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

Advanced Techniques and Tools for Software Development
9 CFU
-
Project Report

SPRING BOOKSHELF: A SIMPLE SPRING BOOT WEB APPLICATION TO MANAGE MY PERSONAL BOOKSHELF

FRANCESCO MUCCI
francesco.mucci@stud.unifi.it
francescomucci.github.io
6173140

Docente: Prof. *Lorenzo Bettini*

Anno Accademico 2020-2021
21 aprile 2022

Francesco Mucci: *Spring Bookshelf*, Versione 1.0.1, Corso di Laurea Magistrale in Informatica, ©, Università degli Studi di Firenze, Anno Accademico 2020-2021

INDICE

INTRODUZIONE

Il progetto qui presentato è stato realizzato come parte della prova d'esame dell'insegnamento **B027540 - Advanced Techniques and Toold for Software Development (9CFU)** del corso di *Laurea Magistrale in Informatica dell'Università degli Studi di Firenze*.

Il lavoro è stato svolto singolarmente dallo studente **Francesco Mucci**:

- matricola: 6173140;
- email: `francesco.mucci@stud.unifi.it`;
- sito web personale: `francescomucci.github.io`.

Tutto il codice sviluppato e questo report sono pubblicamente accessibili sulla seguente repository GitHub:

- `https://github.com/FrancescoMucci/spring-bookshelf`.

STRUTTURA DELLA RELAZIONE

La relazione sarà strutturata come segue:

- nel **Capitolo 1 (Presentazione del progetto)** andremo a illustrare le specifiche e l'architettura del progetto, gli strumenti, le tecniche e i framework usati e la strategia di branching seguita;
- nel **Capitolo 2 (Setup)** tratteremo gli elementi salienti relativi alla configurazione del processo di building e di integrazione continua, la definizione del domain model e delle interfacce comuni per i layer della nostra applicazione;
- nel **Capitolo 3 (Persistence layer)** ci soffermeremo sul testing e l'implementazione della *Repository*, la componente responsabile dell'interazione con il database;
- nel **Capitolo 4 (Service layer)** analizzeremo l'implementazione e il testing del *Service*, la componente che implementa la business logic dell'applicazione;
- nel **Capitolo 5 (Web layer)** andremo a illustrare l'implementazione e il testing delle componenti che fanno parte del *Presentation (o Web) layer*: i *Data Trasfer Objects*, il *Web Controller* e le *Web Views*; vedremo inoltre come configurare la sicurezza web della nostra applicazione;
- nel **Capitolo 6 (Integration e E2E tests)** discuteremo dei test d'integrazione e dei test E2E implementati, ponendo in particolar modo l'accento sull'uso del *Page Object Pattern* per l'interazione con la UI delle nostre pagine web;
- infine, nel **Capitolo 7 (Eseguire la web application)** vedremo come fare a eseguire il processo di building e a lanciare da command-line la web application sviluppata.

1

PRESENTAZIONE DEL PROGETTO

1.1 SPECIFICHE

In questo lavoro progettuale siamo andati ad implementare una semplice *CRUD web application*, **Spring Bookshelf**, per la gestione dei libri di una libreria personale; tale applicazione andrà ad esporre unicamente end-points web tramite cui:

- un qualsiasi utente può:
  - visualizzare tutti i libri presenti nel database;
  - ricercare un libro inserendo il corrispettivo ISBN-13[1];
  - ricercare tutti i libri contenenti una certa stringa nel titolo;
  - autenticarsi (con o senza "remember-me") come amministratore;
- unicamente l'amministratore può:
  - aggiungere nuovi libri al database;
  - modificare i dati di un libro già presente;
  - cancellare un libro dal database;
  - eseguire il logout.

1.2 ARCHITETTURA

Al fine di agevolare la separazione delle responsabilità e aumentare la modularità del progetto, abbiamo implementato la *Spring Bookshelf* web application seguendo il classico **modello architetturale a 3 layer**. Le principali componenti della nostra applicazione saranno la `BookRepository` nel *Persistence (o Data access) layer*, il `BookService` nel *Business logic (o Service) layer* e il `BookWebController` nel *Presentation (o Web) layer*.

1.3 STRUMENTI, FRAMEWORK E TECNICHE USATE

1.3.1 *Linguaggio di programmazione e database*

Il linguaggio di programmazione usato per implementare il progetto è **Java**: in particolare, è stata usata la versione LTS *Java 8*.

Come database abbiamo adottato **MongoDB**, sia per motivi di semplicità, sia perché usato nella prima parte del corso; in particolare, eseguiremo la sua versione 4.4.3 all'interno di un **Docker Container** al fine di garantire un certo livello di riproducibilità; tale DB sarà in ascolto sulla sua porta di default, la 27017.

1 Un ISBN-13 è un International Standard Book Number a 13 cifre: una stringa usata per identificare in maniera univoca una particolare edizione di un libro [1].

1.3.2 *Strumenti e framework per applicazioni web*

Al fine di rendere più semplice lo sviluppo e il testing della nostra applicazione web, ci siamo appoggiati al framework **Spring Boot (v2.4.5)**: in particolare, rispetto a quanto visto nel corso, siamo andati ad approfondire i seguenti meccanismi di *Spring*:

- autenticazione (in-memory) e protezione degli end-points sfruttando il framework *Spring Security*;
- validazione degli input forniti dall'utente[2] sfruttando il meccanismo di *Bean Validation*;
- *custom error-handling* per associare a eccezioni da noi definite delle risposte HTTP personalizzate.

Per realizzare le pagine HTML abbiamo sfruttato la template engine **Thymeleaf**; nello specifico, sono stati indagati i seguenti aspetti:

- l'integrazione di *Thymeleaf* con *Spring Security* con l'obiettivo di nascondere ad utenti anonimi bottoni, links e altro contenuto normalmente riservato ad utenti autenticati;
- l'uso dei *Thymeleaf fragments* al fine di definire "componenti" HTML riusabili per ridurre la duplicazione di codice e strutturare meglio le nostre pagine.

Per lo stile delle pagine ci siamo affidati quasi totalmente a **Bootstrap 4**.

1.3.3 *Testing*

Il testing framework usato è **JUnit 4** e, al fine di aumentare la leggibilità dei test, è stata usata **AssertJ** come *assertion library*.

Come richiesto, abbiamo implementato tutto il codice seguendo la metodologia di sviluppo del **TDD** stando attenti a rispettare la forma della **Piramide dei Test**; nello specifico sono stati scritti 342 test d'unità, 114 test d'integrazione e 22 test E2E. Abbiamo, inoltre, tenuto traccia del **Code Coverage** grazie a *JaCoCo* e sfruttato *PIT* per eseguire la procedura di **Mutation Testing** solo per il `BookService`.

Infine, oltre a quanto visto a lezione, è stata sperimentata la possibilità di testare le web views in maniera indipendente rispetto al web controller struttando le funzionalità di mocking di Spring; inoltre, abbiamo usato il design pattern **Page Object Model**, sia nei test d'integrazione che nei test E2E, per interagire con la UI delle varie pagine web.

1.3.4 *Build automation, VCS e CI*

Per automatizzare la gestione delle dipendenze e il processo di building del software abbiamo utilizzato **Maven**; inoltre, è stato sfruttato il progetto **Maven Wrapper** per andare a "fissare" la versione di Maven alla 3.8.3, andando, di conseguenza, a incrementare la riproducibilità delle nostre *Maven builds*.

Abbiamo usato **git** come *Version Control System* e **GitHub** come host per la nostra *git repository*.

Il *processo di integrazione continua* è stato implementato usando **GitHub Actions** come *CI Server*, **SonarCloud** per analizzare la qualità del nostro codice e **Coveralls** per tener traccia della storia del *Code Coverage*.

2 E.g. dovrebbero essere accettate come ISBN-13 solo stringhe che rispettano lo standard.

1.4 STRATEGIA DI BRANCHING

Anche se il progetto è stato sviluppato da un singolo studente, si è deciso di seguire una strategia per la gestione dei branch che, nel caso di un team di sviluppatori, avrebbe permesso lo sviluppo parallelo ed indipendente delle diverse componenti della *Spring Bookshelf* application; in particolare, è stato adottato il seguente approccio:

1. dopo aver creato il progetto, a partire dal ramo `master` praticamente vuoto, è stato creato un ramo `setup` nel quale abbiamo:
   a) creato lo *Spring Boot project*;
   b) configurato il processo di building;
   c) configurato il processo di integrazione continua;
   d) definito la *Domain Model Class*, `Book`;
   e) definito le interfacce comuni ai diversi layer, `BookRepository` e `BookService`;
2. dopo aver creato il progetto, a partire dal ramo `master` praticamente vuoto, è stato creato un ramo `setup` nel quale abbiamo creato lo *Spring Boot project*, configurato il processo di building e di integrazione continua, definito la *Domain Model Class* `Book` e le interfacce, `BookRepository` e `BookService`, comuni ai diversi layer;
3. una volta concluso il lavoro di "setup", è stata creata una *Pull Request* e le modifiche introdotte in questo ramo sono state unite al ramo `master`;
4. a partire da `master`, sono stati creati i rami `repository` e `service` per lo sviluppo ed il testing rispettivamente della `BookRepository` e del `BookService`;
5. inoltre, sempre a partire dal `master` è stato creato anche il ramo `web-base` nel quale vengono definiti gli elementi comuni del *Web Layer* su cui è necessario concordare prima di andare a implementarne le componenti in rami separati; per la precisione, abbiamo:
   a) definito i DTO, `BookData` e `IsbnData`;
   b) configurato e testato il meccanismo *Bean Validation* per i DTO;
   c) definito l'interfaccia del web-controller, `BookWebController`;
   d) configurato la web security;
6. dunque, a partire dal ramo `web-base`, sono stati creati i rami `web-controller` e `web-views` per lo sviluppo ed il testing rispettivamente del `BookWebControler` e delle HTML web views;
7. dopo aver implementato e testato ogni componente nel proprio ramo, a partire dal `repository`, è stato creato il ramo `integration`; in questo siamo andati a fondere, uno alla volta, i rami delle diverse componenti e, dopo ogni fusione, abbiamo:
   a) aggiunto i test d'integrazione per la data componente;
   b) verificato che tutti i test venissero superati;
   c) creato *Pull Request* al fine di richiedere l'unione dell'`integration` branch nel `master`;
8. una volta integrate tutte le componenti, sempre nel ramo `integration`, sono stati aggiunti i test E2E e, dopo aver verificato che i test fossero tutti positivi, è stata creata un'ulteriore *Pull Request* per fondere il tutto nel `master`;
9. infine, è stato creato un ramo `update` per operazioni di refactoring e modifica del progetto.

2

SETUP

In questo capitolo illustreremo il lavoro svolto nel ramo `setup`, nel quale andremo a definire elementi comuni ai diversi layer su cui è necessario concordare prima di potersi dedicare allo sviluppo delle singole componenti. Nello specifico, definiremo la *Domain Model Class* e le interfacce `BookRepository` e `BookService`[1].

Risulta interessante sottolineare che si è deciso di rimandare la definizione dell'interfaccia del *Web Controller* dal momento che non è necessaria per gli altri layer.

In questo ramo ci siamo anche dedicati alla configurazione del processo di building del software e del processo di integrazione continua e alla definizione di una classe di costanti per il testing.

2.1 BOOK DOMAIN MODEL CLASS

La Spring Bookshelf application deve permettere di gestire i libri di una libreria; risulta, dunque, naturale rappresentare i nostri *Domain Model Objects* attraverso una classe `Book` con i seguenti campi:

- `Long isbn`;
- `String title`;
- `List<String> authors`.

2.1.1 *Campo ISBN*

Un codice ISBN o, per esteso, **International Standard Book Number** è una stringa usata per identificare in maniera univoca una specifica edizione di un libro; tali codici sono emessi dalla *International ISBN Agency* e contengono 10 oppure 13 cifre (parleremo, rispettivamente, di *ISBN-10* e *ISBN-13*), opzionalmente suddivise da spazi o trattini [1].

Avremmo potuto optare per un campo di tipo `String`, tuttavia volevamo imporre una certa consistenza nel formato dell'ISBN: usando il tipo `Long` stiamo implicitamente imponendo che nel database non possano essere salvati libri con un ISBN contenente spazi o trattini.

2.1.2 *Non manca un campo id?*

In generale, ci aspetteremmo anche un campo `id` di tipo `Long` che, nel momento in cui ci occuperemo del *Persistence Layer*, dovrebbe essere usato come chiave primaria per il

1 Tali interfacce sono poi state aggiornate aggiungendo, ad esempio, le annotazioni di *Spring* nel momento in cui siamo andati a lavorare sui singoli rami.

corrispettivo documento nel database; tuttavia, abbiamo deciso di usare l'ISBN stesso come identificativo.
Essendo buona norma delegare la generazione dell'id direttamente al database [2], la scelta fatta risulta inusuale; nonostante ciò, quest'approccio ci permetterà di affrontare qualche sfida implementativa in più: al momento dell'aggiunta di un nuovo libro al DB, sarà compito della nostra applicazione verificare che l'ISBN fornito dall'utente non sia già presente.
Proseguendo per questa strada non sarà, ovviamente, possibile aggiungere più ristampe della stessa edizione; per quanto ciò possa sembrare limitante, bisgna tenere a mente che l'applicazione *Spring Bookshelf* nasce per gestire una libreria domestica dove difficilmente saranno presenti più ristampe della stessa edizione.

2.1.3 *Alternative per il campo authors*

Per il campo `authors`, al posto di una `List<String>`, avremmo potuto optare per una `List<Authors>`; tuttavia, avendo deciso di appoggiarci a un database non relazionale, si è ritenuto opportuno evitare l'inserimento di una relazione di tipo *many-to-many* e mantenere le cose più semplici.

2.2 INTERFACCE CONCORDATE

2.2.1 *Interfaccia BookRepository*

L'interfaccia della *Repository* ci consentirà di astrarre dai dettagli del database usato e di gestire i nosti DMO come se fossero oggetti di una collezione.

Come vediamo nel codice 2.1, la `BookRepository` richiederà l'implementazione dei metodi per:

- recuperare tutti i libri salvati sul database;
- trovare un libro a partire dal suo ISBN;
- trovare tutti i libri con un certo titolo;
- salvare un libro (nuovo o editato) nel database;
- cancellare un libro dal database.

```java
public interface BookRepository {
  public List<Book> findAll(Sort sort);
  public Optional<Book> findById(long isbn);
  public List<Book> findAllByTitleLikeOrderByTitle(String title);
  public Book save(Book book);
  public void deleteById(long isbn);
}
```

Codice 2.1: Interfaccia BookRepository

Le firme dei metodi rispecchiano le convenzioni del progetto **Spring Data**: come vedremo nel capitolo 3, non saremo noi a fornire l'implementazione di questa interfaccia, ma lasceremo tale resposabilità a *Spring Boot* [3].

Nel ramo `update` abbiamo aggiornato l'interfaccia `BookRepository` aggiungendo il metodo `public boolean existsById(long isbn);` anche la sua implementazione sarà

fornita da *Spring Boot* e ci permetterà di verificare se un libro con il dato ISBN è presente o meno nel database.

