HOCHSCHULE COBURG

Hochschule für angewandte Wissenschaften Coburg

Fakultät Elektrotechnik und Informatik

Secure Software Engineering

Gruppe 8

Adrian Becker, Lukas Scheler

09.06.2024

Inhaltsverzeichnis

1 Projektbeschreibung

Im Rahmen des Fachs Secure Software Engineering wurde eine Chat-Anwendung in der Programmiersprache Rust entwickelt. Diese Dokumentation beschreibt die Entwicklung der Anwendung, die es Nutzern ermöglicht, in Echtzeit Nachrichten zu senden und zu empfangen.

1.1 Ziele der Chat-App

Das Hauptziel dieser Chat-App ist es, eine sichere, effiziente und benutzerfreundliche Plattform für die Echtzeitkommunikation bereitzustellen. Besondere Schwerpunkte liegen auf der Gewährleistung der Datensicherheit und dem Schutz der Privatsphäre der Nutzer. Weitere Ziele sind:

- Kenntnisse in der Programmiersprache Rust erlernen
- Erfahrung in der Dokumentation eines Softwareprojekts sammeln
- Konkrete Planung und Durchführung eines Projekts
- Hohe Performance und Zuverlässigkeit
- Einfache und intuitive Benutzeroberfläche
- Modularität und Erweiterbarkeit des Codes

2 Anforderungen

Die Anforderungen für dieses Projekt wurden umfassend in Redmine dokumentiert, einem webbasierten Projektmanagement-Tool zur Verwaltung von Projekten, Aufgaben und Anforderungen. Diese Anforderungen umfassen:

- Funktionale Anforderungen: Beschreiben die grundlegenden Funktionen der Chat-App, wie das Senden und Empfangen von Nachrichten, Benutzerregistrierung und -anmedung sowie die Verwaltung von Benutzerkonten.
- Nicht-funktionale Anforderungen: Betreffen die Qualitätseigenschaften der Anwendung wie Leistung, Benutzerfreundlichkeit, Zuverlässigkeit und Wartbarkeit.
- Sicherheitsanforderungen: Maßnahmen zum Schutz der Datenintegrität, Vertraulichkeit der Nachrichten, sichere Authentifizierung und Autorisierung von Benutzern sowie Schutz vor verschiedenen Arten von Angriffen wie SQL-Injection und Man-in-the-Middle-Angriffen.

3 Aufteilung der Entwicklungsarbeit

Für die Entwicklung der Chat-Applikation in Rust wurden die Aufgaben möglichst gleich verteilt. Dabei wurde darauf geachtet, dass jeder möglichst viel mit Rust arbeitet und sich dennoch in einen Bereich spezialisieren kann. Im folgenden Bild ist die Aufteilung sichtbar:

