

# Sprint 1

Freyschmidt, Henry Lewis (HLF)  
 Hama, Zana Salih (ZSH)  
 Krasnovska, Paula (PK)  
 Krüger, Lucas (LK)  
 Prüger, Marvin Oliver (MOP)  
 Seep, Tom-Malte (TMS)  
 Zabel, Steven (SZ)

18. Februar 2024

	Verantwortung	Inhalt	Grafik	Korrektur
<b>Backlog</b>				
Änderungen am Backlog	SZ	SZ	-	LK, ZSH
Tasks	SZ	SZ	SZ	LK, ZSH
<b>Feinentwurf</b>				
Login Seite Prototyp	ZSH, PK	ZSH	ZSH,PK	HLF, LK
Projekt-Manager Prototyp	ZSH, PK	ZSH	ZSH,PK	HLF, LK
Komponentendiagramm	PK	PK	PK, LK	HLF, LK, ZSH
Klassendiagramm	PK, HLF	HLF, LK	HLF, LK	LK, ZSH
Interfaces für User-Storys zu Account	HLF	HLF	HLF, PK	LK, ZSH
Interfaces für User-Storys zu User-Story	HLF	HLF	HLF,ZSH	LK, ZSH
Modellierung umgesetzte DS	TMS	TMS	LK	HLF, PK,LK, ZS
Sequenzdiagramm zu A2	ZSH	ZSH	HLF	PK, ZSH, LK
Sequenzdiagramm zu A3	ZSH	ZSH	ZSH	PK, ZSH, LK
Sequenzdiagramm zu U1	LK	LK	HLF, PK	PK, ZSH
Sequenzdiagramm zu U3	ZSH	ZSH	HLF,PK	PK, ZSH, LK
Sequenzdiagramm zu U4	ZSH	ZSH	HLF,PK	PK, ZSH, LK
Tracing	LK	LK	-	HLF
<b>Implementation und Tests</b>				
Update Technologie	LK	LK	-	HLF, ZSH
Abweichung von geplanter Architektur	ZSH	ZSH	-	HLF, PK
Automatische Tests	LK	LK	-	HLF
Ingerationstest INT.T1	LK	LK	-	HLF, ZSH
Abnahmetest AT.T1	ZSH, PK	ZSH, PK	-	HLF, LK
Tracing	LK	LK	-	HLF
Laufender Prototyp	PK	PK	PK	HLF, ZSH, LK
Abweichungen von Sprintplanung	PK, ZSH	PK, ZSH	-	HLF, ZSH, LK

Task-ID	Task-Beschreibung/Name	Teammitglied	Beitrag in %
A1.D1	Erstelle Speicher für Accounts	MOP	50%
		TMS	50%
A2.D1	Erstelle Funktionalität zur Account-Speicherung	MOP	50%
		TMS	50%
A2.F1	Erstelle Anzeige zur Account-Registrierung	PK	50%
		ZSH	50%
A2.B1	Erstelle Funktionalität zur Registrierung	LK	80 %
		HLF	20%
A3.D1	Erstelle Funktionalität zum Account-Abgleich	MOP	50%
		TMS	50%
A3.F1	Erstelle Anzeige zum Login	PK	50%
		ZSH	50%
A3.B1	Erstelle Funktionalität zum Login	LK	80%
		HLF	20%
U1.D1	Erstelle Funktionalität zur User-Story-Ausgabe aus dem Speicher	MOP	50%
		TMS	50%
U1.F1	Erstelle Anzeige für User-Storys	PK	50%
		ZSH	40%
		LK	10%
U1.B1	Erstelle Funktionalität zur User-Story-Abbildung	LK	100%
U3.D1	Erstelle Speicher für User-Storys	MOP	50%
		TMS	50%
U3.D1	Erstelle Speicher für User-Storys	MOP	50%
		TMS	50%
U3.D2	Erstelle Funktionalität zur User-Story-Speicherung	MOP	50%
		TMS	50%

Task-ID	Task-Beschreibung/Name	Teammitglied	Beitrag in %
U3.F1	Erstelle Anzeige zur User-Story-Erstellung	ZSH	100%
U3.B1	Erstelle Funktionalität zur User-Story-Erstellung	LK	100%
U4.D1	Erstelle Funktionalität zur User-Story-Überspeicherung	MOP TMS	50% 50%
U4.F1	Erstelle Anzeige zur User-Story-Anpassung	ZSH PK	80% 20%
U4.B1	Erstelle Funktionalität zur User-Story-Anpassung	LK	100%
U5.D1	Speichere zu User-Storys die Priorität	MOP TMS	50% 50%
U5.D2	Erweitere Funktionalität zur User-Story-Speicherung um die Priorität	MOP TMS	50% 50%
U5.D3	Erweitere Funktionalität zur User-Story-Überspeicherung um die Priorität	MOP TMS	50% 50%
U5.F1	Erweitere Anzeige zur User-Story-Erstellung um die Priorität.	ZSH	100%
U5.F2	Erweitere Anzeige zur User-Story-Anpassung um die Priorität	ZSH	100%
U5.B1	Erweitere Funktionalität zur User-Story-Erstellung um die Priorität	LK HLF	40% 60%
U5.B2	Erweitere Funktionalität zur User-Story-Anpassung um die Priorität	LK HLF	40% 60%
	Logic-Tests	HLF	100%
	HTTP-Tests	LK	100%
	DB-Tests	MOP TMS	50% 50%
	Product Owner	SZ	100%
	Anlegen und Verwalten von Tasks	SZ	100%

## Inhaltsverzeichnis

<b>1 Backlog</b>	<b>2</b>
1.1 Änderungen am Backlog . . . . .	2
1.2 Planung des Sprints . . . . .	2
1.2.1 [A] Account . . . . .	2
1.2.2 [U] User-Story . . . . .	3
<b>2 Feinentwurf</b>	<b>7</b>
2.1 Prototyp für Login und Projekt-Manager . . . . .	7
2.2 Modellierung der verfeinerten Struktur . . . . .	11
2.3 Modellierung der verfeinerten Interfaces und Datenstrukturen . . . . .	15
2.3.1 User-Storys zum Account . . . . .	15
2.3.2 User-Storys zur User-Story . . . . .	16
2.3.3 Modellierung umgesetzte Datenstrukturen . . . . .	17
2.4 Modellierung des verfeinerten Verhaltens . . . . .	18
2.5 Tracing . . . . .	20
<b>3 Implementierung und Tests</b>	<b>20</b>
3.1 Updates Technologien . . . . .	20
3.2 Dokumentation der Codequalität . . . . .	20
3.2.1 Abweichung des Codes von der geplanten Architektur/Design . . . . .	20
3.2.2 Automatische Tests . . . . .	22
3.2.3 Manuelle Tests . . . . .	24
3.3 Tracing . . . . .	24
3.4 Laufender Prototyp . . . . .	25
3.5 Abweichungen von Sprintplanung . . . . .	26

# 1 Backlog

## 1.1 Änderungen am Backlog

Es gibt keine Änderungen am Backlog. Das Vorschlagen und Diskutieren von Änderungen am Backlog war stets Thema in den wöchentlichen Meetings. Da sich die Anfertigung des Grobentwurfs mit der Zeitspanne des Sprints überschnitt, wurden alle Änderungen zum ursprünglichen Backlog noch in den Grobentwurf geschrieben, sodass es zum Sprint selbst keine weiteren Änderungen mehr gab.

## 1.2 Planung des Sprints

Die Tasks sind eingeteilt in die Teams [F] Frontend, [B] Backend und [D] Daten-Ebene. Das Frontend umfasst die Funktionalität zur Präsentation, das Backend umfasst die Funktionalität zur Logik und die Daten-Ebene umfasst die Funktionalität zur Datenspeicherung, -ausgabe und -manipulation. Der Backlog ist auch in Trello verfügbar:

<https://trello.com/invite/b/sQO7igKH/ATTI10673b4880074c8b311584edffda5dadD626DC84/tasks>

- Product Owner: SZ
- Frontend-Entwicklung: PK, ZSH
- Backend-Entwicklung: HLF, LK
- Daten-Entwicklung: MOP, TMS

### 1.2.1 [A] Account

- ⊗ **A1** Als Nutzer habe ich einen Account, um mich mit diesem gegenüber des Systems zu authentifizieren.
  - ⊗ **A1.D1** Erstelle Speicher für Accounts.
    - \* Aufwandseinschätzung: 4 Stunden
    - \* Bearbeitung: 4 Stunden durch TMS, MOP
    - \* Review: 1 Stunde durch LK
    - \* Test: 1 Stunde durch MOP, ZSH
- ⊗ **A2** Als potenzieller Nutzer kann ich unter Angabe einer Mail und eines Passworts einen Account erstellen, um mich zukünftig mit diesem gegenüber des Systems zu authentifizieren.
  - ⊗ **A2.D1** Erstelle Funktionalität zur Account-Speicherung, welche einen Account in den Speicher schreibt.
    - \* Aufwandseinschätzung: 12 Stunden
    - \* Bearbeitung: 13 Stunden durch TMS, MOP
    - \* Review: 1 Stunde durch LK
    - \* Test: 1 Stunde durch MOP
  - ⊗ **A2.F1** Erstelle Anzeige zur Account-Registrierung.
    - \* Aufwandseinschätzung: 5 Stunden
    - \* Bearbeitung: 6 Stunden durch ZSH, PK
    - \* Review: 1 Stunde durch LK
    - \* Test: 1 Stunde durch ZSH, PK
  - ⊗ **A2.B1** Erstelle Funktionalität zur Registrierung, welche die Anzeige zur Account-Registrierung ausliest und die Funktionalität zur Account-Speicherung aufruft.

