Hochschule Bochum
Fachbereich Elektrotechnik und Informatik

Dokumentation Rustprojekt

„Webproject Arbeitszeitverwaltung “

Prüfer: Prof. Dr.-Ing. Edmund Coersmeier

Abdelrahman Shahin
Matrikelnummer: 18310070

Studiengang: Informatik
6. Fachsemester, Sommersemester 2023

Inhaltsverzeichnis

1 Einleitung **2**
1.1 Ziele der Dokumentation . . . 2
1.2 Einführung in Rust und Webanwendungen . . . 2

2 Installation **2**
2.1 Rust-Installation . . . 2
2.2 Installation von C++ Build Tools . . . 2
2.3 Einrichtung einer Entwicklungsumgebung . . . 2

3 Datenbank **2**
3.1 Installation von PostgreSQL . . . 2
3.2 Einrichtung eines Datenbankservers . . . 3
3.3 Tabellenstruktur und Schema-Definitionen . . . 3

4 Backend **3**
4.1 Projekt starten . . . 3
4.2 Abhängigkeiten . . . 3
4.3 API . . . 4
4.3.1 main.rs . . . 4
4.3.2 services.rs . . . 5

5 Frontend **10**

6 Deployment **10**
6.1 Docker . . . 10
6.1.1 Installation auf dem Linuxserver . . . 10

7 Zusammenfassung und Ausblick **10**
7.1 Zusammenfassung der wichtigsten Punkte . . . 10
7.2 Ausblick auf zukünftige Entwicklungen . . . 10

8 Anhang **10**
8.1 Glossar . . . 10
8.2 Beispielcode . . . 10
8.3 Referenzen und weiterführende Literatur . . . 10

1 Einleitung

1.1 Ziele der Dokumentation

1.2 Einführung in Rust und Webanwendungen

2 Installation

2.1 Rust-Installation

Um Rust auf einem Windows-System zu installieren, müssen die folgenden Schritte ausgeführt werden:

1. Unter der offiziellen Rust-Website `https://www.rust-lang.org/learn/get-started` müssen den Anweisungen für Windows befolgt werden.
2. Mit den Standardoptionen kann man das Packet installieren.
3. Um sicherzustellen, dass Rust erfolgreich installiert wurde, kann man im Terminal den Befehl `rustc -version` ausführen. Die installierte Rust-Version soll angezeigt werden.

2.2 Installation von C++ Build Tools

Um Rust auf Windows zu installieren, muss man zunächst C++ Build Tools installieren:

1. Zur offiziellen Microsoft-Website für Visual C++ Build Tools unter `https://visualstudio.microsoft.com/de/visual-cpp-build-tools/` gehen.
2. Auf den Download-Button klicken, um das Installationsprogramm herunterzuladen.
3. Das Installationsprogramm starten und die Option "Desktop development with C++äuswählen.
4. Anweisungen im Installationsprogramm befolgen und die Standardoptionen akzeptieren.

2.3 Einrichtung einer Entwicklungsumgebung

Nach der Installation von Rust und den erforderlichen Abhängigkeiten wird eine Entwicklungsumgebung eingerichtet:

1. Als Entwicklungsumgebung (IDE)wird in diesem Projekt mit Visual Studio Code gearbeitet.
2. Das Plugin Rust Analyzer von The Rust Programming Language installieren.
3. Ein neues Rust-Projekt mit dem Befehl `cargo new RustProjektWebanwendung` erstellen.

3 Datenbank

In diesem Abschnitt werden die Schritte zur Installation von PostgreSQL und die Einrichtung eines Datenbankservers auf Ihrem Computer beschrieben. Außerdem werden die Tabellenstruktur und die Schema-Definitionen für die Datenbank vorgestellt.

3.1 Installation von PostgreSQL

Um PostgreSQL auf einem Windows-System zu installieren, müssen die folgenden Schritte ausgeführt werden:

1. Unter der offiziellen PostgreSQL-Website `https://www.postgresql.org/download/windows/` müssen den Anweisungen für Windows befolgt werden.
2. Installationsprogramm starten und den Anweisungen auf dem Bildschirm folgen, um die Installation abzuschließen.
3. Die Option pgAdmin4 auswählen, um pdAdmin4 für die Verwaltung der Datenbank zu installieren.

