



**UNIVERSITÀ
DEGLI STUDI
DI BERGAMO**

Informatica III

Modulo di Progettazione e Algoritmi cod. 38068

&

Modulo di Progettazione, Algoritmi e Computabilità cod. 38090

Relazione e documentazione sul progetto “ticket-generator”

Prof. Patrizia Scandurra

Salvatore Greco matr.1053509

Fabio Gamba matr. 1053157

Alessandro Chaar matr. 1054918

A.A 2022/2023

Indice

Introduzione al progetto	4
Toolchain	5
Organizzazione del team	6
Modello AGILE.....	7
Iterazione 0	9
0.1 Analisi dei requisiti	10
0.2 Attori	10
0.3 Use case stories	10
0.3.1 Utente standard	10
0.3.2 Utente operatore.....	12
0.3.3 Sistema di gestione della coda.....	12
0.3.4 Sistema di autenticazione.....	12
0.3.5 Sistema di visualizzazione.....	12
0.4 Use Case Diagram.....	13
0.5 Use Case Description (descrizione degli use case)	14
0.5.1 UC1: Generazione biglietto.....	14
0.5.2 UC2: Visualizzazione della dashboard della coda	15
0.5.3 UC3: Generazione biglietto prioritario.....	16
0.5.4 UC4: Log-in	17
0.5.5 UC5: Log-out.....	18
0.5.6 UC6: Gestione profilo utente	19
0.5.7 UC7: Registrazione profilo utente.....	20
0.5.8 UC8: Recupero credenziali.....	21
0.5.9 UC9: Evento di avanzamento della coda	22
0.5.10 UC10: Gestione della coda.....	23
0.5.11 UC11: Visualizzazione dashboard utente.....	24
0.6 Analisi delle specifiche.....	25
0.6.1 Introduzione	25
0.6.2 Specifiche funzionali	25
0.6.3 Logica di funzionamento delle code	26
0.7 Analisi dell'architettura	27
0.7.1 Deployment diagram - Architettura.....	27
0.7.2 Deployment diagram – UML.....	28
Iterazione 1	29
1.1 Component diagram – Interfacce	30

1.2 Class diagram - Interface definition	31
1.3 Class diagram - Data type (presentation model)	32
Iterazione 2	33
2.1 Tecnologie utilizzate	34
2.1.1 Database - MongoDB.....	34
2.1.2 Framework Spring	34
2.1.3 Repository - GitHub	35
2.1.4 Testing - Postman	35
2.2 Scelte di implementazione	35
2.3 Component diagram TicketsRepository.....	36
2.4 Sequence diagram di esecuzione dei servizi.....	37
2.5 Metodo di gestione della coda	38
2.5.1 Pseudocodice della classe ServeNextOPE () e analisi complessità e metodo execute().....	39
2.6 Suite di testing delle API (con Postman)	42
2.7 Casi di test con JUnit.....	46
Iterazione 3	47
3.1 Scelte di implementazione	48
3.2 Component diagram white-box.....	49
3.3 Sequence diagram di esecuzione.....	51
3.4 Pseudocodice Autorizzatore (Componente).....	52
3.4.1 Analisi complessità metodo autenticazione	53
Iterazione Finale	55
4.1 Toolchain e Design pattern MVP IO.....	56
4.1.1 Toolchain	56
4.1.2 Design pattern MVP.....	56
4.2 Sistema di stima del tempo residuo	59
4.2.1 Aggiornamento del data class diagram.....	61
4.3 Conclusioni e repository IO.....	61
4.4 Sviluppi futuri	64
4.5 Manuale utente sulla app Android	65
Analisi statica	69

Introduzione al progetto

Il progetto scelto vuole implementare un sistema di generazione automatica dei biglietti per rispettare il proprio turno all'interno di edifici pubblici, come ad esempio i sistemi presenti negli uffici postali o bancari. In particolare, questo sistema avrà una forte personalizzazione per un CUP (Centro Unico di Prenotazione) ospedaliero, rivolgendosi quindi al settore sanitario. Un punto chiave che lo differenzia dal classico strumento "cartaceo", nonché motivo principale dello sviluppo, è la sua funzionalità IOT, ovvero di avere questo sistema connesso ad internet in modo che possa svolgere il servizio anche al di fuori dell'ambiente locale di utilizzo. Questa funzionalità è richiesta per una motivazione molto importante: quella di poter prenotare un turno senza essere fisicamente presenti per ritirare il biglietto stampato, così come la possibilità di prenotare giorno e ora esatta all'interno del sistema slegandosi dal concetto più antiquato di "coda di attesa". Obiettivo principale del progetto è lo sviluppo del software utilizzando la metodologia AGILE, diminuendo costi e tempi di sviluppo rispetto alle metodologie standard

Toolchain

Implementazione del software

- Linguaggio di programmazione: Java
- IDE: IntelliJ IDEA, (Android Studio), Eclipse
- DBMS: MongoDB Atlas
- GUI: Window Builder

Modellazione

- Use case diagram: app.diagrams.net (ex draw.io)
- Deployment diagram, component diagram, class diagram, sequence diagram, datatype diagram: Astah UML

Analisi del software

- Analisi statica: CodeMR
- Analisi dinamica: JUnit

Documentazione, versioning e organizzazione del team

- Documentazione: Microsoft Word
- Versioning: Git, GitHub come hosting
- Organizzazione team: GitHub, Discord

Organizzazione del team

Superate le prime iterazioni, il team ha usufruito di alcune funzionalità di GitHub, come le Issues e una logica di funzionamento dei branch, per scandire e parallelizzare il lavoro sulla base delle funzionalità da implementare, ad esempio assegnando diverse priorità ai task che vengono generati man mano che si procede nello sviluppo assegnando diverse priorità ai task che vengono generati man mano che si procede nello sviluppo

- Issues: consentono agli sviluppatori di capire immediatamente il lavoro che c'è da svolgere sul codice (anche con l'utilizzo di labels) per implementare le funzionalità o correggere comportamenti indesiderati. Uno sviluppatore può creare e assegnare issues ad altri collaboratori in modo da suddividere il lavoro sulla base delle risorse personali a disposizione (tempo, skills, esperienza...)
- Logica dei branch: ogni sviluppatore possiede il proprio branch derivato dal main branch. Su ogni branch personale avvengono le modifiche degli sviluppatori che poi vengono uniti al main branch tramite Pull Request
- Pull Request: si tiene traccia delle implementazioni fatte dagli sviluppatori nel main branch richiedendo la pull request
- Milestones: teniamo traccia delle funzionalità implementate a gruppi con le Milestones, che aiutano anche per il versioning del prodotto finale

Modello AGILE

Con questo metodo siamo riusciti a gestire variabili (come: tempo, risorse e qualità) in modo molto più efficiente che con altre metodologie. Nasce come bisogno di contrapporsi con i modelli tradizionali come il modello a spirale e il modello a cascata che sono spesso bloccanti in alcune fasi dello sviluppo. Questa nuova metodologia pensa il software in modo leggero e ci permette di raggiungere grande efficienza con l'utilizzo di best-practice. I quattro valori agili importanti sono:

- Persone e interazioni sono più importanti del processo e dei tools utilizzati.
- Un software funzionante è più importante di una chiara documentazione.
- La collaborazione del cliente va aldilà del contratto stipulato.
- La capacità di rispondere a cambiamenti è una caratteristica fondamentale.

Può succedere spesso che vengano imposte delle modifiche al prodotto:

bisogna adottare un processo in grado di apportare continui cambiamenti a ciò che stiamo realizzando.

Durante la progettazione di un servizio applicativo per la gestione della coda, il team di sviluppo ha adottato i principi del modello Agile per garantire un processo efficace ed efficiente. Di seguito viene fornita una descrizione descrittiva delle pratiche Agile utilizzate:

1. Auto-organizzazione: Ci siamo organizzati con incontri sistematici e programmati tramite calls e scambio di feedback.
2. Test e feedback continui: Durante lo sviluppo abbiamo continuamente controllato il funzionamento delle features implementate in modo da non perdere la struttura organizzativa del progetto

3. Iterazione e miglioramento continuo: Abbiamo pianificato il lavoro in iterazioni, seguendo un approccio di implementazione continua. Dopo ogni iterazione, abbiamo chiuso le attività dell'iterazione con un controllo sul lavoro svolto
4. Prioritizzazione e gestione del backlog: Abbiamo adottato una pratica di prioritizzazione e gestione del backlog per gestire in modo efficace le funzionalità da sviluppare. Il backlog è stato creato tramite funzionalità di Issues/Milestone presente sulla piattaforma GitHub
5. Adozione di strumenti collaborativi: Abbiamo utilizzato strumenti collaborativi per agevolare la comunicazione e la condivisione delle informazioni. Abbiamo utilizzato i seguenti strumenti
 - GitHub: per la gestione delle implementazioni delle features
 - Discord: per l'organizzazione e lo svolgimento degli incontri
 - OneDrive: come storage condiviso di supporto alle attività di sviluppo

Iterazione 0

0.1 Analisi dei requisiti

Abbiamo svolto questa fase di analisi cercando di immaginare tutti gli scenari possibili sia dal punto di vista del paziente, sia dal punto di vista dell'operatore ospedaliero. Per raggiungere questo obiettivo abbiamo usato la tecnica degli Use Case Stories, grazie alla quale abbiamo preso ogni attore separatamente e abbiamo individuato le funzionalità che l'applicazione dovrà avere.

0.2 Attori

L'analisi degli Use Case ha reso necessario individuare gli attori coinvolti nell'utilizzo del sistema. Gli attori identificati vengono divisi in attori primari e secondari e sono i seguenti:

- Utente standard (primario)
- Utente operatore che si occupa delle richieste degli utenti standard (primario)
- Sistema di visualizzazione della coda (secondario)
- Sistema di autenticazione e profilazione (secondario)
- Sistema di visualizzazione (secondario)
- Monitor (secondario)

0.3 Use case stories

0.3.1 Utente standard

Generazione biglietto

- Un utente che chiede il biglietto vuole avere il numero assegnato

- Un utente può voler scegliere tra code di più servizi
- Un utente vuole stampato il biglietto
- Un utente lo vuole visualizzato sullo schermo della propria app

Registrazione profilo utente

- Un utente può effettuare la registrazione con le proprie informazioni.
- Un utente può richiedere la generazione di una nuova password se ha smarrito quella attuale.

Visualizzazione della dashboard (coda)

- Un utente può voler visualizzare la coda su schermo
- Un utente vuole sapere qual è il suo turno guardando sullo schermo
- Un utente vuole sapere a che sportello è stato assegnato
- Un utente può voler visualizzare il tempo stimato per il suo turno

Gestione della coda (coda prioritaria)

- Un utente può voler generare un biglietto di coda prioritaria anche se non si trova nell'ufficio degli operatori
- Un utente può voler generare un biglietto di coda prioritaria per velocizzare il procedimento
- Un utente può voler guardare la coda attuale senza essere nell'ufficio

Login/logout

- Un utente può voler accedere tramite login per ottenere un biglietto nella coda prioritaria

Gestione profilo utente

- Un utente può voler visualizzare il tempo stimato per il suo turno
- Un utente può voler visualizzare una cronologia di prenotazioni generate all'interno del proprio profilo

- Un utente può voler visualizzare/modificare i suoi dati
- Un utente può richiedere una nuova password

0.3.2 Utente operatore

Gestione della coda

- Un operatore vuole ricevere il numero che deve servire quando passa al successivo
- Un operatore vuole che la coda visualizzata scorra quando seleziona il successivo
- Un operatore, per selezionare il successivo, schiaccia un pulsante

0.3.3 Sistema di gestione della coda

Gestione della coda

- Elabora l'avanzamento della coda
- Genera biglietti standard
- permette l'accesso alla funzionalità di coda prioritaria

0.3.4 Sistema di autenticazione

Gestione della coda

- Il sistema di autenticazione permette l'accesso alla funzionalità di coda prioritaria

0.3.5 Sistema di visualizzazione

Visualizzazione della dashboard (coda)

- Deve visualizzare i biglietti serviti attualmente dagli operatori

- Deve visualizzare i biglietti immediatamente successivi in coda (alcuni)
- Deve visualizzare correttamente lo scorrere della coda

0.4 Use Case Diagram

Riportiamo tutti gli use case specificati nello schema per meglio capire le associazioni tra gli attori e l'inglobamento di varie funzionalità nei sotto-sistemi.

Da questo schema si intravedono tutte le funzionalità del sistema complessivo

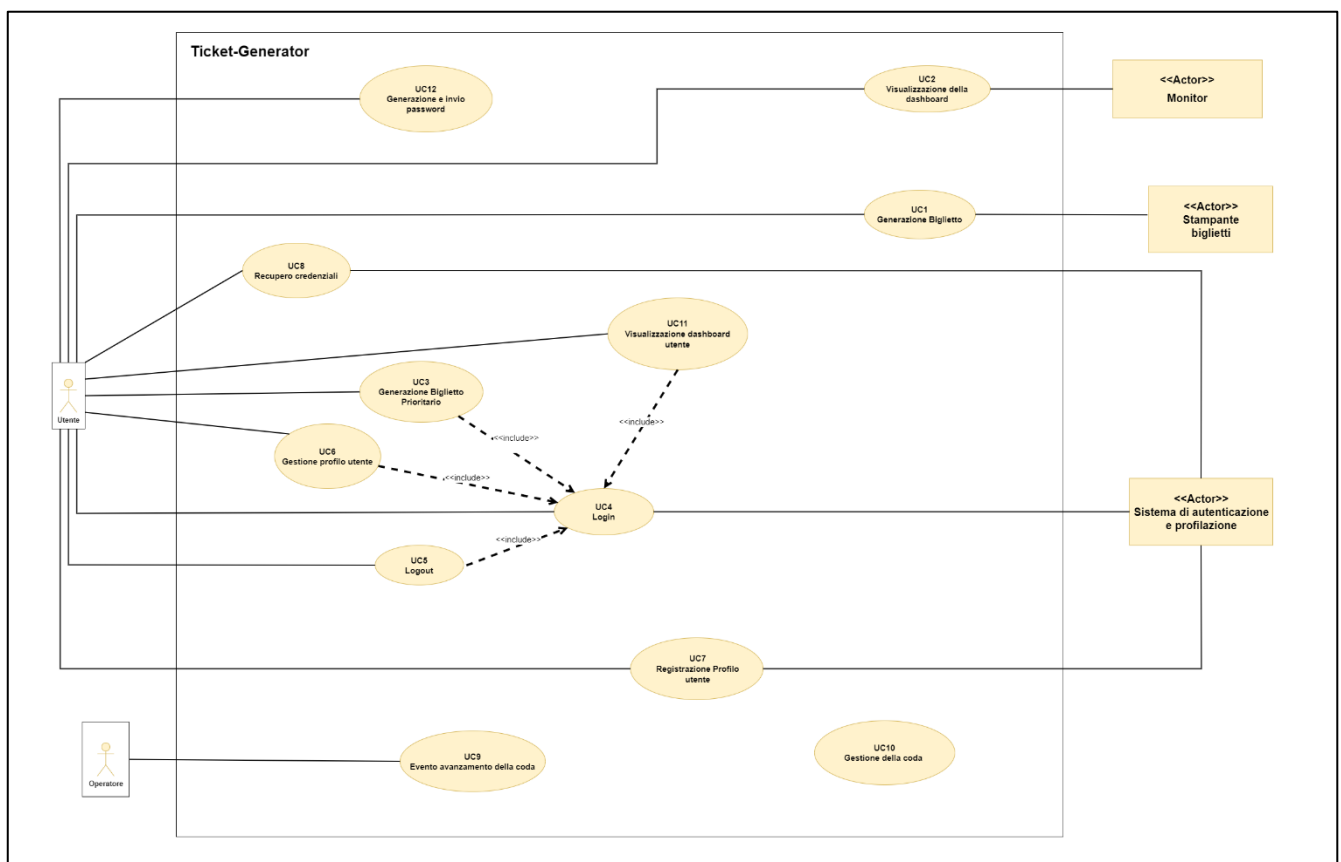


Figura 1: Use Case Diagram

0.5 Use Case Description (descrizione degli use case)

0.5.1 UC1: Generazione biglietto

UseCase	Generazione biglietto
Summary	Generazione del biglietto per l'utente
Actor	Utente, Stampante biglietti
Precondition	L'utente non ha il biglietto
Postcondition	L'utente ha il biglietto
Base sequence	<ol style="list-style-type: none">1. L'utente inizializza la procedura di generazione a schermo2. Dopo aver selezionato il servizio, conferma la scelta3. Il sistema stampa/visualizza a schermo il biglietto generato
Branch sequence	
Exception sequence	
Sub UseCase	
Note	

0.5.2 UC2: Visualizzazione della dashboard della coda

UseCase	Visualizzazione della dashboard (coda)
Summary	Visualizzazione delle informazioni della coda
Actor	Utente, Monitor
Precondition	
Postcondition	
Base sequence	<ol style="list-style-type: none">1. L'utente dopo aver ricevuto il biglietto attende il proprio turno2. Il turno può avvenire quando a schermo viene visualizzato il biglietto con un operatore assegnato. In particolare, sulla dashboard vengono visualizzati per ogni riga<ul style="list-style-type: none">- Biglietto- Operatore (se presente)- Tempo stimato
Branch sequence	
Exception sequence	
Sub UseCase	
Note	

