

Hochschule Osnabrück

University of Applied Sciences

Fakultät

Ingenieurwissenschaften und Informatik

Schriftliche Ausarbeitung zum Thema:

**Umsetzung einer REST-API zur Verwaltung von Buch-
messen eines Verlags**

im Rahmen des Moduls

Software-Architektur – Konzepte und Anwendungen,
des Studiengangs Informatik-Medieninformatik

Autor:	Katharina Schmidt
Matr.-Nr.:	902423
E-Mail:	Katharina.schmidt.1@hs-osnabrueck.de
Themensteller:	Prof. Dr. Rainer Roosmann

Abgabedatum: 22.07.2024

Inhaltsverzeichnis

Inhaltsverzeichnis	2
Abbildungsverzeichnis	4
Tabellenverzeichnis	5
Source-Code Verzeichnis	6
Abkürzungsverzeichnis	7
1 Einführung in die Software-Architektur	1
1.1 Vorstellung des Themas und Ziel der Ausarbeitung	1
1.2 Aufbau der Hausarbeit	2
2 Darstellung der Grundlagen	3
2.1 Grundlagen der Softwarearchitektur	3
2.2 Design-Prinzipien	3
2.2.1 Single-Responsibility-Prinzip	3
2.2.2 Open-Closed-Prinzip	4
2.2.3 Liskovsches Substitutionsprinzip	4
2.2.4 Interface-Segregation-Prinzip	4
2.2.5 Dependency-Inversion-Prinzip	4
2.3 Domain Driven Design (DDD)	4
<i>Layered Architecture</i>	4
2.4 Entity-Control-Boundary-Pattern	5
2.5 Clean Architecture	5
3 Konzeptionierung	7
3.1 C4-Model	7
3.1.1 Context-Diagramm	7
3.1.2 Container-Diagramm	8
3.1.3 Component-Diagramm	8
3.1.4 Code-Diagramm	9
3.2 Klassendiagramm	9
3.3 Entwurf der UI	10
4 Implementierung	12
4.1 Datenpersistierung	12
4.2 Modul „authorAdministration“	12
4.3 Modul „bookFairAdministration“	14
4.4 Modul „waitlistAdministration“	16

4.5	Umsetzung der Authentifizierung und Autorisierung	17
4.6	Logging	18
4.7	OpenAPI	19
4.8	Fault-Tolerance	19
4.9	Umsetzung der Frontend-Architektur	20
4.9.1	Formular Verarbeitung	21
4.10	Finale Struktur der API	21
4.11	Qualitätssicherung	22
4.11.1	Unit-Tests	22
4.11.2	Black-Box-Tests	23
4.11.3	Versionisierung	23
5	Zusammenfassung und Fazit	24
5.1	Zusammenfassung	24
5.2	Fazit	24
5.3	Ausblick	25
6	Referenzen	26
7	Anhang	28

Abbildungsverzeichnis

Abbildung 1: Das Schichtenmodell nach dem Clean-Architecture-Modell [9].....	6
Abbildung 3 Context-Diagramm	7
Abbildung 4 Container-Diagramm	8
Abbildung 5 Component-Diagramm	9
Abbildung 6: Entwurf der UI.....	11
Abbildung 7: Logo	11

Tabellenverzeichnis

Tabelle 1: Kernprinzipien von DDD	4
Tabelle 2: Darstellung der Author-Endpunkte	13
Tabelle 3: Darstellung der Buchmessen-Endpunkte	14
Tabelle 4: Darstellung der Waitlist Endpunkte	17
Tabelle 5: Rollen und ihre Zugriffsberechtigungen	18
Tabelle 6: Fault Tolerance [15]	19
Tabelle 7: Unit-Test-Klassen	22

Source-Code Verzeichnis

Snippet 1 Konfiguration der Datenbank in der Application.properties Datei	12
Snippet 2: GET-Request	13
Snippet 3: Löschen eines Autors	14
Snippet 4: Beziehung zwischen Autoren und Buchmessen	14
Snippet 5: Teilnehmer einer Buchmesse	15
Snippet 6: "signIn"-Methode für die Registrierung eines Autors zur Buchmesse	16
Snippet 7: Einfügen in die Warteliste, beim Editieren der Teilnehmer	16
Snippet 8: Integration der Autoren und Buchmessen in die WaitlistEntry	17
Snippet 9 Keycloak-Setup	18
Snippet 10: Logging-Konfiguration	18
Snippet 11: Ermittlung der Logger-Instanz	19
Snippet 12: Logging-Beispiel	19
Snippet 13: Anfragespezifische OpenAPI Beschreibung	19
Snippet 14: Template	20
Snippet 15: Parameterzugriff	20
Snippet 16: Get-Request mit TemplateInstance als Rückgabe	21
Snippet 17: Bsp. Formular Verarbeitung	21
Snippet 18: Unit-Test "testSignIn"	22
Snippet 19: Path-Attribut der Klasse "BookFairRessource"	23

Abkürzungsverzeichnis

ACL	Anti-Corruption-Layer
API	Application Programming Interface
CDI	Context and Dependency Injection for the Java EE Plattform
CRUD	Create, Read, Update, Delete
DBMS	Datenbankmanagement System
DDD	Domain Driven Design
DI	Dependency Injection
DIP	Dependency-Inversion-Prinzip
DTO	Datentransferobjekt
ECB	Entity-Controller-Boundary Pattern
EJB	Enterprise Java Beans
IEEE	Institute of Electrical and Electronics Engineers
ISP	Interface-Segregation-Prinzip
HTTP	Hypertext Transfer Protocol
JDBC	Java Database Connectivity
JPA	Java Persistence API
Java EE	Java Enterprise Edition, in der Version 7
JSF	Java Server Faces
JSON	JavaScript Object Notation
LSP	Liskovsches Substitutionsprinzip
OCP	Open-Closed Prinzip
OIDC	OpenID Connect
ORM	Object-Relational Mapping
OOP	Objekt orientierte Programmierung
POM	Project Object Model
REST	Representational State Transfer
SRP	Single-Responsibility-Prinzip
SWA	Software-Architektur
SFLB	Statefull Session Bean
SLSB	Stateless-Session Bean
SQL	Structured Query Language
RBAC	Role-Based Access Control
RMM	Richardson Maturity Model
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

1 Einführung in die Software-Architektur

„Moderne Software-Architektur ist ein entscheidender Faktor für den Erfolg von Softwaresystemen“ [1]

Die Verwendung des Begriffs „Software-Architektur“ kann für Laien oft ein falsches Bild vermitteln. „Architekt“ kommt ursprünglich aus dem altgriechischen Wort **architekton**, ἀρχι- („der Anfang, die Grundlage“) + τέκτων („Handwerker“). Man versteht also unter Architektur das Gestalten und Konstruieren von Bauwerken. Dies ist jedoch unter den heutigen Bauarchitekten größtenteils verlorenen gegangen, da sie für die Planung des Baus engagiert werden und nicht für die tatsächliche Bebauung des Gebäudes. Hier liegt der zentrale Unterschied zu Software-Architekten, die sowohl für die Planung als auch die Umsetzung der Architektur von Softwaresystemen zuständig sind. Sie nehmen aktiv am Erstellungsprozess teil und verfügen über fundiertes Wissen über das zu entwickelnde System sowie umfangreiche Erfahrung als Software-Entwickler. Diese ganzheitliche Beteiligung ermöglicht es ihnen, flexible und zukunftssichere Systeme zu gestalten. Im Rahmen des Moduls "Software-Architektur - Konzepte und Anwendungen" wird der Prozess der Erstellung einer REST-API (Representational State Transfer, Application Programming Interface) mit Java erläutert.

1.1 Vorstellung des Themas und Ziel der Ausarbeitung

Die „BookFestival-Planner“-Application ist ein Autorenverwaltungs-Tool eines Verlags, das Nutzern eine Übersicht der anstehenden Buchmessen und teilnehmenden Autoren bietet. Öffentliche Methoden werden mittels URL-Pfaden und HTTP-Verben bereitgestellt, und der Zugriff auf die API wird basierend auf den Nutzerrollen gesteuert. Dies erfolgt durch ein Login-System mit Nutzernamen und Passwort. Autoren können sich für Buchmessen an- und abmelden, während Administratoren Buchmessen und Autoren bearbeiten, hinzufügen und löschen können. Eine Buchmesse hat einen Namen, ein Datum, einen Ort und eine maximale Teilnehmeranzahl. Autoren besitzen einen Vor- und Nachnamen. Es wird eine Wartelistenfunktion integriert: Bei vollständiger Belegung einer Messe werden Autoren automatisch auf die Warteliste gesetzt und rücken nach, wenn Plätze frei werden. Autoren können sich auch von der Warteliste austragen.

Ziel der Ausarbeitung ist es, ein tiefgehendes Verständnis für die Erstellung und Realisierung einer Softwarearchitektur zu schaffen. Ein RESTful-Webservice soll die genannten Anforderungen erfüllen, und eine Benutzeroberfläche soll den Nutzern das Holen, Schreiben, Ändern und Löschen von Daten ermöglichen. Der Umgang mit gängigen Frameworks soll ebenfalls verstanden und sinnvoll eingesetzt werden.

In dieser Hausarbeit bezieht sich der Ausdruck „Nutzer“, „Prüfer“, „Admin“ oder vergleichbare maskuline Worte, auch auf die andersgeschlechtliche Person.

1.2 Aufbau der Hausarbeit

Zunächst werden die fachlichen Grundlagen erläutert, um ein umfassendes Verständnis zu schaffen. Die genutzten Technologien und die Konzeptionierung der API werden mit Skizzen und Diagrammen veranschaulicht. Faktoren einer guten Software-Architektur werden beleuchtet. Anschließend wird der Projektaufbau erläutert und Klassen sowie Methoden anhand ihres Quellcodes näher betrachtet. Abschließend erfolgt eine kritische Betrachtung der Umsetzung und mögliche Ausblicke für die Erweiterung des Projekts.

2 Darstellung der Grundlagen

Das folgende Kapitel bietet das Fundament, welches das Verständnis der Dokumentation und der Implementierung der Software erleichtern soll. Es beinhaltet die wesentlichen Prinzipien und Muster der Softwarearchitektur.

2.1 Grundlagen der Softwarearchitektur

Der Schwerpunkt dieser Projektarbeit basiert darauf, wie eine Softwarearchitektur entwickelt wird und welche Bestandteile dazu gehören. Der IEEE-Standard definiert Softwarearchitektur als „die Organisationsstruktur eines Systems“ [2]. Konkret geht es um die Organisation der Komponenten eines Systems (wie Datenbanken, Benutzeroberflächen und APIs) im Kontext der Entwicklungsumgebung. Dabei sollen reale Sachverhalte auf technische Lösungen übertragen werden. Nach R.C. Martin besteht „das Ziel der Softwarearchitektur [...] darin, den Personalaufwand für den Aufbau und die Wartung des erforderlichen Systems zu minimieren.“ [3, p. 58]). Eine geeignete Softwarearchitektur soll die Qualität der Software sichern, indem sie einfache Erweiterbarkeit und Wartbarkeit ermöglicht. Softwarearchitekturstile und Entwurfsmuster tragen zur Entwicklung einer qualitativ hochwertigen Softwarearchitektur bei.

