Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Laurea Magistrale in Ingegneria Informatica

BookingApp

Applicazione desktop Java per la gestione di prenotazioni sviluppata con TDD, build automation e Continuous Integration

Corso di Advanced Programming Techniques

Autore
Marco Pagliocca

Docente
Prof. Lorenzo Bettini

Anno Accademico 2023/2024

Introduzione

BookingApp è il nome del progetto per il corso di Advanced Programming Techniques a 6 CFU, nonché una semplice applicazione desktop scritta in **Java 11** e sviluppata con le pratiche di *Test Driven Development*, mediante **JUnit 4** e **5**, di *Build Automation*, tramite **Maven**, e di *Continuous Integration*, grazie al supporto di **GitHub Actions**. Particolarità del progetto è la compatibilità sia con un *database NoSQL*, quale **MongoDB**, sia con un *database relazionale*, quale **PostgreSQL**, quest'ultimo accompagnato dal provider JPA **Hibernate**; di conseguenza sono state usate le *transazioni* su Java per entrambi i DBMS. Infine l'applicazione presenta anche un'interfaccia grafica implementata con **Swing**.

Il progetto ha una *repository Git* pubblica disponibile su **GitHub** alla pagina marcopaglio/BookingApp.

Indice

Capitolo 1

BookingApp

L'applicazione proposta è un sistema per amministrare generiche prenotazioni di tipo giornaliero utilizzabile da chiunque abbia dei clienti in modo da mantenere un archivio di prenotazioni. L'interfaccia grafica sarebbe dovuta essere come in Figura 1.1, ovvero una singola finestra che comprende:

- Un box contenente la lista delle prenotazioni; se non ci sono prenotazioni il box è vuoto.
- Un box contenente la lista dei clienti; se non ci sono clienti registrati il box è vuoto, tuttavia se un cliente registrato non ha prenotazioni, appare comunque nella finestra.
- Due campi di testo affiancati ("Nome Cognome", "Data Prenotazione").
- Un bottone "Prenota" sopra le finestre, disabilitato di default e abilitato qualora i campi di testo siano compilati con valori adeguati. Quando cliccato:
  - Aggiunge la prenotazione alla lista e ripulisce i form.
  - Se la prenotazione viene inserita per un nuovo cliente, questo viene aggiunto alla lista dei clienti.
  - Se esiste già una prenotazione per la stessa data, viene restituito un messaggio di errore.
- Un bottone "Elimina Prenotazione" sotto la finestra delle prenotazioni, disabilitato di default e abilitato qualora una prenotazione venga selezionata dalla lista. Quando cliccato rimuove la prenotazione dalla lista, a meno che non fosse già stata eliminata, allora restituisce un messaggio di errore.

- Un bottone "Elimina Cliente" sotto la finestra dei clienti, disabilitato di default e abilitato qualora un cliente venga selezionato dalla lista. Quando cliccato rimuove il cliente e tutte le sue prenotazioni dalle liste, a meno che il cliente non fosse già stato eliminato, allora restituisce un messaggio di errore.

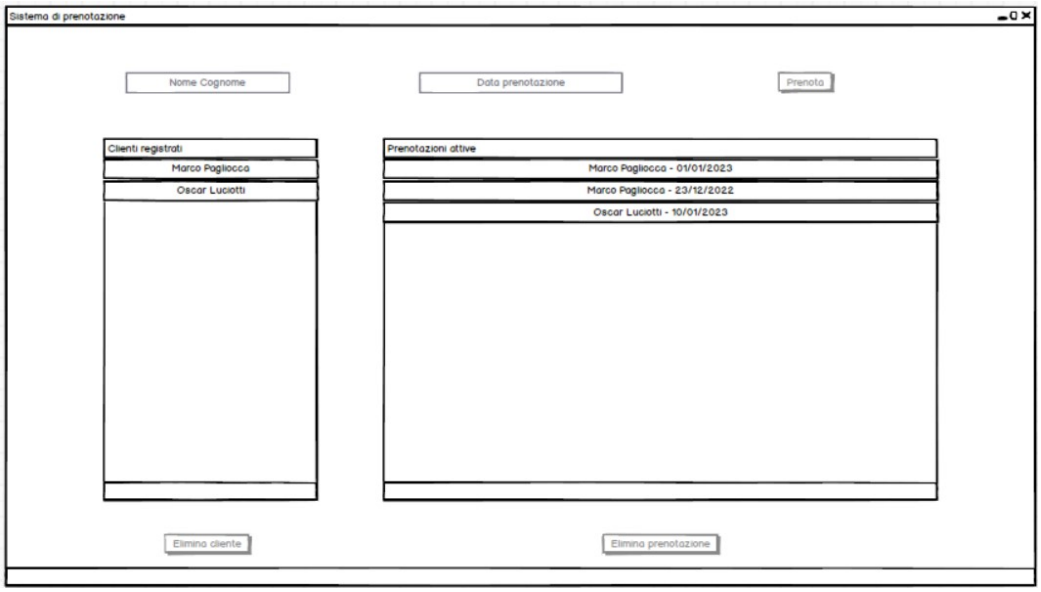

Figura 1.1: Mock-up dell'applicazione appena aperta.

1.1 Modifiche e risultato finale

In corso d'opera ci sono state alcune modifiche e aggiunte che non erano state (correttamente) pensate durante la progettazione.

Interfaccia grafica La GUI realizzata, in Figura 1.2, è simile al mock-up della proposta di progetto, se non per alcune modifiche di seguito descritte:

- Il form "Nome Cognome" è stato suddiviso in due campi, in modo che eventuali secondi nomi o cognomi composti da più parole (e.g. De Lucia) possano essere utilizzati senza problemi.
- Il form "Data Prenotazione" è adesso costituito da "Year", "Month" e "Day" per non avere ambiguità sul formato della data.
- Il bottone "Prenota" è stato sostituito dai bottoni:

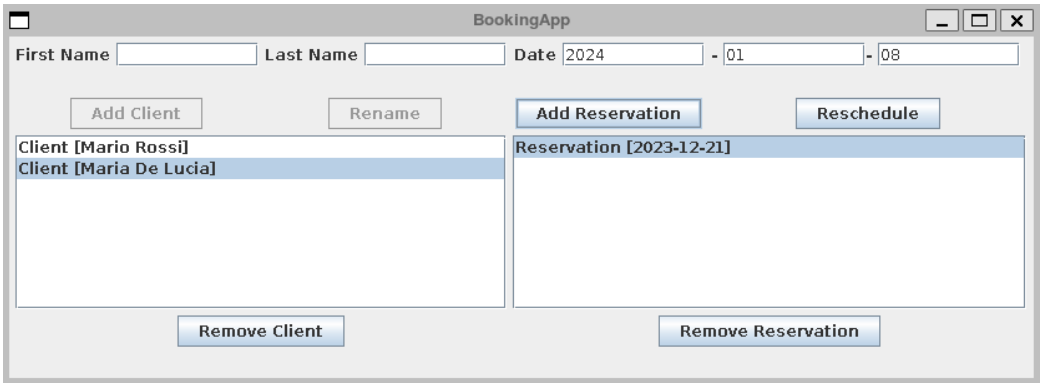

Figura 1.2: Interfaccia grafica di BookingApp.

  - "Add Client" abilitato qualora i campi di "First Name" e "Last Name" siano compilati con caratteri non vuoti. Quando cliccato aggiunge il cliente alla lista e ripulisce i form; se un cliente omonimo esiste già oppure nei form sono stati utilizzati caratteri non alfabetici, allora viene mostrato un messaggio di errore.
  - "Add Reservation" abilitato qualora i campi di "Year", "Month" e "Day" sono compilati da caratteri non vuoti e un cliente venga selezionato dalla lista. Quando cliccato aggiunge la prenotazione alla lista e ripulisce i form; se esiste già una prenotazione per la stessa data oppure nei form sono stati utilizzati caratteri non numerici, allora viene mostrato un messaggio di errore.

- È stato aggiunto il bottone "Rename", disabilitato di default e abilitato qualora i campi "First Name" e "Last Name" siano compilati con caratteri non vuoti e un cliente venga selezionato dalla lista. Quando cliccato:

  - Rinomina il cliente selezionato coi nomi specificati e ripulisce i form.
  - Se i nuovi nomi non sono cambiati o contengono caratteri non alfabetici viene mostrato un messaggio di errore.
  - Se esiste già un cliente omonimo viene mostrato un messaggio di errore.

- È stato aggiunto il bottone "Reschedule" disabilitato di default e abilitato qualora i campi "Year", "Month" e "Day" siano compilati con caratteri non vuoti e una prenotazione venga selezionata dalla lista. Quando cliccato:

- Riprogramma la prenotazione selezionata per la nuova data e ripulisce i form.
- Se la nuova data non è cambiata o contiene caratteri non numerici viene restituito un messaggio di errore.
- Se esiste già una prenotazione per la nuova data viene restituito un messaggio di errore.

Casi d'uso In via definitiva, all'amministratore delle prenotazioni vengono garantiti i casi d'uso di Figura 1.3, ovvero è in grado di:

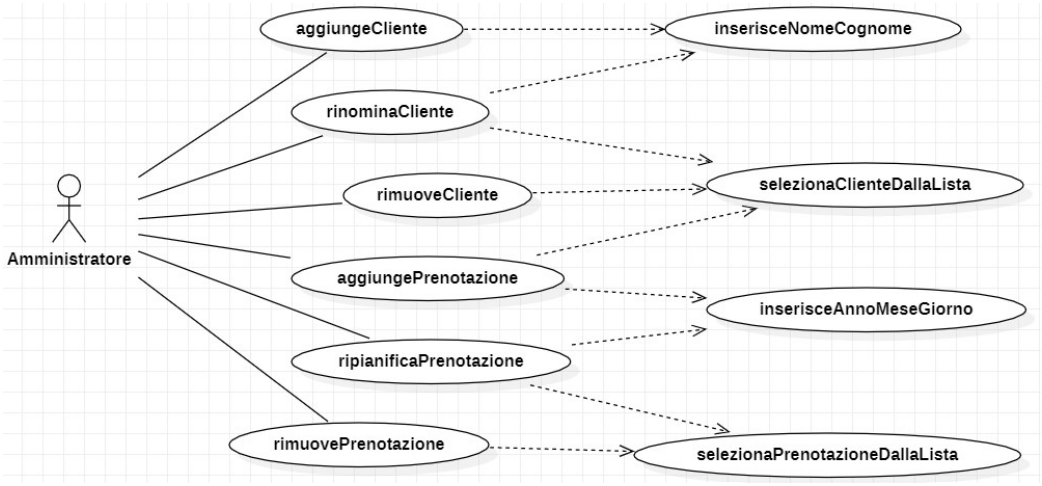

Figura 1.3: Use Case Diagram per l'amministratore di sistema.

