



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

SVILUPPO E TESTING DI
UN'APPLICAZIONE WEB CON SPRING
BOOT

DEVELOPMENT AND TESTING OF A WEB
APPLICATION WITH SPRING BOOT

VANESSA MARRAMI

Relatore: *Lorenzo Bettini*

Anno Accademico 2022-2023

INDICE

1	Introduzione	7
2	Nozioni di base	9
2.1	Sito web	9
2.2	Pattern Model View Controller	10
2.3	REST	11
2.4	JSON	13
2.5	Spring Framework	14
2.5.1	Spring e Inversion Of Control	15
2.5.2	Spring Boot	15
2.6	Hibernate	15
3	Analisi dell'applicazione	17
4	Progettazione	19
4.1	Setup Database	19
4.1.1	MySQL	19
4.1.2	Docker	20
4.2	Creazione del progetto Spring Boot	22
5	Implementazione	25
5.1	Livello di Dominio	26
5.1.1	Metodi equals() e hashCode()	28
5.1.2	Relazione molti a molti tra Book e Author	30
5.1.3	Implementazione classi DTO (Data Transfert Object)	32
5.2	Livello di Persistenza	34
5.2.1	Connessione al database MySQL	35
5.3	Livello di Servizio	36
5.3.1	Transazioni in Spring Boot	40
5.3.2	Eccezioni personalizzate: NotFoundException e DuplicateException	43
5.4	Livello Web o Consumer	43
5.4.1	Componente @RestController	43
5.4.2	Componente @Controller	47
5.4.3	Viste (View)	52
6	Validazione e Sicurezza	57
6.1	Validazione degli Input	57
6.2	Spring Security	61

7	Testing	67
7.1	Livello di dominio	68
7.2	Livello di persistenza	69
7.3	Livello di servizio	74
7.4	Livello Web o Consumer	76
7.4.1	Test componenti @RestController	76
7.4.2	Test componenti @Controller	78
7.4.3	Test delle viste	81
7.4.4	Test di integrazione	85
7.4.5	Integrare lo strato di persistenza e di servizio	85
7.4.6	Integrare le componenti REST	87
7.4.7	Integrare le componenti Controller	89
7.5	Coverage	91
8	Risultati ottenuti	93
9	Conclusioni	97
9.1	Vantaggi e Svantaggi	97
9.2	Considerazioni e Conclusioni	98
10	Ringraziamenti	103

ELENCO DELLE FIGURE

Figura 1	Browser web mondiale NeXT creato da Tim Berners-Lee.	8
Figura 2	Interazione tra i componenti pattern MVC	11
Figura 3	Modello generale di una API REST	12
Figura 4	Panoramica di Spring Framework	14
Figura 5	Relazione molti a molti	20
Figura 6	File docker-compose.yml	21
Figura 7	Comando docker compose up	22
Figura 8	Comando docker exec -ti	22
Figura 9	Creazione del progetto Spring Boot	23
Figura 10	Spring Starter Project Dependencies	23
Figura 11	File pom.xml	24
Figura 12	Livelli dell'architettura Spring Boot	25
Figura 13	Interazione tra i diversi livelli di un'applicazione <i>Spring Boot</i>	26
Figura 14	File application.yml	36
Figura 15	Esempio senza <i>@Transactional</i>	42
Figura 16	Esempio con <i>@Transactional</i>	42
Figura 17	Esempio di gestione delle eccezioni nelle classi Controller	52
Figura 18	Esempio di convalida non rispettata	61
Figura 19	Messaggio di errore login credenziali errate	65
Figura 20	Definizione di message in application.yml	65
Figura 21	Bottoni di Login e Logout	66
Figura 22	databaseH2.properties	69
Figura 23	Coverage	92
Figura 24	Generazione delle tabelle nel database	93
Figura 25	Home page	93
Figura 26	Consultazione del catalogo senza autenticazione	94
Figura 27	Esempio di inserimento nel catalogo	94
Figura 28	Consultazione del catalogo con autenticazione	95
Figura 29	Istruzioni per l'inserimento nella console	95

*"C'è vero progresso soltanto quando i vantaggi di una nuova tecnologia
diventano per tutti."*
— Henry Ford

1

INTRODUZIONE

"Il WorldWideWeb (WWW) è un'iniziativa di recupero di informazioni ipermediali ad ampio raggio che mira a dare accesso universale a un vasto universo di documenti" [1].

Il web è nato negli anni 90, al centro di ricerca del CERN, per soddisfare la necessità di avere un sistema che permetesse lo scambio e la consultazione di articoli scientifici. Nasce come un *progetto ipertestuale* chiamato *WorldWideWeb* (rete di ampiezza mondiale) in cui una *rete di documenti ipertestuali* poteva essere visualizzata dai *browser*.

Alla fine del 1990, Tim Berners-Lee etichettò un computer del CERN con una scritta rossa "This machine is a server. DO NOT POWER IT DOWN!!". Fu così che Tim Berners-Lee sviluppò il codice per il suo server web su un computer *NeXT* e *info.cern.ch* fu il primo sito web al mondo, accessibile anche oggi. La pagina conteneva informazioni sul progetto del WWW, sulla descrizione dell'ipertesto, sulla creazione di server web e collegamenti ad altri server web.

La Figura 1 mostra il browser web *NeXT* creato da Tim Berners-Lee.

Il browser funzionava solo sul sistema operativo *NeXT*, ma era fondamentale che il web fosse accessibile a molti tipi di computer. Questo ostacolo fu superato poco dopo da Nicola Pellow che sviluppò il '*line-mode browser*', il primo browser web che potesse essere installato su tipi diversi di computer. Con il '*line-mode browser*' era possibile visualizzare solo testo e si poteva usare solo tramite la tastiera.

Il WWW è diventato un enorme successo e la sua invenzione fu subito una rivoluzione che cambiò radicalmente la vita e la quotidianità. Oggi, è uno strumento fondamentale ed il più usato per la comunicazione e per la ricerca di informazioni. Dall'invenzione del primo sito web ad oggi, lo sviluppo web ha avuto una forte evoluzione, comportando la diffusione di nuovi strumenti e linguaggi.

In questa tesi, dopo un'introduzione sui concetti necessari per la comprensione del lavoro svolto, viene descritta l'applicazione web realizzata, la sua architettura, le componenti necessarie per il suo sviluppo, la sua implementazione, i test effettuati ed, infine, vengono mostrati i risultati ottenuti. Gli strumenti principalmente utilizzati per lo sviluppo sono:

- *Docker Desktop*, un'applicazione che consente di creare, condividere ed eseguire contenitori, applicazioni e immagini. Permette di visualizzare i contenitori disponibili e di gestire il ciclo di vita delle applicazioni direttamente dal proprio sistema. [2]
- *DBeaver*, è un set di plug-in, funzionalità e prodotti *Eclipse* multi-piattaforma gratuito per sviluppatori, amministratori di database, analisti e chiunque lavori con i dati.[3]
- *Eclipse Spring Tool Suite (STS)*, un IDE per lo sviluppo di applicazioni aziendali basate su *Spring*. Fornisce un ambiente pronto all'uso per implementare, eseguire il debug, eseguire e distribuire applicazioni e servizi su Java. [4]
- *Postman*, una piattaforma per la creazione e l'utilizzo di API. Permette di creare ed inviare richieste e di testare le funzionalità di un'API. Gestisce diversi tipi di formati di risposta come *JSON*, *XML*, *HTML* e testo semplice. [5]

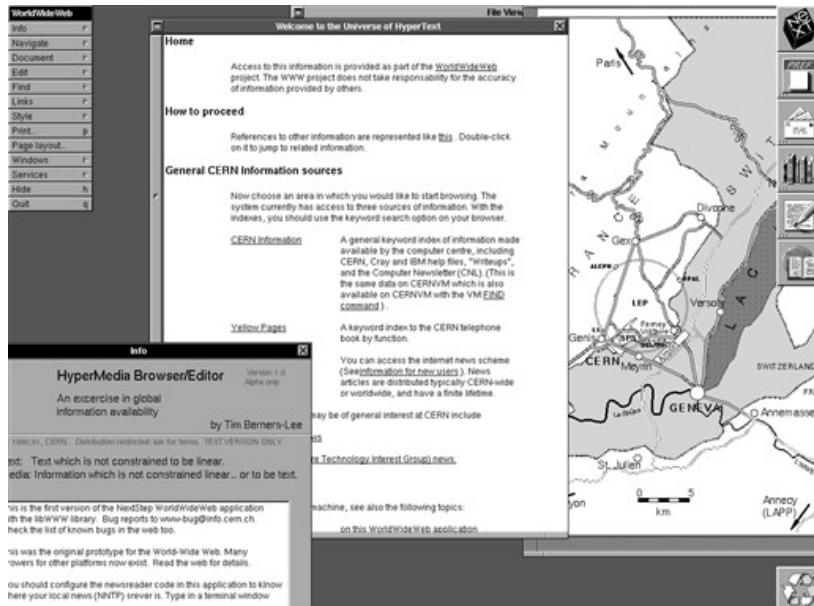


Figura 1: Browser web mondiale NeXT creato da Tim Berners-Lee.

2

NOZIONI DI BASE

Di seguito vengono riportate delle informazioni per la comprensione degli argomenti trattati nella presente tesi.

2.1 SITO WEB

Un sito web (o un applicazione web) è un insieme di pagine web opportunamente correlate tra di loro attraverso dei link e raggiungibili attraverso un specifico dominio.

Le informazioni sul web vengono gestite dal protocollo HTTP (*HyperText Transfer Protocol*). Esso è *stateless*, senza memoria, basato sul meccanismo di richiesta/risposta tra client e server, in cui il client esegue una richiesta (*HTTP request*) e il server restituisce una risposta (*HTTP response*). Quindi, il client (browser) ed il server devono essere in grado di gestire il protocollo HTTP per lo scambio di informazioni.

Il messaggio di richiesta (*HTTP request*) è costituito da:

- *request line*: composto da metodo (GET, POST ...) e dall' URI (*Uniform Resource Identifier*), ovvero una sequenza di caratteri che identifica in modo univoco una risorsa.
- *handler*: contiene un insieme di informazioni, ad esempio l'host (server a cui si riferisce l'URL), browser utilizzato, ecc.
- *body*: corpo del messaggio.

Il messaggio di risposta (*HTTP response*) è costituito da:

- *status line*: un codice a tre cifre che indica l'esito della richiesta del client.
- *handler*: contiene un insieme di informazioni, ad esempio il tipo, la versione del server ed il tipo del contenuto restituito.

- *body*: corpo della risposta.

Nello specifico, quando il client esegue una *request line* viene indicato il metodo da eseguire sulla risorsa identificata dall' URI. I metodi più comunemente utilizzati sono:

- *GET*, che ha la funzione di richiedere una risorsa a un web server.
- *POST*, per inviare quantità di dati al server allegandoli all' header HTTP.
- *OPTIONS*, per richiedere l'elenco dei metodi permessi dal server.
- *HEAD*, che ha la funzione di richiedere solo l'header senza la risorsa (file HTML, l'immagine, ecc).
- *PUT*, per aggiornare le informazioni già presenti sul server.
- *DELETE*, per cancellare una risorsa sul server.

Nella risposta del server, lo *status line* notifica al client se il metodo è consentito sulla risorsa oppure no.

Per sviluppare un'applicazione web moderna, in grado di fornire delle funzionalità avanzate e una maggiore esperienza interattiva all'utente, è opportuno l'utilizzo di almeno un linguaggio *Server side*, codice eseguito ed interpretato dal server, ed un linguaggio *Client side*, interpretato dal browser. Oggi, esistono molteplici linguaggi *Server side* che permettono lo sviluppo di un'applicazione web che interagisca dinamicamente con il client, generando un flusso di codice HTML (*HyperText Markup Language*, linguaggio utilizzato per scrivere pagine web) oppure di altro codice (ad esempio *JSON* o *XML*) che viene restituito al client.

I linguaggi *Server side* più comuni sono: *Java*, *PHP*, *Python* e *.NET*.

I linguaggi *Client side* più comuni sono: *Javascript* e *HTML*.

2.2 PATTERN MODEL VIEW CONTROLLER

Un *pattern* rappresenta un modello per la risoluzione ad un problema ricorrente. Nell'ambito dello sviluppo di applicazioni web, *Model-View-Controller (MVC)* è uno dei pattern architetturali più comunemente usati.

Un'applicazione sviluppata con il pattern MVC avrà tre componenti:

- *Model*, implementa i metodi per l'accesso ai dati e si interfaccia con il database.
- *View*, gestisce le componenti visualizzate dall'utente.
- *Controller*, gestisce le richieste dell'utente attraverso le *view* ed esegue operazioni che possono portare ad un cambiamento di stato della *view*. È l'intermediario tra *model* e *view*, il cui compito è gestire la URL richiesta dall'utente.

Esso permette la netta separazione tra la *business logic* (logica applicativa dell'applicazione) e la logica di presentazione dei dati (le pagine viste dall'utente).

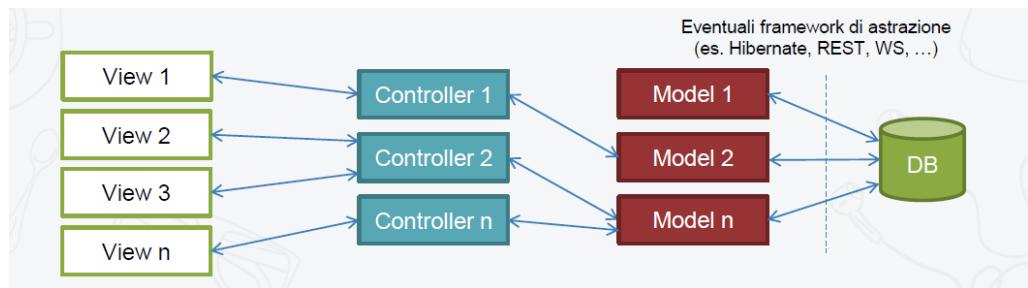


Figura 2: Interazione tra i componenti pattern MVC

2.3 REST

Nel 2000, Roy Thomas Fielding, uno dei principali architetti nella definizione del protocollo HTTP, introduce il concetto di *Representational State Transfer (REST)* definendolo come uno stile architettonico che si basa sul concetto di *Separation Of Concerns (SoC)* [6], ovvero sulla suddivisione di un sistema in moduli, in cui ogni modulo svolge una funzione specifica. REST applica il principio SoC ad un'architettura client-server consentendo lo sviluppo indipendente lato server e lato client.

In un'architettura REST, una risorsa è una rappresentazione di qualcosa di significativo nel dominio applicativo ed essa può essere rappresentata in diversi formati, i più utilizzati sono *Json* o *Xml*. Si accede ad essa, o ad un insieme di esse, tramite un'interfaccia uniforme basata sui metodi HTTP. Con il termine RESTful viene rappresentato un servizio web che implementa l'architettura REST.

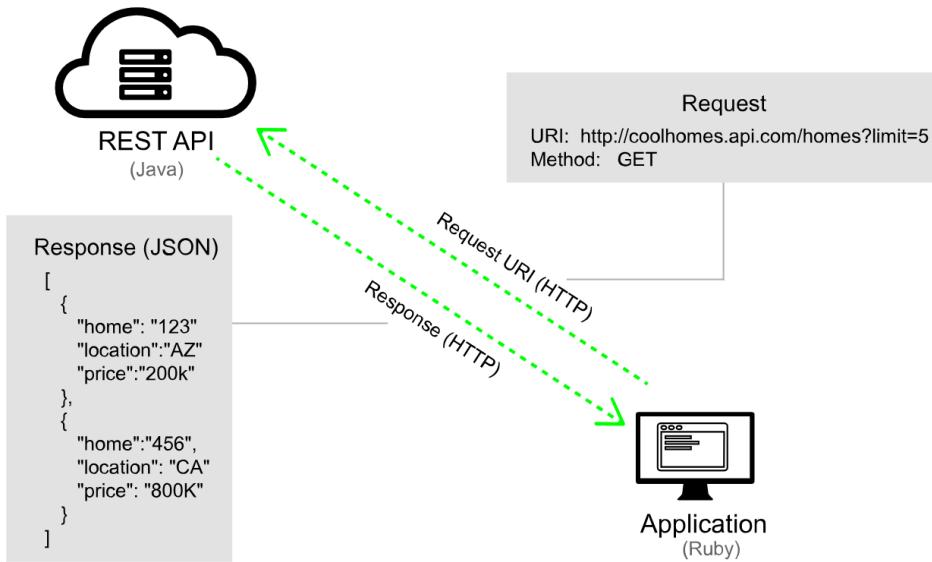


Figura 3: Modello generale di una API REST

La comunicazione client-server è *stateless*, quindi, ogni richiesta del client deve contenere tutte le informazioni necessarie per l'utilizzo di un servizio messo a disposizione dal server. Infatti, ogni richiesta prevede l'utilizzo di URL ben specifici per una risorsa, od un insieme di risorse, e l'utilizzo dei metodi standard HTTP (GET, POST, ecc).

Dopo aver ricevuto la richiesta, il server può eseguirla o respingerla. L'esito della risposta viene rappresentato con un codice di stato HTTP che assume un determinato significato in base alle seguenti categorie:

- *1xx*, codice informativo.
- *2xx*, codice di successo.
- *3xx*, codice di reindirizzamento.
- *4xx*, codice di errore del client.
- *5xx*, codice di errore del server.

I vantaggi derivati dall'utilizzo di un servizio RESTful sono:

- *Indipendenza*, le API REST sono indipendenti dai linguaggi o dal tipo di piattaforme utilizzate. Si ha, dunque, la libertà di scelta sul linguaggio e framework da utilizzare nell'implementazione.

- *Separazione client-server*, consente lo sviluppo indipendente delle due componenti.
- *Scalabilità*, la modifica o l'aggiunta di funzionalità può avvenire con facilità ed in modo indipendente.

2.4 JSON

Nella sezione precedente, è stata descritta l'architettura REST per lo sviluppo delle applicazioni che permettono l'interscambio di dati fra client e server. È possibile perché i dati vengono inviati o ricevuti come risorsa attraverso un formato specifico.

Json (JavaScript Object Notation) è uno dei principali formati che permette l'interscambio dati. È leggero e facile da leggere, i dati restituiti in tale formato, si presentano come testo e sono, quindi, compatibili con la maggior parte dei moderni linguaggi di programmazione (es. *Python*, *Java*, *PHP*, *JavaScript*, ecc).

```

1  {
2      "isbn": "978-88-04-50998-1",
3      "title": "La serie infernale",
4      "genre": "Crime",
5      "plot": "Hercule Poirot riceve una strana lettera in cui un anonimo
           personaggio lo avvisa che presto, nella città di Andover, verrà
           commesso un crimine. E infatti Alice Ascher viene uccisa. Dopo di
           lei, il misterioso assassino annuncia che colpirà a Bexhill, e poi
           a Churston, e così fa. Le vittime sembrano non aver alcun rapporto
           tra loro: il diabolico criminale, che si firma A.B.C., pare
           sceglierle solo in base a un presunto ordine alfabetico.",
6      "publicationDate": "2018-07-04",
7      "numberPages": 244,
8      "authors": [
9          {
10             "id": 2,
11             "firstName": "Agatha",
12             "lastName": "Christie"
13         }
14     ]
15 }
```

Dunque, in un'applicazione RESTful, dal lato server è possibile ottenere le risorse in un formato specifico, come *Json*, e tramite il lato client è possibile consumare e visualizzare i dati ottenuti.

2.5 SPRING FRAMEWORK

"Spring Framework è una piattaforma Java che fornisce un supporto infrastrutturale completo per lo sviluppo di applicazioni Java. Spring gestisce l'infrastruttura in modo che tu possa concentrarti sulla tua applicazione [7]."

Spring è composto da circa 20 moduli che possono essere utilizzati in base alle esigenze dell'applicazione:

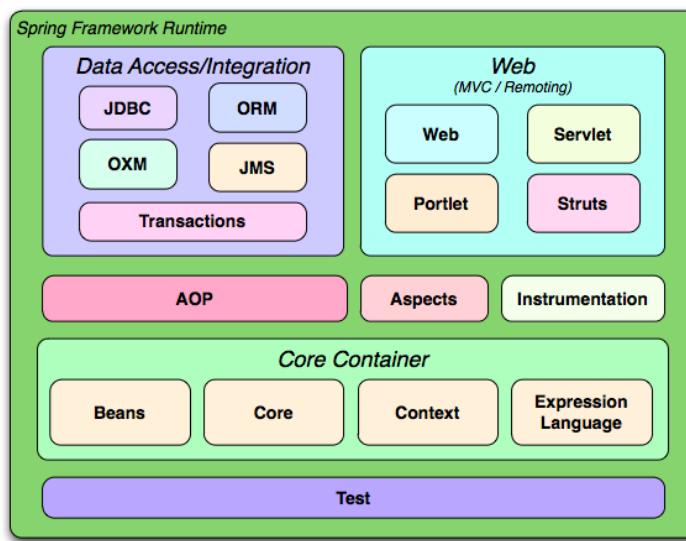


Figura 4: Panoramica di Spring Framework

Dunque, *Spring* è un ecosistema di framework che aiuta nello sviluppo di applicazioni di diversa natura, ad esempio:

- *Spring Core*, parte fondamentale di *Spring* con la quale è possibile gestire il *context*.
- *Spring MVC*, framework che permette lo sviluppo di applicazioni web.
- *Spring Data*, framework che permette l'interazione delle applicazioni web con i database.
- *Spring Security*, framework che gestisce la sicurezza dell'applicazione web.

2.5.1 Spring e Inversion Of Control

Spring è basato sul principio dell'*Inversion Of Control*, "un processo in base al quale gli oggetti definiscono le loro dipendenze, ovvero gli altri oggetti con cui lavorano, solo tramite argomenti del costruttore, argomenti a un metodo factory o proprietà che vengono impostate sull'istanza dell'oggetto dopo che è stata costruita o restituita da un metodo factory. Il contenitore quindi inietta tali dipendenze quando crea il *bean*. I pacchetti *org.springframework.beans* e *org.springframework.context* sono la base per il contenitore *IoC* di *Spring Framework*".[8]

Per creare un progetto *Spring*, è possibile utilizzare *Apache Maven*, un tool di *Apache Software* che facilita la gestione e lo sviluppo di progetti basati su Java. Il *Maven* può essere configurato con un file *.xml* che permette di specificare le informazioni del progetto e i dettagli di configurazione per la costruzione dell'applicazione stessa.

2.5.2 Spring Boot

Spring Boot è un framework che facilita la creazione di applicazioni basate su *Spring* [9]. Esso infatti ha le seguenti caratteristiche:

- *Gestione semplificata delle dipendenze*, ha la capacità di selezionare automaticamente delle dipendenze per delle funzionalità comuni.
- *Configurazione automatica*, configura automaticamente molte delle funzionalità di *Spring Framework*.
- *Deployment semplificato*, permette lo sviluppo di web app *stand alone* pronte all'uso. Quindi, i progetti *Spring Boot* creano un file JAR, i quali hanno tutti gli elementi necessari per il funzionamento. Non c'è bisogno di un server, ad es. *Tomcat*, perché già incluso nel file JAR.