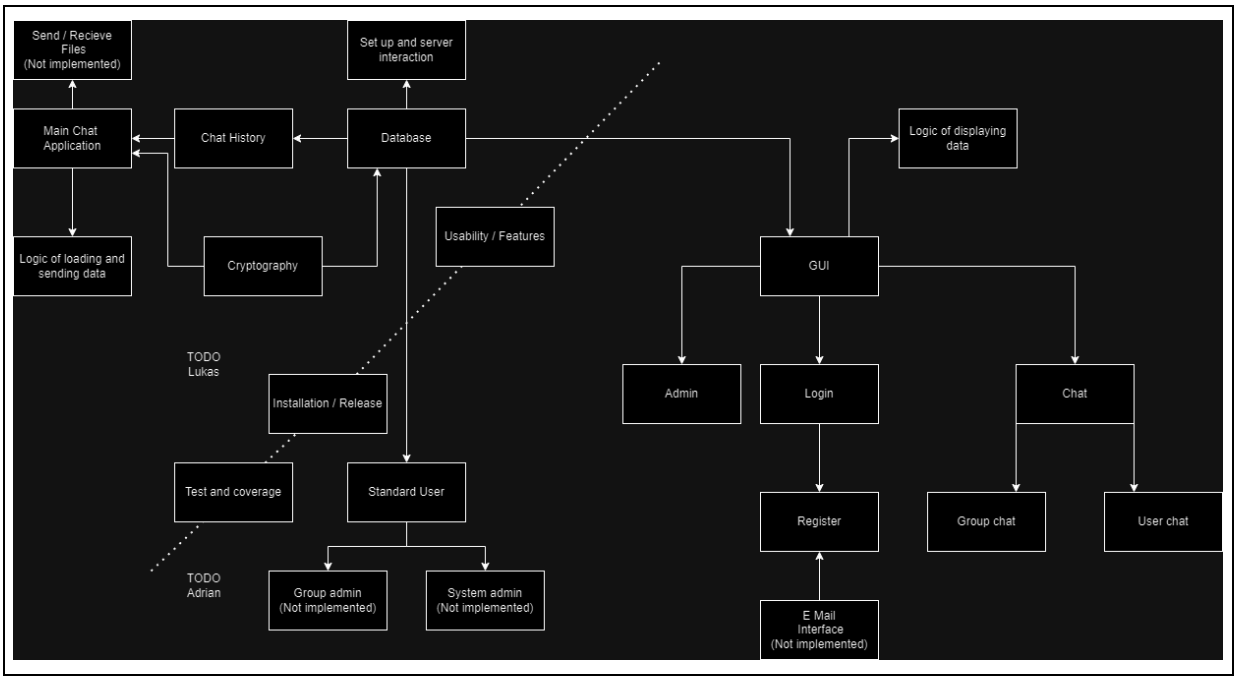


Abbildung 1 Aufteilung der Entwicklungsarbeit

Adrian war für die Benutzeroberfläche und die Benutzerinteraktionen verantwortlich, während Lukas sich um die Datenbank und die allgemeine interne Logik kümmerte. Das Schreiben von Tests und das Erreichen einer bestimmten Code-Coverage wurden aufgeteilt. Dasselbe galt für Usability-Tests und Release-Build-Tests. Einige geplante Features, wie das Versenden von Dateien und eine E-Mail-Schnittstelle zur Verifizierung, wurden aufgrund des hohen Aufwands und Zeitmangels nicht implementiert.

4 Systemarchitektur

Die Systemarchitektur der Chat-App ist darauf ausgelegt, eine effiziente und sichere Kommunikation zwischen den Benutzern zu gewährleisten. Die Hauptkomponenten des Systems sind:

Client-Komponente:

Dies ist die Benutzerschnittstelle, über die Nutzer Nachrichten senden und empfangen. Die Client-Komponente interagiert direkt mit der Datenbank.

Datenbank:

Eine zentrale Datenbank speichert alle Nachrichten und Benutzerdaten. Der Zugriff auf die Datenbank erfolgt über gesicherte Verbindungen.

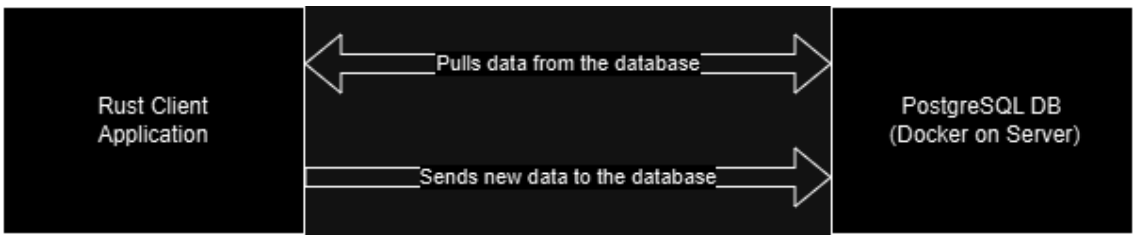
Übersicht über die Komponenten:

- Client: Eine in Rust geschriebene Anwendung, die Benutzereingaben entgegennimmt, Nachrichten in die Datenbank schreibt und neue Nachrichten abruft.
- Datenbank: Eine relationale PostgreSQL-Datenbank, die alle notwendigen Daten speichert. Die Kommunikation erfolgt über Rust Diesel.

