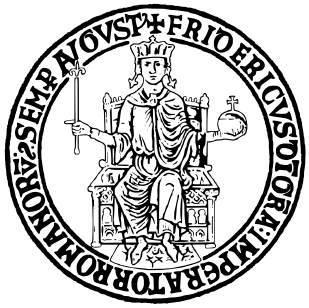
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Analisi e sviluppo di un sistema software per la gestione intelligente delle prenotazioni

Relatore
Ch.mo Tramontana Porfirio

Candidato
Spena Roberto
N86003552

Anno Accademico 2024–2025

Indice

1 Introduzione **7**

1.1 Contestualizzazione del Problema 7

1.2 Organizzazione della Tesi 8

2 Stato dell'arte **9**

2.1 Architettura Client-Server 9

2.1.1 Paradigma REST 10

2.1.2 Sicurezza in una REST API 12

2.1.3 Il ruolo dei framework nello sviluppo di una REST API 14

2.1.4 Alternative a REST per le comunicazioni client-server 15

2.2 Architettura event-driven 16

2.2.1 Message Queues (MQs) 16

2.3 Architettura a microservizi 17

2.3.1 Distribuzione attraverso i container 18

2.4 Modelli di database 18

3 Specifica dei requisiti **21**

3.1 Requisiti funzionali 22

3.2 Requisiti non funzionali 25

4 Design del sistema **27**

4.1 Backend 27

4.1.1 Microservizi implementati 29

4.1.2 Modellazione dei Flussi Asincroni per la Gestione delle Prenotazioni . 31

4.2 Frontend 33

5 Validazione del sistema **37**

5.1 Obiettivo del capitolo 37

5.2 Test unitari 39

5.2.1 Autenticazione 40

5.2.2 Gestione delle prenotazioni 59

5.3 Test a livello di sistema 73

5.3.1 Autenticazione 74

5.3.2 Gestione prenotazione 78

5.4 Conclusioni finali 89

6 Analisi dell'usabilità dell'interfaccia utente **91**

6.1 Principi di usabilità di riferimento 92

6.2 Applicazione dei principi nel design dell'interfaccia 93

6.2.1 Coerenza 93

6.2.2 Riduzione carico di memoria 95

6.2.3 Feedback 98

6.2.4 Reversibilità delle azioni 100

6.2.5 Messaggi di errore 102

6.3 Conclusioni finali 103

7 Conclusioni e prospettive future **105**

Bibliografia **106**

Capitolo 1

Introduzione

L'obiettivo di questa tesi è presentare uno studio delle moderne metodologie e tecnologie dell'ingegneria del software e, sulla base di esso, illustrare il lavoro svolto durante il tirocinio curriculare presso **BC Soft**, azienda operante nel settore della consulenza informatica con diverse sedi sul territorio nazionale. Il progetto realizzato dimostra come tali tecnologie possano essere applicate alla progettazione e allo sviluppo di un sistema complesso, pensato come soluzione a una problematica reale che coinvolge numerose attività che basano il proprio modello di servizio sulle prenotazioni come ristoratori, liberi professionisti e altre figure che lavorano su appuntamento: **la mancata presentazione del cliente (no-show)**[1].

1.1 Contestualizzazione del Problema

Il progetto nasce da un obiettivo iniziale: sviluppare un sistema capace di gestire automaticamente le prenotazioni, inviando promemoria ai clienti prenotati con richiesta obbligatoria di conferma e, in caso di mancata risposta, contattare automaticamente i prenotati in lista d'attesa seguendo una coda ordinata.

A partire da questo obiettivo generale, emergono le principali problematiche affrontate durante la progettazione di sistemi di questo tipo:

- **Gestione di un sistema event-driven**: Poiché il funzionamento si basa su conferme e aggiornamenti continui da parte degli utenti, è necessario implementare un'architettura orientata agli eventi, capace di reagire in tempo reale. Ciò implica l'adozione di tecnologie e pattern propri degli ambienti event-driven.

- **Interfaccia utente e usabilità**: Il sistema è destinato a utenti non esperti. È quindi fondamentale progettare un'interfaccia semplice, intuitiva e in grado di supportare operazioni quali l'inserimento e la visualizzazione delle prenotazioni. Questo richiede tecnologie moderne per lo sviluppo di componenti UI.
- **Persistenza e gestione dei dati**: Il sistema deve archiviare informazioni riguardanti utenti e prenotazioni. È dunque necessario integrare soluzioni di storage affidabili e coerenti con l'architettura adottata.
- **Scalabilità del sistema**: Poiché il target include liberi professionisti, piccole attività e ristoranti, il sistema deve poter gestire un carico crescente senza deteriorare le prestazioni. La progettazione deve quindi orientarsi verso un'architettura modulare, distribuita e scalabile, come ad esempio un'architettura a microservizi.

1.2 Organizzazione della Tesi

In linea con il percorso di studio intrapreso, la tesi è organizzata nei seguenti capitoli:

- **Capitolo 2. Stato dell'arte:** Viene presentata un'analisi comparativa delle tecnologie moderne e delle architetture utilizzate nello sviluppo di software contemporaneo. Tale fase preliminare di ricerca permette di individuare le soluzioni più innovative e di definire le strategie progettuali adottate successivamente.
- **Capitolo 3. Specifica dei requisiti:** Dopo lo studio introdotto nel capitolo precedente, si entra nel merito del sistema sviluppato, descrivendo in dettaglio i requisiti funzionali e non funzionali necessari alla realizzazione di una soluzione per il problema introdotto.
- **Capitolo 4. Design del sistema**: Vengono illustrate le scelte di progettazione effettuate sulla base dei requisiti individuati e delle considerazioni emerse dallo stato dell'arte.
- **Capitolo 5. Validazione del sistema:** Sono presentati i test realizzati per verificare il corretto comportamento delle funzionalità critiche, in particolare quelle legate ai flussi asincroni e alla gestione degli eventi.
- **Capitolo 6. Analisi dell'usabilità dell'interfaccia:** Vengono analizzati i principi e le linee guida adottate nella progettazione dell'interfaccia utente, con particolare attenzione all'usabilità, elemento cardine dell'ingegneria del software.
- **Capitolo 7. Conclusioni e prospettive future:** Si riassume il lavoro svolto, si discutono i limiti del sistema sviluppato e si presentano possibili estensioni e funzionalità future.

Capitolo 2

Stato dell'arte

Questo capitolo fornisce una panoramica strutturata delle principali tecnologie, metodologie e architetture che caratterizzano lo sviluppo software contemporaneo che hanno guidato le scelte progettuali illustrate nei capitoli successivi.

Verranno introdotti i modelli architetturali di riferimento: architettura Client-Server, il paradigma delle REST API, ampiamente utilizzato per la comunicazione tra componenti software, e discusso il ruolo dei framework nello sviluppo di applicazioni modulari e manutenibili.

Il capitolo approfondirà inoltre le differenze tra architetture monolitiche e architetture a microservizi, evidenziandone caratteristiche, vantaggi e limitazioni. Saranno analizzate le tecnologie di containerizzazione, oggi impiegate per semplificare la distribuzione e l'esecuzione di applicazioni, e verrà introdotto il modello dei sistemi event-driven, sempre più diffuso per la gestione di flussi asincroni e reattivi.

2.1 Architettura Client-Server

Un sistema basato su architettura Client-Server è diviso in due componenti principali:

1. **Backend (server):** Responsabile dei dati e dalla logica di business.
2. **Frontend (client):** fornisce l'interfaccia utente e consente l'interazione con i servizi del backend. Può assumere diverse forme, come applicazioni web, desktop o mobile.

I principali vantaggi di questa architettura includono:

- **Prestazioni e Scalabilità:** L'elaborazione viene distribuita tra client e server, alleggerendo il carico di lavoro sulle singole postazioni di lavoro e migliorando le prestazioni complessive del sistema. Inoltre, la natura modulare del client-server permette di aggiungere facilmente nuovi client e server per soddisfare le crescenti esigenze di calcolo

- **Indipendenza dalla Stazione di Lavoro:** Gli utenti non sono vincolati a un tipo specifico di sistema operativo o piattaforma, potendo accedere alle risorse e ai servizi indipendentemente dal dispositivo utilizzato. Questo favorisce la flessibilità e l'adattabilità del sistema a diverse esigenze e contesti.

- **Integrità dei Dati:** I server centralizzati offrono funzionalità avanzate per la protezione e la gestione dei dati, come backup, mirroring, crittografia e controllo degli accessi. Ciò garantisce un elevato livello di sicurezza e affidabilità dei dati.

- **Accessibilità dei Dati:** I dati sono centralizzati sul server, rendendoli facilmente accessibili e condivisibili da parte di più utenti simultanei, favorendo la collaborazione e la condivisione delle informazioni.

2.1.1 Paradigma REST

Nel modello client-server, la comunicazione tra le due parti avviene attraverso la rete. Il client invia delle richieste al server sfruttando un'interfaccia esposta dal backend, nota come **API (Application Programming Interface)**. Al client non interessa conoscere l'implementazione interna del server: gli è sufficiente sapere quali servizi sono disponibili e come invocarli.

REST (Representational State Transfer) è uno **stile architetturale** che definisce un insieme di principi per progettare API che sfruttano gli standard del Web, basandosi sull'uso di **HTTP** e delle **URI** per l'identificazione delle risorse. HTTP (HyperText Transfer Protocol) è il protocollo ampiamente utilizzato per le comunicazioni web e definisce due tipi fondamentali di messaggi: **request** e **response**.

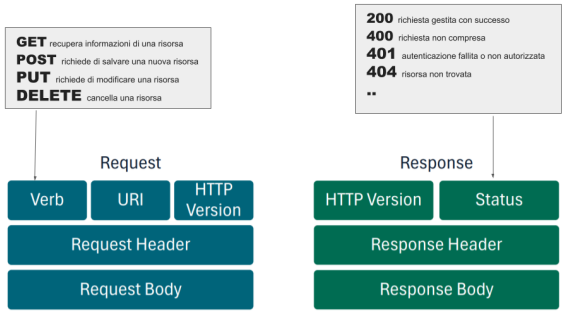


Figura 2.1: Definizioni messaggi http

Lo stile REST, definito da Roy Fielding[2] nella sua tesi di dottorato, si basa su sei vincoli principali. Quando questi vincoli sono rispettati, è possibile ottenere sistemi semplici, scalabili e facilmente integrabili.

1. **Client-Server:** La separazione tra client e server garantisce indipendenza tra interfaccia e logica di business.

2. **Stateless:** Il server non mantiene lo stato della sessione del client; ogni richiesta deve contenere tutte le informazioni necessarie per essere elaborata.

3. **Cache:** Le risposte devono indicare se possono essere memorizzate nella cache, migliorando così le prestazioni del sistema.

4. **Interfaccia uniforme:** Le interazioni tra client e server sono standardizzate, semplificando l'uso e favorendo l'interoperabilità. Include concetti come identificazione delle risorse tramite URI e rappresentazioni standardizzate dei dati.

5. **Sistema a livelli:** Il client non deve conoscere la struttura interna del sistema; le comunicazioni avvengono verso il livello immediatamente superiore, permettendo la presenza di intermediari (proxy, gateway).

6. **Codice su richiesta (facoltativo):** Il server può inviare codice eseguibile al client (ad esempio JavaScript) per estenderne dinamicamente le funzionalità.

REST (*REpresentational State Transfer*) prevede, come conseguenza dell'interfaccia uniforme, che i dati scambiati tra client e server utilizzino formati standardizzati. I formati più diffusi sono **JSON** e **XML**, con JSON oggi largamente preferito per leggibilità e semplicità

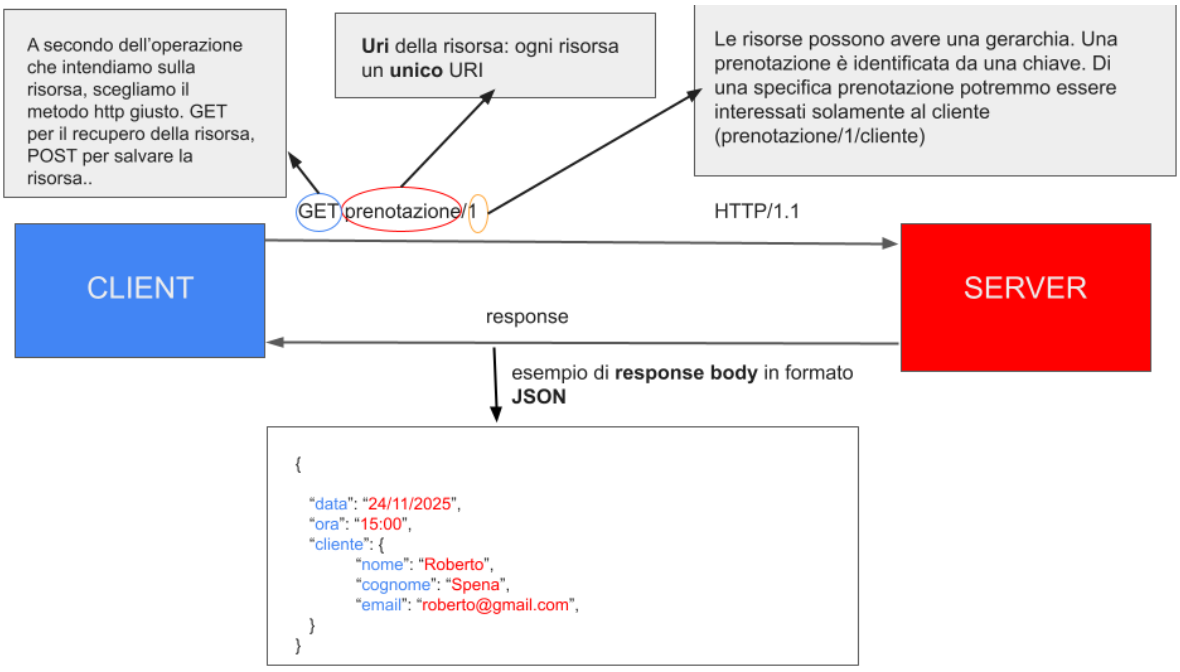


Figura 2.2: Schema architettura REST

2.1.2 Sicurezza in una REST API

Le risorse esposte tramite una REST API non sono sempre di libero accesso. Per questo motivo, la sicurezza rappresenta un aspetto fondamentale nella progettazione di un'API. I due concetti principali alla base dei meccanismi di protezione sono:

1. **Autenticazione:** verifica dell'identità dell'utente che effettua la richiesta.
2. **Autorizzazione:** verifica che l'utente autenticato abbia i permessi necessari per accedere o modificare una determinata risorsa.

Un approccio ampliamente adottato nelle API moderne consiste nell'utilizzo di sistemi di autenticazione **basati su token**. Il flusso tipico può essere riassunto come segue:

1. Il client invia una richiesta di autenticazione (di solito tramite credenziali come username e password).
2. Il server verifica la correttezza delle credenziali. Se l'autenticazione ha successo, genera un token, ovvero una stringa contenente informazioni codificate e firmate digitalmente.
3. Il token viene restituito al client nella risposta.

4. Il client memorizza il token in un proprio storage locale e lo include nelle richieste successive all'API tramite l'**Authorization Header**.

5. Prima di elaborare ogni richiesta, il server verifica la validità del token e determina l'identità dell'utente e i relativi privilegi. Il controllo può avvenire tramite diverse tecniche crittografiche a seconda del tipo di token utilizzato.

Un formato particolarmente diffuso per l'implementazione di sistemi di autenticazione stateless è il **JWT (JSON Web Token)**[3]. Un JWT è una stringra divisa in tre parti Header.Payload.Signature

- L'**header** JWT contiene informazione riguardo al tipo di token e il tipo di funzione hashing usato per la firma. È codificato utilizzando utilizzando **Base64Url** (semplicemente una codifica non una crittografia quindi facilmente decodificabile)

- Il **playload** è un oggetto JSON che contiene **claims** ovvero informazioni personalizzate nel token. Anche il playload è codificato utilizzando **Base64Url**.

- La **signature** serve a garantire la validità del token. Si ottiene applicando una funzione hashing alla stringa ottenuta concatenando header, playload e una chiave segreta conosciuta dal server.

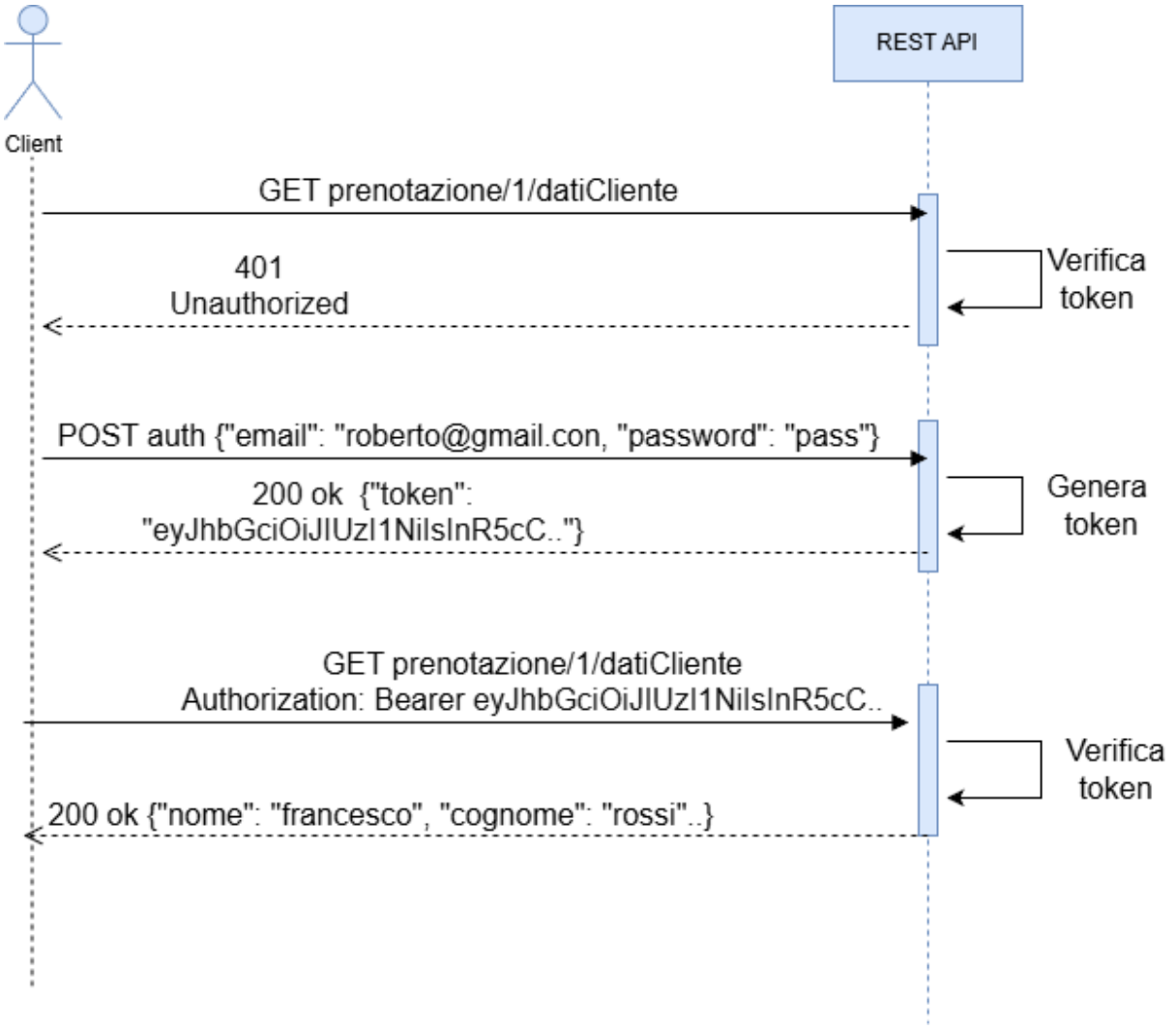


Figura 2.3: Schema autenticazione basato su token

2.1.3 Il ruolo dei framework nello sviluppo di una REST API

Implementare una REST API da zero non è complesso in sé: è sufficiente gestire le richieste HTTP, elaborarle ed inviare una risposta. Nei linguaggi prevalentemente sincroni, come Java, per ogni richiesta viene solitamente creato un nuovo thread. Una volta completata l'elaborazione, la risposta HTTP viene restituita al client. Tuttavia, dietro questo processo si nasconde una notevole quantità di attività ripetitive e tecniche: analizzare la richiesta HTTP, gestire il routing, deserializzare gli oggetti JSON/XML ricevuti, serializzare gli oggetti da restituire, applicare controlli di sicurezza e autorizzazioni.

Gli **web framework** entrano in gioco proprio per semplificare queste operazioni: offrono soluzioni già pronte e ottimizzate per i problemi più comuni, consentendo agli sviluppatori di concentrarsi sulla logica applicativa invece che sull'implementazione di dettagli infrastruttura-

li. Un principio fondamentale dell'ingegneria del software è infatti "*non reinventare la ruota*": i framework incarnano questo concetto fornendo una collezione di componenti, strumenti, librerie e buone pratiche riutilizzabili, che non implementano funzionalità applicative specifiche, ma facilitano e accelerano lo sviluppo di quelle effettive.

