Progetto di Gestione Ticket per Hotel

Contesto del Sistema

Il progetto prevede la realizzazione di un sistema di gestione ticket per un ambiente alberghiero. L'utente principale è l'ospite che soggiorna nella stanza d'albergo e ha la necessità di poter inserire delle segnalazioni attraverso un sistema di ticketing.

Struttura del Ticket

Ogni ticket rappresenta una segnalazione e contiene diverse informazioni fondamentali. La descrizione costituisce il campo principale dove l'utente può comunicare ciò che desidera segnalare. Il sistema registra automaticamente la data e l'ora di apertura del ticket attraverso il campo openDate, che memorizza il momento esatto in cui la segnalazione viene creata.

La gestione temporale del ticket include anche un campo closedDate che indica quando il ticket viene chiuso. Insieme alla data di chiusura esiste un campo closure, un testo descrittivo che specifica le modalità con cui il ticket è stato risolto e chiuso. È importante notare che questi campi relativi alla chiusura non esistono al momento della creazione del ticket, ma vengono popolati soltanto quando un concierge interviene e gestisce la segnalazione. Naturalmente ogni ticket possiede un identificativo univoco.

Attori del Sistema

Il sistema coinvolge due figure principali con responsabilità distinte. L'ospite rappresenta il primo attore e il suo unico caso d'uso consiste nell'inserimento di nuovi ticket attraverso la funzione insertTicket. Il concierge costituisce il secondo attore e dispone di due casi d'uso specifici: può visualizzare i ticket aperti tramite seeOpenTicket e può procedere alla chiusura dei ticket gestiti attraverso closeTicket.

Architettura delle API

Le API rappresentano i punti di accesso al sistema e fungono da interfaccia tra il frontend e la logica applicativa. Per questo progetto è necessario sviluppare una TicketAPI che espone tre endpoint fondamentali, ciascuno mappato su un indirizzo specifico e associato a un metodo HTTP appropriato.

Il primo endpoint utilizza il verbo GET all'indirizzo tickets e ha la responsabilità di restituire esclusivamente i ticket aperti. Il verbo GET viene impiegato convenzionalmente per operazioni di lettura e in questo caso invoca internamente il metodo ticketApi.findOpen per recuperare le segnalazioni ancora in gestione.

Il secondo endpoint adopera il verbo POST sempre sull'indirizzo tickets. Il verbo POST è lo standard per la creazione di nuove entità nel sistema e in questo contesto permette di creare un nuovo ticket attraverso il metodo ticketApi.newTicket. Questo endpoint viene utilizzato dall'ospite quando desidera aprire una nuova segnalazione.

Il terzo endpoint sfrutta il verbo PUT con un indirizzo parametrizzato nella forma tickets seguito dall'identificativo specifico del ticket da aggiornare. Il verbo PUT è dedicato alle operazioni di aggiornamento e consente al concierge di modificare lo stato del ticket, tipicamente per procedere alla sua chiusura con l'aggiunta delle informazioni di closure.

Esistono anche altri verbi HTTP standard come GET per recuperare un singolo ticket tramite il suo identificativo e DELETE per operazioni di cancellazione, sebbene quest'ultimo non sia stato specificato tra i requisiti principali del progetto.

Separazione tra API e Frontend

Un aspetto architetturale fondamentale riguarda l'evoluzione separata tra le API e il frontend. Questa separazione garantisce che lo strato delle API funzioni in modo completo e autonomo, permettendo potenzialmente a diversi client di interfacciarsi con lo stesso sistema backend. L'obiettivo è creare delle API robuste e ben definite che costituiscano un contratto stabile, indipendentemente dall'implementazione del frontend che le consuma.
Certamente, ecco una versione migliorata del testo fornito:

La fase iniziale dello sviluppo del sistema di gestione ticket per l'hotel prevede l'uso di Spring Initializr, uno strumento web che semplifica la creazione della

struttura di base di un progetto Spring Boot.

