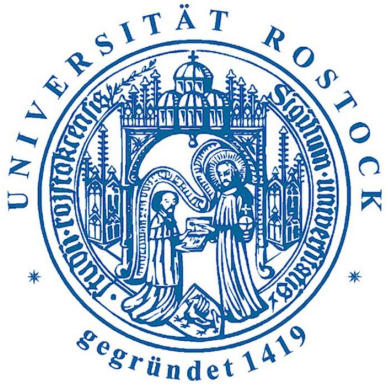


Softwaretechnik WS 2024-25
Projektabgabedokument Sprint

Übung-2 Team-1

Dogukan Karakoyun, 223202023
Hüseyin Kayabasi, 223201801
Helin Oguz, 223202103
Eren Temizkan, 223201982
Cagla Yesildogan, 223201881

Einleitung

In diesem Abschnitt wird der Produkt-Backlog vor und nach dem Sprint dokumentiert. Es wird gezeigt, welche Änderungen vorgenommen wurden und wie sich der Backlog im Laufe des Sprints entwickelt hat.

Produkt-Backlog vor dem Sprint

Vor dem Sprint bestand der Produkt-Backlog aus den folgenden User Stories:

- **US-1.1 Benutzerkonto erstellen**
  Basis-Registrierung für Nutzer mit Name, E-Mail und Passwort.
- **US-3.1 Aufgabe erstellen**
  Nutzer können Aufgaben mit Titel und Beschreibung anlegen.

GitLab-Link zur Testdokumentation: Produktbacklog auf GitLab

Änderungen während des Sprints

Während des Sprints wurden die folgenden Erweiterungen an den User Stories vorgenommen:

US-1.1 Erweiterungen

- Passwort wird nun sicher gehasht (`IPasswordHasher`).
- E-Mail-Verifizierung wurde hinzugefügt (`IEmailVerification`).
- Benutzerrollenverwaltung wurde erweitert (`IAuthorizationService`).

US-3.1 Erweiterungen

- CRUD-Operationen für Aufgaben (`ICRUDTask`, `TaskService`) implementiert.
- Aufgaben können einem Benutzer zugewiesen werden (`assignedUser: User`).
- Aufgaben können Dateien enthalten (`File`-Klasse).
- Aufgaben sind mit Projekten verknüpft (`project: Project`).

Begründung der Änderungen

Die Änderungen wurden aufgrund von Sicherheits- und Benutzerfreundlichkeitsanforderungen vorgenommen. Das Team stellte fest, dass eine Passwort-Hashing-Implementierung für die Sicherheit erforderlich ist. Ebenso wurde eine E-Mail-Verifizierung hinzugefügt, um sicherzustellen, dass nur legitime Benutzer Zugriff auf die Plattform haben. Die Benutzerrollenverwaltung wurde erweitert, um eine differenzierte Berechtigungsstruktur zu ermöglichen.

1+1/1

Sprint-Planung

Einleitung

Die Sprint-Planung basiert auf der Auswahl der wichtigsten User Stories für Sprint 1. Es wurden die folgenden User Stories und deren zugehörigen Tasks in den Sprint-Backlog aufgenommen.

Aufteilung der User Stories und Tasks

Die ausgewählten User Stories wurden in kleinere Tasks aufgeteilt:

US-1.1 Benutzerkonto erstellen

- Registrierung API entwickeln
- Passwort-Hashing implementieren (`IPasswordHasher`)
- E-Mail-Verifizierung hinzufügen (`IEmailVerification`)
- Benutzerrollenverwaltung erweitern (`IAuthorizationService`)

US-3.1 Aufgabe erstellen

- Task-Klasse implementieren
- CRUD-Operationen für Tasks entwickeln (`ICRUDTask`, `TaskService`)
- Aufgaben einem Benutzer zuweisen (`assignedUser: User`)
- Datei-Upload für Aufgaben ermöglichen (`File`-Klasse)
- Aufgaben mit Projekten verknüpfen (`project: Project`)

Um die User Stories auf Tasks herunterzubrechen, wurde die **Story-Point-Schätzmethode** verwendet. Diese Methode wurde gewählt, da sie eine schnelle und effektive Möglichkeit bietet, den Arbeitsaufwand zu bewerten, ohne dass genaue Zeitangaben erforderlich sind. Jedes Teammitglied hat eine unabhängige Schätzung abgegeben, und bei großen Abweichungen wurde eine Diskussion geführt, um den besten Wert zu ermitteln. Diese Methode hat sich als effektiv erwiesen, um komplexe Aufgaben in kleinere, realistisch machbare Einheiten zu unterteilen.

Aufwandsschätzung

Für die Sprint-Planung wurde die Story-Point-Schätzungsmethode verwendet. Jedes Teammitglied hat eine Schätzung abgegeben. Falls es große Unterschiede gab, musste das höchste geschätzte Teammitglied eine Begründung liefern.

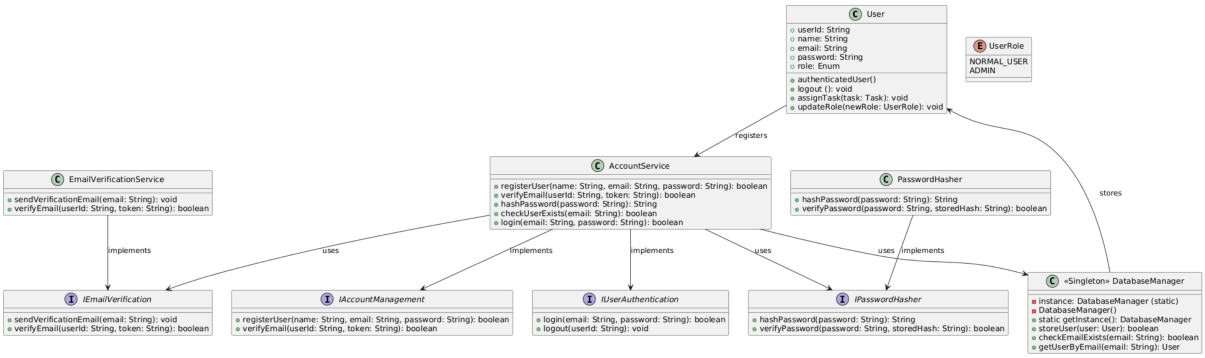
User Story	Task	Helin	Hüseyin	Dogukan	Cagla	Eren	Durchschnittliche Story Points	Begründung
US-1.1	Registrierung API entwickeln	5	3	8	6	5	5.4	Backend könnte komplexer sein, API-
US-1.1	Passwort-Hashing implementieren	2	3	3	2	3	2.6	Keine große Herausforderung, einfache Implementier
US-1.1	E-Mail-Verifizierung hinzufügen	5	4	5	3	4	4.2	SMTP-Server kann Probleme machen,
US-3.1	CRUD-Operationen für Tasks	8	8	13	10	9	9.6	Datenbank-Operationen und Validierung
US-3.1	Aufgaben einem Benutzer zuweisen	5	6	7	5	6	5.8	Benutzerrollen müssen klar definiert werden.
US-3.1	Datei-Upload für Aufgaben	6	7	8	7	7	7.0	Frontend-Backend-Integration muss getestet

1+1/1

2.1.1.1 Benutzerkonto erstellen [US-1.1]

In diesem Abschnitt wird die verfeinerte Struktur für die User Story US-1.1 "Benutzerregistrierung" modelliert. Die Modellierung erfolgt mit Klassendiagrammen, die die wichtigsten Komponenten, Attribute, Methoden und Relationen enthalten. Zusätzlich werden die Verantwortlichkeiten der Klassen erläutert.

