Advanced Technique And Tools For Software Development

Sviluppo di una web application di un semplice e-commerce per la gestione di utenti e ordini

Jacopo Vezzosi Mat. 7103062

Prof. Lorenzo Bettini

Dipartimento di Matematica e Informatica "Ulisse Dini"
Università degli Studi di Firenze
A.A. 2024/2025

1 Applicazione

Ho sviluppato un sistema completo per la gestione di un e-commerce utilizzando Spring Boot. Il progetto si basa su due entità: User e Order.

L'entità User ha come attributi:

- id - Long, generato automaticamente in modo auto-incrementale
- username - String
- name - String
- email - String
- balance - long

L'entità Order ha come attributi:

- id - Long, generato automaticamente in modo auto-incrementale
- item - Enum (BOX1, BOX2, BOX3), non nullo
- price - long
- user - User, referenza esterna dell'utente associato all'ordine

Per la gestione dei dati ho implementato due repository JPA, ciascuna per ogni entità (User e Order).

La logica di business è stata incapsulata in un unico service layer, che gestisce operazioni standard per entrambe le entità: aggiunta e aggiornamento di record; ricerca in base ad alcuni criteri. Inoltre, un User ha a disposizione i metodi Withdraw e Deposit per effettuare operazioni sul proprio Balance; mentre un Order utilizza questi metodi appena citati per effettuare operazioni di inserimento e modifica, infatti, l'inserimento di un Order è possibile quando l'User associato ha abbastanza Balance per poterlo effettuare, concettualmente equivalente è l'update di un Order, dove, prima di effettuare il Withdraw sul Balance dello User associato al nuovo Order, si procede effettuando un Deposit sul Balance dello User associato all'ordine già esistente. Inoltre, il service lancia eccezioni con messaggi opportuni che vengono poi gestite nei Controllers (Web e Rest). Le eccezioni in questione riguardano le seguenti casistiche:

- Eseguire Withdraw/Deposit con Amount negativo

- Eseguire Withdraw/Deposit con Id inesistente
- Eseguire Withdraw con Balance < Amount
- Eseguire Insert/Update di un Order che non va a buon fine a causa di una delle operazioni di Withdraw/Deposit

L'applicazione è accessibile sia tramite API REST, utili per integrazioni e test automatici, sia tramite interfaccia web, che consente di effettuare le operazioni descritte nel service layer. Come detto in precedenza, entrambi i Controllers gestiscono le eccezioni tramite ExceptionHandler restituendo risposte opportune.

2 Tecniche e Framework

Per lo sviluppo dell'applicazione, è stato seguito l'approccio Test-Driven Development (TDD), scrivendo prima i test unitari e poi implementando il codice necessario per farli passare, eseguendo qualche "big step", dato che alcuni comportamenti tra User e Order sono condivisi ed i test erano simili tra di loro. Dopo i test unitari, sono stati eseguiti quelli di integrazione tra le componenti, considerato che i test unitari erano solidi ed esaustivi, il comportamento dei test di integrazione era quello atteso e non ci sono state grosse problematiche nella loro implementazione. Infine per E2E tests mi sono concentrato sulle azioni eseguibili partendo sempre dalla home page, anche per quest'ultimi non ci sono stati grossi problemi, fa eccezione la loro esecuzione in Continuous Integration dove ho notato alcune problematiche sul caricamento delle pagine, per rimediare sono state inserite delle Wait opportune.

