1 Backend - Spring Boot & Kotlin

When referring to a backend in software engineering, typically, this is part of an application that operates behind the scenes (or in the “background”), creating an abstraction between a representation layer and a storage layer. The storage layer, in most cases, is a database, such as PostgreSQL or MongoDB. On the other hand, the representation layer would be referred to as frontend, which will be discussed in Section 2.1.2. The backend always runs on remote hardware, e.g., a server that needs to be reachable through the internet. In most cases, only one instance of such an application is running, but also through scaling, multiple instances of the same application can be running to improve performance on high traffic.

The data access or processing layer handles requests, manages data, and generates a response that can be transmitted to the representation layer. Often, it contains a lot of application logic, such as business rules, algorithms, or processing logic that determines “how” the data is handled and what responses are generated based on the input or other events.

¹<https://www.baeldung.com>

2 Background

To allow such request-response behavior, data management, etc., a framework - in this case, Spring Boot - can be used as a foundation. The Spring Boot framework includes components right out of the box, such as:

- Spring framework² provides a comprehensive programming and configuration model for modern Java-based enterprise applications.
- Embedded web server such as Tomcat or Jetty to handle *Hypertext Transfer Protocol (HTTP)* requests.
- Automatic configuration of Spring and 3rd party libraries whenever possible.

What this means is that Spring Boot, in its latest version 3, provides the “chassis” for creating an executable program that includes the web server that will help us to handle HTTP requests, automatically configure any additional dependencies, and other features that help us to create a production-ready application. Spring also provides extra dependencies for database connection, user authentication, or mail transfer. These dependencies can be included in the dependency management system, which automatically adds and configures them for the Spring Boot application.

Building this API can be easily done using multiple Spring annotations to define a resource, as seen in the example in Listing 2.1. To provide information to the framework that a certain class should be treated as a REST controller, we annotate it with the corresponding annotation and optionally with a base API path using `@RequestMapping()`. Implementing the actual endpoint, defining a method, return value, and annotating it with its corresponding annotation provides enough information for building the endpoint. The example provides only a very basic setup, nevertheless, more complex tasks with query parameters, or other HTTP operations with data are built similarly.

```
1 @RestController
2 @RequestMapping("/api")
3 class MyExampleController() {
4     @GetMapping("/hello-world")
5     fun sendHelloWorld() = "Hello, World"
6 }
```

Listing 2.1: Example REST controller

²<https://spring.io/projects/spring-framework>

2 Background

Testing is as important as the actual implementation, requiring at least one test case when writing a new feature or fixing a bug. For simple unit tests, JUnit as the most commonly known framework, was chosen. When writing more complex integration tests, such as calling an HTTP API endpoint, the Spring application needs to be started and set to testing mode allowing a replacement of certain services with mocked implementations creating reproducible, predictable test cases.

2.1.2 Frontend - Vue & TypeScript

The frontend is the so-called representation layer, visualizing the data retrieved from the backend. This part of the application, in comparison to the backend, runs locally on a user's machine in the browser. This means there can be $0:N$ instances running of the same frontend, all communicating with the same backend.

As is commonly known, a web page usually consists of three components: HTML, CSS, and JS, which are used to build the skeleton, style it, and make it interactive. On top of that, building a modern webpage comes with deciding on a frontend framework that, similarly to the backend, should simplify and support the development experience, e.g., by creating a component-based programming model for building user interfaces. In this case, Vue was already present to create a single-page application (SPA³).

Vue out-of-the-box is only the framework that runs in the browser, it does not come with any build tool or testing engine. These parts have to be installed and configured separately to be able to run the application for development or production.

- **development** - locally runs a web server that serves non-minified HTML, CSS, and JS files, including a reload mechanism to propagate file changes to the browser, making it easy to immediately see changes without performing a full page reload. Additionally, debugging is much easier because all source code is served unoptimized with the drawback of being slower.
- **production** - bundles, minifies, and optimizes the HTML, CSS, and JS files to be an efficient application that can be served through any web server. Debugging is much more complicated or even impossible due to the obfuscated source code.

³SPA is a web app implementation that loads only a single web document, and then updates the body content of it via JavaScript APIs [1]

2 Background

- **testing** - two approaches are available for testing. Unit testing, or in this context *component testing* and *end-to-end testing*. As the name component testing suggests, only a single component is tested at a time, e.g., the navigation component. End-to-end testing, on the other hand, is testing *everything*, multiple screens (including multiple components), optionally including communication with the backend.

2.1.3 Application Programming Interface - API

For the representation layer to be able to communicate with the processing layer, an interface must be defined to allow for data exchange. This interface is defined as a contract on how to communicate with each other using request and response - called API. There are different ways that APIs can be built, mostly those interfaces rely on HTTP(S):

REST APIs “Representational State Transfer“, which is the most popular and flexible way to create modern APIs. It defines a set of functions like GET, POST, PUT, and DELETE for a client to exchange data. The server uses the input to start internal functions and returns the output data to the client after processing [2].

WebSocket APIs enable real-time, bidirectional communication between clients and servers by establishing a persistent connection that remains open, allowing continuous data exchange [3]. Unlike the traditional request-response model of HTTP, WebSockets support full-duplex communication, meaning both parties can send and receive messages independently and simultaneously [4].

A common data format for both REST APIs and WebSocket APIs is JavaScript Object Notation (JSON) due to its lightweight nature and ease of parsing, compared to formats like Extensible Markup Language (XML).

2.2 Deployment and Development Tools

To put the source code into an executable program, be able to collaborate on the same codebase, or help developers with various tools to improve their experience, some other software is required.

2 Background

2.2.1 Docker

As a lightweight, reliable, and effective method of containerization, Docker significantly decreases complexity and issues when running modern applications, especially in the cloud. At the core of Docker's significance lies its ability to encapsulate an application and its dependencies into a single, portable container. This containerization ensures that the application runs identically regardless of the environment. Key to Docker's functionality are its core components [5]:

Docker Engine An open-source containerization technology for building and containerizing your applications. It serves the Docker Daemon & Docker Client.

Docker Daemon Listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.

Docker Client The primary way that users can interact with Docker. The docker command uses the Docker API for CLI calls.

Docker Images Serve as immutable templates for containers, ensuring consistency across different environments.

Docker Containers Instances of these images provide the isolated environments in which applications run.

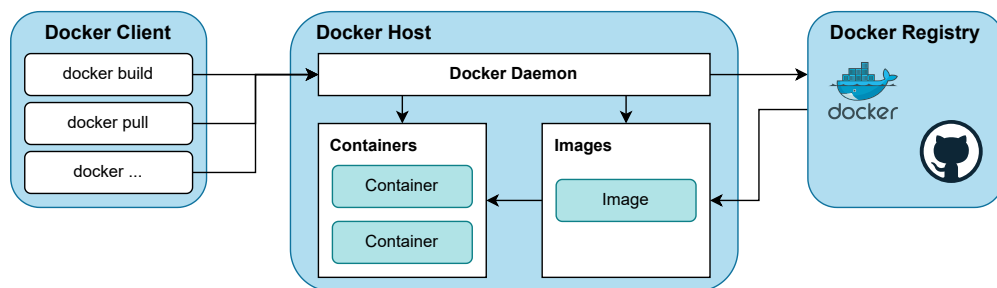


Figure 2.1: Docker architecture & components overview.

These elements work together to create, manage, and run containers (Figure 2.1). The use of Dockerfiles to automate the creation of images through a script of instructions enables reproducibility and version control, which is crucial for maintaining consistency in software delivery. For distribution of such Docker Images, a Docker Registry must be

2 Background

used to be able to serve an application using the Docker Daemon. Often, a local registry that is automatically installed with the Docker Engine can be used, but there are also many free options in the cloud, such as `http://hub.docker.com` to distribute Docker Images across the web.

2.2.2 Version control (Git) & GitHub

Git is a distributed version control system crucial for modern software development. It tracks changes, manages versions, and enables collaboration for multiple developers to work on a project simultaneously without conflicts. Each change is recorded in a commit, forming a detailed project history that allows reverting to previous states and comparing changes [6].

Git supports various collaborative workflows, such as pull requests and code reviews, enhancing team productivity. Platforms like GitHub facilitate these processes, enabling discussions and code reviews before merging changes into the main branch. Git integrates seamlessly with continuous integration and continuous deployment (CI/CD) systems such as GitHub Actions, automating testing, building, and deployment processes, enhancing software quality and delivery speed.

2.2.3 Frontend

The frontend consists of many different tools and software to be executable and testable. Status-quo before starting the thesis, the application was built using Node.js⁴ v14, Webpack for bundling, and WebdriverIO for testing. One goal of the thesis was to modernize this stack, changing the application to now use Node.js v22 (latest as of the thesis), Vite for bundling, and Cypress for testing. Additionally, the package manager for managing all dependencies has been exchanged from the default *npm* (node package manager - shipped with Node.js) with *yarn* to make better use of some caching behavior later needed for the automated builds.

⁴Node.js is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools, and scripts.

2 Background

Yarn is an established open-source package manager used to manage dependencies in JavaScript projects. It assists with the process of installing, updating, configuring, and removing package dependencies [7].

Webpack is a static module bundler for modern JavaScript applications. When Webpack processes the application, it internally builds a dependency graph from one or more entry points and then combines every module the project needs into one or more bundles, which are static assets to serve the content [8].

Vite.js is a build tool that aims to provide a faster and leaner development experience for modern web projects. It consists of two major parts: a dev server that provides rich feature enhancements over native ES modules and a build command that bundles the code with Rollup for production [9].

WebdriverIO is a progressive automation framework built to automate modern web and mobile applications. It simplifies the interaction with the app and provides a set of plugins that help one to create a scalable, robust, and stable test suite [10].

Cypress is a next-generation frontend testing tool built for the modern web [11].

ESLint statically analyzes code to find problems quickly. It is built into most text editors, and it can run ESLint as part of a continuous integration pipeline [12].

2.2.4 Backend

The backend consists mainly of the Spring Boot framework split up into multiple dependencies, e.g., `spring-boot-starter-web` as the basis of the framework, or `spring-boot-starter-security` used for user authentication, and many more. Additionally, the application is written using *Kotlin*. As a foundation for Kotlin, Java Development Kit (JDK) version 17 is required. For dependency management, building, and testing, the following tools are used:

Gradle Build Tool is a fast, dependable, and adaptable open-source build automation tool with an elegant and extensible declarative build language. Gradle is the most popular build system for the Java Virtual Machine (JVM) and is the default system for Android and Kotlin Multi-Platform projects [13].

2 Background

JUnit5 is the current generation of the JUnit testing framework, which provides a modern foundation for developer-side testing on the JVM. This includes focusing on Java 8 and above, as well as enabling many different styles of testing [14].

2.3 Xaminer Overview

The application consists of multiple screens, specifically built for different stakeholders, differentiating between publicly accessible resources and protected resources requiring a user sign-in. Currently, only a single instance at a time can serve a single institute (no multi-tenancy).

We differentiate between three roles:

Students do not require any authentication due to a unique identification key in the URL when joining an exam.

Supervisors which also require no authentication as they are also identified through a unique key in the URL, which is different from the students' identification.

Exam editors are the only stakeholders of the system requiring a sign-in as they can create, modify, and delete exams, manage students on the exam as well as can export the submissions for the students. Authentication is performed through a sign-in, creating a unique "session" on the backend once the user has successfully authenticated.

2.3.1 Architecture

As already discussed, Xaminer uses the server-client architecture pattern with a strong separation of business logic in the backend. Each functionality has a logical "domain" serving specific purposes, such as the `SubmissionController` handling requests of students submitting their results, but also serving endpoints for downloading such submissions for exam correction.

Endpoints can be either protected by the sign-in if the action performed needs proper authentication or can be put to "public" and potentially require some other way of identification based on the requester.

2 Background

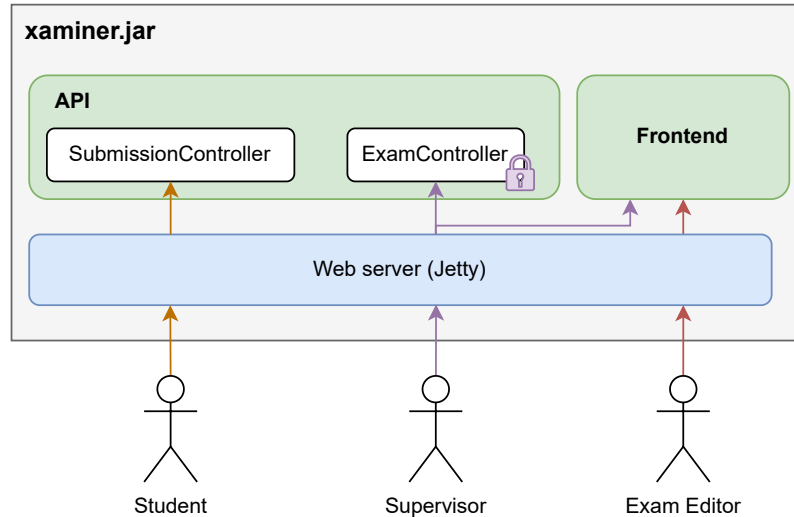


Figure 2.2: Xaminer high-level architecture overview.

Figure 2.2 represents some use cases that the application might serve. As we can see in the overview, the whole application is made available through the built-in web server to serve both the frontend and backend. This means that the API served through the backend can only be running as a single instance in the current setup, meaning that limited scaling options are available for load-balancing high demand on multiple applications to improve user experience. This somewhat limiting factor of the deployment will be further discussed in Section 2.3.2.

2.3.2 Deployment & Hosting

A critical aspect of the thesis is to change the setup of how the application is hosted on a server. The current setup as seen in Figure 2.3 is a .jar file - containing the backend and frontend - directly running on the host operating system (OS). This file is sent to the server from a local computer where Xaminer was built using a *Secure Shell (SSH)* connection to the remote host. After copying, if any existing instance is running, the process is killed, and the new .jar file is executed.