2.6 HIBERNATE

La programmazione ad oggetti permette di rappresentare delle entità del mondo reale in termini di oggetti e relazioni tra di essi. Sviluppare un software orientato agli oggetti che interagisce con un database può essere complesso in quanto è necessario memorizzare i dati in un database, che non è orientato agli oggetti [10]. Dunque, è necessario mappare i dati che

rappresentano un oggetto e le relazioni tra classi in un insieme di tabelle, questa tecnica è chiamata *Object/Relational Mapping (ORM)*.

Hibernate è una soluzione *Object/Relational Mapping (ORM)* per ambienti Java e fornisce anche funzionalità per il recupero dei dati. È un framework Java che facilita lo sviluppo di applicazioni che interagiscono con un database riducendo il tempo di sviluppo e liberando lo sviluppatore dalla gestione relativa alla persistenza dei dati. Non è necessaria una conoscenza in *SQL*, ma è utile per comprendere *Hibernate*. [11]

3

ANALISI DELL'APPLICAZIONE

Per lo sviluppo di un'applicazione web il primo passo è la raccolta delle informazioni che la caratterizzano.

Si vuole sviluppare un'applicazione web che gestisce i dati di un catalogo di una biblioteca.

È possibile accedere ad esso con o senza registrazione, l'utente non registrato può accedere ai dati, ma non modificarli, mentre l'utente registrato ha anche la possibilità di modificare il catalogo.

L'applicazione, quindi, permette ad un utente registrato di consultare, aggiungere, modificare ed eliminare libri dal catalogo.

Il nome dell'applicazione sviluppata è *My Library* e le sue principali funzionalità sono:

- Accesso alla lista dei libri: un utente ha la possibilità di visualizzare la lista dei libri presenti nel catalogo.
- Accesso ai dettagli di un libro: un utente ha la possibilità di visualizzare dettagli aggiuntivi di un libro della lista, quali ad esempio la trama, numero di pagine e data di pubblicazione.
- Accesso alla lista degli scrittori: un utente ha la possibilità di visualizzare la lista degli autori dei libri presenti nel catalogo.
- Ricerca per titolo: un utente ha la possibilità di ricercare un libro in base al titolo.
- Filtro di ricerca: un utente ha la possibilità di filtrare la lista dei libri in base al genere ed alla data di pubblicazione.
- Ricerca di un autore: un utente ha la possibilità di ricercare un autore in base al nome ed al cognome.

- Login: un utente ha la possibilità di autenticarsi per poter gestire il catalogo.
- Eliminazione di un libro: un utente autenticato ha la possibilità di eliminare un libro presente nel catalogo.
- Inserimento di un libro: un utente autenticato ha la possibilità di inserire un libro e relativi autori (se più di uno) nel catalogo.
- Modifica delle informazioni: un utente autenticato ha la possibilità di modificare le informazioni di un libro o di un autore presente nel catalogo.
- Logout: un utente autenticato ha la possibilità di eseguire il logout.

Poiché le operazioni di modifica dei dati nel catalogo sono privilegiate, un utente non autenticato non è in grado di accedere a tali funzionalità, in quanto i pulsanti che permettono di eseguirle risultano nascosti.

La presenza di un autore nel catalogo è strettamente vincolata alla presenza di un libro scritto da questi. Infatti, è previsto solo l'inserimento di libri nel catalogo e se durante questa operazione lo scrittore non è presente, esso viene inserito automaticamente nella lista degli autori. Altrimenti, il libro viene aggiunto alla lista delle sue opere. In tal modo, non sono mai presenti autori senza libro. Di conseguenza, l'eliminazione di tutti i libri appartenenti ad uno stesso autore, comporta la sua eliminazione.

4

PROGETTAZIONE

In questa sezione, si descrive la struttura del database usato e si introduce l'uso di *Spring Starter Project* per la creazione del software.

4.1 SETUP DATABASE

Per poter implementare il software come prima cosa è necessario un database, ovvero "un insieme di informazioni (o dati) strutturate, in genere archiviate elettronicamente in un sistema informatico" [12]. Tra i database più diffusi vi sono, ad esempio, *MySQL*, *PostgreSQL*, *MariaDB*, *SQL Server*. In questa tesi, per memorizzare e gestire le informazioni dei libri e degli autori è stato usato *MySQL*, poiché è compatibile con molti sistemi operativi.

4.1.1 MySQL

MySQL è un database relazionale *open source*. Il nome del software deriva da *Structured Query Language (SQL)*, linguaggio di programmazione utilizzato per creare, leggere, modificare ed eliminare i dati archiviati (chiamate anche operazioni *CRUD*, *Create*, *Read*, *Update*, *Delete*).

Un database relazionale consente di organizzare i dati in tabelle, formate da righe (record) e colonne (attributi), in cui ciascuna tabella può essere collegata ad un'altra attraverso una relazione.

Ad esempio, per l'applicazione sviluppata in questa tesi si hanno due tabelle:

- *Book*, composto da codice ISBN, titolo, genere, data di pubblicazione, trama, numero di pagine e autori.
- *Author*, composto da codice id, nome e cognome.

Le due tabelle hanno una relazione molti a molti. Infatti, un libro può essere scritto da uno o più autori e un autore può scrivere più di un libro. Nei database relazionali, la relazione molti a molti è possibile tramite l'uso di una terza tabella, che contiene le chiavi primarie (attributi che identificano univocamente un record) delle due tabelle che unisce. In Figura 5, è mostrato un esempio.

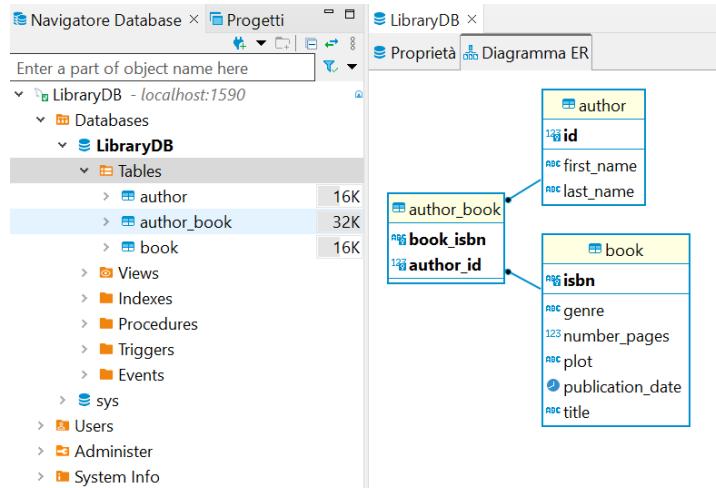


Figura 5: Relazione molti a molti

La costruzione di tale tabella in *Spring Boot* avviene tramite l'annotazione `@JoinTable`, affrontata nel capitolo successivo.

4.1.2 Docker

Docker è una piattaforma per lo sviluppo e l'esecuzione di applicazioni. Consente di separare le applicazioni e le sue componenti impacchettandole in un ambiente isolato, chiamato *contenitore*. Tali contenitori sono leggeri e autosufficienti, dispongono di tutto il necessario per essere eseguiti senza fare affidamento su ciò che è installato sulla macchina host [13].

Dunque, per sfruttare i vantaggi dell'uso del contenitore, in particolare l'isolamento e la portabilità, è stata creata un'immagine *Docker* di un database *MySQL* attraverso il *Docker Compose*, un file in formato *YAML* utilizzato per configurare il servizio.

```

1  version: '3.8'
2
3  services:
4    mysql:
5      image: mysql:8.0
6      restart: unless-stopped
7      container_name: mysqlApplicationDB
8      volumes:
9        - type: bind
10       source: ./dockervol
11       target: /var/lib/mysql
12      ports:
13        - target: 3306
14        published: 1590 #Porta macchina Host.
15        protocol: tcp
16        mode: host
17      environment:
18        MYSQL_ROOT_PASSWORD: LibraryDB
19        MYSQL_DATABASE: LibraryDB
20        MYSQL_USER: sa
21        MYSQL_PASSWORD: Library123
22      volumes:
23        mysql-data:

```

Figura 6: File docker-compose.yml

In Figura 6 è stato configurato un contenitore chiamato *mysqlApplicationDB* in cui è presente un'immagine di un database MySQL. In dettaglio:

- la sezione *service* specifica il nome ed il tipo di servizio.
- la sottosezione *volumes* definisce il meccanismo per la persistenza dei dati. Il volume è di tipo *bind*, montaggio che consente di dividere una directory del *File System* dell'host con il contenitore, garantendo la persistenza dei dati anche dopo averlo arrestato. Il percorso della directory nel computer host è specificato in *source*, viceversa, il percorso della directory montata nel contenitore è specificato in *target*. [14]
- la sottosezione *ports* definisce la porta di esposizione per il contenitore, utilizzata per comunicare con il servizio. Il *docker container* espone il servizio sulla porta 3306, cioè la porta di default per MySQL ed essa è mappata sulla porta 1590 della macchina host. Si accede al servizio eseguito dal container tramite la porta 1590.
- la sottosezione *environment* gestisce le variabili d'ambiente.

Configurato il servizio, il contenitore viene compilato tramite il comando *docker compose up [OPTIONS] [SERVICE...]*. In Figura 7 è presente un esempio:

```
C:\Tesi\Docke Library MySQL>docker compose up -d
[+] Running 2/2
  ▓ Network dockerlibrarymysql_default   Created
  ▓ Container mysqlApplicationDB         Started
                                         0.6s
                                         1.1s

C:\Tesi\Docke Library MySQL>docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
bff0ad8f9b87        mysql:8.0          "docker-entrypoint..."   8 seconds ago     Up 6 seconds      33060/tcp, 0.0.0.0:1590->3306/tcp   mysqlApplicationDB
```

Figura 7: Comando docker compose up

Con *docker ps* è possibile vedere i contenitori in esecuzione e con *docker exec -it <container id> /bash* è possibile connettersi ad un contenitore già attivo:

```
C:\Tesi\Docke Library MySQL>docker exec -ti bff0ad8f9b87 bash
bash-4.4# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.34 MySQL Community Server - GPL

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases
    -> ;
+-----+
| Database |
+-----+
| information_schema |
| LibraryDB |
| mysql |
| performance_schema |
| sys |
+-----+
5 rows in set (0.00 sec)

mysql> ^D
Bye
bash-4.4# exit
C:\Tesi\Docke Library MySQL>
```

Figura 8: Comando docker exec -ti

In Figura 8, dopo l'accesso con le credenziali *root* specificate nel *docker-compose*, è possibile osservare la presenza del database *LibraryDB*.

4.2 CREAZIONE DEL PROGETTO SPRING BOOT

Per la realizzazione dell'applicazione è stato usato *Spring Tool Suite (STS)*, un IDE Java utilizzato per lo sviluppo di applicazioni *Spring*.

La creazione di un progetto può avvenire tramite *Spring Initializr*, accessibile al sito <https://start.spring.io/>, oppure tramite la procedura guidata *Spring Starter Project* usando *Spring Tool Suite*. In questa tesi, il progetto *Spring Boot* è stato creato tramite lo *Spring Starter Project*:

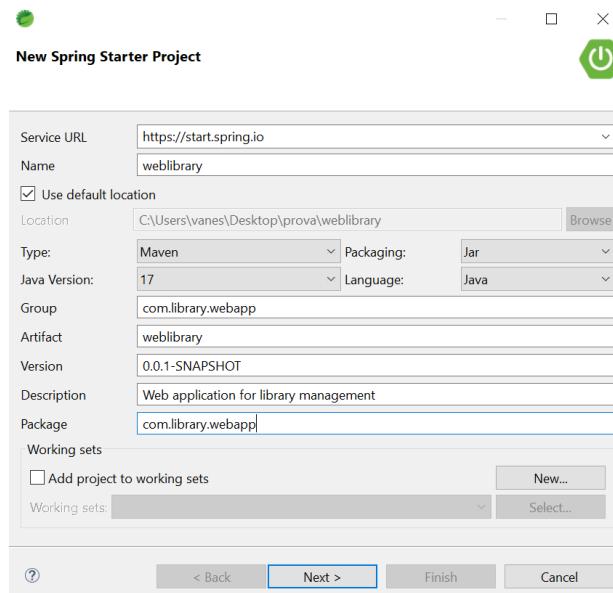


Figura 9: Creazione del progetto Spring Boot

In Figura 9 sono inseriti i dettagli del progetto. Nella finestra successiva, in Figura 10, vengono selezionate la versione di *Spring Boot* e le dipendenze principali per il progetto.

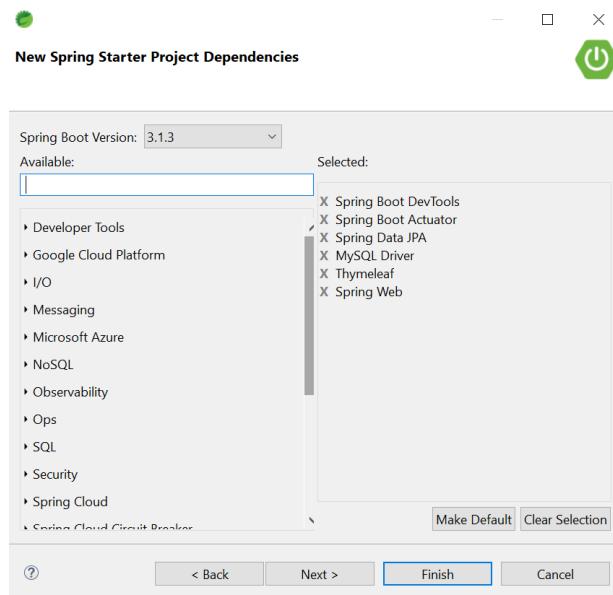


Figura 10: Spring Starter Project Dependencies

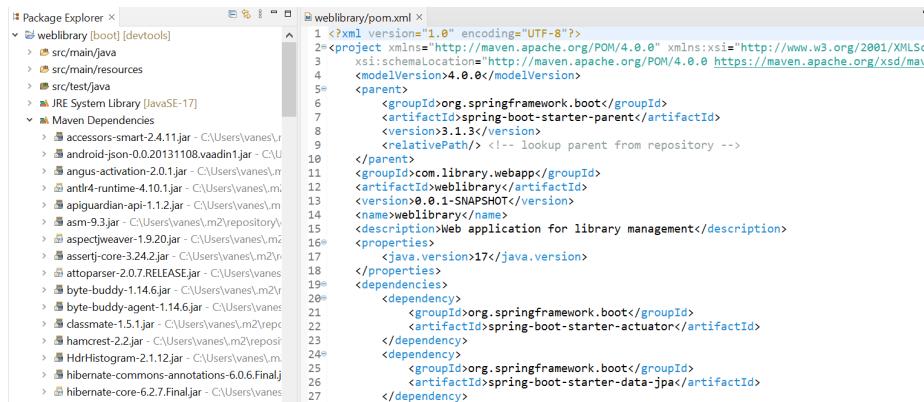
Le dipendenze scaricate sono:

- *Spring Web*, necessario per implementare un'applicazione web.
- *Spring Thymeleaf*, una libreria Java che consente di definire un modello di pagina HTML o HTML5, nella quale vengono inseriti i dati necessari per generare la pagina finale.
- *Spring Boot DevTools*, modulo utilizzato per migliorare i tempi di sviluppo mentre si lavora con l'applicazione *Spring Boot*, rileva le modifiche e riavvia l'applicazione.
- *Spring Boot Actuator*, modulo che permette di monitorare una applicazione *Spring Boot* in modo facile e veloce.
- *Spring Data JPA*, fornisce il supporto del repository per l'*API Jakarta Persistence (JPA)*. Facilita lo sviluppo di applicazioni che devono accedere alle origini dati JPA.
- *MySQL Driver*, modulo per accedere al DB.

Eventuali dipendenze aggiuntive, sono inserite direttamente nel file di configurazione *pom.xml* del progetto.

Cliccato su *Finish*, *Spring Tool Suite* genererà il progetto e scaricherà le dipendenze. Essendo un'applicazione *Spring Boot* un'applicazione Java, nel progetto generato, è presente una classe *main*.

Lo strumento *Spring Starter Project* crea un progetto *Maven* con le dipendenze *Spring Boot* e *Spring Maven*. È presente un file di configurazione chiamato *pom.xml* (*Project Object Model*). La parte più importante di questo file, in Figura 11, è quella che fa riferimento alla versione di *Spring Boot*. Aggiornando la versione, vengono aggiornate a versioni più recenti anche le dipendenze.



```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven_4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.3</version>
    <relativePath><!-- lookup parent from repository --&gt;
  &lt;/parent&gt;
  &lt;groupId&gt;com.library.webapp&lt;/groupId&gt;
  &lt;artifactId&gt;weblibrary&lt;/artifactId&gt;
  &lt;version&gt;0.0.1-SNAPSHOT&lt;/version&gt;
  &lt;name&gt;weblibrary&lt;/name&gt;
  &lt;description&gt;Web application for library management&lt;/description&gt;
  &lt;properties&gt;
    &lt;java.version&gt;17&lt;/java.version&gt;
  &lt;/properties&gt;
  &lt;dependencies&gt;
    &lt;dependency&gt;
      &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
      &lt;artifactId&gt;spring-boot-starter-actuator&lt;/artifactId&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
      &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
      &lt;artifactId&gt;spring-boot-starter-data-jpa&lt;/artifactId&gt;
    &lt;/dependency&gt;
  &lt;/dependencies&gt;
&lt;/project&gt;
</pre>

```

Figura 11: File *pom.xml*

5

IMPLEMENTAZIONE

Un'applicazione *Spring Boot* segue un'architettura a livelli, *layer*, che interagiscono tra loro [15]:

- *Livello di dominio*, rappresenta le entità, attributi, vincoli e relazioni che intercorrono fra di essi. Le classi che appartengono a questo strato sono identificate con `@Entity`.
- *Livello di persistenza*, gestisce l'interscambio e la modifica dei dati con l'applicazione e il database. Le classi che appartengono a questo strato sono identificate con `@Repository`.
- *Livello di servizio*, è composto dalle classi che gestiscono la *business logic* dell'applicazione. Le classi che appartengono a questo strato sono identificate con `@Service`.
- *Livello Web o Consumer*, si occupa di gestire richieste HTTP. Ha il compito di interpretare gli input dell'utente e di fornire la risposta appropriata. Le classi che appartengono a questo strato sono identificate con `@Controller` o `@RestController`.

In Figura 12, è rappresentato il modello architetturale di un'applicazione *Spring Boot*:

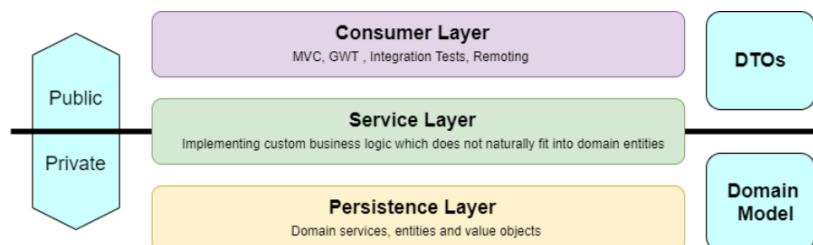


Figura 12: Livelli dell'architettura Spring Boot

In Figura 13 è riportata l'interazione tra i diversi strati:

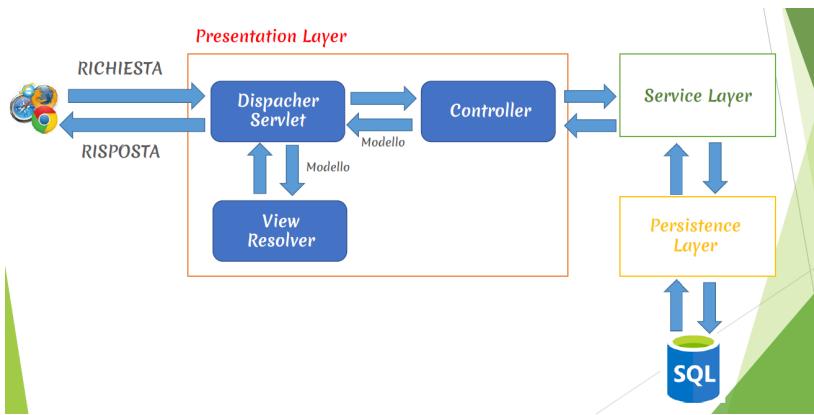


Figura 13: Interazione tra i diversi livelli di un'applicazione *Spring Boot*

Il client richiede un servizio tramite una richiesta HTTP. Essa viene gestita dal *Controller*, che richiama lo strato di servizio, il quale, effettua delle operazioni interagendo con lo strato di persistenza. Dopodiché, il controller restituisce una vista o una risorsa.

In questo capitolo, vengono implementati i diversi livelli dell'applicazione.

5.1 LIVELLO DI DOMINIO

Un'*Entità* è un oggetto di persistenza archiviato nel database. Essa rappresenta una tabella archiviata. Ogni istanza di un'entità rappresenta una riga nella tabella. [16]

Dopo aver creato il progetto, le due classi entità *Book* e *Author* vengono implementate in un apposito package. Per ciascun livello del modello architettonico, viene utilizzato un package apposito, nel caso delle entità si chiama *com.library.webapp.entity*.

Di seguito, è riportata l'implementazione delle classi entità *Book* e *Author*:

```

@Entity
@Table(name = "book")
public class Book {
    @Id
    @Basic(optional = false)
    @Column(unique = true)
    private String isbn;

    @Basic(optional = false)

```

```

10  private String title;
    private String genre;
    @Column(length=2000)
15  private String plot;
    @Column(name = "publication_date")
    @Temporal(TemporalType.DATE)
    private LocalDate publicationDate;
20
    @Column(name = "number_pages", columnDefinition = "int default 0")
    private int numberPages;


---


@Entity
@Table(name = "author")
3 public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

8    @Basic(optional = false)
    @Column(name = "first_name")
    private String firstName;

    @Basic(optional = false)
13   @Column(name = "last_name")
    private String lastName;


---



```

Entrambi le classi entità:

- sono annotate con `@Entity` per indicare che la classe è un entità ed è mappata ad una tabella del database.
- sono annotate con `@Table` per sovrascrivere il nome della tabella con il nome indicato.
- hanno un campo annotato con `@Id` per specificare la chiave primaria dell'entità.
- hanno dei campi annotati con `@Basic(optional = false)` per indicare che la proprietà non può assumere valore null.
- hanno dei campi annotati con `@Column` per specificare dettagli aggiuntivi per una colonna, quali, ad esempio, `name`, `length`, `nullable` e `unique`. L'elemento `name` specifica il nome della colonna nella tabella,

length la lunghezza, *nullable* se la colonna è *nullable* o meno e *unique* se la colonna è univoca. Se non viene specificata l'annotazione, il nome della colonna nella tabella è il nome del campo.

