AST-SessionBuilder

Edoardo Fanciullacci - 7152664

June 2025

1 Introduzione

SessionBuilder è un'applicazione desktop java che permette la creazione di topic da approfondire o svolgere nel corso delle sessioni pianificate dall'utente. Il pattern seguito è MVC con service layer integrato. Il dominio di SessionBuilder si compone di due entità: Topic & StudySession. Esse formano una relazione bidirezionale molti a molti, in cui il Topic ha un riferimento a zero o più sessioni mentre Session un riferimento ad almeno un topic. Session presenta gli attributi date, duration, note e topicList, mentre Topic presenta name, description, difficulty e sessionList. L'interfaccia utente è stata implementata in swing e si compone di tre view: una per la gestione dei Topic esistenti e le altre per la creazione dei topic e delle sessioni rispettivamente.

2 Panoramica sulle Tecniche Applicate e i Frameworks utilizzati

SessionBuilder è stata sviluppata seguendo i principi della metodologia TDD, i cui test sono stati scritti attraverso i framework Junit4, Mockito e Assertj-Swing. Per migliorare la leggibilità dei test è stata impiegata la libreria AssertJ. La granularità dei test è stata monitorata tramite i tools Jacoco e Pit. Per la gestione della persistenza delle istanze del dominio è stato utilizzato JPA (con provider Hibernate) e PostgreSQL è stato scelto come database per l'applicazione. La correttezza delle operazioni transazionali è garantita da una classe utility di gestione che si affida a delle interfacce funzionali col ruolo di funzioni di callback. Il tool Maven è stato utilizzato per la gestione della build e delle dipendenze del progetto ed un file YAML per il workflow Github Actions è stato creato al fine di automatizzare il deployment le operazioni di controllo sulle diverse versioni del progetto. La stesura del DockerFile ha permesso di generare l'image docker dell'applicazione, utilizzata poi nell'orchestrazione del progetto multi-container guidata dal docker-compose. L'image creata viene quindi automaticamente "pushata" su DockerHub, previa login, dal sistema di CI. Il sistema di CI è arricchito tramite la connessione a Coveralls e SonarCloud che forniscono rispettivamente feedBack sulla code coverage e la code quality. Nel progetto è stato utilizzato Google Guice come framework di dependency injection.

3 Organizzazione del Progetto Maven

Il progetto SessionBuilder adotta un'architettura multi-modulo Maven che garantisce una chiara separazione delle responsabilità, favorisce la modularità e facilita la manutenzione del codice. La struttura gerarchica del progetto è orchestrata attraverso un aggregatore principale che amministra diversi moduli specializzati, ciascuno con i propri plugin e dipendenze specifiche.

3.1 Struttura Generale del Progetto

La radice del progetto è costituita dal modulo `sessionBuilder-aggregator`, che funge da contenitore principale del progetto. Questo POM di aggregazione definisce la struttura attraverso la seguente gerarchia:

```
sessionBuilder-app
sessionBuilder-aggregator
  sessionBuilder-bom
  sessionBuilder
    sessionBuilder-core
    sessionBuilder-swing
    sessionBuilder-report
```

3.2 Modulo Aggregator

Il POM aggregatore (`sessionBuilder-aggregator`) rappresenta il punto di ingresso principale per la build del progetto. Questo modulo definisce:

- **Proprietà globali di SonarQube**: Configura le esclusioni per l'analisi statica del codice, ignorando il file di configurazione `SessionBuilderApplication.java`
- **Regole di esclusione**: Definisce una serie di regole per ignorare specifici warning di SonarQube attraverso la proprietà `sonar.issue.ignore.multicriteria` :
  - **java:S120**: impone un ordine superfluo agli elementi delle interfacce
  - **javaarchitecture:S7027**: segnala le dipendenze cicliche. In sessionBuilder è presente una dipendenza ciclica tra la classe Topic e StudySession necessaria perché conseguenza dell'adozione dello standard JPA

3.2.1 Profili

docker-app Questo profilo ha come scopo quello di estendere dinamicamente la lista dei moduli da "buildare" aggiungendo sessionBuilder-app ai moduli base, e delegando ad esso il packaging Docker.