- \* Aufwandseinschätzung: 1 Stunde
  - \* Bearbeitung: 1 Stunde durch LK
  - \* Review: 1 Stunde durch ZSH, PK
  - \* Test: 1 Stunde durch LK
- ⊗ **A3** Als potenzieller Nutzer kann ich mich mit meiner Mail und meinem Passwort in meinen Account einloggen, um mich gegenüber des Systems zu authentifizieren.
- ⊗ **A3.D1** Erstelle Funktionalität zum Account-Abgleich, welche prüft, ob ein Account bereits gespeichert ist.
- \* Aufwandseinschätzung: 1 Stunde
  - \* Bearbeitung: 1 Stunde durch TMS, MOP
  - \* Review: 1/2 Stunde durch LK
  - \* Test: 1/2 Stunde durch MOP
- ⊗ **A3.F1** Erstelle Anzeige zum Login.
- \* Aufwandseinschätzung: 3 Stunden
  - \* Bearbeitung: 4 Stunden durch ZSH, PK
  - \* Review: 1 Stunde durch LK
  - \* Test: 1 Stunde durch ZSH, PK
- ⊗ **A3.B1** Erstelle Funktionalität zum Login, welche die Anzeige zum Login ausliest, die Funktionalität zum Account-Abgleich aufruft und einen Nutzer einloggt, falls der Account bereits gespeichert ist.
- \* Aufwandseinschätzung: 3 Stunden
  - \* Bearbeitung: 3 Stunden durch LK
  - \* Review: 1 Stunde durch ZSH, PK
  - \* Test: 1 Stunde durch LK

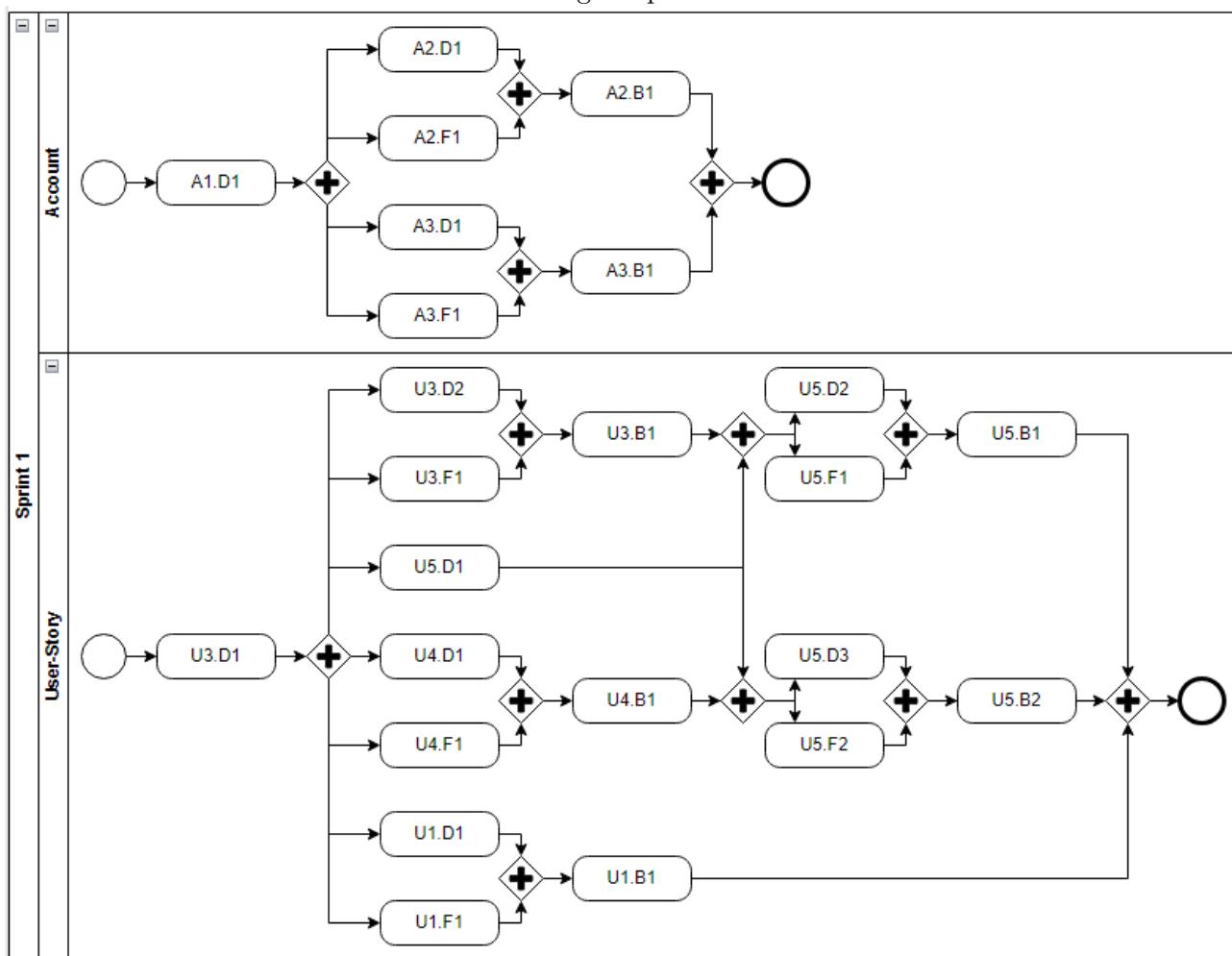
### 1.2.2 [U] User-Story

- ⊗ **U1** Als Nutzer kann ich mir alle User-Storys anzeigen lassen, um eine Übersicht über diese zu bekommen.
- ⊗ **U1.D1** Erstelle Funktionalität zur User-Story-Ausgabe, welche User-Storys aus dem Speicher ausgibt.
- \* Aufwandseinschätzung: 9 Stunden
  - \* Bearbeitung: 9 Stunden durch TMS, MOP
  - \* Review: 1 Stunde durch LK
  - \* Test: 1 Stunde durch MOP
- ⊗ **U1.F1** Erstelle Anzeige für User-Storys.
- \* Aufwandseinschätzung: 5 Stunden
  - \* Bearbeitung: 8 Stunden durch ZSH, PK
  - \* Review: 1 Stunde durch LK
  - \* Test: 1 Stunde durch ZSH, PK
- ⊗ **U1.B1** Erstelle Funktionalität zur User-Story-Abbildung, welche die Funktionalität zur User-Story-Ausgabe aufruft und die User-Storys auf die Anzeige für User-Storys abbildet.
- \* Aufwandseinschätzung: 4 Stunden
  - \* Bearbeitung: 4 Stunden durch LK

- \* Review: 1 Stunde durch ZSH, PK
  - \* Test: 1 Stunde durch LK
- **U3** Als Product Owner kann ich eine User-Story erstellen, um Anforderungen dem Projekt hinzuzufügen.
- ⊗ **U3.D1** Erstelle Speicher für User-Storys.
    - \* Aufwandseinschätzung: 4 Stunden
    - \* Bearbeitung: 4 Stunden durch TMS, MOP
    - \* Review: 1 Stunde durch LK
    - \* Test: 1 Stunde durch MOP, ZSH
  - ⊗ **U3.D2** Erstelle Funktionalität zur User-Story-Speicherung, welche eine User-Story in den Speicher schreibt.
    - \* Aufwandseinschätzung: 12 Stunden
    - \* Bearbeitung: 11 Stunden durch TMS, MOP
    - \* Review: 1 Stunde durch LK
    - \* Test: 1 Stunde durch MOP
  - ⊗ **U3.F1** Erstelle Anzeige zur User-Story-Erstellung.
    - \* Aufwandseinschätzung: 8 Stunden
    - \* Bearbeitung: 5 Stunden durch TMS, MOP
    - \* Review: 1 Stunde durch LK
    - \* Test: 1 Stunde durch MOP
- **U3.B1** Erstelle Funktionalität zur User-Story-Erstellung, welche, wenn der Nutzer die Rechte dazu hat, die Anzeige zur User-Story-Erstellung ausliest und die Funktionalität zur User-Story-Speicherung aufruft.
- \* Aufwandseinschätzung: 5 Stunden
  - \* Bearbeitung: 5 Stunden durch HLF, LK
  - \* Review: 1 Stunde durch ZSH, PK
  - \* Test: 1 Stunde durch HLF
- **U4** Als Product Owner kann ich eine User-Story editieren, um sie an neue Umstände anzupassen.
- ⊗ **U4.D1** Erstelle Funktionalität zur User-Story-Überspeicherung, welche eine User-Story im Speicher überschreibt.
    - \* Aufwandseinschätzung: 4 Stunden
    - \* Bearbeitung: 4 Stunden durch TMS, MOP
    - \* Review: 1 Stunde durch LK
    - \* Test: 1 Stunde durch MOP
  - ⊗ **U4.F1** Erstelle Anzeige zur User-Story-Anpassung.
    - \* Aufwandseinschätzung: 4 Stunden
    - \* Bearbeitung: 8 Stunden durch TMS, MOP
    - \* Review: 1 Stunde durch LK
    - \* Test: 1 Stunde durch MOP
- **U4.B1** Erstelle Funktionalität zur User-Story-Anpassung, welche, wenn der Nutzer die Rechte dazu hat, die Anzeige zur User-Story-Anpassung ausliest und die Funktionalität zur User-Story-Überspeicherung aufruft.
- \* Aufwandseinschätzung: 3 Stunden

- \* Bearbeitung: 3 Stunden durch HLF
  - \* Review: 1 Stunde durch ZSH, PK
  - \* Test: 1 Stunde durch HLF
- ⊗ **U5** Als Product Owner kann ich einer User-Story eine Priorität (Urgent > High > Normal > Low) zuordnen, um die Dringlichkeit der User-Story auszudrücken.
- ⊗ **U5.D1** Speichere zu User-Storys die Priorität.
- \* Aufwandseinschätzung: 2 Stunden
  - \* Bearbeitung: 2 Stunden durch LK
  - \* Review: 1 Stunde durch LK
  - \* Test: 1 Stunde durch MOP
- ⊗ **U5.D2** Erweitere Funktionalität zur User-Story-Speicherung um die Priorität.
- \* Aufwandseinschätzung: 1/2 Stunde
  - \* Bearbeitung: 1/2 Stunde durch TMS, MOP
  - \* Review: 1/4 Stunde durch LK
  - \* Test: 1/4 Stunde durch MOP
- ⊗ **U5.D3** Erweitere Funktionalität zur User-Story-Überspeicherung um die Priorität.
- \* Aufwandseinschätzung: 3/4 Stunde
  - \* Bearbeitung: 1 Stunde durch TMS, MOP
  - \* Review: 1 Stunde durch LK
  - \* Test: 1 Stunde durch MOP
- ⊗ **U5.F1** Erweitere Anzeige zur User-Story-Erstellung um die Priorität.
- \* Aufwandseinschätzung: 1/4 Stunde
  - \* Bearbeitung: 3 Stunden durch TMS, MOP
  - \* Review: 1 Stunde durch LK
  - \* Test: 1 Stunde durch MOP, ZSH und PK
- ⊗ **U5.F2** Erweitere Anzeige zur User-Story-Anpassung um die Priorität.
- \* Aufwandseinschätzung: 1/4 Stunde
  - \* Bearbeitung: 1/4 Stunde durch ZSH, PK
  - \* Review: 1/4 Stunde durch LK
  - \* Test: 1/12 Stunde durch ZSH, PK
- ⊗ **U5.B1** Erweitere Funktionalität zur User-Story-Erstellung um die Priorität.
- \* Aufwandseinschätzung: 1 Stunde
  - \* Bearbeitung: 1 Stunde durch HLF
  - \* Review: 1 Stunde durch ZSH, PK
  - \* Test: 1 Stunde durch HLF
- ⊗ **U5.B2** Erweitere Funktionalität zur User-Story-Anpassung um die Priorität.
- \* Aufwandseinschätzung: 1 Stunde
  - \* Bearbeitung: 1/2 Stunde durch HLF
  - \* Review: 1/4 Stunde durch ZSH, PK
  - \* Test: 1/4 Stunde durch HLF

