Abschlussprüfung Sommer 2025 Fachinformatiker für Anwendungsentwicklung

Dokumentation zur betrieblichen Projektarbeit

Terminalgerät

Entwicklung eines Terminalgeräts zur Zeiterfassung und Serverüberwachung

Abgabedatum: 15.06.2025

Prüfungsbewerber:

Nikita Kolesnikow

Ludwig-Thoma Straße 39

97422 Schweinfurt

Ausbildungsbetrieb:

Werner & Krapf GbR

auftretend unter dem Namen GoDigital24

Am Mühlbach 1

97475 Zeil am Main

Inhaltsverzeichnis

1 Abbildungsverzeichnis

2 Tabellenverzeichnis

3 Abkürzungsverzeichnis

API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
ORM	Object-Relational Mapping
REST	Representational State Transfer
SPA	Single Page Application
SQL	Structured Query Language
UI	User Interface
JWT	JSON Web Token

1 Einleitung

1.1 Einleitung

In dieser Projektdokumentation wird der Ablauf des Abschlussprojektes erläutert, das der Autor im Rahmen seiner Abschlussprüfung zum Fachinformatiker für Anwendungsentwicklung durchführt. Das Projekt wird bei der Firma GoDigital24 umgesetzt, dem Ausbildungsbetrieb des Autors. GoDigital24 ist ein junges IT-Dienstleistungsunternehmen mit Sitz in Zeil am Main, das sich auf die Digitalisierung und Automatisierung von Geschäftsprozessen in kleinen und mittelständischen Unternehmen spezialisiert hat. Aktuell besteht das Team aus drei Personen.

Ziel dieser Dokumentation ist es, die im Rahmen des Projekts durchzuführenden Schritte von der Planung bis zum Deployment nachvollziehbar darzustellen und dabei geeignete Diagramme und Dokumente zur Veranschaulichung heranzuziehen. Der Fokus liegt dabei auf einer klaren Strukturierung des Projektablaufs sowie auf der praktischen Umsetzung innerhalb der betrieblichen Umgebung von GoDigital24.

1.2 Projektbegründung

Die Entscheidung zur Entwicklung eines eigenen Terminalsystems wurde getroffen, da im Arbeitsalltag von GoDigital24 bislang keine einheitliche Lösung zur Erfassung von Arbeitszeiten und zur Aufgabenverwaltung vorhanden ist. Gerade für ein kleines und noch wachsendes Unternehmen ist es hilfreich, einen Überblick darüber zu bekommen, wie viel Zeit für bestimmte Tätigkeiten aufgewendet wird und welche Aufgaben aktuell anstehen.

Das Terminalgerät soll jedem Mitarbeiter zur Verfügung stehen und die Zeiterfassung automatisieren. Darüber hinaus soll es ermöglichen, Aufgaben an Kollegen zu übermitteln und aktuelle Aufgaben direkt am Arbeitsplatz im Blick zu behalten. Auf diese Weise können Abläufe übersichtlicher gestaltet und Missverständnisse bei der Aufgabenverteilung vermieden werden.

Auch die Überwachung von Webseiten und Servern ist im Projekt vorgesehen. Bei technischen Problemen erhalten Mitarbeiter entsprechende Hinweise direkt auf dem Gerät. Das verbessert die Reaktionszeit bei Ausfällen und trägt zu einem reibungsloseren Arbeitsablauf bei.

Eine externe Lösung kommt in diesem Fall nicht infrage, da vorhandene Tools oft nicht flexibel genug sind oder nicht alle benötigten Funktionen kombinieren. Eine eigene Umsetzung bietet die Möglichkeit, genau auf die internen Anforderungen einzugehen und das System bei Bedarf zu erweitern.

1.3 Projektziel

Das Ziel des Projekts ist die eigenständige Entwicklung eines webbasierten Terminalsystems, das bei GoDigital24 zur Erfassung von Arbeitszeiten, zur Anzeige von Aufgaben sowie zur Überwachung von Webseiten und Servern eingesetzt wird. Jeder Mitarbeiter soll ein eigenes Terminalgerät auf Basis eines Raspberry Pi erhalten. Die Software des Terminals wird als Webanwendung realisiert und über ein integriertes Display bedient.

Das Terminal ermöglicht es den Mitarbeitern, ihren Arbeitsbeginn und ihr Arbeitsende bequem zu registrieren. Die Zeiten werden dabei automatisch erfasst und zentral gespeichert, wodurch eine transparente und nachvollziehbare Zeiterfassung entsteht. Um sicherzustellen, dass nur berechtigte Personen Zugriff auf die Funktionen haben, wird ein Authentifizierungsmechanismus integriert. Die Anmeldung am Terminal kann dabei flexibel entweder über eine RFID-Karte oder durch Eingabe eines persönlichen PIN-Codes erfolgen, je nachdem, was der jeweilige Nutzer bevorzugt.

Neben der Zeiterfassung bietet das Terminal die Möglichkeit, dem Mitarbeiter seine aktuellen Aufgaben anzuzeigen. Diese Aufgaben werden zuvor zentral vergeben und auf dem Display des Terminals übersichtlich dargestellt, sodass jeder Nutzer stets einen klaren Überblick über seine anstehenden Tätigkeiten behält.

Ein weiterer Bestandteil des Projekts ist die technische Überwachung der firmeneigenen Webseiten und Server. Diese Funktionalität wird durch denselben Server umgesetzt, der auch für die zentrale Verwaltung der Terminaldaten zuständig ist und mit dem Framework NestJS entwickelt wird. Der Server überprüft regelmäßig, ob bestimmte Webseiten erreichbar sind, ob SSL-Zertifikate gültig sind und ob die Serverauslastung innerhalb akzeptabler Grenzen liegt. Bei Störungen oder Ausfällen wird automatisch eine Warnmeldung auf dem Display des betroffenen Terminals angezeigt, was eine schnelle Reaktion durch die Mitarbeiter ermöglicht.

Zur grafischen Darstellung des Systemzustands wird eine Anbindung an Grafana umgesetzt. Prometheus erfasst dabei kontinuierlich die relevanten Metriken und speichert sie, sodass die aktuelle Situation der Infrastruktur jederzeit visuell nachvollzogen werden kann. Für die technische Umsetzung des Gesamtsystems kommen moderne Webtechnologien zum Einsatz: Vue 3 für das Frontend, Tailwind CSS für das Styling, NestJS für das Backend sowie PostgreSQL als relationale Datenbank. Diese Kombination ermöglicht eine effiziente, wartbare und zukunftssichere Umsetzung des Projekts.

1.4 Projektabgrenzung

Aufgrund des auf 80 Stunden begrenzten Zeitrahmens für das Abschlussprojekt war es erforderlich, bestimmte inhaltliche und technische Einschränkungen vorzunehmen, um die

Fertigstellung innerhalb dieses Rahmens sicherzustellen. Im Projekt wurde ein Instrument in Form eines iframes vorbereitet, das zukünftig die Einbindung von Grafiken und Visualisierungen zur Darstellung der Serververfügbarkeit, Systemzustände und weiterer Monitoring-Daten ermöglicht. Die eigenständige Erstellung und Implementierung eines vollständigen Grafana-Dashboards zur Visualisierung dieser Daten wurde jedoch aus Zeitgründen nicht realisiert und gehörte nicht zum Umfang dieses Projekts.

2 Projektplanung

2.1 Projektphasen

Für die Umsetzung des Projekts standen insgesamt 80 Stunden zur Verfügung. Um diese Zeit effizient zu nutzen, wurde das Projekt in mehrere aufeinanderfolgenden Phasen unterteilt. Ein grober Zeitplan, der diese Phasen umfasst, ist in Tabelle 1 dargestellt.

Zur besseren Übersicht und genaueren Steuerung des Projektablaufs wurden die Hauptphasen zusätzlich in spezifischere Arbeitsschritte gegliedert. Eine detaillierte Auflistung dieser Aufgaben ist im Anhang unter dem Abschnitt „Ursprünglicher Detaillierter Zeitplan“ zu finden.

Projektphase	**Geplante Zeit in Stunden**
• Analysephase	6
• Entwurfsphase	10
• Implementierungsphase	45
• Qualitätskontrolle	7
• Dokumentation	12
Summe	***80***

Tabelle 1: Ursprünglicher Grober Zeitplan

2.2 Abweichung vom Projektantrag

Während der Umsetzung des Projekts mussten einige Anpassungen gegenüber der ursprünglichen Planung vorgenommen werden. Ursprünglich war vorgesehen, im Falle technischer Störungen zusätzlich eine Benachrichtigung per Telefon zu versenden. Diese Funktion wurde jedoch nicht umgesetzt, da sowohl die zur Verfügung stehende Zeit als auch die persönliche Erfahrung im Umgang mit derartigen Benachrichtigungsdiensten nicht ausreichten, um eine zuverlässige Lösung innerhalb des Projektzeitraums zu realisieren.

2.3 Ressourcenplanung

Für die Umsetzung des Projekts wurde ein moderner Technologiestack verwendet, der eine effiziente Entwicklung und eine gute Benutzererfahrung ermöglicht. Die Webanwendung basiert im Frontend auf Vue.js 3 mit der Composition API und wird mithilfe von Vite gebündelt, was schnelle Ladezeiten und eine angenehme Entwicklungsumgebung bietet. Für das Styling kam eine Kombination aus Tailwind CSS und SCSS zum Einsatz, unterstützt durch Animationen mit Framer Motion und Animate.css, um die Oberfläche dynamischer und benutzerfreundlicher zu gestalten.

Die Kommunikation zwischen Frontend und Backend erfolgt über eine REST-API, umgesetzt mit Axios. Diagramme zur Visualisierung verschiedener Daten, etwa im Zusammenhang mit den Arbeitszeiten oder Servermetriken, werden mit Chart.js dargestellt.

Das Backend wurde mit NestJS und TypeScript realisiert. Als Datenbank wird PostgreSQL verwendet, wobei der Zugriff über TypeORM erfolgt.

Zur Absicherung der Anwendung wurde eine Authentifizierung mit JWT in Kombination mit Passport.js implementiert. Die API-Dokumentation wird automatisch mit Swagger erzeugt, was eine einfache Nachvollziehbarkeit der Schnittstellen gewährleistet. Für die Sicherstellung der Codequalität wurden ESLint und Prettier genutzt, während Tests mit Jest durchgeführt wurden. Die gesamte Anwendung wurde zudem containerisiert, um mit Docker eine konsistente Entwicklungs- und Laufzeitumgebung sicherzustellen.

2.4 Entwicklungsprozess

Die Entwicklung des Projekts erfolgte in mehreren aufeinanderfolgenden Iterationen, wobei der Fokus stets auf der Erstellung funktionsfähiger Zwischenstände lag. Diese konnten frühzeitig getestet, bewertet und verbessert werden, was eine flexible Reaktion auf neu auftretende Anforderungen ermöglichte. Die Umsetzung begann mit einer grundlegenden Anforderungsanalyse, auf deren Basis die technische Gesamtarchitektur sowie die benötigten Datenstrukturen definiert wurden.

Anschließend wurden Frontend und Backend getrennt voneinander entwickelt, um eine klare Struktur und bessere Wartbarkeit sicherzustellen. Das Backend wurde mithilfe von NestJS umgesetzt, wobei die Kommunikation über eine REST-Schnittstelle realisiert wurde. Für das Frontend kam Vue 3 mit Composition API zum Einsatz, ergänzt durch Tailwind CSS für die Gestaltung. Die Anbindung beider Komponenten erfolgte über klar definierte Schnittstellen.

Ein besonders wichtiger Aspekt des Entwicklungsprozesses war der Aufbau einer stabilen und flexiblen Architektur auf der Client-Seite. Die einzelnen Komponenten wurden so gestaltet, dass sie leicht wiederverwendet und unabhängig voneinander erweitert werden können. Durch eine saubere Trennung von Zuständigkeiten und eine modulare Struktur ist es nun möglich, neue Funktionen mit geringem Aufwand zu integrieren oder bestehende zu modifizieren. Das sorgt nicht nur für eine langfristige Wartbarkeit, sondern auch für eine hohe Skalierbarkeit des Systems.

