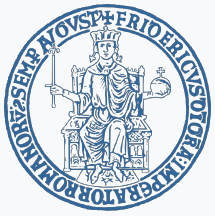


Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Corso di Laurea in Informatica

BUG BOARD 26

Sistema di Gestione Issue Collaborativo

Insegnamento

Ingegneria del Software

Docenti

Prof. Sergio Di Martino

Prof. Luigi Libero Lucio Starace

Candidati

GRUPPO INGSW2526_005

Virginia Antonia Esposito

Matricola: N86004987

Giuseppe Paolo Esposito

Matricola: N86005174

Anno Accademico 2025/2026

Indice

Documento di Specifica dei Requisiti Software

a. Glossario

- **Issue:** Segnalazione creata da un utente per documentare un problema, porre un quesito o richiedere l'inserimento di una funzionalità.
- **Bug:** Tipo di issue che indica un errore o un malfunzionamento del software.
- **Feature:** Tipo di issue che rappresenta una proposta di aggiunta di una nuova funzionalità.
- **Documentation:** Tipo di issue che riguarda errori o mancanze nella documentazione del progetto.
- **Question:** Tipo di issue utilizzata per porre quesiti o chiedere chiarimenti sul progetto.
- **Priorità:** Livello di urgenza associato a una issue.
- **Stato:** Indica la fase corrente della issue nel suo ciclo di esistenza (todo, assegnata, risolta).
- **Admin:** Utente con privilegi elevati, può creare team e progetti, assegnare issue, gestire utenti dei team e archiviare bug.
- **GenericUser:** Utente non ancora loggato, può solo effettuare il login.
- **LoggedUser:** Utente con privilegi base, visualizza, crea e risolve issues assegnate dagli admin dei team.
- **Team:** Gruppo di utenti responsabili di un progetto, gestito da un admin.
- **Filtro:** Funzione che consente di individuare issue in base a tipologia, stato, priorità o altri parametri.
- **Archivio:** Sezione dove i bug archiviati non sono più visibili nella lista principale; solo l'admin può archiviare.
- **Pull Request:** Permette di notificare la risoluzione locale di una issue, che potrebbe essere accettata o meno a discrezione dell'admin del team.

