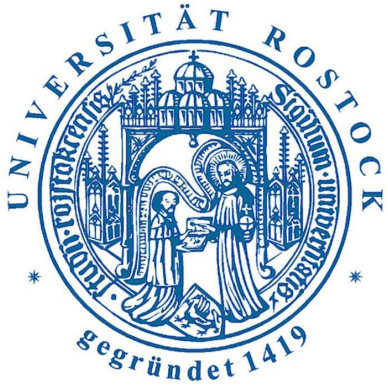


Softwaretechnik WS 2024-25
Projektabgabedokument Sprint 2

Übung-2 Team-1

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

1 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.

1.1 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-1.2 Benutzeranmeldung**
  Nutzer können sich mit E-Mail und Passwort anmelden.
- **US-1.3 Passwort zurücksetzen**
  Nutzer können ihr Passwort zurücksetzen, falls sie es vergessen haben.
- **US-3.1 Aufgabe erstellen**
  Nutzer können Aufgaben mit Titel und Beschreibung anlegen.
- **US-3.2 Aufgaben mit User Stories verlinken**
  Aufgaben können einer User Story zugeordnet werden.
- **US-3.3 Aufgaben priorisieren**
  Aufgaben können nach ihrer Wichtigkeit sortiert werden.
- **US-3.4 Aufgaben als "complete" oder "incomplete" markieren**
  Aufgaben können als erledigt oder offen markiert werden.

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 (**EmailVerificationService**).
- Benutzerrollenverwaltung wurde erweitert (**IAuthorizationService**).

Die sind doch genauso geblieben wie im letzten Sprint?

US-1.2 Erweiterungen

- Login-Logik wurde in **AccountService** implementiert.
- Passwort wird bei der Anmeldung verifiziert (**IPasswordHasher**).
- Sitzungsverwaltung nach erfolgreicher Anmeldung wurde eingeführt.

US-1.3 Erweiterungen

- Passwort-Reset-Formular im **ViewController** erstellt.
- PasswordResetToken (geplant) für zukünftige Implementierung vorbereitet.
- Anbindung an **EmailVerificationService** für Token-Verifikation geplant.

US-3.1 Erweiterungen

- CRUD-Operationen für Aufgaben (**TaskService**) implementiert.
- Aufgaben einem Benutzer zuweisbar (**assignedUser: User**).
- Aufgaben unterstützen Datei-Uploads (**TaskFile**).
- Aufgaben können Projekten zugeordnet werden (**Project**).

US-3.2 Erweiterungen

- Aufgaben wurden mit User Stories verlinkbar gemacht (Erweiterung der **Task**-Klasse).
- Anzeige der verlinkten User Story bei Aufgaben implementiert.

US-3.3 Erweiterungen

- Einführung von Prioritätsstufen (**Priority**-Enum: LOW, MEDIUM, HIGH, URGENT).
- Aufgaben können nach Priorität sortiert werden.

US-3.4 Erweiterungen

- Aufgabenstatus-Management eingeführt (**TaskStatus**: NEW, COMPLETED etc.).
- Aufgaben können als abgeschlossen oder offen markiert werden.

Begründung der Änderungen

Die vorgenommenen Änderungen dienten der Verbesserung der Sicherheit, Benutzerfreundlichkeit und Funktionalität der Plattform. Die Implementierung von Passwort-Hashing erhöhte die Sicherheit bei der Authentifizierung. Durch die Einführung der E-Mail-Verifizierung konnte sichergestellt werden, dass nur legitime Benutzer registriert sind. Die Erweiterung der Benutzerrollenverwaltung ermöglicht eine differenzierte Zugriffskontrolle. Zusätzlich wurden Aufgaben nun mit User Stories verlinkt, mit Prioritäten versehen und ihr Status kann verwaltet werden, um die Transparenz und Effizienz im Aufgabenmanagement zu erhöhen.

Hier ist auch fast alles wie im letzten Sprint

0,5 + 0,5 / 1

1.2 Sprint-Planung

Einleitung

Die Sprint-Planung basiert auf der Auswahl der wichtigsten User Stories für Sprint 2. 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 (**EmailVerificationService**)
- Benutzerrollenverwaltung erweitern (**IAuthorizationService**)

US-1.2 Benutzeranmeldung

- Login-Formular bereitstellen
- Implementierung der Login-Logik (**AccountService**)
- Passwort-Verifikation beim Login (**IPasswordHasher**)
- Erstellung einer Benutzersitzung nach erfolgreichem Login

US-1.3 Passwort zurücksetzen

- Passwort-Reset-Formular im ViewController anzeigen
- Implementierung der Passwort-Reset-Logik (**AccountService**)
- Vorbereitung der Token-Generierung (**PasswordResetToken**, geplant)
- Anbindung an E-Mail-Verifikation (**EmailVerificationService**)

US-3.1 Aufgabe erstellen

- Task-Klasse implementieren
- CRUD-Operationen für Tasks entwickeln (**TaskService**)
- Aufgaben einem Benutzer zuweisen (**assignedUser: User**)
- Datei-Upload für Aufgaben ermöglichen (**TaskFile**)
- Aufgaben mit Projekten verknüpfen (**Project**)

US-3.2 Aufgaben mit User Stories verlinken

- Aufgaben können einer User Story zugewiesen werden
- Anzeige der verlinkten User Story bei Aufgaben

US-3.3 Aufgaben priorisieren

- Einführung des Prioritäts-Enums (**Priority**)
- Aufgabenliste nach Priorität sortierbar machen

US-3.4 Aufgaben als "complete" oder "incomplete" markieren

- Einführung des Status-Feldes (**TaskStatus**)
- Aufgaben als abgeschlossen oder offen markieren

Um die User Stories und 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.

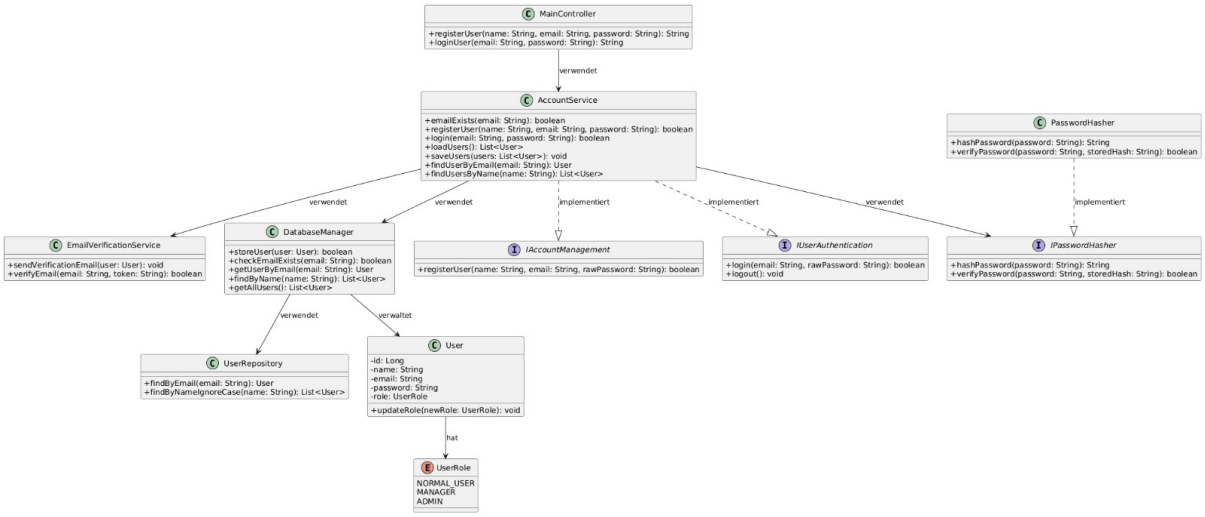
Table 1: Aufwandschätzung (Story Points) mit User Stories und Aufgaben

User Story	**Task**	**Helin**	**Hüseyin**	**Doğukan**	**Çağla**	**Eren**	**Ø Punkte**	**Begründung**
US-1.1	Registrierung API entwickeln	5	3	8	6	5	5.4	Backend könnte komplexer sein, API-Struktur
	Passwort-Hashing implementieren	2	3	3	2	3	2.6	Keine große Herausforderung, einfache Implementierung
	E-Mail-Verifizierung hinzufügen	5	4	5	3	4	4.2	SMTP-Server kann Probleme machen
US-1.2	Login-Funktion umsetzen	3	3	4	4	3	3.4	Standard-Login mit Hash-Prüfung
US-1.3	Passwort zurücksetzen	4	4	3	5	4	4.0	Token + E-Mail-Versand nötig
US-3.1	CRUD-Operationen für Tasks	8	8	13	10	9	9.6	Datenbankoperationen und Validierung
	Aufgaben einem Benutzer zuweisen	5	6	7	5	6	5.8	Benutzerrollen müssen klar definiert werden
	Datei-Upload für Aufgaben	6	7	8	7	7	7.0	Frontend-Backend-Integration muss getestet werden
	Subtasks erstellen/verwalten	6	7	6	5	6	6.0	Unteraufgaben + Status-Logik nötig
US-3.2	Verlinkung mit User Stories	5	4	4	5	5	4.6	Verbindung Task - StoryID
US-3.3	Priorisierung von Tasks	3	3	2	4	3	3.0	Enum + Sortierlogik im UI
US-3.4	Status als done/undone“ markieren	4	4	4	3	4	3.8	UI-Schalter + Status speichern

1+1/1

2.1 Benutzerregistrierung und Anmeldung [US-1.1 und US-1.2]

In diesem Abschnitt wird die verfeinerte Struktur für die User Stories US-1.1 Benutzerregistrierung und US-1.2 Benutzeranmeldung modelliert. Die Modellierung erfolgt mit einem gemeinsamen Klassendiagramm, das die wichtigsten Komponenten, Attribute, Methoden und Relationen enthält. Zusätzlich werden die Verantwortlichkeiten der Klassen erläutert und die Vereinfachungen gegenüber dem ursprünglichen Architekturentwurf dargestellt.