Zur Qualitätssicherung wurde Git zur Versionsverwaltung eingesetzt, um Änderungen nachvollziehbar und strukturiert zu halten. Ergänzend kamen statische Codeanalyse mit ESLint, typisierte Entwicklung mit TypeScript und automatische Formatierung durch Prettier zum Einsatz. Der Entwicklungsprozess wurde zudem durch eigene Testszenarien und regelmäßige manuelle Prüfungen begleitet.

2.5 Ist-Analyse

Vor Beginn des Projekts existierte keine vergleichbare Anwendung im Unternehmen – es gab weder ein Terminalsystem noch eine integrierte Lösung zur Serverüberwachung oder Aufgabenverwaltung. Das Projekt startete vollständig bei null und erforderte daher die komplette Konzeption und Umsetzung aller Komponenten.

Sowohl die Benutzeroberfläche des Terminals als auch der dahinterliegende Server wurden von Grund auf neu entwickelt, um genau den internen Anforderungen zu entsprechen. Ziel war es, eine zentrale, erweiterbare Plattform zu schaffen, die sowohl die tägliche Organisation unterstützt als auch wichtige technische Zustände sichtbar macht. Da keine bestehende Lösung angepasst werden konnte, bot die Eigenentwicklung maximale Flexibilität, Kontrolle und Erweiterbarkeit.

2.6 Soll-Konzept

Die Webanwendung soll Nutzerinnen und Nutzern eine zentrale Übersicht über den aktuellen Arbeitstag bieten. Nach dem Login erscheint ein Dashboard mit Begrüßung, Datum und Uhrzeit sowie mehreren hilfreichen Widgets, darunter ein Timer zur Zeiterfassung, eine Wetteranzeige, eine analoge Uhr und eine grafische Darstellung des bisherigen Arbeitstags mit Informationen zu Arbeits- und Pausenzeiten. Eine Aufgabenliste im Checkbox-Format zeigt alle aktuellen Aufgaben an. Die Aufgaben können hier nicht erstellt oder angenommen werden, sondern erscheinen automatisch, nachdem sie zuvor über die Kanban-Ansicht definiert wurden.

Über die Seitenleiste erhalten die Nutzer Zugriff auf weitere Bereiche wie das Kanban-Board, Statistiken zur Arbeitszeit, grundlegende Einstellungen sowie einen Bereich für Server-Monitoring. Für diesen wurde ein Iframe vorbereitet, der die Einbindung eines externen Dashboards wie zum Beispiel mit Grafana ermöglicht, auch wenn das konkrete Dashboard selbst nicht Teil des Projekts war.

Die Benutzeroberfläche ist modular aufgebaut und auf einfache Erweiterbarkeit ausgelegt. Die Architektur des Frontends wurde bewusst flexibel gestaltet, um spätere Anpassungen und Wiederverwendung einzelner Komponenten zu erleichtern. Rechte und Funktionen sind rollenbasiert organisiert, wodurch die Anwendung unterschiedlichen Benutzergruppen gerecht wird. Die Kommunikation mit dem Backend erfolgt über eine REST-API, die den zuverlässigen Datenaustausch zwischen Client und Server sicherstellt.

Im Anhang ist ein Use-Case-Diagramm beigefügt, das die geplanten rollenbasierten Anwendungsfälle anschaulich darstellt.

2.7 Wirtschaftlichkeitsanalyse

Die Hardwarekosten für ein Raspberry-Pi-Terminal belaufen sich auf rund 56 € bis 63 € für das Raspberry Pi 4 Model B mit 4 GB RAM. Hinzu kommt ein geeignetes Display mit integriertem Gehäuse für etwa 130 €, sowie ein USB-RFID-Reader für rund 16 €. Insgesamt ergeben sich somit circa 202 € bis 209 € pro Gerät.

Ein modernes Tablet wie das Google Pixel Tablet (128 GB) kostet in Europa rund 499 €, also mehr als das Zweieinhalbfache der Pi-Lösung.

Da die Personalkosten für beide Varianten identisch sind (80 Stunden Entwicklungsarbeit), liegt der wesentliche Unterschied in der Hardwareinvestition. Ein Terminal auf Raspberry-Pi-Basis ist mit etwa 205 € deutlich günstiger als ein Tablet für 499 €. Trotz der höheren Anschaffungskosten bietet das Tablet keinen echten Mehrwert für die spezifischen Funktionen wie GPIO-Integration, RFID oder spätere Hardwareerweiterungen.

Fazit: Die Nutzung eines Raspberry-Pi-Terminals ist wirtschaftlich klar vorteilhaft und effektiv kostengünstiger, insbesondere bei mehreren Geräten. Zudem bietet das Terminal Vorteile in Modularität, Kontrolle und Anpassbarkeit, die bei Tablets deutlich eingeschränkter sind.

3 Entwurfsphase

Die Entwurfsphase diente dazu, erste Konzepte für die technische Struktur sowie das visuelle und funktionale Design der Webanwendung zu entwickeln.

3.1 Zielplattform

Die entwickelte Anwendung läuft auf einem speziell für diesen Zweck eingerichteten Terminal, das auf einem Raspberry Pi 4 basiert. Das System ist für die Verwendung mit einem fest verbauten Display konzipiert, wodurch eine direkte Bedienung am Gerät möglich ist.

Diese Hardware-Plattform wurde bewusst gewählt, um eine kostengünstige und zugleich flexible Lösung zu schaffen, die genau auf die Anforderungen des Projekts zugeschnitten ist. Die Software ist auf die Auflösung und Eingabemöglichkeiten des Displays optimiert.

3.2 Architekturdesign

Die Architektur des Frontends basiert auf einem modularen Prinzip, bei dem jeder UI-Komponent wie ein eigenständiges Bauteil funktioniert. Zum Beispiel ist eine Schaltfläche (Button) komplett in sich abgeschlossen, inklusive aller zugehörigen Zustände, Farben, Animationen und Verhaltensweisen. Dies wird durch zentrale Dateien umgesetzt, die alle Varianten und Stile eines Components beschreiben und steuern.

Ein Beispiel ist die button.ts-Datei, die mittels der Bibliothek class-variance-authority verschiedene Varianten des Buttons definiert. Diese Datei importiert den Button-Komponent und exportiert Varianten, die verschiedene Farbschemata (default, destructive, outline, secondary, ghost, link) sowie Größen (default, sm, lg, icon) abdecken. Dadurch ist die Schaltfläche flexibel und konsistent an verschiedenen Stellen der Anwendung nutzbar.

Für die Zustandsverwaltung und das Teilen von Daten zwischen Komponenten wird Pinia als zentraler Store verwendet, kombiniert mit Vue-Hooks zur optimalen Wiederverwendbarkeit von Logik und Reaktivität. Die Komponenten sind so entworfen, dass sie unabhängig voneinander funktionieren, was Wartbarkeit und Erweiterbarkeit erheblich erleichtert.

Diese modulare Frontend-Architektur sorgt für klare Trennung von Verantwortung, hohe Wiederverwendbarkeit und einfache Anpassbarkeit der UI-Elemente bei gleichzeitiger Wahrung eines konsistenten Designs und Verhaltens.

3.3 Entwurf vom Datenmodell

Das Datenmodell der Systemarchitektur ist um eine zentrale Entität, die Mitarbeiter (employees), herum konzipiert. Diese Tabelle dient als primärer Ankerpunkt, an den sich alle anderen systemrelevanten Daten anbinden. Jeder Mitarbeiter ist eindeutig über eine ID

identifizierbar und verfügt über grundlegende Informationen wie Name, PIN, E-Mail, Telefonnummer, Abteilung und Position.

Zur detaillierten Erfassung von Arbeitszeit und Aufgabenmanagement werden weitere spezifische Tabellen verwendet, die über Fremdschlüsselbeziehungen mit der Mitarbeitertabelle verknüpft sind. Die Tabelle rfid_tags speichert Informationen zu den RFID-Karten der Mitarbeiter. Da ein Mitarbeiter mehrere RFID-Tags besitzen kann, besteht hier eine Eins-zu-Viele-Beziehung zur Mitarbeitertabelle. Ähnlich verhält es sich mit der Tabelle work_time_sessions, die individuellen Arbeitszeiten, Pausen und die Dauer der Arbeitssitzungen pro Mitarbeiter erfasst.

Für die Aufgabenverwaltung wird ein Kanban-System genutzt, das durch die Tabellen kanban_columns und kanban_cards repräsentiert wird. Die Tabelle kanban_columns definiert die vordefinierten Spalten des Kanban-Boards wie "Neue Aufgaben", "In Bearbeitung" und "Abgeschlossen", wobei die Reihenfolge der Spalten über ein order-Feld gesteuert wird. Die kanban_cards-Tabelle speichert die eigentlichen Aufgabenkarten, die jeweils einem Mitarbeiter und einer Kanban-Spalte zugeordnet sind. Dies ermöglicht eine flexible Zuordnung von Aufgaben zu Mitarbeitern und eine visuelle Darstellung des Arbeitsfortschritts.

Die Implementierung dieses Datenmodells basiert auf autoinkrementierenden Primärschlüsseln und der konsequenten Verwendung von Fremdschlüsseln zur Gewährleistung der Datenintegrität. Indizes auf häufig verwendeten Feldern optimieren die Abfrageleistung, und Zeitstempel erlauben die Nachverfolgung von Erstellungs- und Änderungsdaten. Durch die Verwendung von Reihenfolgefeldern (order) wird zudem eine intuitive Sortierung von Elementen in der Benutzeroberfläche ermöglicht. Diese robuste Struktur bildet die Grundlage für die effiziente Erfassung von Arbeitszeiten, das Aufgabenmanagement, die Zugangskontrolle über RFID-Tags und die Generierung aussagekräftiger Berichte zur Mitarbeiterproduktivität.

Ein relationales Datenmodell befindet sich im Anhang.

3.4 Entwurf vom Frontend

Die Benutzeroberfläche wurde mit Vue.js, Radix Vue für Komponenten und Tailwind CSS für das Styling entwickelt, ergänzt durch Lucide-Icons und Designsystem-Komponenten. Sie bietet Kernfunktionen wie ein Dashboard, Metriken, ein Kanban-Board, Benutzereinstellungen und Benachrichtigungen. Die Navigation ist intuitiv über ein festes Seitenmenü in Dashboard-Layout.vue organisiert, das Farbcodierungen und Animationen nutzt. Die Routings erfolgen über Vue Router, mit geschützten Pfaden.

Das Design ist minimalistisch und funktional, mit Fokus auf Benutzerfreundlichkeit durch Farbcodierung, Animationen, responsives Design und Lucide-Icons. Informationen werden übersichtlich in Karten dargestellt, während modale Fenster für wichtige Aktionen dienen. Die

Interaktion erfolgt über interaktive Schaltflächen und Drag-and-Drop im Kanban-Board. Visuelle Hierarchie wird durch Typografie, Badges und Spinner für Status und Ladevorgänge verstärkt.

Die strukturierte Informationsdarstellung wird durch ein Karten-Interface, eine Rasterstruktur und hierarchische Überschriften sowie Tabs gewährleistet. Die Benutzeroberfläche wurde spezifisch an die Projektanforderungen angepasst, indem spezialisierte Komponenten für Systemmetriken implementiert wurden. Die Designprinzipien konzentrieren sich auf Minimalismus, Funktionalität, Konsistenz, Zugänglichkeit, Reaktionsfähigkeit und Skalierbarkeit durch einen komponentenbasierten Ansatz.

3.5 Entwurf vom Backend

Die technische Architektur des Systems ist darauf ausgelegt, eine robuste und wartbare Lösung zu bieten, die auf gängigen Webstandards und Best Practices basiert.