b. Modellazione dei Casi d'Uso

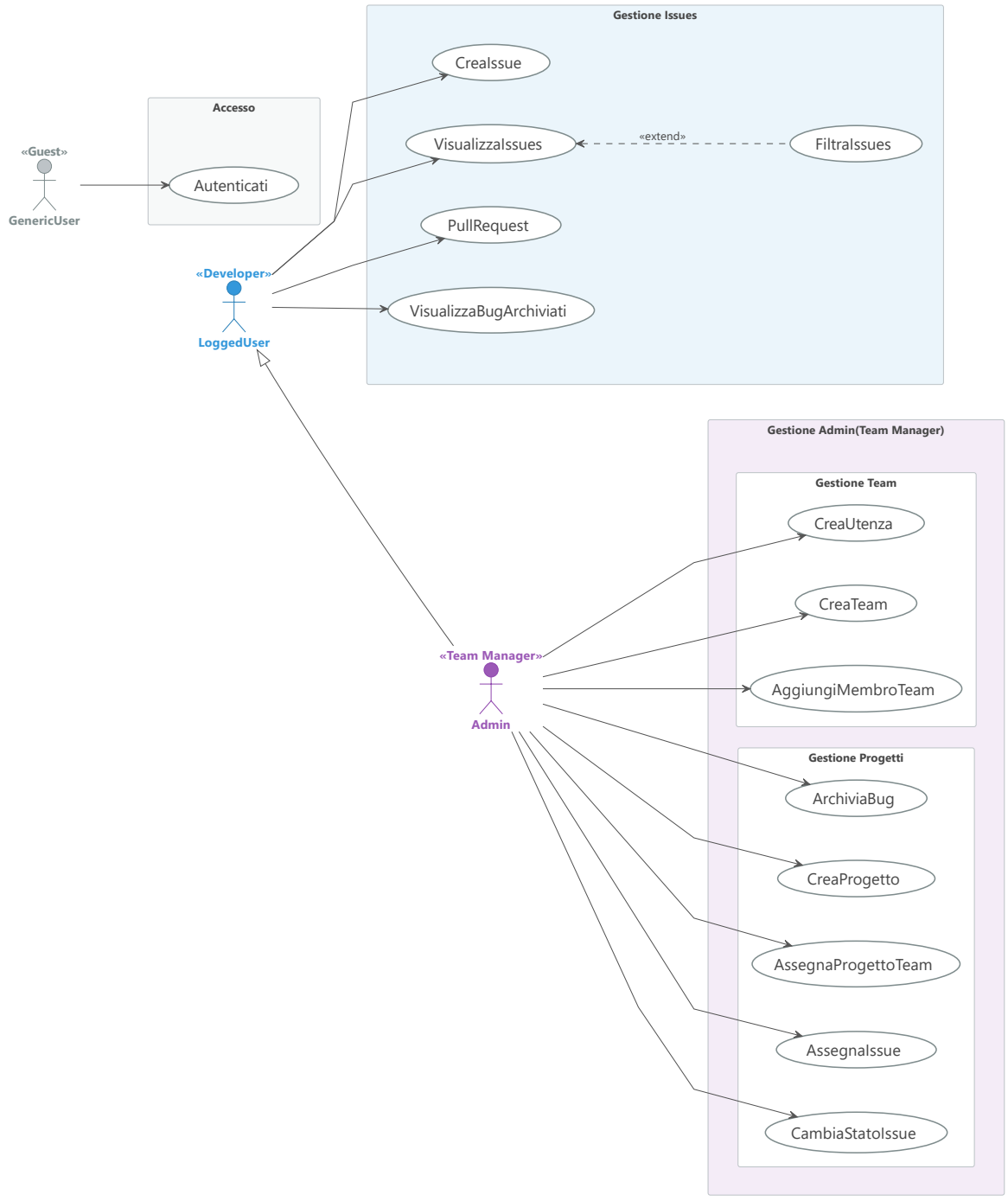


Figura 1: Diagramma Use Case di BugBoard26

c. Personas

1. Sheldon Cooper

Età: 35
Luogo di nascita: Texas, USA
Stato civile: Sposato
Titolo lavorativo: Senior Developer
Tratti caratteriali: Perfezionista, Narcisista, Competitivo, Poco incline al lavoro di squadra

GOALS:

- Dimostrare la propria superiorità nella risoluzione delle issue complesse
- Ottenere il ruolo di amministratore dell'azienda
- Ottenere riconoscimenti per il contributo apportato al team

INTERESSI:

- Fumetti: spende gran parte del suo tempo libero collezionandoli
- Fisica: la sua prima passione prima dell'informatica
- Videogiochi: utilizza i giochi strategici per tenere la mente allenata

BIO: Attualmente Sheldon lavora presso la SoftEngUniNA come Senior Developer e si occupa della analisi e risoluzione di bug complessi, oltre che alla manutenzione del codice.
Le sue eccezionali capacità lo rendono uno sviluppatore molto competente nella risoluzione di bug complessi. Tuttavia la sua natura perfezionista e la difficoltà nel lavorare con gli altri creano spesso tensioni nel team di sviluppo.
Sheldon preferisce interfacce chiare e ben strutturate, che gli consentano di concentrarsi sulla risoluzione dei bug piuttosto che sulla navigazione di interfacce complesse.

2. Armando Caputo

Età: 19
Luogo di nascita: Campobasso, IT
Stato civile: Single
Titolo lavorativo: Junior Developer
Tratti caratteriali: Insicuro, Taciturno, Determinato

GOALS:

- Dimostrare competenza agli amministratori e crescere professionalmente
- Imparare rapidamente tecniche e metodologie per la gestione e risoluzione dei bug
- Completare correttamente le issues assegnate senza dover chiedere aiuto

INTERESSI:

- Computer: Appassionato di hardware, sperimenta nuove configurazioni nel tempo libero
- Serie Tv: Utilizza le pause per distrarsi e staccare la mente dal lavoro
- Musica: Crea playlist personali per concentrarsi mentre lavora

BIO: Armando è al suo primo lavoro come sviluppatore presso SoftEngUniNA. La sua inesperienza e timidezza rendono complicato il lavoro in team, soprattutto quando deve interagire con colleghi più esperti. Per evitare di perdersi o confondersi tra le molte attività in corso, preferisce strumenti che gli permettano di concentrarsi sulle issue più urgenti e rilevanti, con informazioni complete e facilmente accessibili. Ha bisogno di un'interfaccia semplice, chiara e lineare, con indicazioni passo-passo e notifiche che lo guidino senza sovraccaricarlo di informazioni inutili.

3. Tony Stark

Età: 38
Luogo di nascita: New York, USA
Stato civile: Celibe
Titolo lavorativo: Amministratore di sistema
Tratti caratteriali: Carismatico, impulsivo, geniale, ironico, ambizioso, visionario

GOALS:

- Creare un ambiente di lavoro stimolante e competitivo dove i membri del team possano mostrare le proprie capacità.
- Coordinare il team di developer per garantire la risoluzione tempestiva dei bug e il rispetto delle scadenze.
- Vincere il premio di migliore amministratore dell'azienda ogni mese

INTERESSI:

- Tecnologia e innovazione: Creare e sperimentare nuovi dispositivi e software avanzato
- Comunicazione: Eccelle nella gestione dei rapporti con gli stakeholder, trovando sempre soluzioni creative
- Vita sociale: Party di lusso e feste esclusive

BIO: Tony è il visionario dietro la Stark Industries ed è conosciuto per il suo genio ingegneristico. Come amministratore in SoftEngUniNA porta la sua esperienza di gestione di progetti e team ad alto impatto, assicurandosi che gli sviluppatori abbiano sempre gli strumenti migliori per un lavoro efficace. La sua natura carismatica e ironica lo rendono un leader stimolante, anche se spesso impulsivo. Si irrita facilmente se le informazioni non sono immediatamente visibili e richiedono troppi click per essere raggiunte, odia non poter tenere tutto sotto controllo del suo team e non poter monitorare lo svolgimento delle Issues.