UML-Klassendiagramm
Modell der Benutzerkonto-Erstellung



Modellierte Klassen und ihre Verantwortlichkeiten

- **User**: Repräsentiert einen Benutzer im System. Sie enthält grundlegende Informationen wie `userId`, `name`, `email`, `passwordHash` und `role`.
  - *Verantwortung*: Speicherung der Benutzerinformationen.
- **AccountService**: Zentrale Klasse zur Verwaltung der Benutzerregistrierung und Authentifizierung.
  - *Verantwortung*:
    - Registrierung eines neuen Benutzers.
    - Hashing des Passworts.
    - Überprüfung der E-Mail-Adresse.
    - Verwaltung des Login-Prozesses.
- **DatabaseManager**: Speicherung und Abruf von Benutzerdaten.
- **PasswordHasher**: Sichere Speicherung und Verifikation von Passwörtern.
- **EmailVerificationService**: Verwaltung der E-Mail-Bestätigung.

Erweiterungen aus vorherigen Sprints Dieses Modell basiert auf der Architektur des vorherigen Sprints und wurde erweitert. Im Vergleich zum ursprünglichen Entwurf wurden wichtige Änderungen an der Systemarchitektur, Sicherheitsmaßnahmen und zusätzlichen Funktionen vorgenommen.

Verwendung von Design Patterns

- **Singleton Pattern** (für DatabaseManager)
  - Sicherstellt, dass es nur eine Instanz der Datenbankverwaltung gibt.
  - Vorteil: Einheitliche Datenverwaltung und weniger Speicherverbrauch.

Begründung der Modellwahl

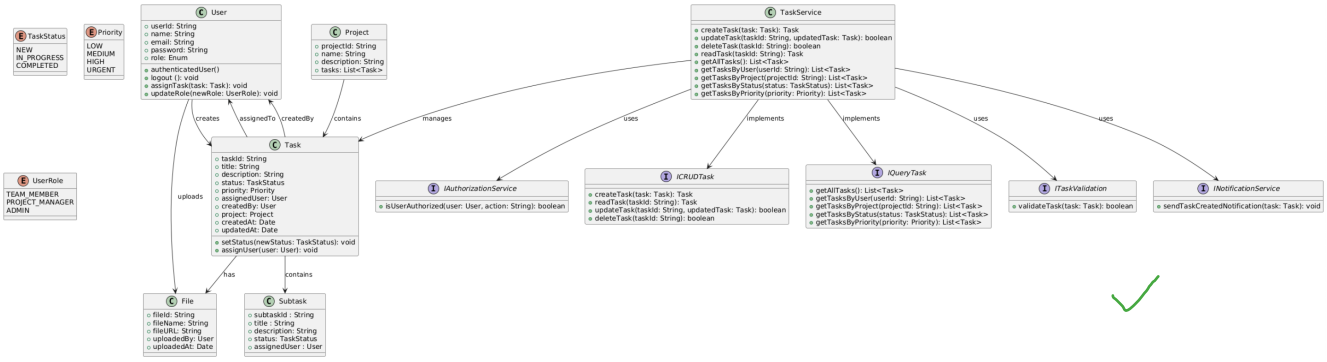
Dieses Modell wurde gewählt, weil es folgende Vorteile bietet:

- Hohe Sicherheit durch Passwort-Hashing und E-Mail-Verifikation.
- Modularität und Erweiterbarkeit durch den Einsatz von Interfaces.
- Trennung von Verantwortlichkeiten (Separation of Concerns), was eine bessere Wartbarkeit ermöglicht.
- Erweiterbar für zukünftige User Stories, z. B. durch einfache Integration neuer Rollen oder Authentifizierungsverfahren.

2.1.2.1 Task-Erstellung [US-3.1]

Dieser Abschnitt beschreibt die detaillierte Modellierung der User Story US-3.1 "Task-Erstellung". Die Modellierung erfolgt anhand eines Klassendiagramms, welches relevante Klassen, Attribute, Methoden und Relationen zeigt. Zusätzlich werden die Verantwortlichkeiten der modellierten Komponenten beschrieben.

UML-Klassendiagramm
Modell der Task-Erstellung



Modellierte Klassen und ihre Verantwortlichkeiten

- **User**: Repräsentiert einen Benutzer im System und enthält grundlegende Informationen wie `userId`, `name`, `email`, `password` und `role`. Verantwortlichkeiten:
  - Verwaltung von Benutzerinformationen und Rollen.
  - Aufgaben zuweisen mittels `assignTask(Task)`.
  - Authentifizierung über `authenticatedUser()`.
- **Task**: Repräsentiert eine Aufgabe mit `title`, `description`, `status`, `priority`, `createdAt` und `updatedAt`. Verantwortlichkeiten:
  - Verwaltung des Aufgabenstatus,updatedAt wird automatisch aktualisiert (`setStatus()`).
  - Zuweisung von Benutzern zu Aufgaben (`assignUser()`).
  - Speicherung von Dateien innerhalb einer Aufgabe (`File`).
  - Enthält eine Liste von Unteraufgaben und wenn alle Subtasks erledigt sind, wird die Hauptaufgabe automatisch abgeschlossen(`Subtask`).
- **Project**: Organisiert mehrere Tasks innerhalb einer Projektstruktur. Verantwortlichkeiten:
  - Speicherung und Verwaltung von Aufgaben innerhalb eines Projekts.
- **TaskService**: Kernkomponente zur Verwaltung von Tasks. Verantwortlichkeiten:
  - Erstellung, Aktualisierung und Löschung von Tasks.

  - Abfrage von Tasks nach Benutzer, Projekt, Status oder Priorität.

- **File**: Repräsentiert eine Datei im System und speichert grundlegende Metadaten wie `fileId`, `fileName`, `fileURL`, `uploadedBy` und `uploadedAt`. Verantwortlichkeiten:
  - Verwaltung von Datei-Metadaten (Dateiname, URL, Hochlade-Datum).
  - Zuordnung der Datei zu einem Benutzer mittels `uploadedBy`, um die Verantwortlichkeit nachzuvollziehen.
  - Speicherung und Zugriff auf hochgeladene Dateien im System.

- **Subtask**: Repräsentiert eine Unteraufgabe innerhalb eines Aufgabenmanagementsystems. Enthält eine eindeutige `subtaskId`, einen Titel (`title`), eine Beschreibung (`description`), einen Status (`status`) und einen zugewiesenen Benutzer (`assignedUser`). Verantwortlichkeiten:
  - Verwaltung von Unteraufgaben mittels `subtaskId`, `title` und `description`.
  - Statusverwaltung der Unteraufgabe über das Attribut `status: TaskStatus`, um den Fortschritt zu dokumentieren.
  - Zuweisung eines verantwortlichen Benutzers über `assignedUser` zur Bearbeitung der Unteraufgabe.

✓

Begründung der Modellwahl

Flexibilität, Erweiterbarkeit und Software Design Prinzipien

Die Wahl der modellierten Klassen und Komponenten basiert auf wesentlichen Softwarearchitekturprinzipien, um eine nachhaltige, erweiterbare und flexible Systemstruktur zu gewährleisten.

- **Flexibilität:**
  - Die Trennung von Aufgaben- und Dateiverwaltung sorgt für eine hohe Anpassungsfähigkeit des Systems.
  - Die Klasse `Subtask` ermöglicht eine dynamische Aufgabenverwaltung durch den Statusmechanismus, wodurch neue Workflows einfach integriert werden können.
  - Die Klasse `File` erlaubt eine flexible Handhabung von hochgeladenen Dateien, sodass unterschiedliche Dateitypen unterstützt werden können.

