

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

STRATEGIE DI TESTING PER WEB API

2022/2023

<i>Professore</i>	<i>Studente</i>
Porfirio Tramontana	Vincenzo Tramo

Indice

Introduzione	3
1 Introduzione e obiettivi	3
Software Under Testing	5
2 Restful Booker Platform	5
2.1 Use Case Diagram	5
2.2 Comprendere il sistema per definire una strategia di testing	6
2.2.1 Usare i Browser Tools per scoprire le Web API	6
2.2.2 Esplorare le Web API e memorizzare le richieste usando Postman	8
2.2.3 Scoprire i servizi e le dipendenze tra i servizi con Wireshark	9
2.2.4 Analizzare il codice sorgente del progetto	10
Strategie di Testing Automatico per Web API	12
3 Testing Automatico	12
3.1 Test Automation Framework	12
3.1.1 Test Automation Libraries	12
3.2 Integration Testing	13
3.2.1 Testare un microservizio insieme al suo database - Auth Service	13
3.3 Component Testing	23
3.3.1 Testare un microservizio in isolamento - Room Service	24
3.4 End-to-end Testing	34
3.4.1 Testing E2E - Making a booking for a room	34
A Compilazione ed Esecuzione Locale del Progetto	43

Introduzione

1 Introduzione e obiettivi

Nel contesto della progettazione e dell’interconnessione di sistemi software, le Web API sono diventate il fondamento di numerosi servizi online e applicazioni. La loro essenziale funzione di consentire la comunicazione e lo scambio di dati tra diverse applicazioni li rende componenti chiave di sistemi complessi. Un elemento cruciale nell’assicurare che tali Web API funzionino correttamente è il **testing**.

In tale contesto, questo elaborato si concentra sull’implementazione e l’analisi di diverse strategie di testing per Web API applicate su un progetto specifico: una piattaforma di servizi web, di nome Restful Booker Platform, che costituisce un sistema di prenotazione per Bed & Breakfast.

Il progetto in esame (che assume il ruolo di *Software Under Testing* e presentato brevemente in Sezione 2), sebbene non sia stato sviluppato personalmente dal sottoscritto, rappresenta un caso di studio significativo. L’obiettivo è quello di valutare l’affidabilità delle Web API all’interno della piattaforma e scoprire potenziali problematiche (o *failures*) attraverso l’applicazione di varie strategie di testing.

Data l’architettura a microservizi di Restful Booker Platform, le strategie di testing adottate sono specifiche per questo tipo di architettura. Poiché i microservizi operano come componenti indipendenti, il testing deve essere progettato per coprire sia gli aspetti di unità, sia quelli di integrazione tra i vari servizi. Inoltre, il testing end-to-end è fondamentale per garantire che l’interazione tra i microservizi funzioni correttamente nell’ambito dell’applicazione complessiva.

Le strategie di testing discusse in questo elaborato copriranno tre livelli della Test Pyramid in Figura 1:

- **Component Testing:** un’applicazione software è composta da vari moduli o componenti. I test specifici per valutare singoli componenti si chiamano test dei componenti. Questi test verificano le funzionalità e l’usabilità di ciascun componente in modo isolato, senza ancora integrarli con gli altri.
- **Integration Testing:** i test di integrazione sono un tipo di test del software in cui le diverse unità, moduli o componenti di un’applicazione software vengono testati come un’entità combinata. Lo scopo del test di integrazione è testare le interfacce tra i componenti ed esporre eventuali difetti che potrebbero verificarsi quando questi componenti sono integrati e devono interagire tra loro;
- **End-to-end Testing:** i test end-to-end (E2E) sono test che valutano l’intero flusso dell’applicazione, dall’inizio alla fine. Garantisce che tutti i componenti funzionino come previsto e che l’applicazione software funzioni correttamente in scenari simili a quelli di produzione. Nei test E2E, il software viene testato dal punto di vista dell’utente finale, simulando uno scenario utente reale, inclusa l’interfaccia utente, i servizi backend, i database e la comunicazione di rete.

Alla base della piramide, troviamo i test unitari. Sono più veloci e meno complessi, perché hanno il compito di testare solo classi e metodi isolati. In cima alla piramide, troviamo i test end-to-end. Sono costosi e inaffidabili perché hanno il compito di testare flussi che possono coinvolgere centinaia di servizi diversi e dipendenze esterne e presentano molti scenari da coprire. In questo elaborato non ci concentreremo sui test di unità ma piuttosto esamineremo, come già accennato, il Component Testing, Integration Testing ed E2E Testing.

Il progetto è disponibile al seguente link GitHub <https://github.com/vtramo/restful-booker-platform>.

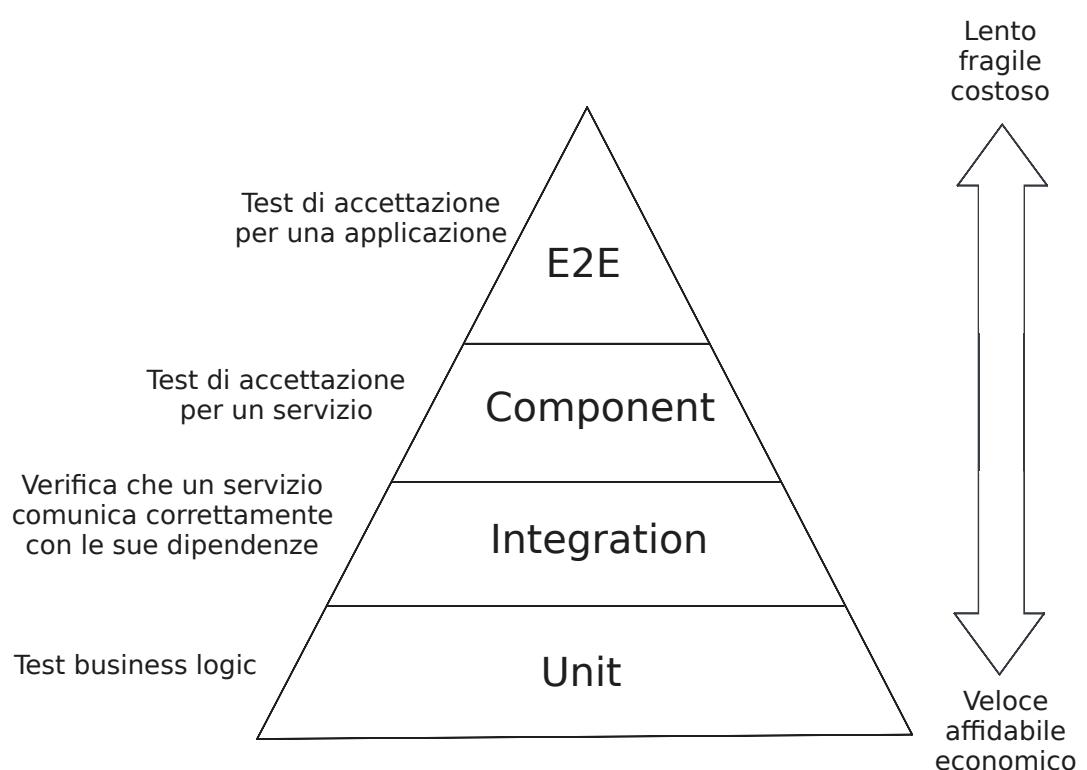


Figura 1: Test Pyramid

Software Under Testing

2 Restful Booker Platform

La piattaforma Restful Booker Platform è stata progettata per i proprietari di Bed and Breakfast (B&B) al fine di gestire i loro siti web e le prenotazioni. Il sistema offre le seguenti funzionalità:

- Creazione di un brand per commercializzare il B&B;
- Aggiungere stanze con dettagli per consentire agli ospiti di effettuare prenotazioni;
- Consentire agli ospiti di creare prenotazioni;
- Consentire agli ospiti di visualizzare dei report sulle prenotazioni per valutare la disponibilità;
- Consentire agli ospiti di inviare messaggi per contattare l'host del B&B.

La piattaforma fu inizialmente sviluppata come progetto personale per un singolo proprietario di B&B, ma col tempo è cresciuta ed è ora utilizzata da diversi proprietari di B&B per gestire le prenotazioni. Il progetto si sta espandendo lentamente sia in termini di funzionalità che di numero di clienti, ma recentemente ha riscontrato alcuni problemi durante la sua crescita. Alcuni proprietari di B&B hanno manifestato frustrazione a causa di errori, tempi di inattività e funzionalità implementate in modo non corretto. L'obiettivo è fornire una strategia di testing che aiuti il team a migliorare la qualità di Restful Booker Platform e assicurare la soddisfazione sia dei proprietari di B&B che degli ospiti nei confronti del prodotto.

2.1 Use Case Diagram

Lo Use Case Diagram in Figura 2 descrive in maniera più chiara le funzionalità del sistema:

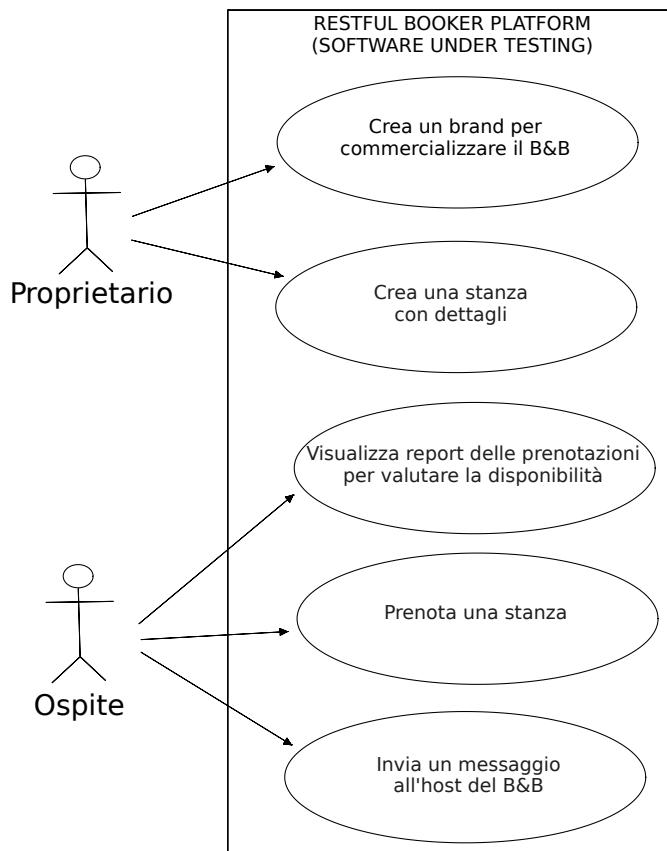


Figura 2: Restful Booker Platform - Use Case Diagram

L'applicazione è costruita per i B&B, il che implica la necessità di tener conto di due diverse categorie di utenti: gli ospiti e i proprietari (o hosts) dei B&B. Il sistema comprende una varietà di funzionalità, suggerendo che più servizi sono gestiti attraverso diverse Web API.

2.2 Comprendere il sistema per definire una strategia di testing

E' necessario avere una buona comprensione del progetto e del prodotto per ideare una valida strategia di testing. Ciò significa ricercare tra le varie fonti di informazione disponibili e utilizzare strumenti che aiutino a conoscere meglio il prodotto per il quale stiamo per costruire una strategia di testing. Per ampliare la nostra comprensione del contesto in cui agiscono le Web API della piattaforma che vogliamo testare possiamo sfruttare varie sorgenti di informazioni e differenti strumenti:

- **Browser Tools:** possiamo sfruttare gli strumenti built-in dei browser come Google Chrome o Firefox (conosciuti anche come *DevTools*). Questi strumenti sono dotati di funzioni come il monitoraggio del traffico HTTP, che ci permette di catturare le richieste che provengono dal browser e le risposte che vengono restituite. Questo traffico può essere analizzato e utilizzato per capire quali Web API vengono chiamate e quali dettagli vengono condivisi;
- **HTTP Testing Client:** possiamo usare le informazioni raccolte usando i DevTools messi a disposizione dai browser per ampliare la nostra comprensione del comportamento di ciascuna Web API utilizzando degli HTTP Client come Postman. Questi strumenti ci permettono di memorizzare le richieste HTTP e di modificarle per capire meglio le caratteristiche di una Web API;
- **Proxy Tools:** le Web API possono comunicare anche tra di loro. Strumenti di analisi di protocolli o packet sniffers come Wireshark ci aiutano ad ampliare la nostra ricerca per iniziare a capire quante Web API esistono in questa piattaforma e come dialogano tra loro;
- **Documentazione:** è piuttosto raro che un progetto non sia dotato di un minimo di documentazione. Una buona documentazione può sicuramente velocizzare il processo di comprensione del sistema e di conseguenza il testing;
- **Source code:** il miglior modo per capire come un componente software o come un intero sistema funziona è quello di consultare direttamente il codice sorgente;
- **Parlare con il team:** un altro modo per comprendere meglio il sistema è parlare direttamente con le persone che hanno costruito quel sistema.

2.2.1 Usare i Browser Tools per scoprire le Web API

La Figura 3 mostra l'interfaccia utente principale dell'applicazione. Per istruzioni dettagliate su come eseguire l'applicazione in locale, fare riferimento all'Appendice A.

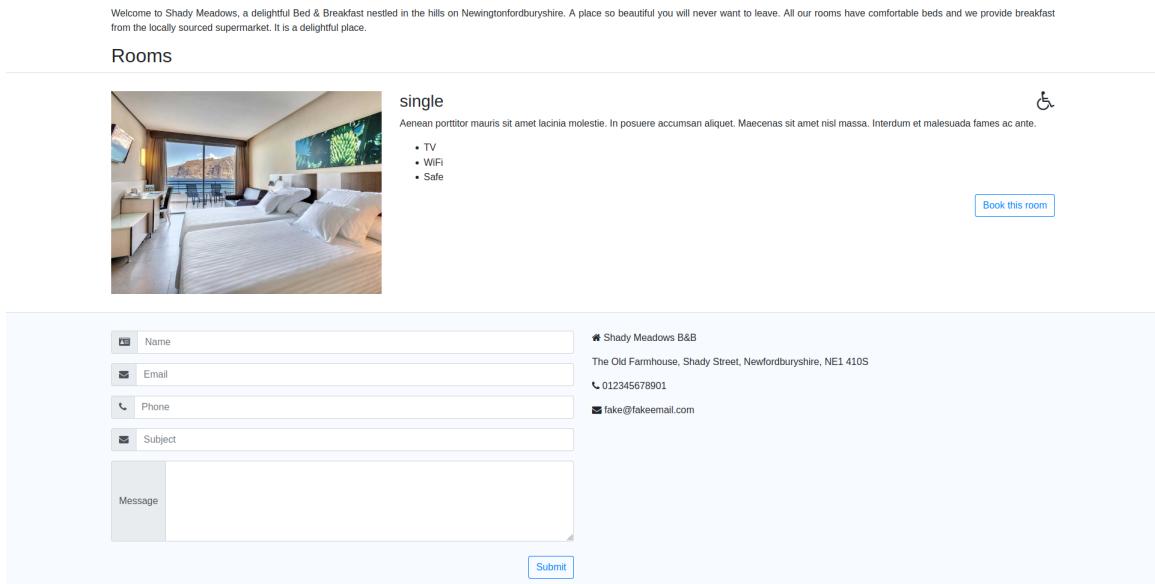


Figura 3: Schermata principale di Restful Booker Platform

Esplorando la piattaforma attraverso l'interfaccia utente, otteniamo un'idea generale del prodotto. Tuttavia, il nostro focus principale sono le Web API che operano dietro le quinte. Per approfondire

la nostra comprensione del sistema, facciamo uso degli strumenti integrati nei browser come Google Chrome e Firefox. Questi strumenti forniscono varie funzioni, tra cui il monitoraggio del traffico HTTP. Questa funzionalità ci permette di registrare le richieste inviate dal browser e le risposte ricevute dalle Web API. Il traffico registrato è poi disponibile per l'analisi, consentendoci di identificare le specifiche Web API coinvolte e i dettagli scambiati tra il browser e la piattaforma di servizi web.

Utilizzando il browser Mozilla Firefox, è possibile accedere allo strumento di monitoraggio del traffico HTTP premendo il tasto F12, quindi selezionando **Network** e, successivamente, **XHR** (che sta per **XMLHttpRequest**). Da questo momento in poi verranno registrate e visualizzate tutte le richieste HTTP eseguite dal browser.