2.2 Design-Prinzipien

Für den Entwurf hochwertiger Software wurden von R. C. Martin die SOLID-Prinzipien formuliert. Sie sind die grundlegenden Designprinzipien in der objektorientierten Softwareentwicklung. Gemäß ([3, p. 58]) zeigen diese Prinzipien, wie Funktionen und Datenstrukturen in Klassen angeordnet und miteinander verbunden sein sollten. Sie dienen als Regeln und Leitfaden für die Gestaltung von Klassen und deren Beziehungen. „Wenn diese Prinzipien eingehalten werden, entsteht besserer Code und die Software wird besser wartbar“ [4]. SOLID ist ein Akronym und steht für:

- **Single-Responsibility-Prinzip**
- **Open-Closed-Prinzip**
- **Liskovsches Substitutionsprinzip**
- **Interface-Segregation-Prinzip**
- **Dependency-Inversion-Prinzip**

2.2.1 Single-Responsibility-Prinzip

Das Single-Responsibility-Prinzip (SRP) besagt laut Martin [3, p. 62], dass ein Modul nur einer einzigen Verantwortlichkeit gerecht werden sollte. Ein Modul wird hier als „zusammenhängender Satz von Funktionen und Datenstrukturen“ [3, p. 62] definiert. Wenn ein Modul mehrere Verantwortlichkeiten übernimmt, können verschiedene Bereiche entstehen, die zukünftige Änderungen erfordern und die Fehleranfälligkeit erhöhen. Dies fördert hohe Kohäsion und besser strukturierten, verständlicheren Code.

2.2.2 Open-Closed-Prinzip

Das Open-Closed-Prinzip (OCP) besagt gemäß [3, p. 70], dass ein Software-Artefakt für Erweiterungen offen, aber für Modifikationen geschlossen sein sollte. OCP wird oft durch den Einsatz von Polymorphismus erreicht. Für erwartete Änderungen können Abstraktionen eingeführt werden, die als Erweiterungspunkte dienen sollen. Das bedeutet, dass Interfaces oder abstrakte Klassen definiert werden können, die spezifizieren, wie neue Funktionalitäten hinzugefügt werden sollen, ohne bestehenden Code zu ändern. Dies fördert die Modularität und erleichtert die Wartung und Erweiterung des Systems.

2.2.3 Liskovsches Substitutionsprinzip

Das Liskovsche Substitutionsprinzip wurde von Barabara Liskov formuliert und besagt, dass eine abgeleitete Klasse alle Eigenschaften der Oberklasse beibehalten muss. Dies gewährleistet die Konsistenz der Vererbungshierarchie (vgl. [3, p. 78 f.]).

2.2.4 Interface-Segregation-Prinzip

Das Interface-Segregation-Prinzip (ISP) zielt darauf ab, dass Bausteine nur Teile von Interfaces implementieren, die sie tatsächlich benötigen. Dadurch wird eine präzisere und effizientere Verwendung der Schnittstellen ermöglicht. (vgl. [3, p. 86]).

2.2.5 Dependency-Inversion-Prinzip

Nach dem Dependency Inversion Principle (DIP) sollen Abhängigkeiten auf Abstraktionen und nicht auf konkrete Implementierungen bezogen werden. Dies erreicht lose Kopplung und vermeidet zyklische Abhängigkeiten. (vgl. [3, p. 87 f.]).

2.3 Domain Driven Design (DDD)

„DDD ist ein Satz von Werkzeugen, die beim Entwerfen und Implementieren von hochwertiger Software helfen, [...] sowohl auf strategischer als auch auf taktischer Ebene.“ [5, p. 1] In dem Kontext der Projektarbeit liegt der Fokus auf den taktischen Werkzeugen, die die Entwicklung der Erstellung nützlicher Software unterstützen. Mit Hilfe von DDD soll eine umfassende Methode zur Modellierung komplexer Geschäftsanforderungen durch präzise Modellierung von Geschäftsdomänen geboten werden. Im Folgenden ist nach [6] eine Erläuterung einige Kernprinzipien von DDD ersichtlich:

Tabelle 1: Kernprinzipien von DDD

Name	Bedeutung
<i>Layered Architecture</i>	Es handelt sich um ein Entwurfsmuster, bei dem die Systemarchitektur in verschiedene Schichten unterteilt wird, sodass die Kommunikation nur zwischen den unmittelbar benachbarten Schichten erfolgt.
<i>Modules</i>	Logische Gruppierungen von zusammengehörigen Klassen, Interfaces und weiteren Programmkomponenten.

<i>Anti-Corruption Layer (ACL)</i>	Mustertyp, der Interaktionen zwischen unterschiedlichen Systemen oder Kontexten isoliert und sicherstellt, dass das interne Modell des neuen Systems nicht durch das externe System „verunreinigt“ wird.
<i>Entitäten</i>	Objekte mit einer eindeutigen Identität, die über den Lebenszyklus konstant bleiben, sind Entitäten.
<i>Wertobjekte</i>	Objekte, welche keine besondere Identität haben, sondern durch ihre Eigenschaften definiert sind. Meistens werden sie als unveränderliche Objekte definiert, damit sie wiederverwendbar sind.
<i>Aggregate</i>	Gruppierung von zusammengehörenden Entitäten und Wertobjekten zu einer gemeinsamen Entität. Ein Aggregate hat eine Root Wurzel) als einzigen Zugriff nach außen
<i>Service Objekte</i>	Wichtige funktionale Konzepte, welche sich auf mehrere Objekte im Domänenmodell beziehen. Sie sind in der Regel zustandslos und ermöglichen daher eine Wiederverwendung. Sie kapseln die Geschäftslogik, die nicht natürlich zu einem bestimmten Objekt gehört.
<i>Fabriken</i>	Werden verwendet, um die Erzeugung von Fachobjekten in spezielle Fabrik-Objekte auszulagern, insbesondere wenn die Erzeugung komplex ist oder zur Laufzeit ausgetauscht werden muss. Üblicherweise werden dabei erzeugende Entwurfsmuster wie abstrakte Fabrik, Fabrikmethode oder Erbauer eingesetzt.
<i>Repositorys</i>	bieten Abstraktionen für das Speichern und Abrufen von Aggregaten. Für alle Fachobjekte, die über die Infrastrukturschicht geladen werden, existiert eine Repository-Klasse, die die Implementierung der Lade- und Suchfunktionen nach außen hin kapselt.

2.4 Entity-Control-Boundary-Pattern

Das Entity-Control-Boundary-Pattern (ECB) wird nach [7] zur Strukturierung der Anwendung in drei Schichten verwendet:

1. **Boundary:** Schnittstelle zur Außenwelt, ermöglicht Kommunikation mit externen Systemen.
2. **Control:** Steuerungseinheit zwischen Entity und Boundary, enthält die Anwendungslogik.
3. **Entity:** Repräsentiert Kernkonzepte und Datenmodelle der Anwendung.

Das Pattern kann um eine **Gateway-Schicht** erweitert werden, die die Kommunikation nach außen verwaltet, während die Boundary-Schicht eingehende Anfragen handhabt. Diese Struktur ermöglicht eine klare Trennung der Komponenten und Logik der Applikation.

2.5 Clean Architecture

Clean Architecture ist ein Softwarearchitektur-Paradigma, welches von Robert C. Martin entwickelt wurde. Ziel ist es Anwendungen so zu gestalten, dass sie modular und leicht wartbar sind. Das Konzept beruht dabei auf der Förderung der Entkopplung verschiedener Systemkomponenten. Nach Martin [8] gibt es folgende Grundprinzipien:

Unabhängigkeit von Frameworks, Leichte Testbarkeit, Unabhängigkeit der UI, Unabhängigkeit von externen Systemen und Datenbank-Unabhängigkeit.

Clean Architecture wird oft durch konzentrische Ringe visualisiert, wobei jede Schicht bestimmte Verantwortlichkeiten hat. Der Aufbau der Schichten sieht dabei wie folgt aus:

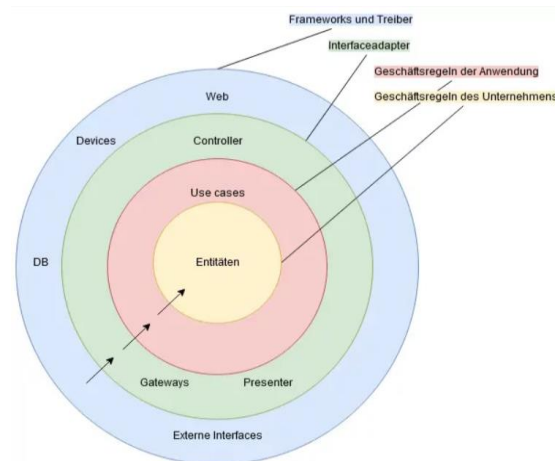


Abbildung 1: Das Schichtenmodell nach dem Clean-Architecture-Modell [9]

In den **Entitäten** sind die Geschäftslogik und die Geschäftsobjekte enthalten. Sie repräsentieren die Kernlogik. Die spezifische Anwendungslogik wird in den **Use Cases** umgesetzt, dabei orchestriert sie die Geschäftslogik und koordiniert die Datenflüsse zwischen Entitäten und anderen Schichten. Die Umwandlung der Datenformate zwischen der Applikation und externen Systemen wird durch die Interface Adapter vollzogen. In der äußersten Schicht sind die Implementierungen von Frameworks und Technologien enthalten. Diese Schicht interagiert direkt mit externen Systemen und Bibliotheken. Der Hauptaspekt der Clean-Architecture ist die Dependency Rule. Sie besagt, dass Abhängigkeiten nur in Richtung der inneren Schicht zeigen dürfen.

3 Konzeptionierung

Nach der Erläuterung der fachlichen Grundlagen, folgt nun die Beschreibung der Konzeptionierung der Software.

3.1 C4-Model

Laut [13] ist das C4-Modell eine grafische Notationstechnik zur Modellierung und Dokumentation von Software-Architekturen. Es wurde von Simon Brown entwickelt und umfasst vier Abstraktionsebenen, die die Aufteilung des Systems in Container und Komponenten sowie deren Beziehungen ermöglichen. Bei Bedarf können auch die Beziehungen zu Benutzern und externen Systemen dargestellt werden. Im Folgenden werden die Ebenen vorgestellt und die Techniken angewendet.

3.1.1 Context-Diagramm

Das Context-Diagramm ist nach [13] der Ausgangspunkt für die Diagrammerstellung und Dokumentation der Software. Dabei liegt der Fokus auf den Akteuren, Rollen oder Persona und dem Softwaresystem. Das Diagramm soll auch für technisch nicht versierte Personen verständlich sein. In diesem Kontext werden für ein besseres Verständnis des Anwendungskontexts zunächst Persona erstellt. Persona seien nach [15] fiktive Typen, die die Bedürfnisse, Fähigkeiten und Ziele“ der Benutzer vertreten. Man unterscheide zwischen drei Persona-Typen:

- **Primäre Persona:** „Stellvertreter der wichtigsten Zielgruppen“ [16]
- **Sekundäre Persona:** „kleinere, speziellere Zielgruppen“ [16]
- **Anti Persona:** „Gruppe von Personen, die explizit nicht die Zielgruppe der Aktivitäten darstellen, für die nicht entwickelt wird.“ [16]

Dafür werden Steckbriefe erarbeitet, sowie die Einstellung und Erwartungen gegenüber der Software. Die Steckbriefe sind in Anhang 3, Anhang 4 und Anhang 5 ersichtlich.

Anhand der Auseinandersetzung mit den Persona kann nun das Context-Diagramm erstellt werden:

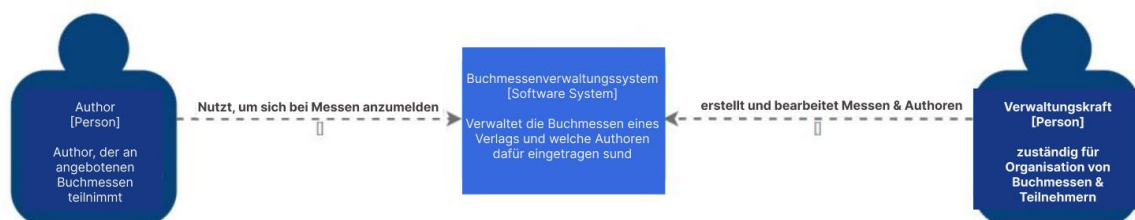


Abbildung 2 Context-Diagramm

3.1.2 Container-Diagramm

In dem Container-Diagramm wird nach [13] der Anwendungsbereich des Softwaresystems dargestellt und wie die Verantwortlichkeiten in der Softwarearchitektur verteilt werden. Außerdem werden die wichtigsten Technologieoptionen und wie die Container miteinander kommunizieren veranschaulicht. Dieses Diagramm ist sowohl für Support-/Betriebsmitarbeiter als auch Softwareentwickler gleichermaßen nützlich.