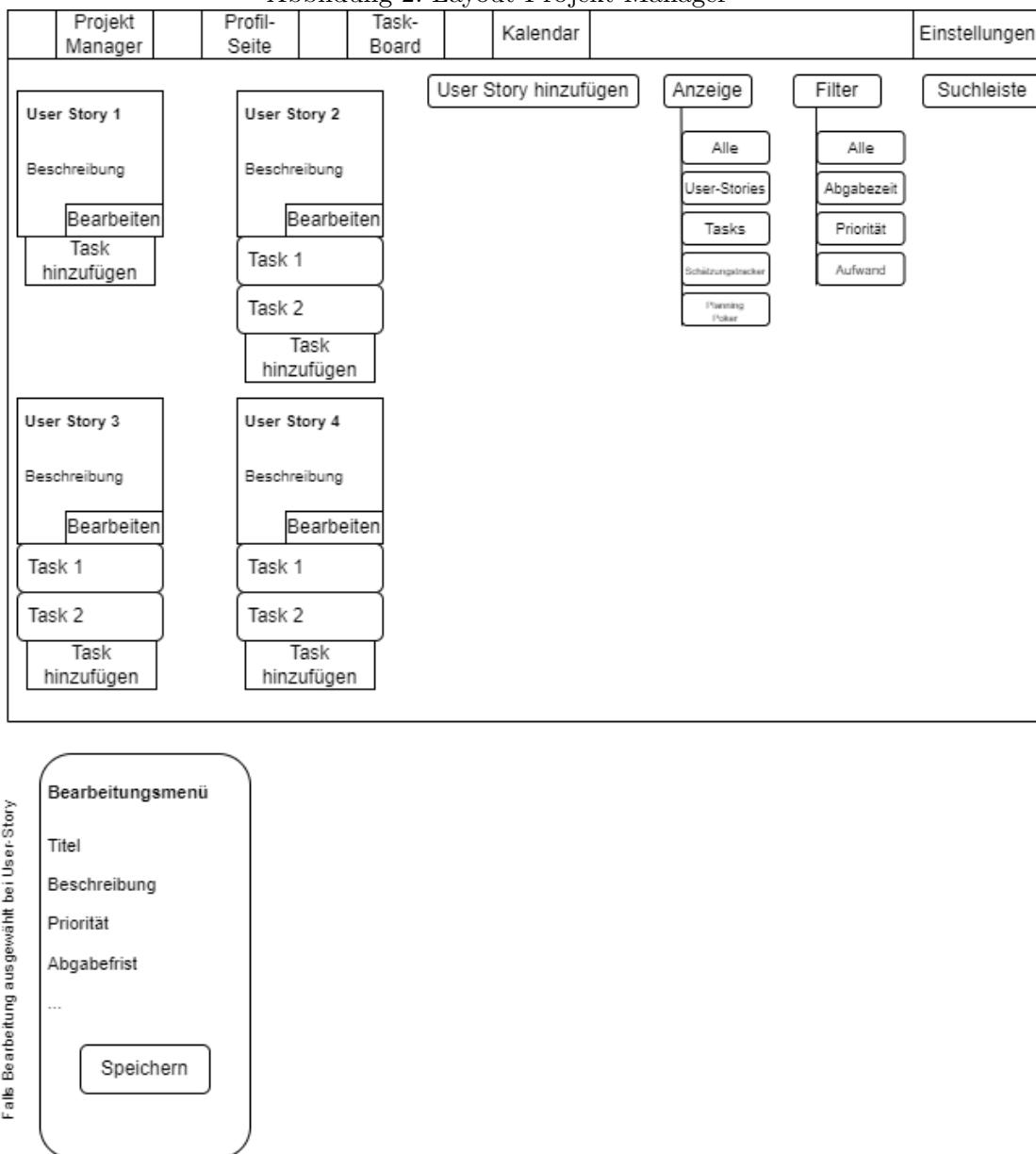
Abbildung 1: Sprint1Flow



## 2 Feinentwurf

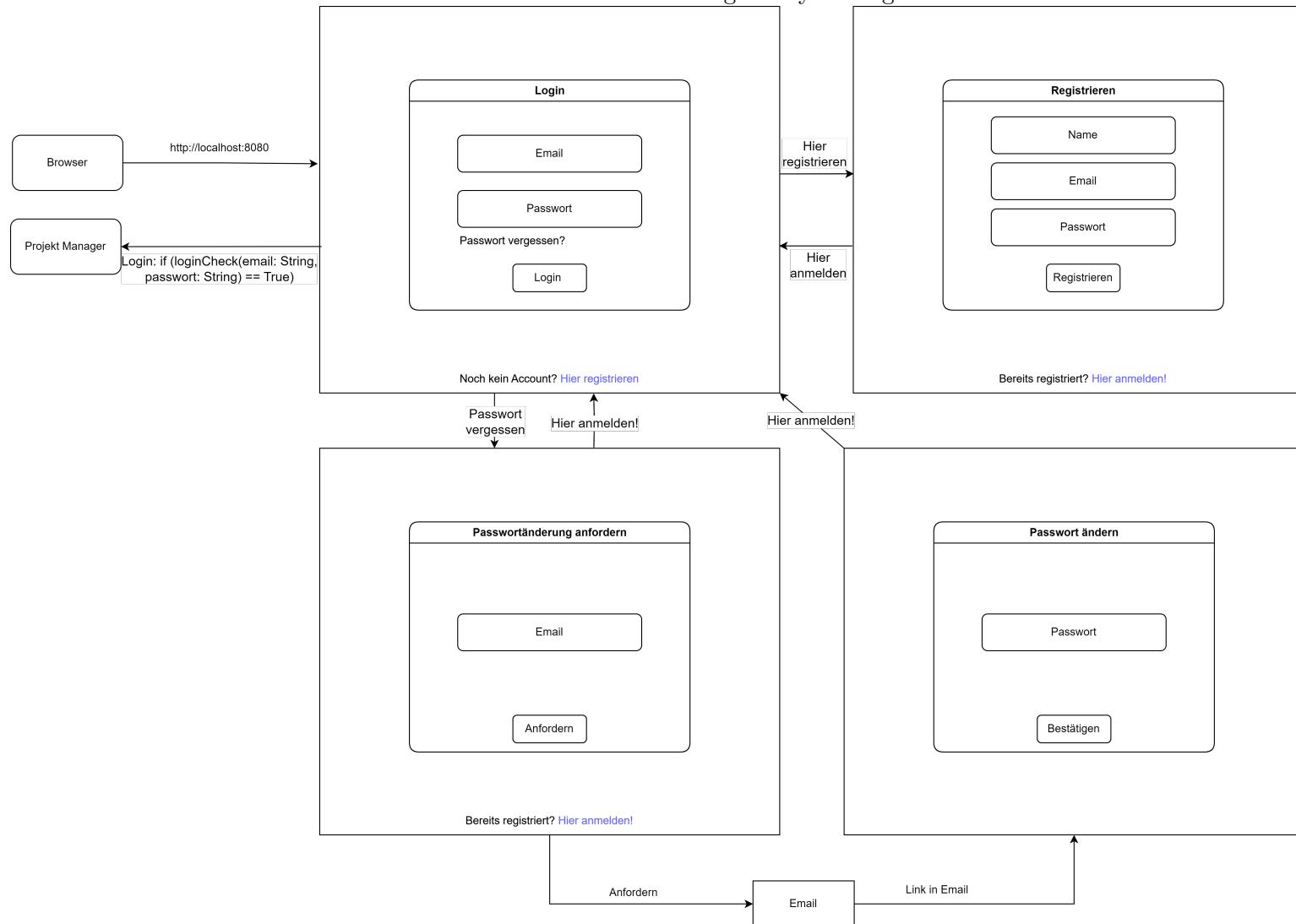
### 2.1 Prototyp für Login und Projekt-Manager

Abbildung 2: Layout Projekt-Manager



In Abbildung 2 ist der designtechnische Prototyp unseres *Projekt-Managers* erkenntlich. Dieser beinhaltet zuallererst einen globalen Reiter, der sich aus dem *Projekt-Manager*, der *Profil*-Seite, das *Task-Board*, dem *Kalender* und den *Einstellungen* zusammensetzt. Diesen haben wir hinzugefügt, um eine vereinfachte Navigation durch die verschiedenen Seiten unserer Website zu ermöglichen. Der globale Reiter bleibt über alle Pages hinweg persistent vorhanden. Die umzusetzende Anforderung in diesem Sprint war es, eine Anzeige für User-Storys zu erstellen (U1.F1), wofür die Seite *Projekt-Manager* genutzt wird. User-Storys können durch den „*User-Story hinzufügen*“-Button hinzugefügt werden. Angemerkt sei, dass der Inhalt einer User-Story zuallererst mit generischen Bezeichnungen initialisiert wird - diese Parameter lassen sich jedoch über das Bearbeitungsmenü konfigurieren (U3.F1, U4.F1), zu welchem man durch den *Bearbeiten*-Button gelangt. Zu diesen konfigurierbaren Optionen zählen sowohl die Titelwahl, eine Beschreibung, eine Priorität (U5.F1,U5.F2), als auch eine Abgabefrist. In weiteren Sprint-Phasen könnte sich die Liste an Optionen erweitern. Des Weiteren bietet eine User-Story die Möglichkeit Tasks hinzuzufügen in Form einer ausklappbaren Liste. Uns ist bewusst, dass in diesem Sprint die Implementierung dessen noch nicht vorgesehen ist, weshalb hierfür keine Funktionalitäten vorhanden sein werden, wollen aber lediglich die Zugehörigkeit zu den User-Storys zeigen. Überdies bietet die Page die Möglichkeit verschiedene Anzeigen/Pages innerhalb des *Projekt-Managers* auszuwählen (lokaler Reiter) und Filteroptionen bezüglich der User-Storys zu nutzen, welche die Suche nach spezifischen Anforderungen erleichtert. Diese werden als Dropdown-Menü umgesetzt. Zur Anzeige gehören *User-Storys*, bei denen nur User-Storys angezeigt werden, *Tasks*, wobei nur Tasks aufgelistet werden, *Schätzungstracker*, wobei Statistiken und Auswertungen gezeigt werden bezüglich geschätztem Aufwand und benötigte Zeit zu den Tasks und *Planning Poker*, wo man im Team Aufwandsschätzung betreiben kann. Hierbei wird die Anforderung 8 der Projektaufgabe berücksichtigt. Dahingegen gehören zu dem Filter die Auswahl, *kein Filter*, Sortierung nach *Priorität* und *Abgabefrist*. Zuallerletzt haben wir eine Suchleiste, durch welche auch die Navigation durch die verschiedensten Elemente der Seite(n) erleichtert wird. Dies erfüllt die Anforderung 9 der Projektaufgabe.