2 Background

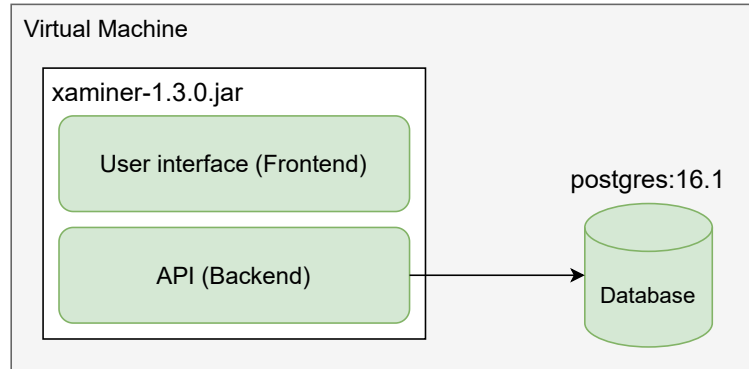


Figure 2.3: Current Xaminer hosting setup on a virtual machine.

Some concerns regarding this setup include:

- Direct access to the host OS causes potential issues if the application is being compromised, as it has full access to all information on the system it is running. Due to the lack of isolation, there is a certain security risk.
- Java and any third-party library must be installed on the host OS (e.g., `wkhtmltopdf` used for generating PDFs). Especially when setting up new instances, all the required software must be documented.
- Updates for libraries on the OS are rather inconvenient and would require central management especially if *Common Vulnerabilities and Exposures (CVE)* are published, depending on the severity one should act quickly mostly due to no isolation (as previously mentioned).

3 Modernize Frontend Stack

This chapter covers the changes performed to upgrade the core framework of the frontend. Upgrading the core framework used for building the application has many benefits for the development experience, user experience, and maintainability.

3.1 Vue 3 Upgrade

Upgrading the core framework for the frontend offers numerous reasons and benefits. One significant advantage is the maintainers' active support and ongoing development, which ensures timely bug fixes, enhancements, and new features essential in the fast-paced world of web development. Performance improvements are another key benefit, achieved through various optimizations leveraging the latest web standards, such as newer JavaScript standards and new APIs for rendering.

Additionally, the ease of online information lookup and documentation for Vue 3 is vastly improved, as online resources typically focus on the most recent version of the framework. This upgrade also introduces several recommended libraries and tools, further enhancing developer productivity and ensuring Xaminer remains future-proof for years to come. One major upgrade includes migrating to the bundling tool Vite.js (Vite), which will be discussed in detail in Section 3.2. Furthermore, Vue 3's Composition API provides more flexible and scalable code organization, making it easier to manage larger applications. The developer experience is enhanced with improved tooling, streamlined workflows, and better integration capabilities.

3.2 Bundling Tool

The suggested setup when using Vue 3 is to use Vite as a bundling tool [15]. When setting up a completely new Vue application via `create-vue`¹, it will automatically use Vite in the background per default. As Vite was created by the same person as the Vue framework, it provides first-class support for Vue Single File Component (SFC). This means, that the old bundling tool Webpack had to be replaced during the migration, causing little to no issues except finding some alternatives for custom logic and 3rd party libraries. To better understand the benefits of this upgrade, we will look at the technical difference between these two bundling tools.

Webpack is a powerful and widely-used module bundler designed to transform and compile modern JavaScript applications. It plays a critical role in the frontend development ecosystem by enabling developers to manage and optimize the assets and dependencies of their applications. At its core, Webpack takes a central configuration file (`webpack.config.js`) and an entry point, then traverses the dependency graph of the application, bundling the various modules, assets, and resources into one or more output files [16] (Figure 3.1). This process not only consolidates the code but also optimizes it for performance, making it ready for deployment. Additionally, Webpack is excellent at maximizing the output for usage in production as it includes advanced features like tree shaking and code splitting.

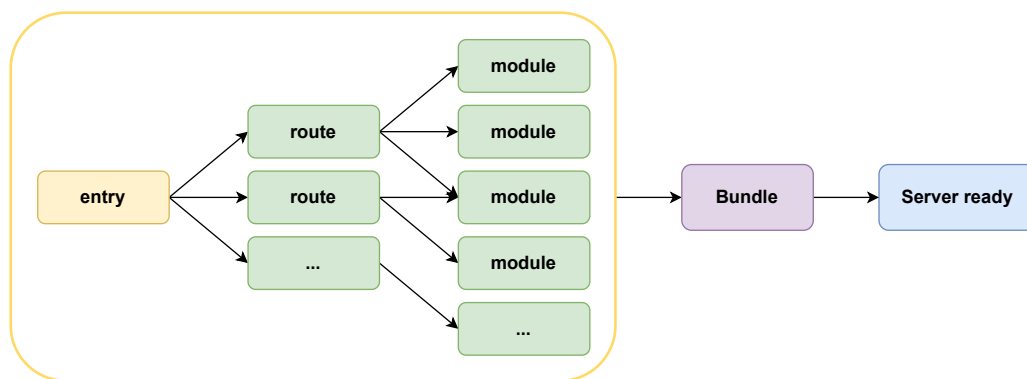


Figure 3.1: Life-cylce of Webpack’s bundling mechanism.

¹<https://github.com/vuejs/create-vue>

3 Modernize Frontend Stack

Vite.js is an advanced build tool that addresses many limitations of traditional bundlers, offering several core features that enhance the development experience. It achieves far faster builds and real-time module updates, which considerably shorten development cycle times, by utilizing native ECMAScript Modules (ESM) and modern browser capabilities [16] (Figure 3.2). Instead of bundling one or more output files that are served to the browser, Vite only transforms and serves source code on demand, letting the browser handle the job of a bundler. Native ESM, therefore, represents a significant advancement in JavaScript’s module system, providing a standardized and efficient way to organize and manage code by improving `import` and `export` statements to define and utilize dependencies, facilitating modularity and reusability.

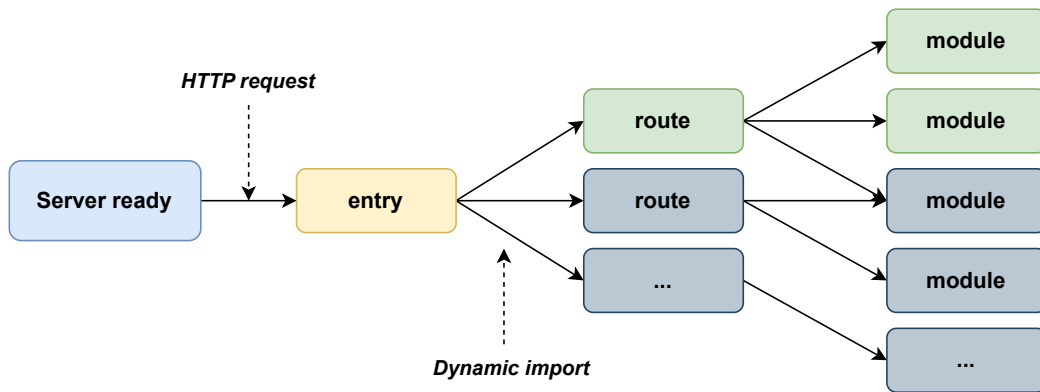


Figure 3.2: Life-cycle of Vite’s bundling mechanism using Native ESM.

Vite’s zero-configuration setup simplifies the initial project setup, making it accessible and efficient for developers. What this means is that little to no configuration is required, as the default settings provide a perfect baseline to get started. Vite also supports hot module replacement, allowing changes to be reflected instantly without full page reloads, thereby improving productivity [16].

In conclusion, using Vite over Webpack offers several compelling advantages. The biggest advantage of Vite is the significantly faster build time and a smoother development experience by leveraging native ESM making it a more modern and efficient choice for building web applications in 2024.

3.3 Migration Path

The migration from Vue 2 to Vue 3 was achieved very straightforwardly by following the official migration guide² while also checking and fixing various errors when trying to build the application. In the following Section 3.3.1, we detail the most complex challenges of the migration from Webpack to Vite. For that, the existing configuration had to be ported to the new build tool, where one bigger challenge needed to be tackled during migration.

3.3.1 Dynamic Route Generation

Based on the folder structure and component names, custom logic using Webpack automatically generates paths that a user can navigate to. For example, a component located in `src/views/demo.ts` with a corresponding `src/views/demo.vue` automatically generates a path `https://xaminer.jku.at/demo.html`.

When using Vite, instead of implementing this custom logic to handle it, a library called “unplugin-vue-router” can be used to provide dynamic route generation based on the folder structure. In addition, it creates typings for the TypeScript components so no typing errors occur when redirecting programmatically between pages.

In Xaminer’s old version, using Webpack, each route generated its own HTML file containing multiple JS files split up into the following structure:

- `<page-path>.js` self-written code for the page navigated to such as handling the exam submission API logic.
- `chunk-commmon.js` a JS "chunk" containing self-written, shared code such as the header navigation or footer component of the page.
- `chunk-vendor.js` external code for the Vue framework and other libraries required for rendering the page.

Figure 3.3 shows that each route a user navigates to initially loads the HTML, then the linked JS files before rendering the actual page.

²<https://v3-migration.vuejs.org>

3 Modernize Frontend Stack

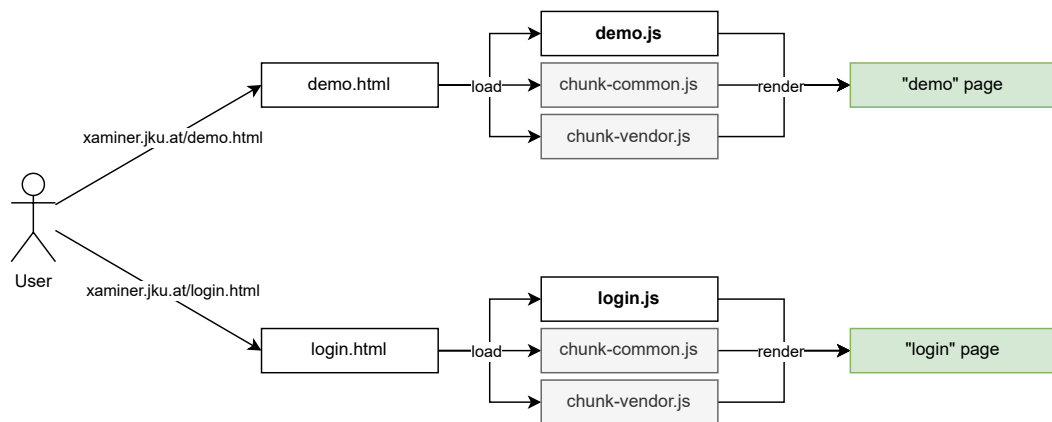


Figure 3.3: Page loading behavior before migrating to Vue 3.

The new approach uses only a single HTML file (`index.html`) that renders the content based on programmatically detecting the path in the URL. This dynamic detection of the route and rendering of the corresponding page is handled by the “vue-router”. This allows for improved performance due to decreased loading times for HTML files and better optimizations of the JS files.

The structure changes to the following:

- *index.html* containing the very basic HTML structure used to apply the page content dynamically based on the route. This file is only loaded **once** when initially opening the Xaminer application.
- *index.js* containing the Vue framework, other external libraries, shared code, and paths to other JS “chunk” files.
- *<chunks>.js* containing logic for self-written components. These chunks are determined by the bundler on build time - usually, each Vue component (TypeScript file) is translated into a chunk.

Compared to Figure 3.3, Figure 3.4 shows that only a single HTML file is required for loading the page, which loads an initial JS file holding links to all other JS files needed to render the page. Upon calling a specific route, the browser automatically loads the file in the background before rendering the page.

3 Modernize Frontend Stack

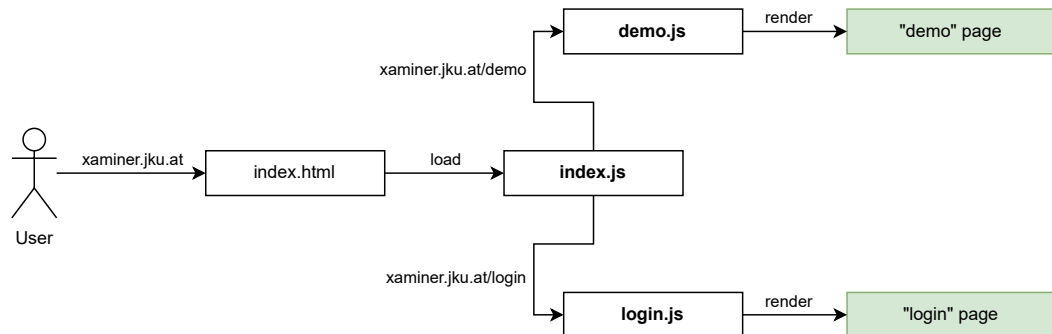


Figure 3.4: Page loading behavior after migration to Vue 3 using “vue-router”.

This allows for improved navigation speed between pages while also reducing the amount of requests for the same files. Additionally, after loading the initial JS file, the loading mechanisms of the framework can be optimized because the browser is already aware of all possible combinations of required JS files on navigation. The old approach required a reset of this for every page change as the browser performs a full reload (reducing the performance and increasing the load on the server) on navigation.

3.4 Testing Tool

In addition to upgrading the framework and bundler, the Vue documentation also highly recommends the use of Cypress for testing. This means that the existing testing framework WebdriverIO, including all test cases, had to be migrated. Even though the test cases remained the same, the syntax for each case had to be rewritten entirely. Additionally, Cypress is limited by exactly one browser window, requiring a complete re-write of all test cases that involve interactions by multiple users, e.g., joining the exam and reactivating a student’s exam.

Syntax change mainly included two steps:

Refactoring of *.page.ts files, which separates the testing logic from how an element can be retrieved from the *Document Object Model (DOM)*. Under some circumstances, getting the element from the DOM required some custom logic, e.g., receiving the exact

3 Modernize Frontend Stack

row and column of a table. For reference, Listing 3.1 and Listing 3.2 should provide an example of the performed changes, each retrieving the “title” of the login page by using their unique ID.

```
1 get heading () {
2   return $('#index-content #login-header')
3 }
```

Listing 3.1: Old *.page.ts syntax (Webdriver.IO)

```
1 get heading () {
2   return cy.get('#index-content #login-header')
3 }
```

Listing 3.2: New *.page.ts syntax (Cypress)

Change of *.spec.ts is required to make Cypress “understand” our test cases again. Every assertion statement needed a syntax change, also sometimes including more refactoring and additional chaining of statements. Further, file names had to be changed from “*.spec.ts” in favor of “*.cy.ts”.