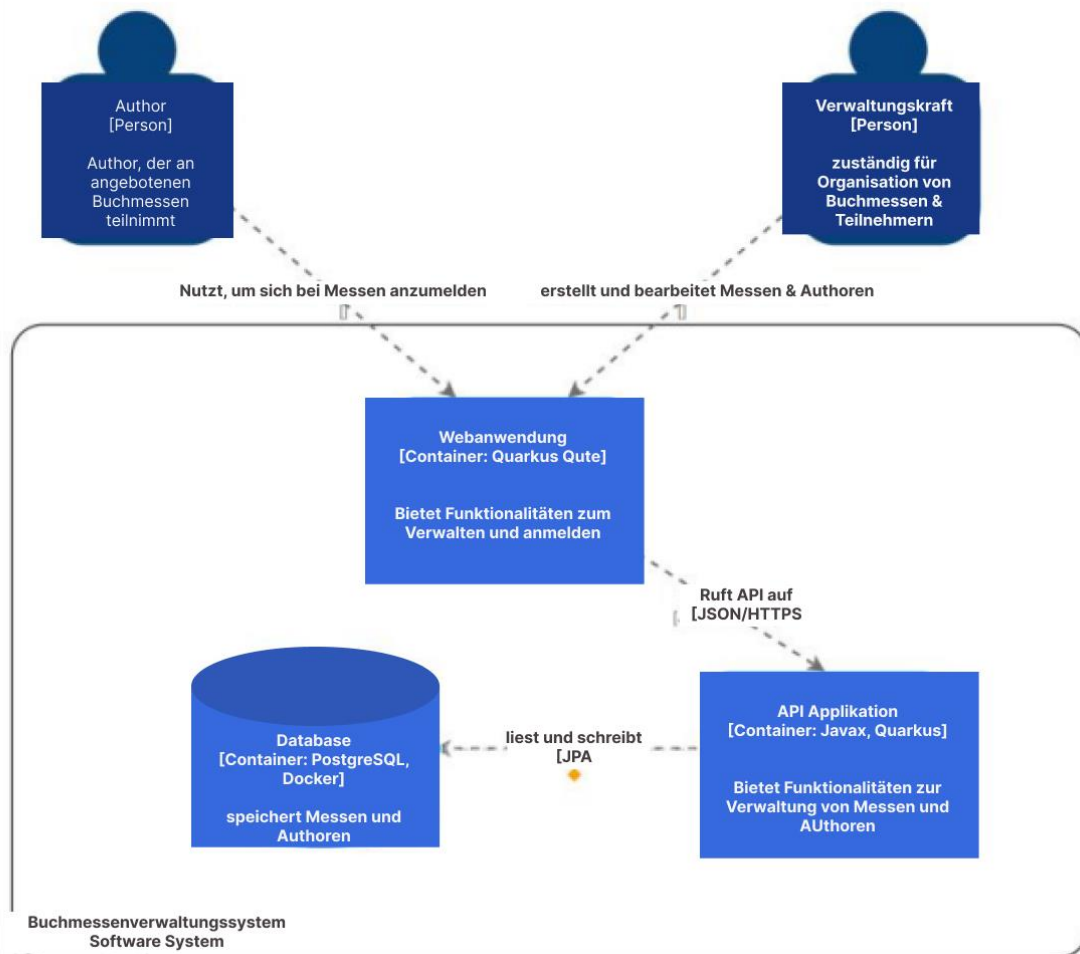


Abbildung 3 Container-Diagramm

3.1.3 Component-Diagramm

In dem Component-Diagramm wird nach [13] gezeigt, welche Komponenten ein einzelner Container beinhaltet und welche Verantwortlichkeiten dieser hat. In diesem Kontext bedeutete Komponente eine Gruppierung von Funktionalitäten, welche in einem Interface implementiert seien.

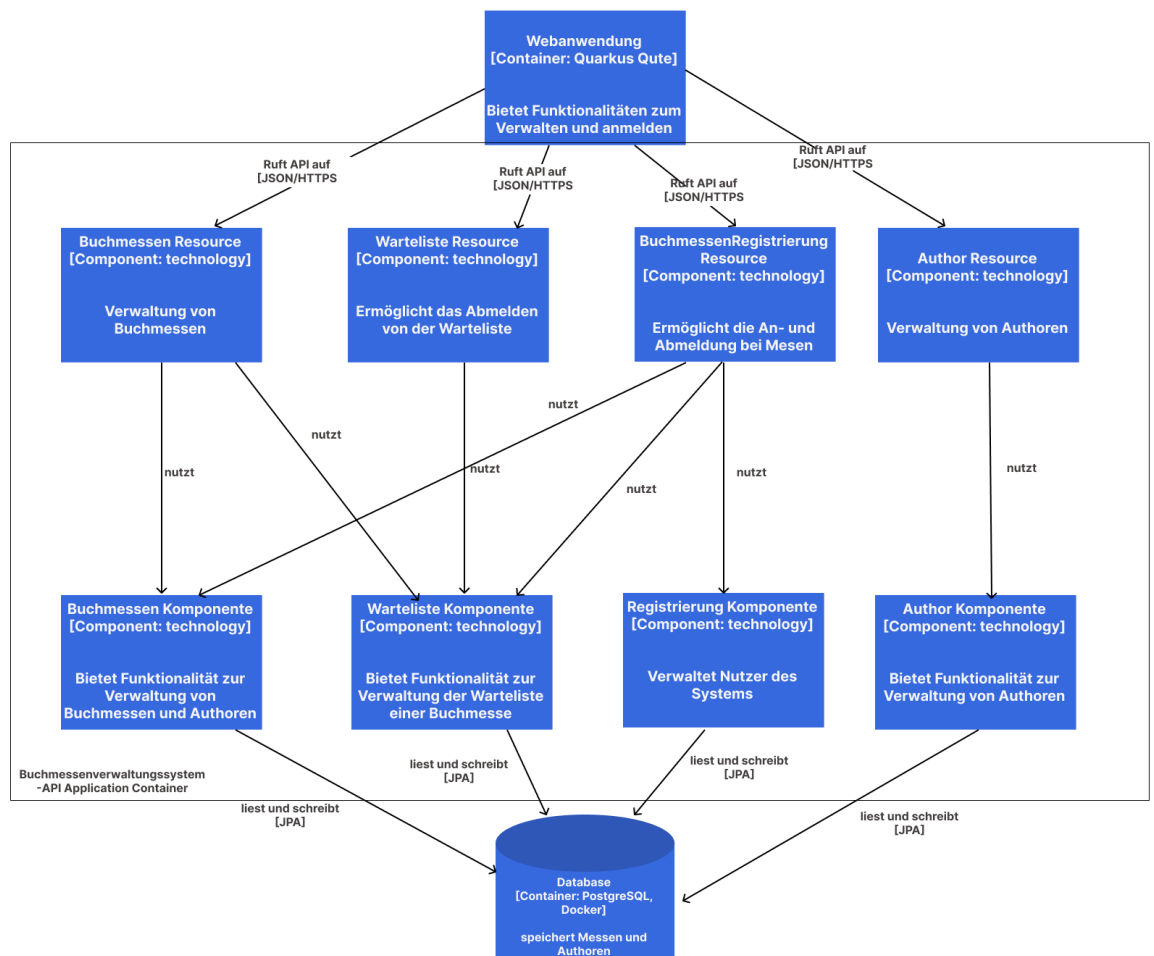


Abbildung 4 Component-Diagramm

3.1.4 Code-Diagramm

Bei dem Code-Diagramm handelt es sich nach [13] um eine optionale Detailebene, die häufig bei Bedarf über Tools wie IDEs verfügbar seien. Es wird dabei jede Komponente vergrößert, um zu veranschaulichen, wie sie im Code implementiert sei. Dies erfolgt unter Verwendung UML-Klassendiagrammen, Entity-Relationship-Diagrammen oder ähnlichem. Das vollständige Klassendiagramm ist Anhang 2 ersichtlich.

3.2 Klassendiagramm

Nach der Festlegung der grundlegenden Projektstruktur folgt die Entwicklung des Klassendiagramms. Der BookFairFestivalPlanner ist in drei Module aufgeteilt: Buchmessenverwaltung, Autorenverwaltung und Wartelistenverwaltung. Das Diagramm folgt dem erweiterten ECB-Muster und implementiert Prinzipien des taktischen DDD und der Clean Architecture. Die Struktur ist in mehrere Schichten unterteilt, die klare Verantwortlichkeiten gewährleisten:

- **Boundary-Layer:** Enthält Ressourcenklassen, unterteilt in Web und Rest. Rest-Paket für API-Methoden (JSON-Format) und Web-Paket für Webanwendung.

Ergänzt durch DTO-Paket (Data Transfer Object) zur externen Datenverwaltung.

- **Control-Layer:** Implementiert die Geschäftslogik und koordiniert Interaktionen zwischen den Schichten.
- **Entity-Layer:** Enthält Domänenmodelle (WaitlistEntry, Author, BookFair), die Kernobjekte und Geschäftsregeln repräsentieren.
- **Gateway-Layer:** Repository-Klassen für Persistenz und Datenbankzugriff.

Die Architektur setzt die SOLID-Prinzipien um: klare Verantwortlichkeiten, Erweiterbarkeit durch Interfaces, austauschbare Implementierungen, spezifische Schnittstellen und Abhängigkeiten von außen nach innen. Clean Architecture wird durch Schichtenanordnung und DTOs für Datentransfer zwischen Schichten umgesetzt. Abhängigkeiten verlaufen von außen nach innen, wobei innere Schichten unabhängig bleiben. Dependency Injection mit CDI (@Inject-Annotation) ermöglicht lose Kopplung und bessere Testbarkeit. Das Klassendiagramm ist in Anhang 2 ersichtlich. Das Klassendiagramm ist in Anhang 2 ersichtlich.

3.3 Entwurf der UI

Der Entwurf der Benutzeroberfläche (UI) orientiert sich an den acht goldenen Regeln des Interface Designs von Ben Shneiderman [15]:

1. *Strebe nach Konsistenz.*
2. *Strebe nach universeller Bedienbarkeit.*
3. *Biete informatives Feedback an.*
4. *Gestalten Sie Dialoge so, dass sie zu einem Abschluss führen.*
5. *Verhindere Fehler.*
6. *Erlaube die Umkehrung von Aktionen.*
7. *Gib dem Nutzer das Gefühl der Kontrolle.*
8. *Reduziere die Belastung für das Kurzzeitgedächtnis*

Ziel ist es mit Hilfe der Regeln eine angenehme User Experience sicherzustellen. Das Design verwendet ein monochromatisches Farbschema mit der Primärfarbe: #E6EBF3, das konsistent in der gesamten Anwendung eingesetzt wird. Listen und Formulare sind einheitlich gestaltet. Bei fehlerhaften Eingaben in Formularen erhält der Nutzer sofortiges Feedback und kann seine Eingabe korrigieren. Eine Navigationsleiste unterstützt die Reduzierung der Belastung für das Kurzzeitgedächtnis. Standardisierte Icons für Lösch- und Bearbeitungsfunktionen gewährleisten einfache und universelle Bedienbarkeit. Beim Starten der Anwendung wird der Nutzer mit einer Willkommensseite begrüßt und kann zwischen Autoren und Buchmessen auswählen. Ein Login-Formular ermöglicht den Zugriff auf die Seiten:



Abbildung 5: Entwurf der UI

Ein in Logo, welches in Abbildung 6 ersichtlich ist, gewährleistet dem Wiedererkennungswert. Details zur Entwicklung des Logos und des Farbschemas sind in Anhang 1 ersichtlich.



