

BOOKIES

PROJEKTDOKUMENTATION

Programmierprojekt, Sommersemester 2025

Umut Avci
Samet Avci
Ayberk Cagli
Halimenur Can
Yusuf Eren Colak
Taha Emircan Gokduman

Inhaltsverzeichnis

1 Einleitung	6
1.1 Kontext der Arbeit	6
1.2 Motivation für diese Arbeit	6
1.3 Zielstellung für diese App	6
2 Technische Grundlagen	7
2.1 Android-Entwicklung mit Kotlin & XML	7
2.2 Retrofit & Netzwerkarchitektur	7
2.3 Java, Spring Boot & REST API	7
2.4 PostgreSQL & Datenbankmodellierung	7
3 Analyse der Anforderungen	8
3.1 Beschreibung des Ist-Standes	8
3.2 Funktionale Anforderungen	8
3.2.1 Mobile Seite	8
3.2.1.1 Benutzeroauthentifizierung und Kontoverwaltung von Halime	8
3.2.1.2 Startseite (HomePage) Komponenten von Halime und Samet	9
3.2.1.3 Such- und Filterfunktionen von Samet	10
3.2.1.4 Bücherverwaltung und Interaktionen von Samet, Halime .	11
3.2.1.5 Benachrichtigungssystem von Ayberk, Samet	12
3.2.1.6 Profilseite und persönliche Inhalte von Ayberk, Samet . .	13
3.2.2 Backend Seite	13
3.2.2.1 Auth Use Cases von Umut	14
3.2.2.1.1 UC-AUTH-01: Benutzerregistrierung	14
3.2.2.1.2 UC-AUTH-02: Benutzer-Login	15
3.2.2.1.3 UC-AUTH-03: Benutzer-Logout	15
3.2.2.2 Book Use Cases von Umut und Yusuf	16
3.2.2.2.1 UC-BOOK-01: Bücher automatisch importieren (Google Books API)	16
3.2.2.2.2 UC-BOOK-02: Buchdetails anzeigen	17
3.2.2.2.3 UC-BOOK-03: Bücher suchen und filtern	18
3.2.2.3 Comment Use Cases von Umut und Yusuf	19
3.2.2.3.1 UC-COMMENT-01: Kommentar erstellen	19
3.2.2.3.2 UC-COMMENT-02: Kommentare anzeigen	19
3.2.2.3.3 UC-COMMENT-03: Kommentar löschen	20
3.2.2.4 Follower Use Cases von Umut und Yusuf	21

3.2.2.4.1	UC-FOLLOW-01: Benutzer folgen	21
3.2.2.4.2	UC-FOLLOW-02: Benutzer entfolgen	22
3.2.2.4.3	UC-FOLLOW-03: Gefolgte Benutzer anzeigen . .	22
3.2.2.4.4	UC-FOLLOW-04: Follower-Liste anzeigen	23
3.2.2.5	List Use Cases von Umut und Yusuf	23
3.2.2.5.1	UC-LIST-01: Buchliste erstellen	23
3.2.2.5.2	UC-LIST-02: Buch zu Liste hinzufügen	24
3.2.2.5.3	UC-LIST-03: Buch aus Liste entfernen	25
3.2.2.5.4	UC-LIST-04: Öffentliche Liste anzeigen	26
3.2.2.5.5	UC-LIST-05: Einer Liste folgen	26
3.2.2.5.6	UC-LIST-06: Liste entfolgen	27
3.2.2.5.7	UC-LIST-07: Gefolgte Listen anzeigen	28
3.2.2.6	Notification Use Cases von Yusuf	28
3.2.2.6.1	UC-NOTIFY-01: Benachrichtigung bei neuem Fol-	
	lower	28
3.2.2.6.2	UC-NOTIFY-02: Benachrichtigung bei List-Follow	29
3.2.2.6.3	UC-NOTIFY-03: Benachrichtigung bei Kommentar-	
	Like	30
3.2.2.6.4	UC-NOTIFY-04: Ungelesene Benachrichtigungen	
	zählen	30
3.2.2.7	Comment Like Use Cases von Umut und Yusuf	31
3.2.2.7.1	UC-LIKE-01: Kommentar liken	31
3.2.2.7.2	UC-LIKE-02: Kommentar-Like zurückziehen . .	32
3.2.2.8	Search Use Cases	33
3.2.2.8.1	UC-SEARCH-01: Benutzer suchen von Umut und	
	Yusuf	33
3.2.2.8.2	UC-SEARCH-02: Öffentliche Listen durchsuchen	33
3.2.2.9	Status Use Cases von Umut	34
3.2.2.9.1	UC-STATS-01: Beliebteste Bücher anzeigen . . .	34
3.2.2.9.2	UC-STATS-02: Meist gefolgte Listen anzeigen . .	35
4	Beschreibung der Lösung	36
4.1	Software Architektur der Lösung	36
4.1.1	Architektur des Frontends (Mobile App) von Samet	37
4.1.1.1	Komponenten der MVVM-Architektur	37
4.1.1.1.1	View	37
4.1.1.1.2	ViewModel	37
4.1.1.1.3	Model / Repository	38

4.1.1.1.4	Retrofit-basierte API-Integration	38
4.1.1.1.5	Datenfluss in der Applikation	39
4.1.1.1.6	Vorteile der Architektur	39
4.1.1.1.7	Navigation und Fragmentübergänge	39
4.1.2	Architektur des Backend (Hexagonale Architektur) von Umut, Yusuf und Taha	40
4.1.2.1	Domain	41
4.1.2.1.1	Model	42
4.1.2.1.2	Ports-Schicht (Inbound Outbound)	43
4.1.2.2	Adapters-Schicht (Inbound Outbound)	45
4.1.2.2.1	Inbound Adapter (Web Layer)	46
4.1.2.2.2	Inbound Adapter (Web Layer)	49
4.1.2.3	Application	53
4.1.2.3.1	Service (UseCases)	53
4.1.2.3.2	Config	54
4.2	Übersicht und Zusammenspiel der Komponenten	58
4.3	Beschreibung der Komponenten	60
4.3.1	Mobile App (Kotlin)	60
4.3.1.1	Gesamtstruktur der App	60
4.3.1.2	Technische Schnittstellen	60
4.3.1.3	Datenmodelle und Klassenstruktur	61
4.3.1.4	Funktionale Teilkomponenten	62
4.3.1.4.1	Authentifizierung und Einstieg	62
4.3.1.4.2	Startseite und Buchanzeigen	63
4.3.1.4.3	Suche und Filter	64
4.3.1.4.4	Buchinteraktionen	66
4.3.1.4.5	Listenverwaltung	68
4.3.1.4.6	Benachrichtigungen	69
4.3.1.4.7	Profil und soziale Funktionen	70
4.3.2	Backend App (Java)	71
4.3.2.1	User Modul von Umut	71
4.3.2.2	Auth Modul von Umut	73
4.3.2.3	Book Modul von Umut und Yusuf	74
4.3.2.4	Comment Modul von Umut und Yusuf	75
4.3.2.5	Follower Modul von Umut und Yusuf	76
4.3.2.6	List Modul von Umut und Yusuf	77
4.3.3	Gemeinsame Komponenten	78

4.4	Qualitätssicherung des Gesamtsystems von Taha	78
4.4.1	Überblick über die Testebenen in BookApp	79
4.4.2	Integrationstests am Beispiel UserServiceIntegrationTest	81
4.4.3	Persistenz-Mapping Tests: Validierung der Datenkonvertierung	81
4.4.4	Sicherheitstests mit CommentSecurityTest	82
4.4.5	Use-Case Tests am Beispiel UserUseCasesTest	82
4.4.6	Web-Layer Tests: UserWebMapperTest	82
4.4.7	Praxisbeispiel: Sicherheitsprüfung beim Benutzerlöschen	83
4.5	Übersicht Verzeichnisse und Dateien	84
4.5.1	Frontend	84
4.5.1.1	data	84
4.5.1.2	di (Dependency Injection)	84
4.5.1.3	domain	84
4.5.1.4	ui	84
4.5.1.5	MyApplication.kt	84
4.5.2	Backend	85
4.5.2.1	domain	85
4.5.2.2	adapter/in (Web Layer)	85
4.5.2.3	adapter/out (Persistence Layer)	85
4.5.2.4	application/config	86
4.5.2.5	application/service	86
4.5.2.6	BookAppBackendApplication.java	86
4.6	Bauen des Gesamtsystems	86
5	Einrichtung und Betrieb der Software-Lösung	87
5.1	Abhängigkeiten zu anderen Software-Systemen	87
5.2	Verfügbarkeit der Software	88
5.3	Installation der Software	88
5.4	Inbetriebnahme und Betrieb	88
5.5	Möglichkeiten zur späteren Anpassung und Weiterentwicklung	89
6	Zusammenfassung und Ausblick	90
6.1	UML Diagrams von Backend	91
6.2	Sequence Diagrams von Backend	96

1 Einleitung

1.1 Kontext der Arbeit

Bookies ist eine neue Art von intelligenten Anwendungen, die Bücherliebhabern bei der Entdeckung und Empfehlung von Büchern zur Seite stehen. *Bookies* bietet eine umfassende Lösung, die darauf abzielt, die Lesegewohnheiten der Nutzer zu unterstützen, ihnen geeignete Buchempfehlungen zu geben und ihnen eine einfache Verwaltung ihrer digitalen Büchersammlungen zu ermöglichen.

Die App digitalisiert herkömmliche Leselisten und ermöglicht es den Nutzern, auf einer einzigen Plattform die Bücher zu verfolgen, die ihnen gefallen, die sie gelesen haben, die sie lesen möchten oder die sie empfehlen. .

Bookies erhöht die soziale Interaktion, indem es den Nutzern erlaubt, einander zu folgen und auf ihre eigenen Listen zuzugreifen.

In diesem Dokument werden der allgemeine Ablauf des Projekts, die Positionierung des Produkts in der Branche sowie die Zielgruppen und Stakeholder erläutert.

1.2 Motivation für diese Arbeit

Die zunehmende Digitalisierung des Alltags und das Medienkonsumverhalten haben dazu geführt, dass traditionelle Buchlisten und Buchempfehlungen oft unübersichtlich und wenig personalisiert sind. Es fehlt an einer mobilen, benutzerfreundlichen und intelligenten Lösung, die das Bücherentdecken, -verwalten und -teilen auf einer modernen Plattform ermöglicht.

Mit *Bookies* soll diese Lücke geschlossen werden, indem eine Anwendung bereitgestellt wird, die gezielt auf die Bedürfnisse von Bücherliebhabern eingeht, innovative Technologien nutzt und die soziale Interaktion fördert.

1.3 Zielstellung für diese App

Das Ziel dieses Projekts ist die Spezifikation, Implementierung und Dokumentation einer robusten, skalierbaren mobilen Anwendung für Bücherliebhaber.

Bookies soll flexible Listenerstellung und soziale Funktionen einen echten Mehrwert bieten. Zudem soll die App eine klare Benutzerführung, hohe Performance und Datensicherheit gewährleisten.

Am Ende dieses Projekts steht eine Anwendung, die sowohl technisch als auch funktional überzeugt und als innovatives Beispiel für digitale Buchplattformen dienen kann.

2 Technische Grundlagen

2.1 Android-Entwicklung mit Kotlin & XML

Die Entwicklung der mobilen Anwendung erfolgt mit Kotlin als Programmiersprache. Die Benutzeroberfläche wird mithilfe von XML beschrieben. Die Nutzung von Architekturmuster wie MVVM (Model-View-ViewModel), Lifecycle-Komponenten und Recycler-Views gewährleistet eine robuste, wartbare und performante Applikation.

Die Trennung von Logik und Präsentation sowie die Nutzung moderner Android-Frameworks sorgen für eine hohe Codequalität.

2.2 Retrofit & Netzwerkarchitektur

Für die Kommunikation zwischen der mobilen App und dem Backend wird Retrofit als HTTP-Client eingesetzt. Retrofit ermöglicht eine einfache und typensichere Anbindung an REST-APIs und unterstützt dabei sowohl synchrone als auch asynchrone Anfragen.

Mit Hilfe von Interceptors werden Authentifizierung, Fehlerbehandlung und das Logging von Netzwerkaufrufen realisiert.

2.3 Java, Spring Boot & REST API

Das Backend-System basiert auf Java und dem Framework Spring Boot. Durch die Verwendung von RESTful APIs wird die Kommunikation zwischen Frontend und Backend standardisiert und modular aufgebaut.

Spring Boot bietet Funktionen wie Dependency Injection, Annotation-basierte Konfiguration sowie die Schichtung von Controller-, Service- und Repository-Layern, was die Wartbarkeit und Testbarkeit des Systems erhöht.

2.4 PostgreSQL & Datenbankmodellierung

Für die ständige Datenspeicherung wird ein relationales Datenbanksystem, in diesem Fall PostgreSQL, eingesetzt. Die Datenbankstruktur wird durch ein Entity-Relationship-Modell (ERM) abgebildet, das die Beziehungen zwischen den verschiedenen Entitäten wie Benutzer, Bücher und Listen beschreibt.

Durch die Nutzung von Constraints, Indizes und Transaktionen wird die Datenintegrität und die Performance der Anwendung sichergestellt.

3 Analyse der Anforderungen

3.1 Beschreibung des Ist-Standes

Vor Projektstart wurde deutlich, dass es für Bücherliebhaber noch keine moderne und benutzerfreundliche mobile Anwendung gibt, mit der sie ihre gelesenen oder noch zu lesenden Bücher digital verwalten können. Aktuell bestehende Lösungen wie Goodreads oder ähnliche Plattformen bieten zwar grundlegende Funktionen zur Bücherverwaltung, zeigen jedoch deutliche Defizite unter anderem in der Benutzerführung, im Design und in der Personalisierung auf.

Insbesondere fehlt es an einer Anwendung, die sich an modernen UX/UI-Standards orientiert, soziale Interaktionen fördert und gleichzeitig intelligente Empfehlungen bereitstellt. Während es für andere Medienbereiche – wie etwa Musik (Spotify) oder Filme (Letterboxd) – etablierte, interaktive Plattformen mit hohem Personalisierungsgrad gibt, existiert eine solche Lösung im Buchbereich bisher kaum.

Das gilt ganz besonders im Hinblick auf die Personalisierung und soziale Interaktion. Konkret fehlt es häufig an einer individuellen Buchempfehlung, die sich gezielt am Leseverhalten und den Interessen der Nutzer orientiert. Auch die Integration sozialer Funktionen, wie zur Vernetzung mit anderen Leserinnen und Lesern oder zum Teilen persönlicher Leselisten, ist nur begrenzt verfügbar.

Darüber hinaus bieten viele Anwendungen keine ausreichend flexible Verwaltung von Buchlisten, die es erlauben würde, Bücher thematisch zu gruppieren, sie zu kommentieren oder dauerhaft zu speichern.

Hinzu kommt, dass die mobile Nutzererfahrung oft nicht konsistent gestaltet ist. Besonders bei Smartphones weist sie Defizite hinsichtlich der Bedienbarkeit und Nutzerführung auf.

3.2 Funktionale Anforderungen

3.2.1 Mobile Seite

Die funktionalen Anforderungen auf der mobilen Seite beschreiben die zentralen Interaktionen der Benutzer mit dem System und die grundlegenden Funktionen einer solchen digitalen, buchbasierten sozialen Plattform. Im Folgenden werden die wichtigsten Funktionalitäten der mobilen Anwendung im Detail dargestellt:

3.2.1.1 Benutzeroauthentifizierung und Kontoverwaltung von Halime Die Anwendung ermöglicht es den Benutzern, ein neues Konto mittels E-Mail und Passwort zu erstellen oder sich mit einem bestehenden Konto anzumelden. Nach erfolgreicher Authen-

tifizierung wird eine Benutzersitzung initiiert, wodurch personalisierte Inhalte zugänglich gemacht werden.

3.2.1.2 Startseite (HomePage) Komponenten von Halime und Samet

Die Startseite fungiert als zentraler Zugangspunkt zur Anwendung und gliedert sich in drei funktional differenzierte Bereiche, die auf unterschiedliche Aspekte der Nutzerinteraktion ausgerichtet sind.

Im Fokus der Bücher-Sektion stehen aktuelle, besonders in der eigenen Freundesgruppe beliebte Werke, die fortlaufend aktualisiert werden. Ergänzt wird dieses Angebot durch die Rubrik „Explore More“, in der personalisierte Buchempfehlungen angezeigt werden. Beide Inhalte werden dynamisch aus der Datenbank bezogen und regelmäßig angepasst, um den Nutzerinnen und Nutzern ein aktuelles und relevantes Angebot zu bieten.

Die Rezensionen-Sektion bietet Raum für Diskussion und bildet den Kern der sozialen Interaktion, indem sie Nutzerrezensionen aggregiert und sichtbar macht. Eine algorithmisch gesteuerte Sortierung nach positiven Bewertungen und Nutzerreaktionen sorgt dafür, dass qualitativ hochwertige Beiträge sichtbar platziert werden.

Zusätzlich gibt es noch eine Listen-Sektion, in der von der Community erstellte Buchsammlungen präsentiert werden. Die Anordnung dieser Inhalte erfolgt datenbasiert und orientiert sich an Interaktionsmetriken wie Likes oder Ansichten. Dadurch wird eine kuriatierte und zugleich soziale Komponente in die Plattform integriert, die individuelle Vorlieben mit kollektiver Bewertung verknüpft.

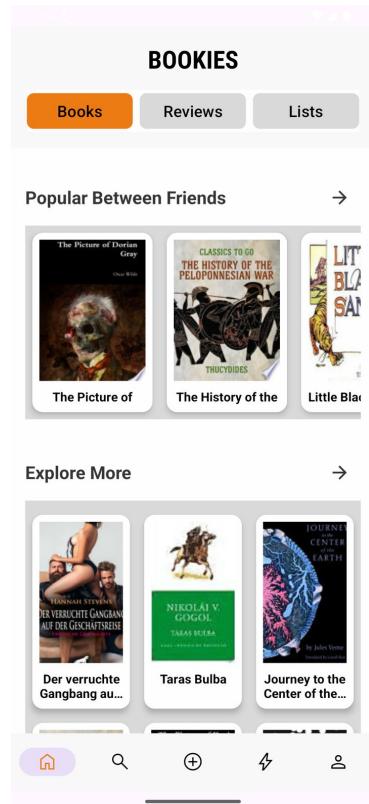


Abbildung 1: Beispiel für die Homepage

3.2.1.3 Such- und Filterfunktionen von Samet

Die Anwendung bietet eine integrierte Suchfunktion, die es ermöglicht, gezielt nach Büchern sowie nach Benutzerprofilen zu suchen. Um die Relevanz der Ergebnisse zu erhöhen, können diese anhand spezifischer Filter wie Genre (Buchkategorie) und Sprache eingegrenzt werden.

Über die Basissuche hinaus stehen erweiterte Kategorien zur Verfügung, die eine differenzierte Exploration des Angebots erlauben. Unter der Kategorie „Most Popular“ werden Bücher mit der höchsten Anzahl an Interaktionen und Likes angezeigt, während „Highly Rated“ jene Werke hervorhebt, die auf Grundlage von Nutzerbewertungen besonders positiv beurteilt wurden. Auf diese Weise unterstützt die Suchfunktion sowohl zielgerichtetes Auffinden als auch inspiratives Stöbern.

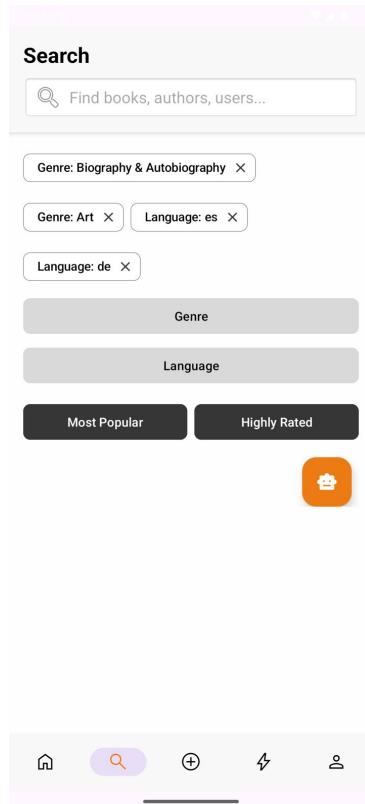


Abbildung 2: Beispiel für die Filter- und Suchfunktion

3.2.1.4 Bücherverwaltung und Interaktionen von Samet, Halime

Nutzerinnen und Nutzer haben die Möglichkeit, Bücher unmittelbar persönlichen Listen wie „Gelesen“, „Will ich lesen“ oder „Favoriten“ zuzuordnen. Darüber hinaus steht eine Interaktionsfunktionalität zur Verfügung, mit der Bücher bewertet, kommentiert und mit Likes versehen werden können.

Sämtliche Nutzeraktionen werden in Echtzeit in der Datenbank erfasst und aktualisiert, wodurch eine unmittelbare Rückmeldung im System sowie eine konsistente Datengrundlage für weitere personalisierte Funktionen gewährleistet wird.

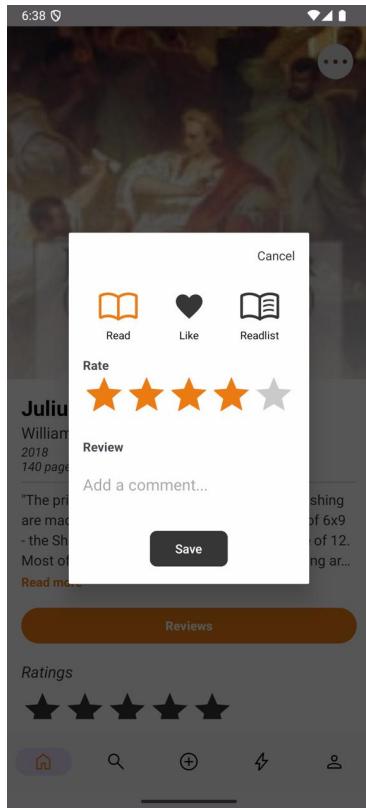


Abbildung 3: Beispiel für die BuchMorePage

3.2.1.5 Benachrichtigungssystem von Ayberk, Samet

Das integrierte Benachrichtigungssystem der Anwendung informiert Nutzerinnen und Nutzer zeitnah über relevante soziale Aktivitäten innerhalb der Plattform. Hierzu zählen unter anderem neue Follower, erhaltene Likes auf eigene Kommentare sowie Reaktionen auf selbst erstellte Buchlisten.

Die Benachrichtigungen werden durch Hintergrunddienste in Echtzeit generiert und ausgeliefert, wodurch eine kontinuierliche Einbindung in soziale Interaktionen und eine erhöhte Nutzerbindung gefördert werden.

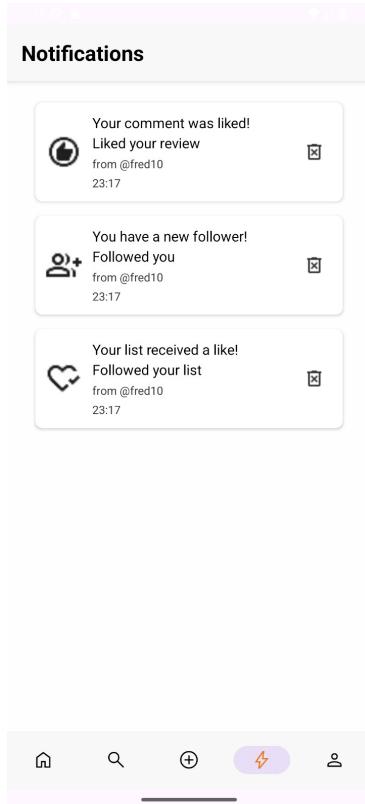


Abbildung 4: Beispiel für die Nachrichten

3.2.1.6 Profilseite und persönliche Inhalte von Ayberk, Samet

Im Profilbereich erhalten Nutzerinnen und Nutzer einen umfassenden Zugriff auf ihre persönliche Bücherwelt. Hier werden sowohl die eigene Lesehistorie als auch individuell erstellte Inhalte übersichtlich dargestellt. Dazu zählen Bücher, die bereits gelesen, für zukünftige Lektüre vorgemerkt oder als Favoriten markiert wurden.

Ebenso sind eigene, individuell zusammengestellte Buchlisten sowie fremde Listen, die als „Followed“ gekennzeichnet wurden, zentral einsehbar. Das Profil bietet damit nicht nur eine visuelle Zusammenfassung der persönlichen Aktivitäten, sondern unterstützt auch eine strukturierte und benutzerfreundliche Verwaltung der eigenen literarischen Interessen.

Zusätzlich kann der Nutzer bestimmte Avatare als Profilbild einsetzen.

3.2.2 Backend Seite

Die Backend-Seite definiert eine Reihe von Use Cases, die die zentralen Funktionen des Systems aus Sicht der Benutzer und der Geschäftslogik beschreiben. Jeder Use Case legt fest, welche Akteure beteiligt sind, welche Voraussetzungen erfüllt sein müssen und welche Ergebnisse erzielt werden. Dadurch wird sichergestellt, dass die Anforderungen klar dokumentiert und die Implementierungen in der Hexagonalen Architektur konsistent ab-

gebildet werden. Die folgenden Use Cases dienen als Grundlage für die Umsetzung und das Testen der Backend-Funktionalitäten.

3.2.2.1 Auth Use Cases von Umut

3.2.2.1.1 UC-AUTH-01: Benutzerregistrierung

Feld	Beschreibung
ID	UC-AUTH-01
Name des Use Case	Benutzerregistrierung
Beschreibung	Ein neuer Benutzer registriert sich mit Benutzername, E-Mail und Passwort, um Zugriff auf alle Plattformfunktionen zu erhalten.
Akteure	Nicht angemeldeter Benutzer (Gast)
Voraussetzungen	Der Benutzer darf noch nicht registriert sein, alle Eingaben sind gültig.
Ergebnis	Neuer Benutzer wird gespeichert und zur Anmeldeseite weitergeleitet.
Fachlicher Auslöser	Benutzer möchte ein Konto erstellen, um Inhalte zu nutzen.
Normalablauf	<ol style="list-style-type: none">1. Formular absenden (Name, E-Mail, Passwort)2. System prüft Duplikate3. Passwort verschlüsseln (BCrypt)4. Benutzer speichern5. ID und JWT-Token zurück6. Automatisches Login

Alternativabläufe

- E-Mail oder Name existiert → 409 Conflict
- Fehlerhafte Eingabe → 400 Bad Request
- Serverfehler → 500 Internal Server Error

API Endpoint POST /api/auth/register

Nicht funktionale Anforderungen Antwortzeit <2s, HTTPS, Passwortregeln (8 Zeichen, gemischt). Token 24h gültig.

3.2.2.1.2 UC-AUTH-02: Benutzer-Login

Feld	Beschreibung
ID	UC-AUTH-02
Name des Use Case	Benutzer-Login
Beschreibung	Ein registrierter Benutzer meldet sich an und erhält ein JWT-Token.
Akteure	Registrierter Benutzer
Voraussetzungen	Benutzer ist im System vorhanden, gültige Daten.
Ergebnis	Benutzer ist authentifiziert.
Fachlicher Auslöser	Benutzer möchte geschützte Funktionen nutzen.
Normalablauf	<ol style="list-style-type: none">1. Name/Passwort senden2. System prüft Daten3. JWT-Token erzeugen4. Zugriff gewähren

Alternativabläufe

- Falsche Daten → 401 Unauthorized
- Leere Eingaben → 400 Bad Request

API Endpoint POST /api/auth/login

Nicht funktionale Anforderungen Antwortzeit ≤1s, HTTPS

3.2.2.1.3 UC-AUTH-03: Benutzer-Logout

Feld	Beschreibung
ID	UC-AUTH-03

Name des Use Case	Benutzer-Logout
Beschreibung	Ein angemeldeter Benutzer beendet die Sitzung.
Akteure	Angemeldeter Benutzer
Voraussetzungen	Ein gültiges JWT-Token ist vorhanden.
Ergebnis	Benutzer ist ausgeloggt.
Fachlicher Auslöser	Benutzer möchte sich sicher abmelden.
Normalablauf	<ol style="list-style-type: none"> 1. Logout-Request senden 2. Token wird gelöscht/invalidiert

Alternativabläufe

- Keine Alternativen (idempotent)

API Endpoint	POST /api/auth/logout
Nicht funktionale Anforderungen	Antwortzeit \leq 1s, HTTPS, Token wird serverseitig verworfen

3.2.2.2 Book Use Cases von Umut und Yusuf

3.2.2.2.1 UC-BOOK-01: Bücher automatisch importieren (Google Books API)

Feld	Beschreibung
ID	UC-BOOK-01
Name des Use Case	Bücher automatisch importieren (Google Books API)
Beschreibung	Das System lädt automatisch Bücher aus der Google Books API und speichert sie in der Datenbank, wenn das System zum ersten Mal gestartet oder eine leere Datenbank erkannt wird.
Akteure	System (BookSeeder als ApplicationRunner)
Voraussetzungen	Datenbank ist leer oder Initialisierungsphase läuft. Google Books API ist erreichbar.
Ergebnis	Bis zu 1000 Bücher werden automatisch importiert und lokal verfügbar gemacht.
Fachlicher Auslöser	Beim Systemstart sollen Bücher automatisch bereitgestellt werden, ohne manuelle Eingaben von Benutzern.

Normalablauf

1. Beim Start erkennt BookSeeder, dass die Datenbank leer ist.
2. BookSeeder ruft Book ApiService auf.
3. Book ApiService lädt bis zu 1000 Bücher von der Google Books API (in Blöcken von 40).
4. Alle Bücher (Titel, ISBN, Autor, Cover, Genre, Sprache) werden konvertiert und in Domain-Modelle transformiert.
5. Bücher werden in der Datenbank gespeichert und stehen allen Abfragen zur Verfügung.

Alternativabläufe

- API nicht erreichbar → Fallback: System startet ohne Bücher, Admin-Benachrichtigung
- Teilweise Ladefehler → Nur erfolgreich geladene Bücher speichern

API Endpoint	N/A (wird automatisch durch ApplicationRunner ausgeführt)
Nicht funktionale Anforderungen	Antwortzeit für gesamten Import \leq 60 Sekunden (bei 1000 Büchern). API-Zugriffe parallelisiert. Wiederholte Importe werden blockiert, wenn DB bereits gefüllt ist.

3.2.2.2 UC-BOOK-02: Buchdetails anzeigen

Feld	Beschreibung
ID	UC-BOOK-02
Name des Use Case	Buchdetails anzeigen
Beschreibung	Ein Benutzer ruft alle verfügbaren Details zu einem Buch ab.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Buch ist öffentlich sichtbar und bereits in der Datenbank.
Ergebnis	Alle relevanten Informationen (Titel, Autor, ISBN, Cover, Genre, Sprache, Beschreibung) werden angezeigt.