For reference, Listing 3.3 and Listing 3.4 showcases an example of the changes.

```
1 it('should open and render welcome text (not logged in)', async () => {
2   await expect(Login.heading).toHaveTextContaining('Please log in.')
3 })
```

Listing 3.3: Old *.spec.ts syntax (Webdriver.IO)

```
1 it('should open and render welcome text (not logged in)', () => {
2   Login.heading.should('contain.text', 'Please log in.')
3 })
```

Listing 3.4: New *.cy.ts syntax (Cypress)

Instead of different methods such as `toHaveTextContaining()`, Cypress **always** requires `should()` with different assertion parameters³ like “contain.text” that can be further chained with other `should()` statements. Secondly, no `async-await` or `expect` method is needed anymore, simplifying the source code required for testing.

³<https://docs.cypress.io/guides/references/assertions>

4 Deployment and Hosting

This chapter covers the steps performed to create an easy-to-deploy and secure way of hosting Xaminer while also adding automation to all steps required for testing and building the application using GitHub Actions.

4.1 Containerization

When talking about “containerization”, we are referring to creating an executable program that can be run using Docker. Creating such programs, namely images, always follows the same principles, with the exception that the building steps defined in the `Dockerfile` are different based on the source code, application type, and programming language.

The basic steps for creating an image include the following, which are defined in a single file called `Dockerfile` (Listing 4.1).

Line 1 **define “base image”** which is usually chosen based on the programming language, e.g., for Java, you would be choosing any OpenJDK image with the desired Java version.

Line 2 **COPY the source code** into the image to later execute build commands.

Line 3 **RUN build commands** to build the actual application, e.g., a `.jar` for Java, inside the image.

Line 4 **defining the command for starting the application** to tell Docker what command it should use when starting the container to also execute the application, e.g., `java -jar xaminer.jar`.

Build Steps

The so-called build step(s) are crucial as they define how the application must be built. For Xaminer, Gradle is used for the build system, which means to create an executable .jar file, a Gradle command is defined. In Listing 4.1, line 3, we can see that the gradle bootJar command is run to build the application.

```
1 FROM gradle:8.8-jdk17 as builder
2 COPY /xaminer-backend/ /
3 RUN gradle bootJar
4 ENTRYPOINT ["java", "-jar", "/build/libs/xaminer-backend.jar"]
```

Listing 4.1: Basic Dockerfile

Depending on the application, multiple build steps could be required to build the application, split up into different build commands. Those commands are always run inside the image that is created, which means no source code, folder structure, or resources from the host file system are modified but only the files inside the image that were copied before. Each instruction in the Dockerfile translates to a layer in the final image, which receives a unique hash that can be used for caching. If the instruction has not changed since the last time the image was built, the cached layer will be used instead of running the corresponding command.

With that knowledge, several optimizations can be performed to reduce the final size of the image while also trying to maximize cached layers for improved build times of the image, e.g., by combining multiple RUN lines into a single line.

4.2 Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It allows users to configure the application's services, networks, and volumes through a single YAML file, typically named "docker-compose.yml". With Docker Compose, one can manage the lifecycle of the application with simple commands, such as starting, stopping, and rebuilding services. This allows an Xaminer instance to be started with a single command in a secure, platform-independent environment.

4 Deployment and Hosting

The `docker-compose.yml` (Listing 4.2) includes the backend application with database, the frontend, and a reverse proxy that allows HTTP traffic to be forwarded into the Docker network while also providing certificate management for HTTPS.

```
1 services:
2   caddy:
3     image: caddy:2.7-alpine
4     ports:
5       - 80:80
6       - 443:443
7     depends_on:
8       - frontend
9       - backend
10    volumes:
11      - ./docker/caddy/Caddyfile:/etc/caddy/Caddyfile
12
13   frontend:
14     image: ghcr.io/neonmika/xaminer/frontend:1.3.0
15
16   backend:
17     image: ghcr.io/neonmika/xaminer/backend:1.3.0
18     depends_on:
19       - postgres
20
21   postgres:
22     image: postgres:16.3
```

Listing 4.2: Xaminer `docker-compose.yml` (simplified)

The `services` section defines all the containers we want to execute when running `docker-compose up` to start the application. Each service has different configurations, e.g., port forwarding, mounting a volume from the host OS, or requiring another service to be running before starting. For the Xaminer application itself (frontend & backend), we can identify the GitHub Registry “`ghcr.io`” that contains the images after automatically building them using a GitHub Action (Section 4.3). If no registry, like `ghcr.io`, is present, the default registry from Docker (`https://hub.docker.com`) is used.

4 Deployment and Hosting

4.2.1 Backend & Database

The backend application of Xaminer is the container containing the executable `.jar` file. Of course, it requires a database to be running to store all data related to users, exams, and submissions. Before being able to start the application, we *require* the database to be running, which is indicated by the `depends_on` attribute. The database itself, in our case, PostgreSQL, is also running inside the Docker network and is isolated from the host OS. Additionally, the backend application is also isolated from the host OS, which means no HTTP(S) traffic can reach the application with this configuration. To forward the traffic to it, we will use a reverse proxy, which will be explained in Section 4.2.3.

4.2.2 Frontend

The frontend of Xaminer is the container containing our production-build Vue application running inside a *NGINX* container. *NGINX* is the most commonly known and used webserver, very lightweight, and secure. This also means that the frontend is now running independently of the backend, removing the dependency on it for hosting. Similarly to the backend, no traffic is being forwarded to the container itself. Instead, it is run completely isolated from outside traffic, requiring the reverse proxy to forward traffic.

For building, the application is first built using Vite in “production” mode to export HTML, CSS, and JS files into the `dist/` folder that is further copied into the *NGINX* container for serving the content.

4.2.3 Reverse Proxy

A reverse proxy is, in general, a component that sits between client devices and backend servers, forwarding client requests to the appropriate service and returning the server’s response to the client. It handles inbound requests from the internet to a server or multiple servers in a private network - in our case the Docker network. It enhances security, load balancing, and performance by distributing client requests among multiple servers or instances of an application, caching content, and masking the identity and structure of the backend servers.

4 Deployment and Hosting

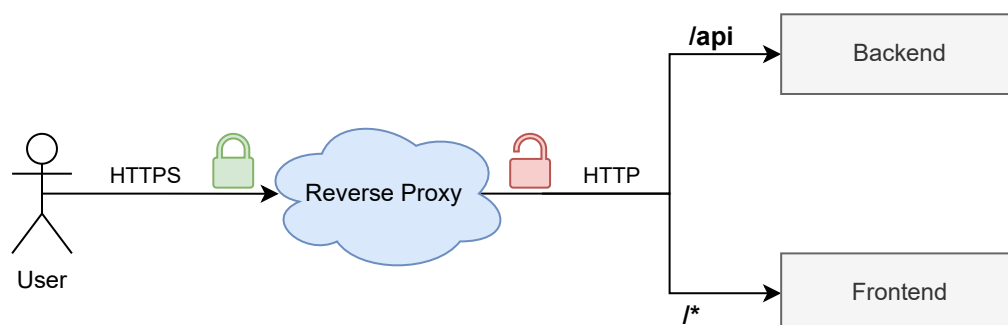


Figure 4.1: Usage of a “Reverse Proxy” for the new hosting setup.

The reverse proxy, using “Caddy”, is configured in a way that all traffic received is split up into two paths (based on Figure 4.1):

- `/api` traffic that indicates the API path will automatically be forwarded to the backend.
- `/*` every other request will be forwarded to the frontend.

Further, the traffic received by the reverse proxy is always HTTPS which is decrypted using TLS termination once traffic is forwarded to the corresponding application.

To run this setup, as defined in Listing 4.2 line 2-11, both port 80 (HTTP) and 443 (HTTPS) need to be forwarded to the host OS so that traffic can be received by the proxy from the internet. Of course, the proxy should also only be started if the backend and frontend are running (dependency defined by `depends_on`). Lastly, the configuration for Caddy is mounted from the host OS using the `volumes` attribute to mount the “Caddyfile” located in the repository, mounted and replaced inside the container on `/etc/caddy/Caddyfile`¹

¹<https://caddyserver.com/docs/running#setup>

4.3 Automated Builds

One important aspect of modern software development is, to have automation wherever possible for building and testing applications. When using Docker, automating the process of building images allows for faster software shipping without requiring developers to care about “how to build” such images. Additionally, we can guarantee a certain quality of the application if automated tests verify the functionality before actually shipping a new version.

4.3.1 GitHub Actions

GitHub Actions² are a convenient, built-in automation tool for GitHub (where the Xaminer repository is hosted) that seamlessly automates building and testing on every commit or pull request. Integrating with a hosted registry for shipping Docker images, we can create an automation pipeline that consists of multiple steps for creating a shippable version of Xaminer. Figure 4.2 shows the pipeline, consisting of seven standalone jobs that might run in parallel or require results of previous jobs, e.g., the “E2E Test” is only executed once the previous jobs “Frontend docker image” and “Backend docker image” could be executed successfully.

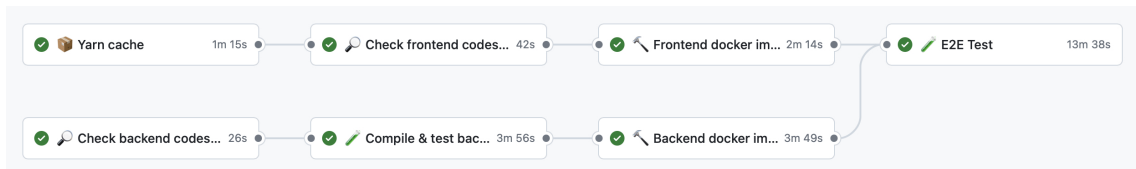


Figure 4.2: Implementation of the GitHub Action workflow for Xaminer.

Currently, the conditions for automatically triggering such a pipeline are:

1. **Pull requests** - opening a pull request or adding new commits.
2. **Branch** - adding commits to the “main” branch.
3. **Tag** - creating a tag for releasing a new version of Xaminer.

²<https://docs.github.com/actions>

4 Deployment and Hosting

Each of the seven jobs performs a dedicated action explained in the following list:

- **Yarn cache** - Downloads and creates a shareable cache consisting of all frontend packages and Cypress for running the E2E tests, shared with three consecutive jobs. This way, the dependencies are only downloaded once instead of three times improving the speed of the pipeline.
- **Check frontend codestyles** - Checks for code stylings of all frontend files (.js, .ts, .vue, .html) to maintain consistency, readability, and quality in the codebase
- **Check backend codestyles** - Checks for code styling of all backend files (.kt) for the same reason as checking it in the frontend.
- **Compile & test backend** - Compiles the backend to run JUnit for executing all unit- and integration tests using Gradle.
- **Frontend docker image** - Builds the frontend Docker image and saves it as an artifact, which is shared with the “E2E Test” job. Optionally, when creating a *Git Tag*, the image is pushed to the GitHub registry.
- **Backend docker image** - Builds the backend Docker image and stores it the same way as the “Frontend docker image” job.
- **E2E Test** - Runs the Xaminer application to perform end-to-end tests using Cypress.

For example, Listing 4.3 shows how such a job in the pipeline is defined. The syntax for creating a pipeline is done using *YAML*³ and is defined using the following structure:

Step 1 define a meaningful name, which will be shown in the overview.

Step 2 decide on which system the job should be executed. This can be either systems hosted in the cloud by GitHub or self-hosted machines.

Step 3 (optionally) defines an array of jobs that must be successfully executed before executing the job. In our case, we require the job for creating the frontend dependency cache to download its artifacts.

Step 4 the actual steps that the job should execute, e.g., checking out the repository, downloading the cache, etc.

³YAML is a human-readable data serialization language commonly used for configuration files

4 Deployment and Hosting

```
1 lint-frontend:
2   name: Check frontend codestyles
3   runs-on: ubuntu-latest
4   needs: ['init-frontend-cache']
5   steps:
6     - name: Checkout
7       uses: actions/checkout@v4
8     - name: Download node_modules cache
9       uses: actions/cache/restore@v4
10      with:
11        key: node_modules-${{ hashFiles('yarn.lock') }}
12        path: |
13          xaminer-frontend/node_modules
14          ~/.cache
15     - name: Setup Node
16       uses: actions/setup-node@v4
17       with:
18         node-version-file: 'xaminer-frontend/.nvmrc'
19     - name: Lint
20       run: cd xaminer-frontend/ && yarn lint
```

Listing 4.3: GitHub Action job definition for “Check frontend codestyles”

The actual steps define a name and optionally a uses key for using pre-built workflows to easily set up libraries and environments, such as installing and setting up the frontend runtime Node.js. As we use some shared, cloud-hosted machines from GitHub, jobs are completely stateless, meaning that all data (if not stored as an artifact) will be wiped after a job ends. Upon job execution, we are also presented with a completely “empty” machine without any installation of our required libraries. Therefore, having such pre-built workflows for setting up the environment speeds up the development of such pipelines drastically.

If no uses key is defined, we are presented with a plain shell to run any command. Line 22 of Listing 4.3 shows two commands, one changing to the `xaminer-frontend/` folder to run `yarn lint` to check the code stylings for potential violations.

4 Deployment and Hosting

Testing

One of the most important aspects of software is testing. When performing changes to the software, it is expected that no existing functionality will be negatively impacted by the change. To verify this, unit tests, integration tests, or end-to-end tests need to pass successfully before being able to merge any change. Previous projects on Xaminer already covered a wide variety of test cases for both backend and frontend. In the backend, mostly unit tests and integration tests are available, providing coverage of all API endpoints consumed by the frontend. The frontend, on the other hand, covers end-to-end flows, meaning that a “real” instance of the backend is running when performing tests.

These tests cover most supervisor and student journeys such as logging in to Xaminer, creating exams, submitting results, and downloading the results.

Backend testing is performed using Gradle and an instance of the database that is started before executing the tests.

Frontend testing is done using Cypress, with a running instance of the backend (including the database). These tests are executed after successfully executing the backend tests and building a Docker image.

Building

For building, we also differentiate between backend and frontend builds. Both jobs use the same foundation for automated Docker image builds using *Docker Buildx*⁴ which is a CLI for building images using *BuildKit*⁵, a concurrent and cache-efficient builder toolkit.

After building, images are pushed to a public registry on GitHub (`ghcr.io`), where institutes or system administrators can access them for running a Xaminer instance on their server through the Docker Compose. Images are only pushed if a *Git Tag* can be associated with the build to keep proper versioning. For single commits on the “main” branch or when opening pull requests, images are only copied to other jobs of the pipeline but will not be uploaded to the registry.