4. Amy Farrah Fowler

Età: 33
Luogo di nascita: California, USA
Stato civile: Sposata
Titolo lavorativo: Amministratrice di sistema
Tratti caratteriali: Introversa, metodica, analitica, paziente

GOALS:

- Mantenere l'efficienza del team e garantire qualità nello sviluppo del codice
- Coordinare i team di sviluppatori con rigidità e chiarezza
- Ridurre il tempo di risoluzione dei bug fornendo strumenti adatti e formazione mirata

INTERESSI:

- Neuroscienze: E' il suo campo scientifico preferito
- Attività sociali: Le piace partecipare a feste a tema
- Lettura scientifica: E' la sua attività preferita nel tempo libero

BIO: Amy, dopo la sua carriera nell'ambito della ricerca nelle Neuroscienze, ha deciso di dedicarsi all'amministrazione di progetti software presso SoftEngUniNA. Coordina i team di sviluppo e supervisiona la gestione delle issue assicurandosi un approccio metodico e garantendo un codice di alta qualità. Tuttavia, il suo perfezionismo e la sua introversione a volte rallentano il flusso decisionale del team e la comunicazione. Vuole gestire personalmente ogni dettaglio e fa fatica a delegare sotto pressione, ha bisogno di strumenti di comunicazione chiari e un sistema di monitoraggio delle Issues sintetico e impeccabile.

d. Requisiti non-funzionali e di dominio

(i) Requisiti non-funzionali

RNF-1 Prestazioni

- **RNF-P1 (Carico Utenti):** Il sistema deve supportare fino a 20 utenti registrati senza degradazione significativa delle prestazioni.

RNF-2 Sicurezza

- **RNF-S1 (Autenticazione):** Il sistema deve impedire l'accesso a utenti non autenticati a tutte le funzionalità e dati, eccetto la funzionalità di login.
- **RNF-S2 (Protezione Dati):** Tutte le password devono essere memorizzate in forma hash (con algoritmo bcrypt) e mai in chiaro nel database.

RNF-3 Affidabilità

- **RNF-R1 (Uptime):** Il back-end deve garantire un uptime maggiore del 95% in un periodo di monitoraggio di 30 giorni.

RNF-4 Usabilità

- **RNF-U1 (Efficienza):** Un utente autenticato deve poter creare una nuova issue in meno di 10 clic/interazioni dall'homepage.
- **RNF-U2 (Apprendibilità):** Un nuovo utente deve poter effettuare login, creare una issue e filtrarle entro 10 minuti dal primo utilizzo.

RNF-5 Manutenibilità

- **RNF-M1 (Disaccoppiamento):** Front-end e back-end devono eseguibili indipendentemente.

RNF-6 Sistema

- **RNF-C1 (Architettura):** Il sistema deve essere composto da almeno due componenti distribuiti: back-end (API REST) e front-end che comunica solo tramite API.
- **RNF-C2 (Tecnologia):** Il codice deve essere sviluppato in linguaggio object-oriented.

(ii) Vincoli di dominio

VD-1 **Protezione dei Dati Personali (GDPR):** Il sistema gestisce dati personali, trattati nel rispetto del **Regolamento UE 2016/679** e del **D.Lgs. 196/2003**.

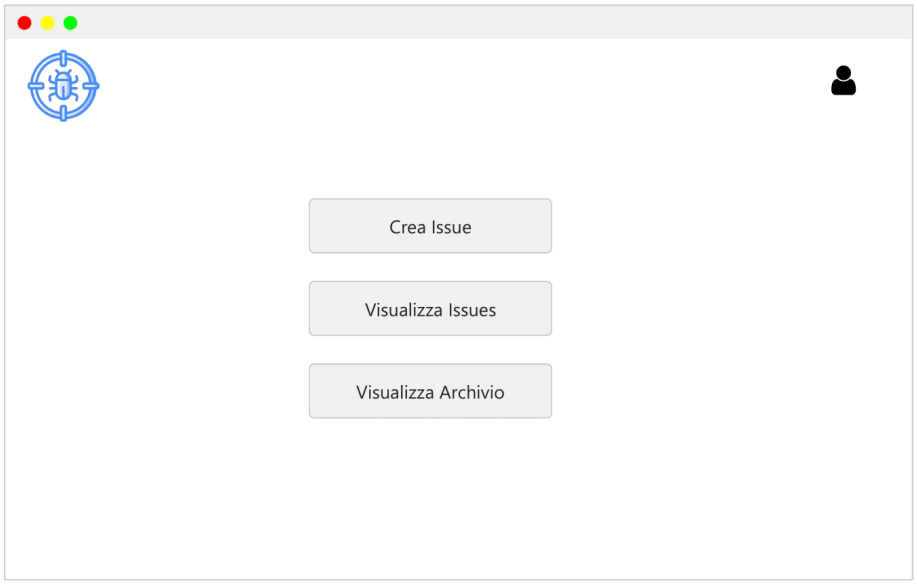
e. Formalizzazione Caso d'Uso: CreaIssue

