PROJEKTARBEIT
SOFTWARE- ENTWICKLUNG 3

14.02.2025

Abgegeben von:

Anna-Lena Miletic

Franziska Landt

Gabriel Eißler

Isaak Kulikowskich

Polina Rogge

Sofia Koumpatidou

YuTing Chen

Inhaltsverzeichnis

1. Einleitung 2
    1.1 Ziel des Projekts 2
    1.2 Funktionsübersicht 2
    1.3 Technologien und Werkzeuge 2
2. Anforderungsanalyse 3
    2.1 Technische Anforderungen 3
    2.3 Fachliche Anforderungen 3
    2.4 Verknüpfung der Anforderung 4
    2.5 Priorisierung 4
3. Softwarearchitektur & Clean Code 5
    3.1 Architekturübersicht 5
    3.2 Architekturmuster 5
    3.3 Architekturdiagramme 6
    3.4 Clean Code Prinzipien 8
    3.5 Architecture Decision Records (ADR) 9
4. Git-Management/Versionierung 13
    4.1 Git-Workflow 13
    4.2 Tagging & Changelog 13
    4.3 Readme-Dokumentation 13
5. Testing 15
    5.1 Unit-Tests 15
6. CI/CD-Pipeline 27
    6.1 Aufbau der Pipeline 27
7.Persistenz 30
    7.1 Systembeschreibung der Datenbank 30
    7.2 CRUD-Operationen 31
8. Schnittstellen (API-Design) 31
    8.1 API-Überblick 31
9. UI 36
    9.1 UI-Architektur 36
    9.2 GUI-Komponenten 37
    9.3 Asynchrone Prozesse 38
10. Reflexion & Lessons Learned 38
    10.1 Projektreflexion 38
    10.2 Individuelle Reflexionen 38
    10.3 Verbesserungsmöglichkeiten - Ausblick in die Zukunft 41

1. Einleitung

1.1 Ziel des Projekts

Das Ziel des Projekts war es, ein Planungstool zu entwickeln, das die Verwaltung und Organisation von Projektarbeiten effizienter gestaltet. Die Software ermöglicht es, Aufgaben zu erstellen, zuzuweisen, zu dokumentieren und den Fortschritt zu verfolgen.

Die wichtigsten Aspekte beinhalten:

• Zuordnung von GitHub-Nutzern zu Projektteilnehmern

• Kurze Beschreibung des Projekts

• Problemstellung und Motivation

Als Wahlthema haben wir **UI** und **Schnittstellen** gewählt.

1.2 Funktionsübersicht

Das Tool ermöglicht es den Nutzern, mehrere Projekte gleichzeitig zu verwalten und diese auf einfache Weise zu organisieren. Es bietet die Möglichkeit, neue Projekte zu erstellen, eine detaillierte Beschreibung hinzuzufügen und Teilnehmer in das Projekt einzubinden. Zudem können Aufgaben priorisiert und ihr Schwierigkeitsgrad festgelegt werden. Für eine bessere Strukturierung lassen sich Bearbeitungszeiten definieren, sodass genau ersichtlich ist, wann eine Aufgabe begonnen und abgeschlossen werden soll. Falls eine große Anzahl an Aufgaben vorliegt, können diese durch Filter- und Sortierfunktionen übersichtlicher dargestellt werden, um eine Überforderung der Nutzer zu vermeiden.

1.3 Technologien und Werkzeuge

Bei der Wahl der Technologien haben wir uns für C# entschieden. Wir wollten eine neue Herausforderung und gleichzeitig eine moderne, leistungsfähige Sprache nutzen, die uns ermöglicht, eine stabile und gut strukturierte Anwendung zu entwickeln.

Neben C# setzen wir auf einige bewährte Frameworks und Tools:

• **.NET**: Da wir eine skalierbare und robuste Anwendung wollten, war .NET die perfekte Wahl. Es bietet viele nützliche Funktionen und erleichtert uns die Entwicklung enorm.

• **Avalonia**: Wir haben uns für Avalonia als UI-Framework entschieden, weil wir unsere Anwendung plattformübergreifend nutzbar machen wollten. So können wir das Tool nicht nur unter Windows, sondern auch unter Linux oder macOS verwenden.

• **PetaPoco**: Eine leichte, unkomplizierte ORM-Lösung, die uns hilft, effizient mit der Datenbank zu arbeiten, ohne unnötigen Overhead.

Als Entwicklungsumgebung haben wir **JetBrains** Rider genutzt. Die IDE bietet uns einfach alles, was wir brauchen – von hilfreichen Code-Analysen bis hin zu einer übersichtlichen Benutzeroberfläche. So konnten wir schneller und produktiver arbeiten. Die Versionskontrolle läuft über **GitHub**, was für uns eine klare Wahl war. Dort konnten wir unseren Code sicher verwalten, gemeinsam daran arbeiten und Änderungen leicht nachvollziehen. Gerade bei einem Teamprojekt wie diesem ist eine gute Versionskontrolle unverzichtbar.

2. Anforderungsanalyse

2.1 Technische Anforderungen

Uns war von Anfang an klar, dass wir eine solide Architektur brauchen, um das Tool langfristig gut pflegbar und erweiterbar und erweiterbar zu machen. Daher haben wir uns für eine **3-SchichtenArchitektur** entschieden, die eine klare Trennung zwischen Datenbank, Logik und Benutzeroberfläche sicherstellt. Um das Ganze noch sauberer zu strukturieren, haben wir das MVVM-Pattern (Model-ViewViewModel) verwendet. Dadurch können wir die Benutzeroberfläche und die dahinterliegende Logik voneinander trennen, was uns nicht nur die Entwicklung erleichtert, sondern auch zukünftige Anpassungen und Erweiterungen vereinfacht. Zusätzlich setzen wir DTOs (Data Transfer Objects) ein, um die Kommunikation zwischen den einzelnen Komponenten der Anwendung effizient zu gestalten. Die API sorgt dafür, dass Daten sicher und strukturiert ausgetauscht werden können.

2.3 Fachliche Anforderungen

Bei der Planung haben wir uns gefragt: Was brauchen wir, damit unser Tool nützlich ist und sich von anderen Tools abhebt? Dabei sind wir auf folgende Kernanforderungen gestoßen:

• **Projekte erstellen und verwalten**: Ein zentrales Element unserer Anwendung ist die Möglichkeit, neue Projekte zu erstellen und sie detailliert zu beschreiben.

• **Aufgaben verwalten**: Die Nutzer müssen Aufgaben erstellen, bearbeiten und löschen können. Dabei können sie Prioritäten setzen, Schwierigkeitsgrade definieren und Bearbeitungszeiträume festlegen.

• **Zuweisung von Aufgaben**: Da wir das Tool für Teamarbeit entwickelt haben, war es uns besonders wichtig, dass Aufgaben an spezifische Teammitglieder zugewiesen werden können.

• **Fortschrittsverfolgung**: Um den Überblick nicht zu verlieren, sollen Nutzer jederzeit den Status einer Aufgabe oder eines gesamten Projekts einsehen können(KanbanBoard).

2.4 Verknüpfung der Anforderung

Bei der Entwicklung unseres Tools war es entscheidend, dass die Architektur unsere funktionalen Anforderungen optimal unterstützt. Daher haben wir eine Struktur gewählt, die eine effiziente Verarbeitung und Verwaltung von Projekten, Aufgaben und Nutzerinteraktionen ermöglicht.

Durch die 3-Schichten-Architektur werden Datenhaltung, Geschäftslogik und Benutzeroberfläche klar voneinander getrennt. Dies sorgt nicht nur für eine bessere Wartbarkeit, sondern ermöglicht auch eine saubere Umsetzung der Kernfunktionen, wie die Verwaltung von Projekten und Aufgaben.

Die Datenverarbeitung und -übertragung erfolgt über eine API, die eine sichere und strukturierte Kommunikation zwischen den Komponenten sicherstellt. So können Aufgaben erstellt, bearbeitet und spezifischen Nutzern zugewiesen werden, während Änderungen direkt mit der Remote-Datenbank synchronisiert werden.

Zur Verbesserung der Benutzerinteraktion haben wir Mechanismen zur Statusverfolgung implementiert. Nutzer können jederzeit den Fortschritt von Projekten und Aufgaben einsehen, was eine transparente Teamarbeit ermöglicht. Die Integration von Animationen und optischen Rückmeldungen verstärkt zudem die Benutzerfreundlichkeit.

Die gewählte Architektur erlaubt es uns, die Anforderungen effizient umzusetzen und gleichzeitig eine Grundlage für zukünftige Erweiterungen zu schaffen.

2.5 Priorisierung

Beim Entwickeln eines neuen Tools stellt sich immer die Frage: Welche Funktionen sind wirklich essenziell, und welche können erst später kommen? Wir haben uns dabei an der **MVP-Methode** (Minimum Viable Product) orientiert, also ein funktionierendes Grundgerüst geschaffen, das direkt einen Mehrwert bietet, ohne mit unnötigen Features überladen zu sein. Unser Fokus lag darauf, die Kernfunktionen stabil und intuitiv nutzbar zu machen. Zusätzliche Features, wie erweiterte Statistiken oder detailliertere Filteroptionen, haben wir bewusst erst einmal nach hinten gestellt. So konnten wir sicherstellen, dass das Grundkonzept steht, bevor wir uns um Erweiterungen kümmern.

3. Softwarearchitektur & Clean Code

3.1 Architekturübersicht

Unsere Softwarearchitektur basiert auf einem Schichtenmodell mit den Ebenen Benutzeroberfläche (UI), Services, Core und Data. Dieses Modell trennt klar die Verantwortlichkeiten und sorgt für eine saubere, wartbare Codebasis.

Die UI-Ebene zeigt die Anwendung und interagiert mit dem Benutzer. Wir nutzen das MVVM-Muster (Model-View-ViewModel), um Darstellung und Logik zu trennen.

Die Services-Ebene vermittelt zwischen UI und den darunterliegenden Schichten. Sie stellt die Geschäftslogik bereit und verarbeitet Anfragen und greift auf Core und Data zu.

Die Core-Ebene enthält Models und Enums. Diese sind die Blueprints der wichtigsten Objekte unseres Projekts. Sie sind die Basis für alle unsere Prozesse.

Die Data-Ebene ist für die Datenpersistenz verantwortlich und stellt Schnittstellen zu den Datenquellen bereit.

Diese Schichtenarchitektur bildet klare Verantwortlichkeiten und eine wartbare Codebasis. Jede Ebene ist unabhängig von den anderen, was die Entwicklung und die Erweiterbarkeit des Projekts erleichtert und übersichtlicher macht.