Interaktionen zwischen den Komponenten:

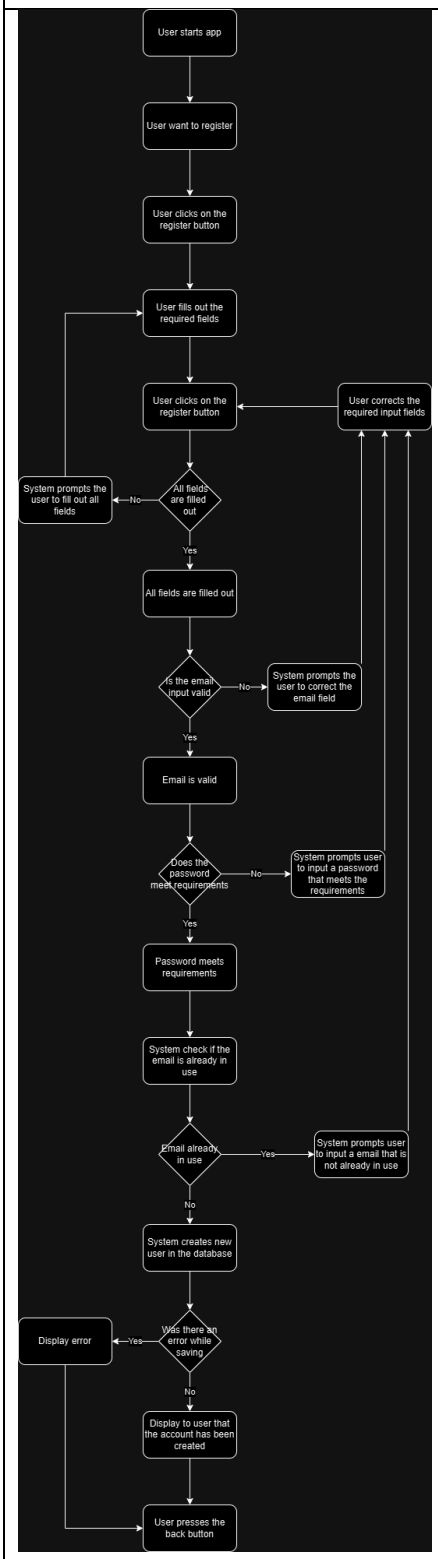
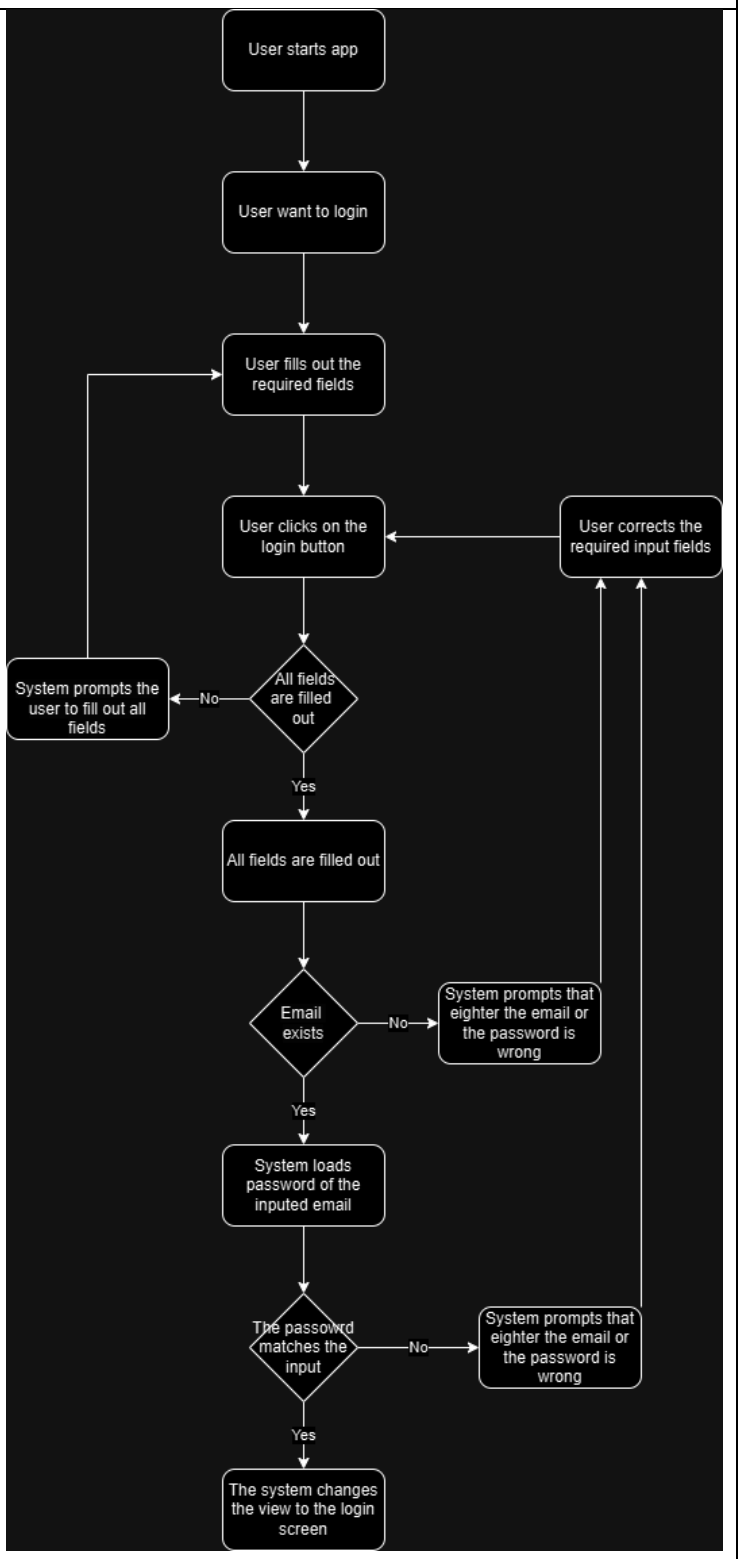
- Nachrichtenversand: Der Client nimmt eine Nachricht vom Benutzer entgegen und speichert sie in der Datenbank.
- Nachrichtenabruf: Der Client führt regelmäßige Abfragen (Polling) durch, um neue Nachrichten aus der Datenbank abzurufen und anzuzeigen. Die Datenbank wird auf dem Server der Hochschule gehostet und ist ausschließlich über das Netzwerk der Hochschule Coburg zugänglich.