Fachlicher Auslöser Benutzer möchte ein Buch bewerten oder seine Details ansehen.

Normalablauf

1. Benutzer sendet Buch-ID.
2. System lädt alle Buchinformationen aus der DB.
3. Details werden zurückgegeben.

Alternativabläufe

- Buch existiert nicht → 404 Not Found

API Endpoint GET /api/books/{id}

Nicht funktionale Anforderungen Antwortzeit ≤ 1 Sekunde

Anforderungen

3.2.2.2.3 UC-BOOK-03: Bücher suchen und filtern

Feld	Beschreibung
ID	UC-BOOK-03
Name des Use Case	Bücher suchen und filtern
Beschreibung	Ein Benutzer sucht nach Büchern anhand von Titel, Autor, Genre, Sprache und erhält paginierte Ergebnisse.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Es sind Bücher in der Datenbank vorhanden.
Ergebnis	Gefilterte und sortierte Bücher werden paginiert zurückgegeben.
Fachlicher Auslöser	Benutzer möchte gezielt Bücher finden oder durchstöbern.

Normalablauf

1. Benutzer gibt Suchparameter (keyword, genres, languages, page, size, sortBy) an.
2. System durchsucht die DB mit diesen Filtern.
3. Ergebnisse werden nach Sortierkriterien sortiert und paginiert zurückgegeben.

Alternativabläufe

- Keine Treffer → Leere Liste (200 OK)
- Ungültige Filter → 400 Bad Request

API Endpoint GET /api/books/search

Nicht funktionale Anforderungen Antwortzeit \leq 2 Sekunden. Index-basierte Suche aktiviert, Pagination standardmäßig auf 12 pro Seite.

3.2.2.3 Comment Use Cases von Umut und Yusuf

3.2.2.3.1 UC-COMMENT-01: Kommentar erstellen

Feld	Beschreibung
ID	UC-COMMENT-01
Name des Use Case	Kommentar erstellen
Beschreibung	Ein authentifizierter Benutzer kommentiert ein Buch.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Buch existiert, Kommentar gültig.
Ergebnis	Kommentar gespeichert und verknüpft.
Fachlicher Auslöser	Benutzer möchte Feedback geben.
Normalablauf	

1. Kommentartext und Buch-ID senden.
2. Validierung.
3. Speichern und Buch zuordnen.

Alternativabläufe

- Ungültiger Text → 400 Bad Request
- Buch fehlt → 404 Not Found

API Endpoint POST /api/comments

Nicht funktionale Anforderungen Antwortzeit \leq 1,5s, Moderation gegen Missbrauch.

3.2.2.3.2 UC-COMMENT-02: Kommentare anzeigen

Feld	Beschreibung
ID	UC-COMMENT-02
Name des Use Case	Kommentare anzeigen
Beschreibung	Ein Benutzer sieht alle Kommentare zu einem Buch.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Buch öffentlich sichtbar.
Ergebnis	Liste der Kommentare angezeigt.
Fachlicher Auslöser	Benutzer möchte Meinungen lesen.
Normalablauf	<ol style="list-style-type: none"> 1. Buch-ID auswählen. 2. Kommentare sortiert (Datum) laden.
Alternativabläufe	
	<ul style="list-style-type: none"> • Keine Kommentare → Leere Liste.
API Endpoint	GET /api/books/{id}/comments
Nicht funktionale Anforderungen	Antwortzeit \leq 1s, Pagination 10 pro Seite.

3.2.2.3.3 UC-COMMENT-03: Kommentar löschen

Feld	Beschreibung
ID	UC-COMMENT-03
Name des Use Case	Kommentar löschen
Beschreibung	Autor oder Admin löscht einen Kommentar.
Akteure	Autor oder Administrator
Voraussetzungen	Kommentar existiert und Berechtigung liegt vor.
Ergebnis	Kommentar entfernt oder als gelöscht markiert.
Fachlicher Auslöser	Benutzer möchte Inhalte kontrollieren.
Normalablauf	<ol style="list-style-type: none"> 1. Kommentar-ID senden. 2. Berechtigung prüfen. 3. Kommentar löschen.

Alternativabläufe

- Keine Berechtigung → 403
- Kommentar fehlt → 404

API Endpoint `DELETE /api/comments/{id}`

Nicht funktionale Antwortzeit \leq 1s, Audit-Log aktiv.

Anforderungen

3.2.2.4 Follower Use Cases von Umut und Yusuf

3.2.2.4.1 UC-FOLLOW-01: Benutzer folgen

Feld	Beschreibung
ID	UC-FOLLOW-01
Name des Use Case	Benutzer folgen
Beschreibung	Ein Benutzer folgt einem anderen Benutzer.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Zielbenutzer existiert und ist nicht der gleiche.
Ergebnis	Neue Follower-Beziehung gespeichert.
Fachlicher Auslöser	Benutzer möchte Social-Features nutzen.
Normalablauf	

1. Follow-Anfrage senden.
2. Prüfen, ob Beziehung existiert.
3. Speichern.

Alternativabläufe

- Selbst-Follow → 400
- Beziehung existiert → 409
- Ziel fehlt → 404

API Endpoint `POST /api/followers`

Nicht funktionale Antwortzeit \leq 1s, Duplikatprüfung aktiv.

Anforderungen

3.2.2.4.2 UC-FOLLOW-02: Benutzer entfolgen

Feld	Beschreibung
ID	UC-FOLLOW-02
Name des Use Case	Benutzer entfolgen
Beschreibung	Ein Benutzer beendet eine Follower-Beziehung.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Benutzer folgt Ziel bereits.
Ergebnis	Beziehung gelöscht.
Fachlicher Auslöser	Benutzer möchte Feeds anpassen.
Normalablauf	<ol style="list-style-type: none">1. Entfolgen senden.2. Beziehung prüfen.3. Löschen.

Alternativabläufe

- Beziehung fehlt → 404
- Nicht eingeloggt → 401

API Endpoint `DELETE /api/followers/{id}`

Nicht funktionale Antwortzeit \leq 1s

Anforderungen

3.2.2.4.3 UC-FOLLOW-03: Gefolgte Benutzer anzeigen

Feld	Beschreibung
ID	UC-FOLLOW-03
Name des Use Case	Gefolgte Benutzer anzeigen
Beschreibung	Ein Benutzer sieht alle Profile, denen er folgt.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Benutzer ist angemeldet.
Ergebnis	Liste der gefolgten Benutzer angezeigt.
Fachlicher Auslöser	Benutzer möchte Netzwerke pflegen.

Normalablauf

1. Profil laden.
2. Gefolgte Benutzer abrufen.

Alternativabläufe

- Benutzer fehlt → 404

API Endpoint GET /api/followers/following/{userId}

Nicht funktionale Antwortzeit \leq 1s, Pagination aktiv.

Anforderungen

3.2.2.4.4 UC-FOLLOW-04: Follower-Liste anzeigen

Feld	Beschreibung
ID	UC-FOLLOW-04
Name des Use Case	Follower-Liste anzeigen
Beschreibung	Ein Benutzer sieht alle, die ihm folgen (abhängig von Privatsphäre).
Akteure	Eingeloggter Benutzer
Voraussetzungen	Zielbenutzer erlaubt Anzeige.
Ergebnis	Liste der Follower angezeigt.
Fachlicher Auslöser	Benutzer möchte Reichweite prüfen.

Normalablauf

1. Profil aufrufen.
2. Follower abrufen.

Alternativabläufe

- Privatsphäre verweigert → 403

API Endpoint GET /api/followers/{userId}

Nicht funktionale Antwortzeit \leq 1s, Sichtbarkeitsregeln greifen.

Anforderungen

3.2.2.5 List Use Cases von Umut und Yusuf

3.2.2.5.1 UC-LIST-01: Buchliste erstellen

Feld	Beschreibung
ID	UC-LIST-01
Name des Use Case	Buchliste erstellen
Beschreibung	Ein Benutzer erstellt eine neue Buchliste.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Benutzer ist authentifiziert.
Ergebnis	Neue Liste gespeichert und verknüpft.
Fachlicher Auslöser	Benutzer möchte persönliche Sammlung anlegen.
Normalablauf	<ol style="list-style-type: none"> 1. Namen und Beschreibung senden. 2. Name validieren. 3. Liste speichern.
Alternativabläufe	
<ul style="list-style-type: none"> • Name fehlt → 400 Bad Request • Liste existiert → 409 Conflict 	
API Endpoint	POST /api/lists
Nicht funktionale	Antwortzeit \leq 2s
Anforderungen	

3.2.2.5.2 UC-LIST-02: Buch zu Liste hinzufügen

Feld	Beschreibung
ID	UC-LIST-02
Name des Use Case	Buch zu Liste hinzufügen
Beschreibung	Ein Benutzer fügt ein Buch seiner Liste hinzu.
Akteure	Listenersteller
Voraussetzungen	Liste und Buch existieren.
Ergebnis	Buch verknüpft.
Fachlicher Auslöser	Benutzer möchte Liste erweitern.

Normalablauf

1. Buch auswählen.
2. Duplikate prüfen.
3. Verknüpfen.

Alternativabläufe

- Buch bereits in Liste → 409 Conflict
- Keine Rechte → 403 Forbidden

API Endpoint POST /api/books-in-list

Nicht funktionale Antwortzeit \leq 1.5s

Anforderungen

3.2.2.5.3 UC-LIST-03: Buch aus Liste entfernen

Feld	Beschreibung
ID	UC-LIST-03
Name des Use Case	Buch aus Liste entfernen
Beschreibung	Ein Benutzer entfernt ein Buch aus seiner Liste.
Akteure	Listenersteller
Voraussetzungen	Verknüpfung existiert.
Ergebnis	Buch nicht mehr Teil der Liste.
Fachlicher Auslöser	Benutzer möchte Liste anpassen.

Normalablauf

1. Buch-ID senden.
2. Verknüpfung prüfen.
3. Entfernen.

Alternativabläufe

- Verknüpfung fehlt → 404 Not Found
- Keine Rechte → 403 Forbidden

API Endpoint DELETE /api/books-in-list/{id}

Nicht funktionale Anforderungen Antwortzeit $\leq 1.5\text{s}$

3.2.2.5.4 UC-LIST-04: Öffentliche Liste anzeigen

Feld	Beschreibung
ID	UC-LIST-04
Name des Use Case	Öffentliche Liste anzeigen
Beschreibung	Ein Benutzer sieht Inhalte einer öffentlichen Liste.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Liste ist öffentlich.
Ergebnis	Liste mit Details und Büchern angezeigt.
Fachlicher Auslöser	Benutzer möchte stöbern.
Normalablauf	<ol style="list-style-type: none">1. Liste-ID abrufen.2. Inhalte laden.

Alternativabläufe

- Privat → 403 Forbidden
- Liste fehlt → 404 Not Found

API Endpoint GET /api/lists/{id}
Nicht funktionale Anforderungen Antwortzeit $\leq 1\text{s}$

3.2.2.5.5 UC-LIST-05: Einer Liste folgen

Feld	Beschreibung
ID	UC-LIST-05
Name des Use Case	Einer Liste folgen
Beschreibung	Ein Benutzer folgt einer öffentlichen Liste.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Liste ist öffentlich und gehört nicht dem Benutzer.
Ergebnis	Beziehung gespeichert.
Fachlicher Auslöser	Benutzer möchte Listen abonnieren.

Normalablauf

1. Folgen klicken.
2. Duplikate prüfen.
3. Beziehung speichern.

Alternativabläufe

- Schon Follower → 409 Conflict
- Eigene Liste → 400 Bad Request

API Endpoint POST /api/list-follow

Nicht funktionale Antwortzeit \leq 1s

Anforderungen

3.2.2.5.6 UC-LIST-06: Liste entfolgen

Feld	Beschreibung
ID	UC-LIST-06
Name des Use Case	Liste entfolgen
Beschreibung	Ein Benutzer beendet das Folgen einer Liste.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Benutzer folgt der Liste.
Ergebnis	Beziehung entfernt.
Fachlicher Auslöser	Benutzer möchte Abos verwalten.

Normalablauf

1. Entfolgen anklicken.
2. Beziehung prüfen.
3. Löschen.

Alternativabläufe

- Beziehung fehlt → 404 Not Found

API Endpoint DELETE /api/list-follow/{id}

Nicht funktionale Antwortzeit $\leq 1\text{s}$
Anforderungen

3.2.2.5.7 UC-LIST-07: Gefolgte Listen anzeigen

Feld	Beschreibung
ID	UC-LIST-07
Name des Use Case	Gefolgte Listen anzeigen
Beschreibung	Ein Benutzer sieht alle Listen, denen er folgt.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Benutzer ist authentifiziert.
Ergebnis	Liste der abonnierten Listen angezeigt.
Fachlicher Auslöser	Benutzer möchte Abos überblicken.
Normalablauf	<ol style="list-style-type: none">1. Profil laden.2. Gefolgte Listen abrufen.

Alternativabläufe

- Keine besonderen Alternativen.

API Endpoint GET /api/list-follow/following/{userId}

Nicht funktionale Antwortzeit $\leq 1\text{s}$, Pagination aktiv.

Anforderungen

3.2.2.6 Notification Use Cases von Yusuf

3.2.2.6.1 UC-NOTIFY-01: Benachrichtigung bei neuem Follower

Feld	Beschreibung
ID	UC-NOTIFY-01
Name des Use Case	Benachrichtigung bei neuem Follower
Beschreibung	Wenn ein Benutzer von einem anderen gefolgt wird, erhält er eine Push-Benachrichtigung über FCM.
Akteure	System (FCM) & gefolgter Benutzer
Voraussetzungen	Benutzer B folgt Benutzer A.
Ergebnis	Benutzer A erhält sofort eine Push-Nachricht.

Fachlicher Auslöser Benutzer soll sofort informiert werden, wenn er einen neuen Follower bekommt.

Normalablauf

1. Benutzer B folgt Benutzer A.
2. Backend erstellt Notification-Entity.
3. FCM-Push wird an Benutzer A gesendet.

Alternativabläufe

- FCM-Server nicht erreichbar → Nachricht wird in DB gespeichert, später zugestellt.

API Endpoint Automatisch (eventbasiert)

Nicht funktionale Anforderungen Antwortzeit \leq 1 Sekunde für Push. Retry-Mechanismus bei Fehlern.

3.2.2.6.2 UC-NOTIFY-02: Benachrichtigung bei List-Follow

Feld	Beschreibung
ID	UC-NOTIFY-02
Name des Use Case	Benachrichtigung bei List-Follow
Beschreibung	Wenn eine Liste gefolgt wird, erhält der Listenersteller eine Push-Benachrichtigung.
Akteure	System (FCM) & Listenersteller
Voraussetzungen	Benutzer B folgt Liste X von Benutzer A.
Ergebnis	Listenersteller (A) wird über den neuen Follower informiert.
Fachlicher Auslöser	Listenersteller soll über Follower auf seinen Listen informiert bleiben.

Normalablauf

1. Benutzer B folgt Liste X.
2. Backend erstellt Notification.
3. Push-Benachrichtigung an Listenersteller.

Alternativabläufe

- FCM-Server nicht erreichbar → DB speichert Nachricht.

API Endpoint Automatisch (eventbasiert)

Nicht funktionale Antwortzeit \leq 1 Sekunde, Retry bei Push-Fehler.

Anforderungen

3.2.2.6.3 UC-NOTIFY-03: Benachrichtigung bei Kommentar-Like

Feld	Beschreibung
ID	UC-NOTIFY-03
Name des Use Case	Benachrichtigung bei Kommentar-Like
Beschreibung	Wenn ein Kommentar geliked wird, erhält der Kommentarautor eine Benachrichtigung.
Akteure	System (FCM) & Kommentarautor
Voraussetzungen	Benutzer B liked Kommentar von Benutzer A.
Ergebnis	Kommentarautor wird sofort informiert.
Fachlicher Auslöser	Autor soll erfahren, wenn sein Kommentar geliked wird.

Normalablauf

1. Benutzer B liked Kommentar.
2. Notification-Entity wird erstellt.
3. Push-Benachrichtigung an Autor.

Alternativabläufe

- Kommentar gelöscht → Kein Notification-Eintrag.

API Endpoint Automatisch (eventbasiert)

Nicht funktionale Antwortzeit \leq 1 Sekunde, Retry bei Fehler.

Anforderungen

3.2.2.6.4 UC-NOTIFY-04: Ungelesene Benachrichtigungen zählen

Feld	Beschreibung
ID	UC-NOTIFY-04
Name des Use Case	Ungelesene Benachrichtigungen zählen

Beschreibung	Ein Benutzer ruft die Anzahl seiner ungelesenen Benachrichtigungen ab.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Benutzer ist authentifiziert.
Ergebnis	Backend gibt die aktuelle Anzahl zurück.
Fachlicher Auslöser	Benutzer möchte in der App sofort sehen, wie viele Benachrichtigungen offen sind.
Normalablauf	<ol style="list-style-type: none"> 1. Benutzer ruft Counter-Endpunkt auf. 2. Backend zählt ungelesene Notifications. 3. Anzahl wird zurückgegeben.

Alternativabläufe

- Keine Benachrichtigungen → 0 wird zurückgegeben.

API Endpoint	GET /api/notifications/count/unread
Nicht funktionale Anforderungen	Antwortzeit \leq 0,5s, Count wird gecached und inkrementell aktualisiert.

3.2.2.7 Comment Like Use Cases von Umut und Yusuf

3.2.2.7.1 UC-LIKE-01: Kommentar liken

Feld	Beschreibung
ID	UC-LIKE-01
Name des Use Case	Kommentar liken
Beschreibung	Ein Benutzer markiert einen Kommentar als 'Gefällt mir'.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Kommentar existiert und Benutzer ist nicht der Autor.
Ergebnis	Kommentar-Like wird gespeichert, Autor erhält ggf. Notification.
Fachlicher Auslöser	Benutzer möchte Zustimmung zu einem Kommentar zeigen.

Normalablauf

1. Benutzer klickt 'Like'.
2. Backend speichert Like (LikedComment-Entity).
3. Autor wird benachrichtigt.

Alternativabläufe

- Benutzer ist Autor des Kommentars → Like blockiert.
- Kommentar gelöscht → 404.

API Endpoint POST /api/comments/id/like

Nicht funktionale Anforderungen Antwortzeit \leq 1 Sekunde, Duplikatprüfung (nur 1 Like pro Benutzer).

3.2.2.7.2 UC-LIKE-02: Kommentar-Like zurückziehen

Feld	Beschreibung
ID	UC-LIKE-02
Name des Use Case	Kommentar-Like zurückziehen
Beschreibung	Ein Benutzer entfernt sein Like von einem Kommentar.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Like existiert für diesen Benutzer und Kommentar.
Ergebnis	Like wird entfernt und Like-Zähler aktualisiert.
Fachlicher Auslöser	Benutzer möchte Like rückgängig machen.

Normalablauf

1. Benutzer klickt 'Unlike'.
2. Backend löscht Like-Entity.
3. Kommentar wird aktualisiert.

Alternativabläufe

- Like nicht vorhanden → 404.
- Kommentar gelöscht → 404.

API Endpoint DELETE /api/comments/id/like

Nicht funktionale Anforderungen Antwortzeit \leq 1 Sekunde

3.2.2.8 Search Use Cases

3.2.2.8.1 UC-SEARCH-01: Benutzer suchen von Umut und Yusuf

Feld	Beschreibung
ID	UC-SEARCH-01
Name des Use Case	Benutzer suchen
Beschreibung	Ein Benutzer sucht nach anderen Benutzern anhand von Namen oder Username.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Benutzer ist authentifiziert.
Ergebnis	Gefilterte Benutzerliste wird zurückgegeben.
Fachlicher Auslöser	Benutzer möchte anderen Benutzern folgen oder deren Profile finden.

Normalablauf

1. Suchparameter (Keyword) senden.
2. DB nach Teiltreffern durchsuchen.
3. Ergebnisse paginieren.

Alternativabläufe

- Keine Treffer → Leere Liste (200 OK).

API Endpoint GET /api/search/users

Nicht funktionale Anforderungen Antwortzeit \leq 1.5 Sekunden, Suchindex optimiert.

Anforderungen

3.2.2.8.2 UC-SEARCH-02: Öffentliche Listen durchsuchen

Feld	Beschreibung
ID	UC-SEARCH-02
Name des Use Case	Öffentliche Listen durchsuchen

Beschreibung	Ein Benutzer durchsucht öffentliche Listen nach Name oder Ersteller.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Nur öffentliche Listen werden durchsucht.
Ergebnis	Gefilterte, paginierte Listen zurückgegeben.
Fachlicher Auslöser	Benutzer möchte interessante Listen entdecken.
Normalablauf	<ol style="list-style-type: none"> 1. Keyword eingeben. 2. System durchsucht nur öffentliche Listen. 3. Ergebnisse paginiert zurück.

Alternativabläufe

- Keine Treffer → Leere Liste.

API Endpoint	GET /api/search/lists
Nicht funktionale Anforderungen	Antwortzeit \leq 1.5 Sekunden, Indexe auf Name und Ersteller.
<hr/>	

3.2.2.9 Status Use Cases von Umut

3.2.2.9.1 UC-STATS-01: Beliebteste Bücher anzeigen

Feld	Beschreibung
ID	UC-STATS-01
Name des Use Case	Beliebteste Bücher anzeigen
Beschreibung	Das System zeigt die beliebtesten Bücher basierend auf Follower und Statusinformationen.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Daten in der DB verfügbar.
Ergebnis	Top 12 Bücher werden zurückgegeben.
Fachlicher Auslöser	Benutzer möchte Trends sehen.

Normalablauf

1. Aggregationsabfrage (COUNT auf BooksStatus).
2. Sortierung DESC.
3. Ergebnisse zurück.

Alternativabläufe

- Keine Daten → Leere Liste.

API Endpoint	GET /api/stats/popular-books
Nicht funktionale Anforderungen	Antwortzeit \leq 2 Sekunden, Abfrage optimiert (JOINS, GROUP BY).

3.2.2.9.2 UC-STATS-02: Meist gefolgte Listen anzeigen

Feld	Beschreibung
ID	UC-STATS-02
Name des Use Case	Meist gefolgte Listen anzeigen
Beschreibung	Das System zeigt Listen mit den meisten Followern.
Akteure	Eingeloggter Benutzer
Voraussetzungen	Follower-Daten vorhanden.
Ergebnis	Top-Listen nach Follower-Anzahl angezeigt.
Fachlicher Auslöser	Benutzer möchte beliebte Listen entdecken.
Normalablauf	

1. Aggregationsabfrage (COUNT auf ListFollow).
2. Sortieren und zurückgeben.

Alternativabläufe

- Keine Listen → Leere Liste.

API Endpoint	GET /api/stats/popular-lists
Nicht funktionale Anforderungen	Antwortzeit \leq 2 Sekunden, Indexe auf ListFollow.

4 Beschreibung der Lösung

Die entwickelte Lösung basiert auf einer klar getrennten Client-Server-Architektur, in der die mobile Anwendung (Frontend) in Kotlin für Android entwickelt wurde und mit einem in Java implementierten Backend über REST-Schnittstellen kommuniziert.

Das Backend wurde mithilfe des Spring-Boot-Frameworks umgesetzt und läuft als eigenständiger Dienst. Beide Systeme wurden unabhängig voneinander, jedoch im engen Teamzusammenhang konzipiert und implementiert, wodurch eine kohärente Gesamtlösung entstanden ist. In diesem Kapitel werden die Softwarearchitektur sowie die einzelnen Systemkomponenten und deren Zusammenspiel detailliert beschrieben.

Der Fokus liegt dabei auf der technischen Umsetzung, der internen Struktur der Komponenten sowie dem Datenfluss zwischen Frontend und Backend.

4.1 Software Architektur der Lösung

Die Architektur des Bookies-Systems folgt einem modularen Ansatz mit klar abgegrenzten Verantwortlichkeiten und zeichnet sich durch eine strukturierte Trennung von Frontend und Backend aus. Diese Aufteilung gewährleistet eine saubere Softwarearchitektur, fördert die Wartbarkeit und unterstützt eine skalierbare Weiterentwicklung.

Das Gesamtsystem gliedert sich in zwei zentrale Komponenten: die mobile Anwendung (Frontend) und das Backend-System. Beide Teile sind logisch und technologisch voneinander entkoppelt, kommunizieren aber über eine standardisierte REST-basierte Schnittstelle. Diese API definiert eindeutig, welche Datenformate und Endpunkte zwischen den Systemteilen ausgetauscht werden, wodurch eine lose Kopplung und eine parallele Entwicklung ermöglicht werden.

Die mobile Anwendung wurde unter Verwendung des MVVM-Architekturmusters in Kotlin entwickelt. Sie bildet die Benutzeroberfläche ab und implementiert sämtliche clientseitigen Logiken, darunter Datenanzeige, Nutzerinteraktion und Zustandsverwaltung. Durch die klare Trennung zwischen View, ViewModel und Model werden Wiederverwendbarkeit und Testbarkeit auf UI-Ebene signifikant erhöht.

Das Backend-System basiert auf dem Spring-Boot-Framework und wurde in Java realisiert. Es verantwortet die zentrale Geschäftslogik, verarbeitet Nutzereingaben und stellt die dauerhafte Datenspeicherung über eine relationale PostgreSQL-Datenbank sicher. Innerhalb des Backends sorgen definierte Schichten wie Controller, Service und Repository für eine kohärente Datenflussstruktur und erleichtern die Erweiterung um zusätzliche Funktionalitäten.

Diese zweigeteilte Architektur bildet die Grundlage für ein robustes, erweiterbares Systemdesign und ermöglicht es, beide Komponenten unabhängig voneinander zu skalieren

oder zu aktualisieren. In den folgenden Abschnitten werden Aufbau und Funktionalität der mobilen Anwendung sowie des Backends detailliert dargestellt.

4.1.1 Architektur des Frontends (Mobile App) von Samet

Die mobile Anwendung wurde unter Verwendung des MVVM-Architekturmusters (Model-View-ViewModel) entwickelt, das sich als bewährter Standard für moderne Android-Applikationen etabliert hat. Diese Architektur gewährleistet eine klare Trennung der Verantwortlichkeiten zwischen Benutzeroberfläche, Anwendungslogik und Datenzugriff. Das ermöglicht eine hohe Wartbarkeit, Erweiterbarkeit und Testbarkeit des Systems.

4.1.1.1 Komponenten der MVVM-Architektur

4.1.1.1.1 View

Die View-Schicht der mobilen Anwendung bildet die visuelle und interaktive Ebene des Systems ab und besteht aus einer Vielzahl XML-basierter Layout-Dateien sowie den zugehörigen *Activity*- und *Fragment*-Klassen. Sie ist verantwortlich für die Darstellung der Inhalte, die Erfassung von Benutzereingaben und die Weiterleitung von Interaktionen an die darunterliegende Logik.

Zu den zentralen Layout-Komponenten zählen unter anderem Dateien wie `activity_sign_in.xml`, `fragment_home_page.xml` und `fragment_search.xml`, die jeweils spezifische UI-Bereiche abbilden. Diese Layouts sind direkt mit den entsprechenden Fragment-Klassen wie `HomePageFragment`, `SearchFragment` oder `ProfileFragment` verbunden.

Ergänzt wird die View-Schicht durch interaktive UI-Elemente wie *BottomSheets* (z. B. `AddBookBottomSheet`) und Dialoge (`BookInfoMoreDialog`), die für kontextabhängige Funktionen und Nutzeraktionen verwendet werden.

Die Benutzerinteraktionen – etwa Klickereignisse, Texteingaben oder Scrollbewegungen – werden mithilfe von *View Binding* unmittelbar an die zuständigen *ViewModels* weitergeleitet. Dadurch ist eine klare Trennung zwischen Darstellung und Logik gewährleistet, was nicht nur die Lesbarkeit und Wartbarkeit des Codes verbessert, sondern auch die Testbarkeit der Anwendungslogik unterstützt.

4.1.1.1.2 ViewModel

Die *ViewModels* übernehmen innerhalb der MVVM-Architektur die Rolle der zentralen Bindeschicht zwischen der *View* und der Daten- bzw. Logikebene. Sie kapseln den

UI-Zustand, steuern die Geschäftslogik für Benutzerinteraktionen und koordinieren die Kommunikation mit den *Repositories*.

Ein wesentlicher Bestandteil der *ViewModels* ist die Verwaltung des UI-Status in Form zustandstragender Klassen wie `SearchUiState` oder `RegisterState`. Diese Zustandsobjekte ermöglichen eine reaktive und konsistente Darstellung der Benutzeroberfläche auf Grundlage aktueller Daten und Interaktionen.

Darüber hinaus enthalten die *ViewModels* die zentrale Anwendungslogik zur Verarbeitung von Nutzeraktionen – etwa das Auslösen von Suchvorgängen, das Validieren von Eingaben oder das Aktualisieren von Profilinformationen. Sie fungieren dabei als Vermittlungsinstanz, die eingehende Ereignisse verarbeitet und entsprechende Datenoperationen über die *Repositories* anstößt.

Zu den implementierten *ViewModels* zählen unter anderem `SearchViewModel`, `ListsViewModel`, `SignInViewModel` und `UserViewModel`. Zur Unterstützung der modularen und testbaren Instanziierung dieser Klassen kommen spezialisierte *Factory*-Klassen wie `SearchViewModelFactory` zum Einsatz. Diese ermöglichen eine flexible Bereitstellung von Abhängigkeiten, insbesondere im Kontext von *Dependency Injection* und *Lifecycle Management*.

4.1.1.3 Model / Repository

In *Bookies* erfolgt der gesamte Datenzugriff über eine klar definierte *Repository*-Schicht, die als Vermittlungsebene zwischen den *ViewModels* und den zugrunde liegenden Datenquellen dient. Jedes Repository operiert auf Basis spezialisierter Datenübertragungsobjekte (*DTOs*) aus dem Paket `data.remote.dto`, wie etwa `BookDto` oder `ReviewDto`, sowie auf Anfrage- und Antwortstrukturen aus dem Paket `model`, etwa `SignInRequest` oder `RegisterRequest`.

Zur Entkopplung von externen Datenformaten und interner Logik werden *Mapper*-Klassen wie `BookMapper` oder `UserMapper` eingesetzt, die eine Umwandlung in domänen-spezifische Modelle vornehmen.

Die Hauptverantwortung liegt bei Repositories wie `BookRepository`, `AuthRepository` oder `ReviewsRepository`, die so eine modulare, testbare und wartbare Datenverarbeitung sicherstellen.

4.1.1.4 Retrofit-basierte API-Integration

Die Netzwerkkommunikation der App basiert auf *Retrofit*, wobei für jeden funktionalen Anwendungsbereich eigene API-Interfaces definiert sind, etwa `Books ApiService`,

`User ApiService` oder `Notification ApiService`.

