Shop Totem Application Report
Advanced Techniques and Tools for Software Development

Claudia Raffaelli

Matricola: 7036222
Department of Information Engineering
University of Florence
Via di Santa Marta 3, Florence, Italy
claudia.raffaelli@stud.unifi.it

Francesco Scandiffio

Matricola: 7033974
Department of Information Engineering
University of Florence
Via di Santa Marta 3, Florence, Italy
francesco.scandiffio@stud.unifi.it

***Abstract*—In questo report viene presentato Shop Totem [1], un'applicazione desktop che simula un totem interattivo di un supermercato attraverso cui poter comprare e restituire dei prodotti. Il report descrive il design e discute alcune scelte implementative dell'applicazione, con riferimenti ai patterns e tools utilizzati. Una particolarità di questa applicazione è quella di poter essere avviata, alternativamente, con un database relazionale o con uno non relazionale.**

***Keywords*—TDD, Java, Testing, MongoDB, MySQL, Maven, Docker, CI, Code Quality.**

I. Introduzione

Shop Totem è un'applicazione desktop sviluppata in Java 1.8 che simula un sistema di acquisto di prodotti in un supermercato. L'implementazione replica alcuni vincoli del mondo reale, come ad esempio limitare la quantità di prodotti acquistabili in base allo stock disponibile in magazzino (il database). L'applicazione può usare sia un database relazionale che un database non relazionale, nello specifico MySQL e MongoDB. Il sistema fa anche uso dei seguenti patterns:

- *Model-View-Controller*: definisce le interazioni tra vista e Service Layer
- *Repository*: nasconde l'implementazione concreta dei metodi utilizzati per accedere ai dati salvati sul database.
- *Service Layer*: implementa la logica di manipolazione dei dati e delle classi di modello
- *TransactionManager*: regola le transazioni, confinando al loro interno tutte le operazioni che richiedono l'accesso al database.

II. Tools

Il sistema è stato prodotto come un progetto Maven e sviluppato in Java 8, utilizzando Eclipse come IDE. In esso abbiamo utilizzato i seguenti tools:

- **Hibernate**: per la gestione delle interazioni con il database relazionale.
- **Docker**: per l'avvio dei containers con database.
- **PIT**: come strumento per la generazione di mutanti.
- **JBehave**: come framework per BDD.

[1] https://github.com/FrancescoScandiffio/shop-totem

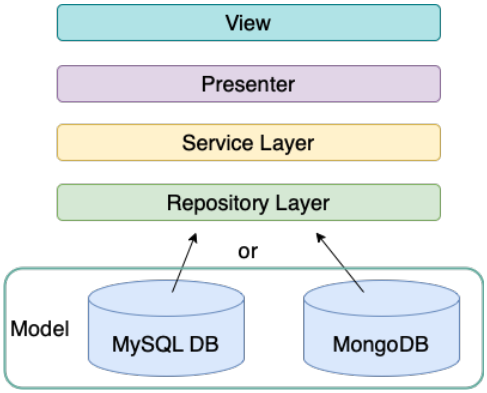

Figure 1: Overview progetto.

- **GitHub Actions**: come servizio per la Continua Integrazione.
- **SonarCloud e Coveralls**: come strumento di code quality e copertura del codice.

III. System design

In questa sezione sono descritte le singole componenti dell'applicazione.

A. Domain Model

Il Domain Model è composto da quattro classi che corrispondono a quattro tipi di entità persistibili sul database: Product, Stock, Order e OrderItem (Fig. 2).

