Simple App to Manage Reservations of a Guesthouse

Advanced Tools and Techniques for Software Development

Università degli Studi di Firenze
Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Magistrale in Informatica - Data Science

Elia Mercatanti - 6425149

Maggio 2021

Indice

1 Struttura del Progetto

Il software sviluppato implementa una semplice app per gestire le prenotazioni di una guesthouse o di un piccolo albergo. Lo scopo di questo progetto è stato quello di costruire una semplice applicazione per sfruttare e mostrare i vari framework per il testing del codice, seguendo la tecnica di sviluppo **TDD**, e le tecniche per automatizzare tutto il processo di build sfruttando anche il **Continuous Integration** (CI).

Il progetto è stato sviluppato in **Java** utilizzando **MongoDB** come database per gestire le prenotazioni, lanciato attraverso i container di **Docker**, **Gradle** come sistema di build automation e **GitHub Actions** come **CI**.

In generale l'applicazione permette all'utente di inserire degli ospiti (**Guest**) nel database, con tutte le informazioni utili per la prenotazione, ovvero il nome, cognome, email e numero di telefono, e inoltre consente di inserire i dati per le prenotazioni (**Booking**) che comprendono le date di check in e check out, la camera scelta per il soggiorno (**Room**), il numero di *guests* che occuperanno la camera, e il *guest* a cui è riferita la prenotazione. Inoltre è possibile ricercare e mostrare sia la lista dei guest inseriti nel database sia eseguire una serie di ricerche sulle prenotazioni, ovvero è possibile ricercare tutte quei *bookings* che rientrano in una coppia di date, tutte quelle relative ad una particolare stanza o ad un determinato guest, oppure semplicemente mostrarle tutte. Infine è anche possibile eliminare sia i *guest* sia i *bookings*.

1.1 Vincoli e Restrizioni

Per gestire correttamente i *guest* e i *bookings* sono stati introdotti alcuni vincoli, anche per rendere l'applicazione più interessante e realistica.

Per i *guest* viene richiesto all'utente di inserire delle email e dei numeri di telefono che abbiano un formato valido:

- *Email*: viene richiesto all'utente di inserire un email con un formato simile a prefix@domain, dove nel prefisso e nel dominio l'utente può inserire qualsiasi carattere.
- *Numero di Telefono*: viene richiesto all'utente di inserire un numero di telefono con un formato simile a (+1)0000000000, dunque solo numeri ad eccezione del prefisso facoltativo (+1) che può contenere un simbolo "+". Inoltre viene richiesto una lunghezza massima di 15 numeri.

Per quanto riguarda invece i *bookings* anche in questo caso viene richiesto all'utente di inserire delle date con un formato valido ma anche sensate, ovvero:

- La data di check in e di check out deve avere un formato simile a dd(/.-)mm(/.-)yyyy o yyyy(/.-)mm(/.-)dd, ovvero le date possono essere inserite sia in formato giorno/mese/anno sia anno/mese/giorno utilizzando un tra i possibili delimitatori: "/", "." o "-".

- Le date inserite devono ovviamente essere delle date valide con dei numeri di giorno, mese e anno sensati.
- La data di check out deve sempre essere successiva a quella di check in. Per prenotare una stanza per un solo giorno l'utente può utilizzare il check-out al giorno successivo.

Inoltre l'applicazione esegue dei controlli per verificare se la richiesta di prenotazione inserita può essere mandata in porto correttamente, ovvero:

- Il numero di *guests* che occuperanno la stanza non può superare la capienza massima di quest'ultima, ovvero il numero di letti disponibili.
- Una stanza può essere prenotata solo se non esistono altre prenotazioni per quella stanza nel periodo selezionato con le date di check in e check out.