0.5.3 UC3: Generazione biglietto prioritario

UseCase	Generazione Biglietto prioritario
Summary	Generazione di un biglietto prioritario
Actor	Utente
Precondition	L'utente ha fatto il login
Postcondition	
Base sequence	<ol style="list-style-type: none">1. L'utente inizializza l'operazione di generazione da app2. L'utente visualizza il tempo stimato3. L'utente conferma la generazione, visualizzando a schermo il biglietto generato
Branch sequence	
Exception sequence	
Sub UseCase	Log-in
Note	

0.5.4 UC4: Log-in

UseCase	Log-in
Summary	Accesso al sistema di priorità
Actor	Utente, Sistema di autenticazione e profilazione
Precondition	L'utente è registrato
Postcondition	
Base sequence	<ol style="list-style-type: none">1. L'utente inizializza la procedura di accesso inserendo i propri dati di autenticazione2. L'utente visualizza a schermo la riuscita dell'operazione
Branch sequence	
Exception sequence	<ol style="list-style-type: none">1. Viene generato un errore di autenticazione poiché l'utente o non è registrato o ha sbagliato l'inserimento dei dati2. L'utente visualizza a schermo la nuova richiesta di dati di autenticazione
Sub UseCase	
Note	

0.5.5 UC5: Log-out

UseCase	Log-out
Summary	Uscita dal sistema di priorità
Actor	Utente
Precondition	L'utente è loggato
Postcondition	
Base sequence	<ol style="list-style-type: none">1. L'utente inizializza la procedura schiacciando il tasto di logout2. L'utente viene riportato alla pagina di autenticazione
Branch sequence	
Exception sequence	
Sub UseCase	
Note	

0.5.6 UC6: Gestione profilo utente

UseCase	Gestione profilo utente
Summary	Modifica e visualizzazione dei dati utente
Actor	Utente
Precondition	L'utente ha fatto il login
Postcondition	
Base sequence	<ol style="list-style-type: none">1. L'utente clicca sul pulsante adibito alla visualizzazione del proprio profilo2. La schermata mostra le informazioni inserite in fase di registrazione Viene mostrato un pulsante per abilitare la modifica del form, oppure tornare indietro3. L'utente alla fine torna alla schermata principale
Branch sequence	<p>B1:</p> <ol style="list-style-type: none">3. L'utente ha cliccato sulla modifica del form abilitandola4. L'utente dopo aver compilato il form può decidere se confermare le modifiche o tornare alla schermata di visualizzazione
Exception sequence	
Sub UseCase	
Note	

0.5.7 UC7: Registrazione profilo utente

UseCase	Registrazione profilo utente
Summary	Per l'accesso al sistema di priorità da parte dell'utente
Actor	Utente, Sistema di autenticazione e profilazione
Precondition	L'utente ha richiesto la registrazione
Postcondition	
Base sequence	<ol style="list-style-type: none">1. Un utente richiede la registrazione tramite app2. L'utente compila un apposito form3. L'utente può accedere ai servizi. [E1: utente già registrato]
Branch sequence	
Exception sequence	E1: <ol style="list-style-type: none">3. L'utente è già registrato4. Viene annullata la registrazione
Sub UseCase	
Note	

0.5.8 UC8: Recupero credenziali

UseCase	Recupero credenziali
Summary	
Actor	Utente, Sistema di autenticazione
Precondition	L'utente ha compilato il form
Postcondition	Il sistema cambia la password scelta dall'utente
Base sequence	<ol style="list-style-type: none">1. Il sistema controlla la validità del form [E1: dati non validi]2. L'utente inserisce una nuova password3. Se la password è valida viene cambiata
Branch sequence	
Exception sequence	<p>E1:</p> <ol style="list-style-type: none">2. I dati da inserire non sono validi3. Il sistema restituisce un errore concorde
Sub UseCase	Registrazione profilo utente
Note	

0.5.9 UC9: Evento di avanzamento della coda

UseCase	Evento di avanzamento della coda
Summary	Funzionalità all'interno del sistema di gestione della coda disponibile per farla avanzare
Actor	Operatore
Precondition	L'operatore ha schiacciato
Postcondition	La coda è avanzata
Base sequence	<ol style="list-style-type: none">1. L'operatore segnala la propria disponibilità schiacciando il pulsante2. L'operatore riceve il biglietto da servire e viene visualizzata a schermo l'assegnazione all'operatore [E1: coda vuota]3. Dalla visualizzazione della coda viene cancellato il biglietto che stava servendo
Branch sequence	
Exception sequence	E1: <ol style="list-style-type: none">2. Non c'è nessuno in coda (non ci sono utenti che hanno richiesto un biglietto), l'operatore riceve un messaggio di default
Sub UseCase	
Note	

0.5.10 UC10: Gestione della coda

UseCase	Gestione della coda
Summary	Funzionalità del sistema di gestione della coda su evento dell'operatore
Actor	Sistema di gestione della coda
Precondition	
Postcondition	La coda è avanzata
Base sequence	<ol style="list-style-type: none">1. Il sistema riceve il numero operatore successivamente alla pressione del pulsante da parte di esso2. A questo punto controlla la coda, eliminando un biglietto associato se presente3. Assegna il primo biglietto disponibile (non ha un operatore associato) all'operatore della richiesta
Branch sequence	
Exception sequence	E1: <ol style="list-style-type: none">3. Non c'è nessuno in coda (non ci sono utenti che hanno richiesto un biglietto), l'operatore riceve un messaggio di default
Sub UseCase	Evento di avanzamento della coda
Note	

0.5.11 UC11: Visualizzazione dashboard utente

UseCase	Visualizzazione dashboard utente
Summary	Visualizzazione delle prenotazioni, visualizzazione tempo stimato
Actor	Utente
Precondition	L'utente ha fatto il login
Postcondition	
Base sequence	<ol style="list-style-type: none">1. L'utente clicca sul pulsante adibito alla visualizzazione delle proprie prenotazioni2. La schermata mostra le prenotazioni effettuate come una lista. Le prenotazioni ancora valide mostreranno un tempo stimato, mentre quelle passate no3. L'utente alla fine torna alla schermata principale
Branch sequence	
Exception sequence	
Sub UseCase	
Note	

0.6 Analisi delle specifiche

0.6.1 Introduzione

Le specifiche descritte sono state ordinate in modo da implementare prima le funzionalità necessarie (e quindi vincoli) alle successive funzionalità del software. Sono quindi divise in specifiche con priorità alta e bassa.

0.6.2 Specifiche funzionali

- R1 Generazione biglietto in sede: l'applicazione possiede la capacità di generare un biglietto per l'utente in sede nell'ufficio. Implica che il biglietto venga fornito e visualizzato dall'utente e inserito nella lista d'attesa del sistema in base anche al tipo richiesto al momento della generazione
- R2 Visualizzazione coda: nella sede è presente un device adibito alla visualizzazione della coda generata. Il sistema sviluppato ha bisogno quindi di questa funzionalità da utilizzare nel device per la visualizzazione agli utenti in coda che hanno richiesto e ottenuto il biglietto
- R3 Avanzamento coda (operatore): il sistema deve poter scalare su richiesta dell'operatore, che richiederà l'operazione quando termina il servizio con l'utente precedentemente servito. Quindi il sistema seleziona il prossimo biglietto da servire in base alla logica progettata
- R4 Profilazione utente: il sistema tiene traccia degli utenti se richiesto, tramite un sistema di profilazione affidato all'applicazione. La profilazione viene poi utilizzata per le successive funzionalità
- R5 Log in/Log out e registrazione utente al sistema online: l'applicazione permette l'accesso esclusivo all'utente che lo richiede
- R6 Generazione biglietto non in sede: l'applicazione permette di generare e tenere traccia dei biglietti tramite applicazione che comunica con il sistema in sede, così da permettere una generazione non in sede

Codice	Nome	Priorità
R1	Generazione biglietto in sede	B
R2	Visualizzazione coda	B
R3	Avanzamento coda (operatore)	A
R4	Profilazione utente	C
R5	Log in/Log out e registrazione	A
R6	Generazione biglietto non in sede	A

Tabella 1: Requisiti funzionali

0.6.3 Logica di funzionamento delle code

In particolare, definiamo la business logic dietro il sistema di espletamento della coda, che avviene alla pressione da parte dell'operatore del pulsante (segnalazione di disponibilità).

Questa funzionalità viene solo abbozzata ma iniziamo a considerare un buon funzionamento per questa funzionalità di sistema fondamentale

- Accettazione (di una visita già prenotata) -> priorità/biglietto A
- Priorità o biglietto online -> priorità/biglietto X
- Prenotazione (con ricetta oppure senza) -> priorità/biglietto B
- Ritiro referti -> priorità/biglietto C1
- Registrazione nel sistema di login priorità -> priorità/biglietto C2

Su 10 serviti, 4 sono accettazione, 3 sono priorità, 2 sono prenotazione, 1 sono in coda C.

Il sistema dovrà quindi ripartire i serviti sulla base di questa suddivisione

0.7 Analisi dell'architettura

Abbiamo realizzato due deployment diagram: il primo serve per avere un'idea generale dell'intero sistema, mentre il secondo per mostrare nel dettaglio la struttura delle componenti software su quelle hardware.

0.7.1 Deployment diagram - Architettura

Questo deployment diagram mostra i dispositivi che vengono utilizzati e come interagiscono fra di loro. L'architettura è formata da un server centrale che interagisce con un database e con le seguenti componenti hardware:

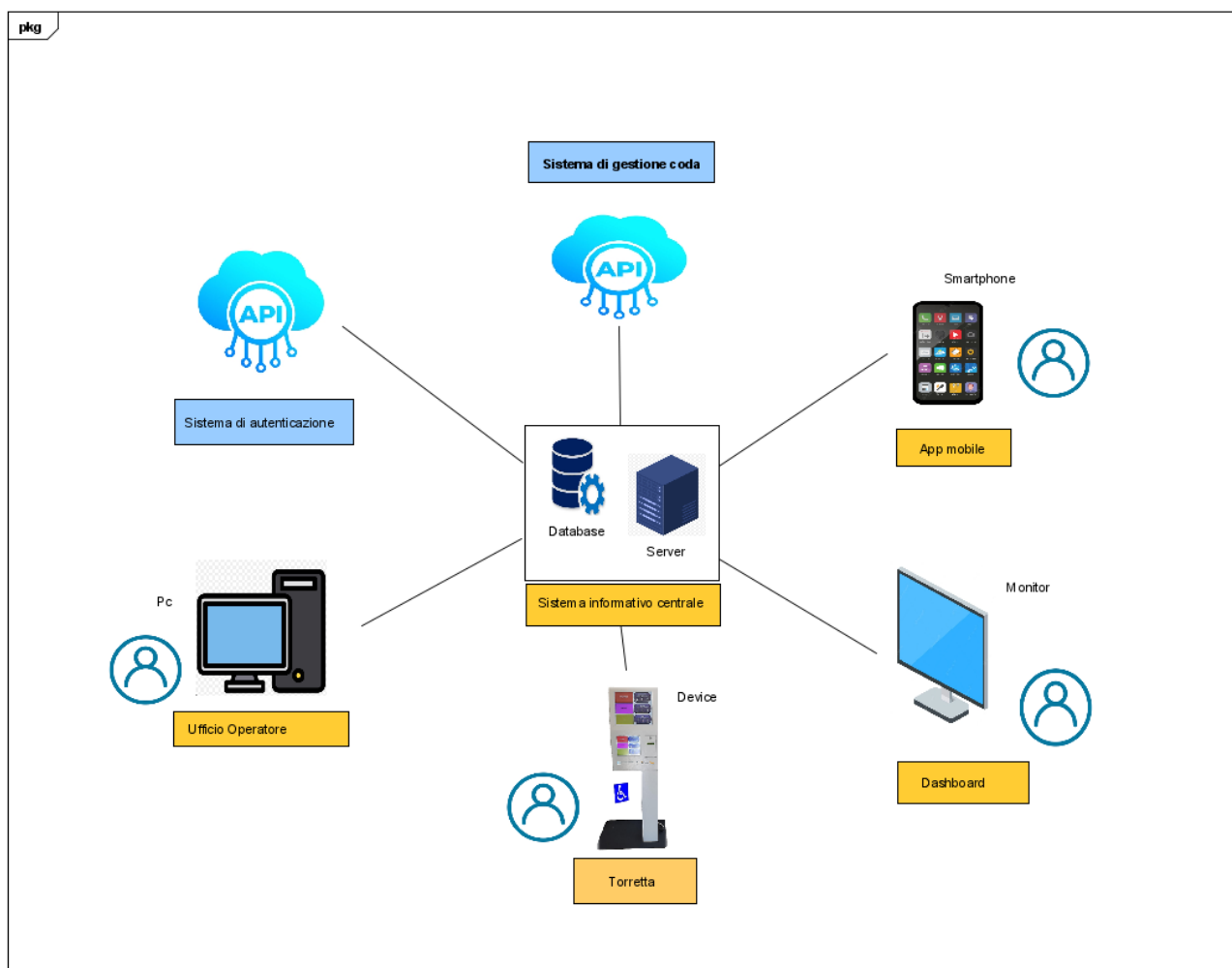


Figura 2: Deployment diagram informale

- Ufficio operatore, dove l'utente può richiedere la procedura di registrazione
- Dashboard, per vedere la coda attuale
- App mobile, per prenotare il biglietto nella coda prioritaria e per vedere le informazioni personali.
- Sistema di gestione coda, che gestisce l'avanzamento della coda.
- Sistema di autenticazione, che gestisce l'autenticazione degli utenti.

0.7.2 Deployment diagram – UML

Questo diagram approfondisce le informazioni contenute nel diagram precedente, specificando la tecnologia di comunicazione utilizzata.

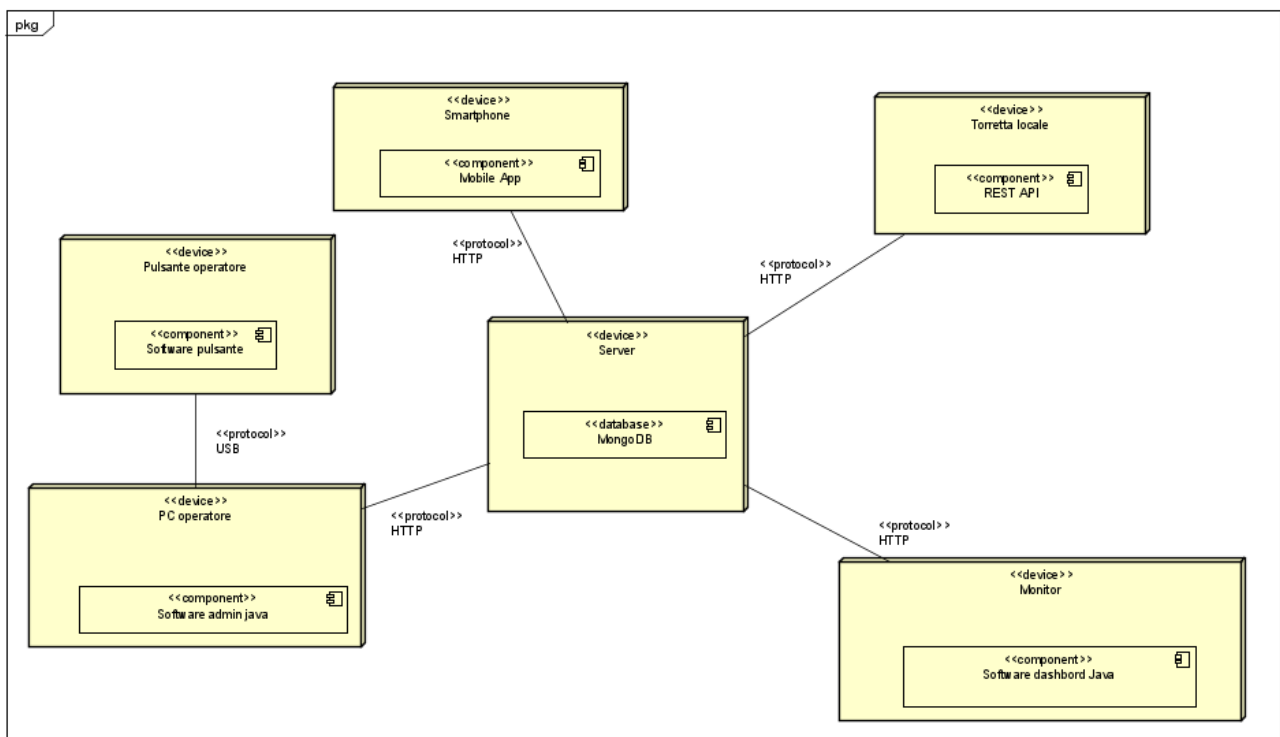


Figura 3: Deployment diagram UML

Iterazione 1

1.1 Component diagram – Interfacce

Il diagramma delle interfacce mostra come i componenti del sistema comunicano ed interagiscono tra di loro per implementare le funzionalità richieste dall'analisi dei requisiti.