- **Erweiterbarkeit:**
  - Durch den modularen Aufbau der Klassen kann das System ohne tiefgreifende Änderungen erweitert werden.
  - Die Verwendung von Interfaces wie `ICRUDTask` erleichtert die Implementierung neuer Aufgabenverwaltungsfunktionen.

- **Software Design Prinzipien:**
  - **Separation of Concerns (SoC):** Jede Komponente ist für eine spezifische Funktion verantwortlich:
    * `File`: Verwaltung und Speicherung von Dateien.
    * `Subtask`: Verwaltung von Unteraufgaben innerhalb eines Tasks.
    * `TaskService`: Steuerung der zentralen Logik für das Aufgabenmanagement.

    Diese klare Trennung verbessert die Wartbarkeit und Skalierbarkeit des Systems.
  - **Single Responsibility Principle (SRP):** Jede Klasse hat genau eine klar definierte Verantwortung. Dadurch wird die Code-Wartung vereinfacht und die Fehleranfälligkeit reduziert.
  - **Open/Closed Principle (OCP):** Das System ist offen für Erweiterungen, ohne dass bestehender Code modifiziert werden muss. Neue Dateiformate oder Task-Typen können problemlos hinzugefügt werden.
  - **Dependency Inversion Principle (DIP):** Die Verwendung von Abstraktionen (z. B. Interfaces für Aufgabenverwaltung und Benachrichtigungssysteme) sorgt für eine lose Kopplung zwischen den Modulen.

Diese Architektur gewährleistet eine nachhaltige Softwarelösung, die leicht erweiterbar und flexibel an neue Anforderungen anpassbar ist.

3/3 ✓

Modellierung verfeinerter Interfaces und Datenstrukturen

Verfeinerung der Interfaces zwischen den Komponenten

Um die Kommunikation zwischen den Komponenten effizient und skalierbar zu gestalten, wurden die folgenden Interfaces modelliert und mit Methoden versehen.

2.1.1.2 User Story – Interface Zuordnung

- **US-1.1: Benutzerkonto erstellen** Als neuer Nutzer möchte ich ein Konto erstellen, um auf die Plattform zugreifen zu können.
  - **IAccountManagement**
    - `registerUser(name, email, password)` Erstellt einen neuen Benutzer, speichert seine Daten in der Datenbank und verschickt eine Bestätigungs-E-Mail.
    - `verifyEmail(userId, token)` Überprüft den Bestätigungstoken und aktiviert den Benutzer.
  - **IEmailVerification**
    - `sendVerificationEmail(email)` Sendet eine E-Mail mit einem Verifizierungslink an den Benutzer.
    - `verifyEmail(userId, token)` Prüft, ob der Token gültig ist, und schaltet den Benutzer frei.
  - **IPasswordHasher**
    - `hashPassword(password)` Erstellt einen sicheren Hash aus dem Passwort und speichert ihn in der Datenbank.
    - `verifyPassword(password, storedHash)` Vergleicht ein eingegebenes Passwort mit dem gespeicherten Hash, um die Identität zu bestätigen.
  - **IUserAuthentication**
    - `login(email, password)` Überprüft die Zugangsdaten und generiert eine Benutzersitzung.
    - `logout(userId)` Beendet die aktuelle Benutzersitzung.

2.1.2.2 User Story – Interface Zuordnung

- **US-3.1: Aufgaben verwalten** Als Nutzer möchte ich Aufgaben erstellen, bearbeiten und verwalten, um meine Arbeit zu organisieren.
  - **ICRUDTask**
    - `createTask(title, description, assignedUser, priority)` Erstellt eine neue Aufgabe und speichert sie in der Datenbank.
    - `readTask(taskId)` Ruft eine Aufgabe anhand ihrer ID aus der Datenbank ab.
    - `updateTaskStatus(taskId, status)` Aktualisiert den Status einer vorhandenen Aufgabe.
    - `deleteTask(taskId)` Entfernt eine Aufgabe endgültig aus dem System.
  - **IQueryTask**
    - `getTasksByUser(userId)` Gibt eine Liste aller Aufgaben zurück, die einem bestimmten Benutzer zugewiesen sind.
    - `getTasksByProject(projectId)` Ruft alle Aufgaben ab, die mit einem bestimmten Projekt verbunden sind.
    - `getTasksByStatus(status)` Filtert Aufgaben basierend auf ihrem aktuellen Status.
    - `getTasksByPriority(priority)` Gibt Aufgaben zurück, die eine bestimmte Priorität haben.

- **ITaskValidation**
  - `validateTask(task)` Überprüft, ob eine Aufgabe alle erforderlichen Felder enthält und valide ist.
- **INotificationService**
  - `sendTaskCreatedNotification(task)` Sendet eine Benachrichtigung an den Benutzer, wenn eine neue Aufgabe erstellt wurde.

IAccountManagement (Implementiert durch: AccountService)

```
public interface IAccountManagement {
    boolean registerUser(String name, String email, String password);
    boolean verifyEmail(String userId, String token);
}
```

- **registerUser(name: String, email: String, password: String): boolean** – Erstellt ein neues Benutzerkonto.
- **verifyEmail(userId: String, token: String): boolean** – Verifiziert die E-Mail-Adresse eines Benutzers.

IEmailVerification (Implementiert durch: EmailVerificationService)

```
public interface IEmailVerification {
    void sendVerificationEmail(String email);
    boolean verifyEmail(String userId, String token);
}
```

- **sendVerificationEmail(email: String): void** – Versendet eine Verifizierungs-E-Mail an die angegebene Adresse.
- **verifyEmail(userId: String, token: String): boolean** – Prüft die Gültigkeit eines Verifizierungstokens und aktiviert das zugehörige Benutzerkonto.

IPasswordHasher (Implementiert durch: PasswordHasher)

```
public interface IPasswordHasher {
    String hashPassword(String password);
    boolean verifyPassword(String password, String storedHash);
}
```

- **hashPassword(password: String): String** – Erstellt einen sicheren Hash aus einem Passwort.
- **verifyPassword(password: String, storedHash: String): boolean** – Vergleicht ein Passwort mit einem gespeicherten Hash.

IUserAuthentication (Implementiert durch: AccountService)

```
public interface IUserAuthentication {
    boolean login(String email, String password);
    void logout(String userId);
}
```

- **login(email: String, password: String): boolean** – Authentifiziert einen Benutzer und gewährt Zugriff.
- **logout(userId: String): void** – Beendet die aktive Benutzersitzung.

ICRUDTask

```
public interface ICRUDTask {
    Task createTask(String title, String description, String assignedUserId, Priority
        priority);
    Task readTask(String taskId);
    boolean updateTask(String taskId, Task updatedTask);
    boolean deleteTask(String taskId);
}
```

- **createTask(title: String, description: String, assignedUserId: String, priority: Priority): Task** – Erstellt eine neue Aufgabe mit den angegebenen Details und weist sie einem Benutzer zu.
- **readTask(taskId: String): Task** – Ruft die Aufgabe mit der angegebenen ‘taskId‘ aus dem System ab.
- **updateTask(taskId: String, updatedTask: Task): boolean** – Aktualisiert eine vorhandene Aufgabe mit neuen Werten.
- **deleteTask(taskId: String): boolean** – Löscht die Aufgabe mit der angegebenen ‘taskId‘.