Die zentrale Konfiguration erfolgt über die Klasse `RetrofitClient.kt`. Diese bindet einen *Interceptor* zur JWT-basierten Authentifizierung ein, ergänzt standardisierte Header wie `Content-Type` und ermöglicht durch den `HttpLoggingInterceptor` eine nachvollziehbare Protokollierung der HTTP-Anfragen während der Entwicklung.

4.1.1.5 Datenfluss in der Applikation

Der Ablauf einer Nutzerinteraktion folgt einem klar strukturierten Schema: Die *View* (z. B. `SearchFragment`) initiiert eine Aktion durch einen Aufruf im zugehörigen *ViewModel*. Dieses leitet die Anfrage an das entsprechende *Repository* weiter (etwa `searchRepository.search(query)`), das über *Retrofit* eine Netzwerkanfrage stellt (z. B. über `Search ApiService.kt`).

Die Antwort (z. B. `CombinedSearchResponse`) wird anschließend über *LiveData* oder *StateFlow* an die View zurückgegeben. Dieses Muster ermöglicht sowohl synchrone Rückmeldungen als auch reaktive UI-Updates, wie sie etwa in `HomePageFragment`, `AddBookBottomSheet` oder `SearchFragment` umgesetzt sind.

4.1.1.6 Vorteile der Architektur

Die gewählte Architektur fordert eine lose Kopplung, wodurch Änderungen an der Benutzeroberfläche keine Anpassungen an Geschäftslogik oder Datenstruktur erfordern. Gleichzeitig gewährleistet sie eine hohe Testbarkeit, da *ViewModel*- und *Repository*-Schichten unabhängig voneinander überprüft werden können. Die Wiederverwendbarkeit von *ViewModel*-Logik über verschiedene UI-Komponenten hinweg erhöht zudem die Effizienz bei der Entwicklung. Dank der modularen Struktur ist das System gut skalierbar und kann unkompliziert um neue Funktionen oder Services erweitert werden. Insgesamt schafft diese klar definierte Architektur eine belastbare Grundlage für kollaborative Entwicklung, langfristige Wartbarkeit und nachhaltige Erweiterbarkeit der mobilen Anwendung.

4.1.1.7 Navigation und Fragmentübergänge

Zur Steuerung der Benutzerführung setzt *Bookies* auf die Android *Jetpack Navigation*-Komponente. Die gesamte Navigationslogik ist deklarativ in der Datei `nav_graph.xml` hinterlegt, wodurch *Fragment*-Übergänge und Argumentweitergaben zentral verwaltet und typsicher umgesetzt werden. Grundlage der Navigation ist ein `NavController`, der

in den jeweiligen Fragment-Klassen initialisiert wird. Übergänge erfolgen über im Navigationsgraph definierten `action`-Elemente mit typisierten Parametern wie `bookId`, `userId` oder `listId`.

Die Navigation ist modular strukturiert und spiegelt die funktionale Gliederung der App wider. Nach dem Start über die `SplashActivity` erfolgt der Einstieg entweder über die `FirstPageActivity` oder direkt die `MainActivity`, abhängig vom Authentifizierungsstatus. Die `MainActivity` enthält eine *Bottom Navigation* zur Ansteuerung von `HomePageFragment`, `SearchFragment`, `ProfileFragment` und `NotificationsFragment`.

Die Detailnavigation umfasst unter anderem folgende Übergänge:

- `HomePageFragment` → `BookInfoPageFragment` → `ReviewsFragment` bzw. `ListsFragment`
- `SearchFragment` → `BookInfoPageFragment`, `ProfileFragment`, `OtherUserProfileFragment`
- `ProfileFragment` und `OtherUserProfileFragment` → `ListsFragment`, `UserListFragment`
- `ListsFragment` → `ThreeColumnFragment`

Zusätzliche Übergänge erfolgen über UI-Komponenten wie `AddBookBottomSheet` oder `BookInfoMoreDialog`, die programmatisch oder als `dialog`-Elemente in das Navigationssystem eingebunden sind.

Besonders hervorzuheben ist der Einsatz von *Safe Args*, der eine typsichere Übergabe von Argumenten ermöglicht und potenzielle Laufzeitfehler reduziert. Die gewählte Navigationsarchitektur erhöht die Konsistenz und Nachvollziehbarkeit der App-Flows und trägt entscheidend zur Wartbarkeit und Skalierbarkeit der Anwendung bei.

4.1.2 Architektur des Backend (Hexagonale Architektur) von Umut, Yusuf und Taha

Das Backend des Systems ist vollständig nach dem Prinzip der *Hexagonalen Architektur* (Ports und Adapter) aufgebaut – einem modernen Ansatz, der eine strikte Trennung zwischen der Geschäftslogik und allen technischen Implementierungen sicherstellt. Im Mittelpunkt dieser Architektur steht die *Domain*-Schicht, welche die zentralen Entitäten und Regeln des Systems beinhaltet und frei von Abhängigkeiten gegenüber Frameworks wie *Spring* oder *JPA* bleibt. Dadurch wird gewährleistet, dass die Kernlogik unabhängig getestet, leicht erweitert und in unterschiedlichen Kontexten wiederverwendet werden kann, ohne dass Änderungen in der Infrastruktur Einfluss auf die zentrale Logik haben.

Die hexagonale Struktur verfolgt das Ziel, ein hochgradig modulares und zukunftsicheres System zu schaffen. Jede Interaktion zwischen der Domain und der Außenwelt erfolgt ausschließlich über klar definierte Schnittstellen, die sogenannten *Ports*. **Inbound Ports** stellen die Services dar, die die Geschäftslogik der Domain für externe Anfragen

zugänglich machen, während **Outbound Ports** als Abstraktionsschicht für Datenbankzugriffe oder externe Integrationen dienen. Diese Architektur sorgt dafür, dass Anpassungen – etwa der Austausch einer Datenbank oder die Integration eines neuen Dienstes – ohne tiefgreifende Änderungen an der Geschäftslogik möglich sind.

Die Kommunikation und Abhängigkeitsrichtung innerhalb dieser Struktur folgt einem eindeutigen Prinzip: Alle Abhängigkeiten verlaufen ausschließlich von außen nach innen. Eingehende Anfragen aus der Außenwelt treffen zunächst auf **Controller**, die als *Inbound Adapter* agieren und über definierte Service-Schnittstellen (*Inbound Ports*) mit der Domain kommunizieren. Die Domain selbst bleibt von technischen Details unberührt und nutzt lediglich *Repository*-Schnittstellen (*Outbound Ports*), um bei Bedarf auf Datenbanken oder externe Systeme zuzugreifen. Da sämtliche Implementierungen und Integrationen in den *Adapter*-Schichten gekapselt sind, bleibt die Kernlogik klar isoliert und vor technologischen Abhängigkeiten geschützt. **Typischer Datenfluss:**

1. Request vom Client (z. B. Mobile App) erreicht einen **Controller** (*Inbound Adapter*).
2. Der Controller übernimmt die Validierung und Transformation der eingehenden Daten, indem die *DTOs* in Domain-Objekte umgewandelt werden.
3. Die Anfrage wird an einen **Service** (*Inbound Port*) weitergeleitet, der die Geschäftslogik verarbeitet.
4. Bei Bedarf ruft der Service **Repositories** (*Outbound Ports*) auf, um Daten aus der Datenbank zu lesen oder externe Systeme (z. B. FCM) anzusprechen.
5. Nach Abschluss der Logik durchläuft die Antwort denselben Weg zurück (Domain → DTO → Controller) und wird an den Client zurückgesendet.

Dank dieser klaren Struktur ist das System nicht nur robust und gut wartbar, sondern auch flexibel und erweiterbar. Neue Module, zusätzliche Schnittstellen oder externe Integrationen lassen sich problemlos ergänzen, ohne dass die Kernlogik des Systems angeastet werden muss. Dieses Architekturprinzip bildet somit die Grundlage für ein stabiles, testbares und langfristig skalierbares Backend.

4.1.2.1 Domain

Die **Domain-Schicht** bildet das Herzstück der Anwendung. Sie enthält die wesentlichen Geschäftsregeln und Modelle (Entities, Value Objects) und stellt sicher, dass diese unabhängig von technischen Details wie Frameworks, Datenbanken oder externen Schnittstellen bleiben.

4.1.2.1.1 Model

Das *Model*-Paket innerhalb der *Domain*-Schicht bildet das fachliche Rückgrat der Anwendung. Hier werden sämtliche Entitäten, *Value Objects* und unterstützende Strukturen definiert, die die Geschäftslogik der Anwendung abbilden. Dieses Paket ist bewusst **frameworkunabhängig** gestaltet und dient ausschließlich der Modellierung fachlicher Konzepte, ohne technische Abhängigkeiten zu *Spring*, *JPA* oder externen Bibliotheken.

Die Struktur des `model`-Pakets ist in mehrere Unterpakete gegliedert, um die unterschiedlichen Domänenaspekte sauber voneinander zu trennen:

- **base**: Enthält die abstrakte Basisklasse `BaseModel`, die allen Entitäten einheitliche Attribute wie `id`, `createdAt` und `updatedAt` zur Verfügung stellt. Zudem ist hier das *Value Object* `NonNegativeInteger` definiert, welches ausschließlich nicht-negative Werte zulässt. Methoden wie `increment()` und `decrement()` stellen sicher, dass der Wert nie unter null fällt.
- **book**: Beinhaltet zentrale Entitäten wie `Book`, `Author`, `AuthorBook` sowie `Comment`. Auch die Konstante `StatusConstants` zur Definition fester Statuswerte ist hier enthalten.
- **follower**: Modelliert Entitäten wie `Follower` und `LikedBook`, die Benutzerinteraktionen und soziale Beziehungen abbilden.
- **id**: Enthält mehrere *Embedded Keys* wie `AuthorBookId`, `BooksInListId`, `ListFollowId` etc., welche die Modellierung von *Composite Keys* ermöglichen. Diese Schlüssel sind gekapselt, um Wiederverwendbarkeit und Trennung sicherzustellen.
- **likedcomment**: Beinhaltet die Entität `LikedComment`, die „*Gefällt mir*“-Angaben bei Kommentaren abbildet.
- **list & listfollow**: Enthalten Entitäten wie `BooksInList`, `List` und `ListFollow`, um die Organisation und Nachverfolgung von Buchlisten zu verwalten.
- **notification**: Enthält die Entität `Notification` sowie die Enumeration `NotificationType`. Letztere definiert typisierte Benachrichtigungsarten wie Benutzer-Follows, Listen-Follows oder Kommentar-Likes.
- **status**: Beinhaltet Klassen wie `BooksStatus` und `Status`, welche den Zustand und Status von Büchern modellieren.
- **user**: Umfasst die Entitäten `User`, `UserAvatar`, `Avatar` sowie die *Value Objects* `Email` und `Password`. Beide *Value Objects* sind *immutable* und führen bereits bei

Instanziierung Validierungen durch, um Sicherheit und Datenintegrität zu gewährleisten.

Durch diese modulare Struktur und die konsequente Verwendung von *Value Objects*, *Embedded Keys* und *Enumerationen* wird das `model`-Paket zu einem robusten, klar strukturierten Fundament. Es bildet sämtliche Geschäftsregeln präzise ab, ohne von technischen Details beeinflusst zu werden. Damit bleibt die Geschäftslogik **isoliert, testbar und langfristig erweiterbar**.

4.1.2.1.2 Ports-Schicht (Inbound Outbound)

Die *Ports*-Schicht bildet die Schnittstelle zwischen der *Domain* und den äußeren Adapterschichten. Sie sorgt dafür, dass die Geschäftslogik nach außen zugänglich ist und gleichzeitig die Domain kontrolliert mit Datenbanken oder externen Diensten kommuniziert. Dank der konsequenten Nutzung von **Interfaces** bleibt die Domain vollständig von technischen Details isoliert, wodurch sie leicht testbar, erweiterbar und unabhängig von Frameworks bleibt.

Die *Ports*-Schicht ist in zwei Hauptbereiche gegliedert:

- **Inbound Ports:** Die *Inbound Ports* stellen alle Schnittstellen bereit, über die externe Komponenten wie Controller auf die Geschäftslogik zugreifen können. Diese Ports definieren die öffentlich zugänglichen Operationen der Domain und kapseln die zugrunde liegenden Anwendungslogiken.

Jedes Service-Interface folgt einer einheitlichen Struktur, indem es das generische `IBaseService<T, ID>` Interface erweitert, das grundlegende CRUD-Funktionalitäten (Create, Read, Update, Delete) bereitstellt:

```
public interface IBaseService<T, ID> {  
    List<T> getAll();  
    Optional<T> getById(ID id);  
    T create(T object);  
    void delete(ID id);  
    T update(ID id, T object);  
}
```

Dieses Basiskonzept ermöglicht eine einheitliche Handhabung aller Entitäten innerhalb der Domain. Konkrete Services wie `IBookService` erweitern dieses Interface, um den spezifischen Datentyp und den Primärschlüsseltyp zu definieren und bei Bedarf zusätzliche Methoden hinzuzufügen:

```

public interface IBookService extends IBaseService<Book, Long> {

    Page<Book> searchBook(String keyword,
                          List<String> genres,
                          List<String> languages,
                          int page,
                          int size);

    // zusätzliche Abfragen oder Buch-spezifische Operationen
}

```

Dank dieser Struktur wird eine klare Wiederverwendbarkeit und Konsistenz innerhalb der gesamten Service-Landschaft sichergestellt, während individuelle Services dennoch flexibel um entitätsspezifische Logik ergänzt werden können.

- **Outbound Ports:** Die *Outbound Ports* stellen die Schnittstellen dar, über die die Domain mit der Persistenzschicht oder externen Systemen interagiert. Sie kapseln sämtliche Datenzugriffsoperationen und abstrahieren die Details der Datenhaltung, wodurch die Domain unabhängig von der zugrunde liegenden Infrastruktur bleibt.

Auch hier wird ein generisches Interface genutzt – das `IBaseRepository<T, ID>` –, das die grundlegenden CRUD-Operationen zur Verfügung stellt:

```

public interface IBaseRepository<T, ID> {
    List<T> findAll();
    Optional<T> findById(ID id);
    T save(T entity);
    void deleteById(ID id);
    T update(T dto, ID id);
}

```

Konkrete Repositories wie `IBookRepository` erweitern dieses Interface, um die Entität und den Schlüsseltyp zu spezifizieren. Sie können zusätzlich eigene Query-Methoden definieren, ohne die wiederverwendbare Basisstruktur zu verändern:

```

public interface IBookRepository extends IBaseRepository<Book, Long> {

    Page<Book> searchBook(String keyword,
                          List<String> genres,

```

```

        List<String> languages,
        int page,
        int size);

    // zusätzliche Abfragen oder Buch-spezifische Operationen
}

```

Durch dieses Muster entsteht eine klare und konsistente Architektur, in der alle Services und Repositories ein gemeinsames Fundament teilen. Gleichzeitig bleibt die Flexibilität erhalten, individuelle Anforderungen pro Entität umzusetzen, ohne die generische Struktur zu beeinträchtigen.

4.1.2.2 Adapters-Schicht (Inbound Outbound)

Die Adapters-Schicht stellt die zentrale Verbindung zwischen der Domain und allen externen Systemen und Nutzern dar. Sie fungiert als vermittelnde Ebene, die sicherstellt, dass sämtliche Interaktionen von außen – sei es durch Frontend-Anwendungen, mobile Clients oder externe Dienste – in geordneten Bahnen ablaufen und die Geschäftslogik der Domain erreichen, ohne diese direkt oder unkontrolliert zu berühren. Durch die klare Trennung dieser Schicht bleibt die Domain vollständig frei von technischen Details, während externe Anfragen und Rückmeldungen nach festen Regeln verarbeitet, validiert und in ein standardisiertes Format gebracht werden.

Diese Schicht ist in zwei funktional unterschiedliche Bereiche unterteilt: **Inbound Adapter**, die dafür verantwortlich sind, eingehende Daten von Clients entgegenzunehmen, zu validieren, an die entsprechenden Inbound Ports (Service-Interfaces) weiterzuleiten und die Ergebnisse der Domain in aufbereiteter Form, oft angereichert mit zusätzlichen Informationen wie HATEOAS-Links, an die Clients zurückzugeben. **Outbound Adapter**, die den Zugriff der Domain auf Datenbanken und externe Systeme ermöglichen, indem sie die in der Domain definierten Outbound Ports implementieren und dabei sicherstellen, dass die Geschäftslogik unabhängig von der eingesetzten Persistenztechnologie oder Integrationslösung bleibt.

Durch diese Struktur fungiert die Adapters-Schicht als Schutz- und Vermittlungsbarriere: Sie kapselt alle technischen Implementierungsdetails und stellt sicher, dass die Kommunikation zwischen Domain und Außenwelt konsistent, sicher und erweiterbar bleibt. So können Änderungen an der Infrastruktur oder neuen externen Schnittstellen umgesetzt werden, ohne die Kernlogik der Anwendung anzupassen.

4.1.2.2.1 Inbound Adapter (Web Layer)

Der Inbound-Adapter innerhalb der Adapters-Schicht ist dafür zuständig, alle eingehenden Anfragen von externen Clients wie Web-Frontends, mobilen Anwendungen oder Drittanbieterdiensten aufzunehmen und in einer kontrollierten Form an die Domain weiterzuleiten. Diese Ebene ist die erste Kontaktstelle für alle HTTP-Anfragen und sorgt dafür, dass Daten korrekt entgegengenommen, validiert und in die Geschäftslogik der Anwendung integriert werden. Gleichzeitig stellt sie sicher, dass die Domain vollständig von technischen Details wie HTTP-Routing, Validierung oder Authentifizierung entkoppelt bleibt.

Ein wesentlicher Bestandteil dieser Schicht sind die Controller-Klassen, die als REST-Endpoints agieren und die primäre Kontaktstelle für alle eingehenden HTTP-Anfragen darstellen. Diese Controller sind mit `@RestController` annotiert, um ihre Rolle als Webschnittstelle zu kennzeichnen, und sie nutzen `@RequestMapping` sowie spezifische HTTP-Mappings wie `@GetMapping`, `@PostMapping`, `@PutMapping` und `@DeleteMapping`, um die Pfade und die Art der Anfrage (Lesen, Erstellen, Aktualisieren, Löschen) eindeutig zu definieren. Jede Anfrage wird dadurch klar klassifiziert und kann gezielt an die passenden Verarbeitungslogiken weitergeleitet werden.

Neben der Definition der Routen übernehmen die Controller auch die API-Dokumentation und Testbarkeit. Mithilfe von Swagger (OpenAPI) werden alle Controller und ihre Endpunkte automatisch dokumentiert und für Entwickler, Tester sowie externe Integrationspartner transparent gemacht. Swagger wurde bewusst gewählt, weil es eine Reihe von Vorteilen bietet. Es erzeugt eine automatische und stets aktuelle Dokumentation, die ohne manuelle Pflege auskommt, da Änderungen an den Controllern – wie neue Routen, geänderte Parameter oder Rückgabewerte – sofort in der Dokumentation sichtbar werden. Zudem bietet es über Swagger UI eine interaktive Benutzeroberfläche, die es ermöglicht, sämtliche Endpunkte direkt im Browser zu testen. Entwickler können auf diese Weise Eingabedaten an die API senden und Rückgabewerte prüfen, ohne zusätzliche Tools wie Postman oder externe Skripte nutzen zu müssen. Darüber hinaus erleichtert Swagger die Zusammenarbeit zwischen Backend- und Frontend-Teams, da die Schnittstellen visuell nachvollziehbar sind und auch von Frontend-Entwicklern oder externen Partnern ohne tiefgehendes technisches Wissen verstanden und getestet werden können. Schließlich reduziert es den Integrations- und Kommunikationsaufwand erheblich, weil Endpunkt-Spezifikationen, erwartete Parameter und Rückgabewerte klar dargestellt sind, wodurch Fehler durch Missverständnisse in der API-Kommunikation deutlich minimiert werden.

Um dies zu erreichen, nutzen die Controller Anmerkungen aus dem Paket `io.swagger.v3.oas.annotations`: `@Tag` gruppiert die Endpunkte thematisch (z. B. „Books“, „Books“)

„Users“) und beschreibt deren Zweck, damit die Oberfläche übersichtlich bleibt. `@Operation` beschreibt detailliert, was ein Endpunkt macht, welche Parameter er benötigt, welche Datentypen er verarbeitet und welche Rückgabeformate erwartet werden. `@ApiResponses` definiert alle möglichen Antwortstatus (z. B. 200, 201, 400, 404) samt deren Bedeutung, damit sowohl Entwickler als auch Tester alle Szenarien nachvollziehen können.

Durch diese Integration verändert sich auch der Workflow: Neue Endpunkte können schneller getestet und sofort validiert werden, da Swagger UI die API bereits in einer benutzbaren Form darstellt, ohne dass ein separates Test-Setup erforderlich ist. Für Qualitätssicherung und externe Partner bedeutet dies, dass sie sofort sehen können, welche Schnittstellen verfügbar sind und wie sie diese nutzen können – inklusive Beispiel-Requests und -Responses, die automatisch aus den Anmerkungen generiert werden.

Swagger beeinflusst dabei nicht die Geschäftslogik oder Architektur: Es handelt sich lediglich um eine deklarative Erweiterung in der Controller-Schicht. Die Domain bleibt weiterhin isoliert und unberührt von diesen technischen Hilfsmitteln. Dennoch sorgt Swagger dafür, dass die API nicht nur maschinenlesbar (für Clients), sondern auch menschenlesbar und direkt testbar ist, was die Entwicklungsgeschwindigkeit und die Wartbarkeit der gesamten Anwendung deutlich erhöht.

Ein zentrales Augenmerk liegt in dieser Schicht auf der Sicherheit und Zugriffskontrolle. Hier kommen Annotationen wie `@PreAuthorize` zum Einsatz, die sicherstellen, dass nur autorisierte Benutzer oder bestimmte Rollen auf einzelne Endpunkte zugreifen können, und `@AuthenticationPrincipal`, mit dem der aktuell authentifizierte Benutzer automatisch in den Methoden verfügbar ist. Diese Mechanismen erlauben es, bereits auf Controller-Ebene fein granular zu steuern, wer welche Aktionen durchführen darf, und verhindern unberechtigte Zugriffe frühzeitig.

Die Controller enthalten selbst keine Geschäftslogik, sondern fungieren ausschließlich als Vermittler. Ihre Aufgaben sind klar umrisSEN:

- Sie nehmen die eingehenden Daten entgegen,
- validieren sie (häufig durch `@Valid` und Validierungsregeln in den Request-DTOs),
- und übergeben die aufbereiteten Informationen an die jeweiligen Inbound Ports (Service-Interfaces).

Von dort wird die eigentliche Geschäftslogik in der Domain ausgeführt. Nach der Verarbeitung übernehmen die Controller die Ergebnisse aus der Domain, wandeln sie mithilfe von Mappern in passende Response-DTOs um und stellen sie den Clients bereit.

Um die Interaktion mit den Clients zu verbessern, werden alle Antworten mit HATEOAS-Links angereichert. Diese Links erweitern die zurückgegebenen Daten um kontextabhängi-

ge Navigationsmöglichkeiten, etwa Verweise auf eigene Ressourcen (`self`), Bearbeitungsaktionen (`update`) oder Löschoptionen (`delete`). Dadurch wird die API selbstbeschreibend und unterstützt die Clients aktiv dabei, mögliche nächste Schritte zu erkennen, ohne dass zusätzliche Logik auf der Client-Seite notwendig ist.

Ein wesentlicher Bestandteil dieser Ebene ist außerdem die DTO- und Mapper-Struktur, die dafür sorgt, dass der Datenfluss zwischen Client, Controller und Domain klar getrennt, performant und zweckmäßig bleibt. Diese Architektur vermeidet nicht nur eine direkte Kopplung der Controller an die Domain-Modelle, sondern optimiert auch die Menge und Struktur der übertragenen Daten, um die Anwendung insgesamt effizienter und wartbarer zu gestalten.

Die DTOs (Data Transfer Objects) werden in verschiedene Varianten unterteilt, um die Datenübertragung situationsabhängig anzupassen:

- Request-DTOs wie `UserLoginRequestDTO` oder `BookCreateRequestDTO` stellen sicher, dass eingehende Daten vom Client in einer validierten und klar strukturierten Form vorliegen. Durch Validierungsanmerkungen wie `@NotNull` oder `@NotBlank` wird bereits auf Controller-Ebene sichergestellt, dass fehlerhafte oder unvollständige Daten nicht in die Geschäftslogik gelangen.
- Response-DTOs sind für die gezielte und optimierte Ausgabe von Daten zuständig. Es gibt sie in unterschiedlichen Varianten, um nur die für die jeweilige Situation benötigten Informationen bereitzustellen.

Ein Beispiel dafür sind sogenannte Short-Response-DTOs, die bewusst nur eine Teilmenge der verfügbaren Daten enthalten – etwa bei Listenansichten, wo lediglich eine kurze Zusammenfassung wie Titel, ID oder ein Vorschaubild nötig ist. Daneben existieren auch detaillierte Response-DTOs wie Profil- oder Detailansichten (`UserProfileResponseDTO`, `BookResponseDTO`), die alle für einen spezifischen Anwendungsfall relevanten Felder liefern.

Diese Aufteilung dient einem klaren Zweck: Nur die wirklich benötigten Daten werden übertragen, was sowohl den Datenverkehr als auch die Last auf Server und Client reduziert. Indem unnötige Informationen weggelassen werden, wird das System performanter und die Antwortzeiten bleiben kurz – besonders bei Anfragen, die große Datenmengen oder viele Objekte betreffen.

Die Mapper übernehmen in diesem Konzept eine zentrale Rolle, indem sie für die Transformation zwischen den Domain-Objekten und den unterschiedlichen DTO-Varianten verantwortlich sind. Spezialisierte Klassen wie `BookWebMapper`, `AuthorWebMapper` oder `AvatarWebMapper` sorgen dafür, dass diese Konvertierungen an einem zentralen Ort stattfinden und konsistent ausgeführt werden.

Diese Mapper verhindern, dass Controller eigene Konvertierungs- oder Geschäftslogik enthalten, und halten diese dadurch schlank und übersichtlich. Sie stellen sicher, dass immer die richtige DTO-Variante je nach Anwendungsfall automatisch erzeugt wird – etwa kompakte Short-DTOs für schnelle Listenansichten oder vollständige DTOs für Detailseiten. Darüber hinaus ermöglichen sie die zentrale Steuerung von wiederkehrenden Prozessen wie der Anreicherung von HATEOAS-Links, der Umwandlung verschachtelter Strukturen oder der Filterung sensibler Informationen, ohne dass Logik mehrfach im Code verteilt ist.

Durch diese klare Trennung und die zentrale Handhabung in den Mappern bleibt das System übersichtlich, modular und effizient. Der Datenfluss wird jederzeit kontrolliert, Übertragungen beschränken sich auf das Nötigste, und Änderungen an der Datenstruktur lassen sich einfach und ohne doppelten Aufwand an einem Ort – den Mappern – durchführen, was die Wartbarkeit und Erweiterbarkeit der gesamten Anwendung deutlich erhöht.

4.1.2.2.2 Inbound Adapter (Web Layer)

Der Outbound-Adapter innerhalb der Adapters-Schicht stellt die verbindende Ebene zwischen der Domain und sämtlichen Persistenz- sowie Integrationsmechanismen dar. Seine Hauptaufgabe ist es, sicherzustellen, dass die fachliche Geschäftslogik vollständig unabhängig von technischen Details wie JPA, SQL, Transaktionssteuerung oder der eingesetzten Datenbanktechnologie bleibt. Sämtliche Implementierungsdetails zur Datenhaltung und -abfrage werden in dieser Schicht gekapselt, sodass die Domain ausschließlich über klar definierte Outbound Ports (Interfaces) mit dem Speicher- oder Integrationssystem interagiert. Diese Entkopplung ermöglicht es, die Persistenztechnologie – sei es ein relationales System wie PostgreSQL, eine NoSQL-Datenbank oder ein externer Dienst – zu wechseln oder zu erweitern, ohne die Geschäftslogik anpassen zu müssen.

Kern dieser Schicht sind die Repository-Adapter, beispielsweise `BookRepositoryAdapter`, `UserRepositoryAdapter` oder `NotificationRepositoryAdapter`. Diese Klassen implementieren die in der Domain definierten Outbound Ports wie `IBookRepository`, `IUserRepository` oder `INotificationRepository`. Sie dienen als Brücke zwischen der fachlich geprägten Domain und den technischen Details der Datenbankanbindung. Die Adapter nutzen intern die Spring-Data-JPA-Repositories, welche im Paket `persistence.repository` definiert sind, um CRUD-Operationen sowie komplexe Abfragen auszuführen. Zugleich binden sie die Persistence-Mapper ein, um die bidirektionale Transformation zwischen JPA-Entities und den Domain-Modellen sicherzustellen. Auf diese Weise arbeitet die Domain aus-

schließlich mit fachlichen Objekten, während sämtliche Details der Datenbanklogik und -optimierung innerhalb der Persistence-Schicht verbleiben.

Ein zentrales Element dieser Schicht sind die JPA-Entities, die im Paket `persistence.entity` zu finden sind. Diese Klassen spiegeln die Struktur der zugrunde liegenden Datenbanktabellen wider und sind mit Anmerkungen wie `@Entity`, `@Table`, `@Column`, `@Id`, `@OneToMany` und `@ManyToOne` ausgestattet. Ihre Hauptaufgabe ist es, die Datenbankstruktur mit allen technischen Aspekten wie Primärschlüsseln, Relationen, Indexen und Spaltenzuordnungen abzubilden. Diese Entities werden innerhalb der Domain bewusst nicht verwendet, um die fachliche Logik vollständig von technischen Abhängigkeiten zu isolieren. Sie dienen ausschließlich als Speicher- und Transportobjekte innerhalb der Persistenz-Schicht und werden bei Bedarf über Mapper in Domain-Modelle überführt.

Die Persistence-Mapper, angesiedelt im Paket `persistence.mapper`, übernehmen die vollständige Transformation zwischen Domain-Modellen und JPA-Entities in beide Richtungen. Sie gewährleisten, dass:

- Änderungen an der Datenbankstruktur, wie neue Spalten, geänderte Tabellenbeziehungen oder zusätzliche technische Felder, nicht die Domain-Modelle beeinflussen.
- Die Domain-Modelle rein fachlich bleiben, frei von technischen Annotationen oder JPA-spezifischen Details.
- Alle Umwandlungen konsistent, nachvollziehbar und zentralisiert erfolgen, wodurch doppelte Logik, Streuung von Mapping-Code und Fehlerquellen vermieden werden.
- Erweiterungen – wie zusätzliche Felder in Entities, neue Relationen oder das Einführen von Audit-Feldern – ohne Risiko für die Geschäftslogik implementiert werden können.