Per illustrare questo concetto, supponiamo di tentare di prenotare una stanza cliccando sul pulsante “**Booking this room**” e seguendo tutti i passaggi necessari per finalizzare la prenotazione. In tal caso, il pannello visualizzerà diverse richieste HTTP, come evidenziato nella Figura 4:

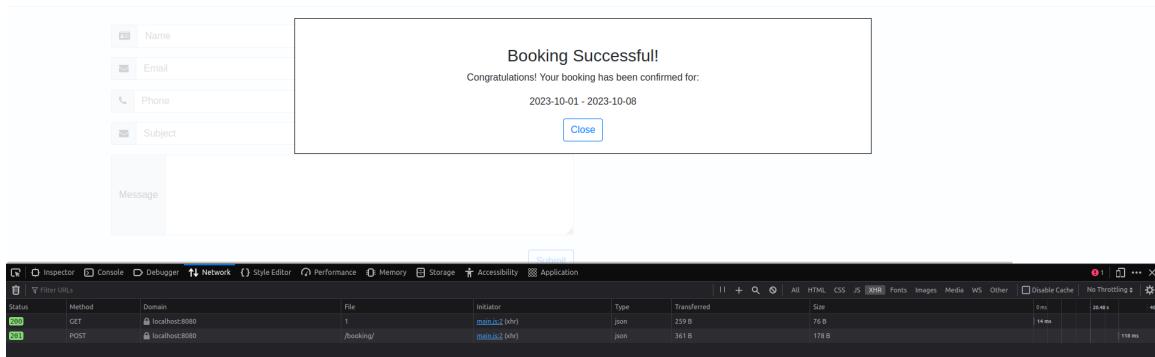


Figura 4: Richieste HTTP inviate dal browser

Come mostrato nella Figura 4, affinché una stanza possa essere prenotata, è necessario effettuare due richieste HTTP: una di tipo GET e una di tipo POST. Analizzando più approfonditamente le richieste HTTP registrate, possiamo constatare quanto segue:

- La prima richiesta HTTP è una GET con path identico a `/report/room/1`. La risposta, oltre a restituire uno status code 200, include un JSON:

```
{
  "report": [
    {
      "start": "2022-02-01",
      "end": "2022-02-05",
      "title": "Unavailable"
    }
  ]
}
```

E' evidente che tale endpoint restituisce un report per la stanza con id 1. In particolare, il report include i giorni in cui la stanza è **Unavailable** e cioè non disponibile per la prenotazione;

- La seconda richiesta HTTP è una POST con path identico a `/booking`. Anche in questo caso, il significato della richiesta è chiaro. Si tratta di una richiesta di tipo POST contenente un JSON che rappresenta i dettagli della prenotazione effettuata da un ospite o cliente. Il JSON include informazioni quali nome, cognome, indirizzo email, numero di telefono e ulteriori dettagli relativi alla prenotazione. La maggior parte di queste informazioni sono state inserite dal cliente tramite interfaccia grafica. Di seguito è riportato il payload JSON associato a questa richiesta:

```
{
  "bookingid": 2,
  "booking": {
    "bookingid": 2,
    "roomid": 1,
```

```

    "firstname": "Vincenzo",
    "lastname": "Tramo",
    "depositpaid": false,
    "bookingdates": {
        "checkin": "2023-10-01",
        "checkout": "2023-10-08"
    }
}
}

```

Questa analisi delle richieste HTTP effettuate durante il processo di prenotazione non solo ci permette di identificare le Web API coinvolte, ma ci fornisce anche informazioni dettagliate su quali dati vengono restituiti o inviati attraverso queste API. Utilizzando la stessa tecnica, è possibile esplorare ulteriormente la piattaforma, scoprendo nuovi endpoint e approfondendo la comprensione delle funzionalità offerte dal sistema.

2.2.2 Esplorare le Web API e memorizzare le richieste usando Postman

Dopo aver utilizzato gli strumenti di sviluppo dei browser per raccogliere e analizzare le richieste HTTP sottostanti, possiamo ora approfondire ulteriormente la nostra comprensione del comportamento di ciascuna Web API. Un utile strumento in questo processo è **Postman**. Postman è una potente piattaforma per la gestione delle API, ampiamente utilizzata dai professionisti dello sviluppo software e dei servizi web. Questo strumento offre un ambiente intuitivo che consente di testare, documentare e collaborare efficacemente con le API. Con Postman, è possibile creare, inviare e ricevere richieste HTTP (e non solo) personalizzate, esplorare le risposte e organizzare le richieste in modo ordinato all'interno di *Collections* per un facile accesso e condivisione.

Possiamo usare Postman per memorizzare agevolmente le richieste HTTP precedentemente scoperte e di apportare modifiche per sperimentare e comprendere appieno le caratteristiche delle Web API del sistema. In tal modo, possiamo esplorare in dettaglio gli endpoint identificati e valutare le risposte ricevute tenendo in mente che l'obiettivo in questo momento è comprendere il sistema e non tanto individuare errori.

In Figura 5 vengono mostrati tutti gli endpoint raccolti con i DevTools del browser memorizzati su Postman. La piattaforma di servizi web espone endpoint per gestire le stanze, le prenotazioni, i messaggi, i report, i brand ed endpoint relativi all'autenticazione.

Figura 5: Raccolta richieste HTTP con Postman

2.2.3 Scoprire i servizi e le dipendenze tra i servizi con Wireshark

L'indagine eseguita fino ad adesso si è concentrata sul flusso di dati tra il browser e le Web API. Tuttavia, non tutto il traffico della piattaforma è limitato a queste interazioni. È necessario ampliare la nostra ricerca per cominciare a comprendere quanti servizi (intesi come istanze indipendenti) sono presenti in questa piattaforma e come questi comunicano tra loro. Per fare ciò, possiamo utilizzare lo strumento **Wireshark**. Wireshark è uno strumento potente e ampiamente utilizzato per l'analisi del traffico di rete. È un packet sniffer che consente agli utenti di catturare e ispezionare pacchetti che transitano attraverso una rete, consentendo un'approfondita comprensione delle comunicazioni in corso.

Utilizzando Wireshark, siamo in grado di esplorare ulteriormente il panorama delle Web API all'interno di restful-booker-platform. Questo strumento ci permette di capire quanti servizi indipendenti sono presenti all'interno della piattaforma e di comprendere come essi comunicano tra di loro. Per eseguire questa attività è necessario avviare l'intera piattaforma sulla propria macchina localmente su cui Wireshark sta eseguendo. Quando avviamo Wireshark sull'interfaccia di rete loopback, il packet sniffer comincerà a monitorare qualsiasi attività che si verifica internamente sulla propria macchina.

Tutte le richieste HTTP inviate dal browser sono indirizzate sempre verso lo stesso host. Poiché la piattaforma è in esecuzione in locale, tutte le richieste HTTP sono indirizzate a `localhost:8080` (questa scoperta è emersa durante l'analisi eseguita con gli strumenti di sviluppo del browser). Se provassimo a inviare una richiesta `GET localhost:8080/report/room/1` (ottenere un report riguardo la disponibilità della stanza con id 1), questo è ciò che otterremmo sulla schermata di Wireshark (protocollo http):

No.	Time	Source	Source Port	Destination	Destination Port	Protocol	Length	Info
2629	32.696202478	::1	50104	::1	8080	HTTP	308	GET /report/room/1 HTTP/1.1
2636	32.698235986	127.0.0.1	39660	127.0.0.1	3905	HTTP	244	GET /report/room/1 HTTP/1.1
2646	32.730466662	127.0.0.1	54560	127.0.0.1	3900	HTTP	225	GET /booking/summary?roomid=1 HTTP/1.1
2654	32.771312329	127.0.0.1	3000	127.0.0.1	54560	HTTP/JSON	71	HTTP/1.1 200 , JavaScript Object Notation (application/json)
2659	32.779866289	127.0.0.1	3005	127.0.0.1	39660	HTTP/JSON	71	HTTP/1.1 200 , JavaScript Object Notation (application/json)
2663	32.780902756	::1	8080	::1	50104	HTTP/JSON	91	HTTP/1.1 200 , JavaScript Object Notation (application/json)

Figura 6: Cattura traffico HTTP richiesta `GET localhost:8080/report/room/1`

Analizzando le richieste HTTP catturate da Wireshark (mostrate in Figura 6) scopriamo quanto segue:

- Il client HTTP apre una connessione TCP (`Source Port 50104`) e invia la richiesta HTTP `GET /report/room/1` verso il servizio in ascolto sulla porta `Destination Port 8080`;
- Il servizio in ascolto sulla porta 8080 riceve la richiesta HTTP e la inoltra verso un altro servizio in ascolto sulla porta 3005;
- Il servizio in ascolto sulla porta 3005 riceve la richiesta HTTP, la elabora, genera una nuova richiesta HTTP `GET /booking/summary?roomid=1` verso un altro servizio web della piattaforma in ascolto sulla porta 3000 e completa l'elaborazione rispondendo al mandante con un `HTTP/1.1 200 OK`;
- Il client HTTP del punto 1. riceve una risposta `HTTP/1.1 200 OK` contenente la risorsa ricercata.

È facile intuire che il servizio scoperto in ascolto sulla porta 8080 è un **reverse proxy**. Analizzando altre richieste HTTP con Wireshark, scopriamo che tutto il traffico generato dal client HTTP è indirizzato verso un unico punto: il reverse proxy appena scoperto (`localhost:8080`). Tale reverse proxy ha la funzione di esporre un unico servizio verso l'esterno su cui i client HTTP possono inviare le richieste.

Dopo aver analizzato ogni richiesta (memorizzate precedentemente con Postman) con Wireshark, possiamo costruire una mappatura dei servizi web della piattaforma e le loro dipendenze come quella mostrata in Figura 7.

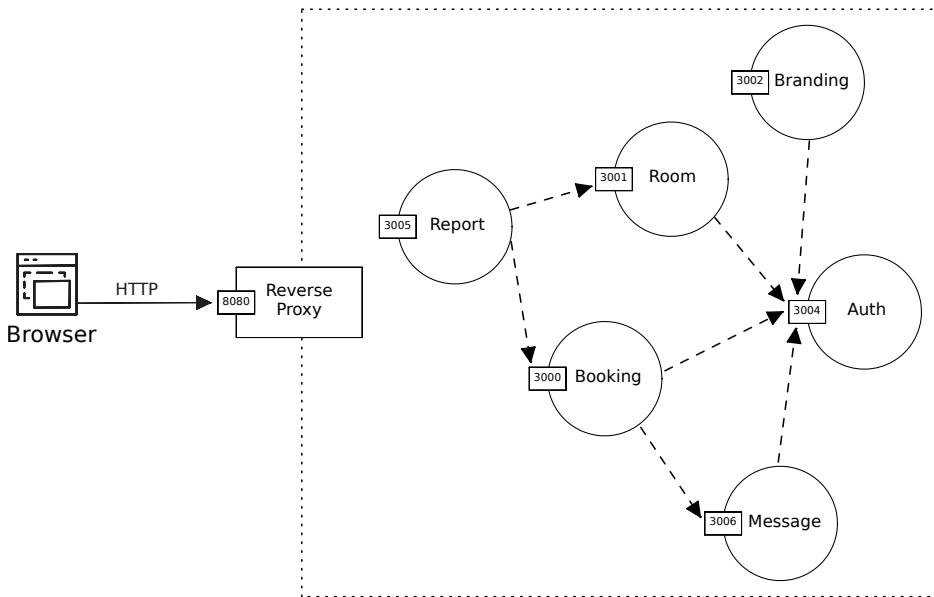


Figura 7: Mappatura dei servizi web di Restful Booker Platform e le loro interazioni

2.2.4 Analizzare il codice sorgente del progetto

Ogni servizio mostrato in Figura 7 è una istanza **Spring Boot 2.7.5** (un popolare framework per realizzare servizi web) implementata con **Java 15**. Queste informazioni, così come i nomi dei vari servizi, sono facilmente reperibili analizzando la struttura e il codice sorgente del progetto.

Di seguito vengono evidenziate a grandi linee le responsabilità di ogni servizio web:

- **Report:** questo servizio ha la responsabilità di realizzare report sulla disponibilità delle stanze. Espone due endpoint:
 1. GET `/report` per ottenere un report di ogni stanza
 2. GET `/report/room/id` per ottenere un report per una specifica stanza
- **Room:** questo servizio gestisce le stanze. Espone cinque endpoint:
 1. GET `/room` per ottenere tutte le stanze
 2. GET `/room/id` per ottenere una specifica stanza
 3. POST `/room` per creare una stanza
 4. DELETE `/room/id` per eliminare una specifica stanza
 5. PUT `/room/id` per aggiornare una stanza già esistente
- **Booking:** questo servizio gestisce le prenotazioni. Espone sei endpoint:
 1. GET `/booking` per ottenere tutte le prenotazioni
 2. GET `/booking/id` per ottenere una specifica prenotazione
 3. GET `/booking/summary?roomId=id` per ottenere la storia delle prenotazioni di una specifica stanza
 4. POST `/booking` per creare una prenotazione per una stanza
 5. PUT `/booking/id` per aggiornare una specifica prenotazione
 6. DELETE `/booking/id` per eliminare una specifica prenotazione
- **Branding:** questo servizio gestisce la brandizzazione. Espone due endpoint:
 1. GET `/branding` per ottenere tutti i branding

2. PUT /branding/id per aggiornare un branding

L'applicazione attualmente non supporta la creazione di nuovi branding

- **Message:** questo servizio è responsabile di gestire i messaggi intesi come notifiche che un'amministratore di un B&B dovrebbe ricevere. Espone sei endpoint:

1. GET /message per ottenere tutti i messaggi
2. GET /count per ottenere il numero di messaggi non letti
3. GET /message/id per ottenere uno specifico messaggio
4. POST /message per creare un messaggio
5. DELETE /message/id per eliminare un messaggio
6. PUT /message/id/read per modificare e marcare uno specifico messaggio come letto

- **Auth:** la piattaforma permette a un amministrazione di autenticarsi e gestire il proprio B&B. Il servizio Auth si occupa di questioni legate all'autenticazione/autorizzazione. Espone tre endpoint:

1. POST /auth/login per autenticarsi e ottenere un token di accesso da inserire in ogni richiesta che lo richiede
2. POST /auth/validate per verificare se un dato token è ancora valido o meno
3. POST /auth/logout per effettuare il logout

- **Reverse Proxy:** questo servizio è un reverse proxy e ha la responsabilità di smistare le richieste ai corretti servizi web della piattaforma. Riceve il traffico sulla porta 8080.

Strategie di Testing Automatico Avanzate per Web API

3 Testing Automatico

Molte persone concordano sul fatto che i test vengano creati per garantire che le cose operino come ci si aspetta. Tuttavia, la questione va oltre. Chiunque abbia esperienza con test manuali sa quanto possano essere lenti, noiosi e suscettibili di errori. Con i test automatizzati, miriamo a superare questi ostacoli, eliminandoli o almeno riducendone l'impatto. Il testing automatico rappresenta quindi un pilastro fondamentale per almeno tre motivi:

1. Ci dà la fiducia che il codice faccia ciò che dovrebbe;
2. Fornisce un feedback veloce, accurato, affidabile e prevedibile;
3. Facilitano la manutenzione, un aspetto che viene comunemente trascurato quando si scrivono i test.

In questo capitolo verranno mostrate diverse strategie di testing automatico avanzate per Web API applicate a Restful Booker Platform.

3.1 Test Automation Framework

Un approccio certo al fallimento nell'automazione consiste nel creare un framework di automazione complicato da gestire o da comprendere. Bisogna sempre tenere in mente che il testing automatico è, in fin dei conti, codice, e dovremmo applicare le medesime tecniche e approcci che adottiamo per mantenere il nostro codice di produzione pulito, leggibile e facilmente manutenibile. A questo scopo, organizziamo i nostri test di integrazione per Web API in tre package:

- **payloads**: molte Web API usano modelli complessi per le richieste e le risposte HTTP, il che si traduce in un gran numero di classi POJO (Plain Old Java Object). Lo scopo di questo package è quello di creare un luogo in cui organizzare in modo pulito queste classi, a seconda che siano utilizzate per i payload di richiesta o di risposta, nonché di riordinarle nelle proprie aree, a seconda dell'API che si sta testando;
- **requests**: questo package include le richieste HTTP relative alle Web API. Ciascuna richiesta è associata ai suoi metodi che verranno invocati all'interno dei test. Questo approccio consente di apportare modifiche all'URL dell'API in un unico punto, semplificando l'aggiornamento;
- **checks**: qui sono organizzati le test suites. L'obiettivo di questo framework è quello di rendere i controlli automatici il più possibile facili da leggere. È importante capire l'intento di una test suite. Tale obiettivo viene raggiunto delegando le azioni di creazione dei payload e di invio/ricezione delle richieste e delle risposte HTTP agli altri due packages del test framework tenendo invece le asserzioni in questo package.

La Figura 8 mostra i tre packages appena descritti e le dipendenze tra di loro.