In entrambe le entità sono generati automaticamente:

- Costruttore nullo (necessario per *Hibernate*).
- Costruttore con tutti gli argomenti.
- *Getter* e *Setter* per i campi privati degli attributi persistenti.
- Metodo *toString()*.

5.1.1 Metodi *equals()* e *hashCode()*

In *Author* la chiave primaria ha l'annotazione `@GeneratedValue(strategy = GenerationType.IDENTITY)`. Essa indica la strategia di generazione per i valori della chiave primaria nel database. Con *IDENTITY* l'assegnazione avviene tramite valori incrementati automaticamente. [17]

Se la chiave primaria è generata automaticamente, essa viene generata da *Hibernate* e *JPA* quando l'entità viene resa persistente nel database. Ciò significa che non è un campo su cui è possibile fare affidamento per calcolare l'*hashCode*. Se l'oggetto viene aggiunto in un *Set* e poi viene salvato nel database, l'oggetto nel *Set* e quello del database risultano diversi perché l'*id* è cambiato (è stato generato automaticamente un nuovo valore durante il salvataggio). Questa situazione può presentarsi con una relazione molti a molti.

Per implementare *hashCode()* ed *equals()*, la documentazione di *Hibernate* raccomanda di non includere campi mutabili nell'*hashCode()*, poiché ciò richiederebbe il *rehashing* di qualsiasi raccolta contenente l'entità ogni volta che il campo viene mutato.

Dato che l'identificatore viene generato solo quando l'istanza dell'entità viene resa persistente, è consigliato non aggiungerla in una raccolta fino al suo salvataggio nel database. In alternativa, nell'*hashcode* è opportuno includere qualsiasi campo immutabile e non generato. La documentazione suggerisce di identificare una *chiave naturale* per ogni entità, ovvero una combinazione di campi che identifichino univocamente un'istanza dell'entità, dal punto di vista del modello dati del programma. [18]

L'entità *Book* non ha un identificativo generato, quindi, l'*hashcode* calcolato prima o dopo il salvataggio nel database è lo stesso. Il metodo

hashCode() è strutturato sulla chiave naturale ISBN dell'entità. Perciò, due libri risultano uguali se e solo se hanno lo stesso ISBN.

```

1  @Override
2      public int hashCode() {
3          return Objects.hash(isbn);
4      }
5
6  @Override
7      public boolean equals(Object obj) {
8          if (this == obj)
9              return true;
10         if (obj == null)
11             return false;
12         if (getClass() != obj.getClass())
13             return false;
14         Book other = (Book) obj;
15         return Objects.equals(isbn, other.isbn);
16     }

```

L'entità *Author* ha la chiave generata automaticamente da *Hibernate*, dunque, è necessario identificare una chiave naturale. Nel mondo reale si identificano gli autori tramite il loro nome e cognome. Perciò, risultano uguali se hanno lo stesso nome e cognome (si suppone che non ci siano omonimi). Si struttura il metodo *hashCode()* sulla combinazione dei campi *firstName* e *lastName*.

```

@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Author other = (Author) obj;
    return Objects.equals(firstName, other.firstName) && Objects.equals(
        lastName, other.lastName);
}

```

5.1.2 Relazione molti a molti tra Book e Author

Nel capitolo precedente è stata descritta la relazione molti a molti tra le entità *Book* ed *Author* tramite l'uso di una terza tabella contenente le chiavi primarie delle due entità che unisce.

Tale tabella, in *Spring boot*, è chiamata *Tabella di Join* e la sua costruzione avviene tramite l'annotazione `@JoinTable`, in cui ne viene definito il nome della tabella. [19]

Ogni libro ha una lista di autori e ogni autore ha una lista di libri scritti. Dunque, in *Book* si include una collezione di *authors* ed in *Author* una collezione di *books*. La relazione molti a molti viene configurata dal lato proprietario ed è possibile scegliere qualsiasi lato come proprietario. In questo caso, è stato scelto *Book* :

```

@ManyToMany(
    fetch = FetchType.LAZY,
3     cascade = { CascadeType.MERGE, CascadeType.PERSIST })
)
@JoinTable(
    name = "author_book",
    joinColumns = @JoinColumn(name = "book_isbn"),
8     inverseJoinColumns = @JoinColumn(name = "author_id"))
)
@JsonManagedReference
private Set<Author> authors = new HashSet<>();

```

È stato definito l'attributo *cascade* per definire come le operazioni su un'entità sono propagate alle entità collegate. Nello specifico:

- il tipo *MERGE* propaga l'operazione di aggiornamento da un'entità padre a un'entità figlio.
- il tipo *PERSIST* propaga l'operazione di persistenza da un'entità padre a un'entità figlio.

Nella configurazione della tabella, sono stati forniti il nome e le chiavi esterne attraverso l'annotazione `@JoinColumn`. L' attributo *joinColumn* si collega al lato proprietario della relazione e l' *inverseJoinColumn* all'altro lato.

Sul lato target, nella classe *Author*, è necessario solo fornire il nome del campo che mappa la relazione:

```
@ManyToMany(
```

```

        fetch = FetchType.LAZY,
        mappedBy = "authors"
4      )
@JsonBackReference
private Set<Book> books = new HashSet<>();

```

Nelle due classi è stato definito l'attributo *FetchType.LAZY*. Esso specifica il tipo di caricamento e può essere *Eager* oppure *Lazy*. Con *Eager* l'inizializzazione dei dati avviene immediatamente, con *Lazy* l'inizializzazione avviene quando si utilizza la relazione. Per non avere problemi prestazionali con una relazione molti a molti, è consigliato l'uso del *FetchType.LAZY*.

Quando viene mappata un'associazione molti a molti, si dovrebbe utilizzare *java.util.Set*. Se l'associazione viene mappata con *java.util.List*, *Hibernate* non gestisce efficientemente la rimozione delle entità. Inoltre, non si dovrebbe utilizzare *CascadeType.REMOVE* (opzione utilizzata per eliminare un'entità insieme a tutte le sue entità figlie) e, dunque, *CascadeType.ALL* nella mappatura, perché *JPA* potrebbe rimuovere tutte le entità associate, anche nel caso in cui essa sia ancora collegata ad altre entità [20].

Per concludere la configurazione della relazione molti a molti, vengono implementati due metodi *helper*, nella classe *Book*:

```

public void addAuthors(Set<Author> authorsToAdd) {
    for (Author author : authorsToAdd) {
        this.authors.add(author);
4      author.getBooks().add(this);
    }
}

public void removeAuthors(Set<Author> authorsToRemove) {
9      Iterator<Author> iterator = authors.iterator();
      while (iterator.hasNext()) {
          Author author = iterator.next();
          if (authorsToRemove.contains(author)) {
              iterator.remove();
14            author.getBooks().remove(this);
          }
      }
}

```

Il primo metodo è utilizzato per aggiungere un autore alla lista degli scrittori di un libro, esso facilita l'associazione tra le due entità durante

il processo di inserimento utilizzando un oggetto *BookDto*, spiegato successivamente. Nello specifico, se ad un libro viene aggiunto un autore, rispettivamente, all'autore stesso viene aggiunto il libro in questione. Il secondo metodo, viene utilizzato per eliminare le entità associate alla relazione molti a molti in modo da evitare l'uso di *CascadeType.REMOVE*. Infatti, se viene rimosso un autore dalla lista degli scrittori del libro, anche il libro stesso viene rimosso dalla lista dei libri di quell'autore. Con l'uso dei due metodi si garantisce la coerenza tra le due collezioni delle due entità.

5.1.3 Implementazione classi DTO (*Data Transfert Object*)

Le componenti *DTO* (*Data Transfer Objects*) permettono di encapsulare i dati per il trasferimento. In breve, servono per rendere pubbliche le risorse, entità, che vengono rappresentate nel livello di dominio. L'uso delle classi *DTO* può comportare i seguenti vantaggi [21]:

- è possibile nascondere delle informazioni sovrabbondanti (ad esempio la trama del libro). Non tutte le informazioni del database sono necessarie.
- è possibile modificare i nomi dei campi delle entità, la struttura e il numero dei campi restituiti.
- si ha una maggiore sicurezza. Con l'uso delle classi *DTO* si evita di dare un'indiretta informazione alla struttura delle relative tabelle sulle quali vengono salvate le informazioni delle entità.

Le classi *DTO* sono implementate nel package *com.library.webapp.dtos*.

La classe *BookDto* è utilizzata per ottenere informazioni di uno o più libri. Ha una lista di *AuthorSlimDto*.

```
public class BookDto {
    private String isbn;
3   private String title;
    private String genre;
    private String plot;
    private LocalDate publicationDate;
    private int numberPages;
8   private Set<AuthorSlimDto> authors;
```

La classe *AuthorDto* è utilizzata per ottenere le informazioni degli autori. Ha una lista di *BookSlimDto*.

```
1 public class AuthorDto {
2     private Long id;
3     private String firstName;
4     private String lastName;
5     private Set<BookSlimDto> books;
```

La classe *BookSlimDto* è di supporto ad *AuthorDto*. Contiene solo due informazioni, ISBN e titolo, cioè quelle necessarie per quando si vogliono ottenere le informazioni degli autori.

```
1 public class BookSlimDto {
2     private String isbn;
3     private String title;
```

La classe *AuthorSlimDto* è di supporto a *BookDto*. Contiene id, nome e cognome.

```
1 public class AuthorSlimDto {
2     private Long id;
3     private String firstName;
4     private String lastName;
```

Per tutte le classi DTO, sono generati automaticamente:

- Costruttore nullo (necessario per *Hibernate*).
- Costruttore con tutti gli argomenti.
- *Getter* e *Setter* per i campi privati degli attributi persistenti.
- Metodi *hashCode()* e *equals()* in coerenza con le classi entità.

Create le classi DTO, è necessario il tool *MapStruct*, per eseguire la mappatura, ovvero il riversamento automatico dei dati dalla classe entità alla classe DTO. Per utilizzare il tool, è necessario aggiungere la dipendenza corrispondente a *MapStruct* nel file *pom.xml*.

Nel package *com.library.webapp.dto.mapper* è implementata l'interfaccia *MapStructMapper*, in cui vengono dichiarati tutti i metodi necessari per la mappatura. È necessaria l'annotazione *@Mapper*.

```
1 @Mapper
public interface MapStructMapper {
    BookSlimDto bookToBookSlimDto(Book book);
```

```

    Book bookDtoToBook(BookDto bookDto);
    BookDto bookToBookDto(Book book);
6   AuthorDto authorToAuthorDto(Author author);
    AuthorSlimDto authorToAuthorSlimDto(Author author);
    Author authorSlimDtoToAuthor(AuthorSlimDto authorPostDto);
}

```

Nello stesso package, viene creata la corrispondente classe di implementazione *MapStructMapperImpl*.

```

1 @Component
public class MapStructMapperImpl implements MapStructMapper{

```

La classe è annotata con *@Component*, che consente a *Spring* di rilevare automaticamente i *bean* personalizzati.

5.2 LIVELLO DI PERSISTENZA

Spring Data è un framework che semplifica e riduce notevolmente la quantità di codice necessaria per l'accesso dei dati archiviati in un database.

Per utilizzare le funzionalità *Spring Data*, è necessario implementare un'interfaccia *DAO* (*Data Access Object*), essa definisce le operazioni che possono essere eseguite nel database per una specifica tabella. L'interfaccia deve essere annotata con *@Repository* e deve estendere l'interfaccia specifica del repository JPA, *JpaRepository*, inserendo nei due parametri dei generici rispettivamente la classe di entity di riferimento (ad esempio *Author* o *Book*) ed il tipo della chiave primaria. Estendendo l'interfaccia *JpaRepository*, la classe DAO eredita dei metodi base per le operazioni CRUD ed è possibile anche definirne di nuovi tramite l'utilizzo delle *query*. [22]

Nell'applicazione sviluppata, all'interno del package *com.library.webapp.repository*, si definiscono due interfacce DAO.

L'interfaccia *BookRepository* definisce le operazioni che possono essere eseguite sull'entità *Book*.

```

@Repository
@Transactional(readOnly = true)
3 public interface BookRepository extends JpaRepository<Book, String> {
    @Query(value = "SELECT * FROM book WHERE genre = ? ", nativeQuery = true)

```

```

List<Book> findByGenreLike(String genre);
@Query(value = "SELECT * FROM book b WHERE b.publication_date > :date
    ORDER BY PUBLICATION_DATE", nativeQuery = true)
List<Book> findByAfterPublicationDate(LocalDate date);
8 @Query(value = "SELECT * FROM book b WHERE TITLE LIKE CONCAT('%', :title,
    '%') ORDER BY PUBLICATION_DATE", nativeQuery = true)
List<Book> findByTitleContains(String title);
}

```

L’interfaccia *AuthorRepository* definisce le operazioni che possono essere eseguite sull’entità *Author*.

```

@Repository
@Transactional(readOnly = true)
public interface AuthorRepository extends JpaRepository<Author, Long> {
    @Query(value = "SELECT * FROM author a WHERE (first_name = :firstName and
        last_name = :lastName)", nativeQuery = true )
5 Author findByNameLike(String firstName, String lastName);
    }
}

```

Le due interfacce, oltre alle operazioni ereditate da *JpaRepository*, definiscono ulteriori operazioni tramite delle query native SQL. L’utilizzo di `@Transactional(readOnly = true)` è buona pratica quando nelle interfacce DAO ci sono più operazioni di lettura che di scrittura [23].

5.2.1 Connessione al database MySQL

Nel capitolo della progettazione, è stato creata un’immagine Docker di un database MySQL. Per poter utilizzare le funzionalità *Spring Data*, è necessario configurare la connessione al database creato tramite il file *application.properties* o *application.yml* presente nella directory *src/main/resource* del progetto *Spring Boot*, che contiene le proprietà di configurazione di *Spring* o altre proprietà dell’applicazione sviluppata.

```

application.yml
1 server:
2   port: 5051
3 spring:
4   datasource:
5     driverClassName: com.mysql.cj.jdbc.Driver
6     url: jdbc:mysql://localhost:1590/LibraryDB
7     username: root
8     password: LibraryDB
9
10 jpa:
11   properties:
12     hibernate:
13       ddl-auto: update
14     show-sql: true
15     generate-ddl: true

```

Figura 14: File application.yml

Nel file di configurazione, sono state definite le seguenti proprietà:

- *port: 5051* indica la porta su cui viene avviata l'applicazione.
- l'applicazione si connette a *LibraryDB*, ovvero un database *MySQL* ospitato sulla macchina corrente (*localhost*) sulla porta *1590*, utilizzando le credenziali specificate.
- è stata impostata la proprietà per il processo di creazione delle tabelle di *Hibernate* in *update*. Essa comunica ad *Hibernate* di aggiornare lo schema del database se viene modificato. Altrimenti, se lo schema non è presente nel database, viene creato.

Dopo aver implementato il livello di persistenza, che permette di eseguire le operazioni sulle entità, ed aver configurato il database dell'applicazione, viene implementato il livello di servizio, che interagisce con quello di persistenza per eseguire le operazioni definite dalla *business logic*.

5.3 LIVELLO DI SERVIZIO

La *business logic* (*logica applicativa*) è il nucleo di un'applicazione. Si trova tra il livello di persistenza e quello web o consumer ed ha il compito di manipolare i dati. Nello strato di servizio è presente tutta la logica che contraddistingue l'applicazione.

All'interno del package *com.library.webapp.service*, è dichiarata l'interfaccia *LibraryService*, in cui vengono definite le operazioni possibili sulle entità.

```

public interface LibraryService {
    List<BookDto> selAllBooks();
3   BookDto selBookByIsbn(String isbn);
    boolean delBook(String isbn);
    Book insertNewBook(BookDto newBook);
    boolean saveBook(BookDto book);
    List<BookDto> findBooksByGenre(String genre);
8   List<BookDto> findBooksAfterPublicationDate(LocalDate date);
    List<BookDto> findBooksTitleContains(String title);
    List<AuthorDto> selAllAuthors();
    AuthorDto selAuthorById(Long id);
    boolean saveAuthor(AuthorSlimDto authorToSave);
13  AuthorDto findAuthorByNameAndSurname(String firstName, String lastName);
    Set<AuthorSlimDto> fillAuthorsSetByStringName(String authorNames);
    String fillStringNameBySet(Set<AuthorSlimDto> authors);
}

```

Nello stesso package, viene creata la corrispondente classe di implementazione, *LibraryServiceImpl*, nella quale viene fornita l'implementazione dei metodi dichiarati nell'interfaccia.

```

@Service
@Transactional
public class LibraryServiceImpl implements LibraryService {
4   @Autowired
    BookRepository bookRepository;
    @Autowired
    AuthorRepository authorRepository;
    @Autowired
9   private MapStructMapper mapstructMapper;
    @Override
    @Transactional(readOnly = true)
    public List<BookDto> selAllBooks() {
        List<Book> allBooks = bookRepository.findAll();
14   if(allBooks.isEmpty()) throw new NotFoundException("There is no book!");
        ;
        List<BookDto> booksDto = new ArrayList<BookDto>();
        for (Book book : allBooks) {
            booksDto.add(mapstructMapper.bookToBookDto(book));
        }
19   return booksDto;
    }
    @Override
    @Transactional(readOnly = true)
    public BookDto selBookByIsbn(String isbn) {
24   BookDto bookdto = mapstructMapper.bookToBookDto(bookRepository.findById

```

```

(isbn).orElse(null));
if(bookdto == null) {
    throw new NotFoundException("Book not present or wrong ISBN!");
}
return bookdto;
29 }
@Override
public boolean delBook(String isbn) {
    Book book = bookRepository.findById(isbn).orElse(null);
    if(book == null){
        throw new NotFoundException(String.format("It is not possible to
delete the book with ISBN %s because it is not present!", isbn));
    }
    Set<Author> authors = new HashSet<Author>();
    authors.addAll(book.getAuthors());
    book.removeAuthors(book.getAuthors());
39 bookRepository.deleteById(isbn);
Set<Author> authorsToDelete = new HashSet<Author>();
for (Author author : authors) {
    if(author.getBooks().isEmpty()) authorsToDelete.add(author);
}
44 authorRepository.deleteAll(authorsToDelete);
return true;
}
@Override
public Book insertNewBook(BookDto newBook) {
49 if( (bookRepository.findById(newBook.getIsbn())).orElse(null) != null)
{
    throw new DuplicateException(String.format("The book with ISBN %s is
already there!", newBook.getIsbn()));
}
Book book = mapstructMapper.bookDtoToBook(newBook);
associateWithExistingAuthor(book);
54 bookRepository.save(book);
return book;
}
@Override
public boolean saveBook(BookDto bookDto) {
59 Book bookFindById = bookRepository.findById(bookDto.getIsbn()).orElse(
null);
if(bookFindById == null){
    throw new NotFoundException(String.format("It is not possible to edit
the book with ISBN %s because it is not present!", bookDto.getIsbn()))
;
}
Book book = mapstructMapper.bookDtoToBook(bookDto);
64 associateWithExistingAuthor(book);
bookRepository.save(book);

```

```

        return true;
    }
    @Override
69   public boolean saveAuthor(AuthorSlimDto authorToSave) {
        if(authorRepository.findById(authorToSave.getId()).orElse(null) == null){
            throw new NotFoundException(String.format("It is not possible to
                modify the author with ID %s because it is not present!", authorToSave.
                getId()));
        }
        authorRepository.save(mapstructMapper.authorSlimDtoToAuthor(
            authorToSave));
        return true;
    }
    @Override
    @Transactional(readOnly = true)
    public AuthorDto findAuthorByNameAndSurname(String firstName, String
        lastName) {
    Author author = authorRepository.findByNameLike(firstName, lastName);
    if(author == null) throw new NotFoundException("There is no author with
        the specified name.");
    AuthorDto authorDto = mapstructMapper.authorToAuthorDto(author);
    return authorDto;
}
84 }
```

La classe è annotata con `@Service` per indicare che appartiene al livello di servizio. Si osservi che nei metodi implementati viene eseguita la mappatura tra *entity* e relativa classe DTO e viceversa. Essa deve avvenire nello strato di servizio.

I metodi della classe *LibraryServiceImpl* non riportati nella seguente tesi sono presenti nel codice sorgente.

Le operazioni sulle entità *Book* e *Author* definite sono:

- *selAllBooks()*: metodo per selezionare tutti i libri presenti nel catalogo.
- *selBookByIsbn()*: metodo per selezionare il libro in base al codice ISBN.
- *delBook()*: metodo per eliminare un libro nel catalogo in base al codice ISBN.
- *insertNewBook()*: metodo per aggiungere un libro nel catalogo.
- *saveBook()*: metodo per aggiornare le informazioni di un libro.

- *associateWithExistingAuthor()*: metodo di supporto per le operazioni di inserimento e aggiornamento. Se esiste un autore con lo stesso nome dichiarato nella lista degli scrittori del libro, che inseriamo o aggiorniamo, allora il metodo associa lo scrittore della lista con l'autore già presente nel database. Se non è presente un autore con lo stesso nome, allora viene creato.
- *findBooksByGenre()*: metodo per filtrare i libri del catalogo in base al genere.
- *findBooksAfterPublicationDate()*: metodo per filtrare i libri del catalogo in base alla data di pubblicazione.
- *findBooksTitleContains()*: metodo per ricerca in base al titolo.
- *selAllAuthors()*: metodo per selezionare tutti gli autori presenti nel catalogo.
- *selAuthorById()*: metodo per selezionare un autore in base al codice ID.
- *saveAuthor()*: metodo per aggiornare un autore.
- *findAuthorByNameAndSurname()*: metodo per la ricerca di un autore per nome e cognome.
- *fillAuthorsSetByStringName()*: metodo di supporto per l'inserimento e l'aggiornamento di un libro. Utilizzato nel controller *BookWebController* durante la fase salvataggio per convertire una stringa, contenente i nomi degli autori, dal formato "*Nome₁ Cognome₁, Nome₂ Cognome₂, ... Nome_N Cognome_N*" in un Set di autori.
- *fillStringNameBySet()*: metodo di supporto per l'aggiornamento di un libro. Utilizzato nel controller *BookWebController* per convertire un Set di autori del libro in una stringa, contenente i nomi degli autori, nel formato "*Nome₁ Cognome₁, Nome₂ Cognome₂, ... Nome_N Cognome_N*" per permettere la loro modifica.

5.3.1 Transazioni in Spring Boot

"La transazione è un'interfaccia disponibile nel pacchetto *org.hibernate* che è associata alla sessione. Possiamo descrivere la transazione con le proprietà ACID:

- Atomicità: tutto il successo o nessuno.
- Coerenza: i vincoli del database non devono essere violati.
- Isolamento: una transazione non dovrebbe effettuarne un'altra.
- Durata: dovrebbe essere nel database dopo il commit. " [24]

Per utilizzare le transazioni si fa uso dell'annotazione `@Transactional`, che viene utilizzata nel livello di servizio, e dell'annotazione `@EnableTransactionManagement`, utilizzata nella classe principale contenente il metodo `main`.