2.2.2 *Interfaccia BookService*

Il *Service* andrà ad esporre i metodi per le varie operazioni CRUD andando, di conseguenza, a nascondere i dettagli della *Repository*; possiamo osservare la definizione dell'interfaccia `BookService` nel codice 2.2.

```
1 public interface BookService {
2   public List<Book> getAllBooks();
3   public Book getBookByIsbn(long isbn);
4   public List<Book> getBooksByTitle(String title);
5   public Book addNewBook(Book newBook);
6   public Book replaceBook(Book editedBook);
7   public void delateBookByIsbn(long isbn);
8 }
```

Codice 2.2: Interfaccia BookService

Sarà compito del *Service*, a seconda del dato metodo, verificare la presenza del particolare libro nel database e decidere se lanciare o meno una `BookNotFoundException` o una `BookAlreadyExistException`.
Entrambe queste eccezioni, che andremo a definire nel ramo `service`, estenderanno la `IllegalArgumentException` e, dunque, saranno delle *unchecked exceptions*: di conseguenza, non dovremo dichiarane il lancio nella firma dei metodi [4]. Sarà, tuttavia, necessario concordare in anticipo quali meteodi potrebbero sollevarle; in particolare:

- la `BookNotFoundException` potrà essere lanciata da:
  - `getBookByIsbn`;
  - `getBooksByTitle`;
  - `replaceBook`;
  - `delateBookByIsbn`;
- mentre la `BookAlreadyExistException` potrà essere sollevata unicamente da:
  - `addNewBook`.

2.3 CONFIGURARE IL PROCESSO DI BUILDING

Il processo di building è stato configurato in modo tale che, se non attiviamo alcun profilo e richiediamo che la build proceda almeno fino alla fase `test`, vengano eseguiti unicamente i test d'unità e non venga tenuto traccia del *Code Coverage*.

I *Profili Maven* definiti sono `jacoco`, `it-tests` e `e2e-tests`.

2.3.1 *Profilo jacoco*

Il profilo `jacoco` ci permette di abilitare su richiesta il `jacoco-maven-plugin`; questo è stato configurato nella sezione `<pluginManagement>` in modo tale che:

1. durante l'esecuzione dei test d'unità, venga tenuta traccia del *Code Coverage*;

2. durante la fase `verify` del *default maven-lifecycle*, vengano generati i report relativi al coverage e venga verificato che la copertura sia del 100%[2].

2.3.2 *Profilo it-tests*

Attiveremo il profilo `it-test` quando sarà nostro interesse eseguire unicamente i test d'integrazione; in particolare, farà sì che:

1. durante la fase `test`, venga saltata l'esecuzione dei test d'unità;
2. durante la fase `pre-integration-test`, venga lanciato un *Docker Container* su cui sarà eseguito un MongoDB in ascolto sulla porta 27017[3];
3. durante la fase `integration-test`, vengano eseguiti unicamente i test d'integrazione;
4. durante la fase `post-integration-test`, venga stoppato e rimosso il *Docker Container*;
5. durante la fase `verify`, venga verificato l'esito dei test d'integrazione.

Risulta importante evidenziare che il `docker-maven-plugin` non è stato configurato all'interno del profilo `it-tests`, ma nella sezione `<pluginManagement>`: in tal modo, non sarà comunque abilitato di default e non dovremo ripetere la sua configurazione nel profilo `e2e-tests`.

2.3.3 *Profilo e2e-tests*

Attiveremo il profilo `e2e-test` quando sarà nostro interesse eseguire unicamente i test E2E; in particolare, farà sì che:

1. durante la fase `process-resources`, venga riservato un numero di porta libero per la nostra applicazione *Spring Boot*;
2. durante la fase `test`, vengano saltati i test d'unità;
3. prima dell'esecuzione dei test E2E, venga eseguita l'applicazione *Spring Boot*;
4. durante la fase `pre-integration-test`, venga lanciato il *Docker Container* con il MongoDB in esecuzione sulla porta 27017;
5. durante la fase `integration-test`, vengano eseguiti solamente i test end-to-end[4];
6. dopo l'esecuzione dei test E2E: venga fermata l'applicazione *Spring Boot*;
7. durante la fase `post-integration-test`, venga stoppato e rimosso il *Docker Container*;
8. durante la fase `verify`, venga verificato l'esito dei test E2E.

Avremmo anche potuto rimandare la definizione di questo profilo fino al momento della scrittura dei test E2E; tuttavia, per una questione di comodità, si è preferito lavorare in blocco alla maggior parte della configurazione del processo di building.

2 In particolare, verrà verificato che la percentuale di *istruzioni del progetto* coperte dai test sia del 100%.

3 La porta 27017 del container sarà mappata nella porta locale attesa dalla nostra applicazione *Spring Boot*; tale numero di porta sarà letto della proprietà `spring.data.mongodb.port` specificata nel file `application.properties`.

4 I test d'integrazione verranno saltati.

2.4 CONFIGURARE IL PROCESSO DI INTEGRAZIONE CONTINUA

Il nostro *GitHub Actions Workflow* principale, **Java CI with Maven**, è stato definito nel file `maven.yml` e verrà eseguito dopo ogni operazione *push* e dopo la creazione o l'aggiornamento di ogni *pull request*. Il workflow avrà un unico *Job*; questo sarà eseguito da un *runner* che usa l'ultima versione di *Ubuntu* e sarà costituito dai seguenti passi:

1. *Clone the GitHub repo in the Runner*, in cui cloniamo nel *runner* la repository GitHub del nostro progetto;
2. *Set up JDK 11 in the Runner*, in cui installiamo nel *runner* la versione 11 di Java, necessaria per poter eseguire il goal `sonar` del `sonar-maven-plugin`;
3. *Cache Maven packages and SonarQube artifacts*, in cui predisponiamo il salvataggio in cache delle dipendenze *Maven* scaricate e degli artefatti necessari per eseguire le analisi con *SonarCloud*;
4. *Run Unit Tests (with JaCoCo)*, in cui eseguiamo la *Maven Build* fino alla fase `verify` abilitando il profilo `jacoco`;
5. *Run Integration Tests in a separate build*, in cui eseguiamo un'altra build, questa volta abilitando il profilo `it-tests`;
6. *Send code coverage report (previously collected) to Coveralls*, in cui inviamo i report precedentemente generati da *JaCoCo* a *Coveralls*;
7. *Run End-to-end tests in a separate build*, in cui eseguiamo un'ulteriore build in cui abilitiamo il profilo `e2e-tests`;
8. *Run SonarQube analysis through SonarCloud*, in cui inviamo il nostro codice a *SonarCloud* al fine di farne analizzare la qualità.

Negli steps *Send code coverage report (previously collected) to Coveralls* e *Run SonarQube analysis through SonarCloud* invochiamo in modo diretto i goal, rispettivamente, del `coveralls-maven-plugin` e del `sonar-maven-plugin`; di conseguenza, è stato sufficiente configurare tali plugin nella sezione `<pluginManagement>` del POM.

Dopo aver verificato che tutto funzionasse correttamente, lo step *Run End-to-end tests in a separate build* è stato "commentato via": non sarà utile fino al momento dell'aggiunta dei test E2E.

Infine, evidenziamo anche il fatto che, per verificare il corretto funzionamento del processo di integrazione continua, abbiamo aggiunto una classe temporanea e il relativo test d'unità: in tal modo, è stato possibile generare un report sul coverage e verificare la corretta interazione con *Coveralls*.

Nel ramo `update` siamo andati ad aggiornare il *Java CI with Maven* workflow: le *Maven Build* verranno lanciate usando lo script `mvnw` del progetto **Maven Wrapper**; in tal modo, aumentiamo la riproducibilità del processo di integrazione continua andando a "fissare" la versione di *Maven* usata.

2.5 CLASSE DI COSTANTI PER IL TESTING

Nella classe `BookTestingConstants` sono state definite alcune costanti comuni per i test-case della nostra applicazione. Quest'approccio è stato adottato per una questione di praticità e tenendo ben a mente l'importanza della leggibilità dei test: per quanto l'uso di costanti non definite all'interno dei singoli test-case possa impattare negativamente su di essa, si ritiene che l'uso di nomi significativi per le costanti stesse vada a compensare ampiamente il problema.

3

PERSISTENCE LAYER

In questo capitolo illustreremo il lavoro svolto nel ramo `repository`, nel quale ci siamo occupati dell'implementazione e del testing del *Persistence layer* della nostra applicazione.

3.1 IMPLEMENTARE LA BOOKREPOSITORY USANDO SPRING DATA

Come specificato nella sezione 1.3.1, andremo ad usare *MongoDB*; questo sarà eseguito all'interno di un *Docker Container* e accessibile dalla porta 27017 del localhost. La *Repository* dovrà, pertanto occuparsi, attraverso l'uso di un `MongoClient`, della lettura-scrittura dei documenti nel DB e della mappatura tra documenti e DMO.

In generale, non saremo noi a fornire l'implementazione dell'interfaccia `BookRepository`, ma lasceremo tale resposabilità a *Spring Boot* andandoci ad appoggiare al progetto *Spring Data*.

3.1.1 *Spring Data e Spring Data MongoDB*

Spring Data è un progetto *Spring* il cui obiettivo è rendere semplice l'uso delle tecnologie per l'accesso ai dati, sia per database relazionali che non-relazionali [3]; tale progetto è costituito da un insieme di sotto progetti, ognuno tarato per uno specifico DB: tra questi, noi useremo *Spring Data MongoDB*.

Per poter usare il progetto **Spring Data MongoDB** dovremmo aggiungere alle nostre dipendenze lo *Spring Boot Virtual Package* `spring-boot-starter-data-mongodb`: questo fornirà alla nostra applicazione tutto il necessario per interagire con il database; in particolare, *Spring Boot* autoconfigurerà quanto serve per comunicare con un'istanza di *MongoDB* server in esecuzione sul localhost e in ascolto sulla porta 27017[1] [3].

3.1.2 *Mappatura tra Book DMO e documenti MongoDB*

La *Repository* dovrà occuparsi della conversione di documenti in DMO e viceversa; con *Spring Data* ci basterà usare:

- l'annotazione a livello di classe `@Document`, per identificare le classi candidate per il mapping;
- l'annotazione a livello di campo `@Id`, per identificare quale variabile usare per la proprietà `_id` del documento associato.

1 Sarà possibile sovrascrivere i default per connettersi a *MongoDB* server remoto andando ad aggiungere le proprietà `spring.data.mongodb.*` nel file `application.properties`.

Fatto ciò, tutto il lavoro di conversione verrà svolto dal `MongoMappingConverter` di *Spring* [5]; questo identificherà i DMO candidati per il mapping e creerà il corrispettivo documento MongoDB che:

- apparterrà alla *Collection* il cui nome è stato specificato come argomento dell'annotazione `@Document`;
- avrà come chiave primaria il valore del campo annotato con `@Id`;
- avrà come campi i restanti campi della classe Java[2].

Dunque, tutto quello dovremo fare per configurare la mappatura sarà annotare la classe `Book` con `@Document("Book")` ed il suo campo `isbn` con `@Id`.

3.1.3 *Far implementare la BookRepository a Spring Boot*

Usando *Spring Data* non dovremo preoccuparci di implementare le operazioni CRUD dell'interfaccia `BookRepository`; basterà farle estendere `MongoRepository<Book,Long>`[3]: fintanto che i metodi esposti dalla `BookRepository` rispettano le convenzioni di `Spring Data` relative alla firma, *Spring* sarà in grado di creare un'implementazione della nostra interfaccia a run-time e di iniettare una sua istanza nell'applicazione [5].

L'interfaccia `MongoRepository` espone diversi metodi per operazioni CRUD standard; tra questi sono compresi anche `findAll`, `findById`, `save`, `deleteById` e `existsById` [6], già dichiarati nella `BookRepository`. L'unico metodo che saremmo obbligati a dichiarare esplicitamente sarebbe `findAllByTitleLikeOrderByTitle`; tuttavia, per una questione di chiarezza e leggibilità, si è preferito lasciare un elenco esplicito dei metodi essenziali che vogliamo vengano esposti dalla nostra interfaccia.

3.2 testare l'implementazione della bookrepository

Anche se non implementeremo noi i metodi della `BookRepository`, risulterà comunque importante testarne il comportamento al fine di essere certi che sia quello atteso: la classe `BookRepositoryIT` conterrà i test per tali metodi.

Prima di eseguire questi test avremo bisogno di lanciare un *MongoDB*. Potremmo usare *Flapdoodle*, un *Embedded MongoDB* per il testing; tuttavia, non essendo un prodotto *MongoDB* ufficiale e, conseguentemente, non avendo garanzie sul fatto che la sua implementazione corrisponda a quella vera [7], abbiamo preferito testare fin da subito con un vero *MongoDB* eseguito all'interno di un *DockerContainer*. Questi test saranno, dunque, a tutti gli effetti dei test d'integrazione, ma verranno trattati come test d'unità: cercheremo, per quanto possibile, di scriverli prima del codice e di coprire tutte le possibili interazioni con il DB.

Per testare l'implementazione della `BookRepository` non avremo bisogno di tutti gli *Spring Beans*, ma ci basterà unicamente avviare le componenti fondamentali per il nostro *Persistence layer*; potremo fare ciò sfruttando il meccanismo delle *testing slices* di Spring: sarà sufficiente annotare il test-case `BookRepositoryIT` con `@DataMongoTest` e Spring farà tutto il resto [3].

2 Il nome dei campi sarà invariato a meno che non si vada a specificare un nome diverso con l'annotazione `@Field("nome")`.

3 L'interfaccia `MongoRepository<T,ID>` si aspetta come parametri il tipo del documento e delle sua chiave primaria.

Infine, proprio perché verificheremo la correttezza dell'implementazione della `BookRepository`, sarà bene evitare di sfruttare i metodi di tale interfaccia per eseguire operazioni di *setup* e *verify*. Per ripulire il contenuto, per salvare e per controllare la presenza di libri nel database, andremo ad utilizzare un oggetto di tipo `MongoOperations`: un'interfaccia fornita da *Spring Data MongoDB* che ci mette a disposizione diversi metodi per interagire con il nostro database [6]. Sfrutteremo la funzionalità di *Dependency Injection* di Spring per farci passare un oggetto che implementa tale interfaccia.

4

SERVICE LAYER

In questo capitolo illustreremo i tratti salienti relativi al testing e all'implementazione di `MyBookService`, la principale componente del *Business logic layer* della nostra applicazione.

La maggior parte del lavoro qui presentato è stato portato avanti nel ramo `service` e solamente alcune operazioni di refactoring sono state eseguite nel ramo `update`.

4.1 TESTARE E IMPLEMENTARE IL MYBOOKSERVICE

4.1.1 *Testare il Service layer in isolamento*

La classe `MyBookService`, dal punto di vista architetturale, è indipendente da ogni altro dettaglio implementativo: per testarla non avremo bisogno che venga eseguita nessuna ulteriore componente dell'applicazione *Spring Boot*. L'unica cosa fondamentale sarà *mockare* e *stubbare* la `BookRepository`: per fare ciò, abbiamo deciso di affidarci alle funzionalità di *Mocking*, *Dependency Injection* e *Stubbing* fornite da **Mockito**.

4.1.2 *Implementare il Book Service*

La classe `MyBookService` dovrà essere annotata con `@Service` e dovrà implementare l'interfaccia `BookService`, precedentemente descritta nella sezione 2.2.2. Inoltre, la classe verrà predisposta per la *constructor-injection*[1]: si andrà a richiedere a *Spring* l'iniezione di un'istanza di `BookRepository`.

L'implementazione dei metodi dell'interfaccia `BookService` è abbastanza immediata, dovremo solamente delegare all'istanza di `BookRepository` e lanciare la corretta eccezione a seconda del particolare caso.