Per questo progetto sono state identificate cinque dipendenze fondamentali, che rappresentano l'ossatura tecnologica dell'applicazione e soddisfano i requisiti architetturali del sistema di ticketing:

- **Spring Web:** È il fulcro della comunicazione HTTP del sistema, fornendo gli strumenti necessari per creare i controller che ricevono ed elaborano le chiamate HTTP dai client.
- **MySQL Driver:** Permette all'applicazione di comunicare con il database MySQL, stabilendo la connessione per la gestione e la persistenza dei ticket.
- **Spring Data JPA:** Gestisce le entity (classi Java che rappresentano le tabelle del database) in modo automatico. Consente di creare i repository senza scrivere manualmente l'implementazione delle operazioni CRUD, generando automaticamente le query SQL necessarie e semplificando l'interazione con il layer di persistenza tramite JPA e Hibernate.
- **Spring Boot DevTools:** Migliora l'esperienza di sviluppo fornendo funzionalità come il riavvio automatico dell'applicazione ad ogni modifica del codice, accelerando il ciclo di sviluppo e testing.
- **Thymeleaf:** È il template engine per la visualizzazione dell'applicazione, creando e gestendo le pagine HTML che costituiscono la view del sistema e integrando dinamicamente i dati dal backend.

Maven è uno strumento di build automation per progetti Java che gestisce dipendenze, compilazione, test e creazione di pacchetti (JAR/WAR) tramite il file pom.xml, standardizzando il ciclo di vita dello sviluppo software.

Il file application.properties (o application.yml) è il principale file di configurazione di Spring Boot, dove vengono definite le impostazioni dell'applicazione come porta del server, configurazione del database, logging e profili di esecuzione, caricato automaticamente da src/main/resources senza necessità di registrazione esplicita.
2 passo → importo progetto creato da spirng initializr su eclipse, creo il database e lo collego

Documentazione Commentata del File application.properties

Ecco una versione professionale e ben strutturata dei commenti presenti nel file application.properties del progetto Ticket.

Nome Applicazione

La proprietà spring.application.name definisce l'identificativo univoco dell'applicazione all'interno dell'ecosistema Spring Boot. In questo caso è stato impostato il valore "ticket" per identificare il sistema di gestione delle segnalazioni alberghiere.

Configurazione Database MySQL

La sezione dedicata alla configurazione del database contiene tutti i parametri necessari per stabilire la connessione con MySQL. Il percorso verso il database viene definito attraverso spring.datasource.url che specifica l'indirizzo JDBC completo inclusi protocollo, host, porta e nome del database. Le credenziali di accesso vengono impostate tramite username e password dell'utente del database. Il connettore è stato già scaricato automaticamente da Maven durante la fase di inizializzazione del progetto grazie alla dipendenza MySQL Driver selezionata.

Configurazione Hibernate per la Creazione Automatica dello Schema

Hibernate è il framework ORM che gestisce la persistenza dei dati e offre diverse funzionalità per la sincronizzazione automatica dello schema del database. La proprietà spring.jpa.hibernate.ddl-auto controlla il comportamento di Hibernate riguardo alla gestione dello schema delle tabelle. Quando impostata su "update", questa proprietà fa sì che Hibernate aggiorni automaticamente il database ogni volta che vengono apportate modifiche alle entity Java, creando o modificando le tabelle in base alla struttura delle classi annotate.

Relazione tra JPA, Hibernate e JDBC

JPA rappresenta una libreria che permette di leggere dal database in modo automatico, offrendo una traduzione automatica delle tabelle del database in entità Java. Hibernate costituisce un tipo specifico di implementazione della specifica JPA, fornendo funzionalità concrete per l'Object-Relational Mapping. Spring Data JPA estende ulteriormente Hibernate aggiungendo ulteriori livelli di astrazione e automazione. Hibernate stesso è un ORM automatico che si appoggia su JDBC per l'effettiva comunicazione con il database. Lo stack completo si struttura quindi come Spring Data JPA che utilizza Hibernate, il quale a sua volta si basa su JDBC per l'accesso ai dati.

