Eberhard Karls Universität Tübingen

Mathematisch-Naturwissenschaftliche Fakultät

Wilhelm-Schickard-Institut für Informatik

Lehrstuhl für Datenbanksysteme

Bachelor of Science Informatik

CRUSHING BOUNDARIES: OVERCOMING TECHNICAL DEBT BY USING FULL-STACK FRAMEWORKS

FLORIAN MARTEL

30.04.2025

Gutachter

Prof. Dr. Torsten Grust

Betreuer

Tim Fischer

Florian Martel:
Crushing Boundaries: Overcoming Technical Debt By Using Full-Stack Frameworks
Bachelor of Science Informatik
Eberhard Karls Universität Tübingen
01.02.2025 – 30.04.2025

Declaration of Authorship

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe. Des Weiteren erkläre ich, dass die Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist.

Tübingen, 30.04.2025
Ort, Datum

F. Martel
Unterschrift

Abstract

With the growing adoption of Software as a Service (SaaS), web applications are increasingly replacing traditional on-premise software solutions. The conventional approach to web application development involves separating the frontend and backend by using different frameworks, such as React, Vue.js, or Angular for the frontend, and Django, Spring Boot, or ASP.NET for the backend. However, this separation causes complexities in development and maintenance. Full-Stack frameworks, such as Next.js, provide a unified development environment that covers both frontend and backend implementation. This thesis shows how a Full-Stack approach simplifies functionalities like database interaction, authentication and rendering, using the development of a digital module manual for the University of Tübingen as a demonstration of its advantages.

Acronyms

API:	Application Programming Interface
CDN:	Content Delivery Network
CI/CD:	Continuous Integration and Continuous Deployment
CLI:	Command Line Interface
GHCR:	GitHub Container Registry
HTTP:	Hypertext Transfer Protocol
IDE:	Integrated Development Environment
ISR:	Incremental Static Regeneration
JWT:	JSON Web Token
npm:	Node Package Manager
ORM:	Object Relational Mapping
OS:	Operating System
REST:	Representational State Transfer
SEO:	Search Engine Optimization
SSH:	Secure Shell
UI:	User Interface
VM:	Virtual Machine

Contents

1.

Introduction

Web applications often evolve from simple tools into complex systems that become harder to maintain over time. This chapter outlines a real example of such a case: the course management tool “Digitales Modulhandbuch” used by students at the University of Tübingen.

1.1. Motivation

Students in the Department of Computer Science at the University of Tübingen are familiar with a web application that provides an overview of which courses can be assigned to which elective modules (Figure 1).

Additionally, the application allows students to view detailed information about each course, including content descriptions and recommended literature (Figure 2).

As a student, this tool helps with deciding which course to take in the semester. In order to do so there is a filtering feature, allowing students to filter courses for specific semesters, lecturers or assignments.

However, the application suffers from several shortcomings in terms of usability and maintainability. For example, the semester filtering of a course is implemented by a hard-coded enum, which requires manual adjustments in the source code from time to time. As well the semester values that indicate when the course is offered have to be updated manually. Furthermore, there is neither an Continuous Integration and Continuous Deployment (CI/CD) pipeline nor a version control system, such as Git. This complicates further development as every code change has to be applied manually on the production server. The original application is built with Django, a Python-based web framework, and follows a server-rendered architecture with minimal client-side functionality. As a result, most user interactions cause full page reloads — for example, switching to the list of master’s courses in Figure 1 reloads the entire interface.

Other features regarding user experience can be improved as well. For instance, when searching for the Medical Informatics subject area, students must scroll horizontally, even if other subjects are irrelevant to them (Figure 1). Additionally, User Interface (UI) elements such as filters sometimes disappear unintentionally during hover interactions. Addressing the mentioned issues is difficult due to technical debt.

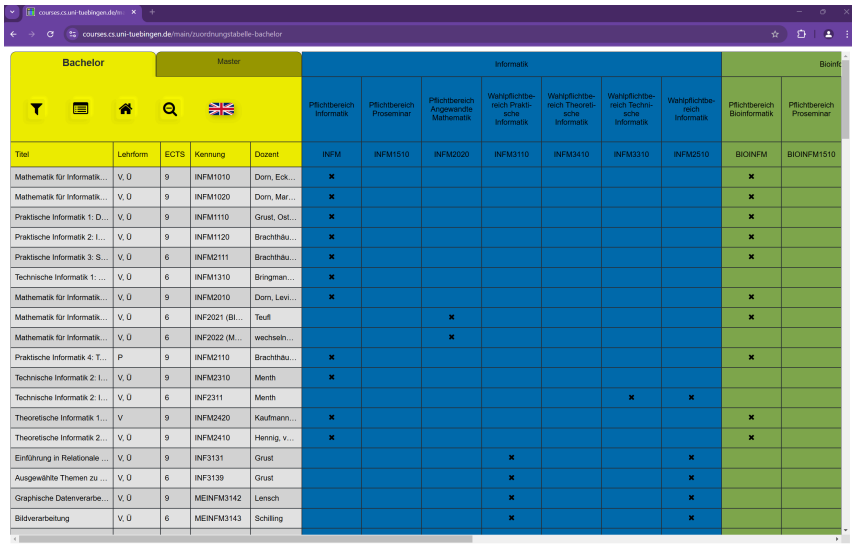


Figure 1: User Interface of the module directory

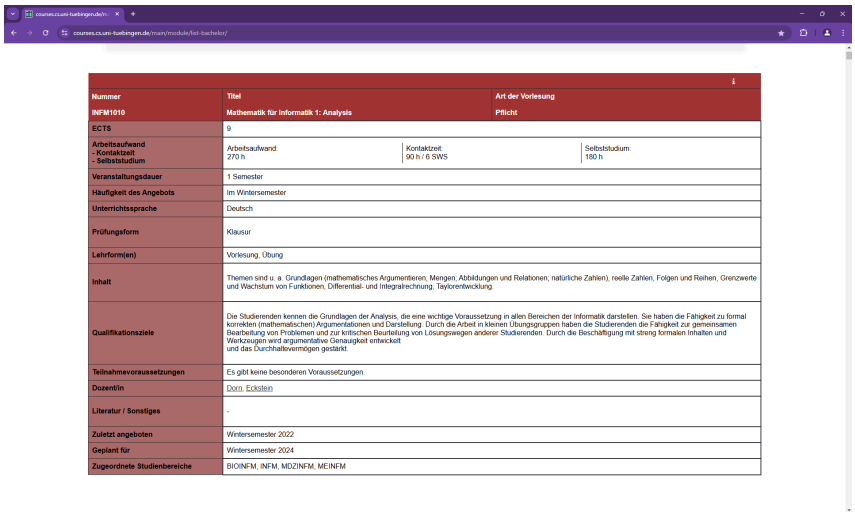


Figure 2: Course details of the module directory

1.2. Goals

Given these problems with the application, the goal of this thesis is to develop a new application that adopts the functionality of the current one while resolving its usability and maintainability issues. The redesigned system should provide a more intuitive and responsive user interface, simplify the development and deployment process, and be easier to maintain and extend.

The scientific goal of this thesis is to explore how the strict boundary between client and server can be blurred using Next.js as Full-Stack framework. In web development the terms client and server both refer to computers. The client is responsible for sending requests to the server and displaying the server's responses, often through a web browser. A server is a computer that provides resources to clients over a network. In the context of web development, the server processes incoming client requests, performs necessary operations and sends back the appropriate responses. Typically, client-side and server-side code is developed using different frameworks in separate code-bases. Server and client communicate with each other through an Application Programming Interface (API) such as Representational State Transfer (REST) or GraphQL. As a result, they do not share the same types, classes and libraries. This strict separation, often referred to as the network boundary, can lead to reduced development ergonomics and technical debt.

1.3. Overview

The source code of the developed application can be accessed on the public GitHub repository [1].

The application is organized as a monolithic[1] repository (Listing 1). The main directory — labeled as the `app` directory in the source code — implements all URL routes using Next.js–specific files (Section 2.1.1).

The project contains three services. First, there is the primary Next.js application found in the `src` directory. Second, a PostgreSQL database is used, with the schema described in `schema.prisma`. Third, an optional cronjob server is included, which triggers an API route to update semester data. Although the application remains functional without the cronjob server, automatic updating of semesters will not occur in its absence (Section 1.1).

In addition, the repository includes a dedicated deployment setup. It provides GitHub Actions[2] workflows for CI/CD, along with a custom deployment script.

```
root/
├── .github/              # GitHub Actions for CI/CD
├── deployment/           # Files for deployment
│   ├── deploy.sh         # Bash script for Docker deployment
│   ├── server.js         # Express.js server for web-hook-based deployment
├── prisma/               # Prisma files for database development
│   ├── migrations/       # SQL migrations applied to the database
│   ├── schema.prisma     # Database schema
├── public/               # Static assets (images etc.)
├── src/                  # Next.js Source code
│   ├── actions/          # Next.js Server Actions for database interaction
│   ├── app/              # Next.js App directory containing all URL routes
│   │   ├── api/          # API routes
│   │   ├── backend/      # Restricted route for lecturers and admins (auth required)
│   │   ├── common/       # Open route for students
│   ├── components/       # Custom React components
│   ├── fonts/            # Fonts
│   ├── lib/              # Library for custom backend logic
│   ├── utils/            # Auth utilities (guard components and logic)
│   ├── auth.ts           # Auth.js configuration
│   ├── paths.ts          # Paths
├── docker-compose.yml    # Compose file for connecting containered services
├── Dockerfile            # Docker image build script
├── package.json          # Node dependencies
└── template.env          # Template for environment variables
```

Listing 1: Simplified folder structure of the repository. Only the most important folders and files are mentioned.

