Automated Software Testing - Una semplice app di blogging

Michael Bartoloni

December 2025

Contents

1 Introduzione

Questo progetto è un'implementazione di un applicativo Java per la gestione di un blog personale, sviluppato per l'esame di Automated Software Testing (Università degli Studi di Firenze). L'implementazione non ha lo scopo di essere utilizzabile nella realtà, è soltanto un esercizio per applicare i vari framework visti nel corso. Si ipotizza di dover gestire una pagina di un blog, in cui è possibile postare articoli, con la possibilità di taggare questi ultimi. Tutti gli

use case dell'app sono destinati al gestore del blog, che aggiunge e tagga gli articoli tramite l'interfaccia grafica.

1.1 Entità e funzionalità

Fondamentalmente le entità in gioco sono due: articoli e tag. Tra i due c'è una relazione molti-a-molti, più precisamente si intende che un articolo può avere molti tag, e lo stesso tag può apparire in più articoli. Un database si occupa di conservare l'insieme di articoli e di tag.
L'applicativo consente all'utente di:

- Salvare nel database un nuovo articolo
- Selezionare un articolo già presente da una lista
- Aggiungere o rimuovere tag all'articolo selezionato, tramite una lista di tag presenti nell'articolo
- Modificare l'articolo selezionato
- Cancellare dal database l'articolo selezionato
- Filtrare gli articoli nella lista in base a uno specifico tag
- Resettare filtri e inserimenti, reimpostando la visualizzazione di default

2 Architettura e implementazione

Per la realizzazione, il codice segue un'architettura MVC con più livelli per gestire modularmente le responsabilità delle varie classi e la gestione del database. In questa sezione viene introdotta l'architettura partendo dal domain model e salendo nei livelli di astrazione fino ad arrivare al livello di presentazione. Tutte le classi sono state implementate seguendo il flusso TDD.

2.1 Data layer

Articoli e tag sono modellati nelle classi Article e Tag rispettivamente. Sono dei POJO con qualche vincolo che ho scelto di imporre sui loro campi, per assicurare che le stringhe che compongono il contenuto leggibile dall'utente non sia vuoto o null.

```
//...
//Nel costruttore di Article.java
if(title == null) throw new IllegalArgumentException("Article title cannot be
↪  null!");
if(title.trim().equals("")) throw new IllegalArgumentException("Article
↪  title cannot be an empty string!");
if(content == null) throw new IllegalArgumentException("Article content
↪  cannot be null!");
```

```java
if(content.trim().equals("")) throw new IllegalArgumentException("Article
↪  content cannot be an empty string!");

//...
//Nel costruttore di Tag.java
if(label == null) throw new IllegalArgumentException("Tag label cannot be
↪  null!");
if(label.trim().equals("")) throw new IllegalArgumentException("Tag label
↪  cannot be blank!");
```

Il campo id degli articoli è volutamente esentato da tali check perché l'utente non ha controllo su di essi. Infatti, è il database stesso a assegnare un id ogni volta che viene inserito un articolo. Inoltre, il database è sfruttato anche per eseguire le operazioni di lettura e scrittura in maniera transazionale.

2.1.1 Repository e transaction manager

Le dependency per la gestione di MongoDB adesso richiedono la creazione di un client tramite una factory, MongoClients, e una stringa per la connessione (host name + porta) come specificato nella documentazione.

Vista la presenza di operazioni con transazioni, la comunicazione con il database richiede di passare ai metodi del driver anche la sessione oltre al client, quindi una repository, implementata dalla classe BlogMongoRepository, viene creata con questi parametri passati nel costruttore.

```java
//...
MongoClient mongoClient = MongoClients.create(uri)
//...
public BlogMongoRepository(MongoClient client,@MongoDbName String
↪  databaseName,
    @MongoCollectionName String collectionName, @Assisted ClientSession
    ↪  session) {
    this.session = session;
    this.articleCollection =
    ↪  client.getDatabase(databaseName).getCollection(collectionName);
}
//...
Document doc = articleCollection.find(session, ...)//Viene passata la session
```