Modellierte Klassen und Verantwortlichkeiten

- **MainController**: Zuständig für die Entgegennahme der Benutzeranfragen zur Registrierung und Anmeldung sowie die Weiterleitung der Daten an den *AccountService*.
- **AccountService**: Verwaltung der Benutzerregistrierung, Anmeldung, Passwort-Hashing und E-Mail-Verifikation durch koordinierte Nutzung mehrerer Services.
- **EmailVerificationService**: Verwaltung der Versendung und Überprüfung von Bestätigungs-E-Mails zur Benutzerverifikation.
- **PasswordHasher**: Implementiert sicheres Hashen und Verifizieren von Passwörtern.
- **User**: Modelliert die Benutzerdaten mit Attributen wie *id*, *name*, *email*, *password* und *role*. Enthält zudem die Methode *updateRole()*.
- **UserRole (Enum)**: Definiert die möglichen Benutzerrollen (*NORMAL_USER*, *MANAGER*, *ADMIN*).
- **DatabaseManager**: Verwaltung der Benutzerpersistenz; speichert und ruft Benutzerobjekte ab.
- **UserRepository**: Zugriff auf die Benutzerdatenbank, insbesondere auf Benutzer über die E-Mail-Adresse.

Erweiterungen gegenüber dem ursprünglichen Entwurf

Im ursprünglichen Entwurf war vorgesehen, die Registrierung und Anmeldung direkt über den *MainController* zu verwalten. Im aktuellen Design wurde eine zusätzliche Service-Schicht eingeführt (*AccountService*), um die Geschäftslogik von der Controller-Schicht zu trennen und eine bessere Wartbarkeit sowie Erweiterbarkeit zu ermöglichen. Zusätzlich wurden spezialisierte Services wie *EmailVerificationService* und *PasswordHasher* eingeführt, um die Modularität und Sicherheit des Systems zu verbessern.

Verwendung von Design Patterns

- **Repository Pattern**: Zugriff auf persistente Daten über das *UserRepository*.
- **Strategy Pattern**: Verwendung eines separaten *IPasswordHasher*-Interfaces für Passwortsicherheitsstrategien.
- **Separation of Concerns**: Trennung der Verantwortlichkeiten zwischen Controller, Services und Datenhaltungskomponenten.

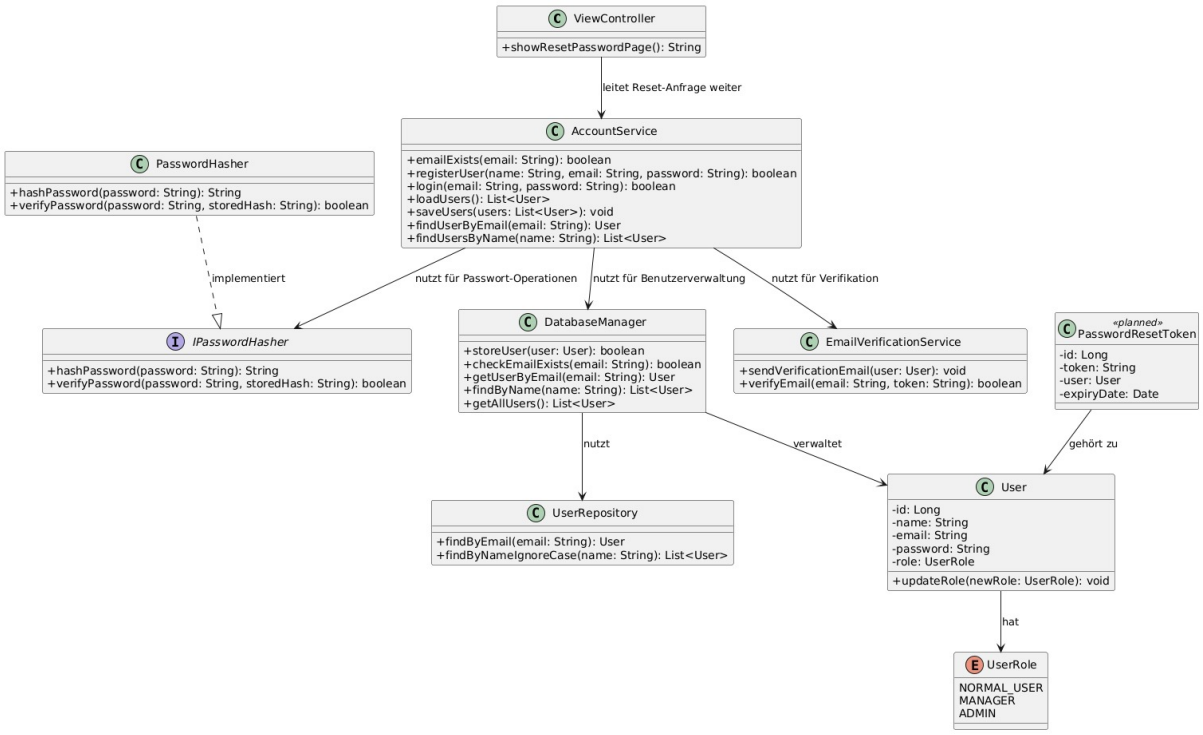
Begründung der Modellwahl braucht es hier nicht

Dieses Modell wurde gewählt, weil es folgende Vorteile bietet:

- **Hohe Sicherheit** durch modulare Passwort-Hashing- und E-Mail-Verifikationsprozesse.
- **Modularität und Erweiterbarkeit** durch Einführung spezialisierter Services.
- **Separation of Concerns**, was die Wartbarkeit erheblich verbessert.
- **Erweiterbarkeit** für zukünftige User Stories, z.B. einfachere Integration von zusätzlichen Authentifizierungsmechanismen oder Benutzerrollen.

Passwortzurücksetzung [US-1.3]

In diesem Abschnitt wird die verfeinerte Struktur für die User Story US-1.3 Passwortzurücksetzung modelliert. Die Modellierung erfolgt mit einem Klassendiagramm, das die wichtigsten Komponenten, Attribute, Methoden und Relationen im Zusammenhang mit dem Passwort-Reset-Prozess enthält. Zusätzlich werden die Verantwortlichkeiten der modellierten Klassen erläutert.



Modellierte Klassen und Verantwortlichkeiten

- **ViewController**: Zeigt die Benutzeroberfläche für die Anforderung einer Passwortzurücksetzung an.
- **AccountService**: Verwaltet das Versenden von Passwort-Reset-Links und die Authentifizierung der Benutzer. Verantwortlich für die Registrierung, Anmeldung und Passwortänderung auf Basis gültiger Tokens.
- **EmailVerificationService**: Verwaltung der E-Mail-Verifikation, insbesondere bei Registrierung und Passwortzurücksetzung.
- **PasswordHasher**: Sicheres Hashen und Verifizieren von Passwörtern mittels bcrypt.
- **PasswordResetToken** (*zukünftig geplant*): Repräsentiert einen Token für die Passwortzurücksetzung mit Tokenwert, Benutzerreferenz und Ablaufdatum.
- **User**: Modelliert die Benutzerdaten mit Attributen wie *id*, *name*, *email*, *password* und *role*.
- **UserRole (Enum)**: Definiert die möglichen Benutzerrollen (*NORMAL_USER*, *MANAGER*, *ADMIN*).
- **UserRepository**: Zugriff auf Benutzer über E-Mail-Adresse und Namenssuche.

Erweiterungen gegenüber dem ursprünglichen Entwurf

Zum Zeitpunkt dieses Sprints ist die Passwortzurücksetzungsfunktionalität nur teilweise implementiert. Das Frontend-Formular und die grundlegende Backend-Architektur sind vorbereitet, aber die vollständige

Implementierung der Token-Verwaltung, Ablaufprüfung und sicheren Passwortänderung wird in einem zukünftigen Sprint ergänzt.