Esempio del comportamento senza @Transactional

A tale scopo, si inserisce un'eccezione non controllata nel metodo `insertNewBook()` in `LibraryServiceImpl`:

```

1  @Override
  public Book insertNewBook(BookDto newBook) {
    if( (bookRepository.findById(newBook.getIsbn())).orElse(null) != null)
    {
      throw new DuplicateException(String.format("The book with ISBN %s is
already there!", newBook.getIsbn()));
6
    }
    Book book = mapstructMapper.bookDtoToBook(newBook);
    associateWithExistingAuthor(book);
    // Test per le transazioni:
    bookRepository.save(book);
11   int n = 10/0;
    return book;
  }
}

```

Se si prova ad inserire un libro, ad esempio con titolo *Senza Transazioni*, si ha l'eccezione *ArithmetricException* e nel database si ottiene:

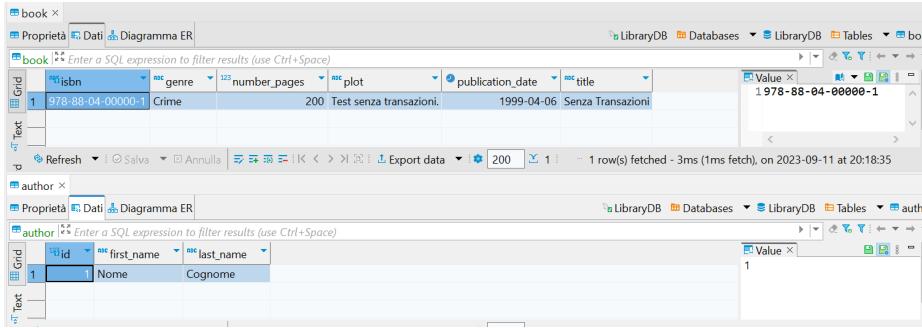


Figura 15: Esempio senza @Transactional

Non si vuole che ciò accada. Se c'è un'eccezione, i dati non dovrebbero persistere, dunque, si inseriscono le transazioni per evitare questo comportamento.

Esempio del comportamento con @Transactional

Si inserisce `@Transactional` a livello di classe di servizio. L'annotazione ha di default l'opzione `rollback` attivata, quindi, se si verifica un'eccezione durante un'operazione, lo stato del database viene ripristinato a quello precedente l'operazione. Per i metodi che recuperano informazioni dal catalogo, senza modificare i dati, si utilizza l'annotazione `@Transactional(readOnly = true)`.

Nella Figura 16, si può osservare che anche se durante l'inserimento, ad esempio del libro con titolo *Con Transazioni*, si verifica l'eccezione `ArithmetcException`, il database non viene modificato.

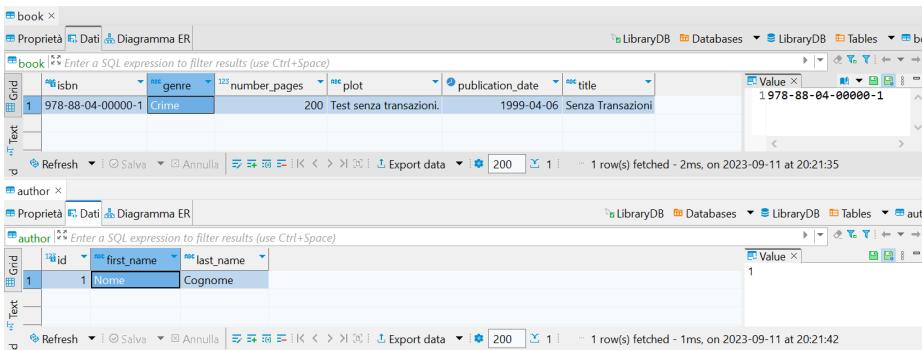


Figura 16: Esempio con @Transactional

5.3.2 Eccezioni personalizzate: *NotFoundException* e *DuplicateException*

Nella classe *LibraryServiceImpl* viene fatto uso di due eccezioni personalizzate, *NotFoundException* e *DuplicateException*.

L'eccezione *NotFoundException* viene sollevata quando non è presente un'entità, od un insieme di entità, nel catalogo. L'eccezione *DuplicateException* viene sollevata quando si cerca di inserire un'entità già presente nel catalogo.

Viene, inoltre, implementato l'oggetto personalizzato *ErrorResponse*, composto da un codice ed un messaggio di errore. Una sua istanza viene restituita dai *RestController* nel caso in cui si presenti un errore.

L'implementazione delle eccezioni e di *ErrorResponse* è presente nel package *com.library.webapp.exceptions* del codice sorgente.

5.4 LIVELLO WEB O CONSUMER

All'inizio del capitolo, è stato descritto il controller come componente che si occupa di interpretare gli input dell'utente e di fornire la risposta appropriata.

Nell'applicazione web sviluppata si ha l'utilizzo sia dell'annotazione *@RestController*, per fornire servizi di *API RESTful* e consentire ad altre applicazioni o servizi di accedere ai dati in formato *JSON* ma non di modificare il catalogo, e sia l'utilizzo dell'annotazione *@Controller* per gestire pagine web tradizionali. Le componenti *@RestController* possono solo accedere ai dati, ma non possono modificarli, la modifica è permessa tramite pagine web dedicate.

5.4.1 Componente *@RestController*

@RestController è un'annotazione che combina *@Controller* e *@ResponseBody*, annotazione che indica al controller che l'oggetto restituito viene serializzato in *JSON* e passato nuovamente all'oggetto *HttpServletResponse*.

All'interno del package *com.library.webapp.restController* sono implementate le due classi controller *BookRestController* e *AuthorRestController*. Entrambe le classi, hanno le annotazioni a livello di classe *@RestController* e *@RequestMapping*, quest'ultima è utilizzata per mappare una richiesta HTTP. I metodi delle classi, come accennato in precedenza, recuperano informazioni dal database, infatti, sono annotati con *@GetMapping*, è una

particolare annotazione di `@RequestMapping` che permette di mappare le richieste HTTP `GET`. Con `@PathVariable` è indicata una parte variabile del path, che il controller riceve come parametro.

La classe `BookRestController` è la classe controller REST che gestisce le richieste relative all'entità `Book`.

```

1 @RequestMapping(value = "/api/book")
2 @RestController
3 public class BookRestController {
4     @Autowired
5     private LibraryService service;
6     @GetMapping
7     public List<BookDto> listAllBook(){
8         List<BookDto> allBook = service.selAllBooks();
9         return allBook;
10    }
11    @GetMapping(value = "{isbn}")
12    public BookDto getByIsbn(@PathVariable("isbn") String isbn){
13        BookDto book = service.selBookByIsbn(isbn);
14        return book;
15    }
16    @GetMapping(value = "/genre/{genre}")
17    public List<BookDto> getByGenre(@PathVariable("genre") String genre){
18        List<BookDto> books = service.findBooksByGenre(genre);
19        return books;
20    }
21    @GetMapping(value = "/title/{title}")
22    public List<BookDto> getByTitleContains(@PathVariable("title") String
23        title){
24        List<BookDto> books = service.findBooksTitleContains(title);
25        return books;
26    }
27    @GetMapping(value = "/afterdate/{date}")
28    public List<BookDto> getAfterPublicationDate(@PathVariable("date")
29        @DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate date){
30        List<BookDto> books = service.findBooksAfterPublicationDate(date);
31        return books;
32    }
33 }
```

Dunque, quando un utente esegue la richiesta sull'URL

- `/api/book`, ottiene la lista di tutti i libri in formato `JSON`, se presenti.
- `/api/book/` e il valore di un codice ISBN, ottiene le informazioni riguardanti il libro con il codice ISBN specificato, in formato `JSON`,

se presente.

- */api/book/genre/* e il valore di un genere letterario, ottiene la lista dei libri che appartengono a quel genere, in formato JSON, se presenti.
- */api/book/title/* e una stringa, ottiene la lista dei libri che contengono la stringa specificata nel titolo, in formato JSON, se presenti.
- */api/book/afterdate/* e il valore di una data, ottiene la lista dei libri che sono stati pubblicati dopo la data specificata, in formato JSON, se presenti.

AuthorRestController è la classe controller REST che gestisce le richieste relative all'entità *Author*. Quando un utente esegue la richiesta sull'URL:

- */api/author* ottiene la lista di tutti gli autori in formato JSON, se presenti.
- */api/author/* e il valore di un codice ID, ottiene le informazioni riguardanti un autore con il codice ID specificato, in formato JSON, se presente.
- */api/author/name/* e il valore del nome e cognome, separati da /, ottiene le informazioni dell'autore con nome e cognome specificato in formato JSON, se presente.

Nel seguente codice, è presente il metodo che permette la ricerca di un autore per nome e cognome. Gli altri due metodi sono visibili nel codice sorgente perché molto simili a quelli implementati in *BookRestController*.

```

@RequestMapping("/api/author")
@RestController
public class AuthorRestController {
    @Autowired
    private LibraryService service;
    @GetMapping(value = "/name/{firstName}/{lastName}")
    public AuthorDto listAuthorByNameAndSurname(@PathVariable("firstName")
        String firstName, @PathVariable("lastName") String lastName){
        AuthorDto author = service.findAuthorByNameAndSurname(firstName,
        lastName);
    }
}

```

Si osservi che i metodi di entrambi le classi hanno meno logica possibile, infatti, il controller dovrebbe solo richiamare lo strato di servizio, il quale contiene la logica dell'applicazione.

Gestione delle eccezioni

Nel livello di servizio, sono state implementate due eccezioni (*NotFoundException*, *DuplicateException*) ed un oggetto *ErrorResponse* da restituire come istanza in caso di eccezioni nelle classi annotate con *@RestController*.

Si implementa la classe *RestControllerExceptionHandler* annotata con *@RestControllerAdvice("com.library.webapp.restController")* per indicare che la classe gestisce le eccezioni per i controller presenti nel package *com.library.webapp.restController*.

```
1 @RestControllerAdvice("com.library.webapp.restController")
2 public class RestControllerExceptionHandler{
3     @ExceptionHandler(NotFoundException.class)
4     public ResponseEntity<ErrorResponse> handleNotFoundException(
5         NotFoundException ex) {
6             ErrorResponse error = new ErrorResponse();
7             error.setCode(HttpStatus.NOT_FOUND.value());
8             error.setMessage(ex.getMessage());
9             return new ResponseEntity<ErrorResponse>(error, HttpStatus.NOT_FOUND);
10        }
11    }
```

Se si verifica un'eccezione *NotFoundException*, viene restituito un'istanza di *ErrorResponse* con codice *404* e messaggio specificato nel servizio, in base al metodo che ha causato l'eccezione. Di seguito, è presente un esempio.

```
1 {
2     "code":404,
3     "message":"There is no author with the specified name."
4 }
```

Nella classe non viene gestita l'eccezione *DuplicateException*, perché tramite i due controller non è possibile l'inserimento.

5.4.2 Componente @Controller

Le principali responsabilità del controller web nella tipica architettura *Spring Model View Controller* sono:

- intercettare le richieste in arrivo.
- convertire il payload della richiesta nella struttura interna dei dati.
- invio dei dati a Model per ulteriori elaborazioni.
- ottenere i dati elaborati dal modello e far avanzare tali dati alla vista per il rendering.

Le applicazioni *Model View Controller* non sono orientate ai servizi; pertanto, esiste un *View Resolver* che esegue il rendering delle visualizzazioni finali in base ai dati ricevuti da un Controller. [25]

All'interno del package *com.library.webapp.controller* sono implementate le classi controller web *HomeWebController*, *BookWebController* e *AuthorWebController*. In tutte le classi web controller, a livello di classe, sono presenti le annotazioni *@Controller* e *@RequestMapping*.

La classe *HomeWebController* gestisce la chiamata alla *home page* dell'applicazione.

```

1 @Controller
  @RequestMapping("/")
  public class HomeWebController {
    @GetMapping
    public String homepage() {
6      return "index";
    }
}

```

La classe ha un unico metodo che risponde all'URL / e restituisce una stringa, che rappresenta il nome di una vista.

La classe *BookWebController* è la classe controller che gestisce le operazioni relative all'entità *Book*.

```

2 @Controller
  @RequestMapping(value = "/library")
  public class BookWebController {
    @Autowired
    LibraryService service;

```

```

    @GetMapping
7   public String listAllBooks(Model model){
        List<BookDto> allBooks = service.selAllBooks();
        model.addAttribute("books", allBooks);
        return "books";
    }
12  @GetMapping(value = "/{isbn}")
    public String getBookByIsbn(@PathVariable String isbn, Model model){
        BookDto book = service.selBookByIsbn(isbn);
        model.addAttribute("book", book);
        return "detBook";
    }
17  @GetMapping(value = "/delete/{isbn}")
    public String deleteBook(@PathVariable String isbn, Model model) {
        service.delBook(isbn);
        return "redirect:/library";
    }
22  @GetMapping("/insert")
    public String newBook(Model model) {
        BookDto bookDto = new BookDto();
        model.addAttribute("message", "New Book");
    }
27  String authorNames = "";
    model.addAttribute("authorNames", authorNames);
    model.addAttribute("bookDto", bookDto);
    return "editBook";
}
32  @PostMapping("/insert/saveNewBook")
    public String saveNewBook(@ModelAttribute BookDto book, @ModelAttribute("authorNames") String authorNames) {
        Set<AuthorSlimDto> authors = service.fillAuthorsSetByStringName(
            authorNames);
        book.setAuthors(authors);
        service.insertNewBook(book);
    }
37  return "redirect:/library";
}
@GetMapping("/update/{isbn}")
public String updateBook(Model model, @PathVariable String isbn) {
    BookDto bookDto = service.selBookByIsbn(isbn);
    String authorNames = service.fillStringNameBySet(bookDto.getAuthors());
    model.addAttribute("authorNames", authorNames);
    model.addAttribute("bookDto", bookDto);
    model.addAttribute("message", "Edit Book");
    return "editBook";
}
47  }
@PostMapping("/update/saveBook")
public String saveBook(@ModelAttribute BookDto book, @ModelAttribute("authorNames") String authorNames) {
    Set<AuthorSlimDto> authors = service.fillAuthorsSetByStringName(

```

```

authorNames);
book.setAuthors(authors);
52    service.saveBook(book);
      return "redirect:/library";
}
}

```

Oltre all'annotazione `@GetMapping`, è presente anche l'annotazione `@PostMapping`, che permette di mappare le richieste HTTP POST.

Dunque, quando un utente esegue la richiesta sull'URL:

- `/library`, il controller recupera la lista dei libri presenti nel catalogo e la invia alla vista `books`.
- `/library/` e il valore di un codice ISBN, il metodo recupera i dettagli del libro, se presente, e li invia alla vista `dettBook`.
- `/library/delete/` e il valore di un codice ISBN, il metodo richiama il servizio per eliminare il libro con il codice ISBN specificato. Dopodiché, l'utente viene reindirizzato alla vista `books`.
- `/library/insert`, l'utente viene reindirizzato alla vista `editBook`, pagina di inserimento di un libro. Dopo aver inserito i dati necessari per l'inserimento in un `form` ed aver cliccato sul bottone `submit`, il controller, con il metodo `saveNewBook` che ottiene i dati del libro inseriti tramite l'annotazione `@ModelAttribute`, salverà il libro nel catalogo. Se il libro è già presente nel catalogo, si ha un errore.
- `/library/update/` e il valore di un codice ISBN, l'utente viene reindirizzato alla vista `editBook`, pagina di modifica di un libro. Dopo aver effettuato le modifiche del libro ed aver cliccato sul bottone `submit`, il controller con il metodo `saveBook` che ottiene i dati tramite l'annotazione `@ModelAttribute`, salverà le modifiche del libro nel catalogo.
- `/library/genre/` e il valore di un genere letterario, il controller recupera la lista dei libri del genere specificato, se presenti, e la invia a `books`. Il metodo è visibile nel codice sorgente perché molto simile a `listAllBooks`.
- `/library/afterDate/` e il valore di una data, il controller recupera la lista dei libri pubblicati dopo la data specificata, se presenti, e la

invia a *books*. Il metodo è visibile nel codice sorgente perché molto simile a *listAllBooks*.

- */library/title/* e una stringa, il controller recupera la lista dei libri che contengono la stringa specificata nel titolo, se presenti, e la invia a *books*. Viene impostata la variabile *AllBooks = true*, ciò permette di visualizzare un bottone che può reindirizzare l'utente alla lista dei libri. Per poter effettuare la richiesta, vengono implementati due metodi, un metodo annotato con *@GetMapping* e l'altro con *@PostMapping* (per recuperare il valore della stringa per la ricerca). I due metodi sono visibili nel codice sorgente perché molto simili a quelli mostrati per la ricerca di un autore in base al nome e cognome della classe *AuthorWebController*.

Si può notare che, sia per l'inserimento sia per la modifica di un libro viene usata la stessa vista, la quale cambierà titolo in base al parametro passato *message*.

L'annotazione *@RequestParam("")* viene utilizzata per richiedere dei dati inseriti in un modulo e associarli al parametro specificato all'interno delle parentesi.

AuthorWebController è la classe controller che gestisce le richieste relative all'entità *Author*. Quando un utente esegue la richiesta sull'URL:

- */library/author*, il controller ottiene la lista di tutti gli autori presenti nel catalogo e la invia alla vista *authors*.
- */library/author/update/* e il valore di un codice ID, l'utente viene reindirizzato alla vista *editAuthor*, pagina di modifica di un autore. Dopo aver effettuato le modifiche e aver cliccato sul bottone *submit*, il controller con il metodo *saveAuthor*, che ottiene i dati tramite l'annotazione *@ModelAttribute*, salverà le modifiche dell'autore.
- */library/author/name/* e il valore del nome e cognome, il controller recupera l'autore, se presente, e lo invia alla vista *authors*. Viene impostata la variabile *AllAuthors = true*, ciò permette di visualizzare un bottone che può reindirizzare l'utente alla lista degli autori.

Nel seguente codice, sono presenti i metodi che permettono la modifica e la ricerca di un autore per nome e cognome. Il metodo per ottenere la lista di tutti gli autori è visibile nel codice sorgente perché molto simile a quello implementato in *BookWebController*.

```

@Controller
@RequestMapping(value = "/library/author")
public class AuthorWebController {
    @Autowired
    LibraryService service;
    @GetMapping("/update/{id}")
    public String updateAuthor(Model model, @PathVariable Long id){
        AuthorDto authorDto = service.selAuthorById(id);
        AuthorSlimDto author = new AuthorSlimDto(id, authorDto.getFirstName(),
        authorDto.getLastName());
        model.addAttribute("author", author);
        return "editAuthor";
    }
    @PostMapping("/update/saveAuthor")
    public String saveAuthor(@ModelAttribute AuthorSlimDto author) {
        service.saveAuthor(author);
        return "redirect:/library/author";
    }
    @GetMapping(value = "/name/{firstName}/{lastName}")
    public String findByName(Model model, @PathVariable("firstName") String
        firstName, @PathVariable String lastName) {
        AuthorDto author = service.findAuthorByNameAndSurname(firstName,
        lastName);
        model.addAttribute("authors", author);
        model.addAttribute("AllAuthors", true);
        return "authors";
    }
    @PostMapping(value="/name")
    public String findByNamePost(@RequestParam("firstNameFound") String first
        , @RequestParam("lastNameFound") String last) {
        return "redirect:/library/author/name/" + first + "/" + last;
    }
}

```

Gestione delle eccezioni

Dobbiamo gestire l'eccezioni che si possono verificare nei controller web. Si implementa la classe *ControllerExceptionHandler* annotata con `@ControllerAdvice("com.library.webapp.controller")` per indicare che la classe gestisce le eccezioni per i controller presenti nel package `com.library.webapp.controller`.

```

1 @ControllerAdvice("com.library.webapp.controller")
public class ControllerExceptionHandler {
    @ExceptionHandler(NotFoundException.class)
    public ModelAndView handleNotFoundException(NotFoundException ex) {

```

```

    ModelAndView modelAndView = new ModelAndView();
6    modelAndView.setViewName("errorPage");
    modelAndView.addObject("errorMessage", ex.getMessage());
    return modelAndView;
}
@ExceptionHandler(DuplicateException.class)
11   public ModelAndView handleDuplicateException(DuplicateException ex) {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("errorPage");
    modelAndView.addObject("errorMessage", ex.getMessage());
    return modelAndView;
16  }
}

```

Se si verifica un'eccezione, l'utente viene reindirizzato ad una pagina di errore *errorPage*, la quale verrà popolata con un messaggio di errore in base al metodo che ha causato l'eccezione. In Figura 17 è presente un esempio.

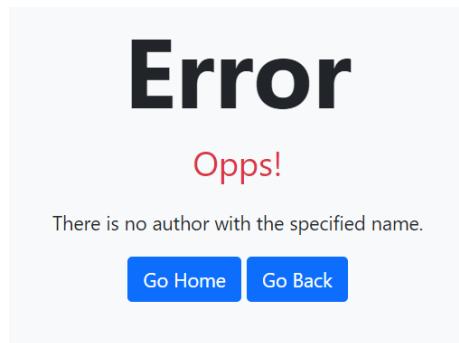


Figura 17: Esempio di gestione delle eccezioni nelle classi Controller

5.4.3 Viste (View)

Questa sezione descrive le viste restituite dai web controller. Esse vengono ricercate automaticamente da *Spring Boot* nella directory *src/main/resource/templates*.

Per lo sviluppo delle viste sono stati usati come base due template di *bootstrap*, ovvero librerie per la creazione di siti e applicazioni web, riadattati alle esigenze. I template usati sono *Shop Item* e la sezione *contact* di *Modern Business*, per la creazione dei form di inserimento e modifica, di *startbootstrap.com*.