Die Web-Anwendung bietet Kernfunktionen wie die Authentifizierung und Autorisierung von Benutzern, umfassendes Aufgabenmanagement, Überwachung der Arbeitszeiten, Personalverwaltung und Server-Monitoring. Diese Funktionalitäten werden über ein REST-API bereitgestellt, das einen einheitlichen Präfix /api für alle Endpunkte verwendet. Für die Interaktion mit diesem API kommen standardisierte HTTP-Methoden zum Einsatz: GET zum Abrufen von Daten (z.B. Aufgabenlisten, Mitarbeiterinformationen), POST zum Erstellen neuer Einträge (z.B. Aufgaben, Benutzerkonten), PUT/PATCH zum Aktualisieren bestehender Daten und DELETE zum Entfernen von Einträgen. Die API-Struktur ist durch Swagger umfassend dokumentiert, und eine globale Validierung mittels ValidationPipe sowie standardisierte DTOs (Data Transfer Objects) für Anfragen und Antworten gewährleisten die Datenintegrität.

Die Sitzungsverwaltung wird durch JWT-Authentifizierung (JSON Web Token) realisiert, wobei ein Bearer-Token in den Anfrage-Headern übergeben wird. Guards schützen die Routen und stellen sicher, dass nur autorisierte Benutzer auf bestimmte Bereiche zugreifen können. Ein umfassendes Log-System erfasst alle HTTP-Anfragen mit Zeitstempeln und protokolliert die Anfragekörper, was für Debugging und Monitoring von großer Bedeutung ist. Die Anwendungskonfiguration erfolgt zentral über ein config-Modul, das Umgebungsvariablen nutzt, und wird mittels Docker-Compose verwaltet, was die Bereitstellung und Skalierung vereinfacht.

Um die Effizienz zu maximieren und Redundanzen zu vermeiden, setzt das System auf Code-Wiederverwendung. Dies wird durch die modulare Architektur von NestJS, die Nutzung gemeinsamer DTOs und Schnittstellen sowie den Einsatz von Guards und Decorators zur Wiederverwendung von Logik erreicht. Die Zugriffsrechte werden durch ein rollenbasiertes Zugriffssystem verwaltet, wobei Guards und Decorators die erforderlichen Berechtigungen überprüfen und durchsetzen.

Die Lesbarkeit und Wartbarkeit des Codes haben hohe Priorität. TypeScript sorgt für starke Typisierung, während die modulare Struktur die Übersichtlichkeit erhöht. Die Swagger-Dokumentation unterstützt das Verständnis der API. ESLint und Prettier stellen eine konsistente Code-Formatierung sicher, und Tests für kritische Komponenten gewährleisten die Zuverlässigkeit und Stabilität der Anwendung.

3.6 Maßnahmen zur Qualitätssicherung

Der Entwicklungsprozess legt großen Wert auf die Sicherstellung der Softwarequalität, Effizienz und Wartbarkeit. Dies wird durch eine Kombination aus präventiven Maßnahmen, stringentem Änderungsmanagement und umfassenden Teststrategien erreicht.

3.6.1 Versionskontrolle und Entwicklungspraktiken

Das Änderungsmanagement erfolgt über die Versionskontrolle mit Git. Jede Codeänderung wird mit einer klaren Beschreibung dokumentiert. Neue Funktionen werden in separaten Entwicklungszweigen bearbeitet, um den Hauptzweig stabil zu halten. Sensible Konfigurationsdateien wie „.env“ werden von der Versionskontrolle ausgeschlossen. Dieser Ansatz ermöglicht die präzise Verfolgung von Änderungen und die Rückkehr zu früheren Versionen bei Bedarf.

3.6.2 Testen, Dokumentation und Fehlerbehebung

Es werden automatisierte Tests (mittels Jest) zur kontinuierlichen Funktionsprüfung eingesetzt. Das API wird über spezialisierte Tools, einschließlich Swagger und Postman, getestet. Eine interaktive API-Dokumentation ist ein zentrales Element für Entwickler und Tester. Die Fehlerbehebung und API-Prüfung profitieren von einer benutzerfreundlichen Schnittstelle, die eine schnelle Überprüfung von Eingabedaten ermöglicht und verständliche Fehlermeldungen liefert. Standardisierte DTOs (Data Transfer Objects) tragen zur Klarheit der Anfragen und Antworten bei.

3.6.3 Früherkennung von Fehlern

Die frühzeitige Fehlererkennung ist ein Eckpfeiler des Entwicklungsprozesses. Dies wird durch automatische Code-Stilprüfung, automatisierte Tests bei jeder Änderung und Typüberprüfung gewährleistet. Eine konsistente Code-Formatierung, die detaillierte API-Dokumentation und die Kontrolle von Datenbankänderungen ergänzen diese Maßnahmen und stellen sicher, dass potenzielle Probleme frühzeitig identifiziert werden.

4 Implementierungsphase

4.1 Implementierung der Datenstrukturen

Die Architektur des Datenbanksystems basiert auf einem modernen ORM-Ansatz (Object-Relational Mapping) unter Verwendung von TypeORM. Dies manifestiert sich in der Konfiguration in typeorm.config.ts und der Verwendung spezifischer Dekoratoren innerhalb der Entitätsklassen.

4.1.1 Struktur und Beziehungen

Tabellen und ihre Beziehungen werden im Code durch Klassen-Entitäten definiert, die von TypeORM-Dekoratoren Gebrauch machen. Hierzu gehören **@Entity()** zur Kennzeichnung einer Klasse als Datenbankentität, **@Column()** zur Definition von Tabellenfeldern und **@ManyToOne()** zur Festlegung von Beziehungen zwischen Tabellen, wie beispielsweise in den Entitäten für **Employee**, **RfidTag** und **WorkTimeSession** ersichtlich. Zur Definition von Einschränkungen und Regeln im Datenmodell werden Annotationen wie **@PrimaryGeneratedColumn()** für autoinkrementierende Primärschlüssel, **@Column({ length: X })** zur Längenbegrenzung von Zeichenketten, **@Column({ nullable: true })** für Nullwerte sowie **@CreateDateColumn()** und **@UpdateDateColumn()** für die automatische Verwaltung von Zeitstempeln verwendet.

Der zentrale Zugriffspunkt zur Datenbank wird durch die Konfiguration in **typeorm.config.ts**, das TypeOrmModule in **app.module.ts** und die Injektion von Repositories mittels **@InjectRepository()** in Services realisiert. Beziehungen zwischen Entitäten werden über Dekoratoren wie **@ManyToOne(() => Employee)** beschrieben und durch Repository-Methoden wie **findOne()** oder **find()** unter Verwendung der relations-Option gehandhabt.

4.1.2 Konfiguration und Erweiterbarkeit

Informationen zur Datenbankverbindung werden sicher in Umgebungsvariablen **(.env-Datei)** sowie in den Konfigurationsdatei **typeorm.config.ts** und gespeichert. Für die Verarbeitung von Beziehungen wird explizites Laden über die relations-Option in Abfragen genutzt.

Die Erweiterbarkeit und Anpassungsfähigkeit des Datenmodells wird durch die modulare Anwendungsstruktur, den Einsatz von Migrationen zur Schemaanpassung, die Typisierung mittels TypeScript und die Möglichkeit zur Hinzufügung neuer Entitäten durch neue Klassen gewährleistet. Änderungen an der Datenbank bei Modelländerungen erfolgen mittels Migrationen, die über **migration:generate** generiert und mit **migration:run** angewendet werden, wobei die automatische Synchronisierung **(synchronize: false)** deaktiviert ist.

4.1.3 Wartbarkeit und Qualitätssicherung

Der objektorientierte Ansatz und die Typisierung tragen maßgeblich zur Wartbarkeit des Projekts bei. Dies äußert sich in der strengen Typisierung aller Entitäten und ihrer Felder, der Bereitstellung von Autovervollständigung in IDEs, der Typüberprüfung zur Kompilierungszeit und einer klaren Datenstruktur durch Klassen. Die Validierung von Daten wird durch class-validator-Dekoratoren unterstützt, und die API-Dokumentation wird durch Swagger-Dekoratoren generiert, was das Verständnis und die Nutzung der Schnittstellen erheblich verbessert.

4.2 Implementierung vom Frontend

Die Organisation des Projekts ist darauf ausgelegt, eine klare Struktur zu bieten und die Wiederverwendung von Komponenten zu maximieren.

4.2.1 Struktur der Quellcode-Ordner

Der **src/**-Ordner, der das Herzstück der Anwendung bildet, ist logisch in mehrere Unterverzeichnisse gegliedert. **assets/** enthält statische Ressourcen wie Bilder und Schriftarten. **components/** beherbergt die verschiedenen Komponenten der Anwendung, wobei diese weiter in **ui/** für allgemeine Benutzeroberflächenelemente und **pages/** für spezifische Seitenkomponenten unterteilt sind. Hilfsbibliotheken und Dienstprogramme sind im **lib/**-Ordner zu finden. Die Konfiguration der Routen befindet sich in **router/**, Skripte und API-bezogene Logik in **scripts/**. Die **views/** enthalten die eigentlichen Seitendarstellungen, während **layout/** für die Maket-Komponenten zuständig ist, die die allgemeine Struktur der Anwendung definieren.

4.2.2 UI-Komponenten

Innerhalb des **src/components/ui/-**Ordners ist eine umfangreiche Sammlung wiederverwendbarer UI-Komponenten angesiedelt. Dazu gehören elementare Bausteine wie **badge/** für Badges, **button/** für Schaltflächen, **card/** für Karten und **checkbox/** für Checkboxen. Komplexe Interaktionselemente wie **dialog/** für Dialoge sind ebenfalls vorhanden. Für Benutzereingaben gibt es **input/** für allgemeine Eingabefelder, **label/** für Beschriftungen, **pin-input/** für PIN-Eingaben und **select/** für Auswahlfelder. Visuelle Indikatoren wie **spinner/** für Ladezustände sind ebenfalls enthalten, zusammen mit spezifischeren Komponenten wie **stats/** für Statistiken, **tasks/** für Aufgaben, **timer/** für Zeiterfassung und **weather/** für Wetteranzeigen.

4.2.3 Prinzip der Komponentenwiederverwendung am Beispiel

Das Prinzip der Komponentenwiederverwendung lässt sich hervorragend am Beispiel einer Schaltfläche veranschaulichen. Eine Schaltflächenkomponente ist typischerweise in einem eigenen Unterordner strukturiert, der eine Hauptkomponentendatei (Button.vue) und eine Konfigurations- und Exportdatei (index.ts) enthält. Die Konfiguration der Schaltfläche in index.ts nutzt class-variance-authority (cva) um verschiedene Varianten (z.B. default, destructive,

outline, secondary, ghost, link) und Größen (z.B. default, sm, lg, icon) zu definieren. Diese Konfiguration legt die Basisstile fest und ermöglicht eine flexible Anpassung. Die Implementierung des Hauptkomponenten in Button.vue nutzt diese Konfiguration, indem sie die definierten Varianten und Größen als Eigenschaften (props) akzeptiert.

4.2.4 Vorteile dieses Ansatzes

Dieser Ansatz bietet mehrere entscheidende Vorteile. Er gewährleistet einen einheitlichen Stil im gesamten Projekt, da alle Instanzen einer Komponente auf derselben Basiskonfiguration aufbauen. Änderungen an dieser Konfiguration werden automatisch auf alle betroffenen Komponenten angewendet, was die Wartung des Stils erheblich vereinfacht. Die Flexibilität ist ebenfalls ein großer Pluspunkt, da eine einzelne Komponente in verschiedenen Varianten und Größen verwendet und durch zusätzliche Eigenschaften erweitert werden kann. Die hohe Wiederverwendbarkeit ist offensichtlich: Eine einmal definierte und konfigurierte Schaltfläche kann einfach importiert und im gesamten Projekt mit spezifischen Varianten- und Größeneinstellungen eingesetzt werden.