- ***Product***: contiene le informazioni su nome e prezzo di un prodotto.
- ***Stock***: contiene l'informazione sulla quantità disponibile in magazzino in riferimento ad un prodotto. Ogni Stock ha un riferimento ad un solo Product e non possono esistere più Stock per lo stesso prodotto. La relazione instaurata è di tipo 1:1 unidirezionale.
- ***Order***: serve a tenere traccia dei prodotti acquistati durante una sessione di shopping. L'ordine può avere due stati, definiti attraverso una Enumeration: aperto o chiuso.
- ***OrderItem***: rappresenta un prodotto all'interno del carrello del cliente, con informazioni circa la quantità acquistata e il costo. Ogni OrderItem fa riferimento ad

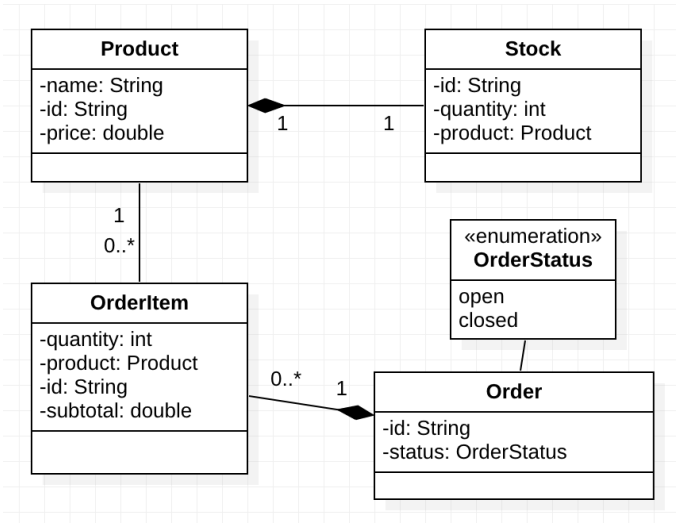

Figure 2: Class diagram.

un solo Product ma clienti diversi possono ovviamente acquistare lo stesso prodotto: la relazione è dunque many to one. La classe possiede anche un attributo subtotal utilizzato per mostrare il costo di un oggetto nel carrello in relazione alla quantità acquistata. Poiché il valore di *subtotal* è calcolabile dalla quantità nel carrello e dal prezzo del prodotto, informazioni reperibili, sarebbe ridondante salvarlo sul database: per questo motivo il campo è transiente.

Per questo progetto è stato utilizzato un modello anemico, ovvero privo di funzioni con una qualche logica. Le classi delle Entità sono dei semplici contenitori di dati mentre la logica di business è implementata altrove.

B. Repository Layer

Come anticipato, Shop Totem può essere avviata scegliendo tra due database di tipo diverso: uno relazionale (MySQL) e uno non relazionale (MongoDB). Un sistema del genere richiede una adeguato livello di astrazione, motivo per cui è stato implementato il Repository Pattern. Abbiamo prima definito delle interfacce pubbliche per esporre i metodi di accesso ai dati sul database e poi ne abbiamo realizzato due implementazioni specifiche, una per ogni database.

C. Transaction Manager

Un transaction manager nasce dall'esigenza di permettere l'esecuzione sul database di un insieme raggruppato di operazioni, che prende il nome di *database transaction*. Dunque una transazione simbolizza una unità di lavoro eseguita su un database. Le transazioni sono necessarie per fare in modo che:

- Programmi o parti di programma che occedono al database in modo concorrente lo facciano in isolamento.
- Operazioni multiple appartenenti a diverse transazioni vengano svolte in modo consistente e indipendente.
- Se si ha un fallimento in almeno un'operazione all'interno di una transazione, tutte le operazioni precedenti vengono annullate: si ha il rollback. Pertanto si dice che le transazioni sono atomiche. Senza questa caratteristica, se l'esecuzione si ferma in modo prematuro o inaspettato con operazioni non portate a termine, il database rimane in uno stato non chiaro.

Dunque se una transazione viene eseguita con successo nella sua totalità verrà salvata, *committed*, sul database ed i cambiamenti avranno effetto. Altrimenti, viene eseguita l'azione di rollback, ripristinando lo stato pre-transaction.