Di seguito si riportano i framework utilizzati:

- **Ambiente di sviluppo**
  - Eclipse (Spring Tools for Eclipse), Maven
  - Spring Boot
- **Docker & Database**
  - Spring Data JPA + Hibernate
  - PostgreSQL, MySQL, MariaDB
  - H2 (database in-memory)
  - Docker, Docker Compose, Docker Maven Plugin
  - Spring Profiles (configurazione dinamica database)
- **Web e View**
  - Thymeleaf
  - HTML
- **Testing (Unit, Integration, E2E)**
  - JUnit 4 (con JUnit Vintage Engine)
  - Mockito, AssertJ, Hamcrest, JSONAssert, JsonPath
  - Selenium, HtmlUnit

  - JUnitParams[1] (test parametrizzati)
  - Spring Boot Test, Spring Test
- **Quality Assurance & Code Analysis**
  - PIT (Mutation Testing)
  - Jacoco (Code Coverage)
  - SonarQube, SonarCloud
  - Coveralls
- **Version Control & Continuous Integration**
  - Git, GitHub
  - GitHub Actions (Continuous Integration)

[1] JUnitParams

3 Design, Sviluppo e Testing

Nel design dell'applicazione ho dato priorità a modularità e facilità di manutenzione. L'adozione dell'architettura Controller → Service → Repository risponde all'esigenza della separazione delle responsabilità: il controller orchestra le richieste e l'interazione con il sistema, il service gestisce la logica applicativa, mentre le repository si occupano unicamente dell'accesso e della persistenza dei dati.

Lo sviluppo è proceduto seguendo un approccio bottom-up: ho iniziato implementando il livello di persistenza con le repository, per poi costruire il livello service contenente la logica di business, e infine ho realizzato i controller che espongono le funzionalità all'esterno. Questa progressione ha permesso di testare e consolidare ogni strato prima di procedere con quello successivo. Come si nota dalle varie Pull Request su GitHub, ci sono state alcune problematiche durante lo sviluppo: a parte dai metodi di Insert e Update di un Order, i quali non erano stati implementati correttamente, infatti durante l'Update i Balance non venivano modificati, successivamente i metodi sono stati corretti catturando anche le eccezioni lanciate da Withdraw/Deposit e rilanciando un'eccezione con un messaggio opportuno. Inoltre inizialmente erano presenti due Service, ma dato che uno dipendeva dall'altro, tutta la logica è stata inserita in un'unico EcommerceService, i Controllers sono stati modificati di conseguenza. Lato test non ci sono state differenze cruciali, infatti la logica rimaneva invariata ma spostata in un unico Service, questo ha comportato che, gradualmente venissero spostati i metodi ed i corrispettivi tests in un'unica classe modificandoli opportunamente: lato OrderService, era presente un'istanza Mock di UserService, di conseguenza, dopo il merge in unico service, le chiamate ai metodi che prima erano di un'istanza Mock, vengono *stubbate* da un'istanza Spy del nuovo unico EcommerceService. Nello specifico, queste chiamate vengono stubbate perché riguardano i metodi Withdraw e Deposit che hanno già una suite di test esaustiva.

Dal punto di vista del testing, ogni componente dell'applicazione è stata testata in isolamento seguendo la tecnica del Test-Driven Development (TDD). In particolare, gli unit test sono stati progettati sfruttando la dependency injection, che permette di fornire a ciascun componente solo le dipendenze necessarie, sostituendo eventualmente le altre con istanze Mock o Spy. In questo modo, ogni classe viene testata indipendentemente dal comportamento delle altre, garantendo che la logica interna funzioni correttamente.

Gli Integration Test, invece, verificano la corretta collaborazione tra i diversi componenti, interagendo con un Docker container contenente un database reale a

scelta tra Postgresql, Mysql o Mariadb (i test sono stati eseguiti con tutti i database citati, sia in locale che nella build in CI). Il container viene avviato automaticamente nella fase di pre-integration test fermato nella fase di post-integration test.

Infine, negli test end-to-end (E2E), l'applicazione web viene avviata automaticamente dopo aver avviato il container, vengono eseguiti scenari completi che simulano l'interazione reale dell'utente con l'interfaccia: partendo dalla home page, si effettuano creazioni e aggiornamenti di Users, gestione del balance; creazione e aggiornamenti di Orders. In questi test si pone una problematica sull'ordine di esecuzione (avvio/stop) del container e dell'app, infatti, l'esecuzione segue la sequenza della build principale, tuttavia è necessario che il container si avvii prima dell'app, però questo implicherebbe anche che si arresti dopo, ma non è l'effetto voluto, l'esecuzione desiderata è: Avvio container, Avvio spring boot app, Stop spring boot app, Stop container, perciò si è deciso di sovrascrivere lo stop del container legandolo alla fase di verify.[2]

[2]L'alternativa era lasciarlo legato alla post-integration, però questo comportava che i log facessero notare l'assenza di connessione al container.