Le interfacce definite non sono definitive (presentano infatti un nome abbozzato) ma semplicemente prototipate per adattarsi poi durante lo sviluppo e le iterazioni successive del processo agile.

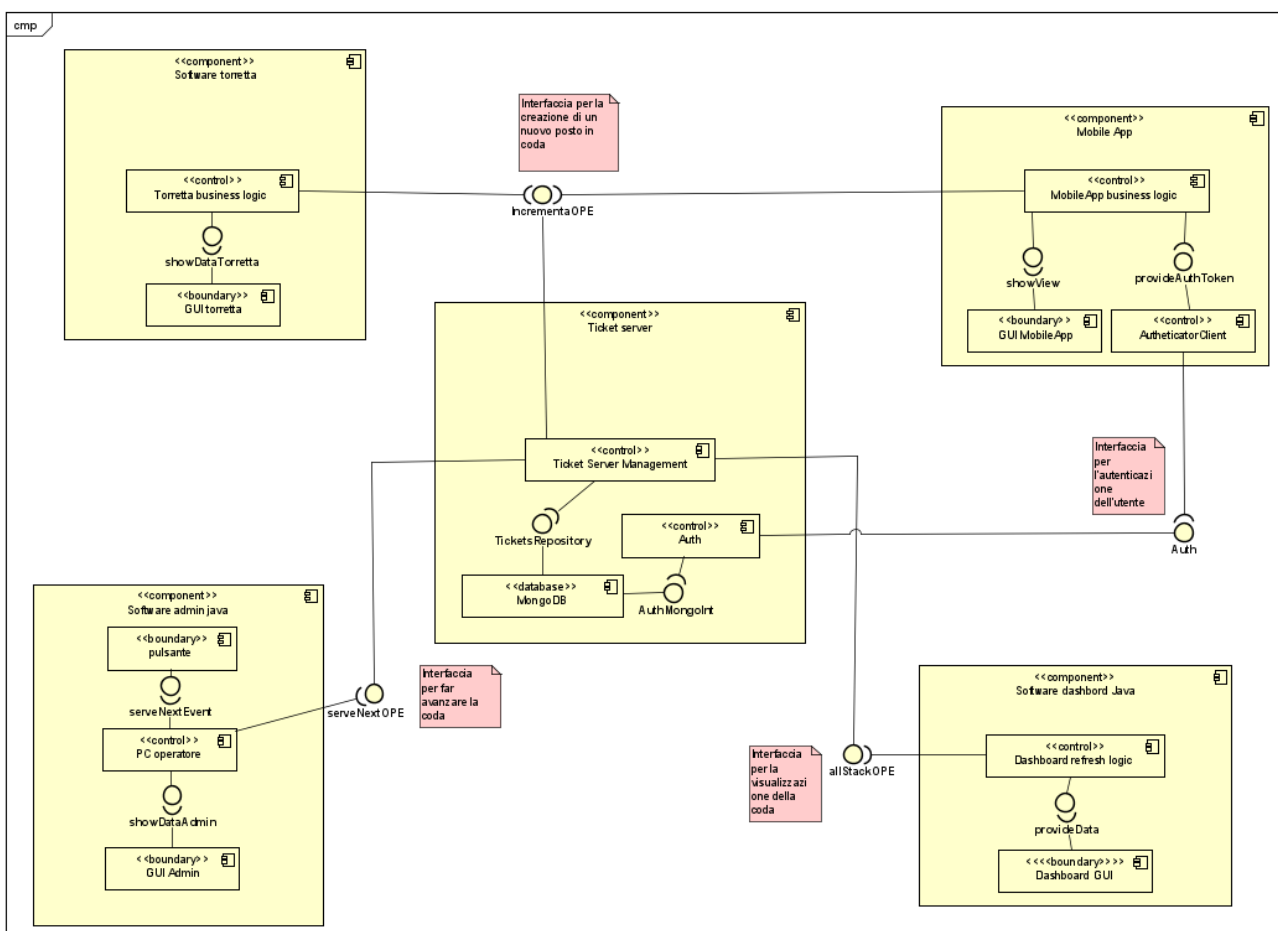


Figura 4: Component diagram delle interfacce

1.2 Class diagram - Interface definition

Le interfacce abbozzate del component diagram vengono definite più nello specifico in questo diagramma, dove specifichiamo per ogni interfaccia i metodi necessari per implementare la funzionalità tra i componenti che la utilizzano

Anche qui le firme dei metodi non sono completamente definite, in modo che vengano specificate al bisogno durante le iterazioni successive, nel processo di sviluppo del software

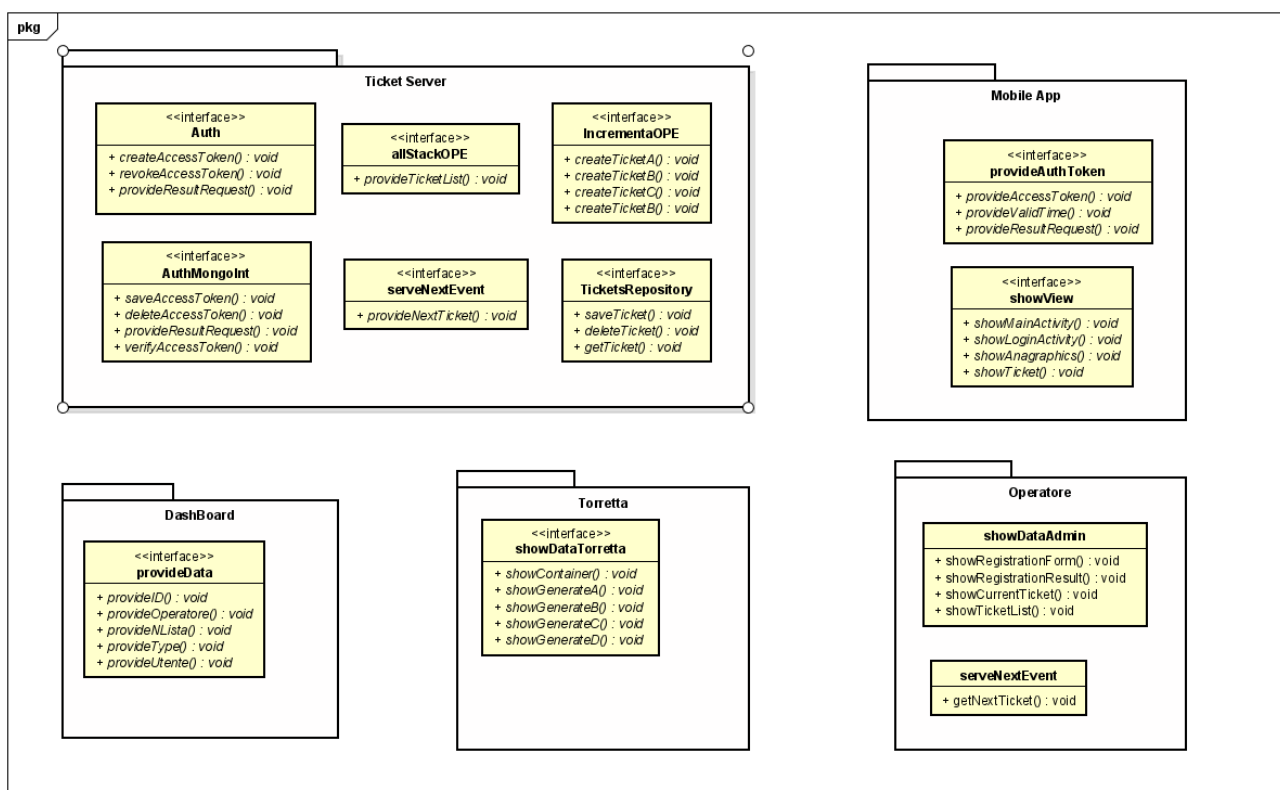


Figura 5: Class diagram, interface definition

1.3 Class diagram - Data type (presentation model)

Rappresenta le relazioni statiche tra gli oggetti che contengono informazioni, così come una possibile struttura della base di dati.

Abbiamo utilizzato una particolare struttura di ereditarietà ed interfacce perché nelle iterazioni successive vogliamo, con un unico oggetto, scambiare sia l'input di un componente che l'output, in base all'API che verrà chiamata. Il principio è che un componente mittente crea l'oggetto e ne popola l'input, mentre un componente destinatario lo riceve e ne popola l'output, restituendolo al mittente.

In questo modo possiamo facilmente tenere traccia delle richieste all'interno del sistema e monitorarne la logica di funzionamento

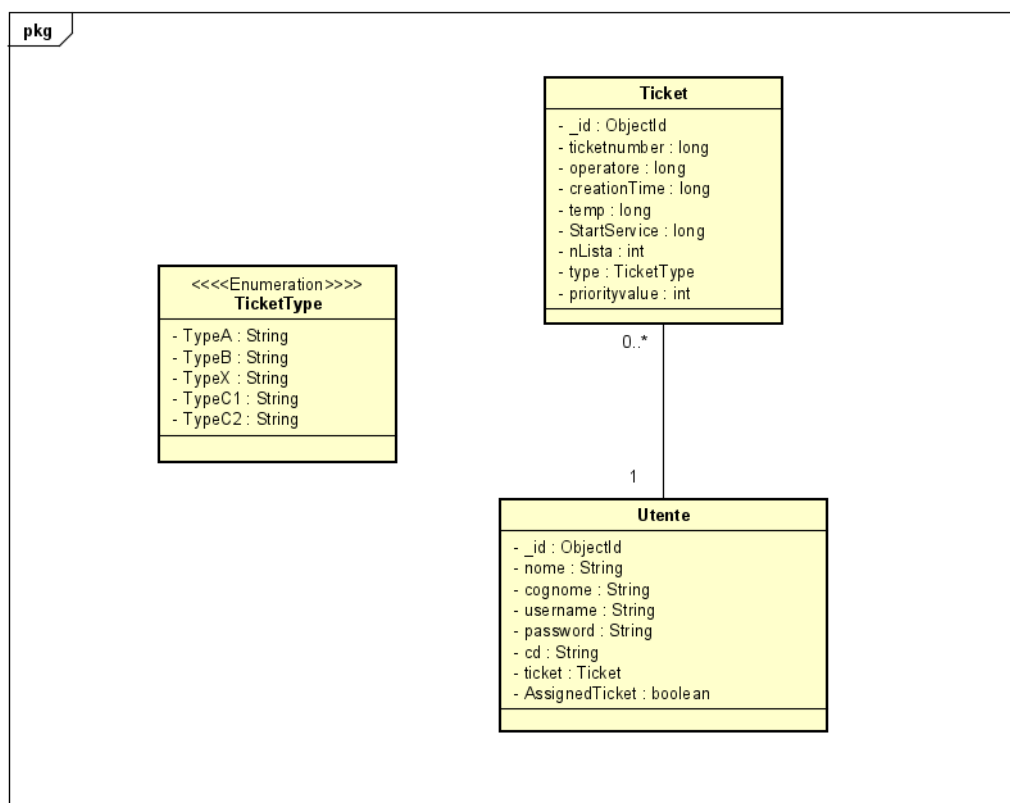


Figura 6: Class diagram, data type

Iterazione 2

2.1 Tecnologie utilizzate

Una volta decisa l'architettura di base nelle fasi iniziali, abbiamo pensato alle migliori tecnologie per implementarne il funzionamento, con un occhio di riguardo verso le novità del momento

2.1.1 Database - MongoDB

La decisione è ricaduta su un database NoSQL per via dell'estrema libertà di utilizzo che consente, un DBMS non relazionale che facilita la memorizzazione degli oggetti Json del backend. L'hosting è a cura di MongoDB Atlas, anche se in fase di deployment è anche possibile utilizzare una macchina con Docker Desktop con un'immagine di MongoDB

2.1.2 Framework Spring

Il framework Spring è un framework open source per lo sviluppo di applicativi Java, in particolare per quanto riguarda la parte backend. All'interno del nostro progetto utilizziamo maggiormente due delle sue componenti

- Spring MVC: per l'implementazione delle API REST esposte dalla parte server, utilizzate sia per la componente di autenticazione, sia per la parte di funzionalità
- Spring Data: mette a disposizione dei driver già implementati per la comunicazione con i DBMS. Abbiamo utilizzato questa funzionalità come driver di comunicazione con MongoDB Atlas

2.1.3 Repository - GitHub

La tecnologia impiegata per il repository è Git, su hosting GitHub. Questo strumento è fondamentale sia per il versioning del progetto, sia per parallelizzare il lavoro su più collaboratori. In particolare, abbiamo utilizzato alcune delle funzionalità di GitHub per scandire la fase di sviluppo del software, come illustrato più avanti

2.1.4 Testing - Postman

Per il testing a lavori in corso delle API esposte dal backend abbiamo utilizzato il software Postman, che si occupa di simulare le richieste inviate dai client verso il server e di riceverne la risposta. In questo modo abbiamo velocizzato lo sviluppo dell'elemento portante del progetto, testando la parte backend, senza avere a disposizione i client che sono stati poi successivamente “cuciti” sulle API del backend.

2.2 Scelte di implementazione

In questa iterazione ci occupiamo dell'implementazione dei servizi cardini nel sistema progettato, quindi la generazione del biglietto (IncrementaTicket), la restituzione di tutta la lista dei biglietti (AllStack), l'aggiornamento delle priorità (UpdateTickets), la funzionalità per passare all'utente successivo (ServeNext), facenti parte delle specifiche funzionali R1, R2 e R3.

In particolare, trattiamo i seguenti Use case:

- UC1: Generazione del biglietto
- UC2: Visualizzazione della dashboard (anche per UC14: visualizzazione dashboard utente)

- UC11: Evento di avanzamento della coda (anche per UC12: gestione della coda)

Visto che si rende necessaria la persistenza dei biglietti generati, passo fondamentale è implementare il loro salvataggio sulla base di dati scelta prima di procedere all'implementazione della logica

2.3 Component diagram TicketsRepository

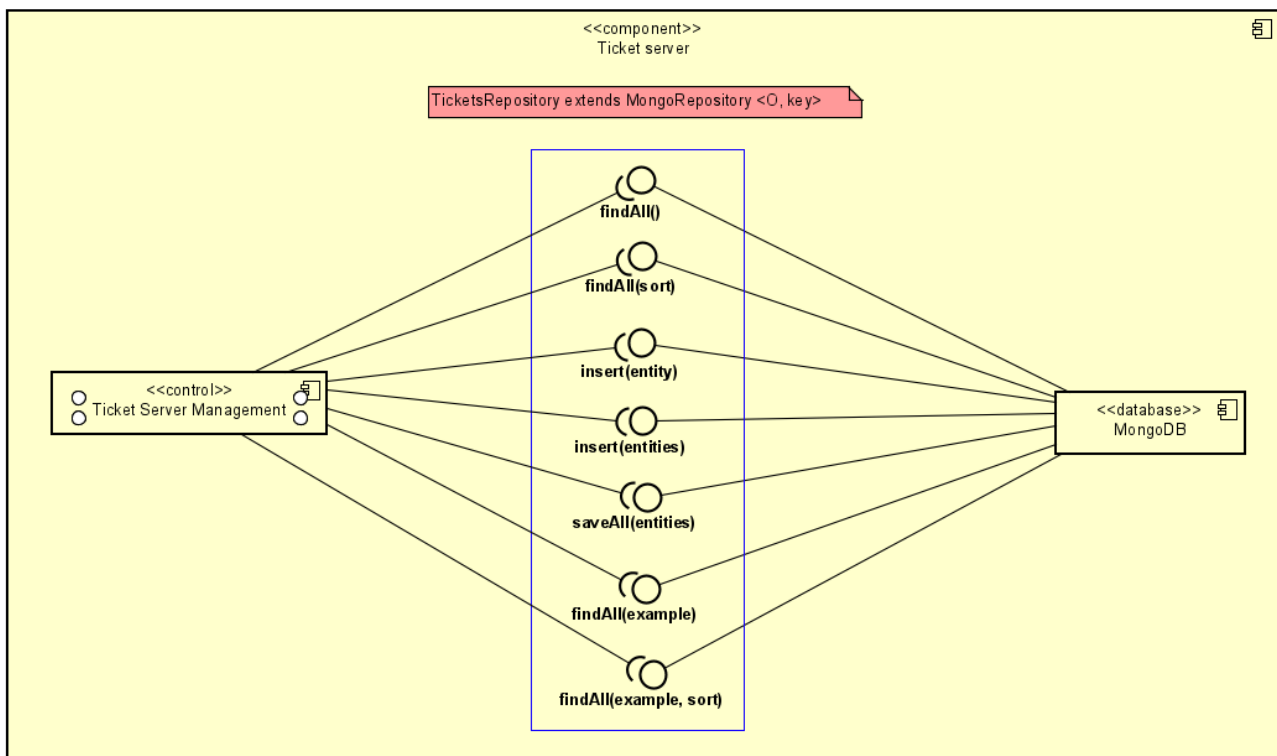


Figura 7 Component Diagram TicketsRepository

Per la comunicazione con il database MongoDB abbiamo sfruttato l'interfaccia e l'implementazione integrata messa a disposizione dal framework Spring.