3.2 Architekturmuster

Wir folgen den SOLID-Prinzipien, um die Qualität und Wartbarkeit des Codes zu erhöhen:

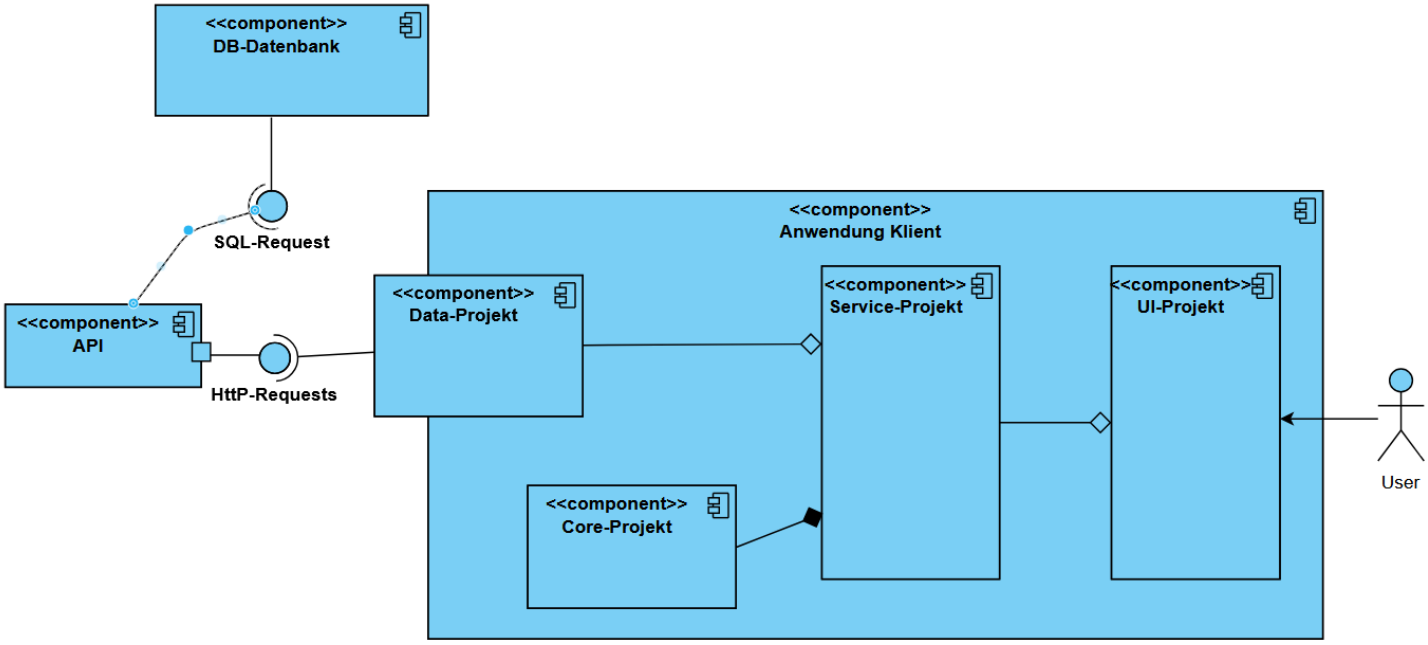
- **SRP:** Jede Klasse hat nur eine Verantwortung, was die Wartung erleichtert.
- **OCP:** Komponenten sind offen für Erweiterungen, aber geschlossen für Modifikationen.
- **LSP:** Abgeleitete Klassen können ohne Änderungen der Funktionalität eingesetzt werden.
- **ISP:** Schnittstellen enthalten nur notwendige Methoden, was die Flexibilität erhöht.
- **DIP:** Höhergestellte Module hängen von abstrakten Schnittstellen ab, nicht von niedriggestellten Modulen.

Vor Allem unsere Service-Schicht haben wir besonders aufgeteilt in separate Bereiche. Für die Wichtigsten Funktionen (erstellen von Tasks, Projects und die Userfunktionen) sind in separaten Serviceklassen, die nur die relevanten Funktionen enthalten.
Zu diesen jeweiligen Service-Klassen haben wir ebenfalls Interfaces implementiert.

Weiterhin haben wir Repository-Klassen, die dafür zuständig sind, den Zugriff auf Datenquellen (wie Datenbanken, APIs, etc.) zu abstrahieren und zu verwalten.
Wir haben uns außerdem dazu entschieden Factories zu verwenden, um die Umwandlung von DTO in Model und andersherum deutlich zu erleichtern. Auch hier sind diese Klassen aufgeteilt in TaskMapper, ProjectMapper und UserMapper.

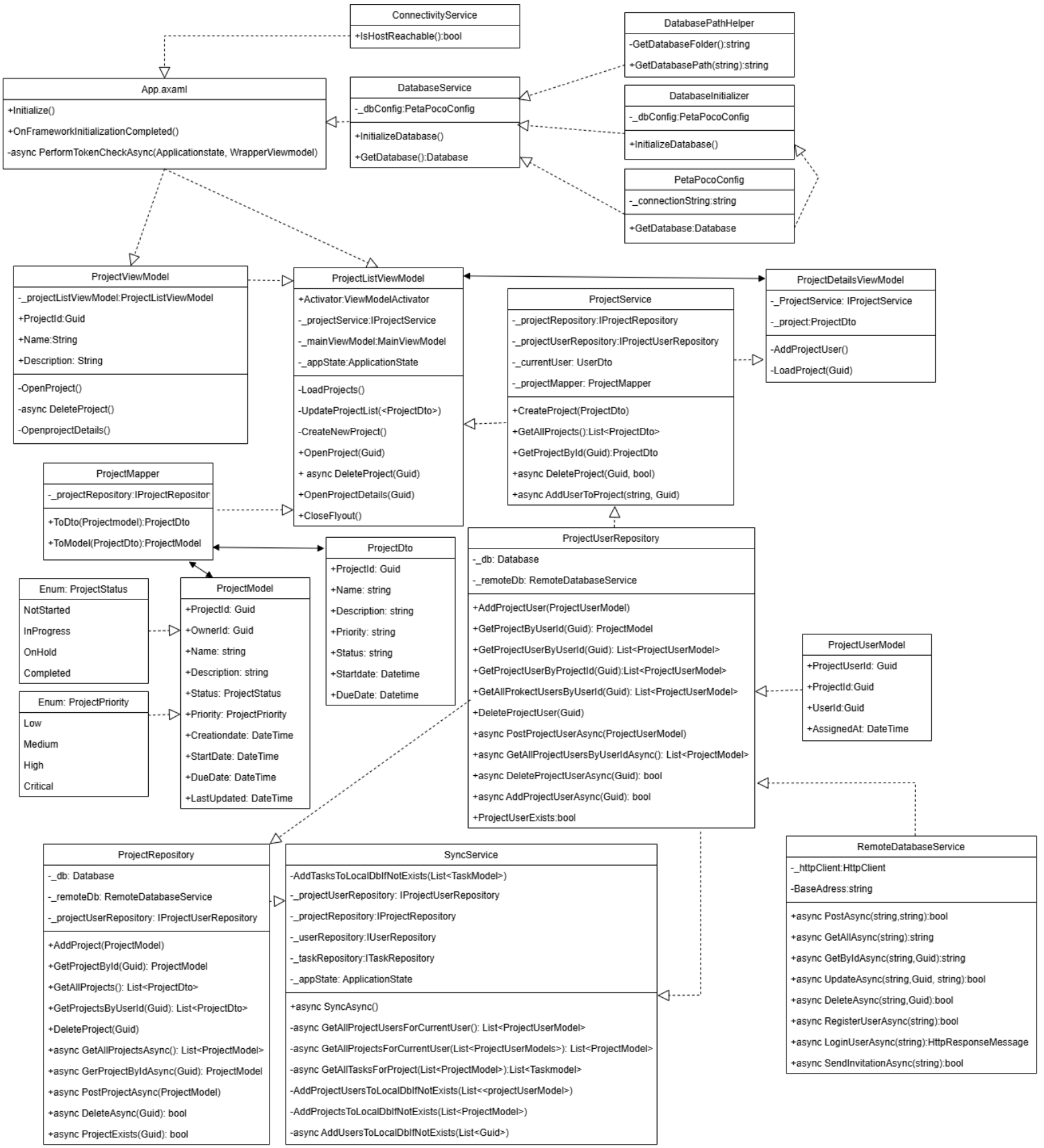
3.3 Architekturdiagramme

Komponentendiagramm

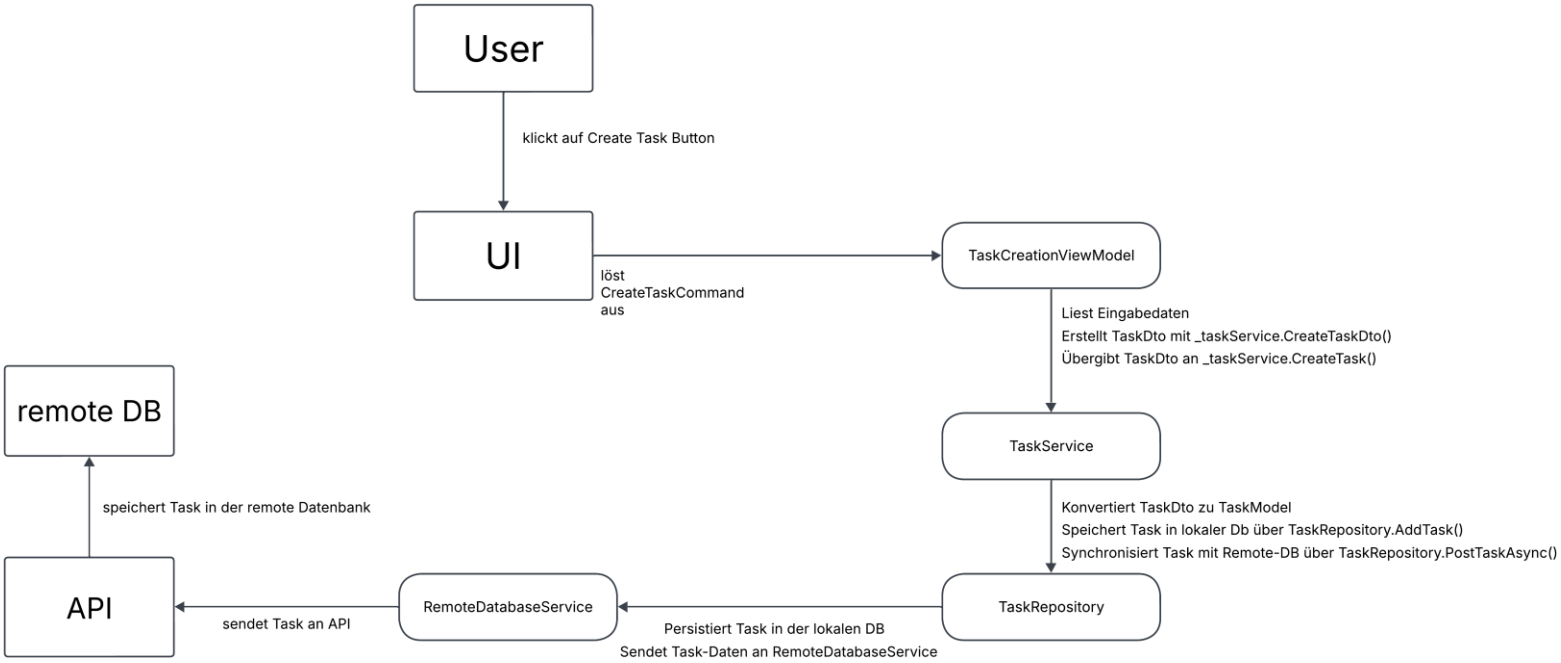


UML-Klassendiagramm

Um Unleserlichkeit zu vermeiden, sind im folgenden Diagramm nicht alle Klassen abgebildet. In unserem Fall haben einige Klassen ähnliche oder gleiche Methoden, die sich beispielsweise nur auf die Task-Prozesse beziehen. Hier sind primär die Methoden und Klassen abgebildet, die sich mit den Project-Prozessen beschäftigen. Dieselbe Struktur ist also auch auf die Task-Prozesse zu beziehen. Auch die Interfaces, die jede Repository- und Service-Klasse hat, sind aus Gründen der Leserlichkeit nicht abgebildet.