4 Tecnologie High Rating e Difficoltà Riscontrate

Tra le "normali" difficoltà incontrate nello sviluppo del progetto, se ne riportano alcune degne di nota inserendo una spiegazione esaustiva dove necessario.

4.1 Test Parametrici

Una difficoltà riscontata riguarda l'utilizzo di test parametrici. In JUnit 4, i test parametrici nativi avevano l'effetto di eseguire tutti i test della classe con i parametri specificati, anche quelli non interessati, perciò ho deciso di utilizzare JUnitParams come libreria esterna che, oltre ad evitare il comportamento appena detto, ha anche una definizione dei parametri più semplice e intuitiva. Tuttavia, le classi coinvolte dovevano usare come Runner JUnitParams e JUnit 4 non permette Runner multipli, questo ha comportato delle problematiche nelle classi seguenti classi di test: EcommerceServiceWithMockitoTest e EcommerceWebControllerTest. Dopo una esaustiva ricerca su internet per una soluzione ai Runner di Spring e Mockito non più utilizzabili, ho trovato implementazioni alternative ed equivalenti nelle documentazioni ufficiali che riporto qui sotto:

- Mockito Rule è usato in alternativa a MockitoJunitRunner[3]
- SpringRunner è an alias per SpringJUnit4ClassRunner[4]
- Sostituire lo SpringRunner tramite le Rule[5]

4.2 Relations and Transactions

Per la gestione della persistenza dei dati su database SQL, ho adottato JPA. Le entità User e Order sono state annotate con le specifiche JPA (@Entity, @Table, @Id, @GeneratedValue) per mappare le classi Java alle corrispondenti tabelle del database. La relazione tra le due entità è stata modellata attraverso l'annotazione @ManyToOne, che permette di associare più ordini a un singolo utente, con FetchType.EAGER per garantire il caricamento immediato dei dati dell'utente quando viene recuperato un ordine.

Per l'accesso ai dati ho utilizzato Spring Data JPA, che fornisce un'astrazione di alto livello attraverso l'interfaccia JpaRepository. Estendendo questa interfaccia

[3]Documentazione MockitoRule
[4]Documentazione SpringRunner
[5]Documentazione SpringClassRule

nelle repository UserRepository e OrderRepository, ho ottenuto automaticamente i metodi CRUD base e la possibilità di definire query aggiuntive in modo automatico come findByUsername e findByUsernameOrEmail, inoltre è stata utilizzata anche l'annotazione @Query per quelle più complesse. L'implementazione concreta è fornita da Hibernate, che traduce le operazioni sugli oggetti Java in query SQL.

Un aspetto fondamentale dell'applicazione è la gestione delle transazioni, implementata attraverso l'annotazione @Transactional di Spring. Nei metodi insertNewOrder e updateOrderById del service layer, le transazioni garantiscono che operazioni multiple (come l'aggiornamento del Balance di User e la save dell'Order) vengano eseguite atomicamente: o tutte le operazioni vanno a buon fine, o in caso di errore viene effettuato il rollback completo, mantenendo la consistenza dei dati nel database.

Il comportamento transazionale è stato verificato in modo esaustivo negli integration test testando le varie casistiche possibili. La difficoltà in questi test è stata capire come procedere, dato che i test annotati con @DataJpaTest sono nativamente transazionali, quindi per testare correttamente i meccanismi di rollback è stato necessario disabilitare questa caratteristica utilizzando @Transactional(propagation = Propagation.NOT_SUPPORTED)[6], altrimenti, il rollback veniva eseguito alla fine del test, e questo rendeva impossibile fare asserzioni sui risultati. In definitiva, si è verificato che, in caso di fallimento durante l'inserimento o l'aggiornamento di un ordine (ad esempio per vincoli di integrità violati o saldo insufficiente), tutte le modifiche apportate al database venissero correttamente annullate, ripristinando lo stato iniziale.