Ein weiteres Schlüsselement sind die Spring-Data-Repository-Interfaces, die im Paket `persistence.repository` definiert sind. Diese Interfaces bieten die technische Grundlage für Standardoperationen wie `findAll()`, `save()`, `deleteById()` und erlauben darüber hinaus abgeleitete Abfragen wie `findByTitleContaining`, `findByUserId`, `existsByUserIdAndListId` oder komplexe Filterabfragen. Ein besonders aussagekräftiges Beispiel hierfür ist die Methode:

```
1 @Query("""
2     SELECT b
3     FROM BookEntity b
4     JOIN BooksStatusEntity bs ON bs.book = b
5     JOIN FollowerEntity f ON f.followedPerson = bs.user
6     WHERE f.follower.id = :userId
7     GROUP BY b
```

```

8   ORDER BY COUNT(bs.id) DESC
9   LIMIT 12
10  """
11 List<BookEntity> findPopularBooksAmongFollowed(@Param("userId") Long
userId);

```

Diese Abfrage ermittelt die zwölf beliebtesten Bücher unter den Personen, denen ein bestimmter Nutzer folgt. Dazu verknüpft sie `BookEntity` mit `BooksStatusEntity`, um die Bücher und deren Statusinformationen (z. B. gelesen, bewertet) zu erhalten, und kombiniert dies mit `FollowerEntity`, um nur die Bücher der vom Nutzer verfolgten Personen zu berücksichtigen. Anschließend werden die Bücher gruppiert, die Häufigkeit ihrer Vorkommen gezählt und absteigend sortiert, bevor die zwölf meistgenutzten Bücher zurückgegeben werden. Diese Logik wird vollständig in der Datenbank ausgeführt, wodurch große Datenmengen effizient verarbeitet werden können. Die von solchen Methoden gelieferten JPA-Entities werden in den Repository-Adaptoren mit Hilfe der Persistence-Mapper in fachliche Domain-Modelle transformiert und an die Service- oder Use-Case-Schichten weitergereicht. Auf diese Weise arbeitet die Domain konsequent gegen abstrakte Interfaces, bleibt vollständig von JPA, SQL und konkreten Datenbankdetails entkoppelt und kann sich ausschließlich auf die fachliche Logik konzentrieren.

Sie stellen die unmittelbare Verbindung zur Datenbank dar. Die Repository-Adapter greifen auf diese Interfaces zurück, übernehmen die Ergebnisse, wandeln sie mit den Persistence-Mappern in Domain-Modelle um und geben sie an die Service- oder Use-Case-Schichten weiter. So entwickelt die Domain vollständig gegen Interfaces und bleibt von der konkreten Datenhaltung oder Implementierungsdetails abstrahiert.

Die Adapter dienen dabei als Bindeglied zur Anwendungslogik, indem sie die von den Repositories gelieferten Daten aufbereiten und direkt an die Domain-Services oder Use-Case-Schicht weitergeben. So bleibt die Geschäftslogik schlank und vollständig von technischen Details isoliert.

Ein entscheidender Aspekt dieser Architektur ist die Testbarkeit. Da die Domain nur die Outbound Ports kennt, können die tatsächlichen Repository-Implementierungen in Tests problemlos durch Mocks oder In-Memory-Fakes ersetzt werden. Dadurch lassen sich Unit-Tests der Geschäftslogik vollständig unabhängig von einer echten Datenbank ausführen. Integrationstests können wiederum die tatsächlichen Adapter und Repositories einbinden, um das Zusammenspiel mit der Datenbank zu prüfen. Diese Trennung erleichtert kontinuierliches Testen, erhöht die Zuverlässigkeit und verkürzt die Entwicklungszyklen.

Die Persistence-Schicht berücksichtigt außerdem Aspekte der Performance und Optimierung. Durch die explizite Kontrolle über Abfragen und Mapping kann die Architektur Probleme wie das klassische N+1-Problem vermeiden, indem Abfragen gezielt mit JOIN FETCH oder maßgeschneiderten Query-Methoden optimiert werden. Pagination und Sor-

tierung (über `Pageable`) sind integrale Bestandteile, um große Datenmengen effizient zu handhaben. Ebenso wird der Einsatz von Lazy und Eager Loading bewusst gesteuert, damit nur wirklich benötigte Daten geladen werden, ohne die Domain mit technischen Feinheiten zu belasten. Sämtliche Optimierungsentscheidungen verbleiben in der Persistence-Schicht, sodass die Domain weder mit der Performance-Logik konfrontiert noch von ihr abhängig ist.

Ein weiterer zentraler Punkt ist das Transaktionsmanagement. Während die Service-Schicht in der Regel die Transaktionsgrenzen definiert, wird innerhalb der Persistence-Schicht sichergestellt, dass alle Datenbankoperationen konsistent ausgeführt werden. Spring-Transaktionen (über `@Transactional`) sorgen dafür, dass Änderungen entweder vollständig übernommen oder vollständig zurückgerollt werden. Die Domain bleibt dabei unbeeinträchtigt, da die Transaktionslogik transparent und ausschließlich auf der Infrastruktur-Ebene implementiert wird.

Neben der lokalen Datenbankanbindung enthält die Persistence-Schicht auch spezialisierte Outbound Adapter wie den `BookApiService`. Dieser Adapter greift nicht auf die interne Datenbank zu, sondern bindet eine externe Datenquelle – die Google Books API – an. Seine Aufgabe ist es, eine bestimmte Anzahl von Büchern aus der API abzurufen, die gelieferten JSON-Daten zu verarbeiten und sie in Domain-Modelle vom Typ `Book` zu transformieren. Der Service verwendet `RestTemplate`, um API-Anfragen zu stellen, und `ObjectMapper`, um die JSON-Antwort in eine `JsonNode`-Struktur zu parsen. Die Methode `fetchBooksFromGoogleApi(int max)` lädt die Ergebnisse in Blöcken von maximal 40 Einträgen, extrahiert Titel, ISBN, Autor, Verlag, Beschreibung, Seitenanzahl, Veröffentlichungsjahr, Genre, Sprache und das Cover-Bild, kürzt überlange Beschreibungen automatisch auf 255 Zeichen und liefert eine Liste von Domain-Book-Objekten zurück. Durch diese Kapselung bleibt die externe API für die restliche Anwendung unsichtbar: Die Domain arbeitet ausschließlich mit vollständig aufbereiteten Domain-Objekten und muss keine Details der API kennen. Sollte die externe Datenquelle ausgetauscht oder erweitert werden, bleibt die übrige Architektur unverändert.

Dank dieser klaren Struktur ist die Outbound-Adapter-Schicht modular, erweiterbar und zukunftssicher. Sie stellt sicher, dass alle technischen Details – von der Datenbankanbindung über Performance-Optimierungen bis hin zu externen Integrationen – isoliert in einer Schicht verbleiben. Die Domain ist so konsequent vor technischen Abhängigkeiten geschützt und kann sich vollständig auf die Geschäftslogik konzentrieren. Änderungen an der Datenbank, der Speicherstrategie oder externen Schnittstellen lassen sich daher ohne Eingriffe in die fachliche Logik umsetzen. Gleichzeitig wird durch die Entkopplung eine hohe Testbarkeit, Flexibilität und langfristige Wartbarkeit der Anwendung gewährleistet, was die Outbound-Adapter-Schicht zu einem stabilen Fundament für den Daten- und

Integrationsfluss der gesamten Architektur macht.

4.1.2.3 Application

4.1.2.3.1 Service (UseCases)

Das `application.service`-Paket bildet die zentrale Schicht für die Implementierung der Anwendungslogik des Systems. Es enthält alle Klassen, die konkrete Use Cases abbilden. Jede Klasse steht für einen klar abgegrenzten Anwendungsfall, der die fachlichen Regeln umsetzt, Entitäten aus dem Domain-`model` koordiniert und bei Bedarf auf die Outbound Ports zugreift. Die Use Cases sind vollständig von technischen Details wie Frameworks, Controllern oder Datenbankimplementierungen entkoppelt und werden ausschließlich über die definierten Inbound Ports (Service-Interfaces) genutzt. Alle Use Cases sind nach einem einheitlichen Muster aufgebaut: Sie implementieren die Inbound Ports (Service-Interfaces) wie `IJwtService`, `IAuthService`, `IBookService` oder `INotificationService`. Sie koordinieren die Entitäten aus dem `model`-Paket, wenden die jeweiligen Geschäftsregeln an und greifen über die Outbound Ports (Repository-Interfaces) auf die Persistenzschicht oder externe Systeme zu. Mapper kommen nur dann zum Einsatz, wenn Daten zwischen Domain-Objekten und DTOs konvertiert werden müssen.

Die Klasse `JwtUseCases` spielt eine zentrale Rolle bei der Datensicherheit und Authentifizierung des Systems. Sie ist für die Erstellung, Signierung und Validierung von JSON Web Tokens (JWTs) zuständig, die verwendet werden, um Benutzer eindeutig zu identifizieren und einen sicheren Zugriff auf geschützte Endpunkte zu ermöglichen. Ein JWT enthält Benutzerinformationen wie Name, ID und eine Ablaufzeit, wird mit einem geheimen Schlüssel signiert, um Manipulationen zu verhindern, und kann von Controllern und Services validiert werden. Die Kapselung dieser Logik in `JwtUseCases` stellt sicher, dass Sicherheitsfunktionen zentral verwaltet werden, keine verteilte Token-Logik in anderen Klassen existiert und Änderungen an der Token-Verwaltung, wie Ablaufzeiten oder zusätzliche Claims, leicht umsetzbar sind. Controller und Services nutzen ausschließlich die bereitgestellten Methoden, wodurch Authentifizierung konsistent, sicher und unabhängig von Frameworks umgesetzt wird.

`AuthUseCases` ist für die Registrierung, Anmeldung und Abmeldung von Benutzern zuständig. Es arbeitet mit `IUserRepository`, um Benutzerinformationen zu speichern oder abzurufen, und verwendet `JwtUseCases`, um Tokens für authentifizierte Benutzer zu erstellen. Diese Tokens identifizieren Benutzer bei weiteren Anfragen und prüfen deren

Berechtigungen, wodurch die Authentifizierung zentralisiert wird, ohne dass Controller direkte Sicherheitslogik beinhalten.

`BookUseCases` verwaltet das Erstellen, Bearbeiten und Abrufen von Büchern. Neben den Standardfunktionen bietet es auch die Möglichkeit, besonders populäre Bücher zu ermitteln. Dazu wird neben `IBookRepository` auch der `MostPopularBooksComparator` genutzt, mit dem Bücher nach Popularität sortiert und entsprechend ausgegeben werden können. Auf diese Weise stellt die Klasse sicher, dass die beliebtesten Bücher dem Benutzer gezielt bereitgestellt werden, ohne dass die zugrunde liegende Architektur verändert werden muss.

`NotificationUseCases` ist für die Erstellung und Verwaltung von Benachrichtigungen zuständig. Es nutzt `INotificationRepository` und verwendet das `NotificationType`-Enum, um verschiedene Benachrichtigungsarten wie Benutzer-Follows, Listen-Follows oder Kommentar-Likes zu unterscheiden und korrekt zu verarbeiten.

Alle Klassen im `service`-Paket sind nach dem *Single-Responsibility-Prinzip* konzipiert und übernehmen nur die Prozesse, für die sie zuständig sind. Sie bilden das Bindeglied zwischen der fachlichen Logik der Domain und den Ports und sorgen dafür, dass die Domain-Schicht klar strukturiert, robust, testbar und erweiterbar bleibt, ohne von technischen Details beeinflusst zu werden.

4.1.2.3.2 Config

Die `application.config`-Schicht übernimmt die zentrale Rolle in der Anwendung, indem sie sämtliche systemweiten Konfigurationen, Sicherheitsmechanismen, Fehlerbehandlung und Initialisierungsprozesse bündelt. Diese Schicht ist als Cross-Cutting Layer konzipiert: Sie unterstützt sowohl die Inbound-Adapter (Web Layer) als auch die Outbound-Adapter (Persistence und externe Integrationen), bleibt jedoch vollständig unabhängig von der Domain-Logik. Alle hier definierten Komponenten dienen dazu, die Architektur konsistent, sicher und erweiterbar zu gestalten, ohne die Geschäftslogik mit technischen Details zu belasten.

Security und Authentifizierung von Umut

Ein wesentlicher Bestandteil dieser Schicht ist die Sicherheitsinfrastruktur, die in mehreren dedizierten Komponenten organisiert ist.

Die Klasse `SecurityConfig` definiert die gesamte Spring-Security-Konfiguration, einschließlich Authentifizierungs- und Autorisierungsregeln, CORS-Richtlinien und der Inte-

gration des JWT-basierten Sicherheitsmechanismus. Sie registriert Filter wie den `JwtAuthFilter`, der jeden eingehenden Request abfängt, das JWT-Token extrahiert, validiert und – falls gültig – den zugehörigen Benutzer über `JwtUtil` identifiziert. Der Filter lädt die Benutzerinformationen (`UserDetails`) und bindet sie an den `SecurityContext`, sodass alle nachfolgenden Schichten die Authentifizierung nahtlos nutzen können.

Zusätzlich sorgen spezialisierte Sicherheitsklassen wie `CommentSecurity`, `CommentLikeSecurity` und `ListSecurity` für eine feingranulare Zugriffskontrolle. Sie prüfen kontextabhängig, ob ein authentifizierter Benutzer berechtigt ist, auf bestimmte Ressourcen zuzugreifen oder Aktionen wie Bearbeiten, Liken oder Löschen durchzuführen. Diese Mechanismen arbeiten eng mit den Controllern zusammen und werden meist über `@PreAuthorize`-Annotationen eingebunden, wodurch die Zugriffslogik zentral, testbar und deklarativ bleibt.

Fehlerbehandlung und Stabilität von Umut

Die zentrale Fehlerbehandlung wird von den Klassen `GlobalExceptionHandler` und `RestExceptionHandler` übernommen, die mit `@RestControllerAdvice` annotiert sind. Diese Komponenten fangen systemweit alle relevanten Ausnahmen ab – von Validierungsfehlern (`MethodArgumentNotValidException`), Datenintegritätsverletzungen (`DataIntegrityViolationException`), Zugriffsverletzungen (`AccessDeniedException`) bis hin zu allgemeinen Laufzeitfehlern. Jede Ausnahme wird in ein einheitliches Format umgewandelt, das durch das `ErrorResponseDTO` repräsentiert wird. Dieses Objekt enthält konsistente Informationen wie HTTP-Statuscode, Fehlermeldung, Zeitstempel, betroffene Pfade und optionale Feldfehler. Dadurch erhalten alle Clients eine standardisierte und nachvollziehbare Antwort, was die Integration und Fehlersuche erheblich erleichtert.

Infrastruktur- und Bean-Konfiguration von Umut und Yusuf

Die Klasse `BeanConfiguration` definiert alle wiederverwendbaren Beans, die in der gesamten Anwendung benötigt werden. Dazu gehören:

- Ein `PasswordEncoder`, der für die sichere Speicherung und Überprüfung von Benutzerpasswörtern verwendet wird.
- Ein vorkonfigurierter `ObjectMapper`, der JSON-Serialisierung und -Deserialisierung mit spezifischen Einstellungen (z. B. für Datum/Zeit) ermöglicht.
- Mapper-Instanzen für DTO-Transformationen, die zentral bereitgestellt werden, um Redundanz zu vermeiden.

Zusätzlich sorgt die Klasse `WebConfig` für globale Webeinstellungen, wie etwa CORS-Konfigurationen, Message Converters und allgemeine Spring-MVC-Anpassungen, sodass die Web-Schicht ohne zusätzliche Konfiguration konsistent funktioniert.

Dateninitialisierung und externe API-Integration

Ein besonders wichtiger Bestandteil dieser Schicht ist die `BookSeeder`-Klasse, die mit `@Component` und `ApplicationRunner` implementiert ist. Diese Komponente wird automatisch beim Start der Anwendung ausgeführt und sorgt dafür, dass die Datenbank mit initialen Buchdaten befüllt wird, falls diese leer ist.

Dazu prüft sie zunächst über das `IBookRepository`, ob bereits Bücher in der Datenbank vorhanden sind. Falls nicht, ruft sie über den `BookApiService` bis zu 1000 Bücher von der Google Books API ab. Die abgerufenen Daten werden automatisch verarbeitet, in Domain-Modelle vom Typ `Book` transformiert und über das Repository in die Datenbank geschrieben. Dieser Prozess stellt sicher, dass die Anwendung auch bei einer frischen Installation oder in Testumgebungen mit einer aussagekräftigen Grundmenge an Daten arbeitet, ohne dass manuell Eingriffe notwendig sind.

Sicherheitsarchitektur und JWT-Integration von Umut

Ein Hauptbestandteil dieser Schicht ist die Sicherheitskonfiguration, die durch mehrere zentrale Klassen umgesetzt wird.

Die Klasse `SecurityConfig` definiert die gesamte Spring-Security-Infrastruktur: Sie konfiguriert Authentifizierungs- und Autorisierungsregeln, stellt CORS-Einstellungen bereit und integriert den JWT-basierten Sicherheitsmechanismus. Ein wesentlicher Teil davon ist der `JwtAuthFilter`, der eingehende HTTP-Requests abfängt, das JWT-Token ausliest, dessen Gültigkeit über `JwtUtil` prüft und – bei Erfolg – den zugehörigen Benutzer in den `SecurityContext` einbindet. So wird sichergestellt, dass nachfolgende Schichten nahtlos mit dem authentifizierten Benutzerkontext arbeiten können.

Zusätzlich kommen spezialisierte Sicherheitsklassen wie `CommentSecurity`, `CommentLikeSecurity` und `ListSecurity` zum Einsatz. Diese Komponenten stellen eine feingranulare Zugriffskontrolle sicher, indem sie kontextbezogen prüfen, ob der aktuell authentifizierte Benutzer berechtigt ist, bestimmte Kommentare, Likes oder Listen zu erstellen, zu bearbeiten oder zu löschen. Diese Sicherheitsprüfungen werden häufig über `@PreAuthorize`-Annotationen in den Controllern eingebunden, wodurch die Zugriffslogik zentral verwaltet und leicht testbar bleibt. **Zentrale Fehlerbehandlung**

Die globale Fehlerbehandlung wird in den Klassen `GlobalExceptionHandler` und

`RestExceptionHandler` gebündelt, die mit `@RestControllerAdvice` annotiert sind. Diese Komponenten fangen eine Vielzahl von Ausnahmen systemweit ab – darunter `EntityNotFoundException`, `BadCredentialsException`, `MethodArgumentNotValidException`, `DataIntegrityViolationException`, `ConstraintViolationException`, `AccessDeniedException` sowie allgemeine `Exception`-Fälle.

Alle Fehler werden in ein standardisiertes Format konvertiert, das durch das `ErrorResponseDTO` repräsentiert wird. Dieses enthält neben Zeitstempel, Statuscode und Fehlerbeschreibung auch Details wie den betroffenen API-Pfad und – bei Validierungsfehlern – eine Liste der fehlerhaften Felder. So erhalten alle Clients konsistente und nachvollziehbare Antworten, was die Fehlersuche und Integration erheblich vereinfacht.

Infrastruktur- und Bean-Konfiguration

Die Klasse `BeanConfiguration` stellt alle wiederverwendbaren Beans der Anwendung bereit. Dazu gehören:

- Ein sicherer `PasswordEncoder`, der für die Verschlüsselung und Überprüfung von Passwörtern genutzt wird.
- Ein vorkonfigurierter `ObjectMapper`, der JSON-Serialisierung und -Deserialisierung mit projektspezifischen Einstellungen ermöglicht.
- Mapper-Instanzen, die für die Transformation zwischen DTOs und Domain-Objekten verwendet und zentral bereitgestellt werden, um Redundanzen zu vermeiden.

Die Klasse `WebConfig` ergänzt diese Infrastruktur durch globale Spring-MVC-Konfigurationen. Sie steuert CORS-Einstellungen, Message Converters und weitere Web-Parameter, damit die Inbound-Schicht ohne zusätzlichen Konfigurationsaufwand konsistent funktioniert.

Automatische Dateninitialisierung mit BookSeeder

Ein besonders wichtiger Bestandteil der `Application.Config`-Schicht ist die `BookSeeder`-Klasse, die als `@Component` implementiert und über das `ApplicationRunner`-Interface ausgeführt wird. Diese Komponente wird beim Start der Anwendung automatisch aktiv und sorgt dafür, dass die Datenbank initial mit Buchdaten gefüllt wird, falls sie leer ist. Zunächst überprüft der Seeder über das `IBookRepository`, ob bereits Einträge in der Datenbank existieren. Falls keine vorhanden sind, ruft er mithilfe des `BookApiService` bis zu 1000 Bücher von der Google Books API ab.

Die Daten werden dabei in Blöcken von maximal 40 Einträgen abgerufen, da die API paginierte Ergebnisse liefert. Für jedes Buch extrahiert der Seeder relevante Metadaten wie Titel, ISBN, Autor(en), Verlag, Beschreibung, Seitenanzahl, Erscheinungsjahr, Genre, Sprache und das Cover-Bild. Um die Kompatibilität mit der Datenbank zu gewährleisten,

werden besonders lange Beschreibungen automatisch auf 255 Zeichen gekürzt.

Alle verarbeiteten Bücher werden in Domain-Modelle vom Typ `Book` konvertiert – nicht in JPA-Entities – damit die Persistenzdetails von der Geschäftslogik getrennt bleiben. Anschließend werden die Bücher über das `IBookRepository` gespeichert und dauerhaft in die Datenbank geschrieben.

Dank dieser automatisierten Initialisierung steht die Anwendung bereits beim ersten Start – sei es in Produktions-, Test- oder Entwicklungsumgebungen – mit einer realistischen Datenbasis bereit, ohne dass manuell Daten importiert werden müssen.

Weitere unterstützende Komponenten

Zusätzlich enthält diese Schicht weitere Hilfsklassen, die spezifische Aufgaben übernehmen:

- `MostPopularBooksComparator`: Stellt eine individuelle Vergleichslogik zur Verfügung, mit der Bücher beispielsweise nach Beliebtheit sortiert werden können, bevor sie an Clients ausgeliefert werden.
- `StatusInitializer`: Initialisiert vordefinierte Statuswerte oder Systemeinstellungen beim Anwendungsstart, sodass diese in der gesamten Anwendung verfügbar sind.

Rolle in der Gesamtarchitektur

Die `Application.Config`-Schicht fungiert als zentrales Fundament für alle technischen und querschnittlichen Funktionen der Anwendung. Sie stellt sicher, dass Authentifizierung, Autorisierung und Zugriffskontrollen konsistent und sicher umgesetzt werden. Zudem gewährleistet sie eine einheitliche und nachvollziehbare Fehlerbehandlung in allen Schichten. Durch die Bereitstellung zentraler Beans und Konfigurationen unterstützt sie die Wiederverwendbarkeit und erleichtert die Wartbarkeit des Systems.

Darüber hinaus stellt sie sicher, dass die Anwendung dank Seeder und Initializer auch in neuen oder frisch aufgesetzten Umgebungen sofort betriebsbereit ist. Diese klare Trennung ermöglicht es der Domain, sich vollständig auf die Geschäftslogik zu konzentrieren, während `Application.Config` alle unterstützenden Mechanismen übernimmt, ohne die Kernlogik zu belasten. Dadurch bleibt die Gesamtarchitektur modular, stabil und langfristig erweiterbar.

4.2 Übersicht und Zusammenspiel der Komponenten

Das Gesamtsystem von *Bookies* folgt einer klar strukturierten Client-Server-Architektur, in der die mobile Anwendung (Frontend) über eine REST-API mit dem Backend kommuniziert.

nisiert. Die beiden Komponenten arbeiten über definierte Schnittstellen eng zusammen, wobei insbesondere auf Standardisierung, Sicherheit und Erweiterbarkeit geachtet wurde.

Kommunikation zwischen Frontend und Backend

Die mobile Anwendung greift mit Hilfe der Bibliothek `Retrofit` auf die REST-Endpunkte des Backends zu. Die Benutzeroberfläche der App wurde mit XML-basierten Layout-Dateien umgesetzt, welche die visuelle Struktur definieren und mit Kotlin-View-Logik über das MVVM-Muster verbunden sind. Die URL-Struktur der Schnittstellen folgt einem konsistenten Schema, beispielsweise `https://<server-url>/api/books` oder `.../api/users/{id}`. Die Endpunkte sind nach funktionalen Modulen gegliedert (z. B. Auth, Books, Lists, Comments) und verwenden Standardmethoden wie `GET`, `POST`, `PUT` und `DELETE`.

Für jede dieser Schnittstellen existiert im Frontend ein entsprechendes `Service-Interface`, das die Kommunikation kapselt. Dadurch wird die Netzwerklogik zentral verwaltet und vom Rest der Anwendung isoliert.

Authentifizierung und Sicherheit

Zur Absicherung der Schnittstellen wird ein JWT-basiertes Authentifizierungssystem eingesetzt. Nach erfolgreichem Login erhält der Client ein Token, das bei jedem weiteren Request im HTTP-Header (`Authorization: Bearer <token>`) mitgesendet wird. Auf Serverseite wird dieses Token validiert, bevor der Zugriff auf geschützte Ressourcen gewährt wird.

Ein zentraler `Retrofit-Interceptor` im Client sorgt dafür, dass das Token automatisch an alle relevanten Anfragen angehängt wird. Darüber hinaus können mit Hilfe desselben Interceptors auch Fehlerzustände (z. B. 401 Unauthorized) abgefangen und gezielt behandelt werden.

Datenfluss und Synchronisation

Der typische Ablauf einer systemweiten Interaktion lässt sich wie folgt beschreiben:

1. Benutzeraktion in der mobilen App (z. B. „Buch hinzufügen“) wird über die UI ausgelöst.
2. Das zuständige `ViewModel` ruft die zugehörige Repository-Methode auf.
3. Die Repository-Schicht führt einen HTTP-Request via `Retrofit` zum Backend aus.
4. Das Backend verarbeitet die Anfrage (z. B. speichert das Buch in der Datenbank) und gibt eine Antwort zurück.
5. Das `ViewModel` erhält die Antwort und aktualisiert die `View` über `LiveData`.

Dadurch entsteht ein asynchroner, aber reaktiver Datenfluss zwischen UI, Logik und Server, der eine nahtlose Benutzererfahrung ermöglicht.

Komponente	Technologie	Aufgabe
Mobile App	Kotlin, XML, MVVM, Retrofit	UI, Nutzerinteraktion, Datenanzeige
API-Kommunikation	Retrofit + Interceptor	REST-Aufrufe, Authentifizierung, Logging
Backend	Spring Boot (Java)	Geschäftslogik, Datenhaltung, Sicherheitsprüfung
Datenbank	PostgreSQL	Persistente Speicherung der Entitäten

Tabelle 31: Überblick über Systemkomponenten und deren Aufgaben

4.3 Beschreibung der Komponenten

4.3.1 Mobile App (Kotlin)

4.3.1.1 Gesamtstruktur der App Die mobile Anwendung *Bookies* wurde mit Kotlin und dem MVVM-Architekturmuster (Model-View-ViewModel) für die Android-Plattform entwickelt. Sie bildet die gesamte clientseitige Schicht des Systems ab und ermöglicht den Zugriff auf sämtliche Funktionen über eine moderne und benutzerfreundliche Oberfläche.

In dieser Komponente werden alle zentralen Funktionen wie Benutzeranmeldung, Buchsuche, Interaktionen mit Büchern (z. B. Bewerten, Kommentieren, Hinzufügen zu Listen), Benachrichtigungen sowie die Verwaltung des Nutzerprofils umgesetzt. Dabei kommuniziert die App über die REST-API mit dem Backend, verarbeitet die Antworten und stellt die Ergebnisse über XML-basierte Layouts dar.

Die Implementierung folgt einem konsequent modularen Aufbau: Die View-Schicht (Activity/Fragment) ist von der Geschäftslogik (ViewModel) sowie der Datenzugriffs-schicht (Repository) klar getrennt. Daten werden über Retrofit vom Backend abgerufen und mit Hilfe von DTOs und Mappern in interne Strukturen überführt. Die Synchronisation mit der Benutzeroberfläche erfolgt reaktiv über `LiveData` oder `StateFlow`.

Im Folgenden werden die einzelnen Bestandteile der mobilen Anwendung detailliert beschrieben – beginnend mit der Aufgabenteilung, den Schnittstellen zu anderen Komponenten, den verwendeten Datenstrukturen, der konkreten Funktionsweise einzelner Module sowie der eingesetzten Qualitätssicherungsmaßnahmen.

4.3.1.2 Technische Schnittstellen Die mobile Anwendung *Bookies* kommuniziert über eine klar definierte, REST-basierte Schnittstelle mit dem serverseitigen Backend. Für jede funktionale Domäne – etwa Authentifizierung, Bücher, Rezensionen, Listen oder

Benachrichtigungen – ist im Frontend ein dediziertes Service-Interface vorgesehen, etwa `Books ApiService` oder `Reviews ApiService`. Diese Interfaces sind als Kotlin-Definitionen im Paket `data.api.service` implementiert und kapseln HTTP-Anfragen mithilfe der `Retrofit`-Bibliothek. Die zugehörigen Endpunkte folgen einem konsistenten Pfadschema wie `GET /api/books/popular` oder `POST /api/liked-books`. Die Authentifizierung erfolgt über ein JWT-Token, das durch einen zentralen Interceptor (`TokenInterceptor`) automatisiert an alle geschützten Anfragen angehängt wird.

Jede Interaktion zwischen der Benutzeroberfläche und dem Backend folgt einem standardisierten Ablauf: Das zuständige `ViewModel` stößt eine Aktion an und delegiert diese an das zugehörige Repository, das über ein `Retrofit`-Service-Interface mit dem Backend kommuniziert. Die Antwort wird in Form eines DTO (Data Transfer Object) empfangen, durch einen `Mapper` in ein domänenspezifisches Modell überführt und anschließend via `LiveData` oder `StateFlow` an die View zurückgegeben.

Diese strukturierte Entkopplung schafft Transparenz und Nachvollziehbarkeit im Datenfluss, erleichtert die Erweiterung um neue Endpunkte und erhöht die Fehlertoleranz durch zentralisiertes Fehlerhandling, beispielsweise über Methoden wie `SafeApiCall`.

4.3.1.3 Datenmodelle und Klassenstruktur Die Daten- und Klassenstruktur der Anwendung *Bookies* folgt einem konsequent modularisierten Aufbau auf Basis der MVVM-Architektur. Sie gliedert sich in drei zentrale Ebenen: Datenmodelle, Logikschicht sowie UI-Komponenten.