ConnectivityService
+IsHostReachable():bool
DatabasePathHelper
-GetDatabaseFolder():string
+GetDatabasePath(string):string
App.axaml
+Initialize()
+OnFrameworkInitializationCompleted()
-async PerformTokenCheckAsync(Applicationstate, WrapperViewmodel)
DatabaseService
-_dbConfig:PetaPocoConfig
+InitializeDatabase()
+GetDatabase():Database
DatabaseInitializer
-_dbConfig:PetaPocoConfig
+InitializeDatabase()
PetaPocoConfig
-_connectionString:string
+GetDatabase:Database
ProjectViewModel
-_projectListViewModel:ProjectListViewModel
+ProjectId:Guid
+Name:String
+Description: String
-OpenProject()
-async DeleteProject()
-OpenprojectDetails()
ProjectListViewModel
+Activator:ViewModelActivator
-_projectService:IProjectService
-_mainViewModel:MainViewModel
-_appState:ApplicationState
-LoadProjects()
-UpdateProjectList(<ProjectDto>)
-CreateNewProject()
+OpenProject(Guid)
+ async DeleteProject(Guid)
+OpenProjectDetails(Guid)
+CloseFlyout()
ProjectService
-_projectRepository:IProjectRepository
-_projectUserRepository:IProjectUserRepository
-_currentUser: UserDto
-_projectMapper: ProjectMapper
+CreateProject(ProjectDto)
+GetAllProjects():List<ProjectDto>
+GetProjectById(Guid):ProjectDto
+async DeleteProject(Guid, bool)
+async AddUserToProject(string, Guid)
ProjectDetailsViewModel
-_ProjectService: IProjectService
-_project:ProjectDto
-AddProjectUser()
-LoadProject(Guid)
ProjectMapper
-_projectRepository:IProjectRepositor
+ToDto(Projectmodel):ProjectDto
+ToModel(ProjectDto):ProjectModel
ProjectDto
+ProjectId: Guid
+Name: string
+Description: string
+Priority: string
+Status: string
+Startdate: Datetime
+DueDate: Datetime
Enum: ProjectStatus
NotStarted
InProgress
OnHold
Completed
Enum: ProjectPriority
Low
Medium
High
Critical
ProjectModel
+ProjectId: Guid
+OwnerId: Guid
+Name: string
+Description: string
+Status: ProjectStatus
+Priority: ProjectPriority
+Creationdate: DateTime
+StartDate: DateTime
+DueDate: DateTime
+LastUpdated: DateTime
ProjectUserRepository
-_db: Database
-_remoteDb: RemoteDatabaseService
+AddProjectUser(ProjectUserModel)
+GetProjectByUserId(Guid): ProjectModel
+GetProjectUserByUserId(Guid): List<ProjectUserModel>
+GetProjectUserByProjectId(Guid):List<ProjectUserModel>
+GetAllProkectUsersByUserId(Guid): List<ProjectUserModel>
+DeleteProjectUser(Guid)
+async PostProjectUserAsync(ProjectUserModel)
+async GetAllProjectUsersByUserIdAsync(): List<ProjectModel>
+async DeleteProjectUserAsync(Guid): bool
+async AddProjectUserAsync(Guid): bool
+ProjectUserExists:bool
ProjectUserModel
+ProjectUserId: Guid
+ProjectId:Guid
+UserId:Guid
+AssignedAt: DateTime
ProjectRepository
-_db: Database
-_remoteDb: RemoteDatabaseService
-_projectUserRepository: IProjectUserRepository
+AddProject(ProjectModel)
+GetProjectById(Guid): ProjectModel
+GetAllProjects(): List<ProjectDto>
+GetProjectsByUserId(Guid): List<ProjectDto>
+DeleteProject(Guid)
+async GetAllProjectsAsync(): List<ProjectModel>
+async GerProjectByIdAsync(Guid): ProjectModel
+async PostProjectAsync(ProjectModel)
+async DeleteAsync(Guid): bool
+async ProjectExists(Guid): bool
SyncService
-AddTasksToLocalDbIfNotExists(List<TaskModel>)
-_projectUserRepository: IProjectUserRepository
-_projectRepository:IProjectRepository
-_userRepository:IUserRepository
-_taskRepository:ITaskRepository
-_appState: ApplicationState
+async SyncAsync()
-async GetAllProjectUsersForCurrentUser(): List<ProjectUserModel>
-async GetAllProjectsForCurrentUser(List<ProjectUserModels>): List<ProjectModel>
-async GetAllTasksForProject(List<ProjectModel>):List<Taskmodel>
-AddProjectUsersToLocalDbIfNotExists(List<<projectUserModel>)
-AddProjectsToLocalDbIfNotExists(List<ProjectModel>)
-async AddUsersToLocalDbIfNotExists(List<Guid>)
RemoteDatabaseService
-_httpClient:HttpClient
-BaseAdress:string
+async PostAsync(string,string):bool
+async GetAllAsync(string):string
+async GetByIdAsync(string,Guid):string
+async UpdateAsync(string,Guid, string):bool
+async DeleteAsync(string,Guid):bool
+async RegisterUserAsync(string):bool
+async LoginUserAsync(string):HttpResponseMessage
+async SendInvitationAsync(string):bool

Ablauf einer Task-Creation



3.4 Clean Code Prinzipien

Um Clean Code sicherzustellen, haben wir einige Regeln aufgestellt, die sich mit der Benamung von Variablen, Klassen, Methoden und Datenbankentabellen beschäftigt.

Klassen, Methoden, Ordner/Projects

Benennung: Klassen und Ordner sollten immer in UpperCamelCase benannt werden.

- Beispiel: DatabaseInitializer, TaskModelFactory, DatabaseConfig

Variablen

Benennung Methodenparameter: sollten in camelCase benannt werden

- Beispiel: totalAmount, customerName, orderList

Benennung Lokale Variablen in Klassen (Parameter/Felder einer Klasse): sollten in UpperCamelCase benannt werden

- siehe oben

Sonderfall: privaten Variablen wird ein Unterstrich „_“ Vorangestellt

- Beispiel: _privateVariable, _amountOfMoney,...

UI-Objekte

Benennung: Name (wenn überhaupt nötig) sollte aus Initialen des Objekttypen und einer Beschreibung der Funktion des Objekts bestehen, getrennt durch einen Unterstrich „_“

- Beispiel: RB_PrioAuswahl (RadioButton, der für die Auswahl der Priorität zuständig ist)

UI-Klassen

Benennung Views: UpperCamelCase mit Anhängsel „View“ ganz am Ende

- Beispiel: KanbanView

Benennung ViewModel: UpperCamelCase mit Anhängsel „Viewmodel“ ganz am Ende (Ausnahme ViewModel)

- Beispiel: KanbanViewModel

Konstanten

Benennung: Konstanten sollten in Großbuchstaben mit Unterstrichen zwischen den Wörtern benannt werden.

- Beispiel: MAX_CHARACTERS, DEFAULT_PRIO

Interfaces

Benennung: Interfaces sollten in UpperCamelCase benannt werden und mit einem vorangestellten "I" beginnen.

- Beispiel: ILogger, IRandomInterface

Datenbanken und Tabellen

Benennung: Datenbanktabellen sollten in snake_case benannt werden, wobei jedes Wort durch einen Unterstrich getrennt ist.

- Beispiel: customer_orders, product_catalog

3.5 Architecture Decision Records (ADR)

1. Verwendung einer Schichtenarchitektur:

Entscheidung:

Wir verwenden eine Schichtenarchitektur, um eine klare Trennung der Verantwortlichkeiten und eine bessere Wartbarkeit des Codes zu gewährleisten.

Kontext:

tschiraplus soll modular und erweiterbar sein. Eine monolithische Struktur würde langfristig zu schwer wartbarem und unübersichtlichem Code führen.

Optionen:

1) Keine Schichtenarchitektur (Monolithisch):

- Vorteil: Einfachere Implementierung am Anfang
- Nachteil: Schlechte Wartbarkeit, starke Kopplung zwischen Komponenten

2) Drei-Schichten-Architektur (Presentation, Business, Data - gewählte Option):

- Vorteil: Klare Trennung von UI, Logik und Daten
- Nachteil: Erhöhter initialer Implementierungsaufwand

Gewählte Lösung:

Wir entscheiden uns für eine Drei-Schichten-Architektur mit folgenden Schichten:

- Data (Core, Data): Core enthält Domänenmodelle und Enums, Data verwaltet Datenbankoperationen
- Business (Services): Beinhaltet die Geschäftslogik
- Presentation (UI): Präsentationsschicht mit Views und ViewModels

Konsequenzen:

- Verbesserte Wartbarkeit und Testbarkeit
- Klare Trennung von Verantwortlichkeiten
- Leichter Overhead durch zusätzliche Schichten

2. Verwendung von MVVM für die UI:

Entscheidung:

Die UI wird nach dem Model-View-ViewModel (MVVM)-Muster entwickelt, um eine klare Trennung zwischen Logik und UI zu erreichen.

Kontext:

Das Projekt verwendet das Avalonia-Framework für die UI. Die UI-Schicht darf nicht direkt mit der Geschäftslogik oder den Datenmodellen interagieren, um eine lose Kopplung zu gewährleisten.

Optionen:

1) Keine Trennung (Code-Behind):

- Vorteil: Einfachere Implementierung
- Nachteil: Schlechte Wartbarkeit, erschwerte Testbarkeit

2) Model-View-Presenter (MVP):

- Vorteil: Gute Trennung von UI und Logik

- Nachteil: Erfordert mehr Boilerplate-Code als MVVM

3) Model-View-ViewModel (MVVM - gewählte Option):

- Vorteil: Bessere Trennung von UI und Logik, direkte Unterstützung in Avalonia
- Nachteil: Steilere Lernkurve für Anfänger

Gewählte Lösung:

MVVM wird umgesetzt mit:

- Model (Service-Klassen): Enthält die Datenmodelle und Logik
- View: UI-Elemente (axaml-Dateien)
- ViewModel: Enthält Bindings, Commands und Interaktionslogik

Konsequenzen:

- Bessere Testbarkeit durch Trennung der Logik
- Verbesserte Wartbarkeit und Skalierbarkeit
- Höherer initialer Aufwand für die Implementierung

3. Verwendung von C# als Programmiersprache:

Entscheidung:

C# wird als Hauptprogrammiersprache für das Projekt verwendet.

Kontext:

tschiraplus soll plattformübergreifend lauffähig sein und eine stabile, moderne Sprache nutzen, die gut mit UI-Frameworks (z. B. Avalonia, WPF) funktioniert.

Optionen:

1) C# - gewählte Option:

- Vorteil: Modern, stark typisiert, plattformübergreifende Entwicklung mit .NET
- Nachteil: Abhängigkeit von Microsofts Ökosystem

2) Java:

- Vorteil: Plattformunabhängig, große Community
- Nachteil: Schlechtere Integration mit .NET und Avalonia/WPF

3) Python:

- Vorteil: Einfachere Syntax, große Bibliotheken
- Nachteil: Schlechte Performance für UI-intensive Anwendungen

Gewählte Lösung:

Wir setzen auf C# mit .NET, weil:

- Gute Integration mit SQLite, PostgreSQL und UI-Frameworks
- Asynchrone Programmierung mit async/await erleichtert UI-Interaktion
- Hohe Performance im Vergleich zu dynamischen Sprachen

Konsequenzen:

- Starke Typisierung, erleichtert Fehlervermeidung
- Gute Performance und plattformübergreifende Möglichkeiten
- Abhängigkeit von Microsofts Tooling

4. Verwendung von DTOs zur Trennung von UI und Core:

Entscheidung:

Data Transfer Objects (DTOs) werden genutzt, um Daten zwischen den Schichten zu übertragen.

Kontext:

Die UI-Schicht kann nicht direkt auf die Core/Data-Schicht zugreifen. Um eine klare Trennung sicherzustellen, werden DTOs als Vermittler verwendet.

Optionen:

1) Direkter Zugriff auf die Domänenmodelle:

- Vorteil: Weniger Boilerplate-Code
- Nachteil: Enge Kopplung zwischen UI und Geschäftslogik

2) Data Transfer Objects (DTOs) - gewählte Option:

- Vorteil: Bessere Trennung von UI und Core, vermeidet direkte Kopplung
- Nachteil: Zusätzlicher Code für Mapping

Gewählte Lösung:

Wir nutzen manuelle DTO-Klassen, um Daten zwischen Services und UI zu transportieren. Mappings erfolgen explizit in der Services-Schicht.

Konsequenzen:

- Starke Entkopplung der UI von der Geschäftslogik
- Erhöhte Sicherheit durch Begrenzung der Felder, die an die UI gesendet werden
- Zusätzlicher Entwicklungsaufwand für DTOs und Mapping

4. Git-Management/Versionierung

4.1 Git-Workflow

Branching-Strategie (Git Flow)

- Git Flow ist ein zuverlässiges Modell für die Verwaltung von Branches. Durch Nutzung dieses Modelles trennen wir Entwicklung-, Feature- und Release-Branches was für Struktur und weniger Chaos sorgt.