sonarqube & sonarcloud Questi due profili configurano l'integrazione dei servizi di analisi della code quality statica:

- **sonarqube**: Per istanze locali (http://localhost:9000)
- **sonarcloud**: Per il servizio cloud con configurazione specifica dell'organizzazione

3.3 Bill of Materials (BOM)

Il modulo `sessionBuilder-bom` implementa il pattern Bill of Materials per centralizzare la gestione delle versioni delle dipendenze. Questo approccio garantisce che tutte le dipendenze del progetto utilizzino versioni coordinate e gestite centralmente. In questo modo infatti le versioni di Log4j e dei moduli interni sono definite in un unico punto e possono essere attivate da tutti i moduli nella sezione dependencyManagement.

3.4 Modulo Parent sessionBuilder

Il POM parent principale (`sessionBuilder`) stabilisce la configurazione comune per tutti i sotto-moduli, che sono:

- sessionBuilder-core, per le classi di dominio e i rispettivi test unitari
- sessionBuilder-swing, per le classi UI e i rispettivi test unitari
- sessionBuilder-it, per i test di integrazione del dominio e dell'UI
- sessionBuilder-e2e, per i test e2e dell'applicazione

3.4.1 Profili

jacoco Il profilo jacoco si compone di tre esecuzioni distinte, che si occupano della configurazione dell'agente (prepare-agent), del controllo sui risultati di copertura (check) e in fine della scrittura dei report (report). Il plugin jacoco, nella sezione pluginManagement, viene attivato solo dai moduli figli sessionBuilder-core e sessionBuilder-swing, dove sono raccolti i test unitari. Sono stati esclusi dall'analisi della copertura tutte classi di test e la classe contenente il metodo main.

mutation-testing il profilo mutation-testing è incaricato di attivare il framework Pit per l'individuazione dei mutanti. La sua esecuzione prevede l'attivazione del goal mutationCoverage, che compie l'analisi Pit e genera i report. Il plugin pitest, nella sezione pluginManagement, ha come package target quello di sessionBuilder-core. I test unitari della UI sono stati volutamente esclusi dal pit-testing, principalmente per evitare interferenze con l'EDT.

3.5 Moduli Implementativi

3.5.1 sessionBuilder-core

Il modulo core contiene la logica di business e l'accesso ai dati:

- **Dipendenze runtime**: PostgreSQL driver per la connettività database
- **Dipendenze compile-time**: Jakarta Persistence API e Hibernate Core
- **Plugin attivi**: Pitest per mutation testing e JaCoCo per code coverage

3.5.2 sessionBuilder-swing

Il modulo GUI implementa l'interfaccia utente desktop:

- **Dipendenze specifiche**: JCalendar (1.4) per il date picker, AssertJ Swing per i test GUI
- **Dipendenza interna**: sessionBuilder-core per accedere alla logica di business
- **Configurazione speciale**: Pitest disabilitato (`skip=true`) data la complessità del testing delle GUI

Nota su Warning di Riflessione durante i Test È importante segnalare che, durante l'esecuzione della build viene generato a console un warning del tipo `InaccessibleObjectException`.

```
WARNING: Exception thrown by a TimerTask
java.lang.reflect.InaccessibleObjectException: Unable to make field
    int java.util.TimerTask.state accessible: module java.base does
    not "opens␣java.util" to unnamed module...
```

L'analisi del warning ha rivelato che non è presente un errore dell'applicazione, ma un'incompatibilità nota tra la libreria di test `assertj-swing` e le versioni di Java superiori alla 9. A partire da Java 9, infatti, il Java Platform Module System (JPMS) incapsula di default le API interne del JDK, proibendo l'accesso riflessivo a membri privati, una tecnica di cui `assertj-swing` fa largo uso per monitorare lo stato dei componenti Swing. Ho tentato di risolvere il warning istruendo la JVM ad autorizzare questo accesso tramite l'argomento `--add-opens java.base/java.util=ALL-UNNAMED`, configurato nel `maven-surefire-plugin`. Tuttavia, questo approccio ha generato un conflitto diretto con il plugin JaCoCo. Entrambi i plugin, infatti, tentano di configurare gli argomenti della JVM tramite la proprietà Maven `argLine`. La configurazione di Surefire sovrascrive quella impostata dall'agente JaCoCo, disabilitando di fatto l'analisi della code coverage. Nonostante i tentativi di unire le configurazioni tramite l'uso della proprietà `${argLine}`, il conflitto si è dimostrato di difficile risoluzione all'interno della complessa configurazione multi-modulo e multi-profilo. Dato che il warning è legato a una libreria di test, non impatta il funzionamento dell'applicazione finale ed è circoscritto all'ambiente di build, ho deciso di accettarlo come debito tecnico necessario per poter utilizzare la libreria `assertj-swing` su una JVM moderna.

