Advanced Programming Techniques Relazione: eShop

Giovanni Burbi
giovanni.burbi@stud.unifi.it

Abstract

In questa relazione viene presento il progetto eShop app, sviluppato per l'esame del corso 'Advanced Programming Techniques'. L'applicazione simula un semplice negozio online. L'obiettivo del progetto non è sviluppare un'applicazione complessa, ma sfruttare metodologie di programmazione che, con il supporto di frameworks e librerie, aiutano lo sviluppo del codice, il testing e forniscono certe garanzie della qualità del codice prodotto. Gli aspetti più importanti del progetto sono le tecniche di test utilizzate, in particolare la metodologia BDD per le feature ad alto livello e TDD per le componenti di più basso livello, l'uso di funzionalità avanzate di un noto database non relazionale, MongoDB, quali le transazioni e le relazioni, e l'impiego di strumenti di supporto quali SonarCloud, Coveralls e Pit. Lo scopo è quello di avere un processo automatico sfruttando la pratica di Containuous Integration con Github Actions tramite la Build Automation di Maven per incrementare la produttività e la qualità del lavoro svolto.

Parole chiave: CI, Docker, Eclipse, Github, Java, Maven, Mocking, MongoDB, Testing, BDD, TDD, Cucumber, Transactions, Replica set

1. Introduzione

Il progetto di esame prevede l'implementazione di un'applicazione che simula un semplice negozio online. Il linguaggio di programmazione usato è **Java 8** e l'architettura principale dell'applicazione è mostrata in Figura 1.

L'architettura consiste di un **Model View Controller** con uno strato **Repository**, un'interfaccia per accedere ad un generico database, e uno strato di gestione (**Management**) che si occupa di chiamare i metodi forniti dallo strato sottostante all'interno di transazioni, garantendo così alle operazioni le proprietà di atomicità, coerenza, isolamento e durabilità (**ACID**). L'interfaccia grafica prevede una lista di prodotti acquistabili che l'utente può aggiungere ad un carrello, da cui si possono anche rimuovere prodotti. L'utente,

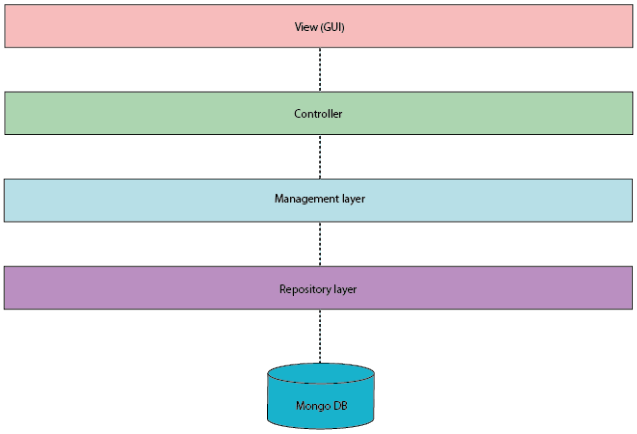


Figure 1. Architettura del progetto

attraverso un campo di testo, potrà cercare specifici prodotti all'interno del catalogo e ripristinare lo stato iniziale della lista dei prodotti dopo una ricerca attraverso un pulsante di *Clear*. Infine l'utente potrà effettuare il checkout del carrello e ricevere un feedback sull'esito dell'operazione. Nell'eventualità che all'interno del carrello ci sia almeno un prodotto con una quantità richiesta superiore alla disponibilità, il checkout fallirà. In tal caso verrà mostrato un messaggio che descrive l'errore specificando il prodotto che ha causato il fallimento dell'operazione. Altrimenti, quando il checkout va a buon fine, il messaggio informa l'utente del costo complessivo pagato e un riepilogo dei prodotti acquistati.

Per questo progetto verrà seguita la metodologia **Behavior-Driven Development** (BDD) usando il framework **Cucumber** per specificare le features ad alto livello dell'applicazione, in particolare quelle che riguardano l'interfaccia grafica, insieme a **JUnit** per il testing delle componenti di basso livello secondo la metodologia **Test-Driven Development** (TDD). L'ambiente di sviluppo integrato (**IDE**) scelto per la realizzazione del progetto è **Eclipse**.

Per il backend dell'applicazione verrà usato **MongoDB**, un database non relazionale basato sui documenti, istanziato su un container **Docker**. Di MongoDB sfrutteremo anche il

suo meccanismo di transizioni e le pipeline di aggregazione. Per l'interfaccia grafica verrà usato **Java Swing**. **Github** è stato scelto come **Version Control System** per la facilità di integrazione con gli strumenti di supporto allo sviluppo del codice usati per questo progetto che vedremo in seguito. Lo sviluppo dell'applicazione è supportato dal meccanismo di **Continuous Integration** di **Github Actions** sfruttando la **Build Automation** con **Maven**. Ora vedremo l'implementazione dell'applicazione seguendo i vari scenari che descrivono le funzionalità ad alto livello del negozio online.

2. Primo Scenario

Come primo scenario, definiamo lo stato iniziale del nostro negozio online che prevede all'avvio dell'applicazione la visualizzazione di una finestra con una lista popolata dai prodotti presenti all'interno del database.

```
Feature: eShop View
  Specifications of the behavior of the
      eShop View

   Scenario: The initial state of the view
    Given The database contains products
        with the following values
      | "1" | "Laptop" | 1300.0 |
      | "2" | "Iphone" | 1000.0 |
      | "3" | "Laptop MSI" | 1250.0 |
     When The eShop View is shown
     Then The list contains elements with
         the following values
      | "1" | "Laptop" | 1300.0 |
      | "2" | "Iphone" | 1000.0 |
      | "3" | "Laptop MSI" | 1250.0 |
```

Lo scenario è definito attraverso la definizione di passi (**steps definition**) che corrispondono all'implementazione delle fasi **Given, When, Then** dello scenario e che guidano lo sviluppo delle componenti di basso livello. Il flusso di lavoro quindi inizia dagli **steps** necessari a completare lo scenario per poi proseguire con l'implementazione delle componenti di basso livello necessarie agli steps su cui stiamo lavorando usando la metodologia **TDD**. Poi verranno integrate insieme le varie componenti, creando così degli **integration tests** che verificano le corrette interazioni tra le stesse. A quel punto lo scenario può essere eseguito con successo.