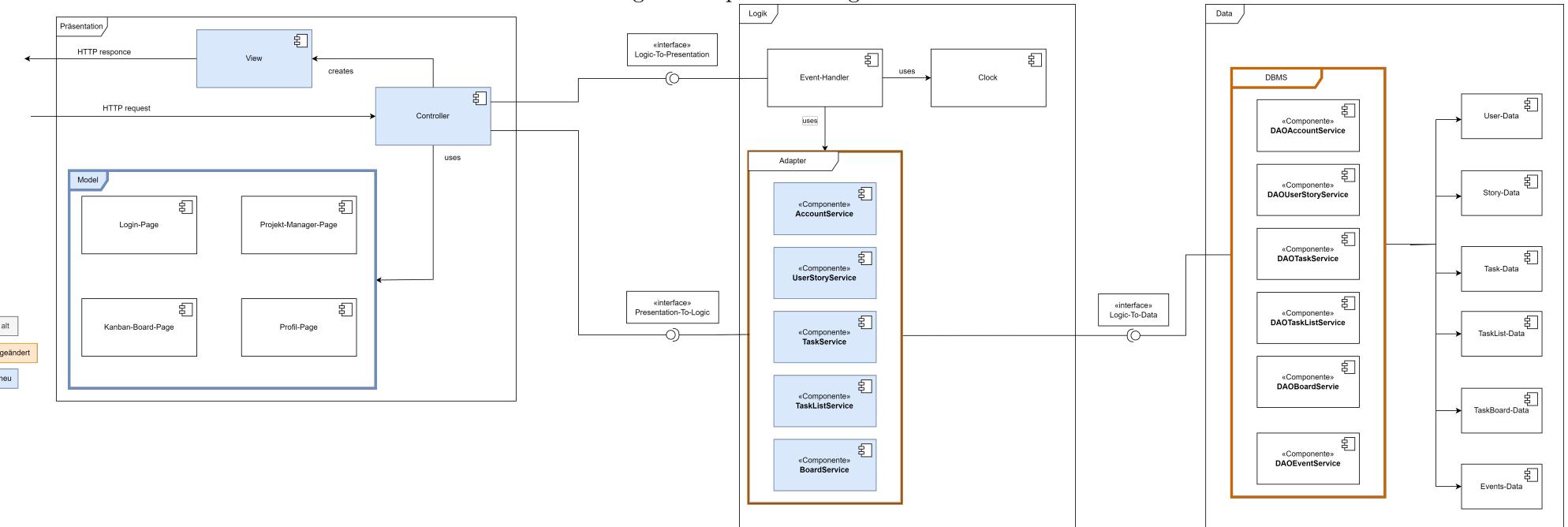
Abbildung 3: Layout Login-Seite



Für die Umsetzung von Anforderung 1 ist in Abbildung 3 unser Prototyp Layout für die Registrierung und die Anmeldung erkennbar. Aus diesen hat sich eine spezialisierte Page herauskristallisiert (Passwort ändern), die nachfolgend beschrieben wird. Zuallererst zur Registrierungsseite (A2.F1): Um sich auf unserer Seite zu registrieren, muss man Angaben zum *Username*, der *E-Mail* und dem *Passwort* treffen. Hat man sich registriert, wird man zur *Login Page* (A3.F1) weitergeleitet. Unter Verwendung der E-Mail und des Passworts kommt man bei erfolgreicher Authentifizierung zur Homepage *Projekt-Manager*. Wir haben uns entschieden zusätzlich Referenzen zwischen der Login- und Registrierungs-Seite hinzuzufügen in Form von Hyperlinks, um die Navigation zwischen diesen zu ermöglichen, falls man sich ungewollt auf einer der beiden befindet und auf die jeweils andere möchte. Außerdem haben wir die Möglichkeit eingebaut, das Passwort zurückzusetzen, falls dieses dem Nutzer entgangen ist. Zur Seite *Passwortänderung anfordern* kommt man durch die *Login-Page* unterhalb des Feldes *Passwort*. Unter Angabe der E-Mail kann man das Passwort zurücksetzen. Der Nutzer erhält einen Link zum Zurücksetzen per E-Mail nach Drücken von *Anfordern*, um das Passwort wie gefordert zu ändern. Falls einem bei der Anforderung zur Änderung des Passworts das Passwort eingefallen ist, kann man mittels des Hyperlinks unterhalb des *E-Mail-Feldes* wieder zurück zur *Login-Seite* kommen. Diese Funktionalität wird nicht in diesem Sprint umgesetzt.

## 2.2 Modellierung der verfeinerten Struktur

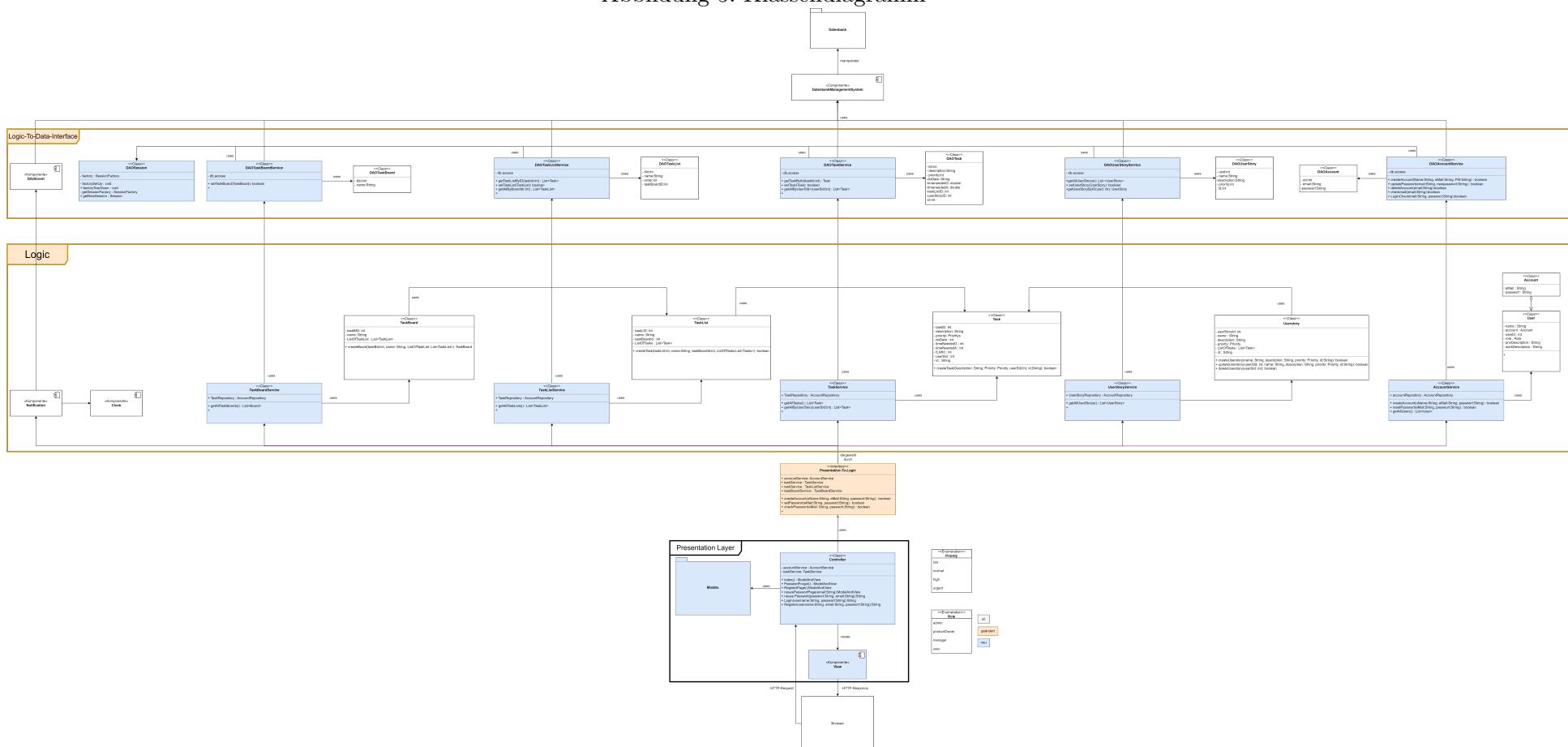
Abbildung 4: Komponentendiagramm



Um die für diesen Sprint geplanten User-Storys zu implementieren, haben wir uns für eine Änderung der benötigten Komponenten entschieden. Diese sind in Abbildung 4 farblich dargestellt. Auf der einen Seite wurde zu der bereits bestehenden *Drei-Schichten-Architektur* die *Model-View-Controller-Architektur* (MVC) innerhalb der Präsentationsschicht eingefügt. Es wurde diese Entscheidung getroffen, um die spätere Implementierung mit dem Framework Spring zu vereinfachen. Bezuglich des Wechsels zum Spring Framework wird allerdings ausführlicher in Technologien eingegangen, weshalb der Wechsel des Web-Frameworks hier nicht genauer erläutert wird. Aufgrund dessen, dass das Spring Web Framework die MVC-Architektur als eines seiner wichtigsten Designprinzipien ansieht, haben wir uns dazu entschieden, uns ebenfalls an das Framework anzupassen und die Architektur in unseren Entwurf zu implementieren. Dies erleichtert sowohl den Empfang von Nutzeranfragen, als auch die Weiterleitung dieser an weitere Komponenten, welche mit diesen weiterarbeiten. Zum anderen kann ein verbessertes Konzept von *Separation of Concerns* implementiert werden, da hier die Nutzersicht, die Bearbeitung von HTTP-Requests und die Lagerung der aktuellen Seiten getrennt werden kann.

Eine weitere Änderung ist der *Adapter*, zu welchem nun verfeinerte Komponenten hinzugefügt werden. Die Servicekomponenten stellen den jeweiligen Anforderungen, wie *Account*, *User-Story*, *Task*, *Task-List* und *Board*, Funktionen, welche vom *Event-Handler* genutzt werden können, zur Verfügung. Diese sollen - wie im vorherigen Komponentendiagramm - Funktionen zur Umwandlung von Anfragen der Präsentationsschicht an die Datenschicht bieten. Die gewählten *Servicekomponenten* wurden bewusst entsprechend der für diesen Sprint geplanten User-Storys ausgewählt. Weitere Komponenten sind in ihrer Funktionsweise grundsätzlich gleich geblieben. Zuletzt wurden im *DBMS* alle *Repository-Komponenten* entsprechend der User-Storys hinzugefügt.