3.2 Einrichtung eines Datenbankservers

Nachdem PostgreSQL erfolgreich installiert wurde, kann ein Datenbankserver auf eingerichtet werden. Hierfür wurde pgAdmin4 benutzt:

1. In der linken Navigationsleiste wird auf „Servers/PostgreSQL/Databases“ geklickt.
2. Durch einen Rechtsklick auf „Databases“ wird eine neue Datenbank erstellt oder im Terminal wird der Befehl „CREATE DATABASE <datenbankname>;“ ausgeführt. Standardmäßig wird eine Datenbank namens "postgresërstellt.
3. Standardmäßig läuft der Server auf Localhost auf Port 5432.

3.3 Tabellenstruktur und Schema-Definitionen

In unserer Beispielanwendung werden zwei Tabellen verwendet: die Tabelle üsersünd die Tabelle "working_days". Die Struktur und Definitionen dieser Tabellen sind wie folgt:

```
-- Tabelle "users"
CREATE TABLE users (
id serial primary key,
first_name varchar not null,
last_name varchar not null
);

-- Tabelle "working_days"
CREATE TABLE working_days (
id SERIAL PRIMARY KEY,
starting_time varchar ,
ending_time varchar ,
working_hours varchar ,
user_id INTEGER REFERENCES "users"(id) ON DELETE CASCADE
);
```

Die Tabelle „users“ enthält drei Spalten: „id“, „first_name“ und „last_name“. Die „id“-Spalte ist der Primärschlüssel der Tabelle. Die Tabelle „working_days“ enthält fünf Spalten: „id“, „starting_time“, „ending_time“, „working_hours“ und „user_id“. Die „id“-Spalte ist auch hier der Primärschlüssel. Die Spalte „user_id“ ist ein Fremdschlüssel, der auf die „id“-Spalte der Tabelle „users“ verweist und das Löschen von Datensätzen in der Tabelle „users“ automatisch das Löschen der entsprechenden Datensätze in der Tabelle „working_days“ auslöst. Die Datentypen von „starting_time“, „ending_time“ und „working_hours“ sind auf VARCHAR gesetzt. Das Handling der Arbeitszeitrechnung wird im Backend in Rust durchgeführt.
Mit diesen Schritten und Definitionen sollte die PostgreSQL-Datenbank einsatzbereit sein.

4 Backend

4.1 Projekt starten

Um ein Rustprojekt zu starten wird der Befehl `cargo new working_time_webproject` ausgeführt. So wird ein neuer Projekt erstellt und man kann in den neu erstellten Ordner mit `cd working_time_webproject` `navigieren` Standardmäßig hat man im Projektverzeichnis den src Ordner sowie die Cargo.toml Datei.

4.2 Abhängigkeiten

Abhängigkeiten werden in der Cargo.toml Datei. Die sieht wie folgt aus:

```
[package]
name = "working_time_webproject"
version = "0.1.0"
edition = "2021"
```

```
# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
actix = "0.13.0"
actix-web = "4.2.1"
dotenv = "0.15.0"
serde = { version = "1.0.145", features = ["derive"] }
serde_json = "1.0.86"
sqlx = { version = "0.6.2", features = ["runtime-async-std-native-tls", "postgres"] }
chrono = "0.4"
```

Im ersten Abschnitt der Datei findet sich zunächst die Paketdefinition, die standardmäßig bei der Erstellung des Projekts gesetzt wird.
Im Abschnitt dependencies werden die Abhängigkeiten aufgeführt, die für das Projekt verwendet werden. In diesem Projekt werden folgende Abhängigkeiten genutzt:

1. actix und actix-web werden verwendet, um den Actix-Webserver zu nutzen.
2. dotenv wird eingesetzt, um auf die Datei ".envßuzugreifen, in der Umgebungsvariablen definiert werden können. In diesem Projekt wird die Umgebungsvariable DATABASE_URL verwendet, um eine Verbindung zur PostgreSQL-Datenbank herzustellen.
3. serde und serde_json werden für die Serialisierung und Deserialisierung von Objekten genutzt, um reibungslose Datenbankoperationen ohne Datentypenkonflikte zu gewährleisten.
4. sqlx wird hier verwendet, um eine Verbindung zu einer PostgreSQL-Datenbank herzustellen und mit ihr zu interagieren.
5. chrono wird verwendet, um den Datentyp NaiveDateTime bereitzustellen, der zur Berechnung der Arbeitszeit in der Anwendung anhand von Anfangs- und Endzeitpunkten genutzt wird.