[1] In this context, monolith means that a single repository contains all parts of the application.
[2] GitHub Actions allow to run tasks like application building on every commit to a remote repository.

2.

Fundamentals

This chapter introduces the frameworks used in the project and explains why they have been chosen. It also provides an overview of the development process, explaining the techniques involved in building and deploying the application.

2.1. Frameworks

For the implementation, only open source frameworks were used. All frameworks are free to use, have large communities behind them, and can be customized if needed.

2.1.1. Next.js

Next.js is a React[3] framework for building Full-Stack web applications [2]. While React itself only allows client-side functionality, Next.js extends it by creating two server-side runtimes in order to run a complete application:

- Node.js[4] Runtime for server-side rendering and API route logic.
- Edge Runtime for middleware

While server-side rendering offers many advantages, such as enhanced security by keeping sensitive data on the server and reducing the need to send large dependency bundles to the client, browser APIs are not accessible on the server (Section 4.2). To address this, Next.js implements the `use client` directive. Files without this directive are executed on the server by default, whereas files marked with `use client` will be sent to the client first and then are executed there.

Next.js supports TypeScript. TypeScript is a syntactic superset for JavaScript that adds static typing, interfaces, and better tooling like auto-complete and type checking for the Integrated Development Environment (IDE). Because of these maintenance improvements compared to vanilla JavaScript the project is developed with TypeScript.

Next.js is sensitive to file naming and folder structure within the `app` directory. Every folder inside the `app` directory turns into a URL route. Files inside these folders cover different functionalities (Table 1).

File Type	Description
page.tsx	Defines a route segment and renders the content of a page. By default, it is server-rendered unless marked with `use client` to enable client-side execution.
layout.tsx	Defines a persistent layout that wraps multiple pages. Layouts maintain their state and do not re-render between navigations.
route.ts	Handles API routes. Allows defining backend logic for handling HTTP requests like GET, POST, PUT, and DELETE.
middleware.ts	Runs logic before a request completes using the Edge runtime. Common use cases include authentication, logging, and redirection.

Table 1: Next.js file types. There are more file types, which are not discussed in this introduction [3].

[3] React is an open source JavaScript library for building user interfaces.
[4] Node.js is an open source JavaScript runtime environment.

2.1.2. Prisma

Prisma is an Object Relational Mapping (ORM) [4]. It consists of three parts:

1. Prisma Client: A type-safe SQL query builder.
2. Prisma Migrate: A Command Line Interface (CLI) for applying database migrations without writing SQL.
3. Prisma Studio: A UI for database interactions during the development process.

By modifying the `schema.prisma` file, developers can update the database schema while simultaneously generating corresponding TypeScript types (Table 2). This enhances maintainability and reduces runtime errors.

Migrations are a way to version-control changes to the database schema. Prisma stores all migrations in a `migrations` directory. The SQL files in this directory can be then used to reconstruct the database.

Prisma Schema	Generated TypeScript Type	Generated SQL migration

`model User {`
`  id  Int @id @default`
`  email String @unique`
`  name  String?`
`}`
	`export declare type User = {`
	`  id: number`
	`  email: string`
	`  name: string | null`
	`}`
		`CREATE TABLE "User" (`
		`  "id" SERIAL PRIMARY KEY,`
		`  "email" VARCHAR(255) UNIQUE NOT NULL,`
		`  "name" VARCHAR(255)`
		`);`

Table 2: Changing `schema.prisma` generates TypeScript types and SQL migrations

2.1.3. Material UI

Material UI is an open source React component library that implements Google's Material Design [5]. It offers a wide range of components like buttons, chips, alerts, tables or icons. The components can be used in Next.js `page.tsx` or `layout.tsx` files as demonstrated in Listing 2.

```tsx
import Button from '@mui/material/Button';

export default function Page() {
  return <Button variant="contained">Hello world</Button>;
}
```

Listing 2: Hello world button with Material UI

2.1.4. Auth.js

Auth.js is an open source authentication library for JavaScript and TypeScript applications [6]. It provides a flexible and extensible solution for handling user authentication in web applications. In this application it is used to handle JSON Web Token (JWT) generation and consumption to authenticate users, so that they can change the courses in the database (Section 4.1.3.1).

2.2. Development Process

2.2.1. Local development

1. Linting with ESLint[5]

ESLint was used to enforce coding standards and detect potential issues in the code-base. By integrating a linter into the development workflow, common syntax errors, anti-patterns, and style inconsistencies can be identified and corrected early in the process. When an inconsistency is found, the application is not compilable and needs to be fixed first. In order to do so there is a `lint` script for the Node Package Manager (npm). This ensures code quality in production.

2. Type Safety Verification

TypeScript's `--noEmit` compiler-flag is used to perform static type checking without generating output files. This ensures that the code conforms to type definitions preventing build errors. By running the `typecheck` script with npm types can be checked. Type errors can then be fixed with the CLI output.

3. Local Feature Testing

Application features were tested in a local development environment to verify their functionality. In order to do so development environment variables like a development database connection string were set in a `.env` file.

4. Database Migrations

Database migrations were used like git commits for the database. Database testing is performed by pushing a new schema with the `prisma db push` CLI command. This command does not create a migration file but runs SQL against the database, which allows testing the database schema before creating a new migration. After successful implementation of a feature a database migration is then created with `prisma migrate dev`.

2.2.2. Deployment

Generally speaking there are three ways to deploy an application to a server (Table 3). As the application consists of three different services (Next.js application, PostgreSQL Database and cronjob server), a maintainable way is to package each service in a Docker[6] container and then connect them via Docker Compose. In comparison to other approaches like a Virtual Machine (VM) or a bare metal deployment, a container-based deployment allows to port the app to different machines anytime. As well containers share the Operating System (OS) host kernel and are therefore lightweight compared to VMs, because they do not need to boot an entire OS. Therefore a container-based deployment with Docker was chosen.

[5] ESLint is an open source linter for JavaScript

[6] Docker is a platform that packages applications and their dependencies into standardized containers, which can run consistently across different computing environments.

Method	Description	Advantages	Disadvantages
Container-Based	Lightweight virtualization that packages applications into isolated units sharing the host OS.	• Resource efficient • Portable • Simple maintenance • Easy horizontal scaling	• Limited isolation (shared kernel) • Requires orchestration for complex deployments
VM-Based	Complete virtualization of hardware resources with dedicated operating systems.	• OS flexibility	• Higher resource overhead • OS maintenance responsibility
Bare Metal	Direct deployment to physical servers without virtualization.	• High performance • Hardware control	• Slow provisioning • Reduced security due to missing isolation

Table 3: Comparing container, VM and bare metal deployment

2.2.3. Continuous Integration and Continuous Deployment

To ensure deployment, a CI/CD pipeline was implemented using GitHub Actions and Docker. This process automates building, testing, and deploying the application to a production environment. The pipeline follows three steps:

1. Building via GitHub Actions

On every push to the main branch, GitHub Actions triggers a workflow that builds the Docker image of the application. This ensures that any change in the code-base results in a deployable artifact. The build uses the `Dockerfile` located at the root of the repository and publishes the resulting image to the GitHub Container Registry (GHCR). This was chosen with respect to the requirement that there is no build process on the server of the University of Tübingen.

2. Web-hook Deployment trigger

The most straightforward way of deploying the images would be to access the server by Secure Shell (SSH) and run bash commands. But the University of Tübingen blocks the SSH protocol on port 22 during several holidays due to security concerns. To ensure deployment availability all over the year a web-hook based deployment was chosen. In order to do so a Express.js web server (`server.js`) is running and waiting for a POST request. Receiving a POST request the Express.js web server will execute the bash deployment script.

3. Bash deployment script

First the deployment script from Listing 3 pulls the remote repository via git. This ensures changes made to `docker-compose.yml` and `deploy.sh` will be respected in the deployment process. Currently the repository publicly accessible. Therefore neither for executing `git pull` nor for pulling the images from GHCR GitHub authentication is required. Should the repository become private in future, authentication can be managed via a GitHub token stored as an environment variable. This is already implemented in the deployment script. Having pulled the repository and the images the app will be deployed by Docker Compose.

```
#!/bin/bash
log "Starte Deployment-Prozess"
log "Lokaler Pfad: $LOCAL_REPO_PATH"
cd "$LOCAL_REPO_PATH" || handle_error "Konnte nicht ins Repo-Verzeichnis wechseln"
git pull || handle_error "Git pull fehlgeschlagen"
if [ ! -f docker-compose.yml ]; then
  handle_error "Docker Compose Datei nicht gefunden"
fi
if [ -n "${GITHUB_TOKEN:-}" ]; then
  log "Authentifiziere bei GitHub Container Registry..."
  echo "$GITHUB_TOKEN" | docker login ghcr.io -u "$GITHUB_USERNAME" --password-stdin ||
handle_error "Docker-Login fehlgeschlagen"
fi
log "Starte Docker Compose Up..."
docker compose up -d || handle_error "Docker Compose up fehlgeschlagen"
log "Bereinige ungenutzte Docker-Ressourcen..."
docker system prune -af --volumes || log "Warnung: Docker system prune fehlgeschlagen"
log "Deployment erfolgreich abgeschlossen!"
exit 0
```

Listing 3: Deployment script

3.