Per gestire l'effettivo svolgimento delle transazioni, è implementato un transaction manager, che si occupa di inizializzare la transazione, eseguire il codice e fare il commit della transazione in caso positivo, o fare rollback in caso di fallimento. Per fare ciò si sfrutta l'interfaccia TransactionManager, implementata da BlogMongoTransactionManager. Questa classe utilizza l'interfaccia funzionale TransactionCode per applicare del codice arbitrario a una repository. Per disaccoppiare ulteriormente il TransactionManager dalla repository, e vista la volatilità delle transazioni, è implementata una factory per creare su richiesta una repository.

```java
//...
```

```
public BlogMongoTransactionManager(MongoClient client, BlogRepositoryFactory
↪  repositoryFactory) {
    this.client = client;
    this.repositoryFactory = repositoryFactory;
}

@Override
public <T> T doInTransaction(TransactionCode<T> code) throws
↪  TransactionException{
    try (ClientSession session = client.startSession()) {
        session.startTransaction();
        try {
            BlogRepository repository =
            ↪  repositoryFactory.createRepository(session);
            T result = code.apply(repository);
            session.commitTransaction();
            return result;
        } catch (Exception e) {
            session.abortTransaction();
            throw new TransactionException("Transaction failed: " +
            ↪  e.getMessage(), e);
        }
    }
}
//...
@FunctionalInterface
public interface TransactionCode<T> extends Function<BlogRepository, T>{
}
//...
public interface BlogRepositoryFactory {
        BlogRepository createRepository(ClientSession session);
}
```

2.2 Business logic

Questo livello separa l'accesso ai dati dalla logica di presentazione, ed è stato creato proprio per togliere responsabilità di business logic dal controller/presenter e trasferirla in un sottile livello intermedio: il service. L'interfaccia BlogService, implementata dalla classe BlogMongoService, si occupa di definire operazioni di alto livello che il controller richiede, e che dovranno essere tradotte in operazioni CRUD della repository. Inoltre, il service è responsabile di validare gli input che riceve (ad esempio, se un articolo esiste quando lo si vuole rimuovere dal database). I metodi del service ricevono puramente stringhe e Set di stringhe, anche se non esplicitamente richiesto dal progetto. Questo perché potrebbero esserci casi, in uno scenario reale, in cui la rappresentazione di un articolo potrebbe essere diversa nella logica di presentazione rispetto a quella del data layer. In questa maniera, il livello di presentazione non è a conoscenza di dettagli per l'accesso ai dati, come le transazioni o il formato di articoli e tag usato nel data layer, ma può comunicare attraverso un'interfaccia che generalizza il servizio

utilizzato.

```
//...
public interface BlogService {
        List<Article> getAllArticles();
        List<Article> getArticlesByTag(String tagLabel);
        Article saveArticle(String title, String content, Set<String> tags);
        Article updateArticle(String id, String title, String content,
        ↪  Set<String> tags);
        void deleteArticle(String id);
}
```

2.3 Presentation

BlogView definisce operazioni per la gestione del frame dell'applicazione, implementate dalla classe BlogSwingView. Qui vengono definiti i layout e la gestione delle componenti, i binding degli eventi e le chiamate al controller.
BlogController gestisce le richieste della GUI e delega al service le richieste per le operazioni con il database. Si occupa della gestione delle eccezioni, inoltrandole all'utente tramite GUI.
Per risolvere la ciclicità della dipendenza del pattern MVC, BlogSwingView espone un setter per il controller. Dato che verrà utilizzato Google Guice, è anche implementata una interfaccia per una factory di controller.
La classe BlogSwingApp.java implementa il main loop, in cui si istanziano le dependency necessarie, con la possibilità di utilizzare argomenti da terminale per host, porta, nome del database e della collection, grazie a Picocli.

2.4 Altre scelte di design

- Ogni articolo tiene la sua lista di tag, sfruttando la rappresentazione con documenti utilizzata da MongoDB.
- Per gestire l'edit di un articolo che ancora non è nel database, l'aggiunta o rimozione di tag sono gestiti unicamente da view e controller, in quanto le operazioni non contengono molta logica. Quindi la persistenza dei tag è gestita insieme a quella degli articoli.
- Visto che lanciare una RuntimeException è troppo generico, per migliorare la leggibilità del codice vengono implementate due eccezioni, TransactionException e ArticleNotFoundException, lanciate rispettivamente per errori di transazioni o del service.
- Per design, salvare un articolo dalla GUI non può fallire (salvo per errori di database), in quanto sono ammessi duplicati. La responsabilità dunque grava sull'utente. Possono fallire invece le operazioni che coinvolgono ricerca tramite ID univoco (modifica o cancellazione).

3 Framework utilizzati

In questa sezione vengono spiegate le configurazioni dei framework principali utilizzati nel progetto.

3.1 Google Guice

Per gestire automaticamente le dipendenze delle varie componenti e negli integration test. Viene quindi implementato un modulo custom, BlogSwingMongoDefaultModule, che si occupa di gestire i binding e istanziare le dependency per i vari layer. In particolare, le istanze di MongoClient, MongoTransactionManager e BlogMongoService sono singleton. MongoSession è creato a runtime ad ogni transazione dal transaction manager, quindi non ha bisogno di essere configurato. Le factory per repository e controller sono configurate con dei FactoryModuleBuilder, e vengono usate nell'iniezione di dipendenze con l'annotazione *@Assisted*. Per rendere configurabile più facilmente la connessione al database, sono definite annotazioni di binding per host, porta, nome del database e nome della collection, in maniera tale che la repository possa utilizzare stringhe specificate a runtime. Metodi *@Provides* sono definiti per la creazione di MongoClient e BlogSwingView, per indicare a guice come istanziare questi tipi di oggetti: il primo richiede una connection string, il secondo deve risolvere una dipendenza ciclica. In questa maniera è molto più semplice istanziare le componenti per testare e creare l'app.