(i) Descrizione testuale strutturata

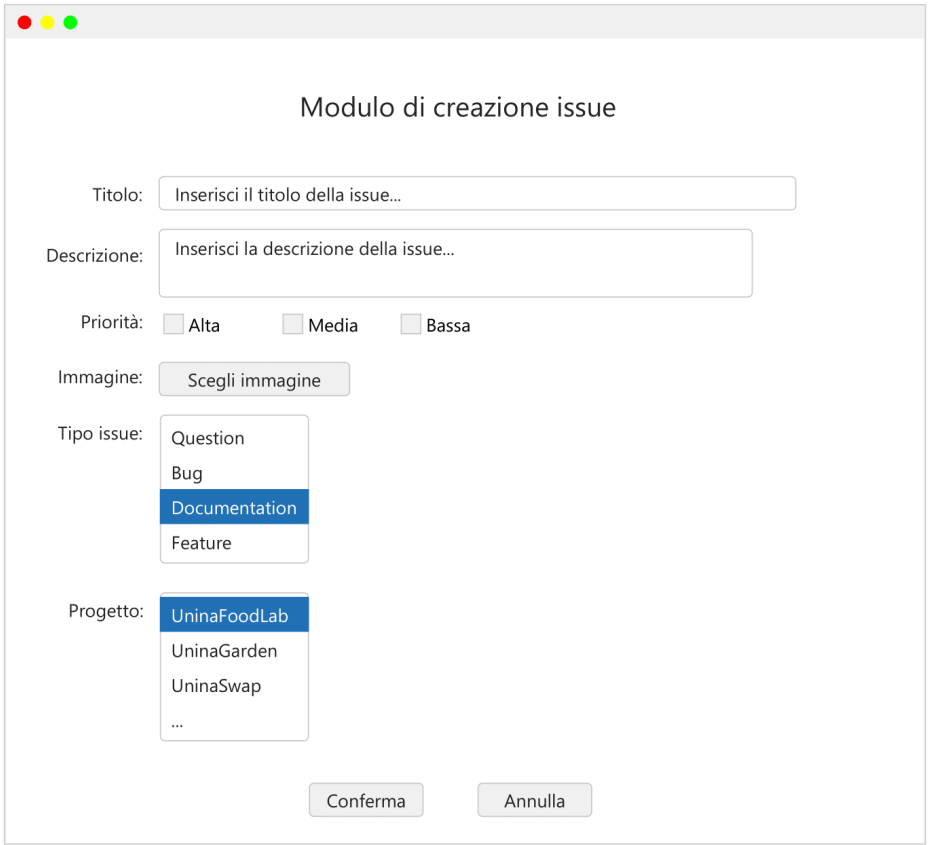
Use Case #1	**CreaIssue**		
Goal in Context	L'utente loggato vuole creare una nuova issue per un progetto a cui partecipa in modo che il sistema la salvi correttamente.		
Preconditions	Partecipare ad almeno un team di sviluppo di un progetto.		
Success End Conditions	Il sistema tiene traccia della creazione e salva correttamente la issue.		
Failed End Conditions	La issue non viene salvata e viene mostrato un messaggio di errore.		
Primary Actor	LoggedUser		
Trigger	L'utente clicca sul pulsante "Crea Issue" dalla Homepage.		
Main Scenario	**Step**	**LoggedUser**	**Sistema**
	1	Clicca "Crea Issue"	✗
	2	✗	Mostra form creazione issue (M2)
	3	Compila i campi e clicca "Conferma"	✗
	4	✗	Salva la issue e mostra conferma (M3)
	5	Clicca "Ok"	✗
	6	✗	Torna alla homepage (M1)
Extension #1 (Dati non validi)	**Step**	**LoggedUser**	**Sistema**
	3a	Inserisce dati errati o mancanti	✗
	3b	✗	Mostra errore di validazione (M4)
	3c	Corregge i dati e clicca "Ok"	✗
	3d	✗	Ritorna al passo 3 (Main Scenario)

Extension #2 (Errore di salvataggio)	**Step**	**LoggedUser**	**Sistema**
	4a	✗	Mostra errore di salvataggio (M5)
	4b	Clicca "Ok'	✗
	4c	✗	Ritorna al passo 4 (Main Scenario)
Extension #3 (Annullamento)	**Step**	**LoggedUser**	**Sistema**
	2a	Clicca "Annulla"	✗
	2b	✗	Mostra homepage (M1)
Notes	Campi obbligatori per ogni issue: **Titolo, Descrizione, Tipo e Progetto**. **M1**: Homepage **M2**: Form creazione issue **M3**: Schermata di avvenuto salvataggio **M4**: Schermata di errore validazione **M5**: Schermata di errore di salvataggio.		