Abbildung 5: Klassendiagramm



Für die Umsetzung der oben genannten User-Storys wurden ebenfalls einige Änderungen im Klassendiagramm hinzugefügt. Hierbei wurden die Klassen in zugehörige Komponenten eingeteilt. Neue Komponenten sind hier die Klassen der neuen *MVC-Architektur* in der *Präsentationsschicht*, die *Service-Klassen*, als auch die *DAO-Service-Klassen*<sup>1</sup>. Dabei wurden lediglich die ursprünglichen Komponenten *Präsentation* und *Logik*, als auch die *Interfaces* abgeändert. Die Begründung der Änderungen werden im obigen Text zum Komponentendiagramm erwähnt, weswegen in folgendem Text nicht näher darauf eingegangen wird.

Zuallererst wurden neue Klassen für die Implementierung der *MVC-Architektur* eingefügt. Der *Controller* greift auf *Model* zu und setzt anschließend den *View* für den Nutzer. Währenddessen besitzt der Controller Methoden, um jegliche Nutzeranfragen zu bearbeiten, weshalb der *Controller* das *Presentation-To-Logic-Interface* benötigt. Dementsprechend wurden einige Methoden des Interfaces abgeändert, um die Nutzeranfragen der *Präsentationsschicht* zu bearbeiten und an die *Logikschicht* weiterzuleiten. Eine weitere Neuerung sind die Service-Klassen, welche die Implementierung des *Presentation-To-Logic-Interfaces* darstellen. Hierbei wurden entsprechend der User-Storys jeweils *Service-Klassen* erstellt, dessen Aufgabe es ist, *Get-* und *Set-Methoden*, als auch Methoden zur Umsetzung von Nutzeranfragen anzubieten. Demzufolge greift jede *Service-Klasse* auf die zugehörige Klasse als Objekt zu, beispielsweise verwendet *UserStoryService* das Objekt *UserStory*. Weiterhin benutzen *Service-Klassen* jeweils eine gleichnamige *DAO-Service-Klasse*, um benötigte Daten aus der *Datenschicht* anfordern zu können, um mit diesen anschließend weiterarbeiten zu können. Die neuen *DAO-Service-Klassen* können hier als die Implementation des *Logic-To-Data-Interfaces* interpretiert werden, welche die benötigten Daten aus der *Datenschicht* für die jeweiligen Serviceanfragen der *Logikschicht* bereitstellen. Um dies umzusetzen, greift jede *DAO-Service-Klasse* auf das *DBMS* zu, welches die Anfragen schlussendlich in der *Datenbank* verarbeitet und an die *DAO-Service-Klassen* zurückgibt. Für jede *DAO-Service-Klasse* existiert ebenfalls ein passendes DAO-Klassenobjekt.

Damit die eingehenden Requests im *Controller* von der *DBMS* verarbeitet werden können, müssen diese umgeformt werden in für die *DBMS* verständlichen Anfragen. Hierfür nutzen wir das Design-Pattern Adapter, welches durch die gleichnamige Komponente umgesetzt wird. Dadurch müssen im *Controller* die Datenstrukturen vom restlichen System nicht integriert sein, was die Übersichtlichkeit und Skalierbarkeit des Systems fördert, indem die Klassen kaum verändert werden müssen durch neue Adapter oder neue Funktionalitäten des existenten Adapters.

Die Interfaces und deren Implementation werden im Code in eine Klasse zusammengeführt, wobei dessen Methoden in anderen Klassen ausgelagert sind. Zum Beispiel wird das *Presentation-To-Logic-Interface* von der *Presentation-To-Logic-Klasse* implementiert, wobei die Methoden in den Klassen *AccountService*, *TaskService* und *UserStoryService* kategorisch verlagert werden. Dies vereinfacht unsere Arbeit, da das Programm lesbarer wird, ohne die Wirkung eines Interfaces zu verlieren. Aufgrund dieser Herangehensweise verwenden wir das Singleton-Design-Pattern auf unsere Interface-Klassen. Somit wird sichergestellt, dass wir keine unnötigen Kopien des Interfaces ansammeln, um Speicherprobleme zu vermeiden. Weiterhin kreiert es ein globales Objekt, auf das alle Klassen zugreifen können. Jedoch kann das Objekt nur in der Klasse des Singleton-Objekts verändert werden, was die Services vor ungewünschten Veränderungen schützt. In den zukünftigen Sprints möchten wir für die Synchronisation und Notifikation das *Observer-Pattern* anwenden. Um die Synchronisation zu integrieren, wird für die möglichen Seiten der Website jeweils ein *Observer* kreiert, der die dazugehörigen Daten in der Datenbank betrachtet. Der *Observer* verfolgt, welche Nutzer aktuell auf der Seite sind. Bei Änderungen in der Datenbank wird die Webseite der aufgelisteten Nutzer durch den *Event-Handler* zum Aktualisieren gezwungen. Für die Benachrichtigung wird aus den *Events* ausgelesen, welche Nutzer beteiligt sind. Sobald die zeitliche Entfernung zum *Event* einen vordefinierten Punkt unterschreitet, werden allen aufgelisteten Nutzern eine Benachrichtigung vom *Event-Handler* geschickt mittels der Methode *Notification()*.

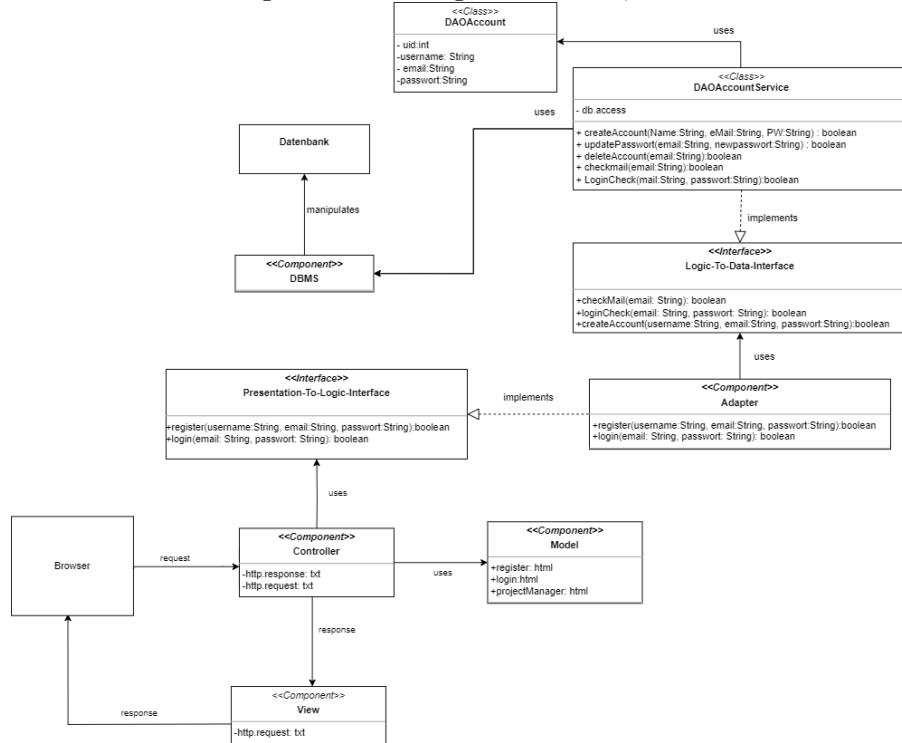
---

<sup>1</sup>DAO = Data Access Object

## 2.3 Modellierung der verfeinerten Interfaces und Datenstrukturen

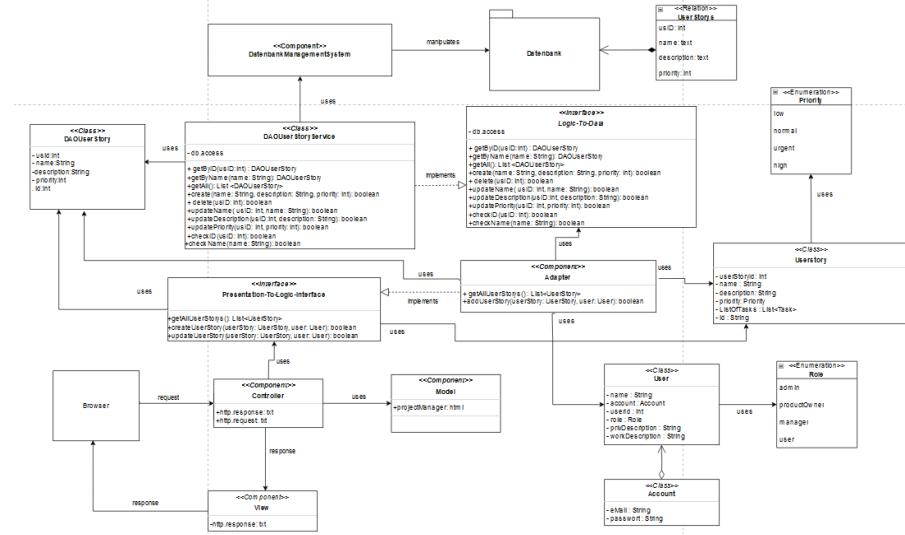
### 2.3.1 User-Storys zum Account

Abbildung 6: Klassendiagramm für A1, A2 und A3



Die User-Story A1 verlangt eine Speichermöglichkeit der Account-Daten, was durch die Relation *Users* realisiert wird. Damit die User-Storys A2 und A3 umgesetzt werden können, wird die Datenbank abgefragt, weshalb die Interfaces zwischen den Schichten des Systems notwendig sind. Insbesondere das *Presentation-To-Logic-Interface* und *Logic-To-Data-Interface*, weil der Prozess eine Anfrage ist, weshalb die Interfaces benötigt werden, die eine Verbindung vom *Controller* zur *Datenbank* kreieren. Der *Controller* nimmt dem Request vom *Browser* mit E-Mail, Passwort bei einem Login und zusätzlich den Nutzernamen im Falle einer Registrierung entgegen. Der *Adapter*, welcher das *Presentation-To-Logic-Interface* implementiert und somit die Methodenaufrufe des *Controllers* durchführt, dient zur Übersetzung des Methodenaufrufs in die äquivalenten Methodenaufrufe über das *Logic-To-Data-Interface* an den *DAOAccountService*, der mittels *DBMS* die Datenbank abfragt. Damit die *DBMS* die Anfrage an die Datenbank stellen kann, wird die E-Mail als Schlüsselattribut benötigt, um die Eindeutigkeit der Ausgabe sicherzustellen. Beim Login wird das Passwort mit dem abgespeicherten Wert verglichen. Für die Registrierung wird, wenn die E-Mail nicht gefunden wurde, E-Mail, Passwort und Nutzernamen benötigt, um ein neues Tupel in der Relation *DAOAccount* einzufügen. Also benötigt die *DBMS* zwei oder drei Strings. Da der *DAOAccountService* die *DBMS* und dessen Operationen verwendet, bekommen die Methoden dieselben zwei oder drei Strings als Argumente übergeben. Somit werden im *Logic-To-Data-Interface* nur Strings benötigt. Durch die Rolle des *Adapters* sind die Argumente dessen Methoden identisch zu den des *Logic-To-Data-Interface*, als auch des *Presentation-To-Logic-Interfaces*. Das heißt E-Mail, Nutzernname und Passwort werden unverändert in den Interfaces genutzt. Somit sind nur Strings und keine zusätzlichen Datentypen benötigt.