- **Aggiungere un cliente** specificandone il nome e il cognome.
- **Aggiungere una prenotazione** per il cliente selezionato dalla lista specificandone l'anno, il mese e il giorno.
- **Eliminare una prenotazione** selezionata dalla lista.
- **Eliminare un cliente** selezionato dalla lista.
- **Rinominare un cliente** selezionato dalla lista specificandone il nuovo nome e cognome.
- **Riprogrammare una prenotazione** selezionata dalla lista specificandone il nuovo anno, mese e giorno.

Capitolo 2

Tecnologie e Framework

Lo sviluppo di BookingApp è avvenuto su **Eclipse**, e il progetto è stato portato a termine mediante l'ausilio di alcuni framework e tecnologie per Java, di cui se ne fornisce il contesto di utilizzo e una breve spiegazione.

2.1 Framework per l'applicazione

Swing BookingApp è un'applicazione desktop scritta in Java e in quanto tale beneficia dell'uso di **Swing** per realizzare la GUI, il cui vantaggio principale è la gestione delle finestre indipendente dal sistema operativo. Inoltre esistono alcuni plugin di Eclipse, come *WindowBuilder*, che ne forniscono un editor visuale e semplificano notevolmente lo sviluppo dell'interfaccia grafica.

Hibernate BookingApp può memorizzare i dati su disco tramite un database relazionale, il quale però necessita delle *Java Persistence API* (JPA) per gestire i dati relazionali. Fra i vari provider conformi alla specifica JPA, è stato scelto **Hibernate** come framework per il *Object/Relational Mapping* (ORM), che a sua volta utilizza *Java DataBase Connectivity* (JDBC) per connettersi al database. La sua configurazione richiede che l'unità di persistenza, ovvero il file `persistence.xml`, sia opportunamente inserita all'interno della cartella `META-INF` tra le risorse del progetto, e definisca: il provider JPA scelto; l'elenco delle classi che dovranno essere messe in ORM; il dialetto del DBMS utilizzato con il quale saranno definite le query SQL; le credenziali per autorizzare l'accesso al DBSM; altre proprietà specifiche del provider che configurano le operazioni in background. Infine, la configurazione viene completata dalle annotazioni poste nelle classi delle entità e interpretate in maniera opportuna da Hibernate.

Picocli Per far partire un'istanza di BookingApp da linea di comando può essere usato il FatJar, così come descritto nell'Appendice A. Il comando accetta alcuni argomenti per personalizzare l'esecuzione, come `-name` o `-psdw`, i quali risultano quantomeno ambigui; a tal proposito, **Picocli** permette di fornire una descrizione a ciascuno di essi in modo da far sparire ogni dubbio. Ma non solo: il framework associa i valori inseriti a delle opportune variabili, per cui vi è un controllo forte sul tipo. A titolo di esempio, BookingApp utilizza l'argomento `-dbms` per scegliere se usare MongoDB o PostgreSQL, a sua volta associato ad una variabile del tipo di enumerazione; qualsiasi valore diverso da MONGO o POSTGRES verrà dunque rifiutato.

2.2 Framework per i test

La maggior parte dei framework utilizzati hanno avuto come scopo principale la produzione e la semplificazione dei test, sotto diversi punti di vista. Alla base di tutti i test vi è **JUnit 5** che viene integrato a:

- **AssertJ** per migliorarne la leggibilità grazie allo stile *fluent* delle API, basato sulla *concatezione dei metodi*.
- **Mockito** per semplificare la creazione dei *test double*, fondamentale negli unit test, così come in molti integration test.
- **Testcontainers** per far partire e fermare automaticamente i *container Docker* delle istanze di database direttamente dai test case. Il framework viene usato negli unit test, ma non negli integration ed E2E test, per i quali si è preferito avviare manualmente le suddette istanze.
- **Awaitility** per sincronizzare operazioni asincrone tramite API *fluent*. Di fatto, il framework viene usato negli integration test sulle race condition per aspettare che tutti i thread abbiano eseguito l'operazione prima di procedere con le verifiche di AssertJ.

Del modulo *Jupiter* di JUnit 5 sono state ampiamente usate le annotazioni per definire le classi annidate (`@Nested`) e per decorare classi e test con un nome (`@DisplayName`); il modus operandi è sempre lo stesso: i test sono suddivisi in classi rispetto a funzionalità comuni (SUT, database, etc), poi ciascun test viene denominato rispetto alla condizione o azione svolta (nella nomenclatura del test è ciò si trova dopo il *when*). In questo modo se ne migliora la leggibilità, soprattutto del test case quando contiene molti test, oltre ad ottenere report ben categorizzati e meglio leggibili.

I test parametrizzati (`@ParameterizedTest`) sono un'altra funzionalità specifica di Jupiter, molto utili nella validazione degli input per le entità. Di fatto, vi sono diversi modi per violare la stessa condizione, ed è risultato particolarmente efficace documentare, mediante i test, le cause principali di ciascuna di esse.

Di JUnit 5 è stato utilizzato anche il modulo *Vintage* per i test che utilizzano **AssertJ Swing**, controparte di AssertJ per semplificare notevolmente la creazione dei test su UI create con *Swing*. Infatti, il framework manca completamente del supporto con Jupiter [2] [1], e dunque richiede di usare **JUnit 4**.

2.3 Iter di sviluppo

L'applicazione, sviluppata interamente tramite TDD, ha seguito un processo iterativo ben consolidato e consistente di 3 passaggi:

1. Verifica del 100% di *test coverage* tramite **JaCoCo**. Tale obiettivo è una diretta conseguenza del TDD, ma vale la pena accertarsene dato che il passo successivo non avrebbe senso senza di esso.

2. Uccisione di tutti i mutanti generati dai *mutation testing* di **PIT** in modalità DEFAULT. L'obiettivo viene raggiunto con (eventuali) ulteriori test volti a verificare il comportamento del codice rispetto a casistiche inesplorate.

3. Superamento della *code quality analysis* di **SonarCloud**. A tal proposito è stato definito un *quality gate* per il nuovo codice, che lo considerasse superato se:

    - non sono stati aggiunti bug, vulnerabilità e debito tecnico, e tutti i security hotspot sono stati controllati; in altre parole, il grado di *reliability*, *security* e *maintainability* rimane massimo, cioè A.
    - il codice è coperto almeno per l'80%; di fatto è stato raggiunto il 100% di test coverage anche su SonarCloud.
    - vi è codice duplicato al di sotto del 3%; di fatto è stato raggiunto lo 0% di linee duplicate.

Il processo è stato dunque utilizzato per ogni SUT sia localmente (N.B: con **SonarQube**, per la qualità del codice) sia su *GitHub Actions*, dove il CI ha il beneficio di controllare automaticamente gli obiettivi preposti.

Capitolo 3

Scelte di configurazione

Il progetto di BookingApp comprende 8 sotto-progetti Maven, ciascuno con una responsabilità ben definita:

- **booking-aggregator** è il reactor, dunque dichiara tutti i moduli Maven usati da BookingApp nel costrutto `<modules>`. Poiché completamente indipendente dai restanti sotto-progetti, necessita di sincronizzare i plugin con la stessa versione.

- **booking-bom** mantiene tutte le dipendenze *di compilazione* utilizzate fra i moduli sottomessi all'interno del tag `<dependencyManagement>`. Poiché interpellato dall'aggregator durante la build di Maven, dichiara anche tutti i plugin che potrebbero essere eseguiti da linea di comando (e.g. se eseguito `mvn clean surefire-report:report-only`, vengono aggiunti `maven-clean-plugin` e `maven-surefire-plugin`).

- **booking-parent** contiene tutte le dipendenze *di test* utilizzate dai suoi sotto-moduli all'interno di `<dependencyManagement>`, nonché tutti i plugin all'interno di `<pluginManagement>`, con l'obiettivo principale di fissarne le versioni. In secondo luogo, si occupa delle configurazioni dei plugin a comune fra i moduli figlio e della strutturazione base dei profili Maven.

- **booking-report** si occupa della generazioni di report composti di JaCoCo e Pitest, ottenuti a partire dai singoli report di ciascun modulo Maven. Il report di Jacoco viene usato anche per monitorare il coverage su Coveralls.

- i moduli **booking-*-module** contengono il codice di produzione e di testing, mentre nel loro POM vengono attivati e configurati dipendenze,

plugin e profili Maven specifici per il modulo stesso (che potrebbero sovrascrivere le configurazioni comuni che si trovano nel parent).

- **booking-app** è indipendente dagli altri sotto-progetti, senonché importa il BOM di progetto in modo da ottenere tutto lo stretto necessario per avviare un'istanza dell'applicazione di BookingApp.

L'organizzazione fra i vari moduli Maven ha svolto un ruolo fondamentale per il progetto di BookingApp, anche in ottica dei file di configurazione di cui di seguito se ne sottolineano gli aspetti caratteristici.

3.1 GitHub Actions

I file di workflow per **GitHub Actions** sono posizionati nell'omonima cartella contenuta in `.github` di `booking-bom`. Principalmente servono per il CI, ma su BookingApp sono stati utilizzati anche per realizzare un sito Web statico e una pagina di release per il progetto.

3.1.1 Continuous integration

I workflow **maven-*** eseguono la build Maven di BookingApp sul sistema operativo che completa il nome del file. L'esecuzione avviene utilizzando **Maven Wrapper** in modo che la versione di Maven sia ben definita e non ci siano problemi di compatibilità. Se i test falliscono, il workflow esegue altri *step* che sono utili al debugging. Per esempio, quando a fallire sono i mutation testing, i report vengono recuperati singolarmente da ciascun modulo. Oltre ai dettagli generici, il workflow di ciascun sistema operativo presenta delle peculiarità:

- Con **Linux** (`ubuntu-22.04`) la build aggiorna il coverage su Coveralls e verifica la qualità del codice con SonarCloud. Vengono attivati anche i profili Maven chiamati `pitest`, per eseguire i mutation testing, e `docker`, che dockerizza l'applicazione e ne verifica la compatibilità coi database.

- Su **macOS** (`macos-12`) la build non attiva alcun profilo Maven. Nonostante ciò, i test richiedono la presenza di Docker che quindi viene installato tramite un'opportuna *azione*. Va detto però che lo step `wait docker running` di `setup-docker` potrebbe fallire a causa di un timeout mal calibrato [4].