User Interface

The application features two separate frontends for different user groups: one for students and one for lecturers and admins.

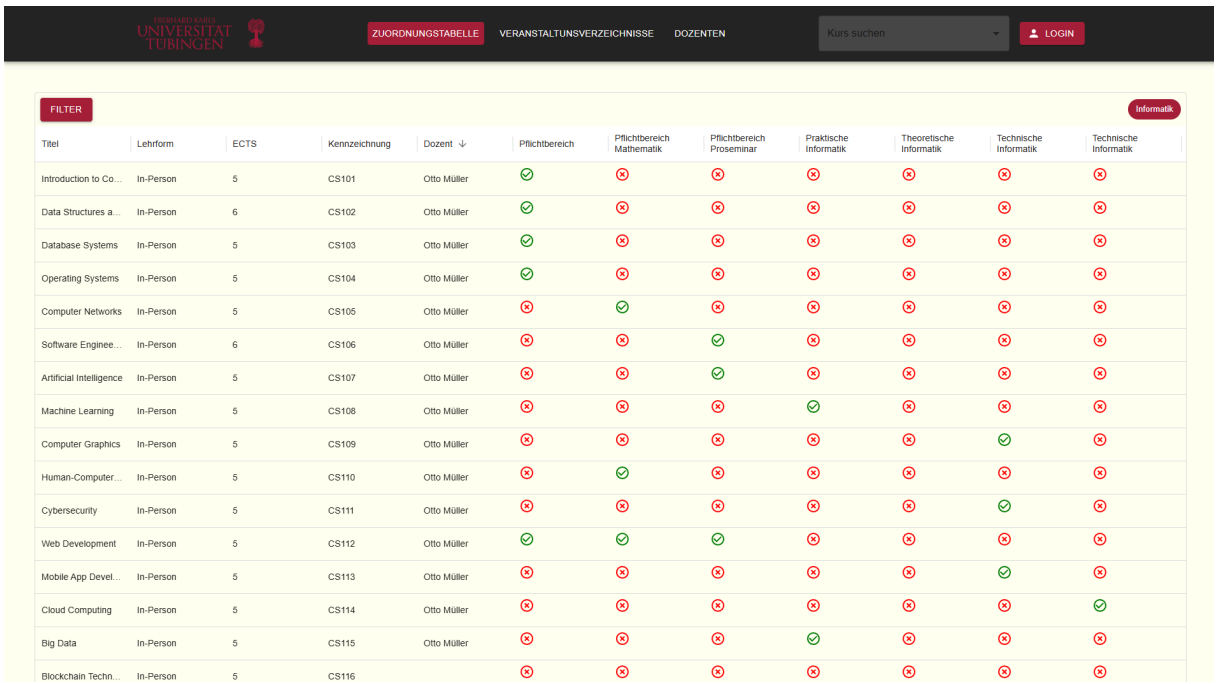
The Public Frontend is accessible without authentication and is used by students to explore available courses, check module assignments and view course details. It offers features like searching and filtering.

The Restricted Backend[7], on the other hand, requires authentication and provides administrative access to course data. Depending on the user role — lecturer or admin — users can view and edit courses. Admins have extended rights to manage all courses, users, and lecturer assignments, while lecturers are limited to their own content.

This chapter gives an overview of both interfaces. The Figures are provided with mockup data.

3.1. Public Frontend

The Public Frontend contains a redesigned course overview page with a new search functionality (Figure 3) and filtering options (Figure 4). Instead of covering all fields of study on a single page now only a single field of study is displayed. Fields of study can be changed through filters. By clicking on a course in the table or searching for a course a details page is opened (Figure 5). Additionally, a page containing all lecturers with their assigned courses is implemented (Figure 6).



Titel	Lehrform	ECTS	Kennzeichnung	Dozent	Pflichtbereich	Pflichtbereich Mathematik	Pflichtbereich Proseminar	Praktische Informatik	Theoretische Informatik	Technische Informatik	Technische Informatik
Introduction to Co…	In-Person	5	CS101	Otto Müller	✓	✗	✗	✗	✗	✗	✗
Data Structures a…	In-Person	6	CS102	Otto Müller	✓	✗	✗	✗	✗	✗	✗
Database Systems	In-Person	5	CS103	Otto Müller	✓	✗	✗	✗	✗	✗	✗
Operating Systems	In-Person	5	CS104	Otto Müller	✓	✗	✗	✗	✗	✗	✗
Computer Networks	In-Person	5	CS105	Otto Müller	✗	✓	✗	✗	✗	✗	✗
Software Enginee…	In-Person	6	CS106	Otto Müller	✗	✗	✓	✗	✗	✗	✗
Artificial Intelligence	In-Person	5	CS107	Otto Müller	✗	✗	✓	✗	✗	✗	✗
Machine Learning	In-Person	5	CS108	Otto Müller	✗	✗	✗	✓	✗	✗	✗
Computer Graphics	In-Person	5	CS109	Otto Müller	✗	✗	✗	✗	✗	✓	✗
Human-Computer…	In-Person	5	CS110	Otto Müller	✗	✓	✗	✗	✗	✗	✗
Cybersecurity	In-Person	5	CS111	Otto Müller	✗	✗	✗	✗	✗	✓	✗
Web Development	In-Person	5	CS112	Otto Müller	✓	✓	✓	✗	✗	✗	✗
Mobile App Devel…	In-Person	5	CS113	Otto Müller	✗	✗	✗	✗	✗	✓	✗
Cloud Computing	In-Person	5	CS114	Otto Müller	✗	✗	✗	✗	✗	✗	✓
Big Data	In-Person	5	CS115	Otto Müller	✗	✗	✗	✓	✗	✗	✗
Blockchain Techn…	In-Person	5	CS116		✗	✗	✗	✗	✗	✗	✗

Figure 3: Table with all courses and possible assignments

[7] The Restricted Backend may sound like a backend service but is a frontend feature in fact.