4.3 API

Nachdem die Abhängigkeiten definiert wurden, kann die API implementiert werden. Die API definiert die Schnittstelle, über die die Anwendung mit dem Benutzer interagiert.
In der Datei `src/main.rs` wird die Hauptfunktion des Projekts definiert. In dieser Funktion wird eine Instanz des Actix-Webservers gestartet und die Routen der API definiert. Die Datei `src/services.rs` enthält Funktionen, die als HTTP-Endpunkte (API) fungieren, die von Benutzern aufgerufen werden können, um Operationen (Create, Read, Update, Delete) auf einer Datenbank durchzuführen.

4.3.1 main.rs

Importieren von Paketen :

```rust
use actix_web::{web::Data, App, HttpServer};
use dotenv::dotenv;
use sqlx::{postgres::PgPoolOptions, Pool, Postgres};

mod services;
use services::{fetch_users, fetch_user_workingdays, create_user, create_user_workingday,
 ↪  update_user, update_user_workingday, delete_user, delete_user_workingday};
```

Hier wird das actix_web-Paket, das dotenv-Paket sowie das sqlx-packet importiert und ein Modul namens services definiert, in dem Funktionen für HTTP-Endpunkte definiert werden. Die Funktionen werden dann mit use deklariert, damit sie in der Main-Funktion verwendet werden können.

Datenbankverbindungspool :

```rust
pub struct AppState {
    db: Pool<Postgres>
}
```

Die Struktur AppState stellt den globalen Zustand der Anwendung dar und enthält eine Datenbankverbindungspool, welche Informationen zum Herstellen einer Verbindung zur PostgreSQL-Datenbank enthält. Diese Informationen werden in der .env-Datei gespeichert, die von der Bibliothek dotenv geladen wird. Der Datenbankverbindungspool wird im Hauptprogramm erstellt und dann in die AppState-Struktur eingebettet, damit er im gesamten Programmzugriff zur Verfügung steht.

Main Funktion :

```rust
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    dotenv().ok();
    let database_url = std::env::var("DATABASE_URL").expect("DATABASE_URL must be set");
    let pool = PgPoolOptions::new()
        .max_connections(5)
        .connect(&database_url)
        .await
        .expect("Error building a connection pool");

    HttpServer::new(move  {
        App::new()
            .app_data(Data::new(AppState { db: pool.clone() }))
            .service(fetch_users)
            .service(fetch_user_workingdays)
            .service(create_user)
            .service(create_user_workingday)
            .service(update_user)
            .service(update_user_workingday)
            .service(delete_user)
            .service(delete_user_workingday)
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

Die Main-Funktion richtet einen Webserver mit dem Actix-Web-Framework ein. Sie erstellt einen Verbindungspool zu der PostgreSQL-Datenbank unter Verwendung der PgPoolOptions-Struktur aus der sqlx-Bibliothek.
Die dotenv-Funktion wird aufgerufen, um die DATABASE_URL Umgebungsvariable aus der .env-Datei zu laden.
Als nächstes wird ein HttpServer mit der App-Middleware von Actix-Web erstellt. Der Verbindungspool wird mithilfe der app_data-Methode zum App-Status hinzugefügt. Anschließend werden alle Servicemethoden zum Bearbeiten von HTTP-Anfragen mithilfe der Service-Methode zur App hinzugefügt. Schließlich wird der Server an die IP-Adresse localhost auf Port 8080 gebunden und mit der run-Methode gestartet.

4.3.2 services.rs

Die Datei definiert Strukturen und Funktionen mit spezifischen HTTP-Methoden-Annotationen, die Daten von einer Anfrage erhalten und verarbeiten, um Operationen (Create, Read, Update, Delete) auf einer Datenbank durchzuführen.

Importieren von Paketen :