Le eccezioni che il *Service* potrebbe lanciare sono `BookNotFoundException` e `BookAlreadyExistException`; queste sono state concordate in anticipo e già descritte nella sezione 2.2.2.

In tal contesto, risulta importante porre l'accento sul fatto che sarà responsabilità del `MyBookService` assicurarsi che:

- al momento dell'aggiunta di un nuovo libro, l'ISBN fornito non sia già presente nel database;
- al momento dell'update di un libro, l'ISBN fornito sia già presente nel database.

1 Avendo un unico costruttore con parametri avremmo anche potuto evitare di usare l'annotazione `@Autowired`, tuttavia, per una questione di chiarezza, abbiamo preferito lasciarla.

Questo implica la necessità di eseguire sempre un controllo, prima di invocare il metodo `save` della `BookRepository`, volto a verificare la presenza o meno del dato ISBN nel database; nel caso questo risulti non superato, verrà lanciata la corretta eccezione. Nel codice 4.1 possiamo osservare la soluzione adottata.

```
//...
  @Override
  public Book addNewBook(Book newBook) {
    checkBookNonExistenceByIsbn(newBook.getIsbn());
    return bookRepository.save(newBook);
  }
  @Override
  public Book replaceBook(Book editedBook) {
    checkBookExistenceByIsbn(editedBook.getIsbn());
    return bookRepository.save(editedBook);
  }
  private void checkBookExistenceByIsbn(long isbn) {
    if (!bookRepository.existsById(isbn))
      throw new BookNotFoundException(isbn);
  }
  private void checkBookNonExistenceByIsbn(long isbn) {
    if (bookRepository.existsById(isbn))
      throw new BookAlreadyExistException(isbn);
  }
//...
```

Codice 4.1: metodi addNewBook e replaceBook del MyBookService

4.2 MUTATION TESTING PER IL SERVICE

Il `MyBookService` contiene la logica della nostra applicazione: sarà importante valutare la qualità dei suoi test d'unità attraverso il processo di *Mutation Testing*.

Sarà nostro interesse non eseguire il processo di *Mutation Testing* durante ogni *Maven Build*; andremo, quindi, a definire il profilo `pitest` che ci permetterà di abilitare su richiesta il plugin `pitest-maven`.

4.2.1 *Profilo pitest*

Il plugin `pitest-maven` sarà configurato nella sezione `<pluginManagement>` del POM in modo tale che:

- vengano generate mutazioni usando gli operatori del set `STRONGER`;
- venga mutata unicamente la classe `MyBookService`;
- si provi a uccidere i mutanti usando unicamente i test di `MyBookServiceTest`[2];
- la build fallisca se tutti i mutanti generati non risultassero uccisi.

In questo profilo andremo a legare il goal `mutationCoverage` alla fase `test-compile`, così da non raggiungere la fase `test` e, di conseguenza, non rieseguire i test d'unità;

2 Restringeremo i test-case da esguire specificando come valore per la chiave `<testClasses>` il contenuto del package `io.github.francescomucci.spring.bookshelf.service`; in tale package sarà incluso anche il test-case d'integrazione `MyBookServiceIT`: dovremo stare attenti ad escluderlo dal processo settando correttamente il valore di `<excludedTestClass>`.

questa scelta è dettata dal fatto che attiveremo `pitest` in build eseguite speficatamente per il *Mutation Testing*.

4.2.2 *PIT Mutation Testing GitHub Actions workflow*

Anche se abbiamo configurato il plugin `pitest-maven` per restringere il *Mutation Testing* al solo *Service*, si è comunque deciso di configurare il *Processo di Integrazione Continua* in modo tale che questa procedura sia eseguita unicamente dopo la creazione o l'aggiornamento di una *Pull Request*.
Per fare ciò è stato necessario definire un ulteriore *GitHub Actions Workflow*, **PIT Mutation Testing on Pull Request**; tale workflow, contenuto nel file `pitest.yml`, è costituito da un solo *Job* con i seguenti steps:

1. *Clone the GitHub repo in the Runner*[3];
2. *Set up JDK 11 in the Runner*[3];
3. *Cache Maven packages*[3];
4. *Run PIT mutation testing*, in cui andiamo ad eseguire una Maven Build fino alla fase `test-compile` abilitando il profilo `pitest`;
5. *Archive PIT mutation testing report on GitHub*, in cui richiediamo che il contenuto della directory `/target/pit-reports` venga immagazzinato in un archivio su *GitHub*.

3 Questi primi tre steps sono praticamente equivalenti a quelli del workflow *Java CI with Maven*; per la loro descrizione si rimanda alla sezione 2.4.

5

WEB LAYER

In questo capitolo andremo a illustrare il *Presentation (o Web) layer* della nostra applicazione; in particolare, vedremo:

1. definizione dei DTO;
2. configurazione della validazione degli input;
3. definizione dell'interfaccia `BookWebController`;
4. configurazione della sicurezza web;
5. implementazione e testing del `MyBookWebController`;
6. implementazione e testing delle *Web Views*.

5.1 DATA TRANSFER OBJECT PATTERN

Per rappresentare i dati passati dagli utenti della nostra applicazione attraverso le *Web Views* andremo ad utilizzare un modello più semplice rispetto a quella fornito da `Book`, la nostra *Domain Model Class*; per far ciò seguiremo il **Data Trasfer Object (DTO)** pattern [8]. Andremo, dunque, a definire:

1. due DTO, `IsbnData` e `BookData`, che forniscono la rappresentazione dei dati per il *Presentation layer*;
2. un *Mapper*, `BookDataMapper`, che verrà usato dal *Web Controller* per convertire `Book` in `BookData` e viceversa.

5.1.1 *IsbnData e BookData*

I DTO svolgeranno un ruolo speculare a quello della *Domain Model Class* e, esattamente come questa, verranno creati come *POJOs*; in particolare:

- `IsbnData` avrà un solo campo `isbn` di tipo `String`;
- `BookData` estenderà `IsbnData` aggiungendo i campi `title` e `authors`, sempre di tipo `String`.

Questo modello semplificato renderà più facile la gestione dei form delle viste HTML, senza però rinunciare alla struttura che abbiamo scelto per i documenti immagazzinati nel database. In generale, avremo a disposizione una rappresentazione dei dati ottimizzata per poter essere usata come modello per le *Web Views*.

Le due DTO class, esattamente come la *Domain Model Class*, non saranno testate e verranno escluse dal calcolo del *Code Coverage*.

5.1.2 *MyBookDataMapper*

L'uso del *Mapper* per incapsulare la logica di conversione tra `Book` e `BookData` ci permetterà di mantenere queste due classi totalmente separate: potranno essere modificate in maniera indipendente e, in tal caso, l'unica altra classe da rimaneggiare sarà il `BookDataMapper` stesso [8]. In sostanza, il *Mapper* renderà il nostro codice più manutenibile.

L'interfaccia `BookDataMapper` verrà implementata dalla classe `MyBookDataMapper`[1]; tale implementazione sarà testata in isolamento nel test-case `MyBookDataMapperTest`.

5.2 MECCANISMO DI BEAN VALIDATION

L'ISBN verrà fornito dall'utente attraverso i form delle nostre pagine web e, di conseguenza, sarà fondamentale assicurarsi che questo sia un ISBN-13 valido: sarà, pertanto, necessario che la nostra applicazione implementi un meccanismo di validazione degli input.

Con Spring abbiamo a disposizione due possibili approcci: possiamo creare un custom-validator implementando l'interfaccia `Validator` oppure possiamo affidarci allo standard **JSR-380 (Bean Validation 2.0)** e lasciare che Spring usi come validatore l'*Hibernate Validator* [9]. Opteremo per questa seconda strada.

5.2.1 *JSR-380 Bean Validation 2.0*

Lo standard *JSR-380*, anche noto come *Bean Validation 2.0*, specifica una API per la validazione di Java Bean; *Hibernate Validator*[2] è la reference-implementation di tale API. Per poterli usare nel progetto, sarà sufficiente aggiungere tra le nostre dipendenze il virtuale package `spring-boot-starter-validation`.

La Bean Validation API ci permetterà di esprimere dei vincoli riguardo ai campi dei nostri DTO: sarà sufficiente usare le annotazioni del package `javax.validation.constraints`; in particolare, useremo [10]:

- `@NotNull` per richiedere che l'elemento annotato non sia `null`;
- `@NotBlank` per richiedere che la sequenza di caratteri annotata non sia `null` e contenga almeno un carattere non-whitespace;
- `@Pattern` per richiedere che la sequenza di caretteri annotata rispetti l'espressione regolare specificata nell'attributo `regexp`.

Oltre a questi, avremo anche a disposizione i vincoli di validazioni del package `org.hibernate.validator.constraints`; nello specifico, sfrutteremo[11]:

- `@ISBN` per richiedere che la sequenza di caratteri annotata rappresenti un valido ISBN-13.

Per ogni vincolo di validazione, attraverso l'attributo `message`, potremo configurare il messaggio di errore che verrà registrato nel caso il processo non vada a buon fine. Tutti i messaggi di errore saranno letti da proprietà definite nel file `ValidationMessages.properties` contenuto in `src/main/resources`.

1 `MyBookDataMapper` è stato aggiunto nel ramo `update`, ma poteva essere sviluppato fin da subito nel ramo `web-base`.

2 L'Hibernate Validator è un progetto separato da Hibernate e, dunque, non legato all'uso di un database relazionale.

Inoltre, con l'attributo `groups` potremo suddividere i vincoli in gruppi [12]: questo ci consentirà di richiedere che, al momento della validazione, vengano considerati unicamente i vincoli di un certo gruppo. Il valore dell'attributo `groups` dovrà essere il nome di un'interfaccia che rappresenta il dato gruppo; a questo scopo andremo a definire le interfacce `IsbnConstraints`, `TitleConstraints` e `AuthorsConstraints` che rappresentano rispettivamente il gruppo dei vincoli relativi ai campi `isbn`, `title` e `authors`.

5.2.2 *Configuriamo la Bean Validation nei DTO*

Il primo passo per configurare il processo di validazione è andare ad annotare i nostri DTO specificando i vincoli di validazione relativi ai loro campi. Come possiamo osservare nel codice 5.1, quando abiliteremo il processo per un oggetto `IsbnData` specificando il gruppo `IsbnConstraints`, il validatore usato andrà a verificare che il campo `isbn` non sia `null` e che sia un ISBN-13 valido.

```
//...
public class IsbnData {
  @NotNull(message = "{Blank.Field.Message}",
    groups = {Default.class, IsbnConstraints.class})
  @ISBN(type = ISBN_13, message = "{Invalid.ISBN.Message}",
    groups = {Default.class, IsbnConstraints.class})
  private String isbn;
  //...
}
```

Codice 5.1: vincoli di validazione per IsbnData

Specificheremo dei vincoli anche per i campi di `title` e `authors` di `BookData`: richiederemo che non siano `null`, che sia costituiti da almeno un carattere non-whitespace e che non contengano alcun carattere all'infuori di quelli elencati nell'espressione regolare specificata.

```
//...
public class BookData extends IsbnData {
  @NotBlank(message = "{Blank.Field.Message}",
    groups = {Default.class, TitleConstraints.class})
  @Pattern(regexp = "^$|[a-zA-Z0-9&,:.!? ]+$", message = "{Invalid.Title.Message}",
    groups = {Default.class, TitleConstraints.class})
  private String title;
  @NotBlank(message = "{Blank.Field.Message}",
    groups = {Default.class, AuthorsConstraints.class})
  @Pattern(regexp = "^$|[a-zA-Z, ]+$", message = "{Invalid.Authors.Message}",
    groups = {Default.class, AuthorsConstraints.class})
  private String authors;
  //...
}
```

Codice 5.2: vincoli di validazione per BookData

Risulta importante evidenziare che ogni vincolo verrà incluso nel gruppo `Default`; in tal modo, se abiliteremo la validazione senza specificare alcun gruppo, i nostri vincoli verranno comunque presi in considerazione.

5.2.3 *Testare la Bean Validation*

In `BookDataValidationTest` sono stati definiti i test per verificare la corretta configurazione dei vincoli di validazione: nella fase di setup a livello di classe, costruiremo un'istanza di `Validator`; durante la fase exercise, useremo tali istanze per andare a validare un dato oggetto `BookData`. Infine, nella fase verify, andremo a verificare se è stata rilevata la corretta violazione del vincolo.
Nel codice 5.3 vediamo uno dei metodi di test scritti.

```
//...
@Test
public void test_IsbnConstr_whenAllFieldsNull_thenNotNullForIsbn {
  BookData bookData = new BookData(null, null, null);
  Set<ConstraintViolation<BookData>> violations =
    validator.validate(bookData, IsbnConstraints.class);
  assertThat(violations).hasSize(1);
  violations.forEach(violation -> {
    assertThat(violation.getPropertyPath()).hasToString("isbn");
    assertThat(violation.getMessage()).isEqualTo("Please fill out this field");});
}
//...
```

Codice 5.3: esempio di test per Bean Validation

5.3 WORKFLOW DELLE PAGINE WEB E END-POINTS HTTP

Per poter implementare il *Web Controller* e le *Web Views* in rami separati è necessario concordare, oltre che sul modello dei dati, anche su quali siano gli end-points HTTP e sul workflow dell'interfaccia web. Andiamo, dunque, a illustrare la struttura e il funzionamento concordato per le nostre pagine web.

5.3.1 *Layout delle pagine web*

Verrà usato un layout comune per tutte le pagine web dell'applicazione; in particolare, in ognuna sarà presente la medesima barra di navigazione che andrà a contenere:

- per tutti gli utenti:
  - un link per la *Book home view* (GET `"/book"`);
  - un link per la *Book list view* (GET `"/book/list"`);
  - un link per la *Book search by ISBN view* (GET `"/book/searchByIsbn"`);
  - un link per la *Book search by title view* (GET `"/book/searchByTitle"`);
- solo per l'amministratore:
  - un link per la *Book new view* (GET `"/book/new"`).
- solo per gli utenti autenticati:
  - un form di logout che sarà composto unicamente da un bottone *Logout* e permetterà l'emissione di richieste POST per l'end-point `"/logout"`;

POST request per `"/logout"`

A seguito di una POST request per l'end-point `"/logout"`, verrà eseguita la procedura di logout per il dato utente, che verrà rendirizzato sulla *Book home view* attraverso l'URI `"/login/?logout"`; in tal caso, sulla pagina sarà mostrato un messaggio che notifica che l'operazione è avvenuta con successo.

5.3.2 *Book home view*

La *Book home view* verrà restituita a seguito di richieste GET per gli end-points `"/book"`, `"/"` e `"/login"`. Questa vista è definita nel file `bookHome.html` e conterrà:

- per utenti autenticati:
  - un messaggio di bentornato.
- per utenti anonimi:
  - un messaggio di benvenuto;
  - un form di login, dotato di remember-me checkbox, che permetterà l'emissione di POST request per l'end-point `"/login"`.

POST request per `"/login"`

A seguito di una POST request per l'end-point `"/login"`, verrà eseguita la procedura di login:

- se questa non ha successo, si verrà reindirizzati, attraverso l'URI `"/login/?error"`, sulla *Book home view*, dove verrà mostrato un consono messaggio di errore;
- se ha successo, si verrà reindirizzati sulla *Book home view* attraverso l'URI `"/book"`.