D. Service layer

Il service layer ShoppingService contiene la logica di validazione dell'applicazione, controllando che gli oggetti con cui fare operazioni sul database siano validi. Controlla anche che le richieste di acquisto e rimozione di prodotti da parte dell'utente siano possibili date le attuali quantità in magazzino. Interagisce con il Transaction Manager per la gestione delle operazioni con il database. Inoltre supponiamo che i Prodotti, una volta inseriti all'interno del database non verranno più modificati. Per quanto riguarda gli Stock, la quantità viene modificata unicamente ogni qualvolta un utente inserisce o rimuove un prodotto dell'ordine.

E. Controller layer

In ShopTotem il controller prende il nome di TotemController. Esso è l'unico soggetto ad interagire con la vista, dalla quale riceve dei messaggi "triggerati" da azioni dell'utente. La vista non è dotata di una logica complessa, pertanto è il controller che, ogni qualvolta riceve una comunicazione dalla vista, risponde fornendo istruzioni su cosa fare. Per assolvere alle richieste della vista TotemController comunica con ShoppingService che fa da intermediario con le repository e gestisce le transazioni.

F. GUI

L'interfaccia grafica è stata realizzata in Swing ed è composta da 4 pannelli navigabili attraverso dei pulsanti:

- *Welcome page*. È il pannello iniziale, contiene solo un pulsante con cui poter avviare una sessione di shopping.
- *Shopping page*. Espone una lista dei prodotti presenti nel database, di cui viene mostrato nome e prezzo, ed elementi interattivi con cui l'utente può specificare quale tra i prodotti mostrati vuole acquistare e in quale quantità.
- *Cart page*. Questo pannello mostra un riepilogo dei prodotti che l'utente desidera acquistare. Questa pagina permette di confermare o annullare l'acquisto e di restituire i prodotti. È anche possibile tornare alla pagina di acquisto e inserire ulteriori prodotti nel carrello.
- *Goodbye page*: successivamente al checkout l'utente viene condotto alla pagina di goodbye che conferma l'acquisto degli oggetti nel carrello. Qui è presente solo un pulsante con cui poter avviare un nuovo ordine.

I pannelli dell'applicazione sono mostrati in Fig. 3.