Il **Domain Model** della nostra applicazione è la classe *Prodotto*.

```
public class Product {
  private String id;
  private String name;
  private double price;
  ...
}
```

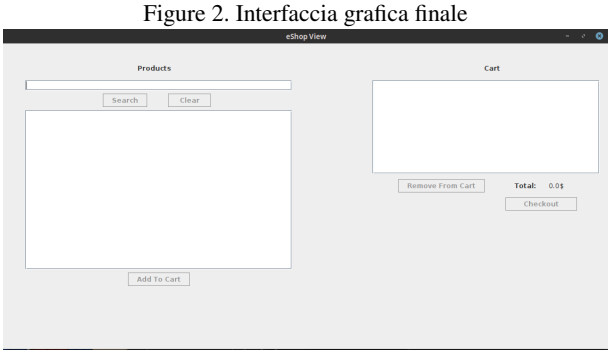
Questa classe ha tre campi per indicare l'*id*, il *nome*, il *prezzo* di un prodotto. Sono stati generati i metodi *getter* e *setter* per ogni campo e il metodo *toString* che traduce in stringa il prodotto usando i valori dei suoi campi.

Il primo step dello scenario, il **Given step**, prevede che ci siano prodotti (catalogo) all'interno del database. Abbiamo creato quindi una interfaccia per accedere al catalogo di prodotti in un generico database, *ProductRepository*. Questa interfaccia verrà implementata per un database **MongoDB** usando il suo **driver Java**. Il database deve mantenere in memoria la quantità disponibile (*storage*) relativa ad ogni istanza di prodotto del catalogo. Per questo aggiungiamo al modello la classe *CatalogItem* che ha come campi: *prodotto*, un'istanza della classe *Prodotto*, e *storage*, la relativa quantità del prodotto disponibile. **MongoDB** è basato sui documenti, nel database lavoriamo con documenti **Bson** in cui definiamo i campi *_id*, *name*, *price* e *storage* assegnando i valori estratti da istanze di *CatalogItem*. Dovremo quindi occuparci di tradurre da documento ad oggetto java e viceversa ogni volta che effettuiamo operazioni di CRUD nel database.

```
private List<Document>
    fromCatalogItemsToDocuments(List
<CatalogItem> items) {
  List<Document> documents = new
      ArrayList<>();
  for (CatalogItem item : items) {
    documents.add(new Document()
      .append("_id",
          item.getProduct().getId())
       .append("name",
           item.getProduct().getName())
       .append("price",
           item.getProduct().getPrice())
       .append("storage",
           item.getStorage())
    );
  }
  return documents;
  }

  private CatalogItem
      fromDocumentToCatalogItem(Document
      doc) {
    return new CatalogItem(new
        Product(""+doc.get("_id"),
        ""+doc.get("name"),
        doc.getDouble("price")),
        doc.getInteger("storage"));
  }
```

Figure 2. Interfaccia grafica finale



MongoDB crea in automatico il campo *_id* alla creazione di un documento Bson ed è un indice univoco nella collezione che identifica il documento. Useremo questo campo per assegnare il valore dell'*id* del prodotto, così che non possiamo avere due prodotti con il medesimo identificativo nella collezione. Come già detto, l'implementazione del codice è preceduta da **Unit test** seguendo la metodologia **TDD**. In questo caso, però, il testing dell'implementazione della repository verrà fatta usando direttamente la vera implementazione del database, quindi tale testing è considerato come **integration test**. Il prossimo passo, il **When step**, del primo scenario prevede la visualizzazione dell'interfaccia grafica. Abbiamo creato un'interfaccia generica per la view a cui aggiungeremo metodi via via con l'aggiunta di nuovi scenari. Abbiamo implementato questa interfaccia usando un *JFrame* di **Java Swing**. L'interfaccia grafica finale avrà l'aspetto della Figura 2. Per quanto riguarda il primo scenario, abbiamo solo creato la finestra con la lista dei prodotti. Per definire il **When step** del primo scenario, prima abbiamo estratto il *main* generato dal *JFrame* di **Java Swing** in una classe apposita, *EShopSwingApp*. Questa classe si occupa di istanziare nel corretto ordine le varie componenti necessarie all'applicazione e avviare quest'ultima in modo che all'utente sia mostrata l'interfaccia grafica. Usiamo **picocli** per specificare argomenti che possono essere passati da riga di comando all'applicazione. Il **When step** userà *EShopSwingApp* per avviare l'applicazione passando gli argomenti per i **BDD** test. Per definire l'ultimo step, **Then step**, di questo scenario, che verifica che la lista di prodotti mostrata nella finestra sia effettivamente popolata con gli elementi all'interno del database, implementiamo l'ultima componente principale dell'architettura **MVC**, il controller. Tale componente accede, per ora, direttamente alla repository e riceve gli input dell'utente attraverso la view. In uno dei successivi scenari verrà creato uno strato di gestione che impedirà l'accesso diretto del controller alla repository, che quindi dovrà delegare a questo nuovo strato tutte le richieste. Ognuna delle implementazioni delle componenti che abbiamo introdotto è accompagnata da **Unit test** usando **JUnit 4** e **Mockito** per testare in isolamento le componenti secondo metodologia **TDD**. Completati con successo gli **Unit test**, abbiamo testato l'integrazione dei componenti attraverso **IT tests**. Visto che abbiamo già una prima implementazione della repository abbiamo usato quest'ultima insieme ad una vera istanza del database all'interno dei nostri **integration test**. Per avviare istanze di **Docker containers** con l'immagine del database scelto direttamente da questi test di integrazione è stato usato il plugin **Test Containers**. Il plugin si occupa anche di fermare i containers e rimuovere eventuali volumi in memoria al termine dei test. Completati gli IT tests, per assicurarci che il primo scenario sia eseguito con successo, si avvia prima un container **Docker** con l'immagine di *MongoDB:4.4.3* localmente e si esegue, usando il runner di **Cucumber**, la classe *EShopAppBDD* in cui abbiamo specificato dove trovare il **feature file**.