5.3.3 *Book list view*

La *Book list view* verrà restituita a seguito di richieste GET per l'end-point `"/book/list"`. Questa vista è definita nel file `bookList.html` e conterrà:

- un messaggio informativo, se non sono presenti libri nel DB (nel model manca l'attributo `"books"`);
- la tabella dei libri, altrimenti.

La tabella sarà costituita da una riga per ogni libro del database e da tre colonne che ospiteranno rispettivamente l'ISBN, il titolo e gli autori; inoltre, se l'utente è autenticato come amministratore, saranno presenti due colonne aggiuntive che conterranno:

- un link per la *Book edit view* (GET `"/book/edit/{isbn}"`);
- un bottone `Delete` che aprirà un dialog-box di conferma per l'operazione di cancellazione; in caso di risposta affermativa, verrà emessa una POST request per l'end-point `"/book/delete/{isbn}"`.

POST request per `"/book/delete/{isbn}"`

Una POST request per l'end-point `"/book/delete/{isbn}"` rappresenta una richiesta di cancellazione per il libro il cui ISBN-13 è fornito come path-variable nell'URI; tale variabile dovrà essere validata:

- se la validazione non ha successo:
  - verrà restituita la *Invalid ISBN-13 error view*;

- se ha successo
  - e l'ISBN-13 fornito non è presente nel database, verrà restituita la *Book not found error view*;
  - e l'ISBN-13 è presente, sarà eseguita l'operazione di cancellazione e si verrà reindirizzati sulla *Book list view* attraverso l'URI `"/book/list"`.

5.3.4 *Book edit view*

A seguito di una richiesta GET per l'end-point `"/book/edit/{isbn}"`, l'ISBN fornito come path-variable verrà validato:

- se non supera la validazione, verrà restituita la *Invalid ISBN-13 error view*;
- se la supera, ma non è presente nel database, verrà restituita la *Book not found error view*; altrimenti, verrà restituita la *Book edit view*.

La *Book edit view*, definita nel file `bookEdit.html`, dovrà contenere un form di editing per il libro specificato; in tale form saranno presenti i campi per editare il titolo e gli autori e un campo nascosto per l'ISBN. Riempiendo i campi e premendo il bottone *Save edit* verrà emessa una POST request per l'end-point `"/book/save"`.

POST request per `"/book/save"`

A seguito di una POST request per l'end-point `"/book/save"`, Spring userà i parametri forniti per costruire un oggetto `BookData` da passare al metodo del *Web Controller* che gestirà tale richiesta. Il contenuto dei campi di tale oggetto verrà validato:

- se l'ISBN-13 non è valido, verrà restituita la *Invalid ISBN-13 error view*;
- se titolo o autori sono non validi, verrà restituita la *Book edit view*, con i campi del form riempiti con i dati precedentemente inseriti e i corretti messaggi d'errore in evidenza;
- se non ci sono errori di validazione
  - e il libro da editare non risulta presente, verrà restituita la *Book not found error view*;
  - e il libro risulta presente, questo verrà rimpiazzato con la nuova versione e si verrà reindirizzati sulla *Book list view* attraverso l'URI `"/book/list"`.

5.3.5 *Book new view*

La *Book new view* verrà restituita a seguito di richieste GET per l'end-point `"/book/new"`. Questa vista è definita nel file `bookNew.html` e conterrà un form per aggiungere nuovi libri al database; tale form sarà composto dai campi per l'inserimento dell'ISBN-13, del titolo e degli autori del libro e permetterà l'emissione di POST request per l'end-point `"/book/add"`.

POST request per `"/book/add"`

A seguito di una POST request per l'end-point `"/book/add"`, Spring userà i parametri forniti per costruire un oggetto `BookData` da passare al metodo del *Web Controller* che gestirà tale richiesta. Il contenuto dei campi di tale oggetto verrà validato:

- nel caso siano presenti errori di validazione, dovrà essere restituita la *Book new view* con i campi del form pre-riempiti e i corretti messaggi d'errore in evidenza;

- nel caso non siano presenti errori di validazione:
  - se il libro risulta già presente nel database, dovrà essere restituita la *Book already exist error view*;
  - altrimenti, il libro verrà aggiunto al modello nell'attributo `books` e si verrà reindirizzati sulla *Book list view* attraverso l'URI `"/book/list"`.

5.3.6 *Book search by ISBN view*

La *Book search by ISBN view* verrà restituita a seguito di richieste GET per l'end-point `"/book/searchByIsbn"`. Questa vista è definita nel file `bookSearchByIsbn.html` e conterrà un form per ricercare un libro a partire dal suo ISBN-13; tale form sarà composto unicamente dal campo per l'inserimento dell'ISBN-13 e permetterà l'emissione di GET request per l'end-point `"/book/getByIsbn"`.

GET request per `"/book/getByIsbn"`

A seguito di una GET request per l'end-point `"/book/getByIsbn"`, Spring userà il parametro `isbn` fornito nella richiesta per costruire un oggetto `BookData`[3] e verrà eseguito il processo di validazione:

- se l'ISBN è invalido, verrà restituita la *Book search by ISBN view* con il campo del form pre-riempito e il corretto messaggio d'errore in evidenza;
- se l'ISBN è valido:
  - se il libro non è presente nel database, verrà restituita la *Book not found error view*;
  - altrimenti, il libro verrà aggiunto al modello nell'attributo `books` e verrà restituita la *Book search by ISBN view* che conterrà il campo ISBN pre-riempito e una tabella con i dati relativi al testo cercato.

5.3.7 *Book search by title view*

La *Book search by title view* verrà restituita a seguito di richieste GET per l'end-point `"/book/searchByTitle"`. Questa vista è definita nel file `bookSearchByTitle.html` e conterrà un form per ricercare i libri per titolo; tale form sarà composto unicamente dal campo per l'inserimento del titolo e permetterà l'emissione di GET request per l'end-point `"/book/getByTitle"`.

GET request per `"/book/getByTitle"`

A seguito di una GET request per l'end-point `"/book/getByTitle"`, Spring userà il parametro `title` fornito nella richiesta per costruire un oggetto `BookData`[3] e verrà eseguito il processo di validazione:

- se il titolo è invalido, verrà restituita la *Book search by title view* con il campo del form pre-riempito e il corretto messaggio d'errore in evidenza;
- se il titolo è valido:
  - se nessun libro contiene la stringa ricercata nel titolo, verrà restituita la *Book not found error view*;

3 Gli altri campi dell'oggetto `BookData` saranno `null`.

– se, invece, sono presenti libri che la contengono, questi verranno aggiunti al modello nell'attributo `books` e verrà restituita la *Book search by title view* contenente il campo pre-riempito e una tabella con i dati dei testi trovati.

5.3.8 *Error views*

Eccezione	Sollevata da	Pagina Web	File HTML
`BookNotFoundException`	`MyBookService`	*Book not found error view*	`bookNotFound.html`
`BookAlreadyExistException`	`MyBookService`	*Book already exist error view*	`bookAlreadyExist.html`
`InvalidIsbnException`	`MyBookWebController`	*Invalid ISBN-13 error view*	`invalidIsbn.html`

Tabella 1: Associazione tra Eccezioni personalizzate e pagine web.

Sfutteremo il meccanismo di gestione delle eccezioni fornito da *Spring* per far sì che, ogni volta che verrà sollevata una delle eccezioni da noi definite, la nostra applicazione servirà la corrispettiva pagina web d'errore: nella tabella 1 possiamo vedere l'associazione tra eccezioni, componenti che le sollevano e file HTML restituiti.

Ognuna delle nostre pagine di errore mostrerà un box informativo contenente:

- lo *status-code* della risposta HTTP ricevuta;
- la *reason-phrase* associata al dato status code;
- il messaggio associato all'eccezione sollevata.

Questi tre dati verranno letti da altrettanti attributi del model, `status`, `error` e `message`; il *Web Controller* non dovrà occuparsi di settare questi attributi, ma, come vedremo a breve, tale compito sarà affidato a un *Exception Handler*.

Unknown error view

Se dovesse venir sollevata un'eccezione di altro tipo, Spring restituirà automaticamente una error page di default; questa sarà sovrascritta dalla *Unknown error view* da noi definita nel file `error.html` nella directory `src/main/resources/templates` [13].

5.4 INTERFACCIA DEL WEB CONTROLLER

Una volta fissati gli end-points HTTP e il workflow delle pagine web, potremo andare a definire l'interfaccia del nostro *Web Controller*; al suo interno, oltre a stabilire le firme dei metodi, useremo delle annotazioni per specificare:

- il tipo di richieste HTTP che i suoi metodi dovranno gestire;
- quando e come abilitare il processo di validazione.

Quando implementeremo il *Web Controller*, ci basterà unicamente annotarne la classe con `@Controller`; tutte le altre annotazioni verranno ereditate dalla sua interfaccia [14]. Quest'approccio, possibile dalla versione 5.1 dello *Spring Framework* [15], ci permette di specificare in modo completo il contratto di mapping del controller.

5.4.1 *Abilitare la Bean Validation*

A seguito di una richiesta HTTP, se il metodo del *Web Controller* che la gestisce ha come parametro uno dei nostri DTO, Spring si occuperà di crearne un'istanza da passare al metodo e mapperà eventuali *path-variables* e *query-parameters* nei corrispettivi campi.

In alcuni casi ci interesserà che, dopo questa operazione di *data-binding*, venga eseguita anche la procedura di validazione. Nella sezione 5.2.2 abbiamo già visto come specificare i vincoli di validazione per i campi dei nostri DTO; a questo punto, tutto quello che dovremo fare sarà abilitarla andando ad annotare il corrispettivo parametro del metodo con `@Valid`: Spring avvierà automaticamente l'*Hibernate Validator*, eseguirà la validazione ed immagazzinerà i risultati in un'istanza di `BindingResult` [9].

Se vogliamo che durante la validazione venga considerato unicamente un particolare sottogruppo dei nostri vincoli, potremo sfruttare l'annotazione `@Validated` del package `org.springframework.validation.annotation` [12]: e.g. specificando il gruppo `TitleConstraints.class`, al momento della validazione, verrà valutato unicamente il rispetto dei vincoli relativi al titolo e tutti gli altri verranno ignorati.

5.4.2 *Interfaccia BookWebController*

Nella classe `BookWebControllerConstants`, elencheremo le costanti concordate per il *Web Controller*: gli URIs (visionabili nella tabella 2), i nomi dei file HTML in cui saranno definite le viste e i nomi degli attributi del *Model* che questi dovrà settare.

L'interfaccia stabilita per il Web Controller è visionabile nel codice 5.4.

```
1  //...
2  public interface BookWebController {
3    @GetMapping({URI_HOME, URI_BOOK_HOME, URI_LOGIN})
4    public String getBookHomeView();
5    @GetMapping(URI_BOOK_LIST)
6    public String getBookListView(Model model);
7    @PostMapping(URI_BOOK_DELETE)
8    public String postDeleteBook(@Valid IsbnData isbn, BindingResult result);
9    @GetMapping(URI_BOOK_EDIT)
10   public String getBookEditView(@Validated(IsbnConstraints.class)
11     BookData editFormData, BindingResult result);
12   @PostMapping(URI_BOOK_SAVE)
13   public String postSaveBook(@Valid BookData editFormData, BindingResult result);
14   @GetMapping(URI_BOOK_NEW)
15   public String getBookNewView(BookData addFormData);
16   @PostMapping(URI_BOOK_ADD)
17   public String postAddBook(@Valid BookData addFormData, BindingResult result);
18   @GetMapping(URI_BOOK_SEARCH_BY_ISBN)
19   public String getBookSearchByIsbnView(BookData searchFormData);
20   @GetMapping(URI_BOOK_GET_BY_ISBN)
21   public String getBookByIsbn(@Validated(IsbnConstraints.class)
22     BookData searchFormData, BindingResult result, Model model);
23   @GetMapping(URI_BOOK_SEARCH_BY_TITLE)
24   public String getBookSearchByTitleView(BookData searchFormData);
25   @GetMapping(URI_BOOK_GET_BY_TITLE)
26   public String getBookByTitle(@Validated(TitleConstraints.class)
```

```
27      BookData searchFormData, BindingResult result, Model model);
28  }
```

Codice 5.4: interfaccia BookWebController

Costante	URI
URI_HOME	"/"
URI_BOOK_HOME	"/book"
URI_LOGIN	"/login"
URI_BOOK_NEW	"/book/new"
URI_BOOK_ADD	"/book/add"
URI_BOOK_LIST	"/book/list"
URI_BOOK_EDIT	"/book/edit/{isbn}"
URI_BOOK_SAVE	"/book/save"
URI_BOOK_DELETE	"/book/delete/{isbn}"
URI_BOOK_SEARCH_BY_ISBN	"/book/searchByIsbn"
URI_BOOK_GET_BY_ISBN	"/book/getByIsbn"
URI_BOOK_SEARCH_BY_TITLE	"/book/searchByTitle"
URI_BOOK_GET_BY_TITLE	"/book/getByTitle"

Tabella 2: Costanti per URIs

Commento sul alcune scelte fatte

A livello di validazione, avere come parametro un `IsbnData` annotato con `@Valid` o un `BookData` annotato con `@Validated(IsbnConstraints.class)` è pressoché equivalente: a scopo dimostrativo, abbiamo scelto di usare entrambi gli approcci.

Se non specifichiamo alcun gruppo, l'annotazione `@Validated` funziona in modo identico a `@Valid` [12]; tuttavia, sempre a scopo dimostrativo, abbiamo preferito sfruttarle entrambe.

5.5 WEB SECURITY CONFIGURATION

Prima di implementare il *Web Controller* andremo a configurare la sicurezza web della nostra applicazione appoggiandoci al framework **Spring Security**: per accedere alle sue funzionalità dovremo aggiungere la dependency `org.springframework.boot:spring-boot-starter-security` [3].

5.5.1 *Configurazione di default*

Autenticazione

Spring Security, senza bisogno di altro se non l'aggiunta della dependency, autoconfigurerà l'autenticazione [16]: ogni volta che questa sarà richiesta, il framework ci ridirezionerà all'URL `"/login"` restituendoci una pagina di login di default; se non configuriamo nient'altro, le credenziali d'accesso saranno *user* per lo username e per la password

una stringa casuale stampata nel log dell'applicazione all'avvio [3]. Completando con successo l'operazione di login, si verrà reindirizzati all'URL precedentemente richiesto. Inviando una richiesta POST per l'URL `"/logout"`, la sessione HTTP verrà terminata, il cookie di sessione `JSESSIONID` verrà cancellato e si verrà reindirizzati all'URI `"/login/?logout"`.

CSRF protection

Spring Security v5, di default, ci fornirà protezione da attacchi di *Cross-Site Request Forgery (CSRF)* [16]. In assenza di tale protezione, un attaccante potrebbe sfruttare una sessione esistente per forzare l'invio, attraverso pagine web malevoli, di richieste non autorizzate e la nostra applicazione non sarebbe in grado di distinguerle da quelle legittime [17]. Per difenderci da questo attacco *Spring Security* sfrutta il *Synchronizer Token Pattern* e, di conseguenza, dovremo garantire che ogni richiesta POST includa un CSRF token generato casualmente server-side e associato alla data sessione: quando una richiesta è inviata, il server confronterà il token ricevuto con quello atteso e, nel caso non fossero uguali, questa verrà scartata [18]. Il sito malevolo non sarà in grado di fornire il CSRF token associato alla sessione e, pertanto, non potranno essere forgiate POST request contraffatte. Affinché la nostra applicazione sia effettivamente protetta, ovviamente, non dovremo mai permettere a richieste GET di alterarne lo stato [17].