Visualizzazione e Formattazione delle Query SQL

La proprietà spring.jpa.show-sql abilitata su "true" permette di visualizzare in console tutte le query SQL che vengono eseguite da Hibernate. Questa funzionalità risulta estremamente utile durante la fase di debug per comprendere quali operazioni vengono effettuate sul database. Le query mostrate in console vengono rese più leggibili attraverso spring.jpa.properties.hibernate.format_sql che applica una formattazione in stile Hibernate al codice SQL generato.

Configurazione del Dialetto MySQL

La proprietà spring.jpa.database-platform specifica quale dialetto SQL deve essere utilizzato da Hibernate per generare le query. Il dialetto non rappresenta uno standard SQL generico ma una specializzazione specifica per MySQL attraverso org.hibernate.dialect.MySQLDialect. Questo permette a Hibernate di sfruttare le caratteristiche peculiari di MySQL e di generare query ottimizzate per questo particolare sistema di gestione database.
Dopo aver inizializzato il progetto con Spring Initializr, importato in ambiente di sviluppo e configurato il file `application.properties`, il passo successivo consiste nell'esporre una parte dell'applicazione al mondo esterno, partendo dalla definizione delle entità. Nel contesto specifico, si andrà a creare la classe `Ticket` all'interno del package `com.generation.ticket.model.entities`.

In Java, le annotation sono elementi fondamentali e parte integrante del sistema, non semplici aggiunte. Hanno un ruolo chiave nella progettazione, al pari di metodi come `equals`, `hashCode` e `toString`. In Spring Boot, le annotation forniscono metadati e istruzioni al framework, consentendo la configurazione automatica dei

componenti dell'applicazione senza la necessità di configurazioni XML complesse. Annotation come `@Component`, `@Service`, `@Repository`, `@Controller` e `@Bean` permettono a Spring di identificare automaticamente le classi tramite reflection, creare istanze dei bean, gestirne il ciclo di vita e iniettare le dipendenze necessarie.

Le annotation in Java sono metadati sintattici che forniscono informazioni aggiuntive al compilatore, alla JVM o ad altri strumenti di sviluppo, e si scrivono usando il simbolo `@` seguito dal nome dell'annotazione. Possono essere applicate a classi, metodi, variabili, parametri e package, rendendo il codice più espressivo e leggibile. A runtime, tramite reflection, possono modificare il comportamento dell'applicazione senza alterare la logica del codice stesso.

È fondamentale includere i getter e i setter nelle entità, altrimenti Spring non sarà in grado di accedere e gestire le proprietà dell'entità stessa.

Layer Repository e Implementazione Automatica

Dopo aver definito le entità che rappresentano le tabelle del database, si procede alla creazione del layer repository che si occupa della persistenza dei dati. L'interfaccia repository estende JpaRepository utilizzando due parametri generici: il primo indica il tipo dell'entità da gestire, mentre il secondo specifica il tipo della chiave primaria. Spring Data JPA implementa automaticamente questa interfaccia a runtime attraverso la classe SimpleJpaRepository, che costituisce l'implementazione concreta predefinita fornita dal framework. Questa classe implementa tutti i metodi dichiarati nell'interfaccia JpaRepository senza richiedere alcuna scrittura manuale di codice da parte dello sviluppatore.

Generazione Automatica dei Metodi attraverso Naming Convention

Spring Data JPA offre un meccanismo potente per la generazione automatica delle query basato sulle convenzioni di denominazione dei metodi. Quando si dichiara un metodo nell'interfaccia repository seguendo specifiche convenzioni di naming, Spring interpreta il nome del metodo e genera automaticamente l'implementazione corrispondente traducendolo in una query JPQL. Un esempio tipico è rappresentato dal metodo findByStatus che accetta un parametro String status e restituisce una lista di Ticket: Spring analizza il nome del metodo,

identifica la parola chiave "findBy" seguita dal nome del campo "Status", e genera automaticamente la query per recuperare tutti i ticket che corrispondono allo status specificato.