4.2.5 Technische Besonderheiten

Technisch gesehen stützt sich dieser Ansatz stark auf die Verwendung von class-variance-authority (cva), das typensichere Stilvarianten und die automatische Generierung von Typen ermöglicht, was die Verwaltung von Varianten vereinfacht. Die Integration mit Tailwind CSS ist nahtlos, da vordefinierte Klassen für verschiedene Zustände verwendet werden, was zu konsistenten und leicht anpassbaren Stilen führt. Die vollständige Unterstützung von TypeScript durch Typisierung gewährleistet Autovervollständigung in IDEs und verhindert Fehler bereits zur Kompilierungszeit. Dies führt zu einer klaren Datenstruktur durch Klassen.

4.2.6 Praktische Anwendung

In der Praxis äußert sich dies in einer einfachen Erstellung neuer Schaltflächen, indem lediglich die gewünschte Variante und Größe als Eigenschaften übergeben werden. Bestehende Schaltflächen können durch zusätzliche CSS-Klassen leicht angepasst werden. Darüber hinaus können diese Komponenten mühelos in anderen Komponenten verwendet werden, was die Strukturierung von komplexen Oberflächenelementen vereinfacht.

4.2.7 Vorteile für die Entwicklung

Für die Entwicklung bringt dieser Ansatz erhebliche Vorteile mit sich. Er steigert die Produktivität, da Code-Duplizierung reduziert wird, die Kompilierung optimiert ist und der Cache effizient genutzt wird. Die Wartbarkeit wird verbessert durch die zentrale Stilverwaltung, die einfache Implementierung von Änderungen und die vereinfachte Testbarkeit. Schließlich fördert es die Skalierbarkeit, indem es das Hinzufügen neuer Varianten, die Erweiterung der Funktionalität und die effiziente Verwaltung von Abhängigkeiten erleichtert. Dieser umfassende Ansatz

zur Komponentenorganisation ermöglicht die Entwicklung einer skalierbaren und wartbaren Anwendung mit einem konsistenten Stil und einem hohen Grad an Code-Wiederverwendung.

4.3 Implementierung vom Backend

Die Backend-Implementierung des Projekts basiert auf NestJS, einem leistungsstarken Framework, das eine robuste Architektur für die API-Entwicklung bietet. Jeder Aspekt, von der API-Exposition über die Authentifizierung bis hin zur Datenbankinteraktion und Leistungsoptimierung, ist sorgfältig konzipiert, um Zuverlässigkeit, Sicherheit und Performance zu gewährleisten.

4.3.1 API-Endpoints und Business Logic

Das Herzstück der Backend-Funktionalität bilden die API-Endpunkte, die über Controller in jedem Modul der Anwendung realisiert werden. Diese Controller sind für die Verarbeitung von HTTP-Anfragen zuständig und nutzen spezifische Dekoratoren, um Routen und HTTP-Methoden zu definieren. Ein signifikanter Vorteil dieses Ansatzes ist die automatische Generierung der API-Dokumentation mittels Swagger, was die Kommunikation und Nutzung der API erheblich vereinfacht. Die Business Logic ist in separaten Diensten (Services) gekapselt. Diese Dienste agieren als zentrale Verarbeitungsstellen, nutzen Dependency Injection zur Verwaltung ihrer Abhängigkeiten und interagieren über ORM mit der Datenbank, wodurch eine klare Trennung der Verantwortlichkeiten gewährleistet wird.

4.3.2 JWT-Authentifizierung und Autorisierung

Für die Benutzerauthentifizierung und -autorisierung setzt das System auf JWT (JSON Web Token). Bei erfolgreicher Authentifizierung wird ein JWT generiert, dass eine Payload mit Benutzerdaten enthält und mit einem geheimen Schlüssel signiert ist. Dieses Token wird dann bei nachfolgenden Anfragen im Header mitgesendet. Spezielle Guards überprüfen die Präsenz und Gültigkeit des Tokens, extrahieren die Benutzerdaten und fügen sie dem Anfrageobjekt hinzu, wodurch geschützte Routen effektiv abgesichert werden. Da JWTs eine begrenzte Lebensdauer haben, ist ein Mechanismus zur Token-Aktualisierung mittels Refresh-Tokens implementiert, um eine nahtlose Benutzererfahrung zu gewährleisten.

4.3.3 Datenbankinteraktion und -verwaltung

Die Persistenzschicht des Backends verwendet TypeORM für die Interaktion mit einer PostgreSQL-Datenbank. Die Datenbankverbindung wird über eine detaillierte Konfiguration definiert, die typische Parameter wie Host, Port, Benutzername, Passwort und Datenbankname umfasst, wobei sensible Daten aus Umgebungsvariablen geladen werden. Die Datenbank selbst wird in einer isolierten Umgebung mittels Docker Compose betrieben, was eine einfache Verwaltung und Reproduzierbarkeit der Infrastruktur ermöglicht. Die Datenbankentitäten sind mittels TypeORM-Dekoratoren klar definiert, inklusive der Beziehungen zwischen den Tabellen

und der Validierung von Feldern. Diese deklarative Definition vereinfacht die Handhabung der Datenstruktur im Code erheblich.

4.3.4 Monitoring, Fehlerbehandlung und Sicherheit

Das System integriert ein mehrstufiges Monitoring. Dies beinhaltet Verfügbarkeitsprüfungen durch regelmäßige Ping-Anfragen an Endpunkte, die Überprüfung der HTTP- und HTTPS-Erreichbarkeit sowie des Status von SSL-Zertifikaten. Leistungskennzahlen wie die API-Antwortzeit werden erfasst. Ein zentralisiertes Log-System mit verschiedenen Log-Levels (Error, Warn, Info) und Request-Tracing unterstützt die Nachverfolgung und Analyse des Systemverhaltens.

Die Fehlerbehandlung erfolgt über einen globalen Exception-Handler, der Fehlermeldungen in einem standardisierten Format bereitstellt, was die Fehlersuche und -behebung vereinfacht. Für die Validierung von Eingabedaten kommen DTOs mit Custom-Validatoren und Datentransformationen zum Einsatz. Die Sicherheit wird durch Maßnahmen wie Rate Limiting für APIs, Schutz vor SQL-Injektionen und umfassende Eingangsdatenvalidierung gewährleistet.

4.3.5 Leistungsoptimierung

Zur Sicherstellung einer hohen Leistung und Skalierbarkeit werden verschiedene Optimierungsstrategien angewandt. Asynchrone Verarbeitung wird für ressourcenintensive Operationen genutzt, indem Aufgaben in Warteschlangen gelegt, als Hintergrundaufgaben ausgeführt oder parallel verarbeitet werden. Für die Skalierung ist das System auf horizontale Skalierung ausgelegt, was die Verteilung der Last auf mehrere Instanzen ermöglicht. Lastverteilung und Datenbankreplikation sind weitere Schlüsselkomponenten, die die Systemrobustheit und -leistung verbessern.

4.3.6 Modulare Architektur des Projekts

Das Projekt ist auf den Prinzipien einer modularen Architektur aufgebaut, die ein hohes Maß an Flexibilität und Skalierbarkeit gewährleistet. Jeder Modul ist ein unabhängiger Bestandteil mit klar definierten Grenzen und Verantwortlichkeiten. Die Anwendung ist in mehrere spezialisierte Module unterteilt, die jeweils einen bestimmten Funktionsbereich abdecken: Das AuthModule ist für alle Aspekte der Authentifizierung und Autorisierung zuständig, verwaltet den Login per Code und RFID, die Handhabung von JWT-Tokens und den Schutz von API-Endpunkten. Das DashboardModule ist dafür verantwortlich, die benötigten Daten für das Dashboard bereitzustellen, inklusive Statistiken zur Arbeitszeit, Informationen über Mitarbeiter und weitere analytische Daten. Das WorkTimeModule verwaltet die gesamte Funktionalität im Zusammenhang mit der Arbeitszeiterfassung, kümmert sich um die Zeiterfassung selbst, die Verwaltung von Arbeitssitzungen und die Erstellung von Berichten. Das MonitoringModule

schließlich ist für die Überwachung des Systems zuständig, überprüft die Verfügbarkeit von Diensten, erfasst Leistungsmetriken und ist für das Log-Management verantwortlich.

Diese modulare Architektur bietet mehrere entscheidende Vorteile für die Entwicklung und Wartung des Projekts. Isolation ist gegeben, da jedes Modul unabhängig funktioniert, sodass Änderungen innerhalb eines Moduls keine unbeabsichtigten Auswirkungen auf andere Module haben, was das Testen einzelner Komponenten erheblich vereinfacht. Die Wiederverwendung von Code wird gefördert, da gemeinsame Komponenten und Logiken in separate Module ausgelagert werden, was das Hinzufügen neuer Funktionen erleichtert. Die Skalierbarkeit des Systems ist hoch, da es sich durch einfaches Hinzufügen neuer Module oder durch horizontale Skalierung bestehender Module flexibel erweitern und ansteigende Anforderungen anpassen lässt. Schließlich verbessert die modulare Struktur die Wartbarkeit, da die klare und definierte Gliederung des Codes in einzelnen Modulen die Lesbarkeit und das Verständnis der Funktionalität erhöht, was die Fehlerbehebung und allgemeine Wartung erheblich vereinfacht. Dieser modulare Ansatz trägt maßgeblich zur Robustheit und Zukunftsfähigkeit der gesamten Anwendung bei.

4.4 Implementierung der JWT-Authentifizierung

Die JWT-Authentifizierung (JSON Web Token) im Projekt ist ein zentraler Bestandteil der Sicherheitsarchitektur und ermöglicht eine sichere Benutzeridentifikation sowie den Zugriff auf geschützte Ressourcen.

4.4.1 Generierung und Konfiguration des Tokens

Die Erstellung eines Tokens erfolgt, indem eine Payload mit den relevanten Benutzerdaten, wie der Benutzer-ID, dem Vornamen und dem Nachnamen, erzeugt wird. Dieser Payload wird anschließend mit einem geheimen Schlüssel signiert. Gemäß der Konfiguration, die in den Umgebungsvariablen env.config.ts definiert ist, hat jeder generierte Token eine Gültigkeitsdauer von 24 Stunden. Der geheime Schlüssel selbst wird sicher in den Umgebungsvariablen gespeichert.

4.4.2 Authentifizierungsmethoden

Das System bietet zwei Hauptmethoden zur Authentifizierung. Die erste ist die Authentifizierung per Code. Hierbei erwartet das System einen sechsstelligen numerischen Code. Dieser Code wird in zwei Teile aufgeteilt: Die ersten beiden Ziffern repräsentieren die Mitarbeiter-ID, während die verbleibenden Ziffern als PIN-Code dienen. Das System sucht dann in der Datenbank nach einem Mitarbeiter, der sowohl der angegebenen ID als auch dem PIN-Code entspricht. Im Falle einer Übereinstimmung wird ein Token generiert. Die zweite Methode ist die Authentifizierung per RFID. Hierbei wird ein RFID-Tag übermittelt. Das System sucht nach einem RFID-Tag-Eintrag in der Datenbank und ruft den damit verbundenen Mitarbeiter ab. Bei

erfolgreicher Zuordnung wird ebenfalls ein Authentifizierungstoken erstellt. In beiden Fällen führt eine nicht erfolgreiche Validierung zu einer UnauthorizedException.

4.4.3 Nutzung und Sicherheit des Tokens

Nach erfolgreicher Authentifizierung erhält der Client den generierten Token. Dieser Token wird für alle nachfolgenden Anfragen im HTTP-Header als Authorization: Bearer <token> mitgesendet. Spezielle Guards auf der Backend-Seite sind dafür verantwortlich, die Gültigkeit des Tokens zu überprüfen. Ist der Token gültig, werden die im Payload enthaltenen Benutzerdaten extrahiert und dem Anfrageobjekt hinzugefügt, wodurch sie für die nachfolgende Geschäftslogik verfügbar sind.