- L'ambiente virtuale di **Windows** (`windows-2022`) integra Docker di default ma può utilizzare solo container Windows. Durante lo sviluppo, i tentativi di usare i container Windows per MongoDB e PostgreSQL già presenti su Docker Hub (vedi [15] e [14], rispettivamente) hanno dato esito negativo, e non è stata presa in considerazione la possibilità di crearli ex-novo. Di conseguenza, il workflow esegue la build fino agli unit test, escludendo, tra l'altro e per lo stesso motivo, quelli che utilizzano Testcontainers.

3.1.2 GitHub Pages

GitHub Pages è un servizio di hosting statico per il sito Web delle repository. Una volta "sbloccato" dalle impostazioni, necessita di un workflow apposito per funzionare [7]. Di fatto, **gh-pages** di BookingApp presenta due *job*. Nel primo viene richiesto il *check out* della repository e tutti gli artefatti per il sito Web (file HTML, CSS e JavaScript). Dato che sono prodotti della build, sono stati aggiunti anche i profili `pitest` e `docker` in modo che se qualcosa dovesse andare storto, la pagina Web non verrebbe generata o aggiornata. Se tutto va per il verso giusto, i file di *staging* richiesti vengono raccolti e caricati, pronti per il *deployment* su GitHub Pages, eseguito dal secondo job.

La documentazione necessaria a costruire la pagina Web viene dichiarata all'interno del tag `<reporting>` nel POM del reactor, in modo che anche i report aggregati (e.g. Javadoc) possano essere aggiunti. Anche altre informazioni, quali il nome del dominio e la paginazione relativa del sito Web, devono essere configurate nel POM [22]. A tal proposito, viene usato il tag `<site>` all'interno di `<distributionManagement>` solamente nei sotto-progetti "indipendenti" (in BookingApp sono `booking-aggregator`, `booking-bom` e `booking-app`). Viceversa, per tutti gli altri sotto-progetti le informazioni sono inferite dalla gerarchia stessa.

Infine, l'ambiente di deployment specificato nel secondo job viene protetto da un'opportuna regola che ne consente le modifiche solo al ramo di default. Una volta caricato il workflow, il sito Web è consultabile dall'URL `marcopaglio.github.io/BookingApp/`.

3.1.3 GitHub Releases

GitHub Actions consente di automatizzare il rilascio di nuove versioni del progetto tramite un workflow opportuno [6]. Le *release* archiviano perennemente alcuni degli artefatti prodotti dalla build, quali il FatJar, il Javadoc e il codice sorgente nel caso di BookingApp. La struttura di **gh-releases** è di

fatto costituita da due job. Nel primo viene eseguito la build del progetto, in modo da verificarne il funzionamento ed ottenere gli artefatti. Nel secondo job vengono pubblicati i file su un apposito ambiente di deployment della repository, previa autorizzazione di GitHub. Nella configurazione viene anche specificata la dicitura che genera le note di rilascio.

Dato che il workflow si attiva solamente con un tag Git, per ottenere una release è prima necessario creare il tag per il progetto, per esempio da Eclipse oppure tramite il comando `git tag -a <version>`. Su BookingApp sono state rilasciate due versioni a solo titolo di esempio, poiché sostanzialmente sono uguali tra di loro.

3.2 Docker

Il sostegno di **Docker** nel progetto è stato determinante ed ha accompagnato lo sviluppo di BookingApp fin dagli unit test, sui quali non si è potuto fare a meno di usare vere istanze di database con Testcontainers.

Le stesse istanze vengono avviate in modo automatico durante la build dell'applicazione, mediante il `docker-maven-plugin` configurato nel POM di `booking-parent`, in modo che possano essere usate negli integration ed e2e test. In tal caso sono necessarie delle accortezze aggiuntive:

- Per usare le transazioni su MongoDB, l'istanza di database richiede l'associazione ad un *replica set* [21] (`mongod -replSet <repl_set_name>`) che, una volta avviata, dev'essere inizializzato con il comando `mongosh -eval "rs.initiate();"` [20] [19].

- D'altro canto, PostgreSQL specifica il massimo numero di connessioni disponibili tramite il comando `postgres -c max_connections=<num>`, dove `<num>` è un valore che dipende dal numero di connessioni contemporanee richieste [8] (in tal caso equivalente al numero di e2e test da eseguire) e dalla dimensione del *connection pool* di Hibernate, definito dalla proprietà `maximumPoolSize` di `persistence.xml` [3].

Per entrambi i container sono dunque serviti dei volumi, generati nella fase di `pre-integration-test`, prima dell'avvio delle istanze, e rimossi nella fase di `post-integration-test`, subito dopo la loro terminazione. In questo modo l'ambiente Docker rimane pulito.

3.2.1 Dockerfile

A partire dal FatJar prodotto dalla build Maven del progetto si può dockerizzare l'applicazione. L'operazione viene automatizzata dal profilo `docker`

e si basa sul **Dockerfile** contenuto in `booking-app`. L'immagine di base è `eclipse-temurin:11-jre` e su di essa vengono installate alcune librerie necessarie al funzionamento del protocollo X dall'interno del container, ovvero utili per renderizzare (`libxrender1`) il display, quindi ricevere (`libxi6`) e sintetizzare (`libxtst6`) gli input utente. Dopodiché, il comando Java che esegue il FatJar di BookingApp (ricevuto come argomento di build) ha bisogno di alcuni argomenti, per cui altrettante variabili d'ambiente dovranno essere definite all'avvio del container.

3.2.2 Display X11

Tra le variabili d'ambiente dev'essere impostata anche quella che dichiara il display X11 utilizzato. A tal proposito, ulteriori procedure potrebbero essere necessarie in base al sistema operativo, così come descritto nell'Appendice A. Lo stesso vale per gli ambienti virtuali di GitHub Actions. Per esempio, su Linux è anche necessario abilitare l'accesso per il **Xvfb** al container dell'applicazione. Il modo più rapido per farlo è quello di disattivare l'*access control* con l'opzione `-server-args="-ac"`, in modo che il virtual buffer sia accessibile a chiunque.

Inoltre, affinché Docker possa utilizzare il display X è necessario condividerne le API nella cartella `/tmp/.X11-unix` del container. Il *bind* del volume richiede la cartella sorgente delle socket, la quale però ha una posizione differente se Docker viene eseguito su **Windows Subsystem for Linux** (WSL). Di conseguenza, nel plugin si usa una proprietà (`xserver.socket.location`) il cui valore viene modificato se riconosce l'uso di WSLg; mentre per Docker Compose è stato definito un altro file (`docker-compose-wslg.yml`) per sovrascrivere i valori del costrutto `volumes:` [5].

3.2.3 Healthcheck

L'esecuzione di BookingApp fallisce se non trova il database attivo e utilizzabile. Lo stesso vale per l'applicazione dockerizzata. Di conseguenza, che sia durante la build Maven oppure tramite Docker Compose, le condizioni di *healthcheck* sono fondamentali per sincronizzare l'avvio dei container.

In **docker-compose-mongo.yml** l'inizializzazione della replica set dev'essere eseguita solo quando MongoDB è attivo. Pertanto il servizio attende una risposta positiva dai ping inviati al server. Non appena anche l'inizializzazione è terminata con successo, l'applicazione può partire. Per quanto riguarda **docker-compose-postgres.yml**, la condizione di healthcheck utilizza le API di PostgreSQL (`pg_isready`) per verificare che l'opportuno

database sia in funzione. In entrambi i casi, i controlli vengono ripetuti *più volte* prima di dichiarare l'inidoneità del servizio.

Durante la build automatica viene invece usato il `docker-maven-plugin`, il quale presenta delle limitazioni [16] su alcune feature che impediscono di usare le stesse condizioni di healthcheck dei file di docker-compose. Tuttavia, il plugin fornisce tag alternativi (e.g. `<wait>`, `<http>`, `<log>`, etc) coi quali realizzare delle strategie di sincronizzazione analoghe. Di fatto, per MongoDB vengono ancora una volta lanciati dei ping HTTP all'indirizzo del server in attesa di una risposta positiva. Viceversa, il container di PostgreSQL è pronto ad accettare richieste non appena il server stampa su console un opportuno messaggio. In entrambi i casi, i controlli vengono ripetuti *entro un timeout* prima di dichiarare l'inidoneità del servizio.

Capitolo 4

Implementazione

La definizione e l'implementazione della struttura applicativa non è funzionale solo agli obiettivi del progetto, ma concorda anche con le buone pratiche di programmazione architetturale.

4.1 Architettura e Responsabilità

Una delle prime difficoltà è stato capire cosa dovesse fare ogni componente architetturale. Ad un primo approccio, l'architettura software di BookingApp è stata suddivisa in 3 livelli di responsabilità:

- Il **modello di dominio** si occupa di definire il modello ad oggetti del dominio, che incorpora sia i dati dell'applicazione sia le regole di utilizzo.
- L'**accesso ai dati** si occupa di comunicare col database tramite transazioni ed operazioni CRUD.
- La **presentazione** mostra le informazioni all'utente e si occupa di gestirne le richieste (e.g. click del mouse, tastiera, etc).

Tuttavia, la logica con la quale la presentazione interroga il *data access layer* (DAL) non sarebbe stata banale: per esempio, fa uso della tattica *find-or-insert* (prima di inserire una nuova entità, controlla che non esista già). In generale, ogni volta che l'accesso ai dati è accompagnato da logica ulteriore, tale logica deve essere eseguita in un livello intermedio. La seconda strutturazione ha dunque riguardato l'inserimento del **servizio** fra l'accesso ai dati e la presentazione, con la responsabilità di fornire una logica più complessa della semplice operazione CRUD. A questo punto il livello presentazione perde l'accesso diretto ai dati, che quindi verrà interrogato dal servizio. Anche

le transazioni vengono inserite nel *service layer* in modo da evitare inconsistenze; di fatto, lo stato complessivo potrebbe non essere recuperabile se un'operazione che coinvolge più entità venisse eseguita in un altro livello e fallisse per una delle entità.

Una volta definita l'architettura di BookingApp, si è posto il problema (minore) della nomenclatura di classi e metodi ad ogni livello, in modo che rispecchiassero il loro ruolo e fossero di aiuto per comprendere la struttura del software anche ad esterni, oltre che allo sviluppatore stesso.

4.1.1 Modulo Domain

La struttura applicativa di **booking-domain-module** contiene l'implementazione del *modello di dominio* e del *DAL*, esemplificata dal diagramma UML di Figura 4.1, in cui sono presentate anche le suddivisioni in package delle classi.