Zusätzlich wurde die Benutzerverwaltung modularisiert, indem separate Services für Passwortsicherheit (*PasswordHasher*) und E-Mail-Verifikation (*EmailVerificationService*) eingeführt wurden.

Verwendung von Design Patterns

- **Strategy Pattern**: Verwendung eines separaten *IPasswordHasher*-Interfaces für unterschiedliche Passwort-Hashing-Strategien.
- **Separation of Concerns**: Trennung der Verantwortlichkeiten zwischen Frontend (ViewController), Authentifizierungslogik (AccountService) und Sicherheit (PasswordHasher, EmailVerificationService).

Begründung der Modellwahl

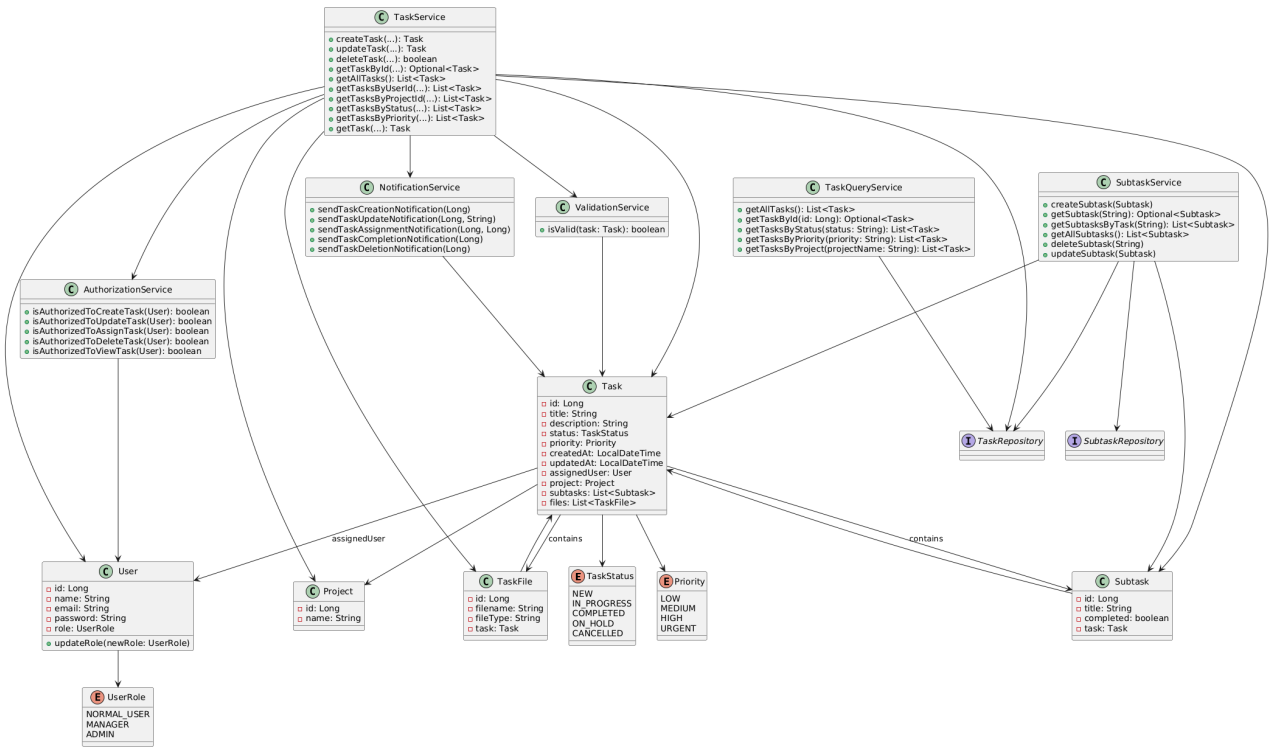
Dieses Modell wurde gewählt, weil es folgende Vorteile bietet:

- **Erhöhte Sicherheit** durch die Nutzung von zeitlich begrenzten Tokens und sicherer Passwort-Hashing-Methoden.
- **Erweiterbarkeit** durch separate, unabhängige Services für Authentifizierung, Token-Verwaltung und E-Mail-Verifikation.
- **Trennung der Verantwortlichkeiten**, was die Wartbarkeit und Weiterentwicklung der Lösung erleichtert.

Task-Erstellung [US-3]

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 mit den Attributen `id`, `name`, `email`, `password` und `role`. Verantwortlichkeiten:
  - Verwaltung von Benutzerinformationen.
  - Rollenverwaltung über `updateRole()`.
  - Aufgaben können Benutzern zugewiesen werden (als `assignedUser`).
- **Task**: Zentrale Entität zur Aufgabenverwaltung mit Attributen wie `id`, `title`, `description`, `status`, `priority`, `createdAt`, `updatedAt`, `assignedUser`, `project`, `subtasks`, `files`. Verantwortlichkeiten:
  - Verwaltung des Aufgabenstatus (`TaskStatus`).
  - Verknüpfung mit User, Project, Subtasks und TaskFile.
  - Speichert Zeitstempel für Erstellung und Aktualisierung.
- **Project**: Organisiert Aufgaben innerhalb eines Projekts. Verantwortlichkeiten:
  - Bündelung und Verwaltung von `Task`-Objekten.
- **Subtask**: Unteraufgabe mit `id`, `title`, `completed` und Referenz zur übergeordneten `Task`. Verantwortlichkeiten:
  - Verwaltung kleinerer Arbeitseinheiten innerhalb einer Aufgabe.
  - Fortschrittskontrolle der Hauptaufgabe.
- **TaskFile**: Repräsentiert hochgeladene Dateien zu Aufgaben. Verantwortlichkeiten:
  - Speicherung von Datei-Metadaten wie `filename`, `fileType`.
  - Verknüpfung mit zugehöriger `Task`.
- **TaskService**: Hauptkomponente zur Aufgabenverwaltung. Verantwortlichkeiten:
  - Erstellung, Aktualisierung und Löschung von Aufgaben.
  - Filterung von Aufgaben nach Benutzer, Projekt, Status, Priorität.
- **SubtaskService**: Verwaltung von Subtasks. Verantwortlichkeiten:
  - CRUD-Operationen auf Unteraufgaben.
- **TaskQueryService**: Bietet spezialisierte Abfragen für Aufgaben. Verantwortlichkeiten:
  - Suche nach Projektname, Priorität, Status, Benutzer.
- **NotificationService**: Versendet Benachrichtigungen bei Task-Ereignissen. Verantwortlichkeiten:
  - Benachrichtigungen zu Erstellung, Zuweisung, Änderung, Abschluss, Löschung.
- **ValidationService**: Prüft die Gültigkeit von Aufgaben. Verantwortlichkeiten:
  - Validierung der Aufgabenstruktur und Pflichtfelder.
- **AuthorizationService**: Prüft Berechtigungen von Nutzern im Task-Kontext. Verantwortlichkeiten:
  - Autorisierungsprüfung für Create, Update, Assign, Delete, View.

Begründung der Modellwahl

Flexibilität, Erweiterbarkeit und Software Design Prinzipien

Die Modellierung basiert auf bewährten Prinzipien der Softwarearchitektur zur Erreichung von Modularität, Wartbarkeit und Erweiterbarkeit.

- **Flexibilität:**
  - Separate Komponenten wie `TaskService`, `QueryService` und `NotificationService` ermöglichen flexible Kombinationen und Erweiterungen.
  - Subtasks und Dateien können unabhängig von der Hauptaufgabe verwaltet werden.
- **Erweiterbarkeit:**
  - Neue Funktionalitäten können über Services und Repositories ergänzt werden, ohne bestehende Klassen zu verändern.
  - Die Enum-Klassen `Priority` und `TaskStatus` ermöglichen einfache Erweiterungen.
- **Software Design Prinzipien:**
  - **Separation of Concerns (SoC):** Klare Trennung zwischen Geschäftslogik (z.B. `TaskService`), Datenzugriff (`Repository`), Validierung, Benachrichtigung und Berechtigung.
  - **Single Responsibility Principle (SRP):** Jede Klasse ist für genau eine Aufgabe zuständig (z.B. Validierung, Autorisierung, Notification).
  - **Open/Closed Principle (OCP):** Das System erlaubt Erweiterungen (z.B. neue Task-Typen oder Filter), ohne bestehende Komponenten zu verändern.
  - **Dependency Inversion Principle (DIP):** Zugriff erfolgt über Interfaces wie `TaskRepository`, was zu loser Kopplung und Testbarkeit führt.

Diese Architektur erlaubt es, das System problemlos an neue Anforderungen (z.B. Rollen, Aufgabenarten, Benachrichtigungswege) anzupassen, ohne das Gesamtsystem neu strukturieren zu müssen.