3. Meccanismi di supporto allo sviluppo

Abbiamo usato i plugin, **Jacoco**, per generare report sul *code coverage*, e **PIT** per effettuare il *mutation testing*. Sono stati creati due profili nel **pom** del progetto per i rispettivi plugin e sono state escluse le classi che non devono essere analizzate. Per valutare la qualità del codice abbiamo usato il plugin per **SonarCloud**. Per il code coverage di Jacoco e SonarCloud sono state escluse dall'analisi la classe che lancia l'applicazione, *EShopSwingApp*, e le classi che appartengono al **Domain Model**. Dal mutation testing sono state escluse anche la classe che implementa la view, *EShopSwingView*, la classe che implementa, con mongo, il manager delle transazioni, che vedremo in seguito, e quasi tutti i test che non fanno parte della suite degli **Unit test**. Gli **integration test** *CartMongoRepositoryIT*, che creeremo in un successivo scenario, e *ProductMongoRepositoryIT* vengono, invece, inclusi nel **mutation testing** in quanto ricoprono anche il ruolo di **Unit test**. Questo deriva dalla decisione di usare vere istanze del database. Abbiamo configurato poi il plugin **Failsafe** per eseguire i BDD test e gli IT test in maniera automatica attraverso la build di **Maven**. Abbiamo, inoltre, usato **docker-maven-plugin** per avviare e rimuovere, durante la Maven build, il container con l'immagine di MongoDB. Come indicato nell'introduzione, è stato sfruttato il processo di **Continuous Integration** di **Github Actions** per automatizzare il testing dell'applicazione attraverso la build automation con **Maven** ad ogni *push* e *pull request* sulla repository di **GitHub**. Sono stati creati i workflows per **Linux** e **Mac OS**. Viene eseguita la build sul **Continuous Integration Server** di **Github Actions** sia con **JDK 8** che **JDK 11**, le versioni LTS di Java. Nella build definita nei workflows sono stati attivati i profili e i goals per effettuare l'analisi di *code coverage* con **Coveralls** e *qualità del codice* con **SonarCloud** in modo che possano pubblicare i loro report direttamente su **GitHub**. Nel caso la build fallisca viene salvato il report generato da **Surefire** per gli Unit test. Invece, nel caso di

pull-request, vengono aggiunti, rispetto al workflow per i *push*, due ulteriori passi dopo l'archiviazione del report di **JUnit**. Questi passi riguardano il *mutation testing* e vengono eseguiti solo quando abbiamo una suite di test verde, quindi solo quando i passi precedenti sono stati eseguiti con successo. In questo caso viene attivato il goal di **PIT** per effettuare l'analisi dei mutanti. Anche in questo caso il report viene salvato in caso di fallimento del passo di *mutation testing* della build della *pull-request*.

```
# Run PIT only if previous steps are
    successfully executed
  - name: Run PIT
    id: pit
    run: mvn org.pitest:pitest-maven:
      mutationCoverage
    working-directory: ${{ env.workdir }}
    # Archive PIT report only if Run PIT
        fails
  - name: Archive PIT Report
    uses: actions/upload-artifact@v2
    if: ${{ always() && steps.pit.outcome
        == 'failure' }}
    with:
      name: pit-report-jdk-${{ matrix.java
          }}
      path: '**/target/pit-reports'
```

4. Secondo Scenario

Il secondo scenario riguarda la funzionalità di ricerca all'interno della lista dei prodotti. Vogliamo che la nostra applicazione individui tutti i prodotti che hanno come sottostringa nel campo '*name*' la parola digitata nel box di ricerca al momento che il pulsante search viene premuto.

```
Scenario: Search a product
  Given The database contains products
      with the following values
    | id | name | price |
    | "1" | "Laptop" | 1300.0 |
    | "2" | "Iphone" | 1000.0 |
    | "3" | "Laptop MSI" | 1250.0 |
  When The eShop View is shown
  And The user enters in the search text
      field the name "la"
  And The user clicks the "Search" button
  Then The list shows products with "la"
      in the name
```

Viene seguito un flusso di lavoro simile a quello visto nel primo scenario per lo sviluppo delle classi del nostro progetto. Per la repository viene sfruttata l'*API* di Java per MongoDB per fare una query all'interno del database nella collezione dei prodotti.

5. Background Scenari

A questo punto è stato fatto un refactoring del **feature file** del **BDD** test aggiungendo uno scenario di **Background** che permette di configurare dei passi che vengono eseguiti prima dell'esecuzione di ogni altro scenario. Il background consiste nella popolazione del database con il catalogo e la visualizzazione dell'interfaccia grafica.

```
Background:
  Given The database contains products
      with the following values
       | id | name | price | storage |
       | 1 | Laptop | 1300.0 | 2 |
       | 2 | Iphone | 1000.0 | 2 |
       | 3 | Laptop MSI | 1250.0 | 1 |
  When The eShop View is shown
```

Il primo scenario quindi avrà solo bisogno del *Then step*, mentre il secondo scenario assume una forma più concisa.