3.5.3 sessionBuilder-it e sessionBuilder-e2e

Questi moduli gestiscono rispettivamente i test di integrazione e end-to-end:

- **Scope delle dipendenze**: Tutte marcate come `test`
- **TestContainers**: Per orchestrare container PostgreSQL durante i test
- **Build Helper Plugin**: Aggiunge directory di test custom (`src/it/java` e `src/end-to-end/java`)
- **Failsafe Plugin**: utilizzato per eseguire i test presenti in tali directory
- **Pitest e Jacoco disabilitati**: Non applicati per test di integrazione

3.5.4 Testing di Integrazione con Testcontainers

I test di integrazione del progetto si affidano al framework Testcontainers affinché i test siano eseguiti in un'istanza reale del database PostgreSQL e non in uno in-memory (come H2). Come mostrato nell'esempio in Figura 1, l'annotazione `@ClassRule` di JUnit4 viene utilizzata per avviare il container PostgreSQL una sola volta per ogni classe di test. Questo riduce significativamente il tempo di esecuzione della suite di test. Tuttavia, per garantire che ogni singolo metodo di test venga eseguito in un ambiente pulito, all'interno del metodo `@Before` viene creata una nuova istanza di `EntityManagerFactory` passando la proprietà `hibernate.hbm2ddl.auto` con valore `create-drop`. Inoltre, viene richiamato il metodo cleanDatabase(), che assicura che il database venga svuotato. In questo modo, lo schema del db viene cancellato e ricreato da zero, prevenendo che i dati di un test possano influenzare l'esito di quello successivo.

```
@SuppressWarnings("resource")
@ClassRule
public static final PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:15-alpine")
    .withDatabaseName(System.getenv().getOrDefault("POSTGRES_DB", "test"))
    .withUsername(System.getenv().getOrDefault("POSTGRES_USER", "test"))
    .withPassword(System.getenv().getOrDefault("POSTGRES_PASSWORD", "test"));

@BeforeClass
public static void setUpContainer() {
    postgres.start();
}

protected void cleanDatabase() {
    EntityManager em = emf.createEntityManager();
    EntityTransaction tx = em.getTransaction();
    try {
        tx.begin();
        em.createNativeQuery("TRUNCATE TABLE topic_studysession CASCADE").executeUpdate();
        em.createNativeQuery("TRUNCATE TABLE studysession CASCADE").executeUpdate();
        em.createNativeQuery("TRUNCATE TABLE topic CASCADE").executeUpdate();
        tx.commit();
    } finally {
        if (tx.isActive()) tx.rollback();
        em.close();
    }
}

protected abstract AbstractModule getTestSpecificModule();

@Before
public void setupBase() {
    Map<String, String> jdbcProperties = new HashMap<>();
    jdbcProperties.put("jakarta.persistence.jdbc.driver", "org.postgresql.Driver");
    jdbcProperties.put("jakarta.persistence.jdbc.url", postgres.getJdbcUrl());
    jdbcProperties.put("jakarta.persistence.jdbc.user", postgres.getUsername());
    jdbcProperties.put("jakarta.persistence.jdbc.password", postgres.getPassword());
    jdbcProperties.put("hibernate.dialect", "org.hibernate.dialect.PostgreSQLDialect");
    jdbcProperties.put("hibernate.hbm2ddl.auto", "create-drop");
    jdbcProperties.put("hibernate.show_sql", "true");
    jdbcProperties.put("hibernate.format_sql", "true");

    injector = Guice.createInjector(
        Modules.override(new AppModule("dummy-persistence-unit", Collections.emptyMap()))
            .with(new TestEntityManagerFactoryModule(jdbcProperties, "sessionbuilder-test"),
                getTestSpecificModule())
    );
    emf = injector.getInstance(EntityManagerFactory.class);
    transactionManager = injector.getInstance(TransactionManager.class);
    cleanDatabase();
    onSetup();
}
```

Figura 1: Estratto della classe `BaseBackendIntegrationTest` che mostra l'uso di `@ClassRule` per il container e la configurazione dell'EntityManagerFactory nel metodo di setup.