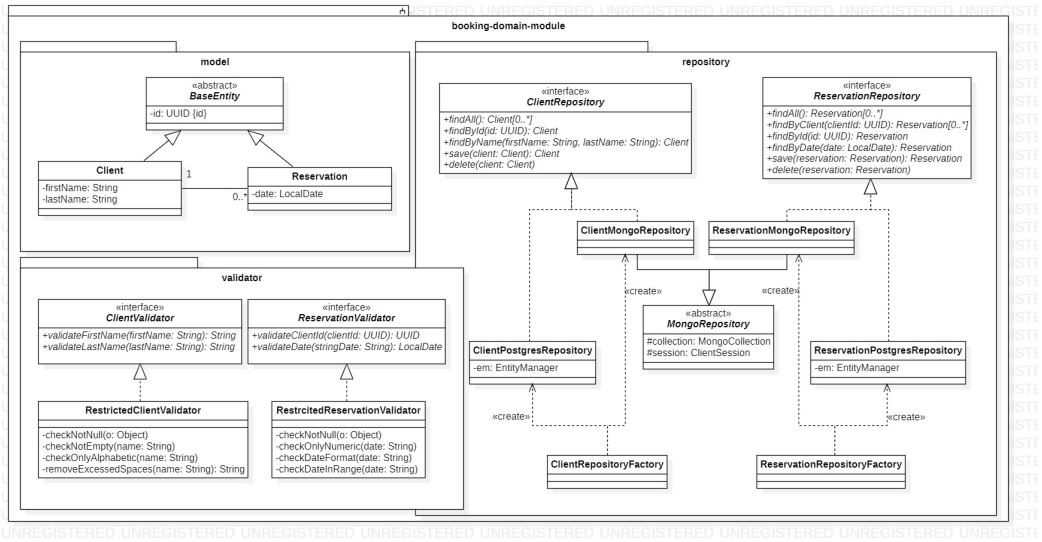

Figura 4.1: Diagramma delle classi per booking-domain-module.

Entità Tutte le classi che modellano le entità, cioè **Client** e **Reservation**, estendono la classe astratta **BaseEntity** che si occupa di associarvi una chiave primaria sotto forma di oggetto `UUID`, necessaria per l'archiviazione su database. Inoltre, le entità sono decorate con annotazioni utili per i database, ma dato che se ne utilizzano due di diverso tipo, allora anche le annotazioni da applicare sono doppie. Per esempio, la definizione di un campo si realizza tramite `@Column` per PostgreSQL e `@BsonProperty` per MongoDB. Va da

sé che le annotazioni non compromettono la struttura basilare delle classi POJO; d'altro canto, devono rispettare i requisiti per la memorizzazione su database:

- Non implementano interfacce o estendono classi da framework.
- Le classi non sono `final`, così come tutti i loro metodi.
- Includono un costruttore vuoto con visibilità pubblica o protetta.
- Tutti i campi da persistere non sono né `static` né `transient` né `final`.
- I metodi getter e setter sono pubblici e seguono la convenzione di nomenclatura.

Infine, le classi delle entità devono sovrascrivere i metodi `hashCode` e `equals` in modo che questi non si basino sull'id, dato che il valore della chiave primaria viene generato solo al momento della persistenza.

Validator Ciascuna entità viene affiancata da una classe che convalida e restituisce ogni suo attributo per mezzo di un opportuno metodo. In particolare:

- **RestrictedClientValidator** controlla che sia il nome che il cognome del cliente siano stringhe non nulle o vuote, contenenti solo caratteri alfabetici; vengono anche rimossi eventuali spazi ridondanti, in modo che non si possano inserire clienti omonimi solo perché hanno un diverso numero di spazi vuoti.
- **RestrictedReservationValidator** controlla che sia il riferimento al cliente che la data della prenotazione non siano nulli. Per la data si controlla anche che contenga solo caratteri numerici, oltre ai delimitatori, nel formato "aaaa-mm-gg"; accertamenti sui range di valore vengono invece lasciati alla classe `LocalData`, dato che presentano molti casi particolari (basti pensare agli anni bisestili).

L'assenza di caratteri estranei viene verificata tramite opportuni `Pattern`, ovvero espressioni regolari incapsulate in oggetti Java, con cui ricercare i simboli non desiderati sulla stringa da controllare.

Repository Per ciascun tipo di entità e per ciascun DBMS è stato applicato il pattern *Repository* con lo scopo di definire le operazioni CRUD e di ricerca. Il punto di accesso al database viene iniettato dall'esterno in modo che tutte le entità siano gestite dalla stessa istanza di database. Nel caso di MongoDB il costruttore riceve sia il `MongoClient` che il `ClientSession`, mentre per PostgreSQL occorre solo l'`EntityManager`. Della struttura generale vale la pena sottolineare che:

- Il metodo `save` cambia comportamento in base all'`id` dell'entità: se il valore è nullo, viene inserito un nuovo record; altrimenti l'oggetto si considera già nel database e dunque viene aggiornato.
- Nel caso si tenti di aggiornare un'entità non più nel database, il metodo lancia un'eccezione e non la re-inserisce.
- Il metodo `findByClient` di **ReservationRepository** è l'unico modo per recuperare tutte le prenotazioni di un cliente.

Andando un po' più nel dettaglio, le classi repository per MongoDB si occupano di configurare la collezione per le opportune entità, così come di definirne gli *indici di unicità*. Inoltre, hanno la responsabilità di popolare l'identificativo di database, così come di gestire i problemi ad esso annessi (e.g. collisioni). I metodi utilizzano le API di `MongoCollection` (e.g. `insertOne`, `deleteOne`, etc) ed operano nella `ClientSession` impostata alla creazione del repository.

Nelle classi relative a PostgreSQL, invece, i metodi alternano query JPQL alle API di `EntityManager`. Diverse API possono svolgere lo stesso compito ma con performance differenti; per esempio, `persist` e `merge` permettono entrambi di inserire nuovi record nel database, ma quest'ultimo è meno performante e meglio indicato per aggiornare le entità. Si ricorda inoltre che la modifica (e anche l'eliminazione) richiede che l'entità venga prima posta nello stato *Managed* mediante `getReference` di `EntityManager`.

4.1.2 Modulo Business

Le *transazioni* sono una componente fondamentale per i database perché ne garantiscono le proprietà ACID di fronte ad accessi concorrenti oppure relazioni fra entità, come quella che lega il cliente alle sue prenotazioni, e viceversa. All'interno di **booking-business-module** vengono definite le classi che si occupano di contenere, gestire e applicare le transazioni. L'intera struttura viene presentata in Figura 4.2.

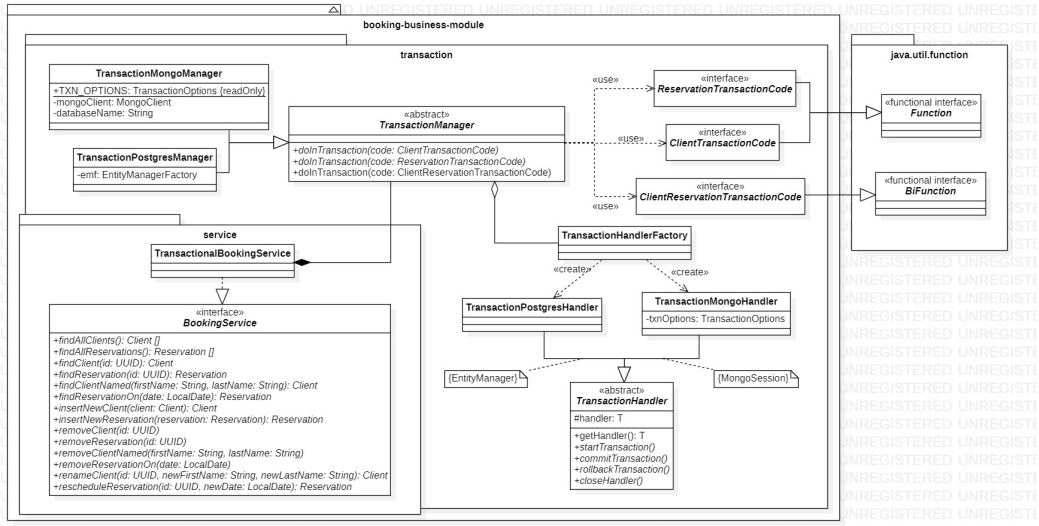

Figura 4.2: Diagramma delle classi per booking-business-module.

Manager & Handler L'uso delle transazioni dev'essere gestita adeguatamente sia per l'esito positivo che per quello negativo, quindi richiede un *manager* che se ne occupi. Di fatto, il **TransactionManager** ha la responsabilità di creare la transazione e le istanze di repository che si interfacciano al database, per poi eseguire del codice al suo interno. Se qualcosa va storto deve revocare la transazione prima di propagare l'errore, e in ogni caso deve chiudere la transazione prima di restituire il comando.

Ciascuna di queste operazioni viene supportata da un *handler* specifico del DBMS: nel caso di PostgreSQL/Hibernate l'operazione è delegata ad un `EntityManager`, mentre su MongoDB vi è un `ClientSession` che si autogestisce l'apertura/chiusura della sessione. Va da sé che la logica dell'operazione diverge tra i due DBMS, per cui nasconderla in un **TransactionHandler** è risultata una scelta vincente per allineare il concetto di transazione all'interno di BookingApp. Anche i test ne beneficiano poiché le interazioni vengono adesso verificate sui metodi del TransactionHandler, piuttosto che direttamente sulle API di terze parti.

Il TransactionManager mette in vita un oggetto TransactionHandler per ogni richiesta da eseguire in una transazione. La ciclicità del pattern *session-per-request* [11] viene agevolata mediante opportune classi *Factory* che mettono in vita sia l'handler che i repository specifici per il DBMS, ma non prima di aver controllato la validità dei parametri.

Vale la pena ricordare che ciascun handler può creare e utilizzare una sola transazione alla volta. Di conseguenza, se si tentasse di farne parti-

re una seconda, l'operazione verrebbe bloccata da un'eccezione. Lo stesso esito si avrebbe se si tentasse di commissionare o revocare una transazione già conclusa. Va da sé che sarebbe più opportuno controllare lo stato della transazione prima di procedere con l'operazione. Tale logica viene dunque affidata al TransactionHandler, in modo che il TransactionManager non debba continuamente interrogarsi sulla sua fattibilità. Questa modalità rende il codice molto più pulito e leggibile.

Infine, durante la creazione di un **TransactionMongoHandler** vengono iniettate delle opzioni che definiscono la tipologia di lettura/scrittura su MongoDB. In particolare, entrambe ricevono `"majority"` come valore, ovvero promettono di agire sulla versione più aggiornata del database [17]. Tale configurazione garantisce la *casual consistency*, un modello di consistenza utile in programmi concorrenti. Esso rappresenta un'ottima via di mezzo fra la più restrittiva *strong consistency*, e la meno affidabile e intelligibile *eventual consistency*.