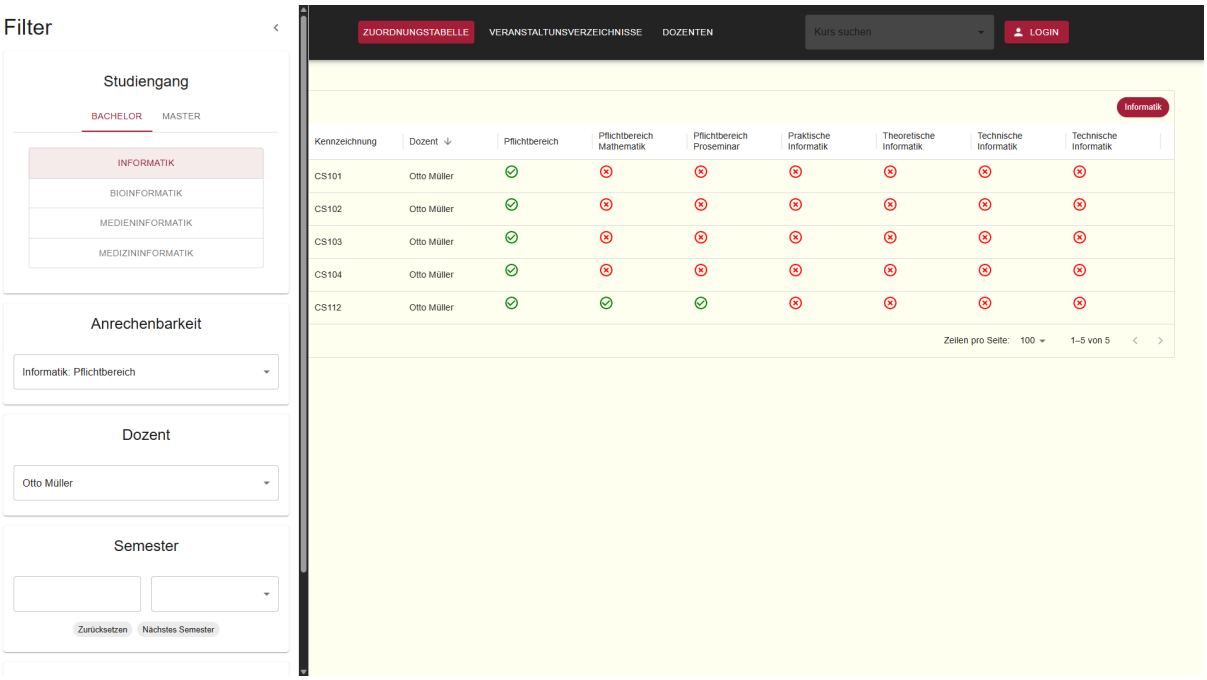


Figure 4: Filter possibilities for the course Table

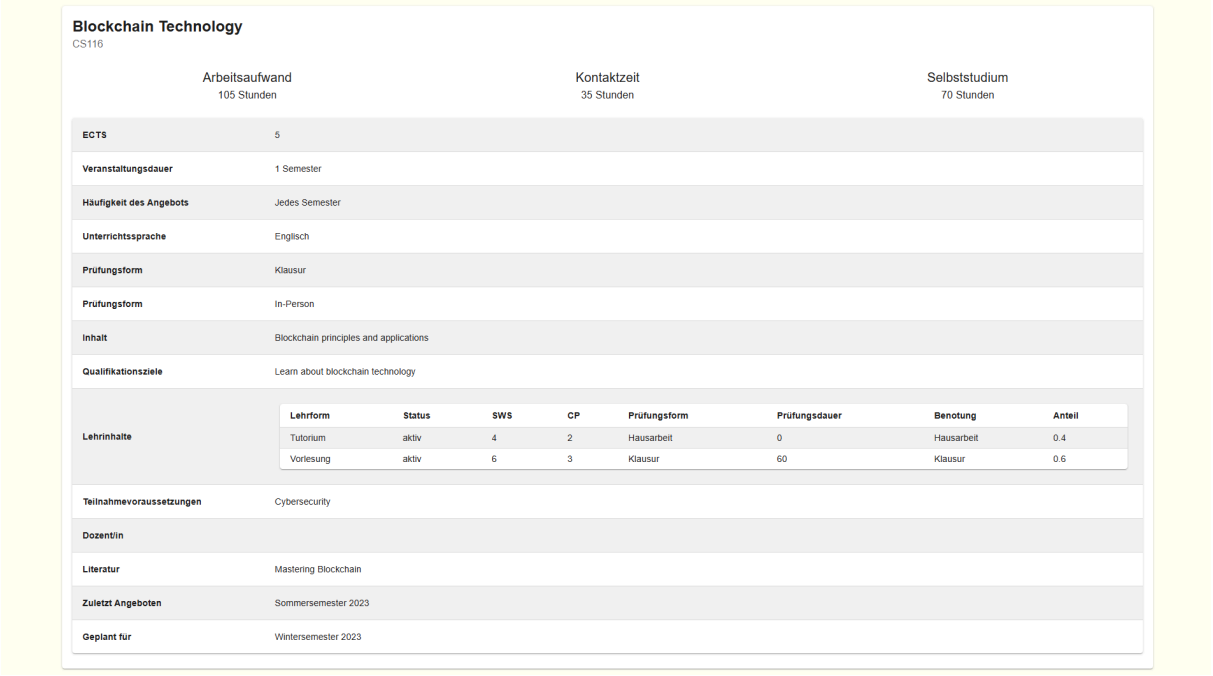


Figure 5: Course details

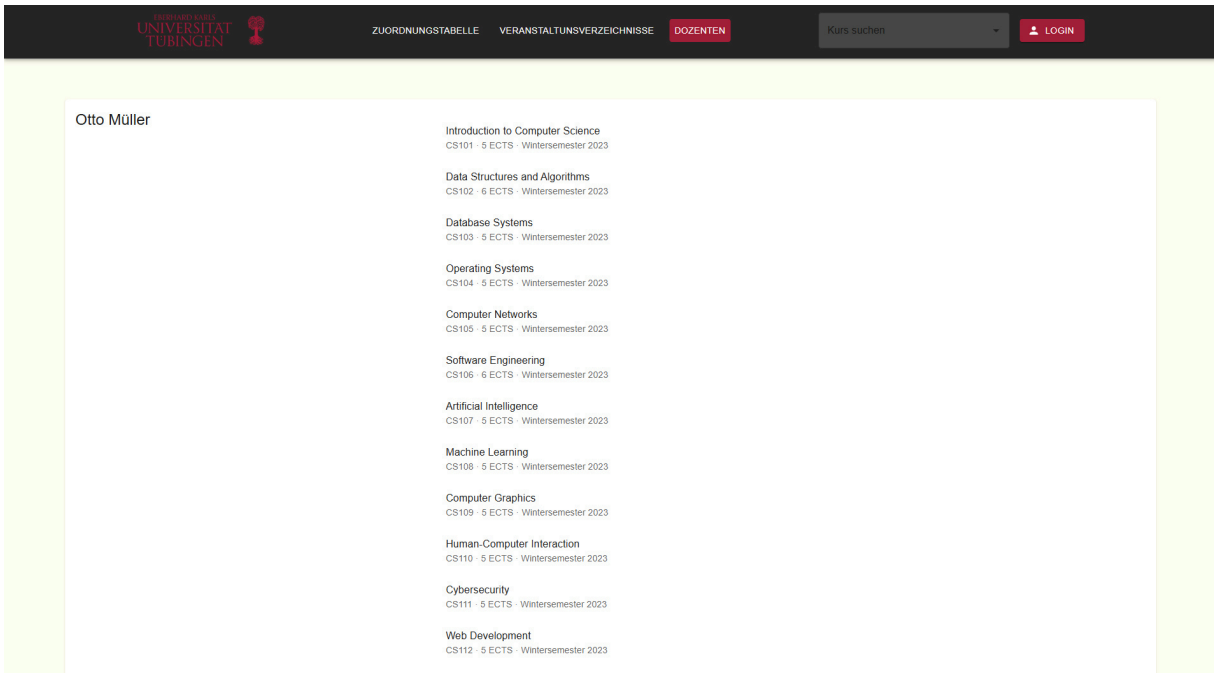


Figure 6: Courses assigned to their corresponding lecturer

3.2. Restricted Backend

The Restricted Backend enables authenticated users to change courses. There are two roles implemented in the authentication service. The `LECTURER` and the `ADMIN` role. Whereas lecturers can only adjust their own courses, admins can adjust any role. The role is assigned in the login process (Figure 7) and then the right backend will be rendered for the user.

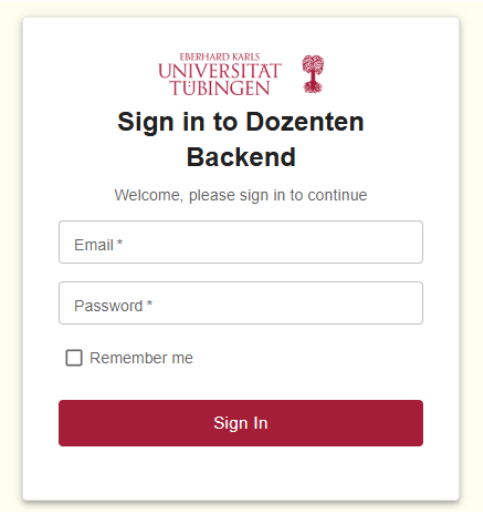


Figure 7: Authentication for backend access

3.2.1. Admin Backend

The Admin Backend implements an overview over all entries (Figure 8) and creation forms like the course creation form (Figure 9), which creates a new course for the Public Frontend. Before the data is written to the database, a preview option is available, giving users the chance to verify the course information. There are similar forms for the creation of a lecturer and a user.[8] In the user form a user entry can be assigned to a lecturer, enabling the user to access all course data assigned to the

[8]Lecturers and users have separate database tables. (Section 4.1.1)

lecturer. Every action which changes values in the database triggers a corresponding success or error notification. (Figure 10)