3.5.5 sessionBuilder-report

Modulo dedicato all'aggregazione dei report di copertura:

- **Packaging**: `pom` (non produce artifacts)
- **Jacoco report-aggregate**: Combina i dati di copertura da tutti i moduli. L'id del profilo è sempre jacoco (come in sessionBuilder). In questo modo se viene attivato tale profilo dalla build ottengo sia il report aggregato che quelli nei singoli moduli.
- **Coveralls integration**: Invia i report aggregati al servizio Coveralls. Il binding è nuovamente con la fase verify.
  OSSERVAZIONE: inizialmente avevo provato a far leggere a coveralls il report aggregato. Sfortunatamente però in questo modo si creava una dipendenza del profilo coveralls da quello jacoco, un anti pattern da evitare. Avevo pensato quindi di inserire il plugin di creazione del report aggregato anche nel profilo coveralls, ma a quel punto ci sarebbero potuti essere conflitti tra i due profili nella stesura del report aggregato. Ho optato quindi in fine per gestire coveralls coi report dei singoli moduli maven

3.6 Modulo sessionBuilder-app

sessionBuilder-app è il modulo applicativo finale orchestrato separatamente dall'aggregatore principale. Il suo scopo è quello generare il fatJar dell'applicazione e attivare il plugin docker-maven-plugin per gestire la creazione dell'immagine e l'esecuzione dei container Docker.

4 Dettagli implementativi

Il codice di sessionBuilder è stato sviluppato seguendo il Dependency Inversion Principle (DIP), privilegiando perciò le dipendenze da astrazioni (interfacce) invece che da implementazioni concrete. Questo lo si può rivedere in diversi aspetti dell'architettura:

4.1 Separazione tra interfacce e implementazioni

Il sistema definisce interfacce chiare per l'accesso ai dati, per i servizi, per il gestore delle transazioni e per l'aggiornamento dell'interfaccia utente:

- `TopicRepositoryInterface` implementata da `TopicRepository`
- `StudySessionRepositoryInterface` implementata da `StudySessionRepository`
- `TopicServiceInterface` implementata da `TopicService`
- `StudySessionInterface` implementata da `StudySessionService`
- `TransactionManager` implementata da `TransactionManagerImpl`
- `sessionViewCallback & TopicViewCallback` implementate da `TopicAndSessionManager`

4.2 Dependency Injection con Google Guice

Per gestire le dipendenze tra i vari componenti nel rispetto del Dependency Inversion Principle, il progetto utilizza il framework Google Guice. La classe principale `SessionBuilderApplication` ha il duplice compito di:

1. Raccogliere i parametri di configurazione dinamici (da CLI e variabili d'ambiente)
2. Avviare il processo di dependency injection

Durante l'avvio, l'applicazione crea l'istanza di `Injector` combinando due moduli:

- `AppModule(persistenceUnit, properties)`: Modulo principale che riceve:
  - Nome dell'unità di persistenza (es. "sessionbuilder-test")
  - Mappa di proprietà JDBC (URL, utente, password)
- **Modulo anonimo**: Definisce i binding per i componenti UI (es. il `JFrame` principale)

Questo design garantisce il disaccoppiamento tra:

- La logica di configurazione (in `SessionBuilderApplication`)
- La definizione delle dipendenze (in `AppModule`)
- La costruzione della UI (modulo anonimo)

Configurazione della Persistenza Il sistema combina configurazione statica e dinamica attraverso due elementi integrati:

1. **Configurazione base**: Definiti nel file `META-INF/persistence.xml`:
   - Unità di persistenza per diversi ambienti (prod/test/e2e)
   - Comportamento DDL (`hbm2ddl.auto`)
   - Proprietà tramite placeholder (es. `$DB_HOST:localhost`)
2. **Override runtime**: Proprietà dinamiche passate via codice

Listing 1: Unità di persistenza in persistence.xml

```
<persistence-unit name="sessionbuilder-prod">
<property name="hibernate.hbm2ddl.auto" value="update"/>
</persistence-unit>
<persistence-unit name="sessionbuilder-test"> <property name="hibernate.hbm2ddl.auto
    " value="create-drop"/> </persistence-unit><persistence-unit name="
    sessionbuilder-e2e"> <property name="hibernate.hbm2ddl.auto" value="update"/> </
    persistence-unit>
```

Meccanismo di Override `SessionBuilderApplication` costruisce una mappa di proprietà JDBC basata su:

- Parametri da riga di comando tramite Picocli (es. `--postgres-host`)
- Variabili d'ambiente (es. `DB_PASSWORD`)
- Valori di default

Questa mappa viene passata a `AppModule`, permettendo di sovrascrivere le proprietà definite in `persistence.xml` secondo la gerarchia:

1. **Max priorità**: Proprietà runtime (mappa)
2. **Media priorità**: Variabili d'ambiente
3. **Min priorità**: Valori nel `persistence.xml`

Nota sulle configurazioni CI: Nella build di GitHub Actions, i parametri da riga di comando non vengono utilizzati - la configurazione avviene esclusivamente tramite variabili d'ambiente. L'implementazione dei parametri CLI è stata mantenuta per completezza. Lo stesso si può dire per la configurazione nel persistence.xml.

Listing 2: Creazione EntityManagerFactory con override

```
@Provides @Singleton
EntityManagerFactory provideEntityManagerFactory() {
// Le proprieta' in dbProperties sovrascrivono:
// - I placeholder nel persistence.xml
// - Le proprieta' statiche nel persistence.xml
return EmfFactory.createEntityManagerFactory(
this.persistenceUnit,
this.dbProperties
);
}
```

4.3 Architettura a Livelli

Ogni livello dipende da astrazioni di livelli sottostanti:

- **Business Layer & Data Access Layer** dipendono dall'interfaccia TransactionManager. Ex: (`TopicRepository`) → dipende da `TransactionManager`

Eccezione alla regola sono i Controller, che dipendono sia dalle interfacce dei servizi che dall'interfaccia della view, di un livello superiore. Quando il metodo di un servizio è richiamato, il controller aggiorna la view di conseguenza.

```
 8 public class StudySessionController {
 9
10     @Inject
11     private StudySessionInterface service;
12
13     @Inject
14     private SessionViewCallback viewCallBack;
15
16     private static final String STRING_ERROR = "Error: ";
17
18
19     public StudySession handleCreateSession(LocalDate date, int duration, String note, List<Topic> topics) {
20         try {
21             StudySession session = service.createSession(date, duration, note, topics);
22             if (viewCallBack != null) {
23                 viewCallBack.onSessionAdded(session);
24             }
25             return session;
26         } catch(Exception e) {
27             if (viewCallBack != null) {
28                 viewCallBack.onSessionError(STRING_ERROR + e.getMessage());
29             }
30             throw e;
31         }
32     }
33
34     public StudySession handleGetSession(long sessionId) {
35         try {
36             return service.getSessionById(sessionId);
37         } catch(Exception e) {
38             if (viewCallBack != null) {
39                 viewCallBack.onSessionError(STRING_ERROR + e.getMessage());
40             }
41             throw e;
42         }
43
44     }
```

