Sprint

March 8, 2025

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.

Der vollständige Produkt-Backlog kann hier eingesehen werden:
Produktbacklog auf GitLab.

Das ist wohl noch nicht der richtige Link, sieht im GitLab aber in Ordnung aus

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

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.

Die für die Sprint-Planung relevanten Issues und Milestone sind unter den folgenden Links verfügbar:
İKİ LİNK
Sprint-Backlog auf GitLab.

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

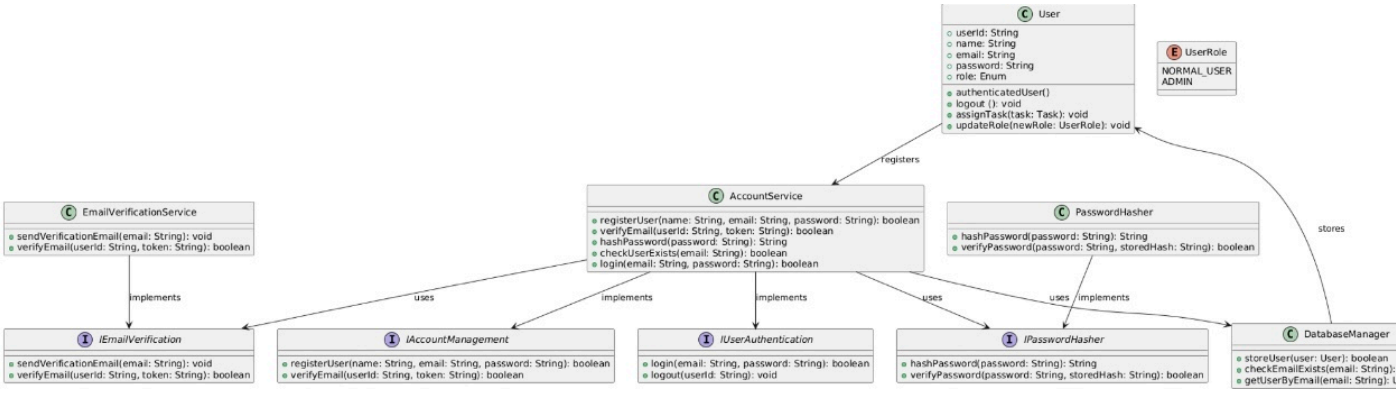
Die X sollen natürlich ersetzt werden. Also diese Kapitel soll in eurem Fall im Prinzip zweimal drin sein, einmal für die Benutzerregistrierung und einmal für die Taskerstellung. Das heißt, ihr macht erstmal alle Diagramme für die eine User Story und dann danach alle Diagramme für die nächste User Story. Die Überschriften könnten dann zum Beispiel lauten:2.1.1 Benutzerregistrierung und 2.1.2 Task-Erstellung.

2.1.X.1 Modellierung der verfeinerten Struktur

US-1.1 Benutzerkonto erstellen

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 relevante Design Patterns und die Verantwortlichkeiten der Klassen erläutert.

UML-Klassendiagramm
Modell der Benutzerkonto-Erstellung



- **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

Könnt ihr machen, wenn ihr wollt, das müsste man dann aber auch bei der Klasse im Diagramm sehen.

- **Singleton Pattern** (für DatabaseManager)
  - Sicherstellt, dass es nur eine Instanz der Datenbankverwaltung gibt.
  - Vorteil: Einheitliche Datenverwaltung und weniger Speicherverbrauch.
- **Strategy Pattern** (für PasswordHasher und EmailVerificationService)
  - Erlaubt flexiblen Austausch der Hashing- und Verifikationsstrategien.
  - Vorteil: Erhöhung der Sicherheit und Anpassungsfähigkeit.

Das ist im Diagramm nicht ersichtlich, aber auch nicht nötig hier. Lasst das raus.

- **Observer Pattern** (für Benachrichtigungen)
  Das `INotificationService` wurde eingeführt, um Benutzer über Änderungen an ihren Aufgaben in Echtzeit zu informieren. Dieses Service nutzt das **Observer Pattern**, um sicherzustellen, dass alle relevanten Nutzer benachrichtigt werden, sobald eine Aufgabe erstellt, aktualisiert oder abgeschlossen wird.

INotificationService existiert nicht im Diagramm und hier würde man auch kein Observer Pattern brauchen

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 und Design Patterns.
- 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.