Naming-Konventionen für Branches

- Einheitliche Namenskonventionen sorgen für mehr Übersicht und erleichtern die Zuordnung von Änderungen. Deshalb haben wir die Benennung der Feature-Branches bewusst einfach gehalten und am Zweck des Branches ausgerichtet.

Verlinkung der Anforderungen mit Issues und Commits

- Commits haben wir mit den Issues verknüpft, damit der Entwicklungsprozess nachvollziehbar bleibt und man weiß, was wozu gehört. Die Commit Messages selbst haben wir so kurz und knapp gehalten wie möglich, damit das spätere Nachvollziehen der Schritte erleichtert wird.

4.2 Tagging & Changelog

Versionsnummerierung und Tags

- Die Versionen haben wir nach dem Prinzip Semantic Versioning getaggt, um Änderungen klar zu kennzeichnen.

Pflege des Changelogs

Wir haben das Changelog so strukturiert, dass neue Features, Fixes und Änderungen klar ersichtlich sind und das Release-Management erleichtert wird.

4.3 Readme-Dokumentation

tschira+ - Projektplanungstool

Projektbeschreibung

tschira+ ist ein leistungsstarkes Projektplanungstool, das entwickelt wurde, um die Verwaltung und Organisation von Projektarbeiten effizienter zu gestalten. Mit dieser Software können Aufgaben einfach erstellt, zugewiesen, dokumentiert und deren Fortschritt verfolgt werden. Sie hilft dabei, den Überblick über alle Phasen eines Projekts zu behalten und fördert eine reibungslose Zusammenarbeit im Team.
Funktionen:

Aufgaben erstellen und verwalten: Erstellen von Aufgaben und Unteraufgaben, Festlegen von Fälligkeitsdaten und Prioritäten.

Zuweisung von Aufgaben: Aufgaben an Teammitglieder zuweisen, um klare Verantwortlichkeiten zu definieren.

Fortschrittsverfolgung: Echtzeit-Tracking des Fortschritts einzelner Aufgaben und des gesamten Projekts.

Aufgaben erstellen und verwalten: Erstellen von Aufgaben und Unteraufgaben, Festlegen von Fälligkeitsdaten und Prioritäten.

Zuweisung von Aufgaben: Aufgaben an Teammitglieder zuweisen, um klare Verantwortlichkeiten zu definieren.

Fortschrittsverfolgung: Echtzeit-Tracking des Fortschritts einzelner Aufgaben und des gesamten Projekts.

Technologie-Stack

Programmiersprache: C#

Framework: .NET

Datenbank: PetaPoco

Weitere Tools: Avalonia-UI;

Installation

Klone dieses Repository auf dein lokales System:

git clone https://github.com/Das-M-e-e/tschiraplus.git

Öffne das Projekt in Visual Studio oder deiner bevorzugten C#-IDE.

Stelle sicher, dass du .NET Core oder das .NET Framework installiert hast. Weitere Infos findest du in der offiziellen .NET-Website(https://dotnet.microsoft.com/en-us/).

Um die Anwendung zu starten führen sie die Program.cs im UI Projekt aus.

Verwendung

Um die Anwendung zu starten führen sie die Program.cs im UI Projekt (To build the app, run Program.cs in the UI Project.)

Erstelle und verwalte Aufgaben über die Benutzeroberfläche.

Verfolge den Fortschritt des Projekts in Echtzeit.

Lizenz

Dieses Projekt ist unter der MIT Lizenz lizenziert. Autoren:
Gabriel; Anna-Lenna; Yuting; Sophia; Franzi; Paulina; Isaak – Entwickler

5. Testing

5.1 Unit-Tests

Vorab sei gesagt, dass wir alle Prinzipien des **F.I.R.S.T.** Einhalten konnten bis auf das **T**imley. Wir haben die Tests erst nach Implementierung der Funktionalitäten erstellt.

Das Framework für unser Unit-Tests war MSTest, welches mit .NET kam. Die Test Klassen sind in der Form TestZuTestendeKlasse.cs Wir haben ein Testing-Projekt(d.i. Modul in c#) mit Namen **Testing** erstellt und für folgende Klassen Tests generiert und ausgeführt:

TestTaskService.cs

Hier wurde ursprünglich Mal die CreateTask Methode getestet, aber letzendlich wurde die Methode anders umgesetzt und schließlich auch der Test verworfen.

TestTaskSortingManager.cs:

Überblick

Die Klasse `TaskSortingManager` enthält Methoden, um eine Liste von `TaskDto`-Objekten nach verschiedenen Kriterien zu sortieren und zu filtern. Diese Tests überprüfen die Funktionalität der Methoden `SortBySingleAttribute` und `FilterByPredicate`.

Die `TaskDto`-Klasse enthält folgende Eigenschaften:

• `TaskId`: Ein eindeutiger Bezeichner für die Aufgabe.

• `Title`: Der Titel der Aufgabe.

• `StartDate`: Das Startdatum der Aufgabe.

• `DueDate`: Das Fälligkeitsdatum der Aufgabe (kann null sein).

• `Status`: Der Status der Aufgabe (z. B. "InProgress", "Ready").

Testmethoden

1. SortBySingleAttribute_SortsByTitleAscending

Zweck: Überprüft, ob die SortBySingleAttribute-Methode die Liste der Aufgaben nach dem Title in aufsteigender Reihenfolge sortiert.

Vorbedingungen: Eine Liste von Aufgaben mit den Titeln "B Task", "A Task" und "C Task".

Erwartetes Ergebnis:

• Die Aufgaben sind nach dem Titel in aufsteigender Reihenfolge sortiert:

1. "A Task"
2. "B Task"
3. "C Task"

2. SortBySingleAttribute_SortsByCreationDateAscending

Zweck: Überprüft, ob die `SortBySingleAttribute`-Methode die Liste der Aufgaben nach dem `StartDate` in aufsteigender Reihenfolge sortiert.

Vorbedingungen: Eine Liste von Aufgaben mit den `Startdaten` "2024-01-05", "2024-01-10" und "2024-01-15".

Erwartetes Ergebnis:

• Die Aufgaben sind nach dem `StartDate` in aufsteigender Reihenfolge sortiert:

1. "2024-01-05"
2. "2024-01-10"
3. "2024-01-15"

3. SortBySingleAttribute_SortsByDueDate_WithNullValues

Zweck: Überprüft, ob die `SortBySingleAttribute`-Methode die Liste der Aufgaben nach dem `DueDate` in aufsteigender Reihenfolge sortiert, wobei `null`-Werte am Ende behandelt werden.

Vorbedingungen: Eine Liste von Aufgaben mit den Fälligkeitsdaten "2024-02-01", "2024-03-01" und `null` für `DueDate`.

Erwartetes Ergebnis:

• Die Aufgaben sind nach dem `DueDate` in aufsteigender Reihenfolge sortiert:

1. "2024-02-01"
2. "2024-03-01"
3. `null` (am Ende der Liste)

4. SortBySingleAttribute_EmptyList_ReturnsEmptyList

Zweck: Überprüft, ob die `SortBySingleAttribute`-Methode eine leere Liste korrekt zurückgibt.

Vorbedingungen: Eine leere Liste von Aufgaben.

Erwartetes Ergebnis:

• Eine leere Liste wird zurückgegeben.

5. SortBySingleAttribute_SingleElementList_ReturnsSameList

Zweck: Überprüft, ob die `SortBySingleAttribute`-Methode eine Liste mit nur einem Element unverändert zurückgibt.

Vorbedingungen: Eine Liste mit einem einzigen `TaskDto`.

Erwartetes Ergebnis:

- Die Liste mit einem einzigen Element wird unverändert zurückgegeben.

6. FilterByPredicate_FiltersByStatus

Zweck: Überprüft, ob die `FilterByPredicate`-Methode die Aufgaben korrekt nach dem `Status` filtert.

Vorbedingungen: Eine Liste von Aufgaben mit verschiedenen Statuswerten, z. B. "InProgress" und "Ready".

Erwartetes Ergebnis:

- Die gefilterte Liste enthält nur Aufgaben mit dem `Status` "InProgress".

7. FilterByPredicate_FiltersByCreationDate

Zweck: Überprüft, ob die `FilterByPredicate`-Methode die Aufgaben korrekt nach dem `StartDate` filtert.

Vorbedingungen: Eine Liste von Aufgaben mit unterschiedlichen `StartDate`–Werten.

Erwartetes Ergebnis:

- Die gefilterte Liste enthält nur Aufgaben, deren `StartDate` nach dem 7. Januar 2024 liegt.

8. FilterByPredicate_EmptyList_ReturnsEmptyList

Zweck: Überprüft, ob die `FilterByPredicate`-Methode eine leere Liste korrekt zurückgibt, wenn nach einem bestimmten Kriterium gefiltert wird.

Vorbedingungen: Eine leere Liste von Aufgaben.

Erwartetes Ergebnis:

- Eine leere Liste wird zurückgegeben.

9. FilterByPredicate_NoMatchingElements_ReturnsEmptyList

Zweck: Überprüft, ob die `FilterByPredicate`-Methode eine leere Liste zurückgibt, wenn keine Elemente den Filterkriterien entsprechen.

Vorbedingungen: Eine Liste von Aufgaben und ein Filter, der keine übereinstimmenden Elemente findet.

Erwartetes Ergebnis:

- Eine leere Liste wird zurückgegeben.

10. FilterByPredicate_AllMatchingElements_ReturnsSameList

Zweck: Überprüft, ob die `FilterByPredicate`-Methode die gesamte Liste zurückgibt, wenn alle Elemente den Filterkriterien entsprechen.

Vorbedingungen: Eine Liste von Aufgaben und ein Filter, der für alle Aufgaben zutrifft.

Erwartetes Ergebnis:

- Die gesamte Liste wird unverändert zurückgegeben.

Zusammenfassung der Testmethoden

• **Sortiermethoden** (`SortBySingleAttribute`) wurden getestet, um sicherzustellen, dass Aufgaben nach verschiedenen Attributen wie Title, `StartDate` und `DueDate` korrekt sortiert werden, auch wenn null-Werte vorhanden sind.

• **Filtermethoden** (`FilterByPredicate`) wurden getestet, um sicherzustellen, dass Aufgaben basierend auf bestimmten Bedingungen (wie Status oder `StartDate`) korrekt gefiltert werden.

Alle Tests stellen sicher, dass die Methoden unter verschiedenen Szenarien wie leeren Listen, fehlenden Werten und nicht übereinstimmenden Filterbedingungen korrekt funktionieren.

TestTokenStorageService.cs:

Überblick

Die Klasse `TokenStorageService` bietet Funktionen zum Speichern, Laden und Entfernen von Authentifizierungstoken in einer Datei. Diese Tests stellen sicher, dass die Methoden des `TokenStorageService` korrekt arbeiten, insbesondere in Bezug auf die Speicherung, das Laden, die Verschlüsselung und das Entfernen von Tokens.

Die Testklasse überprüft die Funktionalität der folgenden Methoden:

• **SaveToken(string token)**: Speichert ein Authentifizierungstoken in einer Datei.

• **LoadToken()**: Lädt das gespeicherte Authentifizierungstoken aus der Datei.

• **RemoveToken()**: Entfernt das gespeicherte Authentifizierungstoken und löscht die Datei.

Testmethoden

1. SaveToken_LoadToken_ReturnsSameToken

Zweck: Überprüft, ob das gespeicherte Token korrekt geladen wird und das geladene Token mit dem ursprünglichen Token übereinstimmt.

Vorbedingungen: Das `SaveToken` wird mit einem Token (z. B. `ey123.fake.jwt.token`) aufgerufen.

Erwartetes Ergebnis:

- Das geladene Token entspricht genau dem ursprünglich gespeicherten Token.
- Das Token ist nicht null.

2. SaveToken_RemoveToken_FileShouldBeDeleted

Zweck: Überprüft, ob die Datei nach dem Entfernen des Tokens mit der Methode `RemoveToken` gelöscht wird.