(ii) Prototipazione visuale via Mock-up



M1



M2

Figura 2: Interfaccia di creazione issue - Homepage e Form creazione

Figura 3: Interfaccia di creazione issue - Notifiche varie

Documento di Design del Sistema

a. Descrizione dell'architettura proposta

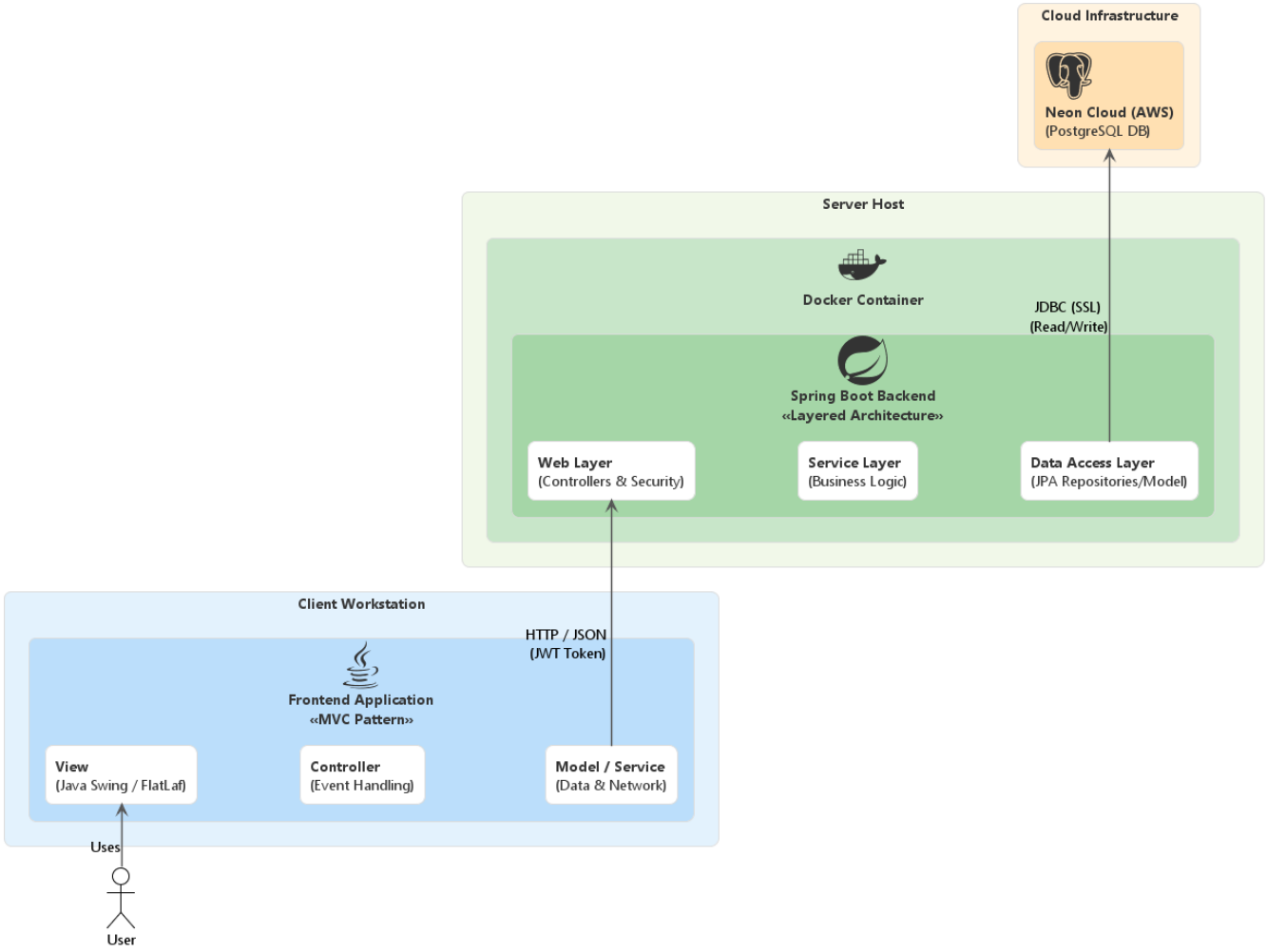


Figura 4: Architettura Distribuita di BugBoard26

Overview Architetturale

Il sistema **BugBoard26** adotta un'architettura distribuita **Client-Server**, progettata per massimizzare il disaccoppiamento (*Low Coupling*) e la coesione funzionale (*High Cohesion*) tra i sottosistemi. La comunicazione inter-processo avviene tramite protocollo HTTP su rete TCP/IP, aderendo allo stile architetturale **REST**.