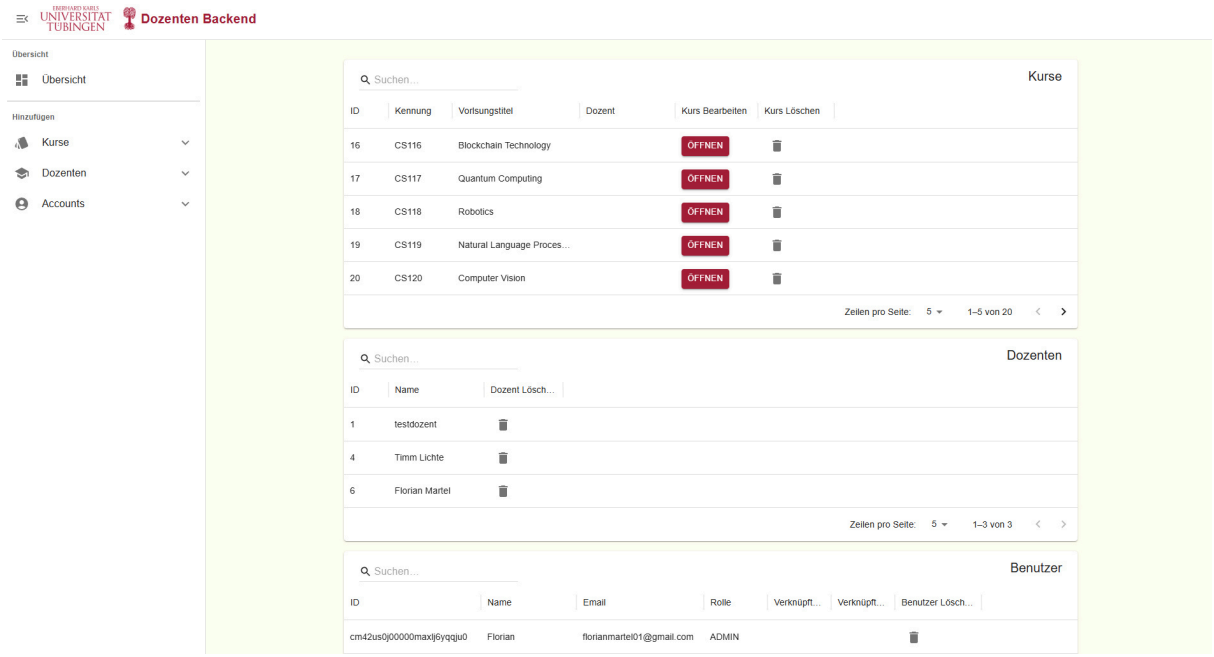


Figure 8: Overview for admins

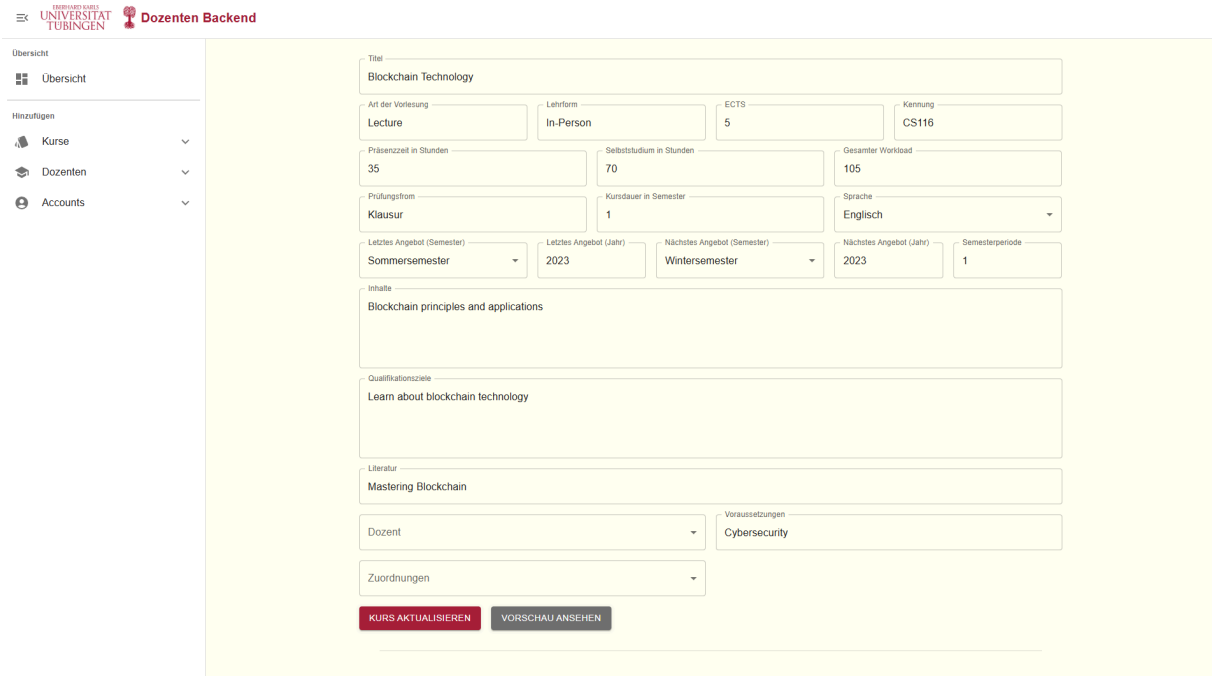


Figure 9: Form for editing (and creating) courses

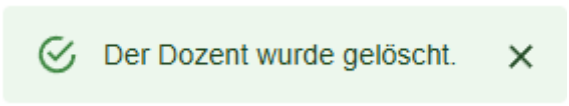


Figure 10: Notifications after actions

3.2.2. Lecturer Backend

The Lecturer Backend can not create courses, but only edit assigned ones. In order to do so, there is a simple overview with all assigned courses for the user (Figure 11).

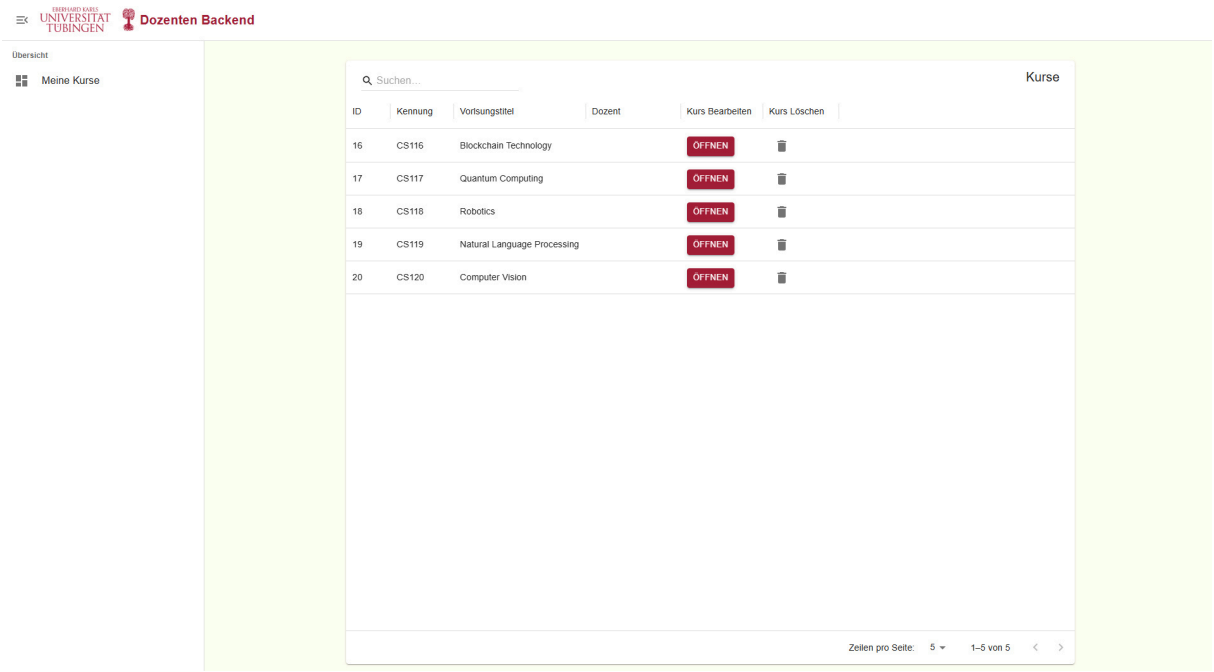


Figure 11: Overview for lecturers

4.

Implementation

In this chapter, the implementation patterns that demonstrate how a Full-Stack framework can blur the traditional network boundary are presented. The focus lies on database interaction and security considerations. Subsequently, different rendering strategies are analyzed to illustrate how Full-Stack frameworks enable the combination of server-side and client-side rendering.

4.1. Database interaction

This section covers how the application interacts with the database using Prisma as an ORM. It explains how the database schema is structured, how data is queried or modified through Prisma's TypeScript API, and how these operations are integrated into both server- and client-side components.

4.1.1. Database Schema

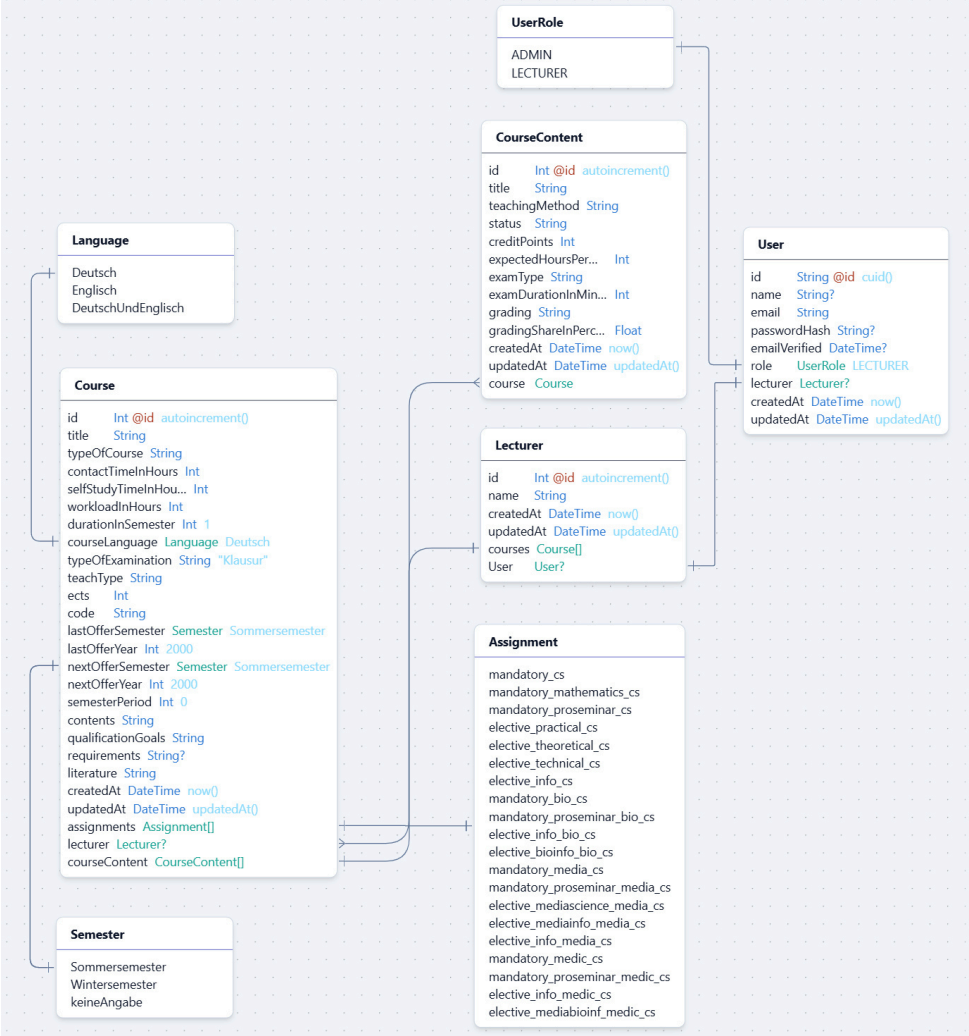


Figure 12: Database Schema implemented in `schema.prisma`[9]