I linguaggi utilizzati per la creazione delle viste sono:

- *HTML (HyperText Markup Language)* è un linguaggio di marcatura ipertestuale. Permette di indicare, tramite i *tag* (etichette), come disporre gli elementi all'interno di una pagina [26].
- *CSS (Cascading Style Sheets o foglio di stile)* permette di specificare delle regole che consentono di definire lo stile che devono assumere gli elementi sulla pagina.
- *Thymeleaf* è un moderno motore di template Java lato server per ambienti web. Con i moduli per *Spring Framework*, *Thymeleaf* è ideale per lo sviluppo web JVM HTML5 moderno. [27]

La vista *index.html* raffigura la *home page* dell'applicazione web. Nella pagina è presente uno sfondo, contenuto all'interno della directory *src/main/resource/static*, un titolo *Welcome to Library!* ed i bottoni per accedere alla lista dei libri, alla lista degli autori e per effettuare il login.

```

1 <body id="bodyhome" background="sfondo.jpg">
2     <!-- Navigation -->
3     <nav id="navBar" class="navbar-expand-lg navbar-light">
4         <div id="loginContainer">
5             <a id="buttonLogin" sec:authorize="!isAuthenticated()"
6                 th:href="@{/login}" class="btn btn-primary
7                 btn-mg">Login</a>
8             <form sec:authorize="isAuthenticated()" th:action="@{/logout}"
9                 method="post">
10                <button id="buttonLogout" type="submit" class="btn
11                  btn-primary btn-mg">Logout</button>
12            </form>
13        </div>
14    </nav>
15    <div class="content">
16        <center>
17            <h1 id="indexTitle">Welcome to Library!</h1>
18            <a id="buttonBooks" class="btn btn-primary btn-lg"
19                 th:href="@{/library}">List of Books</a>
20            <a id="buttonAuthors" class="btn btn-primary btn-lg"
21                 th:href="@{/library/author}">List of Authors</a>
22        </center>
23    </div>
24 </body>
```

Il tag *sec:authorize*, come accennato nel capitolo di analisi, viene utilizzato per nascondere determinati button a seconda che l'utente sia autenticato o meno. Questa autenticazione è basata su un processo di *login*, approfondito nei capitoli successivi.

Le viste *books.html* e *authors.html* sono utilizzate per visualizzare la lista dei libri e degli autori. In particolare, la vista *books.html* è utilizzata anche per il filtraggio dei libri in base alla ricerca per genere, data di pubblicazione e titolo, la vista *authors.html* è utilizzata anche per la ricerca per nome e cognome. Entrambe hanno una *Navigation Bar*, un *footer* ed una tabella, con *id* rispettivamente *booksTable* e *authorsTable*.

Nello script, è presente la tabella *booksTable*, quella degli autori è molto simile, differisce per il numero di colonne e per i tasti presenti nella tabella. In essa, infatti, non essendo permessa la cancellazione o l'inserimento di un autore, è presente solo il tasto per la modifica.

```

1      <table id="booksTable" class="table table-bordered">
2          <thead>
3              <tr>
4                  <th scope="col">Isbn</th>
5                  <th scope="col">Title</th>
6                  <th scope="col">Genre</th>
7                  <th scope="col">Authors</th>
8                  <th scope="col" >
9                      <a sec:authorize="isAuthenticated()" 
10                         th:href="@{/library/insert}">
11                          <button class="btn btn-outline-dark
12                             flex-shrink-0" type="button">New
13                             Book</button>
14                     </a>
15                 </th>
16             </tr>
17         </thead>
18         <tbody>
19             <!-- Sintassi Thymeleaf per iterare una lista. -->
20             <span th:each="book : ${books}">
21                 <tr>
22                     <th th:text="${book.isbn}"
23                         style="width:15%"></th>
24                     <td class="fs-5 mb-5"
25                         th:text="${book.title}"></td>
26                     <td class="fs-5 mb-5"
27                         th:text="${book.genre}"></td>
28                     <td>
29                         <span th:each="a, iter : ${book.authors}">
30                             <span th:text="${a.firstName} + ' ' +
31                               ${a.lastName}"></span>
32                             <span th:if="${!iter.last}">, </span>
33                         </span>
34                     </td>
35                     <td style="width:20%">
```

```

29          <a id="detBook"
30              th:href="@{/library/{isbn}(isbn=${book.isbn})}"
31              class="btn btn-primary">
32                  <i class="bi bi-eye"></i>
33          </a>
34          <a sec:authorize="isAuthenticated()"
35              th:href="@{/library/update/{isbn}(isbn=${book.isbn})}"
36              class="btn btn-primary">
37                  <i class="bi bi-pen"></i>
38          </a>
39          <a sec:authorize="isAuthenticated()"
40              th:href="@{/library/delete/{isbn}(isbn=${book.isbn})}"
41              class="btn btn-primary">
42                  <i class="bi bi-trash"></i>
43          </a>
44      </td>
45  </tr>
46 </tbody>
47 </table>

```

I tasti della tabella che permettono la modifica del catalogo sono nascosti ad utenti non autenticati.

Nella vista relativa ai libri, il filtraggio dei libri in base al genere e data di pubblicazione è permesso tramite due menu a discesa (*dropdown menu*). La ricerca in base al titolo tramite una barra di ricerca.

Nella vista degli autori, la ricerca tramite nome e cognome è permessa mediante un *form*.

detBook.html è utilizzata per visualizzare i dettagli di un libro. Questa vista si può ottenere cliccando sul bottone dei dettagli all'interno della tabella dei libri.

editBook.html è utilizzata per l'inserimento e la modifica di un libro, quando nella vista *books* si clicca sul bottone di inserimento, essa mostra il titolo *New Book*. Altrimenti, se si clicca sul bottone relativo alla modifica, il titolo mostrato è *Edit Book*.

La pagina che permette la modifica degli autori, *editAuthor*, risulta molto simile. Entrambe le viste permettono la modifica dei dati, o l'inserimento di un libro, tramite un *form*.

La vista *errorPage.html* è utilizzata per la visualizzazione degli errori.

Nella corrente sezione sono state descritte delle porzioni di codice delle viste. Il codice completo è presente nel sorgente dell'applicazione.

6

VALIDAZIONE E SICUREZZA

6.1 VALIDAZIONE DEGLI INPUT

Spring Boot fornisce dei meccanismi per definire dei vincoli sui campi di una classe tramite annotazioni basate sulle funzionalità dell'*API Bean Validation*. È possibile visualizzare gli errori di convalida nelle viste di inserimento.

La validazione per il codice ISBN si presenta con una struttura specifica che è possibile definire tramite un'*espressione regolare*, ovvero uno strumento potente quando si deve lavorare con delle stringhe che consente di convalidare dati oppure effettuare ricerche.

Il codice deve avere le seguenti caratteristiche:

- ha una lunghezza tra 10 o 13 cifre.
- è suddiviso in 5 parti da simboli *dash*.
- la prima parte, che identifica il mondo del libro, è uguale a 978 o 979.
- la seconda parte, prefisso dell'area linguistica, può avere da 1 a 5 cifre.
- la terza parte, il prefisso editore, può avere da 2 a 6 cifre.
- quarta parte, numero di identificazione del titolo, può avere un numero di cifre variabile.
- l'ultima parte, il numero di controllo, è un numero da 0 a 9. [28]

Dopo aver scaricato ed installato la dipendenza necessaria per la validazione, *Spring Boot Starter Validation*, all'interno del package `com.library.web.app.validator` si crea l'annotazione `IsbnCodeConstraint`, che definisce i vincoli sul codice ISBN.

```

@Constraint(validatedBy = IsbnCodeValidator.class)
@Target({ ElementType.METHOD, ElementType.FIELD })
3 @Retention(RetentionPolicy.RUNTIME)
public @interface IsbnCodeConstraint {
    String message() default "Invalid ISBN code.";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
8 }

```

Si implementa la classe *IsbnCodeValidator* che applica la logica di validazione in base alle regole definite in precedenza.

```

public class IsbnCodeValidator implements ConstraintValidator<
    IsbnCodeConstraint, String>{
2   @Override
    public boolean isValid(String isbnValue, ConstraintValidatorContext
        context) {
        return isbnValue != null && isbnValue.matches("^(978|979)
            -[0-9]{1,5}-[0-9]{2,6}-[0-9]+-[0-9]{1}$")
            && (isbnValue.length() > 13) && (isbnValue.length() < 18);
    }
7 }

```

Con il metodo *matches* si determina se le stringhe appartengono o meno all'insieme di stringhe definito dall'espressione regolare. L'espressione regolare è caratterizzata da:

- i simboli ^ e \$ indicano, rispettivamente, l'inizio e la fine della stringa.
- | identifica l'operatore logico OR.
- [0-9]{1, 5} identifica il prefisso dell'area linguistica, dunque, una sequenza di numeri composta da 1 a 5 cifre.
- [0-9]{2, 6} identifica il prefisso numero di identificazione del titolo, dunque, una sequenza di numeri composta da 2 a 6 cifre.
- [0-9] rappresenta il numero di identificazione del titolo.
- [0-9]{1} è un numero da 0 a 9.

Dopo aver creato l'annotazione per validare il codice ISBN, nelle classi DTO, poiché l'inserimento avviene tramite oggetti DTO, vengono inserite le annotazioni necessarie per applicare la logica di validazione.

Per Book:

```

public class BookDto {
    @IsbnCodeConstraint
    private String isbn;
    @NotEmpty(message = "Title cannot be empty.")
    private String title;
    @NotEmpty(message = "Genre cannot be empty.")
    private String genre;
    @Size(min = 1, max = 2000, message = "The plot of the book must be
        between 1 and 2000 words.")
    private String plot;
    @NotNull(message = "Publication date cannot be empty.")
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate publicationDate;
    @Min(value = 1, message = "Must be equal or greater than 1.")
    private int numberPages;
    private Set<AuthorSlimDto> authors;

```

Per Author:

```

public class AuthorSlimDto {
    private Long id;
    @NotEmpty(message = "First name cannot be empty.")
    private String firstName;
    @NotEmpty(message = "Last name cannot be empty.")
    private String lastName;

```

Oltre a `@IsbnCodeConstraint`, vengono inserite altre convalide fornite da *Jakarta Bean Validation*, quali `@NotEmpty`, `@Size`, `@NotNull`, `@DateTimeFormat`, `@Min`. Tramite *message* è possibile specificare un messaggio di errore da restituire quando la convalida non viene rispettata.

Il meccanismo di validazione viene implementato nelle classi *BookWebController* e *AuthorWebController*, in particolare nei metodi POST, tramite `@Valid` per attivare la convalida sugli oggetti *book* e *author*.

Convalida per l'oggetto *book*:

```

@PostMapping("/insert/saveNewBook")
public String saveNewBook(Model model, @Valid @ModelAttribute BookDto
    book, BindingResult result, @ModelAttribute("authorNames") String
    authorNames) {
    if (result.hasErrors()) {
        4      model.addAttribute("message", "New Book");
        return "editBook";

```

```

        }
    Set<AuthorSlimDto> authors = service.fillAuthorsSetByStringName(
        authorNames);
    book.setAuthors(authors);
9     service.insertNewBook(book);
    return "redirect:/library";
}
@PostMapping("/update/saveBook")
public String saveBook(Model model, @Valid @ModelAttribute("bookDto")
    BookDto book, BindingResult result, @ModelAttribute("authorNames")
    String authorNames) {
14    Set<AuthorSlimDto> authors = service.fillAuthorsSetByStringName(
        authorNames);
    book.setAuthors(authors);
    if (result.hasErrors()) {
        model.addAttribute("authorNames", service.fillStringNameBySet(
            book.getAuthors()));
        model.addAttribute("message", "Edit Book");
19        return "editBook";
    }
    service.saveBook(book);
    return "redirect:/library";
}
24

```

Convalida per l'oggetto *author*:

```

@PostMapping("/update/saveAuthor")
2  public String saveAuthor(@Valid @ModelAttribute("author") AuthorSlimDto
    author, BindingResult result) {
    if (result.hasErrors()) {
        return "editAuthor";
    }
    service.saveAuthor(author);
7     return "redirect:/library/author";
}

```

Nelle viste, è possibile recuperare i messaggi di errore tramite i tag *th:errorclass* e *th:error*.

ISBN
Invalid ISBN code.

Title
Title cannot be empty.

Genre
Genre cannot be empty.

Publication Date (YYYY-MM-DD)
Publication date cannot be empty.

Number of Pages
0
Must be equal or greater than 1.

Plot
The plot of the book must be between 1 and 2000 words.

Figura 18: Esempio di convalida non rispettata

6.2 SPRING SECURITY

In un'applicazione web la sicurezza è fondamentale, impedisce l'accesso e la modifica dei dati da parte di soggetti ostili e permette anche di definire quali risorse possono essere accedute da determinati utenti. Ad esempio, nell'applicazione *My Library* gli utenti anonimi non posso accedere alle funzionalità di modifica, inserimento ed eliminazione delle risorse. Caratteristiche della sicurezza:

- *Autenticazione*: processo di identificazione dell'utente che procede alla richiesta del servizio.
- *Autorizzazione*: determina a quali risorse l'utente può accedere.
- *Credenziali di accesso al Database*: devono essere criptate e non accessibili.
- *Criptare i dati sensibili*: le password devono essere criptate prima della memorizzazione nel database.

È possibile ottenere le precedenti caratteristiche utilizzando *Spring Security*, il framework che supporta le funzionalità per la gestione della sicurezza in un'applicazione web.

Dopo aver scaricato ed installato la dipendenza di *Spring Security*, all'interno del package *com.library.webapp.security* si implementa *SecurityConfig*, ovvero la classe di configurazione per la sicurezza dell'applicazione.

```

@Configuration
2 @EnableWebSecurity
    public class SecurityConfig{
        @Bean
        public PasswordEncoder passwordEncoder() {
            return new BCryptPasswordEncoder();
        }
        @Bean
        public InMemoryUserDetailsManager userDetailsService() {
            UserDetails admin = User.withUsername("admin")
12            .password(passwordEncoder().encode("adminPass"))
            .roles("ADMIN")
            .build();

            return new InMemoryUserDetailsManager(admin);
        }
        @Bean
        public SecurityFilterChain filterChain(HttpSecurity http,
            MvcRequestMatcher.Builder mvc) throws Exception {
            http
                .authorizeHttpRequests((authz) -> authz
22                .requestMatchers(mvc.pattern("/sfondo.jpg"), mvc.pattern("/css
/**")).permitAll()
                    .requestMatchers(mvc.pattern("/api/**")).permitAll()
                    .requestMatchers(mvc.pattern("/")).permitAll()
                    .requestMatchers(mvc.pattern("/login/**")).permitAll()
                    .requestMatchers(mvc.pattern("/library/update/**")).hasRole("ADMIN")
27                    .requestMatchers(mvc.pattern("/library/insert/**")).hasRole("ADMIN")
                    .requestMatchers(mvc.pattern("/library/delete/**")).hasRole("ADMIN")
                    .requestMatchers(mvc.pattern("/library/author/update/**")).hasRole("ADMIN")
                    .requestMatchers(mvc.pattern("/library/**")).permitAll()
                    .requestMatchers(mvc.pattern("/library/author/**")).permitAll()
32                    .anyRequest().authenticated()
                )
            .formLogin((formLogin) ->

```

```

        formLogin
        .usernameParameter("userId")
37      .passwordParameter("password")
        .loginPage("/login")
        .loginProcessingUrl("/login")
        .failureUrl("/login?error=true")
    )
42      .logout((logout) ->
        logout
        .logoutUrl("/logout")
        .logoutSuccessUrl("/")
    );
47      return http.build();
}
@Bean
MvcRequestMatcher.Builder mvc(HandlerMappingIntrospector introspector) {
    return new MvcRequestMatcher.Builder(introspector);
52 }
}

```

La classe ha le seguenti caratteristiche:

- le annotazioni `@Configuration`, per indicare che la classe ha metodi `@Bean`, ed `@EnableWebSecurity`, appartenente a *Spring Security*.
- il metodo `passwordEncoder()`, che definisce `BCryptPasswordEncoder` come *bean* della configurazione.
- il metodo `userDetailsService()`, che definisce i dettagli di un solo utente, con `username admin`, salvato in memoria.
- il metodo `filterChain()`, che permette di specificare le autorizzazioni che gli utenti possono avere. In questo metodo sono stati anche configurati il *login* ed il *logout*. In particolare, con `loginPage()` si definisce la pagina di login, con `loginProcessingUrl()` l'URL a cui inviare i dati inseriti per l'accesso, con `failureUrl()` si definisce la pagina se l'accesso è fallito, con `logoutUrl()` la pagina per il logout e con `logoutSuccessUrl()` la pagina di reindirizzamento se il logout ha successo.
- il metodo `mvc`, che crea un oggetto `MvcRequestMatcher` utilizzato per gestire gli endpoint nei filtri di sicurezza. [29]

Dunque, come menzionato, nell'applicazione è stato configurato un solo utente e non è presente il processo che permette la registrazione.

Questa scelta implementativa è stata fatta a scopo di test e dimostrazione.

Dopo aver configurato la sicurezza, si crea il controller per gestire il processo di *login*, composto solo da un metodo. Il processo di *logout* viene gestito automaticamente da *Spring Security* sull'URL specificato, ad esempio */logout*.

```

1 @Controller
2 @RequestMapping("/login")
3 public class LoginWebController {
4     @GetMapping
5     public String getlogin(Model model) {
6         return "login";
7     }
8 }
```

Nell'annotazione `@RequestMapping` viene specificato l'URL corrispondente alla pagina di *login* della classe di configurazione. Il metodo `getlogin()` restituisce il nome della vista *login*. Essa si presenta con il titolo corrispondente ed un *form* in cui è possibile inserire le credenziali di accesso.

```

1 <div class="text-center mb-5">
2     <h1 id="titleH1" class="fw-bolder"> Login </h1>
3 </div>
4
5     <div class="row gx-5 justify-content-center">
6         <div class="col-lg-8 col-xl-6">
7
8             <form id="formLogin" name="formLogin" th:action="@{/login}"
9                 method="post">
10                <div th:if="${param.error}">
11                    <div class="alert alert-danger">
12                        <span th:text="#{login.formerrmsg}"></span>
13                    </div>
14                <!-- UserId input -->
15                <div class="form-outline mb-4">
16                    <input type="text" id="userId" name="userId"
17                         class="form-control" />
18                    <label class="form-label" for="userId">Username</label>
19                </div>
20
21                <!-- Password input -->
22                <div class="form-outline mb-4">
```

```

22      <input type="password" id="password" name="password"
23          class="form-control" />
24      <label class="form-label" for="password">Password</label>
25      </div>
26
27      <!-- Submit button -->
28      <input id="submitlogin" name="submitlogin" type="submit"
           value="Log In" class="btn btn-primary btn-block
           mb-4"></input>
29  </form>

```

Se le credenziali non sono corrette, viene visualizzato un messaggio di errore, specificato nel file *message.properties* in *main/java/resources*.

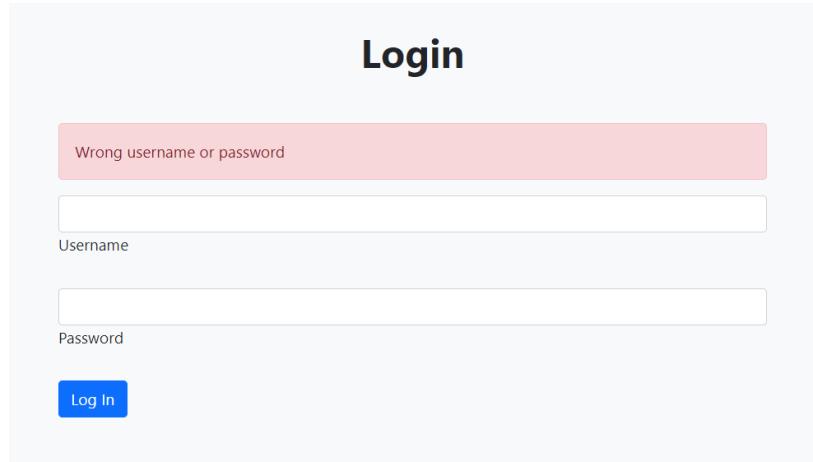


Figura 19: Messaggio di errore login credenziali errate

È necessario informare *Spring* dell'esistenza di *message.properties* e ciò viene fatto in *application.yml*.

```

16
17  messages:
18      basename: message
19

```

Figura 20: Definizione di message in application.yml

A seconda che l'utente sia autenticato o meno, nella *Navigation Bar* compare un bottone di *Logout/Login*.

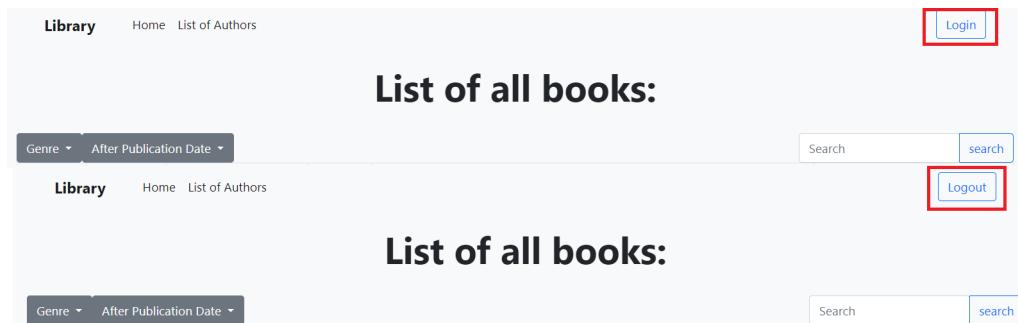


Figura 21: Bottoni di Login e Logout

7

TESTING

La fase di testing è fondamentale per valutare il corretto funzionamento dell'applicazione. È buona pratica testare ogni singolo componente per controllarne l'effettivo funzionamento. Questa attività è chiamata *Unit testing* e con *test di integrazione* si verifica la corretta interazione tra i vari componenti.

I test possono essere scritti prima della stesura del codice effettivo (*Test-Driven Development (TDD)*), ciò garantisce che vengono soddisfatti i requisiti richiesti, oppure dopo la stesura del codice (*Test-Later Development (TLD)*) per verificarne il funzionamento.

Vantaggi nell'utilizzo dei test:

- Qualità: un software opportunamente testato garantisce che ci siano minor problemi possibili.
- Modifica semplificata: con i test è possibile controllare il funzionamento dopo delle modifiche al codice. Se un test fallisce, è facile identificare la modifica che ha causato il fallimento.
- Integrazione dei componenti facilitata: risulta più facile l'integrazione tra componenti testati in precedenza.
- Documentazione: i test supportano la documentazione di un software.

Esistono software e librerie che aiutano lo sviluppatore nell'implementazione dei test, ad esempio *JUnit*, un framework *open-source* che permette di scrivere test unitari e di integrazione. In questa tesi viene utilizzato *Junit 5*, che si pone l'obbiettivo di supportare le nuove funzionalità, come ad esempio le espressioni lambda, di Java 8 e versioni successive.

In questa sezione, non vengono mostrati tutti i test implementati nell'applicazione, ma una porzione di essi, si cerca di esaminare gli aspetti fondamentali. I test completi sono presenti nel sorgente del codice.

7.1 LIVELLO DI DOMINIO

Poiché le classi entità non possiedono logica di business, non hanno bisogno di essere testate.