Figura 8: Test Framework Packages

3.1.1 Test Automation Libraries

In questo progetto, verranno utilizzate le seguenti librerie di testing automatico e librerie di supporto:

- **JUnit 5.10.0**;
- **Hamcrest 2.2**;
- **Rest Assured 5.3.0**;
- **Jackson 2.15.2**;

- **Awaitility 4.2.0;**
- **Testcontainers 1.19.1;**
- **Wiremock 1.0;**
- **Selenium 4.14.1;**
- **Cucumber 7.14.0.**

3.2 Integration Testing

I requisiti del software sono senza dubbio l’artefatto più prezioso quando si parla di testare un software. I requisiti ci dicono esattamente cosa deve fare il software e cosa non deve fare. Descrivono le complessità delle regole di business che il software deve implementare e che bisogna convalidare. Per identificare i test da implementare ed eseguire, verranno adottate una serie di metodologie comprese nell’ambito del cosiddetto **testing basato sulle specifiche** (o *specification based-testing*). Queste tecniche utilizzano i requisiti del software, per esempio i casi d’uso UML (Figura 2), come input di test.

In questa sezione verranno mostrati una serie di esempi di **integration testing** avanzato. I test di integrazione mirano a testare più componenti di un sistema insieme, concentrandosi sulle interazioni tra di esse invece di testare il sistema nel suo complesso. I test di integrazione verificano che i componenti/microservizi indipendenti funzionino correttamente quando sono connessi. In particolare, i test di integrazione nell’ambito dei microservizi presentano le seguenti caratteristiche:

- I test di integrazione verificano la connessione tra moduli interni e componenti esterni, come un database o altri (micro)servizi e non altri moduli interni.
- I test di integrazione utilizzano componenti esterni reali per verificare se è possibile la comunicazione con un servizio reale o con un database reale.
- È probabile che la preparazione dell’ambiente per l’esecuzione dei test sia difficile e/o noiosa.

3.2.1 Testare un microservizio insieme al suo database - Auth Service

In questo paragrafo, ci concentreremo nel verificare il corretto funzionamento del servizio di autenticazione (**Auth**) in concomitanza con il suo database PostgreSQL associato. Questo approccio di testare congiuntamente un servizio e il suo database riveste un ruolo fondamentale, poiché contribuisce ad accrescere la nostra fiducia non solo nella corretta operatività del servizio di autenticazione, ma anche nell’interazione tra il servizio **Auth** e il database. La Figura 9 mostra le componenti del sistema che vogliamo integrare e testare.

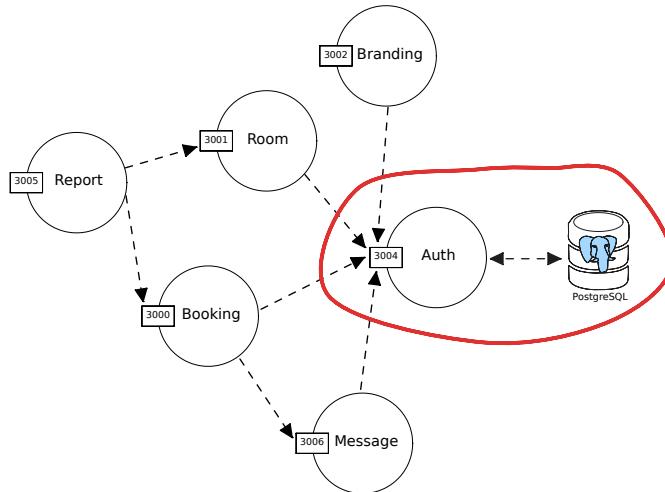


Figura 9: Auth Service - Integration Testing

Il servizio **Auth** è una istanza di Spring Boot 2.7.5 che comunica con un database PostgreSQL. Vogliamo testare il servizio insieme ad una reale istanza di database PostgreSQL. Tuttavia, per fare ciò, abbiamo bisogno di eseguire i seguenti passi:

1. Eseguire una reale istanza del servizio **Auth** in locale in ascolto su una determinata porta;
2. Eseguire una istanza di database PostgreSQL compatibile con il servizio **Auth** e connetterla ad esso;
3. Definire lo schema del database e popolarlo con dati di test;
4. Eseguire i test (eventualmente aggiungendo o eliminando nuovi dati richiesti prima di avviare ogni caso di test).

I primi tre passi appena elencati hanno l'obiettivo di *preparare un'ambiente ideale* su cui eseguire i nostri test. Un'ambiente di testing ideale è un'ambiente che ha le stesse caratteristiche (o quasi) di un'ambiente reale di produzione. È chiaro che eseguire questi passi senza l'aiuto di framework/librerie di testing richiederebbe uno sforzo non banale. Fortunatamente, queste situazioni sono abbastanza comuni, e possiamo semplificare l'implementazione di tali operazioni sfruttando il framework Spring Boot e le librerie di testing **JUnit 5** e **Testcontainers**.

Per implementare il primo passo, basta annotare la classe di test con questa annotazione:

```
@SpringBootTest(
    webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT,
    classes = AuthApplication.class
)
```

Tale annotazione assicura che il servizio **Auth** si avvii prima di ogni altro test su una porta definita nel file di configurazione (`application-test.properties`).

Il secondo passo, cioè quello di eseguire una istanza di PostgreSQL e collegarla al servizio prima dell'esecuzione dei test, richiede uno sforzo in più. Usando la libreria Testcontainers, possiamo avviare una istanza Docker di PostgreSQL programmaticamente. Per fare ciò, bisogna innanzitutto annotare la classe di test con `@Testcontainers`. Questa annotazione integra la libreria JUnit 5 con quella di Testcontainers. Il seguente codice avvia una istanza di PostgreSQL in un Docker container con utente e password `postgres`, nome del database uguale a `test` e definisce lo schema del database eseguendo uno *init script* SQL di nome `init-db.sql`. Lo script crea le tabelle per il servizio **Auth**. Inoltre, espone il container sulla porta 5432 dell'host.

```
@ExtendWith(SpringExtension.class) // Integra Spring TestContext Framework in JUnit 5
@SpringBootTest( // Avvia AuthApplication prima dei test
    webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT,
    classes = AuthApplication.class
)
@ActiveProfiles("test") // Inietta configurazioni presenti in application-test.properties
@Testcontainers // Integra JUnit 5 con Testcontainers
@DisplayName("Auth API")
public class AuthIT {

    @Container
    static PostgreSQLContainer<?> postgres =
        new PostgreSQLContainer<>("postgres:15.2-alpine")
            .withUsername("postgres")
            .withPassword("postgres")
            .withInitScript("init-db.sql")
            .withDatabaseName("test")
            .withExposedPorts(5432);

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        registry.add("auth.db.jdbc-url", postgres::getJdbcUrl);
        registry.add("auth.db.jdbc-username", postgres::getUsername);
        registry.add("auth.db.jdbc-password", postgres::getPassword);
    }

    ...
}
```

Il metodo `configureProperties(...)` annotato `@DynamicPropertySource` setta dinamicamente (cioè a *runtime*) le proprietà di connessione al database, come il Jdbc URL, username e password. È necessario farlo dinamicamente perché il container deve prima avviarsi affinché queste informazioni

possano essere rese disponibili. Dunque, tale metodo connette effettivamente il servizio Auth al database containerizzato. Il contenuto di `init-script.sql` è il seguente e deve essere presente in `src/test/resources` (nel modulo auth):

```
CREATE TABLE IF NOT EXISTS TOKENS (
    tokenid SERIAL PRIMARY KEY,
    token VARCHAR(255),
    expiry TIMESTAMP
);

CREATE TABLE IF NOT EXISTS ACCOUNTS (
    accountid SERIAL PRIMARY KEY,
    username VARCHAR(255),
    password VARCHAR(255)
);
```

La tabella `TOKENS` rappresenta i tokens generati dal servizio e resi persistenti. Un token, oltre ad avere un id e un valore, ha una data di scadenza dopo il quale non è più valido. La tabella `ACCOUNTS` rappresenta i proprietari di un B&B registrati sulla piattaforma.

Usando la libreria Testcontainers, siamo capaci di:

- Usare la stessa versione del database usato in produzione. Possiamo usare la stessa identica immagine del database PostgreSQL usata in produzione;
- Aumentare l'affidabilità del test di integrazione perché sia l'applicazione che i test usano lo stesso tipo di database e la stessa versione evitando così problemi di compatibilità.

Prima di procedere con i casi di test automatici veri e propri, organizziamo le varie richieste da eseguire e i corrispondenti payloads necessari per tali richieste e quelli necessari per le risposte nei package `requests` e `payloads` (Figura 8). Iniziamo col definire i payloads. Definiamo le classi POJO `Auth` e `Token`:

```
package com.automationintesting.it.payloads;

import com.fasterxml.jackson.annotation.JsonProperty;

public class Auth {
    @JsonProperty private String username;
    @JsonProperty private String password;

    public Auth(String username, String password) {
        this.username = username;
        this.password = password;
    }

    // Getters and Setters
    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }
    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }
}

/* ----- */

package com.automationintesting.it.payloads;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonProperty;

import java.time.ZonedDateTime;

public class Token {
    public static final String TOKEN_NAME = "token";

    @JsonProperty private String token;
    @JsonIgnore private ZonedDateTime expiry;
```

```

public Token(String token) { this.token = token; }
public Token() {}

// Getters and Setters
public String getToken() { return token; }
public void setToken(String token) { this.token = token; }
public ZonedDateTime getExpiry() { return expiry; }
public void setExpiry(ZonedDateTime expiry) { this.expiry = expiry; }
}

```

Le classi Auth e Token verranno serializzate (richiesta) e deserializzate (risposta) usando la nota libreria Jackson JSON Parser. Infatti, le proprietà da serializzare e deserializzare sono annotate con `@JsonProperty`.

Successivamente, definiamo le richieste da inviare al servizio Auth. Prima di tutto, definiamo una classe astratta `BaseApi` che ha la responsabilità di mantenere in un unico posto l'URL a cui inviare le richieste:

```

package com.automationintesting.it.requests;

public abstract class BaseApi {
    private final String url;

    public BaseApi(String url) {
        this.url = url.endsWith("/") ? url : url + "/";
    }

    public String url() {
        return url;
    }
}

```

A questo punto definiamo una classe `AuthApi` che estende `BaseApi` e fornisce metodi per eseguire richieste HTTP verso il servizio Auth. Usando la libreria REST Assured, possiamo creare queste richieste in maniera molto pulita, come mostrato di seguito:

```

package com.automationintesting.it.requests;

import com.automationintesting.it.payloads.Auth;
import com.automationintesting.it.payloads.Token;
import io.restassured.http.ContentType;
import io.restassured.response.Response;

import static io.restassured.RestAssured.given;

public class AuthApi extends BaseApi {
    public static final String AUTH_SERVICE_URL_ENV_VAR_NAME = "AUTH_SERVICE_URL";

    public AuthApi(String url) {
        super(url);
    }

    public Response createToken(Auth authentication) {
        return given()
            .contentType(ContentType.JSON)
            .body(authentication)
            .post(url() + "auth/login");
    }

    public Response validateToken(Token token) {
        return given()
            .contentType(ContentType.JSON)
            .body(token)
            .post(url() + "auth/validate");
    }

    public Response clearToken(Token token) {
        return given()

```

```

        .contentType(MediaType.APPLICATION_JSON)
        .body(token)
        .post(url() + "auth/logout");
    }
}

```

La classe `AuthApi` è talmente autoesplicativa che non ha bisogno di particolari spiegazioni. La variabile statica pubblica `AUTH_SERVICE_URL_ENV_VAR_NAME` rappresenta il nome della variabile d'ambiente da usare per l'URL di `AuthApi` nel caso in cui lo si volesse settare tramite una variabile d'ambiente (molto comune). Ogni metodo rappresenta un endpoint esposto dal servizio `Auth`, in particolare:

- `public Response createToken(Auth authentication)`: mappa l'endpoint POST `/auth/login` ed è necessario un body di tipo `Auth`;
- `public Response validateToken(Token token)`: mappa l'endpoint POST `/auth/validate`. È necessario un body di tipo `Token`;
- `public Response clearToken(Token token)`: mappa l'endpoint POST `/auth/logout`. È necessario un body di tipo `Token`.

Aggiungiamo, dunque, nella nostra test suite una variabile d'istanza (*test fixture*) di tipo `AuthApi`. Inoltre, definiamo un metodo `setupAuthApi()` per configurare la fixture prima dell'esecuzione di ogni test. Questo può essere fatto con l'annotazione `@BeforeEach` come mostrato di seguito:

```

...
// Test Fixture
AuthApi authApi;

// Prima di ogni esecuzione, configura AuthApi
@BeforeEach
void setupAuthApi() {
    RestAssured.defaultParser = JSON;
    String authUrl = Optional
        .ofNullable(System.getenv(AuthApi.AUTH_SERVICE_URL_ENV_VAR_NAME))
        .orElse("http://localhost:3004");
    authApi = new AuthApi(authUrl);
}

...

```

In particolare, nel metodo `setupAuthApi()` diciamo alla libreria REST Assured di usare come default parser il parser JSON (Jackson). Successivamente, preleviamo l'URL del servizio `AuthApi` dalla variabile d'ambiente apposita. Se non è stato fornito un valore, allora viene usato `http://localhost:3004` come URL di default.

Arrivati a questo punto, siamo pronti per definire una serie di test case per mettere alla prova il servizio `Auth`. Ogni suo endpoint ha una responsabilità ben precisa, un requisito ben preciso da soddisfare. Come già accennato, sfrutteremo questi requisiti per definire ed eseguire i nostri casi di test.

La specifica dell'endpoint POST `/auth/login` sono definite nella Tabella 1.

Specifiche endpoint POST <code>/auth/login</code>		
Goal	Input	Output
Genera un token di autenticazione per l'utente che ha effettuato l'accesso. Affinché il token possa essere generato, l'utente deve fornire le proprie credenziali per l'autenticazione. Il token restituito è valido per un periodo di tempo specificato dalla proprietà <code>app.token-life-duration</code> . Una volta trascorso tale intervallo, il token non sarà più considerato valido.	HTTP Body application/json { "username": "string", "password": "string" }	- HTTP 200 OK con HTTP Header <code>Set-Cookie</code> impostato, ad esempio: <code>Set-Cookie=token=e14iFOUcZtFy45Rs; Path=/</code> se le credenziali sono corrette. - HTTP 403 Forbidden se le credenziali fornite non sono corrette.

Tabella 1: Specifiche endpoint POST `/auth/login`

Basandoci su questa specifica, possiamo estrarre diversi casi di test. Cominciamo con la verifica della risposta `403 Forbidden` che il servizio `Auth` dovrebbe restituire in caso di forniture di

credenziali non valide. Quando il servizio Auth viene avviato, nel database (PostgreSQL) ad esso associato non esistono account. Questo implica che qualsiasi combinazione di username e password dovrebbe restituire un 403 Forbidden. I seguenti casi di test verificano esattamente questo:

```

...
@Nested
@DisplayName("When trying to create a token using incorrect credentials")
class WhenCreateTokenIncorrectCredentials {

    @ParameterizedTest
    @MethodSource("provideIncorrectCredentials")
    @DisplayName("Should deny the creation of a token to a non-existing user")
    void shouldDenyCreateTokenNonExistingUser(String username, String password) {
        Auth authentication = new Auth(username, password); // Arrange

        Response response = authApi.createToken(authentication); // Act

        assertThat(response.statusCode(), is(equalTo(HttpStatus.SC_FORBIDDEN))); // Assert
    }

    private static Stream<Arguments> provideIncorrectCredentials() {
        return Stream.of(
            Arguments.of(" ", " "),
            Arguments.of(null, null),
            Arguments.of("", ""),
            Arguments.of("\t", "\t"),
            Arguments.of("\n", "\n"),
            Arguments.of("ppippopozzo354L", "18sujudeDDF1L"),
            Arguments.of("admin", "password"),
            Arguments.of("admin", ""),
            Arguments.of("root", ""),
            Arguments.of("root", "root")
        );
    }
}
...