Figura 2: StudySessionController

4.4 TopicAndSessionManager & implementazione delle viewCallBack

TopicAndSessionManager è il JFrame che contiene tutta l'interfaccia grafica dell'applicazione. È strutturato come un CardLayout che permette di passare tramite click dal pannello di gestione delle Jlist dei topic e delle sessioni al pannello col form per la creazione delle sessioni o a quello con il form per la creazione dei topic. La classe implementa le due interfacce TopicViewCallBack e SessionViewCallBack, utilizzate dai controller, fornendo i metodi necessari ad aggiornare la UI sulla base dei cambiamenti avvenuti nel database.

```java
@Override
public void onSessionUpdated(StudySession updatedSession) {
    for (int i = 0; i < studySessionModel.getSize(); i++) {
        StudySession session = studySessionModel.getElementAt(i);
        if (session.getId() == updatedSession.getId()) {
            session.setIsComplete(updatedSession.isComplete());
            studySessionModel.setElementAt(session, i);
            break;
        }
    }
    if (topicPanel != null && topicPanel.getSessionModel() != null) {
        DefaultListModel<StudySession> topicSessionModel = topicPanel.getSessionModel();
        for (int i = 0; i < topicSessionModel.getSize(); i++) {
            StudySession session = topicSessionModel.getElementAt(i);
            if (session.getId() == updatedSession.getId()) {
                session.setIsComplete(updatedSession.isComplete());
                topicSessionModel.setElementAt(session, i);
                break;
            }
        }
    }
}
```

Figura 3: Implementazione del metodo di interfaccia onSessionUpdated. Quando una sessione viene completata questa viene individuata nelle JList dei pannelli in cui è presente e viene aggiornato lo stato di completamento

4.5 Gestione delle Transazioni

Una delle sfide del progetto è stata l'implementazione di un sistema di gestione transazionale che fosse flessibile.

Interfacce Funzionali L'applicazione utilizza diverse interfacce funzionali per la gestione delle transazioni. Ogni interfaccia rappresenta un diverso tipo di operazione:

- `TransactionCode<T>`: Operazioni dirette con EntityManager
- `TopicTransactionCode<T>`: Operazioni specifiche per Topic repository
- `StudySessionTransactionCode<T>`: Operazioni specifiche per StudySession repository
- `MultiRepositoryTransactionCode<T>`: Operazioni che coinvolgono entrambi i repository

4.5.1 Implementazione del Transaction Manager

La classe `TransactionManagerImpl` centralizza tutta la logica transazionale, fornendo metodi specifici per ogni tipo di operazione.

```java
public class TransactionManagerImpl implements TransactionManager {

    private static final ThreadLocal<EntityManager> emHolder = new ThreadLocal<>();
    private EntityManagerFactory emf;
    private TopicRepositoryInterface topicRepository;
    private StudySessionRepositoryInterface sessionRepository;

    @Inject
    public TransactionManagerImpl(EntityManagerFactory emf,
            TopicRepositoryInterface topicRepository,
            StudySessionRepositoryInterface sessionRepository) {
        this.emf = emf;
        this.topicRepository = topicRepository;
        this.sessionRepository = sessionRepository;
    }

    @Override
    public EntityManager getCurrentEntityManager() {
        EntityManager em = emHolder.get();
        if (em == null) {
             throw new IllegalStateException("EntityManager non trovato. La transazione non è stata avviata correttamente");
        }
        return em;
    }

    @Override
    public ThreadLocal<EntityManager> getEmHolder() {
        return emHolder;
    }

    @Override
    public <T> T doInTransaction(TransactionCode<T> code) {
        if(emHolder.get() != null) {
            return code.apply(getCurrentEntityManager());
        }
        EntityManager em = emf.createEntityManager();
        emHolder.set(em);
        EntityTransaction transaction = em.getTransaction();
        try {
            transaction.begin();
            T result = code.apply(em);
            transaction.commit();
            return result;
        } catch (Exception e) {
            if (transaction.isActive()) {
                transaction.rollback();
            }
            throw e;
        } finally {
            emHolder.remove();
            em.close();
        }
    }
```

Figura 4: Implementazione del TransactionManager - metodo base

Pattern Template Method per la Gestione degli Errori Ogni metodo transazionale segue lo stesso pattern template:

1. Creazione di EntityManager
2. Inizio transazione
3. Esecuzione del codice business
4. Commit in caso di successo
5. Rollback in caso di eccezione
6. Chiusura dell'EntityManager

L'utilizzo del ThreadLocal permette ad ogni thread di avere una copia personale dell'entityManager. In tal modo tutte le operazioni che avvengono all'interno della stessa transazione fanno uso di un unico entityManager, senza crearne di nuovi. La generazione di un nuovo contesto di persistenza, infatti, comporterebbe il passaggio di parte delle entità da persisted a detached, impedendo il corretto svolgimento delle operazioni sul database. Questo soluzione è stata guidata dalla necessità di avere metodi di repository che facessero uso dell'em generato dalle transazioni attivate nei service.