Nello strato di dominio vengono definite le classi DTO e la corrispondente classe *MapStructMapperImpl*, che utilizza il tool *MapStruct* per eseguire la mappatura tra entità e DTO e viceversa. Di seguito i test, all'interno del package *com.library.webapp.dto.mapper* in *src/test/java*, per verificare la corretta mappatura eseguita da *MapStructMapperImpl*:

```

@ExtendWith(MockitoExtension.class)
2 class MapStructMapperImplTest {

    @InjectMocks
    private MapStructMapperImpl mapper;

7    @Test
    void testMappingBookToBookDto() {
        Author authorToMapSlim = new Author(1L, "firstName", "lastName", null);
        AuthorSlimDto authorSlimDtoResult = new AuthorSlimDto(1L, "firstName",
        "lastName");
        Book bookToMap = new Book("978-88-000-0000-1", "title", "genre", "plot"
        , LocalDate.parse("2000-01-01"), 100, Set.of(authorToMapSlim));
12    BookDto result = mapper.bookToBookDto(bookToMap);

        assertEquals(result.getIsbn(), bookToMap.getIsbn());
        assertEquals(result.getTitle(), bookToMap.getTitle());
        assertEquals(result.getGenre(), bookToMap.getGenre());
17    assertEquals(result.getPlot(), bookToMap.getPlot());
        assertEquals(result.getPublicationDate(), bookToMap.getPublicationDate
        ());
        assertEquals(result.getNumberPages(), bookToMap.getNumberPages());
        assertThat(result.getAuthors()).contains(authorSlimDtoResult);
    }
22
    @Test
    void testMappingBookDtoToBook(){
        AuthorSlimDto authorSlimToMap = new AuthorSlimDto(1L, "firstName", "lastName");
        Author authorResult = new Author(1L, "firstName", "lastName", null);
27    BookDto bookDtoToMap = new BookDto("978-88-000-0000-1", "title", "genre
        ", "plot", LocalDate.parse("2000-01-01"), 100, Set.of(authorSlimToMap))
        ;
        Book result = mapper.bookDtoToBook(bookDtoToMap);
        authorResult.setBooks(Set.of(result));
    }
}

```

```

    assertEquals(result.getIsbn(), bookDtoToMap.getIsbn());
32  assertEquals(result.getTitle(), bookDtoToMap.getTitle());
    assertEquals(result.getGenre(), bookDtoToMap.getGenre());
    assertEquals(result.getPlot(), bookDtoToMap.getPlot());
    assertEquals(result.getPublicationDate(), bookDtoToMap.
getPublicationDate());
    assertEquals(result.getNumberPages(), bookDtoToMap.getNumberPages());
37  assertThat(result.getAuthors()).contains(authorResult);
}

```

I test per la mappatura che riguardano gli autori hanno la stessa logica e simile implementazione.

L'annotazione `@ExtendWith(MockitoExtension.class)` è utilizzata in `JUnit5` per aggiungere alla classe di test l'estensione `Mockito`. `Mockito` è un framework che permette la realizzazione di oggetti *mock*, ovvero oggetti che simulano il comportamento di oggetti reali. Infatti, con `@InjectMocks` viene creata un'istanza della classe `MapStructMapperImpl` per poterla testare.

7.2 LIVELLO DI PERSISTENZA

Per testare il livello di persistenza si usa l'annotazione `@DataJpaTest`, che disabilita la configurazione automatica completa e applica solo quella relativa alle componenti Jpa. I test annotati con `@DataJpaTest` vengono eseguiti nella propria transazione e ripristinati al termine di ogni test [30].

Dopo aver scaricato la dipendenza di H2, un database incorporato, *open source* ed *in-memory*, si definiscono le proprietà di connessione ad H2 in `src/main/resource`:

```

*databaseH2.properties ×
1#Test Properties
2spring.datasource.driverClassName=org.h2.Driver
3spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE;MODE=MySQL;NON_KEYWORDS=USER
4spring.datasource.username=sa
5spring.datasource.password=sa
6spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect
7spring.jpa.hibernate.ddl-auto=create-drop
8spring.jpa.show-sql=true

```

Figura 22: databaseH2.properties

Si crea la classe di configurazione che cerca il file di proprietà in Figura 22 e crea un *DataSouce* utilizzando le proprietà descritte nel file.

`@TestConfiguration`

```

2 @EnableJpaRepositories(basePackages = "com.library.webapp.repository")
  @EntityScan("com.library.webapp.entity")
  @PropertySource("databaseH2.properties")
  @EnableTransactionManagement
  public class H2Config {
7    @Autowired
      Environment env;
      @Bean
      public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
12      dataSource.setDriverClassName(env.getProperty("datasource.
        driverClassName"));
        dataSource.setUrl(env.getProperty("datasource.url"));
        dataSource.setUsername(env.getProperty("datasource.username"));
        dataSource.setPassword(env.getProperty("datasource.password"));
        return (DataSource) dataSource;
17    }
}

```

Analisi della classe:

- `@TestConfiguration`: per definire *bean* aggiuntivi per i test.
- `@EnableJpaRepositories(basePackages = "com.library.webapp.repository")`: serve per abilitare le componenti JPA definite nel package indicato.
- `@EntityScan("com.library.webapp.entity")`: definisce il package nel quale scansionare le entità.
- `@PropertySource("databaseH2.properties")`: per definire sorgenti di proprietà aggiuntive.
- `@EnableTransactionManagement`: serve per abilitare il supporto transazionale nelle classi di configurazione.

Si usa un oggetto *Environment* per accedere alle proprietà definite dal file specificato in `@PropertySource`.

Successivamente, si descrivono le classi di test implementate.

All'interno del package `com.library.webapp.jpaTest` in `src/test/java`, le classi `BookJpaTest` ed `AuthorJpaTest`, verificano che le entità vengono salvate correttamente nel database e testano la loro uguaglianza basata sull'`hashcode` definito nelle classi entità. Classe `AuthorJpaTest`:

```

@DataJpaTest
2 @ContextConfiguration(

```

```

        classes = { H2Config.class },
        loader = AnnotationConfigContextLoader.class)
class AuthorJpaTest {
    @Autowired
7   private TestEntityManager entityManager;
    @Test
    void testJpaMapping() {
        Author saved = entityManager.persistAndFlush(new Author(null , "
        firstName", "lastName", null));
        assertThat(saved.getFirstName()).isEqualTo("firstName");
12  assertThat(saved.getLastName()).isEqualTo("lastName");
        assertThat(saved.getId()).isNotNull();
        assertThat(saved.getId()).isPositive();
    }
    @Test
17  void testAutoincrementID() {
        Author author1 = entityManager.persistAndFlush(new Author(null , "
        firstName1", "lastName1", null));
        Author author2 = entityManager.persistAndFlush(new Author(null , "
        firstName2", "lastName2", null));
        assertThat(author1.getId()).isNotNull();
        assertThat(author2.getId()).isNotNull();
22  assertThat(author2.getId()).isGreaterThan(author1.getId());
    }
    @Test
    void test2AuthorsNotEqual(){
        Author author1 = new Author(null, "Agatha", "Christie", null);
27  Author author2 = new Author(null, "George", "Orwell", null);
        assertFalse(author1.equals(author2));
    }
    @Test
    void test2AuthorsNotEqual_persist() {
32  Author result1 = entityManager.persistAndFlush(new Author(null, "Agatha"
        , "Christie", null));
        Author result2 = entityManager.persistAndFlush(new Author(null, "George"
        , "Orwell", null));
        assertFalse(result1.equals(result2));
    }
    @Test
37  void test2AuthorsEqual_OneAuthorPersist() {
        Author result = entityManager.persistAndFlush(new Author(null, "Agatha"
        , "Christie", null));
        Author author = new Author(null, "Agatha", "Christie", null);
        assertTrue(result.equals(author));
    }
42  @Test
    void test2AuthorsEqual_AllPersist() {
        Author result1 = entityManager.persistAndFlush(new Author(null, "Agatha"

```

```

    ", "Christie", null));
Author result2 = entityManager.persistAndFlush(new Author(null, "Agatha
", "Christie", null));
assertTrue(result1.equals(result2));
47 }
}

```

La classe di test recupera la configurazione precedentemente descritta. I test implementati eseguono i seguenti controlli sull'entità *Author*:

- *testJpaMapping()* e *testAutoincrementID()* verificano che le entità vengono salvate correttamente, ovvero, controllano che le chiavi primarie vengono generate come previsto.
- *test2AuthorsNotEqual()* e *test2AuthorsNotEqual_persist()* verificano che se due entità non possiedono lo stesso nome e cognome, non risultano uguali.
- *test2AuthorsEqual_OneAuthorPersist()* e *test2AuthorsEqual_AllPersist()* verificano che due entità risultano uguali se hanno lo stesso nome e cognome, indipendentemente dall'id. Ad esempio, nel primo metodo l'id è diverso perché uno è generato dopo la persistenza mentre l'altro no, ma risultano uguali dato che il nome ed il cognome coincidono. Questo comportamento è voluto, se viene inserito un libro ed esiste già nel database un autore con il nome specificato nell'inserimento, il libro viene direttamente associato alle opere di quell'autore.

La classe *BookJpaTest* esegue dei controlli similari sul salvataggio e l'ugualanza basata sul codice ISBN.

Nel package *com.library.webapp.repository*, si implementano le classi *BookRepositoryTest* e *AuthorRepositoryTest*, nelle quali vengono testati solo i metodi in aggiunta all'interfaccia *JpaRepository*, poiché questi ultimi sono già stati testati dagli sviluppatori.

Ad esempio, nella classe *BookRepositoryTest* viene testata la ricerca di un libro in base al titolo:

```

@DataJpaTest
2 @ContextConfiguration(
    classes = { H2Config.class },
    loader = AnnotationConfigApplicationContextLoader.class)
class BookRepositoryTest {
    @Autowired
7     BookRepository bookRepository;
}

```

```

    @Autowired
    private TestEntityManager entityManager;
    @Test
    void findBookTitleContains() {
12     Book bookFound1 = new Book("978-88-000-0000-1", "TitleBookOk", "GenreBook", null, LocalDate.parse("2000-02-01"), 0, null);
        entityManager.persist(bookFound1);
        Book bookFound2 = new Book("978-88-000-0000-2", "AnotherTitleForTheBook", "GenreBook", null, LocalDate.parse("2000-01-01"), 0, null);
        entityManager.persist(bookFound2);
        Book notFound = new Book("978-88-000-0000-3", "TitleNoOk", "GenreBook", null, LocalDate.parse("2000-03-01"), 0, null);
17     entityManager.persist(notFound);
        List<Book> result = bookRepository.findByTitleContains("Book");
        assertThat(result).containsExactly(bookFound2, bookFound1);
    }
    @Test
22    void findBookTitleContains_isEmpty() {
        List<Book> result = bookRepository.findByTitleContains("title");
        assertThat(result).isEmpty();
    }
}

```

Nella classe *AuthorRepositoryTest*, viene testata la ricerca di un autore in base al nome e cognome:

```

@DataJpaTest
@ContextConfiguration(
    classes = { H2Config.class },
    loader = AnnotationConfigContextLoader.class)
4 class AuthorRepositoryTest {
    @Autowired
    private AuthorRepository authorRepository;
    @Autowired
9     private TestEntityManager entityManager;
    @Test
    void testGetAuthorByName() {
        Book book = new Book("978-88-111-1111-1", "titleA", "genre", "plot",
            LocalDate.parse("2000-01-01"),
            100, null);
14     Author author = new Author(null, "firstName", "lastName", Set.of(book));
        book.setAuthors(Set.of(author));
        entityManager.persistAndFlush(author);
        Author result = authorRepository.findByNameLike(author.getFirstName(),
            author.getLastName());
        assertThat(result).isEqualTo(author);
19    }
}

```

```

    @Test
    void testGetAuthorByName_isEmpty() {
        Author result = authorRepository.findByNameLike("FirstName", "LastName"
        );
        assertThat(result).isNull();
24    }
}

```

7.3 LIVELLO DI SERVIZIO

All'interno del package `com.library.webapp.service` in `src/test/java`, viene implementata la classe `LibraryServiceImplTest` per testare il livello di servizio. Nel seguente codice, non sono riportati tutti i metodi di test, ma solo alcuni a titolo di esempio:

```

@ExtendWith(MockitoExtension.class)
class LibraryServiceImplTest {
    @InjectMocks
    private LibraryServiceImpl libraryService;
5     @Mock
    private MapStructMapper mapper;
    @Mock
    private BookRepository bookRepository;
    @Mock
10    private AuthorRepository authorRepository;
    @Test
    void testSelAllBooks() {
        Book book1 = new Book("978-88-000-0000-1", "Book1");
        Book book2 = new Book("978-88-000-0000-2", "Book2");
15    when(bookRepository.findAll()).thenReturn(List.of(book1, book2));
        BookDto bookDto1 = new BookDto("978-88-000-0000-1", "Book1", null, null
            , null, 0, null);
        BookDto bookDto2 = new BookDto("978-88-000-0000-2", "Book2", null, null
            , null, 0, null);
        when(mapper.bookToBookDto(book1)).thenReturn(bookDto1);
        when(mapper.bookToBookDto(book2)).thenReturn(bookDto2);
20    List<BookDto> result = libraryService.selAllBooks();
        assertThat(result).containsExactly(bookDto1, bookDto2);
    }
    @Test
    void testSelAllBook_NoFound() throws Exception {
25    when(bookRepository.findAll()).thenReturn(List.of());
        NotFoundException exception = assertThrows(NotFoundException.class, ()
            -> {
            libraryService.selAllBooks();
        });
}

```

```

        assertEquals("There is no book!", exception.getMessage());
30    }
    @Test
    void testInsertNewBook() {
        Author author = new Author(null, "firstName", "lastName", null);
        Book newBook = spy(new Book("978-88-000-0000-1", "titleBook", null,
            null, null, 0, Set.of(author)));
35    AuthorSlimDto authorSlim = new AuthorSlimDto(null, "firstName",
        "lastName");
        BookDto newBookDto = new BookDto("978-88-000-0000-1", "titleBook", null
            , null, null, 0, Set.of(authorSlim));
        when(bookRepository.findById(anyString())).thenReturn(Optional.empty());
        when(mapper.bookDtoToBook(newBookDto)).thenReturn(newBook);
        when(authorRepository.findByNameLike("firstName", "lastName")).thenReturn(null);
40    when(bookRepository.save(any(Book.class))).thenReturn(newBook);
        Book result = libraryService.insertNewBook(newBookDto);
        assertThat(result).isSameAs(newBook);
        InOrder inOrder = Mockito.inOrder(newBook, bookRepository);
        inOrder.verify(bookRepository).save(newBook);
45    }
    @Test
    void testSaveAuthor() {
        AuthorSlimDto authorToUpdate = spy(new AuthorSlimDto(1L, "authorOldFirstName", "lastName"));
        authorToUpdate.setFirstName("authorNewFirstName");
50    Author updated = new Author(1L, "authorNewFirstName", "lastName", null);
        when(authorRepository.findById(1L)).thenReturn(Optional.of(updated));
        when(authorRepository.save(any(Author.class))).thenReturn(updated);
        when(mapper.authorSlimDtoToAuthor(authorToUpdate)).thenReturn(updated);
        libraryService.saveAuthor(authorToUpdate);
55    assertEquals(authorToUpdate.getFirstName(), updated.getFirstName());
        assertTrue(true);
    }

```

Si osservi l'uso dei metodi di Mockito `when(metodoX).thenReturn(y)` all'interno dei test, essi vengono utilizzati per poter testare il livello di servizio in autonomia. Il metodo definito in `when` non viene eseguito, ma attraverso `thenReturn` viene impostato il valore di ritorno specificato, ad esempio `y`. Dunque, nei metodi di test viene definito il comportamento previsto dallo strato di repository e dalla classe `MapStructMapper`. Dopodiché, si verifica il comportamento dello strato di servizio in base ai risultati previsti.

Ad esempio, nel metodo di inserimento `testInsertNewBook()`, si ipotizza

che il libro e l'autore inseriti non siano già presenti nel database, per evitare l'eccezione *DuplicateException* (testata nel metodo *testInsertNewBook_Exceptions()* del codice sorgente). Si definisce *newBook* come oggetto restituito da *bookRepository.save* e si verifica che *result*, l'oggetto restituito da *libraryService.insertNewBook*, sia uguale a *newBook*.

7.4 LIVELLO WEB O CONSUMER

7.4.1 Test componenti @RestController

All'interno del package *com.library.webapp.restcontroller* in *src/test/java* vengono implementate le classi di test per *BookRestController* e *AuthorRestController*. A titolo di esempio sono riportati due test per le due classi *RestController*.

Nel seguente test di *BookRestControllerTest* si verifica che alla richiesta all'endpoint */api/book* si ottiene la lista di tutti i libri o l'eccezione *NotFoundException*:

```

@WebMvcTest/controllers = BookRestController.class)
@Import(SecurityConfig.class)
3 class BookRestControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private LibraryService service;
8 @Test
    void test GetAllBook() throws Exception {
        when(service.selAllBooks()).thenReturn(List.of(
            new BookDto("978-88-111-1111-1", "Title A", "Genre", "plot",
                LocalDate.parse("2000-01-01"), 100, Set.of(new AuthorSlimDto(1L
            , "FirstNameA", "LastNameA"))),
            new BookDto("978-88-111-1111-2", "Title B", "Genre", "plot",
                LocalDate.parse("2000-02-02"), 200, Set.of(new AuthorSlimDto(2L
            , "FirstNameB", "LastNameB"))))
        ));
        this.mockMvc.perform(get("/api/book"))
            .andExpect(status().isOk())
18         .andExpect(jsonPath("$.isbn", is("978-88-111-1111-1")))
            .andExpect(jsonPath("$.title", is("Title A")))
            .andExpect(jsonPath("$.genre", is("Genre")))
            .andExpect(jsonPath("$.plot", is("plot")))
            .andExpect(jsonPath("$.publicationDate", is("2000-01-01")))
23         .andExpect(jsonPath("$.numberPages", is(100)))
            .andExpect(jsonPath("$.authors.[0].id", is(1)))

```

```

        .andExpect(jsonPath("$.authors.[0].firstName", is("FirstNameA"))
    ))
    .andExpect(jsonPath("$.authors.[0].lastName", is("LastNameA")))
)
.andExpect(jsonPath("$.isbn", is("978-88-111-1111-2")))
28 .andExpect(jsonPath("$.title", is("Title B")))
.andExpect(jsonPath("$.genre", is("Genre")))
.andExpect(jsonPath("$.plot", is("plot")))
.andExpect(jsonPath("$.publicationDate", is("2000-02-02")))
.andExpect(jsonPath("$.numberPages", is(200)))
33 .andExpect(jsonPath("$.authors.[0].id", is(2)))
.andExpect(jsonPath("$.authors.[0].firstName", is("FirstNameB"))
))
.andExpect(jsonPath("$.authors.[0].lastName", is("LastNameB")))
);
}
@Test
38 void testAllBook_NoFound() throws Exception {
    when(service.selAllBooks()).thenThrow(new NotFoundException("There is
        no book!"));
    this.mockMvc.perform(get("/api/book"))
        .andExpect(status().isNotFound())
        .andExpect(jsonPath("$.code", is(404)))
43     .andExpect(jsonPath("$.message", is("There is no book!")));
}
}

```

Nel seguente test di *AuthorRestControllerTest* si verifica il funzionamento della richiesta */api/author/name/FirstName/LastName* per ricercare un autore con nome e cognome specificati, in questo caso *FirstName* e *LastName*:

```

@WebMvcTest/controllers = AuthorRestController.class)
@Import(SecurityConfig.class)
class AuthorRestControllerTest{
    @Autowired
    5 private MockMvc mockMvc;
    @MockBean
    private LibraryService service;
    @Test
    void testFindAuthorByName() throws Exception {
10     AuthorDto author = new AuthorDto(1L, "FirstName", "LastName",
        Set.of(new BookSlimDto("978-88-111-1111-1", "Title")));
        when(service.findAuthorByNameAndSurname("FirstName", "LastName")).
        thenReturn(author);
        this.mockMvc.perform(get("/api/author/name/FirstName/LastName"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.id", is(1)))
15         .andExpect(jsonPath("$.firstName", is("FirstName")))

```

```

    .andExpect(jsonPath("$.lastName", is("LastName")))
    .andExpect(jsonPath("$.books.[0].isbn", is("978-88-111-1111-1")))
    .andExpect(jsonPath("$.books.[0].title", is("Title")));
20  }
}

```

In entrambi le classi, sono presenti le annotazioni `@WebMvcTest`, usata per i test basati solo sulle componenti *Spring MVC*, ed `@Import(SecurityConfig.class)`, utilizzata per importare la classe di configurazione della sicurezza. Si usa l'annotazione `@MockBean` per creare un oggetto mock dello strato di servizio e simularne il comportamento.

7.4.2 Test componenti `@Controller`

All'interno del package `com.library.webapp.controller` in `src/test/java` vengono implementate i test per le classi controller `BookWebController`, `AuthorWebController`, `HomeWebController` e `LoginWebController`. Nelle classi di test si verifica il comportamento dei controller, ovvero, che viene restituita la vista corretta e che ad essa vengono passati i parametri appropriati. Viene importata la configurazione della sicurezza e tramite le annotazioni `@WithAnonymousUser` e `@WithUserDetails("admin")` vengono simulati i ruoli degli utenti.

Ad esempio, nella classe `HomeWebControllerTest` si verifica che se un utente anonimo effettua chiamata endpoint / si ha successo e si ottiene la vista `index`:

```

@WebMvcTest/controllers = HomeWebController.class)
@Import(SecurityConfig.class)
class HomeWebControllerTest {
4   @Autowired
    private MockMvc mockMvc;
    @Autowired
    private WebApplicationContext webApplicationContext;
    @BeforeEach
9   public void setup() {
        MockMvcBuilders
            .webAppContextSetup(webApplicationContext)
            .apply(springSecurity())
            .build();
14  }
    @Test
    @WithAnonymousUser
    void testSuccessEndpointHome() throws Exception {
        mockMvc.perform(get("/")).andExpect(status().is2xxSuccessful());
}

```

```

19    }
20    @Test
21    @WithAnonymousUser
22    void testReturnIndexView() throws Exception {
23        ModelAndViewAssert
24            .assertViewName(mockMvc.perform(get("/"))
25                .andReturn().getModelAndView(), "index");
26    }
27}

```

La classe *LoginWebControllerTest* non viene riportata perché molto simile ad *HomeWebControllerTest*.