⁴<https://github.com/docker/buildx>

⁵<https://github.com/moby/buildkit>

4 Deployment and Hosting

Artifacts

Artifacts, in this context, consolidate files and information related to a specific job in the pipeline. Such artifacts can be shared between the independent jobs inside a pipeline, as previously discussed, or can also be used to show some results of the tests as shown in Figure 4.3.

Cypress Results






Result	Passed 	Failed 	Pending 	Skipped 	Duration 
Passing 	70	0	0	0	715.467s

Figure 4.3: Cypress job result of passed, failed, pending, or skipped tests and duration.

Some of these artifacts include:

Yarn & node_modules caches created by the “Yarn cache” job, which stores all required dependencies for all related frontend jobs (linting, testing, and building).

Docker caches for each build job to speed up the building process.

Docker images are uploaded as an artifact if no tag is created to share the built image with other consecutive jobs.

Testing results and (optional) screenshots to identify why a certain test failed, including logs for the specific failed run.

5 Bidirectional Communication

This chapter introduces mechanisms and standards in the browser for bidirectional communication. The goal of this is, to build a generic foundation for other use cases to come. As a proof-of-concept, the supervisors' administration page will be extended for receiving live updates of certain student actions, such as submitting the exam.

5.1 Technology Evaluation

Efficient communication between clients and servers is a key factor in system performance. Request-response polling, where the client frequently sends requests to the server to check for updates, can be resource-intensive and introduce delays. On the other hand, bidirectional communication offers a more efficient solution, allowing both the client and server to exchange data in real-time over a persistent connection. This method minimizes latency and optimizes resource usage.

Xaminer has many use-cases for using such an efficient, bidirectional communication channel to provide instant feedback to supervisors and students. To name one example, supervisors currently have a simple overview of all students who attend the exam, indicating if they have already joined the exam and if they have already submitted it. A good quality-of-life improvement would be to provide real-time updates of such events to the supervisor without requiring constant polling or reloading.

To evaluate which technology to use for bidirectional communication, it is important to have a basic understanding of the requirements while also knowing what technologies are available. In our case, the initial requirements would only need a one-way live update mechanism to notify supervisors of live updates of the exam. On the other hand, we want to allow a future-proof concept that also allows bidirectional communication, meaning that on-way updates are not suitable for our proof-of-concept.

5 Bidirectional Communication

For our evaluation, three well-known concepts can be taken into consideration:

WebSockets provide a full-duplex communication channel over a single TCP connection, allowing for real-time, two-way interaction between a client and a server [4]. Unlike traditional HTTP requests, WebSockets enable servers to send data to clients proactively without waiting for a client request, making them ideal for applications that require instant data updates, such as chat applications. This protocol reduces latency and overhead compared to polling methods, as it maintains a constant connection, thus ensuring efficient and seamless communication [3].

Server Sent Events (SSE) is a technology that allows servers to push real-time updates to clients over a single, long-lived HTTP connection. Unlike WebSockets, SSE only supports one-way communication from server to client, making it well-suited for applications that need to display live data updates, such as live news updates. SSE is easy to implement and uses the standard HTTP protocol, providing a reliable way to push continuous updates without requiring the client to request new data repeatedly [17].

Long Polling is a technique used to achieve real-time updates in web applications by keeping an HTTP connection open until new data is available. When a client requests the server, the server holds the connection open until it has new information to send. Once the client receives the response, it immediately sends a new request, creating a near-continuous loop of requests and responses. Although more resource-intensive than WebSockets and SSE, Long Polling is a simple method to implement real-time communication and is compatible with older browsers and network configurations that do not support more advanced protocols [18].

As we can see, SSE is just a one-way WebSocket connection, which does not fulfill our requirements. Long polling, on the other hand, would offer similar capabilities as WebSockets but would be more resource-intensive, making it a non-ideal solution for us. This leads to the conclusion that WebSockets are the most efficient, reliable way of handling our bidirectional communication. To have maximum compatibility with older browsers as well, we can make use of a library called Socket.IO that provides more robustness while also improving connection establishment.

5.2 Socket.IO

Based on the evaluation, choosing Socket.IO¹ as a library is an excellent solution as it provides a robust and versatile approach for data transmission. It seamlessly integrates WebSockets and falls back to other methods like Long Polling when necessary, ensuring reliable connectivity across various browsers and network conditions [19]. This automatic transport selection and fallback mechanism guarantees continuous and efficient communication without requiring manual adjustments. Its straightforward API further enhances ease of use, making it an ideal choice for implementing real-time features such as live chats, notifications, and collaborative tools in web applications.

5.2.1 Connection Lifecycle

The connection lifecycle in Socket.IO revolves around establishing and maintaining real-time communication between clients and servers. It begins with a client initiating a connection request to the server through HTTP. Upon receiving this request, Socket.IO attempts to establish a WebSocket connection. If successful, a persistent connection is maintained, enabling efficient bidirectional data exchange. During this phase, clients and servers can emit and listen to events, facilitating real-time updates and interactions [19].

Socket.IO handles disconnections gracefully, automatically attempting to reconnect if the connection is lost due to network issues or server downtime. It employs exponential backoff strategies to minimize connection retries and optimize network usage. This life cycle management ensures robust and reliable communication, crucial for applications requiring continuous updates and responsiveness [19].

5.2.2 Namespaces

In Socket.IO, namespaces provide a way to segment and organize communication channels within a single WebSocket connection (Figure 5.1). Each namespace acts like a separate communication scope, allowing clients and servers to subscribe to specific channels of

¹<https://socket.io>

events. This segmentation is useful for applications that require different types of real-time interactions or wish to isolate functionality. For example, for Xaminer, we might use namespaces to separate public exam chat rooms from private one-to-one conversations. By offering this organizational structure, Socket.IO namespaces enhance scalability, maintainability, and security by controlling the flow and context of real-time data exchanges within a single connection [20].

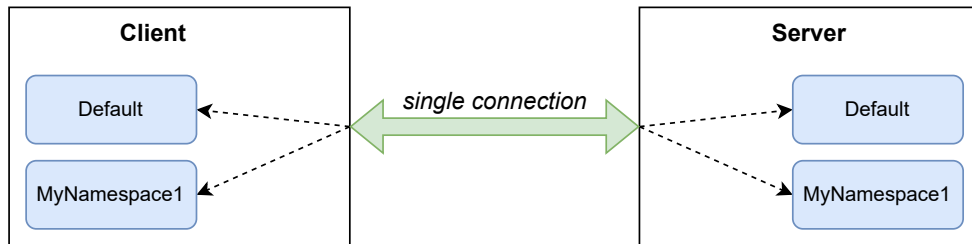


Figure 5.1: Socket.IO Namespace architecture overview.

5.3 Use Cases

The implementation allows for various use cases, such as a notification system for the supervisor if students joined the exam, if they submitted their results, or potentially a chat functionality.

One important aspect of any use case is how users are authenticated determining if users are allowed to join a certain namespace. We are differentiating between two concepts of namespaces, mainly “public” and “protected” namespaces. Socket.IO does not take care of this logic, but when joining multiple namespaces, it efficiently re-uses existing connections only requiring a single connection.

Public namespaces can be joined without any authentication in the context of an exam. This would include chat capabilities within the exam where all students can read and write messages.

Protected namespaces can only be joined if certain conditions for authenticating are fulfilled. This can be any custom mechanism to verify that a specific user is allowed to join the namespace as sensitive data is exchanged, e.g., who submitted the exam.

5 Bidirectional Communication

```
1 fun initProtectedNamespace(examId: Long) {
2     val namespace = getNameSpace(examId)
3     server.addNamespace(namespace)
4
5     server.getNamespace(namespace)?.addConnectListener {
6         val privateKey = it.handshakeData
7             .urlParams["privateKey"]?.get(0)
8
9         if (
10            privateKey == null ||
11            examService.findByPrivateKey(privateKey) == null
12        ) {
13            it.disconnect()
14        }
15    }
16 }
```

Listing 5.1: Adding a protected namespace

Listing 5.1 should provide an overview of how such protected namespaces can be added to the server. This method, specifically, initializes a protected namespace based on the `examId` used for sending notifications to supervisors during an exam. For identification, the `privateKey` from the URL parameters is read and checked for existence. Such a URL containing the `privateKey` can only be retrieved by a supervisor, meaning that students cannot join such namespaces without it. If no corresponding exam could be found, clients are immediately disconnected, meaning that they are unable to receive any events from the server.

Upon successfully connecting to the namespace, clients can receive and send data. As a first proof of concept, the backend is the only emitting component in this scenario, meaning that connected clients are only passively listening. To differentiate between events, a *unique name* is required to implement listeners performing some business logic if a certain event is encountered.

5 Bidirectional Communication

```
1 const socket = new io("/exam-supervisor-namespace", {
2   query: { privateKey: this.examPrivateKey }
3 })
4
5 socket.on("SUBMISSION", ({ matNr }) => {
6   this.notificationStore.show(`${matNr} has submitted`, 'success')
7 })
```

Listing 5.2: Socket.IO listener in client (simplified)

Listing 5.2 shows how clients can connect and listen to topics. The first step is to connect to the supervisor namespace with a `privateKey` as authentication. If the connection has succeeded, the client now listens for `SUBMISSION` events, including a handler that will show a notification popup upon receiving the event. The event data is custom-defined, similar to a REST API response. In our example, it contains an object holding the `matNr` of the submitting student.

There is no limit on how many events a client can listen to, nor are there restrictions to the data format an event can have. The first level of separation is the *namespace* one is connecting to, and secondly, the *event* the client is interested in. There might be more events sent on the namespace, but as long as the client is not listening to it, those are not processed.

6 Question with Sub-questions

This chapter covers the quality-of-life-improvement for both students and lecturers by introducing a new type of question that allows an exam to have questions with sub-questions, e.g., 1a) 1b) 1c), especially focusing on the improvement to visual hierarchy of these elements.

6.1 Motivation

In response to feedback from lecturers, a highly requested feature is the ability to support complex questions with sub-questions, where multiple related questions share a common description and contribute to an aggregate score. Implementing this functionality requires developing an intuitive editor for question creation and ensuring clear, structured formatting during the examination process.

The most important aspect of this feature is to be able to use close to all existing application logic and types of questions available, e.g., `CODE_QUESTION` (questions requiring source code input, e.g., Java), `SINGLE_CHOICE_QUESTION` (questions that require selecting a **single response** from a list of options), or `MULTIPLE_CHOICE_QUESTION` (questions that require selecting **multiple responses** from a list of options.). Additionally, it must be built in a way that adding new types of questions can easily be achieved without needing to adjust the core concept of sub-questions. This means that the goal of the backend and frontend is to make use of existing classes and components for implementation. Further, for grading, all points should be automatically calculated for both students in the exam view and on the printed submission for professors.

6.2 Implementation

The implementation for this new type of question mainly uses recursiveness in the backend and frontend allowing not only “sub-questions” but also “sub-sub-questions”, “sub-sub-sub-questions”, etc. without any technical limitations. The only restriction will occur in the frontend when adding more and more recursive sub-questions as the user interfaces will be unusable at some point.

6.2.1 Recursiveness in the Backend

The backend introduces a new `BlockKind` which determines how a certain question should be rendered in the frontend and defines how the data should be deserialized based on the base class (`Block.kt`) and a corresponding sub-class, e.g., `BlockKind = SINGLE_CHOICE_QUESTION` instantiates a `ChoiceBlock.kt` containing additional data for the specified question type.

The newly introduced `COMBO_QUESTION` instantiates a `ComboBlock.kt` (Listing 6.1) containing a list of `Blocks` allowing a recursive add of all other blocks. For this, the `Block` entity must be extended by the `block_id` to allow this recursive relation.

```
1 @Entity
2 class ComboBlock(
3     @OneToMany
4     @JoinColumn(name = "block_id")
5     var blocks: MutableList<Block> = mutableListOf(),
6 ) : Block(BlockKind.COMBO_QUESTION)
```

Listing 6.1: `ComboBlock.kt` for sub-questions

6 Question with Sub-questions

Through a @OneToMany relation, the framework automatically joins the column when selecting data. This connection can be further inspected in the UML Diagram (Figure 6.1).

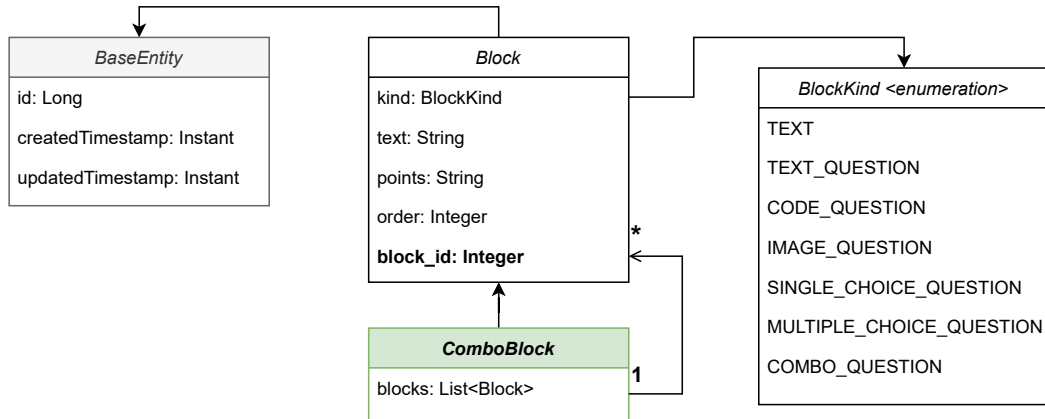


Figure 6.1: Structure of the newly introduced ComboBlock for sub-questions.

6.2.2 Recursiveness in the Frontend

To render questions recursively, multiple changes were required to the existing logic as it was only designed for one level of questions using the position (index) in the array as a unique identifier. Additionally, various stylings were adjusted to create a visual differentiation of sub-questions by applying additional spacings. Lastly, various issues had to be resolved when recursively importing the components for questions as Vue, by default, imports components on usage without “global” component registration, causing recursive loads of the same components even though they were already registered in a different context.

When applying elements to the DOM of the webpage, they need to have a unique ID that can be used to reference the question block for the results and apply the text editor. The index of the question in the array provided a unique id before adding sub-questions, now the actual id of the Block in the database is used. Upon creating an exam where no IDs are available, the *current timestamp* is used as a temporary identifier as its uniqueness can be trusted on a small scale.