Nachfolgend eine grafische Darstellung der Architektur:

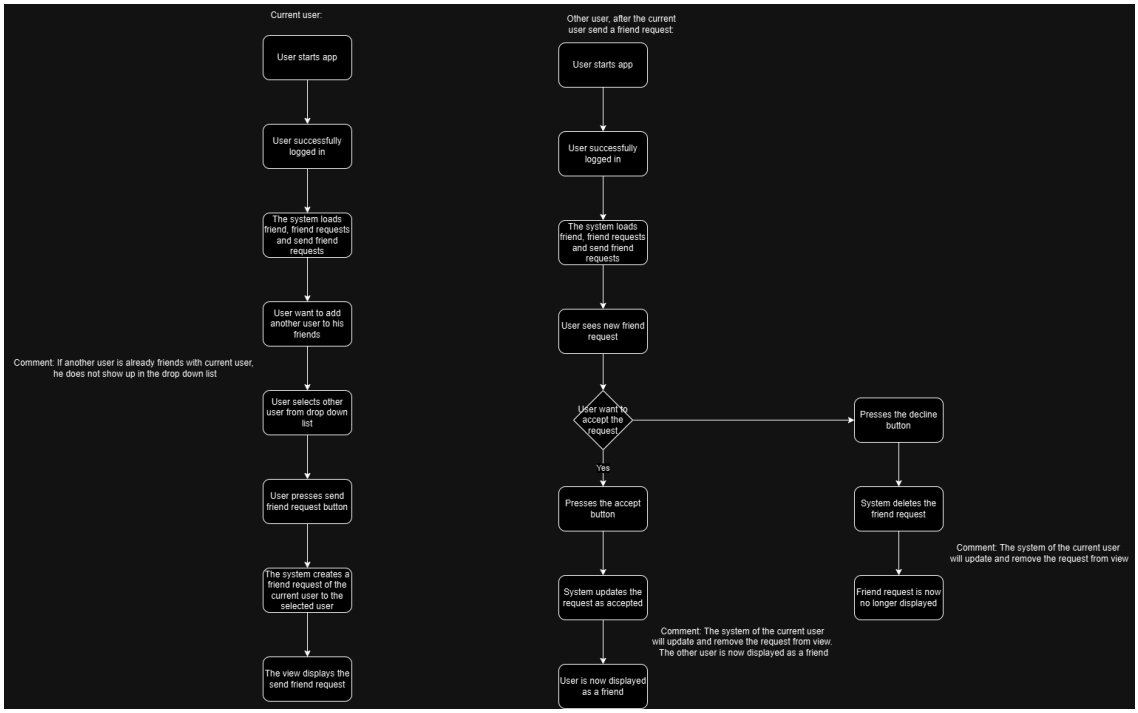


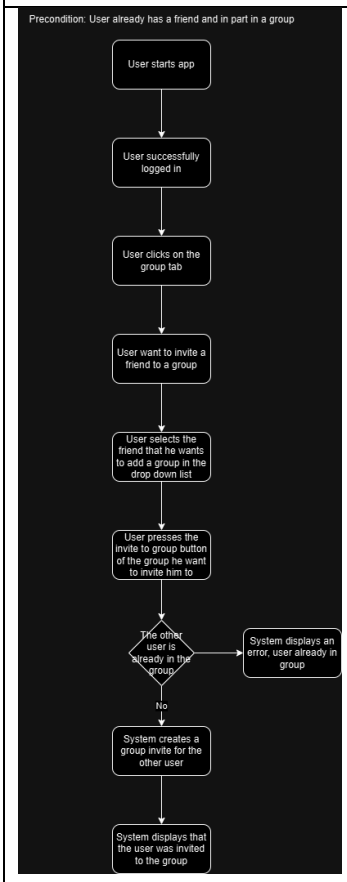
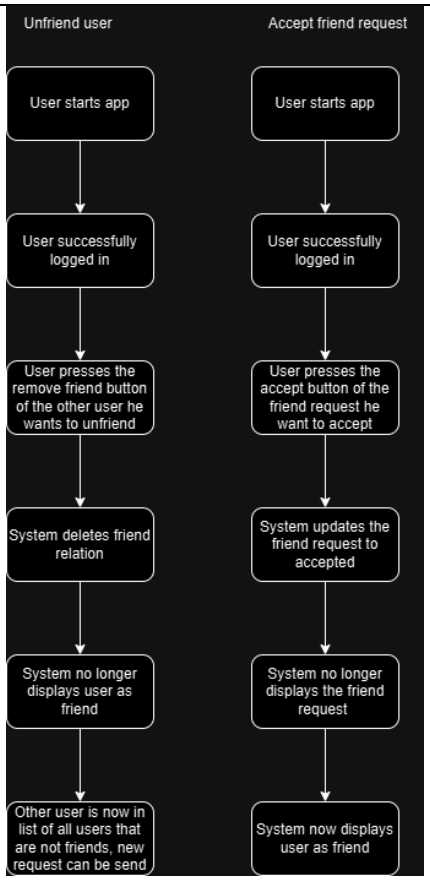
4.1 Use-Cases

Im Folgenden werden einige Use-Cases dargestellt, um die Abläufe bei der Interaktion mit der App und das interne Verhalten der App zu verdeutlichen:

Registrierung:	Login:
	

Freundesanfrage senden:



Freund in eine Gruppe einladen:	Freundschaften beenden /Freundschaftsanfragen annehmen
	

Weitere Use-Cases folgen einem ähnlichen Muster, die Grundstruktur sollte bereits aus den gezeigten Beispielen klar werden.

4.2 Bewertung der Struktur

Die Bewertung umfasst folgende Punkte, die Teil der Projektziele waren:

- Performance: Die Performance der Anwendung ist stark von der Nähe zur Datenbank abhängig. Da der Lademechanismus aktiv auf Daten abfragt, indem er alle 5-10 Sekunden prüft, ob die Datenbank neue Daten hat, kann es zu Leistungseinbußen beim Handling kommen, was zu Verzögerungen führen kann. Dies tritt insbesondere auf, wenn das Hochschul-VPN verwendet wird, um auf die Datenbank zuzugreifen. Vor Ort treten diese Probleme nicht auf.
- Sicherheit: Die Hauptanfälligkeiten betreffen Angriffe auf die Datenbankverbindung, da Diesel verwendet wird, um mit der Datenbank zu interagieren. Diesel verschlüsselt Nachrichten nicht, was bedeutet, dass unsere Anwendung dies selbst tun müsste, was jedoch außer beim Verschlüsseln des Benutzerpassworts mit bcrypt nicht erfolgt. Weitere Schwachstellen sind, dass die Datenbank nur einen Benutzer mit Adminrechten hat, was geändert werden sollte. Ebenso bedenklich ist die Existenz einer .env-Datei, die einen Verbindungsstring zur Datenbank enthält, inklusive des Passworts für den Admin-Benutzer. Darüber hinaus ist die Verwendung von „Root“ als Passwort für den Admin-Benutzer nicht optimal.