Affinché il *Synchronizer Token Pattern* funzioni correttamente, dovremo garantire l'inclusione del CSRF token come hidden-input nei form che permettono l'emissione di POST request: per fortuna, *Thymeleaf* ci solleverà da questa responsabilità andando a svolgere quest'inclusione automaticamente.

5.5.2 *Configurare Spring Security*

Per personalizzare il funzionamento di *Spring Security* dovremo definire una `@Configuration` class annotata con `@EnableWebSecurity` e che estende `WebSecurityConfigurerAdapter` [12].

Come possiamo vedere dal codice 5.5, definendo il metodo `configureGlobal(AuthenticationManagerBuilder auth)` [9], abbiamo impostato una semplice in-memory authentication per l'amministratore: le sue credenziali verranno lette dalle proprietà `admin.username` e `admin.password` oppure, nel caso queste non siano definite, verranno usati i defaults `"Admin"` e `"Password"`. Per codificare la password verrà usata la funzione di hashing *bcrypt* [3].

```
1  //...
2  @Configuration
3  @EnableWebSecurity
4  public class MyWebSecurityConfiguration extends WebSecurityConfigurerAdapter {
5    //...
6    private static final String ROLE_ADMIN = "ADMIN";
7    @Value("${admin.username:Admin}")
8    private String adminUsername;
9    @Value("${admin.password:Password}")
10   private String adminPassword;
11   @Autowired
12   private PasswordEncoder passwordEncoder;
```

```
13   @Bean
14   public PasswordEncoder passwordEncoder() {
15     return new BCryptPasswordEncoder();
16   }
17   @Autowired
18   public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
19     auth.inMemoryAuthentication().passwordEncoder(passwordEncoder)
20       .withUser(adminUsername)
21       .password(passwordEncoder.encode(adminPassword))
22       .roles(ROLE_ADMIN);
23   }
24   //...
25 }
```

Codice 5.5: configurare l'autenticazione

Sovrascrivendo il metodo `configure(HttpSecurity http)`, come possiamo osservare nel codice 5.6, saremo in grado di configurare la sicurezza delle richieste HTTP e l'uso di una pagina di login personalizzata [9].

```
1  //...
2  public class MyWebSecurityConfiguration extends WebSecurityConfigurerAdapter {
3    //...
4    private static final String[] UNSECURED_URIS = {
5      URI_HOME, URI_LOGIN, URI_BOOK_HOME, URI_BOOK_LIST, URI_BOOK_SEARCH_BY_ISBN,
6      URI_BOOK_GET_BY_ISBN, URI_BOOK_SEARCH_BY_TITLE, URI_BOOK_GET_BY_TITLE,
7      "/actuator/health"
8    };
9    private static final String ROLE_ADMIN = "ADMIN";
10   //...
11   @Override
12   protected void configure(HttpSecurity http) throws Exception {
13     http
14       .authorizeRequests()
15         .antMatchers(UNSECURED_URIS).permitAll()
16         .anyRequest().hasRole(ROLE_ADMIN)
17       .and().formLogin()
18         .loginPage(URI_LOGIN)
19         .defaultSuccessUrl(URI_BOOK_HOME, true)
20       //...
21   }
22 }
```

Codice 5.6: configurare la sicurezza delle richieste HTTP e la pagina di login

Con `authorizeRequests()` specifichiamo le regole di autorizzazione per le richieste HTTP [18]; nel nostro caso, solo l'amministratore potrà inviare richieste per creare, editare o cancellare libri.

Con `formLogin().loginPage(URI_LOGIN)` chiediamo al framework di non usare come pagina di login quella di default, ma bensì la *Book home view*; ovviamente, sarà nostra responsabilità far sì che il web controller mappi le richieste GET per `"/login"` nella vista corretta [18]. Dovremo, inoltre, far sì che l'HTML form fornito:

- permetta l'emissione di POST request per l'URI `"/login"`;
- includa un CSRF Token (di questo se ne occuperà automaticamente *Thymeleaf*);

- usi come parametri per la password e l'username, rispettivamente, `username` e `password`.

Inoltre, con `defaultSuccessUrl(URI_BOOK_HOME, true)` richiediamo che, nel caso la procedura di login abbia successo, si venga sempre reindirizzati sulla *Book home view*.

5.5.3 *Spring security e testing*

Elencando tra le nostre dipendenze di test la managed-dependency `spring-security-test` avremo accesso alle funzionalità di testing offerte da *Spring Security* [18], più in dettaglio:

- saremo in grado di specificare per quale utente i nostri test dovranno essere eseguiti;
- avremo accesso all'integrazione di *Spring Security* con `MockMvc`; negli *Spring MVC Test*, senza troppa difficoltà, potremo inviare POST request che includono un CSRF token valido e verificare l'autenticazione e il logout.

Specificare per quale utente i test dovranno essere eseguiti

In assenza di altra configurazione, tutti i nostri test verranno eseguiti per l'utente anonimo (sostanzialmente un utente non autenticato) [18]. Usando l'annotazione `@WithMockUser` potremo specificare il nome e il ruolo di un utente; se tale annotazione:

- sarà presente a livello di metodo, il dato test verrà eseguito per l'utente specificato;
- sarà presente a livello di classe, tutti i test della classe per cui non è specificato un altro utente verranno eseguiti per quello dato; in tal caso, se volessimo che un particolare test venisse effettuato per l'utente anonimo, dovremo specificare l'annotazione `@WithAnonymousUser` a livello di metodo.

Nel progetto eseguiremo molti test per l'amministratore: di conseguenza, per evitare di ripetere più volte gli stessi attributi nell'annotazione `@WithMockUser`, andremo a definire la meta annotazione `@WithMockAdmin` [18]:

```
1 //...
2 @Retention(RetentionPolicy.RUNTIME)
3 @WithMockUser(value = "Admin", roles = "ADMIN")
4 public @interface WithMockAdmin {}
```

Codice 5.7: annotazione WithMockAdmin

Testare la configurazione di sicurezza

Anche la `MyWebSecurityConfiguration` è stata scritta seguendo il TDD: nella classe `MyWebSecurityConfigurationTest` verifichereremo la corretta configurazione del meccanismo di autenticazione, del logout e della sicurezza delle richieste HTTP. Verranno, inoltre, aggiunti dei test per verificare le nostre aspettative sul comportamento di default di *Spring Security*.

In questo test-case useremo l'annotazione `@WebMvcTest` per far sì che vengano caricate solamente le componenti fondamentali per lo *Spring MVC* e per autoconfigurare il `MockMvc`; inoltre, non avendo ancora implementato il *Web Controller*, richiederemo a Spring l'iniezione di un mocked `BookWebController`.

Per comprendere quanto sia semplice inviare POST request che includono un CSRF token valido e verificare l'autenticazione e il logout, vediamo alcuni metodi di test che abbiamo implementato:

```
//...
public class MyWebSecurityConfigurationTest {
  @Autowired
  private MockMvc mvc;
  //...
  @Test
  public void test_login_whenValidCred_redirectToBookHome()
  throws Exception {
    mvc.perform(formLogin().user(VALID_USER_NAME).password(VALID_PASSWORD))
      .andExpect(status().is3xxRedirection())
      .andExpect(redirectedUrl(URI_BOOK_HOME))
      .andExpect(authenticated().withRoles("ADMIN"));
  }
  //...
  @Test
  @WithMockAdmin
  public void test_logout_whenAdmin_redirectToLoginUrlWithLogoutParam()
  throws Exception {
    mvc.perform(logout())
      .andExpect(status().is3xxRedirection())
      .andExpect(redirectedUrl(URI_LOGIN + "?logout"))
      .andExpect(unauthenticated());
  }
  //...
  @Test
  public void test_postSaveBook_whenAnonymousUser_redirectToLogin()
  throws Exception {
    mvc.perform(post(URI_BOOK_SAVE).with(csrf())
      .param("isbn", VALID_ISBN13_WITHOUT_FORMATTING)
      .param("title", NEW_TITLE)
      .param("authors", AUTHORS_STRING))
        .andExpect(status().is3xxRedirection())
        .andExpect(redirectedUrl("http://localhost" + URI_LOGIN));
  }
  //...
  @Test
  public void test_postDeleteBook_whenInvalidCsrf_return403()
  throws Exception {
    mvc.perform(post("/book/delete/" + VALID_ISBN13_WITHOUT_FORMATTING)
      .with(csrf().useInvalidToken()))
      .andExpect(status().isForbidden());
  }
```

Codice 5.8: alcuni test della MyWebSecurityConfigurationTest.class

5.6 WEB CONTROLLER TESTS E IMPLEMENTAZIONE

Nel ramo `web-controller` andremo a sviluppare il `MyBookWebController`; questo implementerà i metodi dell'interfaccia `BookWebController` ed andrà, di conseguenza, a esporre gli end-points HTTP concordati nella sezione 5.3. Il *Web Controller* avrà come collaboratori un `BookService` e un `BookDataMapper`, che userà rispettivamente per richiedere operazioni sul database e per eseguire operazioni di conversione da/a DTO. A seconda del diverso end-point, sarà sua responsabilità verificare l'esito del processo di validazione usando l'istanza di `BindingResult` passatagli da *Spring* e, eventualmente, sollevare una `InvalidIsbnException`. A scopo d'esempio, vediamo l'implementazione dei POST end-points `"/book/save"` e `"/book/add"`:

```
1  //...
2  @Controller("BookWebController")
3  public class MyBookWebController implements BookWebController {
4    @Autowired
5    private BookService service;
6    @Autowired
7    private BookDataMapper map;
8    //...
9    @Override
10   public String postSaveBook(BookData editFormData, BindingResult result) {
11     if (!result.hasErrors()) {
12       service.replaceBook(map.toBook(editFormData));
13       return REDIRECT + URI_BOOK_LIST;
14     }
15     if (result.hasFieldErrors("isbn"))
16       throw new InvalidIsbnException(editFormData.getIsbn());
17     return VIEW_BOOK_EDIT;
18   }
19   //...
20   @Override
21   public String postAddBook(BookData addFormData, BindingResult result) {
22     if (result.hasErrors())
23       return VIEW_BOOK_NEW;
24     service.addNewBook(map.toBook(addFormData));
25     return REDIRECT + URI_BOOK_LIST;
26   }
27   //...
28 }
```

Codice 5.9: POST end-points "/book/save" e "/book/add"

5.6.1 *Gestione delle eccezioni con Spring*

Come illustrato nella sezione 5.3.8, a seconda della particolare eccezione sollevata, dovrà essere restituita una diversa pagina d'errore; per far ciò sfrutteremo il meccanismo di gestione delle eccezioni fornito da *Spring*.

All'interno di una classe annotata con `@Controller` potremo definire dei metodi speciali per gestire eventuali eccezioni sollevate durante l'elaborazione delle richieste HTTP. Tramite l'annotazione `@ExceptionHandler` potremo specificare la tipologia di

eccezione gestita dal particolare metodo; ciascuno di essi preparerà il `Model` e restituirà la corretta vista d'errore [12]. Inoltre, usando l'annotazione `@ResponseStatus` saremo in grado di personalizzare lo status code restituito nella risposta HTTP [12]: se durante l'elaborazione di una richiesta verrà sollevata un'eccezione, *Spring* si occuperà di invocare il particolare metodo di gestione e costruirà una risposta con il corretto status code. Potremmo anche decidere di non definire i metodi di gestione delle eccezioni nel *Web Controller*, ma in una classe dedicata; sarà sufficiente annotare tale classe con `@ControllerAdvice`.

Nel codice 5.10 possiamo osservare la classe per la gestione delle eccezioni usata nel progetto.

```
//...
@ControllerAdvice
public class MyBookWebControllerExceptionHandler {
  @ExceptionHandler(InvalidIsbnException.class)
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  private String handleInvalidIsbnException(InvalidIsbnException exception, Model
     model) {
    addErrorModelAttributes(exception, model, HttpStatus.BAD_REQUEST);
    return ERROR_INVALID_ISBN;
  }
  @ExceptionHandler(BookNotFoundException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  private String handleBookNotFoundException(BookNotFoundException exception, Model
      model) {
    addErrorModelAttributes(exception, model, HttpStatus.NOT_FOUND);
    return ERROR_BOOK_NOT_FOUND;
  }
  @ExceptionHandler(BookAlreadyExistException.class)
  @ResponseStatus(HttpStatus.CONFLICT)
  private String handleBookAlreadyExistException(BookAlreadyExistException
     exception, Model model) {
    addErrorModelAttributes(exception, model, HttpStatus.CONFLICT);
    return ERROR_BOOK_ALREADY_EXIST;
  }
  private void addErrorModelAttributes(Exception exception, Model model, HttpStatus
      httpStatus) {
    model.addAttribute(MODEL_ERROR_CODE, httpStatus.value());
    model.addAttribute(MODEL_ERROR_REASON, httpStatus.getReasonPhrase());
    model.addAttribute(MODEL_ERROR_MESSAGE, exception.getMessage());
  }
}
```

Codice 5.10: classe di gestione delle eccezioni

5.6.2 *Tests per il Web Controller*

La classe `MyBookWebControllerTest` conterrà i test d'unità per il nostro *Web Controller*; anche in questo caso useremo l'annotazione `@WebMvcTest` per caricare unicamente le componenti fondamentali per lo *Spring MVC* e per autoconfigurare il `MockMvc`. Per quanto riguarda i collaboratori, richiederemo a Spring l'iniezione di un mocked `BookService`

e utilizzeremo l'implementazione del `BookDataMapper` a nostra disposizione richiedendone il caricamento attraverso l'annotazione `@Import(MyBookDataMapper.class)` [19].

La scelta di usare un `BookDataMapper` è successiva alla scrittura dei test d'unità per il *Web Controller*; per tal motivo abbiamo deciso di non andare a mockare e stubbare il mapper preferendo l'uso dell'implementazione che abbiamo sviluppato e testato.

5.7 WEB VIEWS TESTS E IMPLEMENTAZIONE

A questo punto, andiamo a illustrare il lavoro svolto nel ramo `web-views`: vedremo come implementare e testare, in maniera indipendente rispetto al *Web Controller*, le pagine web specificate nella sezione 5.3. Queste pagine verranno realizzate usando la template engine *Thymeleaf*; in particolar modo, verranno approfonditi l'integrazione di quest'ultimo con *Spring Security* e l'uso dei *Thymeleaf fragments* per definire componenti HTML riusabili. Per quanto riguarda lo stile delle pagine, ci affideremo quasi totalmente a *Bootstrap 4*.

Per una questione di spazio, non analizzeremo ogni singola vista, ma ci soffermeremo solamente su alcuni frammenti HTML particolarmente interessanti.

5.7.1 *Configurare la sicurezza delle risorse statiche*

Se lasciamo la configurazione di sicurezza invariata, la nostra applicazione non sarà in grado di stilizzare le pagine web per tutti gli utenti, ma solo per l'amministratore: per risolvere la questione dovremo fare in modo che i file di stile presenti nei nostri *WebJars*[4] e nella directory `src/main/resources/static/css` siano sempre accessibili. Per non andare a complicare l'implementazione del metodo `configure(HttpSecurity http)`, potremo sovrascrivere `configure(WebSecurity web)` e specificare le risorse che *Spring Security* dovrà ignorare [12]. Possiamo osservare come abbiamo alterato la `MyWebSecurityConfiguration` nel codice 5.11.

```
1  //...
2  public class MyWebSecurityConfiguration extends WebSecurityConfigurerAdapter {
3    //...
4    private static final String[] STATIC_RESOURCES = {
5      "/webjars/**", "/css/**"
6    };
7    //...
8    @Override
9    public void configure(WebSecurity web) throws Exception {
10     web.ignoring().antMatchers(STATIC_RESOURCES);
11   }
12   //...
13 }
```

Codice 5.11: permettere l'accesso alle risorse statiche

4 Oltre a *Bootstrap 4* andremo a utilizzare un ulteriore *WebJars*, `font-awesome`, che ci metterà a disposizione una libreria di icone da poter usare nelle nostre pagine HTML.