3.2 Testare interagendo con il database

Utilizzare le transazioni in MongoDB richiede di creare una sessione, e per fare unit-testing della repository serve un in-memory database che permetta di utilizzarle. Il database visto durante il corso funziona per accessi regolari, ma non ha implementato ancora l'utilizzo di sessioni e transazioni. Per questo ho scelto di utilizzare Flapdoodle, un altro database in-memory che però permette di testare anche transazioni.

```
//...
import de.flapdoodle.embed.mongo.commands.ServerAddress;
import de.flapdoodle.embed.mongo.distribution.Version;
import de.flapdoodle.embed.mongo.transitions.Mongod;
import de.flapdoodle.embed.mongo.transitions.RunningMongodProcess;
import de.flapdoodle.reverse.StateID;
import de.flapdoodle.reverse.TransitionWalker;

@BeforeClass
public static void setUpServer() {
    Version.Main version = Version.Main.V8_1;
    server = Mongod.instance().transitions(version)
        .walker()
        .initState(StateID.of(RunningMongodProcess.class));
    ServerAddress addr = server.current().getServerAddress();
```

```
    connectionString = "mongodb://" + addr.getHost() + ":" + addr.getPort();
}

@AfterClass
public static void shutDownServer() {
    server.close();
}
//...
client = MongoClients.create(connectionString); //In setUp()
session = client.startSession();
```

Per gli integration test ho scelto di utilizzare Testcontainers. Con un database reale le transazioni richiedono una funzionalità aggiuntiva: il server MongoDB deve essere un Replica Set. Testcontainers offre un metodo per questo tipo di inizializzazione, rendendo semplice l'adattamento.

```
import org.testcontainers.mongodb.MongoDBContainer;
import org.testcontainers.utility.DockerImageName;

@SuppressWarnings({"resource"})
@ClassRule
private static MongoDBContainer mongoContainer =
    new MongoDBContainer(DockerImageName.parse("mongo:5"))
    .withReplicaSet();

@BeforeClass
public static void setUpBeforeClass() {
    mongoContainer.start();
}

@AfterClass
public static void tearDownAfterClass() {
    mongoContainer.stop();
}
```

3.3 Plugin per i test e report

- Sono esclusi dalla coverage con JaCoCo le classi Article, Tag e BlogSwingApp, siccome non contengono logica, e i test associati, se presenti, sono più per documentazione o verifica del funzionamento nel caso dell'app.
- Pitest è configurato per includere le classi BlogMongoRepository, MongoTransactionManager, BlogMongoService, BlogController, che contengono la logica primcipale dell'applicazione. La view è esclusa, in quanto di lunga valutazione per il numero di mutanti generati, che rende difficile e poco utile risolvere tutti i casi.
- Il build-helper-maven-plugin aggiunge le source directory per integration e end-to-end test
- I report di Surefire e Failsafe sono generati a ogni run.

3.3.1 Sonarqube

Il plugin Sonarqube è configurato in maniera tale da inviare i report di Surefire e Failsafe, escludere le classi Article, Tag e BlogSwingApp dall'analisi sulla coverage, e ignorare due regole: java:S117, che marca i nomi delle variabili auto-generate da WindowBuilder come issues, cosa che possiamo ignorare visto che è gestita automaticamente, e java:S2699, che marca i test per la GUI come non contenenti asserzioni, mancando di riconoscere i metodi di AssertJSwing. L'annotazione *NOSONAR* è utilizzata per il nome della classe di test end-to-end, in quanto termina con E2E.java, cosa che invece è corretta nell'ambito del progetto.

3.3.2 Profili maven

Tutti i test possono essere eseguiti con il comando `mvn clean verify`. Questo genererà anche i report di Surefire e Failsafe. Per facilitare lo sviluppo locale, sono definiti tre profili separati per coverage, mutation testing e code quality, rispettivamente chiamati `jacoco, mutation-testing, sonarqube`. Questa separazione consente di concentrare le run locali sul singolo task, in maniera da gestire un aspetto del testing alla volta e contenere i tempi di build. Per l'invio di report a servizi cloud sono definiti due profili, `coveralls` e `sonarcloud`.