In un primo momento infatti avevo cercato di annidare/concatenare le transazioni (i metodi di repository erano transazionali), ma senza successo.

```
    public StudySession findById(long id) {
    EntityManager em = tm.getCurrentEntityManager(); // Recupera EM da ThreadLocal
    return em.createQuery(...).getSingleResult();
}
```

OSS: Ho deliberatamente lasciato la classe `TransactionManagerImpl` invariata nonostante SonarCloud segnalasse Duplication-Code per motivi di leggibilità.

5 Automazione CI/CD con GitHub Actions

Il workflow github actions presenta i seguenti elementi:

- **Esecuzione di Test GUI in Ambiente Headless**: Una sfida notevole per le applicazioni desktop in ambiente CI è l'esecuzione di test dell'interfaccia grafica. Il workflow risolve questo problema utilizzando il comando `xvfb-run`, che fornisce un display server X virtuale in-memory. Questo permette ai test end-to-end basati su AssertJ-Swing di essere eseguiti correttamente in un ambiente headless come quello di GitHub Actions, simulando le interazioni utente senza la necessità di un'interfaccia grafica fisica.
- **Gestione Robusta degli Artefatti**: Per facilitare il debugging, il workflow è configurato per archiviare sempre (`if: ${{ always() }}`) gli artefatti di test (report Surefire, JaCoCo, Pit), anche in caso di fallimento della build. Questo approccio garantisce che gli sviluppatori abbiano sempre accesso ai log dettagliati per diagnosticare rapidamente la causa di un errore.
- **Conditional Deployment su DockerHub**: L'ultimo passo del workflow gestisce il deployment continuo. La pubblicazione dell'immagine Docker su DockerHub è condizionata a due fattori: deve provenire dalla build su JDK 17 (quella di riferimento) e l'evento scatenante deve essere un push sul branch `main`. Questa regola assicura che solo le versioni stabili e validate del software vengano rilasciate, separando gli ambienti di sviluppo da quello di produzione.

6 Orchestrazione Multi-Container con Docker e Docker Compose

6.1 Containerizzazione dell'applicazione

Una delle sfide principali è stata la containerizzazione dell'applicazione. La soluzione adottata è un pattern a tre componenti che permette di eseguire l'interfaccia grafica all'interno del container e di visualizzarla sull'host:

1. **Librerie di Sistema nel Dockerfile**: L'immagine Docker non si limita a includere il JRE, ma installa esplicitamente le librerie di sistema necessarie per il rendering grafico e sonoro (es. `libxext6`, `libxrender1`, `libasound2`). Questo prepara il container a gestire operazioni grafiche.

2. **Forwarding del Display X11**: Nel file `docker-compose.yml`, il servizio `sessionbuilder-app` è configurato per inoltrare la sua interfaccia grafica. Questo avviene tramite:
    - Il mounting del socket X11 dell'host nel container (`- /tmp/.X11-unix:/tmp/.X11-unix`).
    - L'impostazione della variabile d'ambiente `DISPLAY`, che indica all'applicazione nel container dove si trova il display server a cui inviare l'output grafico.

6.2 Gestione delle Dipendenze e dello Stato

Il file `docker-compose.yml` definisce una rete custom (`my-network`) per garantire l'isolamento e la risoluzione dei nomi tra i servizi. La gestione delle dipendenze e dello stato dei servizi stateful è un altro punto chiave:

- **Persistenza dei Dati**: Per i servizi stateful come i database (`postgresdb`, `db`) e SonarQube, vengono utilizzati dei **named volumes** (es. `postgres_data`, `sonarqube_data`). Questa è una best practice che separa il ciclo di vita dei dati da quello dei container, garantendo che le informazioni non vengano perse quando un container viene rimosso o ricreato.
- **Controllo dello Stato dei Servizi (Healthcheck)**: Entrambi i container dei database includono una sezione `healthcheck`. Questo permette a Docker di monitorare non solo se il container è in esecuzione, ma se il servizio al suo interno (PostgreSQL) è effettivamente pronto ad accettare connessioni.
- **Gestione dell'Ordine di Avvio**: Viene fatto uso del comando `depends_on`. Il servizio `sonarqube` attende esplicitamente che il suo database sia in stato *healthy* (`condition: service_healthy`), un approccio volto ad evitare race condition. Il servizio principale `sessionbuilder-app`, invece, utilizza un `depends_on` semplice, che garantisce solo l'avvio del container del database ma non la sua effettiva prontezza. Pertanto il Dockerfile integra lo script wait-for-it.sh nel comando di avvio (CMD). Questo script controlla attivamente la porta del database postgresdb e lancia l'applicazione Java (.jar) solo quando la connessione è possibile.

7 Guida all'Esecuzione dell'Applicazione

L'applicazione SessionBuilder può essere avviata seguendo due flussi distinti, a seconda dello scopo dell'utente. Il primo è pensato per un utente finale che desidera semplicemente utilizzare il software, mentre il secondo è rivolto ai collaboratori fittizzi del progetto.

7.1 Scenario 1: Avvio per l'Utente Finale (tramite Immagine Docker Hub)

Questo scenario sfrutta l'immagine Docker pre-compilata e pubblicata su Docker Hub, garantendo un avvio rapido senza la necessità di buildare il codice.

7.1.1 Procedura di Avvio

1. **Download dell'Immagine Docker:** Poiché il file `docker-compose.yml` è configurato per dare priorità all'uso di immagini locali, il primo passo obbligatorio è scaricare l'immagine dell'applicazione da Docker Hub.

```
docker pull edoardofanciu/sessionbuilder-app:latest
```

2. **Preparazione dell'Ambiente:** Creare una directory dedicata per l'esecuzione e scaricare al suo interno il file di orchestrazione.

```
mkdir sessionbuilder-run
cd sessionbuilder-run
curl -O https://raw.githubusercontent.com/edoardof01/sessionBuilder/main \
/sessionBuilder-app/docker-compose.yml
```

3. **Abilitazione del Display (solo per Linux):** Per permettere al container di mostrare l'interfaccia grafica sull'host, è necessario autorizzare le connessioni dal display server locale.

```
xhost +local:docker
```

4. **Avvio dell'Applicazione:** Eseguire Docker Compose per avviare l'applicazione e il database associato.

```
docker-compose up
```

   L'interfaccia grafica di SessionBuilder apparirà dopo pochi istanti. Per terminare l'esecuzione, premere `Ctrl+C` nel terminale.

7.2 Scenario 2: Avvio per il Collaboratore

Questo flusso è pensato per chi, come richiesto dalle direttive d'esame, clona il repository e intende compilare, testare ed eseguire l'applicazione partendo dal codice sorgente.

7.2.1 Procedura di Avvio

1. **Clonazione del Repository:** Ottenere una copia locale del codice sorgente.

```
git clone https://github.com/edoardof01/sessionBuilder.git sessionBuilder-
    aggregator
cd sessionBuilder-aggregator
```

2. **Compilazione e Creazione dell'Immagine Locale:** Eseguire il build completo tramite Maven. Grazie al profilo `app-docker`, questo comando non solo compilerà il codice e lancerà i test, ma costruirà anche l'immagine Docker dell'applicazione direttamente in locale.

```
mvn clean install -Papp-docker
```

3. **Navigazione nella Directory dell'Applicazione:** Spostarsi nel sottomodulo contenente il file `docker-compose.yml`.

```
cd sessionBuilder-app
```

4. **Abilitazione del Display (solo per Linux):** Come per lo scenario precedente, autorizzare la connessione al display server.

```
xhost +local:docker
```

5. **Avvio dell'Applicazione:** Lanciare la stack con Docker Compose. A differenza dello scenario precedente, Docker Compose rileverà l'immagine buildata localmente al passo 2 e la utilizzerà, senza tentare di scaricarla da Docker Hub.

```
docker-compose up
```