4.3 Spring Profiles

Per quanto riguarda gli Spring Profiles, sono stati creati 3 file application-xxx.yml contenenti ciascuno configurazioni per l'utilizzo del corrispettivo database. Di seguito, le differenti configurazioni:

- application-postgresql.yml per la configurazione di postgresql
- application-mysql.yml per la configurazione di mysql
- application-mariadb.yml per la configurazione di mariadb

[6] Avendo disabilitato questa feature in alcuni test, tramite l'annotazione @Before si puliscono le due repository.

Gli Spring Profiles permettono di utilizzare configurazioni diverse dell'applicazione selezionando il profilo desiderato, in questo caso, i vari profili elencati sono utilizzati per cambiare in modo veloce la configurazione del database utilizzato sia per la fase di testing che per l'avvio della spring boot app. Tutti i database sono stati testati su: Unit Test, Integration Test ed E2E Test, sia in locale che nella build in CI, infatti, come si nota nel file maven.yml è presente "db.profile" dove è necessario specificare il profilo maven da utilizzare per la build in CI. Per ottenere un binding automatico tra Spring Profile e Maven Profile, sono stati definiti 3 differenti profili Maven (postgresql, mysql, mariadb). Ciascun profilo Maven definisce due aspetti fondamentali: primo, specifica quale Spring Profile attivare tramite la property `activeProfile`[7]; secondo, configura e avvia automaticamente il container Docker contenente il database corrispondente tramite il `docker-maven-plugin`. Durante lo sviluppo si sono riscontrate problematiche legate alle tempistiche di avvio dei container, in particolare per MySQL e MariaDB, che richiedevano più tempo per essere completamente operativi. Per garantire una connessione stabile prima dell'esecuzione dei test, sono stati configurati tempi di attesa opportuni nella configurazione del plugin Docker.

[7]Nella build principale, è stato necessario impostare il filtering a `true` per permettere la sostituzione del valore `@activeProfile@` nel file `application.properties`, consentendo così a Spring di attivare il profilo corretto.

5 Istruzioni Riproduzione Build

Per l'esecuzione della build, posizionarsi nella directory principale dove è presente il file `pom.xml`, avviare docker (i container partono automaticamente durante la build). In caso l'esecuzione della build con i mutation-testing fosse troppo lenta, commentare targetClasses e targetTests relativi ai controllers, dato che la logica è contenuta nel service, in entrambi i casi nessun mutante sopravvive. Dipendentemente dal database voluto per i tests, come spiegato in precedenza, specificare un profilo maven tra postgresql, mysql o mariadb (nei comandi sottostanti si riporta l'esempio con postgresql).

Per l'esecuzione di Unit Test, Integration Test, Code Coverage e Mutation Testing:

```
mvn verify -Pjacoco,mutation-testing,postgresql
```

Per l'esecuzione solitaria degli E2E Tests:

```
mvn verify -Pe2e-tests,postgresql
```

Per avviare la spring boot app, nel file application.properties, è necessario commentare la riga contenente `spring.profiles.active=@activeProfile@` e decommentare quella contenente un profilo specifico[8]. Oltre ad attivare un profilo specifico (postgresql, mysql o mariadb), è necessario anche specificare il file docker compose corretto, pertanto sono stati predisposti 3 docker compose file, uno per ogni database, rinominati in modo intuitivo: compose-postgresql.yml, compose-mysql.yml, compose-mariadb.yml. Le configurazioni sono impostate su `jpa.hibernate.ddl-auto: update`, pertanto, se si vuole un database pulito, è necessario eliminare il container relativo al database già utilizzato in precedenza.

[8] Nel file è già presente una riga che attiva postgresql, è sufficiente decommentarla