1.2 Classi Utilizzate

Per gestire tutte queste funzionalità è stata implementata un interfaccia (GUI) sviluppata in Java Swing utilizzando il pattern Model–View–Presenter (MVP), derivazione del pattern Model–View–Controller (MVC). Sono state dunque implementate le seguenti classi:

- **Package App Swing**: Contiene la classe *GuesthouseSwingApp* utilizzata per lanciare l'applicazione e che utilizza *picocli*, come parser per argomenti passati da linea di comando per passare all'applicazione le informazioni per la connessione al database e per specificare le sue collezioni.
- **Package Controller**: Qui sono presenti le classi che gestiscono i due controller dell'applicazione *GuestController* per i *guests* e *BookingController* per i *bookings*. Entrambe ricevono le richieste dalla GUI, controllano che non ci siano problemi e nel caso passano le richieste alle corrette repositories per poi comandare alla GUI di stampare i risultati, altrimenti comunicano alla GUI di lanciare un messaggio di errore. I vincoli e restrizioni spiegati precedentemente vengono gestiti in queste due classi delegando alcuni controlli alla classe di validation.
- **Package Model**: Qui abbiamo le classi dei modelli dell'applicazione Guest, Booking e Room con i vari *getter* and *setter*. Nel caso di *Room*, per non complicare troppo l'applicazione è stato deciso di utilizzare una semplice enumerazione che setta per ogni tipo di stanza disponibile il numero di letti presenti.
- **Package Repository**: Contiene le classi delle due repositories che controllano le operazioni sul database per i *guests* e per i *bookings*. rispettivamente *GuestMongoRepository* e *BookingMongoRepository*. Entrambe utilizzano MongoDB come database e gestiscono richieste, inserimenti e

cancellazioni attraverso il meccanismo dei POJOs di MongoDB. Anche la creazione degli Id per tutti i documenti creati viene lasciata gestire direttamente dal database in modo che siano sempre generati in modo unico.

- **Package Validation**: Contiene la classe *ControllerInputValidator* che gestisce i controlli per la validazione dei dati dei *guests* e delle date dei *bookings* discussi precedentemente.

- **Package View**: Qui invece abbiamo la classe *GuesthouseSwingView* che gestisce la GUI e i suoi controlli. Dialoga con i due controller e stampa i risultati delle richieste di quest'ultimi. La GUI è composta da due tab, una con il form per gestire i *guest* e l'altra con il form per gestire i *bookings*, inoltre è presente un error log comune ad entrambe le tab per la stampa degli errori. Nella tab dei *bookings* la lista che mostra i dettagli delle varie prenotazioni è utilizzata sia per visualizzare i bookings aggiunti al database sia per mostrare i risultati delle ricerche sulle prenotazioni.

2 Tecniche e Strumenti Utilizzati

2.1 Version Control System con Git

Per controllare il versionamento del software durante tutta la fase di sviluppo è stato utilizzato il software **Git**, in congiunzione con la piattaforma **GitHub**. Di seguito viene riportato il link al progetto:
`https://github.com/elia-mercatanti/guesthouse-reservations`.

Nonostante il lavoro sia stato svolto da un singola persona è stato sfruttato comunque il meccanismo delle pull-request offerte da *GitHub*, grazie alle quali è stato possibile effettuare vari check di integrità della build su **GitHub Actions** con **Coveralls** e **SonarCloud**.

2.2 Build Automation con Gradle

Come già accennato, come sistema di *build automation* è stato utilizzato **Gradle**, strumento abbastanza recente rispetto a *Maven*, in continuo sviluppo, e che permette una configurazione pressoché totale della build del progetto. *Gradle* è attualmente il maggior competitor di *Maven* per quanto riguarda la build automation e rispetto a quest'ultimo i suoi maggiori punti di forza sono sostanzialmente quattro: **flessibilità**, **prestazioni**, **esperienza utente** e **gestione delle dipendenze**.

Per quanto riguarda la flessibilità sia *Gradle* che *Maven* si basano sul meccanismo "convention over configuration". Tuttavia, *Maven* fornisce un modello molto rigido che rende la personalizzazione difficoltosa e talvolta impossibile. Sebbene ciò possa rendere più facile la comprensione di qualsiasi build di *Maven*, purché non si abbiano requisiti speciali, lo rende anche non molto adatto

a diversi problemi di automazione. *Gradle*, d'altra parte, è costruito con l'idea di dare all'utente un grande potere di personalizzazione ma anche una grande responsabilità.