US-3.1 Task-Erstellung

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 verwendete Design Patterns erläutert und die Verantwortlichkeiten der modellierten Komponenten beschrieben.

UML-Klassendiagramm
Modell der Task-Erstellung

nicht lesbar am Rand. Am besten das Diagramm nochmal so generieren lassen, dass die Task-Klasse über TaskService ist, oder das Bild horizontal ausrichten

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 Haupttaufgabe 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

0.1 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.
  - **Design Patterns** wie das `Factory Pattern` für Tasks ermöglichen eine einfache Erweiterung um neue Aufgaben- und Unteraufgabentypen.
  - 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.

2/3

2.1.X.2 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.

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.

- **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 ist TaskStatus nur im RAM gespeichert?** → Weil sich der Status oft ändert und ständige Datenbank-Updates vermieden werden sollen, um die Performance zu verbessern.

Das heißt dann aber auch, dass der TaskStatus weg ist, falls das System unerwartet abstürzt!

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.

Diese Strukturen gewährleisten eine optimale Datenverarbeitung und schnelle Abfragen.

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**: Da der Status einer Aufgabe sich oft ändert, wird er nur im Arbeitsspeicher gehalten, um häufige Datenbank-Updates zu vermeiden.
- **Priority**: Die Prioritäten einer Aufgabe sind vordefiniert und ändern sich nicht dynamisch, daher ist eine Speicherung in der Datenbank nicht erforderlich.

Nur weil es nicht geändert werden kann, heißt das ja nicht, dass es nicht gespeichert werden muss?

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

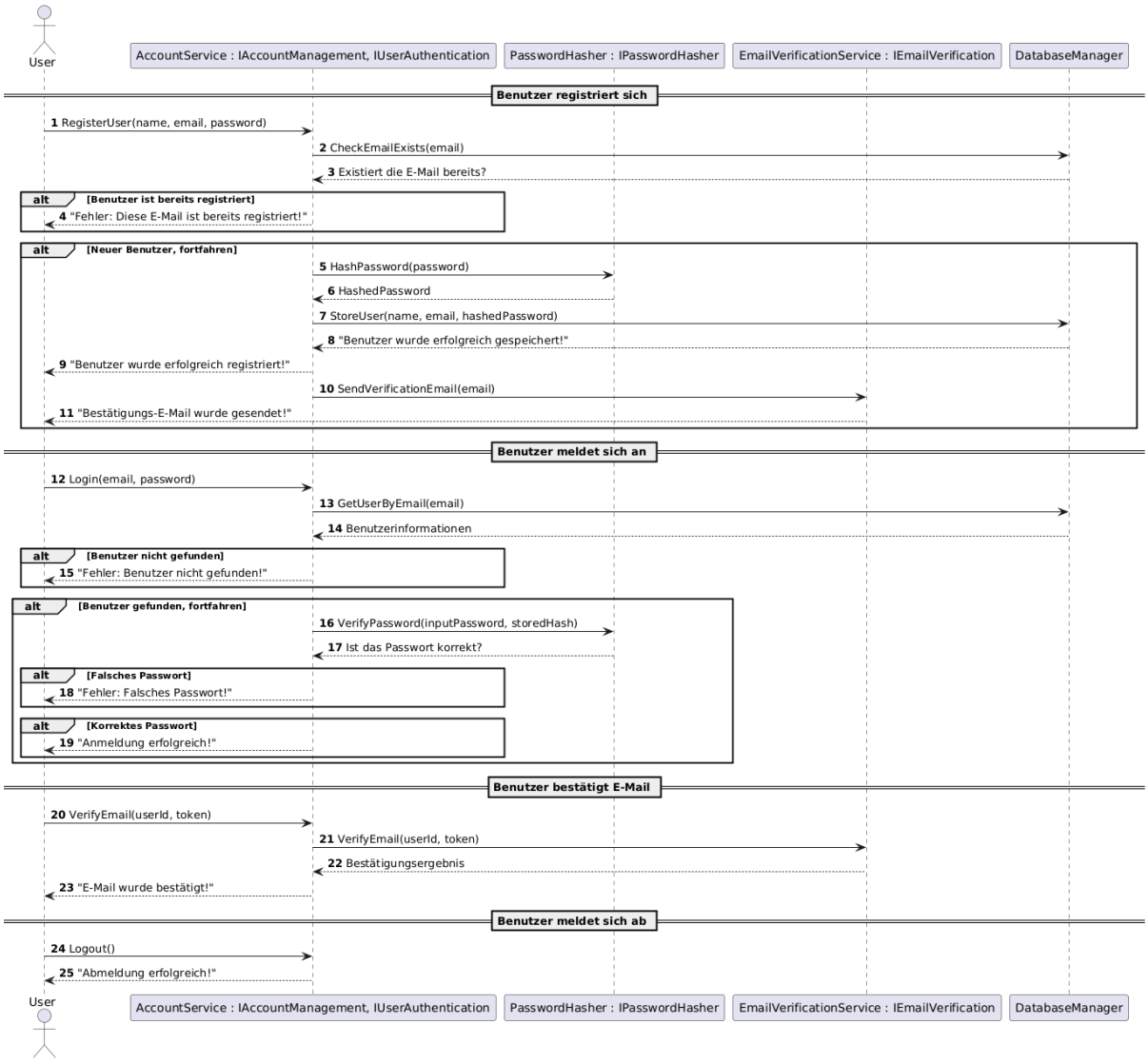
- **Dauerhafte Speicherung**: Benutzer- und Aufgabendaten müssen langfristig erhalten bleiben.
- **Schnelle Abfragen**: Die Struktur ermöglicht effiziente Suchen und Filteroperationen.
- **Minimierung von Schreibzugriffen**: Häufig veränderte Daten wie `TaskStatus` bleiben im Arbeitsspeicher, um unnötige Datenbank-Updates zu vermeiden.
- **Datenintegrität**: Benutzer- und Aufgabendaten müssen konsistent und sicher verwaltet werden.