Die Sicherheit der Authentifizierung wird durch verschiedene Maßnahmen gewährleistet. Die begrenzte Lebensdauer der Tokens minimiert das Risiko bei Kompromittierung. Der geheime Schlüssel zur Signierung der Tokens wird sicher in Umgebungsvariablen gehalten. Eine umfassende Validierung aller eingehenden Daten verhindert Missbrauch. Darüber hinaus schützt die Implementierung von Guards unautorisierten Zugriff auf geschützte Endpunkte, indem nur Anfragen mit gültigen Tokens verarbeitet werden.

4.4.4 Beispielhafte Anwendung und Endpunktschutz

Im Controller wird die Authentifizierungslogik durch separate Endpunkte für die Code- und RFID-Authentifizierung gekapselt, die jeweils die entsprechenden Services aufrufen. Geschützte Endpunkte innerhalb der Anwendung sind mittels des @UseGuards(JwtAuthGuard)-Decorators gesichert. Dies stellt sicher, dass nur erfolgreich authentifizierte Benutzer Zugriff auf die dahinterliegende Funktionalität erhalten.

4.5 Tests

Zur Sicherstellung der Funktionalität und Stabilität der entwickelten Webanwendung wurden sowohl Integrationstests als auch Regressionstests durchgeführt. Diese Tests erfolgten nach Abschluss der Implementierungsphase und dienten dazu, die Interaktion der einzelnen Komponenten zu validieren sowie sicherzustellen, dass neue Änderungen keine bestehenden Funktionalitäten beeinträchtigen.

4.5.1 Testarten und Anwendung

Die Teststrategie gliedert sich in Unit-Tests, die einzelne Komponenten und Services isoliert prüfen, sowie in E2E-Tests, die den vollständigen Ablauf und die Integration der API-Endpunkte validieren. Besonders im Fokus stehen dabei die Authentifizierungsmechanismen, einschließlich der Prüfung von Code-basierten und RFID-basierten Anmeldungen mit verschiedenen Szenarien von gültigen und ungültigen Daten.

4.5.2 Testausführung und Konfiguration

Die Tests werden über definierte Skripte in der package.json gesteuert, z. B. npm run test für alle Tests, npm run test:e2e speziell für E2E-Tests, sowie npm run test:watch für den Watch-Modus. Eine zentrale Jest-Konfigurationsdatei regelt die Umgebungsparameter und das Testmuster.

5 Fazit

5.1 Soll-/Ist-Vergleich

Die Entwicklung der Webanwendung wurde innerhalb des vorgesehenen Zeitrahmens abgeschlossen. Für die Implementierung zusätzlicher Funktionen musste jedoch etwas mehr Zeit aufgewendet werden als ursprünglich geplant. Dies wurde jedoch durch eine schnellere Durchführung der Tests kompensiert, die schneller als erwartet abgeschlossen werden konnten. Insgesamt konnte der geplante Gesamtzeitaufwand eingehalten werden.

5.2 Gelerntes im Projektverlauf

Im Verlauf des Projekts konnten die Kenntnisse in moderner Webentwicklung deutlich erweitert werden. Besonders der Einsatz von Vue 3 mit der Composition API hat zur Entwicklung modularer und wartbarer Komponenten beigetragen. Auch der Umgang mit Tailwind CSS ermöglichte eine zügige und flexible Gestaltung der Benutzeroberfläche.

Auf der Serverseite wurde das Verständnis für NestJS vertieft, was die Umsetzung einer skalierbaren und klar strukturierten Architektur erleichterte. Darüber hinaus wurden Fortschritte im Bereich Testing gemacht – insbesondere durch den gezielten Einsatz von Jest zur Durchführung von Unit- und End-to-End-Tests, was zur Verbesserung der Codequalität beitrug.

Insgesamt spiegelte sich der sichere Umgang mit modernen Technologien in einer erfolgreichen praktischen Umsetzung wider.

5.3 Ausblick

Die bestehende modulare Architektur der Webanwendung ermöglicht eine effiziente Aktualisierung und Erweiterung sowohl des Frontends als auch des Backends. Diese Flexibilität bildet die Grundlage für geplante Weiterentwicklungen, die darauf abzielen, die Funktionalität und den Nutzen der Anwendung signifikant zu steigern.

Für die Zukunft ist die Integration eines Grafana-Dashboards vorgesehen. Dies wird die Visualisierung und Auswertung der gesammelten Daten erheblich verbessern, indem es Benutzern ermöglicht, Leistungsmetriken und Systemstatus auf intuitive Weise zu überwachen.

Des Weiteren ist die Implementierung eines komplexeren Zeiterfassungssystems geplant. Dieses System soll eine präzisere Erfassung und detailliertere Auswertung der Arbeitszeiten der Benutzer ermöglichen.

Diese geplanten Erweiterungen werden maßgeblich dazu beitragen, die Webanwendung noch praxisnäher, leistungsfähiger und benutzerfreundlicher zu gestalten.

4 Glossar

Backend - Serverseitige Anwendung

Frontend - Benutzeroberfläche im Browser

NestJS - Node.js Framework für Backend in TypeScript

PostgreSQL - Relationale Datenbank zur Datenspeicherung

Raspberry Pi - Kleiner Einplatinencomputer als Terminalgerät

RFID - Funkbasierte Authentifizierung per Karte

Grafana - Tool zur Visualisierung von Systemmetriken

Prometheus - System zur Sammlung von Monitoring-Daten

iframe - HTML-Element zum Einbinden externer Inhalte

Tailwind CSS - CSS-Framework für schnelle UI-Gestaltung

Deployment - Softwarebereitstellung auf den Terminalgeräten

Vue 3 - JavaScript-Framework zur Entwicklung der Benutzeroberfläche

Framer Motion / Animate.css - Bibliotheken für Animationen

Vite - Schneller Build-Tool für Frontend-Entwicklung

REST-API - Schnittstelle zur Kommunikation zwischen Client und Server

Axios - HTTP-Client zur API-Kommunikation

Chart.js - Visualisierung von Daten in Diagrammen

TypeORM - ORM zur Datenbankanbindung

JWT / Passport.js - Authentifizierungsmethoden zur Sicherung der Anwendung

Swagger - Generiert automatische API-Dokumentation

ESLint / Prettier - Werkzeuge zur Codequalität und Formatierung

Jest - Test-Framework für JavaScript/TypeScript

Docker - Containerisierung für eine einheitliche Laufzeitumgebung

Git - Versionsverwaltung des Quellcodes

Kanban - Aufgabenverwaltungsmethode, im Projekt als Ansicht umgesetzt

GPIO - Hardware-Schnittstelle am Raspberry Pi für z. B. RFID-Leser

Pinia - Zustandsverwaltung in Vue 3, Alternative zu Vuex

Vue-Hooks - Funktionen zur Wiederverwendung reaktiver Logik in Vue

DashboardLayout.vue - Zentrale Layout-Komponente für Navigation und Struktur

Vue Router - Routing-Lösung für Navigation zwischen Seiten in Vue

Lucide Icons - Icon-Bibliothek für moderne Webanwendungen

DTO (Data Transfer Object) - Struktur zur Definition des Formats von API-Anfragen und -Antworten

ValidationPipe - Middleware zur automatischen Validierung von Eingabedaten in NestJS

JWT (JSON Web Token) - Methode zur Authentifizierung durch digitale Token

Guards - Sicherheitsmechanismen in NestJS zur Zugriffskontrolle

Decorators - TypeScript-Feature zur Annotation von Klassen, Methoden und Eigenschaften

TypeScript - JavaScript mit Typisierung für bessere Wartbarkeit

Swagger / Postman - Tools zur API-Testung und Interaktion

5 Quellenverzeichnis

Ausbildungsbetrieb

https://godigital24.de/

Verwendeten Technologien

https://de.vuejs.org/

https://tailwindcss.com/docs

https://vitejs.dev/guide/

https://docs.nestjs.com/

https://www.postgresql.org/docs/de/

https://swagger.io/docs/

https://docs.docker.com/de/

https://typeorm.io/

Komponente für Terminalgerät

raspberry_pi_4_b_4gb

Display mit Gehäuse

RFID Reader

6 Anhang

Tatsächlicher Detaillierter Zeitplan

Projektphase	Aufgaben	Geplante Stunden
Zielsetzung / Themenbeschreibung	Analyse der Ausgangslage, Definition des Projektziels, Abgrenzung	6
Entwurf / Sollkonzept	Erstellung von Mockups (UI-Design der Benutzeroberfläche)	3
	Entwurf von Datenbankmodellen (ER-Diagramm, Tabellenstruktur)	3
	Erstellung von Systemarchitektur-Diagrammen (Backend, Frontend, API-Flows)	2
	Planung des Monitorings mit Grafana und Prometheus	2
Implementierung	Erstellung der Datenbank (Tabellen, Beziehungen, erste Seed-Daten)	6
	Entwicklung des Backends (NestJS: API, Authentifizierung, Benachrichtigungen)	15
	Entwicklung des Frontends (Vue 3, Tailwind CSS: Benutzeroberfläche, Formular-Design)	14
	Integration der Datenbank mit Backend und Frontend	4
	Einrichtung des Monitorings mit Prometheus und Grafana	4
	Einrichtung von Benachrichtigungssystemen (Display, Telefon)	4
Qualitätskontrolle	Testen der Backend-Logik und API-Endpoints	2
	Testen der Frontend-Funktionalität und Usability	1
	Integrationstests (Verbindung Backend-Frontend-Datenbank)	1
	Sicherheits- und Lasttests	1
Dokumentation	Erstellen einer kurzen Projektdokumentation und Zusammenfassung	12
Gesamt		**80**

Tabelle 2: Tatsächlicher Detaillierter Zeitplan

Ursprünglicher Detaillierter Zeitplan

Projektphase	Aufgaben	Geplante Stunden
Zielsetzung / Themenbeschreibung	Analyse der Ausgangslage, Definition des Projektziels, Abgrenzung	6
Entwurf / Sollkonzept	Erstellung von Mockups (UI-Design der Benutzeroberfläche)	3
	Entwurf von Datenbankmodellen (ER-Diagramm, Tabellenstruktur)	3
	Erstellung von Systemarchitektur-Diagrammen (Backend, Frontend, API-Flows)	2
	Planung des Monitorings mit Grafana und Prometheus	2
Implementierung	Erstellung der Datenbank (Tabellen, Beziehungen, erste Seed-Daten)	6
	Entwicklung des Backends (NestJS: API, Authentifizierung, Benachrichtigungen)	15
	Entwicklung des Frontends (Vue 3, Tailwind CSS: Benutzeroberfläche, Formular-Design)	12
	Integration der Datenbank mit Backend und Frontend	4
	Einrichtung des Monitorings mit Prometheus und Grafana	4
	Einrichtung von Benachrichtigungssystemen (Display, Telefon)	4
Qualitätskontrolle	Testen der Backend-Logik und API-Endpoints	3
	Testen der Frontend-Funktionalität und Usability	2
	Integrationstests (Verbindung Backend-Frontend-Datenbank)	1
	Sicherheits- und Lasttests	1
Dokumentation	Erstellen einer kurzen Projektdokumentation und Zusammenfassung	12
Gesamt		**80**