Vorbedingungen: Ein Token wird mithilfe von `SaveToken` gespeichert.

Erwartetes Ergebnis:

- Nach dem Aufruf von `RemoveToken` sollte die Datei mit dem gespeicherten Token nicht mehr existieren.

3. LoadToken_WhenFileDoesNotExist_ReturnsNull

Zweck: Überprüft, ob die Methode `LoadToken` null zurückgibt, wenn die Datei mit dem Token nicht existiert.

Vorbedingungen: Es gibt keine gespeicherte Datei für das Token.

Erwartetes Ergebnis:

- Die Methode `LoadToken` gibt null zurück.

4. SaveToken_FileContentIsEncrypted

Zweck: Überprüft, ob der gespeicherte Token-Inhalt in der Datei verschlüsselt und nicht im Klartext gespeichert wird.

Vorbedingungen: Ein Token (z. B. `SensitiveToken123`) wird mit der Methode `SaveToken` gespeichert.

Erwartetes Ergebnis:

- Der Klartext des Tokens wird nicht direkt im gespeicherten Dateiinhalt gefunden.
- Der gespeicherte Inhalt sollte verschlüsselt oder anderweitig gesichert sein, sodass der Token nicht im Klartext vorliegt.

5. SaveToken_OverwritePreviousToken_NewTokenIsSaved

Zweck: Überprüft, ob ein bereits gespeichertes Token überschrieben wird, wenn ein neues Token mit `SaveToken` gespeichert wird.

Vorbedingungen: Zwei verschiedene Tokens werden in der Reihenfolge `FirstToken` und `SecondToken` gespeichert.

Erwartetes Ergebnis:

• Das zuletzt gespeicherte Token (`SecondToken`) wird korrekt geladen, wenn `LoadToken` aufgerufen wird.

• Das zuerst gespeicherte Token (`FirstToken`) sollte nicht mehr geladen werden.

Zusammenfassung der Testmethoden

• **Speicher- und Ladeoperationen**: Es wird überprüft, ob das Token korrekt gespeichert und wieder geladen wird, sowohl in normalen Fällen als auch wenn die Datei nicht existiert.

• **Token-Entfernung**: Es wird getestet, ob die Datei nach dem Entfernen des Tokens korrekt gelöscht wird.

• **Sicherheit der Speicherung**: Es wird getestet, ob das Token im Speicher (Datei) sicher und verschlüsselt ist, um zu verhindern, dass es im Klartext gespeichert wird.

• **Überschreibung von Tokens**: Es wird geprüft, ob das neue Token das alte korrekt überschreibt und bei einem erneuten Laden das zuletzt gespeicherte Token zurückgegeben wird.

Die Tests stellen sicher, dass der `TokenStorageService` ordnungsgemäß funktioniert und dass

das gespeicherte Token sowohl sicher als auch korrekt gehandhabt wird.

TestUserInputParser.cs:

Überblick

Die Klasse `UserInputParser` dient dazu, Benutzereingaben zu analysieren und in eine Liste von Befehlen zu übersetzen. Diese Befehle können verschiedene Operationen wie Sortieren und Filtern umfassen, die durch die Eingabe des Benutzers spezifiziert werden. Die Tests überprüfen, ob die Eingaben korrekt geparst werden und ob ungültige Eingaben entsprechend ignoriert oder behandelt werden.

Die `CommandType`-Enumeration enthält verschiedene Befehlsarten, wie `Sort` und `Filter`.

Testmethoden

1. Parse_EmptyInput_ReturnsEmptyList

Zweck: Überprüft, ob der Parser eine leere Eingabe korrekt als leere Liste zurückgibt.

Vorbedingungen: Der Parser wird mit einer leeren Zeichenkette ("") aufgerufen.

Erwartetes Ergebnis:

• Die Rückgabe ist eine leere Liste, da keine Befehle zum Parsen vorhanden sind.

2. Parse_WhitespaceInput_ReturnsEmptyList

Zweck: Überprüft, ob der Parser eine Eingabe, die nur aus Leerzeichen besteht, als leere Liste

zurückgibt.

Vorbedingungen: Der Parser wird mit einer Eingabe bestehend aus Leerzeichen (" ") aufgerufen.

Erwartetes Ergebnis:

- Die Rückgabe ist ebenfalls eine leere Liste, da keine gültigen Befehle vorhanden sind.

3. Parse_ValidSortCommand_ReturnsSortCommand

Zweck: Überprüft, ob der Parser einen gültigen `Sort`-Befehl korrekt erkennt und die entsprechenden Parameter zurückgibt.

Vorbedingungen: Der Parser wird mit einer Eingabe im Format "`sort:Titel`" aufgerufen.

Erwartetes Ergebnis:

- Die Rückgabe ist eine Liste mit einem `Sort`-Befehl, der den Parameter "Titel" hat.

4. Parse_ValidFilterCommand_ReturnsFilterCommand

Zweck: Überprüft, ob der Parser einen gültigen Filter-Befehl korrekt erkennt und die entsprechenden Parameter zurückgibt.

Vorbedingungen: Der Parser wird mit einer Eingabe im Format "`filter`:`Status` Offen" aufgerufen.

Erwartetes Ergebnis:

- Die Rückgabe ist eine Liste mit einem Filter-Befehl, der den Parameter "Status Offen" hat.

5. Parse_MultipleValidCommands_ReturnsAllCommands

Zweck: Überprüft, ob der Parser mehrere gültige Befehle korrekt verarbeitet und alle Befehle in der richtigen Reihenfolge zurückgibt.

Vorbedingungen: Der Parser wird mit einer Eingabe im Format "`sort`:`Titel`;`filter`:`Status` Offen" aufgerufen.

Erwartetes Ergebnis:

- Die Rückgabe ist eine Liste mit zwei Befehlen: einem `Sort`-Befehl und einem `FilterBefehl` mit den entsprechenden Parametern.

6. Parse_CommandWithExtraWhitespace_ParsesCorrectly

Zweck: Überprüft, ob der Parser korrekt mit zusätzlichen Leerzeichen in den Befehlen umgeht und die Eingabe korrekt verarbeitet.

Vorbedingungen: Der Parser wird mit einer Eingabe im Format " `sort:Titel` " aufgerufen.

Erwartetes Ergebnis:

- Die Rückgabe ist eine Liste mit einem `Sort`-Befehl und dem Parameter "Titel", wobei zusätzliche Leerzeichen vor und nach dem Befehl ignoriert werden.

7. Parse_InvalidCommand_Ignored

Zweck: Überprüft, ob der Parser ungültige Befehle, die nicht zu einem bekannten `CommandType` gehören, korrekt ignoriert.

Vorbedingungen: Der Parser wird mit einer ungültigen Eingabe im Format "`invalid:Titel`" aufgerufen.

Erwartetes Ergebnis:

- Die Rückgabe ist eine leere Liste, da der ungültige Befehl ignoriert wird.

8. Parse_InvalidCommandStructure_Ignored

Zweck: Überprüft, ob der Parser ungültige Befehlsstrukturen (z. B. ein `sort`-Befehl ohne Parameter) korrekt ignoriert.

Vorbedingungen: Der Parser wird mit einer Eingabe im Format "`sort`:" aufgerufen.

Erwartetes Ergebnis:

- Die Rückgabe ist eine leere Liste, da der Befehl aufgrund des fehlenden Parameters ungültig ist.

9. Parse_MixedValidAndInvalidCommands_OnlyValidProcessed

Zweck: Überprüft, ob der Parser gemischte Eingaben (gültige und ungültige Befehle) korrekt verarbeitet und nur die gültigen Befehle zurückgibt.

Vorbedingungen: Der Parser wird mit einer Eingabe im Format "`sort:Titel;invalid:Test;filter:Status` Offen" aufgerufen.

Erwartetes Ergebnis:

- Die Rückgabe enthält nur zwei gültige Befehle: einen `Sort`-Befehl und einen `FilterBefehl`. Der ungültige Befehl wird ignoriert.

10. Parse_DoubleSemicolons_ParsesCorrectly

Zweck: Überprüft, ob der Parser mit doppelten Semikolons (z. B. durch falsche Eingabe) korrekt umgeht und die Befehle richtig parst.

Vorbedingungen: Der Parser wird mit einer Eingabe im Format "`sort:Titel;;filter:Status` Offen" aufgerufen.

Erwartetes Ergebnis:

- Die Rückgabe enthält zwei Befehle: einen `Sort`-Befehl und einen Filter-Befehl, wobei das doppelte Semikolon ignoriert wird.

Zusammenfassung der Testmethoden

• **Leer- und Whitespace-Eingaben**: Es wird getestet, ob der Parser leere Eingaben und solche mit nur Leerzeichen korrekt als leere Listen behandelt.

• **Gültige Befehle**: Die Tests überprüfen, ob der Parser die Befehle `sort` und `filter` korrekt verarbeitet, sowohl einzeln als auch in Kombination.

• **Ungültige Befehle**: Es wird überprüft, ob ungültige oder fehlerhafte Befehle (z. B. fehlende Parameter oder unbekannte Befehle) korrekt ignoriert werden.

• **Doppelte Semikolons und zusätzliche Leerzeichen**: Der Parser wird auf seine Fähigkeit getestet, mit doppelten Semikolons und überflüssigen Leerzeichen in der Eingabe umzugehen.

Die Tests stellen sicher, dass der `UserInputParser` auch bei gemischten, ungültigen oder fehlerhaften Eingaben stabil bleibt und nur gültige Befehle verarbeitet.

5.2 Integrationstests

Hier habe ich einfach nur geprüft, ob die Datenbank die Anfragen richtig bearbeitet und habe mir speziell die Task-Modell/Tabellen geprüft. Dafür musste ich Die Klasse `TaskRepository` anschauen.

Die Klasse `TaskRepository` bietet Funktionen zum Hinzufügen, Abrufen, Aktualisieren und Löschen von Aufgaben (`TaskModel`) in einer SQLite-Datenbank. Diese Integrationstests überprüfen die Interaktion zwischen der `TaskRepository`-Klasse und einer echten Datenbankinstanz, um sicherzustellen, dass die Repository-Methoden korrekt mit der Datenbank kommunizieren.

Die Tests verwenden eine in RAM gespeicherte SQLite-Datenbank, die für jeden Test dynamisch erstellt wird. So können schnelle und isolierte Tests durchgeführt werden, ohne die Notwendigkeit für eine persistente Datenbankverbindung.

Testmethoden

1. AddTask_ShouldInsertTask

Zweck: Überprüft, ob eine Aufgabe korrekt in die Datenbank eingefügt wird.

Vorbedingungen:

• Eine `TaskModel`-Instanz wird erstellt und an das Repository übergeben.

Testablauf:

• Der Test erstellt eine neue Aufgabe und fügt diese über die `AddTask`-Methode in die Datenbank ein.

Erwartetes Ergebnis:

• Die eingefügte Aufgabe wird korrekt in der Datenbank gespeichert und kann mit der `GetTaskById`-Methode abgerufen werden.

• Es wird überprüft, ob die Eigenschaften der Aufgabe übereinstimmen (z. B. `TaskId`, `Title`, `Description`).

2. GetTaskById_ShouldReturnTask

Zweck: Überprüft, ob eine Aufgabe korrekt aus der Datenbank abgerufen wird.

Vorbedingungen:

• Eine `TaskModel`-Instanz wird in die Datenbank eingefügt.

Testablauf:

• Nachdem eine Aufgabe hinzugefügt wurde, wird sie mit der `GetTaskById`-Methode abgerufen.

Erwartetes Ergebnis:

• Die Aufgabe wird erfolgreich aus der Datenbank geladen.

• Der `TaskId` der geladenen Aufgabe sollte mit dem `TaskId` der ursprünglich hinzugefügten Aufgabe übereinstimmen.

3. GetTasksByProjectId_ShouldReturnTasks

Zweck: Überprüft, ob Aufgaben korrekt nach `ProjectId` abgerufen werden.

Vorbedingungen:

• Zwei Aufgaben werden für dasselbe Projekt hinzugefügt.