```rust
use actix_web::{
    get, post,put, delete,
    web::{Data, Json, Path},
    Responder, HttpResponse
};
use serde::{Deserialize, Serialize};
```

```
use sqlx::{self, FromRow};
use chrono::{NaiveDateTime};
use crate::AppState;
```

Hier werden actix_web, serde, sqlx und chrono importiert. Die actix_web-Bibliothek wird für die Erstellung von Webanwendungen und APIs in Rust verwendet. Die serde-Bibliothek ist für die Serialisierung und Deserialisierung von Daten zuständig. sqlx ist eine Bibliothek zur Unterstützung von Datenbankzugriffen in Rust. chrono ist eine Bibliothek für Datum und Zeit in Rust. AppState stellt eine Struktur dar, die für die Verwaltung des Anwendungszustands in der Rust-Anwendung verwendet wird.

Definieren von Datenstrukturen :

```
#[derive(Serialize, FromRow)]
struct User {
    id: i32,
    first_name: String,
    last_name: String,
}
#[derive(Serialize, FromRow)]
struct WorkingDay {
    id: i32,
    pub starting_time: Option<String>,
    pub ending_time: Option<String>,
    pub working_hours: Option<String>,
    user_id: i32,
}
#[derive(Deserialize)]
pub struct CreateUser{
    pub first_name: String,
    pub last_name: String,
}
#[derive(Deserialize)]
pub struct CreateWorkingDay{
    pub starting_time: Option<String>,
    pub ending_time: Option<String>,
    pub working_hours: Option<i32>,
}
```

Die Struktur User hat vier Felder: id, first_name, last_name, und ist mit den Traits Serialize und FromRow markiert. Die Struktur WorkingDay hat fünf Felder: id, starting_time, ending_time, working_hours und user_id. starting_time, ending_time und working_hours sind optional, da sie mit dem Datentyp Option definiert sind. Die Strukturen CreateUser und CreateWorkingDay dienen als Eingabe für die Erstellung neuer Benutzer und Arbeitstage. CreateUser hat zwei Felder: first_name und last_name. CreateWorkingDay hat ebenfalls drei Felder: starting_time, ending_time und working_hours.
Diese Strukturen sind wichtig für die Verwaltung von Benutzer- und Arbeitstagdaten. Die Verwendung von Traits wie Serialize und FromRow trägt dazu bei, dass diese Datenstrukturen einfach serialisiert und deserialisiert werden können.

Get-Methoden :

```
#[get("/users")]
pub async fn fetch_users(state: Data<AppState>) -> impl Responder {
    match sqlx::query_as::<_, User>("SELECT * FROM users")
        .fetch_all(&state.db)
        .await
    {
        Ok(users) => HttpResponse::Ok().json(users),
```

```
        Err(err) => HttpResponse::NotFound().json(format!("No users found: {}",
        ↪  err.to_string())),
    }
}
#[get("/users/{id}/workingdays")]
pub async fn fetch_user_workingdays(state: Data<AppState>, path: Path<i32>) -> impl
↪  Responder {
    let id: i32 = path.into_inner();
    match sqlx::query_as::<_, WorkingDay>(
        "SELECT * FROM working_days WHERE user_id = $1"
    )
        .bind(id)
        .fetch_all(&state.db)
        .await
    {
        Ok(workingdays) => HttpResponse::Ok().json(workingdays),
        Err(err) => HttpResponse::NotFound().json(format!("No workingdays found: {}",
        ↪  err.to_string())),
    }
}
```

Die erste Funktion, fetch_users, nimmt eine Data-Instanz des AppState entgegen und gibt alle Benutzer zurück, die in der Datenbank gespeichert sind. Dies wird erreicht, indem eine SQL-Abfrage an die Datenbank gesendet wird, die alle Benutzer abfragt. Das Ergebnis wird dann als JSON formatiert und als HTTP-Antwort zurückgegeben.
Die zweite Funktion, fetch_user_workingdays, nimmt ebenfalls eine Data-Instanz des AppState und eine Path-Instanz entgegen, die den id-Parameter vom User enthält. Die Funktion ruft alle Arbeitstage eines bestimmten Benutzers ab, indem eine SQL-Abfrage an die Datenbank gesendet wird, die die Arbeitstage für den angegebenen Benutzer abfragt. Das Ergebnis wird ebenfalls als JSON formatiert und als HTTP-Antwort zurückgegeben. Diese Funktionen werden verwendet, um Daten von der Datenbank abzurufen und sie als JSON zu formatieren. Die Verwendung von async ermöglicht es, auf die Datenbank zuzugreifen, ohne dass die Anwendung blockiert wird, während die Abfrage ausgeführt wird.

Post-Methoden :