[9]The `Assignment` enum has been shortened due to limited space

The structure of the database schema in Figure 12 was developed with respect to practical requirements. First, not every lecturer needs an account in the system — some just need to be listed as the responsible person for a course. Because of that, courses are linked directly to `Lecturer` entries instead of the `User` model, which is used for authentication.

Second, a course can include different types of content, like lectures, tutorials, or exams. To keep things flexible, `Course` has a one-to-many relation to `CourseContent`. The `CourseContent` rows containing events like exams and lectures will be rendered on the course details page in a separate table.

Finally, courses can be assigned to different modules across various field of studies. Instead of managing this on field of study level, all possible assignments are bundled into a single enum. This enables to assign a course to assignments from different field of studies For example the course "Mathematik 1: Analysis" could be assigned to the field of study "Informatik" on the assignment "Pflichtbereich Informatik" as well as on the field of study "Bioinformatik" on the assignment "Pflichtbereich Bioinformatik".

4.1.2. Queries And Mutations

Prisma provides a TypeScript-based client library that makes it easy to execute SQL queries from the server [7]. These queries can be used to interact with the database in a type-safe and readable way. In the context of this project, most data is fetched on the server before rendering the page (Section 4.2.1), which keeps the client lightweight.

At the same time, there are cases where the client needs to trigger database operations directly. For those situations, Server Actions offer a way to bridge the gap.

4.1.2.1. Client Side Queries Using Server Actions

Server Actions are a Next.js feature that blur the boundary between frontend and backend development. They allow client components to call backend logic without having to create a REST or GraphQL API. A Server Action is created by using the `use server` directive on top of the file. This tells Next.js to execute the code on the server, even if it's being called from the client.

Under the hood, Listing 4 creates a POST endpoint[10], which runs the query on the server. The query will retrieve the `Course` row with the connected `Lecturer` and `CourseContent` rows. The ergonomic benefit of this pattern lies in the usage of this action on the client. Instead of fetching an API endpoint Next.js allows to treat the endpoint as the function `getCourse`.

Listing 5 shows how this is implemented in practice: when a user clicks on a course, the Server Action fetches the relevant data and renders the course details. Using this approach keeps the code clean and readable, and it also opens the door to simple error handling with `try` and `catch` blocks, which is useful for further development and handling unauthorized access (Section 4.1.3.2).

[10]The POST endpoint will be open for everyone. There is some basic security logic implemented in Server Actions, but to make the endpoint truly secure, custom access rules should be added (Section 4.1.3.2).

```
"use server";
export async function getCourse(id: number): Promise<CourseWithLecturerCourseContent | null> {
    const resp = await prisma.course.findUnique(
        {
            where: {
                id: id
            },
            include: {
                lecturer: true,
                courseContent: true
            }
        }
    );
    return resp;
}
```

Listing 4: Example implementation of a basic Server Action. The code will be executed on the server, but the function can be used on the client.

```
"use client";
function CourseDialog({ open, courseId }: PreviewDialog) {
    const [course, setCourse] = React.useState<CourseWithLecturerCourseContent | null>(null);

    React.useEffect(() => {
        if (courseId) {
            getCourse(courseId).then(setCourse)
        }
    }, [courseId]);

    if (!courseId || !course) return null;

    return (
        <Dialog open={open} maxWidth={"xl"} fullWidth>
            <DialogTitle>Kursdetails</DialogTitle>
            <DialogContent>
                <CourseDetails course={course} />
            </DialogContent>
            <DialogActions>
                <Button variant="contained" onClick={() => {
                    setPreviewDialogOpen(prev => ({ ...prev, open: false }));
                }} color="secondary"
                >
                    Schließen
                </Button>
            </DialogActions>
        </Dialog >
    );
}
```

Listing 5: Using Server Actions in client components

4.1.2.2. API Routes

API Routes are another Next.js feature that blur the boundary between frontend and backend development. They allow to implement backend logic directly within the application's folder structure, without the need for a separate server or framework.

To create an API route, a `route.ts` file is added within a folder in the `app` directory. Inside that file, functions can be exported that correspond to HTTP methods like GET, POST, PUT, or DELETE.

In this application, API Routes are used in two scenarios:

- **Authentication**: Auth.js uses API Routes to manage login and session handling.
- **Semester updates**: A cron job triggers an API Route that updates semester values in the database when a new semester begins.

Listing 6 implements the semester API Route, without the need of a stand-alone backend runtime. Using the property `semesterPeriod`, which is set by the lecturer, all course rows will be updated to ensure the correct semester is displayed for the next offering of the course. Nevertheless using an API Route for this use case creates another dependency: a cronjob server. A lightweight alternative for the update process could be using database functions. This will be discussed in Section 5.2.

```
export async function POST(request: Request) {
    const validApiKey = process.env.CRONJOB_API_KEY;
    const providedKey = request.headers.get('x-api-key');

    if (!validApiKey || providedKey !== validApiKey) {
        return NextResponse.json({ error: 'Unauthorized' }, { status: 401 });
    }

    try {
        const courses: Course[] = await prisma.course.findMany();
        const { semester: currentSemester, year: currentYear } = getCurrentSemester();
        courses.map(async (course) => {
            const checkUpdate = course.nextOfferYear <= currentYear
                                && course.nextOfferSemester === currentSemester
            if (checkUpdate) {
                const { semester, year } = increaseSemester(course.nextOfferSemester,
                                           course.nextOfferYear, course.semesterPeriod);
                await prisma.course.update({
                    where: {
                        id: course.id
                    },
                    data: {
                        lastOfferYear: course.nextOfferYear,
                        lastOfferSemester: course.nextOfferSemester,
                        nextOfferYear: year,
                        nextOfferSemester: semester
                    }
                });
            }
        });
        return NextResponse.json({ message: 'Semester updated successfully.' }, { status: 200 });

    } catch (error) {
        if (error instanceof Error) {
            return NextResponse.json({ error: error.message }, { status: 500 });
        }
        return NextResponse.json({ error: 'An unknown error occurred' }, { status: 500 });
    }
}
```

Listing 6: Implementation of an API route for semester updates

4.1.3. Security

This section outlines how authentication and access control are implemented in the project. It covers how users are authenticated and how access to sensitive operations is restricted based on user roles.

4.1.3.1. Authentication

Generally speaking there are two concepts for authentication.

Session-based authentication operates by maintaining session state on the server. When a user logs in successfully, the server creates a new session with a unique identifier, stores session data in a database, and returns the session ID to the client as a cookie. For each request, the server has to validate this session ID against stored sessions.

Token-based authentication offers a stateless alternative. With JWT, the server generates a token containing user identity information and permissions, which is then cryptographically signed [8]. The token is returned to the client. The server validates the token's signature without database interaction. This approach allows verification without maintaining session states. On the other hand it is impossible to invalidate JWT once they are set.

Given the app requirements, JWT authentication was chosen for the following reasons:

- **No Major Security Concerns**: The user base is relatively small, reducing the risk of token misuse. While JWTs cannot be invalidated once issued, the limited number of users minimizes potential threats.
- **Simpler Implementation**: JWT authentication requires no additional session storage on the server, simplifying the database architecture.

The JWTs are stored in HttpOnly-Cookies[11] and every HTTP request will send these cookies in the request header [9]. This allows user authentication to be handled entirely on the server by reading the session from the headers. While it's also possible to access cookies on the client using browser APIs, this project currently has no use case that requires client-side authentication logic.

To protect the Restricted Backend from unauthenticated access, the app uses AuthGuards like Listing 7. AuthGuards are custom React server components designed to wrap both server and client components. They check if a valid session exists and, if not, redirect the user to the login page. This keeps protected parts of the UI secure without cluttering each individual component with access logic.

```
export const AuthGuard: React.FC<AuthGuardProps> = async ({ children }) => {
    const session = await auth();
    if (!session) {
        redirect(paths.signIn);
    }
    if (session) {
        return <>{children}</>;
    }
    return null;
};
```

Listing 7: Basic AuthGuard component

4.1.3.2. Limiting Access to User Groups

Not every user should have access to all data. Lecturers should only be able to view and edit their own courses, while admins can manage everything. Since JWTs are included in the request headers, the server can extract and decode them to determine the user's role. This allows custom security logic to be implemented based on user groups. However, guarding UI components isn't enough — Server Actions also need protection. By default, Server Actions are exposed as POST endpoints. Without proper checks, anyone could trigger these actions. To prevent that, access control must be enforced inside the Server Action itself.