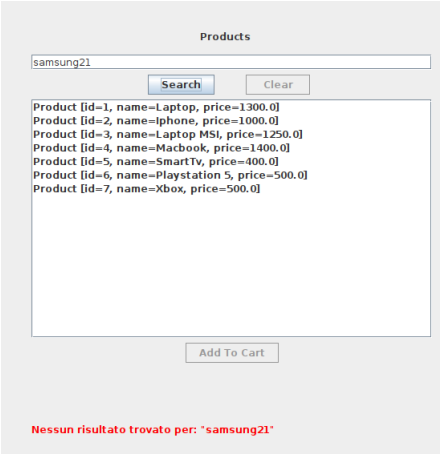
```
Scenario: Search a product
  Given The user enters in the search text
      field the name "la"
  When The user clicks the "Search" button
  Then The list shows products with "la"
      in the name
```

6. Terzo Scenario

```
Scenario: Search a not existing product
  Given The user enters in the search text
      field the name "samsung s21"
  When The user clicks the "Search" button
  Then An error is shown containing the
      name searched "samsung s21"
```

Nel caso in cui la ricerca nella lista dei prodotti non fornisca nessun risultato, viene visualizzato un messaggio di errore che informa l'utente che, appunto, nessun risultato è stato trovato nel catalogo. Il messaggio di errore viene resettato nel momento in cui l'utente preme un tasto mentre il focus della finestra si trova sul box di ricerca.

Figure 3. Feedback ricerca senza risultato



7. Quarto Scenario

Il quarto scenario riguarda il ripristino dello stato iniziale della lista dei prodotti dopo una ricerca avvenuta con successo.

```
Scenario: Clear the search
  Given The user search the product
      "laptop"
  When The user clicks the "Clear" button
  Then The list contains elements with the
      following values
    | id | name | price |
    | 1 | Laptop | 1300.0 |
    | 2 | Iphone | 1000.0 |
    | 3 | Laptop MSI | 1250.0 |
  And The search text box is empty
```

Il bottone di clear ha il compito di resettare la lista dei prodotti andando a recuperare l'intero catalogo dal database, resettare un eventuale errore dopo una ricerca e svuotare il box di ricerca.

8. Quinto Scenario

```
Scenario: Add Product to Cart
  Given The cart contains a product
  And The user select another product from
      the product list
  When The user clicks the "Add To Cart"
      button 2 times
  Then The cart list contains elements
      with the following values
       | id | name | price | quantity |
       | 2 | Iphone | 1000.0 | 1 |
       | 1 | Laptop | 1300.0 | 2 |
  And The view shows the updated total of
      "3000.0$"
```

Per questo scenario implementeremo il carrello e la funzionalità di aggiunta di prodotti in quest'ultimo. L'aggiunta di prodotti al carrello è senza prenotazione, questo vuol dire che il controllo sulla disponibilità in storage viene fatto al momento del checkout. Per prima cosa abbiamo definito una nuova classe nel **Modello** del progetto , *CartItem*, che ha come campi: *prodotto*, istanza della classe *Prodotto*, e *quantity*, la quantità richiesta di tale prodotto. È stato definito anche il metodo *toString()* che chiama a sua volta il metodo *toString()* del *prodotto* che ritorna una stringa con i valori dell'oggetto a cui viene aggiunto il valore del campo *quantity* di *CartItem* . Successivamente, per implementare la funzionalità introdotta da questo scenario, è stata creata una nuova interfaccia nello strato **Repository**, *CartRepository*, che si occupa di definire le operazioni che riguardano il carrello. L'implementazione di questa interfaccia consiste nel creare una nuova collezione nel database, che assume il ruolo di carrello, nella quale andremo ad inserite documenti che contengo un **riferimento** all'*id* univoco del prodotto selezionato per essere aggiunto al carrello e *quantity*, la quantità richiesta di tale prodotto. Se viene aggiunto un prodotto già presente nella collezione del carrello allora viene solo aggiornato il relativo campo *quantity*. Così facendo andiamo a definire una **relazione** fra i documenti nella collezione del carrello con quelli nella collezione dei prodotti attraverso il **riferimento** all'*id* del prodotto. È stato anche implementato un metodo per calcolare il costo totale dei prodotti all'interno del carrello. Nell'interfaccia grafica è stata aggiunta una nuova lista che rappresenta il carrello e un bottone '*Add To Cart*' per aggiungerci il prodotto selezionato nella lista dei prodotti. Per mostrare gli elementi nel carrello all'interno della lista *Cart* nella finestra dell'applicazione è stato creato il metodo *allCart()* che ritorna una lista di *CartItem* degli elementi all'interno della collezione del carrello.

```
public List<CartItem> allCart() {
  // Aggregate documents of this
      collection with related products in
      productCollection
  List<Document> cartJoined =
      aggregateCollections();
  return cartJoined.stream()
     .map(d -> new CartItem(
      fromDocumentToProduct(d.get("product",
          Document.class)),
          d.getInteger("quantity")))
      .collect(Collectors.toList());
  }

private Product
    fromDocumentToProduct(Document d) {
     return new Product(""+d.get("_id"),
         ""+d.get("name"),
         d.getDouble("price"));
```

```
  }
private List<Document>
    aggregateCollections() {
  Bson lookup =
      lookup(productCollectionName,
      "product", "_id", "product");
  Bson project =
      project(fields(include("product",
      "quantity"), excludeId()));
  Bson unwind = unwind("$product");
  return cartCollection
     .aggregate(asList(lookup, project,
         unwind)).into(new ArrayList<>());
  }
```