Auf der Ebene der Datenmodelle erfolgt eine klare Trennung zwischen internen Domänenobjekten und externen Kommunikationsstrukturen. Die `domain.model`-Klassen wie `Book`, `User` oder `CombinedSearchResult` bilden die zentrale Datenstruktur für die interne Logik. Für die API-Kommunikation kommen spezialisierte DTOs zum Einsatz (`data.api.dto`, `data.remote.dto`), darunter etwa `BookDto`, `UserDto` oder `CombinedSearchResponse`. Ergänzend werden Anfrage- und Antwortobjekte (`data.model`) wie `SignInRequest`, `RegisterRequest` oder `AddBookRequest` verwendet. Komplexe API-Antworten werden mithilfe von Wrapper-Klassen wie

`EmbeddedBooksResponse` oder `UserEmbedded` modelliert.

Die Repository- und Service-Schicht fungiert als Schnittstelle zwischen `ViewModels` und Backend. Repositories wie `AuthRepository`, `BookRepository` oder `FollowersRepository` kapseln die Datenzugriffe und binden `Retrofit`-Services wie `Books ApiService`, `User ApiService` oder `Notifications ApiService` ein. Die zentrale Initialisierung erfolgt über `RetrofitClient.kt`, wobei sicherheitsrelevante Aufgaben – insbesondere das Anhängen von JWT-Tokens – durch den `TokenInterceptor` automatisiert übernommen werden.

`Mapper`-Klassen wie `BookMapper` oder `UserMapper` transformieren die externen DTOs

in konsistente Domänenmodelle. Diese Entkopplung sorgt für Robustheit gegenüber API-Änderungen und erhöht die Wartbarkeit der internen Logik.

Die UI-Schicht besteht aus klar abgegrenzten Komponenten: `Fragments` wie `SearchFragment`, `ProfileFragment` oder `BookInfoPageFragment` bilden die Hauptansichten; ergänzend kommen `BottomSheets` und Dialoge wie `AddBookBottomSheet`, `BookInfoMoreDialog` oder `AddBookToListBottomSheet` zum Einsatz. Für die Datenbindung sorgen spezialisierte Adapter wie `BookAdapter`, `CombinedSearchAdapter` oder `NotificationAdapter`. Jedes Modul wird durch ein eigenes `ViewModel` mit zugehöriger `Factory` begleitet (z.B. `SearchViewModel`, `ListsViewModel`). Die gesamte Navigationsstruktur ist in der `nav_graph.xml` hinterlegt und definiert die Übergänge zwischen den Hauptansichten wie `Home`, `Search`, `Profile` und `BookDetails`.

4.3.1.4 Funktionale Teilkomponenten

4.3.1.4.1 Authentifizierung und Einstieg **Aufgabe:** Diese Komponente ermittelt beim Start der Anwendung den Sitzungsstatus des Benutzers und startet anschließend den entsprechenden Bildschirmablauf. Gleichzeitig umfasst sie grundlegende Funktionen zur Identitätsprüfung wie Anmelden (Sign In), Konto erstellen (Register) und Passwort zurücksetzen. Auf diese Weise wird sichergestellt, dass nur autorisierte Benutzer Zugriff auf das System erhalten.

Ebene	Komponente
View	SplashActivity, FirstPageActivity, SignInActivity, RegisterActivity
View Model	SignInViewModel, RegisterViewModel
API-Service	LikedReview ApiService.kt
RetrofitClient	Authorization-Header wird automatisch hinzugefügt

Über die API werden folgende Endpunkte verwendet:

- POST /api/login
- POST /api/register

Beim Start der Anwendung übernimmt die `SplashActivity` die initiale Token-Prüfung. Ist ein gültiges Token vorhanden, wird der Nutzer direkt zur `MainActivity` weitergeleitet; andernfalls erfolgt die Navigation zur `FirstPageActivity`.

Dort wählt der Benutzer zwischen „Anmelden“ oder „Registrieren“. Die eingegebenen Zugangsdaten werden an das Backend übermittelt, das bei erfolgreicher Authentifizierung

ein JWT-Token zurückgibt. Dieses Token wird in den `SharedPreferences` gespeichert, um künftige Sitzungen zu verwalten.

Nach erfolgreicher Anmeldung oder Registrierung erfolgt die Weiterleitung zur `MainActivity`, die den Hauptzugang zur Anwendung bildet.

4.3.1.4.2 Startseite und Buchanzeigen Aufgabe: Das `HomePageFragment` bildet die zentrale Anlaufstelle der mobilen Anwendung und bietet eine dynamisch generierte Übersicht relevanter Buchinhalte. Die Darstellung ist in vier funktionale Kernbereiche unterteilt:

- **Populäre Bücher zwischen Freunden** – Horizontal scrollbare Darstellung aktueller Buchempfehlungen aus dem sozialen Umfeld des Nutzers mittels `RecyclerView`.
- „**Explore More“ Empfehlungen** – Kategorieübergreifende Vorschläge, die auf individuellen Nutzerinteressen basieren und personalisiert bereitgestellt werden.
- **Aktuelle Rezensionen** – Nutzerverfasste Buchbewertungen, sortiert nach Popularität und Interaktionsmetriken.
- **Populäre Listen** – Öffentliche Buchsammlungen anderer Nutzer, gerankt nach Follower-Anzahl, Likes oder Ansichten.

Diese Struktur bietet nicht nur eine thematisch vielfältige Einstiegsebene, sondern verbindet personalisierte Inhalte mit sozialen Komponenten in einem konsistenten UI-Konzept.

Funktion	View (Fragment/Layout)	ViewModel	API / Repository
Populäre Bücher by Friends	<code>MainActivity.kt</code> → <code>HomePageFragment.kt</code> <code>InnerBookFragment.kt</code> → <code>ThreeColumnFragment.kt</code> <code>fragment_home_page.xml</code> <code>fragment_inner_books.xml</code> <code>fragment_three_column.xml</code> <code>item_book_grid.xml</code>	<code>PopularBooksViewModel</code>	<code>BookRepository</code> , <code>Books ApiService</code> GET /api/books
Explore More Bücher	<code>MainActivity.kt</code> → <code>HomePageFragment.kt</code> <code>InnerBookFragment.kt</code> → <code>ThreeColumnFragment.kt</code> <code>fragment_home_page.xml</code> <code>fragment_inner_books.xml</code> <code>item_book_grid.xml</code>	<code>PopularBooksViewModel</code>	<code>BookRepository</code> , <code>Books ApiService</code> GET /api/books
Reviews	<code>MainActivity.kt</code> → <code>HomePageFragment.kt</code> <code>ReviewsFragment.kt</code> <code>fragment_reviews.xml</code> <code>item_review.xml</code>	<code>ReviewsViewModel</code>	<code>ReviewsRepository</code> , <code>Reviews ApiService</code> GET /api/reviews
Populäre Listen	<code>MainActivity.kt</code> → <code>HomePageFragment.kt</code> <code>ListsFragment.kt</code> <code>fragment_lists.xml</code> <code>item_list_other.xml</code>	<code>ListsViewModel</code>	<code>ListsRepository</code> , <code>Lists ApiService</code> GET /api/lists

Tabelle 32: Übersicht der Komponenten der Startseite und zugehöriger Strukturen

Funktionsweise

Beim Laden des `HomePageFragment` erfolgt im Lifecycle-Callback `onViewCreated()` die Initialisierung der zuständigen `ViewModels`, die unmittelbar mit der asynchronen Datenbeschaffung über die jeweilige Repository-Schicht beginnen. Die entsprechenden API-Anfragen werden dabei über `Retrofit` durchgeführt.

Die empfangenen Daten – etwa empfohlene Bücher oder Rezensionen – werden über `LiveData` an die View gebunden und automatisch in den zugehörigen UI-Komponenten dargestellt. Die Ausgabe erfolgt durch spezialisierte `RecyclerView`-Adapter wie `BookAdapter`, `ReviewsAdapter` oder `ListsAdapter`.

Bei Interaktionen, etwa einem Klick auf ein Buchcover, wird der Nutzer mithilfe des `NavController` zur Detailansicht im `BookInfoPageFragment` weitergeleitet.

4.3.1.4.3 Suche und Filter Aufgabe:

Das Modul „Suche und Filter“ stellt ein zentrales Discovery-Element der mobilen App dar. Es erlaubt den Nutzer:innen, gezielt nach Büchern und Benutzerprofilen zu suchen –

mit sofortiger Reaktion auf Eingaben (Live-Suche).

Die Suchergebnisse beinhalten sowohl Bücher als auch Nutzer und werden dynamisch über zwei **RecyclerViews** präsentiert.

Darüber hinaus lassen sich Inhalte anhand von Genre, Sprache und über Sortierkriterien wie „Most Popular“ und „Highly Rated“ weiter eingrenzen. Diese Funktionen tragen maßgeblich zur Personalisierung des Nutzungserlebnisses bei.

Funktion	View (Fragment/Layout)	ViewModel	API / Repository
Populäre Bücher by Friends	MainActivity.kt → HomePageFragment.kt InnerBookFragment.kt → ThreeColumnFragment.kt fragment_home_page.xml fragment_inner_books.xml fragment_three_column.xml item_book_grid.xml	PopularBooksViewModel	BookRepository, Books ApiService GET /api/books
Explore More Bücher	MainActivity.kt → HomePageFragment.kt InnerBookFragment.kt → ThreeColumnFragment.kt fragment_home_page.xml fragment_inner_books.xml item_book_grid.xml	PopularBooksViewModel	BookRepository, Books ApiService GET /api/books
Reviews	MainActivity.kt → HomePageFragment.kt ReviewsFragment.kt fragment_reviews.xml item_review.xml	ReviewsViewModel	ReviewsRepository, Reviews ApiService GET /api/reviews
Populäre Listen	MainActivity.kt → HomePageFragment.kt ListsFragment.kt fragment_lists.xml item_list_other.xml	ListsViewModel	ListsRepository, Lists ApiService GET /api/lists

Tabelle 33: Überblick über die Komponenten der Startseite und zugehöriger Strukturen

Funktionsweise

Im Suchprozess geben Nutzer:innen ihren Begriff in das Eingabefeld ein, wobei ein `onTextChanged`-Listener automatisch einen Live-Request auslöst. Das **SearchViewModel** verarbeitet die Eingabe und delegiert die Anfrage an das **SearchRepository**, welches über den **Search ApiService** eine GET-Anfrage an den Endpunkt `/search?q={query}` stellt.

Die Antwort besteht aus einem kombinierten Datenobjekt (z. B. `CombinedSearchResponse`), das sowohl Buch- als auch Nutzerergebnisse umfasst. Diese werden im UI über zwei getrennte **RecyclerViews** angezeigt.

Filterkriterien wie Genre und Sprache sowie Sortieroptionen wie „Most Popular“ oder „Highly Rated“ werden direkt im **ViewModel** interpretiert oder als Query-Parameter mitgesendet.

Funktionsweise

Die Empfehlungen basieren auf mehreren Faktoren: gelikte Bücher (`liked_books`), häufig genutzte Genres, Schnittmengen mit ähnlichen Nutzerprofilen (z. B. gemeinsame Follower), sowie Aktivitäten in Rezensionen und Buchlisten. Die erhaltenen Ergebnisse werden mithilfe von Mappern in ein geeignetes UI-Modell überführt.

In der Anwendung erfolgt die Darstellung entweder in einer Swipe-Ansicht mit Interaktionsoptionen wie „Will ich lesen“ oder „Nicht interessiert“, oder alternativ in einer Grid-Ansicht. Alle Benutzeraktionen – einschließlich Swipes, Likes oder dem Hinzufügen zu Listen – werden zurück in die Empfehlungshistorie integriert und fließen in zukünftige Vorschläge ein.

Das System bietet vielfältiges Erweiterungspotenzial im Bereich intelligenter Empfehlungen. Eine mögliche Weiterentwicklung ist die Integration eines ML-Modells zur semantischen Ähnlichkeitsbestimmung zwischen Büchern, etwa durch den Einsatz von Embedding-Verfahren. Darüber hinaus könnte eine langfristige Analyse des individuellen Leseverhaltens – inklusive Lesezeit, Genrewchsel oder Abbruchraten – zusätzliche Erkenntnisse für personalisierte Vorschläge liefern. Perspektivisch lassen sich Empfehlungen zudem kontextsensitiv gestalten, beispielsweise abhängig von Tageszeit, saisonalen Präferenzen oder plattformspezifischen Trenddaten.

4.3.1.4.4 Buchinteraktionen Dieses Modul erfasst sämtliche Interaktionen, die Nutzer:innen mit einzelnen Büchern durchführen können. Dazu zählen das Hinzufügen zu vordefinierten Listen wie „Gelesen“, „Will ich lesen“ oder „Favorit“ – entweder direkt aus der `book_info_page` oder über Shortcuts wie `addbook`. Zusätzlich können Bücher gezielt zu individuell erstellten Listen hinzugefügt werden. Weitere Aktionen umfassen das Liken bzw. Entliken von Büchern, das Schreiben von Rezensionen sowie das Vergabe von Bewertungen. Diese Interaktionen sind kontextunabhängig nutzbar, also aus verschiedenen Bereichen der App heraus – etwa von der Startseite, aus Detailansichten oder vom Profil.

Die Kommunikation mit dem Backend erfolgt über dedizierte Repositories, während die Synchronisation der Zustände durch `LiveData`-gebundene ViewModels sichergestellt wird. Das Modul ist eng in das Profil-, Empfehlungs- und soziale System der App eingebunden und bildet eine zentrale Schnittstelle zwischen individuellen Präferenzen und kollektiver Bewertung.

Aktion / Kontext	View (Layout/Dialog/Sheet)	ViewModel	API / Repository
Shortcut: AddBookButton → Buch direkt zur „Gelesen“-Liste hinzufügen, Like / Rezension optional	AddBookBottomSheet.kt, bottom_sheet_add_book.xml	SearchViewModel → loadSearchResults(keyword, genre, lang)	Read ApiService → POST api/books-status/read, LikedBooksRepository, ReviewsRepository
Item-Klick auf Buchkarte → Detailansicht	item_book_grid.xml → fragment_book_info_page.xml, BookInfoPageFragment	BookViewModel	Books ApiService, LikedBooks ApiService, Reviews ApiService
More-Button → erweiterter Dialog, Aktionen: Liken, Rezension schreiben, Bewerten, Gelesenliste oder Leseliste hinzufügen	BookInfoPageFragment.kt → BookInfoMoreDialog.kt, book_info_more_dialog.xml (Dialog mit Sternen & Textfeld)	BookViewModel, ReviewViewModel, LikedBooksViewModel	POST zu api/liked-books, api/reviews, ReviewCreateRequest
Profilseite: Liste → Buch hinzufügen	item_list_profile.xml → add_to_list_card.xml → AddBookToListBottomSheet	—	Lists ApiService → POST /lists/{id}/books

Tabelle 34: Übersicht der Buchinteraktionsmöglichkeiten mit zugehörigen Komponenten

Funktionsweise

Der direkte Zugriff über den `AddBookButton` öffnet das `AddBookBottomSheet`, in dem Nutzer:innen ein Buch suchen, auswählen, liken oder optional rezensieren können. Standardmäßig wird das Buch zur Liste „Gelesen“ hinzugefügt, Review und Bewertung sind optional. Die Aktionen werden über das `LikedBooksRepository` und `ReviewsRepository` mit dem Backend synchronisiert.

Beim Zugriff über Home oder Suchergebnisse wird die Detailansicht mit aggregierter Bewertung angezeigt (berechnet über einen API-Call). Über den Button „Mehr“ lässt sich der `BookInfoMoreDialog` öffnen, in dem Nutzer:innen das Buch als „Gelesen“ markieren (`ReadRepository`), zur Liste „Will ich lesen“ hinzufügen, liken (nur nach Lese-Status), sowie Kommentar und Bewertung speichern können.

In der Listenansicht des Profils führt ein Klick auf die `AddBookToListCardView` zur Öffnung des `AddBookToListBottomSheet`. Nach Auswahl der Ziel-Liste wird ein entsprechender Request an den `Lists ApiService` gesendet.

Technische Highlights des Moduls zeigen eine konsequente Umsetzung des MVVM-Architekturmusters: Die UI-Komponenten wie BottomSheet, Dialog oder Fragment übernehmen ausschließlich die Darstellung und Interaktionserfassung. Die Logik wird in den zugehörigen ViewModels oder innerhalb strukturierter Coroutine-Scopes ausgeführt. Repositories kapseln sämtliche API-Kommunikation über Retrofit und stellen eine klare Trennung zur Datenquelle sicher.

Die Rückmeldungen an die Nutzer:innen erfolgen reaktiv über `LiveData` sowie visuelles Feedback wie Toasts oder aktualisierte Icons. Buchbezogene Interaktionen sind dabei systemweit rückwirkend sichtbar – etwa durch Like-Status im Profil oder Bewertungs-

werte im Home-Bereich – was die Kohärenz und Transparenz der Anwendung wesentlich unterstützt.

4.3.1.4.5 Listenverwaltung Aufgabe: Das Modul Listenverwaltung bietet Nutzer:innen die Möglichkeit, individuelle Buchlisten zu erstellen, zu organisieren und öffentlich zu teilen. Neben der persönlichen Strukturierung – etwa für „Gelesen“ oder „To-Read“ – erfüllt diese Funktion eine soziale Rolle: Listen können von anderen eingesehen, kommentiert oder abonniert werden, wodurch ein gemeinschaftlicher Austausch entsteht.

Typische Anwendungsfälle umfassen die Erstellung thematisch kuratierter Sammlungen (z.B. „Beste Thriller“, „Sommerlektüre“), das Hinzufügen von Büchern aus unterschiedlichen Kontexten – etwa über das Profil (benutzerdefinierte Listen), die Buchansicht (vordefinierte Listen) oder das `AddBookBottomSheet`. Öffentliche Listen anderer Nutzer:innen sind über das `ListsFragment` auf der Startseite auffindbar, wo populäre Sammlungen nach sozialen Metriken angezeigt und direkt gefolgt werden können.

Funktion	View (Layout/Dialog/Fragment)	ViewModel	API / Repository
Eigene Liste erstellen	<code>MainActivity.kt</code> → <code>ProfileFragment</code> → <code>ListsFragment(ShowAddListDialog)</code> <code>item_list_profile.xml</code> (z.B. vom Profil aus aufrufbar)	<code>ListsViewModel</code>	<code>Lists ApiService</code> → POST <code>/api/lists</code>
Buch zur Liste hinzufügen	<code>AddBookToListBottomSheet.kt</code> , <code>item_list_profile.xml</code> → <code>add_to_list_card.xml</code>	<code>ListsViewModel</code>	<code>Lists ApiService</code> → POST <code>/api/books-in-list</code>
Listen auf Profil anzeigen	<code>ProfileFragment</code> → <code>ListsFragment</code> <code>fragment_profile.xml</code> , <code>item_list_profile.xml</code>	<code>ListsViewModel</code>	<code>Lists ApiService</code> → GET <code>/api/lists/user/{id}</code>
Listen anderer Nutzer anzeigen	<code>OtherUserProfileFragment.kt</code> → <code>ListsFragment.kt</code> , <code>fragment_other_user_lists.xml</code>	<code>ListsViewModel</code>	<code>Lists ApiService</code> → GET <code>/api/lists/user/{id}</code>
Einer Liste folgen / entfolgen	Follow/Unfollow-Button in <code>item_list_other.xml</code>	<code>ListFollowViewModel</code>	<code>ListFollows ApiService</code> → POST <code>/api/list-follows/user/{id}</code> , DELETE <code>/api/list-follows/user/{id}</code>

Tabelle 35: Überblick über die Komponenten des Listenverwaltungsmoduls

Funktionsweise

Eigene Listen können direkt über das Profil erstellt werden. Dabei öffnet sich ein Dialogfeld zur Eingabe von Titel und Beschreibung; anschließend wird ein POST-Request an `/lists` gesendet. Das Hinzufügen eines Buchs zu einer bestehenden Liste erfolgt über

das `AddBookToListBottomSheet`, das z. B. aus `item_list_profile.xml` aufgerufen wird. Nach Auswahl einer Liste wird ein POST-Request an `/books-in-list` ausgelöst.

Im `ProfileFragment` werden eigene Listen mithilfe von `item_list_profile.xml` angezeigt, basierend auf einem GET-Request an `/lists/user/{id}`. Öffentliche Listen anderer Nutzer:innen können über das `ListsFragment` auf der Startseite oder über das `OtherUserProfileFragment` eingesehen werden (ebenfalls via GET `/lists/user/{id}`).

Zusätzlich besteht die Möglichkeit, Listen zu folgen oder das Folgen aufzuheben. Gefolgte Listen erscheinen im Profil unter „Gefolgte Listen“. Die zugehörigen Interaktionen werden über den `ListFollows ApiService` abgewickelt.

4.3.1.4.6 Benachrichtigungen Aufgabe:

Das Benachrichtigungssystem informiert Nutzer:innen in Echtzeit über zentrale soziale Ereignisse auf der Plattform. Dazu zählen unter anderem neue Follower, Likes auf eigene Rezensionen sowie neue Follower auf selbst erstellte Listen. Die Benachrichtigungen werden in einem eigenen Bereich (`NotificationsFragment`) gebündelt dargestellt und bieten zugleich direkte Navigationsmöglichkeiten zu den zugehörigen Inhalten – etwa zu einem Buch, einer Liste oder einem Profil. Dieses System fördert nicht nur die Interaktion, sondern erhöht auch die Sichtbarkeit und Reichweite nutzergenerierter Inhalte.

Funktion	View (Fragment/Layout)	ViewModel	API / Repository
Benachrichtigungen anzeigen und löschen	<code>MainActivity.kt</code> → <code>NotificationsFragment.kt</code> <code>fragment_notifications.xml</code> , <code>item_notification.xml</code>	<code>NotificationsViewModel</code>	<code>NotificationsRepository</code> , <code>NotificationApiService</code> GET <code>/notifications</code> DELETE <code>/notifications/{id}</code>
Navigation bei Klick	<code>OnClickListener</code> in <code>NotificationFragment.kt</code>	<code>NotificationViewModel</code>	(Client-seitige Navigation zu Buch-/Listen-/Profilansicht)

Tabelle 36: Übersicht über das Benachrichtigungsmodul und dessen technische Umsetzung

Funktionsweise

Beim Aufruf des `NotificationsFragment` stellt das zugehörige ViewModel einen GET `/notifications`-Request an das Backend, wobei der eingeloggte Nutzer über ein JWT-Token authentifiziert wird. Die Antwort enthält eine Liste von `NotificationDto`-Objekten mit Angaben zu Typ, Quelle, Ziel und Nachricht, etwa:

```

1 {
2   "id": 16,
3   "createdAt": "",
4   "updatedAt": "",
5   "senderId": 1,
6   "receiverId": 2,
```

```

7   "type": "LIKE_COMMENT",
8   "targetId": "3",
9   "read": false,
10  "_links": {
11    "self": {
12      "href": "http://10.0.2.2:8080/api/notifications"
13    }
14  }
15 }

```

Diese Daten werden im UI über einen RecyclerView mithilfe des NotificationsAdapter dargestellt. Die Navigation bei einem Klick auf ein Item erfolgt abhängig vom type oder target: Bei Typ "REVIEW" wird zur entsprechenden Rezension gesprungen, bei "LIST" zum ListFragment und bei "USER" zum OtherUserProfileFragment. So fungieren Benachrichtigungen nicht nur als Informationsquelle, sondern auch als interaktive Navigationselemente.

4.3.1.4.7 Profil und soziale Funktionen Aufgabe:

Das Profilmodul bildet die zentrale Benutzeransicht der Bookies-App und vereint persönliche Verwaltung mit sozialer Interaktion. Nutzer:innen können hier ihre gelesenen Bücher, Favoriten und geplanten Lektüren organisieren, eigene Buchlisten erstellen oder löschen sowie Profile anderer einsehen und folgen bzw. entfolgen. Zusätzlich lassen sich über fremde Profile Leselisten, erstellte Listen und gelikte Inhalte nachvollziehen. Eine Übersicht über Follower und Following unterstützt die soziale Vernetzung. Damit entwickelt sich Bookies über die reine Buchverwaltung hinaus zu einem interaktiven Netzwerk für Literaturinteressierte.

Funktion	View (Fragment/Layout)	ViewModel	API / Repository
Eigenes Profil anzeigen	MainActivity.kt ProfileFragment.kt fragment_profile.xml Tabs: Gelesen / Will ich lesen / Favoriten (Threecolumnfragment)	ProfileViewModel, UserViewModel, ReadViewModel, LikedBooksViewModel	User ApiService → GET /users/{id} ReadBooks ApiService GET books-status/read/{id} Read ApiService, LikedBooks ApiService GET liked-books/{id}
Eigene Listen anzeigen/bearbeiten	ProfileFragment ListsFragment item_list_profile.xml	ProfileListsViewModel	Lists ApiService → GET /lists/user/{id}
Fremdes Profil anzeigen	fragment_other_user_profile.xml item_user=?	OtherUserProfileViewModel	User ApiService → GET /users/{id}
Folgen / Entfolgen	Follow/Unfollow-Button in fragment_other_user_profile.xml item_user=?	FollowersViewModel?, UserListViewModel	Followers ApiService → POST /followers, DELETE /followers/{id}
Gefolgte Nutzer:innen / Follower anzeigen	Tabs in ProfileFragment.kt / OtherUserProfile.kt → UserListFragment	UserListViewModel, FollowerViewModel?	Followers ApiService → GET /followers, GET /following

Tabelle 37: Übersicht über die Funktionen und Komponenten im Profilmodul

Funktionsweise

Beim Öffnen des Profils, realisiert über `fragment_profile.xml`, wird der eingeloggte Nutzer aus den `SharedPreferences` geladen und dessen Daten zur Anzeige vorbereitet. Die Profilansicht gliedert sich in mehrere inhaltliche Bereiche. In den Tabs „Gelesen“, „Will ich lesen“ und „Favoriten“ werden Bücher je nach Interaktionsstatus dargestellt, basierend auf den Daten, die über den Endpunkt `GET /users/{id}/books` vom Backend abgerufen werden. Parallel dazu erfolgt die Anzeige und Verwaltung eigener Buchlisten: Nutzer:innen können ihre erstellten Listen einsehen, bearbeiten, löschen oder um weitere Bücher ergänzen. Die hierfür erforderlichen Informationen werden über `GET /lists/user/{id}` geladen.

Darüber hinaus lassen sich Profile anderer Nutzer:innen direkt aus Kontexten wie Rezensionen oder öffentlichen Listen heraus aufrufen. Diese Profilansichten zeigen unter anderem die Biografie, die Anzahl der Follower sowie eine Vorschau auf veröffentlichte Listen. Der Follow-/Unfollow-Status ist dynamisch steuerbar und basiert auf einer Kombination aus `GET /users/{id}`, `GET /followers`, `GET /following` sowie den Schreiboperationen `POST /followers` und `DELETE /followers/{id}`. Follower- und Following-Informationen sind in Tabs verfügbar, wobei ein Klick auf einen Eintrag zur entsprechenden Profilansicht weiterleitet. Der aktuelle Follower-Status wird im Hintergrund über das `FollowersViewModel` aktualisiert, sodass die Anzeige stets synchron mit dem Systemzustand bleibt.

4.3.2 Backend App (Java)

4.3.2.1 User Modul von Umut

Das User-Modell bildet die Grundlage für alle benutzerbezogenen Funktionen innerhalb des Systems und steht im Zentrum der Domain-Schicht. Es repräsentiert jeden registrierten Anwender und ist die Basis für Authentifizierung, Autorisierung und personalisierte Inhalte. Jeder Benutzer wird durch eine eindeutige ID identifiziert, die aus der von `BaseModel` geerbten Struktur stammt. Neben dieser eindeutigen Kennung enthält das Modell die wesentlichen Stammdaten wie Benutzername und E-Mail-Adresse, welche jeweils als eindeutig hinterlegt sind. Zur sicheren Anmeldung wird das Passwort in Form eines verschlüsselten Hashwerts gespeichert. Das Modell verwaltet außerdem eine Sammlung von Rollen wie `USER`, die als Grundlage für die Zugriffskontrolle dienen.

Das User-Modell ist eng mit anderen zentralen Systemkomponenten verknüpft. Jeder Benutzer kann persönliche Buchlisten erstellen und verwalten, Kommentare zu Büchern verfassen und anderen Nutzern folgen oder selbst Follower haben, wodurch die soziale

Interaktion innerhalb der Plattform unterstützt wird. Zudem ist das Modell mit dem Benachrichtigungssystem integriert, sodass Benutzer über relevante Ereignisse wie neue Follower, Listen oder Kommentare informiert werden. Durch die Vererbung von `BaseModel` verfügt jedes User-Objekt automatisch über Felder für Erstellungs- und Änderungsdatum, was die Nachvollziehbarkeit und Konsistenz der Daten sicherstellt.

Das User-Modell ist so konzipiert, dass es leicht erweiterbar und unabhängig von der technischen Infrastruktur bleibt. Es erfüllt die Anforderungen der Hexagonalen Architektur, indem es keine direkte Abhängigkeit zur Datenbank oder externen Frameworks enthält. Stattdessen wird der Zugriff auf Benutzerinformationen über ein abstrahiertes `UserRepository` abgewickelt, das von einem Persistence-Adapter, typischerweise auf Basis von Spring Data JPA, implementiert wird.