[a4paper,12pt]article graphicx amsmath hyperref
Produkt-Backlog vor und nach dem Sprint March 8, 2025

1,5/2

2.1.X.3 Modellierung des verfeinerten Verhaltens

US-1.1 Benutzerkonto erstellen



dieses

Weshalb wurde ~~ein~~ Sequenzdiagramm gewählt?

Das Sequenzdiagramm wurde verwendet, um die **Kommunikation zwischen den Komponenten über die Interfaces** darzustellen. In diesem Fall interagieren die Systemkomponenten **AccountService, PasswordHasher, EmailVerificationService** und **DatabaseManager** miteinander, um die Benutzerverwaltung effizient umzusetzen.

Ein **Sequenzdiagramm eignet sich besonders gut**, da es die Reihenfolge und den Informationsfluss zwischen den Komponenten klar visualisiert. Alternativ könnten auch andere Modellierungstechniken wie **State Machine Diagramme oder Aktivitätsdiagramme** in Betracht gezogen werden. Diese wären hier jedoch weniger passend, da die Abläufe primär aus **Methodenaufrufen und Berechtigungsprüfungen** bestehen – anstatt aus komplexen Zustandsübergängen oder Workflows.

Da das Verhalten **linear und auf Anfragen/Aktionen ausgerichtet** ist, ermöglicht das Sequenzdiagramm eine präzise und nachvollziehbare Darstellung der Prozesse.

Hier ist die Frage nicht, warum ein Sequenzdiagramm gewählt wurde, sondern warum ihr genau dieses Sequenzdiagramm zeigt mit den dort abgebildeten Interaktionen.
So wie es aussieht, habt ihr allerdings alle möglichen Interaktionen abgebildet, während man normalerweise nur eine kleinere Auswahl gezeigt hätte. In eurem Fall könntet ihr argumentieren, dass ihr hier mal alles zeigen wolltet um einen möglichst detailreichen Einblick zu geben oder so.

Beschreibung des Diagramms

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

0.2 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.

0.3 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.