Una caratteristica peculiare che distingue un framework da una semplice libreria è l'**Inversion of Control (IoC)**: il flusso di controllo è invertito rispetto alla programmazione imperativa tradizionale. Non è più lo sviluppatore a decidere quando eseguire certe operazioni; è il framework che chiama il codice dell'applicazione nei momenti opportuni, gestendo il ciclo di vita delle richieste e dell'applicazione stessa. Questo approccio porta numerosi vantaggi: maggiore coerenza tra progetti diversi, riduzione della complessità, velocità di sviluppo e minore affaticamento decisionale. Tuttavia, presenta anche alcuni svantaggi: una curva di apprendimento più ripida e una possibile riduzione della flessibilità, soprattutto in progetti molto particolari o con esigenze non convenzionali.

2.1.4 Alternative a REST per le comunicazioni client-server

Sebbene le REST API rimangano la soluzione più diffusa, incontrano limitazioni nell'affrontare alcuni requisiti moderni, come prestazioni elevate, funzionalità in tempo reale e una gestione più flessibile dei dati. Per rispondere a tali esigenze sono emerse diverse alternative. Alcuni esempi possono essere:

- **GraphQL:** A differenza dell'approccio basato su molteplici endpoint tipico di REST, GraphQL espone un unico endpoint e consente al client di definire esattamente quali dati richiedere. Ciò riduce over-fetching e under-fetching ed è particolarmente utile quando i dati provengono da più servizi o richiedono aggregazioni complesse.
- **gRPC (Google Remote Procedure Call):** Basato su HTTP/2 e sul formato binario Protocol Buffers, gRPC garantisce prestazioni nettamente superiori rispetto alle API REST tradizionali. Supporta inoltre lo streaming bidirezionale, rendendolo ideale per comunicazioni ad alta efficienza tra microservizi.

Queste alternative trovano impiego soprattutto in progetti con esigenze specifiche: ad esempio, GraphQL è indicato quando l'aggregazione dei dati è complessa, mentre gRPC è preferibile in contesti in cui la velocità di comunicazione è un requisito critico.

Quando tali caratteristiche non sono necessarie, le REST API continuano a rappresentare la scelta più comune, grazie alla loro semplicità, interoperabilità e facilità di adozione.

2.2 Architettura event-driven

Man mano che i sistemi diventano sempre più grandi, a un certo punto potrebbe essere necessario eseguire alcune elaborazioni che non possono essere eseguite in linea durante la gestione di una richiesta, perché creerebbero una latenza inaccettabile.

L'**architettura guidata dagli eventi** è uno stile architetturale software che funge da modello per la progettazione delle applicazioni. Un sistema guidato dagli eventi è pensato per acquisire, comunicare ed elaborare gli eventi che si verificano tra i servizi disaccoppiati. Ciò significa che i sistemi possono rimanere **asincroni** ma continuare a condividere le informazioni e svolgere le attività.

Le applicazioni guidate dagli eventi possono essere create in qualsiasi linguaggio di programmazione perché l'essere "*event driven*" indica un approccio alla programmazione e non un linguaggio. L'architettura guidata dagli eventi consente un livello minimo di accoppiamento, ed è pertanto l'opzione ideale per le architetture applicative moderne e distribuite. Un'architettura guidata dagli eventi viene definita a basso livello di accoppiamento perché chi crea un evento (noto come **producer** eventi) non conosce né il destinatario dell'evento (noto come **consumer** eventi) né le conseguenze che derivano dal verificarsi dell'evento.

2.2.1 Message Queues (MQs)

Le **code di messaggi (MQ)** sono un modo per implementare flussi di lavoro asincroni. Sono componenti progettati per ricevere, conservare e recapitare messaggi. funzionano nel sueguente modo:

1. Un'applicazione pubblica un messaggio nella coda, contenente i dettagli dell'attività da eseguire.

2. Uno o più worker (consumer), in modo asincrono e potenzialmente in un momento successivo, prelevano il messaggio dalla coda ed eseguono l'operazione richiesta. L'utente non resta bloccato in attesa del completamento dell'attività: l'elaborazione avviene in background, quando uno dei worker disponibili è pronto a gestirla.

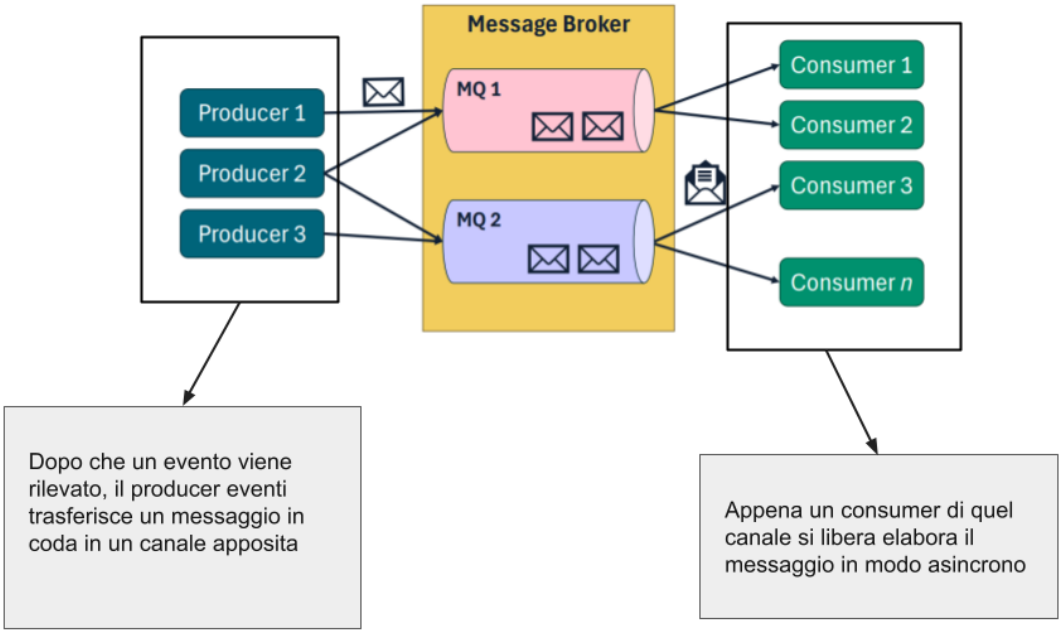


Figura 2.4: Schema di un MQ

2.3 Architettura a microservizi

L'architettura a **microservizi** si contrappone al modello **monolitico** tradizionale, nel quale tutti i componenti e moduli dell'applicazione sono integrati all'interno di un'unica unità software, generando una singola base di codice e un unico artefatto distribuibile. L'architettura monolitica ha l'indubbio vantaggio dell'elevato livello di comunicazione che intercorre tra i suoi componenti ma, per contro, risulta difficile da aggiornare e gestire in maniera agile, come richiesto dalle metodologie di sviluppo di moderna concezione, come DevOps, che ormai costituiscono uno standard sempre meno differibile.

In un'architettura a microservizi, al contrario, l'applicazione è suddivisa in un insieme di servizi indipendenti, ciascuno con un proprio ciclo di vita, le proprie responsabilità e, spesso, il proprio database. Ogni servizio può essere distribuito, scalato e aggiornato in modo autonomo. Questo approccio si adatta particolarmente bene ai sistemi moderni, caratterizzati da esigenze di evoluzione continua, alta disponibilità e rapidità di rilasci.

I vantaggi più significativi dell'architettura a microservizi includono la **scalabilità** (sia verticale che orizzontale, mirata ai singoli servizi) e il **basso accoppiamento**: un malfunzionamento di un microservizio non compromette necessariamente l'intero sistema, aumentando così la resilienza complessiva.

Tuttavia, è importante considerare che l'adozione dei microservizi introduce anche una com-

plessità architetturale non trascurabile: gestione delle comunicazioni tra servizi, osservabilità, orchestrazione, bilanciamento del carico, sicurezza distribuita e maggiore necessità di competenze tecniche avanzate. Per questo motivo, quando il sistema da sviluppare non è particolarmente complesso, l'architettura monolitica può rappresentare una scelta più appropriata. Un monolite è infatti più semplice da sviluppare, distribuire e mantenere, richiede un team più ridotto ed è generalmente più economico nella fase iniziale del progetto.

2.3.1 Distribuzione attraverso i container

I **container** rappresentano una tecnologia particolarmente adatta la distribuzione di applicazioni basate su microservizi. Un container può essere visto come un ambiente di esecuzione leggero e isolato, che include tutto il necessario per far funzionare un'applicazione: codice, dipendenze, librerie, file binari e configurazioni. A differenza delle macchine virtuali tradizionali, i container non richiedono di virtualizzare l'intero hardware, ma condividono il kernel del sistema operativo, risultando così più leggeri, rapidi da avviare e meno onerosi in termini di risorse.

Questo elevato livello di isolamento consente di eseguire più istanze dello stesso servizio senza dover riconfigurare l'ambiente a ogni avvio, rendendo semplice replicare componenti, scalare e garantire un comportamento uniforme in qualsiasi contesto. Inoltre, la possibilità di definire in modo preciso e ripetibile l'ambiente applicativo semplifica notevolmente i cicli di sviluppo, test e rilascio, riducendo tempi e costi.

Utilizzare i container per i microservizi significa migliorare la portabilità delle applicazioni e rendere il deployment indipendente dall'infrastruttura: lo stesso container può essere eseguito on-premise, su cloud privato, pubblico o in ambienti ibridi senza differenze. Questo approccio si integra perfettamente con l'esigenza dei microservizi di essere distribuiti, isolati e facilmente scalabili.

2.4 Modelli di database

Indipendentemente dal tipo di sistema da progettare e sviluppare, nella maggior parte dei casi è necessaria la possibilità di salvare e gestire dati. I database sono progettati per facilitare l'accesso, la persistenza, la gestione e l'aggiornamento delle informazioni, mentre i **DBMS (Database Management System)** sono i software che ne permettono l'utilizzo. Prima di selezionare un DBMS, una delle prime scelte progettuali riguarda il modello di database da adottare. I modelli più diffusi sono due: **relazionale** e **non relazionale (NoSQL)**.

- **Database relazionali (SQL):** Un database relazionale organizza i dati in **tabelle** composte da righe (record) e colonne (attributi). Una caratteristica distintiva di questo modello è la possibilità di stabilire relazioni tra tabelle diverse tramite l'uso di **chiavi primarie** (identificatori univoci dei record all'interno di una tabella) e **chiavi esterne**: attributi che fanno riferimento alla chiave primaria di un'altra tabella. Queste chiavi permettono di modellare relazioni di tipo uno-a-uno, uno-a-molti, molti-a-uno e molti-a-molti, garantendo una struttura dati coerente e ben definita. L'interazione con i database relazionali avviene tramite **SQL (Structured Query Language)**, un linguaggio dichiarativo che consente di creare, leggere, aggiornare ed eliminare i dati (operazioni **CRUD**). I database relazionali seguono il modello **ACID (Atomicity, Consistency, Isolation, Durability)**, che garantisce affidabilità, consistenza e sicurezza nelle transazioni. Sono quindi particolarmente adatti in contesti dove l'integrità del dato è fondamentale, come nei sistemi finanziari o in applicazioni fortemente transazionali.

- **Database non relazionali (NoSQL):** I database non relazionali, o NoSQL (Not Only SQL), non utilizzano la tradizionale struttura tabellare dei sistemi relazionali. Al contrario, si basano su modelli di dati differenti, come: **Key-value, Document-oriented**, **Column-family**, **Graph**. Questi modelli consentono una gestione molto più flessibile dei dati, risultando ideali per informazioni **non strutturate** o **semistrutturate**, o per dataset di grandi dimensioni che cambiano frequentemente.

La scelta tra un database SQL e uno NoSQL dipende principalmente dal tipo di dati da gestire e dai requisiti del sistema. I database relazionali rappresentano la soluzione ideale quando le informazioni hanno una struttura stabile e prevedibile, quando le transazioni devono rispettare rigorosamente le proprietà ACID e quando l'integrità del dato riveste un ruolo centrale. Sono inoltre più adatti nei contesti in cui esistono molte relazioni complesse tra entità diverse, poiché il modello relazionale è pensato proprio per gestire questi legami in modo coerente e sicuro.

I database NoSQL risultano invece più adeguati quando si lavora con dati non strutturati, semistrutturati o soggetti a frequenti cambiamenti. In scenari che richiedono una forte scalabilità orizzontale, come applicazioni distribuite, sistemi di analisi in tempo reale, IoT o gestione di grandi volumi di dati, i database non relazionali offrono prestazioni più elevate e una maggiore flessibilità. L'assenza di uno schema rigido consente di adattare rapidamente il modello dei dati all'evoluzione del progetto, rendendo questo approccio preferibile nei sistemi che devono crescere rapidamente o elaborare flussi continui di informazioni eterogenee.

Capitolo 3

Specifica dei requisiti

Sulla base dello studio delle moderne metodologie e architetture utilizzate nell'ingegneria del software, esposto nella prima parte di questa tesi, questa seconda parte introduce la progettazione e lo sviluppo del progetto realizzato che applica tali metodologie per affrontare una problematica reale che coinvolge attività e liberi professionisti: la mancata presentazione del cliente (no-show).

Questa sezione si apre con il capitolo dedicato alla Specifica dei requisiti. All'interno del ciclo di vita di un software, questa fase preliminare è fondamentale e ha l'obiettivo di identificare, analizzare e documentare in modo sistematico le esigenze e le funzionalità attese dal sistema. Tali informazioni costituiscono la base per le successive attività di progettazione e implementazione.

I requisiti sono stati definiti a seguito di un'approfondita analisi del dominio applicativo, che ha portato alla formulazione dei seguenti obiettivi generali del sistema:

- **Introdurre meccanismi di riduzione dei no-show**, mantenendo un approccio compatibile con contesti caratterizzati da elevata concorrenza e bassa propensione degli utenti ad accettare vincoli troppo rigidi.

- A**utomatizzare l'invio di promemoria** tramite un meccanismo di conferma attiva da parte dell'utente, al fine di diminuire le assenze non comunicate e ridurre il carico operativo del professionista o dell'attività.

- **Automatizzare la gestione delle prenotazioni tramite code dinamiche**, capaci di sostituire automaticamente le prenotazioni annullate o non confermate, mitigando così l'impatto economico delle mancate presentazioni.

Dagli obiettivi generici sono poi stati definiti i requisiti specifici che sono classificati in due categorie principali:

Requisiti funzionali (RF).

Descrivono i comportamenti attesi dal sistema e definiscono con precisione le operazioni che esso deve essere in grado di svolgere. Ogni requisito funzionale specifica un'interazione, un flusso operativo o un output previsto in risposta a determinati input o condizioni.

Requisiti non funzionali (RNF).

Riguardano le proprietà qualitative e i vincoli che il sistema deve rispettare, quali prestazioni, scalabilità, affidabilità, sicurezza e usabilità. Essi stabiliscono le condizioni entro cui le funzionalità devono essere erogate e contribuiscono alla qualità complessiva della soluzione.

3.1 Requisiti funzionali

RF-1 — Gestione dell'autenticazione

Il sistema deve prevedere un meccanismo di autenticazione completo, comprendente registrazione e login. Ogni utente registrato deve essere associato a una singola attività, poiché tutte le operazioni svolte all'interno dell'applicazione fanno riferimento all'attività collegata all'account.

All'avvio dell'applicazione, il sistema verifica lo stato di autenticazione tramite uno stato globale condiviso. Se l'utente non è autenticato, viene automaticamente reindirizzato alla schermata di accesso, dalla quale può eseguire login o registrazione.

RF-2 — Visualizzazione del calendario e delle prenotazioni

L'utente autenticato deve poter visualizzare un **calendario interattivo**, con possibilità di modificare la modalità di visualizzazione (giorno, settimana, mese). Accanto al calendario deve essere presente una **lista delle prenotazioni** relative alla data selezionata. La lista deve permettere filtri aggiuntivi, tra cui:

- email del cliente prenotato
- nome e cognome del cliente
- stato della prenotazione

RF-3 — Visualizzazione dei dettagli di una prenotazione

Interagendo con il calendario o con la lista, l'utente deve poter accedere ai dettagli di una prenotazione, visualizzando tutte le informazioni a essa associate.

RF-4 — Gestione delle prenotazioni

Ogni prenotazione deve contenere le seguenti informazioni:

1. **Dati della prenotazione**
    - titolo dell'evento
    - data e slot oraria
    - stato della prenotazione
    - data di inizio invio dei promemoria
    - frequenza dei promemoria (in giorni)
    - tempo massimo per confermare ciascun promemoria
2. **Dati del cliente**
    - nome
    - cognome
    - email
    - telefono

La slot oraria è rappresentata da un intero compreso tra 0 e 11.

Lo stato di una prenotazione può assumere i valori:

- **CONFERMATO**: prenotazione attiva e valida.

- **RISERVA**: prenotazione inserita in lista d'attesa, confermabile solo se si libera un posto nella stessa slot oraria.
- **ANNULLATA**: prenotazione precedentemente confermata che viene annullata

L'utente deve poter inserire, modificare e cancellare qualsiasi prenotazione.

RF-5 — Configurazione delle preferenze dell'utente

Il sistema deve offrire all'utente una sezione dedicata alla configurazione delle proprie preferenze operative, tra cui:

- tempo massimo concesso per confermare una prenotazione
- numero massimo di prenotazioni confermate per slot oraria
- numero massimo di utenti ammessi in lista di riserva per ciascuna slot
- numero di giorni di anticipo rispetto alla data della prenotazione per l'invio dei promemoria
- frequenza dei promemoria (in giorni)

RF-6 — Meccanismo automatico di riallocazione

Il sistema deve automaticamente gestire il processo di invio dei promemoria e la riallocazione delle prenotazioni sulla base delle conferme ricevute.

Ogni giorno, il sistema individua le prenotazioni per cui è necessario inviare un promemoria, secondo le seguenti condizioni:

- la data della prenotazione non deve essere già trascorsa;
- la data corrente deve essere maggiore o uguale alla data di inizio dei promemoria;
- la data corrente deve rispettare la frequenza configurata per l'invio di promemoria;

Per ciascuna prenotazione individuata, il sistema invia tramite email un link che l'utente deve utilizzare per confermare la propria presenza entro la scadenza indicata.

Alla scadenza del tempo di conferma, se il link è stato cliccato il sistema non prosegue altrimenti:

1. la prenotazione passa dallo stato **CONFERMATO a ANNULLATO**.
2. Il sistema preleva la prima prenotazione in stato **RISERVA** relativa alla stessa data e slot.
3. Invia un'email di notifica al cliente in lista d'attesa, invitandolo a confermare la disponibilità entro il tempo stabilito.
4. Se il cliente conferma, la prenotazione passa allo stato **CONFERMATO**.
5. In caso contrario, il sistema procede iterando sulla coda fino ad assegnare la slot disponibile o fino a esaurimento della coda.

3.2 Requisiti non funzionali

Dato il tempo limitato del tirocinio, sono stati raccolti i requisiti non funzionali ritenuti più essenziali per gli obiettivi e il contesto.

RNF-1 — Sicurezza relativa ad autenticazione e autorizzazione

Il sistema deve garantire un adeguato livello di sicurezza nella gestione dei processi di autenticazione e autorizzazione.

RNF-2 — Scalabilità infrastrutturale

Il sistema deve essere progettato per supportare una crescita futura in termini di volume di utenti, traffico e funzionalità. L'architettura deve quindi poter essere estesa senza richiedere modifiche strutturali significative.

RNF-3 — Interfaccia user-friendly

L'interfaccia utente deve risultare intuitiva e facilmente navigabile. Le azioni principali devono essere chiaramente identificabili e raggiungibili, minimizzando il numero di passaggi necessari per completare le operazioni più comuni.

Capitolo 4

Design del sistema

In questo capitolo vengono illustrate le scelte progettuali adottate per la realizzazione del sistema descritto nel capitolo precedente.

Considerati i vantaggi consolidati dell'architettura Client-Server, evidenziati nello studio preliminare, il sistema è stato progettato seguendo proprio questo modello. L'architettura Client-Server permette di separare chiaramente le responsabilità tra client e server, facilitando la manutenzione, l'estensibilità e la scalabilità del sistema.

Poiché l'analisi dei requisiti non ha evidenziato necessità particolari come la gestione di dati con aggregazioni complesse o la realizzazione di sistemi ad alte prestazioni critiche, si è scelto di adottare lo **stile architetturale REST API** per la comunicazione tra **backend** e **frontend**. Questo approccio, oltre a essere lo standard più diffuso per progetti di complessità ordinaria, garantisce interoperabilità, semplicità di implementazione e facilità di integrazione con eventuali futuri servizi esterni.

Nelle sezioni seguenti viene approfondita la progettazione dei due principali componenti del sistema.

4.1 Backend

Sebbene i requisiti definiti per il sistema realizzato descrivano funzionalità relativamente semplici e potenzialmente gestibili con un approccio monolitico, si è scelto di adottare sin da subito una **struttura a microservizi**. Tale scelta è motivata dall'esigenza di consentire al sistema di evolversi nel tempo, integrando nuove funzionalità e gestendo un volume crescente di utenti.

L'adozione di un'architettura basata su componenti separati garantisce proprietà quali scalabilità e manutenibilità, perfettamente coerenti con i requisiti non funzionali precedentemente definiti.