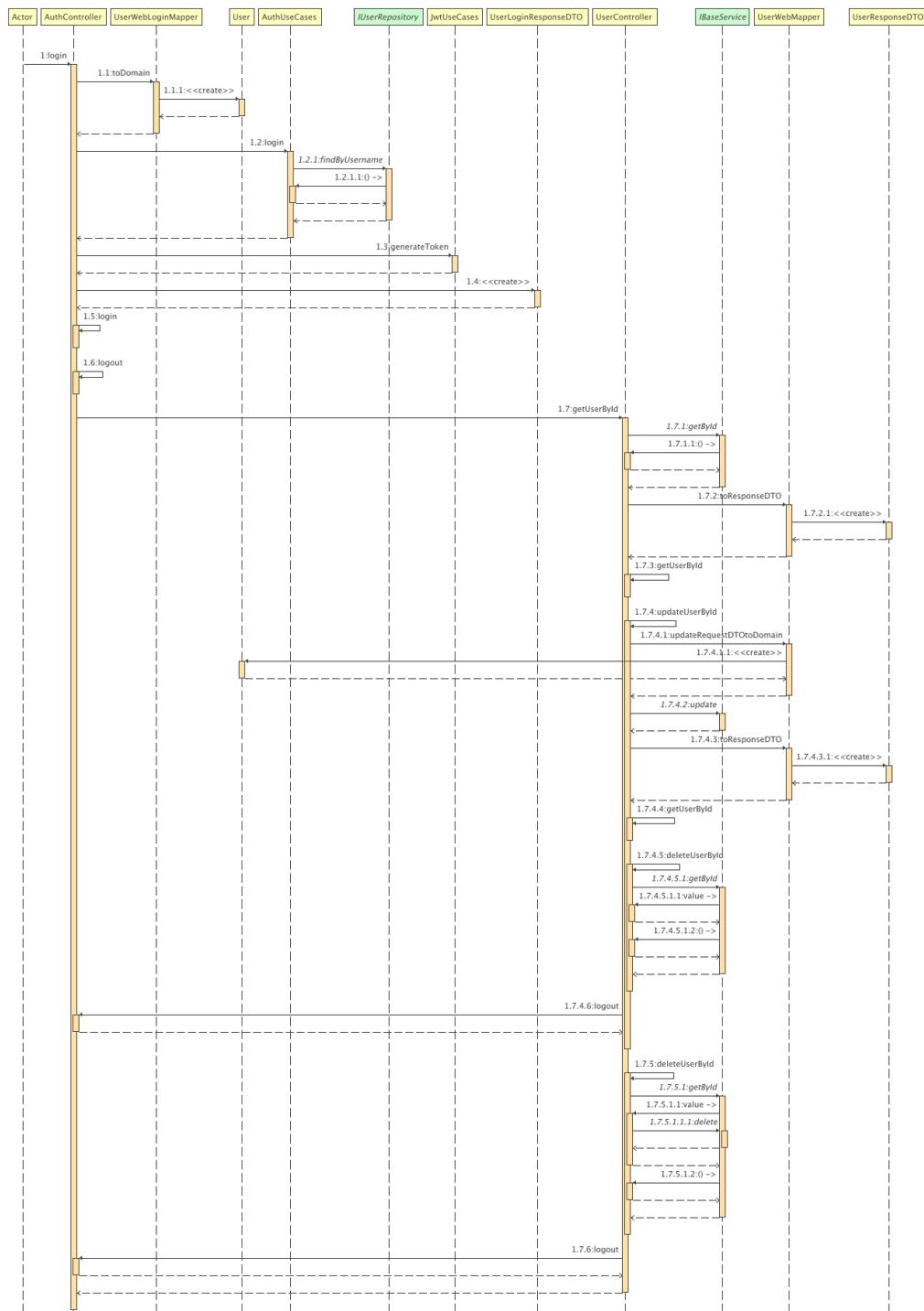


Abbildung 5: AuthController LoginSeqeunce

4.3.2.2 Auth Modul von Umut

Das Authentifizierungsmodul ist für die Registrierung, Anmeldung und Abmeldung von Benutzern im System verantwortlich und stellt sicher, dass nur autorisierte Nutzer Zugriff auf geschützte Funktionen der Anwendung erhalten. HTTP-Anfragen von Clients

werden zunächst vom **AuthController** entgegengenommen, der die eingehenden JSON-Daten verarbeitet, sie mithilfe von DTOs und Mappern in Domain-Objekte umwandelt und anschließend an die Anwendungsschicht weiterleitet. Ein typischer Ablauf bei der Registrierung besteht darin, dass die Benutzereingaben validiert und über die Anwendungsschicht in der Datenbank gespeichert werden.

Die zentrale Geschäftslogik für Registrierung und Login wird in der Klasse **AuthUseCases** umgesetzt, die das Interface **IAuthService** implementiert und damit als Inbound Port des Moduls fungiert. Auf diese Weise ist sie sowohl für Web-Controller als auch für andere Module innerhalb des Systems nutzbar. Ein wesentlicher Bestandteil des Moduls ist die Verwaltung von JWT-Tokens. Nach einem erfolgreichen Login wird ein Token generiert, das bei weiteren Anfragen zur Authentifizierung verwendet wird. Die Prüfung der Gültigkeit dieser Tokens erfolgt durch einen HTTP-Filter, während die Verwaltung und Generierung in der Klasse **JwtUseCases** gekapselt ist.

Der Datenzugriff erfolgt nicht direkt, sondern über ein in der Domäenschicht definierteres Interface **UserRepository**, das von einem Persistence-Adapter auf Basis von Spring Data JPA implementiert wird. Diese Trennung folgt dem Prinzip der Abhängigkeitsumkehr, einem zentralen Bestandteil der Hexagonalen Architektur, und stellt sicher, dass die Geschäftslogik unabhängig von technischen Details bleibt.

Zusammenfassend bietet das Auth-Modul eine klar strukturierte, modulare und leicht testbare Lösung für die gesamte Benutzerverwaltung und setzt dabei auf wiederverwendbare Services und eine sichere Token-basierte Authentifizierung.

4.3.2.3 Book Modul von Umut und Yusuf

Das Book-Modul übernimmt die Verwaltung sämtlicher buchbezogener Funktionen innerhalb des Systems und ermöglicht das Erstellen, Bearbeiten, Abrufen und Suchen von Büchern. Eingehende Benutzeranfragen werden zunächst vom **BookController** entgegengenommen, der als Inbound Adapter fungiert. Der Controller konvertiert die empfangenen JSON-Daten mithilfe von DTOs und Mappern in Domain-Objekte und leitet diese an die Anwendungsschicht weiter, wo die Klasse **BookUseCases** die Ausführung der entsprechenden Geschäftslogik übernimmt.

Beim Anlegen eines neuen Buches stellt die Anwendungsschicht sicher, dass alle Pflichtfelder wie Titel oder ISBN korrekt ausgefüllt sind und keine Duplikate entstehen. Nach erfolgreicher Validierung wird das Buch über die Persistenzschicht gespeichert. Die Anwendungsschicht selbst implementiert das Interface **IBookService**, wodurch ihre Funktionalitäten sowohl von Web-Controllern als auch von anderen Modulen oder Adapters genutzt werden können. Der Zugriff auf die Datenbank erfolgt dabei nicht direkt, sondern

über ein abstrahiertes Interface `IBookRepository`, dessen konkrete Implementierung in der Persistenzschicht auf Basis von Spring Data JPA realisiert wird.

Darüber hinaus unterstützt das Book-Modul eine flexible Suche und Filterung von Büchern. Benutzer können ihre Abfragen anhand von Kriterien wie Titel, Autor oder Genre einschränken. Die entsprechende Filterlogik wird vollständig in der Anwendungsschicht verarbeitet, während die eigentlichen Datenbankabfragen von spezialisierten Methoden im Repository übernommen werden.

Insgesamt stellt das Book-Modul eine klare Trennung zwischen Domäne, Anwendung und Infrastruktur sicher. Dank seines modularen Aufbaus bleibt es unabhängig von der Benutzeroberfläche, leicht testbar und problemlos erweiterbar.

4.3.2.4 Comment Modul von Umut und Yusuf

Das **Comment-Modul** ist für das Erstellen, Anzeigen und Löschen von Kommentaren zu Büchern verantwortlich und bildet eine zentrale Funktionalität der Anwendung, da es die Interaktion zwischen Benutzern und Inhalten ermöglicht. Benutzerinteraktionen beginnen auf der Ebene des `CommentController`, der REST-Endpunkte bereitstellt, über die Kommentare erstellt, abgerufen oder entfernt werden können. Eingehende HTTP-Anfragen werden mithilfe von DTOs und Mappern in Domain-Objekte umgewandelt und anschließend an die Anwendungsschicht weitergeleitet, wo die Geschäftslogik verarbeitet wird.

Die Ausführung der Kernlogik erfolgt in der Klasse `CommentUseCases`, die unter anderem die Validierung der Kommentarinhalte – wie Mindestlänge oder die Vermeidung leerer Einträge – übernimmt. Darüber hinaus sorgt sie für die korrekte Zuordnung der Kommentare zu den entsprechenden Büchern und organisiert die Speicherung über die Repository-Schicht. Bei Bedarf löst die Anwendungsschicht zusätzliche Ereignisse aus, beispielsweise zur Benachrichtigung des Buchautors über neue Kommentare.

Die Anwendungslogik ist über das Interface `ICommentService` gekapselt, das als Inbound Port fungiert und sicherstellt, dass alle zentralen Use Cases – wie das Erstellen von Kommentaren, das Abrufen aller Kommentare zu einem Buch und das Löschen von Kommentaren – von anderen Systemkomponenten genutzt werden können. Für die Persistenz wird ein abstrahiertes Interface `ICommentRepository` verwendet, das von einer JPA-basierten Implementierung in der Infrastruktur-Schicht umgesetzt wird. Dadurch bleibt die Anwendungslogik vollständig unabhängig von der konkreten Datenbanktechnologie.

Ein weiterer essenzieller Aspekt des Moduls ist die Zugriffskontrolle: Nur angemeldete Benutzer dürfen Kommentare verfassen, und das Löschen ist ausschließlich dem jeweiligen Kommentarautor oder einem Administrator gestattet. Diese Regeln sind integraler

Bestandteil der Anwendungslogik und werden innerhalb der Service-Klasse konsequent durchgesetzt. Insgesamt trägt das **Comment-Modul** erheblich zur sozialen Dynamik der Plattform bei und ist dank seiner modularen Struktur, der klaren Verantwortlichkeitstrennung und der Einbettung in die Hexagonale Architektur leicht erweiterbar, testbar und langfristig wartbar.

4.3.2.5 Follower Modul von Umut und Yusuf

Das Follower-Modul ermöglicht es Benutzern, anderen Nutzern innerhalb der Plattform zu folgen, und fördert damit die soziale Interaktion sowie die Personalisierung von Inhalten wie Benachrichtigungen über neue Bücher, Kommentare oder Aktivitäten von gefolgten Personen. Zentrale Anlaufstelle für alle follower-bezogenen Anfragen ist der **FollowController**, der REST-Endpunkte bereitstellt, über die Benutzer anderen Nutzern folgen oder das Folgen beenden können. Darüber hinaus lassen sich über diese Schnittstelle die Listen der Personen abrufen, denen ein Benutzer folgt, oder die ihn selbst verfolgen.

Die Geschäftslogik wird in der Klasse **FollowUseCases** umgesetzt, die das Interface **IFollowService** implementiert und alle Kernfunktionen des Moduls kapselt. Dazu zählen die Validierung, dass ein Benutzer sich nicht selbst folgen kann, die Überprüfung, ob eine Follower-Beziehung bereits existiert, sowie das Anlegen und Entfernen von Follower-Einträgen. Die Persistenz wird über das Interface **IFollowRepository** realisiert, das alle relevanten Methoden für das Speichern, Löschen und Abfragen von Follower-Beziehungen definiert. Dessen konkrete Implementierung erfolgt durch einen Adapter auf Basis von Spring Data JPA, wodurch die Domänenschicht vollständig von der Datenbanktechnologie entkoppelt bleibt.

Das zentrale Domain-Modell in diesem Modul repräsentiert eine Follower-Beziehung als Kombination zweier Benutzerreferenzen – des „folgenden Benutzers“ und des „gefolgten Benutzers“. Diese Beziehung wird mithilfe zusammengesetzter IDs und klar definierter Relationen im Datenmodell abgebildet. Zusätzlich stellt das Modul sicher, dass Integritätsregeln wie das Verhindern doppelter Follows oder das Löschen nicht existierender Beziehungen strikt eingehalten werden. Diese Prüfungen werden innerhalb der Anwendungsschicht implementiert und verhindern inkonsistente Zustände im System.

Durch seine klare Struktur und die strikte Trennung von Controller, Service, Domain und Repository folgt das **Follower-Modul** den Prinzipien der Hexagonalen Architektur. Es ist vollständig entkoppelt, leicht testbar und bietet die notwendige Flexibilität für langfristige Erweiterungen und Wartung.

4.3.2.6 List Modul von Umut und Yusuf

Die drei Module **List**, **BooksInList** und **ListFollow** bilden gemeinsam ein integriertes Subsystem für die Organisation, Verwaltung und soziale Interaktion rund um benutzerdefinierte Buchlisten innerhalb der Anwendung. Sie sind logisch miteinander verknüpft und folgen einem einheitlichen architektonischen Aufbau gemäß den Prinzipien der Hexagonalen Architektur.

Das **List-Modul** ermöglicht es Benutzern, persönliche Buchlisten zu erstellen und zu verwalten, wie etwa Wunschlisten, Leselisten oder thematische Sammlungen. Die HTTP-Endpunkte für das Anlegen, Umbenennen, Löschen und Anzeigen von Listen werden durch den **ListController** bereitgestellt. Jede Liste ist eindeutig einem Benutzer zugeordnet. Die Geschäftslogik liegt in der Anwendungsschicht, wo die Klasse **ListUseCases** die Validierung der Listennamen, die Sicherstellung der Eindeutigkeit pro Benutzer, die Zuordnung zum Ersteller und die zeitliche Markierung von Erstellungs- und Änderungsdatum übernimmt. Die Persistenz erfolgt über das Interface **IListRepository**, das von einem Spring-Data-Adapter implementiert wird. Das zentrale Domain-Modell **List** umfasst Eigenschaften wie Titel, Sichtbarkeit (öffentlich oder privat) und die Zuordnung zum Eigentümer.

Das **BooksInList-Modul** dient der Verknüpfung von Büchern mit Benutzerlisten und verwaltet viele-zu-viele-Beziehungen zwischen beiden. Ein Buch kann in mehreren Listen vorkommen, und eine Liste kann zahlreiche Bücher enthalten. Der **BooksInListController** stellt Endpunkte bereit, über die Benutzer Bücher zu ihren Listen hinzufügen oder daraus entfernen können. Zudem können Inhalte einer Liste abgefragt oder nach Kriterien wie Genre oder Status gefiltert werden. Die Klasse **BooksInListUseCases** in der Anwendungsschicht übernimmt die Validierung, stellt sicher, dass ein Buch nicht mehrfach in derselben Liste enthalten ist und nur der Eigentümer der Liste Änderungen vornehmen kann, und gewährleistet die Konsistenz der Änderungen im Gesamtsystem. Die Verknüpfungen werden als eigenständige Domain-Objekte gespeichert, oft mit einer zusammengesetzten ID aus Buch- und Listen-ID. Die Persistenz wird über **IBooksInListRepository** realisiert, das durch einen spezifischen JPA-Adapter umgesetzt wird.

Das **ListFollow-Modul** erweitert das System um eine soziale Komponente, indem es Benutzern erlaubt, öffentlich geteilten Buchlisten anderer Nutzer zu folgen.

Der **ListFollowController** bietet typische Endpunkte wie das Folgen und Entfolgen von Listen, das Abrufen aller Listen, denen ein Benutzer folgt, sowie die Abfrage der Follower-Anzahl einer bestimmten Liste. Die Geschäftslogik wird in der Klasse **ListFollowUseCases** umgesetzt, die das Interface **IListFollowService** implementiert. Hierbei wird sicher gestellt, dass nur öffentliche Listen gefolgt werden können, Benutzer sich nicht selbst

folgen und doppelte Follow-Beziehungen verhindert werden. Die entsprechenden Daten werden über das Interface `IListFollowRepository` verwaltet, das ebenfalls auf einer JPA-basierten Implementierung beruht.

Diese drei Module greifen eng ineinander und bilden ein kohärentes Subsystem: Das `List`-Modul stellt die Basisstruktur bereit, `BooksInList` verwaltet die dynamischen Inhalte dieser Listen, und `ListFollow` ergänzt das Ganze um eine soziale Dimension, indem Nutzer fremden Listen folgen und Änderungen automatisch nachvollziehen können, etwa durch Benachrichtigungen oder Feed-Aktualisierungen. Dank der klaren Trennung der Verantwortlichkeiten bleibt dieses Subsystem flexibel, testbar und langfristig wartbar, während es gleichzeitig ein konsistentes und nahtloses Nutzererlebnis innerhalb der Plattform sicherstellt.

4.3.3 Gemeinsame Komponenten

Im Rahmen des Projekts wurden mehrere gemeinsam genutzte Komponenten entwickelt, um die Wiederverwendbarkeit, Wartbarkeit und Konsistenz der Benutzeroberfläche zu gewährleisten. Zu diesen gemeinsamen Komponenten zählen unter anderem generische Schaltflächen, die in unterschiedlichen Varianten, Größen und mit Icons verwendet werden können. Sie kommen in verschiedenen Bereichen der Anwendung zum Einsatz, beispielsweise in Formularen, Navigationsleisten oder Dialogfenstern.

Ebenfalls wurde ein flexibles Eingabefeld entwickelt, das Validierungen und Fehlermeldungen unterstützt und somit für alle Formularbereiche geeignet ist. Darüber hinaus existieren wiederverwendbare Layout-Komponenten wie ein Header und ein Footer, die Navigationselemente und Branding-Elemente enthalten und in der gesamten Anwendung einheitlich verwendet werden.

Ergänzt wird diese Sammlung durch eine Modal-Komponente zur Anzeige von Dialogfenstern sowie durch einen Ladeindikator, der bei asynchronen Prozessen den aktuellen Status visualisiert.

tabularx

4.4 Qualitätssicherung des Gesamtsystems von Taha

Die Qualitätssicherung (QS) stellt im Entwicklungsprozess der BookApp einen essenziellen Bestandteil dar, um eine stabile, sichere und funktionale Anwendung sicherzustellen. Dabei kommen verschiedene Testmethoden und -ebenen zum Einsatz, die unterschiedliche Komponenten und Schichten des Systems abdecken. Ziel ist es, potenzielle Fehlerquellen frühzeitig zu identifizieren, die Wartbarkeit langfristig zu gewährleisten und die Grundlage für zukünftige Erweiterungen zu schaffen.

Ein bewährtes Prinzip, das dieser Strategie zugrunde liegt, ist das Test-Pyramiden-Modell. Dieses beschreibt die ideale Struktur automatisierter Tests: Die Basis bilden viele Unit-Tests, darauf folgen weniger, aber umfangreichere Integrations- und Persistenztests, und an der Spitze stehen wenige, aber systemübergreifende End-to-End-Tests. Dieser Aufbau ermöglicht es, Fehler möglichst früh und isoliert zu erkennen und gleichzeitig systemweite Stabilität zu gewährleisten.

Durch die konsequente Umsetzung dieser Testarchitektur kann sichergestellt werden, dass neue Funktionalitäten risikofrei integriert, bestehende Sicherheitsmechanismen zuverlässig kontrolliert und Engpässe oder Integrationsprobleme im Zusammenspiel mehrerer Module frühzeitig erkannt werden.

Um diese Testarchitektur effektiv umzusetzen, wurden eine Reihe moderner Java- und Spring-Testing-Tools eingesetzt. Dazu gehören: Um diese Testarchitektur effektiv umzusetzen, wurden verschiedene moderne Java- und Spring-Testing-Tools verwendet, die unterschiedliche Aspekte des Testens abdecken. Das **Spring Boot Test-Framework** stellt mit Annotationen wie `@SpringBootTest` und `@AutoConfigureMockMvc` einen vollständigen Anwendungskontext für Integrationstests bereit, wodurch realistische Testumgebungen geschaffen werden. Für die Integration mit JUnit 5 kommen Annotationen wie `@ExtendWith` und `@Transactional` – zur Sicherstellung konsistenter Datenbankzustände nach jedem Test – sowie `@TestInstance` und `@ActiveProfiles` für flexible Testkonfigurationen zum Einsatz. Zur Isolierung und Erstellung von Testobjekten in Unit-Tests wird **Mockito** verwendet, unterstützt durch die Annotationen `@Mock` und `@InjectMocks`. Darüber hinaus ermöglicht `MockMvc` die Simulation von HTTP-Anfragen im Web-Layer, während `@Autowired` die automatische Abhängigkeitsinjektion sicherstellt. Schließlich sorgen Annotationen wie `@Test`, `@BeforeEach` und `@TestOrder` für eine kontrollierte Ausführung und Konfiguration der Testabläufe, sodass alle Tests reproduzierbar und nachvollziehbar bleiben.

4.4.1 Überblick über die Testebenen in BookApp

Die BookApp ist in verschiedene Module und Schichten gegliedert, deren Qualität sowohl isoliert als auch im Zusammenspiel getestet wird. Die folgende Übersicht zeigt die eingesetzten Testarten, ihre Schwerpunkte und beispielhafte Testklassen:

Testtyp	Fokus	Beispielklasse	Vorteile
Unit-Test	Einzelne Komponenten isoliert	Mapper, UseCases	Schnelle Fehlererkennung bei geringem Aufwand und niedriger Komplexität.
Integrationstest	Zusammenspiel mehrerer Komponenten	UserServiceIntegrationTest	Frühes Aufdecken von Integrationsproblemen und Schnittstellenfehlern.
Persistenz-Mapping	Datenumwandlung Domain Entity	UserPersistenceMapperTest	Sicherstellung der Datenintegrität und Konsistenz bei Speicherung und Abruf.
Sicherheitstest	Zugriffsrechte, Autorisierungslogik	CommentSecurityTest	Schutz vor unbefugtem Zugriff und Validierung der Rechteverwaltung.
Use-Case Test	Geschäftslogik inklusive Sicherheit	UserUseCasesTest	Überprüfung komplexer Abläufe und Einhaltung fachlicher Regeln.
Web-Layer Test	DTO-Mapping, Web-Security	UserWebMapperTest	Gewährleistung sicherer Schnittstellen und korrekter Datenübergaben.

Diese breite Abdeckung sorgt dafür, dass sowohl technische als auch fachliche Anforderungen abgesichert sind – eine zentrale Voraussetzung für eine skalierbare und wartbare Softwarelösung.

4.4.2 Integrationstests am Beispiel UserServiceIntegrationTest

Integrationstests prüfen, ob verschiedene Komponenten in einem realistischen Anwendungsumfeld korrekt zusammenspielen. Im Beispiel `UserServiceIntegrationTest` wird getestet, ob Service, Repository und Mapper gemeinsam funktionsfähig sind. Diese Tests laufen im Spring Boot Testkontext, inklusive Datenbanksimulation.

```
1 @BeforeEach
2 void setUp() {
3     userService.deleteAll();
4 }
5
6 @Test
7 void testCreateAndFetchUser() {
8     User user = new User();
9     user.setUsername("tester");
10    user.setPassword("1234");
11    userService.create(user);
12
13    Optional<User> result = userService.getById(user.getId());
14    assertTrue(result.isPresent());
15    assertEquals("tester", result.get().getUsername());
16 }
```

Solche Tests tragen wesentlich dazu bei, Integrationsfehler frühzeitig aufzudecken, insbesondere wenn neue Abhängigkeiten oder Schnittstellen eingeführt werden.

4.4.3 Persistenz-Mapping Tests: Validierung der Datenkonvertierung

Persistenztests prüfen, ob Objekte korrekt zwischen Datenbank-Entities und Domain-Modellen konvertiert werden. Dadurch wird gewährleistet, dass gespeicherte Informationen vollständig und unverfälscht verarbeitet werden.

```
1 @Test
2 void testToEntity() {
3     User user = new User();
4     user.setId(1L);
5     user.setUsername("user1");
6
7     UserEntity entity = mapper.toEntity(user);
8     assertEquals("user1", entity.getUsername());
```

```
9 }
```

Diese Absicherung ist besonders im Hinblick auf langfristige Datenintegrität und Konsistenz von entscheidender Bedeutung.

4.4.4 Sicherheitstests mit CommentSecurityTest

Sicherheitstests validieren, ob Zugriffsrechte und Sicherheitslogiken wie z.B. Eigentümerprüfungen korrekt umgesetzt wurden. Am Beispiel des `CommentSecurityTest` wird geprüft, ob nur der Autor eines Kommentars diesen bearbeiten darf.

```
1 @Test
2 void isOwner_shouldReturnTrue() {
3     when(comment.getUser()).thenReturn(user);
4     when(user.getId()).thenReturn(1L);
5     when(authenticatedUser.getId()).thenReturn(1L);
6
7     assertTrue(commentSecurity.isOwner(comment));
8 }
```

Solche Tests sind zentral, um Angriffsvektoren wie Rechteeskalationen oder unautorisierte Datenänderungen zuverlässig zu verhindern.

4.4.5 Use-Case Tests am Beispiel UserUseCasesTest

Use-Case-Tests testen Geschäftslogik im realistischen Kontext inklusive Sicherheitsprüfungen. Beispiel: Nur berechtigte Benutzer dürfen ihren Account löschen.

```
1 @Test
2 void testDelete_accessDenied() {
3     mockSecurityContextWithUserId(2L);
4     AccessDeniedException ex =
5         assertThrows(AccessDeniedException.class, () ->
6             userUseCases.delete(1L));
7     assertEquals("You are not allowed to modify this user.",
8                 ex.getMessage());
9 }
```

Durch diese Tests wird sichergestellt, dass Sicherheitsregeln auch im Anwendungskern strikt durchgesetzt werden – unabhängig von der Benutzerschnittstelle.

4.4.6 Web-Layer Tests: UserWebMapperTest

Diese Tests prüfen die Umwandlung zwischen externen Web-DTOs und internen Domänenobjekten sowie Sicherheitsaspekte wie z.B. das Passwort-Hashing.

```

1  @Test
2  void toDomain_shouldEncodePassword() {
3      UserRegisterRequestDTO dto = new UserRegisterRequestDTO();
4      dto.setPassword("plainPassword");
5
6      User user = mapper.toDomain(dto);
7      assertTrue(passwordEncoder.matches("plainPassword",
8          user.getPassword()));
9 }

```

Gerade bei öffentlich zugänglichen Schnittstellen ist dies entscheidend, um ein sicheres und konsistentes Datenhandling zu garantieren.

4.4.7 Praxisbeispiel: Sicherheitsprüfung beim Benutzerlöschen

Ein klassischer Anwendungsfall, bei dem Sicherheitslogik über automatisierte Tests abgesichert wird, ist das Löschen eines Benutzerkontos.

Erlaubter Zugriff:

```

1  @Test
2  void testDelete_success() {
3      mockSecurityContextWithUserId(1L);
4      when(userRepository.findById(1L)).thenReturn(Optional.of(new
5          User()));
6
7      userUseCases.delete(1L);
8
9      verify(userRepository).deleteById(1L);
}

```

Verweigerter Zugriff:

```

1  @Test
2  void testDelete_accessDenied() {
3      mockSecurityContextWithUserId(2L);
4
5      assertThrows(AccessDeniedException.class, () ->
6          userUseCases.delete(1L));
}

```

Solche Tests leisten einen konkreten Beitrag zur Absicherung der Benutzerrechte und verhindern Datenmanipulation durch Dritte.

Die konsequente Anwendung dieser vielschichtigen Teststrategie stärkt nicht nur die Qualität und Sicherheit der aktuellen BookApp-Version, sondern bildet auch eine robuste Grundlage für zukünftige Erweiterungen, steigende Nutzerzahlen und erhöhte Sicherheitsanforderungen.

4.5 Übersicht Verzeichnisse und Dateien

4.5.1 Frontend

Die Projektstruktur des Frontends folgt den Prinzipien der **Clean Architecture** und ist in mehrere logisch getrennte Schichten unterteilt. Das Root-Package `com.example.frontendbook` enthält alle zentralen Komponenten der Anwendung und sorgt durch die klare Trennung der Verantwortlichkeiten für eine modulare, leicht wartbare und erweiterbare Struktur.

4.5.1.1 data Die `data`-Schicht umfasst sämtliche Datenquellen sowie deren Verarbeitung. Unter `data/api` befinden sich Klassen für die Kommunikation mit externen APIs, einschließlich `dto` (Data Transfer Objects), `mapper` (zur Umwandlung von DTOs in Domain-Modelle) und `service` (Netzwerkdienste wie Retrofit). Die Datei `RetrofitInstance.kt` stellt eine Singleton-Instanz für die API-Verbindung bereit. Darüber hinaus enthält der Ordner `data` weitere Pakete wie `model`, `remote` und `repository`, die für die Definition von Datenmodellen, den Zugriff auf entfernte Datenquellen und die Bereitstellung von Repositories verantwortlich sind. Diese Repositories bilden die Schnittstelle zwischen der Daten- und der Domain-Schicht.

4.5.1.2 di (Dependency Injection) Dieses Paket organisiert alle Klassen zur Bereitstellung und Verwaltung von Abhängigkeiten, typischerweise unter Verwendung von Frameworks wie Dagger oder Hilt. Dies ermöglicht eine modulare und flexible Initialisierung der Anwendungsbausteine.

4.5.1.3 domain Die `domain`-Schicht kapselt die Geschäftslogik der Anwendung und ist vollständig unabhängig von der Benutzeroberfläche oder den Datenquellen. Sie enthält drei Kernkomponenten: `model` für die zentralen Fachmodelle, `repository` als abstrakte Schnittstelle zu den Datenquellen und `usecase` zur Implementierung spezifischer Anwendungslogiken.

4.5.1.4 ui In der `ui`-Schicht befindet sich die Benutzeroberfläche der App, einschließlich Activities, Fragments und Compose-Views. Diese Schicht kommuniziert über ViewModels und UseCases mit der Domain-Schicht und stellt sicher, dass Logik und Präsentation strikt voneinander getrennt bleiben.

4.5.1.5 MyApplication.kt Die Datei `MyApplication.kt` dient als zentrale Einstiegsklasse der Android-Anwendung, in der globale Initialisierungen wie das Setup für Dependency Injection vorgenommen werden.

Durch diese Architektur wird eine klare Trennung der Verantwortlichkeiten erreicht, was nicht nur die Testbarkeit verbessert, sondern auch eine einfache Wartung und Erweiterung der Anwendung ermöglicht.

4.5.2 Backend

Die Projektstruktur des Backends folgt den Prinzipien der **Hexagonalen Architektur** und ist in klar getrennte Schichten gegliedert. Das Root-Package `com.bookapp.backend` enthält alle relevanten Module und Komponenten der Anwendung. Jede Schicht erfüllt eine spezifische Rolle, wodurch die Geschäftslogik unabhängig von technischen Details bleibt und das System leicht wartbar und erweiterbar ist.

4.5.2.1 domain Diese Schicht enthält die zentralen Geschäftsobjekte und Regeln der Anwendung. Unter `domain/model` sind alle Kernmodelle (z. B. `User`, `Book`, `Comment`, `List`, `Follower`) abgelegt, die direkt die Fachlogik repräsentieren. Diese Klassen sind bewusst frei von technischen Abhängigkeiten gehalten. Außerdem definiert die Domain unter `domain/ports` zwei Bereiche:

- `ports/in`: Enthält die Service-Interfaces (Inbound Ports), die die Anwendungslogik nach außen verfügbar machen und von Use-Case-Implementierungen genutzt werden.
- `ports/out`: Enthält die Repository-Interfaces (Outbound Ports), die den Zugriff auf Datenquellen abstrahieren. Diese Schnittstellen werden später von Adapters implementiert.

4.5.2.2 adapter/in (Web Layer) In diesem Paket sind die REST-Controller untergebracht (z. B. `AuthController`, `BookController`, `CommentController`). Sie nehmen eingehende HTTP-Anfragen entgegen, wandeln die empfangenen JSON-Daten mithilfe von DTOs und Mappern in Domain-Objekte um und leiten sie an die Use-Cases weiter. Außerdem sind hier HATEOAS-Links und Response-Strukturen definiert, um die API konsistent und erweiterbar zu gestalten.