5 Implementierung

Für die Entwicklung der Chat-App wurden verschiedene Werkzeuge und eine spezifische Entwicklungsumgebung verwendet:

- Programmiersprache: Rust
- Entwicklungsumgebung (IDE): Visual Studio Code mit den Rust-Tools
- Versionskontrollsystem: Git, gehostet auf GitHub
- Datenbank: PostgreSQL
- Abhängigkeitsmanagement: Cargo (Rust's Paketmanager und Build-System)

5.1 Wichtige Codeabschnitte

Hier werden einige der zentralen Codeabschnitte und ihre Funktionalitäten beschrieben.

Datenbankverbindung:

```
pub fn establish_connection() -> Result<PgConnection, diesel::ConnectionError>
{
    let _guard = CONNECTION_MUTEX
        .lock()
        .expect("Failed to acquire mutex lock");
    dotenv().expect("Failed to read .env file");

    if let Ok(database_url) = env::var("DATABASE_URL") {
        if let Ok(connection) = PgConnection::establish(&database_url) {
            return Ok(connection);
        }
    }

    Err(diesel::ConnectionError::BadConnection(
        "Failed to establish connection to databases".to_string(),
    ))
}
```

Code 1: Code um eine Verbindung zur Datenbank zu erstellen

Nachrichtenaustausch (Pulling-Mechanismus)

Die Nachrichten werden durch regelmäßiges Abfragen (Polling) aus der Datenbank abgerufen. Ein Beispiel für das Abrufen neuer Nachrichten:

```
    fn subscription(&self) -> iced::Subscription<Message> {
        match self.current_page {
            Page::Home => time::every(Duration::from_secs(10))
                .map(|_| Message::HomeMessage(home::HomeMessage::Tick)),
```

```
            Page::UserChat => time::every(Duration::from_secs(5))
                .map(|_| Message::UserChatMessage(user_chat::UserChatMes-
sage::Tick)),
            Page::GroupChat => time::every(Duration::from_secs(5))
                .map(|_| Message::GroupChatMessage(group_chat::GroupChatMes-
sage::Tick)),
            _ => Subscription::none(),
        }
    }
```

Code 2: Code um Pulling zu betreiben

Verschlüsselungsmechanismen:

```
let hashed_password = match hash(user.password, DEFAULT_COST) {
        Ok(h) => h,
        Err(err) => return Err(format!("Failed to hash password: {}", err)),
    };
```

Code 3: Verschlüsselungsfunktion von Passwörtern

6 Testen und Coverage

Während der Entwicklung wurden verschiedene Test- und Codeabdeckungsziele festgelegt und erfolgreich erreicht. Diese lauten wie folgt:

- Jede Funktion, die eine Interaktion mit der Datenbank beinhaltet, sollte getestet werden.
- Funktionen, die sowohl positive als auch negative Fälle abdecken, sollten jeweils Tests für beide Fälle haben.
- Tests müssen vollständig unabhängig voneinander sein, ohne gegenseitige Beeinflussung.
- Tests, die mit der Datenbank interagieren, müssen ihre Änderungen rückgängig machen, um die Datenkonsistenz zu gewährleisten.
- Tests müssen beliebig oft wiederholbar sein.
- Eine Codeabdeckung von mindestens 80 % muss erreicht werden. Aktuell beträgt sie 89 %.
- Nur triviale Codefragmente dürfen nicht in der Codeabdeckung enthalten sein, z. B. UI-Code, der nicht angemessen getestet werden kann.

Insgesamt wurden 102 Tests geschrieben, die beliebig oft wiederholt werden können. Alle generierten Daten werden gelöscht, und wichtige Tests werden sogar mehrfach durchgeführt. Es besteht jedoch eine Abhängigkeit in einigen Tests, die Dummy-Daten aus der Datenbank benötigen. Diese müssen nach dem Neuaufsetzen der Datenbank erneut eingefügt werden.