Tra le prime decisioni da prendere nello sviluppo del backend vi è stata quella relativa alla scelta del linguaggio di programmazione per l'implementazione della REST API e anche la valutazione se realizzarla da zero o sfruttare un framework. Poiché, come evidenziato nella prima parte di questa tesi, l'utilizzo di un framework semplifica e accelera lo sviluppo di una REST API, e considerando che la rigidità imposta dai framework non interferiva con la progettazione dei requisiti individuati, si è deciso di adottare quest'ultima soluzione.

L'implementazione dei servizi che compongono l'API è stata realizzata utilizzando il framework **Spring Boot**. La scelta è motivata da una duplice ragione. Da un lato, si è voluto mantenere coerenza con le tecnologie adottate dall'azienda ospitante, nella quale Spring Boot rappresenta lo standard per lo sviluppo di applicazioni backend. Dall'altro, il framework offre caratteristiche particolarmente adatte alle esigenze del progetto mettendo a disposizione moduli consolidati come **Spring Web** per la creazione di API REST, **Spring Data** per l'accesso ai dati e **Spring Security** per la gestione dell'autenticazione.

La suddivisione logica segue lo schema tipico dell'architettura Spring:

- **Entità (Entity):** rappresentano gli oggetti di dominio del sistema, generalmente mappati sulle tabelle del database.
- **Repository:** gestiscono l'accesso ai dati, interfacciandosi con il database per eseguire le operazioni CRUD (Create, Read, Update, Delete).
- **Servizi (Service):** contengono la logica di business e orchestrano l'interazione tra entità e risorse esterne.
- **Controller:** espongono gli endpoint REST, gestendo le richieste HTTP provenienti dal frontend o da altri sistemi, invocando i servizi appropriati e restituendo le risposte al client.
- **DTO (Data Transfer Object):** rappresentano i dati da trasferire tra client e server. Essi non contengono logica, ma servono a garantire sicurezza e coerenza nelle comunicazioni, separando il livello di persistenza da quello di presentazione.

Di seguito è riportata una figura che mostra l'interazione tra questi moduli:

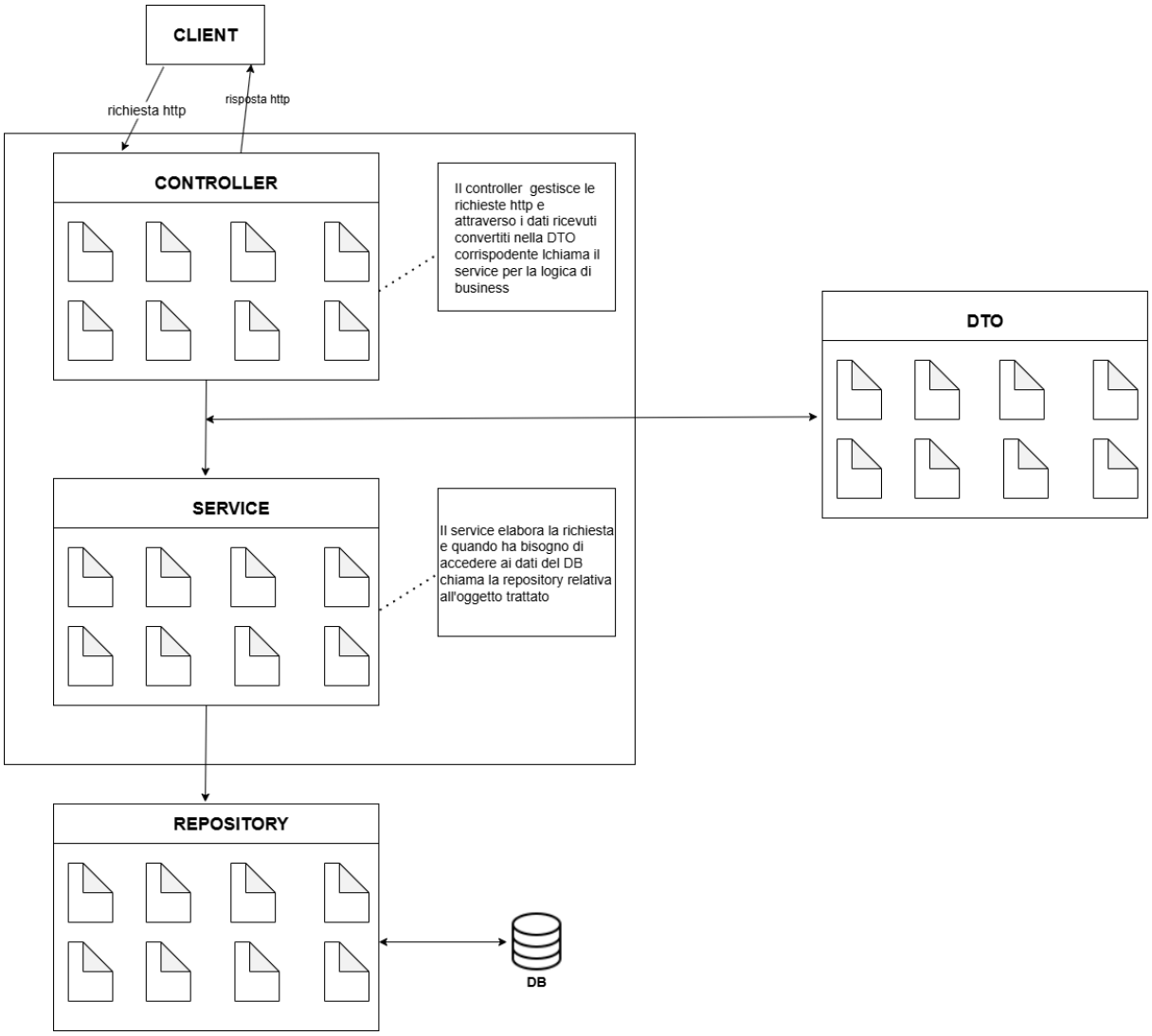


Figura 4.1: Architettura Spring uilizzata

4.1.1 Microservizi implementati

L'API è stata suddivisa in più microservizi con responsabilità ben definite:

- **Auth:** responsabile dell'autenticazione e della gestione degli utenti. Supporta funzionalità di registrazione e login, interagendo con il database tramite le entità Utente e Attività. Durante il processo di login, il microservizio genera un token **JWT**[3] firmato, contenente un claim con l'ID dell'attività associata all'utente. Questo meccanismo permette agli altri microservizi di validare le operazioni rispetto all'attività corretta, incrementando la sicurezza e prevenendo accessi non autorizzati.
- **Booking:** rappresenta il cuore delle operazioni di prenotazione. Gestisce la creazione, modifica, recupero e cancellazione delle prenotazioni, oltre alla conferma della presenza

del cliente. Comunica inoltre con il microservizio Booking-Messages tramite **Apache Kafka**, trasmettendo informazioni come indirizzo email e tipologia di notifica da inviare.

- **Booking-Messages:** dedicato alla gestione delle notifiche email. È in ascolto degli eventi generati dal microservizio Booking e invia email personalizzate tramite template HTML dinamici. Ogni messaggio include un token firmato con scadenza, che incorpora l'ID della prenotazione e viene utilizzato come path per un endpoint esposto dal microservizio Booking. Quest'ultimo si occupa di verificarne la validità, garantendo la sicurezza del flusso di comunicazione.

Oltre ai microservizi core, sono stati introdotti altri servizi per la gestione e il coordinamento del sistema:

- **Eureka:** fornisce il servizio di discovery centralizzato, registrando dinamicamente i microservizi attivi e consentendo il bilanciamento delle richieste.
- **Config:** gestisce la configurazione centralizzata dei vari microservizi, garantendo coerenza e semplificando la manutenzione dei parametri di sistema.
- **Gateway:** costituisce il punto di ingresso unico del sistema. Instrada le richieste verso i microservizi API corretti tramite routing interno, applicando controlli preliminari su sicurezza e **CORS** (Cross-Origin Resource Sharing). In questo modo semplifica l'integrazione tra frontend e backend.
- **Apache Kafka:** utilizzato come broker di messaggistica per la comunicazione asincrona tra microservizi, in particolare per la gestione delle notifiche in tempo reale.
- **PostgreSQL:** sistema di gestione del database relazionale scelto per la persistenza dei dati.

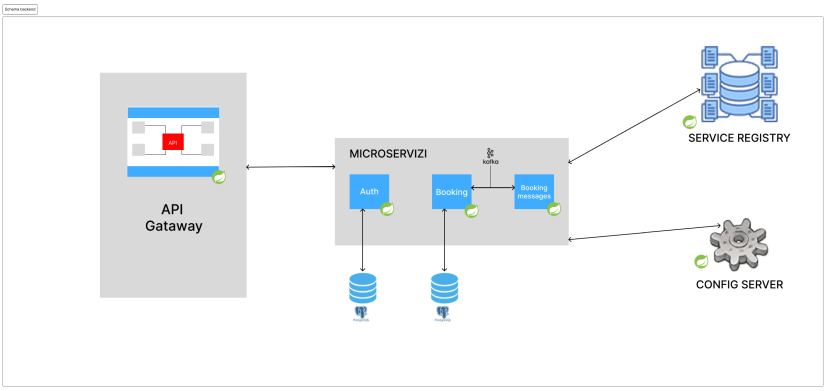


Figura 4.2: Schema microservizi backend

Containerizzazione dei microservizi

In continuità con quanto analizzato nello studio precedente, la containerizzazione dei microservizi consente di ottenere un ambiente di esecuzione indipendente dall'infrastruttura sottostante. Motivo per cui ogni microservizio realizzato viene eseguito all'interno di un container **Docker**, principale tecnologia nell'ambito dei container, che garantisce isolamento, portabilità e una distribuzione non vincolata alle specifiche della macchina ospitante. Questo approccio semplifica in modo significativo il **deployment** e favorisce la scalabilità dell'architettura complessiva.

4.1.2 Modellazione dei Flussi Asincroni per la Gestione delle Prenotazioni

In questa sezione vengono presentati alcuni **sequence diagram** che illustrano i flussi principali alla base dell'algoritmo di gestione automatica delle prenotazioni. I diagrammi sono stati suddivisi in più parti per maggiore leggibilità, ma possono essere interpretati come segmenti consecutivi di un unico flusso completo.

La scelta di includere questi diagrammi nasce dalla volontà di evidenziare in modo chiaro e formale gli elementi tecnici e architetturali che caratterizzano il sistema, in particolare:

- l'utilizzo della tecnologia **JPA + Hibernate** di Spring, che consente la generazione automatica di query tramite metodi derivati dalla struttura delle entità;

- la gestione delle **comunicazioni asincrone** tra microservizi tramite invio di messaggi su topic dedicati;

- la comprensione complessiva della progettazione dell'algoritmo centrale che rappresenta il fulcro funzionale dell'intero sistema.

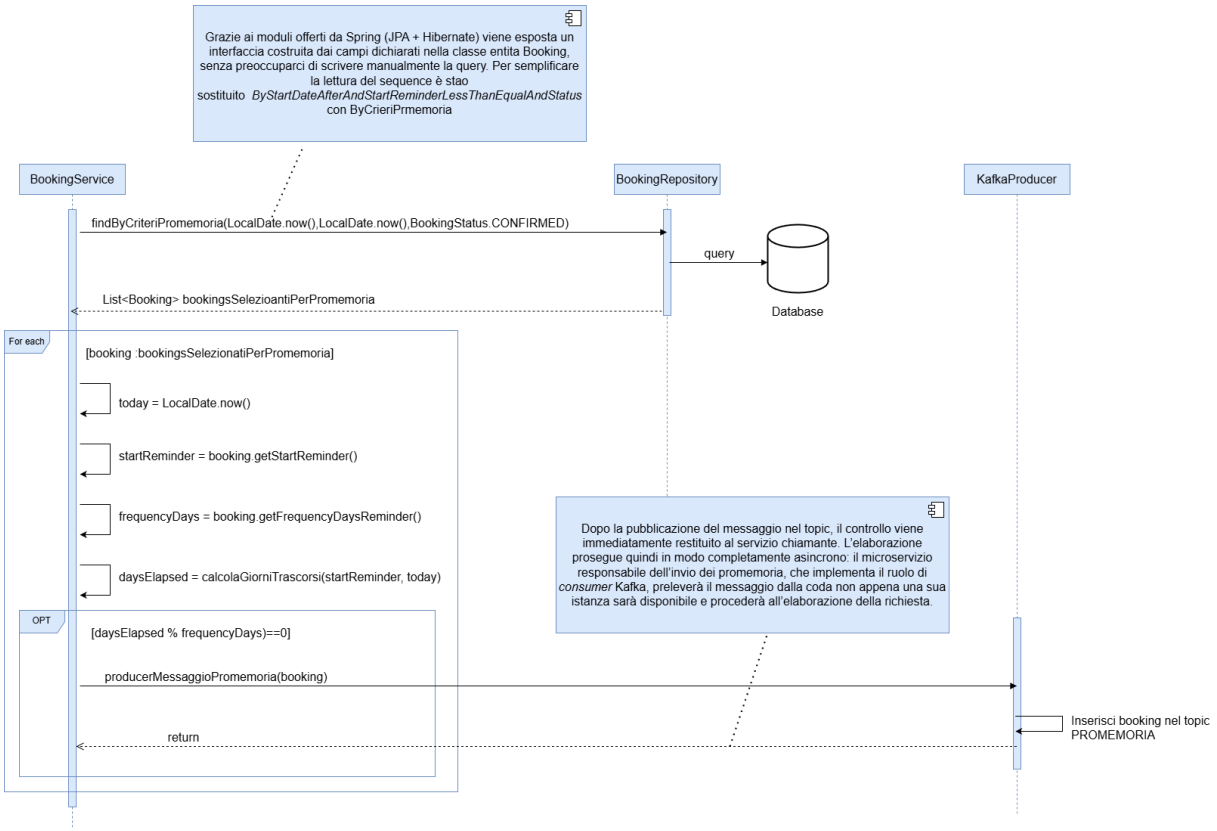


Figura 4.3: Sequence Diagram scenario recupero prenotazioni a cui inviare promemoria - microservizio booking

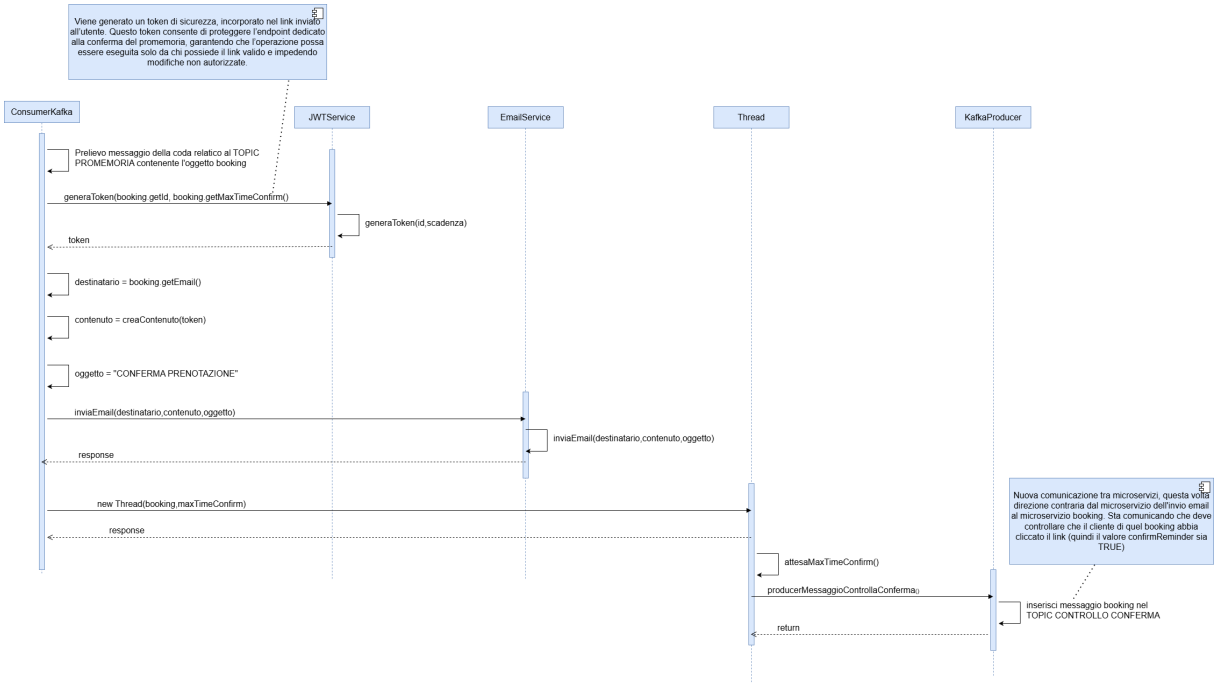


Figura 4.4: Sequence Diagram scenario invio email di un promemoria - microservizio email-Promemoria

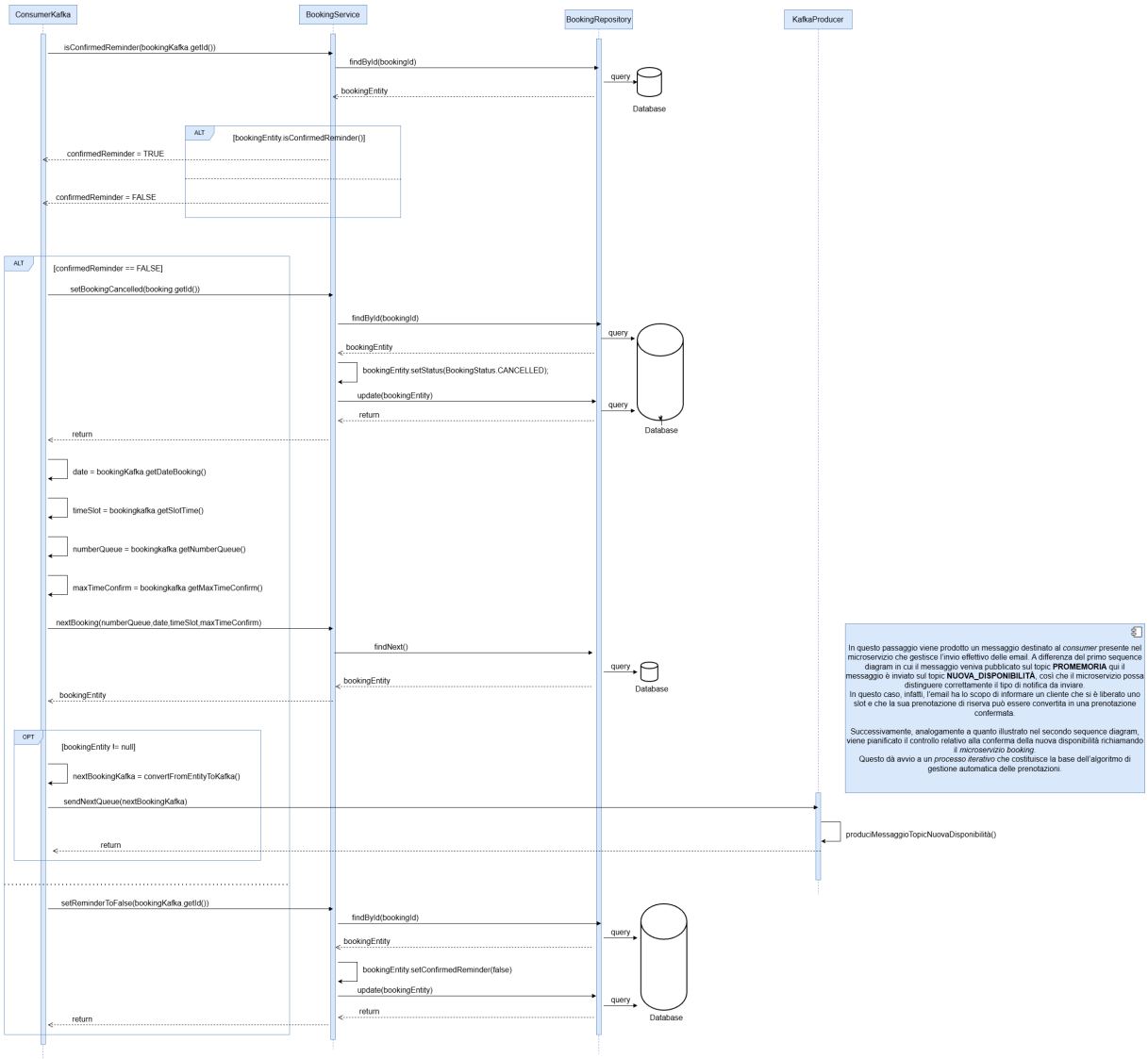


Figura 4.5: Sequence Diagram scenario controllo conferma del promemoria - microservizio booking

4.2 Frontend

Il **frontend** è stato sviluppato utilizzando il framework **Angular**. Anche in questo caso, la scelta tecnologica è stata influenzata dall'ambiente aziendale ospitante, dove Angular rappresenta lo standard per lo sviluppo delle interfacce utente web.

Per ottimizzare l'esperienza d'uso su dispositivi mobili, è stato integrato il framework **Ionic**. L'utilizzo di Ionic ha permesso di progettare l'interfaccia utente adottando un approccio orientato al mobile, beneficiando al tempo stesso dell'ecosistema di Angular.

Un ulteriore vantaggio dell'impiego di Ionic risiede nella possibilità di sviluppare l'interfaccia

come una normale applicazione web, sfruttando pienamente le potenzialità di Angular, e convertire lo stesso codice in un'applicazione mobile nativa. Questo approccio rientra nell'architettura delle **Progressive Web App (PWA)**.