IQueryTask

```
public interface IQueryTask {
    List<Task> getAllTasks();
    List<Task> getTasksByUser(String userId);
    List<Task> getTasksByProject(String projectId);
    List<Task> getTasksByStatus(TaskStatus status);
    List<Task> getTasksByPriority(Priority priority);
}
```

- **getAllTasks(): List<Task>** – Gibt eine Liste aller Aufgaben im System zurück.
- **getTasksByUser(userId: String): List<Task>** – Gibt alle Aufgaben zurück, die einem bestimmten Benutzer zugewiesen sind.
- **getTasksByProject(projectId: String): List<Task>** – Ruft alle Aufgaben ab, die mit einem bestimmten Projekt verbunden sind.
- **getTasksByStatus(status: TaskStatus): List<Task>** – Gibt eine Liste von Aufgaben mit einem bestimmten Status zurück.
- **getTasksByPriority(priority: Priority): List<Task>** – Ruft alle Aufgaben ab, die eine bestimmte Priorität haben.

IAuthorizationService

```
public interface IAuthorizationService {
    boolean isUserAuthorized(User user, String action);
}
```

- **isUserAuthorized(user: User, action: String): boolean** – Überprüft, ob ein Benutzer berechtigt ist, eine bestimmte Aktion auszuführen.

ITaskValidation

```
public interface ITaskValidation {
    boolean validateTask(Task task);
}
```

- **validateTask(task: Task): boolean** – Validiert eine Aufgabe, um sicherzustellen, dass sie den definierten Anforderungen entspricht.

INotificationService

```
public interface INotificationService {
    void sendTaskCreatedNotification(Task task);
}
```

- **sendTaskCreatedNotification(task: Task): void** – Sendet eine Benachrichtigung, wenn eine neue Aufgabe erstellt wird.

Verantwortlichkeiten der Interfaces

Die definierten Interfaces übernehmen spezifische Verantwortlichkeiten, um eine klare Trennung der Logik zu gewährleisten:

- **IAccountManagement**: Zuständig für die Benutzerverwaltung, insbesondere die Registrierung und Aktivierung eines Accounts.
- **IEmailVerification**: Separates Interface für die Verifikation der E-Mail-Adresse, um die Sicherheit zu erhöhen.
- **IPasswordHasher**: Verantwortlich für die sichere Speicherung von Passwörtern durch Hashing-Mechanismen.
- **IUserAuthentication**: Handhabt die Anmeldung und Abmeldung von Benutzern, um Sitzungen zu verwalten.
- **ICRUDTask**: Ermöglicht grundlegende CRUD-Operationen (Erstellen, Lesen, Aktualisieren, Löschen) für Aufgaben.
- **IQueryTask**: Bietet komplexe Abfragefunktionen, um Aufgaben nach Benutzer, Status oder Projekt zu filtern.
- **ITaskValidation**: Validiert Aufgaben, bevor sie gespeichert oder aktualisiert werden, um fehlerhafte Einträge zu verhindern.
- **INotificationService**: Stellt sicher, dass Benutzer über wichtige Ereignisse (z. B. das Erstellen neuer Aufgaben) benachrichtigt werden.

Durch diese Trennung wird sichergestellt, dass jede Komponente nur eine bestimmte Aufgabe übernimmt, was die Wartbarkeit und Erweiterbarkeit des Systems verbessert.

Begründung der Wahl der Interfaces und Datenstrukturen

Die Auswahl der Interfaces und Datenstrukturen basiert auf den Anforderungen der User Stories sowie auf den Prinzipien von Software-Architektur, insbesondere auf Separation of Concerns (SoC), Erweiterbarkeit und Datenintegrität.

Weitere Details:

- **Separation of Concerns (SoC)**: Jede Komponente hat eine klar definierte Aufgabe, wodurch die Wartbarkeit und Erweiterbarkeit des Systems verbessert wird.
- **Erweiterbarkeit**: Neue Funktionen können durch Ergänzung zusätzlicher Interfaces integriert werden, ohne bestehende Module stark zu verändern.
- **Sicherheit**: Kritische Prozesse wie Passwort-Hashing und Benutzerverifizierung sind isoliert, um das System widerstandsfähiger gegen Angriffe zu machen.
- **Effizienz**: Die definierten Datenstrukturen ermöglichen eine schnelle Abfrage und Speicherung, indem oft genutzte Daten optimal organisiert werden.
- **Warum gibt es getrennte Interfaces für CRUD und Queries?** → Weil dies die Systemarchitektur flexibler macht und spezielle Abfragen optimiert werden können.

- **Warum wird TaskStatus nicht nur im RAM gespeichert?** → TaskStatus wird nicht nur im RAM, sondern auch persistent gespeichert, um Datenverlust bei einem unerwarteten Systemabsturz zu vermeiden. Obwohl häufige Änderungen eine effiziente Verarbeitung erfordern, wird eine hybride Lösung eingesetzt: TaskStatus wird im RAM für schnelle Zugriffe gehalten und in regelmäßigen Abständen in eine Datenbank geschrieben.

Informationsbedarf der Komponenten

Die definierten Datenstrukturen sind essenziell für die verschiedenen Systemkomponenten:

- **User**: Wird für Authentifizierung, Autorisierung und Identifikation benötigt.
- **Task**: Enthält die Hauptinformationen zu einer Aufgabe.
- **File**: Dient zur Speicherung und Verknüpfung von Dateien mit Aufgaben.
- **TaskStatus** und **Priority**: Standardisierte Status- und Prioritätsverwaltung.

Datenpersistenz und Speicherung

Die folgenden Datenstrukturen werden in einer relationalen Datenbank gespeichert:

- **User**: Enthält Benutzerinformationen wie ID, Name, E-Mail, Passwort-Hash und Rolle.
- **Task**: Speichert Aufgaben mit Titel, Beschreibung, Status, Priorität und zugewiesenem Benutzer.
- **File**: Repräsentiert hochgeladene Dateien mit Dateinamen, Metadaten und zugehöriger Aufgabe.

Dagegen werden die folgenden Strukturen nur temporär im Speicher gehalten:

- **TaskStatus**: TaskStatus wird im RAM für schnelle Zugriffe gehalten und in regelmäßigen Abständen in eine Datenbank geschrieben.
- **Priority**: Da ~~Priority vordefiniert ist und sich nicht dynamisch andert,~~ wird es in der Datenbank gespeichert, um sicherzustellen, dass es auch nach einem Systemneustart verfügbar bleibt.