```

Per una migliore leggibilità e chiarezza concettuale del codice di test, è una buona pratica organizzare i casi di test in classi innestate sfruttando l'annotazione `@Nested`. Inoltre, per evidenziare ancora di più l'intento del codice di test, possiamo sfruttare l'annotazione `@DisplayName` e il pattern “*Given-When-Then*”. Nel caso di test in questione abbiamo usato l'annotazione `@ParameterizedTest` per parametrizzare il test e fornire valori attraverso un metodo puntato dalla annotazione `@MethodSource`. Il metodo `provideIncorrectCredentials()` fornisce coppie di username e password sottoforma di `Arguments`. Ogni argomento viene fornito attraverso uno `Stream<Arguments>` (Java 8 Stream) e definisce un test case separato. L'annotazione `@ParameterizedTest` è un ottimo modo per riutilizzare lo stesso metodo di test per diversi valori di input (in questo caso username e password). Si noti che sono stati coperti anche casi corner come stringhe vuote, null pointer e caratteri speciali. Il corpo del metodo di test è chiaramente suddiviso in tre parti:

- **Arrange (Preparazione)**: in questa fase, si preparano le condizioni iniziali necessarie per il test. Si configurano oggetti, variabili e stato iniziale. In particolare, qui istanziamo l'oggetto `Auth` fornendo `username` e `password` al costruttore;
- **Act (Esecuzione)**: questa è la fase in cui viene eseguita l'azione o il comportamento che si sta cercando di testare. L'azione in questo caso è inviare una richiesta HTTP usando `authApi` e il metodo `createToken(authentication)` fornendo l'oggetto `Auth` istanziato nella fase di `Arrange`;
- **Assert (Verifica)**: qui si effettuano asserzioni per verificare se il risultato dell'azione corrisponde alle aspettative. In particolare, il tipo `Response` restituito dal metodo `createToken(...)` è una interfaccia della libreria di REST Assured che rappresenta una generica risposta HTTP. Tale interfaccia fornisce differenti metodi utili per ispezionare la risposta. Uno di questi è ottenere lo status code (usando il metodo `getStatusCode()`) che deve corrispondere a 403 `Forbidden`.

Successivamente, possiamo ampliare la suite di test per coprire situazioni in cui vengono fornite credenziali corrette e verificare che il servizio generi un token di autenticazione valido. Tuttavia, questo richiede prima di inserire un record nella tabella ACCOUNTS nel database PostgreSQL. Per fare ciò, è necessario:

1. Aprire una connessione al db;
2. Scrivere il record (un account nella tabella ACCOUNTS);
3. Eseguire il test;
4. Eliminare il record (per non disturbare gli altri casi di test).

Questo si traduce nel seguente codice:

```
...
@Nested
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@DisplayName("When an account exists")
class WhenAnAccountExists {

    Connection connection;

    @BeforeAll
    void createConnectionToPostgres() throws SQLException {
        connection = postgres.createConnection("?" + postgres.getJdbcUrl());
    }

    @AfterAll
    void closeConnectionToPostgres() throws SQLException {
        connection.close();
    }

    String username = "pippo11";
    String password = "pppassword";

    @BeforeEach
    void addAccount() throws SQLException {
        var statement = connection.prepareStatement(
            "INSERT INTO ACCOUNTS (username, password) VALUES (?, ?);"
        );
        statement.setString(1, username);
        statement.setString(2, password);
        statement.execute();
    }

    @AfterEach
    void removeAccount() throws SQLException {
        var statement = connection.prepareStatement(
            "DELETE FROM ACCOUNTS WHERE username = ?;"
        );
        statement.setString(1, username);
        statement.execute();
    }
    ...
}
```

Prima di tutto, nel metodo `createConnectionToPostgres()` annotato `@BeforeAll`, viene creata una connessione al database usando l'oggetto `Connection`. La connessione viene aperta un'unica volta, usata per eseguire operazioni di scrittura/lettura e chiusa soltanto alla fine (nel metodo `closeConnectionToPostgres()` annotato `@AfterAll`). Successivamente, fissiamo due fixture, `username` uguale a “pippo11” e `password` uguale a “pppassword”. Queste fixture vengono usate nel metodo `addAccount()` annotato `@BeforeEach`. Nel metodo `addAccount()` si sfrutta la connessione creata al db per eseguire una `INSERT INTO ACCOUNTS ...` creando difatti un account con tali credenziali. Dopo l'esecuzione di ogni test, l'account creato viene rimosso con il metodo

`removeAccount()` annotato `@AfterEach`. Il metodo elimina il record creato sollevando una `DELETE FROM ACCOUNTS ...` usando la stessa connessione al db creata all'inizio.

A questo punto, possiamo realizzare un caso di test in cui:

1. Viene creato un oggetto `Auth` con campi `username` e `password` uguali a “`pippo11`” e “`pp-password`” (**arrange**);
2. Viene inviata una richiesta di creazione token `POST /auth/token` usando l'oggetto `Auth` come HTTP body (**act**);
3. Si asserisce che lo status della risposta HTTP sia effettivamente `200 OK` e che il response body contenga effettivamente un token (**assert**).

Il caso di test è il seguente:

```
...
@Test
@DisplayName("Should allow him to create a token")
void shouldAllowHimToCreateAToken() {
    // Arrange
    Auth authentication = new Auth(username, password);

    // Act
    Response response = authApi.createToken(authentication);

    // Assert
    String createdToken = response.getCookie(TOKEN_NAME);
    assertThat(response.statusCode(), is(equalTo(HttpStatus.SC_OK)));
    assertThat(createdToken, is(not(emptyString())));
}

...
}
```

Si noti che nella parte *Assert* del caso di test, viene estratto dalla risposta HTTP il token creato usando la libreria REST Assured. In particolare, viene estratto il cookie di nome `token` dall'header `Set-Cookie` verificando che effettivamente esista e che sia una stringa non vuota.

Per testare l'endpoint `POST /auth/validate`, il cui scopo è verificare la validità di un token, seguiremo un approccio simile a quanto fatto per l'endpoint precedente. L'endpoint dovrebbe restituire:

- HTTP `200 OK`: se il token fornito è valido. Un token è valido se esiste e non è scaduto;
- HTTP `403 Forbidden`: altrimenti.

Verifichiamo prima il caso in cui il token fornito esiste e dovrebbe essere valido. Per fare ciò, estendiamo la test suite con il seguente caso di test:

```
...
@Nested
@DisplayName("When he/she creates a token")
class WhenRequestToken {

    Token token;

    @BeforeEach
    void requestToken() {
        Response response = authApi.createToken(new Auth(username, password));
        token = new Token(response.getCookie(TOKEN_NAME));
    }

    @Test
    @DisplayName("When checking it with /auth/validate endpoint, it should be valid")
}
```

```

    void shouldBeValid() {
        Response response = authApi.validateToken(token);

        assertThat(response.statusCode(), is(equalTo(HttpStatus.SC_OK)));
    }
}

...
}

}
...
}

```

In questo caso di test, aggiungiamo una nuova fixture `Token`. Nel metodo `requestToken()` annotato `@BeforeEach` viene prima creato un token usando l'account creato nel metodo `addAccount()`. Tale token viene quindi usato per verificare la sua validità nel metodo di test `shouldBeValid()`.

Come già accennato precedentemente, il token ha una data di scadenza, oltre la quale non dovrebbe essere più considerato valido. Quando un token è scaduto e si solleva una POST `/auth/validate`, l'endpoint dovrebbe restituire HTTP 403 `Forbidden`. Come testare questo comportamento? Testare questo tipo di comportamento risulta essere più complesso rispetto ai test effettuati in precedenza. Per simulare il comportamento desiderato, dovremmo creare un caso di test che comprenda i seguenti passaggi:

1. **Creazione del Token:** iniziamo generando un token valido.
2. **Attesa della Scadenza del Token:** attendiamo che il token raggiunga la data di scadenza prevista.
3. **Invio della richiesta di validazione:** Successivamente, inviamo una richiesta POST `/auth/validate` utilizzando il token scaduto come input.
4. **Verifica della risposta:** infine, verifichiamo che la risposta restituita dall'endpoint sia effettivamente HTTP 403 `Forbidden`.

Tuttavia, si pone un problema evidente: se la durata del token è, ad esempio, un'ora, il caso di test richiederebbe un'attesa di un'ora, il che risulta inaccettabile in termini di efficienza e tempi di esecuzione. La buona notizia è che la durata del token è configurabile tramite una specifica proprietà dell'applicazione. Questa configurabilità rappresenta un importante principio di **controllabilità del software**, fondamentale nell'ambito del software testing. La possibilità di controllare e configurare le condizioni di test aiuta nella progettazione e nell'implementazione di casi di test, consentendo di gestire scenari come quello che stiamo affrontando. A questo punto, possiamo realizzare un caso di test in cui:

1. Viene settata una durata del token a runtime pari a 3 secondi;
2. Viene creato un nuovo token usando il solito account;
3. Si aspettano almeno 3 secondi;
4. Viene inviata una richiesta POST `/auth/validate` utilizzando il token scaduto come input;
5. Si asserisce che la risposta sia effettivamente HTTP 403 `Forbidden`.

Il caso di test è il seguente:

```

...
@Nested
@DisplayName("When he/she creates a token with a lifespan of 3 seconds")
class WhenCreatesATokenWithLifespanEqualTo3sec {

    @Autowired
    AppConfig appConfig; // Configurazioni dell'applicazione

    Token token;

    @BeforeEach
    void requestToken() {
        appConfig.setTokenLifeDuration(Duration.ofSeconds(3));
    }
}

```

```

        Response response = authApi.createToken(new Auth(username, password));
        token = new Token(response.getCookie(TOKEN_NAME));
    }

    @Test
    @DisplayName("Should expire in 3 seconds")
    void shouldExpire() {
        Awaitility.await()
            .with()
            .pollDelay(Duration.ofSeconds(3))
            .atMost(Duration.ofSeconds(4))
            .untilAsserted(() -> {
                Response response = authApi.validateToken(token);
                assertThat(response.statusCode(), is(equalTo(HttpStatus.SC_FORBIDDEN)));
            });
    }
}

...
}

```

Nel caso di test, abbiamo usato la libreria Awaitility che consente di esprimere le aspettative di un sistema asincrono in modo conciso e di facile lettura. Il caso di test aspetta un minimo di 3 secondi prima di eseguire il codice fornito al metodo `untilAsserted(...)`. Se dopo massimo 4 secondi (compresi i 3 secondi iniziali) l'asserzione presente in tale metodo non risulta essere vera, il test fallisce.

Non ci resta verificare che, quando si tenta di validare un token non esistente, l'endpoint dovrebbe sempre restituire HTTP 403 Forbidden. Usando `@ParameterizedTest` come fatto già precedentemente, eseguiamo tale test fornendo differenti valori di input:

```

...
@ParameterizedTest
@NullAndEmptySource
@MethodSource("provideNonExistentTokens")
@DisplayName("When trying to validate a non-existent token with
the /auth/validate endpoint, it should not be valid")
void nonValidToken(String stringToken) {
    Token token = new Token(stringToken);

    Response response = authApi.validateToken(token);

    assertThat(response.statusCode(), is(equalTo(HttpStatus.SC_FORBIDDEN)));
}

private static Stream<Arguments> provideNonExistentTokens() {
    return Stream.of(
        Arguments.of("a"),
        Arguments.of("aa"),
        Arguments.of("B"),
        Arguments.of("BC"),
        Arguments.of("ppipppozzo354L"),
        Arguments.of("fi23r902uig"),
        Arguments.of("admin"),
        Arguments.of("token")
    );
}

```

A questo punto, non ci resta che eseguire la test suite e visualizzare i risultati. La Figura 10 mostra i risultati dei test utilizzando come ambiente di sviluppo IntelliJ IDEA:

Dai risultati dei test, emerge che la gestione della data di scadenza dei token presenta anomalie e non funziona come previsto.

Auth API (com.automationtesting.it.checks)		4 sec 618ms
✓	When trying to validate a non-existent token with the /auth/validate endpoint, it should not be valid	443 ms
✓ [1]	stringToken=null	382 ms
✓ [2]	stringToken=	9 ms
✓ [3]	stringToken=a	7 ms
✓ [4]	stringToken=aa	7 ms
✓ [5]	stringToken=B	6 ms
✓ [6]	stringToken=BC	7 ms
✓ [7]	stringToken=ppippopozzo354L	6 ms
✓ [8]	stringToken=f123r902uig	7 ms
✓ [9]	stringToken=admin	6 ms
✓ [10]	stringToken=tokn	6 ms
✗	When an account exists	4 sec 120 ms
✓	Should allow him to create a token	26 ms
✗	When he/she creates a token with a lifespan of 3 seconds	4 sec 83 ms
✗	Should expire in 3 seconds	4 sec 83 ms
✓	When he/she creates a token	11 ms
✓	When checking it with /auth/validate endpoint, it should be valid	11 ms
✓	When trying to create a token using incorrect credentials	55 ms
✓	Should deny the creation of a token to a non-existing user	55 ms
✓ [1]	username= , password=	5 ms
✓ [2]	username=null, password=null	6 ms
✓ [3]	username= , password=	5 ms
✓ [4]	username= , password=	5 ms
✓ [5]	username= , password=	5 ms
✓ [6]	username=ppippopozzo354L, password=18sujudeDDF1L	4 ms
✓ [7]	username=admin, password=	6 ms
✓ [8]	username=admin, password=admin	7 ms
✓ [9]	username=root, password=	6 ms
✓ [10]	username=root, password=root	6 ms

Figura 10: Risultati test suite Auth service

3.3 Component Testing

Nell’ambito dell’automazione, una delle sfide più significative a cui dobbiamo far fronte è la necessità di ridurre al minimo il numero di “falsi positivi” che possono essere generati durante l’esecuzione dei test automatici. Nel contesto del test automatizzato del software, un falso positivo significa che un caso di test fallisce anche quando il software in esame (SUT) non presenta nessun difetto. Come risultato di un falso positivo, i tester passano il tempo a cercare un bug che non esiste. Questo fenomeno si verifica spesso quando il software da testare dipende da componenti esterni. I microservizi spesso dipendono da altri microservizi per svolgere le proprie attività.

Il **Component Testing** è progettato per verificare il corretto funzionamento di un servizio specifico. Questo tipo di testing ha un campo di applicazione limitato e mira a esaminare la funzionalità end-to-end di un singolo microservizio. Per fare ciò, si isola il microservizio all’interno del sistema, sostituendo le sue dipendenze con test doubles o servizi mock. Questo approccio consente di testare il microservizio in uno stato controllato, riducendo al minimo le interferenze esterne e garantendo che qualsiasi problema rilevato sia direttamente correlato al servizio in esame. In questa sezione, esploreremo il Component Testing utilizzando esempi pratici e tecniche di mocking per simulare il comportamento di altri microservizi.

Le principali caratteristiche del component testing sono le seguenti:

- Sono tipicamente eseguiti insieme agli Unit Test;
- Non dovrebbero dipendere da componenti esterni (i componenti esterni devono essere sostituiti con dei *test doubles*);
- Sono più lenti degli Unit Test ma più veloci degli E2E Test.

La Figura 11 rappresenta lo scopo di un component test. Un component test verifica il comportamento di un servizio in isolamento.

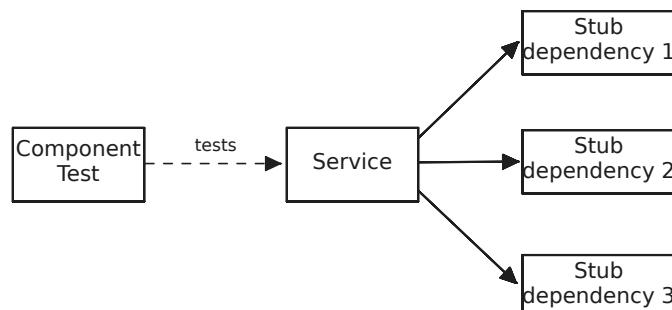


Figura 11: Un component test testa un servizio in isolamento. Usa tipicamente degli stubs per sostituire le dipendenze del servizio.

3.3.1 Testare un microservizio in isolamento - Room Service

In questa sezione, testeremo in isolamento il servizio **Room Service**. Il servizio **Room Service** dipende dal servizio **Auth Service** (si veda Figura 2). Ognqualvolta si effettuano operazioni relative alla creazione, cancellazione o modifica delle stanze in un B&B, il servizio **Room Service** deve fare affidamento su **Auth Service** per verificare l'autenticità e l'autorizzazione degli utenti. Il successo del processo di creazione, modifica o eliminazione di una stanza è dipendente da **Auth Service**. Ciò implica che ogni problematica con **AuthService** risulterà in un fallimento che non è necessariamente legato a **Room Service**. Questo incide sull'affidabilità del test stessi. Al fine di condurre test su queste funzionalità senza la necessità di coinvolgere direttamente il servizio di autenticazione, è necessario sostituire la dipendenza da **Auth Service** con un test double. In questo modo, potremo eseguire i test in un ambiente controllato senza essere dipendenti dall'autenticazione esterna.

Creare un test double che riproduca il comportamento del servizio esterno in modo coerente può essere complesso. Fortunatamente, esistono differenti librerie che facilitano questo compito. Una tra queste è **WireMock**. Con tale strumento possiamo creare test double di API facilmente. La Figura 12 mostra graficamente esattamente quello che in questa sezione andremo a fare.