Il codice sorgente dell'applicazione è organizzato all'interno della cartella `src/app`, mentre gli altri file del progetto includono configurazioni, risorse e dipendenze necessarie al funzionamento del framework.

Angular si basa su un'architettura a **componenti**, ognuno dei quali è solitamente costituito da tre file distinti:

- un file `.html` che definisce il template del componente (struttura e contenuto visivo);
- un file `.scss` che definisce lo stile del componente;
- un file `.ts` che contiene la logica del componente, scritta in TypeScript.

La struttura del progetto frontend è stata organizzata come segue:

- **App Component:** posto alla radice dell'applicazione, rappresenta il punto di ingresso principale dell'interfaccia e avvia l'inizializzazione del frontend.
- **Pages:** contiene i componenti che rappresentano le pagine principali dell'applicazione. Nel caso di Tapp4Sit, la navigazione avviene principalmente tramite modali, pertanto l'unica pagina esplicita è la Home.
- **Components:** raccoglie i componenti riutilizzabili, ossia elementi grafici autonomi che possono essere integrati in più contesti.
- **Services:** include i file TypeScript che implementano la logica di servizio, come la comunicazione con il backend tramite chiamate API REST.
- **Models:** contiene le interfacce TypeScript utilizzate nel progetto, che garantiscono robustezza e tipizzazione sicura dei dati.
- **Entities:** comprende le classi TypeScript che rappresentano le entità del dominio. Esse sono utilizzate in combinazione con l'ORM locale per supportare l'applicazione anche in modalità offline, migliorando l'affidabilità in condizioni di connessione instabile.
- **DTO (Data Transfer Object):** rappresentano gli oggetti scambiati con il backend. Ogni DTO ha un corrispettivo omonimo lato server, assicurando coerenza nella comunicazione client–server.

- **Config:** contiene i file di configurazione del frontend, tra cui quelli necessari alla gestione delle richieste HTTP verso il backend.

Capitolo 5

Validazione del sistema

La fase di **verifica e validazione** rappresenta l'ultimo passo prima del rilascio del software ed è una delle più importanti all'interno del processo di sviluppo. L'esperienza insegna che, una volta completata la scrittura del codice, emergono quasi sempre bug o comportamenti imprevisti.

Il termine *bug* (letteralmente "insetto") venne utilizzato per la prima volta nel 1945, quando una **falena** causò un cortocircuito nel pannello F, Relay #70 del computer **Harvard Mark II**. L'insetto fu rimosso e incollato sul logbook della macchina con la nota "primo caso di bug trovato". Da allora, il termine è rimasto nella terminologia informatica per indicare un malfunzionamento del sistema.

Oggi l'attività di **validazione** è fondamentale per garantire l'affidabilità del software. Per quanto rigoroso possa essere lo sviluppo, nessun processo di testing può assicurare l'assenza totale di difetti; tuttavia, una buona attività di validazione consente di ridurne drasticamente l'impatto.

Il **testing** costituisce la parte principale (ma non l'unica) della validazione. In questo capitolo vengono descritti i test effettuati sul sistema e i relativi risultati. Il testing consiste nel confrontare il comportamento **atteso** (definito dai requisiti e dai modelli del sistema) con il comportamento **osservato** dell'implementazione, con l'obiettivo di individuare eventuali discrepanze e correggere gli errori rilevati.

5.1 Obiettivo del capitolo

Nella letteratura dell'ingegneria del software esistono diverse tipologie di test, generalmente classificate in due macro-categorie: **test funzionali** e **test non funzionali**. I primi verificano

che il sistema soddisfi i requisiti funzionali emersi durante l'analisi, mentre i secondi valutano caratteristiche qualitative come prestazioni, usabilità, sicurezza o affidabilità. Tra i test funzionali più comuni rientrano **test di unità**, **test di integrazione** e **test di sistema**; tra quelli non funzionali, ad esempio, i **test di usabilità** o di **performance**.

Questo capitolo non intende coprire in modo esaustivo tutte le tipologie di test riconosciute in letteratura, ma fornisce una panoramica delle tecniche applicate al *backend* del sistema sviluppato, concentrandosi sulle sue due aree funzionali principali:

- **Autenticazione**, che comprende la registrazione e l'accesso degli utenti.
- **Gestione delle prenotazioni**, che include la creazione di prenotazioni, l'invio dei promemoria e il meccanismo automatico di gestione delle prenotazioni di riserva.

La prima parte del capitolo è dedicata ai **test di unità**, mirati a verificare il comportamento di singoli metodi o componenti isolati dal resto del sistema. In questa sezione vengono approfonditi due aspetti fondamentali del testing:

1. **Tecnologie per il testing automatico**: Vengono introdotti **JUnit** (per le asserzioni sulla correttezza dei risultati) e **Mockito** (per simulare dipendenze esterne come repository e servizi). Questo consente di isolare l'unità testata e analizzare il comportamento del codice in modo controllato. Ogni caso di test riportato sarà accompagnato dal relativo codice JUnit, così da illustrare concretamente il suo funzionamento.
2. **Strategie di progettazione dei test**: Poiché i possibili casi di test sono, in generale, infiniti, è necessario selezionare un insieme significativo ma finito di input. Viene quindi mostrato come applicare tecniche classiche di testing come il partizionamento in classi di equivalenza o criteri di copertura per definire una test suite adeguata.

La seconda parte del capitolo ha l'obiettivo di testare manualmente il sistema nel suo insieme, simulando scenari realistici. Questa tipologia di test si ispira ai **casi di test a livello di sistema** descritti in Software Testing[4, sec. 14.4.3].

Un singolo caso di test viene suddiviso in sottotest sequenziali, ciascuno dei quali rappresenta un passo dello scenario. Il caso di test complessivo si considera superato solamente se tutti i sottotest vengono completati con successo seguendo l'ordine previsto. Di seguito viene riportata una tabella di esempio, tratta dal libro, che illustra questo approccio:

Test case name	Correct PIN entry on first try
Test case ID	TC-1
Description	A customer enters the PIN number correctly on the first attempt.
Preconditions needed to run this test case	1. The expected PIN is "2468"
	2. Screen 2 is displayed
Event Sequence	
Input events (performed by tester)	*Output events (observed by system tester)*
	1. Screen 2 shows '- - - - '
2. Touch digit 2	
	3. Screen 2 shows '- - - * '
4. Customer touches digit 4	
	5. Screen 2 shows '- - * * '
6. Customer touches digit 6	
	7. Screen 2 shows '- * * * '
8. Customer touches digit 8	
	9. Screen 2 shows '- - - * '
10. Customer touches Enter	
	11. Screen 5 is displayed
Cross reference to functions	
Postconditions	Select Transaction is active
Test execution result?	Pass/Fail
Test run by	<tester's name> date

Figura 5.1: tabella di esempio casi di test a livello di sistema

Per eseguire questi test, viene utilizzata la tecnologia **Postman**, che consente di effettuare chiamate HTTP agli endpoint dell'API specificando metodo, header e body, simulando così le richieste generate dal frontend.

5.2 Test unitari

I **test unitari** hanno lo scopo di verificare il corretto funzionamento delle singole unità di codice, tipicamente metodi o classi, in modo isolato dal resto dell'applicazione. Essi rappresentano la base del processo di validazione e consentono di individuare rapidamente errori logici o regressioni nelle componenti fondamentali del sistema.

Nel testing unitario si distinguono due principali approcci metodologici:

- **Testing black-box**: la definizione dei casi di test si basa sui **requisiti funzionali** dell'unità da verificare. In questa strategia non si analizza la struttura interna del codice, ma ci si concentra sugli input e output possibili. L'obiettivo è coprire il maggior numero di scenari rappresentativi del dominio di input, tramite tecniche come le **classi di equivalenza** e l'**analisi dei valori limite**.
- **Testing white-box**: La definizione dei casi di test si fonda sugli aspetti **strutturali del codice** dell'unità. In questo caso è necessario conoscere l'implementazione interna, con

l'obiettivo di ottenere una copertura elevata delle istruzioni o dei rami decisionali del codice (ad esempio *statement e branch coverage*).

Per l'esecuzione dei test unitari sono stati utilizzati i framework **JUnit 5** e **Mockito**. JUnit consente di definire ed eseguire in modo automatizzato i test, verificando la correttezza dei risultati mediante asserzioni, mentre Mockito permette di simulare (mockare) dipendenze esterne come repository o servizi, isolando così la logica dell'unità testata.

5.2.1 Autenticazione

I metodi scelti da testare relativi alla logica di autenticazione sono:

public TemporaryUserEntity temporarySignup(SignUpRequest request):

Il metodo appartiene alla classe AuthService del microservizio auth. Viene invocato nella prima fase del processo di registrazione: se l'operazione va a buon fine, viene creato un utente temporaneo nella relativa tabella del database e gli viene associato un codice casuale di 4 cifre. Tale codice viene inviato all'indirizzo email fornito in fase di registrazione e deve essere inserito dall'utente per confermare la propria email.

In questo metodo si vuole verificare che, per determinate combinazioni di parametri in input, vengano generati errori, mentre per input validi l'operazione vada a buon fine. Il metodo accetta come parametri email, password e nome dell'attività. Il contratto di questo metodo stabilisce che:

- l'email deve rispettare un pattern valido (ad esempio test@dominio.com);
- la password deve rispettare i comuni criteri di sicurezza (almeno una maiuscola, una cifra e lunghezza minima di 8 caratteri);
- nessuno dei tre parametri può essere vuoto e il nome dell'attività deve essere lungo almeno 3 caratteri.

Poiché per questo metodo non è necessario conoscere o analizzare la struttura interna per lo scopo stabilito, ma interessa soltanto verificare che a input validi corrisponda un comportamento corretto e a input non validi corrisponda un errore, è stata adottata una strategia di testing **black-box**. In questo tipo di strategia l'attenzione è posta esclusivamente sugli input forniti al

metodo e sulle risposte prodotte, senza considerare le logiche interne della sua implementazione.

Coerentemente con l'approccio black-box, il dominio di ciascun parametro è stato suddiviso in classi di equivalenza. Più valori appartengono alla stessa classe se producono lo stesso comportamento atteso da parte del metodo. Questa suddivisione consente di selezionare un numero ridotto ma rappresentativo di valori: infatti, un singolo valore appartenente a una classe è sufficiente per verificare il comportamento di altri valori della stessa classe, in quanto rappresenta tutti gli altri valori possibili della stessa classe. Di seguito sono riportate, per ciascun parametro, le tabelle che descrivono le **classi di equivalenza** individuate ai fini della progettazione dei test.

ID Classe	Descrizione	Valore esempio	Validità
E1	L'insieme delle stringhe che non rispettano il pattern email	email == "roberto@spena"	Non Valido
E2	L'insieme delle stringhe che rispettano il pattern email	email == "roberto@unina.it"	Valido

Tabella 5.1: Parametro email

ID Classe	Descrizione	Valore esempio	Validità
P1	L'insieme delle stringhe che non rispettano il pattern password impo-sto	password=="rob"	Non Valido
P2	L'insieme delle stringhe che rispettano il pattern password impostato	request.nome=="Roberto-98"	Valido

Tabella 5.2: Parametro password

ID Classe	Descrizione	Valore esempio	Validità
A1	L'insieme delle stringhe null/vuote	attività==””	Non Valido
A2	L'insieme delle stringhe con lunghezza minore di 3	attività==”r”	Non Valido
A3	L'insieme delle stringhe con lunghezza maggiore di 3	attività==”Roby Srl”	Valido

Tabella 5.3: Parametro attività

Per costruire una test suite efficace era necessario adottare una strategia che non si basasse sui valori di frontiera (come min, min+1, max, max-1, ecc.), poiché i parametri in ingresso (email, password, nome attività) **non presentano un ordinamento naturale**.

Una possibile strategia consiste nel combinare tutte le classi di equivalenza dei parametri (prodotto cartesiano). Questa soluzione permetterebbe di verificare ogni possibile combinazione fra valori appartenenti alle varie classi, ma risulta **poco efficiente**, poiché genera un numero molto elevato di casi di test, spesso sproporzionato rispetto ai benefici ottenuti. Infatti, **molte combinazioni non aggiungerebbero informazioni utili**, portando a ripetere test ridondanti senza aumentare in modo significativo la capacità di individuare difetti.

Un'alternativa più compatta consiste nel garantire che **ogni classe sia coperta da almeno un caso di test**, come previsto dalla strategia **N-WECT (Weak Normal Equivalence Class Testing)**[4, sec. 6.3.1]. Tuttavia, questo approccio non impone alcun vincolo sulle combinazioni tra classi valide e non valide: più parametri potrebbero assumere contemporaneamente valori non validi, rendendo **difficile l'individuazione della causa precisa di un eventuale errore**.

Per questo motivo è stata scelta la strategia **R-WECT (Robust Weak Equivalence Class Testing)**[4, sec. 6.3.3], che rappresenta una variante più robusta della N-WECT e opera come segue:

1. si testano tutte le classi di equivalenza valide di ciascun parametro;

2. si testano tutte le classi di equivalenza non valide, imponendo che in ogni caso di test **solo un parametro alla volta assuma un valore non valido**, mentre gli altri mantengono valori validi.

Si osservi che anche se il test è di tipo black-box e non si conosce l'implementazione interna del metodo, il tester può conoscere il comportamento atteso attraverso il **contratto**[5] del metodo, che definisce precondizioni, postcondizioni e eventuali eccezioni previste. Di conseguenza, sapere quale eccezione deve essere sollevata per input non validi non viola il principio del black-box, ma deriva dalla specifica funzionale del metodo.

Campo	**Descrizione**
ID	TC-1
Titolo	Email non valida
Obiettivo	Verificare che il metodo vada in eccezione in caso l'email non rispetta il pattern
Precondizioni	Dipendenze esterne mockate per garantire isolamento e indipendenza del test
Input	email="rob@"; password="Roberto-98"; attività="Roby Srl"
Output atteso	lancio di eccezione
Postcondizioni	Nessuna postcondizione rilevante

Tabella 5.4: Definizione caso test 1 - metodo temporarySignUp

```
1  //Caso Test 1: classi E1(Non valida); P2(Valida); A3(Valido)
2      @Test
3      public void testSaveTemporaryUser_InvalidEmail(){
4
5          String email = "rob@";
6          String password = "Roberto-98";
7          String attivita = "Roby Srl";
8
9          SignUpRequest request = SignUpRequest.builder()
10                 .email(email)
11                 .password(password)
12                 .attivita(attivita)
13                 .build();
14
15         when(userRepository.existsByEmail(email)).thenReturn(false);
16
17         Exception ex = assertThrows(BadRequestException.class, () ->
               authService.temporarySignup(request));
18
19         assertEquals(ex.getMessage(), "Email non valida");
20     }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Accorgersi che l'email non rispetta il pattern previsto	lancio dell'eccezione BadRequestException con email input non valido	17	✓
Leggere il messaggio dell'eccezione "	getMessage dell'eccezione lanciata è "Email non valida"	19	✓

Tabella 5.5: Descrizione del codice del test

Campo	Descrizione
ID	TC-2
Titolo	Password non valida
Obiettivo	Verificare che il metodo lancia l' eccezione in caso la password non rispetta il pattern
Precondizioni	Dipendenze esterne mockate per garantire isolamento e indipendenza del test
Input	email=”roberto@unina.it”; password=”rob”; attività=”Roby Srl”
Output atteso	lancio di eccezione
Postcondizioni	Nessuna postcondizione rilevante

Tabella 5.6: Definizione caso test 2 - metodo temporarySignUp

```java
//Caso Test 2: classi E2(Valida); P1(Non Valida); A3(Valido)
    @Test
    public void testSaveTemporaryUser_InvalidPassword(){

        String email = "roberto@unina.it";
        String password = "rob";
        String attivita = "Roby Srl";

        SignUpRequest request = SignUpRequest.builder()
                .email(email)
                .password(password)
                .attivita(attivita)
                .build();

        when(userRepository.existsByEmail(email)).thenReturn(false);

        Exception ex = assertThrows(BadRequestException.class, () ->
           authService.temporarySignup(request));

        assertEquals(ex.getMessage(), "Password non valida");
    }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Accorgersi che la password non rispetta il pattern previsto	lancio dell'eccezione BadRequestException con password input non valido	17	✓
Leggere il messaggio dell'eccezione "	getMessage dell'eccezione lanciata è "Password non valida"	19	✓

Tabella 5.7: Descrizione del codice del test

Campo	Descrizione
ID	TC-3
Titolo	Nome attività vuoto
Obiettivo	Verificare che il metodo vada in eccezione in caso il nome dell'attività e vuoto/null
Precondizioni	Dipendenze esterne mockate per garantire isolamento e indipendenza del test
Input	email="roberto@unina.it"; password="Roberto-98"; attività=null
Output atteso	lancio di eccezione
Postcondizioni	Nessuna postcondizione rilevante

Tabella 5.8: Definizione caso test 3 - metodo temporarySignUp

```java
//Caso Test 3: classi E2(Valida); P2(Valida); A1(Non Valido)
    @Test
    public void testSaveTemporaryUser3()_NullActivity{

        String email = "roberto@unina.it";
        String password = "Roberto-98";
        String attivita = null;

        SignUpRequest request = SignUpRequest.builder()
                .email(email)
                .password(password)
                .attivita(attivita)
                .build();

        when(userRepository.existsByEmail(email)).thenReturn(false);
        when(attivitaService.isExistingAttivita(attivita)).thenReturn
           (false);

        Exception ex = assertThrows(BadRequestException.class, () ->
           authService.temporarySignup(request));
        assertEquals(ex.getMessage(), "Activty non valida");
    }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Accorgersi che il nome dell'attività è nullo	lancio dell'eccezione BadRequestException con attività input null	18	✓
Leggere il messaggio dell'eccezione "	getMessage dell'eccezione lanciata è "Activity non valida"	19	✓

Tabella 5.9: Descrizione del codice del test

Campo	Descrizione
ID	TC-4
Titolo	Nome attività troppo corto
Obiettivo	Verificare che il metodo vada in eccezione in caso il nome dell'attività abbia meno di 3 caratteri
Precondizioni	Dipendenze esterne mockate per garantire isolamento e indipendenza del test
Input	email="roberto@unina.it"; password="Roberto-98"; attività="r"
Output atteso	lancio di eccezione
Postcondizioni	Nessuna postcondizione rilevante

Tabella 5.10: Definizione caso test 4 - metodo temporarySignUp

```
//Caso Test 4: classi E2(Valida); P2(Valida); A2(Non Valido)
    @Test
    public void testSaveTemporaryUser_InvalidLengthActivity(){

        String email = "roberto@unina.it";
        String password = "Roberto-98";
        String attivita = "r";

        SignUpRequest request = SignUpRequest.builder()
                .email(email)
                .password(password)
                .attivita(attivita)
                .build();

        when(userRepository.existsByEmail(email)).thenReturn(false);
        when(attivitaService.isExistingAttivita(attivita)).thenReturn
           (false);

        Exception ex = assertThrows(BadRequestException.class, () ->
           authService.temporarySignup(request));
        assertEquals(ex.getMessage(), "Activty non valida, inserire
           almeno un nome lungo 3 caratteri");
    }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Accorgersi che il nome dell'attività ha lunghezza non valida	lancio dell'eccezione BadRequestException con lunghezza 1 dell'attività input	18	✓
Leggere il messaggio dell'eccezione "	getMessage dell'eccezione lanciata è "Activity non valida, inserire almeno un nome lungo 3 caratteri"	19	✓

Tabella 5.11: Descrizione del codice del test

public String signUpConfirmed(SignUpRequest request):

Il metodo appartiene alla classe AuthService del microservizio auth. Viene invocato nella seconda fase del processo di registrazione, quando l'utente inserisce il codice ricevuto via email. Se la verifica ha esito positivo, l'utente viene definitivamente salvato nella tabella degli utenti registrati del database.

Poiché il metodo presenta diversi **flussi decisionali**, come il controllo della validità del codice e della sua eventuale scadenza, è stata adottata una **strategia di testing white-box**, con l'obiettivo di analizzare il comportamento interno del metodo e garantire una copertura adeguata dei possibili percorsi di esecuzione.

Prima di presentare la progettazione del test relativo al metodo signUpConfirmed, viene mostrato il codice del metodo in una forma semplificata rispetto all'implementazione originale, al fine di migliorarne la leggibilità. Si precisa inoltre che alcune porzioni di codice, nell'implementazione effettiva, risultano incorporate in altri metodi.

```
1  @Transactional
2      public String signUpConfirmed(SignUpRequest request){
3
4          TemporaryUserEntity temporaryUserEntity =
               temporaryUserRepository
5                      .findByEmail(request.getEmail());
6
7          if(temporaryUserEntity != null){
8
9              int code = temporaryUserEntity.getCode();
10
11             if(code != request.getConfirmationCode()){
```

```
12
13                  throw new BadRequestException("Codice errato");
14              }
15
16              LocalDateTime after15Minutes =
17                  temporaryUserEntity
18                      .getCreateDate().plusMinutes(15);
19
20              if(LocalDateTime.now().isAfter(after15Minutes)){
21
22                  throw new BadRequestException("Il tempo per la
                        conferma è scaduto");
23              }
24
25              temporaryUserRepository.deleteByEmail(request.getEmail())
                    ;
26
27          }else {
28
29              throw new BadRequestException("Email non registrata");
30          }
31
32          RoleEntity roleEntity = roleRepository.findByName(RoleName.
                ADMIN);
33
34          AttivitaEntity attivitaEntity = AttivitaEntity.builder()
35                  .name(request.getAttivita)
36                  .build();
37
38          UserEntity user = UserEntity.builder()
39                  .email(request.getEmail().toLowerCase())
40                  .password(passwordEncoder.encode(request.getPassword
                        ()))
41                  .role(roleEntity)
42                  .attivita(attivitaEntity)
43                  .build();
44
45          userRepository.save(user);
46
47          return "Registrazione completata con successo";
48      }
```

Di seguito è riportato il **Grafo del Flusso di Controllo (GFC) semplificato** che rappresenta la logica interna del metodo.