```
#[post("/users")]
pub async fn create_user(state: Data<AppState>, body: Json<CreateUser>) -> impl Responder {
    match sqlx::query(
        "INSERT INTO users (first_name, last_name) VALUES ($1, $2)"
    )
        .bind(&body.first_name)
        .bind(&body.last_name)
        .execute(&state.db)
        .await
    {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(err) => HttpResponse::InternalServerError().json(format!("Failed to create
        ↪  user: {}", err.to_string())),
    }
}
#[post("/users/{id}/workingdays")]
pub async fn create_user_workingday(state: Data<AppState>, path: Path<i32>) -> impl
↪  Responder {
    let id: i32 = path.into_inner();
    match sqlx::query_as::<_, WorkingDay>(
        "INSERT INTO working_day (user_id) VALUES ($1) RETURNING id, starting_time,
        ↪  ending_time, working_hours, user_id"
    )
```

```
        .bind(id)
        .fetch_one(&state.db)
        .await
    {
        Ok(workingday) => HttpResponse::Ok().json(workingday),
        Err(err) => HttpResponse::InternalServerError().json(format!("Failed to create
        ↪  working day: {}", err.to_string()))
    }
}
```

Die erste Methode create_user akzeptiert eine JSON-Anfrage mit den Feldern first_name und last_name und fügt diese Werte in die users-Tabelle in der Datenbank ein. Die zweite Methode create_user_workingday akzeptiert eine Anfrage mit der Benutzer-ID und fügt einen neuen Arbeitstag für diesen Benutzer in die working_days-Tabelle ein. Der neu erstellte Arbeitstag wird dann in der Antwort zurückgegeben.

Put-Methoden :

```
#[put("/users/{id}")]
pub async fn update_user(state: Data<AppState>, path: Path<i32>, body: Json<CreateUser>) ->
↪  impl Responder {
    let id: i32 = path.into_inner();
    match sqlx::query(
        "UPDATE users SET first_name = $1, last_name = $2 WHERE id = $3"
    )
        .bind(&body.first_name)
        .bind(&body.last_name)
        .bind(id)
        .execute(&state.db)
        .await
    {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(err) => HttpResponse::InternalServerError().json(format!("Failed to update
        ↪  user: {}", err.to_string()))
    }
}
#[put("/users/{id}/workingdays/{workingday_id}")]
pub async fn update_user_workingday(
    state: Data<AppState>,
    path: Path<(i32, i32)>,
    body: Json<CreateWorkingDay>,
) -> impl Responder {
    let (id, workingday_id): (i32, i32) = path.into_inner();
    // Parse starting_time and ending_time
    let start_time_result =
    ↪  NaiveDateTime::parse_from_str(&body.starting_time.as_ref().unwrap(), "%Y-%m-%d
    ↪  %H:%M:%S");
    let end_time_result =
    ↪  NaiveDateTime::parse_from_str(&body.ending_time.as_ref().unwrap(), "%Y-%m-%d
    ↪  %H:%M:%S");
    // Check if both start_time and end_time were parsed successfully
    if let (Ok(start_time), Ok(end_time)) = (start_time_result, end_time_result) {
        let duration = end_time - start_time;
        let hours = duration.num_hours();
        let minutes = duration.num_minutes() - hours * 60;
        let seconds = duration.num_seconds() - hours * 3600 - minutes * 60;
        let working_hours = format!("{}:{}:{}", hours, minutes, seconds);
        match sqlx::query(
            "UPDATE working_days SET starting_time = $1, ending_time = $2, working_hours =
            ↪  $3 WHERE id = $4 AND user_id = $5"
```