komische Argumentation aber okay. Sie muss auf jeden Fall gespeichert werden

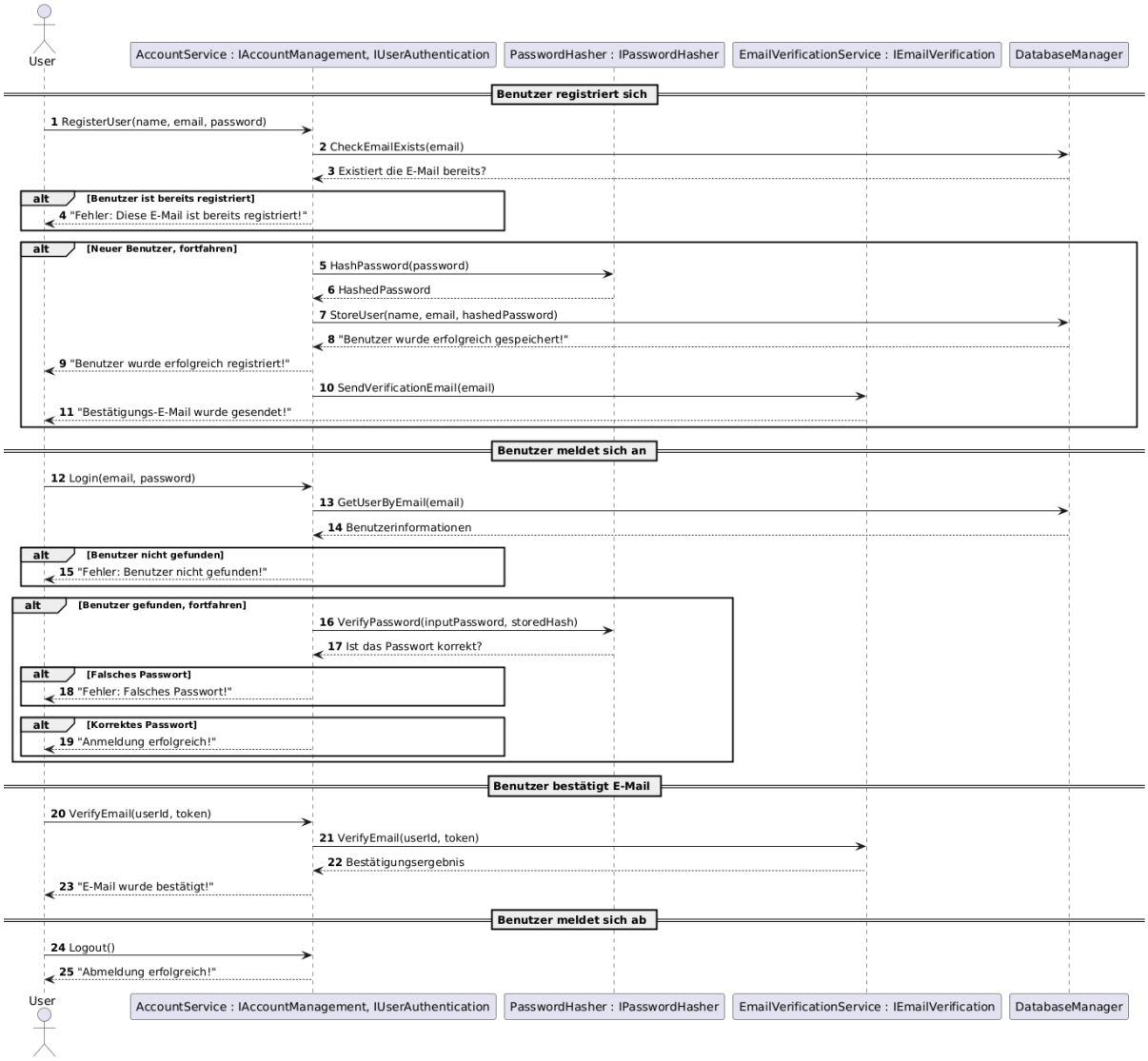
Die Entscheidung für die Speicherung dieser Daten in einer Datenbank basiert auf folgenden Aspekten:

- **Dauerhafte Speicherung**: Benutzer- und Aufgaben- und Prioritätsdaten müssen langfristig erhalten bleiben.
- **Schnelle Abfragen**: Die Struktur ermöglicht effiziente Suchen und Filteroperationen basierend auf Prioritäten.
- **Minimierung von Schreibzugriffen**: Häufig veränderte Daten wie `TaskStatus` bleiben im Arbeitsspeicher, um unnötige Datenbank-Updates zu vermeiden, während statische Daten wie Prioritäten direkt in der Datenbank verwaltet werden, um eine konsistente Datenstruktur sicherzustellen.
- **Datenintegrität**: Benutzer- und Aufgaben- und Prioritätsdaten müssen konsistent und sicher verwaltet werden.

2/2

Modellierung des verfeinerten Verhaltens

2.1.1.3 Benutzerkonto erstellen [US-1.1]



Weshalb wurde dieses Sequenzdiagramm gewählt?

Dieses **Sequenzdiagramm** wurde erstellt, um die **Benutzerverwaltungsprozesse eines Systems von Anfang bis Ende** darzustellen, einschließlich **Registrierung, Anmeldung, E Mail-Bestätigung und Abmeldung.** Diese Prozesse sind direkt miteinander verknüpft und bilden die Grundlage für die **Benutzerauthentifizierung und Kontoverwaltung.**

Anstatt jeden Prozess separat darzustellen, wurde ein **ganzheitliches Modell** gewählt, das die **gesamte Benutzerreise im System** in einem einzigen Diagramm vereint. Dadurch wird klar ersichtlich, **wie verschiedene Komponenten interagieren, welche Überprüfungen und Sicherheitskontrollen durchgeführt werden und wie das System auf erfolgreiche oder fehlerhafte Szenarien reagiert.**

Beschreibung des Diagramms

Das Diagramm zeigt die **zentralen Prozesse der Benutzerverwaltung**, welche durch folgende Interaktionen dargestellt werden:

Benutzerregistrierung (`RegisterUser`)

- Der Benutzer sendet eine Anfrage an den `AccountService`.
- `AccountService` prüft, ob die E-Mail bereits existiert (`CheckEmailExists` in `DatabaseManager`).
- Falls die E-Mail existiert, wird eine Fehlermeldung ausgegeben.
- Falls der Benutzer neu ist, wird das Passwort gehasht (`PasswordHasher`).
- Der Benutzer wird in `DatabaseManager` gespeichert (`StoreUser`).
- Eine Bestätigungs-E-Mail wird über `EmailVerificationService` gesendet.

Benutzeranmeldung (`Login`)

- `AccountService` sucht den Benutzer anhand der E-Mail (`GetUserByEmail` in `DatabaseManager`).
- Falls der Benutzer nicht existiert, wird eine Fehlermeldung gesendet.
- Falls der Benutzer existiert, wird das eingegebene Passwort überprüft (`VerifyPassword` in `PasswordHasher`).
- Falls das Passwort falsch ist, erhält der Benutzer eine Fehlermeldung.
- Falls das Passwort korrekt ist, erfolgt die Anmeldung erfolgreich.

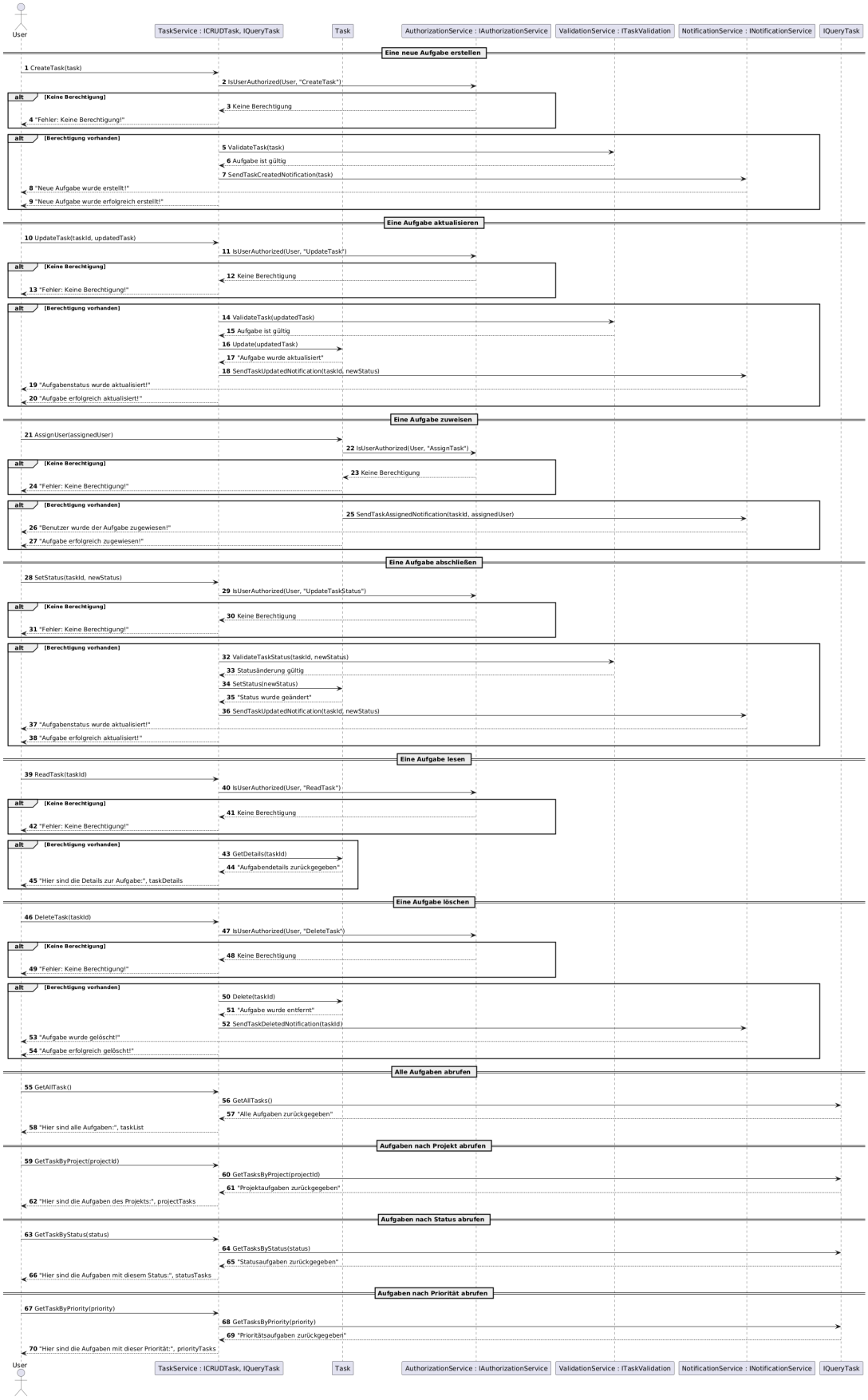
E-Mail-Bestätigung (`VerifyEmail`)