5.7.2 *Testare le web views*

Per scrivere i test d'unità delle nostre viste web ci affideremo a *HtmlUnit*; in tali test andremo a mockare e stubbare il *Web Controller* sfruttando le funzionalità di mocking offerte da *Spring*. Nel codice 5.12 possiamo osservare la coppia di test per la *Book home view* che verificano il contenuto dell'*header-message*, rispettivamente, per gli utenti anonimi e per quelli autenticati[5].

```
//...
@WebMvcTest(controllers = BookWebController.class)
public class BookHomeViewTest {
  @Autowired
  private WebClient webClient;
  @MockBean
  private BookWebController bookWebController;
  @Before
  public void setup() {
    webClient.setCssErrorHandler(new SilentCssErrorHandler());
    webClient.getCookieManager().clearCookies();
  }
  @Test
  public void test_whenNotAuth_welcomeMsg() throws Exception {
    when(bookWebController.getBookHomeView()).thenReturn(VIEW_BOOK_HOME);
    HtmlPage bookHomeView = webClient.getPage(URI_BOOK_HOME);
    HtmlHeader header = (HtmlHeader)
      bookHomeView.getElementsByTagName("header").get(0);
    assertThat(
      BookViewTestingHelperMethods.removeWindowsCR(header.asText()))
      .isEqualTo("Welcome to my book library!" + "\n" +
        "Feel free to explore, but remember that you need" +
        "administrator privileges to create, edit or delete books");
  }
  @Test
  @WithMockAdmin
  public void test_whenAuth_welcomeBackMsg() throws Exception {
    when(bookWebController.getBookHomeView()).thenReturn(VIEW_BOOK_HOME);
    HtmlPage bookHomeView = webClient.getPage(URI_BOOK_HOME);
    HtmlHeader header = (HtmlHeader)
      bookHomeView.getElementsByTagName("header").get(0);
    assertThat(
      BookViewTestingHelperMethods.removeWindowsCR(header.asText()))
      .isEqualTo("Welcome back!" + "\n" +
        "Feel free to explore, create, edit or delete books");
  }
  //...
```

Codice 5.12: HtmlUnit tests d'esempio per la Book Home view

Risulta importante sottolineare che, quando non useremo l'annotazione `@WithMockAdmin` a livello di classe[6], nel metodo di setup sarà necessario cancellare i cookie

5 Nella nostra applicazione l'unico utente che potrà autenticarsi sarà l'amministratore.
6 Useremo `@WithMockAdmin` a livello di classe nei test-case per le pagine web destinate ad uso e consumo dell'admin.

dell'istanza dell'*HtmlUnit* `WebClient`: in tal modo garantiremo che il risultato dei test non sia influenzato dal loro ordine di esecuzione. Se non prendessimo quest'accorgimento, dopo aver eseguito un test per l'amministratore, i successivi test privi di annotazione, quindi per l'utente anonimo, verrebbero comunque eseguiti con l'admin: questo perché l'istanza del `WebClient` avrà memoria del cookie di sessione.
Alternativamente, avremmo anche potuto risolvere il problema specificando a livello di classe l'annotazione `@WithAnonymousUser`; tuttavia, poiché risulta più chiaro cosa si stia facendo e perché, abbiamo preferito prediligere il primo approccio.

Infine, spieghiamo per quale motivo, sempre nel metodo di setup, andremo a modificare l'*handler* usato dal web client per gestire problemi legati alle risorse CSS.
Quando eseguiremo i nostri *HtmlUnit* tests, il `DefaultCssErrorHandler` ci segnalerà più volte, attraverso un messaggio di warning stampato sulla console, la presenza di un *CSS Error* in `/webjars/bootstrap/4.5.2/css/bootstrap.min.css`. Nonostante l'avvertimento, tutto funzionerà correttamente; oltretutto il problema evidenziato non risulterà legato a un nostro file e, di conseguenza, potremo tranquillamente ignorarlo. Seguendo quando indicato nelle FAQ di *HtmlUnit* [20], silenzieremo questo warning usando come *CSS error handler* per il web client il `SilentCssErrorHandler`.

5.7.3 *Frammenti Thymeleaf*

Progettando l'interfaccia per la nostra applicazione web risulterà facile rendersi conto che molto codice HTML sarà comune a più pagine: ad esempio, in ognuna sarà sempre presente un header, un footer e una barra di navigazione. Per ridurre la duplicazione e strutturare meglio le nostre pagine, andremo allora a definire dei *Thymeleaf fragments*, cioè delle componenti HTML riutilizzabili [21].

Definire e usare frammenti Thymeleaf

All'interno di un template HTML, in un qualsiasi tag, potremo definire un frammento *Thymeleaf* con l'attributo `th:fragment` [21]. Ogni frammento che useremo sarà specificato in un proprio file HTML contenuto all'interno della cartella `src/main/resources/templates/shared`: ad esempio, nella sottocartella `fragment/layout/non-parametric` creeremo il file `footer.html` che andrà a contenere il footer delle nostre pagine web. Nel codice 5.13 possiamo visionare tale frammento.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
  <footer th:fragment="fragment" class="site-footer font-small fixed-bottom">
    <div class="text-center py-3">&copy; 2021 Copyright: Francesco Mucci</div>
  </footer>
</body>
</html>
```

Codice 5.13: footer.html: frammento per il footer della pagine web

Come vediamo nel codice 5.14, potremo rimpiazzare un qualsiasi tag presente in un template Thymeleaf con il frammento appena definito: per far ciò, ci basterà impostare l'attributo `th:replace` specificando il relative-path del file che contiene il frammento e il nome del frammento stesso [22].

```
<!-- ... -->
<footer th:replace="shared/fragment/layout/non-parametric/footer::fragment">
    Placeholder footer</footer>
<!-- ... -->
```

Codice 5.14: usare il frammento del footer

Alternativamente avremmo anche potuto richiedere l'inserimento del frammento o del suo contenuto nel corpo del tag impostando, rispettivamente, l'attributo `th:insert` o `th:include` [22].

Frammenti con parametri

Oltre ai frammenti normali, potremo anche crearne di parametrizzati: questi, esattamente come una funzione, saranno definiti specificando un set di parametri e ci permetteranno di ottenere degli stralci di codice HTML ancora più riusabili. Nel codice 5.15 vediamo, ad esempio, il frammento per il bottone di submit che useremo nei nostri form: questo sarà parametrizzato rispetto all'icona e al testo ad esso associati.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
<!-- ... -->
  <div th:fragment="fragment(btnIcon, btnText)">
    <button class="btn btn-block btn-outline-dark"
      type="submit" name="submit-button">
      <span>
        <i class="fas" th:classappend="${btnIcon}" aria-hidden="true"></i>
      </span>
    <span th:text="${btnText}">Placeholder for button text</span>
    </button>
  </div>
<!-- ... -->
</body>
</html>
```

Codice 5.15: formButton.html: frammento per il bottone submit dei form

A seconda del template in cui decideremo di usarlo, andremo a fornire diversi valori per i parametri; ad esempio, nel codice 5.16 e 5.17, vediamo a confronto l'inclusione del bottone di submit nel form di login e in quello per l'aggiunta di nuovi libri.

```
<form name="login-form" id="login-form" th:action="@{/login}" method="post">
  <!-- ... -->
  <div class="form-group pt-3"
    th:include="shared/fragment/parametric/formButton::fragment('fa-sign-in-alt', '
    Login')">Placeholder for button</div>
  <!-- ... -->
</form>
```

Codice 5.16: usare il formButton fragment nel login form

```
<form name="new-book-form" th:action="@{/book/add}" method="post"
  th:object="${bookData}">
  <!-- ... -->
```

```
4   <div class="form-group pt-3"
5     th:include="shared/fragment/parametric/formButton::fragment('fa-plus', 'Add
      book')">Placeholder for button</div>
6 </form>
```

Codice 5.17: usare il formButton fragment nell'add-book form

Frammento per il layout comune

Come evidenziato nella sezione 5.3.1, tutte le nostre pagine web avranno lo stesso layout: nel codice 5.18 possiamo vedere il frammento in cui lo andremo a definire.

```
1 <!DOCTYPE html>
2 <html th:fragment="fragment(title, content)"
3   xmlns:th="http://www.thymeleaf.org" lang="en">
4 <head>
5   <title th:replace="${title}">Placeholder title</title>
6   <th:block th:include="shared/fragment/layout/non-parametric/head::fragment">
      Placeholder header<th:block>
7 </head>
8 <body>
9   <nav th:replace="shared/fragment/layout/non-parametric/navbar::fragment">
      Placeholder navigation bar</nav>
10  <section th:replace="${content}">Placeholder content</section>
11  <footer th:replace="shared/fragment/layout/non-parametric/footer::fragment">
      Placeholder footer</footer>
12 </body>
13 </html>
```

Codice 5.18: viewLayout.html: frammento per il layout

Tale frammento ci permetterà di riusare in tutte le nostre viste la stessa intestazione, la stessa barra di navigazione e lo stesso footer; inoltre, questo verrà parametrizzato rispetto alle variabili `title` e `content` che verranno usate per settare, rispettivamente, il titolo e i restanti elementi della vista.

Come possiamo osservare nel codice 5.19, nei template per le web views richiederemo di rimpiazzarne l'intero contenuto con il frammento di layout: forniremo come valore per i parametri, rispettivamente, i frammenti ottenuti dai blocchi `<title>` e `<section>` del template chiamante; questi verranno estratti usando l'espressione `~{this::tag}` [22].

```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org" xmlns:sec="http://www.thymeleaf.org"
3   th:replace="shared/viewLayout::fragment(~{this::title}, ~{this::section})">
4 <head>
5   <title>Book home view</title>
6 </head>
7 <body>
8   <section>
9     <!-- Contenuto della home view -->
10  </section>
11 </body>
12 </html>
```

Codice 5.19: bookHome.html: template per la Book home view

5.7.4 *Integrazione di Thymeleaf con Spring Security*

Per migliorare la user-experience andremo a nascondere agli utenti anonimi ogni bottone, link o altro contenuto normalmente riservato all'amministratore; per far ciò aggiungeremo al nostro POM la managed-dependency `org.thymeleaf.extras:thymeleaf-extras-springsecurity5` che ci metterà a disposizione il dialetto Thymeleaf `SpringSecurityDialect` [23]. Grazie ad esso saremo in grado di aggiungere ai tag dei nostri template l'attributo `sec:authorize`; questo ci permetterà di controllare il rendering del contenuto del tag sulla base della valutazione di un'espressione Spring da noi fornita. In particolare, andremo a usare [24] `sec:authorize="isAuthenticated()"` e `sec:authorize="hasRole('ROLE_ADMIN')"` per richiedere che il contenuto venga mostrato unicamente, nel primo caso, agli utenti autenticati e, nel secondo, all'amministratore.

5.7.5 *Errori di validazione nei form con Thymeleaf*

Nel caso in cui il processo di validazione degli input non dovesse avere successo, vorremmo che nei form delle pagine web venissero mostrati i corretti messaggi d'errore. In questo contesto verrà in nostro aiuto Thymeleaf [25]: potremo usare la funzione `hasErrors(...)` del `#fields` object per controllare la presenza di eventuali errori di validazione nel campo passato come argomento; inoltre, con l'attributo `th:errors` saremo in grado di rimpiazzare il corpo del dato tag con i messaggi d'errore associati al campo specificato.
Nel codice 5.20, dove definiamo il frammento per i form-inputs, possiamo osservare quanto abbiamo appena illustrato: risulta importante evidenziare come il nome del form-input, essendo parametro del frammento, dovrà essere preprocessato[7] prima di poter essere usato nell'attributo `th:errors` e come argomento della funzione `hasErrors(...)`.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
<!-- ... -->
<div
  th:fragment="fragment(inputLabel, inputPlaceholder, inputName)">
  <label th:id="|${inputName}-label|"
    th:text="${inputLabel}">Placeholder label</label>
  <input class="form-control form-control-lg text-center"
    required type="text" th:name="${inputName}"
    th:field="*{__${inputName}__}" th:placeholder="${inputPlaceholder}"/>
  <div class="alert alert-warning" th:id="|${inputName}-validation-error|"
    th:if="{#fields.hasErrors('__${inputName}__')}"
    th:errors="*{__${inputName}__}">Placeholder valid. error</div>
  </div>
<!-- ... -->
</body>
</html>
```

Codice 5.20: formInput.html: frammento per form inputs

7 Per richiedere a Thymeleaf di preprocessare un'espressione andremo ad usare la notazione `__${...}__` [22].

5.7.6 *Dialog-box con Thymeleaf e Bootstrap 4*

Come illustrato nella sezione 5.3.3, se l'utente è autenticato come amministratore, la tabella dei libri dovrà contenere, per ogni libro presente, un bottone `Delete` che consentirà l'apertura di un dialog-box per confermare l'operazione di cancellazione. Creeremo questa finestra di dialogo attraverso le *modal-component* di *Bootstrap 4*[8], componenti che risultano non visibili fintanto che non verrà premuto un particolare bottone d'innesco. Potremo definire tali bottoni usando l'attributo `data-toggle="modal"` e specificando con `data-target` l'identificativo della *modal-component* da aprire [26].
Come possiamo vedere nel codice 5.21, appoggiandoci a Thymeleaf genereremo un distinto dialog-box per ogni riga della tabella; questa è una soluzione pratica, ma non la più efficiente: tecnicamente avremmo anche potuto definire un'unica finestra di dialogo facendone variare il contenuto attraverso del codice JavaScript[9] [27].

```
<!-- ... -->
<div th:fragment="fragment" id="book-table-fragment">
  <div class="container my-5 pb-5">
    <table class="table table-striped table-bordered text-center"
      id="book-table" aria-label="Book table">
      <thead><!-- ... --></thead>
      <tbody>
        <tr th:each="book : ${books}">
          <!-- altre celle della tabella -->
          <td sec:authorize="hasRole('ADMIN')" class="align-middle" colspan="2">
            <button type="button" class="btn btn-link text-muted"
              th:id="|getDeleteBookDialogButton-${book.isbn}|" data-toggle="modal"
              th:data-target="|#deleteBookDialog-${book.isbn}|">
              <span>
                <i class="fas fa-trash-alt" aria-hidden="true"></i>
              </span>
              <span>Delete</span>
            </button>
            <div class="modal fade" th:id="|deleteBookDialog-${book.isbn}|">
              <div class="modal-dialog modal-dialog-centered">
                <!-- contenuto del dialog-box -->
              </div>
            </div>
          </td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
<!-- ... -->
```

Codice 5.21: bookTable.html: frammento per la tabelle dei libri

8 Per una trattazione dettagliata riguardo alle *modal-component* si rimanda alla documentazione ufficiale di *Bootstrap 4*.

9 In tal caso sarebbe stato necessario aggiungere un attributo `data-*` nei bottoni d'innesco al fine di salvare i dati del libro presente nella particolare riga.

5.7.7 *Custom login con remember-me*

Nella sezione 5.3.2 abbiamo specificato che il form di login della *Book home view* dovrà essere dotato di un remember-me checkbox; tuttavia, nella configurazione di *Spring Security*, definita nel ramo `web-base` e trattata nella sezione 5.5.2, non abbiamo impostato la possibilità di autenticarsi con l'opzione remember-me. Questa scelta non è stata casuale; se avessimo aggiunto subito tale funzionalità, non saremmo stati in grado di verificarne la corretta configurazione: difatti, nel test-case `MyWebSecurityConfigurationTest`, usando `MockMvc`, non avremmo avuto modo di accedere al remember-me cookie. Abbiamo, dunque, rimandato l'impostazione dell'autenticazione con remember-me al momento dell'implementazione del template per la *Book home view*: questo perché, nei test per tale vista, potremo usare il web client di *HtmlUnit* per accedere facilmente ai cookie.

Nel codice 5.22 possiamo vedere il test in cui andremo a verificare che la scadenza del remember-me cookie sia impostata correttamente. Si noti che viene richiesto `"remember-me"` come nome per il checkbox input, scelta dettata da *Spring Security*: facendo diversamente quest'ultimo non sarebbe in grado di eseguire correttamente il processo di autenticazione [16].