```
        )
        .bind(&body.starting_time)
        .bind(&body.ending_time)
        .bind(&working_hours)
        .bind(workingday_id)
        .bind(id)
        .execute(&state.db)
        .await
        {
            Ok(_) => HttpResponse::Ok().finish(),
            Err(err) => HttpResponse::InternalServerError().json(format!("Failed to update
            ↪  working day: {}", err.to_string()))
        }
    } else {
        // Handle case where either start_time or end_time could not be parsed
        HttpResponse::BadRequest().json("Invalid date format")
    }
}
```

Die erste Methode update_user aktualisiert die Informationen des Benutzers in der Datenbank. Der Endpunkt nimmt den Benutzer-ID-Pfadparameter und einen JSON-Body mit den aktualisierten Informationen des Benutzers entgegen. Die SQL-Anweisung wird dann von der Datenbank ausgeführt.
Die zweite Methode update_user_workingday aktualisiert die Informationen eines Arbeitstags des Benutzers in der Datenbank. Der Endpunkt nimmt die Benutzer-ID und die Arbeitstags-ID als Pfadparameter und einen JSON-Body mit den aktualisierten Informationen des Arbeitstags entgegen. Dann werden die die start_time und end_time aus dem Body als NaiveDateTime geparst und daraus die Arbeitsstunden des Arbeitstags berechnet. Wenn die Start- und Endzeit erfolgreich geparst wurden, bindet der Code die Arbeitszeit an die SQL-Anweisung und führt sie aus.

Delete-Methoden :

```
#[delete("/users/{id}")]
pub async fn delete_user(state: Data<AppState>, path: Path<i32>) -> impl Responder {
    let id: i32 = path.into_inner();
    match sqlx::query(
        "DELETE FROM users WHERE id = $1"
    )
        .bind(id)
        .execute(&state.db)
        .await
    {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(err) => HttpResponse::InternalServerError().json(format!("Failed to delete
        ↪  user: {}", err.to_string()))
    }
}

#[delete("/users/{id}/workingdays/{workingday_id}")]
pub async fn delete_user_workingday(state: Data<AppState>, path: Path<(i32, i32)>) -> impl
↪  Responder {
    let (id, workingday_id): (i32, i32) = path.into_inner();
    match sqlx::query(
        "DELETE FROM working_days WHERE id = $1 AND user_id = $2"
    )
        .bind(workingday_id)
        .bind(id)
        .execute(&state.db)
        .await
```

```
26      {
27          Ok(_) => HttpResponse::Ok().finish(),
28          Err(err) => HttpResponse::InternalServerError().json(format!("Failed to delete
            ↪  working day: {}", err.to_string()))
29      }
30  }
```

Die erste Metode delete_user löscht den Benutzer anhand seiner ID aus der Datenbank. Wenn ein Nutzer gelöscht wird, werden auch seine Arbeitstage gelöscht, dies wurde in der Datenbank implementiert.
Die zweite Metode delete_user_workingday löscht einen Arbeitstag eines Benutzers. Der Arbeitstag wird anhand der IDs des Benutzers und des Arbeitstags in der Datenbank gelöscht. Wenn der Löschvorgang erfolgreich ist, wird eine HTTP-Antwort mit dem Statuscode 200 zurückgegeben. Andernfalls wird eine HTTP-Antwort mit dem Statuscode 500 zurückgegeben, die den Fehler enthält, der beim Löschen des Arbeitstags aufgetreten ist.

5 Frontend

6 Deployment

6.1 Docker

6.1.1 Installation auf dem Linuxserver

1. auf den Linuxserver gehen
2. den Befehl sudo apt-get update ausführen
3. den Befehl sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin ausführen. (https://docs.docker.com/engine/install/ubuntu/)
4. um zu testen ob Docker funktioniert kann der Befehl sudo docker run hello-world ausgeführt werden.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung der wichtigsten Punkte

7.2 Ausblick auf zukünftige Entwicklungen

8 Anhang

8.1 Glossar

8.2 Beispielcode

8.3 Referenzen und weiterführende Literatur