Macro-Componenti del Sistema:

- **Backend (Server-Side):** Costituisce il *Core Business* del sistema. È implementato come un'applicazione monolitica modulare containerizzata. Espone un'interfaccia pubblica (API RESTful) per la manipolazione delle risorse (Issues, Utenti, Team, Progetti) ed è responsabile della validazione, dell'autorizzazione e della persistenza dei dati. L'architettura è **Stateless**: il server non mantiene lo stato della sessione utente tra le richieste, delegando l'autenticazione al passaggio di token JWT per ogni interazione.
- **Frontend (Client-Side):** Un'applicazione *Rich Desktop Client*. A differenza di un client web tradizionale, il frontend gestisce autonomamente la logica di presentazione, il rendering dell'interfaccia grafica e la validazione preliminare degli input, riducendo il carico di elaborazione sul server.

Pattern Architetturali e Design di Dettaglio

1. Backend: Layered Architecture (N-Tier)
Il backend adotta il pattern architetturale a strati (*Layered Architecture*), che impone una rigida *Separation of Concerns*. Ogni livello comunica esclusivamente con il livello inferiore:

- **Web Layer (Boundary):** Gestisce l'interfaccia verso l'esterno. Implementato tramite *REST Controllers*, si occupa del parsing delle richieste HTTP, della deserializzazione dei payload JSON in DTO e della validazione sintattica. Include la **Security Chain** (Filtri JWT).
- **Service Layer (Control):** Cuore della logica applicativa. Implementa i casi d'uso del sistema (es. *archiviazione bug*) e gestisce i limiti transazionali (`@Transactional`).
- **Persistence Layer (Data Access):** Isola l'accesso ai dati. Utilizza il pattern **Repository** e un ORM (*Hibernate*) per astrarre le operazioni CRUD sul database relazionale.

2. Criteri di Design Trasversali

- **Information Hiding (DTO):** Il sistema non espone mai le entità di persistenza (`@Entity`) direttamente nelle API, ma utilizza DTO mappati.
- **Inversion of Control (IoC):** Nel backend, la gestione delle dipendenze è affidata al Container Spring (Dependency Injection).

3. Frontend: Model-View-Controller (MVC)
Il client desktop struttura il codice secondo il pattern MVC:

- **Model:** Rappresenta lo stato dell'applicazione e i dati di business (DTO). Include i *Service* che incapsulano le chiamate REST.
- **View:** Costituita da componenti passivi (pannelli Swing estesi con *FlatLaf*) responsabili del rendering.
- **Controller:** Intercetta gli eventi utente (Listener), orchestra il flusso applicativo invocando il Model e aggiorna la View.
- **Gestione della Concorrenza (Responsive UI):** Per rispettare il *Single Thread Rule* di Swing, tutte le operazioni di I/O (rete) sono delegate a **Worker Threads** separati, prevenendo il blocco dell'interfaccia (EDT).