Code Per eseguire del codice all'interno di una transazione si utilizzano le *functional interface* di Java, grazie alle quali vengono strutturate delle funzioni lambda che prendono le repository come parametri.

In TransactionManager ne vengono usate di tre categorie: le interfacce **ClientTransactionCode** e **ReservationTransactionCode** estendono `Function` e definiscono il tipo del singolo parametro con quello del corrispondente repository; d'altra parte, **ClientReservationTransactionCode** estende `BiFunction`, anch'essa una functional interface, con cui definisce entrambe i repository come parametri d'ingresso. Per tutti e tre i casi il valore di ritorno rimane inespresso e viene inferito direttamente dalla funzione lambda. I metodi del TransactionManager che fanno uso dei TransactionCode hanno la stessa implementazione, ma la duplicazione del codice non può essere tolta proprio per il fatto che i tre tipi non hanno una superclasse comune. A tal proposito è stata definita una *rule exclusion* di SonarQube per il solo TransactionManager.

Va anche detto che tali interfacce sono escluse dai test, mentre il codice vero e proprio della funzione lambda dev'essere testato quando definito dalla classe chiamante, che in BookingApp è rappresentata dal servizio.

Service Il *service* racchiude tutti i possibili "servizi" offerti a BookingApp e che coinvolgono le entità sul database. Nonostante ciò, la scelta del DBMS è nascosto dal manager e non influenza la logica di programmazione.

Viene usata un'unica classe **TransactionalBookingService** (e non una classe per ogni entità) perché alcune operazioni coinvolgono più tipi di enti-

tà (ciò motiva anche l'esistenza di ClientReservationTransactionCode). Per esempio, l'eliminazione di un cliente richiede di liberare anche tutte le sue prenotazioni. A tal proposito, nonostante venga fatto all'interno di una transazione, l'ordine delle operazioni vuole lasciare il database il meno corrotto possibile. Pertanto, si eliminano prima tutte le sue prenotazioni e alla fine il cliente, così da non lasciare prenotazioni orfane sul database.

Essendo dunque molto specifico per l'applicazione stessa, sono stati definiti alcuni servizi che si basano sui dettagli del progetto, quali la ricerca tramite nome e cognome per i Client (`findClientNamed`), e quella tramite data per le Reservation (`findReservationOn`), ovvero rispetto ai campi univoci delle entità. In generale, i nomi dei metodi richiamano le azioni da intraprendere per soddisfare un certo caso d'uso.

Infine, prima di eseguire una certa operazione sul database viene verificata la sua realizzabilità. In questo modo si alleggerisce il database da richieste inutili, e si ottengono eccezioni più dettagliate sul perchè un'operazione non possa essere espletata.

4.1.3 Modulo UI

La rappresentazione dell'interfaccia grafica di BookingApp e l'utilizzo dei servizi sottostanti viene implementata nel **booking-ui-module**. Il diagramma delle classi in Figura 4.3 mostra la mutua interazione fra le due componenti nel contesto del pattern *Model-View-Presenter* (MVP), che ben si presta al TDD.

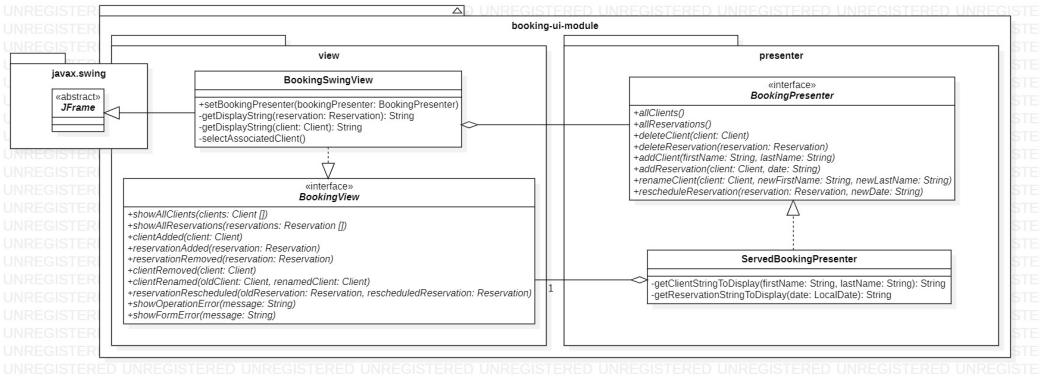

Figura 4.3: Diagramma delle classi per booking-ui-module.

Presenter Il pattern MVP è una semplificazione del Model-View-Controller in cui il *presenter* svolge il ruolo di *man-in-the-middle* per gestire le richieste

utente e notificare la vista circa i cambiamenti del modello. All'interno di BookingApp, **ServedBookingPresenter** utilizza un'unica vista e delega le richieste utente al database tramite il service, in modo che siano eseguite in una transazione. La nomenclatura dei metodi ricalca quella dei casi d'uso.

I metodi per inserire le entità (`addClient`, `addReservation`) ricevono i dati prelevati dai form direttamente dalla vista. Per creare l'entità, ServedBookingPresenter deve prima controllarne la validità, dunque è sua responsabilità chiamare i *validator*. Se qualcosa non va con gli input, non viene nemmeno fatto partire il flusso di operazioni verso il database. Anche i metodi che aggiornano le entità (`renameClient`, `rescheduleReservation`) convalidano gli input, tuttavia se la modifica non apporta dei cambiamenti effettivi (e.g. una prenotazione che viene riprogrammata per la stessa data), non viene delegata al service.

Ogni volta che una richiesta utente non può essere accolta, il presenter mostra un messaggio all'utente coi dettagli di quanto accaduto e, se possibile, su come intervenire per portarla a termine. Inoltre, viene aggiornata la vista al modello più recente, dato che BookingApp è un'applicazione multiutente; se così non fosse, l'utente potrebbe annoiarsi di fronte a continui messaggi di errore. Per esempio, se volessimo eliminare delle prenotazioni di un cliente che è già stato eliminato da qualcun altro (e dunque anche tutte le sue prenotazioni), riceveremmo un messaggio di errore per ciascuna prenotazione inesistente, anziché constatare fin dopo il primo tentativo il nuovo stato del modello. Affinché questa stessa casistica non accada anche per il singolo utente sotto forma di *race condition* a livello di applicazione, tutti i metodi di ServedBookingPresenter che modificano il modello sono definiti `synchronized`.

View L'interfaccia grafica definisce i controlli necessari al presenter per applicare gli aggiornamenti del modello. A differenza di quanto succede nel MVC, nel MVP anche **BookingSwingView** ha un riferimento a BookingPresenter che utilizza per inoltrargli le richieste utente. Per quanto riguarda la GUI, già descritta nel Capitolo 1, vale la pena sottolineare che:

- Le entità in lista vengono visualizzate tramite un opportuno metodo `getDisplayString` che non dipende dai dettagli di basso livello.
- Non appena cliccato, il pulsante viene subito disabilitato per evitare quanto più possibile richieste concorrenti, nonostante siano gestite dal presenter.
- Quando una prenotazione viene selezionata dalla lista, anche il cliente che ne ha fatto richiesta viene selezionato.

- I messaggi di errore possono essere dovuti ad una scorretta compilazione dei form, mostrati sotto ai campi stessi; oppure ad un'operazione fallita, mostrati in fondo alla finestra.

4.1.4 Applicazione

Mediante opportuni argomenti, BookingApp dà la possibilità all'utente di scegliere con quale DBMS avviare l'applicazione (`-dbms`), così come su quale porta (`-port`) e host (`-host`) mettersi in ascolto del database con il nome specificato (`-name`). Ci sono altri due argomenti (`-user` e `-pswd`) che vengono usati per accedere al database; tuttavia solo per PostgreSQL/Hibernate viene fornito un sistema di autenticazione di default. Nel caso di MongoDB si sarebbe dovuto impostare, ma dato che il controllo sugli accessi non è mai stato un requisito di progetto, i suddetti argomenti vengono semplicemente ignorati.

Viene dunque utilizzata un'unica classe **BookingSwingApp** la cui struttura, in Figura 4.4, non dipende da quali o quanti DBMS l'applicazione può utilizzare. Al suo avvio, viene istanziato un oggetto **DatabaseHelper** che si occupa di aprire la connessione col database, nonché di creare un TransactionManager specifico per il DBMS scelto. Per PostgreSQL l'operazione consiste nella creazione dell'`EntityManagerFactory` a partire dall'unità di persistenza, i cui valori vengono aggiornati con quelli inseriti dall'utente. D'altra parte, MongoDB richiede la definizione di un `MongoClient` configurato tramite le regole di serializzazione/deserializzazione in BSON (Binary JSON) [9]. A tal proposito, viene usato un `CodecRegistry`, ovvero una lista di `Codec` per la conversione. Talvolta, però, se i tipi per i `Codec` non sono conosciuti a *compile time*, si deve usare un `CodecProvider` che ne ritarda la costruzione a *runtime*. Dunque, la configurazione è avvenuta andando a [10]:

1. **Impostare il CodecProvider** considerando che un'impostazione può essere ripetuta più volte, ma viene utilizzata solo l'ultima definita; di fatto, `automatic(true)` viene messo in fondo e serve per automatizzare le operazioni.

2. **Compilare il CodecRegistry** in modo che contenga tutte le varie regole di conversione; in particolare, il precedente `CodecProvider` è stato messo come ultimo elemento della lista per evitare sovrascritture.

3. **Configurare il MongoDatabase** in modo che utilizzi il `CodecRegistry` definito al passo precedente.

Nel `main` viene anche definito un *hook di shutdown*, invocato qualora l'applicazione termini (anche a causa di un'eccezione), che si occupa di chiudere

la connessione col database mediante, per l'appunto, il DatabaseHelper. Se il fallimento si verifica all'avvio dell'applicazione, il thread si occupa anche di rilasciare le risorse dell'ambiente grafico.