Il metodo *aggregateCollections()* aggrega i documenti della collezione del carrello, *cartCollection*, con i documenti che rappresentano i prodotti all'interno della *productCollection*. Per fare questo viene usata una **pipeline di aggregazione** che consiste nel definire delle **fasi** per processare i documenti. Ogni fase effettua operazioni sui documenti in ingresso. I documenti che escono in output da una fase diventano input per la fase successiva. La prima fase che è stata definita è il *lookup* in cui viene effettuato un join della *cartCollection* con la *productCollection* usando i campi "*product*" e "*_id*" dei documenti nelle rispettive collezioni. Il risultato è un array field, con il nome dell'ultimo parametro passato al metodo *lookup()*, all'interno dei documenti della *CartColletion*, in cui vengono inseriti i campi dei documenti del *productColletion* che soddisfano l'uguaglianza fra i campi usati per il join. L'output della prima fase diventa l'input della seconda fase, *project*, che per ogni documento rimuove il campo *_id*, generato automaticamente da MongoDB, del documento originario nella *cartCollection*, non quello del prodotto, e restituisce il documento risultante in output per la terza fase. Nell'ultima fase, *unwind*, viene decomposto l'array field dei documenti in input restituendo in output un documento per ogni elemento dell'array field rimpiazzandolo con il valore dell'elemento. Nel nostro caso, dopo la seconda fase, ogni documento in *cartCollection* ha un array field con un solo documento che rappresenta un prodotto. La fase di unwind rimpiazza l'array field con i valori dei campi del prodotto rappresentato dal documento nell'array.

9. Sesto Scenario

```
Scenario: Remove Product from Cart
  Given The cart contains 2 item of a
      product
  When The user select that product in the
      cart
  And The user clicks the "Remove From
      Cart" button
  Then The cart list is empty
  And The view shows the updated total of
      "0.0$"
```

Il sesto scenario consiste nella funzionalità di rimozione di elementi dal carrello. La rimozione viene applicata su tutti gli elementi di uno stesso prodotto; quindi se un prodotto ha un certa quantità all'interno del carrello, tutti vengono rimossi quando l'utente preme il bottone '*Remove From Cart*' nella finestra.

10. Transazioni in MongoDB

Il prossimo scenario consiste nel checkout del carrello. Prima di vedere questo scenario, però, è necessaria fare una digressione sulle **transazioni** con MongoDB. Le transazioni in MongoDB sono associate ad una *sessione* e ogni operazione all'interno della transazione deve essere associata con la *sessione*. Se proviamo ad effettuare una transizione usando un database standalone come abbiamo fatto fino ad ora otteniamo un'eccezione:
com.mongodb.MongoQueryException: Query failed with error code 20 and error message 'Transaction numbers are only allowed on a replica set member or mongos' on server localhost:27017.
Abbiamo quindi bisogno che il nostro database standalone sia configurato come **Replica set**. Per fare ciò, **localmente**, bisogna prima avviare il database specificando l'appartenenza ad un replica set attraverso il comando:
sudo docker run –name mongoRs -p 27017:27017 –rm mongo:4.4.3 –replSet rs0
Poi, attraverso un nuovo terminale, andiamo a configurare il replica set con la configurazione standard attraverso il comando:
sudo docker exec -it mongoRs mongo –eval 'rs.initiate()'
Per quanto riguarda la **build automation** con **Maven**, il *docker-maven-plugin* che si occupa di avviare e rimuovere il container *Docker* con l'immagine standalone di mongoDB non è più sufficiente per le nostre necessità. Dobbiamo quindi definire un container **custom** con **MongoDB Replica Set** attraverso un file **Docker-compose**.

```
version: "2"
services:
 mongo:
   hostname: mongodb
   image: mongo:4.4.3
   restart: always
   ports:
     - "27017:27017"
   volumes:
     - ./scripts:
         /docker-entrypoint-initdb.d/
   command: ["--replSet", "rs0",
       "--bind_ip_all"]
```

L'immagine di MongoDB di *Docker* fornisce il percorso */docker-entrypoint-initdb.d* in cui vengono eseguiti file di setup custom con estensione .js o .sh una volta sola, all'inizializzazione del database. Abbiamo creato un file, **Init.js**, che contiene il comando per la configurazione: *rs.initiate()*. È stata modificata, quindi, la configurazione del **docker-maven plugin** per usare il **Docker-compose** file che abbiamo creato invece che l'immagine standard di MongoDB.

```
<image>
   <external>
     <type>compose</type>
     <basedir>${project.basedir}</basedir>
   </external>
   ...
</image>
```

Per gli **integration test** abbiamo usato **Test Containers** per gestire i containers **Docker**. Quindi per istanziare un database **MongoDB Replica Set** come abbiamo già visto, è necessario modificare la configurazione dei containers attraverso l'*API* offerta da **Test Containers**:

```
@ClassRule
   public static GenericContainer mongo =
       new GenericContainer("mongo:4.4.3")
          .withExposedPorts(27017)
          .withCommand("--replSet rs0");

@BeforeClass
  public static void mongoConfiguration() {
    // configure replica set in MongoDB
       with TestContainers
    try {
      mongo.execInContainer("/bin/bash",
          "-c", "mongo --eval
          'printjson(rs.initiate())' " +
          "--quiet");
      mongo.execInContainer("/bin/bash",
          "-c",
           "until mongo --eval
                \"printjson(rs.isMaster())\"
                | grep ismaster | grep true
                > /dev/null 2>&1;"
                 + "do sleep 1;done");
    } catch (Exception e) {
      throw new
          IllegalStateException("Failed to
          initiate rs.", e);
    }
  }
```