To help mitigate some of the risks, Next.js assigns Action IDs to Server Actions. These IDs allow the client to reference the correct action without exposing implementation details. By default, the IDs are

[11] Using the HttpOnly flag will ensure that the cookie isn't changeable through client side JavaScript.

cached for up to 14 days. This mechanism helps to limit the risk of abuse if an authentication layer is missing — but it's not a substitute for proper access control. Server Actions should always be treated like public HTTP endpoints [10].

To enforce access control, the helper function shown in Listing 8 is used to verify whether a user has permission to access a course. This function is called within the Server Action itself. If access is denied, an error is thrown (Listing 9). On a stand-alone frontend/backend architecture this error could not be handled on the client. Instead it would be necessary to return an error HTTP response and then handle the response on the client. However, since Next.js Server Actions can be used like regular functions on the client, the error can be caught directly using try and catch, as shown in Listing 10. This allows for cleaner and more ergonomic error handling in React components.

As an alternative to implementing custom logic in Server Actions, PostgreSQL's Row Level Security (RLS) [11] could also be used to restrict access at the database level. Although RLS is more secure, it would be more difficult to display error messages to the user. For this reason, the project uses application-level logic for access control.

```
async function checkLecturerAccess(courseId: number) {
    const session = await auth();
    if (session?.user?.role == UserRole.ADMIN) return true;

    const checkCourse: Course | null = await prisma.course.findUnique({
        where: {
            id: courseId
        },
    });
    const lecAccess = checkCourse?.lecturerId == session?.user?.lecturerId &&
                      session?.user?.role == UserRole.LECTURER
    return lecAccess
}
```

Listing 8: Helper function for Securing Server Actions

```
    if (!await checkLecturerAccess(id)) throw new Error("Not authorized");
```

Listing 9: Example of Error throwing inside a Server Action

```
  try {
      const resp = await updateCourse(courseData.id, course);
      notification.show(resp.title + ' wurde erfolgreich aktualisiert.', {
          severity: "success",
          autoHideDuration: 3000,
      });
  } catch (error) {
      notification.show('Fehler beim Aktualisieren des Kurses:' + error, {
          severity: "error",
          autoHideDuration: 3000,
      });
  }
```

Listing 10: Error handling on the client after triggering Server Actions

4.2. Finding the best render strategy

In general, all websites follow the same Request-Response-Lifecycle:

1. User Action: The cycle begins when a user interacts with a web interface by navigating to a URL, clicking a link, or submitting a form.
2. HTTP Request: The client constructs an Hypertext Transfer Protocol (HTTP) request containing the method (GET, POST etc.), headers, and any necessary data.
3. Server Processing: The server processes the request through routing, authentication, data operations, and application logic.
4. HTTP Response: The server returns an HTTP response containing a status code, headers, and the requested resources.
5. Client: The client interprets the status code and parses the resources to render the UI.

Next.js is capable of handling different rendering strategies (Table 4). The chosen rendering strategy determines at which stage of the Request-Response-Lifecycle the page content is generated. In this project client-side rendering, server-side rendering and Incremental Static Regeneration (ISR) were used for different use cases.

Render strategy	Description
Client-Side Rendering	The browser receives minimal HTML and JavaScript, then renders the page dynamically on the client. All data fetching and page assembly occurs in the browser after initial load.
Server-Side Rendering	The server generates the full HTML for a page on each request. The client receives HTML, enabling faster initial page loads and improved Search Engine Optimization (SEO) compared to client-side rendering.
Static Site Generation	Pages are pre-rendered at build time rather than on each request. This approach creates static HTML files that can be served quickly from a Content Delivery Network (CDN), providing optimal performance for pages with content that doesn't change frequently.
Incremental Static Regeneration	An extension of Static Site Generation that allows specific pages to be regenerated in the background after deployment. This enables static content to be updated without rebuilding the entire site.
Streaming SSR	An advanced form of server-side rendering where HTML is streamed to the client progressively as it's generated, allowing parts of the page to be displayed before the entire response is complete.

Table 4: Comparing different render strategies. There are more possible render strategies, which are not implemented by Next.js. [12]

4.2.1. Server-Side Rendering

Server-side rendering is the default render method, when a `page.tsx` file is created within the Next.js `app` directory. One major advantage of server-side rendering is that updated data from the database is reflected immediately after a page reload. Additionally, content is rendered as part of the initial page

load, contributing to a faster First Contentful Paint[12].

For example, Listing 11 shows how the dashboard of an authenticated user is rendered entirely on the server. First, the user's role is extracted from the request headers. Based on that role, the corresponding database queries are constructed. If the user is a lecturer, only their own courses are fetched and passed to the `CourseOverview` component. If the user is an admin, two additional queries are executed to retrieve all `User` and `Lecturer` entries. In this case, a more detailed version of the dashboard is rendered, providing broader management capabilities.

```
export default async function Dashboard() {
    const session = await auth();
    const userRole = session?.user?.role;

    let query: FindManyQuery = {
        include: {
            lecturer: true,
        },
    }
    if (userRole == UserRole.Lecturer) {
      query = { ...query, where: { lecturerId: session.user.lecturerId } }
    }
    const allCourses = await prisma.course.findMany(query);
    if (userRole == UserRole.Lecturer)
        return (
            <CourseOverview courses={allCourses} />
        );
    const allLecturers = await prisma.lecturer.findMany();
    const allUsers = await prisma.user.findMany(
        {
            include: {
                lecturer: true
            }
        }
    );
    return (
        <Grid container spacing={2}>
            <Grid size={12}>
                <CourseOverview courses={allCourses} />
            </Grid>
            <Grid size={12}>
                <LecturerOverview lecturers={allLecturers} />
            </Grid>
            <Grid size={12}>
                <UserOverview users={allUsers} />
            </Grid>
        </Grid>
    )
}
```

Listing 11: Using server-side rendering to implement role-based logic

4.2.2. Client-Side Rendering

Client-side rendering allows the usage of browser APIs and React hooks such as `useState` or `useEffect`. This makes it the preferred choice for interactive components — especially those that require dynamic updates based on user input. A common pattern is to render the overall structure of the page (such as navigation bars or grid layouts) on the server, and delegate only small, interactive components to the client as implemented in Figure 13.

[12]First Contentful Paint is a performance metric that measures the time from when a page starts loading to when any part of its content (text, image, canvas, etc.) is first rendered on the screen.

Rendering the table component on the client simplifies the implementation of filtering. While the initial course data is fetched on the server, it is passed to a client-side component where filtering takes place. Instead of rendering all entries at once, users can dynamically choose which courses they want to display in the table. Each time a filter is changed, the table component re-renders on the client without needing to communicate with the server.[13]

This behavior is made possible by React's `useState` hook [13], which updates the component state and automatically triggers a re-render whenever the state changes.

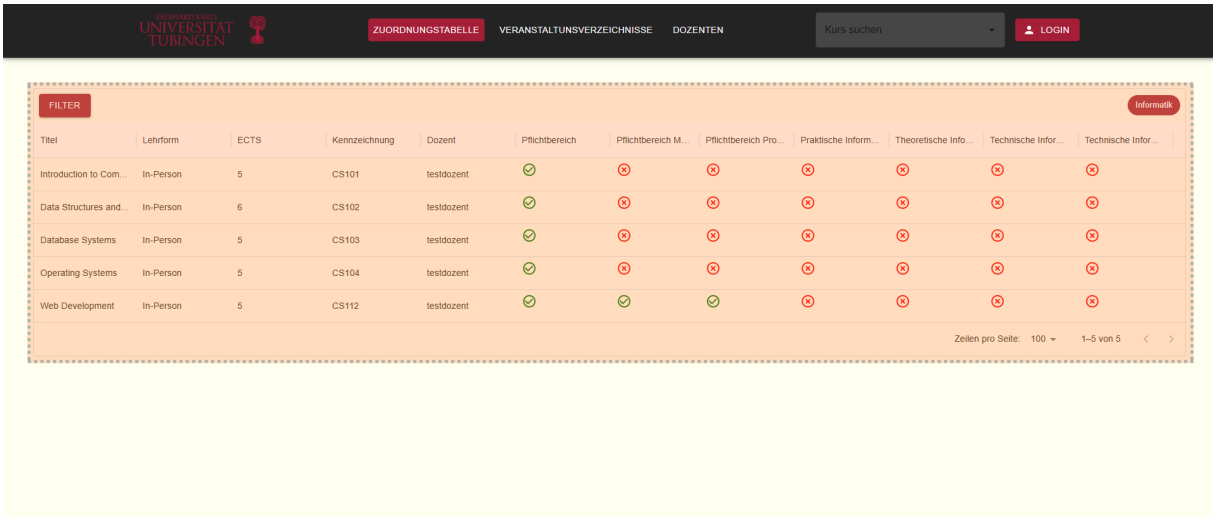


Figure 13: Demonstration of a client component. Only the orange marked component is rendered on the client. The rest of the page is served statically or rendered on the server.