6 Question with Sub-questions

For visualization, each question first renders a `BlockVisualizationHeader`, i.e., the question's header containing the question's description, and then decides based on the `BlockKind` if it should render the question using the `UniversalBlockVisualization` or, if `BlockKind == COMBO_QUESTION` it should recursively call itself with the blocks contained in the question block (Figure 6.2).

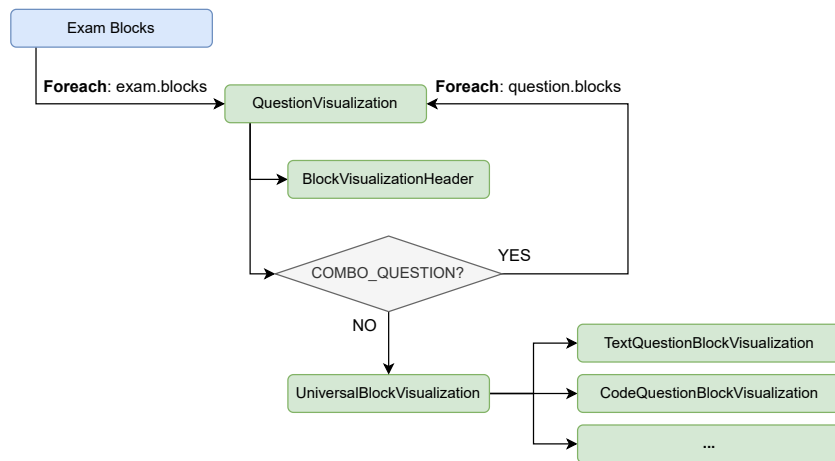


Figure 6.2: Component call graph for the recursive rendering of components in the frontend.

7 Question Search

This chapter covers the search functionality for questions when creating exams as a lecturer. It allows one to search for any type of question in all existing exams, making it easy to copy questions from past exams to the one currently being edited.

7.1 Motivation & Concept

Currently, lecturers encounter difficulties if questions from past exams want to be reused due to the lack of a dedicated search and copy functionality. Without the ability to search across all existing exams, finding and reusing questions becomes a manual, time-consuming task as they have to be copied to the editing exam by selecting the text from the old one and, depending on the question type, additional answers. Implementing a search feature will greatly simplify this process, allowing lecturers to quickly locate relevant questions by typing a simple search term, reducing the effort involved in question creation, and improving the overall workflow of exam preparation.

Conceptually, the goal is to allow a search mechanism in the database table (Figure 7.1) for a given search term (string) which can be utilized by the frontend for copying them into the opened exam without modifying the existing question in the database.

7 Question Search

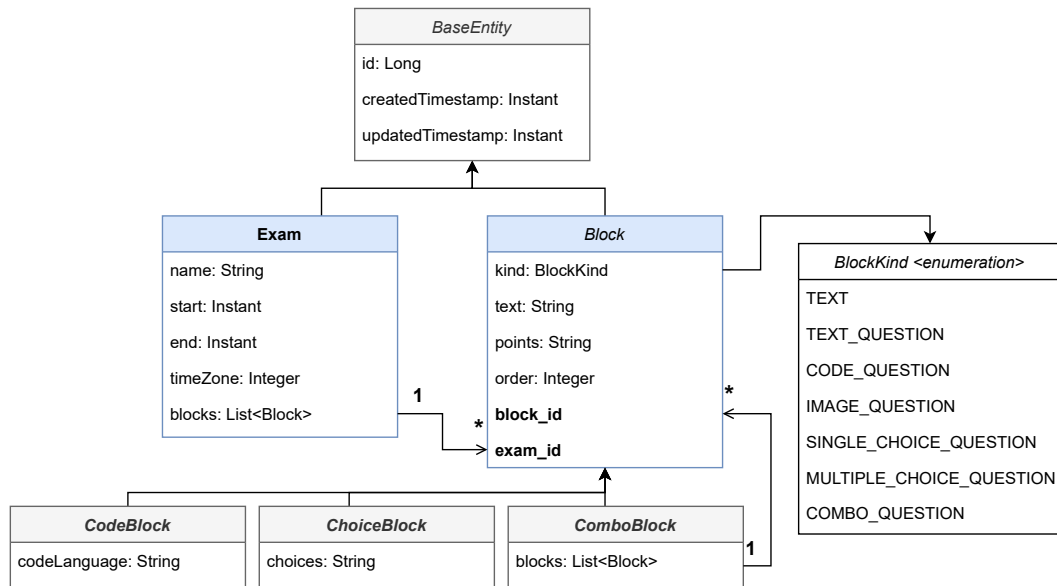


Figure 7.1: UML diagram of Exam & Block, as an exam consists of a Exam entity that is referencing (joining) a second entity Block containing questions related to the exam, joined using the exam id. To access the Block table for searching, we only need to define a Spring repository, which allows us to query the table as needed.

Secondly, we need to define the criteria for searching the exam blocks. In addition to the text attribute, the search considers various other fields depending on the type of question. This can be summarized into the following criteria:

- **All** - search based on the text attribute, which correlates to the title and description of a question.
- **Single- & Multiple-Choice** - search based on the choices, which is a list of possible options to choose from.
- **Code** - based on the codeLanguage attribute for searching based on the programming language of a code question.
- **Combo** - the newly introduced combo question is searched by the blocks attribute referencing a list of questions (recursive).

7.2 Implementation

For implementation, we are only required to build the correct SQL statement for fetching the corresponding Blocks. This is done using a `Specification` to build a criteria, which is similar to writing SQL. Writing such a `Specification` has many benefits in our use-case compared to writing a custom SQL query inside a `@Query()` annotation:

- They provide a modular way to define query conditions, allowing one to create smaller, reusable specifications that can be combined to create complex queries.
- Specifications are type-safe, meaning the query conditions are checked at compile time. This reduces the likelihood of runtime errors due to syntax issues of incorrect field references.
- Improved query performance through Hibernate's¹ query optimization especially when dealing with complex relationships and lazy loading.

Listing 7.1 shows a simple example of how such a statement can be written, which is automatically translated to an optimized query through Hibernate.

```

1 fun searchInText(pattern: String) = Specification<Block> {
2     root, _, builder ->
3         builder.like(
4             builder.lower(builder.trim((root.get("text")))),
5             pattern.like
6         )
7 }

```

Listing 7.1: Question search Specification (simple)

In comparison, Listing 7.2 shows the “translated” SQL statement used for filtering the same Block table.

```

1 SELECT * FROM block
2 WHERE lower(trim(BOTH FROM text)) LIKE '%pattern%'

```

Listing 7.2: Question search translated to SQL (simple)

¹Hibernate is an open-source object-relational mapping (ORM) framework for Java applications that simplifies the interaction between object-oriented code and relational databases [21].

7 Question Search

All defined Specifications can be combined using an OR operator to create a single SQL statement, which is further used for fetching all entries from the database. We want to let the database handle all the filtering efficiently instead of loading all entries into memory to perform filter operations. Nevertheless, one significant challenge arose when filtering for the ComboQuestions, which also can be filtered.

7.2.1 Recursive SQL Join

Searching for string properties is very simple compared to looking up properties referencing itself recursively as it has to be done for the newly introduced ComboQuestion. Listing 7.3 should provide an overview of how this could be achieved using a sub-query on the same type of table, joined by the ids of the Block.

```
1 fun searchInComboBlock(pattern: String)
2   = Specification<Block> { root, query, builder ->
3
4   val subQuery = query.subquery(Block::class.java)
5   val subRoot = subQuery.from(Block::class.java)
6   subRoot.join<Block, Block>("blocks")
7
8   subQuery.select(subRoot).where(
9     builder.equal(
10      subRoot.get<String>("id"),
11      root.get<String>("id")
12    ),
13    builder.or(
14      searchInText(pattern),
15      searchInChoices(pattern),
16      searchInCodeBlock(pattern)
17    )
18  )
19   builder.exists(subQuery)
20 }
```

Listing 7.3: Question search Specification (advanced)

7 Question Search

The purpose of this query is, to apply the same search predicates to blocks contained in a `ComboQuestions`. For this, a `subQuery` is created to be applied to the joined `subRoot`, allowing us to apply the predicates to not only the root (in this case, a `ComboQuestions`) but also all contained blocks via the `subRoot`. If a `Block` in the `subRoot` matches, we add it to the result as well as the `ComboQuestion` itself.

Listing 7.4 puts this specification into an SQL statement for better readability. It is important to note that it only shows a snippet of the whole sub-query for the `ComboQuestion`. The statement below is combined with a larger statement with several `OR` operators. In addition to checking if the current block `b1` meets the criteria, it is also checked if some sub-block `b2` potentially also meets the criteria. This is done by joining the same table on a block `b3` where the `block_id` (used for joining) equals the id of `b2`. In order to avoid a loop, we cannot join the sub-block `b2` on the block `b1`, therefore introducing `b3`.

```
1 SELECT * FROM block b1
2   -- sub-query for ComboQuestion
3   WHERE EXISTS(
4       SELECT * FROM block b2 JOIN block b3 ON b2.id=b3.block_id
5       -- includes the searches via pattern (text OR choice OR codeblock)
6       WHERE b2.id=b1.id AND ((...) OR (...) OR (...))
7   )
```

Listing 7.4: Question search translated to SQL (simple)

For example, when searching for the `BlockKind = SINGLE_CHOICE_QUESTION`, the result will return **two** results.

Given the following blocks:

- Block A (`BlockKind = TEXT`)
- Block B (`BlockKind = TEXT_QUESTION`)
- Block C (`BlockKind = COMBO_QUESTION`)
 - Block C1 (`BlockKind = TEXT`)
 - Block C2 (`BlockKind = SINGLE_CHOICE_QUESTION`)

Result: Block C (including Block C1 & C2) and the single Block C2.

8 Preliminary Submission

This chapter introduces the changes made to Xaminer to store preliminary results of students on the backend, in the database. Those can be automatically restored to the student's browser if needed. Further, supervisors can mark a preliminary submission as a final submission if necessary.

8.1 Motivation & Status Quo

Xaminer allows students to submit their results **only once** during final submission, which is confirmed two times before acting. This limitation means that storing any preliminary results on the backend is not possible. The results are only stored locally in the student's browsers using the *LocalStorage*¹ for persisting results for each question, e.g., storing only the string for a code question.

Each time students modify any value in a text area by typing their answer or selecting an answer via a radio button, this value is *immediately* persisted into this storage. Upon refreshing the browser - and being "reactivated" by the supervisor - the storage is checked for the current exam ID to read the values into memory, to further prefill the question answers of the exam.

Unfortunately, some students have experienced fatal browser crashes, completely wiping or corrupting all local data, which means their answers filled out during the exam could not be restored. In such cases, we want to assure that at least *some state* can be restored by retrieving a state from the backend, which stores their preliminary results. These results should be uploaded in the background using an interval without requiring any student interaction and without performing an actual final submission.

¹LocalStorage is a web storage mechanism that allows web applications to store data persistently in the browser with no expiration date [22]

8.2 Concept & Security Considerations

A crucial step to enhance students' experience is addressing the current limitation that permits only a single submission. This means that the backend should provide a new API to store results the same way students would perform a final submission. Allowing the storage of preliminary results in the backend will also enable supervisors to perform a final submission in favor of the student on fatal browser crashes if the exam ends before the student can rejoin and perform the final submission on their own.

Above all, though, the persistence of submissions and preliminary submissions must be completely separated without rebuilding the submission logic from scratch. Additionally, upon refreshing the browser we need to intelligently determine the latest result from either the LocalStorage or the backend storage.

Figure 8.1 provides an overview of the mechanism of how the data is persisted in both the browsers and the backend storage. Blue boxes indicate existing components or logic, while green boxes indicate newly introduced areas. Each of the three steps will be explained in the following sections.

Result submission to the backend is automatically done in the background using an interval. Before sending the data to the server, the local state is processed and shaped into a preliminary submission.

Result restoration on browser refresh, the local and server state must be fetched into memory and compared before allowing the student to join the exam view. The comparison of states is done using timestamps, always "preferring" the newer answer if available. If no data related to the exam and question can be found, the fallback value is used.

Supervisor submission for students allows for "final submissions" without the interaction of a student. This is a final option for lecturers to use the last state submitted to the backend for grading - this needs to be aligned accordingly for each student. This prevents the loss of a submission if, for some reason, the browser had a fatal crash right before the student performed a final submission.

8 Preliminary Submission

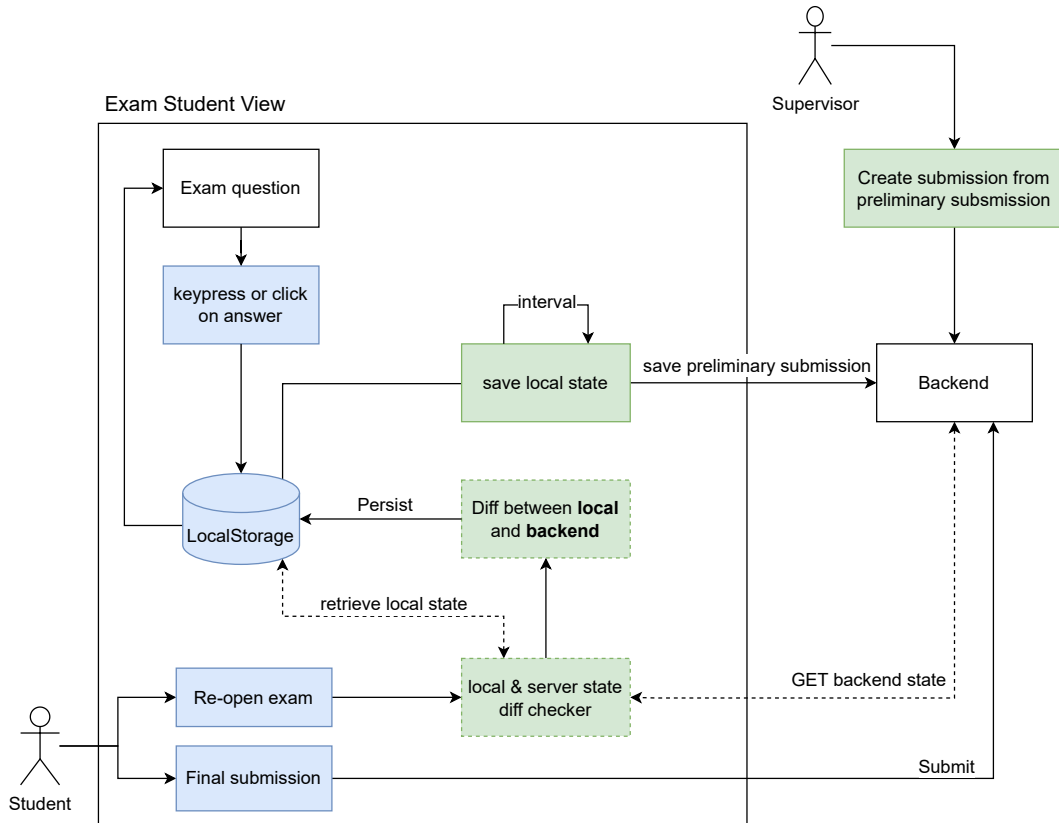


Figure 8.1: Concept of preliminary submissions showing a student and supervisor perspective.