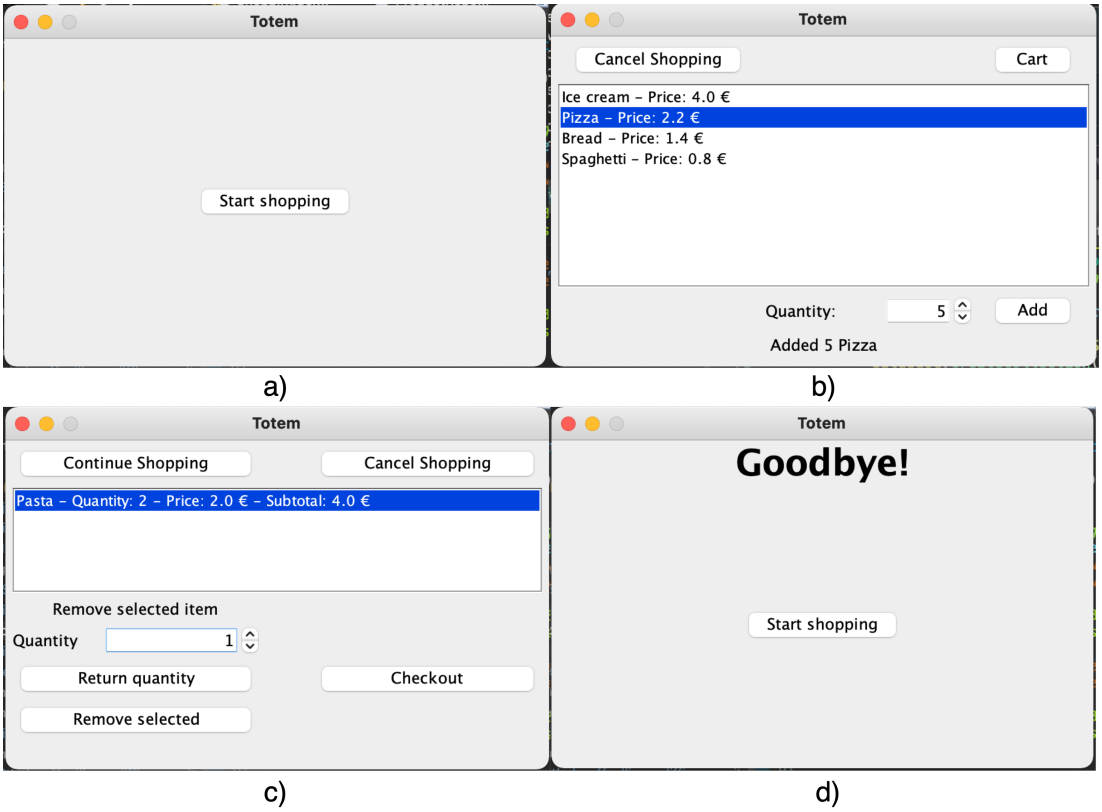

Figure 3: Le schermate: welcome a), shopping b), cart c) e goodbye d).

IV. Dettagli di implementazione

Come anticipato, la particolarità di questa applicazione è quella di poter essere eseguita utilizzando due database molto diversi tra loro, MySQL e MongoDB. Sebbene i patterns precedentemente elencati siano stati fondamentali per il raggiungimento di un buon livello di astrazione, le differenze intrinseche di questi due diversi database hanno più volte condizionato i dettagli implementativi, a partire dalla struttura delle classi di modello.

A. Assegnazione degli ID

Lo scenario simulato da questa applicazione non permette che sia l'utente ad assegnare manualmente id agli oggetti. L'apertura di un nuovo ordine e l'inserimento di prodotti nel carrello sono operazioni di responsabilità dell'applicazione stessa, che deve quindi essere in grado di generare un id in modo automatico. In generale, questo può essere fatto a livello applicativo o a livello di database. Avendo in mente anche una implementazione multi-istanza, abbiamo optato per un id generato a livello di database (o più precisamente generato dal Driver di interfaccia col database). Dopo aver scelto la strategia di generazione è stato necessario individuare un Java type che fosse compatibile sia per MySQL che per MongoDB. La scelta è ricaduta su String, tipo in grado di rappresentare sia UUID, generati attraverso Hibernate, che gli ObjectId di Mongo. Una scelta alternativa avrebbe potuto essere quella di usare un array di bytes, ma la differenza di dimensioni tra UUID e ObjectId - rispettivamente 16 e 12 bytes - e la maggiore difficoltà di debug che ne sarebbe seguita ci hanno fatto preferire String. In generale nei database relazionali è preferibile evitare di avere un UUID come chiave primaria, per motivi di efficienza. La soluzione più performante sarebbe quella di avere un intero come chiave primaria e un UUID come secondaria, da utilizzare solo per effettuare riferimenti al di fuori del database. Tuttavia il nostro interesse principale è stato quello di mantenere compatibilità fra l'implementazione adottata per MongoDB e MySQL, accettando di rinunciare a varie ottimizzazioni.

B. Persistence xml

Hibernate è un tool molto utile per gestire l'interazione tra codice Java e database relazionali, ma per poter funzionare richiede una adeguata configuazione. Tale configurazione viene specificata nel persistence.xml che va collocato nella cartella `resources/META-INF`

Nel caso di database MySQL, Hibernate prevede l'utilizzo della EntityManager API, per accedere al database. Esso è utilizzato per creare e rimuovere delle entità persistenti, per trovare entità con la chiave primaria e in generale per eseguire query sulle entità. Il persistence.xml fornisce la flessibilità per configurare adeguatamente l'EntityManager. Ciò avviene con la definizione di una persistence-unit. Questa fornisce tutte le informazioni che permettono la connessione al database. Sono presenti anche ulteriori dettagli, come il dialect da utilizzare

(nel nostro caso MySQL8Dialect) per la comunicazione con il database, il driver per la connessione al database, ed il metodo di gestione del database ad ogni avvio. Infatti è possibile specificare, per citare alcuni esempi, se aggiornare le tabelle ad ogni avvio, se crearne di nuove, se eliminare le eventuali presenti e ri-crearne di nuove.