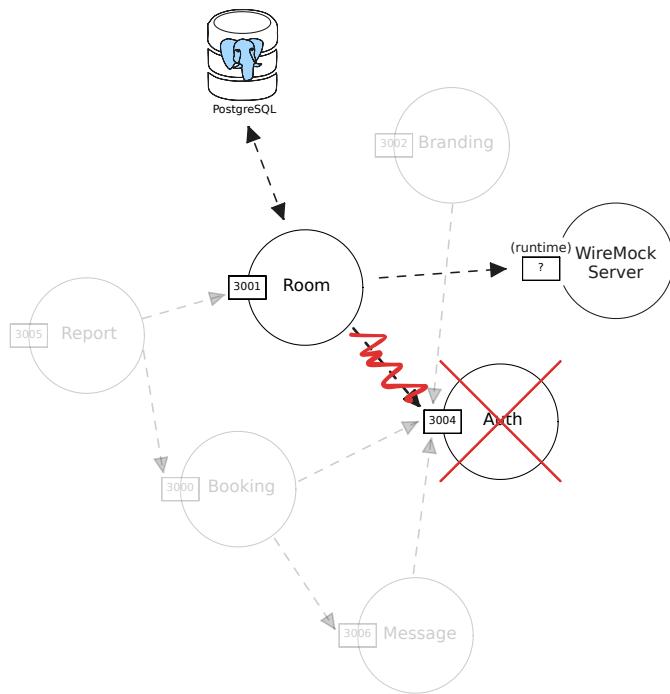


Figura 12: Rimpiazzare la dipendenza da **Auth Service** con un mock usando **WireMock**.

Come evidenziato nella Figura 12, il servizio **Auth Service** verrà sostituito da **Room Service** tramite l'utilizzo di un **WireMockServer**. Un **WireMock Server** è un server HTTP che può essere utilizzato come un servizio standard. La caratteristica distintiva di **WireMock** è la sua configurabilità, che consente di definire le risposte che dovrebbero essere restituite per richieste specifiche, simulando così il comportamento del servizio originale, come **Auth Service**.

Il nostro scopo è quello di testare le funzionalità di creazione, modifica ed eliminazione delle stanze attraverso le API fornite da **Room Service**. Tuttavia, come già accennato, dobbiamo creare un test double di **Auth Service** utilizzando **WireMock**. Per raggiungere questo obiettivo, è necessario eseguire i seguenti passaggi:

1. Creare un **WireMockServer**;
2. Configurare il **WireMockServer** in maniera tale da simulare il comportamento di **Auth Service**;
3. Prima di avviare i test, configurare **Room Service** affinché punti al test double di **Auth Service** appena creato.

Nella Figura 12, notiamo un'altra dipendenza di **Room Service**: il suo database PostgreSQL. Tuttavia, è importante sottolineare che in questo caso non è necessario creare un mock del database. Questo perché ciascun servizio ha il proprio database e ogni database è utilizzato esclusivamente

dal servizio corrispondente. In altre parole, esiste una relazione uno a uno tra il database e il servizio. Di conseguenza, questo database è accessibile esclusivamente dal servizio Room Service. Poiché questa dipendenza non è osservabile dall'esterno (ma è una parte interna del servizio stesso), le comunicazioni con questa dipendenza diventano dettagli di implementazione che devono essere inclusi all'interno dei componenti test.

Per aumentare la comprensibilità della classe di test, possiamo inizialmente creare una classe denominata `RoomConfigCT` (dove CT sta per Component Test). Questa classe conterrà tutte le configurazioni necessarie per l'esecuzione dei test, compresa la creazione di test doubles.

```

@SpringBootTest(
    webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT,
    classes = RoomApplication.class
)
@EnableWireMock({@ConfigureWireMock(
    name = RoomConfigCT.AUTH_SERVICE_WIREMOCK_NAME,
    property = RoomConfigCT.AUTH_SERVICE_WIREMOCK_PROPERTY_URL
)})
@ExtendWith(SpringExtension.class)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@Testcontainers
@ActiveProfiles("test")
@DisplayName("Room API")
public class RoomConfigCT {

    static final String AUTH_SERVICE_WIREMOCK_NAME = "auth-service";
    static final String AUTH_SERVICE_WIREMOCK_PROPERTY_URL = "app.auth-service-client-url";
    static final String PG_IS_READY_COMMAND = "pg_isready -U postgres";

    @Container
    static PostgreSQLContainer<?> postgres =
        new PostgreSQLContainer<>("postgres:15.2-alpine")
            .withUsername("postgres")
            .withPassword("postgres")
            .withInitScript("init-db.sql")
            .withDatabaseName("test")
            .waitingFor(Wait.forSuccessfulCommand(PG_IS_READY_COMMAND))
            .withExposedPorts(5432);

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        postgres.start(); // Necessario quando si usa Lifecycle.PER_CLASS
        registry.add("auth.db.jdbc-url", postgres::getJdbcUrl);
        registry.add("auth.db.jdbc-username", postgres::getUsername);
        registry.add("auth.db.jdbc-password", postgres::getPassword);
    }

    ...
}

```

Le annotazioni presenti sulla classe `RoomConfigCT` sono le stesse annotazioni presentate nelle sezioni precedenti. La novità è l'annotazione `@EnableWireMock` che permette di integrare la libreria WireMock con il framework Spring Boot e JUnit 5. L'annotazione accetta due fondamentali parametri:

- `name`: l'identificativo del `WireMockService`. In questo caso è uguale a `"auth-service"`;
- `property`: questo parametro specifica il nome della proprietà nell'ambiente Spring in cui sarà impostato l'URL del `WireMockService`. Il nome proprietà è impostata uguale a `"app.auth-service-client-url"` che è esattamente la proprietà usata dal servizio `RoomService` per effettuare chiamate verso il servizio `AuthService`.

Successivamente, viene avviata una istanza di PostgreSQL con `Testcontainers` e bindata al servizio `RoomService`. Lo schema del database viene creato usando lo script `init-db.sql` presente nella cartella `src/test/resources`. In particolare, viene creata una tabella `ROOMS`.

Il seguente codice, poi, configura il comportamento di `WireMockService`:

```
...
```

```

    @InjectWireMock(AUTH_SERVICE_WIREMOCK_NAME)
    WireMockServer wiremock;

    @BeforeEach
    void configureWireMockAuthService() {
        wiremock.stubFor(post("/auth/validate")
            .willReturn(aResponse()
                .withStatus(200)));
    }

    ...
}

```

Con `@InjectWireMock` otteniamo l'istanza di `WireMockServer` identificata da `"auth-service"`. Successivamente, nel metodo `configureWireMockAuthService()` annotato `@BeforeEach`, configuriamo questa istanza di `WireMockServer` in modo che restituisca sempre una risposta HTTP 200 OK quando viene effettuata una chiamata POST `/auth/validate`. Questa configurazione riguarda la chiamata effettuata dal servizio `RoomService` per convalidare il token di autenticazione prima di eseguire operazioni di creazione o modifica sulle stanze. Impostando questo comportamento, siamo in grado di utilizzare qualsiasi token di validazione quando eseguiamo chiamate di creazione o modifica delle stanze verso il servizio `RoomService`. È opportuno precisare che il giusto nominativo da dare in questo caso al test double che abbiamo appena creato è **stub** e NON mock. Questo perché la risposta alla chiamata che stiamo cercando di simulare (cioè la risposta a POST `/auth/validate`) non ha nessun side-effect lato `AuthService`.¹

Infine, configuriamo la variabile d'istanza `RoomApi` che fornisce i metodi per inviare richieste HTTP a `RoomService`.

```

    ...
    RoomApi roomApi;

    @BeforeAll
    void setupRoomApi() {
        RestAssured.defaultParser = JSON;
        String roomUrl = Optional
            .ofNullable(System.getenv(RoomApi.ROOM_SERVICE_URL_ENV_VAR_NAME))
            .orElse("http://localhost:3001");
        roomApi = new RoomApi(roomUrl);
    }
}

```

A questo punto, siamo pronti per testare le funzionalità di creazione, modifica ed eliminazione delle stanze. Iniziamo con l'endpoint POST `/room/` per la creazione delle stanze. La richiesta ha bisogno del seguente payload che rappresenta una stanza:

```
{
    "roomId": 0,
    "roomName": "",
    "type": "",
    "accessible": false,
    "image": "",
    "description": "",
    "features": [],
    "roomPrice": 0
}
```

Per questo motivo, definiamo una classe POJO `Room` che rappresenti tale payload nel package `payloads`:

```

public class Room {
    @JsonProperty private int roomid;
    @JsonProperty private String roomName;
    @JsonProperty private String type;
    @JsonProperty private boolean accessible;
    @JsonProperty private String image;
}

```

¹<https://vkhorikov.medium.com/dont-mock-your-database-it-s-an-implementation-detail-8f1b527c78be>

```

    @JsonProperty private String description;
    @JsonProperty private String[] features;
    @JsonProperty private int roomPrice;

    // Getters and Setters
}

```

Vogliamo verificare che, quando si richiede la creazione di una stanza valida con POST /room/, RoomService dovrebbe restituire HTTP 201 Created. Per eseguire più test usando lo stesso metodo di test, useremo `@ParameterizedTest` come già fatto in precedenza. Tuttavia, in questo caso, abbiamo a che fare con un oggetto più complesso costituito da differenti campi. Usando JUnit 5, è possibile usare un file .csv per fornire i dati di input al metodo. Ogni riga rappresenta una istanza di Room che viene data in input al metodo di test in questione. Abbiamo perciò bisogno di fornire un sistema per deserializzare le righe del csv in una istanza Room. La libreria JUnit 5 permette di implementare una interfaccia `ArgumentsAggregator` finalizzata a questo scopo. Dunque, creiamo la classe `RoomAggregator` che implementa `ArgumentsAggregator`:

```

public class RoomAggregator implements ArgumentsAggregator {
    @Override
    public Object aggregateArguments(
        ArgumentsAccessor argumentsAccessor,
        ParameterContext parameterContext
    ) throws ArgumentsAggregationException {
        String roomName = argumentsAccessor.getString(0);
        String roomType = argumentsAccessor.getString(1);
        int roomPrice = argumentsAccessor.getInteger(2);
        String roomDescription = argumentsAccessor.getString(3);
        String[] roomFeatures = argumentsAccessor.getString(4).split("\\|");
        String roomImageUrl = argumentsAccessor.getString(5);

        Room room = new Room();
        room.setRoomName(roomName);
        room.setType(roomType);
        room.setRoomPrice(roomPrice);
        room.setDescription(roomDescription);
        room.setFeatures(roomFeatures);
        room.setImage(roomImageUrl);

        return room;
    }
}

```

Tale classe consentirà di convertire una riga del csv del genere:

```
Sunset Room,Single,45,Scenic View,TV|Wi-Fi,https://t.ly/12345
```

in una istanza Room. Una volta realizzato il nostro aggregatore, possiamo realizzare il seguente test:

```

public class RoomCT extends RoomConfigCT {

    static final Token DUMMY_TOKEN = new Token("");

    @ParameterizedTest
    @CsvFileSource(resources = "/correct-room-instances.csv", numLinesToSkip = 1)
    @DisplayName("When creating a room, it should return HTTP 201 Created")
    void whenCreatingARoomShouldReturn201(@AggregateWith(RoomAggregator.class) Room room) {
        Response response = roomApi.createRoom(room, DUMMY_TOKEN);

        assertThat(response.getStatusCode(), is(equalTo(HttpStatus.SC_CREATED)));
        assertThat(response.as(Room.class), hasProperty("roomid", is(notNullValue())));
    }

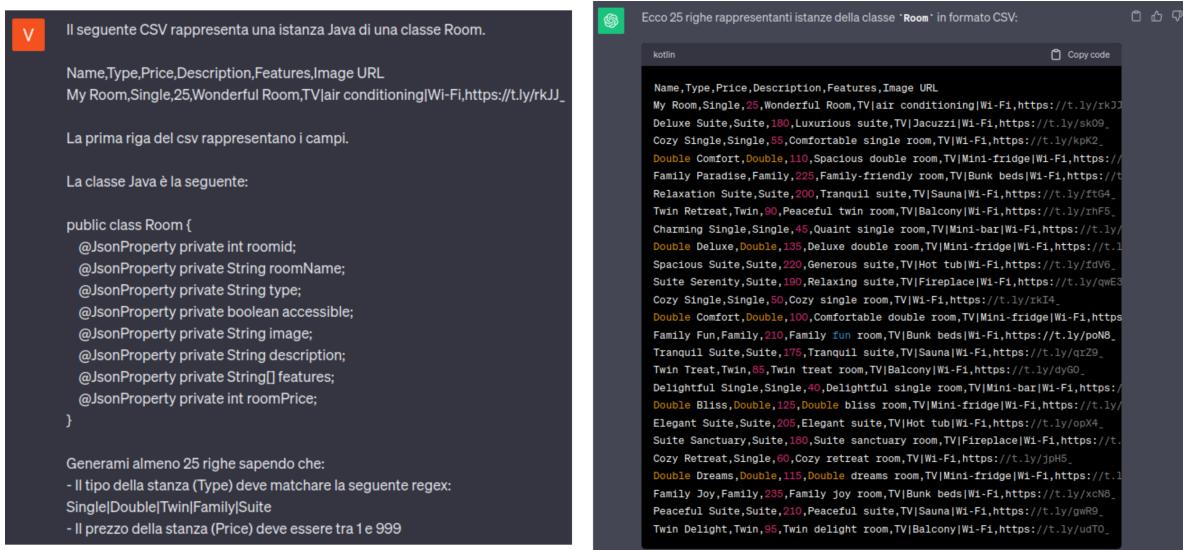
    ...
}

```

Il metodo di test è annotato `@ParameterizedTest`. In particolare, l'annotazione `@CsvFileSource` permette di prelevare i dati da dare in input al metodo da un csv di nome `correct-room-instances.csv`

presente in `src/main/test`. Inoltre, settiamo `numLinesToSkip` uguale a 1 perché la prima riga è di intestazione e serve, di solito, dare i nomi alle colonne. A questo punto, usando l'aggregatore `RoomAggregator` e l'annotazione `@AggregateWith`, siamo capaci di convertire ogni riga in una istanza `Room`. Ogni istanza rappresenta un caso di test apparte. Ogni test case dovrebbe restituire HTTP 201 `Created` e il body contenuto nella risposta ricevuta dovrebbe essere una `Room` e dovrebbe contenere un campo `roomid` diverso da `null`. Si noti che il token passato al metodo `createRoom(...)` è la stringa vuota. Ciò non è un problema, perché sappiamo di aver creato uno stub del servizio `AuthService` che, indipendentemente dal valore del token, restituirà sempre 200 HTTP OK ad ogni richiesta POST `/auth/validate`.

Possiamo sfruttare ChatGPT per realizzare un CSV di stanze ben formate da dare in pasto al test come mostrato in Figura 13.



The screenshot shows two panels from ChatGPT. The left panel contains the following text:

V Il seguente CSV rappresenta una istanza Java di una classe Room.

```
Name,Type,Price,Description,Features,Image URL
My Room,Single,25,Wonderful Room,TV|air conditioning|Wi-Fi,https://t.ly/rkJJ_
```

La prima riga del csv rappresentano i campi.