Nelle classi *BookWebControllerTest* ed *AuthorWebControllerTest*, in aggiunta ai controlli sull'endpoint, si hanno anche i controlli sulle validazioni degli input. Ad esempio, nei test seguenti della classe *BookWebControllerTest* si effettua il controllo sulla chiamata all'endpoint */library*, analizzando il caso in cui sono o non sono presenti i libri nel catalogo, e sull'inserimento considerando parametri di input validi e non.

```

@WebMvcTest/controllers = BookWebController.class)
@Import(SecurityConfig.class)
3 class BookWebControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private LibraryService service;
8    @Autowired
    private WebApplicationContext webApplicationContext;
    @BeforeEach
    public void setup() {
        MockMvcBuilders
            .webAppContextSetup(webApplicationContext)
            .apply(springSecurity())
            .build();
    }
    @Test
18    @WithAnonymousUser
    void testReturnListBooks_BooksView() throws Exception {
        List<BookDto> books = List.of(new BookDto("978-88-111-1111-1", "Title",
            "Genre", "Plot", LocalDate.parse("2000-01-01"), 100, Set.of(new
            AuthorSlimDto(1L, "FirstName", "LastName"))));
        when(service.selAllBooks()).thenReturn(books);
23    mockMvc.perform(get("/library"))
        .andExpect(view().name("books"))
        .andExpect(model().attribute("books", books));
    }
}

```

```

        }

    @Test
28    @WithAnonymousUser
        void testNoBooks_ErrorView() throws Exception {
            when(service.selAllBooks())
                .thenThrow(new NotFoundException("There is no book!"));

33    mockMvc.perform(get("/library"))
            .andExpect(view().name("errorPage"))
            .andExpect(model().attribute("errorMessage", "There is no book!"));
        }

    @Test
38    @WithUserDetails("admin")
        void testInsertNewBook_RedirectBooksView() throws Exception {
            BookDto bookDto = new BookDto("978-88-111-1111-1", "Title", "Genre", "Plot",
                LocalDate.of(2000, 01, 01), 100,
                Set.of(new AuthorSlimDto(1L, "FirstName", "LastName")));
            Book book = new Book("978-88-111-1111-1", "Title", "Genre", "Plot",
                LocalDate.of(2000, 01, 01), 100,
                Set.of(new Author(1L, "FirstName", "LastName", null)));
            when(service.fillAuthorsSetByStringName("FirstName LastName"))
                .thenReturn(Set.of(new AuthorSlimDto(1L, "FirstName", "LastName")));
            when(service.insertNewBook(bookDto)).thenReturn(book);

48    mockMvc.perform(post("/library/insert/saveNewBook").with(csrf())
            .param("isbn", "978-88-111-1111-1")
            .param("title", "Title")
            .param("genre", "Genre")
            .param("plot", "Plot")
53    .param("numberPages", "100")
            .param("publicationDate", "2000-01-01")
            .param("authorNames", "FirstName LastName")
        )
            .andExpect(view().name("redirect:/library"))
58    .andExpect(model().errorCount(0));
            verify(service).insertNewBook(bookDto);
        }

    @Test
    @WithUserDetails("admin")
63    void testInsertNewBook_invalidInputBook() throws Exception {
        mockMvc.perform(post("/library/insert/saveNewBook").with(csrf())
            .param("isbn", "isbn")
            .param("title", "")
            .param("genre", "")
68    .param("plot", "")
            .param("numberPages", "0")
            .param("publicationDate", "")
            .param("authorNames", "FirstName LastName")
        )
    }
}

```

```

    )
73   .andExpect(view().name("editBook"))
    .andExpect(model().hasErrors())
    .andExpect(model().attributeHasFieldErrorCode("bookDto", "isbn", "IsbnCodeConstraint"))
        .andExpect(model().attributeHasFieldErrorCode("bookDto", "title", "NotEmpty"))
        .andExpect(model().attributeHasFieldErrorCode("bookDto", "genre", "NotEmpty"))
78   .andExpect(model().attributeHasFieldErrorCode("bookDto", "plot", "Size"))
        .andExpect(model().attributeHasFieldErrorCode("bookDto", "numberPages", "Min"))
        .andExpect(model().attributeHasFieldErrorCode("bookDto", "publicationDate", "NotNull"));
    }
}

```

7.4.3 Test delle viste

Per testare le viste della nostra applicazione si utilizza *HTMLUnit*, un browser senza GUI utilizzato a livello di programmazione e non direttamente da un utente. Ci permette di analizzare il codice HTML di un sito, interagendo con esso come farebbe un utente dal browser. Dopo aver scaricato la dipendenza, per utilizzare *HTMLUnit*, bisogna iniettare un'istanza di *WebClient*, classe utilizzata per simulare il browser web [31].

Nella classe *HomeWebControllerHtmlUnitTest*, presente nel codice sorgente, si verifica che:

- il titolo della pagina web sia *Library* e che l'intestazione all'interno della pagina sia *Welcome to Library!*.
- cliccando su *List of Books*, *List of Author* o *Login* l'utente viene reindirizzato agli URL corretti, rispettivamente, */library* e */library/author* e */login*.

Nella classe *LoginWebControllerHtmlUnitTest*, presente nel codice sorgente, si verifica che:

- il titolo della pagina e l'intestazione siano *Login*.
- se inserite le credenziali giuste, l'utente viene reindirizzato alla pagina *home*. Altrimenti, l'utente viene reindirizzato alla pagina corrente di login.

Nelle classi *BookWebControllerHtmlUnitTest* e *AuthorWebControllerHtmlUnitTest* la fase di test è più complicata essendo presenti più elementi. Nella prima, si verifica che:

- gli elementi della navigation bar siano *Home*, *List Of Author* e *Login* (oppure di *Logout* se l'utente risulta autenticato) e che se premuti l'utente viene reindirizzato agli URL corretti.
- se presenti dei libri nel catalogo, essi si ottengono nella tabella della vista *books*. Altrimenti, si ottiene una pagina di errore che contiene il bottone *Insert your first book!* se l'utente risulta autenticato. Se non è autenticato, è presente il bottone di *Login*. Si hanno anche i controlli sulla visibilità dei buttoni nella tabella che permettono la modifica del catalogo se l'utente è autenticato.
- se l'utente clicca sul bottone dei dettagli di un libro si ottiene la pagina corrispettiva e con gli attributi del libro corretti. Si hanno anche i controlli sulla visibilità dei buttoni di modifica ed eliminazione.
- se viene eliminato un libro, sia dalla tabella che dalla vista dei dettagli del libro, l'utente viene reindirizzato alla pagina della tabella dei libri.
- se inserito un libro con input validi, l'utente viene reindirizzato alla tabella dei libri. Altrimenti, se gli input non sono validi, l'utente viene reindirizzato alla pagina di inserimento. Se è già presente un libro con il codice ISBN inserito, si ha una pagina di errore. Per la modifica dei dati di un libro, si hanno dei controlli analoghi.
- nella vista *books*, è possibile scegliere, tramite un menu a discesa, un genere letterario e se presenti dei libri del genere specificato, si ottiene la lista. Altrimenti, si ha una pagina di errore. Si verifica anche l'uso del menu a discesa per filtrare i libri in base alla data di pubblicazione.
- tramite l'uso della barra di ricerca è possibile cercare uno o più libri in base al titolo.

In *AuthorWebControllerHtmlUnitTest* si verifica che:

- gli elementi della navigation bar siano *Home*, *List Of Books* e *Login* (oppure di *Logout* se l'utente risulta autenticato) e che se premuti l'utente viene reindirizzato agli URL corretti.

- se presenti degli autori nel catalogo, si ottengono nella tabella della vista *authors*. Altrimenti, si ottiene una pagina di errore che contiene il bottone *Insert your first book!* se l'utente risulta autenticato. Se non è autenticato, è presente il bottone di *Login*. Si hanno anche i controlli sulla visibilità dei bottoni nella tabella che permettono la modifica di un autore.
- se si modifica un autore con input validi, l'utente viene reindirizzato alla tabella degli autori. Altrimenti, viene reindirizzato alla pagina di modifica. Viene effettuato anche il controllo sugli attributi dell'autore presenti in *editAuthor* ed il controllo sulla visibilità del bottone di modifica.
- è possibile ricercare un autore in base al nome tramite le due barre di ricerca (una per il nome e l'altra per il cognome).

Di seguito, sono riportati degli esempi di test della classe *BookWebControllerHtmlUnitTest*:

```

1 @WebMvcTest(BookWebController.class)
2   @Import(SecurityConfig.class)
3 class BookWebControllerHtmlUnitTest {
4     @Autowired
5     private WebClient webClient;
6     @MockBean
7     LibraryService service;
8     @Autowired
9     private WebApplicationContext webApplicationContext;
10    @BeforeEach
11    public void setup() {
12      MockMvcBuilders
13          .webAppContextSetup(webApplicationContext)
14          .apply(springSecurity())
15          .build();
16    }
17    @Test
18    @WithAnonymousUser
19    void testTableBooks() throws Exception{
20      AuthorSlimDto author = new AuthorSlimDto(1L, "FirstName", "LastName");
21      when(service.selAllBooks()).thenReturn(List.of(
22          new BookDto("978-88-111-1111-1", "titleA", "Genre", "Plot",
23          LocalDate.parse("2000-01-01"), 100, Set.of(author)),
24          new BookDto("978-88-111-1111-2", "titleB", "Genre", "Plot",
25          LocalDate.parse("2000-01-01"), 100, Set.of(author))
26      ));
27      HtmlPage page = webClient.getPage("/library");

```

```

    HtmlTable table = (HtmlTable) page.getElementById("booksTable");
    assertEquals(table.asNormalizedText(),
28      "Isbn Title Genre Authors \n"
      + "978-88-111-1111-1 titleA Genre FirstName LastName \n"
      + "978-88-111-1111-2 titleB Genre FirstName LastName ");
    }
    @Test
33  @WithUserDetails("admin")
    void testinsertNewBook_validInput() throws Exception{
        AuthorSlimDto author = new AuthorSlimDto(1L, "FirstName", "LastName");
        BookDto book = new BookDto("978-88-111-1111-1", "Title", "Genre", "Plot",
        "LocalDate.parse(\"2000-01-01"), 100, Set.of(author));

38  HtmlPage page = webClient.getPage("/library/insert");
    HtmlForm form = page.getFormByName("bookForm");
    HtmlTextInput isbn = page.getElementByName("isbn");
    isbn.setValue("978-88-111-1111-1");
    HtmlTextInput title = page.getElementByName("title");
43  title.setValue("Title");
    HtmlTextInput genre = page.getElementByName("genre");
    genre.setValue("Genre");
    HtmlTextInput publicationDate = page.getElementByName("publicationDate");
    );
    publicationDate.setValue("2000-01-01");
48  HtmlNumberInput numberPages = page.getElementByName("numberPages");
    numberPages.setValue("100");
    HtmlTextArea plot = page.getElementByName("plot");
    plot.setText("Plot");
    HtmlTextInput authorNamesInput = page.getElementByName("authorNames");
53  authorNamesInput.setValueAttribute("Firstname Lastname");
    HtmlInput submit = form.getInputByName("submitButton");
    HtmlPage resultPage = submit.click();
    assertEquals("/library", resultPage.getUrl().getPath());
    verify(service).insertNewBook(book);
58  }
    @Test
    @WithUserDetails("admin")
    void testinsertNewBook_invalidInput() throws Exception{
        HtmlPage page = webClient.getPage("/library/insert");
63  final HtmlForm form = page.getFormByName("bookForm");
        HtmlTextInput isbn = page.getElementByName("isbn");
        isbn.setValue("978-88-111-1111-1");
        HtmlInput submit = form.getInputByName("submitButton");
        HtmlPage resultPage = submit.click();
68  assertEquals("/library/insert", resultPage.getUrl().getPath());
    }
}

```

7.4.4 Test di integrazione

Con i test di integrazione (*Integration Test*) si verifica l'integrazione tra uno o più moduli. Precedentemente, sono stati testati i livelli in maniera separata tramite l'uso delle annotazioni e dei metodi forniti da *Mockito*. Nei test seguenti non vengono effettuate delle simulazioni, ma delle operazioni reali.

Per ottenere dei test realistici, si dovrebbe utilizzare un ambiente più vicino possibile a quello della nostra applicazione. Dunque, sarebbe più opportuno testare l'applicazione in un ambiente *MySQL* e non con *H2*. Con *Spring*, non è necessario creare un altro contenitore *Docker* per un database di test *MySQL*, è possibile usare *TestContainer*, una libreria *open source* compatibile con *JUnit* (4 o 5) che ha lo scopo di gestire ed integrare i container *Docker* nei test.

Dopo aver scaricato le dipendenze necessarie per *TestContainer*, per utilizzarlo in una classe di test è necessaria l'annotazione `@Testcontainers`, che attiva l'estensione di *JUnit* che consente di avviare i contenitori. Successivamente, è necessario dichiarare un contenitore che può essere statico, ovvero condiviso dai test della classe e, dunque, avviato prima dell'esecuzione del primo test e distrutto alla fine dell'ultimo, oppure non statico, creato e distrutto per ogni test.

Con la versione *Spring Boot 3.1*, è stato aggiunto un supporto a *TestContainer*, tramite l'annotazione `@ServiceConnection`, utilizzata sui campi dell'istanza del contenitore, per configurare automaticamente la connessione con esso [32].

I test di integrazione, sono implementati nel package `con.library.webapp`. *IntegrationTest* nella cartella `src/test/java`.

Nelle sezioni successive sono presenti degli esempi di test di integrazione. Si omette la spiegazione dei test, poiché molto simili a quelli implementati in isolamento. I test di integrazione non aumentano il coverage, ma hanno lo scopo di testare l'integrazione dei vari componenti.

7.4.5 Integrare lo strato di persistenza e di servizio

Nella classe *LibraryServiceRepositoryIT*, si verifica la collaborazione tra il livello di persistenza e di servizio. Infatti, non vengono utilizzate le annotazioni di *Mockito*, ma vengono iniettate le istanze di *LibraryService*, *BookRepository* ed *AuthorRepository*. Nel codice seguente, sono riportati alcuni test a titolo di esempio:

```
@Testcontainers
@SpringBootTest
class LibraryServiceRepositoryIT {
    @Autowired
    private LibraryService service;
    @Autowired
    private MapStructMapper mapper;
    @Autowired
    private BookRepository bookRepository;
    @Autowired
    private AuthorRepository authorRepository;
    @Container
    @ServiceConnection
    private static final MySQLContainer<?> mySQLContainer = new
        MySQLContainer<>("mysql:8.0")
            .withDatabaseName("test")
            .withUsername("user")
            .withPassword("pass");

    @BeforeEach
    void init() {
        bookRepository.deleteAll();
        bookRepository.flush();
        authorRepository.deleteAll();
        authorRepository.flush();
    }
    @Test
    void testServiceSelAllBooksByRepository() {
        Book book = new Book("978-88-111-1111-1", "Title", "Crime", "Plot",
            LocalDate.of(2000, 01, 01), 100, null);
        Author author = new Author(1L, "FirstName", "LastName", Set.of(book));
        book.setAuthors(Set.of(author));
        bookRepository.save(book);
        List<Book> allRepository = bookRepository.findAll();
        List<Book> allService = service.selAllBooks()
            .stream()
            .map(mapper :: bookDtoToBook)
            .collect(Collectors.toList());
        assertEquals(allRepository, allService);
    }
    @Test
    void testServiceCanInsertBookToRepository() {
        Book saved = service.insertNewBook(
            new BookDto("978-88-111-1111-1", "Title", "Crime", "Plot",
                LocalDate.of(2000, 01, 01), 100, Set.of(new AuthorSlimDto(null, "FirstName", "LastName"))));
    }
}
```

```

        assertTrue(bookRepository.findById(saved.getIsbn()).isPresent());
45    }
    @Test
    void saveAuthorByService() {
        Author author = new Author(null, "FirstName", "LastName", null);
        Book book = new Book("978-88-111-1111-1", "Title", "Crime", "Plot",
        LocalDate.of(2000, 01, 01), 100, Set.of(author));
50    author.setBooks(Set.of(book));
        authorRepository.save(author);
        AuthorSlimDto update = new AuthorSlimDto(author.getId(), "NewName", "LastName");
        service.saveAuthor(update);
        assertThat(authorRepository.findById(author.getId())).contains(mapper.
        authorSlimDtoToAuthor(update));
55    }
}

```

Nella classe, viene configurato un database *MySQL* in un container *Docker* statico. Con l'annotazione `@SpringBootTest` viene avviato l'intero contesto dell'applicazione. Dato che tutti i test dovrebbero iniziare con stato noto (ad esempio con il database vuoto), nel metodo `init()`, annotato con `@BeforeEach` ed eseguito prima di ogni test della classe, vengono eliminati tutti gli elementi dal catalogo.

7.4.6 Integrare le componenti REST

Per testare i controller `BookRestController` ed `AuthorRestController` si è utilizzato il *RestAssured*, framework utile per testare i servizi REST in Java. In entrambi le classi, viene utilizzata l'annotazione `@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)` per avviare il server con una porta casuale e l'annotazione `@LocalServerPort` per ottenere il numero della porta utilizzata, necessaria per configurare *RestAssured*. Viene utilizzato il metodo `init()` per eliminare tutti i dati nel catalogo prima di ogni test.

Di seguito, sono riportati alcuni test a titolo di esempio della classe `BookRestControllerIT`:

```

@Testcontainers
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class BookRestControllerIT {
4   @LocalServerPort
   private int port;
   @Autowired
   private BookRepository bookRepository;
}

```

```

    @Autowired
9   private AuthorRepository authorRepository;
    @Container
    @ServiceConnection
    private static final MySQLContainer<?> mySQLContainer = new
        MySQLContainer<>("mysql:8.0")
            .withDatabaseName("test")
14          .withUsername("user")
            .withPassword("pass");

    @BeforeEach
    void init() {
19      RestAssured.port = port;
      bookRepository.deleteAll();
      bookRepository.flush();
      authorRepository.deleteAll();
      authorRepository.flush();
24    }
    @Test
    void testGetAllBooks() {
      Book book = new Book("978-88-111-1111-1", "Title", "Genre", "Plot",
      LocalDate.of(2000, 01, 01), 100, null);
      Author author = new Author(1L, "FirstName", "LastName", Set.of(book));
29      book.setAuthors(Set.of(author));
      bookRepository.save(book);

      given().get("/api/book")
          .then()
34          .statusCode(200)
          .body("$.size()", equalTo(1))
          .body("isbn", hasItems("978-88-111-1111-1"));
    }
    @Test
39    void testGetAllBooks_NotFound() throws Exception {
      given().get("/api/book")
          .then().statusCode(404)
          .body("message", equalTo("There is no book!"));
    }
44    @Test
    void testFindBooksByTitle() {
      Book bookA = new Book("978-88-111-1111-1", "TitleA", "Genre", "Plot",
      LocalDate.of(2000, 02, 01), 100, null);
      Book bookB = new Book("978-88-111-1111-2", "TitleB", "Genre", "Plot",
      LocalDate.of(1999, 01, 01), 100, null);
      Author authorA = new Author(null, "FirstNameA", "LastNameA", Set.of(
      bookA));
49      Author authorB = new Author(null, "FirstNameB", "LastNameB", Set.of(
      bookB));
    }

```

```

bookA.setAuthors(Set.of(authorA));
bookB.setAuthors(Set.of(authorB));
bookRepository.saveAllAndFlush(List.of(bookA, bookB));

54 given().get("/api/book/title/TitleA")
      .then().statusCode(200)
      .body("$.size()", equalTo(1))
      .body("isbn", hasItems("978-88-111-1111-1"));
}
59 }

```

7.4.7 Integrare le componenti Controller

Per i test di integrazione dei controller web, sono state scaricate ed installate le dipendenze di *Selenium* e *HtmlUnitDrive*, un set di strumenti per l'automazione del browser Web utili per controllare le istanze del browser ed emulare l'interazione dell'utente con il browser [33]. Di seguito, sono riportati alcuni test a titolo di esempio:

```

1 @Testcontainers
  @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
  class BookWebControllerIT {
    @LocalServerPort
    private int port;
6   private String url;
    private WebDriver driver;
    @Autowired
    BookRepository bookRepository;
    @Autowired
11   AuthorRepository authorRepository;
    @Container
    @ServiceConnection
    private static final MySQLContainer<?> mySQLContainer = new
      MySQLContainer<>("mysql:8.0")
        .withDatabaseName("test")
16        .withUsername("user")
        .withPassword("pass");

    @BeforeEach
    void init() {
21      url = "http://localhost:" + port + "/library";
      driver = new HtmlUnitDriver();
      bookRepository.deleteAll();
      bookRepository.flush();
      authorRepository.deleteAll();

```

```

26     authorRepository.flush();
}
@Test
void testTableInBooksPage() {
    Book book = new Book("978-88-111-1111-1", "Title", "Genre", "Plot",
    LocalDate.of(2000, 01, 01), 100, null);
31    Author author = new Author(null, "FirstName", "LastName", Set.of(book))
    ;
    book.setAuthors(Set.of(author));
    bookRepository.save(book);
    driver.get(url);
    assertThat(driver.findElement(By.id("booksTable")).getText())
36        .contains("Isbn Title Genre Authors \n"
            + "978-88-111-1111-1 Title Genre FirstName LastName  ");
    driver.quit();
}
@Test
void testTableEmptyBooksPage_Admin() {
    driver.get(url);
    driver.findElement(By.id("buttonLogin")).click();
    driver.findElement(By.id("userId")).sendKeys("admin");
    driver.findElement(By.id("password")).sendKeys("adminPass");
46    driver.findElement(By.id("submitlogin")).click();
    driver.findElement(By.id("buttonBooks")).click();
    assertThat(driver.findElement(By.id("errMessage")).getText())
        .contains("There is no book!");
    assertThat(driver.getPageSource()).doesNotContain("Login");
51    assertEquals(driver.findElement(By.cssSelector("button")).getText(), "Insert your first book!");
    driver.quit();
}
@Test
void testInsert_Table() {
56    Book book = new Book("978-88-111-1111-1", "Title", "Genre", "Plot",
        LocalDate.of(2000, 01, 01), 100, null);
    Author author = new Author(null, "FirstName", "LastName", Set.of(book))
    ;
    book.setAuthors(Set.of(author));
    bookRepository.save(book);
61    driver.get(url);
    driver.findElement(By.id("buttonLogin")).click();
    driver.findElement(By.id("userId")).sendKeys("admin");
    driver.findElement(By.id("password")).sendKeys("adminPass");
    driver.findElement(By.id("submitlogin")).click();
66    driver.findElement(By.id("buttonBooks")).click();
    driver.findElement(By.cssSelector("[href='/library/insert']")).click();
    assertEquals(driver.getTitle(), "New Book");
    driver.findElement(By.id("isbn")).sendKeys("978-88-111-1111-2");
}

```

```

    driver.findElement(By.id("title")).sendKeys("Title 2");
71   driver.findElement(By.id("genre")).sendKeys("Genre");
    driver.findElement(By.id("publicationDate")).sendKeys("2001-01-01");
    driver.findElement(By.id("numberPages")).sendKeys("150");
    driver.findElement(By.id("plot")).sendKeys("Plot");
    driver.findElement(By.name("authorNames")).sendKeys("FirstnameB
    LastnameB");
76   driver.findElement(By.name("submitButton")).click();
    assertEquals(driver.findElement(By.id("booksTable")).getText(),
    "Isbn Title Genre Authors New Book\n"
    + "978-88-111-1111-1 Title Genre FirstName LastName \n"
    + "978-88-111-1111-2 Title 2 Genre FirstnameB LastnameB ");
81   driver.quit();
}
@Test
void testInsertBook_InvalidInput() {
    driver.get(url + "/insert");
86   driver.findElement(By.id("userId")).sendKeys("admin");
    driver.findElement(By.id("password")).sendKeys("adminPass");
    driver.findElement(By.id("submitlogin")).click();
    assertEquals(driver.getTitle(), "New Book");
    driver.findElement(By.id("isbn")).sendKeys("");
91   driver.findElement(By.id("title")).sendKeys("");
    driver.findElement(By.id("genre")).sendKeys("");
    driver.findElement(By.id("publicationDate")).sendKeys("");
    driver.findElement(By.id("numberPages")).sendKeys("");
    driver.findElement(By.id("plot")).sendKeys("");
96   driver.findElement(By.name("authorNames")).sendKeys("FirstnameB
    LastnameB");
    driver.findElement(By.name("submitButton")).click();
    assertEquals(driver.getTitle(), "New Book");
    driver.quit();
}
101 }

```

7.5 COVERAGE

Nella Figura 23 è presente il *coverage*, percentuale delle linee di codice testate dell'applicazione sviluppata. La copertura delle classi *entity* e DTO non risulta al 100% perché presenti metodi, come ad esempio i *getter*, che non contengono logica complessa da testare.