Migliorare il tempo di build è uno dei modi più diretti per distribuire il codice più velocemente. Sia *Gradle* che *Maven* impiegano una qualche forma di building e risoluzione delle dipendenze in parallelo. Le differenze maggiori sono i meccanismi di *Gradle* per evitare il lavoro ripetuto inutilmente e l'incrementalità. Le tre principali caratteristiche che rendono Gradle molto più veloce di Maven sono:

- *Incrementalità*: *Gradle* evita il lavoro monitorando l'input e l'output delle attività ed eseguendo solo ciò che è necessario e elaborando solo i file modificati quando possibile.
- *Build Cache*: riutilizza gli output di qualsiasi altra build *Gradle* con gli stessi input, anche tra macchine diverse.
- *Gradle Daemon*: un processo di lunga durata che mantiene "calde" le informazioni sulla build nella memoria.

Queste e altre caratteristiche sulle prestazioni rendono *Gradle* almeno due volte più veloce per quasi tutti gli scenari.

Dato il mandato più lungo *Maven* ha in genere però un supporto tramite IDE migliore per molti utenti. Tuttavia, il supporto IDE di *Gradle* continua a migliorare rapidamente. Sebbene gli IDE siano importanti, un gran numero di utenti preferisce eseguire operazioni di compilazione tramite un'interfaccia a riga di comando. *Gradle* fornisce anche una moderna CLI che ha funzionalità di logging e completamento della riga di comando migliori. Infine, *Gradle* fornisce un'interfaccia utente interattiva basata sul Web per il debug e l'ottimizzazione delle build: *build scans*. Queste scan possono anche essere ospitate in locale per consentire a un'organizzazione di raccogliere la cronologia delle build e fare analisi delle tendenze, confrontare le build per il debug o ottimizzare i tempi di build.

Entrambi i sistemi di compilazione forniscono funzionalità integrate per risolvere le dipendenze da archivi configurabili ed entrambi sono in grado di memorizzare nella cache le dipendenze in locale e scaricarle in parallelo. *Maven* consente di ignorare una dipendenza, ma solo in base alla versione. *Gradle* fornisce una selezione delle dipendenze personalizzabile e regole di sostituzione che possono essere dichiarate per gestire le dipendenze indesiderate a livello di progetto. *Maven* inoltre ha pochi scope di dipendenza incorporati, che impongono architetture di moduli scomode in alcuni scenari comuni. *Gradle* consente invece scope di dipendenza personalizzati, che forniscono build modellate meglio e più veloci. Inoltre, in qualità di produttore di librerie, *Gradle* consente ai produttori di dichiarare le dipendenze "api" e "implementation" per impedire che le librerie indesiderate penetrino nei percorsi di classe dei consumatori. *Gradle* infine supporta completamente le varianti di funzionalità e le dipendenze opzionali.

La procedura di assemblamento del software è però intrinsecamente diversa: se in *Maven* abbiamo un ciclo di vita rigido e ben definito, in cui i vari *goal* vengono eseguiti in un determinato ordine e in maniera sequenziale, *Gradle* si basa sulla definizione di *tasks* (intuitivamente possiamo vederli come un corrispettivo dei *goal*), il cui ordine di esecuzione viene definito dal task graph, un grafo aciclico che definisce l'ordine di esecuzione dei vari compiti.

2.2.1 Configurazione di build.gradle

Il file in cui vengono definite le dipendenze del progetto, i plugins e altre configurazioni è il file "build.gradle". Uno dei suoi vantaggi è indubbiamente l'utilizzo di un DSL Groovy-like, al posto del verboso xml utilizzato da *Maven*.

Nel caso di questo progetto sono stati utilizzati i seguenti plugins 1:

```
plugins {
    // Apply the application plugin to add support for building an application in Java.
    id 'application'

    // Apply PIT plugin to perform mutation testing.
    id 'info.solidsoft.pitest' version "${pitPluginVersion}"

    // Apply JaCoCo plugin to generate code coverage.
    id 'jacoco'

    // Apply Coveralls plugin to send coverage data to coveralls.io.
    id 'com.github.nbaztec.coveralls-jacoco' version "${coverallsPluginVersion}"

    // Apply SonarQube plugin to analyze the project with SonarCloud.
    id 'org.sonarqube' version "${sonarQubePluginVersion}"

    // Plugin for generating fatjars, add shadowJar task to manage that.
    id 'com.github.johnrengelman.shadow' version "${fatjarPluginVersion}"

    // Plugin that provides a task to determine which dependencies have updates.
    id "com.github.ben-manes.versions" version "${gradleVersionsPluginVersion}"

    // Plugin that allows specifying test sets.
    id 'org.unbroken-dome.test-sets' version "${testSetsPluginVersion}"

    // Plugin that allows managing Docker images and containers.
    id 'com.bmuschko.docker-remote-api' version "${gradleDockerPluginVersion}"
}
```