La classe Java è la seguente:

```
public class Room {
    @JsonProperty private int roomid;
    @JsonProperty private String roomName;
    @JsonProperty private String type;
    @JsonProperty private boolean accessible;
    @JsonProperty private String image;
    @JsonProperty private String description;
    @JsonProperty private String[] features;
    @JsonProperty private int roomPrice;
}
```

Generami almeno 25 righe sapendo che:

- Il tipo della stanza (Type) deve matchare la seguente regex: Single|Double|Twin|Family|Suite
- Il prezzo della stanza (Price) deve essere tra 1 e 999

The right panel shows the generated CSV content:

```
Name,Type,Price,Description,Features,Image URL
My Room,Single,25,Wonderful Room,TV|air conditioning|Wi-Fi,https://t.ly/rkJJ_
Deluxe Suite,Suite,180,Luxurious suite,TV|Jacuzzi|Wi-Fi,https://t.ly/sk09_
Cozy Single,Single,55,Comfortable single room,TV|Wi-Fi,https://t.ly/pk2_
Double Comfort,Double,110,Spacious double room,TV|Mini-fridge|Wi-Fi,https://t.ly/1qE_
Family Paradise,Family,225,Family-friendly room,TV|Bunk beds|Wi-Fi,https://t.ly/1qE_
Relaxation Suite,Suite,200,Tranquil suite,TV|Sauna|Wi-Fi,https://t.ly/ftG4_
Twin Retreat,Twin,90,Peaceful twin room,TV|Balcony|Wi-Fi,https://t.ly/hf5_
Charming Single,Single,45,Quaint single room,TV|Mini-bar|Wi-Fi,https://t.ly/
Double Deluxe,Double,135,Deluxe double room,TV|Mini-fridge|Wi-Fi,https://t.ly/1qE_
Spacious Suite,Suite,220,Generous suite,TV|Hot tub|Wi-Fi,https://t.ly/rdv6_
Suite Serenity,Suite,190,Relaxing suite,TV|Fireplace|Wi-Fi,https://t.ly/qwE3_
Cozy Single,Single,50,Cozy single room,TV|Wi-Fi,https://t.ly/rK14_
Double Comfort,Double,100,Comfortable double room,TV|Mini-fridge|Wi-Fi,https://t.ly/1qE_
Family Fun,Family,210,Family fun room,TV|Bunk beds|Wi-Fi,https://t.ly/pnR8_
Tranquil Suite,Suite,175,Tranquil suite,TV|Sauna|Wi-Fi,https://t.ly/rz29_
Twin Treat,Twin,85,Twin treat room,TV|Balcony|Wi-Fi,https://t.ly/dyG0_
Delightful Single,Single,40,Delightful single room,TV|Mini-bar|Wi-Fi,https://t.ly/1qE_
Double Bliss,Double,125,Double bliss room,TV|Mini-fridge|Wi-Fi,https://t.ly/1qE_
Elegant Suite,Suite,205,Elegant suite,TV|Hot tub|Wi-Fi,https://t.ly/pxP4_
Suite Sanctuary,Suite,180,Suite sanctuary room,TV|Fireplace|Wi-Fi,https://t.ly/1qE_
Cozy Retreat,Single,60,Cozy retreat room,TV|Wi-Fi,https://t.ly/jph5_
Double Dreams,Double,115,double dreams room,TV|Mini-fridge|Wi-Fi,https://t.ly/1qE_
Family Joy,Family,235,Family joy room,TV|Bunk beds|Wi-Fi,https://t.ly/xnR8_
Peaceful Suite,Suite,210,Peaceful suite,TV|Sauna|Wi-Fi,https://t.ly/gwR9_
Twin Delight,Twin,85,Twin delight room,TV|Balcony|Wi-Fi,https://t.ly/gUOZ_
```

Figura 13: Realizzare un CSV da dare in pasto ai metodi di test annotati `@CsvFileSource`

Usando lo stesso approccio, possiamo estendere la suite di test creando un csv di stanze non valide:

```
...
@ParameterizedTest
@CsvFileSource(resources = "/incorrect-room-instances.csv", numLinesToSkip = 1)
@DisplayName("When creating an invalid room, it should return HTTP 400 Bad Request")
void whenCreatingAnInvalidRoomShouldReturn400(@AggregateWith(RoomAggregator.class) Room room) {
    Response response = roomApi.createRoom(room, DUMMY_TOKEN);

    assertThat(response.getStatusCode(), is(equalTo(HttpStatus.SC_BAD_REQUEST)));
    Error error = response.getBody().as(Error.class);
    assertThat(error.getFieldErrors(),
        either(hasItemInArray("Room name must be set"))
        .or(hasItemInArray("Type must be set"))
        .or(hasItemInArray("must not be null"))
        .or(hasItemInArray("Type can only contain the room options Single, Double, Twin, Family or Suite"))
        .or(hasItemInArray("must be less than or equal to 999")));
}
...
}
```

In questo caso, siccome la stanza che si sta cercando di creare è invalida, `RoomService` dovrebbe ritornare HTTP 400 `Bad Request`. Inoltre, il body della risposta dovrebbe essere di tipo `Error`, e i messaggi di errore dovrebbero essere almeno uno di quelli specificati nelle asserzioni.

Adesso, con l'aggiunta di una singola stanza nota identificata con ID 1 nel database, ci proponiamo di sviluppare i seguenti casi di test:

Test Case 1 Quando la si elimina con `DELETE /room/1`, dovrebbe ritornare HTTP 202 Accepted;

Test Case 2 Quando la si elimina con `DELETE /room/1` due volte, l'ultima richiesta dovrebbe ritornare HTTP 404 Not Found;

Test Case 3 Quando la si ottiene con `GET /room/1`, dovrebbe ritornare HTTP 200 OK;

Test Case 4 Quando si ottengono tutte le stanze `GET /room`, dovrebbe ritornare soltanto una stanza;

Test Case 5 Quando si aggiorna la stanza con `PUT /room/1`, dovrebbe ritornare HTTP 202 Accepted.

Prima di procedere con l'esecuzione dei casi di test, è fondamentale preparare l'ambiente. Questo processo richiede la messa a punto delle seguenti condizioni:

1. Creazione di una connessione al database PostgreSQL.
2. Eliminazione di tutte le stanze presenti prima dell'esecuzione di ciascun caso di test. Questa operazione è essenziale per garantire la pulizia dello stato del database tra un test case e l'altro.
3. Creazione di una stanza con dettagli noti (nome, descrizione, ecc...) e assegnazione dell'ID 1 prima dell'avvio di ogni caso di test.
4. Esecuzione dei casi di test.
5. Chiusura della connessione al database al termine dell'intero processo di test.

Per mantenere l'organizzazione e la leggibilità, suddividiamo i casi di test in classi innestate. Inoltre, avviamo una connessione al database (tramite il metodo `createConnectionToPostgres()` annotato con `@BeforeAll`) e la chiudiamo quando sono stati eseguiti tutti i casi di test (mediante il metodo `closeConnectionToPostgres()` annotato con `@AfterAll`):

```
...
@Nested
@DisplayName("When creating a room with id 1")
class WhenCreatingARoom extends RoomConfigCT {

    Connection connection;

    @BeforeAll
    void createConnectionToPostgres() throws SQLException {
        connection = postgres.createConnection("?" + postgres.getJdbcUrl());
    }

    @AfterAll
    void closeConnectionToPostgres() throws SQLException {
        connection.close();
    }

    ...
}

}
```

Una volta completati questi passaggi preliminari, prima di avviare ciascun caso di test, procediamo con l'eliminazione di tutti i record presenti nella tabella ROOMS. Successivamente, creiamo la stanza con ID 1 e con dettagli noti (nome, descrizione, ecc...) che verranno usati nelle asserzioni. Per garantire che l'eliminazione dei record avvenga prima della creazione della stanza, è utile annotare i metodi `deleteAllRoomsFromDatabase()` e `createRoom()` rispettivamente con `@Order(1)` e `@Order(2)`. Anche se questo non è strettamente necessario (dato che l'ordine dei metodi definisce anche l'ordine di esecuzione), tali annotazioni comunicano a chi legge i casi di test che l'ordine *deve* essere questo, e dunque non va cambiato.

```
...
Connection connection;
```

```

@BeforeAll
void createConnectionToPostgres() throws SQLException {
    connection = postgres.createConnection("?" + postgres.getJdbcUrl());
}

@AfterAll
void closeConnectionToPostgres() throws SQLException {
    connection.close();
}

@Order(1)
@BeforeEach
void deleteAllRoomsFromDatabase() throws SQLException {
    var statement = connection.createStatement();
    statement.execute("TRUNCATE ROOMS");
}

final int ROOM_ID = 1;
final String ROOM_NAME = "Business Suite";
final String ROOM_TYPE = "Suite";
final int ROOM_PRICE = 320;
final int ROOM_BEDS = 3;
final boolean ROOM_ACCESSIBLE = false;
final String ROOM_URL_IMAGE = "https://t.ly/67890";
final String ROOM_DESCRIPTION = "For Professional";
final String[] ROOM_FEATURES = new String[] { "TV", "mini bar", "Wi-Fi" };

@Order(2)
@BeforeEach
void createRoom() throws SQLException {
    var statement = connection.prepareStatement(
        "INSERT INTO ROOMS VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?);"
    );

    statement.setInt(1, ROOM_ID);
    statement.setString(2, ROOM_NAME);
    statement.setString(3, ROOM_TYPE);
    statement.setInt(4, ROOM_BEDS);
    statement.setBoolean(5, ROOM_ACCESSIBLE);
    statement.setString(6, ROOM_URL_IMAGE);
    statement.setString(7, ROOM_DESCRIPTION);
    statement.setArray(8, connection.createArrayOf("VARCHAR", ROOM_FEATURES));
    statement.setInt(9, ROOM_PRICE);

    statement.execute();
}
}

...
}

}
}

```

Procediamo con l'implementazione dei casi di test **Test Case 1** e **Test Case 2**:

```

...
@Test
@DisplayName("""
    When deleting the room with DELETE /room/1,
    it should return HTTP 202 Accepted
""")
void deleteRoom() {
    Response deleteResponse = roomApi.deleteRoom(ROOM_ID, DUMMY_TOKEN);
    Response getResponse = roomApi.getRoom(1, DUMMY_TOKEN);

    assertThat(deleteResponse.getStatusCode(), is(equalTo(HttpStatus.SC_ACCEPTED)));
}

```

```

        assertThat(response.getStatusCode(), is(equalTo(HttpStatus.SC_NOT_FOUND)));
    }

    @Test
    @DisplayName(""""
        When deleting the room twice with DELETE /room/1,
        it should return HTTP 404 Not Found
    """)
    void deleteRoomTwice() {
        roomApi.deleteRoom(ROOM_ID, DUMMY_TOKEN);
        Response response = roomApi.deleteRoom(ROOM_ID, DUMMY_TOKEN);

        assertThat(response.getStatusCode(), is(equalTo(HttpStatus.SC_NOT_FOUND)));
    }

    ...
}

}

```

Il metodo di test `deleteRoom()` non solo verifica che la risposta alla richiesta `DELETE /room/1` sia `HTTP 202 Accepted`, ma anche che una successiva richiesta `GET /room/1` restituisca `HTTP 404 Not Found`. Il secondo metodo di test, `deleteRoomTwice()`, verifica che una prima `DELETE /room/1` restituisca `HTTP 202 Accepted`, e una seconda richiesta identica restituisca `HTTP 404 Not Found`, confermando così che la stanza è stata effettivamente eliminata.

I casi di test **Test Case 3** e **Test Case 4** possono essere implementati nel seguente modo:

```

...
@Test
@DisplayName("When getting the room with GET /room/1, it should return HTTP 200 OK")
void getRoom() {
    Response response = roomApi.getRoom(ROOM_ID, DUMMY_TOKEN);

    assertThat(response.getStatusCode(), is(equalTo(HttpStatus.SC_OK)));
    Room room = response.as(Room.class);
    assertThat(room.getRoomName(), is(equalTo(ROOM_NAME)));
    assertThat(room.getRoomPrice(), is(equalTo(ROOM_PRICE)));
    assertThat(room.getDescription(), is(equalTo(ROOM_DESCRIPTION)));
    assertThat(room.getType(), is(equalTo(ROOM_TYPE)));
    assertThat(room.getImage(), is(equalTo(ROOM_URL_IMAGE)));
    assertThat(room.getRoomid(), is(equalTo(ROOM_ID)));
    assertThat(room.getFeatures(), is(equalTo(ROOM_FEATURES)));
    assertThat(room.getBeds(), is(equalTo(ROOM_BEDS)));
}

@Test
@DisplayName("When call GET /room, it should return only one room")
void getRooms() {
    Response response = roomApi.getRooms(DUMMY_TOKEN);

    assertThat(response.getStatusCode(), is(equalTo(HttpStatus.SC_OK)));
    Rooms rooms = response.as(Rooms.class);
    assertThat(rooms.size(), is(equalTo(1)));
    Room firstRoom = rooms.first();
    assertThat(firstRoom.getRoomName(), is(equalTo(ROOM_NAME)));
    assertThat(firstRoom.getRoomPrice(), is(equalTo(ROOM_PRICE)));
    assertThat(firstRoom.getDescription(), is(equalTo(ROOM_DESCRIPTION)));
    assertThat(firstRoom.getType(), is(equalTo(ROOM_TYPE)));
    assertThat(firstRoom.getImage(), is(equalTo(ROOM_URL_IMAGE)));
    assertThat(firstRoom.getRoomid(), is(equalTo(ROOM_ID)));
    assertThat(firstRoom.getFeatures(), is(equalTo(ROOM_FEATURES)));
    assertThat(firstRoom.getBeds(), is(equalTo(ROOM_BEDS)));
}

...

```

```

    }
}
}
```

Il metodo di test `getRoom()` verifica semplicemente che la richiesta GET `/room/1` restituisca HTTP 200 OK. Inoltre, esso verifica che il body della risposta rappresenti una stanza di tipo `Room` e che i dettagli di questa stanza siano esattamente quelli utilizzati per crearla prima dell'esecuzione del caso di test. D'altra parte, il metodo `getRooms()` esegue una GET `/room` per ottenere tutte le stanze. Successivamente, verifica che venga restituito HTTP 200 OK, che l'elenco delle stanze contenga esattamente una sola stanza e che questa unica stanza abbia gli stessi campi e dettagli della stanza creata precedentemente all'esecuzione del test.

Il test case **Test Case 5** ha l'obiettivo di verificare la funzionalità di aggiornamento delle stanze. In particolare, questo caso di test si concentra sull'endpoint PUT `/room/{id}`, che accetta un body con i campi da modificare della stanza. L'obiettivo è di verificare che questa richiesta modifichi con successo la stanza, assicurandosi che l'ID rimanga invariato. Per implementare questo caso di test, organizziamo il contesto, come al solito, in una classe innestata `WhenUpdateRoom`:

```

...
@Nested
@DisplayName("Given we want to update the room with id 1")
class WhenUpdateRoom {
    final String NEW_ROOM_NAME = "Economy Twin";
    final int NEW_ROOM_PRICE = 100;
    final String NEW_ROOM_TYPE = "Twin";
    final int NEW_ROOM_BEDS = 2;
    final String NEW_ROOM_DESCRIPTION = "Value Stay";
    final String[] NEW_ROOM_FEATURES = new String[] {"TV", "Wi-Fi"};
    Room newRoom;

    @BeforeEach
    void instantiateNewRoom() {
        newRoom = new Room();
        newRoom.setRoomId(ROOM_ID);
        newRoom.setRoomPrice(NEW_ROOM_PRICE);
        newRoom.setImage(ROOM_URL_IMAGE);
        newRoom.setFeatures(NEW_ROOM_FEATURES);
        newRoom.setType(NEW_ROOM_TYPE);
        newRoom.setDescription(NEW_ROOM_DESCRIPTION);
        newRoom.setAccessible(ROOM_ACCESSIBLE);
        newRoom.setRoomName(NEW_ROOM_NAME);
        newRoom.setBeds(NEW_ROOM_BEDS);
    }

    ...
}

}
```

All'interno di questa sottoclasse, definiamo come variabili di istanza i campi che intendiamo modificare. In seguito, nel metodo `instantiateNewRoom()` annotato con `@BeforeEach`, creiamo un'istanza dell'oggetto `Room`. Procediamo, dunque, con l'implementazione del caso di test **Test Case 5**:

```

...
@Test
@DisplayName("When updating the room with PUT /room/1, it should return HTTP 202 Accepted")
void updateRoom() {
    Response response = roomApi.updateRoom(newRoom, ROOM_ID, DUMMY_TOKEN);

    assertThat(response.getStatusCode(), is(equalTo(HttpStatus.SC_ACCEPTED)));
    Room room = response.as(Room.class);
    assertThat(room.getRoomName(), is(equalTo(NEW_ROOM_NAME)));
    assertThat(room.getRoomPrice(), is(equalTo(NEW_ROOM_PRICE)));


}
```

```
        assertThat(room.getDescription(), is(equalTo(NEW_ROOM_DESCRIPTION)));
        assertThat(room.getType(), is(equalTo(NEW_ROOM_TYPE)));
        assertThat(room.getImage(), is(equalTo(ROOM_URL_IMAGE)));
        assertThat(room.getRoomid(), is(equalTo(ROOM_ID)));
        assertThat(room.getFeatures(), is(equalTo(NEW_ROOM_FEATURES)));
        assertThat(room.getBeds(), is(equalTo(NEW_ROOM_BEDS)));
    }
}
```

Ora possiamo eseguire l'intera suite di test e verificare i risultati ottenuti. La Figura 14 rappresenta i risultati dei test quando si utilizza l'ambiente di sviluppo IntelliJ IDEA:

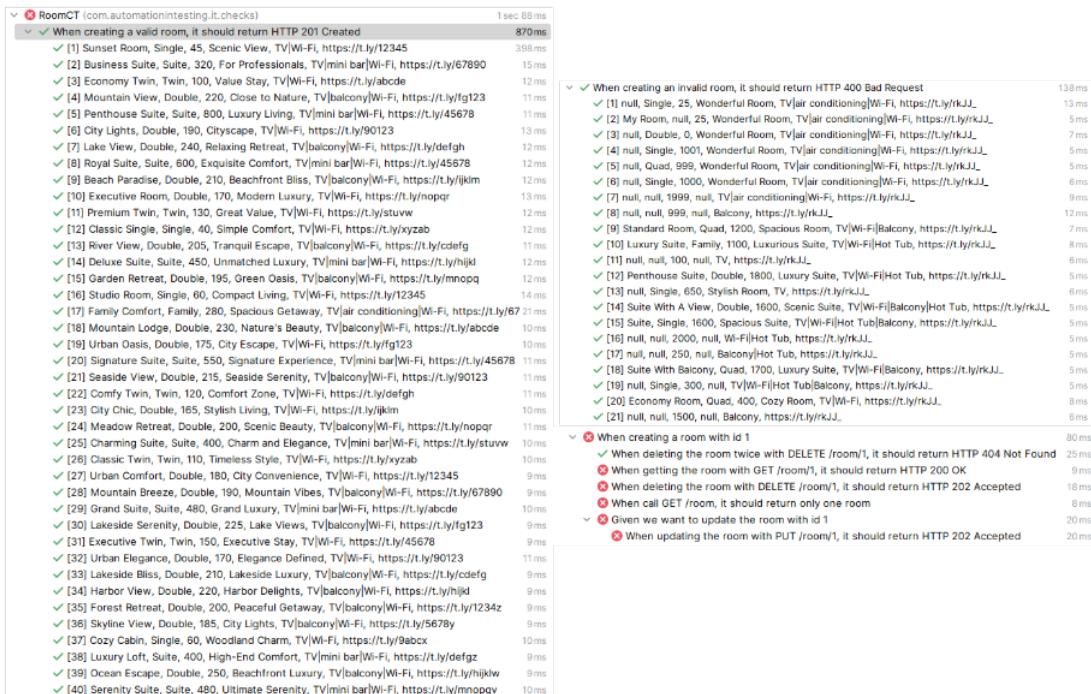


Figura 14: Risultati test suite Room service

Da quanto mostrato in Figura 14, è evidente che la suite di test ha rilevato più di un failure. In particolare:

- Il caso di test **Test Case 2** ha restituito un codice di stato HTTP 500 Internal Server Error, anziché HTTP 404 Not Found come previsto. Questo risultato anomalo sembrerebbe essere dovuto a un'eccezione, nello specifico una `PSQLEException`, che viene lanciata quando si tenta di effettuare una richiesta GET /room/id su una stanza inesistente.

```
@Test
@DisplayName("""
    When deleting the room with DELETE /room/1,
    it should return HTTP 202 Accepted
""")
void deleteRoom() {
    Response deleteResponse = roomApi.deleteRoom(ROOM_ID, DUMMY_TOKEN);
    Response getResponse = roomApi.getRoom(1, DUMMY_TOKEN);

    assertThat(deleteResponse.getStatusCode(), is(equalTo(HttpStatus.SC_ACCEPTED)));
    assertThat(getResponse.getStatusCode(), is(equalTo(HttpStatus.SC_NOT_FOUND)));
}
```

- I casi di test **Test Case 3**, **Test Case 4** e **Test Case 5** condividono tutti un'asserzione comune:

```
...
    assertThat(firstRoom.getBeds(), is(equalTo(ROOM_BEDS)));
...
}
```

Questa specifica asserzione è quella che causa il fallimento. In particolare, indipendentemente dal fatto che si stia recuperando o modificando una stanza (attraverso gli endpoint `GET /room/1`, `GET /rooms`, e `PUT /room/1`), il corpo della risposta HTTP restituisce un valore errato per il campo `beds` (che rappresenta il numero di posti letto).

In questa sezione, abbiamo esaminato a fondo il componente testing, un’approccio che ci permette di testare un microservizio in un ambiente isolato. Nel prossimo paragrafo, ci addentreremo in una delle sfide più intricate e dispendiose all’interno della struttura piramidale dei test, ovvero il testing end-to-end. Durante questa fase, affronteremo test che coinvolgono l’intero sistema, verificando come diverse parti interagiscono tra loro per garantire il corretto funzionamento dell’applicazione.

3.4 End-to-end Testing

I test **end-to-end** costituiscono un’estensione dei test di integrazione, i quali a loro volta sono costruiti su tutte le altre forme di test esaminate nei paragrafi precedenti. La caratteristica distintiva dei test end-to-end, come suggerisce il nome, è quella di esplorare l’intero flusso dell’applicazione, dal suo inizio alla sua fine. In teoria, questi test dovrebbero emulare le interazioni di un utente reale con l’applicazione o almeno simulare le azioni che un utente effettuerebbe.

Nella pratica, la scrittura di questi test risulta spesso la più complessa e richiede un notevole dispendio di tempo per svilupparli. I test end-to-end tendono a essere più lenti rispetto ad altri tipi di test, pertanto vengono frequentemente eseguiti in modo separato dal normale processo di sviluppo principale, spesso su un build server come Jenkins.

Idealmente, un test end-to-end dovrebbe ricreare un ambiente che somigli il più possibile a quello di produzione. Questi test sono fondamentali per garantire che l’input venga elaborato correttamente, partendo dall’interfaccia utente del frontend fino a raggiungere il backend. La sfida principale consiste nel coordinare e far interagire tra loro tutti i microservizi autonomi in esecuzione su una singola macchina.

Questi test possono essere svolti in ambienti “white-box” o “black-box”, o in una combinazione di entrambi. Un ambiente “white-box” coinvolge elementi visibili all’esterno dell’applicazione, mentre un ambiente “black-box” verifica la funzionalità effettiva in background.

- In un ambiente “white-box”, simuliamo l’interazione dell’utente con l’interfaccia utente stessa, poiché l’utente può vedere le azioni in corso.
- Nello stesso contesto, le azioni dell’utente possono attivare processi nel server sottostante. Questi processi, noti come backend, sono invisibili all’utente ma influenzano ciò che l’utente sperimenta. Questa parte è un ambiente “black-box” perché le azioni sono eseguite senza visibilità diretta.

Nella pratica, i test end-to-end combinano spesso entrambi gli ambienti, specialmente nelle applicazioni basate su browser. Il browser rappresenta il “white-box” poiché l’utente può interagire e osservare, mentre il server è il “black-box” in quanto elabora le richieste in modo invisibile e fornisce risposte.

In questa sezione esploreremo tecniche di testing end-to-end usando **Selenium Web Driver API**, **Cucumber** e **Testcontainers**. Selenium Web Driver API è una API che consente di automatizzare il controllo di un browser web, permettendo l’esecuzione di azioni come il click di pulsanti, la compilazione di form e la navigazione tra le pagine web. Cucumber consente di ideare test che tendono ad essere più di alto livello. Cucumber è un framework di testing che permette di scrivere test in linguaggio naturale (in inglese) in modo che siano facilmente comprensibili ai non sviluppatori. Questi test possono essere utilizzati per verificare il comportamento del software in base a scenari di utilizzo specifici. Cucumber aiuta a tradurre i requisiti del software in test automatici, consentendo una comunicazione più chiara tra team di sviluppo e stakeholder. Permette di scrivere test in linguaggio umano e automatizzarli per verificare il software.

3.4.1 Testing E2E - Making a booking for a room

In questa sezione, eseguiremo un test end-to-end completo della funzionalità di prenotazione di una stanza, testando l’intero flusso dell’applicazione, a partire dall’interfaccia utente. Per fare ciò,

innanzitutto descriviamo degli scenari in termini di *executable specifications* (specifiche eseguibili). Le specifiche eseguibili sono scritte utilizzando termini e concetti del dominio aziendale. Sono progettate per essere facilmente comprese dagli utenti finali e sono definite in modo collaborativo dall'intero di uno o più team. Questa tipologia di test vengono chiamati **acceptance tests**: sono test rivolti al business, test che gli utenti finali possono utilizzare per verificare che una funzionalità funzioni come previsto.

Per descrivere i test di accettazione per la funzionalità di prenotazione, sfruttiamo il pattern *Given-When-Then* e definiamo il seguente *scenario*:

Feature: Making a booking for a room

```
Scenario: User makes a booking
  Given There is a free room
  And I am on the homepage
  When I make a booking
  Then I will see "Booking Successful" on the screen
  Then I will see "Unavailable" on newly booked calendar days
```

Un utente finale può facilmente comprendere uno scenario scritto in questo modo. Fornisce obiettivi chiari e oggettivi in termini di ciò che deve essere sviluppato e di ciò che deve essere testato. Lo scenario appena descritto utilizza il linguaggio **Gherkin**. Gherkin utilizza un insieme di parole chiave speciali per dare struttura e significato alle specifiche eseguibili. Con strumenti appropriati, gli scenari scritti in questa forma possono essere trasformati in criteri di accettazione automatici che possono essere eseguiti automaticamente ogni volta che risulta necessario.

In Gherkin, i requisiti relativi a una particolare funzionalità sono raggruppati in un singolo file di testo chiamato *feature file*, che contiene una breve descrizione della funzionalità, seguita da una serie di scenari, o esempi formalizzati del funzionamento della funzionalità. Il feature file presentato sopra è molto semplice e contiene un solo scenario. Ogni scenario è composto da un certo numero di *steps*, in cui ogni step inizia con una particolare keyword (*Given*, *When*, *Then*, *And*, *But*).

È possibile automatizzare una specifica eseguibile come quella appena descritta scrivendo del codice che corrisponda a ciascun step. Strumenti di **Behavior-driven development** come **Cucumber** consentono di “mappare” ogni step dello scenario in codice di test. Tuttavia, prima di iniziare scrivere il codice che implementa questa specifica eseguibile, è necessario configurare l’intera infrastruttura per testare la funzionalità end-to-end. Questo rappresenta uno dei passaggi più complessi nell’implementazione di un test end-to-end. È necessario avviare sulla propria macchina (come già accennato, questa tipologia di test vengono di solito eseguiti su un build server) tutti i servizi di Restful Booker Platform (Figura 2). Per facilitare questo compito, useremo ancora una volta la libreria Testcontainers. Possiamo definire un Docker Compose file per definire e avviare l’intera infrastruttura di servizi necessaria per eseguire i test end-to-end. Successivamente, possiamo usare Testcontainers per avviare questo file Docker Compose direttamente dal nostro codice. Creiamo dunque una classe **E2EConfig** in un package denominato **e2e** che ha esattamente questo compito e che verrà eseguita prima dei test end-to-end:

```
public class E2EConfig {

    private static final File DOCKER_COMPOSE_FILE = new File(
        "src/test/resources/docker/docker-compose-test.yaml"
    );

    @BeforeAll
    public static void setupServices() {
        DockerComposeContainer<?> compose =
            new DockerComposeContainer<>(DOCKER_COMPOSE_FILE)
                .waitingFor("rbp-proxy", Wait.defaultWaitStrategy())
                .withExposedService("rbp-proxy", 8080)
                .withExposedService("rbp-booking", 3000)
                .withExposedService("rbp-room", 3001)
                .withExposedService("rbp-assets", 3003)
                .withExposedService("rbp-auth", 3004)
                .withExposedService("rbp-report", 3005)
                .withExposedService("rbp-message", 3006)
                .withExposedService("postgres", 5432);
        compose.start();
    }
}
```

```

    }
}

```

Il Docker Compose file è posizionato in `src/test/resources/docker/docker-compose-test.yaml`. Il metodo annotato `@BeforeAll` (questa annotazione proviene dalla libreria di Cucumber e non da quella di JUnit) crea una istanza di `DockerComposeContainer<?>` e avvia ogni servizio definito nel Docker Compose file.

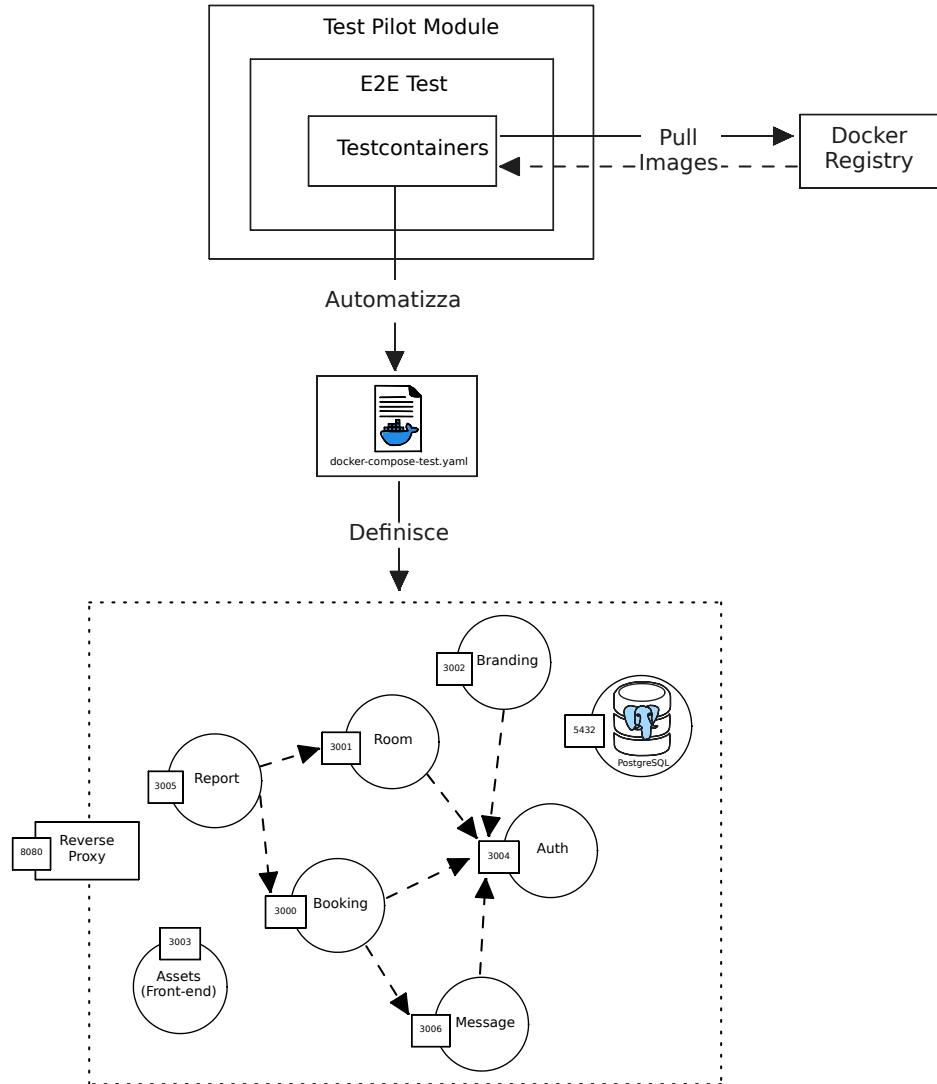


Figura 15: Avviare i servizi di Restful Booker Platform prima di eseguire i test end-to-end

La Figura 15 offre una panoramica delle operazioni eseguite dalla classe `E2EConfig` prima dell'avvio dei test end-to-end. È importante notare che i test end-to-end sono organizzati all'interno di un modulo dedicato denominato `test-pilot`. Inoltre, osserviamo che le immagini Docker devono essere scaricate da un registro, spesso privato.

Successivamente, è fondamentale integrare la libreria JUnit 5 con quella di Cucumber. Questa integrazione è essenziale per assicurarsi che l'engine Cucumber esegua la classe `E2EConfig` prima di eseguire i test end-to-end stessi. Per fare ciò, inseriamo nel package `e2e` la seguente classe:

```

package e2e;

import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.Suite;

import static io.cucumber.junit.platform.engine.Constants.GLUE_PROPERTY_NAME;

@Suite
@IncludeEngines("cucumber")

```

```

@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value = "e2e")
public class RunCucumberTest {}
```

Le annotazioni `@Suite` e `@IncludeEngines("cucumber")` permettono di integrare la libreria JUnit 5 con l'engine Cucumber. Successivamente, tramite l'annotazione `@ConfigurationParameter` stiamo dicendo all'engine integrato (in questo caso Cucumber) che le varie classi di configurazioni e le implementazioni dei file `.feature` si trovano sotto il package `e2e`. Inoltre, i file `.feature` vengono implicitamente ricercati sotto la cartella `src/test/resources`. Adesso è possibile eseguire i test Cucumber con Maven usando il seguente comando:

```
mvn test -Dcucumber.features=src/test/resources -f pom.xml
```

Il percorso `src/test/resources` del modulo `test-pilot` contiene un solo file `.feature` denominato `bookings.feature` il cui contenuto è stato presentato all'inizio di questa sezione. Tuttavia, al momento, non esiste un'implementazione per questi step. Il passo successivo consiste quindi nell'implementare ogni step all'interno di questo file.