Of course, when storing a preliminary submission for a student we need to guarantee that under no circumstances a different student can access the results of others through the server. Before, only one submission was sent to the server, and there was no option to ever retrieve it for the student. With this new functionality, we not only provide multiple submits but also options to retrieve the information again.

Security mechanism for protection relies on the same concept as the old way for verifying a student's identity through a special code, which is only known to the student's browser and the server. The code is a uniquely stored secret for each student joining the exam, which can be loaded from the server only once. To fetch the preliminary submission, the code must match upon sending the request. If the code does not match, an error is

8 Preliminary Submission

returned without the actual preliminary submission. The only way to retrieve this code is if student A shares his unique exam link with student B, allowing student B to join the exam, which is prohibited for obvious reasons.

8.3 Implementation

The implementation consists of three main parts. We are introducing the new API endpoint(s) for allowing to submit and fetch preliminary submissions, extending the existing business logic for storing such results, and introducing the algorithm in the frontend for identifying the most recent answer and correctly restoring the exam's state.

Firstly, the new API is introduced for fetching (GET) a preliminary submission and for sending (POST) the content of such a submission to the backend. Upon fetching, the result needs to be uniquely determined using the `examId` and matriculation number in the path of the API. For authentication, the code is used as previously discussed. If all checks are successful, and a preliminary submission exists, the result is returned containing all `ExerciseAnswers`.

Sending data to the backend works similarly, requiring the code for authentication and a payload containing the identifiers `examId` and `matNr` together with an array of the `exerciseAnswers`.

The resulting API endpoints and paths can, therefore, be defined as:

```
1 GET /api/preliminary-submission/{examId}/{matNr}?code=<code>
2 POST /api/preliminary-submission/submit?code=<code>
```

For storing, a new table `PreliminarySubmission` (Figure 8.2) is being introduced, requiring one SQL join with the existing `ExerciseAnswer` table, which stores the actual answer of a certain question. The table already exists for real submissions, meaning that the table is extended by another column (`preliminary_submission_id`) for joining not to mix joins from `PreliminarySubmission` with `Submission`.

8 Preliminary Submission

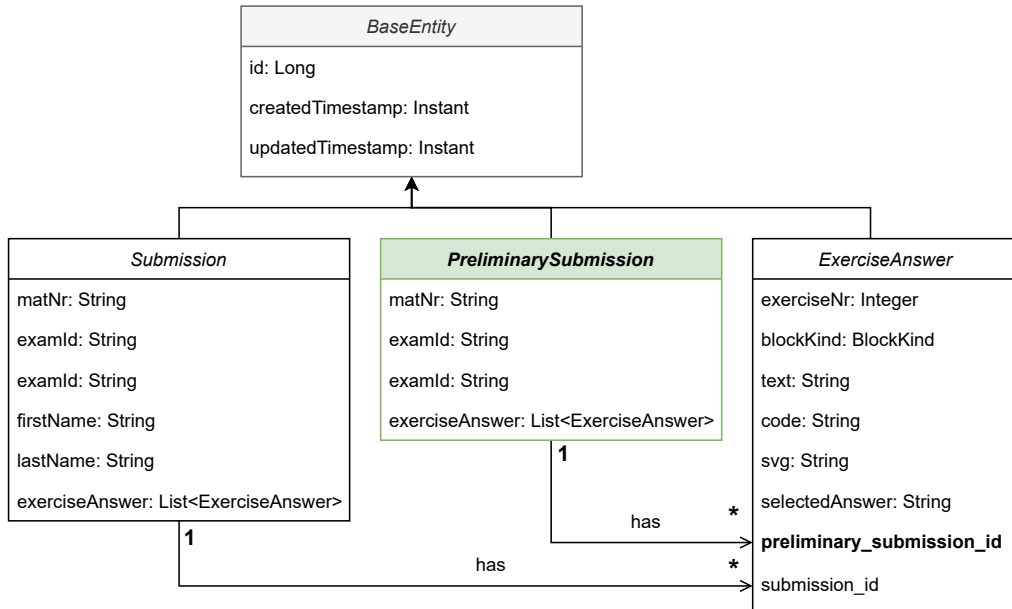


Figure 8.2: UML diagram of preliminary submissions showing the relation between the new table and the existing *ExerciseAnswer* table. For comparison, the existing *Submission* table is also shown, indicating that it also references the same *ExerciseAnswer* joined via a different ID. For simplicity, the *BlockKind* enumeration is not shown in this UML diagram as it was previously shown in Figure 6.1 and Figure 7.1

Table 8.1 shows a simplified version of the table in the database. Most importantly, exercise answers are joined with the other table using the corresponding ID. In this case, the question “What is Java” stores the answer for a real submission (with `id = 1`) while the two entries below relate to a preliminary submission (`id = 2`).

exerciseNr	text	submission_id	preliminary_submission_id
1	What is Java	1	NULL
3	Headline	NULL	2
4	Write a Kotlin program	NULL	2

Table 8.1: *ExerciseAnswer* table columns (simplified)

8 Preliminary Submission

Secondly, the existing LocalStorage needs to be extended with a timestamp, indicating the last time the answer for a specific question has been saved. This timestamp is saved for each question individually, changing the schema of the storage from simple Strings to JSON. The key, in the format of EXAM_ID@BLOCK_ID, required no further changes. Table 8.2 and Table 8.3, indicate examples for an exam with ID = 18 and ID = 24 comparing the existing with the new schema.

Key	Value
18@1	Java is a programming language
18@2	System.out.println("Hello, World");
24@6	fun isEven (number: Int) = number % 2 == 0

Table 8.2: Current LocalStorage schema

Key	Value
18@1	{ "timestamp": 1722179976590, "value": "Java is a programming language" }
18@2	{ "timestamp": 1722179966120, "value": "System.out.println(\"Hello, World\");" }
24@6	{ "timestamp": 1891799784801, "value": "fun isEven (number: Int) = number % 2 == 0" }

Table 8.3: New LocalStorage schema

Lastly, determining the most recent result upon reloading the browsers on the student's browser is the last crucial piece. The concept for this is relatively simple, as discussed in the previous section. For simplicity, Figure 8.3 shows the flowchart of the algorithm used to compare the local and server state, applying the most recent version for each question based on the timestamp. It is worth mentioning that the server state is always checked first to see if it exists. If it doesn't, we rely purely on the local state to render the student's view. If it exists, we try to compare the timestamps to override the local state or completely restore the local state if it cannot be found, e.g., a fatal browser crash that cleared all local browser data.

8 Preliminary Submission

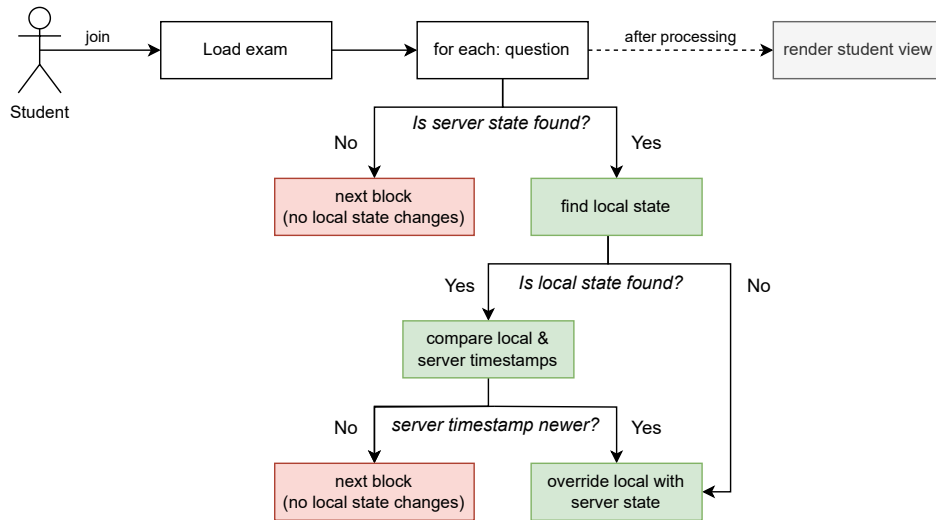


Figure 8.3: Flowchart for determining the most recent answer for restoring the results. The local state can always be set equally to the `LocalStorage` for persistence. For consistency across the application logic, the state is ever only loaded into memory (JavaScript) if really needed, e.g., when creating a preliminary or final submission, otherwise it is always immediately persisted into the storage.

9 Single PDF Export

This chapter covers both the concept and implementation of the single PDF export. This feature lets the exam creator download all student submissions for a given exam, ready for immediate printing.

9.1 Motivation & Concept

Currently, exam results are exported and downloaded as a .zip file, where each student's submission is stored in an individual PDF file. However, a key improvement would be the ability to export all submissions into a single PDF file formatted for efficient double-sided printing. Such an extension would ensure that each sheet of paper contains only one student's submission, enhancing usability and reducing administrative overhead as currently, each submission PDF has to be printed separately, requiring several minutes of work. Additionally, for grading and student exam reviews, it is essential to ensure that no page ever contains submissions from more than one student. Lastly, when downloading this PDF, it should be possible to allow multiple pages on a single page, e.g., two pages or four pages per side.

Conceptually, the existing mechanism for generating the PDF for a single submission can be re-used entirely. The main idea is to detect how many blank pages are required to achieve the requirement of having always only a single submission on one (or multiple) printed sides merged into a single PDF file. For reading the metadata and modifying PDF files, a new library needs to be introduced, namely `pdfbox`¹ that can read the number of pages of the submission file as well as create new PDFs in-memory.

¹<https://pdfbox.apache.org>

9 Single PDF Export

Figure 9.1 shows the steps needed for generating a single PDF.

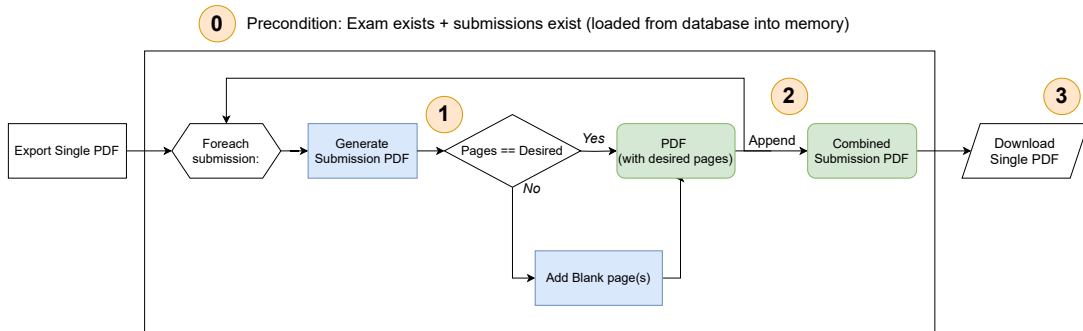


Figure 9.1: Flowchart of the single PDF export process for combining PDFs.

Step 0 loads all the submissions for the exam into memory to be able to generate the submission PDF. After generating the PDF (Step 1), we need to read the content of the file to detect how many pages were generated to check how many blank pages need to be added. For this, a simple modulo calculation can be performed:

$$P_{empty} = P_{requested} - (P_{submission} \bmod P_{requested})$$

where P_{empty} is the number of pages to add to the current submission PDF, based on the $P_{requested}$ that holds the value of how many pages are expected on a double-sided A4 sheet (e.g., two, four, or eight pages) and $P_{submission}$ indicating the current amount of pages generated based on the submission.

Lastly, once the submission PDF has the correct number of pages, we add it to the results “Combined Submission PDF” (Step 2), which will be provided as the downloadable PDF file once all submissions are processed (Step 3).

9.2 Implementation

The implementation mainly consists of the mechanism to determine the current amount of pages that have been generated and adding empty pages to a resulting, combined PDF.

9 Single PDF Export

To generate the PDF, the existing library called `wkhtmltopdf` is used that simply converts HTML to PDF files. By design, it only *generates* files, meaning that it does not provide any meta-information about the PDF itself, e.g., page count. Additionally, it cannot merge nor modify PDF files which means it can only be used to transform the submission to a file.

Listing 9.1 shows a simplified version of the method used. `PDFMergerUtility` and `PDDocument()` are imports of the `pdfbox` library that allow us to create the resulting PDF in memory. To generate the PDFs based on the submission, the already available `convertToPdf()` method is used. The resulting file is loaded into memory using the `Loader.loadPDF()` to further determine the page count. Using the page count, we can determine how many empty pages need to be added (the empty page is loaded into memory before, which consists of a blank page) that is added to the submission PDF. The resulting file is then appended to the `combinedPDF` returned for the download.

```
1 val combinedPdf = PDDocument()
2
3 for (submission in submissions) {
4     val pdf = convertToPdf(exam, submission)
5     val pdfFile = Loader.loadPDF(pdf)
6     val P_submission = pdfFile.pages.count
7
8     if (P_submission % P_requested == 0)
9         continue
10
11     val P_empty = P_requested - (P_submission % P_requested)
12     for (i in 1..P_empty) {
13         pdfFile.addPage(emptyPage)
14     }
15
16     PDFMergerUtility.appendDocument(combinedPdf, pdfFile)
17 }
18
19 return combinedPdf
```

Listing 9.1: Single PDF export (simplified)

10 Usage and Evaluation

This chapter covers how the new features and improvements can be used by lecturers and developers. The first part covers developer-related sections, while the second part covers the four quality-of-life improvements for lecturers.