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

- **US1-1, 1-2: Benutzerregistrierung und Anmeldung** Als Nutzer möchte ich ein Konto erstellen, um auf die Plattform zugreifen zu können.
  - **IAccountManagement**
    - `registerUser(name, email, rawPassword)`
      Erstellt einen neuen Benutzer, speichert seine Daten in der Datenbank und sendet ggf. eine Bestätigungsmail.
  - **IUserAuthentication**
    - `login(email, rawPassword)`
      Authentifiziert einen Benutzer mit E-Mail und Passwort.
    - `logout()`
      Beendet die aktive Benutzersitzung.
  - **IPasswordHasher**
    - `hashPassword(password)`
      Erstellt einen sicheren Hash aus dem Passwort zur Speicherung.
    - `verifyPassword(password, storedHash)`
      Überprüft, ob ein eingegebenes Passwort mit dem gespeicherten Hash übereinstimmt.

IAccountManagement (Implementiert durch: AccountService)

```
public interface IAccountManagement {
    boolean registerUser(String name, String email, String password);

}
```

- `registerUser(name, email, password)` – Erstellt einen neuen Benutzer und speichert ihn in der Datenbank.

IUserAuthentication (Implementiert durch: AccountService)

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

- `login(email, rawPassword)` – Authentifiziert einen Benutzer anhand der Zugangsdaten.
- `logout()` – Beendet die Benutzersitzung.

IPasswordHasher (Implementiert durch: PasswordHasher)

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

- `hashPassword(password)` – Wandelt ein Passwort in einen sicheren Hash um.
- `verifyPassword(password, storedHash)` – Vergleicht das Passwort mit dem gespeicherten Hash.

User Story – Interface Zuordnung

US-3: Aufgaben verwalten Als Nutzer möchte ich Aufgaben erstellen, bearbeiten und verwalten, um meine Arbeit zu organisieren.

- **TaskRepository**
  - `findByAssignedUserId(userId)` Gibt alle Aufgaben zurück, die einem bestimmten Benutzer zugewiesen sind.
  - `findByProjectId(projectId)` Ruft alle Aufgaben ab, die mit einem bestimmten Projekt verbunden sind.
  - `findByProject(project)` Ruft alle Aufgaben ab, die dem gegebenen Projektobjekt zugeordnet sind.
  - `findByStatus(status)` Filtert Aufgaben basierend auf ihrem aktuellen Status.
  - `findByPriority(priority)` Gibt Aufgaben zurück, die eine bestimmte Priorität haben.
- **SubtaskRepository**
  - `findByTask(task)` Gibt alle Subtasks zurück, die einer bestimmten Aufgabe zugeordnet sind.

TaskRepository (Implementiert durch: Spring Data JPA)

```
public interface TaskRepository extends JpaRepository<Task, Long> {
    List<Task> findByAssignedUserId(Long userId);
    List<Task> findByProjectId(Long projectId);
    List<Task> findByProject(Project project);
    List<Task> findByStatus(TaskStatus status);
    List<Task> findByPriority(Priority priority);
}
```

- `findByAssignedUserId(userId)` – Gibt alle Tasks zurück, die einem bestimmten Nutzer zugewiesen sind.
- `findByProjectId(projectId)` – Gibt alle Tasks eines bestimmten Projekts anhand der Projekt-ID zurück.
- `findByProject(project)` – Gibt alle Tasks zurück, die zu einem gegebenen Projekt gehören.
- `findByStatus(status)` – Gibt alle Tasks mit dem angegebenen Status zurück.
- `findByPriority(priority)` – Gibt alle Tasks mit der angegebenen Priorität zurück.

SubtaskRepository (Implementiert durch: Spring Data JPA)

```
public interface SubtaskRepository extends JpaRepository<Subtask, Long> {
    List<Subtask> findByTask(Task task);
}
```

- `findByTask(task)` – Gibt alle Subtasks zurück, die zu einer bestimmten Task gehören.

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.
- **IPasswordHasher**: Verantwortlich für die sichere Speicherung von Passwörtern durch Hashing-Mechanismen.
- **IUserAuthentication**: Handhabt die Anmeldung und Abmeldung von Benutzern, um Sitzungen zu verwalten.
- **TaskRepository**: Zuständig für den Datenbankzugriff auf Aufgaben. Ermöglicht das Filtern nach Benutzer, Projekt, Status oder Priorität.
- **SubtaskRepository**: Ermöglicht das Abrufen aller Subtasks, die einer bestimmten Aufgabe zugeordnet sind.

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**: Durch die Verwendung spezieller Interfaces wie IPasswordHasher und IUserAuthentication werden sicherheitskritische Prozesse (z.B. Passwort-Hashing, Authentifizierung) isoliert behandelt.
- **Effizienz**: Die definierten Datenstrukturen und Repository-Schnittstellen ermöglichen eine gezielte und performante Abfrage von Aufgaben, z.B. über TaskRepository-Methoden wie findByStatus(...) oder findByPriority(...).
- **Warum gibt es getrennte Interfaces für CRUD und Queries?** → Interfaces wie TaskRepository und SubtaskRepository sind auf Datenabfragen spezialisiert und erlauben eine flexible und performante Umsetzung, ohne die Basisfunktionen zu überladen.

- **Warum wird TaskStatus nicht nur im RAM gespeichert?** → Obwohl TaskStatus für schnellen Zugriff im RAM gehalten wird, wird es zusätzlich regelmäßig persistiert, um Datenverlust bei Systemfehlern 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**: Für Authentifizierung, Autorisierung und Verwaltung von Accounts (IAccountManagement, IUserAuthentication).
- **Task**: Repräsentiert die Kerninformationen zu einer Aufgabe, verwaltet durch TaskRepository.
- **File**: Verknüpft Dateien mit Aufgaben.
- **TaskStatus** und **Priority**: Standardisierte Status- und Prioritätsverwaltung.

Datenpersistenz und Speicherung

Unsere Architektur unterscheidet zwischen persistenten und temporären Daten, um sowohl Leistung als auch Datensicherheit zu gewährleisten.

Persistente Speicherung in der Datenbank
Folgende Strukturen werden dauerhaft in einer relationalen Datenbank gespeichert:

- **User**: Enthält Informationen wie ID, Name, E-Mail, Passwort-Hash und Rolle. Grundlage für Authentifizierung und Autorisierung.
- **Task**: Speichert Aufgaben mit Titel, Beschreibung, Status, Priorität und dem zugewiesenen Benutzer.
- **File**: Repräsentiert Dateien mit Metadaten und Verknüpfung zur zugehörigen Aufgabe.

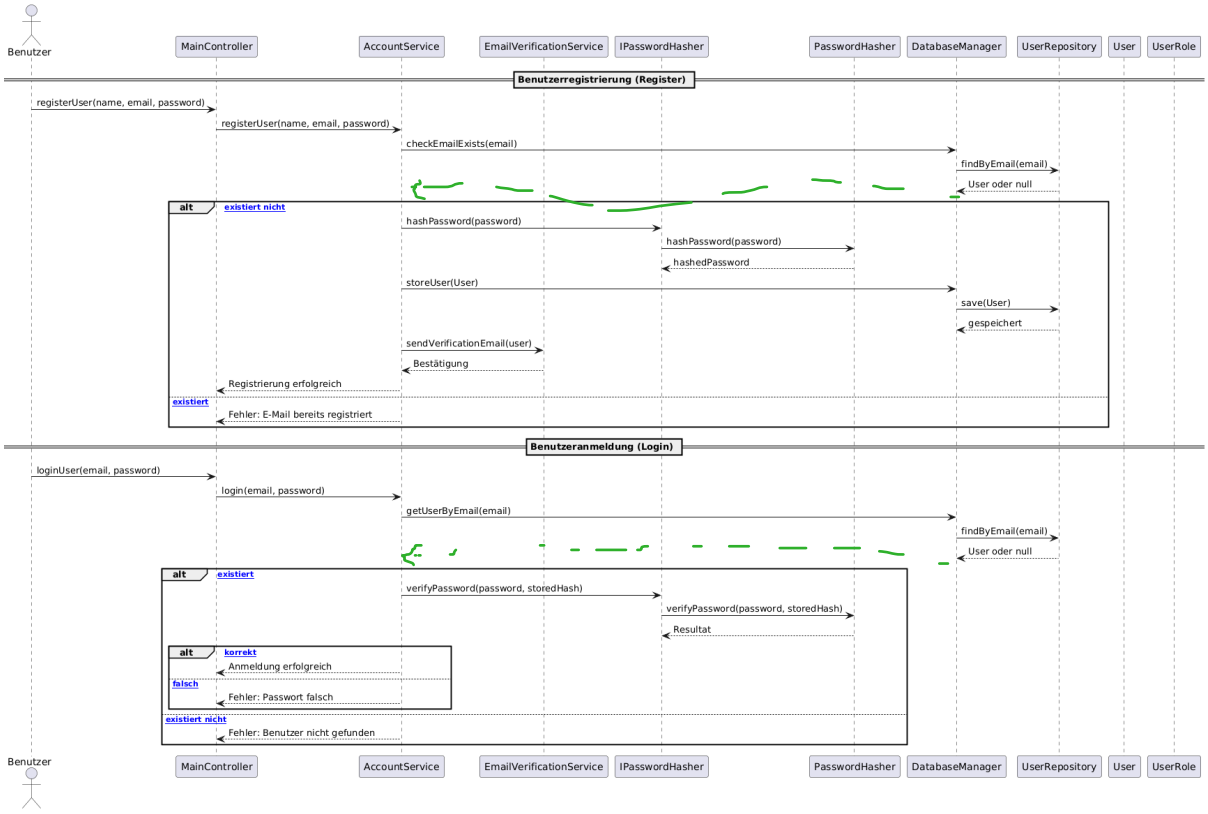
Dagegen werden die folgenden Strukturen nur temporär im Speicher gehalten:

- **TaskStatus**: TaskStatus wird im RAM für schnelle Zugriffe gehalten und in regelmäßigen Abständen in eine Datenbank geschrieben.
- **Priority**: Die Prioritätsinformationen sind integraler Bestandteil der Aufgabenstruktur und werden daher in der Datenbank gespeichert. Dies ermöglicht konsistente Verknüpfungen mit den jeweiligen Aufgaben und erlaubt gezielte Abfragen und Filterfunktionen auf Basis der Priorität. Eine persistente Speicherung stellt außerdem sicher, dass das System auch nach einem Neustart auf vollständige und konsistente Aufgabeninformationen zugreifen kann – inklusive der zugewiesenen Priorität.

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

- **Dauerhafte Speicherung**: Benutzer-, Aufgaben- und Prioritätsdaten sind grundlegend für das System und müssen auch nach einem Neustart verfügbar bleiben.
- **Schnelle Abfragen**: Die Struktur ermöglicht effiziente Suchen und Filteroperationen basierend auf Prioritäten.
- **Minimierung von Schreibzugriffen**: Daten wie TaskStatus, die sich häufig ändern, werden im RAM gehalten und nur periodisch in die Datenbank geschrieben, um unnötige Schreiboperationen zu vermeiden.
- **Datenintegrität**: Persistente Daten wie User, Task und Priority werden strukturiert und sicher verwaltet, um Konsistenz im gesamten System zu gewährleisten.

Benutzerregistrierung und Anmeldung [US-1.1 und US-1.2]



Warum wurde dieses Sequenzdiagramm gewählt?

Dieses Sequenzdiagramm wurde erstellt, um die Benutzerverwaltungsprozesse des Systems – Benutzerregistrierung und Benutzeranmeldung – von Anfang bis Ende darzustellen. Die Modellierung erfolgt in einem einzigen zusammenhängenden Ablaufdiagramm, das die zentralen Interaktionen zwischen den Komponenten *MainController*, *AccountService*, *IPasswordHasher*, *PasswordHasher*, *EmailVerificationService*, *DatabaseManager*, *UserRepository* und *User* zeigt.

Dadurch wird deutlich, wie Benutzeraktionen in der Anwendung verarbeitet werden, welche Prüfungen (z. B. E-Mail-Existenz, Passwortüberprüfung) durchgeführt werden und wie das System auf verschiedene Szenarien – erfolgreiche und fehlerhafte Anfragen – reagiert.

Das Sequenzdiagramm folgt der tatsächlichen Implementierung im Code und stellt sicher, dass die Modellierung konsistent mit der Systemarchitektur ist.

Beschreibung des Diagramms

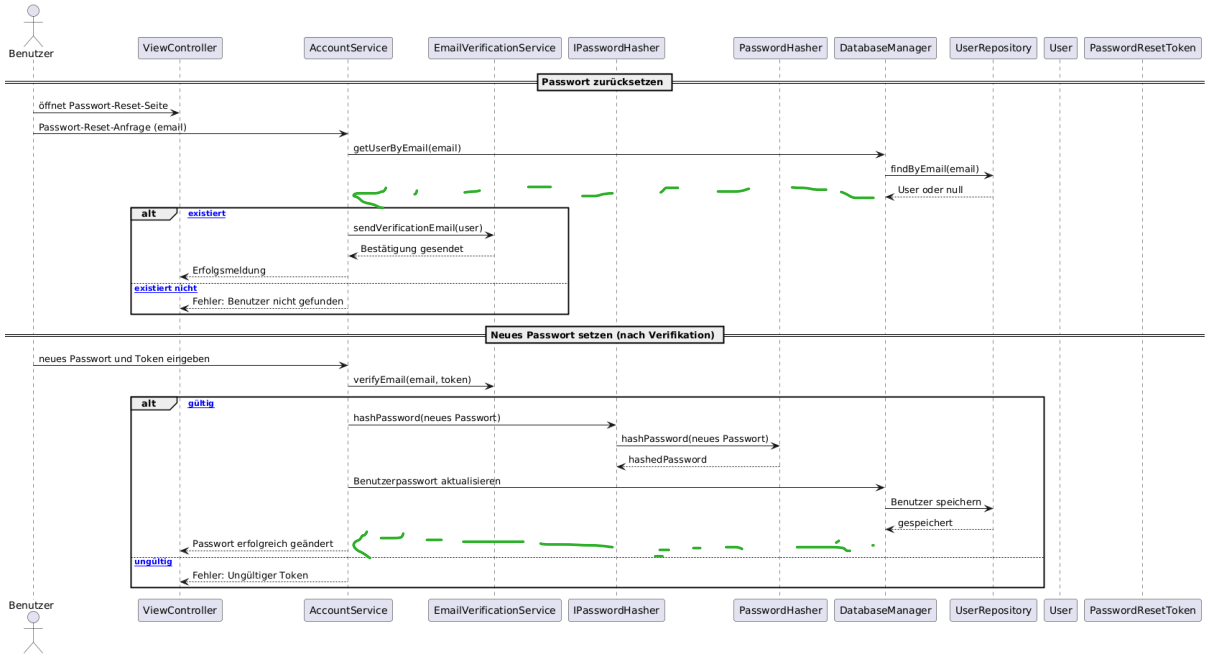
Das Diagramm zeigt die zentralen Prozesse der Benutzerverwaltung:

- **Benutzerregistrierung (RegisterUser)**
  - Der Benutzer sendet eine Anfrage an den *MainController* mit Name, E-Mail und Passwort.
  - *MainController* leitet die Anfrage an den *AccountService* weiter.
  - *AccountService* prüft über *DatabaseManager* und *UserRepository*, ob die E-Mail bereits existiert.
  - Falls die E-Mail existiert, wird eine Fehlermeldung zurückgegeben.
  - Falls die E-Mail neu ist:

        - Das Passwort wird über *IPasswordHasher* und *PasswordHasher* gehasht.
        - Die Benutzerinformationen werden über *DatabaseManager* und *UserRepository* gespeichert.
        - Eine Verifikations-E-Mail wird über den *EmailVerificationService* versendet.
    * Eine Erfolgsmeldung wird an den Benutzer zurückgegeben.
- **Benutzeranmeldung (Login)**
    * Der Benutzer sendet seine E-Mail und Passwort an den *MainController*.
    * *MainController* leitet die Anfrage an den *AccountService* weiter.
    * *AccountService* sucht den Benutzer über *DatabaseManager* und *UserRepository*.
    * Falls der Benutzer nicht existiert, wird eine Fehlermeldung gesendet.
    * Falls der Benutzer existiert:
        - Das eingegebene Passwort wird über *IPasswordHasher* und *PasswordHasher* mit dem gespeicherten Hash verglichen.
        - Bei korrektem Passwort erfolgt die Anmeldung; bei falschem Passwort wird eine Fehlermeldung gesendet.

Das ist in Ordnung so, idealerweise ist das aber ein zusammenhängender kürzerer Text, der den Inhalt des Diagramms etwas abstrakter zusammenfasst. Das Aufschreiben jedes Schrittes ist ja im Prinzip unnötig, weil ich das ja auch genauso im Diagramm sehen kann.

Passwortzurücksetzung [US-1.3]



Warum wurde dieses Sequenzdiagramm gewählt?

Dieses Sequenzdiagramm wurde erstellt, um den Prozess der Passwortzurücksetzung im System darzustellen – von der Anforderung eines Reset-Links bis zur Änderung des Passworts nach erfolgreicher Verifikation. Die Modellierung erfolgt anhand der zentralen Interaktionen zwischen den Komponenten *ViewController*, *AccountService*, *EmailVerificationService*, *IPasswordHasher*, *PasswordHasher*, *DatabaseManager*, *UserRepository*, *User* und dem geplanten *PasswordResetToken*.