Abbildung 7: Klassendiagramm für U1, U3, U4 und U5



### 2.3.2 User-Storys zur User-Story

Die User-Storys U1, U3, U4 und U5 verlangen, dass man alle User-Storys anzeigen, neue User-Storys erstellen und existierende User-Storys bearbeiten kann. Um eindeutige Anfragen an die Datenbank zu stellen, benötigt die DBMS eine *User-Story-ID*, das Schlüsselattribut der Relation User-Story. Jedoch werden alle weiteren Attribute einer User-Story gebraucht, sobald eine User-Story erstellt oder verändert werden soll. Da die Methoden der *DAO-Service-Klassen* atomar aufgebaut sind, sind die Datentypen derer Argumente primitiver Art (wie Integer und String). Die Rückgaben der Methoden *getByID*, *getByName* sind Objekte des neudefinierten Datentyps *DAOUserStory*. Da der *DAOUserStoryService* das *Logic-To-Data-Interface* implementiert, benutzt das Interface neben den primitiven Datentypen den *DAOUserStory*-Typ. Im *Adapter* werden die Methoden des *DAOUserStoryServices* verwendet, weshalb der Adapter den Datentyp *DAOUserStory* verwendet. Damit der Controller die Rückgaben der *Adapter*-Methoden verwerten kann, muss der *Adapter* die *DAOUserStory*-Objekte in Objekte des Typs *UserStory* umwandeln. *DAOUserStory* und *UserStory* ähneln sich stark, jedoch differenzieren sie sich im Datentyp für die Priorität. Hier verwendet der Typ *UserStory* keinen Integer wie in *DAOUserStory*, sondern einen selbstdefinierte Typ *Priorität*. Dieser besteht aus vier Elementen *low*, *normal*, *high* und *urgent*. Wir haben zwei verschiedene Datentypen verwendet, weil die Speicherung der Priorität mit Integer im Gegensatz zu Strings besser ist und beim Programmieren es deutlich übersichtlicher ist, mit *low*, *normal*, *high* und *urgent* anstatt mit Integer zu arbeiten. Zum Erstellen und Bearbeiten von User-Storys benötigt der Nutzer gewisse Rechte, weshalb die Methode *addUserStory()* nicht nur ein Objekt vom Typ *User-Story*, sondern auch ein Objekt des Typs *User* übergeben bekommt. Das Prüfen der Rolle wird in diesem Sprint nicht umgesetzt, weshalb im Programm nur die User-Story übergeben wird. Somit werden *User-Story* und *DAOUserStory* (und *User*) mit deren spezifizierten Datentypen *Priorität* (und *Role*) im *Presentation-To-Logic-Interface* verwendet, weil der *Adapter* das Interface implementiert.

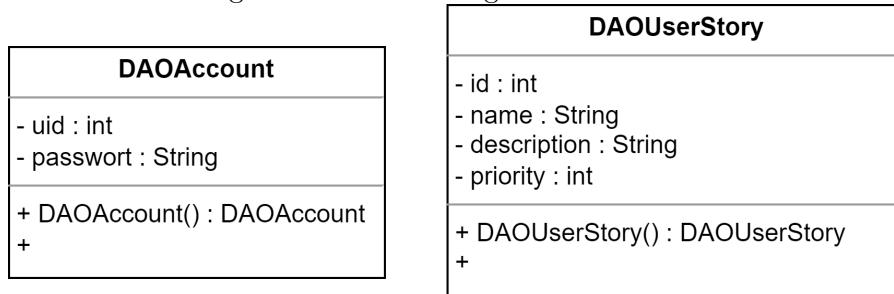
### 2.3.3 Modellierung umgesetzte Datenstrukturen

Accounts, sowie User-Storys sollten auch nach dem Neustart des Programms erhalten bleiben, daher ist das Speichern der Daten unabdingbar. Aus diesem Grund benutzen wir die SQLite Datenbank. Da Accounts und User-Storys jedoch nicht gleich sind, müssen diese in unterschiedlichen Tabellen gespeichert werden. Hinsichtlich der Erweiterungen in den nächsten Sprints, haben wir uns dazu entschieden den Account schon in die Tabelle für User zu integrieren. Dies hält die Datenbankstruktur klein und übersichtlich, was das Arbeiten mit der Datenbank erleichtert, jedoch die Funktionalitäten von Hibernate nicht eingeschränkt.

Jeder User hat eine einzigartige ID, welche durch die Integration des Accounts in die User Tabelle auch für den Account selbst gilt. Für den Account werden E-Mail und Passwort gespeichert, dabei muss die E-Mail auf der Datenbank einzigartig sein. So verhindern wir, dass eine Mail mehrere User hat. Das Passwort wird in zukünftigen Sprints als Hash-Wert gespeichert.

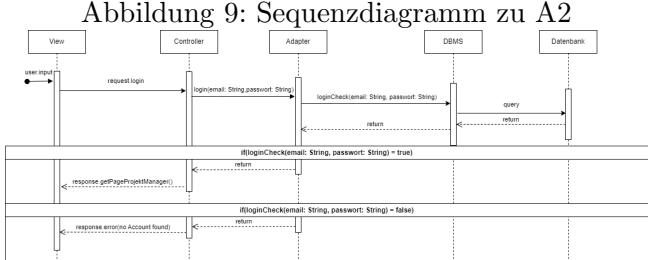
User Story besitzen eine einzigartige ID, sowie einen einzigartigen Namen. So können User-Storys leicht unterschieden werden. Weiterhin verhindert dies Verwechslungen. Die ID wird zusätzlich für die Datenbank selber verwendet. Des Weiteren wird für eine User Story die Beschreibung und die Priorität gespeichert.

Abbildung 8: Als Klassen umgesetzte Datenstrukturen

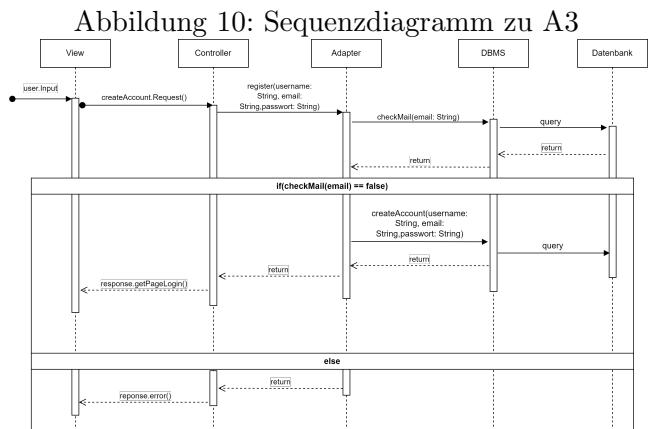


Die bisher umgesetzten Datenstrukturen repräsentieren die gespeicherten Daten der Datenbank und sind deshalb sehr nah an der Umsetzung der Datenbank modelliert. Grundlegend umfassen die Methoden der beiden Datenstrukturen Getter und Setter, welche zur Übersichtlichkeit in den Klassendiagrammen 8 nicht eingebunden wurden.

## 2.4 Modellierung des verfeinerten Verhaltens



DMBS um mit *loginCheck()* über *DAOAccountServices* zu überprüfen, ob die eingegebenen Daten mit dem in der Datenbank existierenden Datensatz übereinstimmen. Die *DMBS* nimmt diese Anfrage entgegen und stellt der Datenbank die hierzu benötigte Anfrage, welche in der Query verarbeitet wird. Sie schickt dann eine entsprechende Antwort an das *DMBS* zurück, der die Rückgabe des nun beendeten Methodenaufrufs dem Adapter übergibt. Liegt eine “True“-Reponse vor, so wird durch die View die Seite des Projekt-Managers geladen, andernfalls wird eine Fehlermeldung angezeigt, dass kein zugrundeliegender Datensatz zu diesem Account existiert, das heißt der Account nicht gefunden wurde. Ist der User im Projekt-Manager, so kann nun auf alle weiteren Seiten zugegriffen werden.

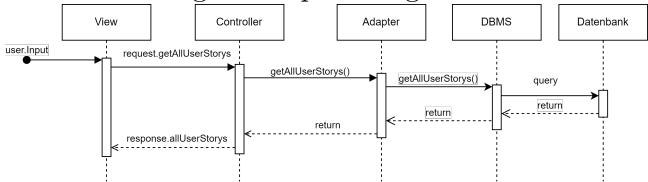


weiter, welches dann in die Datenbank übernommen wird. Dabei wandelt es die Eingabe in eine Query um, so dass die Datenbank nach Bearbeitung der Anfrage der *DBMS* antwortet. Somit endet der Methodenaufruf und die *DBMS* übermittelt das Ergebnis an den Adapter. Liegt eine “True“-Response vor, so heißt das es liegt bereits ein Account mit der angegebenen E-Mail vor und es wird eine Fehlermeldung übermittelt in Form einer Nachricht an die Webpage. Andernfalls wird vom Adapter *createAccount()* aufgerufen und in der Datenbank wird folgend ein neuer Datensatz durch *DAOAccountService* angelegt. Das erfolgreiche Anlegen des Datensatzes wird, in Form einer “True“-Response bis an den Controller rückübermittelt. Dieser ändert den View dann auf die *Login-Page*. Dort kann sich der User nun mit seinem neu angelegten Account anmelden.