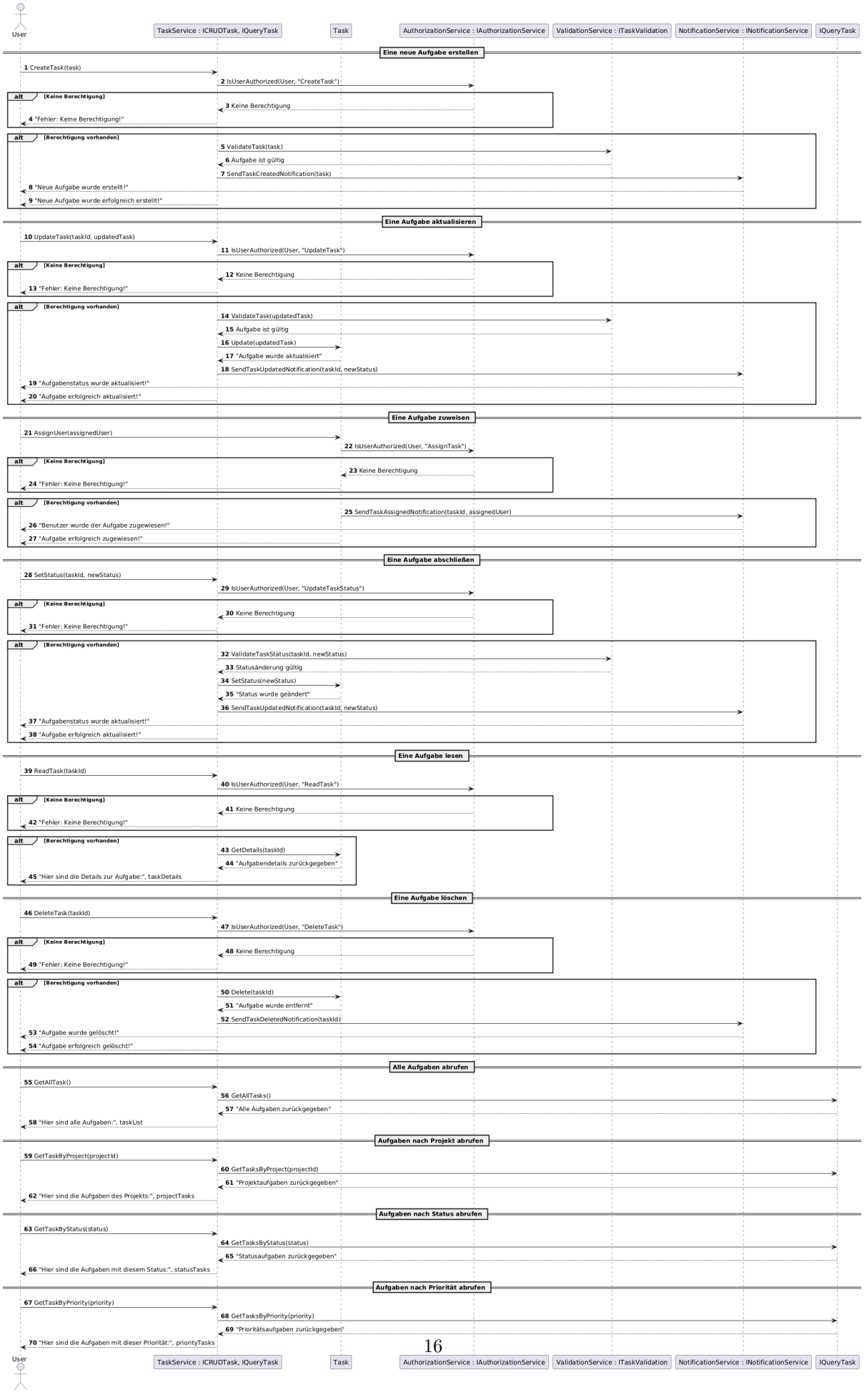
0.4 E-Mail-Bestätigung (`VerifyEmail`)

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

0.5 Abmeldung (`Logout`)

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

US-3.1 Aufgabe erstellen



dieses

Weshalb wurde ein Sequenzdiagramm gewählt?

Das Sequenzdiagramm veranschaulicht die Kommunikation zwischen den verschiedenen Systemkomponenten über ihre jeweiligen Schnittstellen. In diesem Fall interagieren **TaskService, AuthorizationService, ValidationService, NotificationService** und **Task** miteinander, um die Aufgabenverwaltung effizient umzusetzen.

Alternativ könnten auch andere Modellierungstechniken wie **State Machine Diagramme oder Aktivitätsdiagramme** verwendet werden. Diese wären hier jedoch weniger passend, da der Prozess vor allem durch **Methodenaufrufe und Berechtigungsprüfungen** geprägt ist – statt durch komplexe Zustandsübergänge oder parallele Workflows.

Da das Verhalten linear und **ereignisgesteuert** abläuft, ermöglicht das Sequenzdiagramm eine präzise und nachvollziehbare Darstellung der Abläufe.

Gleiches Problem wie oben

Beschreibung des Diagramms

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

0.6 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.

0.7 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.