In particolare, è così definita come mostrato nell'immagine Figura 7

In questo modo non dobbiamo preoccuparci dell'implementazione, basta definire solo gli oggetti dell'interfaccia, successivamente Spring istanzierà il driver di comunicazione su misura dei nostri oggetti scambiati

2.4 Sequence diagram di esecuzione dei servizi

Il sequence diagram, raffigurato in questa sezione, descrive il comportamento di client, server e database quando il client fa una richiesta al server, per ottenere la lista completa di tutti i biglietti in coda per poterli visualizzare a schermo.

Il client, quindi, effettua questa richiesta al server tramite API Rest.

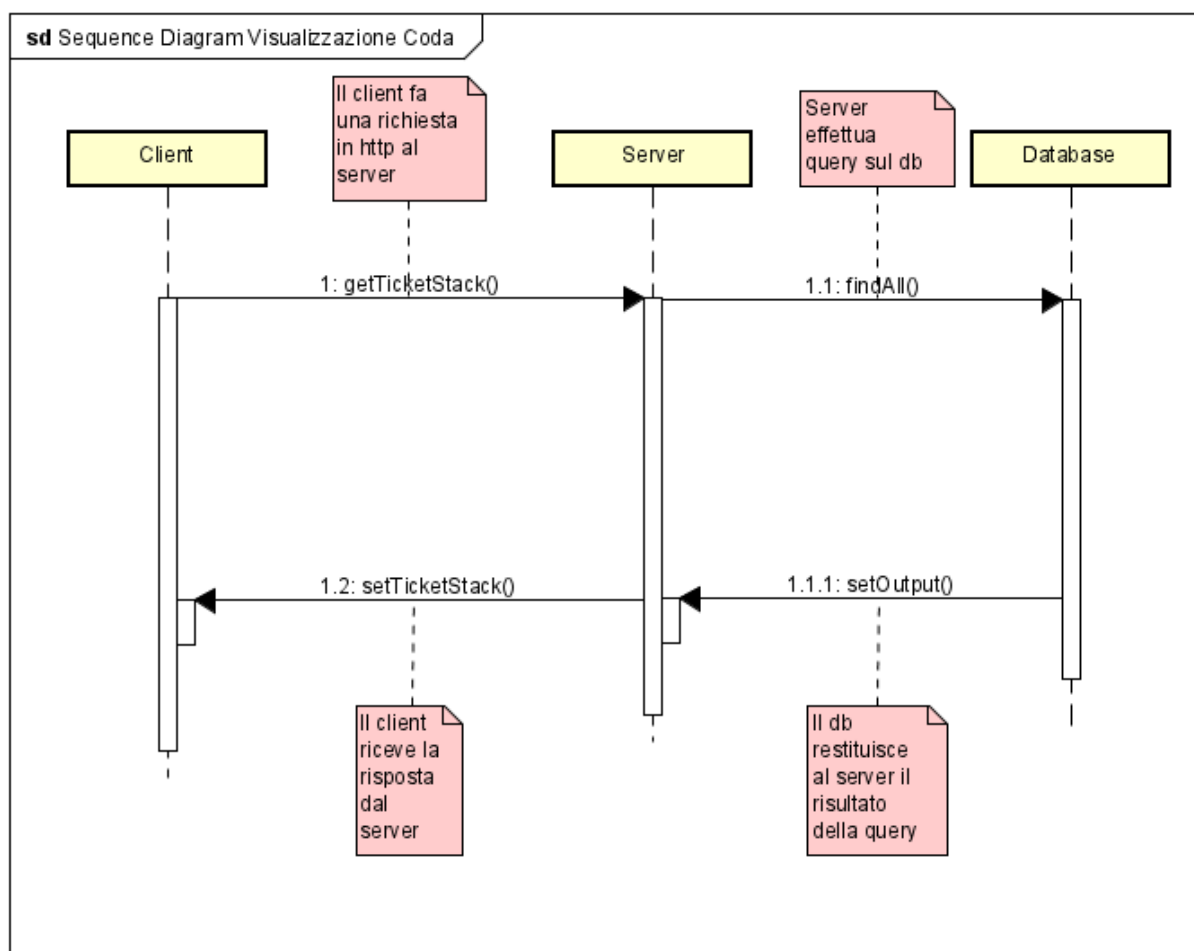


Figura 8 Sequence Diagram Visualizzazione Coda

Il server a sua volta effettua una query sul database e aspetta di ricevere i risultati. Una volta ricevuti i dati sotto forma di Json, li invia al client che è rimasto in attesa e successivamente dopo aver effettuato il dovuto parsing si occupa di visualizzare i dati ottenuti.

2.5 Metodo di gestione della coda

Per gestire una coda che gestisce 5 biglietti (A, X, B, C1, C2) con priorità decrescente, è stato utilizzato un algoritmo di gestione delle code basato sul concetto di "tempo di vita". Questo algoritmo consiste nell'attribuire un "tempo di vita" ad ogni biglietto, ovvero un intervallo di tempo entro cui quel biglietto deve essere servito altrimenti scatta l'aumento della priorità in modo da andare ad evitare l'attesa infinita per alcune tipologie di biglietto. In particolare, i biglietti con priorità maggiore avranno un tempo di vita più breve rispetto a quelli con priorità minore.

L'algoritmo funziona come segue:

1. Inizializza la coda con i 5 biglietti (A, X, B, C1, C2) a cui sono assegnate le seguenti priorità (4,3,2,1,1), assegnando loro i rispettivi tempi di vita in base alla loro priorità decrescente: A avrà il tempo di vita più breve (ad esempio, 5 minuti), seguito da B (ad esempio, 10 minuti), C (ad esempio, 15 minuti), D (ad esempio, 20 minuti) ed infine E (ad esempio, 25 minuti).
2. Quando un cliente si presenta allo sportello, gli viene assegnato il biglietto in base al servizio che deve usufruire.
3. Se un cliente con una priorità più alta si presenta durante l'attesa di un cliente con una priorità più bassa, il cliente con la priorità più alta viene servito prima.

4. Se un biglietto ha esaurito il suo tempo di vita e non è stato ancora servito, gli viene aumentato il valore di priorità fino a che raggiunge un valore massimo.
5. La contesa tra biglietti con stesso valore di priorità viene risolta con algoritmo FIFO.

Questo algoritmo garantisce che tutti i biglietti vengano eventualmente serviti, evitando attese infinite per i biglietti con priorità più bassa, che comunque hanno un tempo di vita limitato e proporzionale alla loro priorità. Inoltre, il decremento dei tempi di vita in modo proporzionale al tempo trascorso permette di evitare di far attendere troppo i clienti.

2.5.1 Pseudocodice della classe **ServeNextOPE()** e analisi complessità e metodo **execute()**

La classe **ServeNextOPE()** rappresenta un servizio che viene utilizzato per gestire la coda dei biglietti e assegnare un biglietto al prossimo operatore disponibile.

Il metodo **execute** è composto da diverse operazioni che influenzano la sua complessità. Inizialmente, il metodo recupera tutti i ticket presenti nel repository tramite l'operazione **ticketsRepository.findAll()**.

La complessità di questa operazione dipende dal numero totale di ticket nel repository e può essere considerata come $O(N)$, dove N rappresenta il numero di ticket. Successivamente, viene eseguito un controllo per verificare se esiste un ticket assegnato all'operatore corrente tramite il metodo **anyMatch()** che scansiona la pila e ha complessità $O(N)$.

Se presente, il ticket viene rimosso dalla lista dei ticket attraverso il metodo nidificato **ticketsRepository.delete(n)** che ha complessità $O(N)$. La complessità di questa parte di codice è quindi $O(N^2)$.

Successivamente, il metodo itera attraverso la lista dei ticket per trovare il primo ticket disponibile per l'operatore corrente.

Questa iterazione richiede di scorrere l'intera lista dei ticket, quindi, ha una complessità lineare $O(N)$, dove N è il numero di ticket.

Alla fine del metodo, viene eseguita un'operazione **`ticketsRepository.saveAll(pila)`** per salvare i ticket aggiornati nel repository.

La complessità di questa operazione dipende dal numero totale di ticket presenti nella lista pila e può essere considerata come $O(N)$, dove N rappresenta il numero di ticket.

Considerando queste operazioni, possiamo affermare che la complessità complessiva del metodo `execute` è influenzata principalmente dalle operazioni di rimozione dei ticket che ha complessità $O(N^2)$.

Pertanto, la complessità del metodo `execute` può essere approssimata come $O(N^2)$, dove N rappresenta il numero totale di ticket presenti nel repository.


```

public ServeNextCb.0 execute(ServeNextCb.I i) {
    //se il token è valido l'autenticatore restituisce true
    se autenticatore.autenticazione(i.getToken()) è vero:
        ServeNextCb.0 out = nuovo ServeNextCb.0()
        long op = i.getNumero()
        List<Ticket> pila = ticketsRepository.findAll()

        // elimino il biglietto assegnato precedentemente (se esiste) all'operatore perchè ha finito di servirlo
        se pila.stream().anyMatch(n -> n.getOperatore() == op) è vero:
            pila.removeSe(n -> {
                se n.getOperatore() == op:
                    ticketsRepository.delete(n)
                    restituisci vero
                altrimenti:
                    restituisci falso
            })

        //assegno il biglietto in base alla priorità all'operatore che ha fatto richiesta
        Ticket temp = nuovo Ticket()
        se pila non è vuota:
            pila.forEach(n -> {
                se n.getOperatore() == 0:
                    // assegno il primo biglietto disponibile all'operatore oppure controllo la priorità con il successivo
                    se temp.get() == null oppure n.comparePriority(temp.get()) è vero:
                        temp.set(n)
            })

        // restituisco il biglietto, se è stato trovato nella coda
        Ticket insert = temp.get()
        se insert non è nullo:
            insert.assegnaOp(cast(int) op)
            out.setBiglietto(insert)
            ticketsRepository.saveAll(pila)
            restituisci out

        // non avendo trovato nessun biglietto libero, restituisco null
        ticketsRepository.saveAll(pila)
        restituisci nullo

    //il token non è valido oppure chi ha chiesto il servizio non ha il permesso
    altrimenti:
        // autenticazione fallita
        restituisci nullo
}

```

Figura 9: Pseudocodice ServeNextOPE()

2.6 Suite di testing delle API (con Postman)

1. Utilizziamo l'API incrementaOPE per aggiungere biglietti alla nostra collezione, verificandone i campi

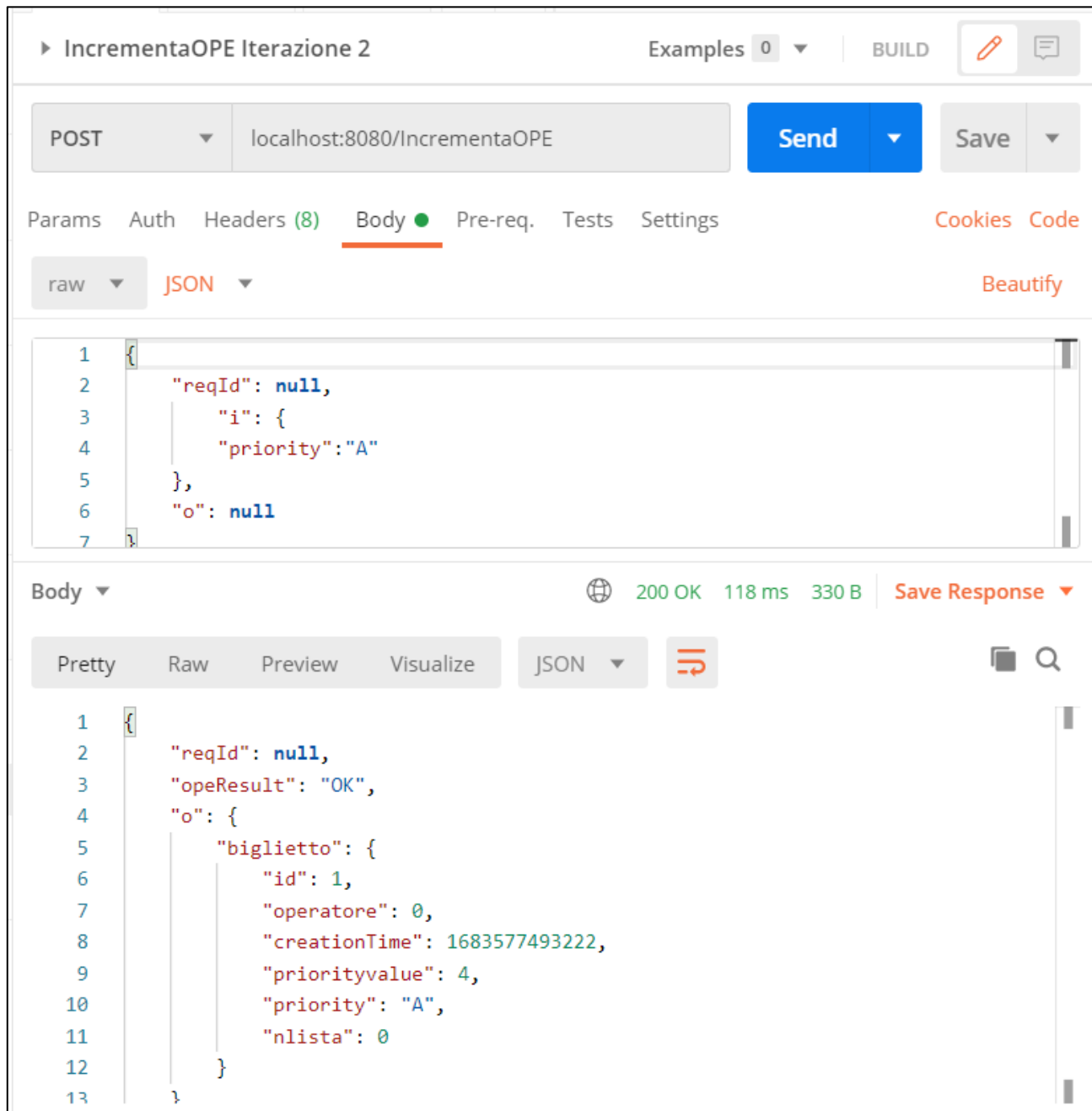


Figura 10 Postman API incrementaOPE

- Controlliamo la presenza dei biglietti utilizzando l'API allStackOPE, verificando la presenza e le caratteristiche prima affermate

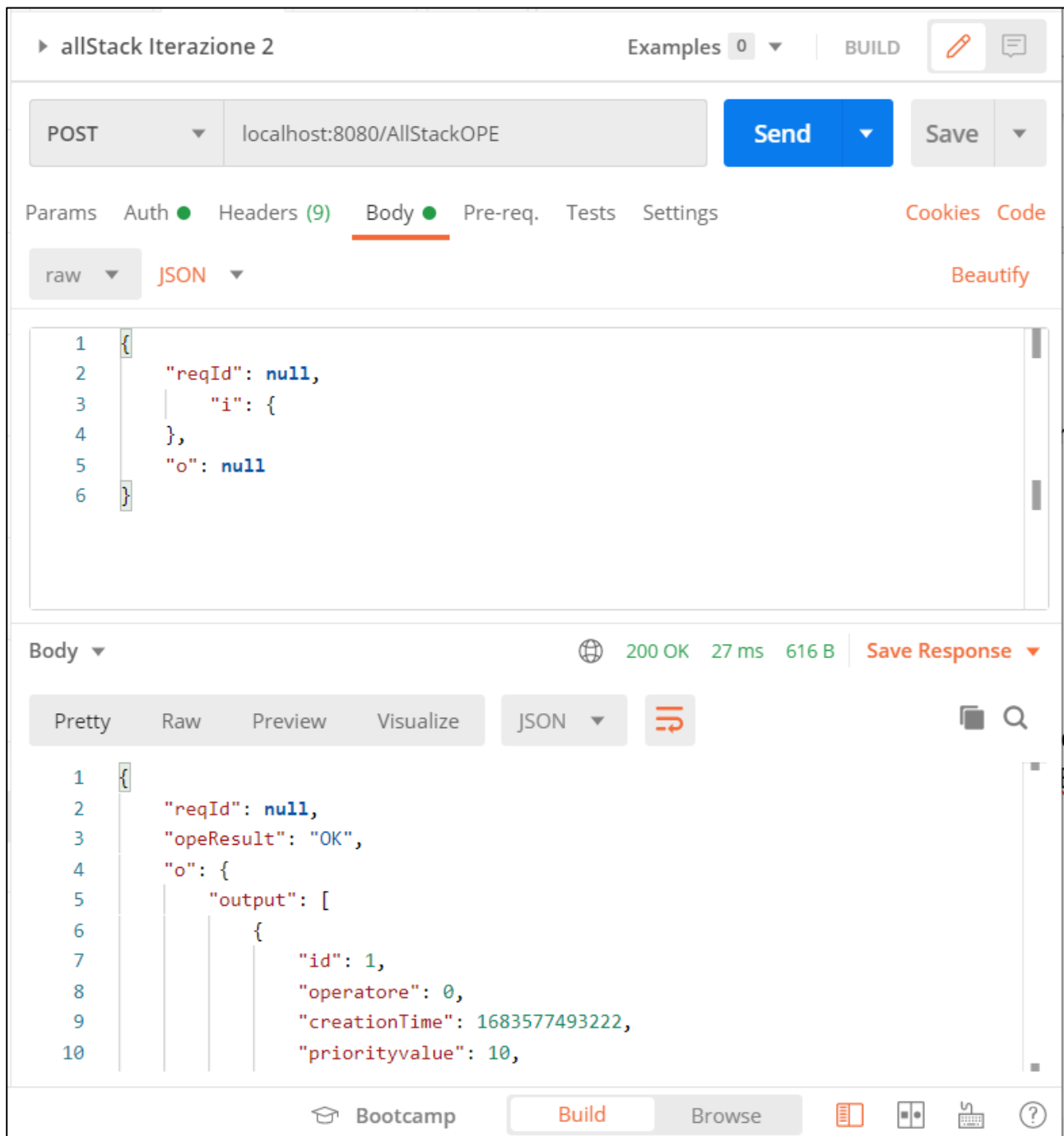


Figura 11 Postman API allStackOPE

- Utilizziamo l'API serveNextOPE per simulare il servizio ai vari utenti, verificando l'assegnamento agli operatori del primo utente disponibile