Tabelle 3: Ursprünglicher Detaillierter Zeitplan

Ein relationale Datenmodell befindet sich in Anhang

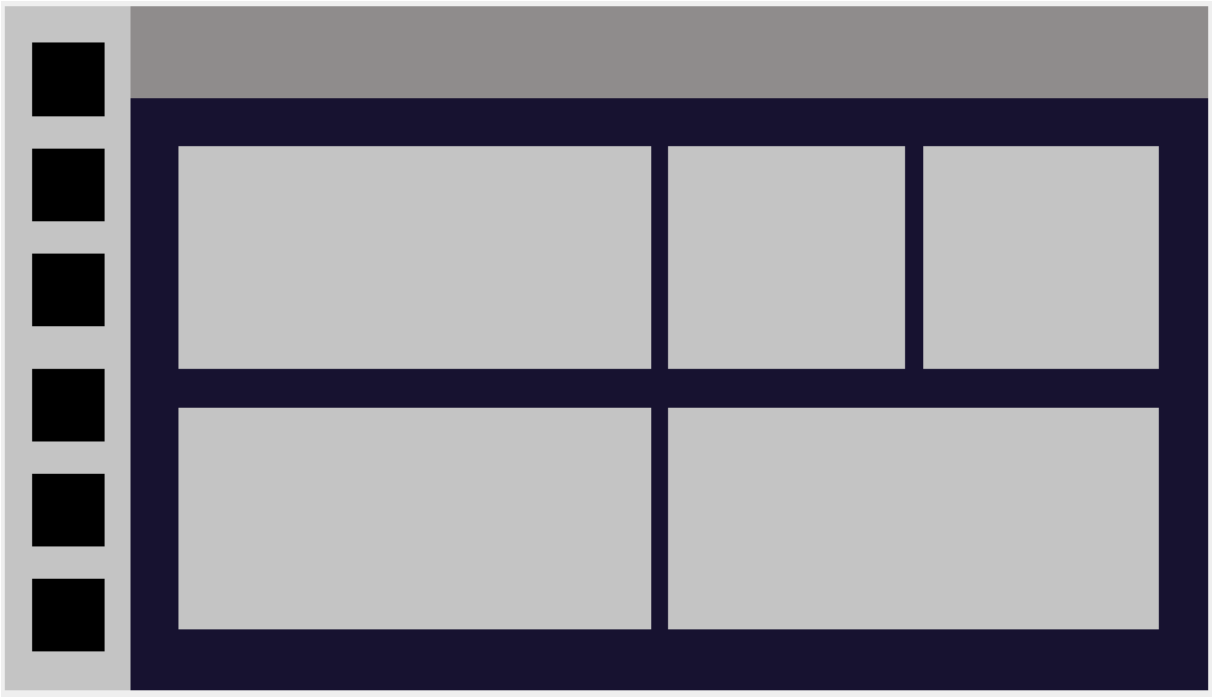

Abbildung 1:Dashboard Entwurf

Kanban Entwurf

Abbildung 2: Kanban Entwurf

Relationales Datenbankmodell

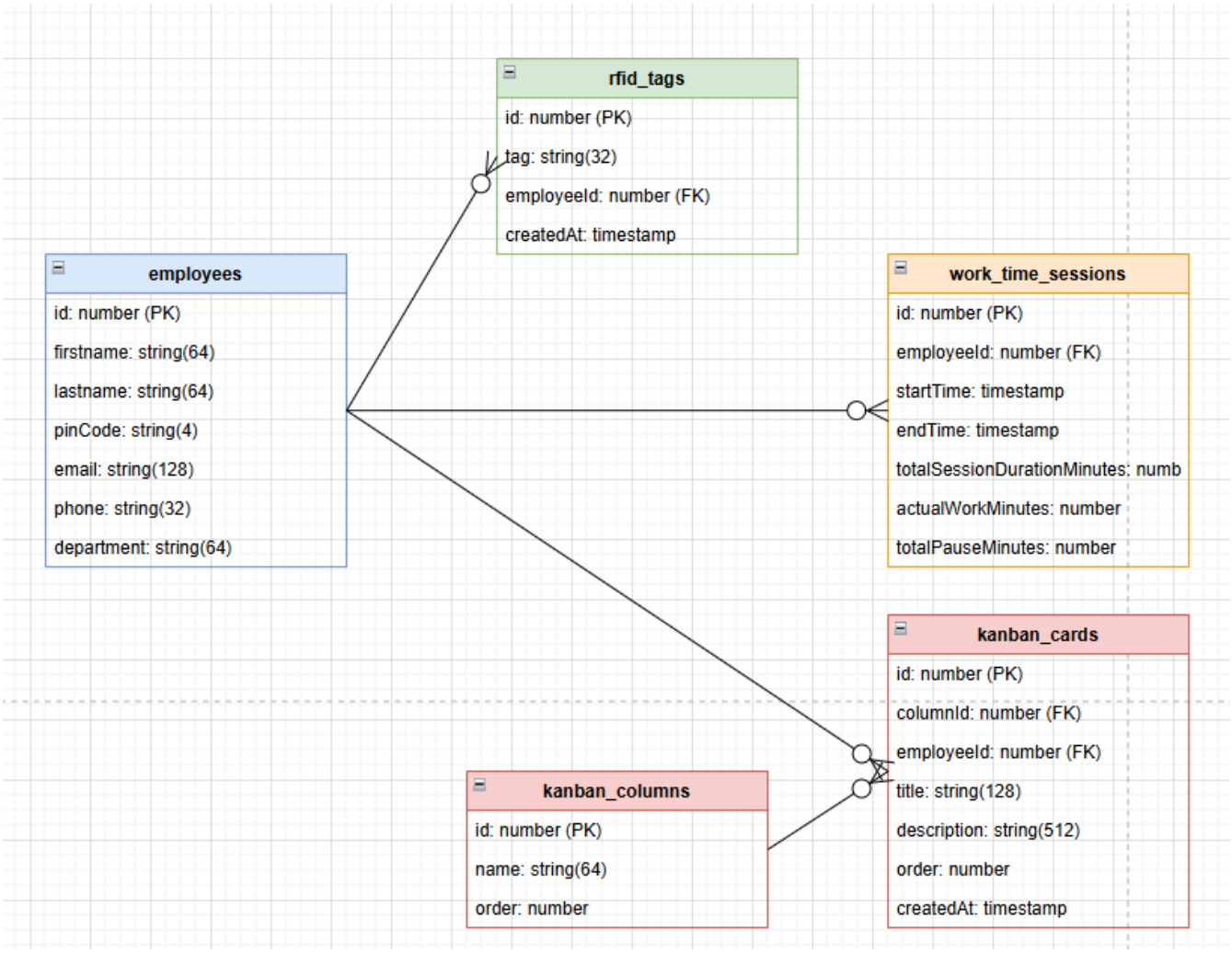

Abbildung 3: Relationales Datenbankmodell

Use Case Diagramm

Abbildung 4: Use Case Diagramm

Swagger

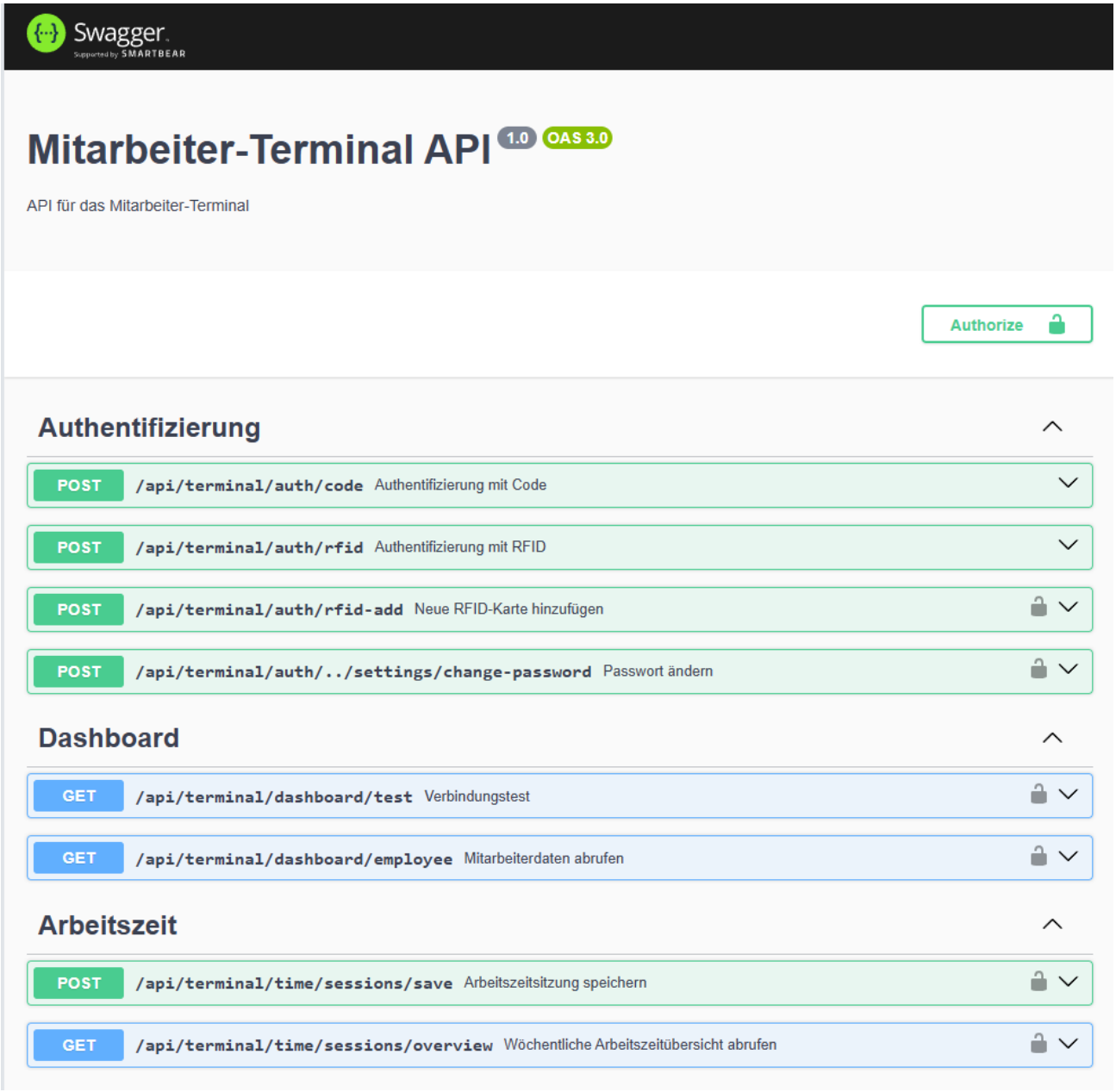

Abbildung 5: Swagger Interface

StandbyView

Abbildung 6: Standby View

Pin-eingabe Dialog

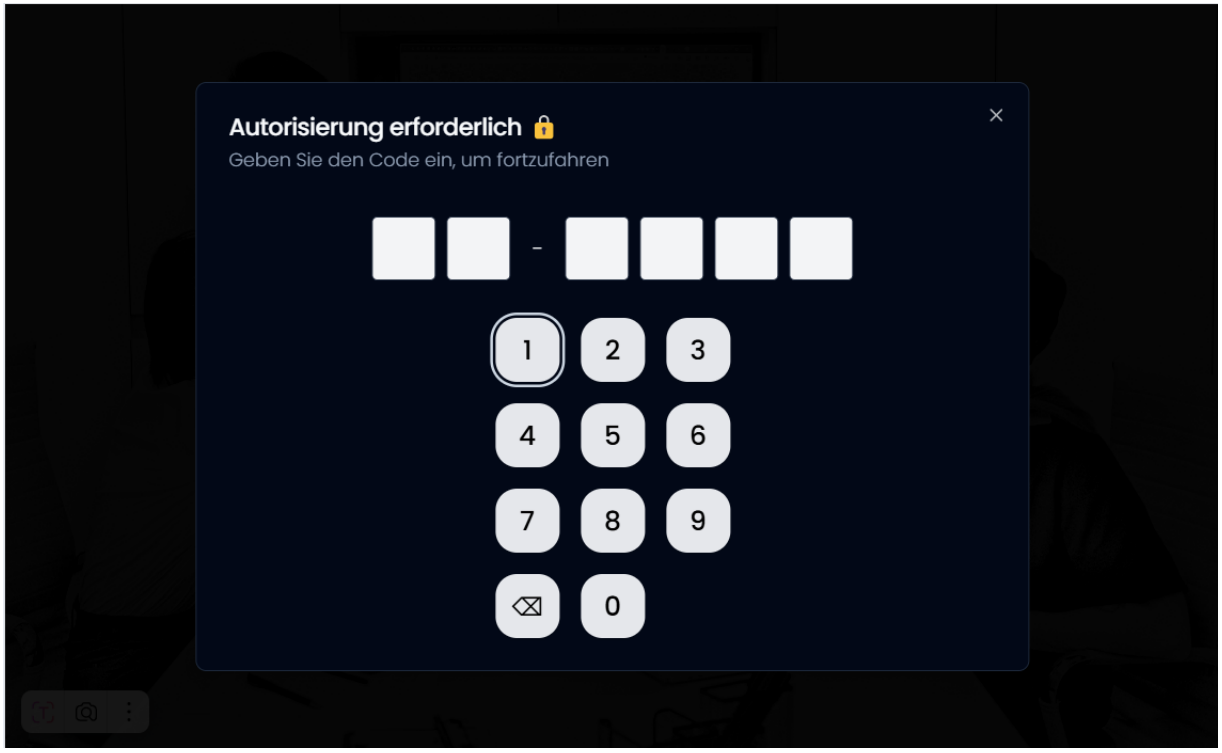

Abbildung 7: Pin-Eingabe Dialog

Begrüßung Dialog

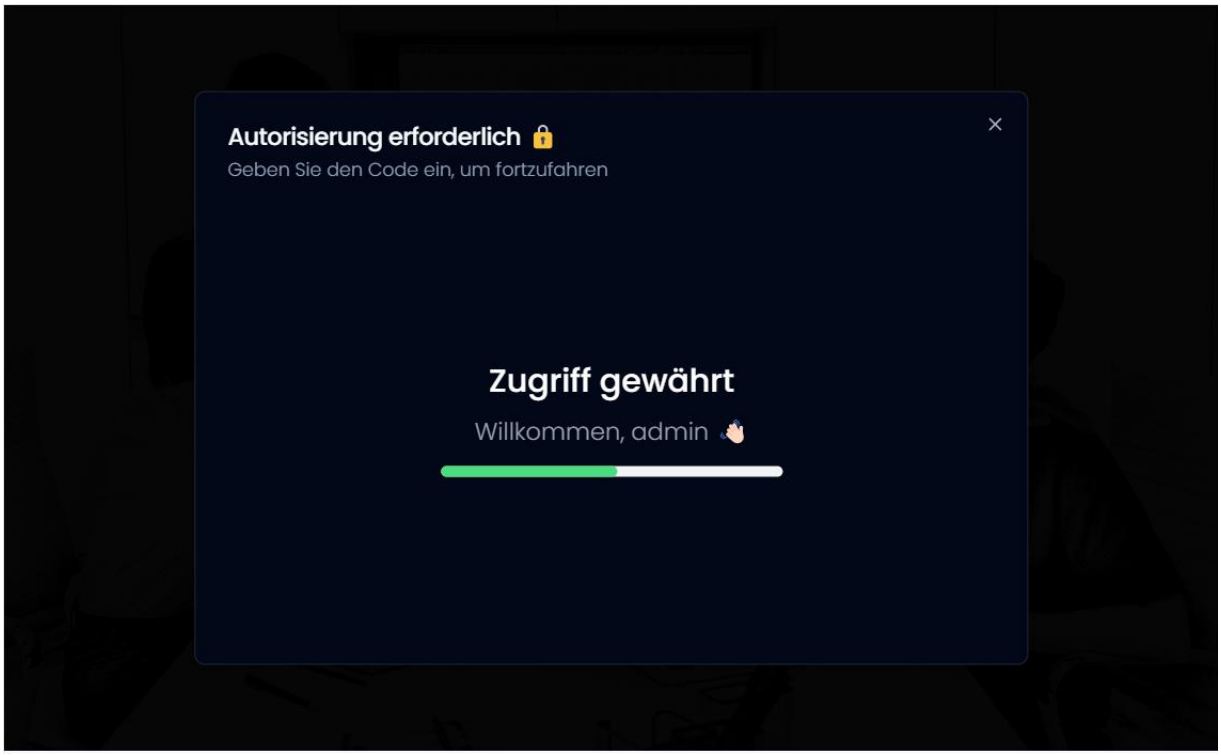

Abbildung 8: Begrüßung Dialog

DashboardView

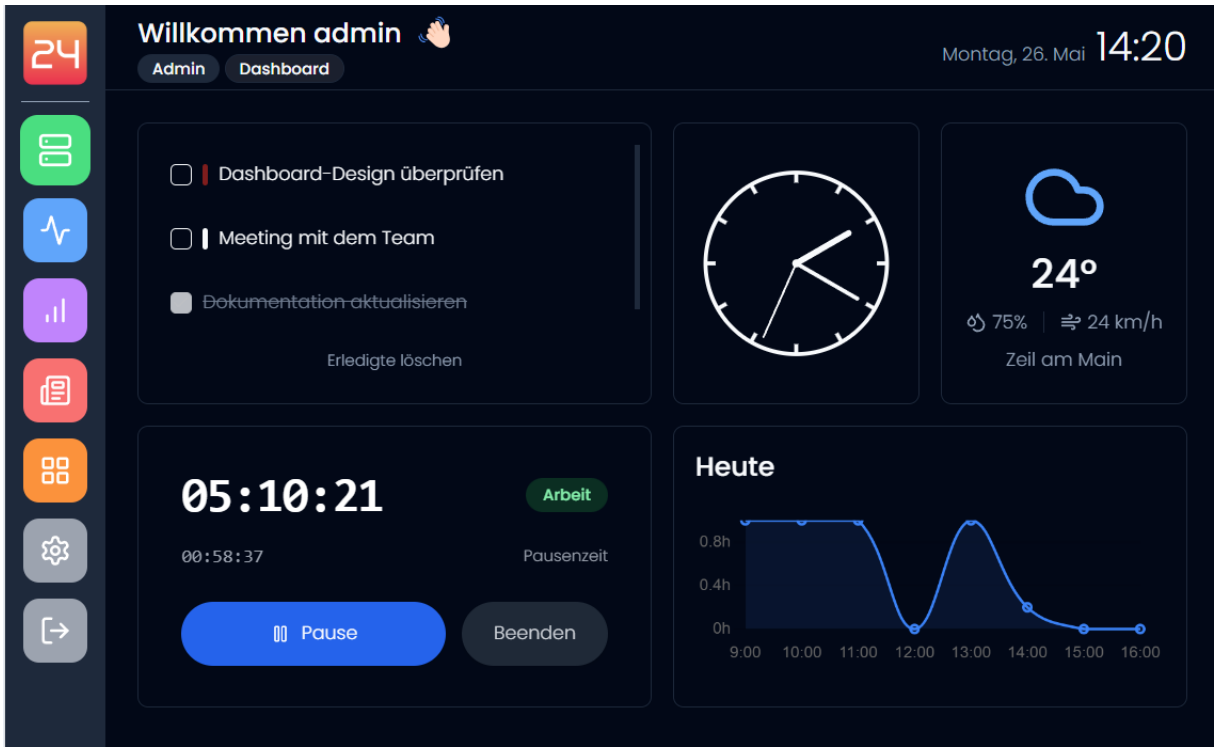

Abbildung 9: Dashboard View

MonitoringView

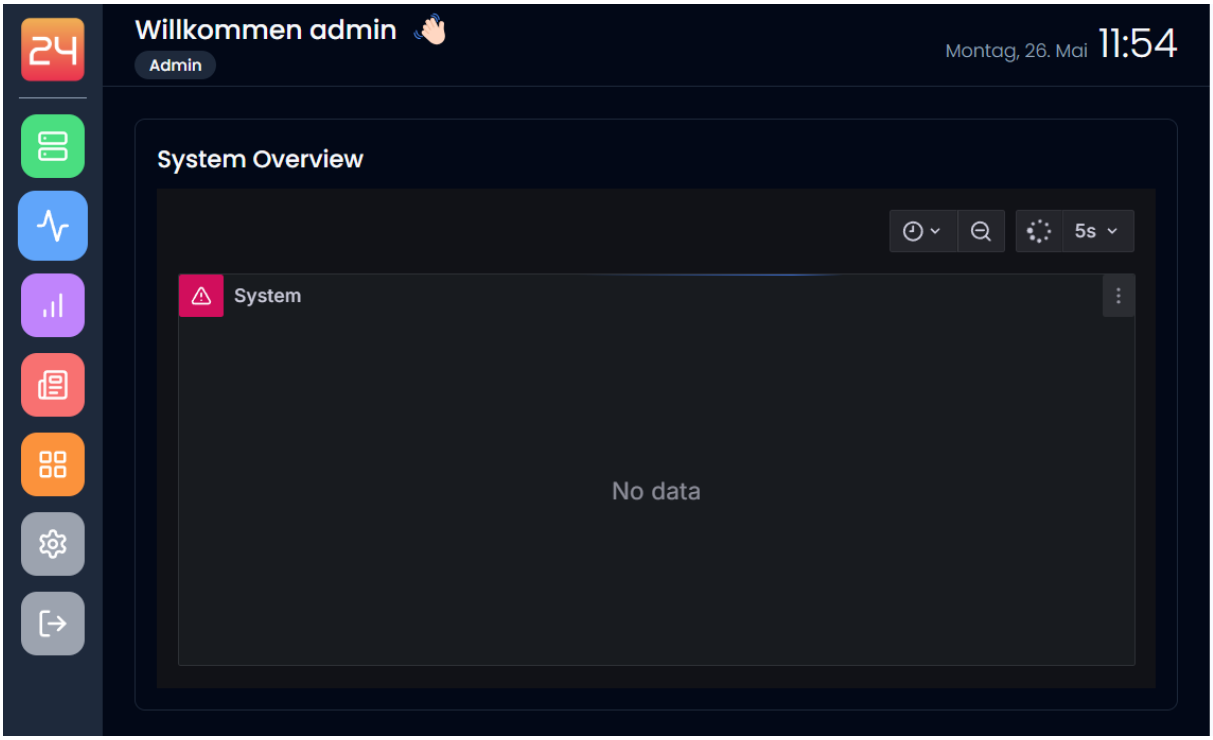

Abbildung 10: Monitoring View

StatistikenView

Abbildung 11: Statistiken View

KanbanView

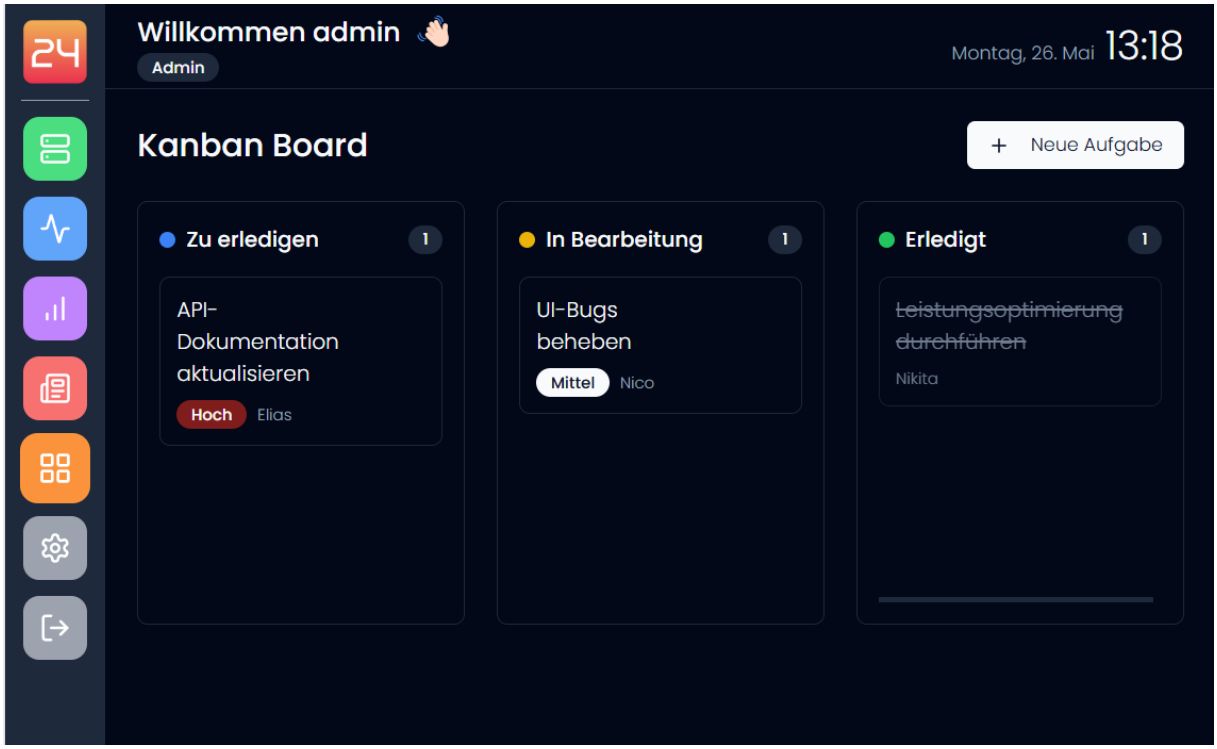

Abbildung 12: Kanban View

SettingsView

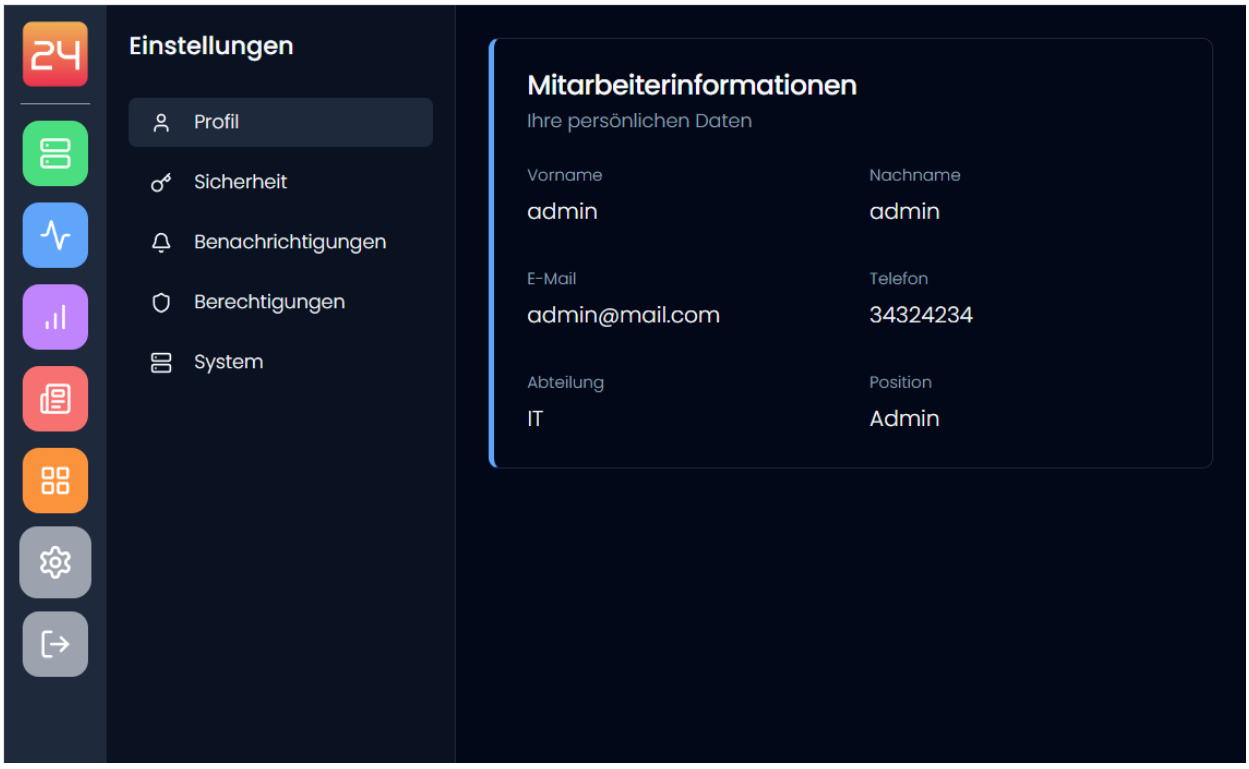

Abbildung 13: Settings View

NotfallAufgabe Dialog

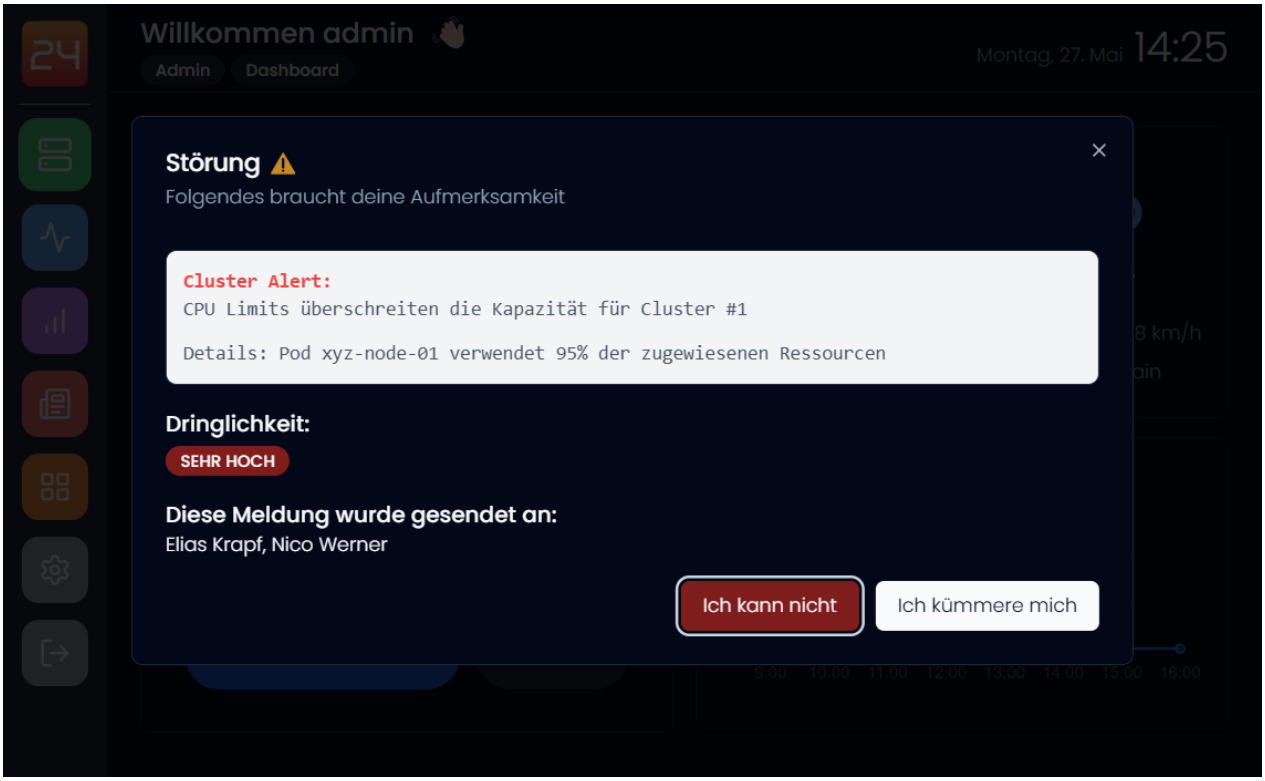

Abbildung 14: Notfallaufgabe Dialog

Code Button.vue

src > components > ui > button > Button.vue > ...

```vue
<script setup lang="ts">
import type { HTMLAttributes } from 'vue'
import { cn } from '@/lib/utils'
import { Primitive, type PrimitiveProps } from 'radix-vue'
import { type ButtonVariants, buttonVariants } from '.'

interface Props extends PrimitiveProps {
  variant?: ButtonVariants['variant']
  size?: ButtonVariants['size']
  class?: HTMLAttributes['class']
}

const props = withDefaults(defineProps<Props>(), {
  as: 'button',
})
</script>

<template>
  <Primitive
    :as="as"
    :as-child="asChild"