Figura 1: Gradle Plugins

- **application**: aggiunge una serie di task per la creazione ed il lancio di un'applicazione in Java, estende il plugin **java** di gradle che comprende diversi task come ad esempio quelli per la compilazione del codice sorgente e dei test.
- **pitest**: aggiunge una serie di task per eseguire *mutation testing*.
- **jacoco**: aggiunge una serie di task per eseguire *code coverage*.
- **coveralls-jacoco**: aggiunge una serie di task per eseguire la *coverage* con coveralls.

- **sonarqube**: aggiunge una serie di task per eseguire una scansione del codice per misurare la code quality da analizzare poi tramite *SonarCloud*.
- **shadow**: aggiunge una serie di task per creare in modo semplificato dei fat/uber JARs dell'applicazione.
- **gradle-versions-plugin**: aggiunge una serie di task per determinare quali dipendenze hanno degli aggiornamenti.
- **test-sets**: che consente di specificare in modo semplice source set e tasks che possono lanciare test come integration o end to end tests.
- **gradle-docker-plugin**: plugin che consente la gestione di immagini e contenitori Docker attraverso l'uso di task personalizzabili.

Di seguito vengono invece riportate tutte le dipendenze utilizzate 2. Da notare che alcune di esse sono solo legate all'implementazione dei test attraverso lo scope *testImplementation* oppure anche solo a runtime durante i test con lo scope *testRuntimeOnly*, inoltre le dipendenze vengono risolte consultando la repository di *Maven Central*. Le versioni delle dipendenze e dei plugins sono specificate nel file "gradle.properties" per mantenere le build facilmente riproducibili.

```
53 repositories {
54     // Use Maven Central.
55     mavenCentral()
56 }
57
58 // Dependencies declarations.
59 dependencies {
60     // Use JUnit 5 Jupiter API and Engine for testing.
61     testImplementation 'org.junit.jupiter:junit-jupiter-api:' + junitJupiterApiVersion
62     testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:' + junitJupiterEngineVersion
63
64     // Use JUnit 4 for E2E GUI tests.
65     testCompileOnly 'junit:junit:' + junit4Version
66     testRuntimeOnly 'org.junit.vintage:junit-vintage-engine:' + junitVintageEngineVersion
67
68     // Use Mockito with JUnit 5 for mocking during tests.
69     testImplementation 'org.mockito:mockito-junit-jupiter:' + mockitoVersion
70
71     // Use AssertJ and Swing for fluent style testing and GUI tests.
72     testImplementation 'org.assertj:assertj-swing-junit:' + assertJSwingVersion
73
74     // Use MongoDB Java Server, fake implementation of the core MongoDB server, for unit tests.
75     testImplementation 'de.bwaldvogel:mongo-java-server:' + mongoDBJavaServerVersion
76
77     // Use MongoDB Java Driver for managing a MongoDB database.
78     implementation 'org.mongodb:mongodb-driver-legacy:' + mongoDBJavaDriverVersion
79
80     // Use Logback Classic Module to see Mongo Java Driver logs.
81     implementation 'ch.qos.logback:logback-classic:' + logbackClassicVersion
82
83     // Use Picocli for java command line parser.
84     implementation 'info.picocli:picocli:' + picocliVersion
```

Figura 2: Gradle Dependencies

Inoltre il progetto è stato configurato con Java 8 attraverso l'uso del toolchain Java che permette di configurare con quale versione di Java compilare tutto il

codice del progetto e consente a *Gradle* di andare automaticamente a scaricare la corretta versione del JDK nel caso in cui non venga trovata alcuna JVM corrispondente nel sistema dell'utente.