È stato creato un metodo *@BeforeClass* in cui andiamo ad eseguire nel container creato il comando per configurare il database. La seconda istruzione che eseguiamo nel container fa in modo che si attenda che la configurazione sia effettivamente completata prima di continuare l'esecuzione dei test. Il modo di configurare il **Replica set** attraverso il **Docker-compose** file mostrato funziona solo con **Linux containers**. L'ambiente *Windows* di **GitHub Actions** permette il solo uso di **Windows containers**. In questo caso l'immagine di MongoDB è diversa da quella usata per Linux containers, in particolare i file nel percorso */docker-entrypoint-initdb.d* non vengono eseguiti all'inizializzazione del database, di conseguenza il modo di configurare il **Replica set** visto in precedenza non funziona. [1] Bisogna quindi usare un metodo alternativo per configurare il database senza sfruttare la funzionalità offerta dal percorso */docker-entrypoint-initdb.d* dell'immagine di MongoDB per **Linux containers**. È stata creata una rete di containers di cui uno contiene l'immagine di mongoDB che definisce il **Replica set** di cui fa parte mentre l'altro si occupa di configurare il **Replica set** tramite il file *setup-replica.sh*.

```
version: "2"
services:
 mongo:
   hostname: mongodb
   container_name: mongodb
   image: mongo:4.4.3
   networks:
     - my-network
   restart: always
   ports:
     - "27017:27017"
   command: ["--replSet", "rs0",
       "--bind_ip_all"]

 replica-setup:
   hostname: mongosetup
   container_name: mongosetup
   image: mongo:4.4.3
   networks:
     - my-network
   restart: on-failure
   ports:
```

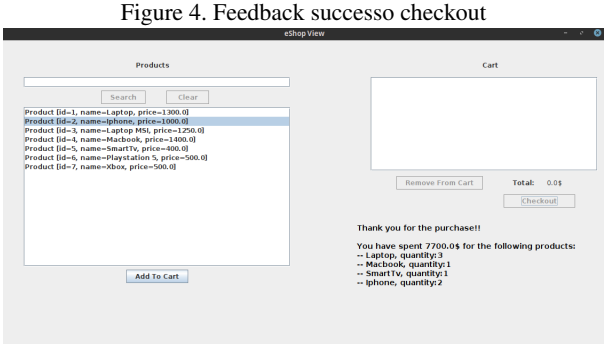
```
    - "27018:27018"
  volumes:
    - ./scripts/setup-replica.sh:/
        scripts/setup-replica.sh
  entrypoint: ["bash"
      ,"/scripts/setup-replica.sh"]
  depends_on:
    - mongo
networks:
 my-network:
  driver: bridge
```

Il file *setup-replica.sh* attende l'avvio di MongoDB del container *mongo* e apre un **MongoDB shell** in cui esegue il comando di configurazione per il **Replica set**.

```
echo "Starting replica set initialize"
until curl
   http://27017:27017/serverStatus\?text\=1
       2>&1 |
   grep uptime | head -1;
do
 sleep 1
done
echo "Connection finished"
echo "Creating replica set"
mongo --host mongodb:27017 <<EOF
rs.initiate();
EOF
echo "replica set created"
```

È necessario, inoltre, permettere la creazione automatica di networks custom nella configurazione di **docker-maven plugin** nel pom. Questo metodo di configurazione del **Replica set** risolve il problema visto in precedenza ma non è sufficiente a risolvere il problema di configurare il **Replica set** in **Windows containers**. Appare infatti un problema di conflitto fra processi in **Windows**, impedendo così la corretta configurazione del database all'interno del container.
I/O Error [Unable to start container id [118d0203f4e3] : "message":"failed to create endpoint mongo-1 on network nat: failed during hnsCallRawResponse: hnsCall failed in Win32: The process cannot access the file because it is being used by another process. (0x20)" (Internal Server Error: 500)]
Su **Linux containers** invece tutto continua a funzionare come la precedente versione. Di conseguenza si consiglia, se si utilizza il sistema operativo **Windows**, di utilizzare **Linux containers** invece di**Windows containers** prima di avviare l'applicazione. [2] Dato che su **Github Actions** non è possibile impostare **Linux containers** nell'ambiente di **Windows**, non è stato aggiunto il workflow relativo a quest'ultimo sistema operativo.

Figure 4. Feedback successo checkout



11. Settimo Scenario

```
Scenario: Checkout successfully
  Given The cart contains some products
  When The user clicks the "Checkout"
      button
  Then The cart list is empty
  And The view shows the updated total of
      "0.0$"
  And The view shows a message about the
      successful checkout
  And The database storage of the
      purchased products is updated
```

Il settimo scenario definisce la funzionalità di checkout del carrello. Una volta completata l'operazione la finestra svuota il carrello, ne resetta il costo e mostra un messaggio per informare l'utente del successo del checkout, Figura 4. Inoltre il database aggiorna le quantità dei prodotti venduti. Per implementare la funzionalità di checkout è necessario usare le **transazione** di MongoDB. Abbiamo già visto come configurare il database come **Replica set**, sia nel caso che vogliamo avviare l'applicazione localmente che durante la build automation con **Maven**. Per gestire le transazioni è stato creato lo strato di **Management** in cui è stata definita l'interfaccia del manager delle transazioni che si occupa di gestire le operazione della strato di **Repository** attraverso **transazioni**.

```
public interface TransactionManager {
  <T> T doInTransaction(TransactionCode<T>
      code);
}
```

Il metodo astratto *doInTransaction* dell'interfaccia *TransactionManager* riceve come parametro una **espressione lambda** definita da un'interfaccia funzionale che possiede un metodo astratto che accetta due parametri di ingresso e restituisce un tipo generico:

```
@FunctionalInterface
public interface TransactionCode<T> {
   T apply(ProductRepository
       productRepository, CartRepository
       cartRepository);
}
```