4.2.3. Incremental Static Regeneration

In contrast to the Restricted Backend, the Public Frontend does not require role-based rendering or real-time data updates. All users see the same content, and the information displayed — such as course listings — does not change frequently. These conditions make it a great use case for ISR. Next.js supports ISR through the `revalidate` directive, which allows pages to be statically generated at build time and then updated in the background at specified intervals. This approach combines the performance of static pages with the flexibility of server-rendered updates.

Listing 12 shows how ISR is implemented for the course details overview. The revalidate directive defines a revalidation interval of 3600 seconds. Within this period, Next.js follows the following process:

1. Initially renders the page at build time or upon the first request
2. Caches the generated HTML and serves it to all visitors
3. On the first request after the revalidation interval, triggers a background regeneration of the page
4. Replaces the cached content with the updated version once regeneration is complete

This mechanism effectively combines the benefits of static site generation with those of server-side rendering. It reduces load on the database and amplifies page load times, while still keeping the content reasonably up to date.

Although this time-based strategy is already quite efficient, Next.js supports a more flexible approach: tag-based revalidation. Instead of waiting for a timer to expire, content can be revalidated when data actually changes. This works well with REST API–based data fetching via the fetch API as shown in

[13] After the initial page load, the user could lose the internet connection and filtering would still work since all logic is handled locally.

Listing 13. In this example the function `revalidateTag('posts')` can be then used in Server Actions to regenerate all pages implementing the tagged endpoint. In the current application this could be implemented when fetching API Routes, but due to the usage of Prisma as ORM instead of a REST API the taggable fetch API is not usable. Instead the `unstable_cache` function has to be used for tagging ORM database request for revalidation [14]. Although this would be possible this project does not implement `unstable_cache` because the API is unstable and may change in the future.

```
export const revalidate = 3600

export default async function Veranstaltungsverzeichnisse() {
    const rows = await prisma.course.findMany(
        {
            include: {
                lecturer: true,
                courseContent: true
            }
        }
    );
    return (
        <Stack spacing={5} sx={{ mx: 12, mt: 8 }}>
            {rows.map((row) => (
                <CourseDetails key={row.id} course={row} />
            ))}
        </Stack>
    );
}
```

Listing 12: Using the revalidate directive to enable ISR

```
const data = await fetch('https://api.vercel.app/blog', {
    next: { tags: ['posts'] },
})
```

Listing 13: Tagging routes with the fetch API

5.

Discussion

This chapter reflects on the advantages of the chosen architecture, points out areas where the project could be improved, and looks ahead at how future developments might shape similar applications.

5.1. Advantages of the Full-Stack Approach

1. Reduced Network Boundaries

Using Next.js as a Full-Stack framework significantly reduces the traditional separation between frontend and backend development. With features like Server Actions, client components can directly interact with server-side logic and the database — avoiding the need for manually defining REST API endpoints. Server and client rendering are separated using the `use client` directive. The application is designed to render as much as possible on the server, while client-side rendering is reserved for interactive components, such as filtering tables. The Full-Stack approach allows using the use case specific advantages of different render strategies within the same code-base.

2. Improved Developer Experience

Thanks to Prisma and TypeScript, the application benefits from strong type safety throughout the entire stack. Linting and type checking are integrated using npm scripts to catch issues early in the development process. Additionally, a CI pipeline ensures that the application is built and tested automatically, removing the need for manual deployments on the production server.

5.2. Future Improvements

1. Test Coverage

Currently, the project includes only basic checks like linting and type checking. There are no unit or integration tests for actual functionality. Introducing proper test coverage, including unit tests, integration tests and end-to-end tests, would improve the application's reliability.

2. Mobile Optimization

The application is currently designed for desktop screens. While it is accessible on mobile devices, the user experience is not fully optimized. Future iterations could include responsive design improvements to better support smaller screens.

3. Enhanced Cache Invalidation

The current implementation of ISR uses time-based revalidation. While this is effective, a more advanced method could be achieved using the `unstable_cache` API once it becomes stable. This would allow for tag-based invalidation, enabling updates to be triggered on actual data changes, rather than time intervals.

4. Simplifying The Service Architecture

The semester update process currently requires a separate cronjob server, which triggers an API route to run database mutations. A simpler alternative could involve using a PostgreSQL extension such as pg_cron [15] to run a database function allowing the cronjob and the function to run directly within the database engine. This removes the need for an external cronjob service.

5.3. Outlook

The implementation presented in this thesis illustrates a unified Full-Stack architecture that collapses the traditional boundaries between frontend and backend. By using a framework like Next.js in combination with TypeScript-aware tools (Prisma, Auth.js, Material UI), teams can maintain a single code-base where data models, API logic, and UI components share the same types and conventions.

But despite the ergonomic benefits and shared data models enabled by Full-Stack architectures, this development approach also presents several limitations. One key drawback is performance. While TypeScript offers static typing and improved tooling, it remains a superset of JavaScript and thus inherits its runtime characteristics. In compute-intensive scenarios, it cannot match the raw performance of languages like Java with Spring Boot or .NET. These alternatives benefit from real static compilation, which allow for deep performance optimizations that JavaScript engines, such as V8 used in Node.js, are inherently limited in providing.

Although recent efforts in the TypeScript ecosystem — such as the 2025 introduction of the new compiler written in Go — aim to optimize the development process, these improvements focus on build-time performance. The runtime environment remains bound to JavaScript, and as a result, the performance ceiling is relatively fixed. Therefore, from a long-term perspective, although the Full-Stack Next.js monolith brings several ergonomic benefits it may not represent the future of web application development. While the ergonomic benefits simplify development workflows and reduce boilerplate, they do not necessarily translate into technical superiority. An equivalent application could be implemented using a distributed architecture — separating frontend and backend services — without sacrificing performance.

Furthermore, the growing integration of AI-assisted development tools like GitHub Copilot, Claude Code, and Cursor AI might fundamentally shift the importance of developer ergonomics. These tools are capable of reasoning over entire code-bases — including data schemas — regardless of whether type annotations exist. This could reduce the cognitive overhead traditionally mitigated by TypeScript. As AI models become more context-aware and deeply embedded in the development process, the trade-off between ergonomic convenience and architectural flexibility may shift. It is possible that applications developed with distributed frontends and backends could reach similar levels of productivity.

In conclusion, although Next.js monoliths currently offer a developer-friendly approach, their long-term viability will depend on how future tools and performance demands evolve. The increasing influence of AI in development workflows may favor architectures that prioritize performance over ergonomic consistency.

Bibliography

[1] Florian Martel, "Application GitHub repository." Accessed: Apr. 23, 2025. [Online]. Available: https://github.com/flovalle1/digitales-modulhandbuch

[2] Vercel, Inc., "Next.js Documentation." Accessed: Feb. 25, 2025. [Online]. Available: https://nextjs.org/docs

[3] Vercel, Inc., "Next.js Project Structure." Accessed: Mar. 21, 2025. [Online]. Available: https://nextjs.org/docs/app/getting-started/project-structure

[4] Prisma Data, Inc., "Prisma Documentation." Accessed: Feb. 25, 2025. [Online]. Available: https://www.prisma.io/docs/orm/overview/introduction/what-is-prisma

[5] Material UI SAS, "Material UI Documentation." Accessed: Apr. 15, 2025. [Online]. Available: https://mui.com/material-ui/getting-started/

[6] Balázs Orbán and Team, "Auth.js Documentation." Accessed: Apr. 26, 2025. [Online]. Available: https://authjs.dev/getting-started

[7] Prisma Data, Inc., "Prisma TypeScript library." Accessed: Apr. 14, 2025. [Online]. Available: https://www.prisma.io/docs/orm/prisma-client

[8] Internet Engineering Task Force, "RFC7519: JSON Web Token (JWT)." Accessed: Feb. 28, 2025. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519

[9] Internet Engineering Task Force, "RFC6265: HTTP State Management Mechanism." Accessed: Feb. 28, 2025. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6265

[10] Vercel, Inc., "Next.js Securing Server Actions." Accessed: Apr. 17, 2025. [Online]. Available: https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations#security

[11] The PostgreSQL Global Development Group, "Postgres Documentation on RLS." Accessed: Feb. 27, 2025. [Online]. Available: https://www.postgresql.org/docs/current/ddl-rowsecurity.html

[12] V. N. S. K. Challa, "Comprehensive Analysis of Modern Application Rendering Strategies: Enhancing Web and Mobile User Experiences," *Journal of Engineering and Applied Sciences Technology*, vol. 4, no. 4, pp. 1–6, 2022, doi: 10.47363/JEAST/2022(4)248.

[13] Meta Platforms, Inc, "React useState API Reference." Accessed: Apr. 14, 2025. [Online]. Available: https://react.dev/reference/react/useState

[14] Vercel, Inc., "Next.js unstable_cache API." Accessed: Mar. 13, 2025. [Online]. Available: https://nextjs.org/docs/app/api-reference/functions/unstable_cache

[15] Citus Data, "Pg_cron - a simple cron-based job scheduler." Accessed: Mar. 19, 2025. [Online]. Available: https://github.com/citusdata/pg_cron