Dadurch wird klar ersichtlich, welche Schritte erforderlich sind, um die Identität des Benutzers zu überprüfen, das Passwort sicher zu aktualisieren und mögliche Fehlerszenarien zu behandeln.

Das Sequenzdiagramm spiegelt die aktuelle Implementierung sowie geplante Erweiterungen wider und stellt sicher, dass die Modellierung konsistent mit der Systemarchitektur und zukünftigen Entwicklungen ist.

Beschreibung des Diagramms

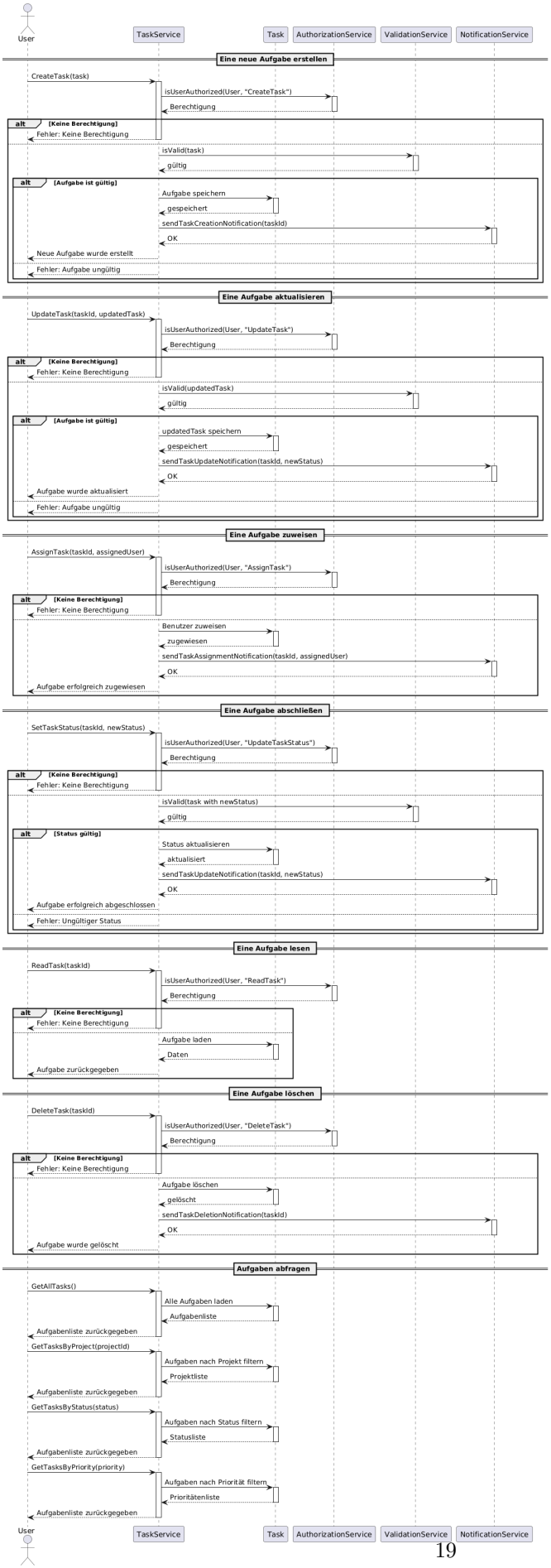
Das Diagramm zeigt die zentralen Prozesse der Passwortzurücksetzung:

- **Anforderung eines Reset-Links**
  - Der Benutzer öffnet die Passwort-Reset-Seite über den *ViewController*.
  - Der Benutzer sendet eine Anfrage zur Passwortzurücksetzung an den *AccountService*.
  - *AccountService* sucht den Benutzer über *DatabaseManager* und *UserRepository*.
  - Falls der Benutzer existiert, wird eine Verifikations-E-Mail über *EmailVerificationService* gesendet.
  - Andernfalls wird eine Fehlermeldung zurückgegeben.
- **Setzen eines neuen Passworts nach Verifikation**
  - Der Benutzer gibt das neue Passwort und den Verifikationstoken ein.
  - *AccountService* überprüft den Token über *EmailVerificationService*.
  - Falls der Token gültig ist:
    - Das neue Passwort wird über *IPasswordHasher* und *PasswordHasher* gehasht.

- Die Benutzerdaten werden über *DatabaseManager* und *UserRepository* aktualisiert.
- Eine Erfolgsmeldung wird an den Benutzer gesendet.

* Falls der Token ungültig ist, wird eine Fehlermeldung ausgegeben.

Aufgabe erstellen [US-3.1]



Hier hättet ihr auch nicht alles zeigen müssen.
Die Interaktionen sind ja doch alle sehr ähnlich.

Weshalb wurde dieses Sequenzdiagramm gewählt?

Dieses Sequenzdiagramm wurde gewählt, um sämtliche Abläufe im Task-Management-System visuell darzustellen – von der Erstellung und Bearbeitung bis zur Statusänderung, Löschung und Abfrage von Aufgaben. Es zeigt die Interaktionen zwischen den wichtigsten Komponenten des Systems: *TaskService*, *AuthorizationService*, *ValidationService*, *NotificationService* und der zentralen Entität *Task*.

Das Diagramm bildet sowohl normale als auch alternative Kontrollflüsse ab (z. B. fehlende Berechtigungen, ungültige Eingaben) und ermöglicht so eine realitätsnahe Modellierung. Durch Rückgabewerte und Prüfungen wird die Robustheit und Logik der Abläufe sichergestellt.

Das Sequenzdiagramm spiegelt die tatsächliche Implementierung wider und unterstützt die Nachvollziehbarkeit sowie die Übereinstimmung mit der Systemarchitektur.

Beschreibung des Diagramms

Das Diagramm besteht aus mehreren Teilprozessen, die zusammen das vollständige Aufgabenmanagement abbilden:

- **Eine neue Aufgabe erstellen**
  - Der Benutzer sendet ein neues Task-Objekt an den *TaskService*.
  - Zuerst wird geprüft, ob der Benutzer berechtigt ist, über den *AuthorizationService*.
  - Ist die Berechtigung gegeben, wird die Aufgabe durch den *ValidationService* validiert.
  - Danach erfolgt die Speicherung und Benachrichtigung über den *NotificationService*.
- **Eine Aufgabe aktualisieren**
  - Der Benutzer sendet aktualisierte Aufgabendaten.
  - Wieder wird zunächst die Berechtigung geprüft.
  - Anschließend erfolgt die Validierung und Speicherung sowie eine Benachrichtigung über die Änderung.
- **Eine Aufgabe zuweisen**
  - Ein Benutzer weist eine Aufgabe einem anderen Benutzer zu.
  - Nach Autorisierungsprüfung erfolgt die Zuweisung und die Zustellbenachrichtigung.
- **Eine Aufgabe abschließen**
  - Der Aufgabenstatus wird geändert (z. B. zu `COMPLETED`).
  - Berechtigung und Statusgültigkeit werden überprüft.
  - Bei Erfolg wird der neue Status gespeichert und eine Statusaktualisierung übermittelt.
- **Eine Aufgabe lesen**
  - Ein Benutzer möchte Details zu einer bestimmten Aufgabe sehen.
  - Nach erfolgreicher Berechtigungsprüfung werden die Daten geladen.
- **Eine Aufgabe löschen**
  - Ein Benutzer versucht, eine Aufgabe zu löschen.
  - Nach erfolgreicher Prüfung wird sie entfernt und eine Löschbenachrichtigung gesendet.
- **Aufgaben abfragen**
  - Aufgaben können nach verschiedenen Kriterien (Projekt, Status, Priorität) abgefragt werden.
  - Die entsprechenden Ergebnislisten werden dem Benutzer zurückgegeben.

Strukturmodelle: 3/3
Interfaces: 2/2
Verhaltensmodelle: 2/2

2.2 Tracing der User Storys und relevante Anforderungen

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: 1.2) und relevante Anforderungen

Als registrierter Nutzer möchte ich mich mit meinen Zugangsdaten einloggen können, damit ich auf meine Daten und personalisierte Inhalte zugreifen kann.

Tracing der User-Story (ID: 1.3) und relevante Anforderungen

Als registrierter Nutzer möchte ich mein Passwort zurücksetzen können, falls ich es vergessen habe, damit ich wieder Zugriff auf meinen Account erhalte.

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 der User-Story (ID: 3.2) und relevante Anforderungen

Als Teammitglied möchte ich Tasks mit den entsprechenden User Stories verlinken können, damit die Zusammenhänge klar ersichtlich sind.

Tracing der User-Story (ID: 3.3) und relevante Anforderungen

Als Teammitglied möchte ich die Tasks priorisieren können, damit ich sicherstellen kann, dass die wichtigsten Aufgaben zuerst erledigt werden.