10.1 Docker & Docker-Compose

Deploying and updating Xaminer on a server must be done using Docker and a Docker-Compose file, which can be found in the root of the applications repository¹.

Two variants of the compose file start the application(s) with the same configuration but with different docker images. The local version (`docker-compose.local.yml`) runs the build process on the local machine where the command is executed, while the non-local version (`docker-compose.yml`) pulls the tested and built images from a remote registry (recommended way of running Xaminer). For development purposes, one can use the `docker-compose.local.yml`, but it is **not** the recommended way of running Xaminer on a server for an institute.

For application configuration, e.g., which version of the application should be used or database credentials, a configuration file in the project's root named `.env` must be created. This configuration file should **never** be distributed if credentials are included. The `.env.example` in the repository can be used as a reference when creating the configuration file for deploying an Xaminer application.

¹<https://github.com/NeonMika/xaminer>

10 Usage and Evaluation

```
1 FRONTEND_IMAGE=ghcr.io/neonmika/xaminer/frontend:1.3.0
2 BACKEND_IMAGE=ghcr.io/neonmika/xaminer/backend:1.3.0
3
4 POSTGRES_URL=jdbc:postgresql://postgres:5432/postgres
5 POSTGRES_USER=postgres
6 POSTGRES_PASSWORD=password
```

Listing 10.1: .env configuration file (simplified)

Listing 10.1 represents a simplified version of the most relevant configurations. `FRONTEND_IMAGE` and `BACKEND_IMAGE` define the version and remote registry from where the images need to be pulled. Additionally, database credentials and URLs are defined using this file. If the database defined in the Docker Compose should be used, the hostname of the container, “postgres”, must be chosen.

Lastly, to run the application, the following command can be used to start every defined container of the compose file:

```
1 $ docker-compose [-f docker-compose.local.yml] up -d
```

Using their name as the last argument of the command can start individual components of the compose file. This can also be used to restart or update individual containers.

```
1 $ docker-compose [-f docker-compose.local.yml] up -d frontend
2
3 $ docker-compose [-f docker-compose.local.yml] restart backend
```

Lastly, to completely stop the application for bigger maintenance, the Docker Compose can also be used.

```
1 $ docker-compose [-f docker-compose.local.yml] down
```

Full reference to all available commands can be found via <https://docs.docker.com/reference/cli/docker/compose/>

10.2 Questions with Sub-Questions

To create an exam section containing sub-questions, creators are only required to add the new question type “Combo question” as seen in Figure 10.1.

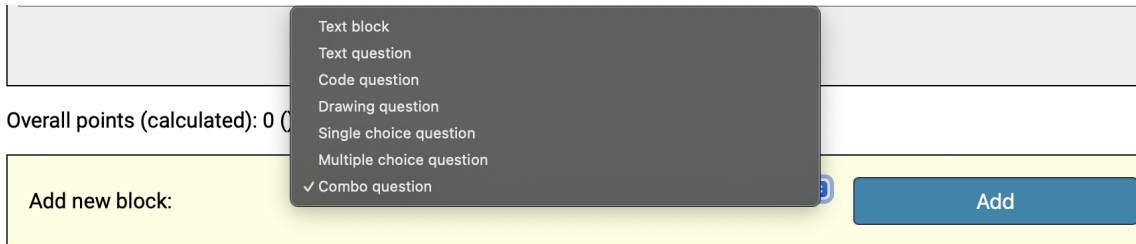


Figure 10.1: Option showcase for newly introduced Combo Question selection.

After adding the new section, a second selector inside the newly added combo question (Figure 10.2) appears to add further sub-questions to it. As the implementation does not limit any usage of other question types nor the depth of how many combo questions can be inside another combo question, creators can freely add as many elements as desired.

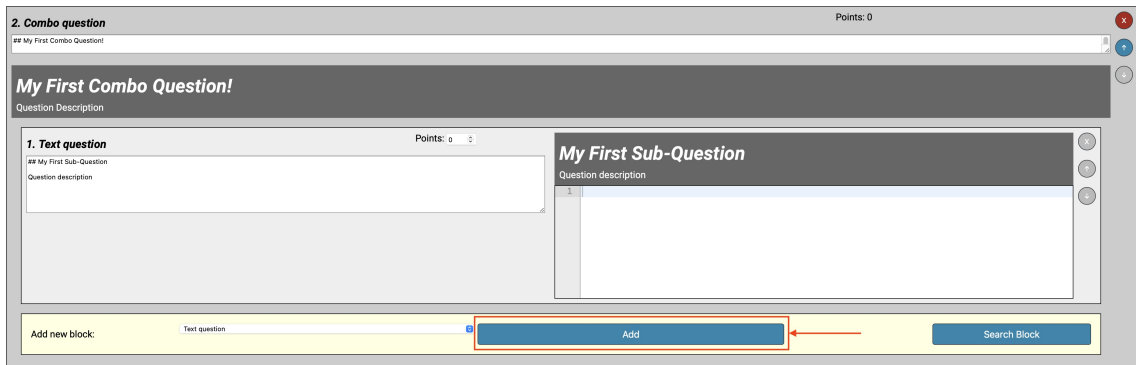


Figure 10.2: Showcase of newly introduced “Add” button for Combo Question.

Added sub-questions can be moved or deleted the same way as before. Deleting the combo question itself will delete all questions contained within. As a small improvement and to prevent accidental deletes of questions, a new confirmation dialog has been added before removing the question.

10.3 Question Search

The question search has been added as a separate button next to adding an empty question type as seen in Figure 10.3. As mentioned in the previous Section 10.2, each combo question has its selector for adding questions, which means that it will also include its question search button for adding existing tasks as sub-questions.

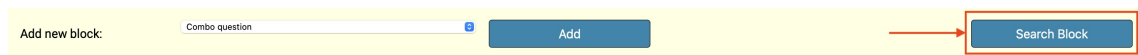


Figure 10.3: Location of the new button for the introduced Question Search.

Upon clicking the button, creators are presented with a modal containing a text input used for searching (Figure 10.4). The default search term is empty, leading to all available questions being loaded from the backend to choose from. Entering a specific search term automatically triggers a search (or upon pressing enter) on the backend filtering for all criteria mentioned in Chapter 7. If a desired question has been found, it can be added via the “Add to Exam” button, copying the question into the area where the question search has been opened.

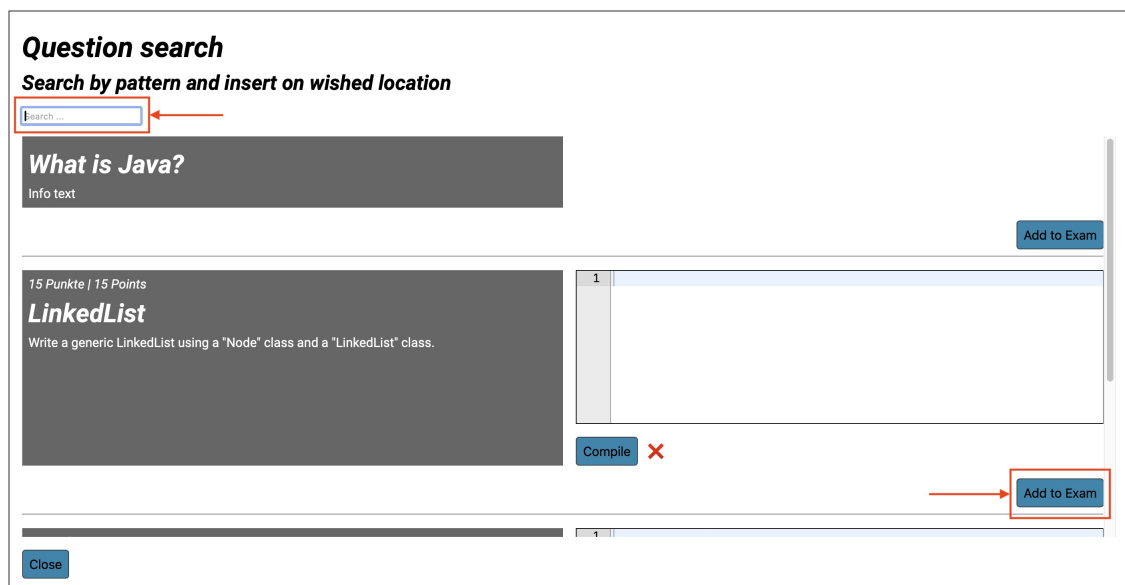


Figure 10.4: Overview of the modal and search input for adding questions through Question Search.

10.4 Single PDF Export

For exporting all submissions at once to a single, printable PDF, a new button below the existing download buttons (Figure 10.5) has been introduced. Upon clicking, a modal opens with a set of configurations to choose from. Each radio button indicating a number determines how many pages should be added to the A4 sheet, e.g., "2" prepares the PDF to be printed double-sided with two pages on each page.

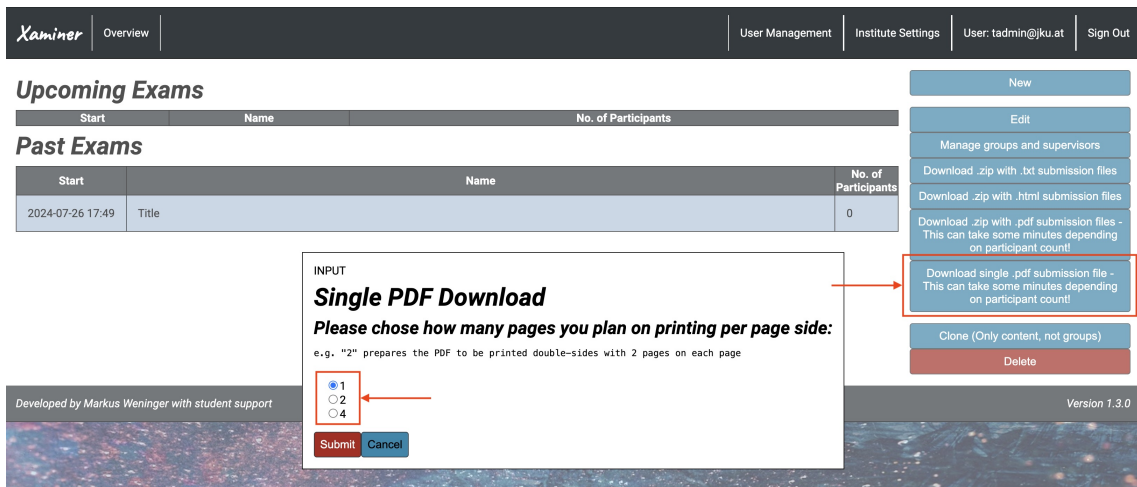


Figure 10.5: Button and dialog for exporting all submissions as a single PDF.

It is worth mentioning, that the duration for downloading the PDF might vary based on how many students participated, as well as how many questions had to be answered. With many students participating, the download might take a couple of minutes.

10.5 Bi-directional Communication

The bi-directional communication introduced in the Thesis opens up opportunities for many features requiring live updates from the backend to clients or to generally broadcast information to certain clients.

The first use-case implemented adds live updates for exam supervisors, informing them if students join and submit the exam. No further action is required from supervisors as

all logic is handled in the background. Figure 10.6 indicates how such notification would look.

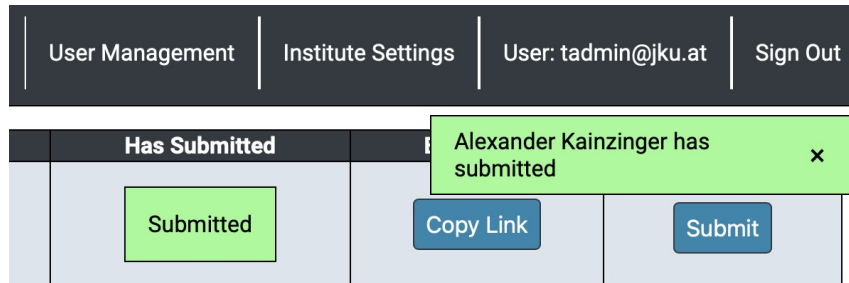


Figure 10.6: Example notification of a student submission notification using bi-directional communication.

To implement new use cases, both the backend and frontend can be easily extended with custom logic to handle things such as authentication. Socket.IO will efficiently handle multiple connections to the backend.

10.6 Developer Experience Improvements

Lastly, future development of Xaminer has improved a lot. By modernizing and introducing new technologies to the application stack, developers can create features more efficiently and faster while also automating manual-intensive tasks such as testing and building.

10.6.1 Frontend Tooling

Replacing Webpack with Vite.js drastically improves performance when it comes to build times and CPU consumption. For testing, three consecutive runs were performed and are listed in Table 10.1 for build time and Table 10.2 for CPU consumption.

The tests were conducted on a **Apple MacBook Pro M2 (64GB)** running **MacOS Sonoma 14.5** with **NodeJS 22.4.1**.

10 Usage and Evaluation

Webpack (Dev)	Vite (Dev)	Webpack (Prod)	Vite (Prod)
11.29s	0.13s	53.13s	6.82s
11.32s	0.12s	50.23s	6.90s
11.27s	0.13s	50.30s	6.86s
11.29s (avg.)	0.126s (avg.)	51.22s (avg.)	6.86s (avg.)

Table 10.1: Compilation times test results (lower better)

Webpack (Dev)	Vite.js (Dev)	Webpack (Prod)	Vite.js (Prod)
237%	8%	438	208
243%	11%	436	208
249%	11%	441	185
243% (avg.)	10% (avg.)	438% (avg.)	200% (avg.)

Table 10.2: CPU consumption test results (lower better)

For comparison, Figure 10.7 provides an overview of how many resources could be saved (in percent). The development build time shows a drastic improvement as it does not pre-compile the whole project but only if requested, significantly reducing the cold start.