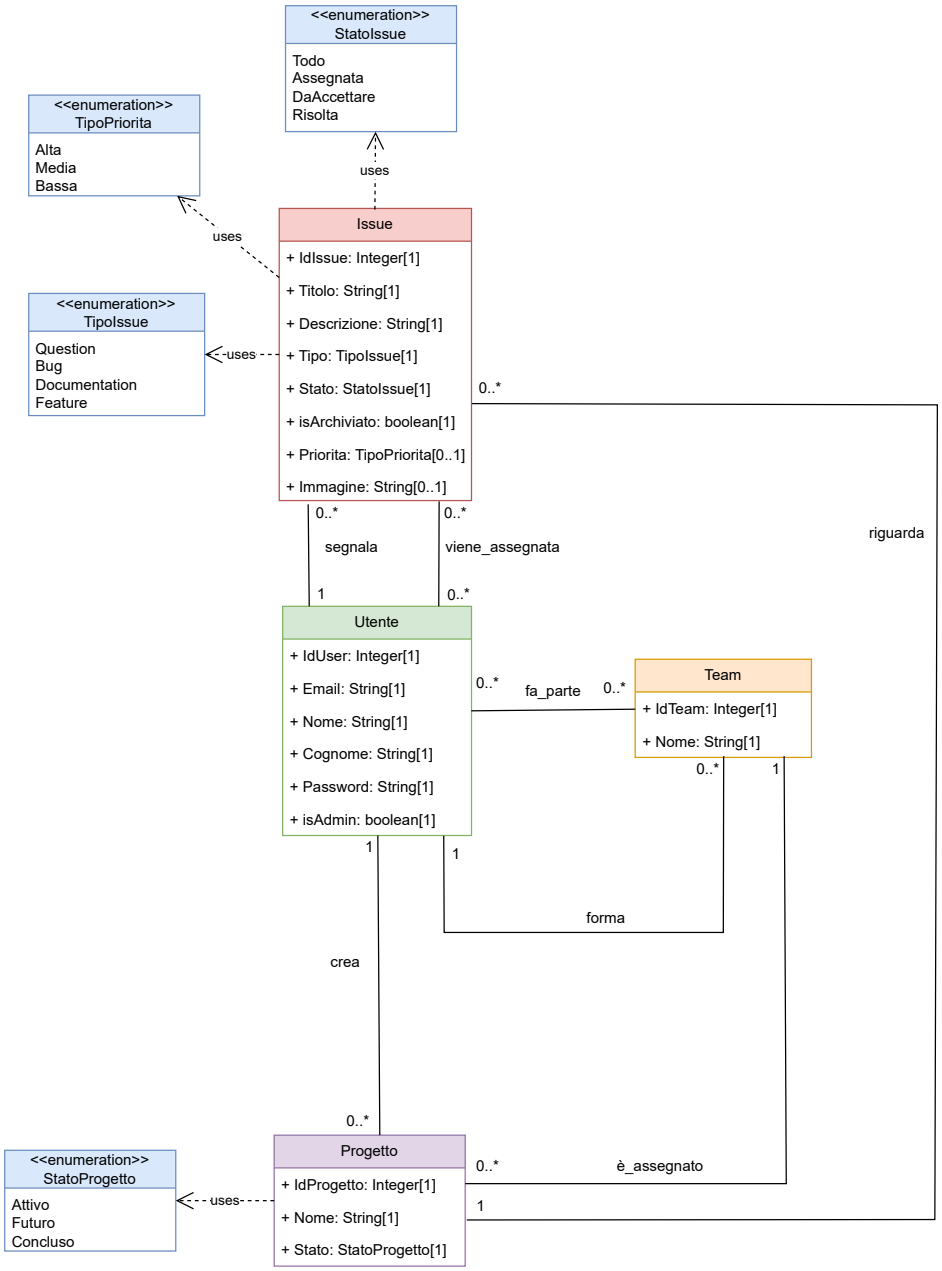
b. Scelte tecnologiche adottate

Stack Tecnologico	
Ambito	**Tecnologia e Motivazione Architetturale**
Backend Framework	**Spring Boot 3 (Java 21):** Scelto per la riduzione del codice di configurazione (*Boilerplate*) grazie alla *Convention over Configuration*. Fornisce supporto nativo per IoC, Security e JPA. Java 21 LTS garantisce stabilità e performance a lungo termine.
Frontend GUI	**Java Swing + FlatLaf:** Swing (parte del JDK) garantisce portabilità totale. L'integrazione di **FlatLaf** modernizza il *Look & Feel* (Flat Design, supporto HiDPI, temi Dark/Light), superando i limiti estetici dello Swing nativo.
Database/Cloud	**PostgreSQL su Neon (Hostato su AWS Frankfurt):** Utilizzato esclusivamente come RDBMS standard per la persistenza relazionale (ACID). ***Nota:*** Non vengono utilizzate funzionalità proprietarie di Neon (es. Auth, Serverless Functions) né servizi MBaaS. Neon funge unicamente da host per il database PostgreSQL, garantendo che la logica di accesso ai dati resti sotto il controllo del codice Java (Repository Pattern).
ORM & Dati	**Hibernate (JPA) & Jackson:** **Hibernate** gestisce il mapping Oggetto-Relazionale in modo trasparente. **Jackson** è lo standard industriale per la serializzazione/deserializzazione JSON ad alte prestazioni.

Stack Tecnologico (continua)	
Boilerplate Reduction	**Lombok:** Automatizza la generazione di metodi ripetitivi (Getter, Setter, Builder), migliorando la leggibilità e riducendo il codice sorgente (*Clean Code*).
API Documentation	**OpenAPI (Swagger UI):** Integrato via *SpringDoc*, genera automaticamente la documentazione interattiva delle API REST, facilitando il testing degli endpoint e l'integrazione frontend-backend.
DevOps	**Docker & Maven: Maven** gestisce il ciclo di vita del build. **Docker** garantisce la *parità ambientale* tra sviluppo e produzione, incapsulando il backend in container isolati.

Documento di Design del Software

a. Schema per la persistenza dati



Descrizione del Modello UML

Il livello di persistenza è implementato su **PostgreSQL**, incapsulato nello schema dedicato `BugBoard26`.

1. Entità Principali e Tipizzazione
Per massimizzare l'integrità dei dati, sono stati definiti tipi enumerativi (`ENUM`) per limitare i valori ammissibili di stati e priorità:

- **Utente:** Rappresenta l'attore che interagisce con il sistema, identificato univocamente dall'indirizzo email. La sicurezza è un requisito critico: la password non è salvata in chiaro ma come hash cifrato (`VARCHAR(60)`, BCrypt). L'attributo booleano `isAdmin` implementa un controllo degli accessi basato sui ruoli (RBAC), discriminando gli utenti che possono creare e amministrare nuovi Team, Issue e Progetti.

- **Team:** Entità di aggregazione che definisce il perimetro di collaborazione. Ogni Team è posseduto da un amministratore ed è caratterizzato da un nome univoco. La relazione con l'entità *Utente* è di tipo **Many-to-Many** (gestita dalla tabella associativa *Partecipanti*), permettendo flessibilità organizzativa: un utente può contribuire a più gruppi di lavoro contemporaneamente.