ShopTotem dispone di due persistence files, uno dedicato ai test ed uno alla produzione. Entrambi sono dotati di un tag `<mapping-file>` che va ad indicare la posizione dei file di annotazione delle classi di modello, così che possano essere recuperati per la mappatura.

C. Hibernate XML mapping

JPA definisce un processo di mappatura delle classi Java in tabelle relazionali basato su annotazioni. Questa funzionalità permette di modellare facilmente il database relazionale a partire dalle specifiche di dominio Java, semplicemente applicando delle annotazioni su campi o metodi delle classi da mappare. Nel nostro caso, dovendo lavorare anche con MongoDB, non è stato possibile applicare le annotazioni all'interno delle classi Java. Per ottenere lo stesso risultato abbiamo dovuto dunque definire un file XML di configurazione, detto appunto *Hibernate XML mapping*. Questo tipo di mappatura ci ha permesso di mantenere il codice di produzione totalmente indipendente dal tipo di database utilizzato, al prezzo di una maggiore difficoltà di configurazione rispetto alla versione con annotazioni. Ad ogni classe persistibile sul database è associato un file di configurazione con estensione `.hbm.xml`, tutti contenuti all'interno della cartella `/src/main/resources/META-INF`.

Abbiamo realizzato quattro XML mapping files, uno per ciascuna classe di modello. Ne mostriamo, uno quello di OrderItem.

OrderItem Hibernate XML mapping file:

```
<hibernate-mapping>
  <class name="OrderItem" table="ORDER_ITEMS">
    <id name="id" type="string" access="field">
      <generator class="uuid2"/>
    </id>
    <property name="quantity" column="quantity" type
    ="int" access="field"/>
    <many-to-one name="product" class="Product"
    access="field"></many-to-one>
    <many-to-one name="order" class="Order" column="
    order_id"  access="field"></many-to-one>
  </class>
</hibernate-mapping>
```

Osserviamo che l'elemento `<id>`, mappa un attributo ID univoco nella classe alla chiave primaria della tabella di database. L'elemento `<generator>`, invece viene usato per specificare la generazione automatica dei valori di chiave primaria. In particolare viene impostata la strategia di generazione. Inoltre come già definito, un oggetto OrderItem possiede un riferimento ad un oggetto Product, di cui ne rappresenta l'istanza nell'ordine. Il tag `<many-to-one>` vinee usato proprio per impostare la relazione fra le entità OrderItem e Product.

D. Transaction Manager

Dal punto di vista teorico, il concetto di transazione è indipendente dal database utilizzato. Si tratta di una porzione di codice con cui è possibile eseguire un insieme di operazioni da applicare sul database con la filosofia del *o tutto o niente*: o tutte le operazioni si concludono con successo oppure nessuna modifica deve essere confermata sul database. Questa equivalenza concettuale ha favorito l'astrazione, permettendoci di definire una comune interfaccia con cui mascherare le implementazioni concrete. Il transaction manager è stato realizzato con le seguenti componenti:

- La `@FunctionalInterface TransactionCode`, i.e. una interfaccia con un solo metodo astratto. Questo metodo non è altro che una lambda function, la quale contiene il codice che dovrà essere eseguito all'interno della transazione.
- L'interfaccia pubblica `TransactionManager` contiene il metodo `runInTransaction` che al suo interno può eseguire del codice definito come `TransactionManager`.
- Le due implementazioni di `TransactionManager` per Mongo e MySQL. In particolare:
  - Per MySQL viene fatto uso della gestione di transazioni offerta dal `JPA EntityManager`: questo oggetto consente l'accesso ad un persistence context su cui vengono eseguite le operazioni in attesa del commit che le applicherà effettivamente sul database. L'`EntityManager` viene iniettato nelle mysql repositories dal `TransactionManager`.
  - Mongo utilizza invece una `ClientSession`, un oggetto che isola le operazioni di scrittura rendendole visibili solo ai possessori della stessa session. Come per l'`EntityManager`, anche la sessione viene iniettata alle mongo repositories che la useranno durante le query sulle collezioni.
- Una `TransactionException`, estensione di `RuntimeException`. Viene lanciata dall'entity manager per notificare gli altri layer che si è verificata una rollback.