Das Neuaufsetzen der Datenbank kann durch folgenden Code erreicht werden:

```
fn setupfortests() {
    let _ = create_user("John", "Doe", "test1@email.de", "n)+L8ZVWw$qKXDQo")
        .unwrap_or_else(|err| panic!("Failed to create user for {}: {}",
"test1@email.de", err));
    let _ = create_user("Jane", "Doe", "test2@email.de", "n)+L8ZVWw$qKXDQo")
        .unwrap_or_else(|err| panic!("Failed to create user for {}: {}",
"test2@email.de", err));
    let _ = create_user_friend(1, 2, true, true).unwrap();
    let _ = create_group("Default Group").unwrap();

    let _ = create_user_group(1, 1, true);

    let _ = create_group_message(1, 1, "Hello, World!");

    let _ = create_user_message(2, 1, "Hello, User 1!");
}
```

Imports der Funktionen wurden vernachlässigt.

7 Zusammenfassung und Ausblick

Die Entwicklung der Chat-App in Rust hat gezeigt, dass Rust eine leistungsfähige und sichere Programmiersprache ist, die sich gut für die Erstellung von sicherheitskritischen Anwendungen eignet. Im Laufe des Projekts wurden folgende Hauptziele erreicht:

- Sichere Kommunikation: Durch die Implementierung von TLS und sicheren Authentifizierungsmechanismen wurden die Daten der Nutzer geschützt.
- Effiziente Implementierung: Der Einsatz von Rust und asynchronen Programmiertechniken hat zu einer performanten Anwendung geführt.
- Robuste Datenhaltung: Die Verwendung von PostgreSQL gewährleistet eine zuverlässige Speicherung und Abruf von Nachrichten.
- Umfassende Tests: Durch Unit-Tests, Integrationstests und Sicherheitstests konnte die Qualität und Sicherheit der Anwendung sichergestellt werden.

7.1 Ausblick und mögliche Erweiterungen

Es gibt mehrere Möglichkeiten, die Chat-App in Zukunft weiterzuentwickeln und zu verbessern:

Erweiterung der Funktionalität

- Echtzeitkommunikation: Implementierung von WebSockets, um Echtzeitkommunikation zu ermöglichen und die Notwendigkeit des Pollings zu eliminieren.
- Dateiübertragungen: Ermöglichen des Sendens und Empfangens von Dateien innerhalb der Chat-App.

Verbesserung der Sicherheit

- Zwei-Faktor-Authentifizierung (2FA): Einführen zusätzlicher Sicherheitsschichten durch Zwei-Faktor-Authentifizierung.
- Erweiterte Verschlüsselung: Implementierung von Ende-zu-Ende-Verschlüsselung, um die Sicherheit der Nachrichten weiter zu erhöhen.

Benutzerfreundlichkeit

- Mobile Unterstützung: Entwicklung von mobilen Anwendungen für Android und iOS

7.2 Lessons Learned

Während der Entwicklung der Chat-App wurden mehrere wertvolle Erkenntnisse gewonnen:

Bedeutung der Sicherheit:

Die Integration von Sicherheitsmaßnahmen von Anfang an ist entscheidend, um eine sichere Anwendung zu gewährleisten. Sicherheitslücken lassen sich schwer beheben, wenn sie erst in späteren Entwicklungsphasen entdeckt werden.

Rust als Programmiersprache:

Rust hat sich als gute Wahl für sicherheitskritische Anwendungen erwiesen. Die strengen Typprüfungen und das Ownership-System helfen dabei, viele gängige Programmierfehler zu vermeiden, die zu Sicherheitslücken führen können.

Testen und Qualitätssicherung:

Umfassende Tests sind unerlässlich, um die Zuverlässigkeit und Sicherheit der Anwendung zu gewährleisten. Automatisierte Tests helfen dabei, sicherzustellen, dass neue Änderungen keine bestehenden Funktionen beeinträchtigen.

Teamarbeit:

Teamarbeit und Kommunikation: Regelmäßige Meetings und ein gut dokumentiertes Vorgehen sind entscheidend für den Projekterfolg.

Projektmanagement:

Die Einhaltung der Zeitpläne und das Priorisieren von Aufgaben sind essenziell, um die Projektziele zu erreichen.