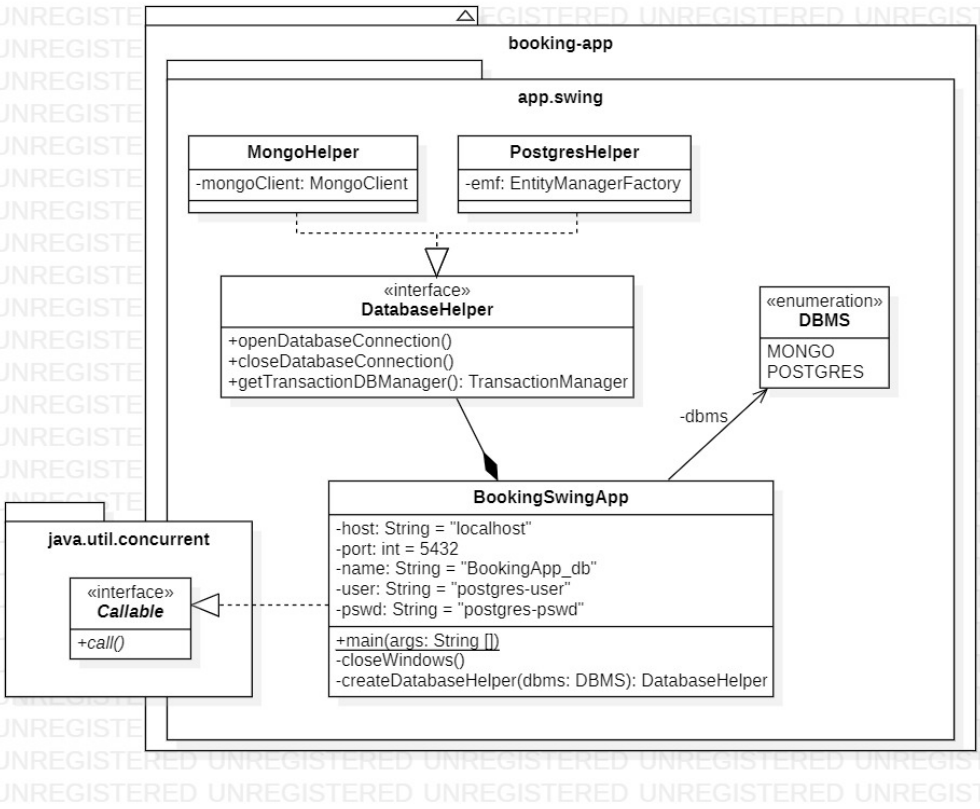

Figura 4.4: Diagramma delle classi per booking-app.

4.2 Allineare i database

Un'altra difficoltà del progetto è stata la necessità di modellare il DAL in modo che il contratto dei dati venisse rispettato indipendentemente dal database utilizzato, non solo fra livelli diversi, ma anche all'interno dello stesso DAL. Non è stato semplice definire cosa dovesse svolgere ogni componente poiché i database in questione (MongoDB e PostgreSQL) hanno strutture e modalità di funzionamento talmente diversi da non poterli allineare, a meno di un compromesso.

Per esempio, una prima differenza è nella gestione dell'identificativo di database; di fatto, sebbene entrambi supportano l'uso dello UUID, solamen-

te PostgreSQL, tramite il provider Hibernate, ne automatizza l'operazione, mentre per MongoDB l'inserimento e la gestione dello UUID dev'essere programmato manualmente (a meno di usare altri framework, come *Spring Boot*). Allo stesso modo, per PostgreSQL la gestione dei vincoli di non nullità è configurabile con Hibernate aggiungendo la proprietà `nullable` in `@Column`, mentre su MongoDB la funzionalità non è proprio disponibile. In altre parole, il DAL lato MongoDB deve compensare anche il ruolo di Hibernate.

D'altro canto, MongoDB riconosce la violazione di altri vincoli (e.g. di unicità) non appena viene chiamato il metodo che agisce sul database, mentre su PostgreSQL l'esecuzione delle query viene ritardata al *commit* o alla chiusura dell'Entity Manager. Di conseguenza, le eccezioni vengono generate in contesti differenti. A tal proposito, sono state identificate 3 possibili soluzioni:

- Usare i *bean validation* di Hibernate per eseguire un controllo a livello di applicazione Java. In questo modo non viene nemmeno generata la query se vi è una violazione di vincoli, ma il loro utilizzo richiede di chiamare il validatore per ogni creazione e modifica, per cui non è molto indicato a meno di utilizzare framework che ne automatizzino la chiamata (e.g. *Spring Data JPA*).

- Spostare la cattura e gestione delle eccezioni a livello delle transazioni. Tuttavia, in questo modo non è possibile unit testare la violazione dei vincoli (dato che il livello sovrastante ne sarebbe inconsapevole), rinviando tali controlli in integration o e2e test.

- Aggiungere un'operazione di *flush* all'interno dei metodi in modo che l'eventuale violazione si palesi immediatamente. Questa modalità permette di allineare l'interfaccia del DAL con MongoDB, ma forzare il flush potrebbe portare a problemi di performance, deadlock e inconsistenza dei dati.

Nonostante i possibili svantaggi, la scelta è ricaduta sull'uso di flush dato che è l'unica opzione efficace per l'allineamento dei database ed è concorde al TDD, poiché consente di unit testare la violazione dei vincoli.

Anche il tipo di eccezione lanciata per la stessa condizione è diversa nei due casi. A tal proposito, si è ricorso alla buona pratica di racchiudere le eccezioni di terze parti all'interno di eccezioni personalizzate e dello stesso tipo per entrambi i database.

4.3 Gestione delle eccezioni

A partire dall'allineamento dei database, si è manifestato un problema più generale, cioè la gestione delle eccezioni. La buona pratica consiste nel catturare l'eccezione lanciata dalle API di terze parti, stamparne il messaggio e rilanciare un'eccezione personalizzata. Tuttavia, la procedura non è stata applicata per ogni possibile eccezione documentata dalle API. Anzitutto si deve considerare di lavorare nel giusto contesto, senza sovraccaricare le classi delle responsabilità altrui. Per esempio, le classi *repository* non controllano se è stata fatta partire una transazione, nonostante sia fondamentale per il funzionamento di alcune API di database (e.g. `persist`). O ancora, prima di optare per un lancio di un'eccezione, si cerca sempre di risolvere o aggirare il problema. Basti pensare alle classi *handler* che si occupano di prevenire lanci di eccezioni quando si tenta di aprire (o chiudere) una sessione già attiva (o chiusa).

Per decidere se un caso particolare richieda il lancio di un'eccezione bisogna dunque appellarsi alle responsabilità del metodo. Si consideri l'esempio in cui non viene trovato un record del database da eliminare. Dato che per le classi *repository* lo scopo dell'operazione viene comunque raggiunto, il metodo `delete` nasconde l'eventuale lancio dell'eccezione; viceversa, controllare se un'eliminazione abbia apportato delle modifiche al database è una responsabilità del *service* che, per l'appunto, lancia un'eccezione se non trova l'oggetto. Dunque, anche le operazioni di ricerca nei *repository* (e.g. `findById`, `findByName`, etc) mettono in conto che potrebbero non trovare l'entità (di fatto, a volte si usano proprio per scongiurare che un'entità sia già nel database) e nel caso non sia presente, viene restituito un oggetto `Optional` vuoto. Viceversa, il *service* implementa quella logica per cui la deviazione dal comportamento atteso è determinante per il lancio di un'eccezione.

Infine, il *wrapping* in eccezioni personalizzate è stato esteso all'intera architettura, dove ciascun livello si occupa di catturare le eccezioni provenienti dal livello sottostante, stamparne il messaggio e rilanciare un'eccezione specifica per il livello, e pertanto meno dettagliata. Il presenter è dunque il punto più alto dell'architettura dove vengono catturate le eccezioni, per poi essere trasformate in opportuni messaggi da visualizzare sulla vista. Tutto ciò che trascende questo limite rappresenta dunque un'eccezione non prevista in fase di sviluppo, e viene catturata dal `main` col fine di rilasciare le risorse acquisite, prima di interrompere l'applicazione.

Capitolo 5

Tecniche di Testing

Le strategie di testing dipendono essenzialmente dalla tipologia del test da scrivere, nonché dalle "problematiche" che si porta dietro. Oltre a verificare il corretto comportamento del codice di produzione, l'obiettivo è stato suddividere i test in modo che fossero coerenti e completi nel loro insieme, sovrapponendosi il meno possibile. Per di più, i test sono stati scritti in modo che fossero il più leggibili e facilmente intepretabili possibile.

5.1 Unit Test

Il fatto che gli **unit test** devono basarsi unicamente sul SUT costituisce il loro vincolo prioritario. A tal proposito, nelle fasi sia di *setup* sia di *verify* potrebbero servire funzioni ad hoc, implementate all'interno del test case, oppure metodi getter, messi a disposizione dalla classe testata. Allo stesso modo, eventuali interazioni coi collaboratori devono essere sostituiti con opportuni *test double*, dunque anche la strategia con cui utilizzare Mockito svolge un ruolo importante.

Per le classi caratterizzate da molti collaboratori (e.g. i punti di snodo e di gestione all'interno dell'architettura), si è scelto di verificare per filo e per segno tutte le interazioni con gli oggetti mock (e anche che non ne siano fatte ulteriori) per i soli *positive case*, mentre tutti gli altri test (*negative* e *corner case*) si concentrano sulle interazioni peculiari del caso. Per esempio, se il metodo `insertNewClient` di TransactionalBookingService lancia un'eccezione, è importante verificare *solamente* che il metodo `save` del repository non venga chiamato, dal momento che ulteriori controlli sono effettuati quando l'operazione va a buon fine. In questo modo se ne aumenta la leggibilità, senza lasciare scoperta alcuna porzione di codice.

Mockito viene usato anche per forzare opportune casistiche da testare. Per esempio, il *data access layer* di MongoDB si occupa di fornire alle entità un identificativo di database, scelto in modo casuale fra un grande insieme di valori, il cosiddetto *Universally Unique IDentifier*. Per forzare l'inserimento dello stesso UUID in modo da testare una situazione rara dal verificarsi, ma non impossibile, quale appunto che venga dato ad un nuovo oggetto un identificativo già fornito in precedenza, è stato sufficiente usare lo *stubbing with callbacks* sul metodo `setId` dell'entità, come di seguito:

```
1 doAnswer( invocation -> {
2     ((Reservation) invocation.getMock())
3         .setId(A_RESERVATION_UUID);
4     return null;
5 }).when(spied_reservation)
6     .setId(not(eq(A_RESERVATION_UUID)));
```

In modo del tutto analogo può essere forzato l'inserimento di uno UUID diverso da quanto già inserito, utile nel caso si voglia essere sicuri che venga rispettato il vincolo di unicità.

Sempre a proposito del DAL, gli unit test utilizzano una reale implementazione dei database dato che quella *in-memory* è insufficiente (MongoDB) o inesistente (PostgreSQL) per usare le transazioni su Java. Questa scelta fa sì che l'esito di alcuni test dipenda da come viene impostato l'ambiente di testing, ovvero nella stessa sessione su cui opera il SUT o meno. Per i suddetti casi, sono stati eseguiti doppi test in cui l'unica cosa a variare è il tipo di configurazione iniziale.