Figura 5.2: GFC del metodo signUpConfirmed

I test implementati sono stati concepiti in modo da verificare tutte le possibili decisioni rappresentate nel grafo di flusso del metodo.

Campo	Descrizione
ID	TC-5
Titolo	Email non registrata
Obiettivo	Copertura cammino nodi 1-5
Precondizioni	Le dipendenze esterne sono state sostituite con mock per garantire l'isolamento e l'indipendenza del test. In particolare, il mock della repository temporaryUserRepository restituisce un oggetto TemporaryUserEntity con email uguale a quella dell'input quando viene chiamato il metodo findByEmail(email), **per simulare l'esistenza di un utente già registrato con l'email di input**.
Input	email="roberto@unina.it"; password="Roberto-98"; attività="Roby Srl"
Output atteso	lancio di eccezione
Postcondizioni	Nessuna postcondizione rilevante.

Tabella 5.12: Definizione caso test 5 - metodo signUpConfirmed

```
1  /*
2      Test 1
3      Copertura: 1-5
4  */
5  @Test
6  public void testSignUpConfirmed()_emailNotRegistered {
7
8      String email = "roberto@unina.it";
9      String password = "Roberto-98";
10     String attivita = "Roby Srl";
11
12     SignUpRequest request = SignUpRequest.builder()
13             .email(email)
14             .password(password)
15             .attivita(attivita)
16             .build();
17
18     TemporaryUserEntity temporaryUserMock = null;
19
```

```
20      when(temporaryUserRepository.findByEmail(email)).thenReturn(
            temporaryUserMock);
21
22      Exception ex = assertThrows(BadRequestException.class, () ->
            authService.signUpConfirmed(request));
23
24      assertEquals(ex.getMessage(), "Email non registrata");
25  }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Accorgersi che non ci sono riferimenti alla prima fase della registrazione e quindi l'user temporaneo non esiste	Lancio dell'eccezione BadRequestException con il riferimento di temporaryUserEntity a null	22	✓
Leggere il messaggio dell'eccezione "	getMessage dell'eccezione lanciata è "Email non registrata"	24	✓

Tabella 5.13: Descrizione del codice del test

Campo	Descrizione
ID	TC-6
Titolo	Errore codice di conferma
Obiettivo	Copertura cammino nodi 1-2-6
Precondizioni	Le dipendenze esterne sono state sostituite con mock per garantire l'isolamento e l'indipendenza del test. In particolare, il mock della repository temporaryUserRepository restituisce un oggetto TemporaryUserEntity con code diverso dal code di input quando viene chiamato il metodo findByEmail(email), **per simulare il caso di un codice input diverso da quello generato dal sistema**.
Input	email="roberto@unina.it"; password="Roberto-98"; attività="Roby Srl"
Output atteso	lancio di eccezione
Postcondizioni	Nessuna postcondizione rilevante.

Tabella 5.14: Definizione caso test 6 - metodo signUpConfirmed

```java
/*
   Test 2
   Copertura: 1-2-6
*/
@Test
public void testSignUpConfirmed_ErrorCode() {

    String email = "roberto@unina.it";
    String password = "Roberto-98";
    String attivita = "Roby Srl";
    int code = 1386;

    SignUpRequest request = SignUpRequest.builder()
            .email(email)
            .password(password)
            .attivita(attivita)
            .confirmationCode(code)
            .build();

    TemporaryUserEntity temporaryUserMock = TemporaryUserEntity.
       builder()
            .code(1300)
            .build();

    when(temporaryUserRepository.findByEmail(email))
        .thenReturn(temporaryUserMock);

    Exception ex = assertThrows(BadRequestException.class, () ->
       authService.signUpConfirmed(request));

    assertEquals(ex.getMessage(), "Codice errato");
}
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Accorgersi che il codice nella richiesta non è quello generato e inviato per email	Lancio dell'eccezione BadRequestException con il codice della richiesta differente dal codice salvato e associato all'user temporaneo	27	✓
Leggere il messaggio dell'eccezione "	getMessage dell'eccezione lanciata è "Codice errato"	29	✓

Tabella 5.15: Descrizione del codice del test

Campo	Descrizione
ID	TC-7
Titolo	Codice di conferma scaduto
Obiettivo	Copertura cammino nodi 1-2-3-7
Precondizioni	Le dipendenze esterne sono state sostituite con mock per garantire l'isolamento e l'indipendenza del test. In particolare, il mock della repository temporaryUserRepository restituisce un oggetto TemporaryUserEntity, con codice uguale a quello di input ma data di creazione uguale a 30 minuti prima della data corrente, quando viene chiamato il metodo findByEmail(email), **per simulare l'esistenza di un utente già registrato con l'email di input ma con codice scaduto**.
Input	email="roberto@unina.it"; password="Roberto-98"; attività="Roby Srl"
Output atteso	lancio di eccezione
Postcondizioni	Nessuna postcondizione rilevante.

Tabella 5.16: Definizione caso test 7 - metodo signUpConfirmed

```java
 /*
   Test 3
   Copertura: 1-2-3-7
*/
@Test
public void testSignUpConfirmed_expired() {

    String email = "roberto@unina.it";
    String password = "Roberto-98";
    String attivita = "Roby Srl";
    int code = 1386;

    SignUpRequest request = SignUpRequest.builder()
            .email(email)
            .password(password)
            .attivita(attivita)
            .confirmationCode(code)
            .build();

    TemporaryUserEntity temporaryUserMock = TemporaryUserEntity.
       builder()
            .createDate(LocalDateTime.now().minusMinutes(30))
            .code(code)
            .build();

    when(temporaryUserRepository.findByEmail(email))
        .thenReturn(temporaryUserMock);

    Exception ex = assertThrows(
        BadRequestException.class, () -> authService.signUpConfirmed(
           request)
    );

    assertEquals(ex.getMessage(), "Il tempo per la conferma è scaduto
       ");
}
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Accorgersi che il tempo per la conferma della registrazione è scaduto	Lancio dell'eccezione BadRequestException con la data di creazione del user temporaneo maggiore di 15 minuti fa.	28	✓
Leggere il messaggio dell'eccezione "	getMessage dell'eccezione lanciata è "Il tempo per la conferma è scaduto"	32	✓

Tabella 5.17: Descrizione del codice del test

Campo	Descrizione
ID	TC-8
Titolo	Registrazione completata
Obiettivo	Copertura cammino nodi 1-2-3-4
Precondizioni	Le dipendenze esterne sono state sostituite con mock per garantire l'isolamento e l'indipendenza del test. In particolare, il mock della repository temporaryUserRepository restituisce un oggetto TemporaryUserEntity(con data corrente e codice uguale a quello dell'input) quando viene chiamato il metodo findByEmail(email), **per simulare l'esistenza di un utente già registrato temporaneamente**.
Input	email="roberto@unina.it"; password="Roberto-98"; attività="Roby Srl"
Output atteso	lancio di eccezione
Postcondizioni	Nessuna postcondizione rilevante.

Tabella 5.18: Definizione caso test 8 - metodo signUpConfirmed

```
/*
      Test 4
      Copertura: 1-2-3-4
   */
    @Test
    public void testSignUpConfirmed_notException() {

        String email = "roberto@unina.it";
        String password = "Roberto-98";
        String attivita = "Roby Srl";
        int code = 1386;

        SignUpRequest request = SignUpRequest.builder()
                .email(email)
                .password(password)
                .attivita(attivita)
                .confirmationCode(code)
                .build();

        TemporaryUserEntity temporaryUserMock = TemporaryUserEntity.
           builder()
                .createDate(LocalDateTime.now())
                .code(code)
                .build();

        when(temporaryUserRepository.findByEmail(email)).thenReturn(
           temporaryUserMock);

        assertEquals(
            "Registrazione completata con  successo",
                                authService.signUpConfirmed(request)
        );
    }
}
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Seguire tutto il flusso del codice senza eccezioni	con gli input corretti il metodo non lancia eccezioni e prosegue fino alla fine restituendo ”Registrazione completata con successo”)	27	✓

Tabella 5.19: Obiettivo e descrizione test

5.2.2 Gestione delle prenotazioni

I metodi scelti per questa macro funzionalità sono quelli più rilevanti per la gestione automatica delle prenotazioni di riserva.

public BookingEntity insertBooking(BookingRequest request)

Si è scelto di testare il metodo `insertBooking` della classe `BookingService`, poiché rappresenta una delle funzionalità principali per la gestione delle prenotazioni. La progettazione dei test case per questo metodo segue una strategia diversa rispetto a quelle descritte nella sezione precedente, in quanto l'elemento critico della logica risiede nella query utilizzata per determinare la priorità della prenotazione.

In particolare, quando si inserisce una nuova prenotazione di riserva, è necessario assegnare il corretto numero di priorità (`queueNumber`). Questo valore viene calcolato in base al numero di utenti già presenti in riserva per lo stesso slot temporale (`timeSlot`) e per la stessa attività (`activityId`).

La query che calcola il numero di priorità è definita nel BookingRepository come segue:

```
1    @Query(SELECT MAX(b.queueNumber) FROM BookingEntity
2    b WHERE b.activityId =  :activityId AND
3    b.timeSlot = :timeSlot AND
4    b.startDate = :date)
5    Integer findMaxQueueNumber(
6    @Param("activityId") Long activityId,
7    @Param("timeSlot") Integer timeSlot,
8    @Param("date") LocalDate date
9    );
```

id	activityId	startDate	timeSlot	queueNumber
1	1	14-11-2025	9	0
2	1	14-11-2025	9	1
3	1	15-11-2025	11	0
4	2	14-11-2025	9	0
5	2	14-11-2025	9	1
6	2	14-11-2025	9	2

Tabella 5.20: Stato Iniziale della tabella Booking per il testare il metodo insertBooking

Campo	Descrizione
ID	TC-9
Titolo	Numero di coda 0
Obiettivo	Verificare che la nuova prenotazione inserita con status CONFRMED viene assegnato un numero di coda a 0
Precondizioni	È utilizzato il database di test H2. Prima di ogni esecuzione del test, la tabella booking viene inizializzata secondo lo stato riportato in Tabella 5.20; id Attvità del contesto sicurezza Spring = 1
Input	bookingRequest.setStartDate(LocalDateTime.of(2025,11,16,9)); bookingRequest.setTimeSlot(9); bookingRequest.setStatus(”CONFIRMED”);
Output atteso	null
Postcondizioni	nuova prenotazione nel db, con i dati dell’input e queueNumber=0

Tabella 5.21: Definizione caso test 9 - metodo insertBooking

```
1
2     /*
3      Caso test 1: Inserire una prenotazione della attività 1 alla
          data 16/11/25 slot orario 9 con stato CONFIRMED
4      Obiettivo: Verificare che viene assegnato il numero 0 come
          numero di coda
5     */
6     @Test
7     void testInsertBooking_expectedQueue_0() {
8
9         MockedStatic<SecurityContexUtils> securityContexMocked =
10            mockStatic(SecurityContexUtils.class);
```

```
11
12          securityContexMocked.when
13          (SecurityContexUtils::getIdAttivitaFromContext)
14          .thenReturn(1);
15
16          bookingRequest.setStartDate(
17              LocalDateTime.of(2025,11,16,9,0)
18              );
19          bookingRequest.setTimeSlot(9)
20          bookingRequest.setStatus("CONFIRMED");
21
22          BookingEntity booking =
23              bookingService.insertBooking(bookingRequest);
24
25          assertEquals(0,booking.getQueueNumber());
26
27          securityContexMocked.close();
28
29      }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Verificare che quando si sta inserendo una prenotazione confermata viene impostato 0 come numero di coda poiché non è una prenotazione di riserva	booking.getQueueNumber è 0	25	✓

Tabella 5.22: Descrizione del codice del test

Campo	Descrizione
ID	TC-10
Titolo	Numero di coda 1
Obiettivo	Verificare che la nuova prenotazione inserita con status PENDING viene assegnato un numero di coda a 1
Precondizioni	È utilizzato il database di test H2. Prima di ogni esecuzione del test, la tabella booking viene inizializzata secondo lo stato riportato in Tabella 5.20; mockato il contesto di sicurezza di Spring simulando che l'operazione di insert lo sta svolgendo un utente associato ad id attività = 1
Input	bookingRequest.setStartDate(LocalDateTime.of(2025,11,15 ,11 ,15); bookingRequest.setTimeSlot(11);
Output atteso	null
Postcondizioni	nuova prenotazione nel db, con i dati dell'input e queueNumber=1

Tabella 5.23: Definizione caso test 10 - metodo insertBooking

```
1     /*
2         Caso test 2: Inserire una prenotazione di riserva della
              attività 1 alla data 15/11/25 slot orario 11.
3         Obiettivo: veirificare che viene assegnato il numero 1 come
              numero di coda
4      */
5     @Test
6     void testInsertBooking_expectedQueue_1() {
7
8         MockedStatic<SecurityContexUtils> securityContexMocked =
              mockStatic(SecurityContexUtils.class);
9
10        securityContexMocked.when(SecurityContexUtils::
              getIdAttivitaFromContext).thenReturn(1);
11
12        bookingRequest.setStartDate(
13            LocalDateTime.of(2025,11,15,11,15)
14        );
15        bookingRequest.setTimeSlot(11);
16
17        BookingEntity booking = bookingService.insertBooking(
              bookingRequest);
18
19        assertEquals(1,booking.getQueueNumber());
20
21        securityContexMocked.close();
22
23    }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Verificare che quando si sta inserendo una prenotazione di riserva in una slot di una data con già altre riserve, relative alla stessa attività, viene assegnato il giusto numero di coda. In questo caso tenendo in considerazione lo stato iniziale del DB, ci aspettiamo 1	booking.getQueueNumber è 1	19	✓

Tabella 5.24: Obiettivo e descrizione test

Campo	Descrizione
ID	TC-11
Titolo	Numero di coda 2
Obiettivo	Verificare che la nuova prenotazione inserita con status PENDING viene assegnato un numero di coda a 2
Precondizioni	È utilizzato il database di test H2. Prima di ogni esecuzione del test, la tabella booking viene inizializzata secondo lo stato riportato in Tabella 5.20; mockato il contesto di sicurezza di Spring simulando che l'operazione di insert lo sta svolgendo un utente associato ad id attività = 1
Input	bookingRequest.setStartDate(LocalDateTime.of(2025,11,14 ,9 ,30); bookingRequest.setTimeSlot(9);
Output atteso	null
Postcondizioni	nuova prenotazione nel db, con i dati dell'input e queueNumber=2

Tabella 5.25: Definizione caso test 11 - metodo insertBooking

```
1
2     /*
3         Caso test 3: Inserire una prenotazione di riserva della
              attività 1 alla data 14/11/25 slot orario 9.
4         Obiettivo: veirificare che viene assegnato il numero 2 come
              numero di coda
5      */
6     @Test
7     void testInsertBooking_expectedQueue_2() {
8
9         MockedStatic<SecurityContexUtils> securityContexMocked =
              mockStatic(SecurityContexUtils.class);
10
11        securityContexMocked.when(SecurityContexUtils::
              getIdAttivitaFromContext).thenReturn(1);
12
13        bookingRequest.setStartDate(
14            LocalDateTime.of(2025,11,14,9,30)
15        );
16        bookingRequest.setTimeSlot(9);
17
18        BookingEntity booking = bookingService.insertBooking(
              bookingRequest);
19
20        assertEquals(2,booking.getQueueNumber());
21
22        securityContexMocked.close();
23    }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Verificare che quando si sta inserendo una prenotazione di riserva in una slot di una data con già altre riserve, relative alla stessa attività, viene assegnato il giusto numero di coda. In questo caso tenendo in considerazione lo stato iniziale del DB, ci aspettiamo 2	booking.getQueueNumber è 2	19	✓

Tabella 5.26: Obiettivo e descrizione test

Campo	Descrizione
ID	TC-12
Titolo	Numero di coda 3
Obiettivo	Verificare che la nuova prenotazione inserita con status PENDING viene assegnato un numero di coda a 3
Precondizioni	È utilizzato il database di test H2. Prima di ogni esecuzione del test, la tabella booking viene inizializzata secondo lo stato riportato in Tabella 5.20; mockato il contesto di sicurezza di Spring simulando che l'operazione di insert lo sta svolgendo un utente associato ad id attività = 2
Input	bookingRequest.setStartDate(LocalDateTime.of(2025,11,14 ,9 ,45); bookingRequest.setTimeSlot(9);
Output atteso	null
Postcondizioni	nuova prenotazione nel db, con i dati dell'input e queueNumber=3

Tabella 5.27: Definizione caso test 12 - metodo insertBooking

```
1     /*
2         Caso test 4: Inserire una prenotazione di riserva della
              attività 2 alla data 14/11/25 slot orario 9.
3         Obiettivo: veirificare che viene assegnato il numero 3 come
              numero di coda
4      */
5     @Test
6     void testInsertBooking_expectedQueue_3() {
7
8         MockedStatic<SecurityContexUtils> securityContexMocked =
              mockStatic(SecurityContexUtils.class);
9
10        securityContexMocked.when(SecurityContexUtils::
              getIdAttivitaFromContext).thenReturn(2);
11
12        bookingRequest.setStartDate(
13            LocalDateTime.of(2025,11,14,9,45)
14        );
15        bookingRequest.setTimeSlot(9);
16
17        BookingEntity booking = bookingService.insertBooking(
              bookingRequest);
18
19        assertEquals(3,booking.getQueueNumber());
20
21        securityContexMocked.close();
22
23    }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Verificare che quando si sta inserendo una prenotazione di riserva in una slot di una data con già altre riserve, relative alla stessa attività, viene assegnato il giusto numero di coda. In questo caso tenendo in considerazione lo stato iniziale del DB, ci aspettiamo 3	booking.getQueueNumber è 3	19	✓

Tabella 5.28: Obiettivo e descrizione test

public List<Long> retrieveUpcomingBooking())

Il metodo `retrieveUpcomingBooking` della classe `BookingService` è uno dei più rilevanti per la gestione automatica delle prenotazioni. Questo metodo viene eseguito automaticamente ogni giorno tramite un'istruzione di tipo `Scheduled`. La sua funzione principale è recuperare tutte le prenotazioni presenti nel database relative a tutte le attività, filtrando quelle che non sono più rilevanti. In particolare, il metodo:

1. Esclude le prenotazioni con date già trascorse.

2. Considera solo le prenotazioni il cui `reminderDate` è uguale o successivo alla data corrente.

3. Include esclusivamente le prenotazioni con stato `CONFIRMED`, evitando quelle ancora in riserva.

Una volta recuperate le prenotazioni, il metodo le prepara per l'invio dei messaggi di promemoria ai clienti, ricordando loro l'avvicinarsi della data della prenotazione e richiedendo conferma. Prima di inviare i messaggi, viene eseguito un ulteriore controllo relativo alla **frequenza del reminder**. Ogni prenotazione, oltre alla data di reminder, memorizza anche la frequenza con cui il promemoria deve essere inviato (ad esempio ogni giorno, ogni due giorni, ecc.). Per verificare se una prenotazione rientra nella frequenza prevista, il metodo calcola il numero di giorni trascorsi tra la data di reminder e la data attuale; se questo numero è un **multiplo dei giorni di frequenza impostati**, la prenotazione viene inclusa tra quelle a cui inviare il promemoria.