Abbildung 6: Logo

4 Implementierung

Nun folgt die Beschreibung der Umsetzung der zu entwickelnden Software. Dabei werden zu Beginn die grundlegenden Mechanismen zur Erstellung der REST-API erläutert. Anschließend wird auf die Implementierung eingegangen.

4.1 Datenpersistierung

Für die persistente Speicherung der Daten wird eine Datenbank genutzt. Es kommt jedoch zu Mapping-Problemen aufgrund des Paradigmenwechsels zwischen objektorientierter Programmierung (OOP) und des Datenbankmanagementsystems (DBMS). Deswegen wird das Mapping in einen eigenen Layer den Gateway-Layer, welches auf dem Data Access Layer basiert, verschoben. Diese Schicht ermöglicht die Interaktion mit der Datenbank, ohne dass Entwickler selbstständig mit JDBC (Java Database Connectivity) arbeiten müssen. Stattdessen wird zur Erleichterung der Datenbankanbindung Hibernate ORM mit Panache verwendet. Hibernate ORM ist eine JPA-Implementierung, die das Mapping von Java-Objekten auf Datenbanktabellen definiert. Die Einbindung der Maven-Dependency erfolgt in der pom.xml Datei. Anschließend wird der JDBC-Treiber für PostgreSQL deklariert. In der Konfigurationsdatei application.properties wird die Art der Datenbank spezifiziert. In diesem Fall wird PostgreSQL verwendet, und es wird festgelegt, dass die Datenbank während der Entwicklungszeit in einem Docker-Container ausgeführt wird:

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=postgres
quarkus.datasource.password=password
quarkus.datasource.devservices.enabled=true
quarkus.datasource.devservices.port=9999
quarkus.datasource.devservices.db-name=project30
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.datasource.jdbc.transactions=ENABLED
```

Snippet 1 Konfiguration der Datenbank in der Application.properties Datei

Panache kann auf zwei Arten genutzt werden: Entweder erbt die zu speichernde Entitätsklasse von PanacheEntity, oder PanacheRepository wird verwendet, um das zugehörige Repository zu implementieren und die Entität als reguläre JPA-Entität zu definieren.

4.2 Modul „authorAdministration“

Das Modul "authorAdministration" verantwortet die Verwaltung von Autoren in dem System. Es ermöglicht das Abfragen, Erstellen, Editieren und Löschen von Autoren Daten und bildet damit eine zentrale Komponente der Anwendung. In Tabelle 2 sind die von der „AuthorResource“ unterstützten Methoden dargestellt. Nicht registrierte Nutzer haben keinen Zugriff auf die Endpunkte. Die Ordnerstruktur des Moduls orientiert sich am erweiterten ECB-Pattern.

Tabelle 2: Darstellung der Author-Endpunkte

/api/v1/author	Security	Funktion
GET	Admin, Autor	Gibt Auflistung aller Autoren zurück
GET /{id}	Admin, Autor	Gibt einen Autor zurück
GET /authors/{id}	Admin	Gibt Buchmesse eines Autors aus
POST	Admin	Erzeugt einen neuen Autor
PUT /{id}	Admin	Verändert bestehen Autor
DELETE /{id}	Admin	Löscht bestehenden Autor

Das **Boundary-Package** ist für den externen Ein- und Ausgang von Daten verantwortlich. Es enthält: den Ressourcen-Ordner und ein DTO-Ordner. Im DTO-Ordner ist die Klasse „*AuthorDTO*“ enthalten. DTOs werden verwendet, um die Validierung vom eigentlichen Objekt zu trennen und nur die notwendigen Daten zu übertragen. Wenn ein Nutzer einen neuen Autor erstellen möchte oder einen vorhandenen editieren möchte, sollte die Validierung nicht am eigentlichen Objekt erfolgen. Im Ressourcen-Paket dient die „*AuthorResource*“ Klasse zur Ausführung der CRUD-Operationen (Create, Read, Update, Delete). Die „*AuthorResource*“-Klasse nutzt JAX-RS zur Implementierung des RESTful-Webservices, was die Kommunikation mit dem Autorenverwaltungs-Backend ermöglicht. Ein Beispiel für einen GET-Request ist die `getAll()`-Methode in Snippet 2. Diese Methode ruft Autoren ab und gibt sie als JSON-Response zurück. Bei leerer Liste wird ein 404-Status zurückgegeben. Die Methode demonstriert die einfache Handhabung von HTTP-Anfragen und -Antworten mit JAX-RS.

```
@GET
public Response getAuthors() {
    LOG.info("Liste alle Autoren auf");
    Collection<Author> authors = this.authorManagement.getAll();
    if (authors == null) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
    return Response.ok(authors).build();
}
```

Snippet 2: GET-Request

Das **Control Paket** enthält das Interface „*IAuthorManagement*“, dass die Methoden für den Zugriff auf die Datenbank definiert. Diese Methoden werden vom „*AuthorRepository*“ im **Gateway-Paket** implementiert. Das Repository dient als Schnittstelle zwischen der Anwendung und der Datenbank und erlaubt es, Methoden für eigene Abfragen zu definieren.

Der Datentransfer mit der Datenbank erfolgt im „*AuthorRepository*“, welches das Interface „*PanacheRepository<Author>*“ implementiert. Panache ist eine Erweiterung von Hibernate ORM, die von Quarkus bereitgestellt wird. Es vereinfacht den Datenbankzugriff, indem es vordefinierte Methoden für gängige Datenbankoperationen bereitstellt. Die Klasse „*AuthorRepository*“ nutzt diese Funktionalitäten wie folgt:

- Die Methode „*listAll()*“ gibt alle Author-Entitäten aus der Datenbank zurück
- „*findByld(id)*“ sucht einen spezifischen Autor anhand seiner ID
- „*persist(author)*“ speichert einen neuen Autor in der Datenbank
- „*deleteByld(id)*“ löscht einen Autor anhand seiner ID

Die „*@Transactional*“-Annotation in Snippet 3 stellt sicher, dass die Datenbankoperationen in einer Transaktion ausgeführt werden, was die Datenintegrität gewährleistet.

```
@Override
@Transactional
public boolean delete(long id) {
    return deleteById(id);
}
```

Snippet 3: Löschen eines Autors

Im Entity-Paket befindet sich die „*Author*“-Klasse. Sie repräsentiert das Domänenmodell eines Autors und ist mit `@Entity` und `@Table` annotiert, was sie als JPA-Entity kennzeichnet und die Abbildung auf eine Datenbanktabelle ermöglicht. Ein Autor hat folgende Attribute: „id, firstname und lastname“. Wobei die Id automatisch beim Einfügen in die Datenbank generiert wird. Zudem hat sie eine Many-to-Many-Beziehung zu „*BookFair*“-Entitäten. Ein Autor kann an mehreren Buchmessen teilnehmen, und eine Buchmesse kann mehrere Autoren als Teilnehmer haben. Das `mappedBy`-Attribut in Snippet 4 zeigt an, dass die Beziehung in der „*BookFair*“-Klasse durch ein Feld namens „*participants*“ verwaltet wird. Das `cascade = CascadeType.REMOVE` stellt sicher, dass beim Löschen eines Autors auch alle zugehörigen Einträge in der Verbindungstabelle (bookfair_participants) gelöscht werden.

```
@ManyToMany(mappedBy = "participants", cascade = CascadeType.REMOVE)
private Set<BookFair> bookFairs = new HashSet<>();
```

Snippet 4: Beziehung zwischen Autoren und Buchmessen

Ein wichtiger Anspruch bei der Implementierung der Ressourcen war es, für jedes HTTP-Verb pro Klasse maximal eine Methode zu implementieren. Dies wurde durch die Verwendung der `@Path`-Annotation sowohl auf Klassen- als auch auf Methodenebene realisiert. Dadurch können innerhalb einer Klasse mehrere Methoden für eine GET-Anfrage implementiert werden, die sich durch ihre relativen Pfade mit Pfadparametern unterscheiden.

4.3 Modul „bookFairAdministration“

Das Modul „*bookFairAdministration*“ ist verantwortlich für die Verwaltung von Buchmessen in dem System. Es ermöglicht das Abfragen, Erstellen, Editieren und Löschen von Buchmessen sowie die Registrierung und Abmeldung von Autoren für Buchmessen. Dieses Modul bildet eine zentrale Komponente der Anwendung und unterstützt die Verwaltung von Buchmesse-Ereignissen, Teilnehmerlisten und Warteliste. Die Ordnerstruktur des Moduls orientiert sich am erweiterten ECB-Pattern. In Tabelle 3 sind die von der „*BookFairResource*“ und „*BookFairRegistrationResource*“ unterstützten Methoden dargestellt.

Tabelle 3: Darstellung der Buchmessen-Endpunkte

/api/v1/bookFair	Security	Funktion
GET	Admin, Autor	Gibt Auflistung aller Buchmessen zurück
GET /{id}	Admin, Autor	Gibt eine Buchmesse zurück
GET /name?name={name}	Admin, Autor	Gibt alle Buchmessen mit dem angegebenen Namen zurück
GET /name?name={location}	Admin, Autor	Gibt alle Buchmessen am angegebenen Ort zurück
POST	Admin	Erzeugt eine neue Buchmesse
PUT /{id}	Admin	Verändert bestehende Buchmesse
DELETE /{id}	Admin	Löscht bestehende Buchmesse

GET /{bookFair_id}/participants	Admin, Autor	Gibt alle Teilnehmer einer Buchmesse zurück
---------------------------------	--------------	---

/api/v1/registration	Security	Funktion
POST /{bookFairId}/authors/{authorId}	Admin, Autor	Meldet einen Autor für eine Buchmesse an, oder auf die Warteliste
DELETE /{bookFairId}/authors/{authorId}	Admin, Autor	Meldet einen Autor von einer Buchmesse ab

Die Ordnerstruktur des Moduls orientiert sich am erweiterten ECB-Pattern und ist analog zum „*authorAdministration*“-Modul. Das Boundary Paket enthält im Ressourcen Ordner neben dem DTO-Ordner und der „*BookFairResource*“-Klasse zusätzlich noch die Klasse „*BookFairRegistrationResource*“, welche Endpunkte für die Registrierung und Anmeldung von Autoren zu Buchmessen bietet. Das Control-Package definiert die Geschäftslogik und enthält die Interface „*IBookFairManagement*“, welches die Methoden für den Zugriff auf die Datenbank beschreibt. Die Implementierung erfolgt im „BookFairRepository“ im Gateway-Package, was als Schnittstelle zwischen der Datenbank und Anwendung dient. Es implementiert das Interface „*PanacheRepository<BookFair>*“. „BookFair“ und hat im Entity Paket folgende Attribute: **id, name, location, date, max participants und participants**. Die Implementierung von „Participants“, die in Snippet 5 ersichtlich ist, enthält die @ManyToMany Annotation für die Verwaltung der Teilnehmer einer Buchmesse. Die Verwendung von Eager Loading für die Teilnehmer-Beziehung (FetchType.EAGER) ist ein weiterer Unterschied zum typischen Lazy Loading in Many-to-Many-Beziehungen und wurde gewählt, um die Performanz bei häufigen Zugriffen auf die Teilnehmerliste zu optimieren. Die „cascade“-Annotation enthält die Cascade-Typen „PERSIST“ und „MERGE“. Dies ermöglicht, dass Änderungen an der Buchmesse-Entität automatisch an die verknüpften Autor-Entitäten übertragen werden, was das Speichern und Aktualisieren von Buchmessen mit ihren Teilnehmern vereinfacht.