0.8 Zuweisen einer Aufgabe (`AssignUser`)

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

0.9 Abschließen einer Aufgabe (`SetStatus`)

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

0.10 Lesen einer Aufgabe (`ReadTask`)

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

0.11 Löschen einer Aufgabe (`DeleteTask`)

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

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

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

1,5/2

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

macht hier lieber eine große Tabelle mit der Zuordnung für beide User Stories.
Hier muss auch nicht jede Klasse/Interface auftauchen, sondern nur die wichtigsten.

Als neuer Nutzer möchte ich einen Account anlegen können, damit ich die Plattform nutzen und auf personalisierte Inhalte zugreifen 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.
IAccountManagement	Definiert die Schnittstelle für Benutzerregistrierung.	User-Story 1.1: Verwaltung von Benutzerkonten.
IPasswordHasher	Stellt Funktionen für sicheres Hashing von Passwörtern bereit.	User-Story 1.1: Passwort sicher speichern (gehasht).
PasswordHasher	Implementiert das Hashing von Passwörtern.	User-Story 1.1: Sichere Passwortspeicherung.
EmailVerificationService	Sendet und überprüft Bestätigungs-E-Mails.	User-Story 1.1: Nutzer erhält eine E-Mail zur Bestätigung.
IEmailVerification	Definiert die Schnittstelle für E-Mail-Bestätigung.	User-Story 1.1: Bestätigung der Registrierung per E-Mail.
DatabaseManager	Speichert und verwaltet Benutzerkontodaten.	User-Story 1.1: Speicherung der Benutzerdaten in der Datenbank.
IAuthorizationService	Erweitert die Benutzerrollenverwaltung.	User-Story 1.1: Benutzerrollenverwaltung erweitern.
User	Repräsentiert den Benutzer und speichert dessen Daten.	User-Story 1.1: Nutzer kann sich mit gültigen Daten registrieren.

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.
- Benutzerrollen werden über **IAuthorizationService** verwaltet, um verschiedene Berechtigungsstufen zu erlauben.
- Der Nutzer wird durch die **User**-Klasse repräsentiert, die alle relevanten Benutzerinformationen speichert.

[a4paper,10pt]article [utf8]inputenc array booktabs
Tracing der User-Story (ID: 3.1) und relevante Anforderungen

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**
TaskService	Verwaltet die Erstellung und Verwaltung von Tasks.	Erstellung einer neuen Aufgabe.
ICRUDTask	Definiert Schnittstelle für CRUD-Operationen.	CRUD-Operationen für Tasks.
INotificationService	Sendet Benachrichtigungen bei Erstellung.	Benachrichtigung bei Erstellung.
Task	Repräsentiert eine einzelne Aufgabe.	Speicherung der Aufgabeninformationen.
User	Kann Aufgaben zugewiesen bekommen.	Aufgaben einem Benutzer zuweisen.
File	Ermöglicht Datei-Upload.	Datei-Upload für Aufgaben.
Project	Verknüpft Aufgaben mit Projekten.	Aufgaben mit Projekten verknüpfen.
IAuthorizationService	Prüft Berechtigungen zur Erstellung.	Berechtigungsprüfung.
ITaskValidation	Validiert die Aufgabe vor Speicherung.	Validierung der Aufgabe.
IQueryTask	Ermöglicht Aufgabenabfrage nach Kriterien.	Abrufen nach Kriterien.

Table 2: Verknüpfung von User Story mit Modellklassen

Detaillierte Beschreibung der Implementierung

- **Die Aufgabenerstellung** wird durch die Klasse `Task` verwaltet, die alle relevanten Attribute einer Aufgabe enthält.
- **CRUD-Operationen für Tasks** werden durch `TaskService` ausgeführt, der `ICRUDTask` implementiert.
- **Berechtigungsprüfung** wird durch `IAuthorizationService` durchgeführt.
- **Aufgabenvalidierung** erfolgt durch `ITaskValidation`, um sicherzustellen, dass die Aufgabe korrekt formatiert ist.
- **Benachrichtigungen** bei der Erstellung oder Zuweisung einer Aufgabe werden durch `INotificationService` gesendet.
- **Aufgaben können einem Benutzer zugewiesen werden**, indem `Task` eine Verbindung zur Klasse `User` herstellt.
- **Dateien können einer Aufgabe hinzugefügt werden**, indem die Klasse `File` verwendet wird.
- **Aufgaben sind mit Projekten verknüpft**, sodass sie über die Klasse `Project` organisiert werden können.
- **Aufgaben können nach bestimmten Kriterien abgefragt werden**, indem `IQueryTask` verwendet wird.

1,5/2