Testablauf:

• Zwei Aufgaben mit dem gleichen `ProjectId` werden in die Datenbank eingefügt.

• Anschließend wird mit der Methode `GetTasksByProjectId` nach Aufgaben mit diesem `ProjectId` gesucht.

Erwartetes Ergebnis:

- Beide Aufgaben werden korrekt zurückgegeben.
- Die Anzahl der zurückgegebenen Aufgaben sollte 2 betragen.

4. UpdateTask_ShouldUpdateExistingTask

Zweck: Überprüft, ob eine bestehende Aufgabe erfolgreich aktualisiert wird.

Vorbedingungen:

- Eine Aufgabe wird in die Datenbank eingefügt.

Testablauf:

- Die Aufgabe wird zunächst mit bestimmten Werten hinzugefügt.
- Anschließend werden die Werte der Aufgabe geändert (z. B. Title, Description) und mit der `UpdateTask`-Methode aktualisiert.

Erwartetes Ergebnis:

- Die aktualisierte Aufgabe wird korrekt gespeichert.
- Beim Abrufen der Aufgabe nach der Aktualisierung sollten die geänderten Eigenschaften wie Title und Description den neuen Werten entsprechen.

5. DeleteTask_ShouldDeleteTask

Zweck: Überprüft, ob eine Aufgabe korrekt aus der Datenbank gelöscht wird.

Vorbedingungen:

- Eine Aufgabe wird in die Datenbank eingefügt.

Testablauf:

- Die Aufgabe wird mit der `DeleteTask`-Methode aus der Datenbank gelöscht.
- Danach wird versucht, die gelöschte Aufgabe mit der `GetTaskById`-Methode abzurufen.

Erwartetes Ergebnis:

- Die gelöschte Aufgabe ist nicht mehr in der Datenbank vorhanden.
- Die Rückgabe der `GetTaskById`-Methode sollte null sein, da die Aufgabe gelöscht wurde.

Testablauf

Testumgebung:

- **Datenbank**: SQLite im RAM (wird für jeden Test neu erstellt).

• **Datenmodell**: Die `Tasks`-Tabelle wird bei jedem Test in der Datenbank erstellt.

• **Testdaten**: Es werden dynamisch generierte `Guid`-Werte verwendet, um die Aufgaben zu erstellen, was sicherstellt, dass jede Aufgabe eindeutig ist.

Teststruktur:

• **TestInitialize**: Vor jedem Test wird eine neue SQLite-Datenbankverbindung hergestellt, und die Tabelle `Tasks` wird erstellt.

• **TestCleanup**: Nach jedem Test wird die Tabelle `Tasks` geleert und die Datenbankdatei gelöscht, um sicherzustellen, dass jeder Test in einer sauberen Umgebung läuft.

Ergebnis:

• **Bestätigung der Funktionalität**: Alle Tests bestätigen, dass die Methoden des `TaskRepository` korrekt arbeiten, indem sie Daten in der SQLite-Datenbank speichern, abrufen, aktualisieren und löschen.

• **Datenintegrität**: Alle Tests stellen sicher, dass die Daten korrekt gespeichert und verarbeitet werden, sodass die Integrität der Daten im Repository gewahrt bleibt.

Zusammenfassung der Testmethoden

Die Integrationstests des `TaskRepository` überprüfen, ob die grundlegenden CRUDOperationen (Erstellen, Lesen, Aktualisieren, Löschen) auf der SQLite-Datenbank korrekt durchgeführt werden. Die Tests bestätigen, dass das Repository:

• Aufgaben korrekt hinzufügt und abruft,

• Aufgaben nach einer Aktualisierung richtig speichert,

• Aufgaben erfolgreich löscht,

• Aufgaben korrekt nach `ProjectId` abruft.

Durch den Einsatz einer in RAM gespeicherten SQLite-Datenbank können die Tests schnell und isoliert durchgeführt werden, was die Entwicklung und das Testen effizient macht.

Fazit

Schließlich sei auch gesagt, dass längst nicht alle Funktionalitäten ausreichend getestet wurden (so wurde z.B. die UI und ihre Controller vernachlässigt), da wir die Test erst gegen Ende der Projektabgagen erstellt haben. Dementsprechend haben wir uns nur auf die Wichtigsten Klassen in Service beschränkt.

6. CI/CD-Pipeline

6.1 Aufbau der Pipeline

Überblick

Diese CI/CD-Pipeline wird über **GitHub Actions** verwaltet und dient der Automatisierung des Build-, Test- und Integrationsprozesses für ein .NET-Projekt. Sie sorgt dafür, dass bei jeder Änderung des Quellcodes, die in das Repository gepusht wird, der Code gebaut, getestet und validiert wird, um die Integrität des Projekts sicherzustellen. Diese Pipeline wird automatisch ausgelöst, wenn Code in den develop-Branch gepusht wird oder wenn ein Pull-Request zu diesem Branch gestellt wird.

Die Pipeline führt mehrere Schritte aus, um sicherzustellen, dass der Code den Standards entspricht und keine Fehler auftreten.

Workflow-Details

Name des Workflows

- **.NET**: Der Name dieses Workflows in GitHub Actions.

Auslöser des Workflows

Der Workflow wird unter folgenden Bedingungen ausgelöst:

1. **Push in den** develop**-Branch:**
    - Wenn Änderungen an Dateien mit den Erweiterungen `.cs`, `.csproj` oder an der Datei `tschiraplus`/tschiraplus.sln vorgenommen werden, wird der Workflow automatisch ausgelöst.
2. **Pull-Request an den** develop**-Branch:**
    - Der Workflow wird auch ausgeführt, wenn ein Pull-Request für den `develop`-Branch geöffnet wird.
3. **Manuelle Auslösung (Workflow Dispatch):**
    - Der Workflow kann manuell über die GitHub Actions UI ausgelöst werden, unabhängig von den Push- oder Pull-Request-Änderungen.

Workflow-Job: `build`

Der Job `build` führt die notwendigen Schritte aus, um das Projekt zu bauen und zu testen. Er wird auf einer **Ubuntu-Latest**-Virtuellen Maschine ausgeführt.

Schritte im Build-Job

1. **Checkout des Codes**
    - `uses: actions/checkout@v4`
    - Dieser Schritt prüft den Code aus dem GitHub-Repository aus, um sicherzustellen, dass die neueste Version des Codes für die folgenden Schritte verfügbar ist.
2. **Setup von .NET**
    - `uses: actions/setup-dotnet@v4`
    - In diesem Schritt wird die benötigte Version von .NET auf der Build-Maschine installiert. Hier wird **.NET 8.0.x** verwendet.
    - Es wird die neueste Version von .NET 8 installiert, die für die folgenden Schritte (wie das Bauen und Testen des Projekts) erforderlich ist.
3. **Wiederherstellen der Abhängigkeiten**
    - `run: dotnet restore ./tschiraplus/tschiraplus.sln`
    - Dieser Schritt stellt alle NuGet-Pakete und Abhängigkeiten wieder her, die im Projekt definiert sind. Dabei wird die tschiraplus.sln-Lösung verwendet, um sicherzustellen, dass alle Abhängigkeiten korrekt aufgelöst werden.
4. **Bauen des Projekts**
    - `run: dotnet build ./tschiraplus/tschiraplus.sln --no-restore`
    - Dieser Schritt baut die Lösung (.sln-Datei), wobei die Option --no-restore bedeutet, dass der vorherige Wiederherstellungs-Schritt nicht erneut ausgeführt wird, sondern der Fokus auf dem tatsächlichen Build liegt.
    - Falls der Build fehlschlägt, wird der Workflow hier stoppen und der Fehlerbericht wird angezeigt.
5. **Ausführen der Tests**
    - `run: dotnet test ./tschiraplus/tschiraplus.sln --no-build --verbosity normal`
    - In diesem Schritt werden die Tests des Projekts ausgeführt. Die Option --no-build stellt sicher, dass nur die Tests ausgeführt werden, ohne dass der Build-Prozess erneut gestartet wird (da der Build bereits in einem vorherigen Schritt durchgeführt wurde).
    - Die `--verbosity` normal-Option stellt sicher, dass die Test-Ergebnisse in einem detaillierten, aber gut strukturierten Format angezeigt werden.

- Wenn ein Test fehlschlägt, wird der gesamte Workflow fehlschlagen und die Fehlerdetails werden angezeigt.

Pipeline-Durchlauf

1. **Code-Änderung (Push oder Pull-Request)**
    - Der Workflow wird ausgelöst, sobald ein Commit in den `develop`-Branch gepusht wird oder ein Pull-Request zum `develop`-Branch geöffnet wird.
2. **Checkout des Repositories**
    - Der aktuelle Stand des `Repositories` wird auf der GitHub-Action-Maschine überprüft, damit die neuesten Änderungen getestet werden können.
3. **Installation von .NET 8**
    - .NET wird auf der Maschine eingerichtet, um das Projekt in der benötigten Version zu bauen und zu testen.
4. **Wiederherstellung von Abhängigkeiten**
    - NuGet-Pakete werden wiederhergestellt, um sicherzustellen, dass alle benötigten Abhängigkeiten für den Build und die Tests vorhanden sind.
5. **Projekt-Build**
    - Das Projekt wird gebaut, wobei alle Projektdateien (einschließlich der Lösung) auf Fehler geprüft werden.
6. **Testausführung**
    - Alle Unit-Tests werden ausgeführt, um die Funktionalität des Projekts zu überprüfen. Bei einem Fehler wird der Workflow abgebrochen.

Ergebnisse

- **Erfolgreicher Workflow**: Wenn alle Schritte ohne Fehler abgeschlossen werden, zeigt der Workflow, dass der Build und die Tests erfolgreich waren.
- **Fehlerhafte Tests oder Builds**: Wenn der Build oder ein Test fehlschlägt, wird der Workflow mit einem Fehlerstatus beendet. Details zu den Fehlern werden in den Logs angezeigt, um das Problem zu beheben.

Zusammenfassung

Diese GitHub Actions CI/CD-Pipeline sorgt dafür, dass bei jeder Änderung im develop-Branch oder bei jedem Pull-Request der Code automatisch gebaut und getestet wird. Sie stellt sicher, dass Fehler frühzeitig erkannt und behoben werden, sodass die Qualität und Stabilität des Projekts gewährleistet bleiben.

7.Persistenz

7.1 Systembeschreibung der Datenbank

Diese Datenbank dient zur Verwaltung von Benutzern, Projekten, Aufgaben, Sprints, Kommentaren, Benachrichtigungen und weiteren relevanten Entitäten für ein Projektmanagement-System. Die Struktur gewährleistet eine effiziente Organisation von Aufgaben und Ressourcen innerhalb von Projekten und erleichtert die Kommunikation und Zusammenarbeit zwischen Benutzern. Die Benutzerverwaltung erfolgt über die Users-Tabelle, in der grundlegende Informationen wie Benutzername, E-Mail und Profilinformationen gespeichert werden. Weitere Einstellungen werden in der UserSettings-Tabelle verwaltet, während Benutzerbeziehungen in der UserFriends-Tabelle abgebildet werden.

Projekte werden in der Projects-Tabelle gespeichert und erlauben eine strukturierte Verwaltung von Aufgaben und Sprints innerhalb eines Projekts. Die Sprints-Tabelle gruppiert Aufgaben in zeitlich begrenzte Abschnitte, während die Tasks-Tabelle spezifische Aufgaben eines Projekts oder Sprints speichert. Mithilfe der TaskTags-Tabelle können Aufgaben mit Tags versehen und durch die UserTaskAssignments-Tabelle einzelnen Benutzern zugewiesen werden.

Die Kommunikation und Zusammenarbeit der Benutzer erfolgt über Kommentare und Anhänge. Die Comments-Tabelle ermöglicht das Hinterlassen von Kommentaren, während die Attachments-Tabelle das Hochladen und Verwalten von Dateien innerhalb von Projekten und Aufgaben erlaubt. Die Notifications-Tabelle informiert Benutzer über relevante Ereignisse und sorgt für eine effiziente Benachrichtigung.