```
@ManyToMany(fetch = FetchType.EAGER, cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinTable(
    name="bookfair_participants",
    joinColumns = {@JoinColumn(name="bookfair_id")},
    inverseJoinColumns = {@JoinColumn(name="author_id")}
)
```

Snippet 5: Teilnehmer einer Buchmesse

Ein wesentlicher Unterschied zum „*authorAdministration*“-Modul liegt in der Implementierung der Teilnehmerverwaltung. Die Methoden „*signIn*“ und „*signOut*“ im „*BookFairRepository*“ behandeln die Logik der An- und Abmeldung von Autoren für Buchmessen, einschließlich der Wartelistenverwaltung. Diese Methoden verwenden benutzerdefinierte Enums (RegistrationResult und SignOutResult) zur Rückgabe detaillierter Statusinformationen. In der „*signIn*“-Methode in Snippet 6 werden zunächst die Entitäten abgerufen und mit Hilfe des EntityManagers aus der Datenbank geladen. Falls entweder die Buchmesse oder der Autor nicht gefunden werden, wird FAILED zurückgegeben. Wenn der Autor bereits für diese Buchmesse angemeldet ist, wird ALREADY_SIGNEDIN zurückgegeben. Wenn noch Plätze frei sind, wird der Autor hinzugefügt und REGISTERED zurückgegeben. Wenn keine Plätze mehr frei sind, wird der Autor automatisch zur Warteliste hinzugefügt und WAITLISTED zurückgegeben.

```

@Override
@Transactional
public RegistrationResult signIn(long bookFairId, long authorId) {
    BookFair bookFair = this.em.find(BookFair.class, bookFairId);
    Author author = this.em.find(Author.class, authorId);
    if (bookFair == null || author == null) {
        return RegistrationResult.FAILED;
    }
    if (bookFair.isParticipantSignedIn(author)) {
        return RegistrationResult.ALREADY_SIGNEDIN;
    }
    if (bookFair.getSignedInAuthorsCount() < bookFair.getMaxParticipants()) {
        bookFair.addParticipant(author);
        return RegistrationResult.REGISTERED;
    }
    waitlistManagement.addToWaitlist(bookFairId, author);
    return RegistrationResult.WAITLISTED;
}

```

Snippet 6: "signIn"-Methode für die Registrierung eines Autors zur Buchmesse

Zusätzlich implementiert das „bookFairAdministration“-Modul eine Integration mit dem „WaitlistManagement“-Service aus dem „waitlistAdministration“-Modul. So wird die Wartelistenverwaltung direkt in den Bearbeitungsprozess einer Buchmesse miteingebunden. Dies ist notwendig, da die Anwendung die Funktionalität bietet die maximale Teilnehmeranzahl zu verändern. Die Implementierung erfolgt in der „edit“-Methode in Snippet 7 im „BookFairRepository“. Es wird zwischen zwei Hauptszenarien unterschieden. Wenn die maximalen Teilnehmeranzahl verringert wird, dann wird eine Liste der überzähligen Teilnehmer erstellt und von der Buchmesse entfernt. Anschließend wird diese Liste zur Warteliste hinzugefügt. Wenn die maximale Teilnehmeranzahl erhöht wird, dann wird die aktuelle Warteliste für die Buchmesse geholt. Die Autoren von der Warteliste werden zur Buchmesse hinzugefügt, bis die maximale Teilnehmeranzahl erreicht ist oder die Warteliste leer ist. Anschließend werden die hinzugefügten Autoren von der Warteliste entfernt.

```

if (newMaxParticipants < currentParticipantsCount) {
    List<Author> participantsToWaitlist = new ArrayList<>(bookFair.getParticipants());

    for (int i = currentParticipantsCount - 1; i >= newMaxParticipants; i--) {
        Author author = participantsToWaitlist.get(i);
        bookFair.removeParticipant(author);
        waitlistManagement.addToWaitlist(bookFairId, author);
    }
} else if (newMaxParticipants > currentMaxParticipants) {
    List<WaitlistEntry> waitlist = waitlistManagement.getWaitlistByBookFairId(bookFairId);
    while (bookFair.getParticipants().size() < newMaxParticipants && !waitlist.isEmpty()) {
        WaitlistEntry entry = waitlist.remove(0);
        bookFair.addParticipant(entry.getAuthor());
        waitlistManagement.removeFromWaitlist(bookFairId, entry.getAuthor());
    }
}

```

Snippet 7: Einfügen in die Warteliste, beim Editieren der Teilnehmer

4.4 Modul „waitlistAdministration“

Das Modul "waitlistAdministration" ist für die Verwaltung von Wartelisten für Buchmessen zuständig. Die Warteliste wird in ein zusätzliches Modul ausgelagert, um es für künftige Erweiterungen bereitzustellen. Es ermöglicht das automatische Hinzufügen von Autoren zu Wartelisten, wenn eine Buchmesse bereits voll ist, sowie das Verwalten dieser Wartelisten und die Austragung aus der Warteliste. In Tabelle 4 sind die von der

"WaitlistResource" unterstützten Methoden dargestellt. Nur angemeldete Nutzer haben Zugriff auf diese Endpunkte, um die Integrität der Wartelisten zu gewährleisten.

Tabelle 4: Darstellung der Waitlist Endpunkte

/api/v1/waitlist/	Security	Funktion
GET /api/v1/waitlist/{bookFairId}	Admin, Autor	Gibt die Warteliste für eine bestimmte Buchmesse zurück
DELETE /api/v1/waitlist/{bookFairId}/{authorId}	Admin, Autor	Meldet einen Autor von der Warteliste einer Buchmesse ab

Die Ordnerstruktur ist analog zum „bookFairAdministration“-Paket. Ein wesentlicher Unterschied zu den anderen Modulen liegt in der engen Integration mit dem „bookFairAdministration“- und „authorAdministration“-Modul. Dies wird durch die Beziehungen in der „WaitlistEntry“-Entität deutlich:

```
@ManyToOne
@JoinColumn(name = "bookfair_id", nullable = false)
private BookFair bookFair;

@ManyToOne
@JoinColumn(name = "author_id", nullable = false)
private Author author;
```

Snippet 8: Integration der Autoren und Buchmessen in die WaitlistEntry

Die @ManyToOne Annotation in Snippet 8 definiert eine Viele-zu-Eins-Beziehung zwischen Entitäten. In diesem Fall bedeutet es: Viele WaitlistEntry-Objekte können zu einer BookFair gehören und viele WaitlistEntry-Objekte können zu einem Autor gehören. Mit der @JoinColumn Annotation wird die Spalte in der Datenbanktabelle, die als Fremdschlüssel für die Beziehung dient, spezifiziert. Diese Struktur ermöglicht es, dass: Jeder WaitlistEntry genau einer BookFair und einem Author zugeordnet ist, eine BookFair mehrere WaitlistEntries haben kann und dass ein Author mehrere WaitlistEntries haben kann. In der Datenbank führt dies zu einer Tabelle für WaitlistEntry, die Spalten für bookfair_id und author_id enthält, welche auf die entsprechenden Tabellen für BookFair und Author verweisen.

4.5 Umsetzung der Authentifizierung und Autorisierung

Die Sicherstellung der Grundwerte der IT-Sicherheit: Vertraulichkeit, Verfügbarkeit und Integrität ist elementar für die Sicherheit einer REST-API. Zur Sicherung der Anwendungsendpunkte kann Keycloak verwendet werden. Keycloak ist eine Open-Source-Software für Identity und Access Management, die eine zentrale Verwaltung der Authentifizierung und Autorisierung ermöglicht. Dadurch wird sichergestellt, dass nur Benutzer mit den entsprechenden Rollen Zugriff auf die geschützten Ressourcen haben. Der Zugriff basiert auf dem Role-Based Access Control (RBAC) Steuerungsmodell, welches Benutzerrechte auf Basis von Rollen verwaltet. Berechtigungen werden Rollen zugeordnet und diese Rollen werden dann den Benutzern zugeteilt. Es gibt folgende Rollen in der Anwendung:

Tabelle 5: Rollen und ihre Zugriffsberechtigungen

Rolle	Zugriffsberechtigung
„admin“	Buchmessen einfügen, bearbeiten und löschen, Autor*innen einfügen, bearbeiten und löschen
„author“	An- und Abmeldung in Buchmessen, Abmeldung aus der Warteliste

Keycloak unterstützt RBAC nativ und stellt eine zentrale Rollenverwaltung sowie eine benutzerfreundliche Oberfläche zur Verwaltung von Rollen und Berechtigungen. Für die Authentifizierung und Autorisierung verwendet Keycloak das OpenID Connect (OIDC) Protokoll, eine Erweiterung von OAuth 2.0. OIDC ermöglicht eine sichere Authentifizierung und den Austausch von Benutzerinformationen zwischen der Anwendung (Client) und dem Identitätsanbieter (Keycloak). Dies gewährleistet eine standardisierte und sichere Methode zur Benutzerauthentifizierung und zum Informationsaustausch.

Die Integration von Keycloak in die Anwendung erfolgt über die Konfiguration in der `application.properties`-Datei. Hier werden wichtige Parameter für die OIDC-Verbindung und das Keycloak-Setup definiert:

```
quarkus.oidc.client-id=bookFestival
quarkus.oidc.credentials.secret=bookFestival-secret
quarkus.oidc.application-type=web-app
quarkus.oidc.logout.path=/logout
quarkus.oidc.logout.post-logout-path=/
%prod.quarkus.keycloak.devservices.enabled=false
quarkus.keycloak.devservices.port=8280
quarkus.keycloak.devservices.realm-name=quarkus
#keycloak user and roles
quarkus.keycloak.devservices.users.emma=author
quarkus.keycloak.devservices.roles.emma=author
quarkus.keycloak.devservices.users.anna=123456
quarkus.keycloak.devservices.roles.anna=admin
```

Snippet 9 Keycloak-Setup

Diese Konfiguration definiert die URL des Keycloak-Servers, die Client-ID und das Secret für die Anwendung, sowie weitere OIDC-spezifische Einstellungen. Zusätzlich werden Entwicklungsdienste für Keycloak aktiviert, was die Einrichtung und das Testen der Sicherheitsfunktionen in der Entwicklungsumgebung erleichtert. Zusätzlich werden Nutzer, deren Rollen und ihre Passwörter festgelegt. Die Ressourcen werden über die `@RolesAllowed`-Annotation geschützt.

4.6 Logging

Logging ist ein wichtiger Aspekt der Softwareentwicklung, der es ermöglicht, den Ablauf und den Zustand einer Anwendung zu überwachen und zu debuggen. Es hilft bei der Fehlerdiagnose. In diesem Projekt erfolgt das Logging in der Ausgabe. In Quarkus kann das Logging mit Hilfe von Konfigurationsparametern angepasst werden, die wie folgt gewählt wurden:

```
quarkus.log.console.format=%d{HH:mm:ss} %-5p [%c{2.}] (%t) %s%n
quarkus.console.color=false
quarkus.log.category."io.quarkus".level=INFO
```

Snippet 10: Logging-Konfiguration

Es wird ein benutzerdefiniertes Format für das Konsolenprotokoll festgelegt, das Zeit, Log-Level, Klassenname, Thread und Nachricht enthält. Dabei wird die Log-Stufe auf

INFO gesetzt, welche allgemeine Informationen und Nachrichten ausgibt. Um innerhalb einer Klasse die Operationen zu dokumentieren, wird das Logging-Framework importiert und eine Instanz erstellt:

```
LOG.info("Entferne Author mit der ID: " + authorId + " aus der Warteliste der Buchmesse mit der ID: " + bookFairId);
```

Snippet 11: Ermittlung der Logger-Instanz

In Snippet 12 ist ersichtlich wie nun geloggt werden kann.