Layer REST Controller e Esposizione delle API

Il passo successivo consiste nella creazione di un controller REST annotato con @RestController, che rappresenta una specializzazione dell'annotazione @Controller. La differenza fondamentale tra queste due annotazioni risiede nel fatto che @RestController combina implicitamente @Controller e @ResponseBody, producendo automaticamente risposte in formato JSON o XML senza necessità di annotare individualmente ogni metodo. Questo comportamento risulta ideale per la costruzione di API RESTful dove le risposte devono contenere dati strutturati piuttosto che pagine HTML renderizzate.

Organizzazione dei Package per le API

Le classi che implementano le API dovrebbero essere organizzate in un package separato, tipicamente denominato qualcosa come com.generation.ticket.api. Questa separazione architetturale riflette il fatto che queste classi rappresentano la porzione dell'applicazione esposta pubblicamente via internet e costituiscono l'interfaccia di comunicazione con i client esterni. Mantenere questa separazione facilita la manutenzione del codice e rende più chiara la distinzione tra i diversi layer dell'applicazione.

Dependency Injection e Application Context

Quando si dichiara una dipendenza nel controller utilizzando l'annotazione @Autowired, si sta richiedendo a Spring di iniettare automaticamente un'istanza della classe specificata prelevandola dall'Application Context. L'Application Context rappresenta il contenitore IoC di Spring che si occupa di creare, configurare e gestire il ciclo di vita di tutti i bean dell'applicazione. Nel caso specifico del repository, Spring ha già creato automaticamente un'implementazione concreta basata su SimpleJpaRepository e l'ha registrata nel context durante la fase di avvio dell'applicazione. L'annotazione @Autowired si limita a prelevare quella istanza già esistente e a iniettarla nel campo o nel costruttore annotato.

Se la dipendenza richiesta tramite @Autowired non è disponibile nel context, l'applicazione non riesce ad avviarsi e termina con un errore durante la fase di startup. Questo comportamento garantisce che tutte le dipendenze necessarie siano soddisfatte prima che l'applicazione diventi operativa, evitando errori a runtime difficili da diagnosticare.

Mappatura degli Endpoint con @RequestMapping

L'annotazione @RequestMapping applicata a livello di classe definisce il percorso base per tutti gli endpoint gestiti da quel controller. Quando si specifica ad esempio @RequestMapping("ticketservice/api/tickets") sulla classe del controller, si stabilisce che tutti i metodi all'interno di quella classe risponderanno a URL che iniziano con quel prefisso. Nei singoli metodi si utilizzano poi annotazioni più specifiche come @GetMapping, @PostMapping, @PutMapping o @DeleteMapping per mappare i diversi verbi HTTP, specificando solo la porzione rimanente del percorso.

L'URL completo che deve essere utilizzato nei test con Postman o nel browser viene costruito concatenando il path definito a livello di classe con quello specificato nel singolo metodo. Questo approccio gerarchico permette di organizzare gli endpoint in modo logico e facilita la manutenzione quando è necessario modificare il percorso base di un'intera sezione dell'API senza dover intervenire su ogni singolo metodo.

Implementazione e Test degli Endpoint REST

Mappatura dei Metodi con le Annotazioni Spring

Una volta scritta la classe controller, si procede all'implementazione dei metodi che gestiranno le diverse operazioni e alla loro mappatura con le annotazioni appropriate. Quando si utilizza l'annotazione @GetMapping su un metodo specifico, si stabilisce che quel metodo verrà invocato ogni volta che un utente effettuerà una chiamata HTTP all'indirizzo configurato utilizzando il verbo GET. Nel caso del sistema di gestione ticket, quando un client effettua una richiesta all'indirizzo ticketservice/api/tickets con il verbo GET, Spring instrada

automaticamente la richiesta al metodo annotato con @GetMapping corrispondente.