La strategia di test scelta consiste nel definire uno **stato iniziale della tabella booking nel database** e verificare che la lista degli ID delle prenotazioni selezionate per l'invio del messaggio restituita dal metodo corrisponda esattamente a quella prevista.

id	startDate	startReminder	status	frequency Days Reminder
1	dataCorrente + 2 giorni	data corrente - 2 giorni	CONFIRMED	3
2	data corrente - 2 giorni	data corrente - 6 giorni	CONFIRMED	4
3	data corrente + 5	data corrente - 2 giorno	PENDING	2
4	data corrente + 4 giorni	data corrente - 4	CONFIRMED	2
5	data corrente + 1	data corrente - 3	PENDING	1
6	data corrente + 10	data corrente + 2	CONFIRMED	2
7	data corrente + 10	data corrente	CONFIRMED	5
8	data corrente + 10	data corrente	PENDING	5
9	data corrente + 5	data corrente + 1	CONFIRMED	3

Tabella 5.29: Stato Iniziale della tabella Booking per il testare il metodo retrieveUpcomingBooking

Nella seguente tabella si mostra le prenotazioni che il metodo dovrebbe selezionare

id	Da selezionare	Motivazione
1	NO	2 (giorni trascorsi) non è un multiplo di 3
2	NO	La data di prenotazione non è più valida
3	NO	È una prenotazione di riserva
4	SI	2 (giorni trascorsi) è un multiplo di 2
5	NO	È una prenotazione di riserva
6	NO	La data di promemoria ancora deve arrivare
7	SI	la data corrente corrisponde alla prima data di promemoria
8	NO	È una prenotazione di riserva
9	NO	La data per iniziare il promemoria ancora deve arrivare

Tabella 5.30: Aspettative delle prenotazioni selezionate dal metodo

Prima di presentare il test implementato, si riporta il metodo oggetto di verifica. Come si può osservare, all'interno del metodo viene aggiunto l'ID di ogni prenotazione alla lista che verrà restituita ogni volta che si entra nello scope per il processo di invio dei messaggi di promemoria. Nel test, pertanto, si verifica la dimensione della lista restituita e si controlla che gli ID attesi siano effettivamente presenti al suo interno. Poiché la dimensione della lista è nota, la presenza degli ID previsti garantisce implicitamente che non siano stati selezionati altri elementi indesiderati.

```
1
2     @Scheduled(cron = "0 00 10 * * *", zone = "Europe/Rome")
3     public List<Long> retrieveUpcomingBooking(){
4
5         List<Long> idBookingsWithMessageSent = new ArrayList<>();
6
7         List<BookingEntity> bookingEntityList = bookingRepository.
              findByStartDateAfterAndStartReminderLessThanEqualAndStatus
              (LocalDate.now(),LocalDate.now(),BookingStatus.CONFIRMED);
```

```
8
9           for(BookingEntity bookingEntity : bookingEntityList){
10
11              LocalDate today = LocalDate.now();
12              LocalDate startReminder = bookingEntity.getStartReminder
                    ();
13              int frequencyDays = bookingEntity.
                    getFrequencyDaysReminder();
14
15              long daysElapsed =
16                  java.time.temporal.ChronoUnit.DAYS.between(
17                      startReminder, today
18              );
19
20              if( (daysElapsed % frequencyDays)==0 ) {
21
22                  Booking booking =
                        convertFromBookinEntityToBookingKafka(
                        bookingEntity);
23
24                  producerService.sendBookingReminder(booking);
25
26                  idBookingsWithMessageSent.add(bookingEntity.getId());
27              }
28          }
29
30          return idBookingsWithMessageSent;
31      }
```

Campo	Descrizione
ID	TC-13
Titolo	Recupero delle prenotazioni per promemoria
Obiettivo	Recuperare solo le prenotazioni che rispettano i criteri per l'invio del promemoria
Precondizioni	È utilizzato il database di test H2. Prima di ogni esecuzione del test, la tabella booking viene inizializzata secondo lo stato riportato in Tabella 5.29;
Input	null
Output atteso	la lista delle prenotazioni recuperate deve essere in linea con le aspetative della tabella Tabella 5.30
Postcondizioni	Nessuna postcondizione rilevante

Tabella 5.31: Definizione caso test 13 - metodo retrieveUpComing

```
1
2     @Test
3     void testRetrieveUpcomingBooking_selectedBooking(){
4
5         List<Long> selectedIdBookings = bookingService.
              retrieveUpcomingBooking();
6
7         assertEquals(2, selectedIdBookings.size());
8
9         assertTrue(selectedIdBookings.contains(4L));
10        assertTrue(selectedIdBookings.contains(7L));
11    }
```

Obiettivo	Comportamento Atteso	Riga Test	Esito
Verificare che la dimensione della lista delle prenotazioni selezionate sia 2, in coerenza con quanto riportato nella tabella delle aspettative.	bookingService. retrieveUpcomingBooking().size == 2	7	✓
Verificare che nella lista delle prenotazioni selezionate siano presenti la prenotazione con ID 4 e quella con ID 7, in coerenza con quanto riportato nella tabella delle aspettative.	bookingService. retrieveUpcomingBooking().contains(4) and bookingService. retrieveUpcomingBooking().contains(7)	9 and 10	✓

Tabella 5.32: Obiettivo e descrizione test

5.3 Test a livello di sistema

Prima di concludere questo capitolo dedicato ai test, vengono presentati la descrizione e i risultati dei test a livello di sistema, come anticipato nella sezione introduttiva. Questi test rappresentano il livello di verifica più completo dopo i test unitari, in quanto mirano a validare scenari realistici di utilizzo, assicurando che tutti i componenti del sistema (API, logica applicativa, database, servizi esterni e microservizi) funzionino correttamente nel loro insieme.

Un test è considerato verificato (esito positivo) se tutti i sottotest che lo compongono hanno esito positivo.

Ogni caso di test è suddiviso in più sottotest eseguiti in ordine sequenziale, in modo da rispecchiare i passaggi di uno scenario reale. I test sono manuali: i sottotest che richiedono chiamate all'API vengono eseguiti con Postman, mentre la verifica dello stato del database e di altre componenti viene effettuata manualmente.

Seguendo l'approccio adottato nelle sezioni precedenti, anche in questa fase vengono analizzate solo alcune funzionalità rappresentative, ritenute sufficienti a dimostrare il corretto funzionamento dei moduli critici già verificati. In particolare, l'attenzione è focalizzata su due aspetti fondamentali del sistema: il processo di autenticazione e la gestione delle prenotazioni.

5.3.1 Autenticazione

Scenario 1: Registrazione e login con esito positivo (flusso ideale)

Campo	Descrizione
ID	TC-14
Titolo	Registrazione e login con successo
Obiettivo	Completare la registrazione ed effettuare il login con le nuove credenziali.
Precondizioni	nessuna
Postcondizioni	Nuovo utente nel database con le credenziali inserite nel test

Tabella 5.33: Definizione caso test 14 - flusso autenticazione completo

N. passo	Descrizione	Input	Output atteso	Esito
1	Richiesta prima fase di registrazione	richiesta post http pb/auth/temporary-signup con body { email: ”badox31679@gyknife.com”, password: ”PasswordTest.98”, attività:”Roby srl }”	risposta http 200 OK	✓
2	Verifica ricezione codice di conferma	null	email ricevuta con codice di conferma	✓
3	Invio codice di conferma per completare la registrazione	richiesta http post pb/auth/confirmed-signup con body { email: ”badox31679@gyknife.com”, password: ”PasswordTest.98”, attività:”Roby srl”, confirmationCode: ”codice ricevuto” }	risposta http 200 OK	✓
4	Login con credenziali create	richiesta http post /pb/auth/signin con body { email: ”badox31679@gyknife.com”, password: ”PasswordTest.98”, }	risposta http 200 OK con body: {email: ”badox31679@gyknife.com”, password: ”PasswordTest.98”, token: ”token generato dal server”	✓

Tabella 5.34: Descrizione scenario testato

Scenario 2: registrazione con esito negativo: email già esistente

Obiettivo: Verificare che il sistema impedisca la registrazione di un utente utilizzando un indirizzo email già presente nel database.

Campo	Descrizione
ID	TC-15
Titolo	Registrazione con email già esistente
Obiettivo	Verificare l'impedimento di una registrazione con email già presente nel sistema
Precondizioni	nessuna
Postcondizioni	Nessun utente inserito nel database

Tabella 5.35: Definizione caso test 15 - flusso autenticazione - email già registrata

N. passo	Descrizione	Input	Output atteso	Esito
1	Invio della richiesta di registrazione (prima fase)	richiesta post http pb/auth/temporary-signup con body { email: "badox31679@gyknife.com", password: "PasswordTest.98", attività:"Roby srl }"	risposta http 400 BAD_REQUEST	✓

Tabella 5.36: Descrizione scenario testato

Scenario 3: login con esito negativo: credenziali errate

Campo	Descrizione
ID	TC-16
Titolo	Login non autorizzato
Obiettivo	Verificare l'impedimento del login con credenziali non corrette
Precondizioni	nessuna
Postcondizioni	Nessuna

Tabella 5.37: Definizione caso test 16 - flusso autenticazione - login non autorizzato

.

N. passo	Descrizione	Input	Output atteso	Esito
1	Invio della richiesta di login con email sbagliata	richiesta post http /pb/auth/-signin con body { email: ”badox@gyknife.com”, password: ”PasswordTest.98”, }”	risposta http 403 FORBIDDEN	✓
2	Invio della richiesta di login con password sbagliata	richiesta post http /pb/auth/signin con body { email: ”badox31679@gyknife.com”, password: ”PasswordTest”, }”	risposta http 403 FORBIDDEN	✓

Tabella 5.38: Descrizione scenario testato

Scenario 4: registrazione con esito negativo: codice di conferma errato

Campo	Descrizione
ID	TC-17
Titolo	Registrazione fallita per codice di conferma errato
Obiettivo	Verificare l’impedimento della registrazione completa quando si inserisce un codice di conferma non valido.
Precondizioni	nessuna
Postcondizioni	Nessun utente creato nel db

Tabella 5.39: Definizione caso test 16 - flusso autenticazione - registrazione non valida - codice conferma non valido

.

N. passo	Descrizione	Input	Output atteso	Esito
1	Richiesta prima fase di registrazione	richiesta post http pb/auth/temporary-signup con body { email: ”badox31679@gyknife.com”, password: ”PasswordTest.98”, attività:”Roby srl }”	risposta http 200 OK	✓
2	Verifica ricezione codice di conferma	null	email ricevuta con codice di conferma	✓
3	Invio codice di conferma diverso da quello ricevuto per completare la registrazione	richiesta http post pb/auth/confirmed-signup con body { email: ”badox31679@gyknife.com”, password: ”PasswordTest.98”, attività:”Roby srl”, confirmationCode: ”codice diverso da quello ricevuto” }	risposta http 400 BAD_REQUEST	✓
4	Invio dopo la scadenza il codice di conferma ricevuto per completare la registrazione	richiesta http post pb/auth/confirmed-signup con body: {email: ”badox31679@gyknife.com”, password: ”PasswordTest.98”, attività:”Roby srl”, confirmationCode: ”codice ricevuto dopo 15min”	risposta http 400 BAD_REQUEST	✓

Tabella 5.40: Descrizione scenario testato

5.3.2 Gestione prenotazione

I test descritti in questa sezione hanno l’obiettivo di verificare scenari relativi all’invio di email ai clienti prenotati, simulando l’automazione del sistema nel ricontattare i clienti in lista di prenotazione di riserva qualora un cliente con prenotazione confermata non risponda o non completi la conferma entro i tempi stabiliti.

l flusso di test parte dalla creazione di un insieme di prenotazioni iniziali e prosegue valutando differenti scenari alternativi. In particolare, vengono confrontate due evoluzioni possibili del

processo:

1. **Scenario A**: Il cliente con prenotazione confermata risponde al promemoria entro la scadenza.

2. **Scenario B**: Il cliente con prenotazione confermata non risponde al promemoria, causando l'attivazione del meccanismo automatico di ricontatto verso i clienti in prenotazione di riserva.

Prima dell'avvio di ciascun test, viene eseguito un reset completo del database al fine di garantire condizioni iniziali consistenti e prive di interferenze da esecuzioni precedenti. Successivamente, vengono inseriti due utenti appartenenti a due attività diverse, così da disporre delle condizioni minime necessarie per l'esecuzione degli scenari di test.

Email	**password**	**ID Attività**
utente1_test001@gmail.com	PasswordTest.98	1
utente2_test001@gmail.com	PasswordTest.98	2

Tabella 5.41: Stato database pre test: Utenti

ID Attività	**date**	**timeSlot**	**status**	**email cliente**

Tabella 5.42: Stato database pre test: Booking - parte 1

ID Attività	**confirmed Booking**	**confirmed reminder**	**max Time Confirmed (min)**	**queue Number**	**start Reminder**	**frequency Day Reminder**

Tabella 5.43: Stato database pre test: Booking - parte 2

Scenario 1: Inserimento di prenotazioni

Campo	Descrizione
ID	TC-18
Titolo	Inserimenti di prenotazioni con stati diversi e da parte di atività diversi
Obiettivo	Verificare che ogni prenotazione venga inserito con i dati giusti in particolare nei campi settati dal sistema come queueNumber e ID attività che viene recuperato dal contesto Spring Security
Precondizioni	tabella utenti inizializzata allo stato riportato nella Tabella 5.41 e tabella booking inizializzato con 0 tuple come nella tabella Tabella 5.42 e Tabella 5.43
Postcondizioni	tabella booking deve essere nello stato riportato alla Tabella 5.48 e Tabella 5.49

Tabella 5.44: Definizione caso test 18 - flusso inserimenti prenotazioni

.

N. passo	Descrizione	Input	Output atteso	Esito
1	Login di utente1_001@gmail.com allo scopo di eseguire le operazioni da parte dell'attività con id 1.	richiesta http /pb/auth/signin con body { email: "utente1_001@gmail.com", password: "PasswordTest.98", }	risposta http 200 OK con body: {email: "utente1_001@gmail.com", password: "PasswordTest.98", token: "token generato dal server"	✓
2	Inserire una prenotazione confermata il giorno 17 Novembre slot 9. Si allega il token ricevuto alla risposta del login dell'utente 1.	richiesta http post /booking con header { Authorization: Bearer token ricevuto } e body { date: 18-11-2025, timeSlot: 9, status: CONFIRMED, email cliente: testingReminder@gmail.com, startReminder: 14-11-2025, max Time Reminder: 3, frequency Day Reminder: 1 }	risposta http 200 OK	✓
3	Verificare lo stato del DB	null	Nuova tupla nella tabella Booking con id Attività: 1, date: 18-11-2025, slot: 9, status: CONFIRMID, queue number: 0, email cliente: testingReminder@gmail.com, startReminder: 14-11-2025, max Time Reminder: 3, frequency Day Reminder: 1, confirmed Booking: false, confirmed Reminder: false, already sent: false	✓
4	Inserimento prenotazione sempre dell'utente1 con stessa data e stessa slot ma con status PENDING	richiesta http post /booking con header { Authorization: Bearer token ricevuto } e body { date: 18-11-2025, timeSlot: 9, status: PENDING, email cliente: testingReminder@gmail.com, startReminder: 14-11-2025, max Time Reminder: 3, frequency Day Reminder: 1 }	risposta http 200 OK	✓

Tabella 5.45: Descrizione scenario testato

N. passo	Descrizione	Input	Output atteso	Esito
5	Verificare lo stato del DB	null	Nuova tupla nella tabella Booking con id Attività: 1, date: 18-11-2025, slot: 9, status: PENDING, queue number: 1, email cliente: testingReminder@gmail.com, startReminder: 14-11-2025, max Time Reminder: 3, frequency Day Reminder: 1, confirmed Booking: false, confirmed Reminder: false, already sent: false	✓
6	Login di utente2 allo scopo di eseguire le operazioni da parte dell'attività con id 2	richiesta http /pb/auth/signin con body { email: "utente1_002@gmail.com", password: "PasswordTest.98", }	risposta http 200 OK con body: {email: "utente2_001@gmail.com", password: "PasswordTest.98", token: "token generato dal server"	✓
7	Inserire una prenotazione confermata il giorno 17 Novembre slot 9. Si allega il token ricevuto alla risposta del login dell'utente 1.	richiesta http post /booking con header { Authorization: Bearer token ricevuto } e body { date: 18-11-2025, timeSlot: 9, status: CONFIRMED, email cliente: testingReminder@gmail.com, startReminder: 14-11-2025, max Time Reminder: 3, frequency Day Reminder: 1 }	risposta http 200 OK	✓
8	Verificare lo stato del DB	null	Nuova tupla nella tabella Booking con id Attività: 2, date: 18-11-2025, slot: 9, status: CONFIRMID, queue number: 0, email cliente: testingReminder@gmail.com, startReminder: 14-11-2025, max Time Reminder: 3, frequency Day Reminder: 1, confirmed Booking: false, confirmed Reminder: false, already sent: false	✓

Tabella 5.46: Descrizione scenario testato

N. passo	Descrizione	Input	Output atteso	Esito
9	Inserimento prenotazione sempre dell'utente2 con stessa data e stessa slot ma con status PENDING	richiesta http post /booking con header { Authorization: Bearer token ricevuto } e body { date: 18-11-2025, timeSlot: 9, status: PENDING, email cliente: testingReminder@gmail.com, startReminder: 14-11-2025, max Time Reminder: 3, frequency Day Reminder: 1 }	risposta http 200 OK	✓
10	Verificare lo stato del DB	null	Nuova tupla nella tabella Booking con id Attività: 2, date: 18-11-2025, slot: 9, status: PENDING, queue number: 1, email cliente: testingReminder@gmail.com, startReminder: 14-11-2025, max Time Reminder: 3, frequency Day Reminder: 1, confirmed Booking: false, confirmed Reminder: false, already sent: false	✓

Tabella 5.47: Descrizione scenario testato

Nuovo stato DB tabella Booking:

ID Prenotazione	ID Attività	date	timeSlot	status	email cliente
1	1	18-11-2025	9	CONFIRMED	testinfReminder@gmail.com
2	1	18-11-2025	9	PENDING	testinfReminder@gmail.com
3	2	18-11-2025	9	CONFIRMED	testinfReminder@gmail.com
4	2	18-11-2025	9	PENDING	testinfReminder@gmail.com

Tabella 5.48: Stato database post inserimenti: Booking - parte 1

ID Prenotazione	confirmed reminder	max Time Confirmed (min)	queue Number	start Reminder	frequency Day Reminder
1	FALSE	5	0	14-11-2025	1
2	FALSE	5	1	14-11-2025	1
3	FALSE	5	0	14-11-2025	1
4	FALSE	5	1	14-11-2025	1

Tabella 5.49: Stato database post inserimenti: Booking - parte 2

Dopo l'esecuzione del test relativo allo scenario che prevede l'inserimento di più prenotazioni da parte di due utenti appartenenti ad attività differenti, e dopo aver osservato il nuovo stato della tabella *booking* nel database, è possibile notare che i valori generati risultano tra loro molto *simili*. Questa scelta non è casuale: avere prenotazioni con caratteristiche quasi identiche permette infatti di impostare lo scenario successivo con condizioni **di confine** e quindi più significative, rendendo i risultati dei test più affidabili.

La strategia utilizzata è stata la seguente:

1. **Sono state inserite quattro prenotazioni**, due per ciascun utente:
   - l'utente 1 (associato all'attività 1) ha inserito due prenotazioni;
   - l'utente 2 (associato all'attività 2) ha inserito altre due prenotazioni.
2. **Per ogni utente è stata inserita una prenotazione confermata e una prenotazione di riserva**, mantenendo la stessa data e lo stesso orario (slot) per entrambe.
3. Lo scenario di test successivo dovrà verificare che **le prenotazioni appartenenti ad attività diverse siano completamente indipendenti tra loro**, anche quando condividono la stessa data e lo stesso slot. Questo permette di confermare che il sistema gestisce correttamente flussi separati per attività distinte, senza interferenze indesiderate.

Scenario 2: Meccanismo automatico di gestione delle prenotazioni di riserva

Obiettivo: Verificare il corretto funzionamento del meccanismo automatico che gestisce le prenotazioni di riserva. In particolare, lo scenario ha due finalità:

1. **Accertare che il sistema contatti automaticamente i clienti in prenotazione di riserva** quando una prenotazione confermata viene annullata o quando il cliente non completa la conferma entro i tempi previsti.

2. **Verificare la corretta riallocazione delle prenotazioni**, controllando che la prenotazione di riserva venga promossa allo stato "confermata" in modo coerente e che la struttura del database (tabella booking) rispecchi correttamente la nuova situazione.

A questo scopo è stato creato un endpoint dedicato ai test che invoca manualmente il metodo retrieveUpcomingBooking. Tale metodo, in condizioni normali, viene eseguito automaticamente dal server una volta al giorno tramite un meccanismo di Scheduled Task. L'introduzione dell'endpoint consente quindi di riprodurre in modo controllato il comportamento giornaliero del sistema, facilitando la verifica del flusso di gestione automatica delle prenotazioni di riserva durante i test end-to-end.

Campo	Descrizione
ID	TC-19
Titolo	Meccanismo automatico promemoria e cambiamento di stato delle prenotazioni
Obiettivo	Basandoci sulle nuove tuple di prenotazioni create nel test precedente, si vuole invocare il servizio che viene invocato ogni giorno per il recupero delle prenotazioni per il promemoria con lo scopo di verificare poi i nuovi stati delle prenotazioni in conseguenza al comportamento svolto nei test (alcune prenotazioni clicchiamo il link di conferma altre no).
Precondizioni	tabella utenti inizializzata allo stato riportato nella Tabella 5.48 e Tabella 5.49
Postcondizioni	tabella booking deve essere nello stato riportato alla Tabella 5.53 e Tabella 5.54

Tabella 5.50: Definizione caso test 19 - flusso meccanismo promemoria e automatizzazione dei cambi di stato delle prenotazioni

.