- `AccountService` leitet die Bestätigungsanfrage an `EmailVerificationService` weiter.
- `EmailVerificationService` verarbeitet die Anfrage und sendet das Bestätigungsergebnis an den Benutzer.

Abmeldung (`Logout`)

- Der Benutzer meldet sich über `AccountService` ab und erhält eine Bestätigungsmeldung.

2.1.2.3 Aufgabe erstellen [US-3.1]



Weshalb wurde dieses Sequenzdiagramm gewählt?

Dieses **Sequenzdiagramm** wurde entworfen, um die **zentralen Interaktionen** im System ganzheitlich darzustellen. Anstatt isolierte Prozesse einzeln darzustellen, haben wir uns bewusst dafür entschieden, **zusammenhängende und zentrale Abläufe** in einem Diagramm zu vereinen.

Der Grund für diese Entscheidung liegt darin, dass das **Task-Management-System** aus Prozessen besteht, die von mehreren Komponenten gemeinsam genutzt werden und voneinander abhängig sind. Die **Erstellung von Aufgaben, Statusänderungen und das Abrufen von Aufgaben** bilden die Kernfunktionen des Systems. Anstatt diese Prozesse als unabhängige Abläufe darzustellen, haben wir sie in einem einzigen Fluss integriert, um die **Gesamtlogik des Systems** klarer darzustellen.

Beschreibung des Diagramms

Das Diagramm zeigt die zentralen Prozesse der Aufgabenverwaltung, die durch folgende Interaktionen gekennzeichnet sind:

Erstellen einer Aufgabe (`CreateTask`)

- Der Benutzer sendet eine Anfrage an den `TaskService`.
- Die Berechtigungsprüfung erfolgt durch den `AuthorizationService`.
- Nach Freigabe wird die Aufgabe vom `ValidationService` überprüft.
- Eine Benachrichtigung wird über den `NotificationService` versendet.

Aktualisieren einer Aufgabe (`UpdateTask`)

- Ähnlicher Ablauf wie bei `CreateTask`, jedoch mit einer Aktualisierung der bestehenden Aufgabe.
- Die Änderungen werden gespeichert, und eine Benachrichtigung wird ausgelöst.

Zuweisen einer Aufgabe (`AssignUser`)

- Die Berechtigung des Benutzers wird durch den `AuthorizationService` überprüft.
- Die Zuweisung erfolgt, und der zugewiesene Benutzer wird benachrichtigt.

Abschließen einer Aufgabe (`SetStatus`)

- Die Berechtigung wird geprüft.
- Der Status der Aufgabe wird aktualisiert.
- Eine entsprechende Benachrichtigung wird gesendet.

Lesen einer Aufgabe (`ReadTask`)

- Der Benutzer ruft die Details einer Aufgabe ab.
- Nach der Berechtigungsprüfung gibt der `TaskService` die Aufgabendaten zurück.

Löschen einer Aufgabe (`DeleteTask`)

- Nach erfolgreicher Berechtigungsprüfung wird die Aufgabe entfernt.
- Der Benutzer erhält eine Bestätigung per Benachrichtigung.

Abrufen von Aufgaben (`GetAllTask`, `GetTaskByProject`, `GetTaskByStatus`, `GetTaskByPriority`)

- Der Benutzer kann Aufgaben nach verschiedenen Kriterien filtern.
- Die Abfragen werden ausgeführt, und die entsprechenden Listen zurückgegeben.

2.2 Tracing der User-Story (ID: 1.1) und relevante Anforderungen

Als neuer Nutzer möchte ich einen Account anlegen können, damit ich die Plattform nutzen und auf personalisierte Inhalte zugreifen kann.

Tracing der User-Story (ID: 3.1) und relevante Anforderungen

Als Teammitglied möchte ich Tasks erstellen können, damit ich die Arbeit in kleinere Einheiten aufteilen kann.

Tracing-Tabelle: Verknüpfung von User Story und Modellklassen

Klasse	**Verantwortlichkeit/Funktion**	**Relevante Anforderungen/User-Story**
AccountService	Verwaltet die Registrierung und Anmeldung von Nutzern.	User-Story 1.1: Registrierung API entwickeln.
PasswordHasher	Hashing von Passwörtern für mehr Sicherheit.	User-Story 1.1: Sichere Passwortspeicherung.
EmailVerificationService	Sendet und überprüft Bestätigungs-E-Mails.	User-Story 1.1: E-Mail-Verifikation der Registrierung.
DatabaseManager	Speichert und verwaltet Benutzerkontodaten.	User-Story 1.1: Speicherung der Benutzerdaten in der Datenbank.
User	Repräsentiert den Benutzer mit seinen Daten.	User-Story 1.1 / 3.1: Nutzer kann sich registrieren und Aufgaben zugewiesen bekommen.
TaskService	Verwaltung und Erstellung von Tasks.	User-Story 3.1: Erstellung und Verwaltung von Aufgaben.
IAuthorizationService	Prüft Nutzerrechte für Registrierung und Aufgabenmanagement.	User-Story 1.1 / 3.1: Berechtigungsprüfung für Nutzer- und Task-Management.
NotificationService	Benachrichtigungen bei Aufgaben- oder Registrierungsereignissen.	User-Story 3.1: Benachrichtigung bei Task-Erstellung.
Project	Verknüpft Tasks mit Projekten.	User-Story 3.1: Organisation von Aufgaben innerhalb eines Projekts.

Table 1: Zusammenführung der User Stories 1.1 und 3.1 mit den wichtigsten Klassen

Detaillierte Beschreibung der Implementierung

- Die Registrierung erfolgt über **AccountService**, der die gesamte Logik zur Nutzererstellung verwaltet.
- Passwortsicherheit wird durch **PasswordHasher** sichergestellt, der das Passwort vor der Speicherung hasht.