Möchte sich der User einloggen, so stellt Sequenzdiagramm A2 das Verhalten des Systems auf diese Anfrage dar. *user.input* beschreibt dabei die Eingabe der E-Mail und des Passworts innerhalb der View, also der Login Page. Der Betätigung des Login-Buttons durch den User folgend wird vom Browser eine HTTP-Request an den Controller geschickt, der durch den Controller die *login()*-Methode initialisiert. In Folge dieses Ereignisses schickt der Adapter eine Anfrage an das

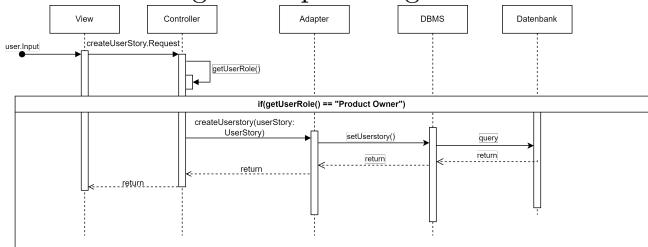
Möchte sich der User hingegen registrieren, so charakterisiert Sequenzdiagramm A3 das Verhalten des Systems auf diese Anfrage. Wie bereits im Sequenzdiagramm zuvor folgt nach der Aktion des Nutzers, welche durch *user.input* modelliert wird, eine Registrieranfrage vom View zum Controller. Der Controller verarbeitet diesen HTTP-Request und unter Zuhilfe des Adapters wird *register()* ausgeführt. Bevor der Account erstellt wird mittels *createAccount()*, wird durch *checkMail()* geprüft, ob bereits ein Account in den Datensätzen der Datenbank existiert, dass mit der Mail verbunden ist. Dafür ruft der Adapter die eben genannte Methode über *DAOAccountService* auf und leitet ein *User-Object* an das *DBMS*

Abbildung 11: Sequenzdiagramm zu U1



mittels des Adapters die `getAllUserStorys()`-Methode ausführt. Diese Methode stellt eine Anfrage an das DBMS dar und gibt aus der Datenbank alle gespeicherten `DAOUserStory`-Objekte zurück. Der Adapter strukturiert daraufhin die zurückgegebenen `DAOUserStory` zu `UserStory`-Objekten um und gibt diese dann an den Controller weiter. Letztlich übergibt der Controller einen neuen View mit den eingesetzten Daten an den Nutzer. Auf der Seite kann er nun alle User-Storys einsehen.

Abbildung 12: Sequenzdiagramm zu U3

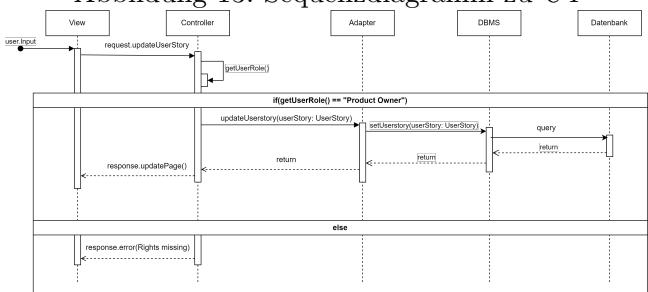


iert hat. Ist das der Fall, so wird vom Controller im Adapter der Methodenaufruf `createUserstory()` gestartet. Dieser strukturiert das `UserStory`-Objekt in ein `DAOUserStory`-Objekt um und ruft `setUserstory()` über das `DBMS` auf. Dieser sendet eine Anfrage an die `Datenbank` durch die `Query`, um einen neuen Datensatz in der Tabelle `UserStorys` zu erstellen. Ist dies erfolgreich, so wird eine "True"-Reponse bis an den Controller weitergeleitet.

Entscheidet sich ein Nutzer nach User-Story U1 alle User-Storys seines Projekts anzeigen zu lassen, also den Projekt-Manager zu öffnen, so beschreibt das Sequenzdiagramm U1 das darauf folgende Verhalten. *user.input* beschreibt die Aktion des Nutzers, den Projekt-Manager öffnen zu wollen. Der Aktion des Nutzers folgt ein HTTP-Request an den Controller, welcher daraufhin

Will ein Nutzer eine User-Story erstellen, so wird im Moment des Speichern innerhalb der Bearbeitungsmenüs ein *user.input* registriert (dieser beeinhaltet Titel, Beschreibung und Priorität), der mittels eines Formulars abgeschickt wird. Danach wird von der View mittels eines HTTP-Requests die Anfrage an den Controller übermittelt mit einer *UserStory*-Instanz. In diesem wird nun die Methode *getUserRole()* aufgerufen, bei dem geprüft wird, ob der Product Owner die Anfrage der User-Story Erstellung initiiert.

Abbildung 13: Sequenzdiagramm zu U4



Möchte man eine bereits vorhandene User-Story verändern, so gibt Sequenzdiagramm U4 Auskunft über die Reaktion des Prozesses auf diese Anfrage. Durch einen *user.input* durch das Bearbeitungsmenü (dieser beinhaltet Titel, Beschreibung und Priorität) und dem darauffolgenden Speichern wird vom View eine Aktualisierungsrequest bezüglich der User-Storys an den Controller übermittelt, als auch eines *Userstory-Objekts*. Dieser prüft wie in Sequenzdiagramm U3 durch einen Selbstaufruf mittels *getUserRole*, ob der Product Owner diese Änderungsanfrage

sendete. Ist dies nicht der Fall, so wird eine Nachricht an die View übermittelt, dass die Rechte zur Änderung fehlen. Andernfalls wird vom Controller über den Adapter `updateUserstory()` aufgerufen und vom Adapter über das DBMS die Methode `setUserstory()`. Das DBMS stellt nun die Änderungsanfragen an die Datenbank in der *Query*. Hat die Datenbank die Datensätze erfolgreich aktualisiert, so gibt diese eine “True-Response“ bis an den Controller, und in der View wird die entsprechende User-Story aktualisiert.

## 2.5 Tracing

Tabelle 1: Tracing umgesetzter User-Storys - Sprint 1

ID	modellierte Klassen	Umsetzung
A1	Abbildung 8	Implementation der modellierten Datenstruktur in der Datenbank
A2	Abbildung 6	Erstellen von visueller Benutzeroberfläche, Requestmapping für HTTP Request und Implementation Speicherung der Daten
A3	Abbildung 6	Erstellen von visueller Benutzeroberfläche, Requestmapping für HTTP Request und Implementation Abgleich der Daten mit DB
U1	Abbildung 7	Erstellen von visueller Benutzeroberfläche, Requestmapping für HTTP Request und Implementation Ausgabe aller User-Storys aus DB
U3	Abbildung 7	Erstellen von visueller Benutzeroberfläche, Requestmapping für HTTP Request und Implementation Speicherung der Daten
U4	Abbildung 7	Erstellen von visueller Benutzeroberfläche, Requestmapping für HTTP Request und Implementation Speicherung der neuen Daten
U5	Abbildung 8	Erweitern der <i>DAOUserStory</i> Datenstruktur und die <i>UserStory</i> -Relation der Datenbank um <i>priority</i>

## 3 Implementierung und Tests

### 3.1 Updates Technologien

Tabelle 2: Ersetzte Technologien

Technologie	Ersetzt durch	Motivation
Jakarta Servlet	Spring	<ol style="list-style-type: none"> <li>1. Benutzerfreundlichkeit</li> <li>2. Bessere Dokumentation und Tutorials</li> <li>3. Integriertes einfaches HTTP Request Handling</li> <li>4. Annotations für wichtige/häufig genutzte Befehle</li> </ol>
Apache Tomcat	Spring Boot	<ol style="list-style-type: none"> <li>1. Für Spring entwickelt</li> <li>2. Ermöglicht deployment ohne externe Software in eigener IDE</li> </ol>

### 3.2 Dokumentation der Codequalität

#### 3.2.1 Abweichung des Codes von der geplanten Architektur/Design

Erste minimale designtechnische Abweichungen vom Projekt-Manager Prototyp sind bereits in der User-Story zu sehen bezüglich des jetzigen umgesetzten Prototyps. Der ursprünglich geplante *Bearbeiten-Button* befindet sich nicht mehr im unteren Bereich, sondern neben dem Titel - quasi als Fensterbereich mit Symbolen – da es intuitiver/platzsparender ist. Der „*User Story hinzufügen*“-Button ist nun oberhalb der User-Story und über diesen sind die Dropdown-Menüs *Ansicht* und *Filter*. Deren Optionen

haben sich aber nicht geändert. Zuletzt hat sich die Anordnung der User-Story verändert. Diese sind nun nicht mehr als Paare untereinander, sondern alle untereinander, da es für den jetzigen Stand einfacher umzusetzen war und keine Größenanpassungen erfordert. Grundsätzlich lassen wir uns für zukünftige Sprints die Freiheit je nach Belieben designtechnische Änderungen zu treffen, wenn nötig.

Zusätzlich zur geforderten Editier- und Erstellungsfunktion haben wir noch eine Zoomfunktion implementiert, um das gesamte User-Story Konstrukt in seiner Gesamtheit zu betrachten mitsamt der Tasks. Da es im Sprint nicht gefordert war, ist es eine Zusatzfunktion. Da diese im Verlauf des Sprints zu Fehlern führte, wurde die Funktionalität erstmal herausgenommen. Hierbei ist geplant, die Zoom-Funktion in späteren Sprints zu implementieren. Um die geplante Funktion trotzdem zu zeigen, existiert nun eine Lupe auf der User-Story. Gedacht war es, da wir [vorerst] die User-Storys untereinander anordneten und es lästig sein kann, bei [theoretisch unendlich] langen, ausgeklappbaren *Task-Listen* durch die Seite zu navigieren oder ohne Nutzung der noch nicht umgesetzten Suchleiste eine bestimmte User-Story zu finden. Die Tasks einer User-Story wurden allerdings erstmal im Code auskommentiert und sind somit nicht vollständig umgesetzt, da wir uns bei automatischen Positions-anpassungen der Tasks beim Ausklappen unsicher waren und die User-Storys sich dadurch im jetzigen Stadium überschneiden würden.