```
private static final Logger LOG = Logger.getLogger(Wait-  
listResource.class.getName());
```

Snippet 12: Logging-Beispiel

Die Wahl des INFO-Levels ist eine bewusste Entscheidung, um relevante Informationen zu erfassen, ohne die Logs mit zu vielen Details zu überladen.

4.7 OpenAPI

Quarkus generiert automatisch eine OpenAPI-Dokumentation für die REST-Endpunkte basierend auf den JAX-RS-Annotationen und den Ressourcenklassen. Die API wird einheitlich beschrieben und zur Visualisierung wird Swagger UI unter <http://localhost:8080/swagger-ui/> bereitgestellt. Die OpenAPI-Definition wird in der Klasse BookFairPlannerAPIAppli angepasst, um grundlegende Metadaten wie Titel, Version, Kontakt und Lizenz zu definieren. Ressourcenklassen enthalten OpenAPI-Annotationen wie @Tag, @Operation und @APIResponse, um detaillierte Informationen zu den Endpunkten bereitzustellen. Die Anwendung der Annotationen ist in diesem Beispiel ersichtlich:

```
@APIResponses(value = {  
    @APIResponse(responseCode = "200", description = "Erfolgreiche Opera-  
tion"),  
    @APIResponse(responseCode = "404", description = "Keine Autoren ge-  
funden")  
})
```

Snippet 13: Anfragespezifische OpenAPI Beschreibung

4.8 Fault-Tolerance

Um die Zuverlässigkeit und Robustheit der Anwendung zu verbessern, wird das Konzept der Fault Tolerance verwendet. Dafür werden Fehlerquellen identifiziert und bewältigt. Quarkus bietet dafür SmallRye Fault Tolerance, als Implementation der MicroProfile Fault Tolerance:

Tabelle 6: Fault Tolerance [15]

Annotation	Funktion
@Retry	Ermöglicht es, eine Operation bei Auftreten eines Fehlers automatisch zu wiederholen. Sie kann konfiguriert werden, um die maximale Anzahl der Wiederholungsversuche festzulegen
@Timeout	Setzt eine maximale Ausführungszeit für eine Operation. Wenn die Operation die angegebene Zeit überschreitet, wird sie abgebrochen.

@Fallback	Bietet eine alternative Methode, die aufgerufen wird, wenn die Hauptoperation fehlschlägt. Sie ermöglicht es, eine Ersatzfunktionalität bereitzustellen, wenn die primäre Funktion nicht verfügbar ist.
@Circuit-Breaker	Es schützt das System vor wiederholten Aufrufen einer fehlerhaften Komponente. Wenn eine bestimmte Anzahl von Fehlern auftritt, "öffnet" sich der Circuit Breaker und verhindert weitere Aufrufe für eine bestimmte Zeit.

In der Anwendung werden hauptsächlich @Retry, @Timeout und @CircuitBreaker auf den Endpunkten angewendet, da das System eine externe zur Datenbank besitzt kann keine Alternative mittels Fallback geboten werden.

4.9 Umsetzung der Frontend-Architektur

Die Frontend-Architektur wird mithilfe eines serverseitigen Rendering-Ansatzes umgesetzt, der durch Quarkus Qute unterstützt wird. Quarkus Qute ist ein leistungsfähiges Templating-System, das die effiziente Erstellung dynamischer HTML-Seiten ermöglicht. Für das Design der HTML-Seiten wird Bootstrap, ein Frontend-CSS-Framework verwendet, um ein responsives Layout zu erstellen. Bootstrap bietet vordefinierte CSS-Klassen und JavaScript-Komponenten, die ein konsistentes Layout und interaktive Elemente ermöglichen. Zudem werden Buttons und Icons für eine intuitive Benutzerinteraktion bereitgestellt.

Eigene Ressource-Klassen werden erstellt, um die Views zu implementieren, sodass die JSON-API weiterhin ohne Einschränkungen funktioniert. Die separaten Ressourcen behandeln spezifische URL-Pfade und HTML-Methoden. Für die Datenübermittlung werden HTML-Formulare verwendet, unterstützt durch die Verwendung von @BeanParam Annotationen und @Consumes(MediaType.APPLICATION_FORM_URLENCODED) in den POST-Methoden.

Qute-Templates werden für die Implementierung genutzt. Diese Templates enthalten HTML-Dateien mit Qute-spezifischer Syntax, die eine dynamische Inhaltsgenerierung ermöglichen. Sie werden mit der @Location-Annotation injiziert:

```
@Inject
@Location("BookFairResource/bookfairForm.html")
Template bookfairForm;
```

Snippet 14: Template

In den Ressourcenmethoden wird die Render-Logik implementiert, wobei Daten an die Templates übergeben werden. Dadurch wird eine enge Integration zwischen Backend-Logik und Frontend-Darstellung ermöglicht. Die HTML-Generierung erfolgt serverseitig. In den HTML-Templates kann auf Parameter durch Namen in geschweiften Klammern zugegriffen werden:

```
<a href="/api/v1/web/bookFair/{bookFair.id}/edit" class="btn btn-warning btn-sm">
```

Snippet 15: Parameterzugriff

Zusätzlich lassen sich auch Objektmethoden, Bedingungen und Schleifen umsetzen, um dynamische Datensätze darzustellen. Diese sind mit geschweiften Klammern und einer

Raute zu Beginn der Bedingung oder Schleife gekennzeichnet, das Ende wird durch einen Backslash markiert.

Es werden zwei verschiedene Rückgabetypen: `TemplateInstance` und `Response` in den Ressource Methoden verwendet. `TemplateInstance` wird verwendet, wenn eine HTML-Seite gerendert werden soll, wie bei der Anzeige von Formularen oder Listen. Ein Beispiel für die Verwendung von `TemplateInstance` ist der GET-Request, der ein Formular zum Hinzufügen eines neuen Autors anzeigt. Dieser Endpunkt rendert eine HTML-Seite mit einem Formular, das nur für Administratoren zugänglich ist:

```
@GET
@Path("/add")
@RolesAllowed({"admin"})
public TemplateInstance showAddForm() {
    LOG.info("Zeige Formular zum Hinzufügen eines Autors");
    return authorForm.data("update", false);
}
```

Snippet 16: Get-Request mit `TemplateInstance` als Rückgabe

`Response` wird eingesetzt, wenn mehr Kontrolle über die HTTP-Antwort benötigt wird. Es ist besonders nützlich, wenn http-Statuscodes zurückgegeben oder zusätzliche Header gesetzt werden müssen. `Response` ermöglicht die Rückgabe von detaillierteren Antworten, die über das Rendern von HTML hinausgehen. Zum Beispiel nach dem Hinzufügen, Bearbeiten oder Löschen von Einträgen.

4.9.1 Formular Verarbeitung

Für das Hinzufügen und Editieren einer Buchmesse oder eines Autors werden Formulare bereitgestellt. Die Verarbeitung von Formulardaten erfolgt in den POST-Methoden. Dafür wird die `@BeanParam`-Annotation verwendet, um Formulardaten automatisch in ein `BookFairForm`- oder `AuthorForm`-Objekt zu binden. Diese Formulare werden dann jeweils mittels der `toDTO()`-Methoden in ein `BookFairForm`- oder `AuthorForm`-Objekt umgewandelt und zur Weiterverarbeitung an das `bookFairManagement` oder `authorManagement` übergeben. Dies ermöglicht eine saubere Trennung zwischen der Darstellung, der Datenübertragung und der Geschäftslogik:

```
public Response addBookFair(@BeanParam BookFairForm form) {
    BookFairDTO dto = form.toDTO();
    BookFair bookFair = bookFairManagement.addBookFair(dto);
    if (bookFair != null) {
        return Response.seeOther(URI.create("/api/v1/web/bookFair")).build();
    } else {
        LOG.info("Hinzufuegen fehlgeschlagen");
        return Response.serverError().build();
    }
}
```

Snippet 17: Bsp. Formular Verarbeitung

4.10 Finale Struktur der API

Die finale API ist in Anhang 6 ersichtlich und entspricht dem Level 2 des Richardson Maturity Models, was eine fortgeschrittene Implementierung von REST-Prinzipien darstellt. Das Model beschreibt nach [15] die einzelnen Stufen einer sauber strukturierten REST-API. Die Stufe zeichnet sich durch folgende Merkmale aus:

1. Ressourcenorientierung: Jede Entität wird durch eine eigene, eindeutige URL repräsentiert. Dies ermöglicht eine klare und intuitive Strukturierung der API
2. Korrekte Verwendung von HTTP-Methoden: Die API nutzt die standardisierten HTTP-Verben (GET, POST, PUT, DELETE) entsprechend ihrer vorgesehenen Funktionen.
3. Aussagekräftige HTTP-Statuscodes: Die API kommuniziert den Erfolg oder Misserfolg von Anfragen durch die Verwendung angemessener HTTP-Statuscodes.

4.11 Qualitätssicherung

Ein wesentlicher Bestandteil der Softwareentwicklung ist die Qualitätssicherung. Sie umfasst eine Reihe von Aktivitäten und Methoden, die Fehler vermeiden sollen und die Produktqualität verbessern sollen. Durch das eigenständige Testen über die UI oder externe Tools wie „Postman“ wurden eigenständige Tests durchgeführt. Weitere Methoden gilt es im Folgenden zu erläutern.

4.11.1 Unit-Tests

Um einzelne Einheiten oder Komponenten einer Software isoliert zu testen, werden Unit-Tests erstellt. Dabei bezeichnet eine Einheit nach [16] die kleinste testbare Komponente einer Anwendung, wie eine Methode oder eine Klasse. Unit Tests werden verwendet, um sicherzustellen, dass jede Komponente der Anwendung wie erwartet funktioniert. Zudem werden Fehler bereits früh im Entwicklungsprozess erkannt, was die Behebung erleichtern soll. Die resultierende REST-API verwendet vier Ressourcen-Klassen, somit wurde für jede Klasse ein Unit-Test angelegt:

Tabelle 7: Unit-Test-Klassen

Klasse	Erläuterung
„ <i>AuthorResourceTest</i> “	Testet alle Methoden der „ <i>AuthorResource</i> “
„ <i>BookFairResourceTest</i> “	Testet alle Methoden der „ <i>BookFairResource</i> “
„ <i>BookFairRegistrationResourceTest</i> “	Testet alle Methoden der „ <i>BookFairRegistrationResource</i> “
„ <i>WaitlistResourceTest</i> “	Testet alle Methoden der „ <i>WaitlistResource</i> “

Die bereits definierten Sicherheitseinstellungen für die Integrationstests werden mit der Annotation `@TestSecurity` konfigurieren. Sie ermöglicht es Benutzerauthentifizierung und -autorisierung zu simulieren und die geschützten Ressourcen zu testen. Die meisten Tests haben einen ähnlichen Aufbau, deswegen wird im Folgenden beispielhaft eine Methode der Klasse „*BookFairRegistrationResourceTest*“ vorgestellt:

```
@Test
@TestSecurity(user = "admin", roles = "admin")
public void testSignIn() {
    given().contentType(ContentType.JSON)
        .when().post("/api/v1/registration/4/authors/5")
        .then()
        .statusCode(200);
}
```

Snippet 18: Unit-Test "testSignIn"

Die Methode testet das Hinzufügen eines Autors zu einer Buchmesse. @Test markiert die Methode als Junit-Testmethode, sodass sie jederzeit ausgeführt werden kann. Die „contentType“-Methode setzt den Content-Type der Anfrage auf JSON. Die „post“-Methode sendet eine Anfrage an den Endpunkt `/api/v1/registration/1/authors/1` und simuliert die Registrierung des Autors mit ID 1 für die Buchmesse mit ID 1. Der Test erwartet eine erfolgreiche Registrierung, angezeigt durch den Statuscode 200. Ansonsten würde der Test fehlschlagen.

4.11.2 Black-Box-Tests

Blackbox-Tests sind eine Methode des Softwaretestens, bei der die innere Struktur des zu testenden Systems nicht bekannt ist. Der Fokus liegt auf der Funktionalität und dem Verhalten der API aus Sicht des Endnutzers. Diese Tests sollen sicherstellen, dass die API die erwarteten Ergebnisse liefert und korrekt auf verschiedene Eingaben reagiert. Um die API regelmäßig zu testen, wurde Swagger-UI verwendet, um die korrekte Funktionsweise sicherzustellen.

4.11.3 Versionisierung

Ein wichtiger Bestandteil bei der Entwicklung der REST-API ist die Versionisierung. Um sicherzustellen, dass Änderungen an Parametern oder URL-Strukturen keine Fehler bei bestehenden Clients verursachen, wird jeder Ressourcen-Klasse eine Version hinzugefügt. Nach [15] hat es den größten Vorteil, wenn man die Versionsnummer am Anfang des Pfads einfügt. So ist sichergestellt, dass alle Ressourcen mit allen Clients abwärts-kompatibel sind. Durch diese Strategie wird sichergestellt, dass ältere Clients weiterhin mit der ursprünglichen API-Version arbeiten können, während neue Funktionen in neueren Versionen eingeführt werden. Jeder Ressourcen-Klasse wird somit eine Version hinzugefügt, indem dem Pfad „v1“, wie in es in Snippet 19 ersichtlich ist, vorangestellt wird. Bei größeren Änderungen wird eine neue Ressourcen-Klasse mit „v2“ im Pfad erstellt.