- E-Mail-Verifikation wird durch **EmailVerificationService** durchgeführt, der eine Bestätigungs-E-Mail sendet.
- Benutzerdaten werden von **DatabaseManager** gespeichert, um den Zugriff zu ermöglichen.
- Der Nutzer wird durch die **User** Klasse repräsentiert, die alle relevanten Benutzerinformationen speichert und mit der Task-Klasse verknüpft ist, um Aufgaben einem Benutzer zuzuweisen.
- CRUD-Operationen für Tasks werden durch **TaskService** ausgeführt, der ICRUDTask implementiert.
- Die Berechtigungsprüfung und Verwaltung der Benutzerrollen werden über den **IAuthorizationService** durchgeführt, um verschiedene Berechtigungsstufen zu ermöglichen.
- Benachrichtigungen bei der Erstellung oder Zuweisung einer Aufgabe werden durch **INotificationService** gesendet.
- Aufgaben sind mit Projekten verknüpft, sodass sie über die Klasse **Project** organisiert werden können.

2/2

3.1 Updates zu genutzten Technologien

In unserem Projekt gab es keine Änderungen an den genutzten Technologien. Wir verwenden weiterhin **Java** als Programmiersprache mit dem **Spring Boot** Framework sowie **MySQL** als Datenbank.

Allerdings haben wir unsere Entwicklungsumgebung von vorherigen Tools auf **IntelliJ IDEA** umgestellt, um eine effizientere Entwicklung und bessere Integration mit Spring Boot zu ermöglichen.Diese Änderung betrifft jedoch nicht das verwendete Framework, sondern lediglich das Entwicklungstool.

Da keine wesentlichen Änderungen an den genutzten Technologien vorgenommen wurden, bleibt unser technischer Stack unverändert.

1/1

3.2 Dokumentation der Codequalität

Abweichungen des Codes zur geplanten Architektur

a) Passwort-Hashing wird nicht korrekt genutzt

- `IPasswordHasher` ist zwar implementiert, wird aber nicht in `AccountService.registerUser()` genutzt.
- Dadurch werden Passwörter unverschlüsselt gespeichert.

b) Analyse der Ursachen

- Die Funktion wurde nicht priorisiert, sodass die Implementierung unvollständig blieb.
- Testdaten wurden direkt in Klartext gespeichert, um die Benutzerregistrierung zu überprüfen.

c) Planung für den nächsten Sprint

- `AccountService.registerUser()` sollte `passwordHasher.hashPassword(password)` aufrufen, bevor das Passwort gespeichert wird.
- Dadurch werden Passwörter sicher abgelegt und vor unbefugtem Zugriff geschützt.

a) E-Mail-Verifizierung nicht funktionsfähig

- `EmailVerificationService.sendVerificationEmail()` zeigt nur eine Nachricht im Terminal an, versendet aber keine echte E-Mail.
- `AccountService.registerUser()` ruft keine Verifikationsmethode auf.

b) Analyse der Ursachen

- Die Integration eines echten E-Mail-Services wurde aufgrund von Zeitmangel verschoben.
- Die Priorität lag auf der Benutzerregistrierung, nicht auf der Verifizierung.

c) Planung für den nächsten Sprint

- `EmailVerificationService.sendVerificationEmail()` sollte einen echten Mail-Service nutzen (z. B. `JavaMailSender`).
- `AccountService.registerUser()` muss nach erfolgreicher Registrierung die Verifizierungsmail auslösen.

a) Benutzer werden nicht dauerhaft gespeichert

- `DatabaseManager` speichert Benutzer in einer `ArrayList`, was bedeutet, dass nach einem Neustart des Servers alle Daten verloren gehen.

b) Analyse der Ursachen

- Es wurde eine einfache Implementierung für Testzwecke genutzt.
- Die Anbindung einer echten Datenbank wurde bisher nicht umgesetzt.

c) Planung für den nächsten Sprint

- `DatabaseManager` sollte eine SQL-Datenbankverbindung verwenden.
- `storeUser()` sollte mit JPA oder JDBC umgesetzt werden, damit Benutzer dauerhaft gespeichert bleiben.

a) Sicherheitskonfiguration für Admin-Zugriff fehlt

- Es gibt keine Sicherheitskonfiguration in Spring Boot, alle Endpunkte sind öffentlich zugänglich.

b) Analyse der Ursachen

- Sicherheitskonfiguration wurde nicht als höchste Priorität betrachtet.
- Die Implementierung der Benutzerverwaltung stand im Vordergrund.

c) Planung für den nächsten Sprint

- Spring Security sollte eingerichtet werden, um `/admin`-Endpunkte nur für Administratoren freizugeben.

Durchgeführte manuelle Tests und Testergebnisse

In diesem Sprint haben wir uns für den Einsatz manueller Tests entschieden, da sie eine detaillierte und flexible Überprüfung von spezifischen Anwendungsfällen ermöglichen. Besonders in Bereichen wie UI-Verhalten, Benutzerinteraktion und Fehlermeldungen sind manuelle Tests oft effektiver, da sie unerwartete Probleme besser aufdecken können. würde ich nicht so unterschreiben
Unsere Teststrategie kombiniert Black-Box-Tests zur Validierung der funktionalen Anforderungen mit White-Box-Tests zur Analyse der internen Systemlogik. Durch die Wahl von manuellen Tests konnten wir sicherstellen, dass kritische Prozesse, wie Benutzerregistrierung und E-Mail-Versand, ordnungsgemäß

funktionieren.
Die vollständige Dokumentation der durchgeführten manuellen Tests befindet sich in unserem GitLab-Repository. Diese enthält detaillierte Testergebnisse, die getesteten Funktionen sowie die entsprechenden Protokolle.
GitLab-Link zur Testdokumentation: Link

Tests sind für den Anfang okay, aber versucht noch etwas mehr die Randbedingungen abzutesten. Was passiert wenn eines der Felder leer gelassen wird? Was passiert bei Eingabe von Leerzeichen? Was passiert bei Eingabe von sehr langen Strings? Das führt dann auch dazu, dass man die Anforderungen noch präzisiert, zum Beispiel definiert man dann erlaubte Zeichen oder Längen.

3.3 Tracing

2,5/3

In unserem Projekt wurde ein systematisches Tracing zwischen dem UML-Diagramm und der Code-Implementierung durchgeführt. Dafür haben wir Annotationen direkt im Code verwendet, um jede Klasse und jedes Interface mit ihrem entsprechenden UML-Element zu verknüpfen. Zusätzlich wurde eine ausführliche Dokumentation in GitLab erstellt, die eine Tabelle mit der Zuordnung von UML-Komponenten zu Code-Klassen und -Interfaces enthält.
GitLab-Link zur Tracing: Link
Dieses Tracing gewährleistet eine transparente Nachvollziehbarkeit zwischen der Modellierung und der tatsächlichen Implementierung.

1/1

3.4 Laufender Prototyp

GitLab-Link zum Prototyp: Link
In diesem Projekt wird eine Webanwendung entwickelt, die es Benutzern ermöglicht, ein Konto zu erstellen und die Plattform zu nutzen, um auf personalisierte Inhalte zuzugreifen.

Benutzergeschichte

ID: US-1.1
Beschreibung: Als neuer Nutzer möchte ich ein Konto erstellen können, um die Plattform nutzen und auf personalisierte Inhalte zugreifen zu können.
Verfolgt zu: ANF-001

Voraussetzungen

IntelliJ IDEA oder eine andere IDE mit Java-Unterstützung, Gradle, Spring Boot, Java Development Kit (JDK 17 oder höher) Postman oder ein Terminal mit curl zur API-Testung, Git zur Versionskontrolle.