Infine, la rapidità è un altro requisito fondamentale per gli unit test. Purtroppo però, ancora una volta, la necessità di usare un'implementazione reale dei database fa sì che test che fanno partire un container Docker siano piuttosto lenti. Altra circostanza in cui la velocità viene a mancare è coi test sulla UI, in cui la generazione della finestra e le interazioni robotizzate ne provocano un rallentamento. In aggiunta, non è possibile eseguire PIT con le Swing UI [13] [12], per cui il criterio con cui sono stati scritti gli unit test è stato particolamente pedante, in modo da rimpiazzare (per quanto possibile) il ruolo dei mutation testing. In definitiva, ben 164 dei 623 unit test totali fanno parte di BookingSwingViewTest. Per aggirare il problema è stato aggiunto un profilo Maven, denominato `skip-gui-tests`, per l'esecuzione senza di essi, in modo da non averne a che fare se non strettamente necessario.

5.2 End-to-end Test

A differenza degli unit test, gli **e2e test** non usano mai test double, quindi da questo punto di vista risultano più semplici. Tuttavia, la difficoltà sta nello scrivere i test in modo che simulino l'utente finale, ovvero che venga utilizzata la sola interfaccia utente per le fasi sia di *execute* che di *verify*. Per sottostare al limite è stato sufficiente decidere a priori i cambiamenti sulla UI in base all'operazione effettuata, per poi tradurre le decisioni nelle asserzioni AssertJ più consone. Per esempio, quando viene premuto il pulsante "Remove Client" il cliente selezionato non deve più apparire nella lista; ciò viene tradotto in AssertJ andando a controllare che nessun elemento della lista abbia lo stesso nome e cognome del cliente selezionato, proprio come farebbe un ipotetico utente finale (ovviamente vengono controllati solo il nome e il cognome perché l'assenza di omonimia è un requisito di BookingApp).

Proprio perché tutti gli e2e test del progetto utilizzano l'interfaccia utente, questi sono circoscritti a `booking-ui-module`. In teoria, il modulo conterrebbe due *test case* per testare l'applicazione con MongoDB e PostgreSQL, per un totale di 44 test. Tuttavia, la scelta di quale DBMS viene usato (così come qualsiasi altro dettaglio di basso livello) non deve influenzare la UI, e di fatto le due test case si distinguono solo per il *setup* dell'ambiente di testing. Per enfatizzare la struttura generale degli e2e test, i test sono contenuti in una classe astratta e vengono duplicati per ciascuna classe che ve ne deriva e ne implementa le opportune funzioni di setup in base al DBMS utilizzato. Le configurazioni iniziali comprendono anche l'avvio dell'applicazione da testare. Ciò è possibile definendo una classe inner contenente un metodo main specifico per il database all'interno di ciascuna classe derivata, e sfruttando le API di AssertJ Swing. La scelta di isolare i test in un'unica classe migliora la qualità di lettura all'interno del test case, non più "sporcato" dalla presenza delle configurazioni di server, applicazione e quant'altro.

Infine, per le classi contententi gli e2e test è stata disabilitata la regola SonarQube che ne segnala i suffissi non standard; di fatto, SonarQube considera solo `Test` e `IT` come suffissi per i test, non `E2E`.

5.3 Integration Test

Rispetto ai precedenti, gli **integration test** sono meno soggetti a vincoli perché possono dipendere dai dettagli di basso livello, per cui non c'è bisogno di generalizzare come negli e2e test, ma a differenza degli unit test non ne dipendono così tanto, e ciò gli permette di concentrarsi sul cuore del test. Il più delle volte un integration test verifica i cambiamenti di stato,

mentre le interazioni servono solo per controllare che le chiamate abbiano opportuni parametri. Per esempio, nei ValidatedServedBookingPresenterIT vengono testate le classi ServedBookingPresenter e RestrictedValidator, le quali non modificano alcuno stato, per cui i test si limitano a verificare che un metodo opportuno del livello di servizio o della vista venga chiamato coi valori convalidati o con l'errore da mostrare, rispettivamente.

Si può già notare come un IT non necessariamente utilizzi reali implementazioni per tutte le dipendenze del SUT. Tuttavia, usare il mocking per dipendenze annidate avrebbe violato la scatola grigia degli IT, e dunque l'uso dei test double è risultato utile solo per le dipendenze dirette del SUT. Si faccia riferimento ai ServedBookingPresenterIT che si distinguono dai precedenti per l'utilizzo di una reale implementazione del *service layer*; in questo caso non viene falsificata nessuna delle dipendenze necessarie per creare il TransactionalBookingService. Come conseguenza, i test possono servirsi del collaboratore per la fase di *verify*.

Altro dettaglio a cui è stata data particolare attenzione è sull'inutilità di fare IT per tutte le combinazioni di collaboratori. Per esempio, se vi sono collaboratori annidati su più livelli, è sufficiente eseguire gli IT al livello più alto, oppure anche ad un livello intermedio se questo è uno snodo. L'importante è che ogni possibile coppia di interazioni fra collaboratori venga testata. Per questo motivo i TransactionalBookingServiceIT sono gli unici integration test di `booking-business-module` e verificano che il contratto dei dati venga rispettato fra tutte le classi sottostanti al livello di servizio.

L'obiettivo di non duplicare gli integration test è stato raggiunto decidendo a priori cosa avrebbe dovuto ispezionare ogni integration test. Per esempio, per BookingSwingView vi sono i ServerBookingPresenterSwingViewIT, i quali verificano che le azioni del presenter siano accompagnate dalle opportune modifiche sull'interfaccia grafica, e i ModelSwingViewServedPresenterIT, i quali agendo sulla vista modificano sia la GUI che il database, ma verificano solamente i cambiamento di quest'ultimo in modo da distinguersi dai precedenti.

Per concludere, nonostante gli integration test siano i meno vincolati, la loro realizzazione non è altrettanto semplice e guidata (di fatto, il TDD aiuta con gli unit test, mentre i casi d'uso servono per gli e2e test). Tuttavia, grazie ad un'adeguata progettazione è stato possibile realizzare integration test il cui ruolo non si sovrapponesse, ma anzi ne fosse complementare, e in numero tale da rispettare la forma della piramide del test (ovvero 148, appena un quarto rispetto agli unit test, e più di tre volte gli e2e test).

Conclusioni

Da quanto svolto nel progetto, ci si è resi conto che l'uso del TDD può prevenire gli errori di programmazione e agevolare lo sviluppo software, a patto di conoscere a menadito gli strumenti a disposizione (e.g. linguaggio di programmazione, framework, database, etc). A posteriori si riconosce anche l'importanza dei *learning test*, coi quali lo sviluppatore può colmare le eventuali lacune. Di fatto, nonostante non fosse un requisito di progetto, la scrittura di alcuni learning test è risultata utile per comprendere il funzionamento dei DBMS. Lo sviluppo ha dunque richiesto anche un minimo di conoscenza circa le architetture software e i design pattern, entrambi importanti per utilizzare database di diverso tipo. Inoltre, è maturata la consapevolezza che tecniche di Build Automation applicate al Continuous Integration sono fondamentali soprattutto nei lavori in gruppo. Infatti, consentono di definire a priori i requisiti minimi (di qualità) del progetto a cui attenersi per poter contribuire. Tuttavia, ancora una volta, queste agevolazioni non sono gratuite ma richiedono la conoscenza e la configurazione degli strumenti da impiegare.

Appendice A

Replicare l'esecuzione

La suddetta appendice è un estratto in lingua italiana del file README di BookingApp che non comprende alcuni dettagli specifici, come sulla compatibilità della macchina.

A.1 Programmi da installare

Per *eseguire l'applicazione* di BookingApp devono essere installati almeno i seguenti programmi:

- **Java 11**
- **Docker Engine**

Per *replicare build*, test e quant'altro, il progetto richiede anche:

- **Git**
- **Docker Compose**

È possibile eseguire la build da linea di comando. In tal caso è necessario installare **Maven**, preferibilmente nella versione usata per il progetto, ovvero 3.8.6, oppure usare gli script di **Maven Wrapper**.

Nel caso voglia essere usato un IDE, si consiglia **Eclipse** dato che è servito anche per lo sviluppo di BookingApp.

A.2 Clonare la repository

Per clonare il progetto di BookingApp, scegliere una cartella qualsiasi ed eseguire su terminale:

```
git clone https://github.com/marcopaglio/BookingApp.git
```

Nella cartella scelta apparirà la **directory principale del progetto**, ovvero `BookingApp/`.

A.2.1 Importare su Eclipse

Una volta clonato, è possibile importare il progetto di BookingApp su Eclipse. A partire dalla barra in alto a sinistra: **File** > **Open Projects from File System...** > usa **Directory..** per scegliere la *directory principale del progetto* > assicurati di selezionare l'opzione `Search for nested projects` > da **Folder list** importa tutte le sottocartelle tranne `BookingApp` > **Finish**.

A.3 Eseguire la build

Per eseguire la build di BookingApp è consigliato avere le immagini Docker dei DBMS in locale; a tal proposito, devono essere eseguiti i seguenti comandi singolarmente sul terminale:

```
docker pull mongo:6.0.7
docker pull postgres:15.3
```

A.3.1 Da linea di comando

Una volta aperto il terminale nella *directory principale del progetto*, il comando base per eseguire la build del progetto è il seguente:

```
<MVN> -f booking-aggregator/pom.xml clean install
```

dove `<MVN>` dev'essere sostituito con un comando di script adeguato:

- con **Maven** si usa `mvn`.
- con **Maven Wrapper** si usa `./mvnw` sui sistemi Unix, oppure `./mvnw.cmd` su Windows.

Profili Maven

Build alternative possono essere eseguite aggiungendo uno o più profili Maven alla fine del comando precedente:

- `-Pjacoco` aggiunge il test coverage, i cui report sono visualizzabili da `/booking-report/target/site/jacoco-aggregate/index.html` una volta terminata l'esecuzione.

- `-Ppitest` aggiunge il mutation testing, i cui report sono visualizzabili da `/booking-report/target/pit-reports/index.html` una volta terminata l'esecuzione.
- `-Pskip-gui-tests` non esegue i test che richiedono un'interfaccia grafica, utile per velocizzare l'esecuzione.
- `-Pdocker` crea un'immagine Docker dell'applicazione che viene denominata `booking-app`; è prima necessario impostare l'ambiente grafico per Docker.