Um eine differenzierte Rollen- und Zugriffsverwaltung sicherzustellen, werden Benutzer über die ProjectUsers-Tabelle mit Projekten verknüpft. Die UserProjectRoles-Tabelle weist Benutzern spezifische Rollen innerhalb eines Projekts zu. Die Organisation von Projekten und Aufgaben wird durch die Nutzung von Tags unterstützt. In der Tags-Tabelle werden verschiedene Tags gespeichert, die über die TaskTags-Tabelle mit spezifischen Aufgaben verknüpft werden können.

Die Datenbankstruktur stellt sicher, dass jeder Benutzer mehrere Projekte erstellen oder daran teilnehmen kann. Ein Projekt kann mehrere Sprints enthalten, die wiederum mehrere Aufgaben beinhalten. Aufgaben können mit Tags versehen und mehreren Benutzern zugewiesen werden. Kommentare und Anhänge erleichtern die Kommunikation, während Benachrichtigungen über wichtige Ereignisse informieren. Durch die UserProjectRoles-Tabelle wird eine differenzierte Rechtevergabe sichergestellt. Diese durchdachte Struktur ermöglicht eine flexible und skalierbare Verwaltung von Projekten, Aufgaben und Benutzerinteraktionen.

7.2 CRUD-Operationen

PetaPoco erstellt die SQL Befehle hier ein Beispiel:
public void AddTask(TaskModel task)

```
{
 _db.Insert("Tasks", "TaskId", task);

 }
```

Jede von uns genutzte Tabelle besitzt ein Repository, das Methoden enthält, die PetaPoco verwenden, um SQL-Anfragen zu erstellen und auszuführen.

8. Schnittstellen (API-Design)

8.1 API-Überblick

Beschreibung der Schnittstellenarchitektur

Die tschiraplusAPI dient als Schnittstelle zwischen der Client-Anwendung tschiraplus und einer remote gehosteten PostgreSQL-Datenbank. Die API ist als RESTful Web API konzipiert und basiert auf ASP.NET Core. Sie ermöglicht eine sichere und effiziente Interaktion zwischen Clients und der Datenbank durch standardisierte HTTP-Methoden und eine JWT-Token-basierte Authentifizierung. Die API-Anfragen werden asynchron verarbeitet, damit die Anwendung während der Kommunikation mit dem Server nicht blockiert wird und eine reaktionsfähige Benutzererfahrung gewährleistet bleibt.

Technologie-Stack

Die API verwendet folgende Technologien und Frameworks:

- ASP.NET Core 9.0 als Web-Framework
- Entity Framework Core für die Datenbankinteraktion
- PostgreSQL als Datenbank-Backend
- JWT (JSON Web Tokens) zur Authentifizierung und Autorisierung
- Swagger/OpenAPI zur Dokumentation und Testbarkeit
- Docker & Docker Compose für containerisierte Bereitstellung

Architekturübersicht

Die API folgt einer schichtenbasierten Architektur mit den folgenden Hauptkomponenten:

1. Controller-Schicht: Definiert Endpunkte für CRUD-Operationen und spezielle Funktionen.

2. Service-Schicht: Kapselt die Geschäftslogik der API.
3. Repository-Schicht: Kapselt die Datenzugriffe über Entity Framework Core.
4. Datenbank-Schicht: PostgreSQL-Datenbank mit verschiedenen Tabellen zur Speicherung von Nutzern, Projekten, Aufgaben und weiteren relevanten Daten.

Sicherheitsmechanismen

- JWT-Authentifizierung: Clients müssen sich authentifizieren, um Zugriff auf geschützte Endpunkte zu erhalten.
- Role-Based Access Control (RBAC): Benutzerrollen und Berechtigungen steuern den Zugriff auf bestimmte Ressourcen.

Bereitstellung und Skalierung

Die API ist für eine containerisierte Bereitstellung mit Docker optimiert und kann über Docker Compose zusammen mit der PostgreSQL-Datenbank ausgeführt werden. In einer Produktionsumgebung kann sie über Kubernetes oder andere Orchestrierungstools skaliert werden.

8.2 Endpunkte und Operationen

Standard-CRUD-Endpunkte

Jede der folgenden Tabellen besitzt fünf Standard-Endpunkte für CRUD-Operationen:

- GET /api/{resource} → Gibt alle Einträge der Tabelle zurück
- GET /api/{resource}/{id} → Gibt einen spezifischen Eintrag nach ID zurück
- POST /api/{resource} → Erstellt einen neuen Eintrag
- PUT /api/{resource}/{id} → Aktualisiert einen bestehenden Eintrag
- DELETE /api/{resource}/{id} → Löscht einen Eintrag

Diese Endpunkte sind für die folgenden Tabellen verfügbar:

- Users
- Projects
- Tasks
- ProjectUsers
- UserTaskAssignments
- Und weitere Tabellen, die wir noch nicht verwenden

Spezielle Endpunkte

Authentifizierung:

- POST /api/Auth/Register → Registriert einen neuen Benutzer
- POST /api/Auth/Login → Authentifiziert einen Benutzer und gibt ein JWT-Token zurück
- POST /api/Auth/VerifyToken → Überprüft die Gültigkeit eines JWT-Tokens

Erreichbarkeit

- GET / → Prüft, ob der API-Server erreichbar ist (Health Check)

Synchronisation

- GET /api/Sync/projects/{userId} → Synchronisiert Projekte für einen Benutzer
- GET /api/Sync/tasks/{projectId} → Synchronisiert Aufgaben für ein Projekt

ProjectUsers-spezifische Endpunkte

- GET /api/ProjectUsers/ByUserId/{userId} → Gibt alle Projektzuordnungen eines Benutzers zurück
- GET /api/ProjectUsers/ByProjectId/{projectId} → Gibt alle Benutzer eines Projekts zurück
- PUT /api/ProjectUsers/DeleteByProjectId/{projectId} → Löscht alle Benutzer-Zuordnungen eines Projekts

Task-spezifische Endpunkte

- GET /api/Tasks/ByProjectId/{projectId} → Gibt alle Aufgaben eines Projekts zurück

Nutzung von HTTP-Verben und Statuscodes

- GET → Holt Daten (Statuscodes: 200 OK, 404 Not Found bei fehlenden Einträgen)
- POST → Erstellt neue Daten (Statuscodes: 201 Created, 400 Bad Request bei fehlerhaften Anfragen)
- PUT → Aktualisiert bestehende Daten (Statuscodes: 200 OK, 400 Bad Request, 404 Not Found)
- DELETE → Entfernt Daten (Statuscodes: 204 No Content, 404 Not Found)

8.3 API-Dokumentation

OpenAPI-Spezifikation

Die tschiraplusAPI verwendet Swagger zur automatisierten API-Dokumentation und Testbarkeit. Die API-Spezifikation wird beim Start im Entwicklungsmodus unter /Swagger bereitgestellt.

Die OpenAPI-Spezifikation beschreibt:

- Alle verfügbaren Endpunkte mit unterstützten HTTP-Methoden
- Erwartete Request-Parameter und Response-Schemata
- HTTP-Statuscodes für unterschiedliche Antwortszenarien
- Notwendige Authentifizierungsmechanismen

Zugriff auf Swagger UI

Während der Entwicklung kann die API-Dokumentation über folgenden Endpunkt aufgerufen werden: http://api.tschira.plus:42069/swagger/index.html

In der Produktionsumgebung ist Swagger nicht verfügbar.

8.4 Synchronisation und Datenkonsistenz

Synchronisierung zwischen Clients und Server

Die Synchronisation zwischen dem Client und dem Server erfolgt durch regelmäßige Abfragen und Updates der Projektdaten sowie der zugehörigen Aufgaben. Dies stellt sicher, dass alle relevanten Informationen stets aktuell und konsistent sind.

Funktionsweise der Synchronisation

Die Synchronisation wird über einen SyncService im Client verwaltet. Dieser kommuniziert mit dem Server über den RemoteDatabaseService, der HTTP-Anfragen zur Datenübertragung behandelt.

Synchronisation der Projekte

Methode: SyncProjectsAsync(Guid userId)

Vorgehen:

1. Der Client sendet eine Anfrage an den Server (sync/projects), um Projekt- und Nutzerdaten zu erhalten.
2. Falls ein Nutzer oder ein Projekt lokal nicht vorhanden ist, wird es hinzugefügt.
3. Falls das Projekt bereits existiert, wird geprüft, ob die Serverversion neuer ist. Falls ja, wird das Projekt aktualisiert.
4. Die Verknüpfungen zwischen Projekten und Nutzern werden synchronisiert.

5. Die Synchronisation wird in einem separaten Task-Thread in Intervallen von 30 Sekunden durchgeführt.

Synchronisation der Aufgaben

Methode: SyncTasksAsync(Guid projectId)

Vorgehen:

1. Der Client sendet eine Anfrage an den Server (sync/tasks), um die aktuellen Aufgaben für ein Projekt abzurufen.
2. Falls eine Aufgabe lokal nicht existiert, wird sie hinzugefügt.
3. Falls die Serverversion neuer ist als die lokale Version, wird die Aufgabe aktualisiert.
4. Die Synchronisation wird ebenfalls in einem separaten Task-Thread in Intervallen von 30 Sekunden durchgeführt.

Steuerung der Synchronisation

- Starten der Projektsynchronisation: StartProjectSync(Guid userId)
- Beenden der Projektsynchronisation: StopProjectSync()
- Starten der Aufgabensynchronisation: StartTaskSync(Guid projectId)
- Beenden der Aufgabensynchronisation: StopTaskSync()

Jede dieser Methoden nutzt CancellationTokenSource, um eine laufende Synchronisation bei Bedarf sicher zu beenden.

Kommunikation mit dem Server

Die Kommunikation zwischen Client und Server erfolgt über den RemoteDatabaseService, der verschiedene HTTP-Anfragen an den Server sendet.

Die wichtigsten Methoden sind:

- PostAsync(string endpoint, string data): Sendet Daten per HTTP-POST an den Server.
- GetAllAsync(string endpoint): Ruft alle Einträge eines Endpunkts per HTTP-GET ab.
- GetByIdAsync(string endpoint, Guid id): Ruft einen spezifischen Eintrag per HTTP-GET anhand einer ID ab.
- UpdateAsync(string endpoint, Guid id, string data): Aktualisiert einen Eintrag per HTTP-PUT.
- DeleteAsync(string endpoint, Guid id): Löscht einen Eintrag per HTTP-DELETE.

Alle Anfragen enthalten einen Authorization-Header mit einem Bearer-Token zur Authentifizierung.

Datenkonsistenz

- Um sicherzustellen, dass keine veralteten Daten überschrieben werden, wird das LastUpdated-Feld geprüft.
- Falls die Serverversion neuer ist als die lokale Version, wird das Objekt aktualisiert.
- Falls eine lokale Version nicht existiert, wird das Objekt neu angelegt.

Fehlerbehandlung

- Jeder Synchronisationsprozess ist in einen try-catch-Block eingeschlossen, um mögliche Fehler abzufangen.
- Fehler werden in der Konsole ausgegeben, um Debugging zu erleichtern.
- Falls keine Daten vom Server empfangen werden, erfolgt ein entsprechender Hinweis.

9. UI

9.1 UI-Architektur

• Erklärung des MVVM-Musters:

Unsere UI-Architektur folgt dem **MVVM (Model-View-ViewModel)**-Muster, um eine klare Trennung zwischen Datenhaltung, Geschäftslogik und der Darstellungsebene zu gewährleisten. Dieses Muster verbessert die Wartbarkeit und Skalierbarkeit des Projekts, indem es die Zuständigkeiten in drei Hauptbereiche aufteilt.