Notiamo che, seppur i test di `TransactionManager` lo riguardino esclusivamente, dovendo utilizzare delle implementazioni reali dei database verranno da noi considerati non come unit tests, ma più propriamente come integration tests.

E. Vincoli e gestione degli errori

L'applicazione è stata implementata seguendo il domain model anemico, che prevede la non inclusione di business logic all'interno di domain objects. Pertanto azioni come validazioni degli argomenti sono tutte eseguite dal service layer. Inoltre i valori che potrebbero essere inseriti all'interno di classi di modello, derivanti da input dell'utente rappresentano ovviamente una criticità. Nel nostro caso l'utente può inserire una quantità richiesta di prodotto da acquistare, ed una quantità di prodotto da restituire che deve essere rimossa dal carrello. Entrambi questi input sono validati direttamente sulla vista, facendo uso dell'oggetto `JSpinner`. Infatti, allo spinner può

essere associato un modello, dotato di vincoli sui valori minimi, che non possono essere inferiori ad 1. Questo garantisce che non possano essere richiesti degli acquisti negativi o di valori zero. Allo stesso modo è garantito che vengano accettati solo numeri grazie all'implementazione di un filtro RegEx. In ogni caso non è possibile per l'utente richiedere aggiunte e rimozioni con valori non validi poiché su spinner è aggiunto un listener che attiva e disattiva i pulsanti in base al contenuto immesso. I listener sono anche posti a rilevare la selezione e deselezione del contenuto delle liste e ad agire di conseguenza. Ci siamo assicurati che questo fosse il comportamento atteso attraverso i test di unità.

Questo non impedisce all'utente di richiedere l'acquisto di prodotti che però non sono disponibili in magazzino. In tal caso il controller, comunicando con il service layer, metterà a conoscenza la vista della assenza della disponibilità richiesta per il prodotto selezionato. Degli errori sono mostrati anche in tutti quei casi in cui si verifichi un'eccezione che ha una rilevanza per l'utente.

F. JUnit5 GUITestExtension

Utilizzando JUnit5 come unit testing framework abbiamo dovuto rinunciare all'uso del `GUITestRunner` per l'esecuzione dei test d'interfaccia. Abbiamo dunque fatto ricorso ad una `GUITestExtension` [2]. Proprio come il `GUITestRunner` di JUnit4, questa estensione è in grado di effettuare degli screenshot dei failed GUI tests. Il setup ha richiesto qualche impostazione manuale, quale la registrazione della extension, annotando la classe di test con `@ExtendWith(GUITestExtension.class )` e l'istallazione di `FailOnThreadViolationRepaintManager` prima dell'inizio dei test. Questo ultimo controlla che tutto l'accesso alle componenti Swing avvenga nell'EDT.

G. GitHub Repository

Come Version Control System abbiamo adottato GitHub, il che ci ha anche permesso di operare con una collaborazione più efficiente. Per ogni nuova feature aggiunta e modifica alla codebase abbiamo quindi creato nuovi branch, integrati sul main branch attraverso Pull Requests (PR). Abbiamo inoltre sperimentato con l'apertura di Issues e relativa chiusura con PR. Inoltre, grazie a GitHub abbiamo potuto accedere allo strumento di Continua Integrazione, ottimo per l'esecuzione di build automatizzate ed esecuzione di test.