    :class="cn(buttonVariants({ variant, size }), props.class)"
  >
    <slot />
  </Primitive>
</template>
```

Abbildung 15: Code-Button.vue

Code button/index.ts

src > components > ui > button > TS index.ts > ...

```ts
import { cva, type VariantProps } from 'class-variance-authority'

export { default as Button } from './Button.vue'

export const buttonVariants = cva(
  [
    'inline-flex items-center justify-center gap-2',
    'whitespace-nowrap rounded-md text-sm font-medium',
    'ring-offset-background transition-colors',
    'focus-visible:outline-none focus-visible:ring-2 focus-visible:ring-ring',
    'focus-visible:ring-offset-2',
    'disabled:pointer-events-none disabled:opacity-50',
    '[&_svg]:pointer-events-none [&_svg]:size-4 [&_svg]:shrink-0'
  ].join(' '),
  {
    variants: {
      variant: {
        default: 'bg-primary text-primary-foreground shadow hover:bg-primary/90',
        destructive: 'bg-destructive text-destructive-foreground shadow-sm hover:bg-destructive/90',
        outline: 'border border-input bg-background shadow-sm hover:bg-accent hover:text-accent-foreground',
        secondary: 'bg-secondary text-secondary-foreground shadow-sm hover:bg-secondary/80',
        ghost: 'hover:bg-accent hover:text-accent-foreground',
        link: 'text-primary underline-offset-4 hover:underline',
      },
      size: {
        default: 'h-9 px-4 py-2',
        sm: 'h-8 rounded-md px-3 text-xs',
        lg: 'h-10 rounded-md px-8',
        icon: 'h-9 w-9',
      },
    },
    defaultVariants: {
      variant: 'default',
      size: 'default',
    },
  },
)

export type ButtonVariants = VariantProps<typeof buttonVariants>
```

Abbildung 16: Code-button/index.ts

Code Button erstellen

```vue
<Button variant="destructive" @click="open = false">Ich kann nicht</Button>