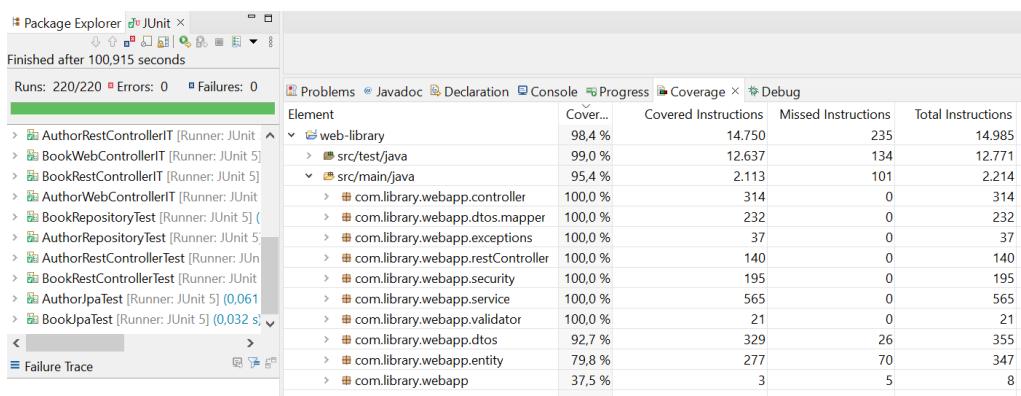


Figura 23: Coverage

8

RISULTATI OTTENUTI

Nella sezione corrente, sono riportati degli esempi di risultati ottenuti dallo sviluppo dell'applicazione.

Al primo avvio, è possibile osservare, all'interno della console di *Spring*, le istruzioni *SQL* generate da *Hibernate* per la creazione delle tabelle.

```
Hibernate: create table author (id bigint not null auto_increment, first_name varchar(255) not null, last_name varchar(255) not null, primary key (id)) engine=InnoDB
Hibernate: create table author_book (book_isbn varchar(255) not null, author_id bigint not null, primary key (book_isbn, author_id)) engine=InnoDB
Hibernate: create table book (isbn varchar(255) not null, genre varchar(255), number_pages int default 0, plot varchar(2000), publication_date date, title varchar(255)
Hibernate: alter table author_book add constraint FKg7j6ud9d32l1z32o9mg98s57 foreign key (author_id) references author (id)
Hibernate: alter table author_book add constraint FKamxyxkwdxsjvm7b7q18qgne5 foreign key (book_isbn) references book (isbn)
```

Figura 24: Generazione delle tabelle nel database

In Figura 25 è presente uno screenshot della home page, in cui è possibile eseguire l'accesso tramite il bottone di *Login* oppure è possibile consultare il catalogo come utente anonimo, vedi Figura 26.



Figura 25: Home page

The screenshot shows a web-based library catalog interface. At the top, there are navigation links for 'Library', 'Home', 'List of Authors', and a 'Login' button. Below this, the title 'List of all books:' is displayed in bold. A search bar with a 'Search' button is located above a table. The table has columns for 'Genre' (sorted by 'After Publication Date'), 'ISBN', 'Title', 'Genre', and 'Authors'. Each book entry includes a small blue circular icon with a white symbol. The table lists 12 books, such as 'La serie infernale' by Agatha Christie and 'Il guardiano degli innocenti. The Witcher. Vol. 1' by Andrzej Sapkowski.

Genre	After Publication Date	ISBN	Title	Genre	Authors
		978-88-04-50998-1	La serie infernale	Crime	Agatha Christie
		978-88-04-51924-9	Il terrore viene per posta	Crime	Agatha Christie
		978-88-04-58327-1	Una giornata nell'antica Roma. Vita quotidiana, segreti e curiosità	History	Alberto Angela
		978-88-04-77198-2	Luce e tenebra. Dal mondo di Percy Jackson	Children	Mark Oshiro, Rick Riordan
		978-88-06-17324-1	Un infinito numero	Classic	Sebastiano Vassalli
		978-88-07-92420-0	Le più belle storie per diventare amici	Children	Susanna Mattiengeli, Giulia Tudori, Daniele Movere
		978-88-23-50956-6	Il pensatore solitario	Comic	Ermanno Cavazzoni
		978-88-459-2998-3	Follia	Novel	Patrick McGrath
		978-88-50-26672-2	Il guardiano degli innocenti. The Witcher. Vol. 1	Fantasy	Andrzej Sapkowski
		978-88-6160-340-0	La bambina che amava Tom Gordon	Horror	Stephen King
		978-88-6836-119-8	L'uomo in fuga	Novel	Stephen King

Figura 26: Consultazione del catalogo senza autenticazione

Successivamente, è presente un esempio di inserimento nel catalogo.

The screenshot shows a 'New Book' form. At the top, there are navigation links for 'Library', 'Home', 'List of Books', 'List of Authors', and a 'Logout' button. The main section is titled 'New Book'. It contains fields for 'ISBN' (978-88-04-66306-5), 'Title' (Il signore delle mosche), 'Genre' (Novel), 'Publication Date (YYYY-MM-DD)' (2016-05-31), and 'Number of Pages' (281). A large text area for 'Plot' contains the following text: 'Nel corso di un conflitto planetario, un aereo precipita su un'isola deserta. Sopravvivono solo alcuni ragazzi, che provano a riorganizzarsi senza l'aiuto e il controllo degli adulti. I primi tentativi di dare vita a una società ordinata hanno successo, ma presto esplodono tensioni latenti ed emergono paure irrazionali e comportamenti asociali: lo scenario paradisoico dell'isola tropicale si trasforma in un inferno. "Il Signore delle Mosche" è un romanzo a tesi in cui Golding si serve delle forme dell'utopia negativa ("distopia") per mettere a nudo gli aspetti più selvaggi e repressi della natura umana ed esporre la sua concezione del mondo e dell'uomo.' Below the plot, there is a note about separating authors with commas. At the bottom, there are 'Submit' and 'Reset' buttons.

Figura 27: Esempio di inserimento nel catalogo

Dopo aver aggiunto i dati del libro, se corretti, l'utente viene reindirizzato alla lista dei libri. Si noti che l'utente è autenticato, quindi sono presenti i tasti di *logout* e per la modifica del catalogo.

Genre	After Publication Date		Search	search
ISBN	Title	Genre	Authors	
978-88-04-50998-1	La serie infernale	Crime	Agatha Christie	
978-88-04-51924-9	Il terrore viene per posta	Crime	Agatha Christie	
978-88-04-58327-1	Una giornata nell'antica Roma. Vita quotidiana, segreti e curiosità	History	Alberto Angela	
978-88-04-66306-5	Il signore delle mosche	Novel	William Golding	
978-88-04-77198-2	Luce e tenebra. Dal mondo di Percy Jackson	Children	Mark Oshiro , Rick Riordan	
978-88-06-17324-1	Un infinito numero	Classic	Sebastiano Vassalli	
978-88-07-92420-0	Le più belle storie per diventare amici	Children	Susanna Mattiengeli , Giulia Tudori , Daniele Moverelli	
978-88-23-50956-6	Il pensatore solitario	Comic	Ermanno Cavazzoni	
978-88-459-2998-3	Follia	Novel	Patrick McGrath	
978-88-50-26672-2	Il guardiano degli innocenti. The Witcher. Vol. 1	Fantasy	Andrzej Sapkowski	
978-88-6160-340-0	La bambina che amava Tom Gordon	Horror	Stephen King	
978-88-6836-119-8	L'uomo in fuga	Novel	Stephen King	

Figura 28: Consultazione del catalogo con autenticazione

Nella console di *Spring* è possibile osservare le istruzioni eseguite per l'inserimento.

```
Hibernate: select b1_0.isbn,b1_0.genre,b1_0.number_pages,b1_0.plot,b1_0.publication_date,b1_0.title from book b1_0 where b1_0.isbn=?
Hibernate: SELECT * FROM author a WHERE (first_name = ? AND last_name = ?)
Hibernate: select b1_0.isbn,a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name,b1_0.genre,b1_0.number_pages,b1_0.plot,b1_0.publication_date,b1_0.title from book b1_0 join author a1_0 on a1_0.id=a1_1.author_id
Hibernate: insert into book (genre,number_pages,plot,publication_date,title,isbn) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into author (first_name,last_name) values (?, ?)
Hibernate: insert into author_book (book_isbn,author_id) values (?, ?)
Hibernate: select b1_0.isbn,b1_0.genre,b1_0.number_pages,b1_0.plot,b1_0.publication_date,b1_0.title from book b1_0
Hibernate: select a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name from author_book a1_0 join author a1_1 on a1_1.id=a1_0.author_id where a1_0.book_isbn=?
Hibernate: select a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name from author_book a1_0 join author a1_1 on a1_1.id=a1_0.author_id where a1_0.book_isbn=?
Hibernate: select a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name from author_book a1_0 join author a1_1 on a1_1.id=a1_0.author_id where a1_0.book_isbn=?
Hibernate: select a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name from author_book a1_0 join author a1_1 on a1_1.id=a1_0.author_id where a1_0.book_isbn=?
Hibernate: select a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name from author_book a1_0 join author a1_1 on a1_1.id=a1_0.author_id where a1_0.book_isbn=?
Hibernate: select a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name from author_book a1_0 join author a1_1 on a1_1.id=a1_0.author_id where a1_0.book_isbn=?
Hibernate: select a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name from author_book a1_0 join author a1_1 on a1_1.id=a1_0.author_id where a1_0.book_isbn=?
Hibernate: select a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name from author_book a1_0 join author a1_1 on a1_1.id=a1_0.author_id where a1_0.book_isbn=?
Hibernate: select a1_0.book_isbn,a1_1.id,a1_1.first_name,a1_1.last_name from author_book a1_0 join author a1_1 on a1_1.id=a1_0.author_id where a1_0.book_isbn=?
```

Figura 29: Istruzioni per l'inserimento nella console

Successivamente, sono visibili i formati *Json* ottenuti tramite la richiesta */api/book/978-88-04-77198-2* e tramite la richiesta */api/author/name/stephen-king*.

96 RISULTATI OTTENUTI

```
1 {
2     "isbn": "978-88-04-77198-2",
3     "title": "Luce e tenebra. Dal mondo di Percy Jackson",
4     "genre": "Children",
5     "plot": "Per Nico Di Angelo, figlio di Ade, segnato da troppi traumi e  
dolorose perdite, non esiste tregua. C'è solo un raggio di sole  
nella sua vita, letteralmente: il suo ragazzo, Will Solace, il  
figlio di Apollo. E ora perfino i suoi sogni sono tormentati: una  
voce invoca il suo aiuto dal Tartaro, e Nico crede di sapere chi  
sia... il titano Bob, un amico che si è sacrificato per chiudere le  
Porte della Morte e che Percy e Annabeth hanno dovuto lasciare  
indietro. Quando l'Oracolo predice l'impresa, Nico deve tornare nel  
più profondo degli Inferi, dove già una volta ha rischiato di  
perdere se stesso, con l'unico conforto di Will, che intende  
accompagnarla, costi quel che costi.",
6     "publicationDate": "2023-05-23",
7     "numberPages": 488,
8     "authors": [
9         {
10            "id": 6,
11            "firstName": "Mark",
12            "lastName": "Oshiro"
13        },
14        {
15            "id": 7,
16            "firstName": "Rick",
17            "lastName": "Riordan"
18        }
19    ]
20 }
```

```
1 {
2     "id": 1,
3     "firstName": "Stephen",
4     "lastName": "King",
5     "books": [
6         {
7             "isbn": "978-88-6836-119-8",
8             "title": "L'uomo in fuga"
9         },
10        {
11            "isbn": "978-88-6160-340-0",
12            "title": "La bambina che amava Tom Gordon"
13        }
14    ]
15 }
```

9

CONCLUSIONI

L'obiettivo che si pone questa tesi è studiare ed illustrare possibili tecnologie per lo sviluppo ed il testing di un'applicazione web, con particolare attenzione sui test, descrivendo come e quali vantaggi essi permettono di ottenere.

Nel capitolo corrente vengono analizzati i vantaggi ed i problemi riscontrati.

9.1 VANTAGGI E SVANTAGGI

Nelle sezioni precedenti, sono stati introdotti ed analizzati gli strumenti impiegati per lo sviluppo. Il loro utilizzo può comportare dei vantaggi ed eventuali svantaggi. Ad esempio, nel caso di *MySQL* si possono osservare i seguenti vantaggi:

- Portabilità: *MySQL* è un server di database multipiattaforma e, quindi, compatibile con diverse piattaforme come Linux, Windows, ecc.
- Sicurezza: è molto sicuro ed affidabile perché protetto da funzioni di sicurezza dei dati.
- Facilità d'uso: è semplice da usare ed apprendere.

Tuttavia, esso risulta essere inefficiente nella gestione di dati di grandi dimensioni ed ha alcuni problemi di stabilità. Durante lo sviluppo dell'applicazione, è stato adottato *MySQL* poiché i vantaggi offerti risultano superiori ai possibili svantaggi riscontrabili.

Anche l'uso di *Spring* può comportare benefici e complicazioni.
I vantaggi osservati sono:

- Flessibilità: supporta l'uso delle annotazioni basate su Java per la configurazione degli *Spring Beans*.

- Test semplificati: la funzionalità dell’iniezione *Spring Dependency* migliora la testabilità.
- Server integrato: *Spring* fornisce un contenitore leggero che può essere attivato senza l’uso di un server.
- Modularità: *Spring* permette agli sviluppatori di scegliere quali pacchetti o classi possono essere utilizzati.

Gli svantaggi osservati sono:

- Complessità: il framework non è di facile utilizzo e richiede esperienza od una fase di apprendimento.
- Molteplici funzionalità: la modularità di *Spring* può essere anche uno svantaggio perché gli sviluppatori devono conoscere le funzionalità più adatte alle loro necessità durante lo sviluppo. Scegliere delle funzionalità inadeguate può comportare complicazioni.

9.2 CONSIDERAZIONI E CONCLUSIONI

Durante lo sviluppo, sono emersi alcuni miglioramenti che è possibile attuare, quali ad esempio:

- non archiviare in memoria l’utente di test, ma fare in modo che le informazioni riguardanti tutti gli utenti siano archiviate nel database e recuperate da *Hibernate* e *JPA*.
- aggiungere nuove funzionalità, come, ad esempio, vedere la disponibilità di un libro.
- migliorare l’interfaccia grafica.
- implementare un servizio che utilizza i dati ottenuti in formato JSON per analizzarli e utilizzarli per creare delle viste.

Essendo un ambito in continua evoluzione, è necessario prestare costantemente attenzione alle novità introdotte. Esse, infatti, possono portare numerosi vantaggi nello sviluppo, ad esempio l’introduzione di `@ServiceConnection` di *TestContainer*, per configurare automaticamente un contenitore e, dunque, permettere di scrivere meno codice.

Il sorgente dell’applicazione sviluppata nella presente tesi è disponibile e consultabile nel repository *spring-boot-mylibrary* su GitHub [34].

BIBLIOGRAFIA

- [1] World wide web. The World Wide Web project. Available at: <http://info.cern.ch/hypertext/WWW/TheProject.html> (Cited on page 7.)
- [2] Docker Desktop overview (2021) Docker Documentation. Available at: <https://docs.docker.com/desktop/> (Cited on page 8.)
- [3] DBeaver Community | Free Universal Database Tool. Available at: <https://dbeaver.io/> (Cited on page 8.)
- [4] Spring Tools 4 is the next generation of Spring tooling. Available at: <https://spring.io/tools> (Cited on page 8.)
- [5] Postman Learning Center. Available at: <https://learning.postman.com/docs/introduction/overview/> (Cited on page 8.)
- [6] Frongillo, D. Introduzione a rest, ItalianCoders. Available at: <https://italiancoders.it/introduzione-a-rest/> (Cited on page 11.)
- [7] Spring.io, Panoramica di Spring Framework. Available at: <https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/overview.html> (Cited on page 14.)
- [8] Spring.io, Introduction to the Spring IoC container and beans. Available at: <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html> (Cited on page 15.)
- [9] Spring Boot Reference Documentation. Available at: <https://docs.spring.io/spring-boot/docs/current/reference/html/index.html> (Cited on page 15.)
- [10] Chehab, G. (2023, April 2). What is an orm? how does it work? how should we use one?. Baeldung on Computer Science. Available at: <https://www.baeldung.com/cs/object-relational-mapping> (Cited on page 16.)

- [11] Hibernate. Final User Guide. Available at:
https://docs.jboss.org/hibernate/orm/6.1/userguide/html_single/Hibernate_User_Guide.html (Cited on page 16.)
- [12] Oracle, Che Cos'è un database? Oracle Italia. Available at:
<https://www.oracle.com/it/database> (Cited on page 19.)
- [13] Docker (2023) Docker Overview, Docker Documentation. Available at: <https://docs.docker.com/get-started/overview/> (Cited on page 20.)
- [14] Docker (2023), Manage data in Docker. Available at:
<https://docs.docker.com/storage/> (Cited on page 21.)
- [15] Spring Validation in the Service Layer | Baeldung. Available at:
<https://www.baeldung.com/spring-service-layer-validation> (Cited on page 25.)
- [16] Balasubramaniam, W. by: V. (2023, June 3). Defining JPA entities. Baeldung. Available at: <https://www.baeldung.com/jpa-entities> (Cited on page 26.)
- [17] An Overview of Identifiers in Hibernate/JPA | Baeldung. Available at: <https://www.baeldung.com/hibernate-identifiers> (Cited on page 28.)
- [18] Hibernate reactive 2.0.4. Final reference documentation. Available at:
https://hibernate.org/reactive/documentation/2.0/reference/html_single/#_equals_and_hashcode (Cited on page 28.)
- [19] Many-to-many relationship in JPA. Baeldung. Available at:
<https://www.baeldung.com/jpa-many-to-many> (Cited on page 30.)
- [20] Peterlić, A. (2023, May 6). Remove entity with many-to-many relationship in JPA. Baeldung. Available at: <https://www.baeldung.com/jpa-remove-entity-many-to-many> (Cited on page 31.)
- [21] Rick-Anderson. Creare oggetti di Trasferimento Dati (DTO). Microsoft Learn. Available at:
<https://learn.microsoft.com/it-it/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5> (Cited on page 32.)

- [22] Oliver Gierke. Spring Data JPA. Available at:
<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories> (Cited on page 34.)
- [23] Vlad Mihalcea. (2022, July 1). Spring transaction best practices. Available at: <https://vladmihalcea.com/spring-transaction-best-practices/> (Cited on page 35.)
- [24] What is transaction management in Hibernate. JavaTute. (2019, February 12). Available at: <https://javatute.com/what-is-transaction-management-in-hibernate/> (Cited on page 41.)
- [25] Dutta, W. (2023, August 15). Quick Guide to spring controllers. Baeldung. Available at: <https://www.baeldung.com/spring-controllers> (Cited on page 47.)
- [26] Andrea Marzilli. Introduzione all'html. HTML.it. Available at:
<https://www.html.it/pag/16026/introduzione22/> (Cited on page 53.)
- [27] Thymeleaf. Available at: <https://www.thymeleaf.org/> (Cited on page 53.)
- [28] Il codice ISBN. Available at: <https://www.isbn.it/CODICEISBN.aspx> (Cited on page 57.)
- [29] Spring.io, Authorize HttpServletRequests Spring Security. Available at:
docs.spring.io/spring-security/reference/servlet/authorization/authorize-http-requests.html (Cited on page 63.)
- [30] Webb, P., Yudovin, A., Frederick, S. Annotation interface DataJpaTest. DataJpaTest (Spring Boot 3.1.3 API). Available at:
<https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/orm/jpa/DataJpaTest.html> (Cited on page 69.)
- [31] Introduction to HtmlUnit. Baeldung. Available at:
<https://www.baeldung.com/htmlunit> (Cited on page 81.)
- [32] Spring Blog. Improved Testcontainers Support in Spring Boot 3.1. Available at: <https://spring.io/blog/2023/06/23/improved-testcontainers-support-in-spring-boot-3-1> (Cited on page 85.)

- [33] A deeper look at selenium. Selenium. Available at:
<https://www.selenium.dev/documentation/overview/details/> (Cited on page 89.)
- [34] Marrami, V. Available at: <https://github.com/vanessamarrami/spring-boot-mylibrary> (Cited on page 98.)

10

RINGRAZIAMENTI

In questa sezione, vorrei dedicare qualche parola a tutte le persone che mi hanno aiutato, supportato e che mi sono state accanto durante il mio percorso universitario.

Ringrazio il professor Bettini, relatore della tesi, per avermi guidato, per i suoi consigli e per la sua disponibilità durante la stesura di questo lavoro.

Ringrazio la mia famiglia per il sostegno e aiuto che mi hanno dato e in particolare a mio fratello Dimitri, che ha condiviso con me, fin da quando ero molto piccola, la sua passione per l'informatica.

Ringrazio Alessandro, il mio ragazzo e collega, per questi bellissimi anni passati insieme e per essere sempre stato disponibile e accanto a me nei momenti più difficili.

Ai miei amici di Radi, alle serate, alle risate e al vostro supporto. Vi ringrazio per tutti i bei momenti passati insieme.

A Elena ed Elisa, le mie carissime amiche di sempre, per essere al mio fianco. Grazie per i momenti divertenti, di conforto e per la vostra sincerità.

Ringrazio tutte le mie ex coinquiline Giulia, Erica, Maria Chiara e coinquilino acquisito Federico, per aver reso la nostra convivenza veramente indimenticabile.

Ringrazio i miei colleghi dell'università, per tutte le collaborazioni, consigli, aiuti e per le risate di questi anni. Prometto di organizzare una serata tutti insieme.