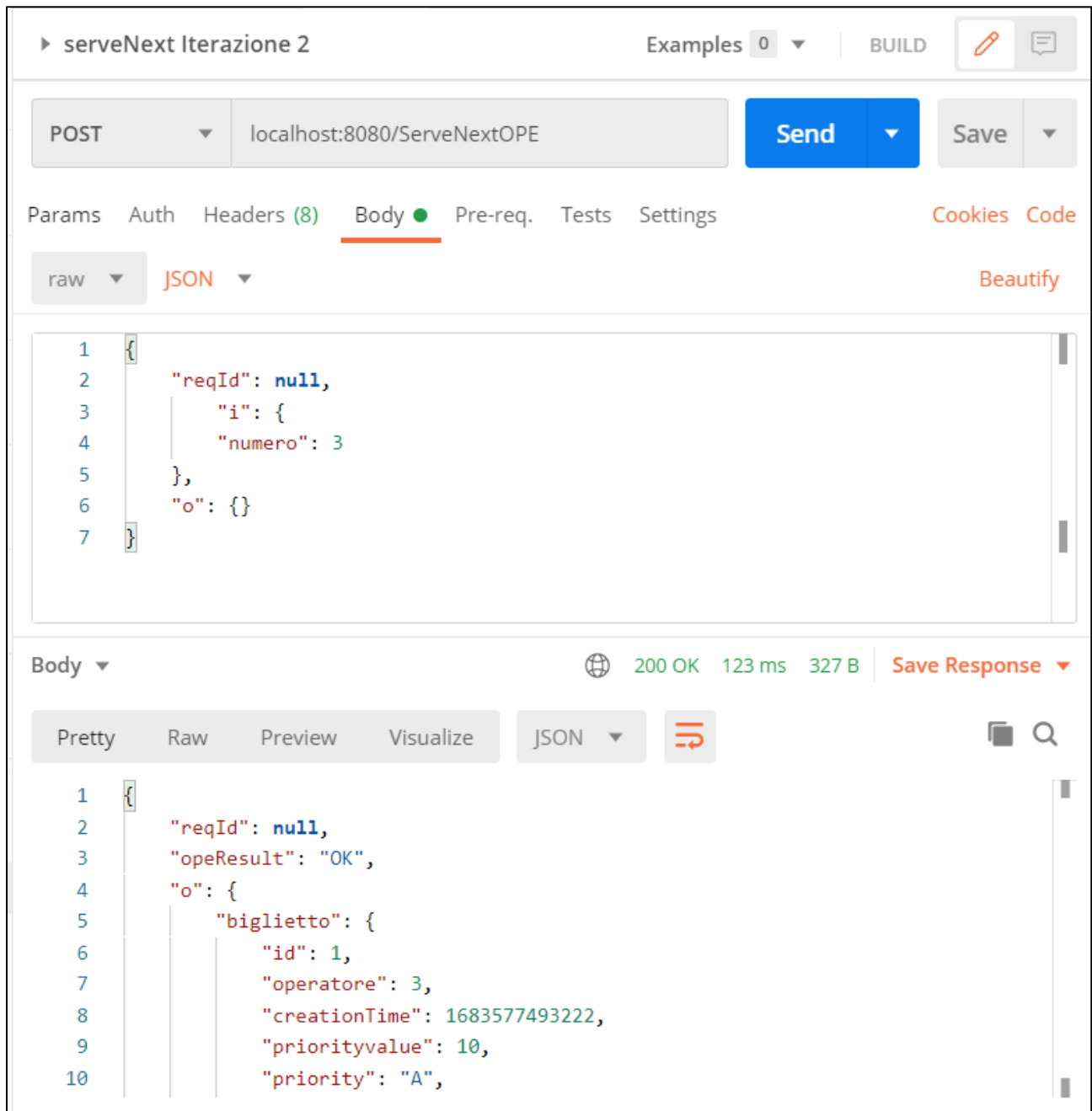


Figura 12 Postman API serveNextOPE

4. Controlliamo l'evoluzione della coda con sempre con l'API allStackOPE

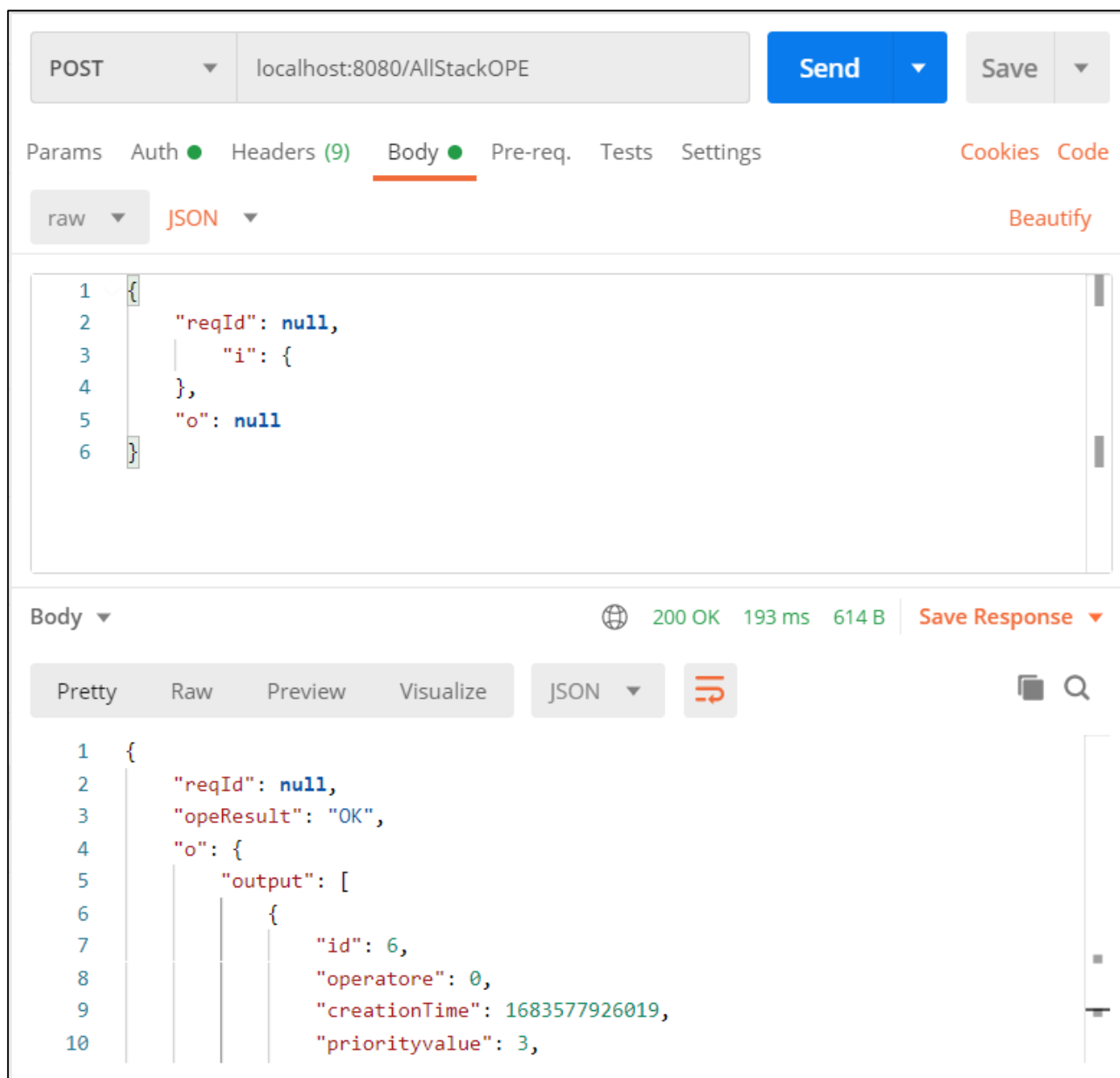


Figura 13 Postman API allStackOPE

2.7 Casi di test con JUnit

Eseguiamo un testing più approfondito dell'algoritmo di selezione tramite priorità, simulando la TicketsRepository in uno scenario a biglietti di varia priorità e non in ordine

```
@DataMongoTest
@RunWith(SpringRunner.class)
public class ServeNextOpeTest {

    6 usages
    @Autowired
    private TicketsRepository ticketsRepository;

    Salvatore Greco

    @TestConfiguration
    static class serveNextOpeTest {
        Salvatore Greco
        @Bean
        public ServeNextOPE serveNextOPE() { return new ServeNextOPE(); }
    }

    6 usages
    @Autowired
    private ServeNextOPE serveNextOPE;

    Salvatore Greco

    @Test
    public void checkServeNext()
    {
        Ticket priorityA = new Ticket( ticketNumber: 1, op: 0, TicketType.A, nLista: 1);
        Ticket priorityB = new Ticket( ticketNumber: 2, op: 0, TicketType.B, nLista: 1);
        Ticket priorityC1 = new Ticket( ticketNumber: 3, op: 0, TicketType.C1, nLista: 1);
        Ticket priorityC2 = new Ticket( ticketNumber: 4, op: 0, TicketType.C2, nLista: 1);
        Ticket priorityBsecondo = new Ticket( ticketNumber: 5, op: 0, TicketType.B, nLista: 1);
        Ticket priorityAsecondo = new Ticket( ticketNumber: 6, op: 0, TicketType.A, nLista: 1);

        ticketsRepository.save(priorityA);
        ticketsRepository.save(priorityB);
        ticketsRepository.save(priorityC1);
        ticketsRepository.save(priorityC2);
        ticketsRepository.save(priorityBsecondo);
        ticketsRepository.save(priorityAsecondo);

        ServeNextCb.I in = new ServeNextCb.I();
        in.setNumero(1L);
        ServeNextCb.O out = serveNextOPE.execute(in);

        assert out.getBiglietto().getPriority().equals(TicketType.A);

        out = serveNextOPE.execute(in);

        assert out.getBiglietto().getPriority().equals(TicketType.A);

        out = serveNextOPE.execute(in);

        assert out.getBiglietto().getPriority().equals(TicketType.B);

        out = serveNextOPE.execute(in);

        assert out.getBiglietto().getPriority().equals(TicketType.B);

        out = serveNextOPE.execute(in);

        assert out.getBiglietto().getPriority().equals(TicketType.C1) || out.getBiglietto().getPriority().equals(TicketType.C2);

        out = serveNextOPE.execute(in);

        assert out.getBiglietto().getPriority().equals(TicketType.C1) || out.getBiglietto().getPriority().equals(TicketType.C2);
    }
}
```

Figura 14: svuotamento della coda con priorità

Iterazione 3

3.1 Scelte di implementazione

Al fine di proteggere le API esposte dal backend e dare una sicurezza all'intero sistema progettato, è necessario che solo i device pensati nel deployment utilizzino effettivamente le funzionalità implementate dal sistema.

È anche vero che non è necessario un vero e proprio sistema di accesso per device di uso comune, come ad esempio la torretta pubblica disponibile sul luogo di produzione.

Abbiamo pensato ad un sistema di autenticazione, che permette poi l'autorizzazione, in modo da limitare l'utilizzo delle API solo a chi ne ha i permessi

- Con Autenticazione intendiamo la richiesta di un token, simil-JWT, al componente preposto del sistema.

L'autenticazione viene effettuata dagli utenti registrati tramite classico username : password.

- Con Autorizzazione intendiamo l'utilizzo effettivo del token per ottenere il permesso di utilizzo delle varie API

L'autorizzazione viene effettuata inviando, insieme alla richiesta, anche il token JWT ottenuto precedentemente

Il sistema è implementato utilizzando due librerie, una per la generazione dei JWT e una per la convalida, implementato sull'utilizzo di ogni API di funzionalità

3.2 Component diagram white-box

La classe JwtUtils viene utilizzata come @Component di Spring all'interno sia delle classiche API esposte, sia di un altro componente chiamato Autenticatore utilizzato anch'esso dalle API.

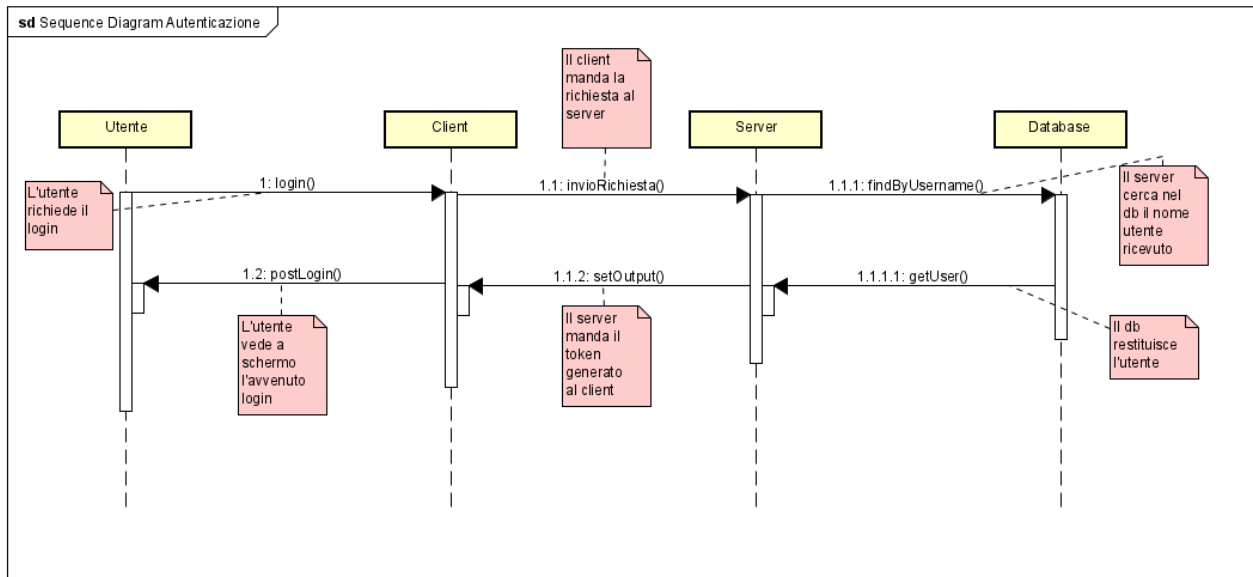
- Nel primo caso utilizziamo il componente per implementare la funzionalità di Autenticazione (ovvero di login)
- Nel secondo caso invece lo includiamo in una logica (quella dell'Autenticatore) per implementare la funzionalità di Autorizzazione

Nel component diagram di dettaglio viene mostrato un esempio solo per l'API IncrementaOPE.

In realtà, visto che questo componente viene "iniettato" in tutte le API esposte, ogni API ha a disposizione la funzionalità del componente Autenticatore (e quindi i suoi metodi utilizzati)

3.3 Sequence diagram di esecuzione

Questo Sequence Diagram mostra come si effettua l'operazione di Login richiesta dall'utente, necessaria per poter usufruire degli altri servizi messi a disposizione.



Questo Sequence Diagram mostra come il client e il server interagiscono per la generazione di un biglietto richiesta dall'utente dopo che ha effettuato correttamente il login.