Successivamente grazie all'uso del plugin "test-sets" sono stati definiti due nuovi source set per gli integration e end to end tests, come si può vedere in 3. In automatico il plugin va anche a creare degli scope delle dipendenze specifici per questi source set e due task con lo stesso nome dei set.

```
87 // Add a new source set, dependency configurations and tasks to the build for IT an E2E tests.
88 testSets {
89     integrationTest { dirName = 'it' }
90     endToEndTest { dirName = 'e2e' }
91 }
```

Figura 3: Test Sets

Gradle, come accennato precedentemente, permette di definire dei task personalizzati che possono essere inseriti all'interno del task graph e quindi eseguiti ogni volta che viene eseguita la build del progetto. In questo caso possiamo immaginarli, con un certo livello di astrazione, come un goal di *Maven* che viene attaccato a una fase oppure eseguirli in maniera autonoma. Uno degli aspetti fondamentali nella definizione di nuovi task è la loro indipendenza rispetto agli altri, ma è comunque possibile definire un ordinamento fra essi, utilizzando i comandi *dependsOn* e *finalizedBy*.

In particolare per gestire i container di Docker per l'utilizzo del database MongoDB sono stati creati alcuni task personalizzati sfruttando il plugin "gradle-docker-plugin", mostrati in 4 e 5. Tra questi abbiamo *pullImage*() per il recupero dell'immagine di MongoDB da Docker Hub, *createContainer*() per la creazione del container, *startContainer*() per far partire il container, *startAndWaitOnHealthyContainer*() per attendere che il container sia pronto per essere utilizzato correttamente (*healthy*), *getContainerPortBindings*() per ottenere la porta esposta sull'host scelta casualmente in fase di creazione del container e per renderla disponibile come *system property* durante l'esecuzione dei test, e infine *stopContainer*() per fermare l'esecuzione del container.

Dopodiché dato che sia negli integration che negli end to end tests è stato scelto l'utilizzo di un database MongoDB reale, i task relativi alla loro esecuzione, rispettivamente "*integrationTest*" e "*endToEndTest*" sono stati configurati per supportare l'utilizzo dei container di Docker per l'uso del database. Per ottenere questo i due task sono stati configurati per dipendere dal task *getContainerPortBindings*() e per essere finalizzati dal task *stopContainer*(), come mostrato in 6. Inoltre i due task sono stati associati al task *check* di *Gradle*, utilizzato per raccogliere i task che eseguono verifiche sul codice e che dunque comprende anche il task per gli unit test (*test*), ma gli integration test sono stati configurati per essere lanciati solo dopo che gli unit test sono stati completati e gli end to end test sono stati configurati per essere lanciati solo dopo la conclusione gli integration test.

```
101 // Pull image from Docker.
102 task pullImage(type: DockerPullImage) {
103     image = 'mongo:4.4.5'
104 }
105
106 // Create container for the image.
107 task createContainer(type: DockerCreateContainer) {
108     dependsOn pullImage
109     targetImageId(pullImage.image)
110     hostConfig.portBindings = [":27017"]
111     hostConfig.autoRemove = true
112 }
113
114 // Start the container.
115 task startContainer(type: DockerStartContainer) {
116     dependsOn createContainer
117     targetContainerId(createContainer.containerId)
118 }
119
120 // Wait an healthy container.
121 task startAndWaitOnHealthyContainer(type: DockerWaitHealthyContainer) {
122     dependsOn startContainer
123     awaitStatusTimeout = 60
124     targetContainerId(createContainer.containerId)
125 }
```

Figura 4: Tasks per la gestione dei container di Docker

2.2.2 Come Eseguire una Build del Progetto

Un altro degli aspetti fondamentali di Gradle è il massiccio utilizzo del *Gradle Wrapper*. Il wrapper consente di poter effettuare build con *Gradle* senza averlo installato sulla propria macchina. Lavorando in team di sviluppo il vantaggio è ancora maggiore in quanto assicura che tutti i membri effettuino le build utilizzando la stessa versione di *Gradle*, senza bisogno di alcuna gestione aggiuntiva.