H. Continuous Integration (CI) using GitHub Actions

Per la Continuous Integration (CI) abbiamo utilizzato il servizio offerto da GitHub: GitHub Actions. Per utilizzare i processi automatizzati di GitHub Actions è necessario configurare almeno file YAML di workflow. I file `.yml` di configurazione devono essere posti all'interno di un'apposita cartella `.github/worflows`, localizzata nella project root directory. Per la nostra applicazione abbiamo realizzato due workflows che definiamo di seguito:

- **maven_ubuntu.yml** è il workflow principale. E' eseguito ogni qualvolta viene eseguita un'azione di push o pull request sul repository, eccetto nei casi in cui la modifica apportata riguardi unicamente file non relativi al processo di build (il README, la licenza e simili).
- **maven_macos.yml** è un workflow secondario, triggerato sulle sole azioni di pull request. E' configurato per l'utilizzo di Java8 e si occupa unicamente dell'esecuzione dei test di unità e integrazione. inizialmente questo workflow utilizzava matrix strategy per testare il codice sia su MacOS che su Windows. Tuttavia, non potendo disporre anche su Windows della stessa immagine Docker che utilizziamo per gli altri sistemi operativi, abbiamo deciso di rimuovere questo SO dal workflow.

Entrambi i workflow sfruttano il sistema di caching delle dipendenze di Maven in modo da velocizzare il processo di build. Il file principale è diviso in più step. Ripercorriamo brevemente i più salienti:

1) Sono eseguiti i test di unità con il comando *xvfb-run*. Questo permette di eseguire l'applicazione in un ambiente "headless".
2) Esegue i mutation test di PIT solo in caso di pull requests e unicamente se l'outcome al passo precedente, riguardante l'esecuzione dei test di unità va a buon fine. Non avrebbe infatti senso eseguire questa tipologia di test su una codebase con test di unità non-green.
3) Sono eseguiti i test di integrazione, anche questi attivando lo schermo virtuale. Facciamo notare che tra i parametri forniti al comando `mvn verify` ne è presente uno - `<skipTests>` - che permette di non eseguire i test di unità. Questa scelta è motivata dal fatto che i test di unità sono già stati eseguiti in uno step precedente, quindi non avrebbe senso ripeterli nuovamente.
   Inoltre facciamo notare che in CI non eseguiamo i test E2E con BDD, per problemi relativi alla configurazione di Hibernate. Viene infatti segnalato un errore per cui la persistence-unit di produzione non viene individuata, il che fa fallire i test. Eseguito in locale il processo di build non segnala invece errori. Durante lo step che esegue il verify, viene anche verificata la quality code con SonarCloud e generato il report di code coverage di Coverall. Sia SonarCloud che Coverall sono collegati a GitHub Actions grazie all'utilizzo di appositi token, salvati come GitHub Secret. In questo workflow, come anticipato facciamo uso di Java11 anziché 8: questo è necessario perché SonarCloud richiede Java11 per essere eseguito.
4) Negli step finali vengono generati i report relativi ai singoli step di test. Questi report sono generati solamente se si verifica un fallimento durante l'esecuzione dei test o la build del progetto. Qui sono salvati anche gli eventuali screenshots prodotti durante i test falliti legati all'interfaccia. Se nessuno screenshot è stato prodotto, questo step viene ignorato grazie al comando `if-no-files-found: ignore`, il quale sopprime il warning legato alla non presenza di nessun file da archiviare nella cartella definita.

[2]https://github.com/assertj/assertj-swing/issues/259

I. Testcontainers vs Docker containers with Maven build

Inizialmente i test relativi ai database facevano uso di Testcontainers. Tuttavia, poiché non è facile configurare Testcontainers per poter eseguire i test E2E in stile BDD, abbiamo deciso di delegare l'avvio e la distruzione dei containers al pom, in particolare alle fasi di pre-integration-test e post-integration-test. Riguardo alle Docker images utilizzate, abbiamo selezionato le seguenti:

- Per MySQL l'immagine ufficiale
- Per MongoDB[3] avevamo bisogno di una immagine che implementasse un Replica-set avviabile in locale. Un replica set in MongoDB è un gruppo di processi mongod che mantengono uno stesso insieme di dati.