```

Abbildung 17: Codeausschnitt Button erstellen

Codeausschnitt 1 Router

```
path: '/',
component: () => import('../layout/dashboard/DashboardLayout.vue'),
meta: {
    requireAuth: true,
},
children: [
    {
        path: '/dashboard',
        name: 'home',
        component: () => import("../views/DashboardView.vue"),
    },
```

Abbildung 18: Codeausschnitt 1 - router

Codeausschnitt 2 Router

```
router.beforeEach((to, from, next) => {
    const requiresAuth = to.matched.some(record => record.meta.requireAuth);
    const token = localStorage.getItem('g24-terminal-jwt');
    console.log(from);
    if (requiresAuth && !token) {
        next({name: 'standby'});
    } else {
        next();
    }
});
```

Abbildung 19: Codeausschnitt 2 - router

Code .env

```
.env
1   NODE_ENV=development
2   PORT=3000
3   JWT_SECRET=dfjsfns5499fbBF03BDHBFDJbfh87dsjk675HF342dfj
4   # Database
5   DB_HOST=localhost
6   DB_PORT=5432
7   DB_USERNAME=postgres
8   DB_PASSWORD=postgres
9   DB_DATABASE=employee_terminal
10
```

Abbildung 20: Code .env

Code env.config.ts

src > config > TS env.config.ts > ...

```typescript
export interface EnvironmentVariables {
  NODE_ENV: string;
  PORT: number;
  DB_HOST: string;
  DB_PORT: number;
  DB_USERNAME: string;
  DB_PASSWORD: string;
  DB_DATABASE: string;
  JWT_SECRET: string;
}

export const envConfig = () => ({
  port: parseInt(process.env.PORT || '3000'),
  database: {
    type: 'postgres' as const,
    host: process.env.DB_HOST,
    port: parseInt(process.env.DB_PORT || '5432'),
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE,
    synchronize: false,
    autoLoadEntities: true,
  },
  jwt: {
    secret: process.env.JWT_SECRET,
    expiresIn: '24h',
  },
});
```

Abbildung 21: Code env.config.ts