Una build completa del progetto può essere eseguita utilizzando il seguente comando nella cartella principale del progetto:

```
./gradlew build jacocoTestReport pitest
```

oppure semplicemente:

```
./gradlew build
```

dove:

- *./gradlew*: invoca il *Gradle Wrapper* per inizializzare la versione corretta di *Gradle* con cui eseguire la build del progetto.
- *build*: configura la build e calcola il task graph, compila i file sorgente andando a scaricare le dipendenze, esegue i test (unit, integration, end-to-end) e produce il fatjar dell'applicazione attraverso il plugin *shadow*.

```
// Inspect the container to get the host port.
task getContainerPortBindings(type: DockerInspectContainer) {
    dependsOn startAndWaitOnHealthyContainer
    targetContainerId(createContainer.containerId)
    onNext {
        it.networkSettings.getPorts().getBindings().each { k, v ->
            def containerPort = "" + k.getPort()
            def hostPort = "" + v[0].hostPortSpec
            println "Container Port ${containerPort} is exposed on host as port ${hostPort}"
            tasks.integrationTest {
                systemProperty 'mongo.port', hostPort
            }
            tasks.endToEndTest {
                systemProperty 'mongo.port', hostPort
            }
        }
    }
}

// Stop the container.
task stopContainer(type: DockerStopContainer) {
    targetContainerId(createContainer.containerId)
}
```

Figura 5: Tasks per la gestione dei container di Docker

```
// Ensure that integration tests are run before the check task but after unit test and E2E test after IT.
check.dependsOn integrationTest
check.dependsOn endToEndTest
integrationTest.mustRunAfter test
endToEndTest.mustRunAfter integrationTest

// Ensure that integration tests are run after getting container port bindings and are ended with a container stop.
integrationTest {
    dependsOn getContainerPortBindings
    finalizedBy stopContainer
}

// Ensure that E2E tests are run after getting container port bindings and are ended with a container stop.
endToEndTest {
    dependsOn getContainerPortBindings
    finalizedBy stopContainer
}
```

Figura 6: Configurazione dei task integrationTest e endToEndTest

- *jacocoTestReport*: esegue il task del plugin di JaCoCo per calcolare la code coverage.
- *pitest*: esegue il task del plugin di pitest per eseguire *mutation testing*.

In alternativa è possibile inserire il comando *clean* prima di *build* per eseguire il task che cancella la cartella di build del progetto per poi eseguire una build completa senza che *Gradle* riutilizzi qualche compilazione effettuata nelle build precedenti. Oppure semplicemente il comando *build* se non si vuole eseguire gli altri due task.

2.3 Continuous Integration

Per il progetto è stato configurato come sistema di *Continuous Integration* **GitHub Actions**, in modo tale da poter eseguire le build complete in remoto e poter continuare a lavorare localmente sul software. Dato che il progetto

è stato sviluppato principalmente su *Windows* il workflow principale (*gradle-windows.yml*) che esegue le build sia per richieste push che pull request è stato configurato sul sistema operativo *windows-2019*. In questo workflow prima di tutto viene configurato Java 11 per poter poi utilizzare correttamente il servizio di SonarCloud, poi vengono salvate le cache di *Gradle* e di *SonarQube*, viene configurato *Docker* su *Windows* e in seguito viene eseguita la build con *Gradle* con il seguente comando:

```
./gradlew build jacocoTestReport coverallsJacoco
          pitest sonarqube -i
```

dove rispetto a quanto visto prima sono stati aggiunti i seguenti comandi:

- *coverallsJacoco*: task che cerca il file .xml prodotto da JaCoCo e invia i risultati a coveralls.
- *sonarqube*: task che effettua l'analisi su SonarCloud.
- *-i*: argomento che permette di generare un log della build più verboso.

Infine il workflow carica con il meccanismo degli *artifacts* una serie di report dei vari test eseguiti solo però nel caso la build fallisca, mentre in caso di successo viene caricato il fatjar dell'applicazione.

Inoltre è stato aggiunto anche un ulteriore workflow che viene richiamato solo in caso di pull-request che esegue una build con *Gradle* su *Linux* con sistema operativo Ubuntu-20.04, per verificare il corretto funzionamento dell'applicazione anche su sistemi *Linux*. In quest'ultimo workflow vengono fatte le medesime configurazioni precedenti ma la build con *Gradle* viene lanciata solo con il semplice comando *build* per eseguire solo le varie compilazioni del codice, i test e il packaging dell'applicazione.