Installation und Kompilation

- **Java JDK Installation:** Laden Sie die neueste Version von Java JDK herunter und installieren Sie es auf Ihrem Computer.
- **Projektdateien vorbereiten:** Öffnen Sie IntelliJ IDEA und wählen Sie *File - Open*, um das heruntergeladene Projekt auszuwählen.
- **Abhängigkeiten installieren:** Öffnen Sie das Terminal und führen Sie im Wurzelverzeichnis des Projekts den Befehl *./gradlew build* aus, um die notwendigen Abhängigkeiten zu installieren.

Projekt Öffnen und Testen

1. Starten Sie IntelliJ IDEA.
2. Wählen Sie *File - Open* und navigieren Sie zum Projektordner, den Sie vorbereitet haben.
3. Wählen Sie den Projektordner und klicken Sie auf *OK*, um das Projekt in IntelliJ zu laden.
4. Stellen Sie sicher, dass alle Abhängigkeiten korrekt geladen und keine Bibliotheken fehlen.

Die Anleitung ist hier so okay. Schreibt das beim nächsten Mal in die README.md direkt beim Code.
Das ist eine übliche Vorgehensweise, weil man dann die Anleitung auch gleich dort hat wo man sie braucht.

Tests Durchführen

- **JUnit Tests:** Führen Sie alle vorbereiteten JUnit-Tests aus, um die Kernfunktionalitäten der Anwendung zu überprüfen. Dies können Sie tun, indem Sie im Projektexplorer mit der rechten Maustaste auf das Verzeichnis *test* klicken und *Run 'Tests in 'test"* wählen. Es gibt keine?
- **Integrationstests:** Führen Sie Integrationstests durch, um die Interaktion zwischen verschiedenen Teilen der Anwendung zu überprüfen.
- **Manuelle Tests:** Starten Sie die Anwendung durch Ausführen von *./gradlew bootRun* im Terminal. Öffnen Sie einen Webbrowser und navigieren Sie zu *http://localhost:8080/hello*, um die Benutzeroberfläche der Anwendung zu testen.

Test Szenarien

- **Benutzerregistrierung:** Ein neuer Benutzer kann durch eine HTTP-POST-Anfrage registriert werden. Dabei müssen Name, E-Mail und Passwort angegeben werden. Nach der Registrierung werden die Benutzerinformationen auch unter "http://localhost:8080/hello" angezeigt.
- **Anmeldung:** Ein registrierter Benutzer kann sich mit seiner E-Mail und seinem Passwort anmelden. Bei erfolgreicher Anmeldung erhält der Benutzer ein Authentifizierungs-Token.

Fehlerbehebung

Falls Probleme bei der Installation oder Nutzung auftreten, können folgende Maßnahmen helfen:

- Prüfen, ob der Server läuft.
- Überprüfen, ob alle Abhängigkeiten korrekt installiert wurden.
- Falls die Registrierung oder Anmeldung fehlschlägt, die Log-Dateien auswerten.
- Bei Problemen mit der Datenbank die Verbindungsparameter kontrollieren.

√

2/3

3.5 Abweichung Sprintplanung

Nicht oder nur teilweise umgesetzte User Stories

- **US-1.1 Benutzerkonto erstellen:**
  - Die Registrierung funktioniert grundsätzlich, jedoch gibt es Probleme mit der Speicherung der Benutzer in der Datenbank.
  - Das Passwort-Hashing wurde noch nicht vollständig implementiert.
  - Die E-Mail-Verifizierung sendet keine echte Bestätigungsmail.
  - Endpunkte zur Nutzerverwaltung sind nicht vollständig funktionsfähig und müssen überarbeitet werden.
  - Sicherheitsmechanismen zur Authentifizierung und Zugriffskontrolle sind noch nicht integriert.
- **US-3.1 Aufgabe erstellen:**
  - Die Modellierung wurde abgeschlossen, aber die Implementierung der Codebasis hat noch nicht begonnen.
  - CRUD-Operationen für Tasks wurden noch nicht entwickelt.

Es fehlt auch für beide User Stories ein UI!

Gründe für die Abweichungen

- **Technische Herausforderungen:**
  - Unsere Gruppe hatte Schwierigkeiten mit Java, insbesondere bei der Integration von Sicherheitsmechanismen und Datenbankanbindungen.
  - Bestimmte Funktionen, wie die E-Mail-Verifizierung und das Passwort-Hashing, erwiesen sich als komplexer als erwartet.
- **Teaminterne Schwierigkeiten:**
  - Unser Team war nicht vollständig auf einem einheitlichen Wissensstand, was die Entwicklungsarbeit verlangsamte.
  - Ein Teammitglied konnte krankheitsbedingt nicht aktiv teilnehmen, wodurch die Arbeitslast auf weniger Personen verteilt wurde.

Lessons Learned & Planung für den nächsten Sprint

Effizientere Aufgabenverteilung:

- Jedes Teammitglied muss sich aktiver einbringen, um den Rückstand aufzuholen.
- Mehr Fokus auf individuelle Vorbereitung und Einarbeitung in Java.

Mehr Unterstützung von Übungsleitern einholen:

- Falls technische Probleme auftreten, sollten wir schneller Hilfe suchen, anstatt zu lange eigenständig nach Lösungen zu suchen.

Bessere Sprint-Planung:

- Mehr realistische Einschätzungen der Komplexität der Aufgaben.
- Sicherstellen, dass alle Teammitglieder zumindest grundlegende Kenntnisse der benötigten Technologien haben, bevor neue Features implementiert werden.

Ziel für den nächsten Sprint

- Funktionierende Datenbankanbindung für Benutzerverwaltung.
- Passwort-Hashing vollständig integrieren
- E-Mail-Verifizierung implementieren, sodass Nutzer tatsächlich eine Bestätigungsmail erhalten.
- Beginn der Implementierung der Aufgabenverwaltung basierend auf dem bereits erstellten Modell.

1/1

Dokumentation individuelle Beiträge

Die folgende Tabelle dokumentiert den Beitrag der Teammitglieder zur Erstellung dieses Projektabgabedokuments für den Sprint.

	Verantwortliche Teammitglieder	**Anwesende während der Meetings**	**(Online-) Beiträge zum Inhalt durch**	**Korrektur-gelesen durch**
Sprint	Dogukan, Helin, Hüseyin, Eren, Cagla	Dogukan, Helin, Hüseyin, Eren, Cagla	Dogukan, Helin, Hüseyin, Eren, Cagla	Dogukan, Helin, Hüseyin, Eren, Cagla

Table 2: Beitrag der Teammitglieder zum Projektabgabedokument

Die nächste Tabelle dokumentiert die Beiträge der Teammitglieder zu den im Sprintbacklog durchgeführten Tasks.

Task ID	**Task Beschreibung/ Name**	**Teammitglied**	**Beitrag in %**
US-1.1-T1	Registrierung API entwickeln	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.1-T2	Passwort-Hashing implementieren (IPasswordHasher)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.1-T3	E-Mail-Verifizierung hinzufügen (IEmailVerification)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.1-T4	Benutzerrollenverwaltung erweitern (IAuthorizationService)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-3.1-T1	Task-Klasse implementieren	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-3.1-T2	CRUD-Operationen für Tasks entwickeln (ICRUDTask, TaskService)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-3.1-T3	Aufgaben einem Benutzer zuweisen (assignedUser: User)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-3.1-T4	Benachrichtigungssystem hinzufügen (INotificationService)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-3.1-T5	Datei-Upload für Aufgaben ermöglichen (File-Klasse)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-3.1-T6	Aufgaben mit Projekten verknüpfen (project: Project)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils

Table 3: Beitrag der Teammitglieder zu den Sprint-Tasks

Die für US-3.1 angegebenen Inhalte wurden ausschließlich für den Abschnitt 'Verfeinerte Architektur' erläutert und nicht für den Abschnitt 'Prototyp'.

20,5/20