```
//...
@Test
public void test_rememberMe_expiresAfter30Min() throws Exception {
  when(bookWebController.getBookHomeView()).thenReturn(VIEW_BOOK_HOME);
  HtmlPage bookHomeView = webClient.getPage(URI_BOOK_HOME);
  HtmlForm loginForm = bookHomeView.getFormByName("login-form");
  loginForm.getInputByName("username").setValueAttribute(VALID_USER_NAME);
  loginForm.getInputByName("password").setValueAttribute(VALID_PASSWORD);
  loginForm.getInputByName("remember-me").click();
  loginForm.getButtonByName("submit-button").click();
  Date expires = webClient.getCookieManager()
    .getCookie("spring-bookshelf-remember-me").getExpires();
  Calendar expectedExpires = Calendar.getInstance();
  expectedExpires.add(Calendar.MINUTE, 30);
  assertThat(expires.toInstant())
    .isCloseTo(expectedExpires.toInstant(), within(1, ChronoUnit.SECONDS));
}
//...
```

Codice 5.22: testare la scadenza del remember-me cookie

Come possiamo osservare nel codice 5.23, per abilitare e configurare l'autenticazione con remember-me sarà sufficiente aggiungere poche linee di codice all'implementazione del metodo `configure(HttpSecurity http)` della `MyWebSecurityConfiguration` [16].

```
//...
public class MyWebSecurityConfiguration extends WebSecurityConfigurerAdapter {
  //...
  private static final String REMEMBER_ME_TOKEN = "spring-bookshelf-remember-me";
  private static final int TOKEN_VALIDITY_SECONDS = 1800;
  //...
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
```

```
10        //...
11        .and().rememberMe()
12          .rememberMeCookieName(REMEMBER_ME_TOKEN)
13          .tokenValiditySeconds(TOKEN_VALIDITY_SECONDS);
14    }
15  }
```

Codice 5.23: configurare il remember-me

6

INTEGRATION E E2E TESTS

In questo capitolo illustreremo i test d'integrazione ed E2E implementati, soffermandoci in particolar modo sul'uso del *Page Object Pattern* per l'interazione con la UI delle nostre pagine web.

Il codice qui presentato è stato completamente sviluppato nel ramo `integration` seguendo l'approccio descritto nella sezione 1.4.

6.1 MYBOOKSERVICE IT

Il primo ramo che andremo a fondere in `integration`, che verrà creato a partire da `repository`, sarà `service`; una volta completata la fusione, andremo ad eliminare la `TemporaryBookRepository` proveniente da `service` e aggiungeremo, nella classe `MyBookServiceIT`, i test d'integrazione per verificare che il *Service* e la *Repository* interagiscano correttamente. In tale test-case useremo l'annotazione `@DataMongoTest` per avviare le componenti fondamentali per il *Persistence layer* e `@Import(MyBookService.class)` per richiedere a Spring di caricare anche l'implementazione del `BookService`. Come da prassi per i test d'integrazione, andremo a testare solamente i casi positivi [2].

6.2 MYBOOKWEBCONTROLLER IT

Una volta completata l'integrazione del `service`, andremo a fondere anche il ramo `web-controller` in `integration`; conclusa quest'operazione, elimineremo il `TemporaryBookService` e aggiungeremo, nella classe `MyBookWebControllerIT`, i test d'integrazione per verificare che il *Web Controller* e le restanti componenti interagiscano correttamente. In questi test avremo bisogno di tutti i layer della nostra applicazione e, di conseguenza, andremo ad usare l'annotazione `@SpringBootTest`. Risulta importante evidenziare che non andremo ad eseguire i test con un vero web server, ma useremo un mock web environment; per fare ciò ci basterà non specificare alcun valore per l'attributo `webEnvironment` di `@SpringBootTest` [28]: di default verrà dunque usato `WebEnvironment.MOCK` e, conseguentemente, Spring non avvierà l'embedded server. A questo punto, per testare gli end-points esposti dal *MyBookWebController* configureremo il `MockMvc` usando l'annotazione `@AutoConfigureMockMvc` e ne richiederemo l'iniezione con `@Autowired` [28]. Andremo a testare unicamente i casi positivi in cui sarà presente un'interazione tra il *Web Controller* e il *Service*.

6.3 WEB VIEWS ITS

Per concludere il processo di integrazione andremo a fondere il ramo `web-views` in `integration`; fatto ciò, dopo aver eliminato il `TemporaryBookWebController`, definiremo per ogni pagina web un distinto IT test-case. In questi test avremo bisogno di tutte le nostre componenti; inoltre, vorremo che venissero eseguiti in un vero web environment: per soddisfare entrambe le richieste andremo a usare l'annotazione `@SpringBootTest` specificando, come valore per l'attributo `webEnvironment`, `WebEnvironment.RANDOM_PORT`. Il numero di porta casuale, sul quale l'embedded server avviato da Spring sarà in ascolto, potrà allora essere iniettato a livello di campo usando l'annotazione `@LocalServerPort` [28]. Questa volta, oltre ai positivi, andremo a verificare anche il comportamento atteso in alcuni casi d'eccezione di particolare interesse.

Diversamente da quanto fatto nei test d'unità, per simulare l'interazione dell'utente con le nostre pagine web ci appoggeremo a un *Selenium* `WebDriver`; in particolare, vorremmo usare l'`HtmlUnitDriver`, un'implementazione di `WebDriver` per l'*HtmlUnit* headless browser. Come già illustrato nella sezione 5.7.2, vorremmo che nel web client venisse usato un `SilentCssErrorHandler`: dal momento che il metodo `getWebClient` di `HtmlUnitDriver` è `protected`, per impostarlo saremo costretti a definire un nostro `SilentHtmlUnitDriver` che, come possiamo vedere nel codice 6.1, estende la classe `HtmlUnitDriver`.

```
1 //...
2 public class SilentHtmlUnitDriver extends HtmlUnitDriver {
3   public SilentHtmlUnitDriver() {
4     super();
5     this.getWebClient().setCssErrorHandler(new SilentCssErrorHandler());
6   }
7 }
```

Codice 6.1: classe SilentHtmlUnitDriver

Un'altra differenza rispetto ai test d'unità per le pagine web risiede nel fatto che l'interazione con la UI non verrà realizzata facendo riferimento agli elementi HTML direttamente nei test, ma bensì affidandosi a un approccio più efficiente: andremo a seguire il *Page Object Pattern* [29].

6.3.1 *Page Object Pattern*

Il *Page Object Pattern* è un Design Pattern usato nel contesto dei test automatici per ridurre la duplicazione di codice e per migliorare la manutenibilità dei test stessi [30]. L'idea alla base di tale pattern è relativamente semplice: andremo a definire dei *Page Objects*, classi che avvolgono una pagina o un frammento HTML e usate nei test per manipolare gli elementi della pagina stessa [29]. Il vantaggio di quest'approccio risiede nel fatto che, se cambieremo la struttura di una pagina, non dovremo cambiare il codice di test, ma solamente quello del corrispettivo page object [30]; inoltre, nascondendo i dettagli riguardo agli elementi HTML, renderemo i test più leggibili [29].

I *Page Object* dovrebbero contenere la rappresentazione di una pagina web ed esporre, attraverso i propri metodi pubblici, i servizi da questa offerti; inoltre, se l'uso di un particolare servizio porterà su una nuova pagina, allora il metodo associato dovrà restituire il corrispettivo page object [30]: così facendo saremo in grado di specificare in

modo realistico il flusso di navigazione della nostra applicazione.
Risulta anche importante evidenziare che un page object non dovrà obbligatoriamente rappresentare tutti gli elementi della particolare pagina, ma solamente quelli necessari per i test [30]. Sarà anche possibile definire dei *Page Component Objects* per modellare frammenti ricorrenti in più pagine e, di conseguenza, migliorare la manutenibilità e ridurre la duplicazione di codice [30]. Nella figura 1 possiamo osservare il diagramma delle classi dei page objects che andremo a definire.

```
//...
public abstract class APageObject {
  protected WebDriver webDriver;
  public APageObject(WebDriver webDriver) {
    this.webDriver = webDriver;
    PageFactory.initElements(webDriver, this);
  }
}
```

Codice 6.2: APageObject: superclasse comune a tutti i nostri page objects

Come possiamo vedere dal codice 6.2 e 6.3, per localizzare gli elementi HTML e inizializzare i corrispettivi `WebElement`, oltre a sfruttare il metodo `findElement(...)` del `WebDriver`, potremo usare l'annotazione a livello di campo `@FindBy` in combinazione con il metodo statico `initElements(...)` della classe `PageFactory`[1] [31].

```
//...
public class MyPage extends APageObject {
  @FindBy(tagName = "header") private WebElement header;
  @FindBy(linkText = "Mucci's bookshelf") private WebElement homeLink;
  @FindBy(linkText = "Show book list") private WebElement bookListLink;
  @FindBy(linkText = "Search book by isbn") private WebElement searchByIsbnLink;
  @FindBy(linkText = "Search book by title") private WebElement searchByTitleLink;
  @FindBy(linkText = "Add new book") private WebElement addNewBookLink;
  @FindBy(name = "logout-button") private WebElement logoutButton;
  public MyPage(WebDriver webDriver) {
    super(webDriver);
  }
  public MyPage(WebDriver webDriver, String expectedTitle) {
    super(webDriver);
    String actualTitle = getPageTitle();
    if (!actualTitle.equals(expectedTitle))
      throw new IllegalStateException(
        "Expected page: " + expectedTitle + ", Actual page: " + actualTitle);
  }
  public String getPageTitle() {
    return webDriver.getTitle();
  }
  //...
}
```

Codice 6.3: MyPage: classe che rappresenta il layout comune delle pagine web

1 Se il metodo `PageFactory.initElements(...)` non venisse specificato nel costruttore, dovremmo usarlo, al posto di quest'ultimo, per istanziare i vari page objects [31].

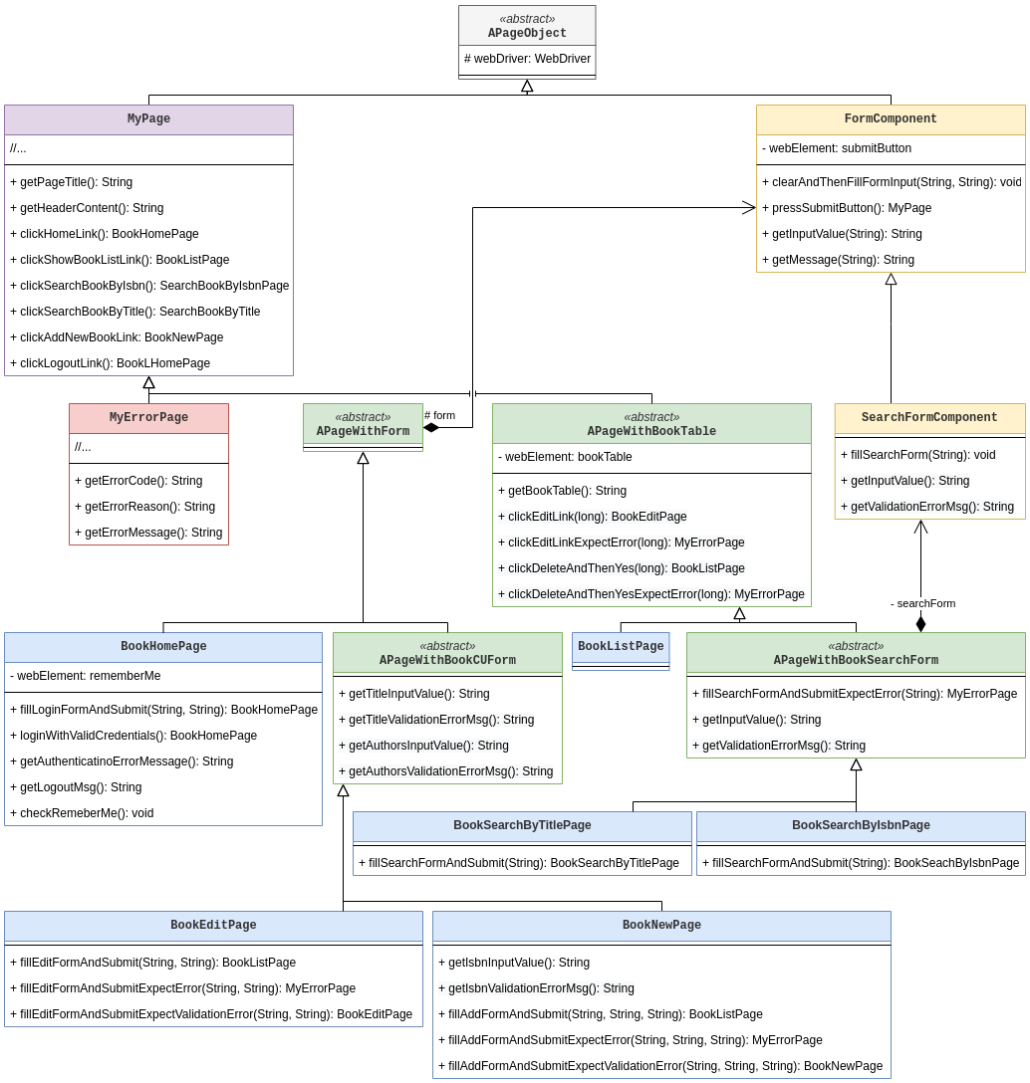

Figura 1: class diagram dei page objects

Sempre osservando il codice 6.3, possiamo notare come il secondo costruttore ci permetterà di istanziare oggetti `MyPage` verificando che il `WebDriver` si trovi effettivamente sulla pagina attesa: se il titolo della pagina non sarà quello atteso, verrà lanciata una `IllegalStateException`. Alternativamente, avremmo anche potuto decidere di esprimere le nostre aspettative attraverso un'assertion; tuttavia, ci sono opinioni contrastanti riguardo al fatto che un page object le debba includere o meno: da una parte, i fautori dell'inclusione sostengono che questa permetta di ridurre la duplicazione di codice nei test [29], dall'altra, molte fonti autorevoli ritengono che la responsabilità di specificare le aspettative riguardo al comportamento atteso debba essere esclusiva dei test [29] [30].

Infine, è importante specificare come andremo a modellare i servizi che restituiscono viste diverse in base alla particolare interazione con l'utente. Secondo quanto illustrato sulla documentazione di *Selenium* [30], per ogni vista restituita da un determinato servizio dovremo definire un particolare metodo nel relativo page object. Seguendo tale approccio, per rappresentare, ad esempio, la ricerca per titolo, un oggetto della classe `BookSearchByTitlePage` dovrà esporre due metodi distinti: `fillSearchFormAndSubmit(String inputValue)`, che restituisce una `BookSearchByTitlePage`, e `fillSearchFormAndSubmitExpectingError(String inputValue)`, che restitusce una `MyErrorPage`. Sarà poi nostro compito, nei metodi di test, invocare il metodo corretto a seconda della particolare situazione.

6.4 E2E TESTS

Dopo aver integrato tutte le componenti, aggiungeremo, sempre nel ramo `integration`, i test E2E. In questi, per simulare l'interazione dell'utente con le nostre pagine, ci appoggeremo ancora una volta a *Selenium*, ma, diversamente da quanto fatto in precedenza, andremo a usare come `WebDriver` un'istanza di `ChromeDriver` che ci permetterà di comunicare con la nostra applicazione attraverso un vero *Google Chrome* web browser. È chiaro che, per manipolare gli elementi della pagine web, potremo riusare i *Page Objects* precedentemente definiti.

Nei test verificheremo i casi positivi e alcune altre situazioni interessanti; tra questi risulta importante soffermarsi sul metodo di test per l'autenticazione con remember-me. Per prima cosa è necessario comprendere che, senza un'ulteriore configurazione, il `ChromeDriver` non sarà in grado di ricordarsi dell'utente precedentemente loggato con remember-me; questo succederà perché, di default, verrà creata una *user data directory* temporanea per ogni singola sessione [32]. Per ovviare a questa situazione dovremo indicare al `ChromeDriver` una directory da usare a tale scopo; per far ciò, nel metodo di setup, andremo a configurare la `ChromeDriver` session [32]. Anzitutto, creeremo un'istanza di `ChromeOptions`, che ci fornirà diversi metodi per impostare le opzioni di configurazione; in particolare, ci permetterà di specificare il path per la *user data directory* aggiungedo l'argomento `"user-data-dir=..."` [32]. Per completare la configurazione passeremo l'oggetto `ChromeOptions` al costruttore del `ChromeDriver` [32].
Nel codice 6.4 possiamo osservare il metodo di setup e i test per il remember-me definiti.