2.4 Coveralls

Coveralls riceve i dati della coverage direttamente da *GitHub Actions* tramite il task *coverallsJacoco*, i risultati della code coverage sono visibili al seguente link `https://coveralls.io/github/elia-mercatanti/guesthouse-reservations`. In particolare come mostrato in 7 possiamo vedere che nella configurazione di JaCoCo sono state escluse alcune classi non necessarie per la coverage, ovvero tutte le classi appartenenti ai pacchetti "model" e "app", ed inoltre sono stati abilitati i report in formato *xml* per renderli compatibili con *SonarCloud*.

2.5 SonarCloud

SonarCloud viene utilizzato per analizzare il codice per la *Code Quality*, i risultati delle analisi sono visibili al seguente link `https://sonarcloud.io/dashboard?id=elia-mercatanti_guesthouse-reservations`. In 8 invece viene mostrata la configurazione del task *sonarqube*, che consiste nei settaggi per l'esclusione delle classi non necessarie per la code coverage e per l'esclusione di

```
// Jacoco plugin configurations.
jacocoTestReport {
    // Generate html and xml Jacoco report.
    reports {
        xml.enabled = true
        html.enabled = true
    }

    // Exclude some classes from jacoco report.
    afterEvaluate {
        classDirectories.setFrom(files(classDirectories.files.collect {
            fileTree(dir: it, exclude: ['**/model/**', '**/app/**'])
        }))
    }
}
```

Figura 7: Configurazione del task jacocoTestReport

alcuni *issues* presenti nel codice ma che rappresentato in realtà dei falsi positivi, in particolare:

- java:S110: l'albero dell'ereditarietà delle classi non dovrebbe essere troppo profondo, nella classe *GuesthouseSwingView* perché estende *JFrame*.
- java:S2699: i test dovrebbero includere asserzioni, nella classe *GuesthouseSwingViewTest* perché *SonarCloud* non riconosce le asserzioni della libreria *Assertj Swing*.
- java:S3577: il nome delle classi dei test dovrebbero rispettare il regex "^((Test|IT)[a-zA-Z0-9]+|[A-Z][a-zA-Z0-9]*(Test|IT|TestCase|ITCase))$", nella classe *GuesthouseSwingAppE2E* perchè il suo nome finisce con "E2E".

Infine vengono configurati esplicitamente le inclusioni dei report e le cartelle dei sorgenti per tutti i tipi di test (unit, integration ed end to end tests).

```
// SonarQube plugin configurations.
sonarqube {
    properties {
        // Properties for project configuration.
        property 'sonar.projectKey', 'elia-mercatanti_guesthouse-reservations'
        property 'sonar.organization', 'elia-mercatanti'
        property 'sonar.host.url', 'https://sonarcloud.io'

        // Property for excluding some classes from code coverage report.
        property 'sonar.coverage.exclusions', ['**/model/**', '**/app/**']

        // Properties for ignoring some code issues.
        property 'sonar.issue.ignore.multicriteria', ['e11', 'e12', 'e13']
        property 'sonar.issue.ignore.multicriteria.e11.ruleKey', 'java:S110'
        property 'sonar.issue.ignore.multicriteria.e11.resourceKey', '**/GuesthouseSwingView.java'
        property 'sonar.issue.ignore.multicriteria.e12.ruleKey', 'java:S2699'
        property 'sonar.issue.ignore.multicriteria.e12.resourceKey', '**/GuesthouseSwingViewTest.java'
        property 'sonar.issue.ignore.multicriteria.e13.ruleKey', 'java:S3577'
        property 'sonar.issue.ignore.multicriteria.e13.resourceKey', '**/GuesthouseSwingAppE2E.java'

        // Properties for specifying the directories of junit reports for unit, integration and e2e tests.
        property 'sonar.junit.reportPaths', ['build/test-results/test', 'build/test-results/integrationTest', 'build/test-results/endToEndTest']
        properties['sonar.tests'] += 'src/it/java'
        properties['sonar.tests'] += 'src/e2e/java'
        properties['sonar.java.test.binaries'] += 'build/classes/java/integrationTest'
        properties['sonar.java.test.binaries'] += 'build/classes/java/endToEndTest'
    }
}
```