4.5.2.3 adapter/out (Persistence Layer) Diese Schicht implementiert die Repository-Interfaces aus `ports/out`. Hier befinden sich die JPA-Entities (z. B. `UserEntity`, `BookEntity`) sowie die Spring-Data-Repositories, die den Datenbankzugriff kapseln. Auf diese Weise bleibt die Geschäftslogik vollständig unabhängig von der verwendeten Datenbanktechnologie oder der konkreten ORM-Implementierung.

4.5.2.4 application/config Dieses Paket enthält sämtliche systemweiten Konfigurationen und Sicherheitsmechanismen. Dazu gehören Klassen wie `SecurityConfig` (Spring Security, JWT-Filter, CORS-Einstellungen) und globale Exception-Handler. Es dient als zentrale Konfigurationsschicht, die sowohl den Web Layer als auch die Persistenz unterstützt, ohne die Geschäftslogik zu beeinflussen.

4.5.2.5 application/service Hier befinden sich die Implementierungen der Geschäftslogik in Form von Use-Case-Klassen (z. B. `AuthUseCases`, `BookUseCases`, `CommentUseCases`). Diese Klassen implementieren die Inbound Ports aus der Domain-Schicht und kapseln die Anwendungslogik vollständig. Sie steuern den Ablauf der Anfragen, validieren Eingaben, interagieren mit den Outbound Ports (z. B. `Repositories`) und stellen sicher, dass alle Geschäftsregeln eingehalten werden.

4.5.2.6 BookAppBackendApplication.java Die Hauptklasse des Backends, die den Einstiegspunkt der Spring-Boot-Anwendung darstellt und die Initialisierung des gesamten Systems steuert.

Diese Struktur sorgt für eine strikte Trennung von Domäne, Anwendung und Infrastruktur. Dadurch bleibt das Backend modular, testbar und flexibel, selbst wenn zukünftige Änderungen an Datenbanken, Frameworks oder externen Integrationen vorgenommen werden müssen.

4.6 Bauen des Gesamtsystems

Zur Ausführung des Gesamtsystems ist zunächst das zentrale Git-Repository unter folgendem Link zu klonen: <https://bitbucket.student.fiw.fhws.de:8443/scm/ppss25/gruppe-12—book-app.git>

Für eine vollständige Ausführung des Systems müssen zwei Branches separat ausgecheckt werden: der `main`-Branch für das Backend (Spring Boot) sowie der `frontendapp`-Branch für die Android-Anwendung, welche die Benutzeroberfläche sowie die Anbindung über Retrofit umfasst.

Nach dem erfolgreichen Klonen des Repositories ist das Backend mit Maven zu bauen. Dies geschieht durch Ausführen des Befehls `mvn clean install` im Wurzelverzeichnis des Backend-Projekts. Dadurch wird eine ausführbare JAR-Datei im `target`-Verzeichnis erzeugt. Um eine konsistente und plattformunabhängige Laufzeitumgebung sicherzustellen, erfolgt der Betrieb des Backends in einem Docker-Container. Das Projekt beinhaltet hierzu ein `Dockerfile` sowie eine vorkonfigurierte `docker-compose.yml`-Datei, in der auch die PostgreSQL-Datenbank als separater Container eingebunden ist.

Vor dem Start müssen Docker und Docker Compose lokal installiert sein. Anschließend kann das gesamte System mittels `docker-compose up -build` gestartet werden. Der Build-Vorgang erstellt dabei das Docker-Image für das Backend, startet den Container und sorgt für eine automatische Verbindung zur Datenbank. Die PostgreSQL-Instanz wird mit dem Datenbanknamen `bookapp`, dem Benutzer `postgres` und dem Passwort `postgres` initialisiert. Die Kommunikation zwischen Backend und Datenbank erfolgt intern über den Container-Namen `db`, welcher in der Umgebungsvariable `SPRING_DATASOURCE_URL` korrekt hinterlegt ist. Die Spring Boot-Anwendung lauscht nach dem Start standardmäßig auf Port 8080 und ist somit über `http://localhost:8080` erreichbar.

Die Android-Anwendung greift über Retrofit automatisch auf das lokal laufende Backend zu, ohne dass zusätzliche Konfigurationsanpassungen notwendig sind. Die Basis-URL ist in der Anwendung entsprechend auf `localhost:8080` gesetzt, wodurch eine sofortige Verbindung zum Backend beim Start der App erfolgt. Es ist zu beachten, dass bei der Entwicklung oder dem Testen auf realen Geräten innerhalb desselben Netzwerks ggf. die IP-Adresse des Hosts anstelle von `localhost` angegeben werden muss.

Zusammenfassend ermöglicht die Kombination aus Maven, Docker und Docker Compose einen automatisierten und reproduzierbaren Build- und Ausführungsprozess für das gesamte System. Die Android-App kann direkt mit dem lokal bereitgestellten Backend kommunizieren, wodurch eine effiziente Entwicklung und ein funktionaler Gesamtablauf gewährleistet sind.

5 Einrichtung und Betrieb der Software-Lösung

5.1 Abhängigkeiten zu anderen Software-Systemen

Die Anwendung ist für ihre Kernfunktionalität auf mehrere externe Systeme und Dienste angewiesen, die eine reibungslose Bereitstellung und den Betrieb ermöglichen. Im Mittelpunkt steht der **Backend-Service**, der über klar definierte REST-Endpunkte sämtliche Datenoperationen abwickelt – darunter die Verwaltung von Nutzern, Büchern, Kommentaren und personalisierten Listen. Alle Anfragen des Frontends werden über HTTPS an diesen Service gesendet, wobei ein **JWT-basiertes Authentifizierungssystem** den sicheren Zugriff auf geschützte Bereiche gewährleistet.

Die Persistenz der Daten wird durch eine **relationale Datenbank** (z. B. PostgreSQL) sichergestellt, die sämtliche Benutzerinformationen, Bücher und weiteren Inhalte speichert. Der Zugriff auf die Datenbank erfolgt ausschließlich über das Backend, wodurch eine konsistente Datenhaltung und ein hohes Maß an Sicherheit gewährleistet werden.

Für den Betrieb wird die gesamte Anwendung **containerisiert mit Docker** bereitge-

stellt. Diese Architektur ermöglicht eine plattformunabhängige Ausführung, ein standarisches Deployment sowie eine einfache Skalierbarkeit. Die Konfiguration erfolgt zentral über Umgebungsvariablen, welche API-Schlüssel, Datenbankzugänge und sicherheitsrelevante Parameter beinhalten. Eine **stabile Netzwerkverbindung** und die korrekte Einrichtung dieser Konfiguration sind essenziell, um einen unterbrechungsfreien Betrieb der Anwendung zu gewährleisten.

5.2 Verfügbarkeit der Software

Die Verfügbarkeit der Software hängt maßgeblich von der erfolgreichen Ausführung des Docker-Containers und der Erreichbarkeit externer Dienste ab. Solange der Container aktiv ist und keine Fehler im Build oder in der Konfiguration auftreten, steht die Anwendung lokal unter der definierten Portweiterleitung (standardmäßig: `http://localhost:8080`) zur Verfügung.

Für den **Produktivbetrieb** kann die Anwendung auf einem Server oder in einer Cloud-Umgebung gehostet werden. In diesem Kontext sind zusätzliche Maßnahmen wie **Lastverteilung (Load Balancing)**, **Monitoring**, **automatisierte Backups** und **Sicherheitsmechanismen** (z. B. TLS, Zugriffskontrollen) erforderlich, um eine hohe Verfügbarkeit und Stabilität sicherzustellen. Die Zuverlässigkeit ist außerdem davon abhängig, dass externe Dienste wie Google APIs und das Backend-System kontinuierlich erreichbar sind.

5.3 Installation der Software

Die Installation erfolgt nicht über ein klassisches Setup-Programm, sondern vollständig über die Einrichtung und Ausführung eines Docker-Containers. Voraussetzung ist eine funktionierende Docker-Installation auf dem Host-System.

Nach dem Klonen des Repositorys wird aus dem Quellcode ein Docker-Image erstellt, das **alle notwendigen Abhängigkeiten** enthält. Die Konfiguration erfolgt entweder über Umgebungsvariablen oder direkt über das Dockerfile. Anschließend wird das Image gestartet und der Container ausgeführt. Es sind keine weiteren Installationsschritte notwendig, da sämtliche Bibliotheken und Konfigurationen bereits im Image enthalten sind. Dies ermöglicht eine **einheitliche, wiederholbare und plattformunabhängige Installation** auf jedem System, das Docker unterstützt.

5.4 Inbetriebnahme und Betrieb

Die Inbetriebnahme beginnt mit dem erfolgreichen Build und Start des Docker-Containers. Sobald der Container aktiv ist, übernimmt er automatisch den Betrieb der Anwendung,

ohne dass manuelle Konfigurationsschritte erforderlich sind.

Dabei muss sichergestellt werden, dass alle **notwendigen Dienste** (z. B. Datenbanken, externe APIs) erreichbar sind. Nach dem Start ist die Anwendung sofort über den definierten Port verfügbar.

Für den **stabilen Dauerbetrieb** empfiehlt es sich, Mechanismen wie:

- Automatisierte Neustarts des Containers bei Fehlern oder Systemausfällen,
- Health Checks und Monitoring via Docker oder externen Tools,
- Optional die Einbindung in ein Orchestrierungssystem wie Kubernetes,

zu nutzen, um langfristige Zuverlässigkeit und Skalierbarkeit sicherzustellen.

5.5 Möglichkeiten zur späteren Anpassung und Weiterentwicklung

Die Architektur der Anwendung ist modular aufgebaut und erlaubt Erweiterungen ohne tiefgreifende Umstrukturierungen. Geplante und denkbare Erweiterungen sind unter anderem:

- **Integration von Künstlicher Intelligenz (KI):** Personalisierte Buchempfehlungen, Analyse des Nutzerverhaltens und adaptive Benutzeroberflächen (z. B. lernfähige Empfehlungssysteme).
- **Erweiterte Benutzeroberfläche:** Anzeige des Profilbilds des Nutzers auf allen relevanten Seiten zur Personalisierung und besseren Wiedererkennung.
- **Nachrichtensystem:** Direkte Kommunikation zwischen Nutzern (z. B. über Bücher, Rezensionen oder gemeinsame Interessen), inklusive Echtzeit-Funktionalitäten über **WebSockets**.
- **Benachrichtigungssystem:** Push-Benachrichtigungen bei neuen Nachrichten, Kommentaren oder Systemereignissen; ergänzend auch E-Mail-Benachrichtigungen oder In-App-Banner.
- **Upload-Funktion für Benutzer-Avatare:** Ermöglichung eines eigenständigen Hochladens von Profilbildern durch die Nutzer, inklusive Validierung und serverseitiger Speicherung. Dies unterstützt eine höhere Individualisierung des Nutzerprofils.

Diese geplanten Weiterentwicklungen verdeutlichen, dass die Anwendung eine **zukunftsähnliche und flexible Basis** bietet, die sich schrittweise erweitern lässt, ohne die bestehende Architektur grundlegend anzupassen.

6 Zusammenfassung und Ausblick

Bei dem vorliegenden Projekt handelt es sich um eine Android-Anwendung, die im Rahmen eines studentischen Softwareentwicklungsprojekts konzipiert und umgesetzt wurde. Die Applikation basiert auf der Programmiersprache Kotlin und folgt dem Prinzip der **Clean Architecture**, wodurch eine klare Trennung zwischen Daten-, Domänen- und Präsentationsschicht sichergestellt wird. Diese Strukturierung des Quellcodes ermöglicht eine hohe Wartbarkeit sowie eine einfache Erweiterbarkeit der Anwendung.

Die Software wird containerisiert mithilfe von **Docker** betrieben, was eine plattformunabhängige Ausführung und ein standardisiertes Deployment ermöglicht. Alle benötigten Abhängigkeiten werden bereits im Docker-Image gebündelt, sodass keine zusätzliche Konfiguration auf dem Zielsystem erforderlich ist. Der gesamte Build- und Ausführungsprozess lässt sich sowohl über die Kommandozeile als auch über grafische Werkzeuge wie Android Studio durchführen.

Zur Laufzeit greift die Anwendung auf externe Systeme zurück, insbesondere auf die **Books API**, die als Datenquelle für Buchinformationen dient. Darüber hinaus kommuniziert die App mit einem separaten Backend-Service, über den Nutzerdaten, Bücher, Kommentare und weitere Inhalte verwaltet und synchronisiert werden.

Im aktuellen Entwicklungsstand stellt die Anwendung bereits eine funktionale Grundlage für eine digitale Buchplattform bereit – inklusive Benutzeroberfläche, Datenmodellen, API-Anbindung und einem modular aufgebauten Codegerüst. Die Architektur ist so gestaltet, dass zukünftige Funktionserweiterungen – wie beispielsweise ein Nachrichtensystem oder die Integration von Benachrichtigungen – ohne tiefgreifende Änderungen umsetzbar sind.

Für die nächste Projektphase ist die Umsetzung weiterer Features geplant, die die Benutzerinteraktion und Personalisierung verbessern sollen. Dazu zählen unter anderem die Anzeige von Profilbildern auf allen relevanten Seiten, die Einführung eines Chat-Systems sowie die Implementierung verschiedener Benachrichtigungsmechanismen. Langfristig ist auch der Einsatz **künstlicher Intelligenz (KI)** zur Analyse des Nutzerverhaltens und zur Generierung personalisierter Inhalte vorgesehen.

Insgesamt stellt die Anwendung eine solide technologische Basis dar, die sich sowohl für den praktischen Einsatz als auch für akademische Weiterentwicklungen eignet. Durch die saubere Trennung der Komponenten und den Einsatz moderner Technologien ist das System zukunftsfähig und flexibel erweiterbar.