Comunicazione in Formato JSON

I metodi del controller REST producono automaticamente risposte in formato JSON grazie all'annotazione @RestController applicata alla classe. JSON, acronimo di JavaScript Object Notation, rappresenta il formato standard per lo scambio di dati nelle moderne architetture API RESTful. La comunicazione con le API avviene bidirezionalmente attraverso questo formato: le API restituiscono oggetti serializzati in JSON come risposta alle richieste, e ricevono dati in formato JSON quando il client invia informazioni al server. Questa standardizzazione del formato di scambio garantisce l'interoperabilità tra diversi sistemi e linguaggi di programmazione.

Strategie di Testing delle API

Dopo aver implementato la prima API, diventa fondamentale verificarne il corretto funzionamento attraverso processi di testing. Esistono due approcci complementari per testare gli endpoint: il testing manuale e il testing automatizzato. Il testing manuale permette di esplorare rapidamente il comportamento dell'API durante la fase di sviluppo, mentre il testing automatizzato garantisce la verifica sistematica della correttezza del sistema anche dopo modifiche successive al codice.

Testing Manuale con Postman

Postman rappresenta lo strumento principale per il testing manuale delle API e si configura come un programma dedicato specificamente all'esecuzione di richieste HTTP. Questo software permette di costruire richieste HTTP complete specificando il metodo, l'URL, gli header, il body e altri parametri necessari, visualizzando poi la risposta del server in modo strutturato e leggibile. Postman offre funzionalità avanzate come la possibilità di salvare le richieste in collezioni, gestire variabili d'ambiente, e documentare automaticamente le API testate. Durante lo sviluppo del sistema di gestione ticket, Postman consente di verificare immediatamente se l'endpoint GET per recuperare i ticket aperti restituisce i dati corretti nel formato JSON atteso.

Testing Automatizzato con JavaScript

Parallelamente al testing manuale, è possibile implementare test automatizzati utilizzando JavaScript e framework specifici per il testing delle API. Questi test vengono scritti come script che eseguono richieste HTTP programmaticamente e verificano automaticamente che le risposte corrispondano alle aspettative definite. L'automazione dei test permette di eseguire rapidamente l'intera suite di verifiche ogni volta che viene apportata una modifica al codice, garantendo che le funzionalità esistenti continuino a operare correttamente e rilevando immediatamente eventuali regressioni introdotte durante lo sviluppo.

Implementazione dell'Endpoint POST per la Creazione di Nuovi Ticket

Mappatura del Metodo POST

Dopo aver verificato con successo il funzionamento della prima API attraverso la prova manuale con Postman, si procede all'implementazione della seconda API dedicata all'inserimento di nuovi ticket nel sistema. Questa operazione richiede l'utilizzo dell'annotazione @PostMapping, che identifica il metodo come gestore delle richieste HTTP POST. Il verbo POST rappresenta lo standard HTTP per la creazione di nuove risorse sul server, distinguendosi dal GET che viene utilizzato esclusivamente per la lettura dei dati esistenti.

Struttura del Metodo newTicket

Il metodo responsabile della creazione si presenta con la firma public Ticket newTicket e accetta come parametro un oggetto Ticket annotato con @RequestBody. Questa annotazione riveste un ruolo fondamentale nel processo di deserializzazione automatica: indica a Spring che il client invierà lo stato del ticket in formato JSON all'interno del corpo della richiesta HTTP, e che il framework dovrà convertire automaticamente quella rappresentazione JSON in un'istanza Java della classe Ticket. Il metodo restituisce l'oggetto Ticket salvato, che verrà automaticamente serializzato in JSON e inviato come risposta al client.