```
1 //...
2 public class SpringBookshelfApplicationE2E {
3   //...
4   private static final String BASE_URL = "http://localhost:" + PORT_NUMBER;
5   //...
```

```
6   private WebDriver webDriver;
7   private ChromeOptions options;
8   private BookHomePage bookHomePage;
9   //...
10  @Before
11  public void setup() {
12    //...
13    options = new ChromeOptions();
14    options.addArguments("start-maximized",
15      "user-data-dir=target/web-driver-data/chrome-user-data");
16    webDriver = new ChromeDriver(options);
17    webDriver.get("http://localhost:" + PORT_NUMBER);
18    bookHomePage = new BookHomePage(webDriver);
19  }
20  @After
21  public void teardown() {
22    webDriver.manage().deleteCookieNamed(REMEMBER_ME_TOKEN);
23    webDriver.quit();
24    //...
25  }
26  //...
27  @Test
28  public void test_login_whenNotRememberMe() {
29    bookHomePage.fillLoginFormAndSubmit(VALID_USER_NAME, VALID_PASSWORD);
30    assertThat(bookHomePage.getHeaderContent()).contains("Welcome back");
31    webDriver.quit();
32    webDriver = new ChromeDriver(options);
33    webDriver.get(BASE_URL);
34    bookHomePage = new BookHomePage(webDriver);
35    assertThat(bookHomePage.getHeaderContent()).contains("Welcome to my book
      library");
36  }
37  @Test
38  public void test_login_whenRememberMe() {
39    bookHomePage.checkRemeberMe();
40    bookHomePage.fillLoginFormAndSubmit(VALID_USER_NAME, VALID_PASSWORD);
41    assertThat(bookHomePage.getHeaderContent()).contains("Welcome back");
42    webDriver.quit();
43    webDriver = new ChromeDriver(options);
44    webDriver.get(BASE_URL);
45    bookHomePage = new BookHomePage(webDriver);
46    assertThat(bookHomePage.getHeaderContent()).contains("Welcome back");
47  }
48  //...
49 }
```

Codice 6.4: E2E: remember-me test

Concludiamo la nostra trattazione sui test E2E evidenziando che, per eseguire operazioni di setup, accederemo in maniera diretta al database usando un'istanza di `MongoClient`. Come possiamo osservare dal codice 6.5, tale istanza verrà creata attraverso la *factory class* `MongoClients` [3] e verrà usata per ripulire il DB prima dell'esecuzione di ogni singolo test. Risulta interessante notare come, sfruttando il meccanismo di

value injection offerto da Spring, sarà possibile leggere il numero di porta e il nome del database direttamente dalle proprietà definite nel file `application.properties`; in tal caso, sarà necessario eseguire i test usando lo `SpringJUnit4ClassRunner` e specificando, attraverso l'annotazione `@TestPropertySource`, il file di proprietà da usare [12]. Per aggiungere libri alla test fixture ci appoggeremo al metodo privato `setupAddingBookToDatabase` che permetterà la creazione e l'inserimento, attraverso l'oggetto `MongoClient`, di un nuovo documento nel database.

```
//...
@RunWith(SpringRunner.class)
@TestPropertySource("classpath:application.properties")
public class SpringBookshelfApplicationE2E {
  //...
  @Value("${spring.data.mongodb.port:27017}")
  private int db_port_number;
  @Value("${spring.data.mongodb.database:test}")
  private String db_name;
  private MongoClient mongoClient;
  //...
  @Before
  public void setup() {
    mongoClient = MongoClients.create("mongodb://localhost:" + db_port_number);
    mongoClient.getDatabase(db_name).drop();
    setupAddingBookToDatabase(VALID_ISBN13, TITLE, AUTHORS_LIST);
    setupAddingBookToDatabase(VALID_ISBN13_2, TITLE_2, AUTHORS_LIST_2);
    //...
  }
  @After
  public void teardown() {
    //...
    mongoClient.close();
  }
  //...
  private void setupAddingBookToDatabase(long isbn, String title, List<String>
    authors) {
    mongoClient.getDatabase(db_name).getCollection(DB_COLLECTION)
      .insertOne(new Document()
        .append("_id", isbn)
        .append("title", title)
        .append("authors", authors));
  }
}
```

Codice 6.5: E2E: setup usando MongoClient

7

ESEGUIRE LA WEB APPLICATION

In questo capitolo conclusivo andremo a vedere come eseguire il processo di building e come lanciare da command-line la nostra web application.

7.1 ESEGUIRE IL PROCESSO DI BUILDING

Per creare il *FatJar* dovremo lanciare, attraverso lo script `mvnw`, una maven build eseguita almeno fino alla fase `package`. Avendo già testato l'applicazione sul CI server, potremo decidere di saltare i test; dunque, nella root directory della git repository, andremo ad eseguire il seguente comando:

```
./mvnw -f spring-bookshelf -Dskip.unit-tests=true clean package
```

Se, oltre a creare il *FatJar*, volessimo anche rieseguire tutti i test riproducendo quanto fatto sul server di integrazione continua, potremo lanciare le seguenti maven build:

```
./mvnw -f spring-bookshelf clean verify -Pjacoco &&
./mvnw -f spring-bookshelf test-compile -Ppitest &&
./mvnw -f spring-bookshelf verify -Pit-tests &&
./mvnw -f spring-bookshelf verify -Pe2e-tests
```

7.2 ESEGUIRE LA WEB APPLICATION DA COMMAND-LINE

La nostra applicazione si aspetterà di trovare un *MongoDB* server in ascolto sulla porta 27017; di conseguenza, prima di lanciarla, eseguiremo tale database in un Docker container attraverso il comando:

```
docker run -p 27017:27017 --name mongoDB mongo:4.4.3
```

A questo punto, ci basterà eseguire il seguente comando nella root directory della git repository:

```
java -jar spring-bookshelf/target/spring-bookshelf-0.0.1-SNAPSHOT.jar
```

In tal caso l'applicazione sarà configurata con alcuni valori di default: in particolare, verrà usato *"test"* come nome per il database e *"Admin"* e *"Password"*come credenziali di autenticazione per l'amministratore. Per modificare questi defaults potremo impostare, rispettivamente, le proprietà `spring.data.mongodb.database` [28], `admin.username` e `admin.password`; ad esempio, l'applicazione potrà essere lanciata con il comando:

```
java \
-Dspring.data.mongodb.database="spring-bookshelf-db" \
-Dadmin.username="Homer" -Dadmin.password="D'Oh!" \
-jar spring-bookshelf/target/spring-bookshelf-0.0.1-SNAPSHOT.jar
```

BIBLIOGRAFIA

[1] International ISBN Agency: *What is an ISBN?* https://www.isbn-international.org/content/what-isbn, consultato in data 2022-01-26. (Cited on pages 2 and 5.)

[2] Lorenzo Bettini: *Test-Driven Development, Build Automation, Continuous Integration with Java, Eclipse and friends*. Leanpub, 2021. https://leanpub.com/tdd-buildautomation-ci. (Cited on pages 6 and 41.)

[3] Felipe Gutierrez: *Pro Spring Boot 2: An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices*. Apress, 2019. (Cited on pages 6, 10, 11, 25, 26, and 46.)

[4] Cay S. Horstmann: *Core Java Volume I – Fundamentals (Eleventh Edition)*. Addison-Wesley, 2019. (Cited on page 7.)

[5] Francesco Marchioni: *MongoDB for Java Developers: Design, build, and deliver efficient Java applications using the most advanced NoSQL database*. Packt Publishing, 2015. (Cited on page 11.)

[6] Pivotal Software Inc.: *Spring Data MongoDB 3.1.8 API*. https://docs.spring.io/spring-data/mongodb/docs/3.1.8/api/, consultato in data 2022-02-03. (Cited on pages 11 and 12.)

[7] Flapdoodle OSS: *Flapdoolde project on GitHub*. https://github.com/flapdoodle-oss/de.flapdoodle.embed.mongo, consultato in data 2022-02-02. (Cited on page 11.)

[8] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee e Randy Stafford: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003. (Cited on pages 16 and 17.)

[9] Iuliana Cosmina, Rob Harrop, Chris Schaefer e Clarence Ho: *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools - Fifth edition*. Apress, 2017. (Cited on pages 17, 24, 26, and 27.)

[10] Red Hat Inc.: *Jakarta Bean Validation 2.0*. https://beanvalidation.org/2.0/, consultato in data 2022-02-14. (Cited on page 17.)

[11] Red Hat Inc.: *Hibernate Validator 6.1.7.Final*. https://docs.jboss.org/hibernate/validator/6.1/api/, consultato in data 2022-02-14. (Cited on page 17.)

[12] Iuliana Cosmina: *Pivotal Certified Professional Core Spring 5 Developer Exam: A Study Guide Using Spring Framework 5 - Second Edition*. Apress, 2020. (Cited on pages 18, 24, 25, 26, 31, 32, and 47.)

[13] Marten Deinum: *Spring Boot 2 Recipes: A Problem-Solution Approach*. Apress, 2018. (Cited on page 23.)

[14] Andrew Shcherbakov: *Interface Driven Controllers in Spring*. https://www.baeldung.com/spring-interface-driven-controllers, consultato in data 2022-02-24. (Cited on page 23.)

[15] *What's New in Spring Framework 5.x*. https://github.com/spring-projects/spring-framework/wiki/What's-New-in-Spring-Framework-5.x, consultato in data 2022-02-24. (Cited on page 23.)

[16] Carlo Scarioni e Massimo Nardone: *Pro Spring Security: Securing Spring Framework 5 and Boot 2-based Java Applications - Second Edition*. Apress, 2019. (Cited on pages 25, 26, and 39.)

[17] OWASP CheatSheets Series Team: *Cross-Site Request Forgery Prevention Cheat Sheet*. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html, consultato in data 2022-02-21. (Cited on page 26.)

[18] Ben Alex, Luke Taylor, Rob Winch, Gunnar Hillert, Joe Grandja, Jay Bryant, Eddú Meléndez, Josh Cummings e Dave Syer: *Spring Security Reference - Version 5.4.6*. https://docs.spring.io/spring-security/site/docs/5.4.6/reference/pdf/spring-security-reference.pdf. (Cited on pages 26, 27, and 28.)

[19] Pivotal Software Inc.: *Spring Framework Documentation - Version 5.3.6*. https://docs.spring.io/spring-framework/docs/5.3.6/reference/html/, consultato in data 2022-02-24. (Cited on page 32.)

[20] Gargoyle Software Inc.: *HtmlUnit: Frequently Asked Questions*. https://htmlunit.sourceforge.io/faq.html, consultato in data 2022-02-26. (Cited on page 34.)

[21] Wim Deblauwe: *Taming Thymeleaf: Practical Guide to building a web application with Spring Boot and Thymeleaf - Version 1.1.1*. Leanpub, 2021. https://leanpub.com/taming-thymeleaf. (Cited on page 34.)

[22] The Thymeleaf Team: *Tutorial: Using Thymeleaf*. https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html, Document version: 20181029 - 29 October 2018. (Cited on pages 34, 35, 36, and 37.)

[23] The Thymeleaf Team: *Thymeleaf: Spring Security integration modules*. https://github.com/thymeleaf/thymeleaf-extras-springsecurity, consultato in data 2022-03-02. (Cited on page 37.)

[24] José Miguel Samper: *Thymeleaf + Spring Security integration basics*. https://www.thymeleaf.org/doc/articles/springsecurity.html, consultato in data 2022-02-28. (Cited on page 37.)

[25] The Thymeleaf Team: *Tutorial: Thymeleaf + Spring*. https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html, Document version: 20181029 - 29 October 2018. (Cited on page 37.)

[26] Jörg Krause: *Introducing Bootstrap 4: Create Powerful Web Applications Using Bootstrap 4.5 - Second Edition*. Apress, 2020. (Cited on page 38.)

[27] Bootstrap team: *Bootstrap v4.6 documentation: Modal*. https://getbootstrap.com/docs/4.6/components/modal/, consultato in data 2022-03-04. (Cited on page 38.)

[28] Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons, Vedran Pavić, Jay Bryant, Madhura Bhave, Eddú Meléndez e Scott Frederick: *Spring Boot v2.4.5 Reference Documentation*. https://docs.spring.io/spring-boot/docs/2.4.5/reference/pdf/spring-boot-reference.pdf, consultato in data 2022-03-05. (Cited on pages 41, 42, and 48.)

[29] Martin Fowler: *PageObject*. https://martinfowler.com/bliki/PageObject.html, 10 Settembre 2013. (Cited on pages 42 and 45.)

[30] *Selenium documentation: Page object models*. https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/, consultato in data 2022-03-06. (Cited on pages 42, 43, and 45.)

[31] Chaitanya Pujari: *Page Object Model and Page Factory in Selenium*. https://www.browserstack.com/guide/page-object-model-in-selenium, 16 Maggio 2021. (Cited on page 43.)

[32] *ChromeDriver: Capabilities and ChromeOptions*. https://sites.google.com/chromium.org/driver/capabilities, consultato in data 2022-03-17. (Cited on page 45.)