3.4 GitHub Actions

La cartella `.github/workflows` contiene il file `maven.yml` che configura gli step di CI per il progetto. In particolare, prima si costruiscono gli artefatti, usando i profili `coveralls, mutation-testing`, poi si effettua l'analisi con Sonarcloud, con il profilo `sonarcloud`. Infine, tutti gli artefatti di report sono salvati, così da permettere una migliore comprensione in caso qualcosa andasse storto nella build.
Nota: gli artefatti vengono salvati sempre, specificando `if: {{ always() }}` nella configurazione degli step di salvataggio e report, fatta eccezione per gli screenshot di test GUI, salvati soltanto in caso di fallimento della build. Per ovviare al problema dei test con AssertJSwing, l'opzione `xvfb-run` è aggiunta prima di eseguire la build con maven, assicurandosi che venga utilizzato il virtual frame buffer di X11.

3.5 Warning e Problemi

Ho soppresso gli warning relativi ai resource leak del container MongoDB negli integration test, perché il compilatore non rileva la gestione manuale nei metodi annotati con *@BeforeClass* e *@AfterClass*.
Vista la quantità di componenti nella GUI del progetto, può succedere che in CI alcuni test relativi a dei bottoni falliscano perché non sono visibili. Ho dovuto ispezionare i log e visionare gli screenshot dei test falliti per capire che dovevo sfruttare un po' di più lo spazio, al costo di una UI più affollata.

4 Packaging e riproducibilità

La classe BlogSwingApp.java è il target del plugin `maven-assembly-plugin`, configurato per generare un *FatJar*. Per dockerizzare l'applicazione, è definito un file Dockerfile:

```
FROM eclipse-temurin:17

RUN apt-get update && apt-get install -y \
    libxext6 \
    libxrender1 \
    libxtst6 \
    libxi6 \
    libxrandr2 \
    fontconfig \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY target/blog-0.0.1-SNAPSHOT-jar-with-dependencies.jar /app/blog.jar

ENV DISPLAY=:0

ENTRYPOINT ["java", "-jar", "/app/blog.jar"]
```

E un file `docker-compose.yml`

```
services:
  mongodb:
    image: mongo:5
    container_name: blog-mongodb
    command: ["--replSet", "rs0", "--bind_ip_all"]
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data:/data/db
    healthcheck:
      test: ["CMD", "mongosh", "--quiet", "--eval",
       "try { rs.status() } catch (err) { rs.initiate({_id:'rs0',
       ↪  members:[{_id:0, host:'mongodb:27017'}]}) }"]
      interval: 5s
      timeout: 10s
      retries: 5
      start_period: 20s
    networks:
      - blog-network

  blog-app:
    build:
      context: .
      dockerfile: Dockerfile
```

```
    container_name: blog-swing-app
    depends_on:
      mongodb:
        condition: service_healthy
    environment:
      - DISPLAY=${DISPLAY}
    volumes:
      - /tmp/.X11-unix:/tmp/.X11-unix:rw
    network_mode: host
    command: [
      "--mongo-host", "${MONGO_HOST:-mongodb://localhost}",
      "--mongo-port", "${MONGO_PORT:-27017}",
      "--db-name", "${DB_NAME:-blog}",
      "--db-collection", "${DB_COLLECTION:-articles}"
    ]

volumes:
  mongodb_data:

networks:
  blog-network:
    driver: bridge
```

Il container dell'app scarica le librerie di X11, ripulendo poi i file di apt per mantenere più leggero possibile il container. L'uso di `ENTRYPOINT` nel Dockerfile permette di configurare un comando di esecuzione a cui potranno venire attaccati argomenti dalla configurazione quando si lancia `docker compose up`, tramite variabili di environment. Ancora una volta le transazioni richiedono un setup aggiuntivo per la gestione dei Replica set, che andiamo a configurare nel `docker-compose` con dei comandi per l'inizializzazione del set, più un `healthcheck` che controlla attivamente lo status del database per aspettare che sia inizializzato correttamente. I container comunicano attraverso un network, e i dati dell'app sono salvati in un volume dedicato.

4.1 Avvio dell'app

1. Clonare la repository:
   `git clone https://github.com/MichaelUnifi/blog.git`
2. Spostarsi nella cartella root della repository:
   `cd blog`
3. Generare gli artefatti:
   `mvn clean package`
4. Autorizzare docker a visualizzare la GUI sulla macchina locale (solo per Linux):
   `xhost +local:docker`

5. Assicurarsi che l'utente sia aggiunto al gruppo Docker (opzionale se già configurato) (potrebbe richiedere un reboot)
   `sudo usermod -aG docker ${USER}`

6. Avviare i container con l'opzione build per costruire l'immagine Docker dal Dockerfile:
   `docker compose up --build`