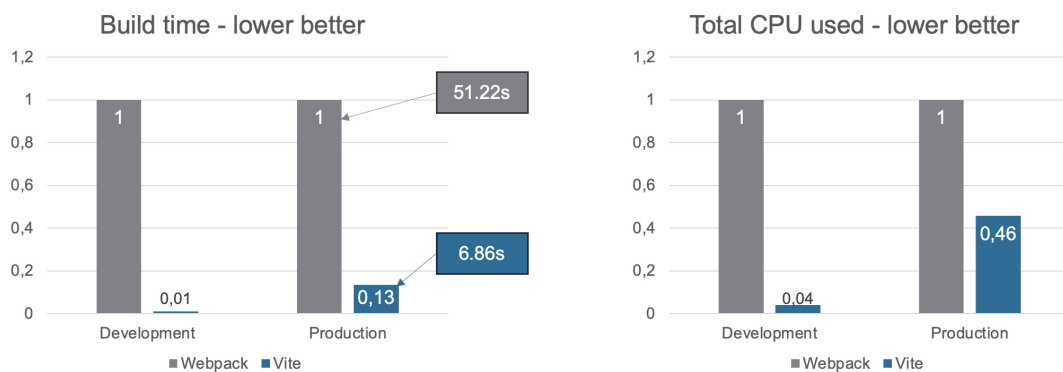


Figure 10.7: Compilation time & CPU consumption comparison of Webpack and Vite.js.

10.6.2 Automated Builds

With the addition of a GitHub Action automation pipeline, developers can ensure that changes performed to Xaminer do not break any existing functionality as well as ship a new version of the application with a single click.

10 Usage and Evaluation

Each time a pull request is opened, a pipeline is automatically triggered, running all jobs without requiring any manual action (Figure 10.8). Upon successfully finishing **all** jobs, including a job to check code styles, a job to run unit- & integration-tests in the backend, and finally, a full end-to-end test suite performing programmatic clicks in the user interface to test certain functionality, developers can merge their changes into the main codebase.

chore: bump dependencies #35

alexkainzinger wants to merge 2 commits into master from bump-dependencies

Conversation 0 Commits 2 Checks 3 Files changed 4

alexkainzinger commented now • edited

This PR updates frontend and backend dependencies

alexkainzinger added 2 commits 10 minutes ago

- chore(backend): bump to spring boot 3.3.2 755f47b
- chore(frontend): bump dependencies c8bca11

Some checks haven't completed yet 2 in progress and 1 successful checks

- .github/workflows/main.yml / Yarn cache (pull_request) In progress — This check has starte... Details
- .github/workflows/main.yml / Check backend codestyles (pull_request) Successful in 15s Details
- .github/workflows/main.yml / Compile & test backend (pull_request) In progress — This che... Details

This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request You can also open this in GitHub Desktop or view command line instructions.

Figure 10.8: Automatically triggered GitHub Action on a pull-request, which is in progress.

On average, a full run of the pipeline takes around **22 minutes** if all jobs can be run successfully and no consecutive job was skipped due to a previous one failing. If a job fails, a glance at the summary can help the developer to identify the issue to resolve, as seen in Figure 10.9. Additional information can be found by checking the logs of a job by selecting it in the web interface.

10 Usage and Evaluation

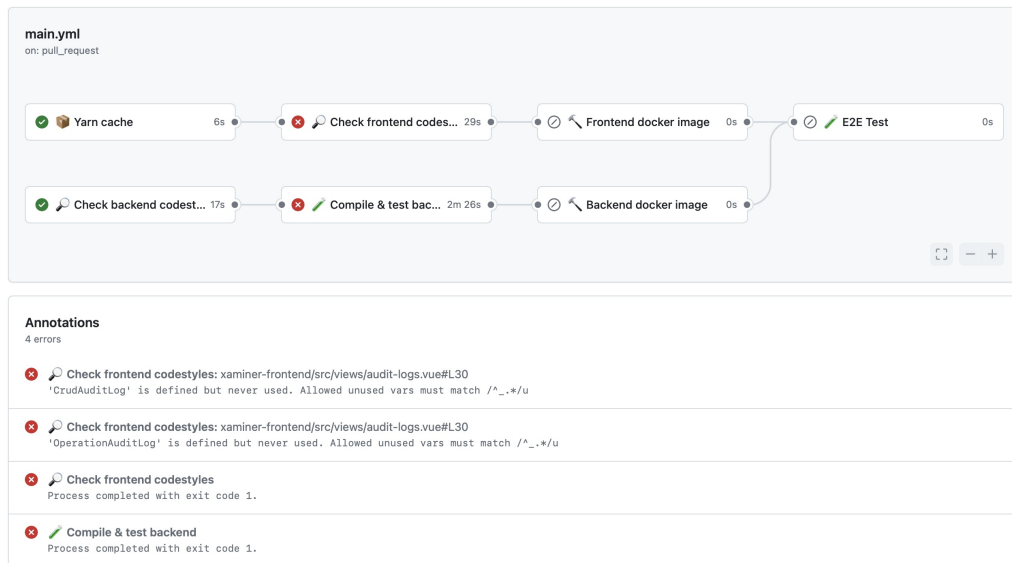


Figure 10.9: Example of a failed GitHub Action with results and reasons for failure.

Creating a new version of Xaminer is also fully automated by leveraging GitHub Releases (<https://github.com/NeonMika/xaminer/releases/new>, Figure 10.10), where the same GitHub Action is run for performing several checks before building the application and pushing it to a remote registry.

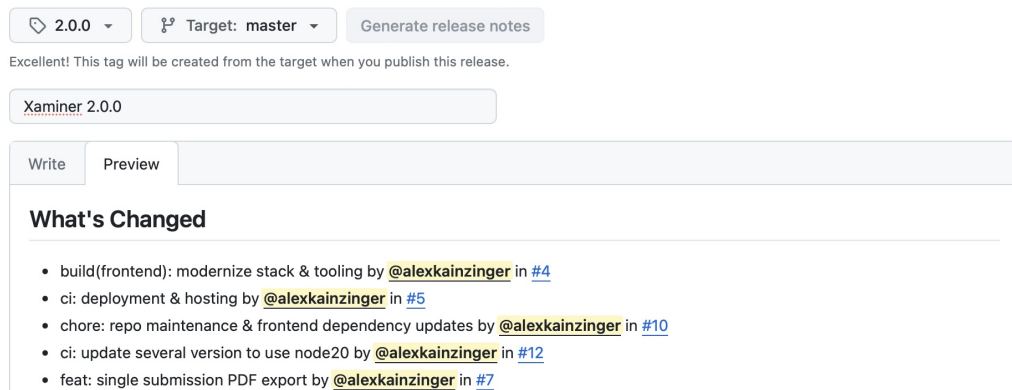


Figure 10.10: GitHub Release page for creating new versions of Xaminer.

11 Conclusion and Outlook

One primary focus for Xaminer's future development is the continuous modernization of its technology stack. The recent upgrade from Vue.js 2 to Vue.js 3 is a critical step in keeping the platform's frontend framework current, enhancing both performance and security. To maintain a smooth and reliable user and developer experience, Xaminer should be updated in the future to take advantage of the most recent developments in software development.

In addition to technological updates, the platform's architecture has been improved to support more flexible and efficient deployment. The decoupling of frontend and backend components and the adoption of Docker and CI pipelines with GitHub Actions streamline the development and deployment process. This modularity not only simplifies maintenance and updates but also enhances system reliability and scalability, allowing Xaminer to better handle increased user demand and complex functionalities.

User experience enhancements remain a key priority, with recent updates introducing features such as single-file PDF exports, preliminary exam submissions, and support for questions with sub-questions. These improvements address specific user needs and contribute to a more efficient and user-friendly platform. Future developments should continue to prioritize user-centric features, potentially exploring real-time chat functionalities and enhanced proctoring tools.

To sum up, Xaminer's improvements indicate a considerable advancement in the platform's evolution and guarantee that it will always be a useful resource for online examination. The focus on modernization, efficient deployment, and user experience has not only improved the system's functionality but also positioned it well for future growth. Xaminer offers an effective and flexible solution for digital exams, making it well-suited to meet the changing demands of the academic community as digital education grows.

List of Figures

2.1	Docker architecture & components overview.	8
2.2	Xaminer high-level architecture overview.	12
2.3	Current Xaminer hosting setup on a virtual machine.	13
3.1	Life-cycle of Webpack's bundling mechanism.	15
3.2	Life-cycle of Vite's bundling mechanism using Native ESM.	16
3.3	Page loading behavior before migrating to Vue 3.	18
3.4	Page loading behavior after migration to Vue 3 using "vue-router".	19
4.1	Usage of a "Reverse Proxy" for the new hosting setup.	25
4.2	Implementation of the GitHub Action workflow for Xaminer.	26
4.3	Cypress job result of passed, failed, pending, or skipped tests and duration.	30
5.1	Socket.IO Namespace architecture overview.	34
6.1	Structure of the newly introduced ComboBlock for sub-questions.	39
6.2	Component call graph for the recursive rendering of components in the frontend.	40
7.1	UML diagram of Exam & Block, as an exam consists of a Exam entity that is referencing (joining) a second entity Block containing questions related to the exam, joined using the exam id. To access the Block table for searching, we only need to define a Spring repository, which allows us to query the table as needed.	42
8.1	Concept of preliminary submissions showing a student and supervisor perspective.	48

List of Figures

8.2	UML diagram of preliminary submissions showing the relation between the new table and the existing ExerciseAnswer table. For comparison, the existing Submission table is also shown, indicating that it also references the same ExerciseAnswer joined via a different ID. For simplicity, the BlockKind enumeration is not shown in this UML diagram as it was previously shown in Figure 6.1 and Figure 7.1	50
8.3	Flowchart for determining the most recent answer for restoring the results. The local state can always be set equally to the LocalStorage for persistence. For consistency across the application logic, the state is ever only loaded into memory (JavaScript) if really needed, e.g., when creating a preliminary or final submission, otherwise it is always immediately persisted into the storage.	52
9.1	Flowchart of the single PDF export process for combining PDFs.	54
10.1	Option showcase for newly introduced Combo Question selection.	58
10.2	Showcase of newly introduced “Add” button for Combo Question.	58
10.3	Location of the new button for the introduced Question Search.	59
10.4	Overview of the modal and search input for adding questions through Question Search.	59
10.5	Button and dialog for exporting all submissions as a single PDF.	60
10.6	Example notification of a student submission notification using bi-directional communication.	61
10.7	Compilation time & CPU consumption comparison of Webpack and Vite.js.	62
10.8	Automatically triggered GitHub Action on a pull-request, which is in progress.	63
10.9	Example of a failed GitHub Action with results and reasons for failure.	64
10.10	GitHub Release page for creating new versions of Xaminer.	64

List of Tables

8.1	ExerciseAnswer table columns (simplified)	50
8.2	Current LocalStorage schema	51
8.3	New LocalStorage schema	51
10.1	Compilation times test results (lower better)	62
10.2	CPU consumption test results (lower better)	62

Listings

2.1	Example REST controller	5
3.1	Old *.page.ts syntax (Webdriver.IO)	20
3.2	New *.page.ts syntax (Cypress)	20
3.3	Old *.spec.ts syntax (Webdriver.IO)	20
3.4	New *.cy.ts syntax (Cypress)	20
4.1	Basic Dockerfile	22
4.2	Xaminer docker-compose.yml (simplified)	23
4.3	GitHub Action job definition for “Check frontend codestyles”	28
5.1	Adding a protected namespace	35
5.2	Socket.IO listener in client (simplified)	36
6.1	ComboBlock.kt for sub-questions	38
7.1	Question search Specification (simple)	43
7.2	Question search translated to SQL (simple)	43
7.3	Question search Specification (advanced)	44
7.4	Question search translated to SQL (simple)	45
9.1	Single PDF export (simplified)	55
10.1	.env configuration file (simplified)	57

Bibliography

- [1] *SPA (Single-page application)*. <https://developer.mozilla.org/en-US/docs/Glossary/SPA>. Accessed: 2024-08-06 (cit. on p. 6).
- [2] *What is a RESTful API?* <https://aws.amazon.com/what-is/restful-api/>. Accessed: 2024-08-05 (cit. on p. 7).
- [3] *The WebSocket API (WebSockets)*. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. Accessed: 2024-08-05 (cit. on pp. 7, 32).
- [4] *WebSocket*. <https://www.ibm.com/docs/en/was/9.0.5?topic=applications-websocket>. Accessed: 2024-08-05 (cit. on pp. 7, 32).
- [5] *Docker overview Docker architecture*. <https://docs.docker.com/guides/docker-overview/#docker-architecture>. Accessed: 2024-07-01 (cit. on p. 8).
- [6] *About Git*. <https://git-scm.com/about/branching-and-merging>. Accessed: 2024-08-04 (cit. on p. 9).
- [7] *Yarn Introduction*. <https://yarnpkg.com/getting-started>. Accessed: 2024-07-03 (cit. on p. 10).
- [8] *Webpack Concepts*. <https://webpack.js.org/concepts>. Accessed: 2024-07-03 (cit. on p. 10).
- [9] *Vite.js Getting Started*. <https://vitejs.dev/guide>. Accessed: 2024-07-03 (cit. on p. 10).
- [10] *Why Webdriver.IO?* <https://webdriver.io/docs/why-webdriverio>. Accessed: 2024-07-03 (cit. on p. 10).
- [11] *Why Cypress?* <https://docs.cypress.io/guides/overview/why-cypress>. Accessed: 2024-07-03 (cit. on p. 10).
- [12] *ESLint*. <https://eslint.org>. Accessed: 2024-07-06 (cit. on p. 10).

Bibliography

- [13] *Gradle User Manual*. <https://docs.gradle.org/current/userguide/userguide.html>. Accessed: 2024-07-06 (cit. on p. 10).
- [14] *JUnit5*. <https://junit.org/junit5>. Accessed: 2024-07-06 (cit. on p. 11).
- [15] *Vue Tooling*. <https://vuejs.org/guide/scaling-up/tooling.html>. Accessed: 2024-07-25 (cit. on p. 15).
- [16] *Vite.js Getting Started*. <https://vitejs.dev/guide/why.html#slow-server-start>. Accessed: 2024-07-25 (cit. on pp. 15, 16).
- [17] *Server-sent events*. https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events. Accessed: 2024-08-05 (cit. on p. 32).
- [18] *What is Long Polling?* <https://www.pubnub.com/guides/long-polling/>. Accessed: 2024-08-05 (cit. on p. 32).
- [19] *Socket.IO - How it works*. <https://socket.io/docs/v4/how-it-works/>. Accessed: 2024-08-05 (cit. on p. 33).
- [20] *Socket.IO - Namespaces*. <https://socket.io/docs/v4/namespaces/>. Accessed: 2024-08-05 (cit. on p. 34).
- [21] *Hibernate ORM*. <https://hibernate.org/orm/>. Accessed: 2024-09-29 (cit. on p. 43).
- [22] *Window: localStorage property*. <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>. Accessed: 2024-08-03 (cit. on p. 46).