Tracing der User-Story (ID: 3.4) und relevante Anforderungen

Als Teammitglied möchte ich Tasks als complete“ oder incomplete“ markieren können, damit der Fortschritt der Arbeit klar sichtbar ist.

Das hätte es nicht unbedingt gebraucht

Tracing der User-Storys und relevante Anforderungen

Table 2: Tracing der User-Storys und relevante Anforderungen

Klasse	**Verantwortlichkeit/Funktion**	**Relevante User Story**
	Account Management	
MainController	Verarbeitung der Anfragen zur Registrierung, Anmeldung und Passwortzurücksetzung.	US-1.1, US-1.2, US-1.3
AccountService	Verwaltung der Benutzerregistrierung, Anmeldung und Passwortänderung.	US-1.1, US-1.2, US-1.3
ViewController	Anzeige der Formulare für Login, Registrierung und Passwort-Reset.	US-1.3
EmailVerificationService	Versand und Verifikation von Bestätigungs-E-Mails.	US-1.1, US-1.3
IPasswordHasher	Interface zur Passwort-Hashing- und Verifikationsstrategie.	US-1.1, US-1.2, US-1.3
PasswordHasher	Implementierung des sicheren Passwort-Hashings und Verifikation.	US-1.1, US-1.2, US-1.3
DatabaseManager	Zugriff und Verwaltung der Benutzerdaten in der Datenbank. Zugriff auf Benutzerobjekte über die E-Mail oder Name.	US-1.1, US-1.2, US-1.3 US-1.1, US-1.2, US-1.3
User	Modellierung der Benutzerinformationen.	US-1.1, US-1.2, US-1.3
UserRole	Definition der Rollen (Normal, Manager, Admin).	US-1.1, US-1.2, US-1.3
PasswordResetToken (geplant)	Repräsentiert Token für Passwortzurücksetzung (zukünftig).	US-1.3
	Task Management	
TaskService	Erstellung, Aktualisierung und Verwaltung von Aufgaben.	US-3.1, US-3.2, US-3.3, US-3.4
TaskQueryService	Abfragen von Aufgaben nach verschiedenen Kriterien (Status, Priorität, Projekt).	US-3.1, US-3.3
SubtaskService	Erstellung und Verwaltung von Unteraufgaben.	US-3.1
NotificationService	Senden von Benachrichtigungen bei Aufgabenereignissen.	US-3.1, US-3.4
ValidationService	Validierung der Eingaben bei der Aufgabenerstellung.	US-3.1
AuthorizationService	Überprüfung von Nutzerrechten bei Aufgabenoperationen.	US-3.1, US-3.2, US-3.3, US-3.4
TaskRepository	Persistenz der Aufgaben in der Datenbank.	US-3.1
SubtaskRepository	Persistenz der Subtasks.	US-3.1
Task	Modellierung der Aufgaben inklusive Status, Priorität und Projektzuordnung.	US-3.1, US-3.2, US-3.3, US-3.4
Subtask	Modellierung von Unteraufgaben innerhalb eines Tasks.	US-3.1
TaskFile	Speicherung von Dateianhängen an Aufgaben.	US-3.1
TaskStatus (Enum)	Definition der Aufgabenstatus (NEW, IN_PROGRESS, COMPLETED, etc.).	US-3.3
Priority (Enum)	Definition der Prioritätenstufen (LOW, MEDIUM, HIGH, URGENT).	US-3.3
Project	Verknüpfung von Aufgaben mit Projekten.	US-3.2

Detaillierte Beschreibung der Implementierung

- Die Benutzeranfragen zur Registrierung, Anmeldung und Passwortzurücksetzung werden durch den **MainController** verarbeitet.
- Die zentrale Logik der Benutzerregistrierung, Anmeldung und Passwortänderung wird im **AccountService** verwaltet.
- Die Anzeige der Login-, Registrierungs- und Passwort-Reset-Formulare erfolgt über den **ViewController**.
- Die Bestätigungs-E-Mails zur Benutzerverifikation werden durch den **EmailVerificationService** versendet und überprüft.
- Das Interface **IPasswordHasher** definiert die Methoden für Passwort-Hashing und -Verifikation.
- Die sichere Implementierung des Passwort-Hashings erfolgt durch die Klasse **PasswordHasher**.
- Der **DatabaseManager** verwaltet das Speichern und Abrufen von Benutzerdaten.
- Der Zugriff auf Benutzerobjekte anhand von E-Mail-Adressen wird durch das **UserRepository** ermöglicht.
- Die Klasse **User** modelliert Benutzerprofile mit Name, E-Mail, Passwort und Rolle.
- Benutzerrollen wie Normal, Manager und Admin werden durch das Enum **UserRole** definiert.
- Das **PasswordResetToken** (geplant) stellt einen Token für die Passwortzurücksetzung dar.
- Aufgaben werden durch den **TaskService** erstellt, aktualisiert und verwaltet.
- Der **TaskQueryService** erlaubt das Abfragen von Aufgaben nach Status, Priorität und Projektzugehörigkeit.
- Unteraufgaben werden mit Hilfe des **SubtaskService** erstellt und verwaltet.
- Benachrichtigungen bei Aufgabenereignissen werden über den **NotificationService** verschickt.
- Die Eingaben bei Aufgabenoperationen werden durch den **ValidationService** validiert.
- Die Überprüfung von Nutzerrechten erfolgt durch den **AuthorizationService**.
- Aufgaben werden in der Datenbank über das **TaskRepository** gespeichert.
- Unteraufgaben werden im **SubtaskRepository** gespeichert.
- Die Klasse **Task** modelliert Aufgaben inklusive Status, Priorität und Projektverlinkung.
- Die Klasse **Subtask** modelliert Unteraufgaben, die zu einer Hauptaufgabe gehören.

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 **H2** 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.

Zusätzlich nutzen wir **GitLab** als zentrale Plattform zur gemeinsamen Projektarbeit. Dort pflegen wir nicht nur alle relevanten Dokumente, sondern verwalten auch den aktuellen Stand des Quellcodes unseres Prototyps. So kann jedes Teammitglied jederzeit auf die aktuellste Version zugreifen und lokal weiterentwickeln. Diese kollaborative Arbeitsweise unterstützt die parallele Entwicklung und stellt sicher, dass alle Teammitglieder stets synchron arbeiten.

Daran hat sich ja nichts geändert. GitLab an sich benutzt ihr ja seit Beginn.

1/1

3.2 Dokumentation der Codequalität

Abweichungen des Codes zur geplanten Architektur

a) Beschreibung der Abweichung: Geplant war ein einzelner Controller für alle Aufgabenfunktionen. Beim Umsetzen des Prototyps hat sich jedoch gezeigt, dass es sinnvoller ist, den Controller zu splitten. So entstand neben dem `TaskController` für die Web-Oberfläche (Thymeleaf) auch ein `TaskRestController`, der REST-Endpunkte zur Verfügung stellt.

b) Begründung der Abweichung: Der Hauptgrund für diese Entscheidung war die bessere Testbarkeit und Trennung der Verantwortlichkeiten. Mit dem REST-Controller konnten wir z.B. die automatisierten Tests deutlich einfacher aufbauen. Auch aus Sicht der Wartbarkeit und Erweiterbarkeit erschien uns die Trennung als sauberere Lösung.

c) Nächste Schritte: Diese Aufteilung soll in der Architektur nachgezogen werden. Im nächsten Sprint wird unser Architekturdiagramm aktualisiert, um beide Controller sichtbar zu machen. Damit ist die Abweichung offiziell dokumentiert und in die langfristige Struktur integriert.

Durchgeführte automatische Tests und Testergebnisse

Teststrategie: Wir wollten von Anfang an sicherstellen, dass unsere Kernfunktionen wie erwartet laufen. Daher haben wir sowohl Unit-Tests als auch REST-Tests geschrieben:

- **White-box-Tests:** direkt im Service (`TaskService`), um Logik zu prüfen
- **Black-box-Tests:** via MockMvc für den `TaskRestController`, um die API wie ein Nutzer zu testen

Testübersicht:

Table 3: Testübersicht

Test-ID	Zeitpunkt	Herkunft / Beschreibung	Ergebnis
TC1	30.04.2025	Unit Test: `TaskService.getTask()` mit gültiger ID	Pass
TC2	30.04.2025	Unit Test: `TaskService.getTask()` mit ungültiger ID	Pass
TC3	30.04.2025	Unit Test: `TaskService.getTask()` mit nicht existierender ID	Pass
TC4	02.05.2025	REST-Test: POST /api/tasks mit gültigem JSON	Pass
TC5	02.05.2025	REST-Test: POST /api/tasks mit leerem JSON (Validation)	Pass

sehr schön!