Inizializzazione Automatica dei Campi

All'interno del metodo, prima di procedere al salvataggio, vengono impostati alcuni campi che non devono essere specificati dal client ma vengono gestiti automaticamente dal server. La data e l'ora di apertura del ticket vengono impostate attraverso ticket.setOpenOn utilizzando LocalDateTime.now, che cattura il momento esatto in cui il ticket viene creato sul server. Questo approccio garantisce l'accuratezza temporale e previene manipolazioni da parte del client. Analogamente, lo stato del ticket viene impostato automaticamente su "Open" attraverso ticket.setStatus, assicurando che ogni nuovo ticket inizi con lo stato corretto indipendentemente dai dati inviati dal client.

Persistenza attraverso il Repository

Il salvataggio effettivo del ticket nel database avviene attraverso la chiamata repo.save passando l'oggetto ticket preparato. Il repository, precedentemente iniettato tramite @Autowired, si occupa di gestire l'interazione con il database attraverso JPA e Hibernate. Il metodo save genera automaticamente la query SQL INSERT necessaria, assegna un identificativo univoco al nuovo ticket attraverso l'auto_increment configurato sulla chiave primaria, e restituisce l'oggetto completo con tutti i campi popolati incluso l'ID generato dal database.

Configurazione degli Header HTTP

Quando si effettua una richiesta POST con Postman, la sezione Headers riveste un'importanza cruciale per la corretta comunicazione con il server. Gli header HTTP definiscono i metadati della richiesta specificando informazioni come il tipo di contenuto inviato, il formato accettato per la risposta, eventuali token di autenticazione e altre configurazioni necessarie. Nel caso specifico della creazione di un ticket, è necessario configurare l'header Content-Type con il valore application/json per comunicare al server che il corpo della richiesta contiene dati in formato JSON. Parallelamente, l'header Accept viene impostato anch'esso su application/json per indicare che il client si aspetta di ricevere la risposta nello stesso formato.

Composizione del Body della Richiesta

Il corpo della richiesta POST contiene i dati effettivi che devono essere inviati al server per la creazione della nuova risorsa. Nella sezione Body di Postman viene inserito il payload JSON che rappresenta lo stato iniziale del ticket da creare. Un esempio di payload potrebbe essere un oggetto JSON contenente il campo opening con una descrizione testuale della segnalazione, come "Manca Giovanni. Dove è Giovanni? Non è una vacanza senza un Giovanni." Questo JSON viene inviato come contenuto della richiesta HTTP e il server, grazie all'annotazione @RequestBody, lo deserializza automaticamente nell'oggetto Ticket Java corrispondente.

Differenze tra Verbi HTTP e Utilizzo del Body

Esiste una distinzione fondamentale nel modo in cui i diversi verbi HTTP gestiscono il corpo della richiesta. Nelle richieste GET il corpo viene completamente ignorato dal server, poiché questo verbo è destinato esclusivamente alla lettura di dati e tutte le informazioni necessarie vengono tipicamente passate attraverso i parametri dell'URL o gli header. Il corpo della richiesta assume invece rilevanza con i verbi POST e PUT, dove rappresenta il canale principale per trasmettere i dati al server. Con POST si inviano i dati necessari per creare un nuovo oggetto, includendo tutti i campi richiesti per l'inizializzazione della risorsa. Con PUT si invia la rappresentazione completa o parziale dell'oggetto esistente che deve essere aggiornato, specificando i campi modificati.

Relazione tra Header e Body

Mentre gli header HTTP si occupano di comunicare i metadati e le caratteristiche della richiesta, il body trasporta il contenuto effettivo dei dati che devono essere processati dal server. Questa separazione permette al server di comprendere prima come interpretare i dati ricevuti attraverso gli header, e successivamente di processare il contenuto vero e proprio presente nel body. Nel contesto del sistema di gestione ticket, l'header Content-Type application/json informa il controller Spring che deve utilizzare un deserializzatore JSON per convertire il body della richiesta, mentre l'header Accept application/json comunica che la risposta deve essere formattata utilizzando un serializzatore JSON prima di essere inviata al client.