A.3.2 Da Eclipse

È possibile eseguire la build di BookingApp da Eclipse utilizzando i *launch file* che si trovano nella cartella `launches` dei vari sotto-progetti: basta cliccarvi sopra col tasto destro > **Run As** > cliccare sulla configurazione Maven che porta lo stesso nome.

La nomenclatura dei file di launch consiste in una *radice*, corrispondente al nome del sotto-progetto, e un *suffisso*, che indica la peculiarità della build, come descritto in Tabella A.1.

Eseguire i test

Nel progetto di BookingApp ci sono tre moduli che contengono dei test:

- `booking-domain-module` contiene solo unit test.
- `booking-business-module` contiene sia unit che integration test.
- `booking-ui-module` contiene unit, integration ed e2e test.

Per eseguire tutti i test contenuti nel modulo direttamente da Eclipse: fare click con il tasto destro del mouse sul sotto-progetto > **Run As** > **JUnit Test**. Prima, però, è necessario assicurarsi che Docker sia correttamente operativo. Dopodiché, per gli unit test non servono altre configurazioni dal momento che eseguono con Testcontainers; viceversa, per gli integration ed end-to-end test è necessario far partire un'istanza Docker di MongoDB e una di PostgreSQL, tramite i seguenti comandi di Docker Compose, da eseguire nella *directory principale del progetto*:

```
1 docker compose -f docker-compose/MongoDB/docker-compose.
    yml up
2 docker compose -f docker-compose/PostgreSQL/docker-compose
    .yml up
```

Suffisso	**Cosa fa**
`-verify`	Esegue tutti i test.
`-install`	Esegue tutti i test e installa le dipendenze localmente per ogni modulo.
`-test-without -docker`	Esegue i soli test che non richiedono l'uso di Docker.
`-junit-report`	Esegue tutti i test e ne genera i report: per gli unit test in `/target/site/surefire-report.html`, mentre per gli integration ed end-to-end test sono in `/target/site/failsafe-report.html`.
`-jacoco`	Fa la stessa cosa del profilo Maven `-Pjacoco`; se eseguito su un sotto-modulo, i risultati del test coverage si trovano in `/target/site/jacoco/index.html`.
`-pitest`	Fa la stessa cosa del profilo Maven `-Ppitest`; se eseguito su un sotto-modulo, i risultati del test coverage si trovano in `/target/pit-reports/index.html`.
`-pages`	Genera un sito Web statico per il progetto di BookingApp che può essere visitato da `/target/staging/index.html`.
`-docker`	Fa la stessa cosa del profilo Maven `-Pdocker`.
`-without-gui -tests`	Fa la stessa cosa del profilo Maven `-Pskip-gui-tests`.
`-docs`	Genera un archivio jar del codice sorgente e uno del javadoc nella cartella `/target/`.
`-reset -dependencies`	Rimuove le dipendenze di progetto dalla repository locale, utile se ve ne sono di inutili o in conflitto.

Tabella A.1: File di launch del progetto di BookingApp: a sinistra i possibili suffissi, a destra la loro peculiarità nella build.

A.4 Eseguire l'applicazione

È possibile eseguire l'applicazione di BookingApp attraverso il suo file jar o utilizzando la sua immagine Docker. Si ricorda che l'applicazione è compatibile sia con MongoDB che con PostgreSQL, quindi è necessario anche decidere con quale di essi lanciarla.

A.4.1 Tramite FatJar

Una copia del FatJar di BookingApp può essere ottenuta in seguito alla build del progetto, posizionata in `/booking-app/target/`, oppure scaricandola dalla pagina di Releases della repository.

PostgreSQL

Anzitutto è necessario far partire un'istanza Docker di PostgreSQL:

```
docker run -d --name booking-postgres -p 5432:5432 -e
    POSTGRES_DB=<YOUR_DB_NAME> -e POSTGRES_USER=<YOUR_USER
    > -e POSTGRES_PASSWORD=<YOUR_PSDW> postgres:15.3 -N 10
```

dove i segnaposto `<YOUR_DB_NAME>`, `<YOUR_USER>` e `<YOUR_PSWD>` devono essere rimpiazzati con un nome e delle credenziali a piacere per il database, rispettivamente. Anche il valore del parametro `-N 10` può essere personalizzato: esso definisce il massimo numero di istanze dell'applicazione di BookingApp che il database può accettare.

Una volta che l'istanza di PostgreSQL è pronta all'uso, è sufficiente aprire un terminale nella cartella del FatJar, ed eseguire l'applicazione tramite il seguente comando (ancora una volta, i segnaposto devono essere sostituiti coi valori decisi prima):

```
java -jar booking-app-1.1.0-jar-with-dependencies.jar --
    dbms=POSTGRES --host=localhost --port=5432 --name=<
    YOUR_DB_NAME> --user=<YOUR_USER> --pswd=<YOUR_PSWD>
```

MongoDB

L'istanza Docker di MongoDB può essere fatta partire con lo stesso comando di Docker Compose usato per i test; a tal proposito, aprire il terminale nella *directory principale del progetto* ed eseguire:

```
docker compose -f docker-compose/MongoDB/docker-compose.
    yml up
```

Dopodiché, è suffiente aprire un terminale nella cartella del FatJar per eseguirvi il seguente comando:

```
java -jar booking-app-1.1.0-jar-with-dependencies.jar --
    dbms=MONGO --host=localhost --port=27017 --name=<
    YOUR_DB_NAME>
```

dove il segnaposto `<YOUR_DB_NAME>` dev'essere sostituito con un nome a piacere che identifichi il database.

A.4.2 Tramite Docker

L'immagine dell'applicazione di BookingApp è ottenibile dalla build del progetto con il profilo Maven `-Pdocker`. Si ricorda di impostare l'ambiente grafico per Docker prima di procedere con questa modalità.

Per eseguire sia l'applicazione di BookingApp che un'istanza ben configurata del database, è sufficiente posizionarsi nella *directory principale del progetto* ed eseguire il seguente comando di Docker Compose sul terminale:

```
docker compose -f <COMPOSE_FILE> up
```

Per usare PostgreSQL come DBMS è sufficiente sostituire `<COMPOSE_FILE>` con `docker-compose-postgres.yml`; viceversa, per MongoDB, il segnaposto dev'essere sostituito con `docker-compose-mongo.yml`.

WSL Nel caso Docker sia stato fatto partire su WSL, il comando diventa:

```
docker compose -f <COMPOSE_FILE> -f docker-compose-wslg.
    yml up
```

A.5 Impostare l'ambiente grafico per Docker

BookingApp richiede un ambiente X Window System per funzionare adeguatamente. In base al sistema operativo utilizzato, l'ambiente grafico potrebbe o meno essere nativo. Dopodiché, l'X server deve essere reso fruibile a Docker.

A.5.1 Linux

Linux ha già un'ambiente grafico X11, per cui l'unica cosa da fare è di condividerlo con Docker. Il modo più semplice è quello di disabilitare il controllo sugli accessi dell'X server tramite il comando:

```
xhost +
```

A.5.2 Windows

Windows non ha integrato un ambiente grafico X di default, ma può usare quello di **Windows Subsystem for Linux**, compatibile con Windows 10 (Build 19041 o successive) e 11.

Per usare WSLg, è sufficiente installare l'ultima versione di WSL dal Microsoft Store o aggiornarlo se è già installata una versione precedente tramite il comando:

```
wsl --update
```

Dopodiché, è sufficiente impostare la variabile DISPLAY in modo che punti a WSLg tramite il comando:

```
setx DISPLAY :0.0
```

Sitografia

[1] Issue assertj/asserj swing. Add junit-jupiter support. https://github.com/assertj/assertj-swing/pull/265.

[2] Issue assertj/asserj swing. Support for junit 5. https://github.com/assertj/assertj-swing/issues/259.

[3] GitHub: brettwooldridge/HikariCP. About pool sizing. https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing, Dec 2021.

[4] Issue docker-practice/actions-setup docker. Macos: group 'wait docker running' sometimes fails due to timeout. https://github.com/docker-practice/actions-setup-docker/issues/34.

[5] Docker Docs. Extend your compose file. https://docs.docker.com/compose/multiple-compose-files/extends/.

[6] GitHub Docs. About releases. https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases, 2024.

[7] GitHub Docs. Publishing with a custom github actions workflow. https://docs.github.com/en/pages/getting-started-with-github-pages/configuring-a-publishing-source-for-your-github-pages-site#publishing-with-a-custom-github-actions-workflow, 2024.

[8] PostgreSQL Documentation. Connections and authentication. https://www.postgresql.org/docs/current/runtime-config-connection.html#GUC-MAX-CONNECTIONS.

[9] MongoDB Java Driver. Document data format: Pojos. https://www.mongodb.com/docs/drivers/java/sync/v4.11/fundamentals/data-formats/document-data-format-pojo/.

[10] MongoDB Java Driver. Pojo customization. https://www.mongodb.com/docs/drivers/java/sync/v4.11/fundamentals/data-formats/pojo-customization/.

[11] Hibernate ORM User Guide. Session-per-request pattern. https://docs.jboss.org/hibernate/orm/6.4/userguide/html_single/Hibernate_User_Guide.html#session-per-request.

[12] Issue hcoles/pitest. Headlessexception when running tests that open frames. https://github.com/hcoles/pitest/issues/1033.

[13] Issue hcoles/pitest. Tests requiring assertj swing as a maven dependency encounter a noclassdeffounderror under pit. https://github.com/hcoles/pitest/issues/881.

[14] Docker Hub. intland/postgres image. https://hub.docker.com/r/intland/postgres/tags.

[15] Docker Hub. Mongodb official image for windows container. https://hub.docker.com/_/mongo.

[16] Roland Huß. fabric8io/docker-maven-plugin docs - limitations. https://dmp.fabric8.io/#limitations, Feb 2024.

[17] Mat Keep and Henrik Ingo. Performance best practices: Transactions and read / write concerns. https://www.mongodb.com/blog/post/performance-best-practices-transactions-and-read-write-concerns, Nov 2022.

[18] LearningFromExperience. A step-by-step guide to use git tag with eclipse. https://www.youtube.com/watch?v=KTaWgm12s9o, Sep 2022.

[19] MongoDB Manual. Deploying a mongodb cluster with docker. https://www.mongodb.com/compatibility/deploying-a-mongodb-cluster-with-docker.

[20] MongoDB Manual. Initiates a replica set. https://www.mongodb.com/docs/manual/reference/method/rs.initiate/.

[21] MongoDB Manual. Replica set options. https://www.mongodb.com/docs/manual/reference/program/mongod/#replication-options.

[22] Apache Maven Project. Building multi-module sites. https://maven.apache.org/plugins/maven-site-plugin/examples/multimodule.html, Dec 2023.