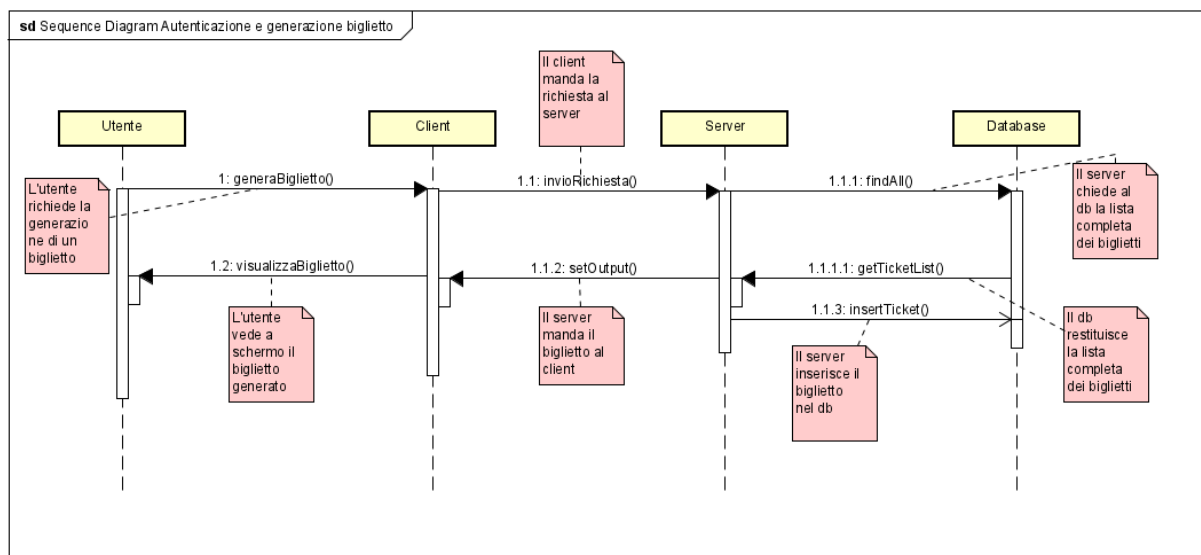


Figura 17: Sequence diagram generazione del biglietto

3.4 Pseudocodice Autorizzatore (Componente)

Il componente “Autorizzatore” rappresenta una classe che gestisce l’autorizzatore basata su token.

```
//metodo che restituisce true se il token è valido e l'utente ha il permesso di effettuare la richiesta, in ogni altro
//caso restituisce false
public boolean autorizzatore(String token) {
    //se il token passato con la chiamata non è nullo
    se token non è nullo:
        //estraggo lo username dal token
        String username = jwtUtils.getUserNameFromJwtTokenNoExpired(token)
        //se è il token associato al visualizzatore restituisco true
        se username è uguale a "visualizzatore":
            restituisci vero
        //altrimenti cerco l'utente nel database
        Utente utente = utenteRepository.findByUsername(username)
        //se l'utente è stato trovato
        se utente non è vuoto:
            Date attuale = nuova Date()
            Date exp = jwtUtils.getExpirationFromJwtToken(token)
            //se la data di scadenza del token è maggiore di quella attuale
            se attuale è prima di exp:
                restituisci vero // autorizza
            altrimenti:
                restituisci falso // token scaduto
        altrimenti:
            restituisci falso // nome utente non trovato, il token non è valido
    altrimenti:
        restituisci falso // token nullo, quindi non autenticato
}
```

Figura 18: Pseudocodice autorizzatore

All'interno della classe, ci sono due annotazioni **@Autowired** utilizzate per iniettare le dipendenze delle classi "JwtUtils" e "UtenteRepository".

Queste dipendenze saranno automaticamente risolte dal framework Spring e le istanze delle classi corrispondenti saranno fornite al componente.

La classe contiene un metodo chiamato "autorizzatore" che prende in input una stringa chiamata "token" e restituisce un valore booleano. Il metodo implementa la logica di autorizzazione basata sul token.

Se il token non è nullo, il metodo estrae il nome utente dal token utilizzando il metodo **"getUserNameFromJwtTokenNoExpired"** della classe **"JwtUtils"**.

Se il nome utente è **"visualizzatore"**, il metodo restituisce true, indicando un'autorizzazione riuscita per il visualizzatore.

Altrimenti, il metodo cerca il nome utente nel database utilizzando il metodo **"findByUsername"** della classe **"UtenteRepository"**.

Se l'utente viene trovato, il metodo confronta la data corrente con la data di scadenza del token.

Se la data corrente è precedente alla data di scadenza, il metodo restituisce true, indicando che l'autenticazione è riuscita.

Altrimenti, restituisce false, indicando che il token è scaduto. Se l'utente non viene trovato nel database, il metodo restituisce false, indicando che il nome utente nel token non è valido.

Se il token è nullo, il metodo restituisce false, indicando che l'autenticazione non è riuscita.

In sintesi, la classe **"Autorizzatore"** fornisce un componente per gestire l'autenticazione basata su token.

Utilizza le dipendenze delle classi **"JwtUtils"** e **"UtenteRepository"** per implementare la logica di autenticazione e verifica la validità del token per autorizzare l'accesso.

3.4.1 Analisi complessità metodo autenticazione

La complessità è $O(1)$, scrivi che i metodi in libreria importata e a repository sono trattati come istruzioni semplici (niente cicli)

3.5 Suite di testing dei metodi (con JUnit)

Abbiamo testato il servizio di generazione dei biglietti in vari casi. Il risultato dei test è stato positivo per tutti i casi.

```
@Test
public void checkIncrementa()
{
    //caso in cui il token JWT è nullo -> il risultato restituito dalla OPE deve essere nullo
    IncrementaCb.I in = new IncrementaCb.I();
    in.setPriority("A");
    in.setToken(null);
    IncrementaCb.O out = incrementaOPE.execute(in);
    assert out == null;

    //caso in cui il token JWT non è valido -> il risultato restituito dalla OPE deve essere nullo
    in = new IncrementaCb.I();
    in.setPriority("A");
    in.setToken("eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ3aXN1YWxpenphdG9yZSIsImVudCI6MTY3MjQwOTEyMSwiZXhwIjoxNjcyNDA5MTIxfQ.Qq4ZUYDUURnLj");
    out = incrementaOPE.execute(in);
    assert out == null;

    //caso in cui il token JWT è valido ma la priorità scelta non esiste -> la OPE lancia l'eccezione InvalidPriorityException
    assertThrows(InvalidPriorityException.class, () -> {
        IncrementaCb.I inL = new IncrementaCb.I();
        inL.setPriority("F1");
        inL.setToken("eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ2aXN1YWxpenphdG9yZSIsImVudCI6MTY3MjQwOTEyMSwiZXhwIjoxNjcyNDA5MTIxfQ.Qq4ZUYDUURnLj");
        incrementaOPE.execute(inL);
    });

    //caso in cui il token JWT è valido, la priorità esiste e la coda è vuota -> il risultato restituito dalla OPE
    // è un biglietto con: id = 1, priority = A, nlista = 0, operatore = 0
    in = new IncrementaCb.I();
    in.setPriority("A");
    in.setToken("eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ2aXN1YWxpenphdG9yZSIsImVudCI6MTY3MjQwOTEyMSwiZXhwIjoxNjcyNDA5MTIxfQ.Qq4ZUYDUURnLj");
    out = incrementaOPE.execute(in);
    assert out.getBiglietto().getTicketNumber() == 1;
    assert out.getBiglietto().getPriority() == TicketType.valueOf(name: "in.getPriority()");
    assert out.getBiglietto().getNlista() == 0;
    assert out.getBiglietto().getOperatore() == 0;

    //caso in cui il token JWT è valido, la priorità esiste e la coda non è vuota -> il risultato restituito dalla OPE
    // è un biglietto con: id = 2, priority = B, nlista = 1, operatore = 0
    in = new IncrementaCb.I();
    in.setPriority("B");
    in.setToken("eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ2aXN1YWxpenphdG9yZSIsImVudCI6MTY3MjQwOTEyMSwiZXhwIjoxNjcyNDA5MTIxfQ.Qq4ZUYDUURnLj");
    out = incrementaOPE.execute(in);
    assert out.getBiglietto().getTicketNumber() == 2;
    assert out.getBiglietto().getPriority() == TicketType.valueOf(name: "in.getPriority()");
    assert out.getBiglietto().getNlista() == 1;
    assert out.getBiglietto().getOperatore() == 0;
}
```

Figura 19: test generazione del biglietto

Iterazione Finale

4.1 Toolchain e Design pattern MVP IO

4.1.1 Toolchain

- RxJava: utilizzato per la gestione della back pressure, fornisce un'implementazione del pattern Observable/Observer per Android
- Dagger/Hilt: utilizzato per la dependency injection dei componenti
- Retrofit/OkHttp: utilizzato per le chiamate http alle API
- Google Gson: utilizzato per il parsing dei JSON ricevuti
- Material per Android 13: utilizzato per l'implementazione dell'interfaccia della app Android

4.1.2 Design pattern MVP

Il design pattern strutturale MVP (Model View Presenter) è una derivazione del design pattern architetturale Model View Controller (MVC). Il suo funzionamento può essere descritto dal seguente schema:

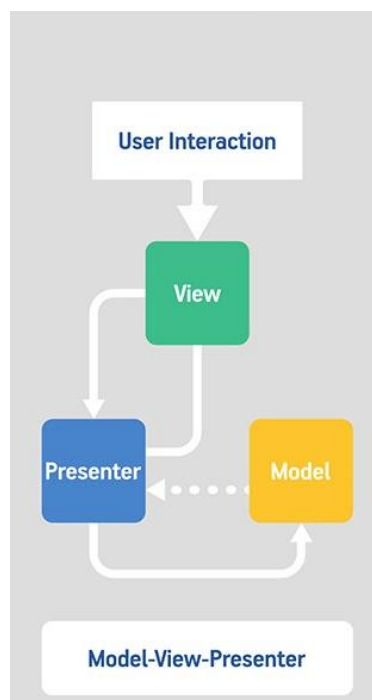


Figura 20: schema MVP

La differenza sostanziale, rispetto al pattern MVC, è che le View e il Model sono separati e il Presenter fa da mediatore tra i due.

Inoltre, ogni View ha il proprio Presenter, che può richiamare metodi per interagire con il Model e la propria View.

4.1.2.1 Struttura del progetto

Prendendo come esempio la Main Activity e seguendo il pattern MVP, si ha che a questa activity sono associate due classi e una interfaccia.

Le due classi sono una per la view (MainActivity) e una per il presenter (MainPresenter), mentre l'interfaccia prende il nome di MainContract e contiene i metodi che la view può richiamare nel presenter e viceversa.

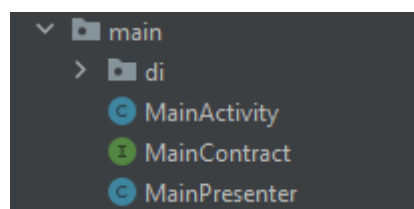


Figura 21: esempio di activity/contract/presenter

Il presenter e la view contengono l'uno il riferimento dell'altro e questo permette quindi di richiamare i metodi voluti.

```
@Inject  
MainContract.Presenter mMainPresenter;
```

Figura 22: dependency injection del presenter

Questi metodi sono presenti nell'interfaccia MainContract, che contiene a sua volta due interfacce:

- View
- Presenter

```
public interface MainContract {  
  
    interface View {...}  
  
    interface Presenter extends BasePresenter {...}  
}
```

Figura 23: interfaccia di contratto tra view e presenter

La parte di Model non è presente all'interno delle directory delle activity o dei fragment in quanto è un insieme di repository condivise tra tutti i presenter.

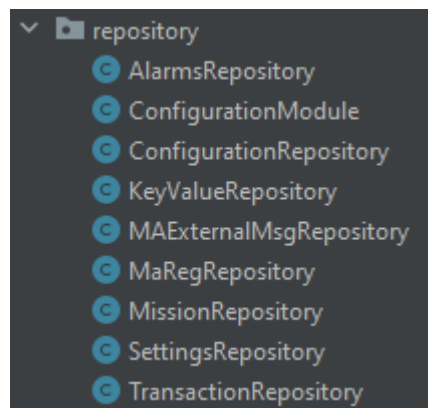


Figura 24: esempio di model

Per potervi accedere, ogni presenter ha il riferimento ai repository di cui ha bisogno.

```

@Inject
AlarmsRepository mAlarmsRepository;

@Inject
MAExternalMsgRepository mExternalMsgRepository;

@Inject
MissionRepository mMissionRepository;

@Inject
KeyValueRepository mKeyValueRepository;

@Inject
ConfigurationRepository configurationRepository;

@Inject
SettingsRepository mSettingsRepository;

```

Figura 25: dependency injection dei model nei presenter

4.2 Sistema di stima del tempo residuo

Il metodo **UserQueue** è progettato per stimare il tempo di coda per un determinato utente in base alle informazioni presenti in una coda di biglietti. Esso prende diversi parametri, tra cui l'utente stesso, la coda, la lista dei biglietti nella coda e il biglietto dell'utente per cui si desidera stimare il tempo di coda.

All'interno del metodo, viene creato un iteratore per scorrere i biglietti nella lista. Successivamente, vengono inizializzate alcune variabili di conteggio per tenere traccia del numero di biglietti in coda di diversi tipi.

Il metodo utilizza un ciclo while per iterare su ogni biglietto nella lista. Per ogni biglietto, viene verificato se ha una priorità superiore a quella del biglietto dell'utente e se non è uguale ad esso. Se queste condizioni sono soddisfatte, il conteggio per il tipo di biglietto appropriato viene incrementato.

Dopo aver completato l'iterazione di tutti i biglietti nella coda, viene calcolato il tempo totale di coda moltiplicando il numero di biglietti in coda di ogni tipo per il tempo medio corrispondente a quel tipo di biglietto.

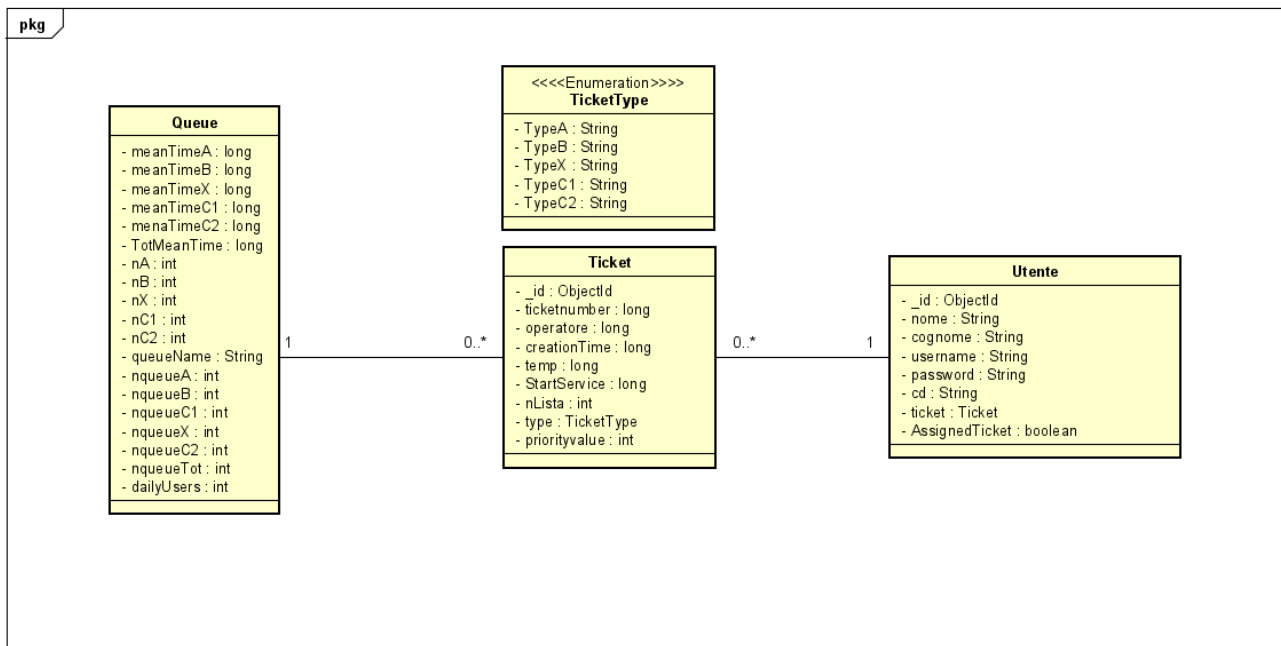
Il tempo totale di coda viene quindi convertito in un oggetto **LocalTime** utilizzando un metodo ausiliario **convertSecondsToLocalTime**. Infine, il tempo di coda stimato viene restituito come una stringa utilizzando il metodo **toString()** dell'oggetto **LocalTime**.