Ablageort der Tests:

- `src/test/java/com/example/demo/service/TaskServiceTest.java`
- `src/test/java/com/example/demo/controller/TaskRestControllerTest.java`

Reflexion zur Teststrategie: Automatisierte Tests geben uns Sicherheit bei Änderungen am Code. Sobald etwas bricht, sehen wir es direkt im Testlauf. Außerdem spart es langfristig Zeit, da wir bei Refactorings nicht manuell alles gegenchecken müssen. Die Mischung aus White- und Black-box-Tests ist für unser kleines Projekt mehr als ausreichend.

Durchgeführte manuelle Tests

Testdurchführung: Zwei Teammitglieder haben händisch getestet, ob man Aufgaben korrekt anlegen und anzeigen kann. Dabei wurden sowohl Fehlerfälle als auch normale Abläufe geprüft.

Testergebnisse:

Table 4: Testergebnisse

Rolle	Ziel des Tests	Version	Ergebnis
Teammitglied	Test der Validierung (leeres Feld bei Task-Eingabe)	UI vom 01.05.2025	Pass
Teammitglied	Anzeige der Taskliste mit leerer DB	UI vom 01.05.2025	Pass

3/3

3.3 Tracing

In unserem Projekt wurde auch in Sprint 2 ein systematisches Tracing zwischen dem UML-Diagramm und der Code-Implementierung durchgeführt. Dafür haben wir eine ausführliche Tabelle in GitLab erstellt, die die Zuordnung zwischen UML-Komponenten und den entsprechenden Code-Klassen sowie -Interfaces enthält.

Diese Tabelle dokumentiert präzise, wie Modellierung und Implementierung miteinander verknüpft sind – zum Beispiel für Klassen wie `TaskService`, `NotificationService` oder `AuthorizationService`.

GitLab-Link zur Tracing-Dokumentation: 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 2/3

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.

Bitte entfernt unbedingt den Ordner ".idea" aus eurem Repository! Das ist ein Ordner von IntelliJ, der die lokale Entwicklungsumgebung in der IDE widerspiegelt und deshalb nicht versioniert sein sollte, weil diese Umgebung bei jedem etwas anders sein kann.

Benutzergeschichte

Account Management

- **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
- **ID:** US-1.2
  Beschreibung: Als registrierter Nutzer möchte ich mich mit meinen Zugangsdaten einloggen können, damit ich auf meine Daten und personalisierte Inhalte zugreifen kann.
  Verfolgt zu: ANF-001
- **ID:** US-1.3
  Beschreibung: Als registrierter Nutzer möchte ich mein Passwort zurücksetzen können, falls ich es vergessen habe, damit ich wieder Zugriff auf meinen Account erhalte.
  Verfolgt zu: ANF-001

Task Management

- **ID:** US-3.1
  Beschreibung: Als Teammitglied möchte ich Tasks erstellen können, damit ich die Arbeit in kleinere Einheiten aufteilen kann.
  Verfolgt zu: ANF-003
- **ID:** US-3.2
  Beschreibung: Als Teammitglied möchte ich Tasks mit den entsprechenden User Stories verlinken können, damit die Zusammenhänge klar ersichtlich sind.
  Verfolgt zu: ANF-003
- **ID:** US-3.3
  Beschreibung: Als Teammitglied möchte ich die Tasks priorisieren können, damit ich sicherstellen kann, dass die wichtigsten Aufgaben zuerst erledigt werden.
  Verfolgt zu: ANF-003
- **ID:** US-3.4
  Beschreibung: Als Teammitglied möchte ich Tasks als complete“ oder incomplete“ markieren können, damit der Fortschritt der Arbeit klar sichtbar ist.
  Verfolgt zu: ANF-003

Was soll das hier bedeuten?

Hier fehlt ansonsten die Nutzeranleitung

Installation und Kompilation

Um die entwickelte Anwendung lokal auszuführen, sind bestimmte Schritte erforderlich. Diese beinhalten das Klonen des Repositories, das Installieren der notwendigen Abhängigkeiten sowie das Starten der Anwendung über die entsprechende Entwicklungsumgebung oder das Terminal. Eine detaillierte Schritt-für-Schritt-Anleitung zur Installation und Ausführung befindet sich in der bereitgestellten README-Datei.

Die vollständige Anleitung ist im folgenden Link verfügbar:

3.5 Abweichung Sprintplanung

In Sprint 2 kam es zu einigen Abweichungen von der ursprünglichen Planung. Während manche User Stories vollständig umgesetzt wurden, konnten andere nur teilweise abgeschlossen werden.

Vollständig umgesetzt:

- **US-1.2 Login**: Fast alle Aufgaben wurden erfolgreich abgeschlossen. Nur ein Akzeptanzkriterium zur Session-Speicherung steht noch aus.

Teilweise umgesetzt:

- **US-1.1 Benutzerkonto erstellen**: Registrierung und E-Mail-Verifizierung wurden umgesetzt, aber Passwort-Hashing und Benutzerrollen fehlen noch.
- **US-1.3 Passwort-Reset**: Reset-Seite und Logik sind vorhanden, aber Token-Erstellung und E-Mail-Verifikation fehlen noch.
- **US-3.1 Aufgabe erstellen**: Task-Klasse, CRUD-Operationen und Projektverknüpfung wurden umgesetzt. Zuweisung, Benachrichtigungen und Datei-Upload sind offen.
- **US-3.2 Tasks verlinken**: Nur die Task-Klasse wurde angepasst. UI-Verknüpfung und Darstellung fehlen noch.
- **US-3.3 Tasks priorisieren**: Das Priority-Enum ist vorhanden, aber die Auswahl im Formular wurde noch nicht implementiert.
- **US-3.4 Tasks als complete/incomplete markieren**: Status-Enum wurde ergänzt, jedoch fehlen Buttons und Anzeige in der Oberfläche.

Gründe für die Abweichungen

Zeitliche & technische Herausforderungen:

- Einige geplante Aufgaben, vor allem im Frontend-Bereich, dauerten länger als erwartet.
- E-Mail-Verifikation, Token-Handling und Rollenlogik waren technisch schwieriger umzusetzen als gedacht.

Teaminterne Schwierigkeiten:

- Es gab unterschiedliche Kenntnisstände im Team, wodurch einige Funktionen langsamer entwickelt wurden.

Lessons Learned & Planung für den nächsten Sprint

Bessere Zeitplanung und Kommunikation:

- Wir müssen den Aufwand für Aufgaben mit Frontend oder Formularlogik realistischer einschätzen.
- Bei technischen Blockaden wollen wir schneller kommunizieren und gemeinsam Lösungen suchen.

Fokus auf Umsetzung statt Perfektion:

– Es ist besser, zuerst einfache und funktionierende Versionen umzusetzen, bevor wir Zusatzfunktionen einbauen.
– Komplexe Aufgaben (z. B. Token-System) sollten früher im Sprint behandelt werden.

Ziel für den nächsten Sprint

– Aufgaben, die nur teilweise umgesetzt wurden (Token, Session, UI), sollen abgeschlossen werden.
– Mehr Features sollen auch auf der Oberfläche sichtbar gemacht werden.
– Die Zusammenarbeit im Team soll strukturierter ablaufen, z. B. durch kürzere Meetings und klare Zuständigkeiten.

1/1

Dokumentation individuelle Beiträge

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

Sprint	**Verantwortliche Teammitglieder**	**Anwesende während der Meetings**	**(Online-) Beiträge zum Inhalt durch**	**Korrektur-gelesen durch**
Sprint 2	Dogukan, Helin, Hüseyin, Eren, Cagla	Dogukan, Helin, Hüseyin, Eren, Cagla	Dogukan, Helin, Hüseyin, Eren, Cagla	Dogukan, Helin, Hüseyin, Eren, Cagla

Table 5: Beitrag der Teammitglieder zum Projektabgabedokument

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

20/20

Task ID	**Task Beschreibung/ Name**	**Teammitglied**	**Beitrag in %**
US-1.2-T1	Login-Formular im ViewController bereitstellen	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.2-T2	Implementierung der Login-Logik im AccountService	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.2-T3	E-Mail und Passwort-Eingaben validieren	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.2-T4	Passwort mit IPasswordHasher verifizieren	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.2-T5	Benutzersitzung (Session) bei erfolgreicher Anmeldung erstellen	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.3-T1	Passwort-Reset-Seite im ViewController bereitstellen	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.3-T2	Implementierung der Reset-Logik im AccountService	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-1.3-T5	Passwort mit IPasswordHasher neu setzen	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-T6	Aufgaben mit Projekten verknüpfen (project: Project)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-3.2-T1	Erweiterung der Task-Klasse um eine Referenz zu User Stories	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-3.3-T1	Einführung des Priority-Enums (LOW, MEDIUM, HIGH, URGENT)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils
US-3.4-T1	Erweiterung der Task-Klasse um Status (TaskStatus Enum)	Dogukan, Helin, Hüseyin, Eren, Cagla	20% jeweils

Table 6: Beitrag der Teammitglieder zu den im Sprint 2 erledigten Tasks