### 3.2.2 Automatische Tests

Tabelle 3: Automatische Tests

TestArt.TestID.Run	Erstellungszeit	Durchlauf	Herkunft	Ergebnis	Grund
HTTP.T1.1	1.02.2024	1.02.2024 21.30	Abbildung 10	Fail	400 Bad Request: Falsche Request Methode
HTTP.T1.2	1.02.2024	8.02.2024 12.30	Abbildung 10	Fail	Null Exception, weil Request Parameter nicht übermittelt
HTTP.T1.3	1.02.2024	8.02.2024 13.00	Abbildung 10	Pass	-
HTTP.T2.1	1.02.2024	1.02.2024 21.30	Abbildung 9	Fail	400 Bad Request: Falsche Request Methode
HTTP.T2.2	1.02.2024	8.02.2024 12.30	Abbildung 9	Fail	Null Exception, weil Request Parameter nicht übermittelt
HTTP.T2.3	1.02.2024	8.02.2024 13.00	Abbildung 9	Pass	-
HTTP.T3.1	1.02.2024	1.02.2024 21.30	Abbildung 5	Fail	400 Bad Request: Falsche Request Methode
HTTP.T3.2	1.02.2024	8.02.2024 12.30	Abbildung 5	Pass	-
HTTP.T4.1	1.02.2024	1.02.2024 21.30	Abbildung 5	Fail	400 Bad Request: Falsche Request Methode
HTTP.T4.2	1.02.2024	8.02.2024 12.30	Abbildung 5	Pass	-
HTTP.T5.1	9.02.2024	9.02.2024 10.30	Abbildung 12 / 13	Pass	-
Logic.T1.1	9.02.2024	9.02.2024 15.30	Abbildung 12	Pass	-
Logic.T2.1	9.02.2024	9.02.2024 15.30	Abbildung 13	Pass	-

TestArt.TestID.Run	Erstellungszeit	Durchlauf	Herkunft	Ergebnis	Grund
DB.Acc.T1.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-
DB.Acc.T2.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-
DB.Acc.T3.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-
DB.Acc.T4.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-
DB.Acc.T5.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-
DB.Acc.T6.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-
DB.US.T1.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-
DB.US.T2.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-
DB.US.T3.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-
DB.US.T4.1	9.02.2024	9.02.2024 14.30	DS (Grobentwurf)	Pass	-

Für die automatischen Tests haben wir uns allgemeinen gegen generierte Tests entschieden um die Möglichkeit zu haben auf jede Anforderung für unser System darauf speziell angepasste Tests zu schreiben. Darauf hinaus wurden alle Tests als White-Box Tests konzipiert um eine möglichst große Line-Coverage zu gewährleisten. Dadurch umfassen die derzeitigen eine mindest Line-Coverage von 95% und eine Branch-Coverage von mindestens 65%.

(DB.Acc.T1, DB.Acc.T5 und DB.Acc.T6 laufen nur einzeln, da die DB derzeitig nur einzelne Anfragen ähnlicher Art abhandeln kann)  
(DS = Datenstruktur)

### 3.2.3 Manuelle Tests

TestID.Durchlauf	getestet um	Resultat
INT.T1.1	10.02.2024 19.45	Pass
INT.T1.2	10.02.2024 20.00	Pass
INT.T1.3	11.02.2024 17.10	Pass
INT.T1.4	11.02.2024 17.10	Pass
AT.T1.1	11.02.2024 17.15	Pass

## 3.3 Tracing

Tracing vom umgesetzten Komponenten im Code:

Komponente	Verweis zum Code
Login-Page	<a href="#">index.html</a>
Project-Manager-Page	<a href="#">projectManager.html</a>
Controller	<a href="#">Controller.java</a>
AccountService (Adapter)	<a href="#">AccountService.java</a>
UserStoryService (Adapter)	<a href="#">UserStoryService.java</a>
DAOAccountService	<a href="#">DAOAccountService.java</a>
DAOUserStoryService	<a href="#">DAOUserStoryService.java</a>
Datenbank	<a href="#">DB/database.db</a>

### 3.4 Laufender Prototyp

Abbildung 14: User-Story Buttons

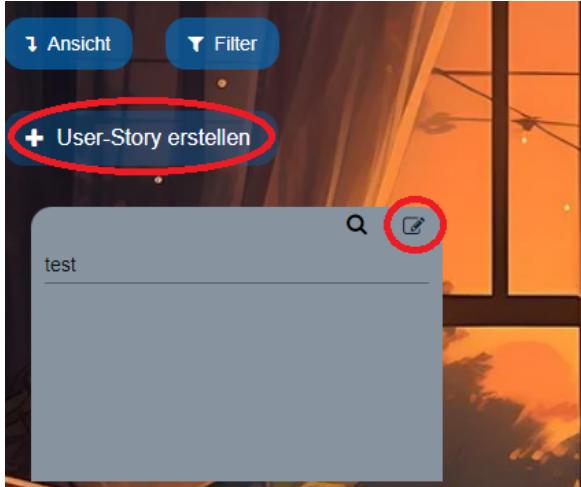


Abbildung 15: Bearbeitungsmenü



Im laufenden Prototyp sind die im Sprint-Backlog genannten User-Storys und Tasks aufzufinden. Zu Beginn erscheint die Login-Seite. Hier kann man sich bei Eingabe seiner Accountdaten (E-Mail und Passwort) einloggen oder registrieren, indem man unten bei "kein Account" auf den Hyperlink drückt. Dies erfüllt sowohl die Anforderung 1 in der Projektaufgabe, als auch die User-Storys A1, A2, A3 und die entsprechenden Tasks dazu. Es existiert ebenfalls ein Hyperlink für das Zurücksetzen des Passworts. Diese Funktion wurde allerdings noch nicht implementiert, da dies für den nächsten Sprint vorgesehen ist.

Hat man sich erfolgreich angemeldet, wird man zum Projekt-Manager weitergeleitet. Hier kann man beim Klicken auf den "User-Story erstellen"-Button User-Storys hinzufügen (Anforderung), aber auch User-Storys editieren, indem man auf den *Editieren-Button* auf einer User-Story drückt (Buttons in Abbildung 14 eingekreist). Dies erfüllt die geplanten User-Storys U1, U3, U4 und deren Tasks. Dabei wird auch die Anforderung 2 erfüllt, wobei noch alle Nutzer diese Funktion besitzen. Es wurde noch nicht implementiert, dass lediglich der *Product Owner* User-Storys erstellen kann. Beim Drücken des *Editieren-Buttons* sollte ein Fenster zum Editieren angezeigt werden, bei welchem der Titel, die Beschreibung, Priorität (U5, Teil von Anforderung 3) und die Abgabefrist angegeben werden kann (in Abbildung 15 dargestellt). Die Funktionen für die *Reiter-Buttons*, als auch die *Filter-, Suche-, Anzeige- und Zoom-Buttons* sind noch nicht implementiert. Die Suche- und Filterfunktion wird für die Anforderung 9 benötigt, die Anzeige beinhaltet den *Schätzungstracker* für Anforderung 8. Diese sind allerdings nicht für diesen Sprint geplant.

Eine Installationsanweisung kann in GitLab unter <https://git.informatik.uni-rostock.de/softwaretechnik-ws-2023-24/uebung3/team3/projekt-team3-uebung3/-/blob/main/Sprint1/Installationsanweisung.pdf> gefunden werden.

### 3.5 Abweichungen von Sprintplanung

Fast alle für diesen Sprint geplanten User-Storys wurden durch die Teams vollständig implementiert. Lediglich Task U3.B1 und U4.B1 wurden nur partiell umgesetzt, da User noch keine Rechte haben aufgrund fehlender Implementation. Dies ist dem zuschulden, dass die Recherchearbeit den zeitlichen Rahmen dieses Sprints gesprengt hätte.

Für die zeitliche Abweichung der Sprintplanung gibt es eine Vielzahl an Gründen. Zum einen wurde der Sprint kurz vor der vorlesungsfreien Zeit begonnen, weshalb ein verzögerter Start des Sprints eingetreten ist. Durch diese Verzögerung wurde der Zeitraum in die Prüfungsphase ausgelagert, was zu noch mehr Verzögerungen bezüglich der Planung führte. Zum anderen wurde zuerst die Implementierung des Prototyps, anstelle der korrekten Planung des Prototyps von den Projektmitgliedern fokussiert. Aufgrund dessen, dass sich jedes Mitglied zunächst in die benötigten Technologien für seine einteilte Gruppe einlesen und dementsprechende Recherchearbeit leisten musste, wurde die eigentliche Planung zeitlich nach hinten verschoben. Dies führte zu vielen Fehlern während der Implementierung. Beispielsweise konnten einige Tasks erst angefangen werden, wenn eine andere Task erfüllt wurde. Das heißt, es gab Abhängigkeiten zwischen Tasks, die nicht beachtet wurden, wodurch die Zeit nicht effektiv genutzt werden konnte (siehe Abhängigkeiten im BPMN Sprint 1). Da es jedoch Verzögerungen innerhalb der Teams bezüglich den Bearbeitungszeitpunkten gab, kam es zu längeren Phasen, wo eine Gruppe nicht weiterarbeiten konnte. Somit wurden einige Tasks zeitlich weiter verzögert.

Die längere Bearbeitungszeit kann ebenfalls damit begründet werden, dass sich innerhalb der Implementierung wiederkehrend Fehler eingebaut haben, welche ebenfalls durch die Unwissenheit der Mitglieder entstanden sind. Dies hatte ebenfalls zu Folge, dass die oben genannten Tasks U3.B1 und U4.B1 nicht implementiert wurden.

Durch die mangelnde zeitliche Planung der Tasks kam es außerdem dazu, dass Mitglieder zuerst überflüssige Zusatzfunktionen implementiert hatten, welche letztendlich keine konkrete User-Story erfüllt haben. Demzufolge wurden wichtigere Tasks anfangs übersehen, und somit nicht effizient bearbeitet. Ebenfalls wurde zu Beginn des Sprints kein konkretes Layout entworfen, sondern erst während der Implementierungen. Aus diesem Grund wurden einige visuelle Funktionen nicht richtig abgesprochen und teilweise auch komplett verworfen.

Ein weiterer Aspekt ist die Schwierigkeiten bei der Verwendung von Technologien. Hierbei wurde sich zunächst auf die Technologien Jakarta Servlet und Apache Tomcat fokussiert. Nach mehreren Fehlschlägen diese Anwendungen umzusetzen, aufgrund von mangelnder Dokumentation und Erklärungen/Verständnis, wurde sich letztendlich für das Framework Spring entschieden, welches die Implementierung um einiges erleichtert hatte.

Zuallerletzt wurde zwar die Agile Praktik des Test-Driven-Developements angesetzt, aber nicht erfolgreich durchgeführt. Die Tests wurden erst nahezu am Ende des Sprints richtig geschrieben, sodass im Verlauf der Implementierung nicht ganz klar war, welche Funktionen für die Umsetzung der User-Storys wirklich gefordert sind. Weiterhin hätten bereits bestehende Tests die Fehlervermeidung um einiges erleichtert und künftige Wartungsarbeiten womöglich verringert, da nur das geschrieben wird, was zur Erfüllung des Tests benötigt wird.

**Lessons Learned:**

1. Test-Driven-Developement sollte eingehalten werden, um Fehleranfälligkeit zu reduzieren
  2. Vorherige strukturelle Planung der Tasks durch das Modell BPMN, um lange Wartezeiten aufgrund von Abhängigkeiten zu vermeiden
  3. Tiefgründigere Auseinandersetzung mit den Tasks, um keine unnötigen zusätzlichen Funktionen zu implementieren
  4. Früherer Entwurf des Seiten-Layouts, um Missverständnisse bei der Implementierung zu vermeiden
  5. Verwendung eines Programms für die Planung von Teamaufgaben
  6. Genauere Absprache zwischen Product Owner und Entwickler, als auch zwischen den Entwicklern, um Missverständnisse zu vermeiden
-