In sintesi, il metodo **UserQueue** analizza i biglietti nella coda, tiene traccia del numero di biglietti di diversi tipi e calcola il tempo totale di coda basato su tali conteggi e sui tempi medi di attesa associati a ciascun tipo di biglietto. Fornisce quindi una stima del tempo di coda per un utente specifico in base alle priorità dei biglietti nella coda.

```
public String UserQueue(Optional<Utente> utente, Queue queue, List<Ticket> pila, Ticket userTicket){
    Iterator<Ticket> it=pila.iterator();
    int nqueueA=0;
    int nqueueB=0;
    int nqueueX=0;
    int nqueueC1=0;
    int nqueueC2=0;
    while(it.hasNext()){
        Ticket temp=it.next();
        if(temp.comparePriority(userTicket) && !temp.equals(userTicket)){
            if(temp.getTypeOfTicket()==TicketType.A){
                nqueueA=nqueueA+1;
            }else if(temp.getTypeOfTicket()==TicketType.X){
                nqueueX=nqueueX+1;
            }else if(temp.getTypeOfTicket()==TicketType.B){
                nqueueB=nqueueB+1;
            }else if(temp.getTypeOfTicket()==TicketType.C1){
                nqueueC1=nqueueC1+1;
            }else if(temp.getTypeOfTicket()==TicketType.C2){
                nqueueC2=nqueueC2+1;
            }
        }
    }
    long TotSec=nqueueA*queue.getMeanTimeA()+nqueueB*queue.getMeanTimeB()+nqueueX*queue.getMeanTimeX()
        +nqueueC1*queue.getMeanTimeC1()+nqueueC2*queue.getMeanTimeC2();
    LocalTime QueueTime=convertSecondsToLocalTime(TotSec);
    return QueueTime.toString();
}
```

Figura 26: codice generazione stima tempo residuo

4.2.1 Aggiornamento del data class diagram



4.3 Conclusioni e repository IO

Il codice sorgente di progetto con relativa documentazione, breve descrizione e suite di test è disponibile alla seguente repository ->

<https://github.com/atusghen/ticket-generator>

Nella sezione “Releases” è disponibile un installation kit completo di eseguibile Java (.jar) e Android (.apk)

I requisiti implementati, con il loro stato dell’arte sono

Use Case	Descrizione breve	Descrizione risultato	Risultato
UC1: Generazione biglietto	l’utente ha la possibilità di generare il biglietto dalla torretta	Funzionalità implementata come API, mancanza di una interfaccia	

UC2: Visualizzazione della dashboard della coda	Consente di visualizzare a tutti gli utenti nell'ufficio lo stato della coda tramite uno schermo	Funzionalità implementata come API, mancanza di una interfaccia	
UC3: Generazione biglietto prioritario	Consente ad un utente in possesso di applicazione di generare un biglietto speciale	Funzionalità completamente implementata	
UC4: Login	Consente ad un'utente di accedere al sistema di generazione dei biglietti via app	Funzionalità completamente implementata	
UC5: Logout	Uscita dal sistema di priorità	Funzionalità completamente implementata	
UC6: Gestione profilo utente	Memorizzazione, visualizzazione e modifica dei dati aggiuntivi di profilazione	Funzionalità non implementata	
UC7: Registrazione profilo utente	Ottenimento della possibilità di accesso al sistema di generazione del biglietto via app	Funzionalità completamente implementata	
UC8: Recupero credenziali	Recupero della password di accesso al sistema via app	Funzionalità non implementata	
UC9: Evento di avanzamento della coda	Funzionalità lato operatore per ottenere le indicazioni successive di biglietto e segnalare l'evento al sistema	Funzionalità completamente implementata	

UC10: Gestione della coda	Operazioni da eseguire all'evento operatore per la corretta gestione della coda e avanzamento	Funzionalità completamente implementata	
UC11: Visualizzazione dashboard utente	Visualizzazione delle prenotazioni, visualizzazione tempo stimato via app utente	Funzionalità implementata, manca la possibilità di visualizzare lo storico (attualmente si visualizza solo la prenotazione attuale)	

Tabella 2: risultato delle implementazioni degli UC

Riassumendo in termini di requisiti funzionali:

Codice	Nome	Priorità	Implementato
R1	Generazione biglietto in sede	B	Parzialmente
R2	Visualizzazione coda	B	Parzialmente
R3	Avanzamento coda (operatore)	A	Si
R4	Profilazione utente	C	No
R5	Log in/Log out e registrazione	A	Si
R6	Generazione biglietto non in sede	A	Si

Tabella 3: risultato delle implementazioni dei requisiti funzionali

4.4 Sviluppi futuri

Lo stato attuale dell'implementazione del progetto prevede solamente che alcune delle funzionalità siano correttamente funzionanti, segnatamente le funzionalità di visualizzazione delle dashboard, generazione dei biglietti e di gestione della coda.

Gli sviluppi futuri dovranno certamente riguardare la progettazione e l'implementazione di varie componenti sul server e il miglioramento di alcune componenti attualmente implementate.

Alcune possibili idee possono essere l'implementazione della gestione di diverse tipologie di code in modo da poter scegliere quella più adatta allo scopo.

Un miglioramento che reputiamo importante consiste nel migliorare l'attuale sistema di monitoraggio e di stima del tempo di attesa in coda, dato che l'attuale algoritmo si basa sul classico metodo di calcolo della media aritmetica, che con l'aumentare dell'uso della coda porterebbe a un errore di "overflow" dei dati di tipo "long".

Un possibile algoritmo che potrebbe essere utilizzato al posto del calcolo della media sarebbe **Adaptive Recursive Least Squares** con forgetting factor.

Si tratta di un algoritmo ricorsivo che può essere utilizzato online, il quale permette di fare previsioni su quanto tempo un utente potrebbe rimanere in coda, essendo anche adattivo consente di dare importanza ai dati più recenti rispetto ai dati più remoti; questo è particolarmente utile in un contesto dinamico come la gestione di una coda, in cui le caratteristiche degli utenti possono cambiare nel tempo e l'importanza dei dati storici può diminuire.

4.5 Manuale utente sulla app Android

Per poter utilizzare l'applicazione Android, basta installare l'apk fornito, sul dispositivo che l'utente vuole utilizzare per usufruire del servizio. L'applicazione non ha bisogno di alcun permesso speciale che deve essere dato da parte dell'utente.

Al primo avvio l'app si presenta con questa schermata.

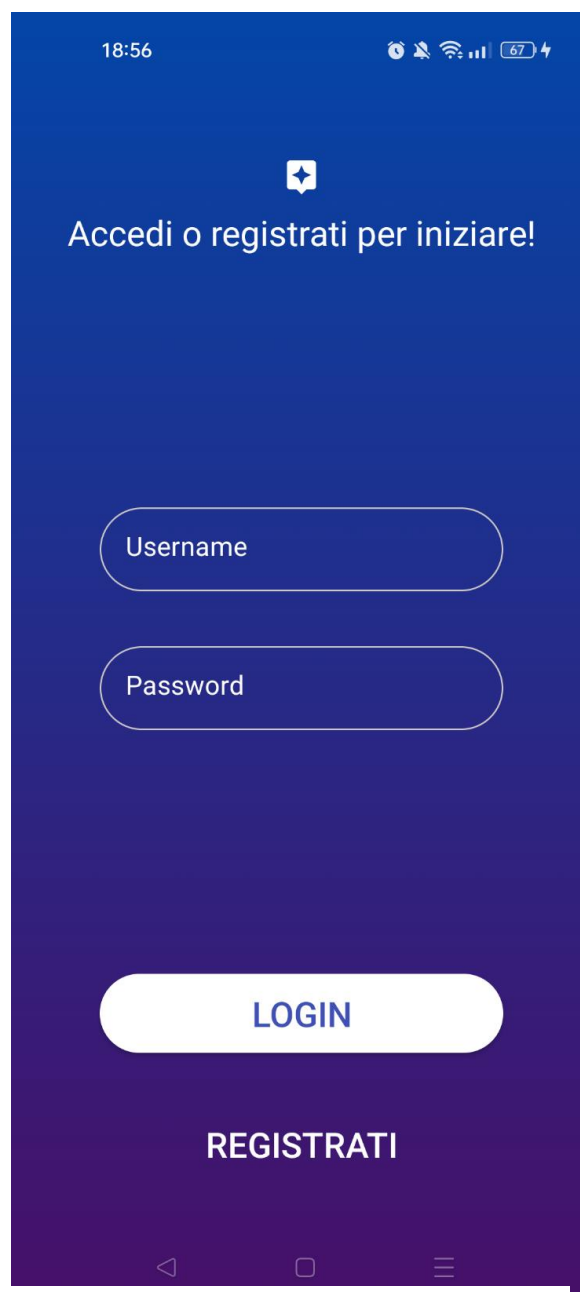
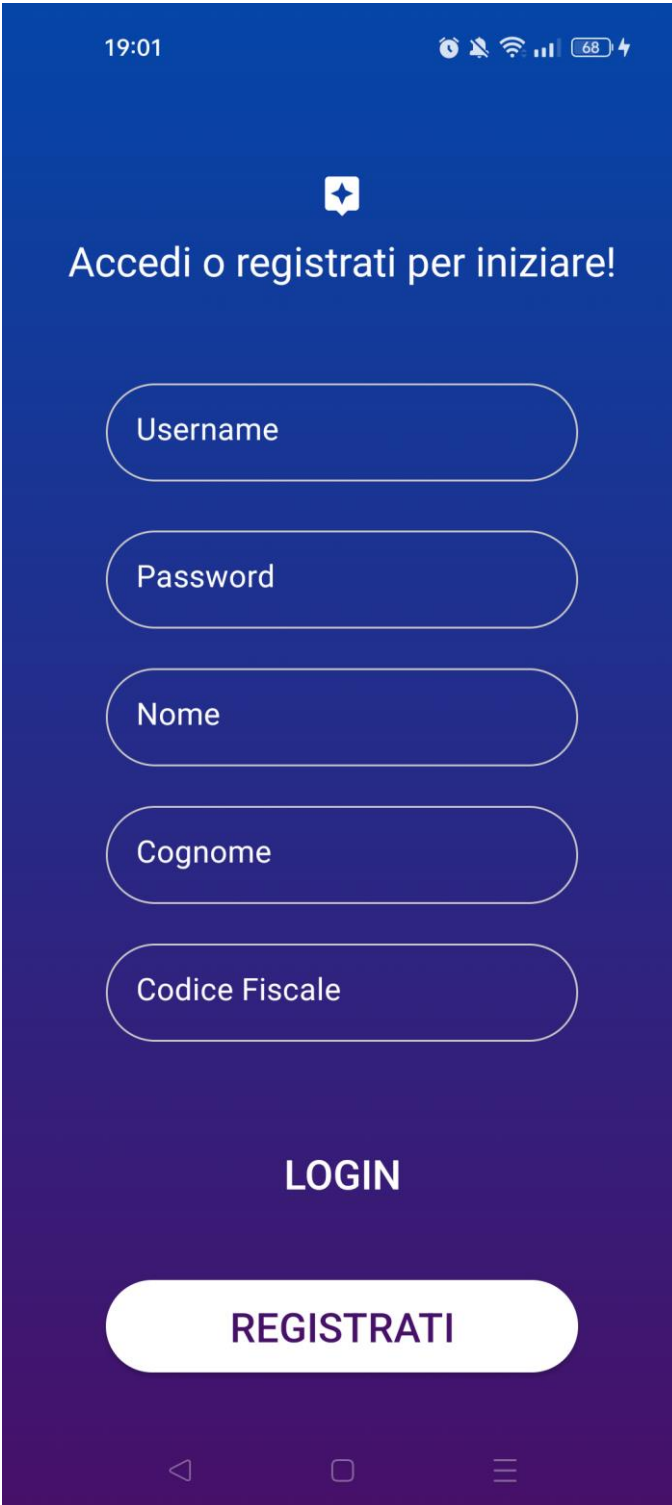


Figura 27: schermata di login

L'utente può accedere inserendo username e password, se ha già effettuato la procedura di registrazione, altrimenti cliccando su REGISTRATI si aprirà il form di registrazione.

A mobile application registration screen with a dark blue gradient background. At the top, the status bar shows the time 19:01 and various icons. Below the status bar is a white star icon. The main heading is "Accedi o registrati per iniziare!". There are five rounded rectangular input fields stacked vertically, labeled "Username", "Password", "Nome", "Cognome", and "Codice Fiscale". Below these fields, the word "LOGIN" is displayed in white capital letters. At the bottom, there is a large white rounded rectangular button with the text "REGISTRATI" in dark blue capital letters. The bottom of the screen shows the standard Android navigation bar with back, home, and app drawer icons.

19:01

Accedi o registrati per iniziare!

Username

Password

Nome

Cognome

Codice Fiscale

LOGIN

REGISTRATI

Figura 28: schermata di registrazione

Una volta effettuata la registrazione o l'accesso, si aprirà la schermata principale dove l'utente potrà effettuare la prenotazione del biglietto.

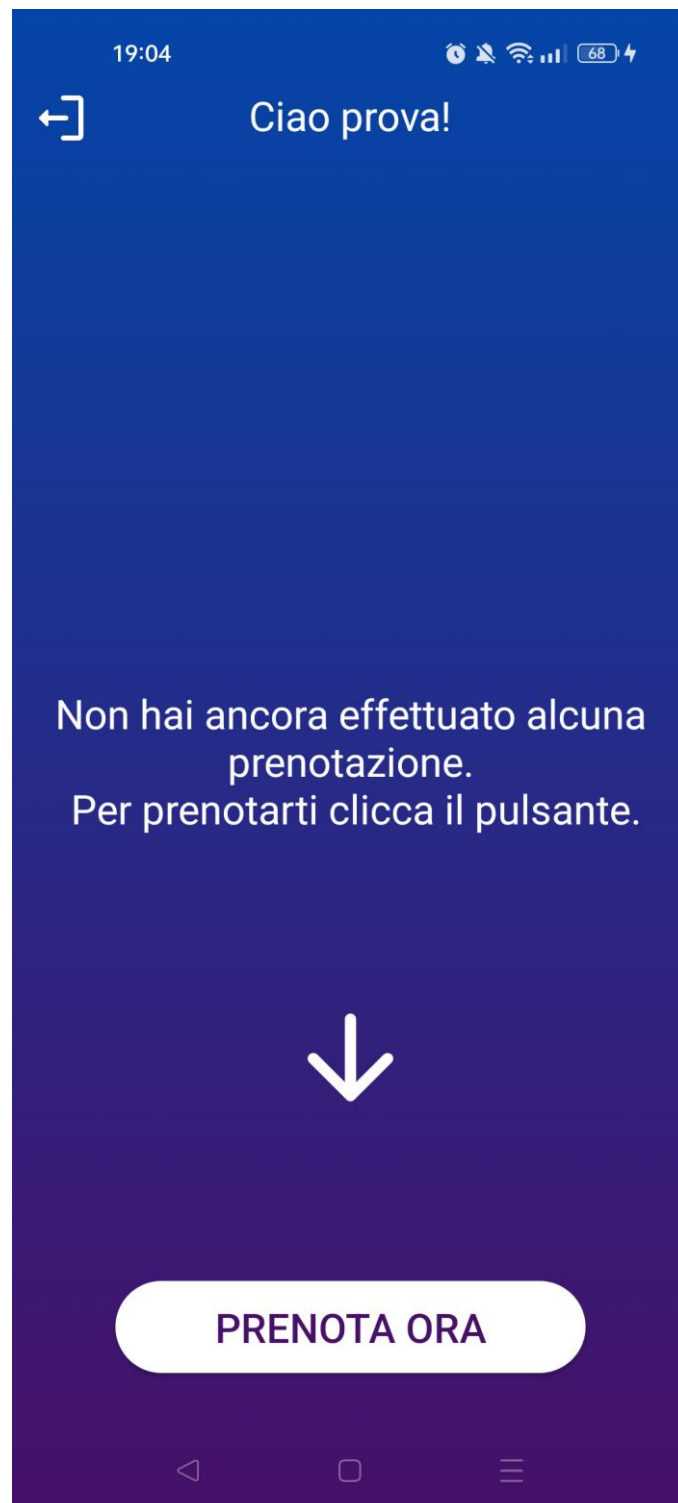


Figura 29: dashboard applicazione

Se invece l'utente dovesse aver già effettuato una prenotazione, senza che essa sia stata ancora servita, allora verrà visualizzato il biglietto con le relative informazioni.



Figura 30: dashboard con biglietto generato

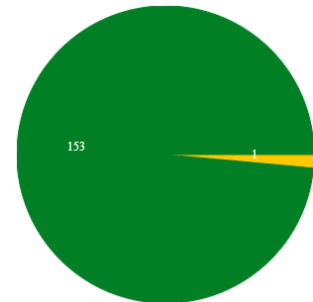
Analisi statica

Analisi Statica:

Lo strumento utilizzato per l'analisi statica è stato codeMR, un plugin fornito dall'ambiente IntelliJ.

Informazioni generali:

Total lines of code	2253
Number of classes	154
Number of packages	31
Number of external classes	207
Number of external packages	43
Number of problematic classes	1
Number of highly problematic classes	0



Pie Chart classi

L'analisi ha rilevato un'unica classe problematica all'interno dell'applicazione. Ciò potrebbe indicare un potenziale problema nel codice di quella specifica classe, che richiederà ulteriori verifiche e correzioni da parte degli sviluppatori.

Tuttavia, non sono state individuate classi altamente problematiche, il che suggerisce che l'applicazione nel complesso non presenta criticità gravi o strutture di codice estremamente complesse.

Sommario metriche della componente server:

	WMC	LOC	#(C&I)	#C	AC	EC	Abs	Ins	ND	#I	Size	Coupling
main	5	55	4	4	0	2	0.0	1.0	0.0	0	L	L
dao	7	12	3	0	13	3	1.0	0.19	0.19	3	L	L
entities	87	285	4	4	18	3	0.0	0.14	0.86	0	L-M	L
dto	23	36	41	31	14	6	0.03	0.3	0.67	0	H	L-M
library	32	138	5	5	12	4	0.4	0.25	0.35	0	L	L
ope	60	353	11	11	2	11	0.0	0.85	0.15	0	M-H	M-H

Analizzando la tabella, possiamo ottenere un'idea generale delle caratteristiche dei diversi pacchetti presenti nella componente server.

Iniziamo osservando la complessità delle classi, rappresentata dalla metrica WMC (Weighted Methods per Class). Notiamo che le classi "entities" e "ope" presentano valori significativamente più elevati rispetto alle altre classi. Ciò suggerisce che queste classi potrebbero essere più complesse in termini di numero di metodi e complessità logica.

Guardando alla dimensione del codice, rappresentata dalle righe di codice (LOC), notiamo che le classi "entities" e "ope" hanno un numero significativo di linee di codice rispetto ad altre classi come "main" o "library". Questo potrebbe indicare una maggiore implementazione o una logica più dettagliata in tali classi.