- **Model (Core-Ebene):**
  Diese Schicht enthält die Datenstrukturen und Geschäftslogik unseres Projekts. Hier werden die grundlegenden Modelle wie Aufgaben, Benutzer und Projekte definiert.
- **ViewModel (UI-Ebene):**
  Das ViewModel fungiert als Vermittler zwischen den Models und den Views. Jede View besitzt ein eigenes ViewModel, das die notwendigen Daten aus dem Model lädt, verarbeitet und an die View weitergibt.
- **View (UI-Ebene):**
  Die Views sind für die Darstellung der Benutzeroberfläche unseres Projektplaners verantwortlich. Sie bestehen aus zwei Komponenten: Die `.axaml`-Datei enthält die XAML-Definitionen der UI-Elemente und nutzt Datenbindung zum ViewModel, um Inhalte anzuzeigen. Die `.axaml.cs`-Datei enthält

beispielsweise Event-Handler oder spezielle UI-Logik, die sich nicht direkt über das ViewModel abbilden lässt.

9.2 GUI-Komponenten

Sidebar & Tabs:

Wir benutzen Sidebar und Tabs zur Navigation zwischen den verschiedenen Ansichten. Die Sidebar bietet eine übersichtliche Struktur, die es dem Nutzer ermöglicht, zur allgemeinen Projektliste zu navigieren und einzelne Projekte auszuwählen. Hinzufügend gibt es uns weitere Navigationselemente für das User-Profil und die Benutzereinstellungen. Die Tabs ermöglichen es dem Nutzer, zwischen einer Kanban-Ansicht und einer Aufgabenlisten-Ansicht für jedes Projekt zu wechseln. Diese Ansichtsmöglichkeiten sind auf zwei Tabs beschränkt, die nicht geschlossen werden können, wodurch der Überblick bewahrt bleibt. Diese Kombination aus Sidebar und Tabs sorgt für eine klare Struktur und ermöglicht eine schnelle, übersichtliche Interaktion, ohne dass der Nutzer zwischen mehreren Menüs wechseln muss.
Flyouts:

In unserem Projekt haben wir Flyouts für mehrere Zwecke eingesetzt.
Einerseits werden sie für die Projekt- und Task-Detailansicht verwendet, in der zusätzliche Informationen angezeigt werden, die direkt bearbeitet werden können. Falsche Eingaben werden verhindert, indem der Benutzer keine ungültigen Daten eingeben kann, da die Auswahlmöglichkeiten meist auf eine Flyout-Liste beschränkt sind. Dies optimiert den Benutzerfluss und ermöglicht schnelle und einfache Änderungen. Andererseits werden Flyouts auch für die Erstellung von Projekten und Aufgaben genutzt.

Buttons und Aktionen:

Unsere Buttons sind in der Regel beschriftet, sodass der Nutzer sofort weiß, welche Funktion sie ausführen. Zusätzlich haben wir Buttons mit unterschiedlichen Farben versehen, zum Beispiel sind Buttons zum Erstellen einer Aufgabe für die Bestätigung grün. Das Schließen einer Ansicht erfolgt über ein X-Symbol in der rechten oberen Ecke. Die Buttons bieten visuelle Rückmeldungen, wenn der Nutzer mit ihnen interagiert. Beim Speichern der Kommentar-Sektion haben wir zusätzlich eine grüne Farbänderung integriert, die signalisiert, dass der Kommentar erfolgreich gespeichert wurde. Diese visuellen Effekte verbessern die Benutzererfahrung, indem sie dem Nutzer sofortige Bestätigung über die erfolgreiche Durchführung der Aktion geben.

9.3 Asynchrone Prozesse

• Umsetzung langlaufender Abfragen asynchron:

In unserem Projekt setzen wir asynchrone Prozesse ein, um langlaufende Abfragen und Datenoperationen effizient zu verwalten und die Benutzeroberfläche nicht zu blockieren. Beispiele für diese asynchronen Methoden finden sich unter anderem in unsere Repositories, wie etwa im **`TaskRepository`**, **`ProjectRepository`** oder **`UserRepository`**. Diese Repositories greifen asynchron auf unsere Remote-Datenbank zu, um Projekte und Benutzerinformationen zu erstellen, zu aktualisieren oder abzurufen. Außerdem werden Projekte und Tasks asynchron aus der lokalen Datenbank geladen.

10. Reflexion & Lessons Learned

10.1 Projektreflexion

- **Was lief gut und welche Probleme gab es?**

Die Teamkommunikation verlief während des Projekts weitgehend reibungslos. Unterstützung wurde stets angeboten, wenn sie benötigt wurde, und offene Fragen konnten gemeinsam geklärt werden. Da das Projekt parallel zum Studium lief, gab es vereinzelt Unklarheiten bei Aufgaben oder deren Umsetzung. Diese Herausforderungen konnten jedoch durch Abstimmung und Zusammenarbeit schnell bewältigt werden. Auch die Organisation des Projekts funktionierte gut, da alle Teammitglieder verantwortungsvoll zu den jeweiligen Aufgaben beigetragen haben.

10.2 Individuelle Reflexionen

Anna-Lena Miletic

Was habe ich gelernt?

Das Projekt war für mich lehrreich auf verschiedenen Ebenen. Im Sinne des Programmierens war es neu für mich mit C# zu arbeiten. Auch war das Strukturieren des Codes durch unser Schichtenmodell und die Verwendung von Models und Dto's zu Beginn noch gewöhnungsbedürftig. Später war dies aber sehr einfach zu verwenden.

Wo habe ich noch Lücken?

Bei der Größe unseres Projekts und unserer Projektgruppe haben wir natürlich unsere Aufgaben aufgeteilt. Aus diesem Grund habe ich nur ein Grundverständnis für Bereiche, die ich selbst nicht geschrieben habe. Bei komplexeren Aspekten wie der API müsste

ich mich wahrscheinlich noch einmal näher mit der tatsächlichen Implementierung auseinandersetzen.

Bei diesen Themen bin ich über meinen Schatten gesprungen

Besonders angsteinflößend war zu Beginn die Entscheidung, im Rahmen des Projekts eine neue Programmiersprache zu lernen. Die Ähnlichkeit zu Java war hilfreich, aber an manchen Stellen war ich ratlos und musste mir Hilfe suchen. Inzwischen fühle ich mich aber deutlich wohler in C# und bin froh, dass wir diese Idee hatten.

Franziska Landt

Was habe ich gelernt?

Ich habe eine neue Sprache gelernt und somit neue Syntax kennengelernt. Zudem habe ich erfahren, wie es ist, in größeren Teams zu arbeiten und die Kommunikation abzustimmen. In meinem Bereich habe ich an der Remote-Datenbank gearbeitet und dadurch neue Kenntnisse darüber erlangt, wie man Daten abfragt, löscht und bearbeitet. Ich habe auch gelernt, wie man eine View in AXAML hardcodet und wie Events in einer anderen Sprache funktionieren. Zudem habe ich besser verstanden, wie Gruppendynamik innerhalb eines Teams funktioniert. Ein weiterer wichtiger Punkt ist, dass ich mich mit Git Flow vertraut gemacht habe, was in früheren Projekten nicht der Fall war.

Wo habe ich noch Lücken?

Vor allem bei Themen wie Modellen und Factories habe ich mich eher zurückgehalten. Ich fühle, dass ich in diesen Bereichen noch tieferes Verständnis und praktische Erfahrung entwickeln könnte, um die Konzepte besser zu beherrschen.

Bei diesen Themen bin ich über meinen Schatten gesprungen
Direkt über meinen Schatten springen musste ich bei keinem Thema. In meiner Gruppe konnte ich offen kommunizieren, was meine Anliegen sind und was meine Motivation ist. Daher gab es keine Themen, bei denen ich mich überwinden musste, um sie anzusprechen oder auszuführen.

Gabriel Eißler
Was habe ich gelernt?

- Architekturstrukturen (MVVM; DTOs; 3-Schichten-Modell)
- Api-Design und Netzwerkaufbau
- Einrichten des Servers
- Allgemein mehr über Programmierprinzipien gelernt

Wo habe ich noch Lücken?

Testing. Allgemeine Projektplanung. Wir mussten den Plan mehrmals mittendrin ändern

Bei diesen Themen bin ich über meinen Schatten gesprungen

Beim Server. Ich hatte mir schon immer eien eigenenen Home-Server einrichten. Und konnte dies im Zuge des Projekts auch durchführen.

Isaak Kulikowskich

Was habe ich gelernt?

- Neue Sprache (C#)
- Architekturstrukturen (MVVM; DTOs; 3-Schichten-…)
- Gruppenarbeit (und damit zusammenhängend Krisenmanagement)
- Ein bisschen Testing-Framework

Wo habe ich noch Lücken?

API und allgemein Netzwerkdesign

Bei diesen Themen bin ich über meinen Schatten gesprungen

Jetz nicht vielleicht über meinen Schatten gesprungen, aber ich hatte (und habe) immer das Gefühl gehabt, dass ich zu wenig in der Gruppe geleistet habe. Es könnte aber auch einfach darin liegen das wir ein paar sehr gute Experten in userm Team hatten.

Polina Rogge
Was habe ich gelernt?

- C# (neue Sprache)
- Arbeit in einer großen Gruppe
- Architekturstrukturen
- Datenbankeinbindung
- UI mit AvaloniaUI

Wo habe ich noch Lücken?

- API
- Testing

Bei diesen Themen bin ich über meinen Schatten gesprungen

Die neue Sprache und die neue sichereren Architekturstrukturen haben mir am Anfang schon Angst eingejagt, jedoch als ich mich damit auseinandergesetzt habe war es gar nicht mehr so schlimm.

Sofia Koumpatidou

Was habe ich gelernt?

Durch das Projekt konnte ich erstmals mit der Programmiersprache C# arbeiten, was mir viel Spaß gemacht hat. Neben der Sprache habe ich auch viel über die effiziente und strukturierte Organisation eines großen Projekts sowie über Konfliktlösung gelernt.

Besonders hat es mir gefallen, mich intensiver mit UI-Design zu beschäftigen und mein bisheriges Wissen praktisch anzuwenden. Auch die intensive Auseinandersetzung mit der Backend-Programmierung hat mir wertvolles Wissen vermittelt.

Wo habe ich noch Lücken?

Mit weniger Zeitdruck hätte ich mich noch intensiver mit C# beschäftigt, um ein tieferes Verständnis der Sprache zu entwickeln. Außerdem hätte ich mehr Wert auf eine noch sauberere und durchdachtere Programmierung gelegt, um den Code effizienter und nachhaltiger zu gestalten. Außerdem habe ich mich während des Projekts weniger mit Themen wie Modelle auseinandergesetzt, somit besteht da viel Lernbedarf.

Bei diesen Themen bin ich über meinen Schatten gesprungen

Während des Projekts gab es kein spezielles Thema, bei dem ich wirklich über meinen Schatten springen musste. Das Einzige, was dem nahekommt, war, mich dazu zu überwinden, mir komplexere Aufgaben/Issues zuzuweisen. Diese gingen oft mit mehr Verantwortung einher, was anfangs eine Herausforderung war, mich aber letztendlich weitergebracht hat.

YuTing Chen

Was habe ich gelernt?

Neue Sprache (C#) Architekturstrukturen (MVVM; DTOs; 3-Schichten-...) UI-Design bischen Backend

Wo habe ich noch Lücken?

Ich hatte bisschen Probleme mit Backend

Bei diesen Themen bin ich über meinen Schatten gesprungen

Allgemein hatte ich dieses Semester sehr viel zu tun und ich bin sehr froh darüber zum Projekt beitragen zu können.

10.3 Verbesserungsmöglichkeiten - Ausblick in die Zukunft

Weitere Features, die wir gerne noch implementiert hätten, sind die Möglichkeit, Custom Tags zu Projekten hinzuzufügen, sowie ein Belohnungssystem, das Nutzer für das Abschließen von Aufgaben motiviert. Zusätzlich war geplant, ein Sprint-Feature zu integrieren sowie eine Drag-and-Drop-Animation zu entwickeln, um das Verschieben der Tasks im Kanban-View visuell ansprechender zu gestalten. Es gibt einige Features, die von uns schon angesetzt wurden, jedoch noch ausgebaut werden müssen.