Abbiamo poi implementato *doInTransaction* dell'interfaccia del manager nella classe *TransactionalShopManager*, usando l'*API* di MongoDB per le transazioni:

```
   @Override
  public <T> T
      doInTransaction(TransactionCode<T>
      code) {
    ClientSession session =
        client.startSession();
    // create a transaction
    session.startTransaction(
    TransactionOptions.builder()
    .writeConcern(WriteConcern.MAJORITY)
    .build());
    // create a repository instance in the
        transaction
    ProductMongoRepository
        productRepository = new
        ProductMongoRepository(client,
        databaseName,
         productCollectionName, session);
    CartMongoRepository cartRepository =
        new CartMongoRepository(client,
        databaseName, cartCollectionName,
        productCollectionName, session);
    // call a lambda passing the
        repository instance
    T result =
        code.apply(productRepository,
        cartRepository);
    session.commitTransaction();
    Logger.getLogger(getClass()
       .getName()).log(Level.INFO,
           SUCCESSFUL_TRANSACTION);
    // close the transaction
    session.close();
    Logger.getLogger(getClass()
       .getName()).log(Level.INFO,
           TRANSACTION_ENDED);
    return result;
  }
```

Successivamente è stato implementato *ShopManager* che riceve nel costruttore, come parametro, una istanza di una classe che estende l'interfaccia generica del manager transazionale e delegata ad essa l'esecuzione delle operazioni di database in modo siano eseguite come transazioni.

Questa struttura permette alla classe *ShopManager* di essere indipendente dall'implementazione del manager delle transazioni, il quale usa la specifica *API* per le transazioni definita da una implementazione del database.

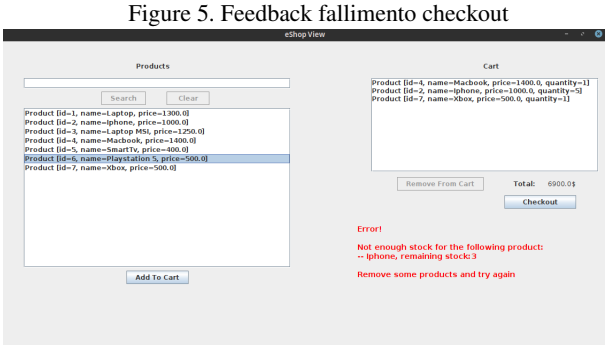
```
public ShopManager(TransactionManager
    transactionManager) {
    this.transactionManager =
        transactionManager;
  }

  public void
      setShopController(EShopController
      shopController) {
    this.shopController = shopController;
  }

  public void checkout() {
    transactionManager.doInTransaction(
       (productRepository, cartRepository)
           -> {
      List<CartItem> items =
          cartRepository.allCart();
      for (CartItem item : items) {
        productRepository
            .removeFromStorage(item);
        cartRepository
            .removeFromCart(item.getProduct());
      }
      shopController.checkoutSuccess();
      return null;
    });
  }
```

Adesso che abbiamo implementato un manager per le transazioni, il controller non dovrebbe più accedere direttamente alle repository, ma dovrebbe farlo solo attraverso questo manager. Abbiamo quindi implementato in *ShopManager* le chiamate dei metodi della repository in una **espressione lambda** passandola al metodo *doInTransaction* di una classe che estende *TransactionManager*. Abbiamo inoltre modificato l'implementazione del controller in modo che deleghi a *ShopManager* le chiamate ai metodi della repository . Otteniamo così l'architettura precedentemente mostrata in Figura 1. *ShopManager*, nel metodo *checkout*, si occupa anche di chiamare il metodo per visualizzare il successo del checkout del controller. Per concludere l'implementazione di questo scenario, è necessario, nelle implementazioni delle repository, eseguire le operazioni di CRUD sul database con l'*API* di MongoDB tramite la sessione creata da *TransactionalShopManager* e passata tramite costruttore alle repository. Infine, per quanto riguarda il testing della classe *ShopManager*, lo stubbing del metodo *TransactionManager.doInTransaction* deve eseguire le lambda expressions passando un mock delle repository così da permettere di

Figure 5. Feedback fallimento checkout



verificarne il comportamento. Per eseguire il codice basato sul valore a runtime dell'argomento passato a *TransactionManager.doInTransaction* abbiamo usato un **Mockito answer** nello stubbing.

12. Ottavo Scenario

```
Scenario: Checkout failure
  Given The cart contains some products of
      which one has quantity greater than
      the stock
  When The user clicks the "Checkout"
      button
  Then The cart list contains elements
      with the following values
     | id | name | price | quantity |
     | 1 | Laptop | 1300.0 | 1 |
     | 2 | Iphone | 1000.0 | 3 |
     | 3 | Laptop MSI | 1250.0 | 1 |
  And The view shows the updated total of
      "5550.0$"
  And The view shows a message about the
      outcome of the checkout
  And The database storage of the products
      has not changed
```

Ora andiamo ad implementare il comportamento dell'applicazione nel caso di fallimento nel checkout per mancanza di prodotti in storage. Questo succede quando nel carrello è presente almeno un prodotto con quantità maggiore della disponibilità definita per quello stesso prodotto all'interno del database. In tal caso, dopo il checkout, il carrello rimane invariato e l'utente deve ricevere un feedback che lo informa di quale prodotto non c'è una disponibilità sufficiente a soddisfare la domanda, Figura 5. Per garantire quindi la consistenza del database, la classe che si occupa della transazione effettua un **rollback** dello stato del database al precedente stato ammissibile prima dell'ultima operazione di checkout La *ProductMongoRepository* lancia una eccezione custom, *RepositoryException*, che estende la classe *Exception* e in cui è possibile memorizzare il prodotto ha causato tale eccezione.

```
@Override
public void removeFromStorage(CartItem
    item) throws RepositoryException {
     int quantityRequested =
         item.getQuantity();
     Bson filterIdProduct =
         Filters.eq(ID_FIELD_NAME,
         item.getProduct().getId());
     CatalogItem productInStorage =
       fromDocumentToCatalogItem(
         productCollection.find(session,
             filterIdProduct).first());
     int quantityInStorage =
         productInStorage.getStorage();
     if (quantityInStorage <
         quantityRequested)
       throw new
           RepositoryException("Insufficient
           stock", productInStorage);
     Bson update = Updates.inc("storage",
         -quantityRequested);
     productCollection.findOneAndUpdate(
       session, filterIdProduct, update);
   }
```