Testing Completo delle API con Postman

Verifica dell'Endpoint PUT per l'Aggiornamento

Per testare l'endpoint PUT implementato nel metodo updateTicket, Postman richiede la configurazione di una richiesta HTTP completa con i parametri corretti. L'URL da utilizzare sarà del tipo localhost:8080/ticketservice/api/tickets seguito dall'identificativo specifico del ticket da aggiornare, ad esempio localhost:8080/ticketservice/api/tickets/1. Il metodo HTTP deve essere impostato su PUT e il corpo della richiesta deve contenere un oggetto JSON con almeno i campi id, status, closedOn e closure popolati con i valori desiderati. Il campo id è essenziale per identificare quale ticket deve essere aggiornato, mentre gli altri campi contengono i nuovi valori da applicare. Spring deserializza automaticamente questo JSON in un oggetto Ticket che viene passato al metodo updateTicket, permettendo l'aggiornamento selettivo dei campi specificati senza necessità di inviare l'intera entità completa.

Implementazione e Testing dell'Ultima API GET Parametrica

L'ultimo endpoint da implementare e testare è quello per il recupero di un singolo ticket dato il suo identificativo, configurato con l'annotazione @GetMapping("/{id}"). Questa API si distingue per la parametrizzazione dell'URL dove la porzione {id} viene sostituita dinamicamente dal valore fornito dal client. Il metodo findTicket riceve l'identificativo attraverso il parametro annotato con @PathVariable("id") int id, che collega automaticamente la parte variabile dell'indirizzo alla variabile Java del metodo. Poiché si tratta di una richiesta GET, non viene inviato alcun corpo e tutti i parametri necessari sono estratti direttamente dall'URL della chiamata.

Configurazione del Test in Postman per GET /{id}

Per testare questo endpoint con Postman, si configura una nuova richiesta con il metodo GET e l'indirizzo localhost:8080/ticketservice/api/tickets/1, dove il numero 1 sostituisce il segnaposto {id} nell'URL. Gli header Accept devono essere impostati su application/json per specificare il formato desiderato della risposta,

mentre non è necessario configurare alcun Content-Type poiché una richiesta GET non include body. Premendo il pulsante Send, Postman invia la richiesta al server che estrae l'ID 1 dall'URL, lo inietta nel parametro del metodo findTicket, recupera il ticket corrispondente dal database e restituisce i dettagli completi serializzati in JSON.

Differenze Architetturali tra le API Implementate

Le quattro API implementate nel sistema presentano pattern distinti che rispettano i principi RESTful. L'endpoint GET /api/tickets restituisce la collezione completa dei ticket aperti senza parametri nel path. L'endpoint POST /api/tickets crea nuovi ticket utilizzando il body JSON per trasmettere i dati di inizializzazione. L'endpoint PUT /api/tickets/{id} aggiorna ticket esistenti combinando parametri nel path per l'identificazione e body JSON per i nuovi valori. Infine, GET /api/tickets/{id} recupera dettagli specifici attraverso parametri estratti dall'URL senza body. Questa progressione da operazioni di collezione a operazioni su singole risorse riflette la maturità architetturale del sistema API.

Output Atteso dal Test GET tickets/1

Quando si esegue il test localhost:8080/ticketservice/api/tickets/1 con Postman, la risposta attesa consiste in un oggetto JSON contenente tutti i campi del ticket con identificativo 1. Il JSON includerà l'id, l'opening con la descrizione originale, l'openOn con la data di creazione, lo status corrente, il closedOn se il ticket è stato chiuso, e il closure con il testo di risoluzione se applicabile. La presenza di questo endpoint completa il ciclo di operazioni CRUD permettendo ai client di navigare fluidamente dall'elenco generale dei ticket ai dettagli specifici di ciascuna segnalazione attraverso il suo identificativo univoco.