N. passo	Descrizione	Input	Output atteso	Esito
1	eseguire il metodo retrieveUpcomingBooking attraverso endpoint di test, per inviare i promemoria alle giuste prenotazioni	richiesta http get /test/retrieveUpcomingBooking	risposta http 200 OK	✓
2	Verificare che siano arrivate le email	null	esclusivamente 2 email all'indrizzo testingReminder@gmail.com, un email legata alla prenotazione id 1, un email legata alla prenotazione id 3. Entrambi email devono riportare il link di conferma con esplicita scadenza entro 5 min da quando è partito il test	✓

Tabella 5.51: Descrizione scenario testato

N. passo	Descrizione	Input	Output atteso	Esito
3	Confermare solo 1 delle 2 email ricevute	click sul link dell'email legata alla prenotazione id 1 entro il termine di scadenza	il link mostra un popup di avvenuta conferma	✓
4	Dopo aver aspettato la scadenza dei link, visualizzare il nuovo stato del database	null	cambiamento solo per la prenotazione con id 3 con nuovo status: CANCELED	✓
5	Verificare che è arrivata una, e solo una, nuova email che informa che c'è disponibilità per convertire la prenotazione di riserva in confermata.	null	una sola email ricevuta all'indirizzo testingReminder@gmail.com legata alla prenotazione con ID 4, informa il cliente della nuova disponibilità della prenotazione a cui ha espresso interesse e lo invita a cliccare il link, entro la esplicita scadenza, per confermare la prenotazione.	✓
6	Confermare la prenotazione per il posto liberato	click link dell'email entro la scadenza	popup di avvenuta conferma	✓
7	Verificare il nuovo stato del DB	null	cambio di status da PENDING a CONFERMED solo per la prenotazione con ID 4	✓

Tabella 5.52: Descrizione scenario testato

Nuovo stato DB tabella Booking:

ID Prenotazione	ID Attività	date	timeSlot	status	email cliente
1	1	18-11-2025	9	CONFIRMED	testinfReminder@gmail.com
2	1	18-11-2025	9	PENDING	testinfReminder@gmail.com
3	2	18-11-2025	9	CANCELED	testinfReminder@gmail.com
4	2	18-11-2025	9	CONFIRMED	testinfReminder@gmail.com

Tabella 5.53: Stato database post conferme email: Booking - parte 1

ID Prenotazione	confirmed reminder	max Time Confirmed (min)	queue Number	start Reminder	frequency Day Reminder
1	FALSE	5	0	14-11-2025	1
2	FALSE	5	1	14-11-2025	1
3	FALSE	5	0	14-11-2025	1
4	FALSE	5	1	14-11-2025	1

Tabella 5.54: Stato database post conferme email: Booking - parte 2

Analizzando il nuovo stato del database dopo l'esecuzione del test, si nota che per la **prenotazione 1** il campo **confirmedReminder** risulta ancora impostato a **FALSE**, nonostante l'utente abbia effettivamente confermato il promemoria. Questo comportamento è corretto ed è dovuto al funzionamento del processo di gestione dei promemoria.

Il valore **confirmedReminder viene impostato a TRUE** solo nel momento in cui l'utente clicca il link presente nell'email di conferma del promemoria. Successivamente:

1. Il server esegue automaticamente, allo scadere del tempo previsto per la conferma, un controllo sulla prenotazione.
2. Durante tale controllo, il sistema verifica il valore di confirmedReminder.
3. Una volta rilevato che la conferma è avvenuta correttamente, procede alla gestione della prenotazione e r**eimposta il campo a FALSE**, in modo da poterlo riutilizzare nelle eventuali future comunicazioni di promemoria relative alla stessa prenotazione.

Questo meccanismo garantisce che il campo confirmedReminder possa essere utilizzato più volte nel ciclo di vita della prenotazione senza richiedere ulteriori colonne o stati aggiuntivi.

5.4 Conclusioni finali

I test effettuati coprono solo una parte del dei possibili scenari, input e funzionalità. Lo scopo è stato quello di illustrare alcune metodologie e tecnologie moderni di testing riconosciuti nell'ingegneria del software.

Così come lo sviluppo di un sistema richiede una progettazione accurata prima dell'implementazione, anche il testing deve essere pianificato con criteri chiari, logici e basati su strategie adeguate. La progettazione dei test non rappresenta un semplice passaggio operativo, ma una fase autonoma e strutturata che, in contesti professionali, può essere affidata a un team distinto da quello di sviluppo.

Il lavoro svolto in questo capitolo riflette questa visione: sono state analizzate e testate alcune funzionalità significative del sistema seguendo un percorso metodologico, razionale e coerente. Ciò ha permesso di mostrare concretamente come la fase di testing sia fondamentale per garantire qualità, affidabilità e robustezza del software sviluppato.

Capitolo 6

Analisi dell'usabilità dell'interfaccia utente

Durante lo sviluppo dell'interfaccia utente di un sistema, è fondamentale tenere conto della sua usabilità. Con questo termine si indica la misura qualitativa della facilità, dell'efficienza e dell'efficacia con cui un utente è in grado di utilizzare le funzionalità offerte dal sistema.

Non in tutti i contesti, tuttavia, l'usabilità rappresenta la priorità assoluta. Nei sistemi critici (come il controllo del traffico aereo o la gestione di reattori nucleari) anche piccoli errori o ritardi possono provocare conseguenze gravi. In tali scenari, gli operatori sono altamente formati e l'interfaccia può sacrificare parte dell'intuitività a favore della massima efficienza operativa.

Al contrario, l'applicazione sviluppata è destinata a un pubblico eterogeneo e non specializzato: per questo motivo la facilità di apprendimento, la riduzione degli errori e la soddisfazione percepita diventano elementi centrali. Inoltre, il settore delle applicazioni per prenotazioni è molto competitivo: un utente che non riesce a raggiungere rapidamente un obiettivo tende ad abbandonare l'applicazione e rivolgersi a un servizio concorrente.

Esistono diversi approcci per valutare e garantire l'usabilità di un'interfaccia. Nel presente lavoro, per assicurare la progettazione di un'interfaccia usabile, il design è stato sviluppato facendo riferimento a principi ampiamente riconosciuti nell'ambito dell'**Human–Computer Interaction**[6]. In particolare, sono state considerate le **dieci euristiche di Nielsen**[7] e le **otto regole d'oro di Shneiderman**[8], linee guida consolidate che supportano la realizzazione di interfacce intuitive, coerenti ed efficienti.

6.1 Principi di usabilità di riferimento

Prima di illustrare come i principi di usabilità siano stati applicati nel frontend sviluppato, viene fornita una breve panoramica delle linee guida considerate e del loro significato..

Un sistema dovrebbe innanzitutto presentare un **dialogo semplice e naturale**. Ogni elemento aggiuntivo dell'interfaccia rappresenta un'informazione in più da apprendere: un eccesso di dettagli può generare confusione e rallentare l'interazione. Questo aspetto è coerente con la **legge di Hick**, secondo cui il tempo necessario per prendere una decisione aumenta in modo logaritmico all'aumentare del numero di scelte disponibili. Inoltre, i dialoghi devono utilizzare **terminologia familiare agli utenti**, evitando riferimenti tecnici orientati al funzionamento interno del sistema.

Un altro principio fondamentale consiste nel **ridurre il carico di memoria dell'utente**. A differenza dei calcolatori, gli esseri umani dispongono di una capacità mnemonica limitata; è quindi opportuno mostrare esempi, suggerimenti o istruzioni contestuali ogni volta che viene richiesto un input, per evitare che l'utente debba ricordare informazioni non visibili.

Un sistema deve inoltre essere **coerente**, sia internamente sia esternamente. La coerenza interna implica che l'utente possa aspettarsi sequenze d'azione simili in situazioni simili; la coerenza esterna riguarda invece l'adesione a convenzioni e modelli già diffusi in applicazioni analoghe, facilitando così la familiarità con l'interfaccia.

L'uso appropriato dei **feedback** costituisce uno dei principi più importanti per un'esperienza d'uso efficace. Il sistema dovrebbe informare costantemente l'utente riguardo allo stato delle operazioni in corso e all'interpretazione dei comandi inseriti. Sono essenziali non solo i messaggi di errore, ma anche i feedback positivi che confermano il corretto completamento di un'azione.

L'utente dovrebbe sempre poter contare su funzionalità di **reversibilità delle azioni**, ovvero la possibilità di annullare o correggere operazioni precedenti. Le modalità per farlo devono essere chiare e sempre accessibili.

Gli utenti più esperti traggono beneficio da **scorciatoie e meccanismi che aumentano l'efficienza**, come combinazioni da tastiera o percorsi d'uso semplificati. Questi elementi consentono di ridurre i tempi necessari per operazioni frequenti o complesse.

I **messaggi di errore** devono essere chiari, privi di codici tecnici e orientati alla risoluzione del problema, aiutando l'utente a comprendere quale azione intraprendere per correggere la si-

tuazione.

Ancora più importante della gestione degli errori è la loro **prevenzione**. Quando possibile, il sistema dovrebbe evitare che l'utente si trovi in situazioni ad alto rischio di errore. Ad esempio, anziché richiedere l'inserimento manuale di una città (con il rischio di errori ortografici) risulta preferibile offrire una lista precompilata da cui selezionare l'opzione desiderata.

6.2 Applicazione dei principi nel design dell'interfaccia

Vengono presentati alcune delle applicazioni dei principi di usabilità, discussi nella sezione precedente, seguiti nella progettazione dell'iterfaccia utente.

Sono presentate di seguito alcune applicazioni dei principi di usabilità, illustrati nella sezione precedente, adottati durante la progettazione dell'interfaccia utente.

6.2.1 Coerenza

Coerenza interna

Gli screen riportati di seguito mostrano come tutte le sezioni dell'applicazione adottino uno **stile visivo uniforme**, garantendo un'elevata coerenza interna. Il colore principale dell'interfaccia è il **blu**, utilizzato per i pulsanti primari e per gli elementi che richiedono maggiore attenzione. Le etichette (label) dei pulsanti principali sono presentate in **bianco**, mentre le etichette testuali indipendenti adottano il **nero**, al fine di mantenere una chiara distinzione visiva tra contenuti e azioni.

Ogni finestra modale presenta un **header** caratterizzato dal colore principale dell'applicazione, con testo in bianco, in linea con il tema generale. I pulsanti e gli elementi secondari utilizzano invece un **grigio neutro**, che consente di differenziare visivamente i livelli di importanza senza compromettere la coerenza complessiva dell'interfaccia.

Nonostante questa uniformità, è presente un **pulsante quadrato verde**, di dimensioni leggermente maggiori rispetto agli altri. Tale differenziazione non compromette la coerenza interna, in quanto il pulsante ha la funzione di richiamare un'**azione principale della home**, ovvero l'operazione di inserimento: la maggiore evidenza visiva risulta quindi coerente con la sua rilevanza funzionale.

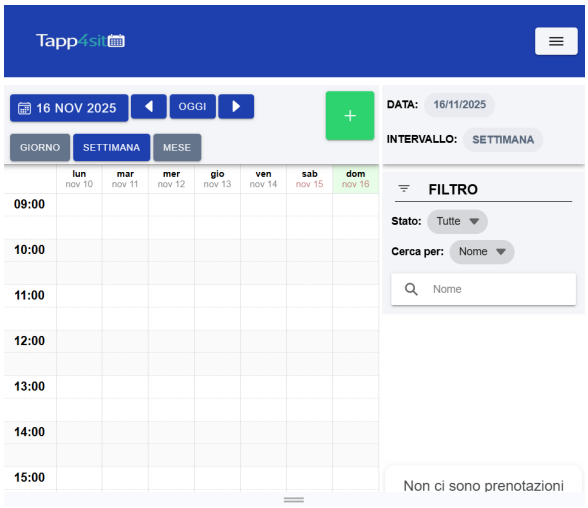


Figura 6.1: Home

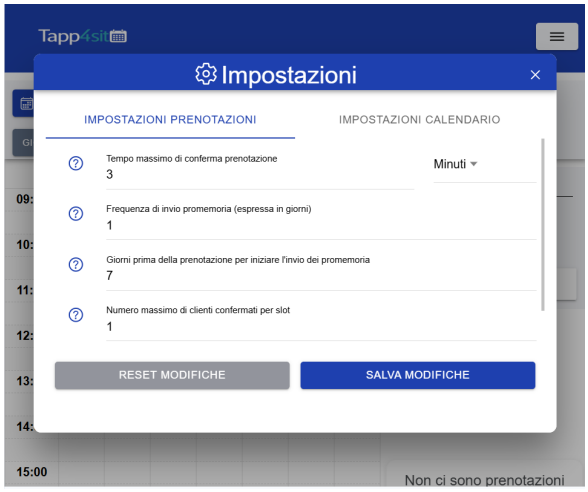


Figura 6.2: Impostazioni

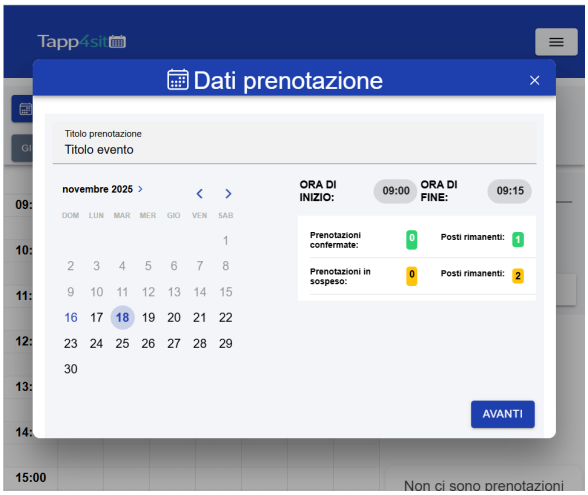


Figura 6.3: Inserimento prenotazione

Coerenza esterna

Di seguito sono riportate alcune immagini a confronto tra il calendario presente nella home di *Tapp4Sit* e il calendario di **Microsoft Teams**. L'analogia tra i due layout è evidente: entrambi presentano una **suddivisione degli slot orari simile**, un **header superiore** contenente le azioni principali e una struttura visiva che permette all'utente di orientarsi facilmente. L'adozione di convenzioni già diffuse in applicazioni note facilita l'apprendimento dell'interfaccia e contribuisce a ridurre il carico cognitivo dell'utente, migliorando così l'usabilità complessiva.

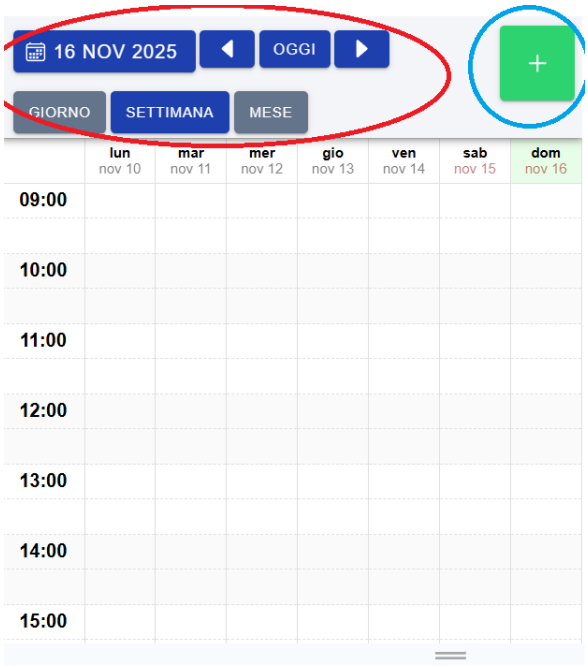


Figura 6.4: Calendario Tapp4Sit

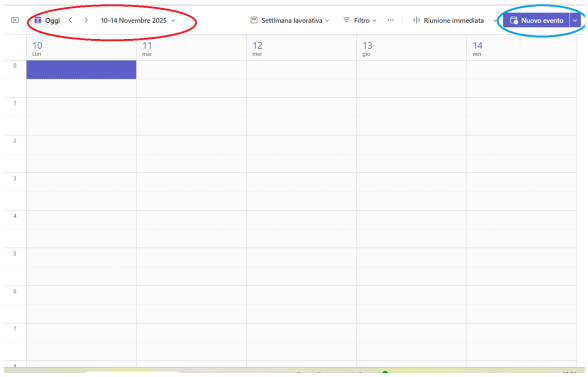


Figura 6.5: Calendario Teams

6.2.2 Riduzione carico di memoria

Il principio di **riduzione del carico di memoria** trova massima applicazione nell'interfaccia dedicata all'inserimento di una nuova prenotazione. Per evitare di sovraccaricare l'utente con troppe informazioni simultanee, il processo è stato suddiviso in **tre sezioni distinte**, in linea con il principio del dialogo semplice.

Prima sezione: informazioni principali sulla prenotazione

La prima parte del form raccoglie i dati fondamentali della prenotazione ed è ulteriormente suddivisa in due colonne.

- **Colonna sinistra**
  Qui si trovano l'input *Titolo evento* e il selettore della data. L'utente può comprendere immediatamente il significato di ogni campo grazie alla label posta sopra l'input, senza dover ricordare informazioni precedenti. Per la data è disponibile un **calendario interattivo**, che permette di selezionare il giorno desiderato evitando qualsiasi sforzo legato alla **formattazione** (ad es. dd-mm-yyyy, mm-dd-yyyy, ecc.).

- **Colonna destra**
  In questa area è presente un selettore dedicato all'**orario**, che consente di impostare ore e

minuti tramite un componente visuale, evitando così errori di inserimento o necessità di ricordare la sintassi corretta. Sotto il selettore è inoltre mostrata una **sezione informativa dinamica** che aggiorna in tempo reale la **disponibilità dello slot** in base alla data e all'ora selezionate.

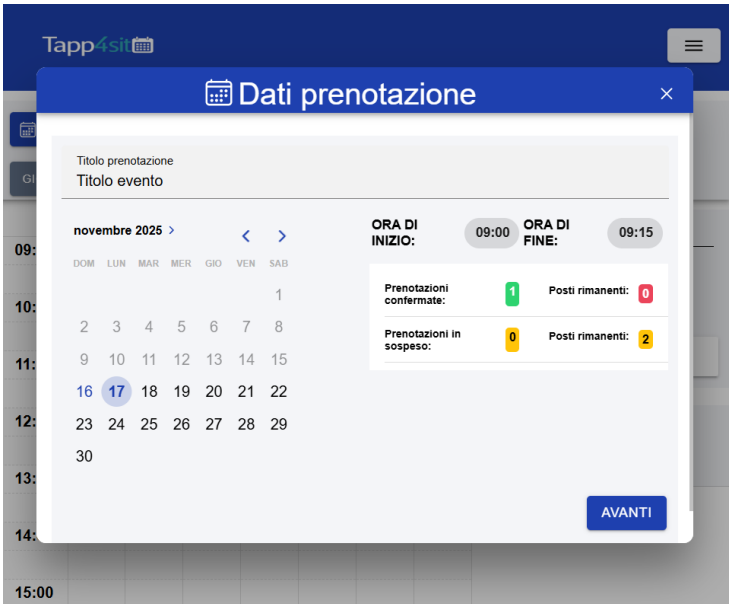


Figura 6.6: Prima sezione dell'inserimento di una prenotazione

Seconda sezione: informazioni del cliente

La seconda parte del form è dedicata ai dati del cliente. Ogni campo è accompagnato da **icone e label esplicative**, che guidano l'utente nella compilazione, riducendo la necessità di ricordare quali informazioni inserire.

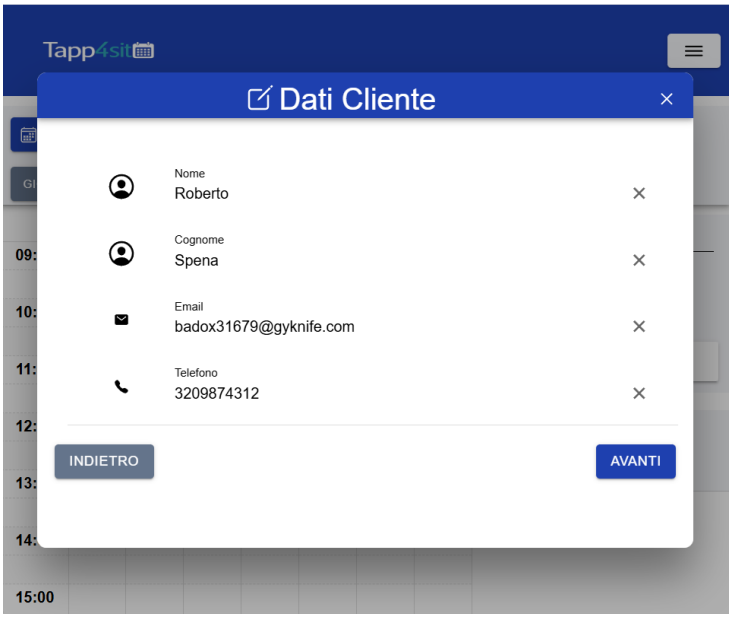


Figura 6.7: Seconda sezione dell'inserimento di una prenotazione

Terza sezione: riepilogo finale

Nell'ultima sezione viene fornito un **riepilogo completo** di tutti i dati inseriti nelle fasi precedenti. Questa panoramica permette all'utente di verificare con facilità la correttezza delle informazioni, riducendo l'ansia di commettere errori e minimizzando il carico cognitivo complessivo.