Per gestire l'eccezione lanciata dall'implementazione della *ProductRepository* è stata modificata l'interfaccia funzionale, *TransactionCode*, in modo che possa lanciare a sua volta un'eccezione del tipo *RepositoryException*.

```
 @FunctionalInterface
public interface TransactionCode<T> {
  T apply(ProductRepository
      productRepository, CartRepository
      cartRepository) throws
      RepositoryException;
}
```

Questa eccezione è catturata nel metodo *checkout* di *ShopManager*, il quale si occupa, in questo caso, di chiamare il metodo del controller, *checkoutFailure()*, per mostrare lo stato di errore a causa di mancanza di stock e propagare la *RepositoryException* dall'interno della lambda expression passata al metodo *doInTransaction*.

```
public void checkout() {
     transactionManager
       .doInTransaction((productRepository,
           cartRepository) -> {
        List<CartItem> items =
            cartRepository.allCart();
        try {
          for (CartItem item : items) {
            productRepository
                .removeFromStorage(item);
```

```
            cartRepository
                .removeFromCart(
                item.getProduct());
          }
        } catch (RepositoryException e) {
          shopController
              .checkoutFailure(e.getItem());
          throw new
              RepositoryException("Insufficient
              stock", e.getItem());
        }
        shopController.checkoutSuccess();
        return null;
      });
    }
```

L'eccezione propagata dallo *ShopManager* è catturata dall'implementazione con mongo del manager transazionale, *TransactionalShopManager*, il quale effettua il **rollback**.

```
@Override
  public <T> T
      doInTransaction(TransactionCode<T>
      code) {
    ClientSession session =
        client.startSession();
    try {
      ...
      // call a lambda passing the
          repository instance
      T result =
          code.apply(productRepository,
          cartRepository);
      session.commitTransaction();
      Logger.getLogger(getClass()
          .getName()).log(Level.INFO,
              SUCCESSFUL_TRANSACTION);
      return result;
    } catch (Exception e) {
      session.abortTransaction();
      Logger.getLogger(getClass()
          .getName()).log(Level.INFO,
              ROLLBACK_TRANSACTION);
    } finally {
      // close the transaction
        session.close();
        Logger.getLogger(getClass()
            .getName()).log(Level.INFO,
                TRANSACTION_ENDED);
     }
    return null;
  }
```

Facendo così, manteniamo *ShopManager* disaccoppiato dall'implementazione del database usata.

13. Conclusioni

Abbiamo implementato un'applicazione con un'architettura **Model-View-Controller** in cui è stato aggiunto uno strato di **Management** che si occupa di gestire le query allo strato di **Repository** attraverso **transazioni**. Lo sviluppo ha seguito la metodologia **BDD** per le feature ad alto livello e la metodologia **TDD** con **JUnit 4** e **Mockito** per le componenti di basso livello. **Docker** è stato usato per creare container con l'immagine del database **MongoDB**, sia durante la build automation che localmente. L'interfaccia grafica è stata creata usando **Java Swing**. Il tutto è stato supportato da meccanismi automatici di testing e qualità del codice forniti da vari plugin e frameworks integrati nella build automation di **Maven** eseguita dal servizio di **Continuous Integration** di **GitHub Action**. Le metodologie seguite e gli strumenti utilizzati, che, in gran parte, possono essere usati con molti dei linguaggi di programmazione maggiormente usati, hanno dimostrato di aumentare la **produttività** durante lo sviluppo e la **robustezza** del codice prodotto. Abbiamo anche ottenuto un codice con certe garanzie circa la sua **qualità**, con *code coverage* al 100% e senza *technical debts*.

14. Risorse

Il codice è disponibile pubblicamente su **GitHub**:
`https://github.com/GiovanniBurbi/e-shop`

15. Bibliografia

1. https://github.com/docker-library/mongo/issues/291#issuecomment-410813034
2. https://docs.docker.com/desktop/windows/#switch-between-windows-and-linux-containers
3. L. Bettini, Test-Driven Development, Build Automation, Continuous Integration. LeanPub, 2021.
4. Maven official website: https://maven.apache.org/
5. JaCoCo: https://www.jacoco.org/jacoco/index.html
6. Testcontainers library: https://www.testcontainers.org/
7. Mockito framework: https://site.mockito.org/
8. Java Swing: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
9. picocli CLI: https://picocli.info/
10. PIT framework: https://pitest.org/
11. Docker: https://www.docker.com/
12. Docker Compose: https://docs.docker.com/compose/
13. SonarQube: https://www.sonarqube.org/
14. GitHub Actions Doc: https://docs.github.com/en/actions
15. Cucumber: https://cucumber.io/

16. MongoDB Java Driver: https://mongodb.github.io/mongo-java-driver/
17. MongoDB replication: https://docs.mongodb.com /manual/replication/
18. MongoDB replica set: https://docs.mongodb.com/manual /tutorial/convert-standalone-to-replica-set/
19. MongoDB aggregation: https://docs.mongodb.com/manual/meta/aggregation-quick-reference/aggregation-expressions
20. MongoDB relationships: https://docs.mongodb.com/manual/applications/data-models-relationships/