6.1 UML Diagrams von Backend

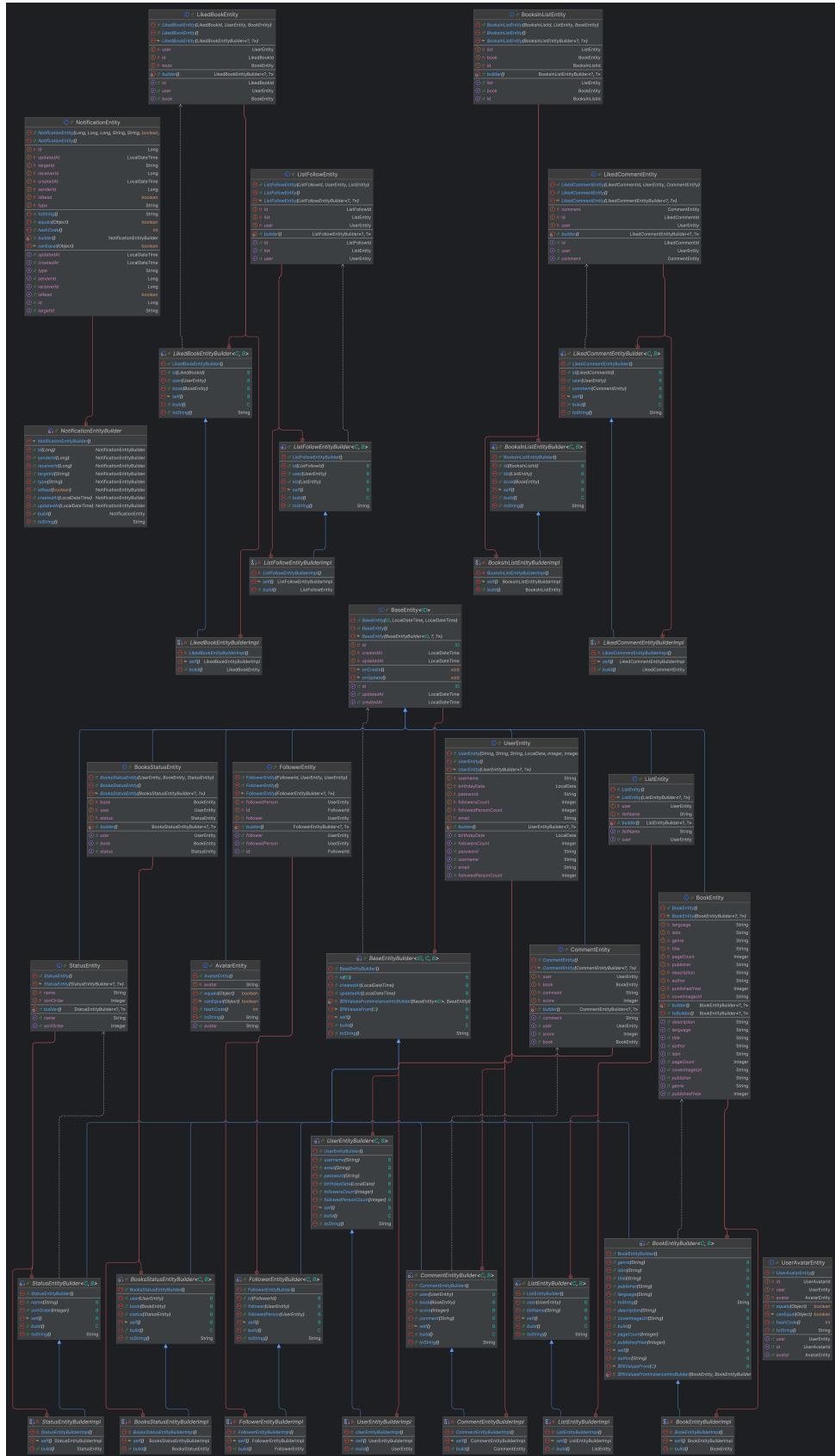


Abbildung 6: Entity UML Diagramm

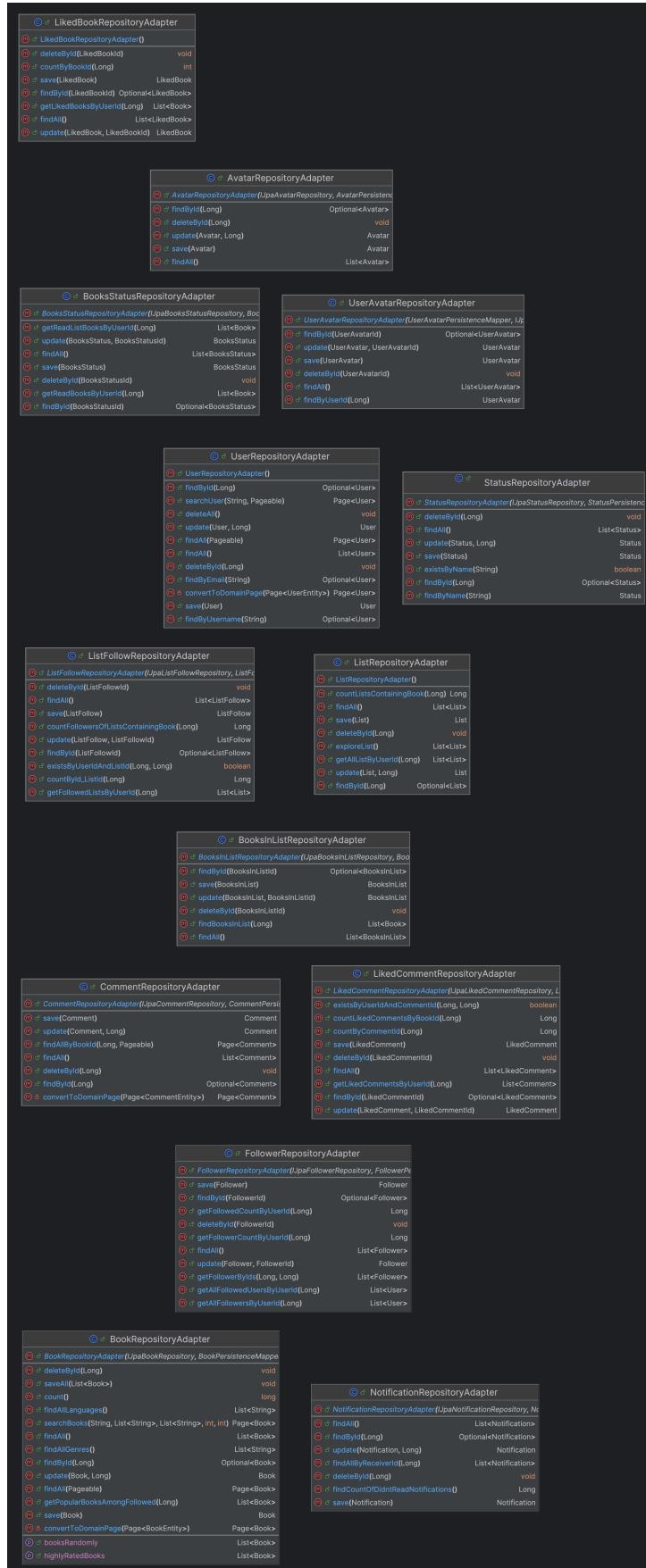


Abbildung 7: Adapter UML Diagramm

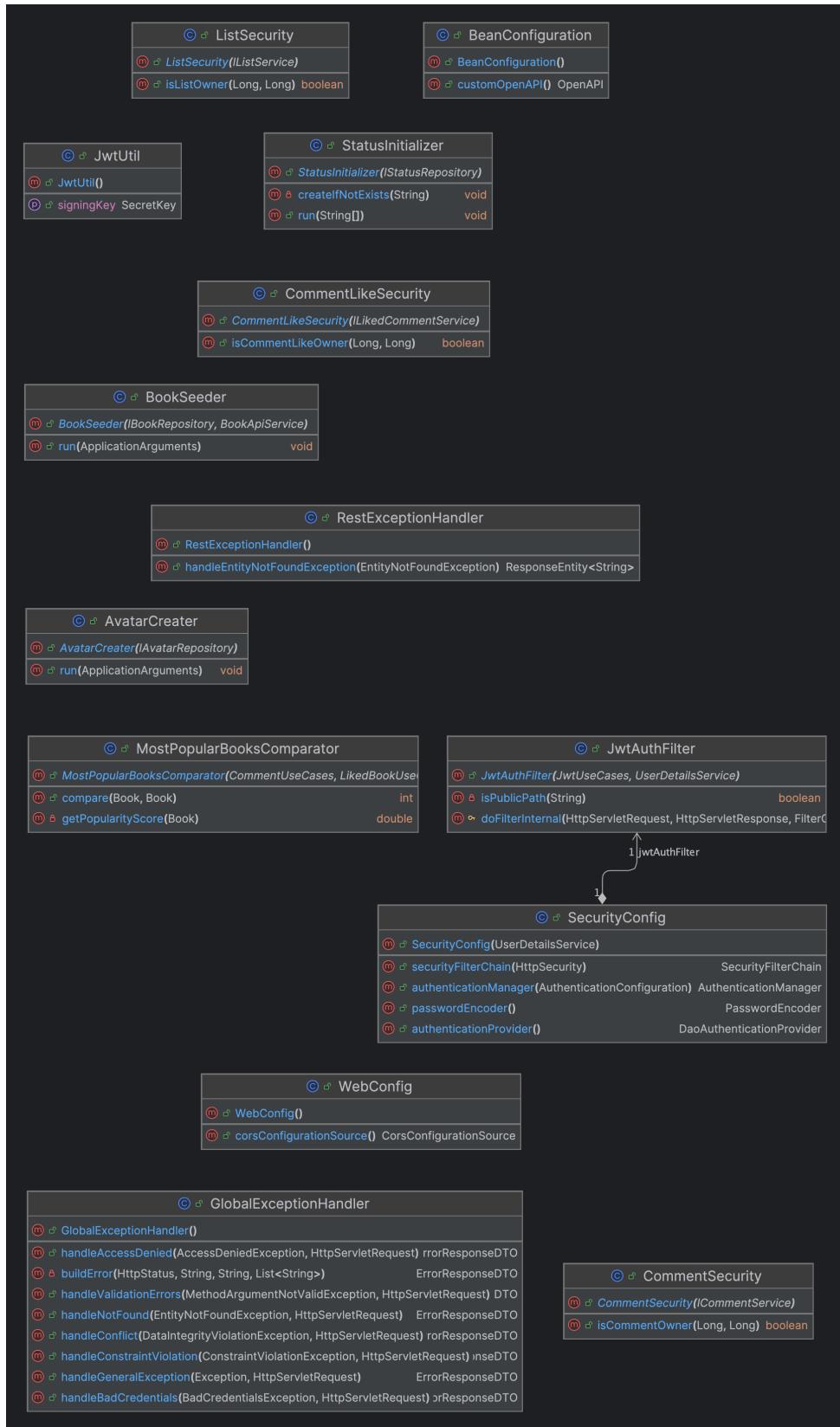


Abbildung 8: Config UML Diagramm

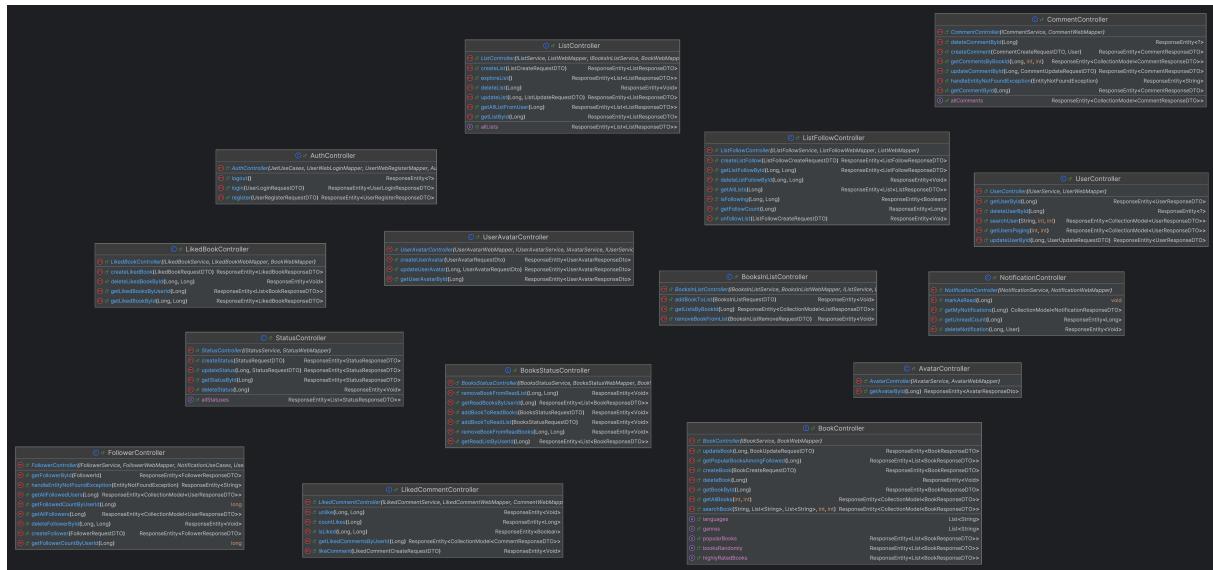


Abbildung 9: Controller UML Diagramm

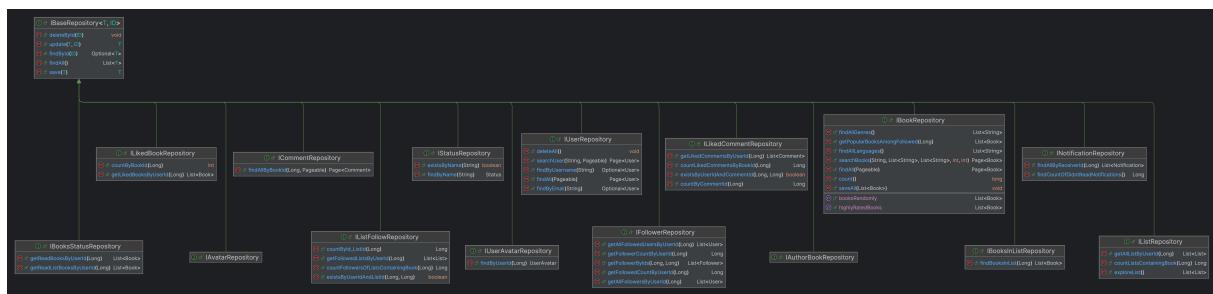


Abbildung 10: IRepository UML Diagramm

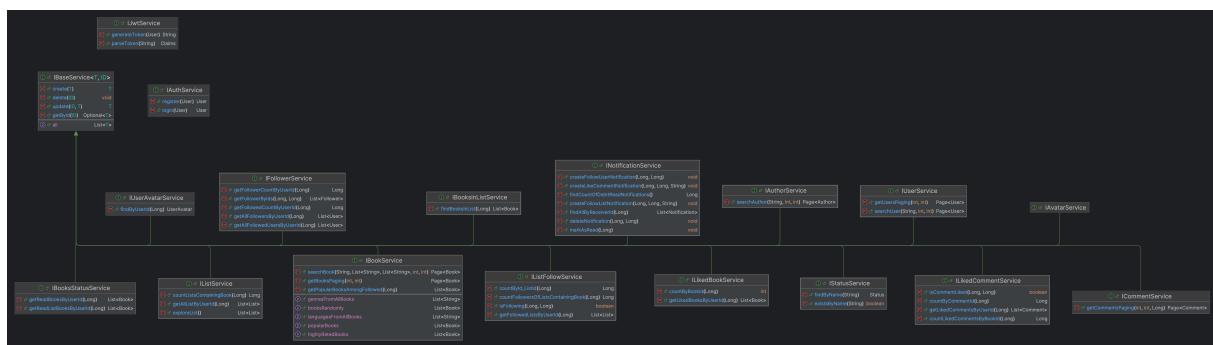


Abbildung 11: IService UML Diagramm

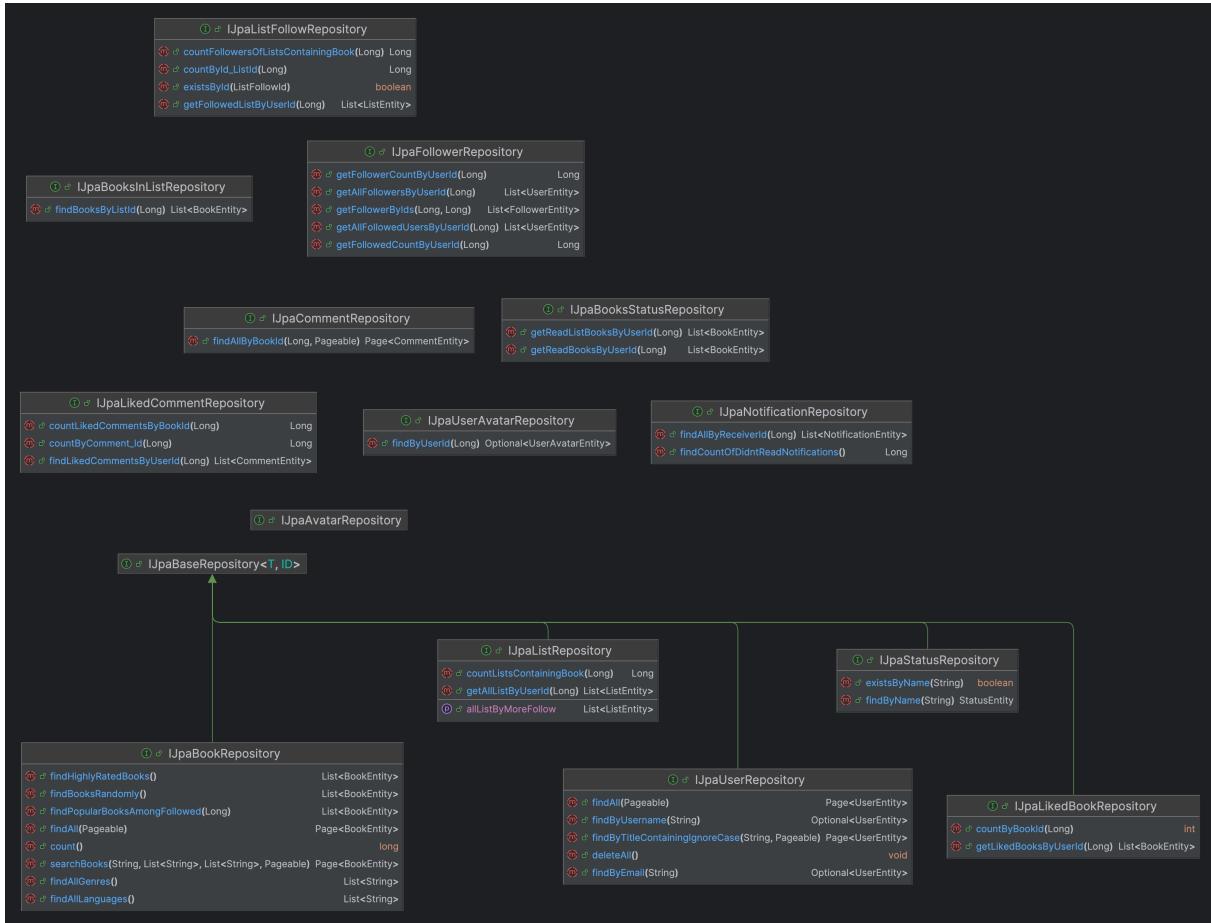


Abbildung 12: JPARepository UML Diagramm

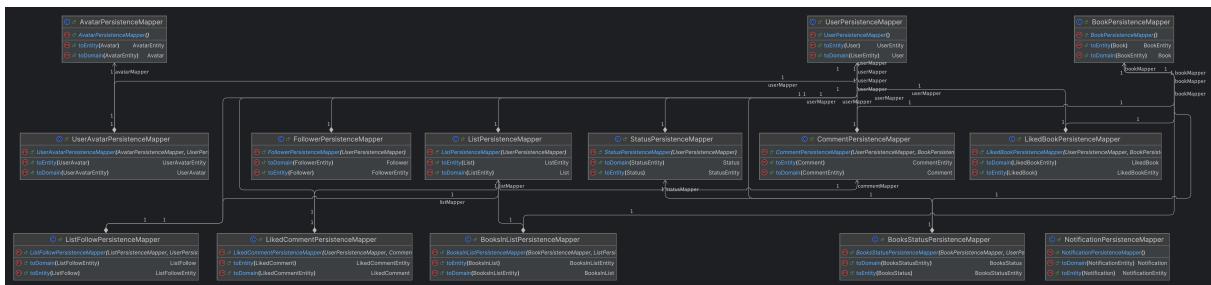


Abbildung 13: Persistence Mapper UML Diagramm

6.2 Sequence Diagrams von Backend

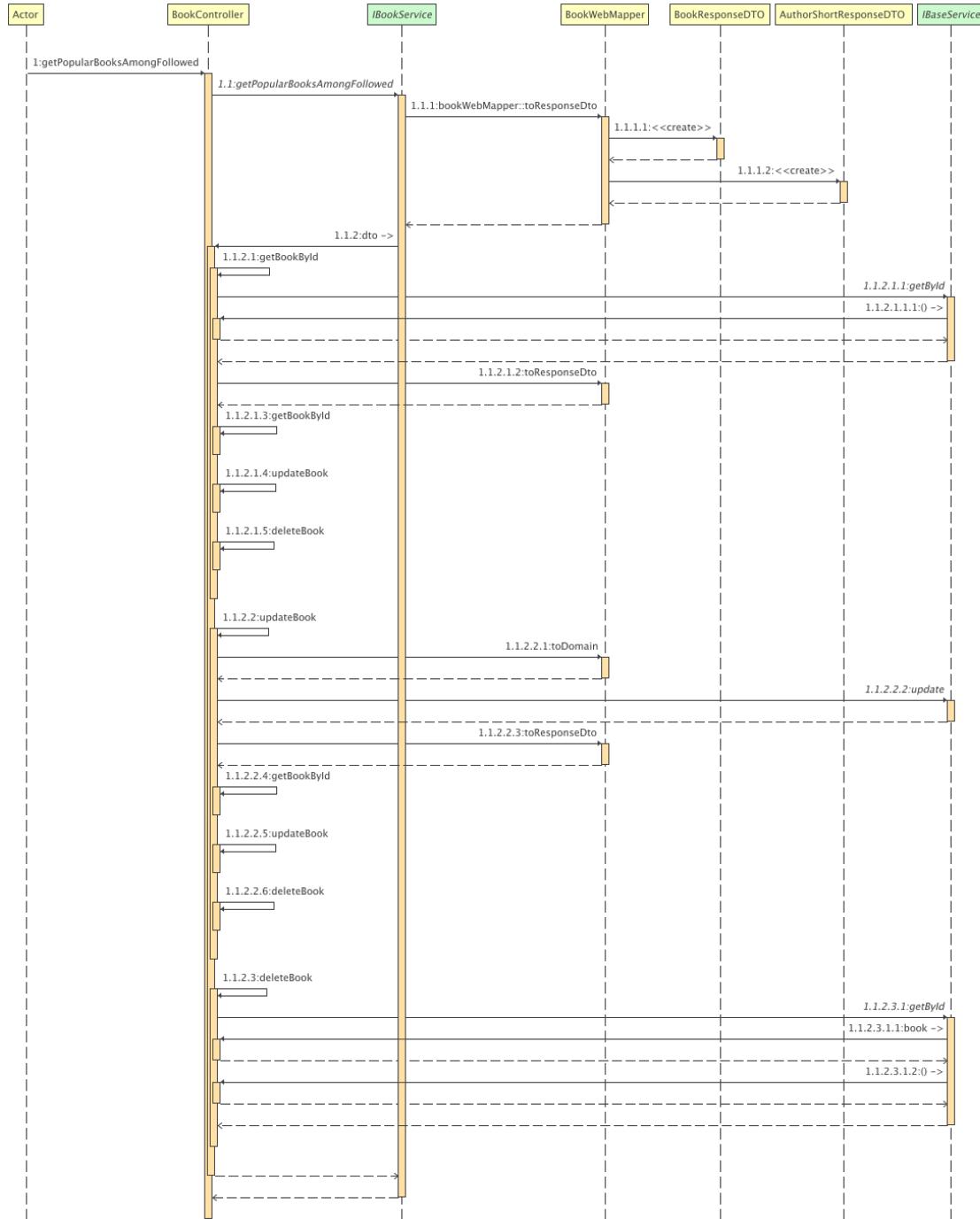


Abbildung 14: Get Popular Books Among Followed Users Sequence Diagramm

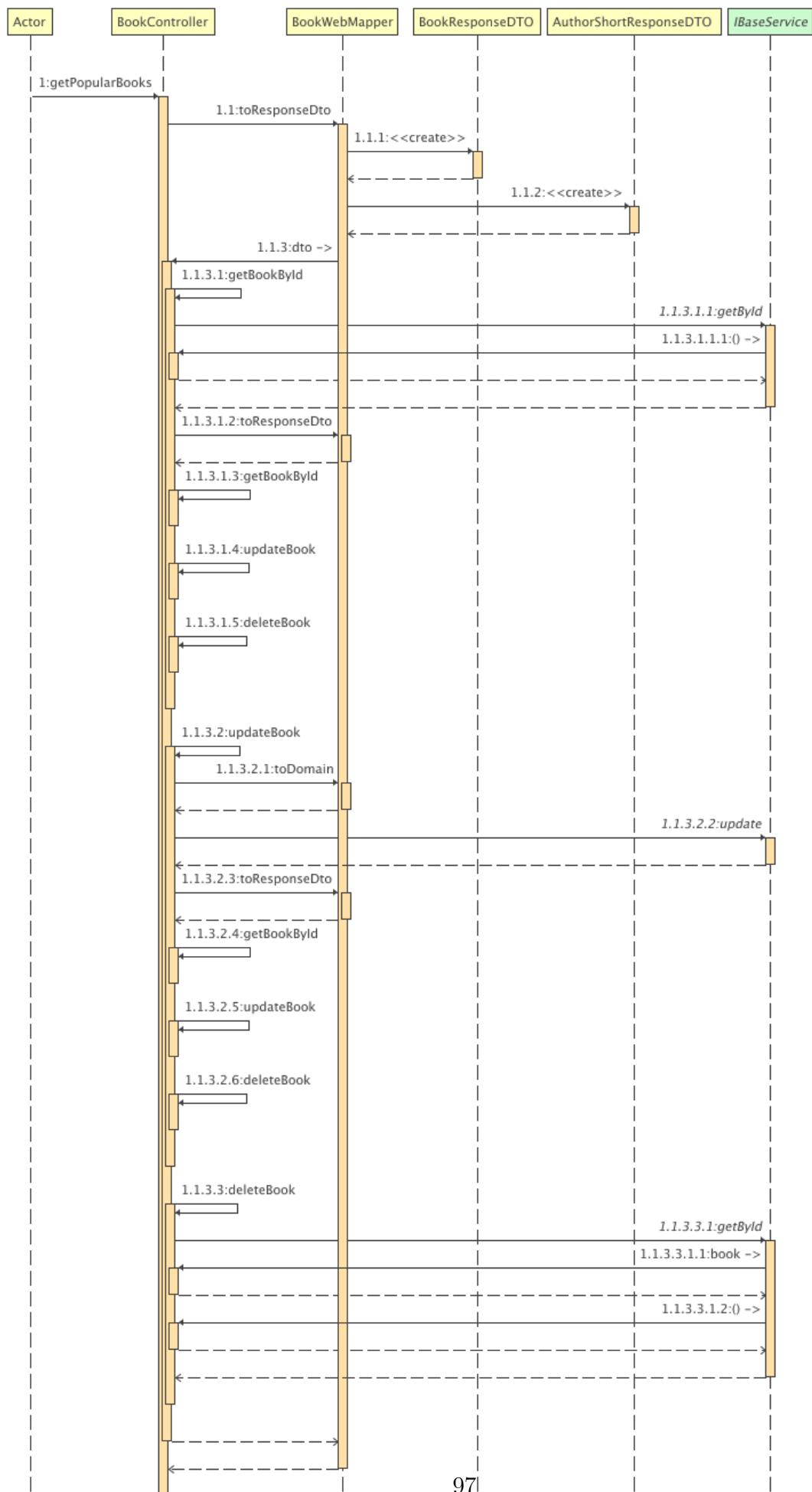


Abbildung 15: Get Popular Books Sequence Diagramm

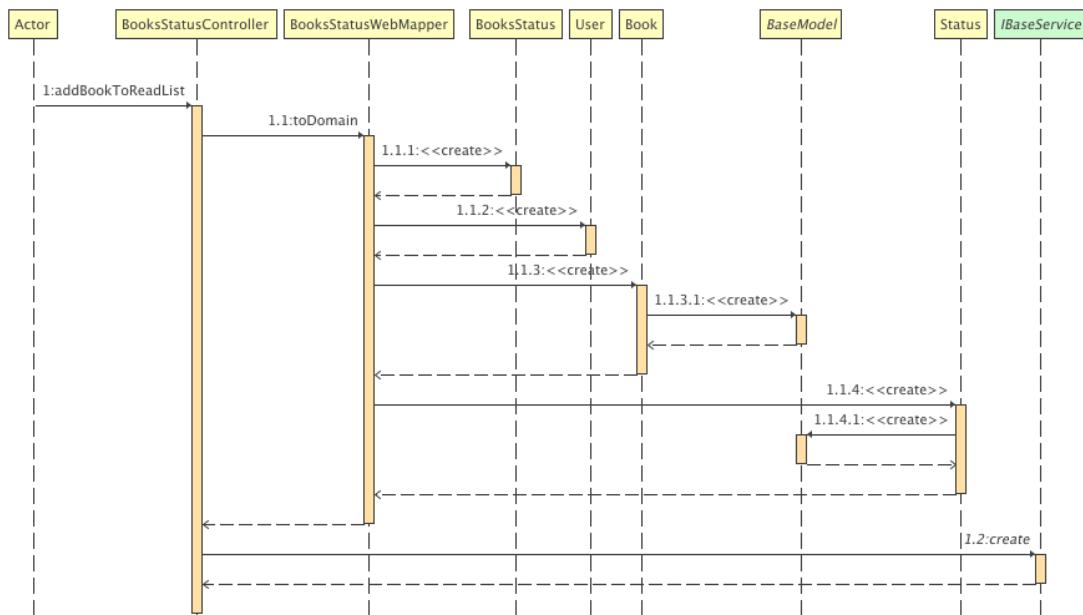


Abbildung 16: Add Book to Readlist sequence diagramm

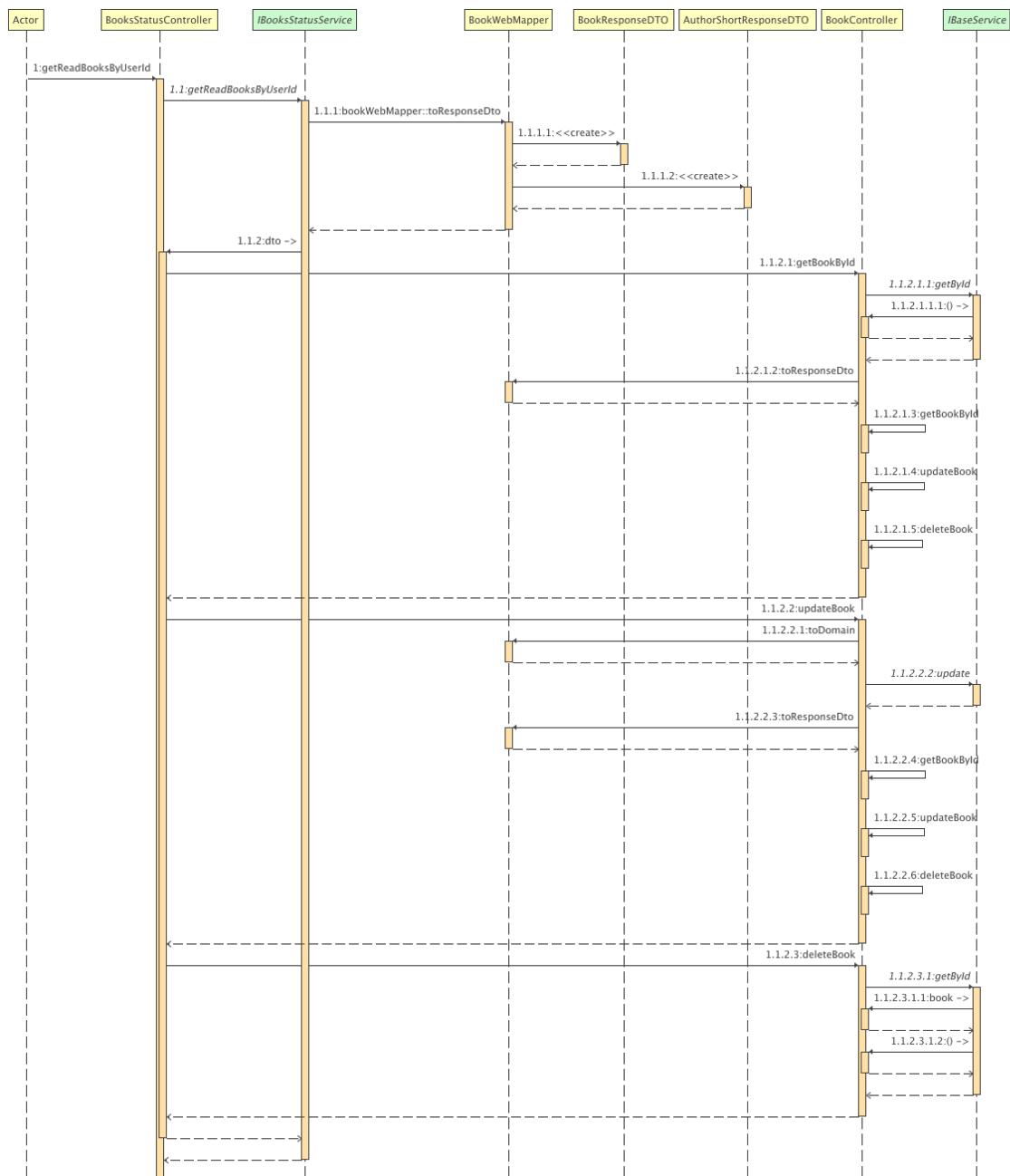


Abbildung 17: Get Readlist by User Sequence Diagramm

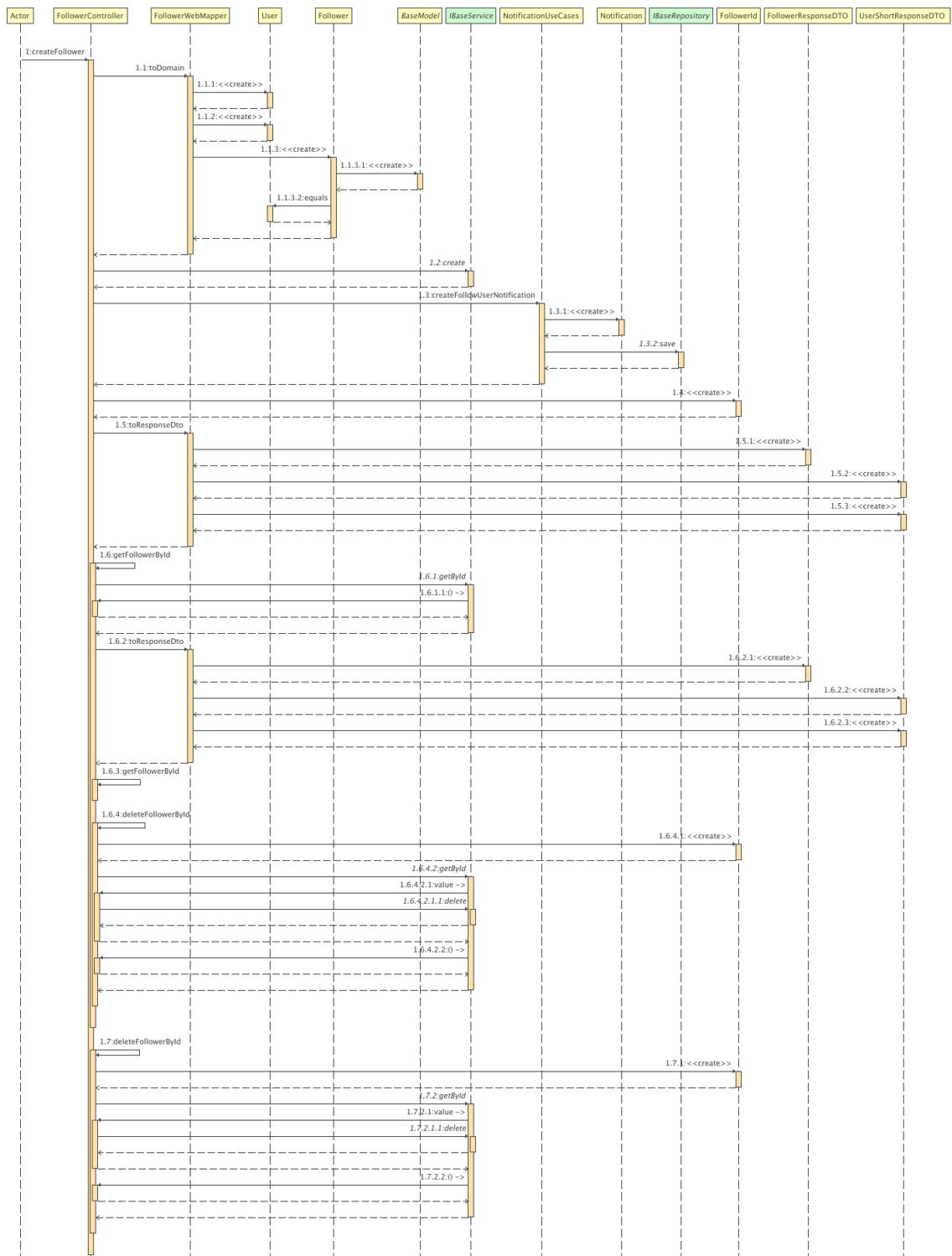


Abbildung 18: Create Follow Relationship Sequence Diagramm

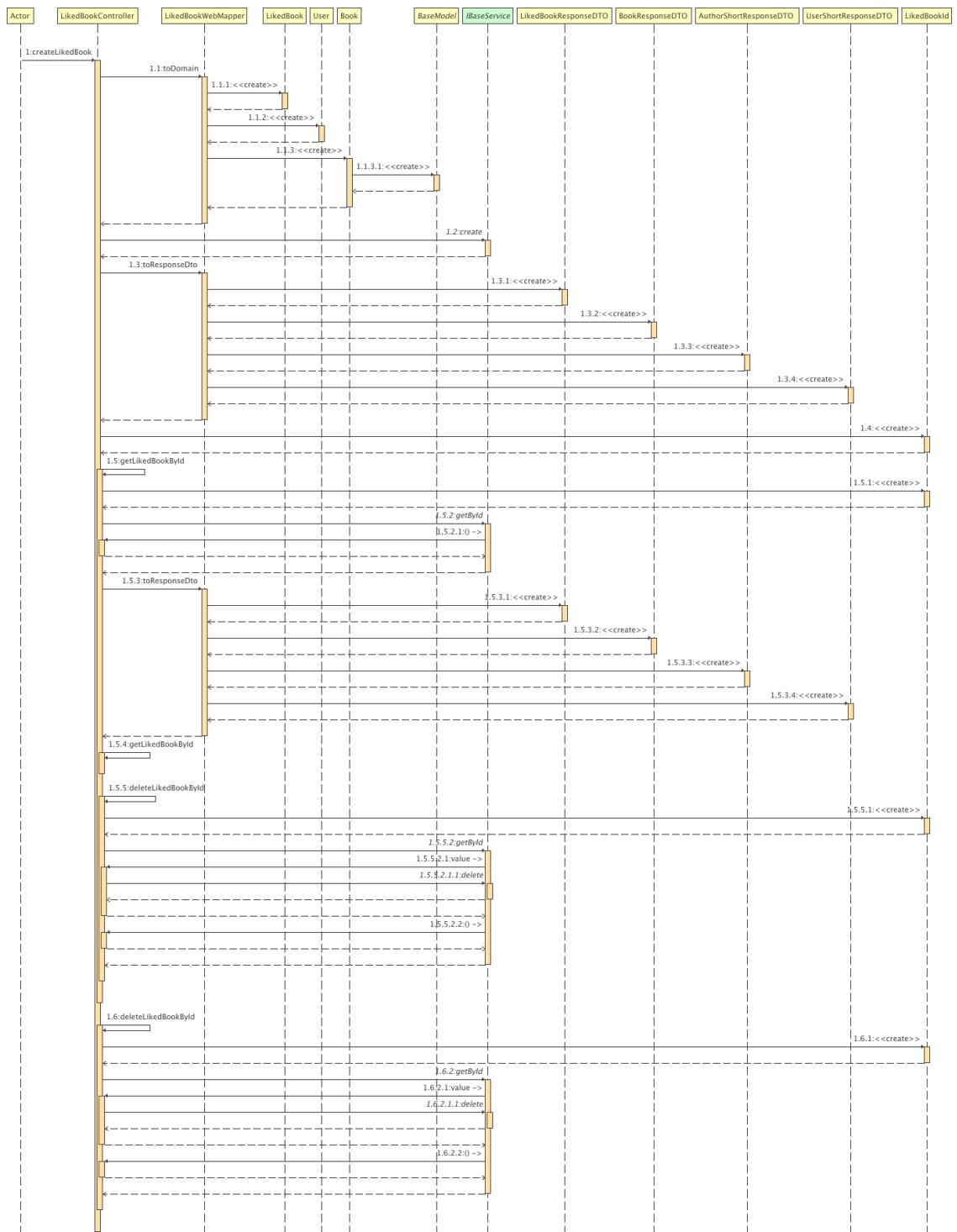


Abbildung 19: Like Book Sequence Diagramm

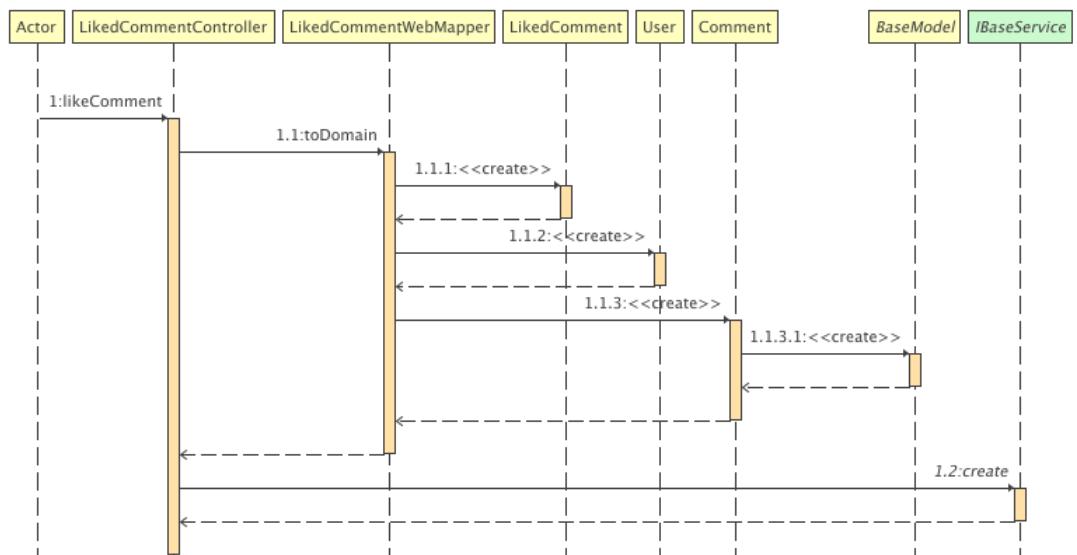


Abbildung 20: Like Comment Sequence Diagramm

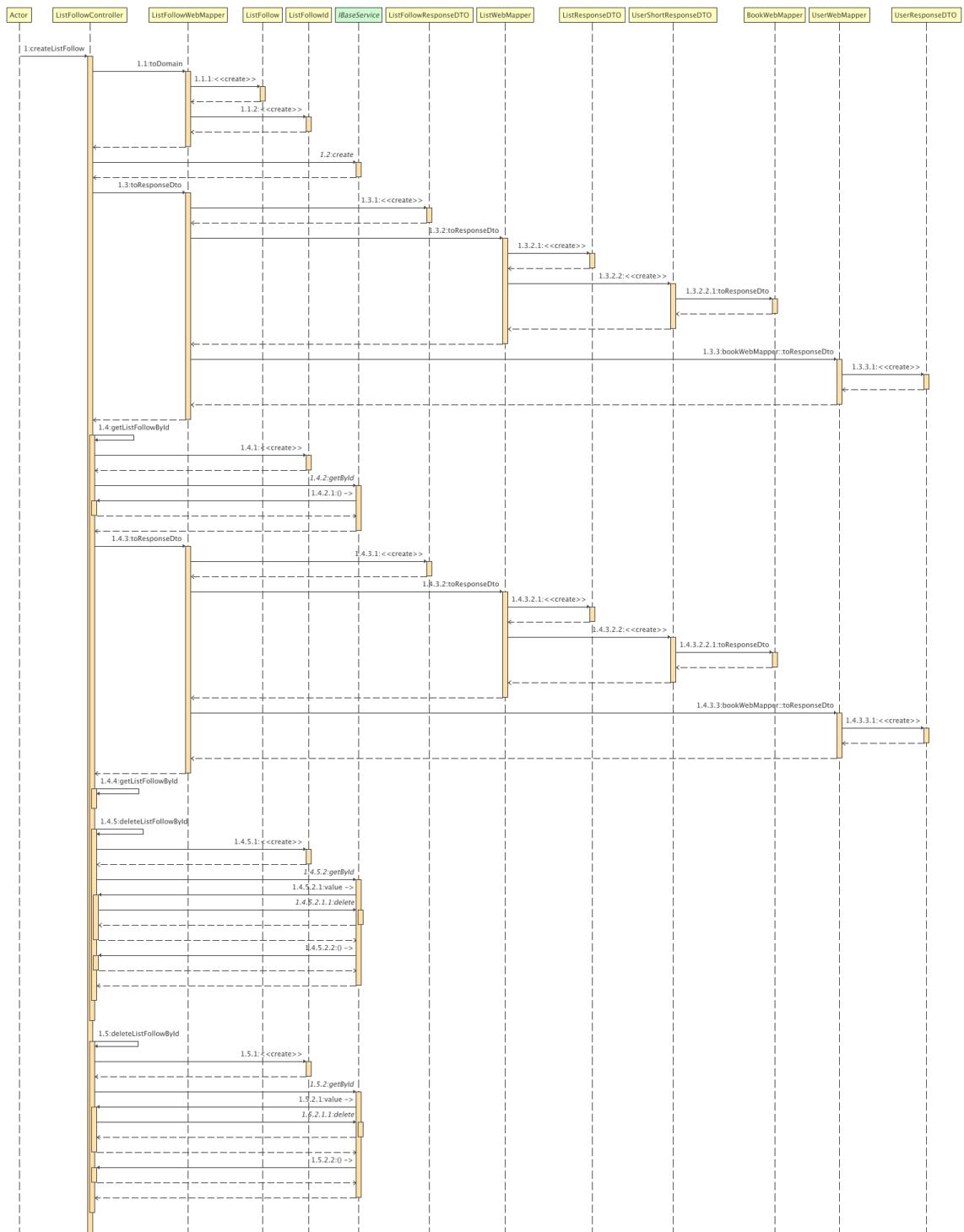


Abbildung 21: Follow List Sequence Diagramm