Come già detto Maven ci permette di automatizzare il processo di build e testing, ma in produzione è necessario avviare manualmente i database prima dell'applicazione. Vediamo due modi per farlo.

- *Docker compose*: nella root folder del progetto è presente un `docker-compose.yml` che permette di avviare con un unico comando sia l'immagine di MySQL che di MongoDB, esponendo automaticamente le porte necessarie alla nostra applicazione.
- *Avviare manualmente i containers*:
  - MySQL:

```
docker run -d -p 3306:3306 -e MYSQL_DATABASE
    ="totem" -e MYSQL_ROOT_PASSWORD="" -e
    MYSQL_ALLOW_EMPTY_PASSWORD="yes" mysql
    :8.0.28
```

  - MongoDB:

```
docker run -d -p 27017:27017 -p 27018:27018
    -p 27019:27019 candis/mongo-replica-set
```

J. Concorrenza

ShopTotem presenta due tipi di multithreading:

- Un multithreading dettato dalle interazioni dell'utente con l'interfaccia. Le azioni da intraprendere per soddisfare le richieste da parte dell'utente, derivanti ad esempio dal click su un pulsante, vengono gestite su un Thread separato. In questo modo l'applicazione risulta sempre responsive.
- Un multithreading multi-istanza ovvero predisposto ad avere più istanze della stessa applicazione che eseguono operazioni concorrenti sullo stesso database. La sincronia a livello di database è gestita dalle transazioni già descritte.

Purtroppo questo tipo di multithreading è stato raggiunto solo su MySQL per i motivi spiegati di seguito.

1) MySQL: In Hibernate le transazioni sono gestite attraverso l'EntityManager che effettua un rollback automatico qualora si verificasse un'eccezione all'interno della transazione. Il multithreading multi-sessione è stato verificato eseguendo su Threads separati più istanze della stessa applicazione, isolate fra loro, ma simulanti le stesse operazioni.

Questo tipo di concorrenza è stato reso possibile dall'utilizzo di Lock di tipo `PESSIMISTIC_WRITE`: si tratta di un lock esclusivo che impedisce le *non-repeatable reads*, ovvero il fenomeno per cui due letture dello stesso dato all'interno della stessa transazione restituiscono valori diversi.

2) MongoDB: Le transazioni di Mongo su un singolo documento sono sempre atomiche e non necessitano di una gestione aggiuntiva, mentre le transazioni multi-documento necessitano di una adeguata configurazione. Nello scenario proposto dalla nostra applicazione le operazioni interessano quasi sempre più di un documento, anche in collezioni diverse.

Come per MySQL era nostra intenzione ottenere un multithreading multi-sessione anche per Mongo. La strada più promettente, come da documentazione[4], è riguarda l'utilizzo di una Write Concern *majority* in combinazione con una Read Concern *linearizable*. Queste due insieme consentono una effettiva sequenzializzazione delle transazioni, come se ci fosse solo un Thread ad eseguirle. Purtroppo non è stato possibile applicare il Concern *linearizable* a causa di un errore che si presentava al momento del loro inserimento:

> The readConcern level must be either 'local' or 'majority'

K. Behaviour Driven Development (BDD)

Come strumento di Behaviour Driven Development (BDD) abbiamo adottato JBehave. Esso permette di specificare un file testuale di storie, il quale esprime un comportamento da verificare. Ad ogni storia è collegato almeno uno scenario composto a sua volta da una collezione di steps.

L. Come avviare ShopTotem

Per prima cosa è necessario far partire i database utilizzando un metodo dei due esposti in *Testcontainers vs Docker containers with Maven build*. Dopodiché è sufficiente eseguire `mvn verify`.

[3]https://github.com/CandisIO/mongo-replica-set

[4]https://www.mongodb.com/docs/manual/reference/read-concern-linearizable/