```
@Path("/api/v1/bookFair")
```

Snippet 19: Path-Attribut der Klasse "BookFairRessource"

5 Zusammenfassung und Fazit

Abschließend soll das Erlernte wiederholt und zusammengefasst werden.

5.1 Zusammenfassung

Durch die Anwendung des Quarkus-Frameworks wurde eine solide Grundlage geschaffen, um die REST-Schnittstelle zu entwickeln. Die Integration von Quarkus mit verschiedenen Technologien wie JPA für die Datenpersistenz, OpenAPI für die API-Dokumentation und PostgreSQL als Datenbank-Engine ermöglichte die Erstellung einer robusten, skalierbaren und gut strukturierten Anwendung. Durch die Verwendung von Domain-Driven Design (DDD) konnte eine übersichtliche Struktur der einzelnen Endpunkte erreicht werden. Diese Aufteilung fördert die Wartbarkeit und Erweiterbarkeit des Systems. Die Implementierung einer rollenbasierten Zugriffskontrolle schützt die Grundwerte der IT-Sicherheit. Die Verwendung von Annotationen, einschließlich MicroProfile Fault Tolerance (@Retry, @Timeout, @CircuitBreaker), ermöglicht eine einfache Konfiguration der verwendeten Technologien und trägt zur Lesbarkeit des Codes bei. Zusätzlich wurde ein umfangreiches Logging implementiert, um die Überwachung und Fehlerbehebung zu erleichtern.

5.2 Fazit

Abschließend lässt sich sagen, dass die Muss- und Wunschkriterien erfüllt wurden und zusätzliche Funktionalitäten erfolgreich integriert werden konnten. Dabei wurden wertvolle Erfahrungen gesammelt, die die Bedeutung der Einhaltung von Prinzipien auch bei kleineren Softwareprojekten verdeutlichen. Eine ausführliche Planung im Voraus hat sich als essenziell erwiesen. Insbesondere die Integration der Wartelistenfunktionalität in einem separaten Modul hat sich bewährt, da sie ermöglicht, weitere Funktionalitäten hinzuzufügen, ohne Änderungen an anderen Modulen vornehmen zu müssen.

Ein Verbesserungsvorschlag ist die Erstellung eines interaktiven Prototyps vor der eigentlichen Entwicklung, um Funktionalitäten testen und wichtige Kriterien verdeutlichen zu können. Die frühzeitige Integration von Usability-Richtlinien würde die Nutzerbedürfnisse besser berücksichtigen. Besonders die Berücksichtigung von Usability-Richtlinien ermöglicht es, die Bedürfnisse der Nutzer klarer herauszuarbeiten und festzulegen, welche Anforderungen das System erfüllen muss.

Dies konnte durch die Entwicklung der Web-Oberfläche simuliert werden. Durch das Testen der Web-Oberfläche konnten einige Funktionen integriert werden, um die Nutzererfahrung angenehmer zu gestalten. So kann man sich beispielsweise die Buchmessen anzeigen lassen, an denen ein Autor teilnimmt, die Warteliste einer Buchmesse einsehen und nach weiteren Suchbegriffen Buchmessen suchen. Insgesamt hat die Entwicklung der Web-Oberfläche auch Defizite offenbart, die behoben werden konnten. Beispielsweise übernimmt das System nun die Verwaltung der Teilnehmer bei einer Buchmesse automatisch. Wenn die Teilnehmeranzahl reduziert wird, werden die betroffenen Teilnehmer automatisch auf die Warteliste gesetzt. Außerdem wurde die Möglichkeit des Ausloggens implementiert.

Insgesamt konnte durch die Entwicklung der Anwendung erlernt werden, wie klare Schnittstellen und klare Trennungen zwischen den einzelnen Komponenten der

Software geschaffen und sinnvoll eingesetzt werden können. Auch der Einsatz von gängigen Bibliotheken und das Absichern der API mit Sicherheitsmechanismen konnten vertieft werden.

5.3 Ausblick

Die Nutzung einer API bietet zahlreiche Vorteile, dennoch würde die Erweiterung der API auf die dritte Stufe des RMM die Gestaltung des Frontends erheblich vereinfachen, da Interaktionslinks nicht manuell generiert werden müssen, sondern von der API bereitgestellt werden. Durch diese Plattformunabhängigkeit wird es ermöglicht eine Webanwendung zu entwickeln, die anschließend mit Hilfe von Frameworks auf mobile Geräte übertragen werden kann.

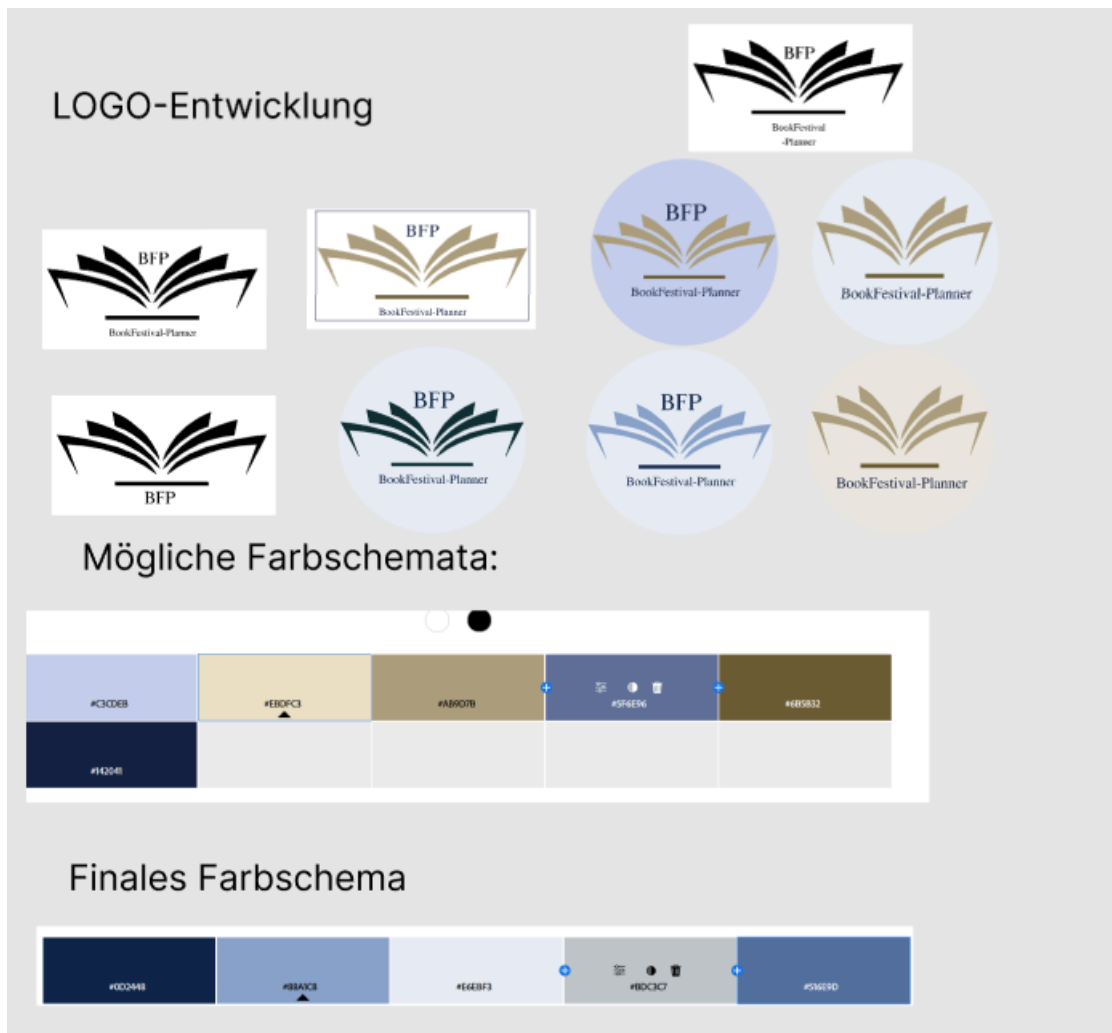
6 Referenzen

- [1] I. E. Lars Röwekamp, „entwickler,“ 02 07 2024. [Online]. Available: <https://entwickler.de/software-architektur/software-architektur-entwicklung>. [Zugriff am 15 07 2023].
- [2] „ieeexplore,“ 15 07 2023. [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=159342>.
- [3] R. C. Martin, Clean Architecture - A Craftsman's Guide to Software Structure and Design, 2017.
- [4] F. Listing. [Online]. Available: https://www.microconsult.de/blog/2019/05/fl_solid-prinzipien/. [Zugriff am 02 07 2024].
- [5] V. Vernon, Domain-Driven Design kompakt, Heidelberg: dpunkt.verlag, 2017.
- [6] F. M. A. Avram, Domain-Driven Design, USA, 2006.
- [7] „vaclavkosar,“ [Online]. Available: <https://vaclavkosar.com/software/Boundary-Control-Entity-Architecture-The-Pattern-to-Structure-Your-Classes>. [Zugriff am 03 07 2024].
- [8] R. Martin, „blog.cleancoder,“ 13 08 2012. [Online]. Available: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>. [Zugriff am 28 06 2024].
- [9] „c4model,“ [Online]. Available: <https://c4model.com/>. [Zugriff am 10 07 2024].
- [10] „Ronja Hoffman,“ [Online]. Available: <https://www.usability.de/leistungen/methoden/personas.html>. [Zugriff am 03 07 2024].
- [11] D. A. Kang, „spiegel-institut,“ [Online]. Available: <https://www.spiegel-institut.de/kompetenzen/usability-engineering/personas>. [Zugriff am 04 07 2024].
- [12] „cybay,“ [Online]. Available: <https://www.cybay.de/blog/die-8-goldenen-regeln-fuer-interface-design>. [Zugriff am 11 07 2024].
- [13] „quarkus,“ [Online]. Available: <https://quarkus.io/guides/smallrye-fault-tolerance>. [Zugriff am 18 07 2024].
- [14] M. Fowler. [Online]. Available: <https://martinfowler.com/articles/richardsonMaturityModel.html>. [Zugriff am 16 07 2024].
- [15] „it-agile,“ [Online]. Available: <https://www.it-agile.de/agiles-wissen/agile-entwicklung/unit-tests/>. [Zugriff am 18 07 2024].
- [16] „postman,“ [Online]. Available: <https://www.postman.com/api-platform/api-versioning/>. [Zugriff am 16 07 2024].

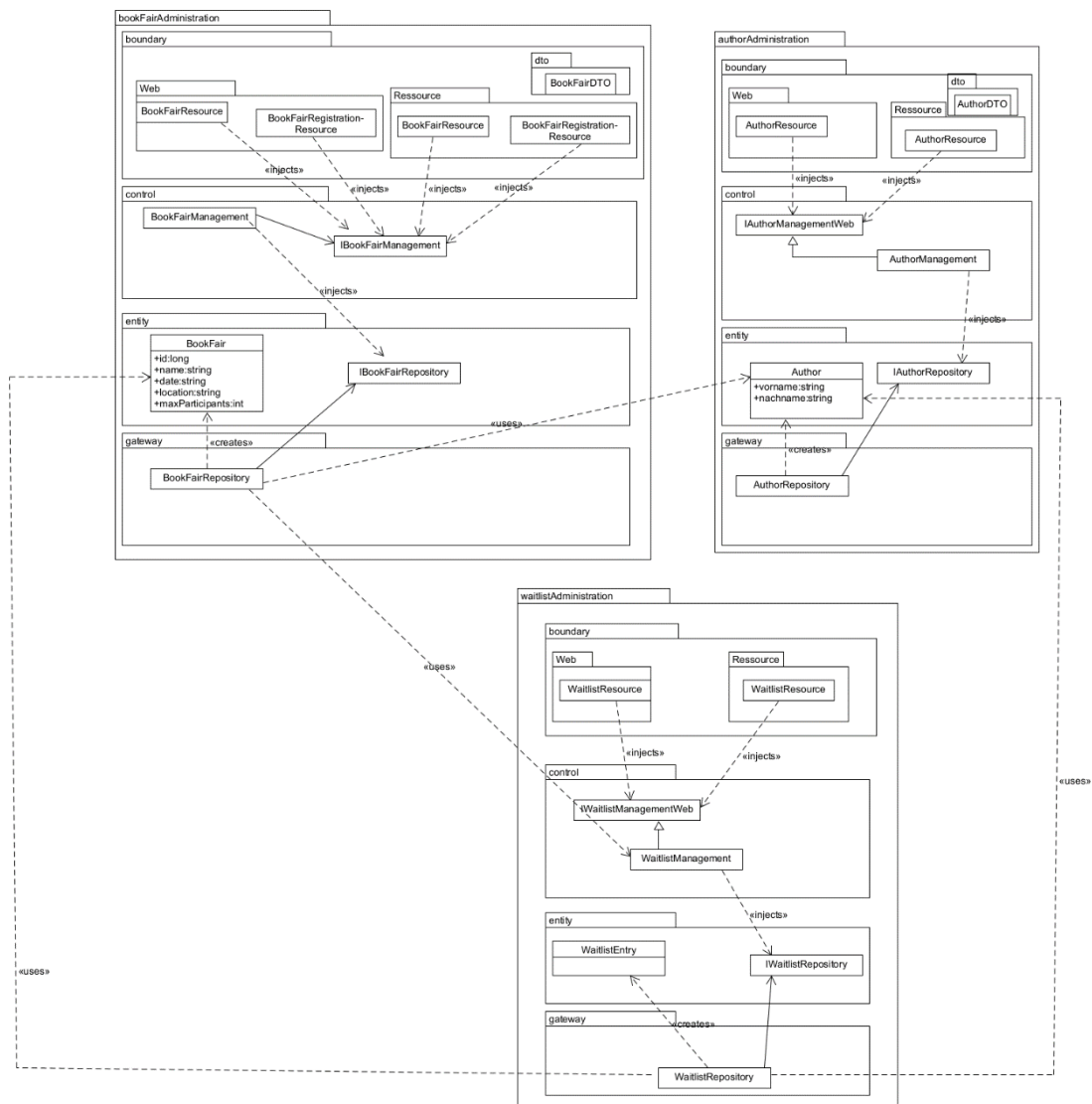
- [17] „vaclavkoslar,“ [Online]. Available: <https://vaclavkosar.com/software/Boundary-Control-Entity-Architecture-The-Pattern-to-Structure-Your-Classes>. [Zugriff am 15 07 2023].
- [18] D. H. e. al., „<https://bmu-verlag.de/software-entwurfsmuster/>,“ 02 07 2024. [Online].
- [19] „logistikknowhow,“ [Online]. Available: <https://logistikknowhow.com/it-und-software/definition-des-software-architektur-prinzips-clean-architecture/>. [Zugriff am 26 06 2024].
- [20] iso25000, „<https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>,“ [Online]. [Zugriff am 15 07 2024].

7 Anhang


Anhang 1: Logo und Farbschema



Anhang 2: Klassendiagramm



Anhang 3: Primärpersona



Lisa Schreiber

- Alter: 28
- Beruf: Marketing Managerin
- Wohnort: Hamburg
- Geschlecht: weiblich

Biographie


Lisa Schreiber arbeitet als Marketing Managerin in einem Verlag. Sie will sich über aktuelle Trends und Neuigkeiten in der Buchbranche informieren. Sie hat einen Abschluss in Betriebswirtschaftslehre mit Schwerpunkt Marketing Management und ist seit etwa vier Jahren in dieser Position tätig. Ihre Arbeit erfordert ein hohes Maß an Organisation und Effizienz, da sie für die Planung und Umsetzung verschiedener Projekte verantwortlich ist

Persönlichkeit

introvertiert	<div style="width: 80%;"></div>	Extrovertiert
Analytisch	<div style="width: 40%;"></div>	Kreativ
ausgelastet	<div style="width: 10%;"></div>	Zeitreich
chaotisch	<div style="width: 60%;"></div>	ordentlich
einzelgänger	<div style="width: 85%;"></div>	Team player
passiv	<div style="width: 80%;"></div>	aktiv
sicher	<div style="width: 20%;"></div>	riskant

<p>Interesse</p> <p>Lisa zeigt Interesse an effizienten und gut organisierten Tools, die ihr helfen, komplexe Projekte zu planen und umzusetzen. Sie ist daran interessiert herauszufinden, wie das Tool ihr dabei helfen kann, Buchmessen und die Teilnahme von Autoren effizient zu verwalten</p>	<p>Einflüsse</p> <p>Ihre Arbeit als Projektmanagerin und ihre Ausbildung im Bereich Betriebswirtschaftslehre beeinflussen ihre Erwartungen an das Tool. Sie legt Wert auf Funktionen, die ihr helfen, Ressourcen effektiv zu verwalten und Zeit zu sparen.</p>	<p>Motivation und Ziele</p> <div style="display: flex;"> <div style="flex: 1;"> <p>Motivation: Durch den Einsatz des "BookFestival-Planners" ihre Projektmanagement-Aufgaben besser zu erfüllen und dabei Zeit zu sparen.</p> </div> <div style="flex: 1;"> <p>Ziele: Organisation und Koordination von Buchmessen und Autoren effizienter und transparenter zu gestalten.</p> </div> </div>
<p>Bedürfnisse und Erwartungen</p> <p>Lisa benötigt ein Tool, das benutzerfreundlich ist und ihr eine klare Übersicht über kommende Buchmessen sowie die Möglichkeit bietet, Autoren für diese Messen einzutragen und zu verwalten. Sie erwartet, dass das Tool ihr hilft, den Überblick zu behalten und administrativen Aufwand zu reduzieren.</p>	<p>Einstellung zum Produkt</p> <p>Lisa wird das Produkt wahrscheinlich positiv gegenüberstehen, wenn es ihre Arbeitsabläufe verbessert und ihr ermöglicht, Buchmessen und Autoren effizient zu koordinieren. Sie könnte skeptisch sein, wenn das Tool unübersichtlich oder umständlich in der Anwendung ist.</p>	<p>Herausforderungen und Unannehmlichkeiten</p> <p>Lisa wird vor der Herausforderung stehen, sicherzustellen, dass das Tool ihren Anforderungen und denen ihres Teams gerecht wird. Sie wird darauf abzielen, dass das Tool insbesondere die Funktionalitäten für Anmeldung, Verwaltung von Buchmessen und Autoren sowie die Wartelisten-Funktion.</p>

Anhang 4: Sekundärpersona



Emma Prante

- Alter: 39
- Beruf: Freiberufliche Autorin
- Wohnort: Hannover
- Geschlecht: weiblich


Biographie

Emma ist Hauptberuflich Psychologin und arbeitet zusätzlich als freiberufliche Autorin. Sie möchte die Vermarktung ihrer Bücher stärker priorisieren. Dafür will sie an Buchmessen teilnehmen, um ihre Bücher vorzustellen und weiter zu vermarkten. Sie will sich über Termine und Veranstaltungsorte von Buchmessen informieren und herausfinden, wie viele Plätze auf den Messen noch verfügbar sind und an welchen Orten die Buchmessen stattfinden.

Persönlichkeit

introvertiert	Extrovertiert
Analytisch	Kreativ
ausgelastet	Zeitreich
chaotisch	eordentlich
einzelgänger	Team player
passiv	aktiv
sicher	riskant

Anhang 5: Antipersona



Alex Maiere

- Alter: 68
- Beruf: Buchhalter
- Wohnort: Moers
- Geschlecht: männlich

Biographie

Alex ist Buchhalter in einem Industrieunternehmen. Er hat kein Interesse an der Buchbranche oder Buchmessen. Dem entsprechend benötigt er die Anwendung nicht für berufliche Zwecke. Er würde die Anwendung nur aus Neugier einmal starten und danach nicht mehr verwenden.

Persönlichkeit

introvertiert	Extrovertiert
Analytisch	Kreativ
ausgelastet	Zeitreich
chaotisch	eordentlich
einzelgänger	Team player
passiv	aktiv
sicher	riskant

Anhang 6: Finale Struktur der API

/api/v1/bookFair	/api/v1/author	/api/v1/registration	/api/v1/waitlist
Alle Buchmessen abrufen: GET /api/v1/bookFair	Alle Autoren abrufen: GET /api/v1/author		
Eine Buchmesse abrufen: GET /api/v1/bookFair/{id}	Einen Autor abrufen: GET /api/v1/author/{id}		Warteliste einer Buchmesse: GET /api/v1/waitlist/{bookFairId}
Buchmesse nach Namen suchen: GET /api/v1/bookFair/search/?s=In Bologna	Buchmessen, an denen ein Autor teilnimmt: GET /api/v1/author/{authorId}/bookfairs		
Buchmesse hinzufügen: POST /api/v1/bookFair	Autor hinzufügen: POST /api/v1/author	Registrierung eines Autors zur Buchmesse: POST /api/v1/registration/{bookFairId}/authors/{authorId}	
Buchmesse bearbeiten: PUT /api/v1/bookFair/{id}	Autor bearbeiten: PUT /api/v1/author/{id}		
Buchmesse löschen: DELETE /api/v1/bookFair/{id}	Autor löschen: DELETE /api/v1/author/{id}	Austragen eines Autors einer Buchmesse: DELETE /api/v1/registration/{bookFairId}/authors/{authorId}	Autor aus Warteliste löschen: DELETE /api/v1/waitlist/{bookFairId}/remove/{authorId}
Teilnehmer der Buchmesse: GET /api/v1/bookFair/{bookFairId}/participants			

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche einzeln kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Münster, 21.07.2024

Ort, Datum



Unterschrift