Iniziamo col trovare un modo per automatizzare le interazioni con un browser. Questo è dove entrano in gioco gli strumenti forniti da Selenium. In particolare, utilizzeremo Selenium WebDriver, uno strumento essenziale per l'automazione dei test con i browser. Lo scenario che vogliamo testare è un utente che esegue una prenotazione per una stanza libera. Questo corrisponde a:

`Scenario: User makes a booking`

Il primo step (`Given`) è il seguente:

```

Given There is a free room
And I am on the homepage
```

Per completare questo step, è fondamentale avviare un'istanza del browser. Sebbene sia possibile eseguire i test end-to-end più volte su diversi browser con poco sforzo e in maniera automatizzata, in questo esempio ci concentreremo sull'utilizzo di Google Chrome. Per farlo, utilizzeremo il driver di Google Chrome fornito da Selenium. Creiamo una classe `MakeAReservationStepDefs` nel package `e2e.stepdefs`. Questa classe conterrà l'implementazione dello scenario che vogliamo testare.

```

public class MakeAReservationStepDefs {

    WebDriver driver = new ChromeDriver();

    ...

}
```

Definiamo subito una istanza di `WebDriver`, in particolare una istanza `ChromeDriver`. Si noti che, a questo punto, l'intera infrastruttura di Restful Booker Platform è completamente operativa, compreso il servizio `assets` che fornisce risorse necessarie al browser. Per implementare il primo step, è necessario annotare il metodo che lo rappresenta con le annotazioni `@Given` e `@And` come mostrato di seguito:

```

...
@Given("There is a free room")
@And("I am on the homepage")
public void i_am_on_the_homepage() {
    driver.manage()
        .timeouts()
        .implicitlyWait(Duration.ofSeconds(2));

    driver.get("http://localhost:8080");
}
```

Il contenuto di queste annotazioni deve corrispondere esattamente a quanto definito nello scenario nel file `bookings.feature`. Nel metodo `i_am_on_the_homepage()`, impostiamo il driver in modo da essere “tollerante” nei confronti di eventuali elementi non trovati, consentendo una tolleranza massima di 2 secondi durante l'analisi del DOM. Questa impostazione è cruciale poiché potrebbe verificarsi che alcuni elementi non siano ancora stati renderizzati completamente. Invece di restituire immediatamente un errore in caso di mancata rilevazione di un elemento, il driver attende

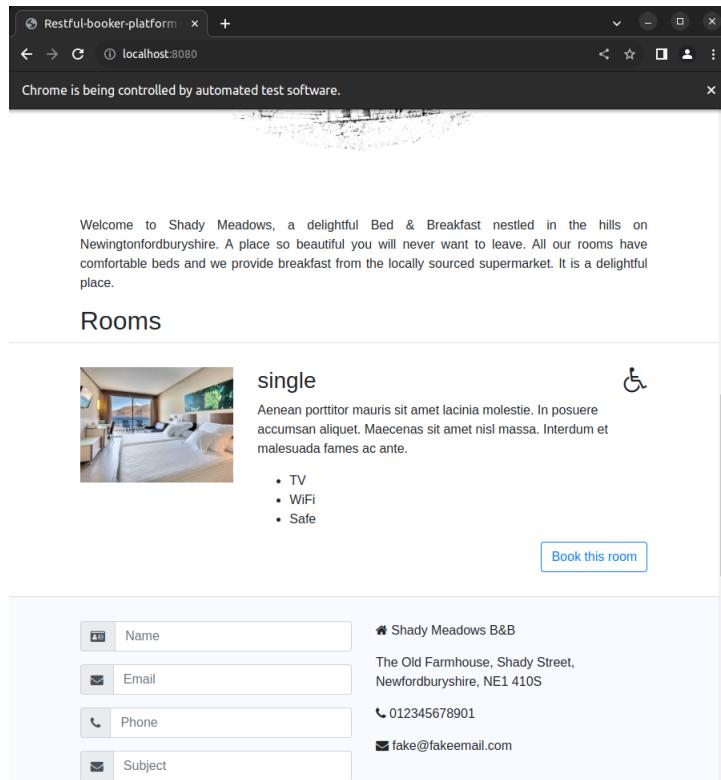


Figura 16: Selenium Homepage

fino a un massimo di 2 secondi e ritenta l'operazione se necessario. Successivamente, indichiamo al driver di seguire l'URL `http://localhost:8080`, che corrisponde alla homepage di Restful Booker Platform, come mostrato in Figura 16.

Abbiamo quindi soddisfatto questo step:

- Esiste una stanza libera (senza prenotazioni);
- L'utente si trova sulla homepage.

Il prossimo step è il seguente:

When I make a booking

Questo è lo step più complicato perché è necessario eseguire tutti gli step sull'interfaccia grafica per eseguire una prenotazione. I passi necessari per effettuare una prenotazione da interfaccia grafica sono evidenziati in Figura 17.

Il metodo che implementa i passi in Figura 17 è il seguente:

```
...
@When("I make a booking")
public void when_i_make_a_booking() {
    // Implementazione passi
    ...
}

...
```

Di seguito viene elencata l'implementazione di ogni passo (Figura 17):

1. Il primo passo richiede di cliccare il pulsante `Book this room`. Un riferimento al pulsante (così come tutti gli `WebElement`) può essere ottenuto eseguendo una `findElement(...)` specificando una *strategia di localizzazione* che riesca ad identificare univocamente l'elemento cercato. In questo caso, sfruttiamo uno dei valori del suo attributo `class` per ottenere un suo riferimento ed eseguire un click:

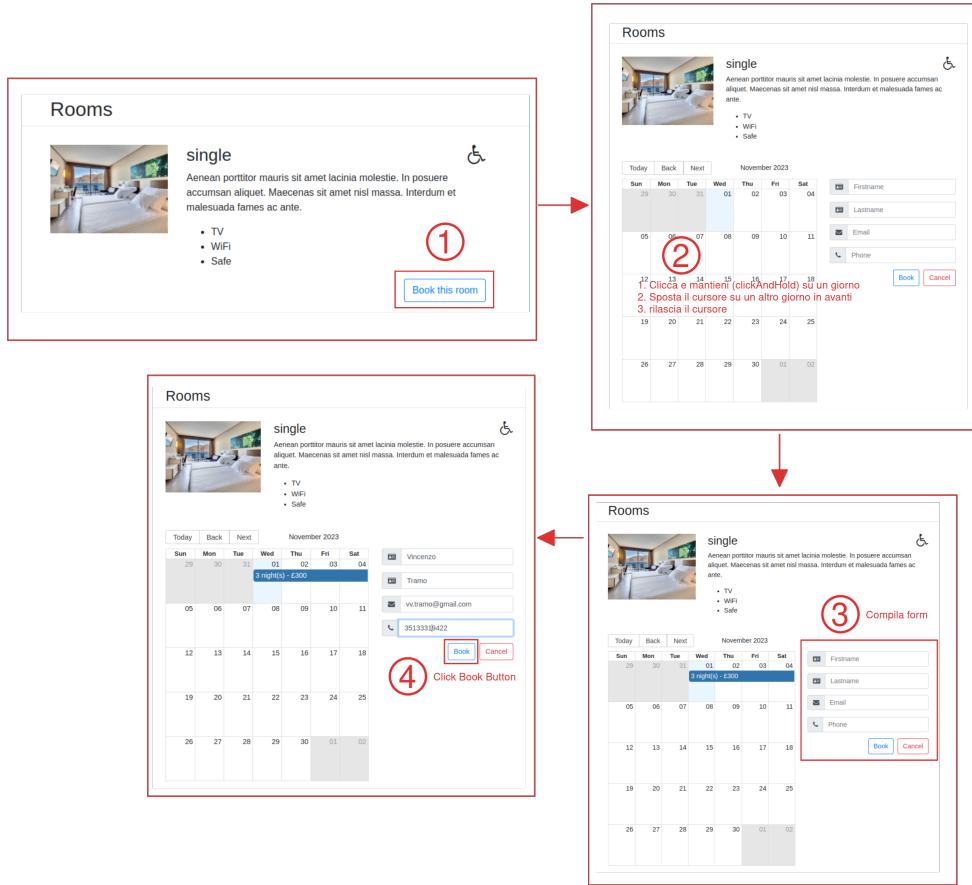


Figura 17: Passi necessari per eseguire una prenotazione da interfaccia grafica

```
...
WebElement openBookingButton = driver.findElement(By.className("openBooking"));
openBookingButton.click();
...
2. Il secondo passo è più complesso poiché coinvolge l'automazione di diverse azioni, tra cui il "click and hold", il "move to", e il "release". Prima di eseguire queste azioni, dobbiamo identificare due caselle sul calendario che rappresentano l'inizio e la fine del periodo di prenotazione. Tuttavia, per semplificare il processo e concentrarci sulla prenotazione di una stanza senza dare troppa importanza ai giorni da scegliere, possiamo impostare start come il giorno corrente ed end come l'ultimo giorno del mese corrente. Il seguente codice fa esattamente questo:
```

```
...
List<WebElement> webElements = driver.findElements(By.className("rbc-day-bg"));
WebElement start = null;
WebElement end = webElements.get(webElements.size() - 1);

for (WebElement webElement: webElements) {
    if (webElement.getAttribute("class").equals("rbc-day-bg rbc-today")) {
        start = webElement;
        break;
    }
}
...

```

Inizialmente, otteniamo una lista di tutte le caselle che rappresentano i giorni del mese corrente. Successivamente, attraverso un ciclo `for`, identifichiamo la casella che corrisponde al giorno corrente. È importante sottolineare che il codice mostrato potrebbe essere notevolmente semplificato se gli sviluppatori front-end fornissero identificatori univoci per ciascun elemento. Successivamente, possiamo facilmente eseguire le operazioni di "click and hold",

“move to” e “release” con il seguente codice:

```
...
Actions actions = new Actions(driver)
.clickAndHold(start)
.moveToElement(end)
.release();
actions.perform();
...
```

3. Il terzo passo riguarda la compilazione del form:

```
...
WebElement firstnameForm = driver.findElement(By.name("firstname"));
WebElement lastnameForm = driver.findElement(By.name("lastname"));
WebElement emailForm = driver.findElement(By.name("email"));
WebElement phoneForm = driver.findElement(By.name("phone"));

firstnameForm.sendKeys("Vincenzo");
lastnameForm.sendKeys("Tramo");
emailForm.sendKeys("vv.tramo@gmail.com");
phoneForm.sendKeys("35122259422");
...
```

4. Il quarto e ultimo passo riguarda il click del pulsante Book:

```
...
WebElement bookButton = driver.findElement(By.className("btn-outline-primary"));
bookButton.click();
...
```

Il prossimo step dello scenario è un **Then**. In questo step si va a verificare il comportamento atteso.

Then I will see "Booking Successful" on the screen

L’utente, arrivato a questo punto, dovrebbe visualizzare la seguente schermata in Figura 18:

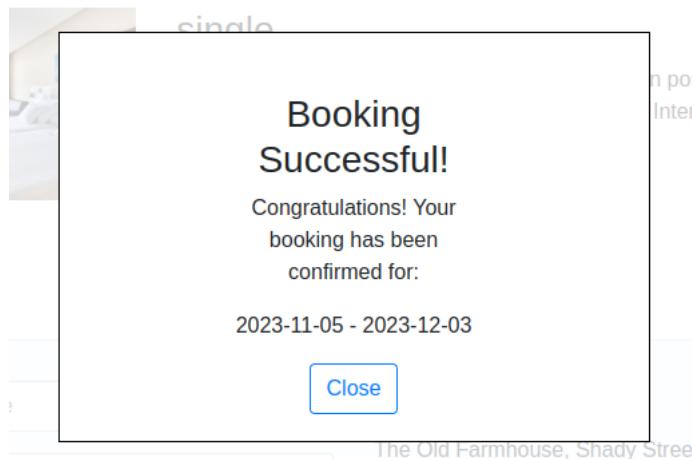


Figura 18: Schermata prenotazione effettuata con successo

L’obiettivo dello step è dunque quello di asserire che:

- La schermata in Figura 18 esiste;
- Il messaggio visualizzato sia “Booking Successful!”

```
...
@Then("I will see \"Booking Successful\" on the screen")
public void i_will_see_booking_successful_on_the_screen() {
    WebElement confirmationModal = driver.findElement(By.className("confirmation-modal"));
    WebElement successfulMessage = confirmationModal.findElement(By.tagName("h3"));
```

```

        WebElement detailedSuccessfulMessage = confirmationModal.findElement(By.tagName("p"));
        assertThat(successfulMessage.getText(), is(equalTo("Booking Successful!")));
        assertThat(detailedSuccessfulMessage.getText(), is(equalTo("Congratulations! Your booking
                                has been confirmed for:")));
    }

...

```

L'ultimo step è il seguente:

Then I will see "Unavailable" on newly booked calendar days

L'ultimo passo è ancora un passo di tipo Then e ha lo scopo di verificare che, dopo aver effettuato la prenotazione, se l'utente apre nuovamente il calendario per quella stanza, tutti i giorni disponibili del mese corrente dovrebbero essere contrassegnati come non disponibili, esattamente come illustrato nella Figura 19.



Figura 19: Schermata giorni non disponibili

L'implementazione di questo step è piuttosto semplice. Dobbiamo ripercorrere il passo per aprire il calendario associato alla stanza in questione. Prima, però, è necessario cliccare sul pulsante Close in Figura 18. Successivamente, si verifica che gli elementi che contrassegnano i giorni non disponibili effettivamente esistano e che contengano il testo "Unavailable".

```

...
@Then("I will see \"Unavailable\" on newly booked calendar days")
public void i_will_see_unavailable() {
    WebElement confirmationModal = driver.findElement(By.className("confirmation-modal"));
    WebElement closeButton = confirmationModal.findElement(By.className("btn-outline-primary"));
    closeButton.click();

    WebElement openBookingButton = driver.findElement(By.className("openBooking"));
    openBookingButton.click();

    WebElement unavailable = driver.findElement(By.className("rbc-event-content"));
    assertThat(unavailable.getText(), is(equalTo("Unavailable")));
}
}

```

I test end-to-end rappresentano un importante tassello nella strategia complessiva di testing di un'applicazione, in quanto consentono di valutare l'interazione tra tutte le componenti e i servizi in un ambiente simile a quello di produzione. Questi test mirano a verificare l'intero flusso dell'applicazione, dall'interazione dell'utente con l'interfaccia utente fino al backend e persino al database.

La loro implementazione richiede la creazione di scenari dettagliati, la gestione dell'infrastruttura e l'utilizzo di strumenti specializzati come Selenium e Cucumber. È necessario garantire che l'ambiente di test rifletta il più possibile quello di produzione, il che può essere complesso ma è cruciale per ottenere risultati significativi. Tuttavia, l'implementazione di questi test è una sfida, poiché richiede una pianificazione accurata, l'infrastruttura appropriata e la sincronizzazione tra

diverse componenti. In conclusione, i test end-to-end sono uno strumento fondamentale per garantire che l'applicazione soddisfi le aspettative degli utenti e funzioni in modo coerente in tutti i suoi aspetti.

A Compilazione ed Esecuzione Locale del Progetto

Per compilare ed eseguire il progetto in locale, è necessario soddisfare i seguenti requisiti:

- JDK 17+ (testato con JDK 19)
- Maven 3.6.3
- Node 16.15.1
- NPM 8.11.0

Una volta soddisfatti i requisiti di cui sopra, seguire questi passi:

1. Clone o scarica la repository Git dal seguente link: <https://github.com/vtramo/restful-booker-platform>
2. Navigare nella cartella `restful-booker-platform`.
3. Eseguire uno dei seguenti comandi a seconda del sistema operativo:
 - Per sistemi Linux o Mac:
`./build_locally.sh`
 - Per sistemi Windows:
`build_locally.cmd`
4. Nota: La prima esecuzione potrebbe richiedere del tempo poiché verranno scaricate le dipendenze necessarie.
4. Accedere al sito tramite il browser all'indirizzo: <http://localhost:8080>

Una volta che l'applicazione è stata creata con successo almeno una volta, è possibile eseguirla senza doverla ricompilare:

- Per sistemi Mac o Linux:
`./run_locally.sh`
- Per sistemi Windows:
`run_locally.cmd`

Ulteriori dettagli relativi all'esecuzione dei controlli, alla creazione di API e alla documentazione per lo sviluppo sono disponibili nei file README all'interno delle rispettive cartelle API.

Lo stesso principio vale per l'esecuzione dei test end-to-end. Per maggiori informazioni, consultare il modulo `test-pilot` e leggere il file `README.md` per eseguire il test end-to-end presentato in questo elaborato.