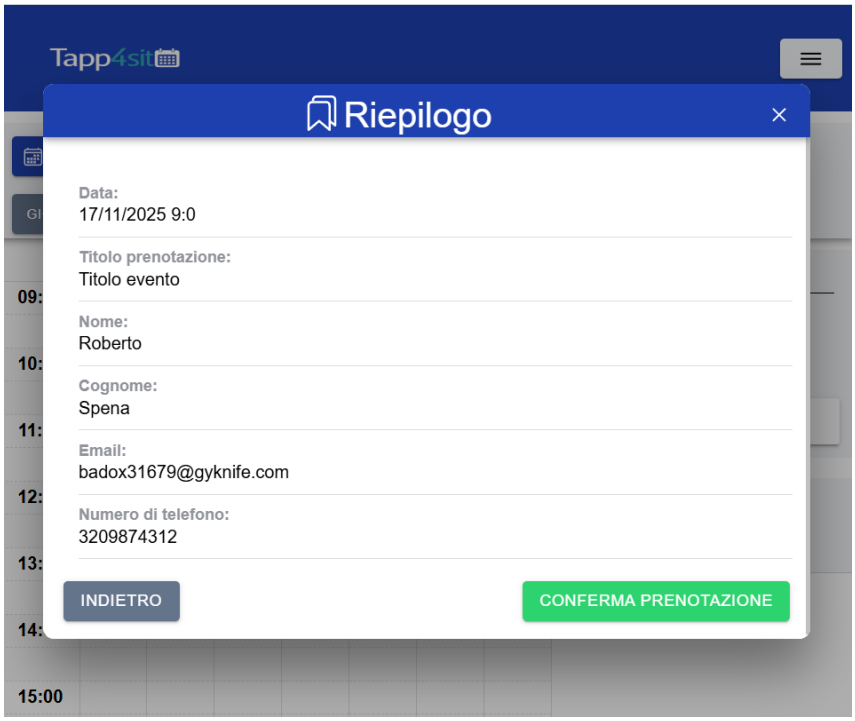


Figura 6.8: Riepilogo dati dell'inserimento di una prenotazione

6.2.3 Feedback

L'applicazione fa un uso estensivo del feedback per garantire all'utente consapevolezza continua dello stato del sistema. I diversi tipi di feedback sono progettati con livelli di permanenza proporzionati all'importanza dell'operazione eseguita.

Feedback istantanei

I feedback istantanei sono implementati tramite **toast**, utilizzati per comunicare messaggi a bassa permanenza relativi a operazioni non definitive. Un esempio significativo riguarda il cambio di tab all'interno del modal delle impostazioni: quando l'utente passa a una scheda diversa senza aver salvato le modifiche, un toast lo informa che i valori sono stati ripristinati alle impostazioni precedenti.

Questo tipo di feedback è stato preferito perché l'utente non ha compiuto alcuna operazione permanente e non è quindi necessario un messaggio modale più invasivo.

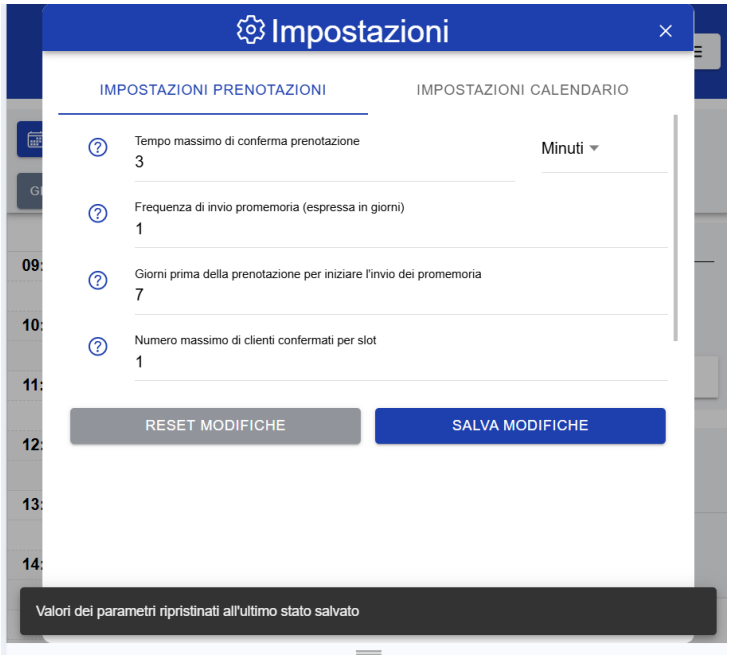


Figura 6.9: Feedback istantaneo: valori ripristinati

Feedback con conferma esplicita

Per operazioni più importanti, come:

- creazione di una nuova prenotazione,

- salvataggio delle impostazioni,
- cancellazione di una prenotazione,
- modifica di una prenotazione,

l'applicazione utilizza dei **feedback con conferma esplicita**. Dopo l'esecuzione dell'azione, compare una notifica chiara e ben visibile che conferma l'avvenuto completamento dell'operazione. Questo approccio assicura che l'utente non abbia dubbi sull'esito dell'azione eseguita, specialmente quando si tratta di operazioni che modificano il contenuto persistente del sistema.

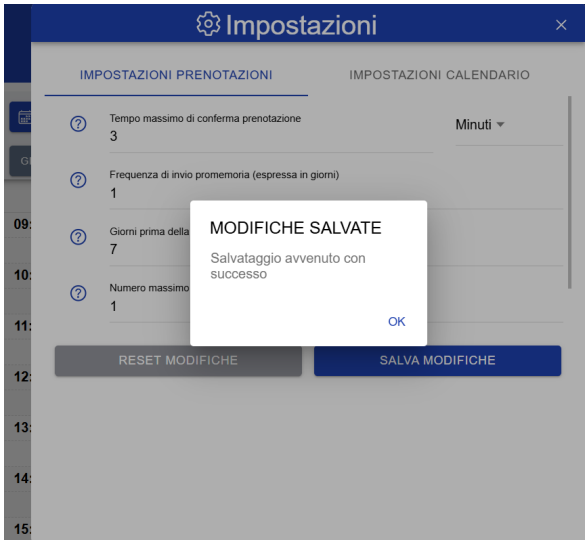


Figura 6.10: Feedback modifiche impostazioni

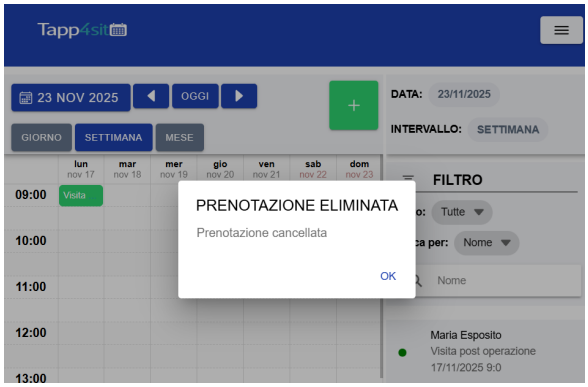


Figura 6.11: Feedback cancellazione prenotazione

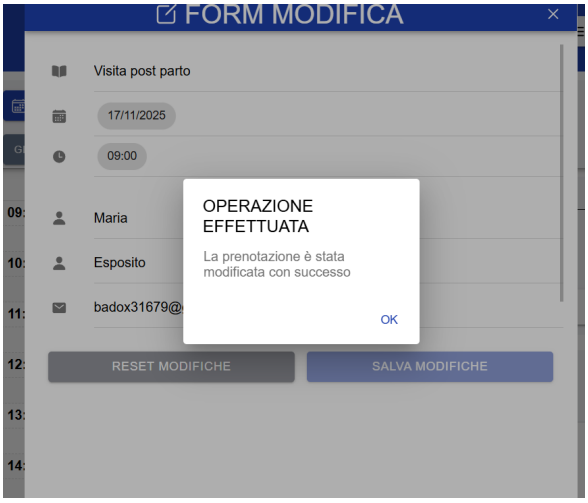


Figura 6.12: Feedback modifica prenotazione

Feedback permanenti

Infine, l'applicazione fornisce **feedback permanenti** nel contesto del calendario. Dopo il salvataggio di una prenotazione, l'utente può immediatamente visualizzare l'evento inserito nello slot corretto del calendario. La presenza costante dell'elemento, insieme ai suoi dettagli accessibili tramite un semplice click, garantisce la piena visibilità del risultato dell'azione eseguita.

Questo tipo di feedback è particolarmente utile poiché mantiene la coerenza tra dati salvati e rappresentazione visuale, riducendo la possibilità di dubbi o errori di interpretazione.

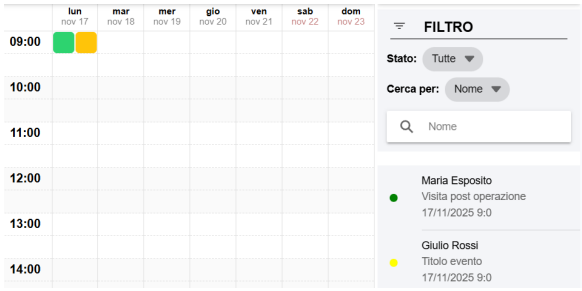


Figura 6.13: Feedback prenotazione inserita

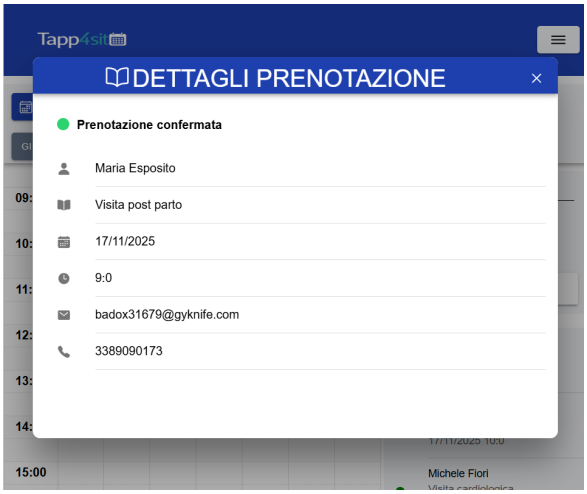


Figura 6.14: Feedback dettagli operazione

6.2.4 Reversibilità delle azioni

Il principio di **reversibilità delle azioni** è stato applicato in modo sistematico durante la progettazione dell'interfaccia, al fine di ridurre l'ansia da errore e permettere all'utente di correggere facilmente eventuali operazioni non desiderate.

Conferme prima delle operazioni permanenti

Per tutte le principali operazioni del sistema (come salvataggi, modifiche o cancellazioni), dopo il tentativo iniziale di completamento viene mostrato un **dialog di conferma**. Questo dialog chiede all'utente se desidera effettivamente procedere e offre sempre la possibilità di **annullare l'azione**, consentendo così un immediato ripensamento o una correzione prima che l'operazione diventi definitiva.

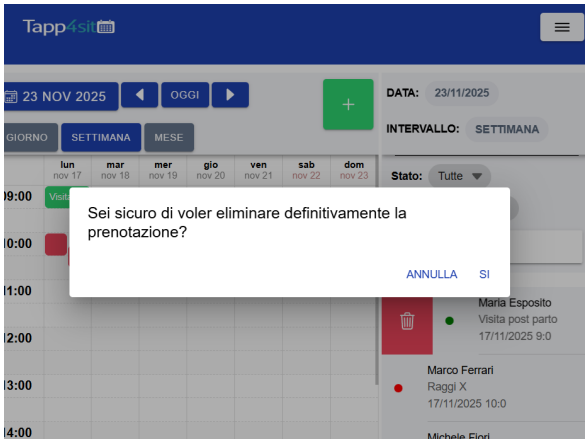


Figura 6.15: Dialog cancellazione prenotazione

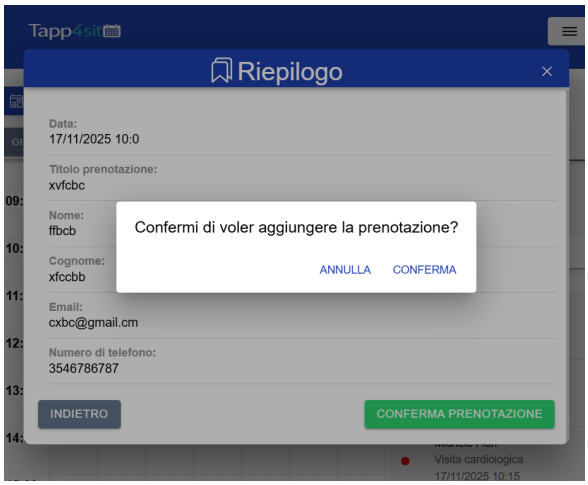


Figura 6.16: Dialog inserimento prenotazione

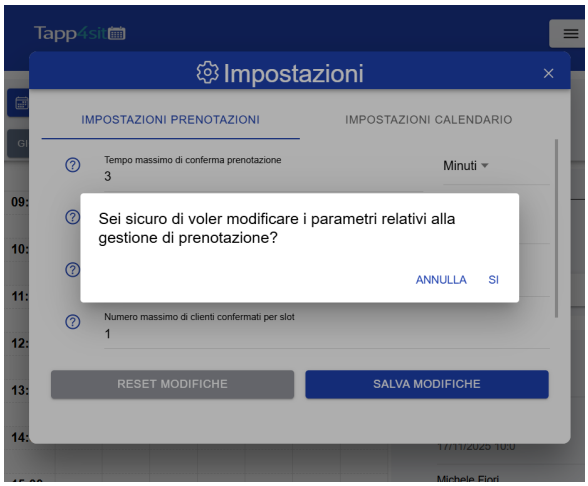


Figura 6.17: Dialog modifiche impostazioni



Figura 6.18: Dialog modifiche prenotazione

Possibilità di annullare modifiche in corso

Oltre alle conferme esplicite, l'interfaccia offre strumenti dedicati per ripristinare lo stato precedente dei dati. In particolare, è presente un tasto "**Reset modifiche**" sia nel modal delle impostazioni sia nel modal di modifica di una prenotazione. Con un semplice click, tutti i campi tornano ai valori dell'ultimo salvataggio, permettendo all'utente di annullare rapidamente modifiche errate o indesiderate.



Figura 6.19: Reset nella finestra di impostazioni



Figura 6.20: Reset nel form di modifica prenotazione

6.2.5 Messaggi di errore

I **messaggi di errore** svolgono un ruolo fondamentale nel supportare l'utente durante la compilazione del form di inserimento di una nuova prenotazione e nella prevenzione di possibili incoerenze o operazioni non valide. L'interfaccia è stata progettata per fornire indicazioni chiare, tempestive e facilmente interpretabili..

All'interno del form, ogni campo può presentare **messaggi di errore mirati**, che compaiono qualora l'input non rispetti i requisiti previsti.

Un ulteriore messaggio di errore di particolare rilevanza compare nella sezione informativa relativa alla disponibilità della slot. Se l'utente tenta di prenotare una fascia oraria già piena secondo le impostazioni precedentemente configurate, il sistema mostra un **messaggio di avviso ben visibile**, che impedisce fraintendimenti e previene la conferma di prenotazioni non valide.



Figura 6.21: Errori form informazioni prenotazione



Figura 6.22: Errori form informazioni cliente

6.3 Conclusioni finali

Il capitolo ha dimostrato che la progettazione dell'**usabilità dell'interfaccia** non è stata trascurata e che il design è stato sviluppato seguendo le **linee guida consolidate della letteratura HCI (Human–Computer Interaction)**. Tuttavia, è importante sottolineare che il dominio dell'usabilità è molto ampio e oggetto costante di studi e perfezionamenti. A tale riguardo, si possono menzionare i principali metodi utilizzati per valutare e migliorare l'usabilità.

Expert Review

In questo approccio vengono coinvolti esperti di usabilità o di applicazioni specifiche del dominio. Gli esperti, attraverso un processo rigoroso, forniscono **feedback mirati al miglioramento del design**. Il vantaggio di questo metodo è la rapidità rispetto al coinvolgimento di utenti reali, e la possibilità di applicarlo in qualsiasi fase del ciclo di vita dello sviluppo. Tuttavia, le raccomandazioni ottenute possono riflettere più le preferenze degli esperti che quelle degli utenti finali.

Testing dell'usabilità

In questo approccio si osservano gli utenti reali mentre interagiscono con l'interfaccia, verificando se il sistema risulta **intuitivo e risponde alle esigenze degli utenti**. I dati raccolti vengono utilizzati per perfezionare l'interfaccia, e i test possono essere ripetuti in modo iterativo fino al raggiungimento di un risultato soddisfacente.

Sondaggi

I **sondaggi strutturati** possono essere utilizzati per raccogliere opinioni mirate sugli aspetti dell'interfaccia. Possono integrare altri metodi di valutazione, anche dopo il lancio dell'applicazione, e consentono di raggiungere un **numero molto maggiore di utenti** rispetto al testing diretto.

Valutazione durante l'uso attivo

Nonostante la progettazione dell'interfaccia sia stata condotta in modo accurato, solo l'**uso reale dell'applicazione** può evidenziare ulteriori criticità o opportunità di miglioramento. È quindi fondamentale monitorare costantemente la soddisfazione degli utenti e adattare l'interfaccia in maniera continua, seguendo un processo di **miglioramento iterativo**.

Capitolo 7

Conclusioni e prospettive future

Durante lo svolgimento di questa tesi sono state affrontate le principali fasi del ciclo di vita dello sviluppo software, a partire dalla definizione dei requisiti fino alla progettazione, implementazione e validazione del sistema. Fin dalle prime settimane di tirocinio presso l'azienda *BC Soft S.r.l.*, il lavoro è stato caratterizzato da un intenso confronto con il relatore aziendale, che in questo contesto ha ricoperto il ruolo di committente. I requisiti iniziali non erano completamente definiti, poiché l'idea dell'applicazione presentava ancora margini di incertezza. Questo scenario ha rappresentato un'opportunità significativa per sperimentare una situazione tipica dei progetti reali: trasformare requisiti di dominio vaghi in specifiche tecniche attraverso analisi, interviste e interazioni iterative con gli stakeholder. Man mano che le discussioni procedevano, le funzionalità attese diventavano più precise e il quadro complessivo del sistema prendeva forma.

Durante la definizione dei requisiti è stato inoltre necessario considerare le tempistiche del tirocinio e le competenze tecniche iniziali, valutando il tempo richiesto per apprendere nuove tecnologie e implementare le funzionalità pianificate. Questa fase si è rivelata particolarmente formativa, poiché ha simulato condizioni tipiche dell'ambiente lavorativo, dove risulta essenziale stimare correttamente tempi e complessità. Per questo motivo, si è scelto di dare priorità alla realizzazione completa dei requisiti fondamentali, evitando di definire funzionalità troppo ambiziose che avrebbero potuto compromettere la consegna del progetto.

Una volta validata la documentazione dei requisiti, il lavoro è proseguito con la progettazione architetturale e la relativa implementazione. Sebbene le specifiche iniziali orientassero verso un'architettura monolitica, il committente ha richiesto un approccio basato sui microservizi, in previsione di una futura evoluzione del sistema. Non avendo precedenti esperienze dirette con tale paradigma architetturale, questa fase ha rappresentato un ulteriore momento di crescita, poiché ha richiesto studio, autonomia e capacità di adattamento, elementi fondamentali in un

contesto professionale reale.

Completata l'implementazione, il periodo conclusivo del tirocinio è stato dedicato alla revisione dell'interfaccia utente, svolta con il supporto di colleghi più esperti, con l'obiettivo di migliorare l'usabilità complessiva del sistema.

La stesura del presente elaborato ha consentito di analizzare il lavoro svolto con una visione più ampia e sistematica, evidenziando l'importanza della progettazione, della validazione e del testing nel ciclo di vita di un software. Alla luce dei risultati ottenuti, è possibile delineare alcune **prospettive di evoluzione futura** del sistema.

Una prima direzione riguarda l'integrazione di tecniche di **intelligenza artificiale**. Considerando la natura del dominio applicativo, la raccolta progressiva dei dati sulle prenotazioni consentirebbe di costruire un dataset utile per addestrare modelli predittivi. Ciò permetterebbe, ad esempio, di stimare automaticamente la probabilità di mancata presentazione, ottimizzare la gestione delle prenotazioni confermate e di riserva, e suggerire configurazioni dinamiche basate sui tempi di attesa osservati.

Un'ulteriore evoluzione potrebbe riguardare la **sincronizzazione del calendario** con servizi esterni diffusi, quali Google Calendar o Microsoft Teams. L'integrazione permetterebbe agli utenti di gestire i propri appuntamenti anche tramite strumenti già utilizzati quotidianamente, migliorando l'interoperabilità e ampliando l'adozione della piattaforma.

Con queste riflessioni si conclude l'elaborato. Un ringraziamento particolare è rivolto all'azienda *BC Soft S.r.l.* per l'opportunità di tirocinio offerta, e al relatore accademico, prof. **Porfirio Tramontana**, per la disponibilità e il supporto forniti durante tutto il percorso di realizzazione di questa tesi.

Bibliografia

[1] Amberger C., Schreyer D. *What do we know about no-show behavior? A systematic, interdisciplinary literature review* Disponibile su: `https://onlinelibrary.wiley.com/doi/10.1111/joes.12534`

[2] Lange K. *THE LITTLE BOOK ON REST SERVICES* Disponibile su: `https://www.kennethlange.com/books/The-Little-Book-on-REST-Services.pdf`

[3] Documentazione ufficiale Disponibile su: `https://www.jwt.io/introduction#when-to-use-json-web-tokens`.

[4] Jorgensen, P. C. (2014) *Software Testing: A Craftsman's Approach (4th ed.)*

[5] B. Meyer, 1997 "Design by Contract." *Object-Oriented Software Construction* 2nd ed., Prentice Hall.

[6] A. Suhas, *Introduction to Human Computer Interaction* Volume 10 Issue 9, Semptember 2021, International Journal of Science and Research (IJSR), 33(3), 450-455.

[7] Molich, R., Nielsen, J. (1990). *Improving a human-computer dialogue* Pearson Education. Communications of the ACM, 33(3), 338-348.

[8] Shneiderman, B., Plaisant, C. (2010). *Designing the user interface: strategies for effective human-computer interaction.* Pearson Education.