Esaminando le metriche di accoppiamento, possiamo notare che la classe "dao" ha un valore elevato di afferent coupling (AC), indicando che molte altre classi dipendono da essa.

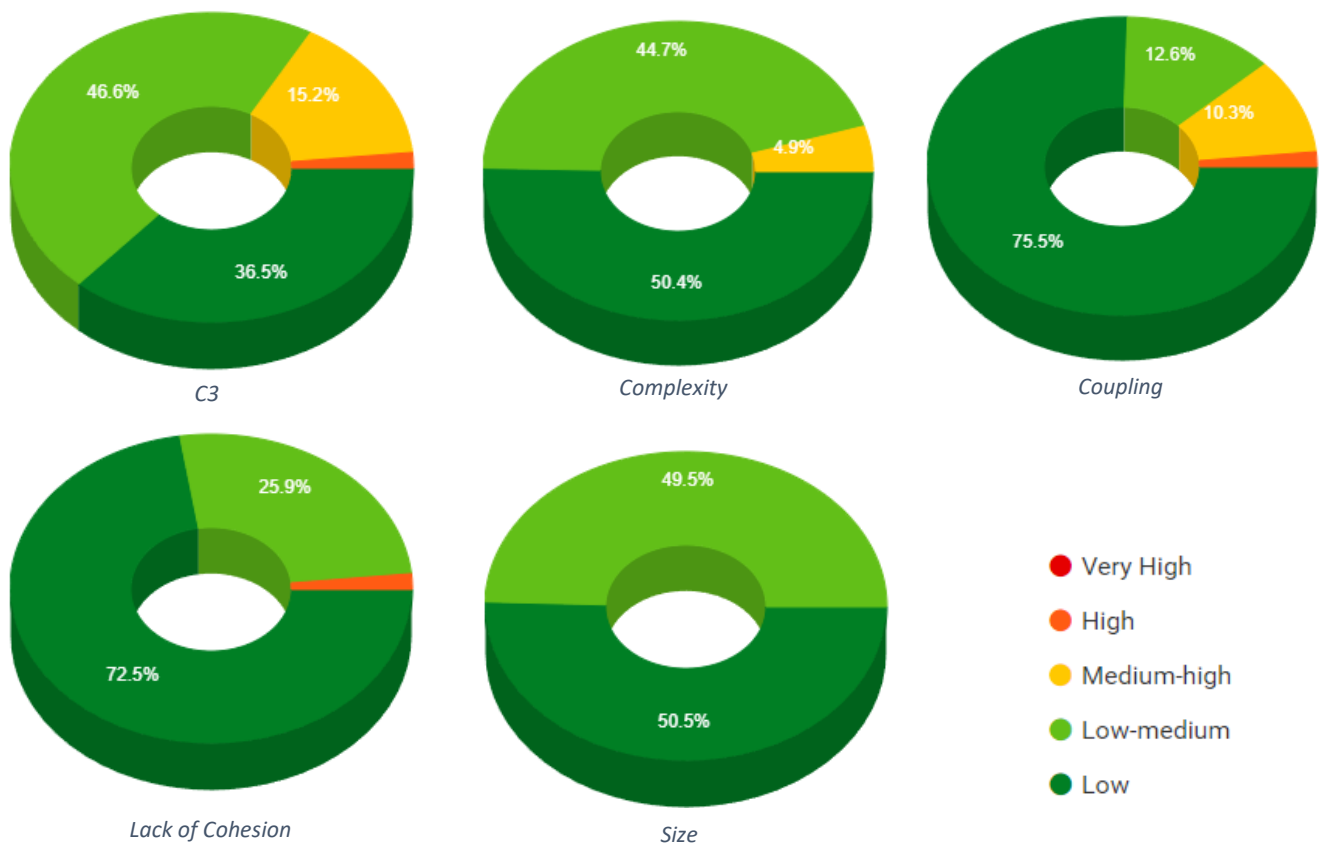
D'altra parte, la classe "ope" ha un valore elevato di efferent coupling (EC), suggerendo che dipende da numerose altre classi. Questi valori possono fornire indicazioni sulle dipendenze tra le classi e l'interazione complessiva del sistema.

Osservando le metriche di astrazione e instabilità, notiamo che la classe "dto" ha una percentuale elevata di metodi astratti (Abs) e una stabilità relativamente alta (Ins). Questo potrebbe indicare che la classe "dto" è progettata per essere estesa o implementata da altre classi e ha una bassa dipendenza da altre classi.

Infine, la metrica di accoppiamento (Coupling) ci fornisce informazioni sulla quantità di dipendenze tra le classi. Tutte le classi nella tabella mostrano un livello di accoppiamento basso o medio-basso.

In conclusione, l'analisi della tabella ci offre una panoramica delle diverse classi all'interno del server, inclusa la loro complessità, dimensione del codice, dipendenze e accoppiamento.

Pie Chart delle caratteristiche dell'applicazione:



Iniziamo osservando l'accoppiamento delle classi.

Il 36.5% delle metriche di accoppiamento è considerato basso, il che significa che molte classi all'interno dell'applicazione hanno un basso livello di dipendenza reciproca. Questo è un buon segno, poiché indica una buona modularità e una maggiore facilità di manutenzione del codice.

Inoltre, il 46.6% delle metriche di accoppiamento è classificato come medio-basso, il che suggerisce che la maggior parte delle dipendenze tra le classi è gestibile e non eccessivamente complessa.

Tuttavia, il 15.2% delle metriche indica un livello medio di accoppiamento, che richiede una certa attenzione per garantire una corretta gestione delle dipendenze. Il resto, che rappresenta una piccola percentuale, ha un livello medio-alto di accoppiamento, il che potrebbe indicare la presenza di alcune classi che dipendono fortemente da altre. Queste classi potrebbero richiedere ulteriori valutazioni per ridurre l'accoppiamento e migliorare la modularità complessiva del sistema.

Passando alla coesione, il 72.5% delle classi ha una bassa mancanza di coesione. Questo è un buon risultato, indicando che la maggior parte delle classi all'interno

dell'applicazione è logicamente coesa e presenta una chiara responsabilità all'interno del sistema.

Tuttavia, il 26% delle classi ha una mancanza di coesione medio-bassa, il che suggerisce che potrebbero essere presenti alcune classi con responsabilità multiple o incerte. Queste classi potrebbero richiedere un'ulteriore analisi per identificare le aree in cui migliorare la coesione.

Passando alla complessità, il 50.4% delle classi ha una bassa complessità, indicando che gran parte del codice è relativamente semplice da comprendere e gestire.

Il 44.7% delle classi ha una complessità medio-bassa, mentre solo il 4.9% ha una complessità media. Questi risultati suggeriscono che l'applicazione di ticket generator è generalmente ben strutturata, con classi di dimensioni e complessità contenute. Tuttavia, è sempre importante monitorare la complessità delle classi in modo da evitare l'accumulo di codice complesso che potrebbe essere difficile da gestire nel tempo.

Tuttavia, alcune classi potrebbero richiedere un'attenzione particolare per migliorare la coesione e ridurre l'accoppiamento. Nel complesso, questi risultati offrono indicazioni utili per la valutazione della struttura del codice e l'identificazione di potenziali aree di ottimizzazione per garantire un sistema di ticket generator ben progettato e manutenibile.

Classi Problematiche

L'unica classe problematica è la classe ServiceBean, classe utilizzata per gestire le richieste http e fornire le risposte corrispondenti.

Complexity	L	RFC	30	LOC	36
Coupling	H	SRFC	1	CMLOC	34
Size	L	DIT	2	NOM	11
Lack of Cohesion	H	LOC	36	LCOM	0.0
CBO	21	WMC	11	LCAM	0.82

L'analisi statica della classe ServiceBean fornisce una serie di risultati che ci aiutano a comprendere le caratteristiche del codice e la sua qualità.

Iniziamo con la complessità della classe, che viene valutata come bassa (L) con un valore di RFC (Response for a Class) pari a 30. Questo valore indica che la classe ha un numero moderato di metodi richiamati da altre classi. Una complessità bassa è un aspetto positivo poiché rende il codice più chiaro e facile da comprendere.

Passando all'accoppiamento, notiamo che è valutato come alto (H) con un valore di SFRC (Sum of Fan-out for a Class) pari a 1. Questo indica che la classe dipende da un numero limitato di altre classi. Un alto accoppiamento può rendere il codice più difficile da mantenere e modificare in futuro, è importante prestare attenzione a questa metrica.

La dimensione della classe, valutata tramite il numero di linee di codice (LOC), è bassa (L) con un totale di 36 linee. Una dimensione ridotta è un buon segno poiché semplifica la comprensione del codice e ne facilita la manutenzione.

Passando alla mancanza di coesione, notiamo che la classe è valutata come alta (H) sia per LCOM (Lack of Cohesion of Methods) che per LTCC (Lack of Tight Class Cohesion). Questo suggerisce che i metodi della classe potrebbero non essere strettamente legati tra loro e potrebbero svolgere diverse responsabilità. Una mancanza di coesione può rendere il codice più difficile da comprendere e mantenere, è necessario considerare un'eventuale riorganizzazione dei metodi per migliorare la coesione.

La metrica CBO (Coupling Between Objects) indica il numero di classi dipendenti dalla classe ServiceBean. In questo caso, il valore di CBO è 21, il che significa che la classe dipende da 21 altre classi. È importante gestire attentamente le dipendenze per mantenere un basso accoppiamento complessivo del sistema.

Altre metriche come WMC (Weighted Methods per Class), LCAM (Lack of Cohesion Among Methods), NOM (Number of Methods) e ATFD (Access to Foreign Data) forniscono ulteriori informazioni sulla struttura e sulle caratteristiche della classe.

In conclusione, l'analisi statica della classe ServiceBean indica che il codice ha una buona complessità e dimensione, ma richiede un'attenzione particolare per migliorare l'accoppiamento e la coesione, che potrebbe essere risolta suddividendo la classe.

È consigliabile valutare attentamente le dipendenze e organizzare i metodi in modo più coeso. Questi risultati forniscono indicazioni utili per ottimizzare il codice e migliorare la qualità dell'applicazione di ticket generator.

Ecco una descrizione delle metriche utilizzate nell'analisi statica:

- **Lack of Cohesion:** Misura quanto bene i metodi di una classe sono correlati tra loro. L'elevata coesione (bassa mancanza di coesione) tende ad essere preferibile, poiché l'elevata coesione è associata a diversi tratti desiderabili del software tra cui robustezza, affidabilità, riusabilità e comprensibilità. Al contrario, una bassa coesione è associata a tratti indesiderabili come la difficoltà di mantenere, testare, riutilizzare o persino comprendere.
- **Complexity:** Implica la difficoltà di comprensione e descrive le interazioni tra un numero di entità. Livelli più elevati di complessità nel software aumentano il rischio di interferire involontariamente con le interazioni e quindi aumenta la possibilità di introdurre difetti quando si apportano modifiche.
- **Size:** La dimensione è una delle forme più antiche e comuni di misurazione del software. Misurato dal numero di righe o metodi nel codice. Un conteggio molto alto potrebbe indicare che una classe o un metodo sta tentando di svolgere troppo lavoro e dovrebbe essere suddiviso. Potrebbe anche indicare che la classe potrebbe essere difficile da mantenere.
- **Weighted Methods per Class (WMC):** La somma ponderata di tutti i metodi di classe rappresenta la complessità McCabe di una classe. È uguale al numero di metodi, se la complessità è considerata pari a 1 per ciascun metodo. Il numero di metodi e la complessità possono essere utilizzati per prevedere lo sviluppo, il mantenimento e la stima dello sforzo di test. Nell'ereditarietà se la classe base ha un numero elevato di metodi, influisce sulle sue classi figlie e tutti i metodi sono rappresentati nella sottoclasse. Se il numero di metodi fosse elevato, quella classe potrebbe essere specifica del dominio. Quindi sono meno riutilizzabili. Anche queste classi tendono a essere più soggette a cambiamenti e difetti.
- **Number of Children and Interfaces (#C&I):** indica il numero di classi figlie e interfacce che estendono o implementano la classe considerata.
- **Number of Coupled Classes (#C):** rappresenta il numero di classi con cui la classe considerata è accoppiata, cioè dipende da esse o interagisce con esse.
- **Coupling (Coupling):** indica il livello di accoppiamento della classe con le altre classi (basso, medio-basso, medio-alto o medio-alto-alto).
- **Depth of Inheritance Tree (DIP):** La posizione della classe nell'albero di ereditarietà. Ha un valore 0 (zero) per le classi root e non ereditate. Per l'ereditarietà multipla, la metrica mostra la lunghezza massima. Classe più profonda nell'albero dell'ereditarietà, probabilmente ereditata. Pertanto, è più difficile prevedere il suo comportamento. Anche questa classe è

relativamente complessa da sviluppare, testare e mantenere. Attributi di qualità correlati: Complessità

- **Coupling Between Object Classes (CBO):** Il numero di classi a cui è accoppiata una classe. Viene calcolato contando le altre classi i cui attributi o metodi sono utilizzati da una classe, oltre a quelle che utilizzano gli attributi o i metodi della classe data. Sono escluse le relazioni ereditarie. Come misura dell'accoppiamento, la metrica CBO è correlata alla riusabilità e alla testabilità della classe. Più accoppiamento significa che il codice diventa più difficile da mantenere perché i cambiamenti in altre classi possono anche causare cambiamenti in quella classe. Pertanto, queste classi sono meno riutilizzabili e richiedono più sforzi di test. Attributi di qualità correlati: Accoppiamento.
- **Response For a Class (RFC):** Il numero dei metodi che possono essere potenzialmente invocati in risposta a un messaggio pubblico ricevuto da un oggetto di una particolare classe. Include il grafico completo delle chiamate di qualsiasi metodo chiamato dal metodo di origine. Se il numero di metodi che possono essere richiamati in una classe è elevato, la classe è considerata più complessa e può essere altamente accoppiata ad altre classi. Pertanto, è necessario uno sforzo maggiore per testare e mantenere. Attributi di qualità correlati: Complessità.
- **Simple Response For a Class (SRFC):** Il numero dei metodi che possono essere potenzialmente invocati in risposta a un messaggio pubblico ricevuto da un oggetto di una particolare classe. Include metodi richiamati direttamente dalla classe. Se il numero di metodi che possono essere richiamati in una classe è elevato, la classe è considerata più complessa e può essere altamente accoppiata ad altre classi. Pertanto è necessario uno sforzo maggiore per testare e mantenere. Attributi di qualità correlati: Complessità.
- **Lack of Cohesion of Methods (LCOM):** Misura in che modo i metodi di una classe sono correlati tra loro. Bassa coesione significa che la classe implementa più di una responsabilità. Una richiesta di modifica da parte di un bug o di una nuova funzionalità, su una di queste responsabilità comporterà il cambiamento di quella classe. La mancanza di coesione influenza anche la comprensibilità e implica che le classi dovrebbero probabilmente essere divise in due o più sottoclassi. Attributi di qualità correlati: Coesione.
- **Number of Fields (NOF):** Il numero di campi (attributi) in una classe. Attributi di qualità correlati: Dimensione.
- **Number of Methods (NOM):** Il numero di metodi in una classe. Attributi di qualità correlati: Dimensione.

- **Number of Static Fields (NOSF):** Il numero di campi statici in una classe.
Attributi di qualità correlati: Dimensione.
- **Number of Static Methods (NOSM):** Il numero di metodi statici in una classe.
Attributi di qualità correlati: Dimensione.
- **Class-Methods Lines of Code (CM-LOC):** Numero totale di tutte le righe di metodi non vuote e non commentate all'interno di una classe. Attributi di qualità correlati: Dimensione.
- **Efferent Coupling (EF):** Il numero di classi in altri pacchetti da cui dipendono le classi nel pacchetto è un indicatore della dipendenza del pacchetto dalle esternalità. Attributi di qualità correlati: Accoppiamento, Accoppiamento in uscita.
- **Afferent Coupling (AC):** Il numero di classi in altri pacchetti che dipendono dalle classi all'interno del pacchetto è un indicatore della responsabilità del pacchetto. Attributi di qualità correlati: Accoppiamento, Accoppiamento in entrata.
- **C3:** Il valore massimo delle metriche di accoppiamento, coesione e complessità. Attributi di qualità correlati: accoppiamento, coesione, complessità.
- **Instability (I):** La metrica di instabilità viene utilizzata per misurare la relativa suscettibilità della classe ai cambiamenti. È definito come il rapporto delle dipendenze in uscita rispetto a tutte le dipendenze del pacchetto e accetta valori da 0 a 1. $I = \frac{EC}{EC+AC}$ dove: EC – dipendenze in uscita, AC – dipendenze in entrata
- **Abstractness (Abs):** La metrica di astrattezza viene utilizzata per misurare il grado di astrazione del pacchetto. l'astrattezza è definita come il numero di classi astratte nel pacchetto rispetto al numero di tutte le classi.
- **Normalized Distance (ND):** La metrica della distanza normalizzata viene utilizzata per misurare l'equilibrio tra stabilità e astrattezza ed è calcolata come $ND = |Abs + I - 1|$