Figura 8: Configurazione del task sonarqube

2.6 Testing

Per Il progetto è stato utilizzato JUnit 5 come testing framework, sfruttando la possibilità di rinominare i test e di raggrupparli in classi di test che testano comportamenti simili dell'applicazione. L'unica eccezione riguarda la classe *GuesthouseSwingAppE2E* per gli end to end test che sono stati eseguiti attraverso *JUnit Vintage*, progetto che fornisce un'implementazione di un motore di test per l'esecuzione di test JUnit 4 sulla nuova piattaforma JUnit, questo per sfruttare la classe *AssertJSwingJUnitTestCase* per testare più facilmente l'applicazione ed il suo lancio.

2.7 Mutation Testing

Per il *mutation testing* è stato utilizzato *PIT*. Come mostrato in 9 per i mutatori sono stati attivati gli *strong mutator* e oltre alle stesse classi escluse per la coverage è stata esclusa anche la classe che implementa la GUI per la sua complessità che non si sposa bene con la gestione dei mutatori. Inoltre è stata impostata una soglia di successo quando almeno l'80% dei mutanti vengono uccisi, anche in questo caso per evitare eccessive restrizioni sopratutto per quanto riguarda le build eseguite sul CI.

```
175 // PIT mutation testing plugin configurations.
176 pitest {
177     junit5PluginVersion = '0.12'
178     mutators = ['STRONGER']
179     mutationThreshold = 80
180     excludedClasses = ['**.app.**', '**.model.**', '**.swing.**']
181     excludedTestClasses = ['**.swing.**']
182 }
```

Figura 9: Configurazione del task pitest

3 Difficoltà Durante lo Sviluppo

3.1 Gradle

Passare dalla build-automation con Maven a quella di Gradle non è stato semplicissimo, principalmente per l'enorme differenza con cui viene gestito il processo di build. Tuttavia, grazie a caratteristiche come un DSL per la specifica della build, *Gradle* si presenta piuttosto intuitivo ed immediato una volta approfondito in contrapposizione a *Maven* e l'xml. Uno degli aspetti più ostici è stato entrare nella mentalità di Gradle che impone una forte modularità nella definizione dei vari task personalizzati. È stato necessario studiare non solo in che modo i task vengono eseguiti ma anche come Gradle stesso gestisce le risorse e le varie parti di un progetto. Un altro aspetto che ha portato alcune problematiche nell'utilizzo di *Gradle* è stata indubbiamente la sua "novità" rispetto a *Maven*.

Per quest'ultimo infatti sono presenti svariati tutorial e plugin ad-hoc per arricchire la build, mentre per *Gradle*, essendo questo più giovane, risulta un pò più difficile trovare materiale online, un esempio calzante è stata la ricerca dei vari plugin alternativi a Maven e la loro configurazione, sopratutto quelli relativi a Docker, che ha impegnato un pò di tempo dello sviluppo e della preparazione del progetto.

3.2 GUI in Swing

Un altra parte che è risultata un pò ostica è stato gestire i test sulla GUI dell'applicazione, che inizialmente davano alcuni problemi sul sistema operativo *Linux*. Per una lunga parte dello sviluppo del progetto i test sulla GUI infatti fallivano a volte in modo apparentemente casuale solo sul CI di *GitHub Actions* con *Linux*, ed è stato molto difficile cercare di riprodurre il problema dato che in locale i test passavano sempre. Dopo svariate prove, effettuate anche su distro diverse mediante il comando xvfb-run, necessario per eseguire dei test sulla GUI solo da riga di comando, è stato fortunatamente possibile scovare il problema che era legato ad un scorretta gestione del focus dell'applicazione quando veniva lanciata nei test e che spesso non consentiva alla libreria AssertJ Swing di eseguire dei click corretti in determinate situazioni come ad esempio il cambio delle tab, usate in modo massiccio nei test della GUI di questo progetto.