- **Progetto:** Unità logica contenuta univocamente all'interno di un Team (**One-to-Many**). Il ciclo di vita del progetto è governato da una macchina a stati finiti definita dall'enumerativo `Stato_Progetto` (`ATTIVO`, `FUTURO`, `CONCLUSO`); vincoli specifici (implementati via Trigger) impediscono la creazione di nuove attività su progetti ormai conclusi o lo spostamento di un progetto da un team all'altro.

- **Issue:** L'entità core del dominio, rappresentante l'unità di lavoro atomica. È fortemente tipizzata tramite gli Enum `Tipo_Issue` (es. BUG, FEATURE) e `Stato_Issue`. Oltre ai dati descrittivi, include un flag `IsArchiviato` che permette l' *archiviazione* (applicabile solo ai Bug risolti).

2. Relazioni e Cardinalità Le relazioni tra le entità sono state modellate per riflettere le dinamiche di collaborazione:

- **Utente ↔ Issue (1:N):** Relazione di segnalazione. Ogni issue è creata da un singolo utente; il sistema mantiene traccia dell'autore originale.

- **Utente ↔ Issue (N:M):** Gestita dalla tabella associativa `Assegnazioni`. Una issue può essere assegnata a più sviluppatori contemporaneamente.

- **Utente ↔ Team (1:N):** Ogni team è amministrato da un singolo utente con privilegi elevati, responsabile della gestione dei membri.

- **Utente ↔ Team (N:M):** Gestita dalla tabella associativa `Partecipanti`. Un utente può collaborare attivamente in più team distinti con lo stesso account.

- **Utente ↔ Progetto (1:N):** Ogni progetto è supervisionato da un responsabile (Admin) che li crea per il suo team.

- **Team → Progetto (1:N):** Relazione di ownership. Un team può gestire molteplici progetti (con uno solo attivo alla volta) , ma ogni progetto appartiene univocamente a un solo team.
- **Progetto → Issue (1:N):** Una issue esiste solo all'interno del contesto di un progetto specifico; non esistono issue "globali".

b. Diagramma delle classi di design

Poiché l'architettura è distribuita, il modello delle classi è stato suddiviso in due diagrammi distinti per **Backend** e **Frontend** al fine di migliorarne la leggibilità.

Backend Class Diagram

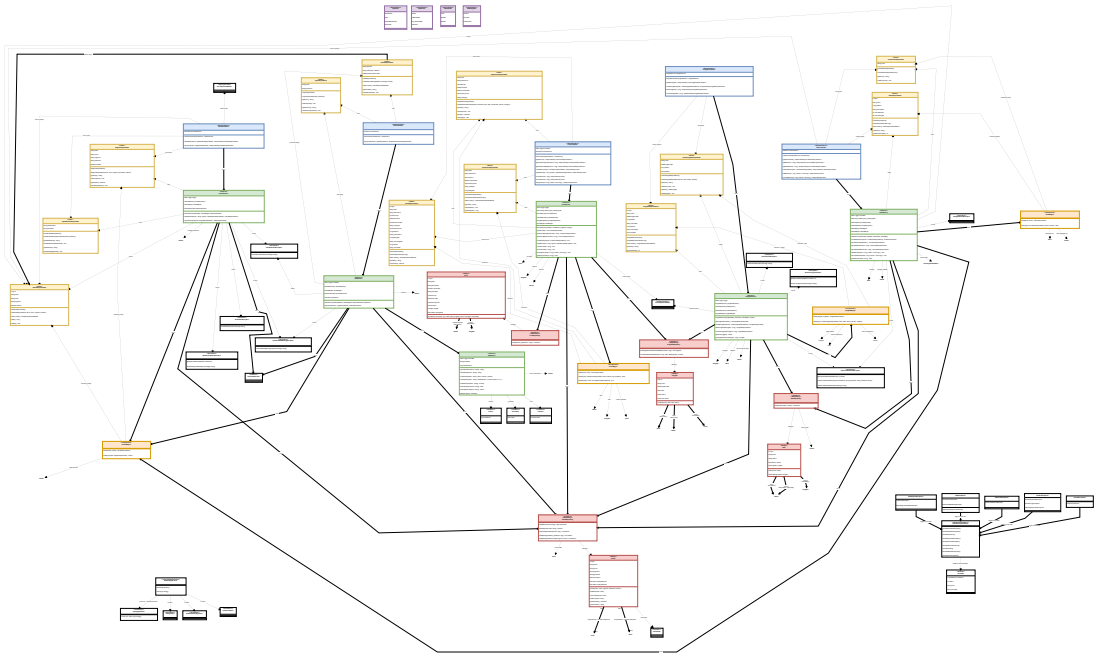


Immagine poco leggibile?
Cliccare qui per scaricare da Github il diagramma Backend ad alta risoluzione

Frontend Class Diagram

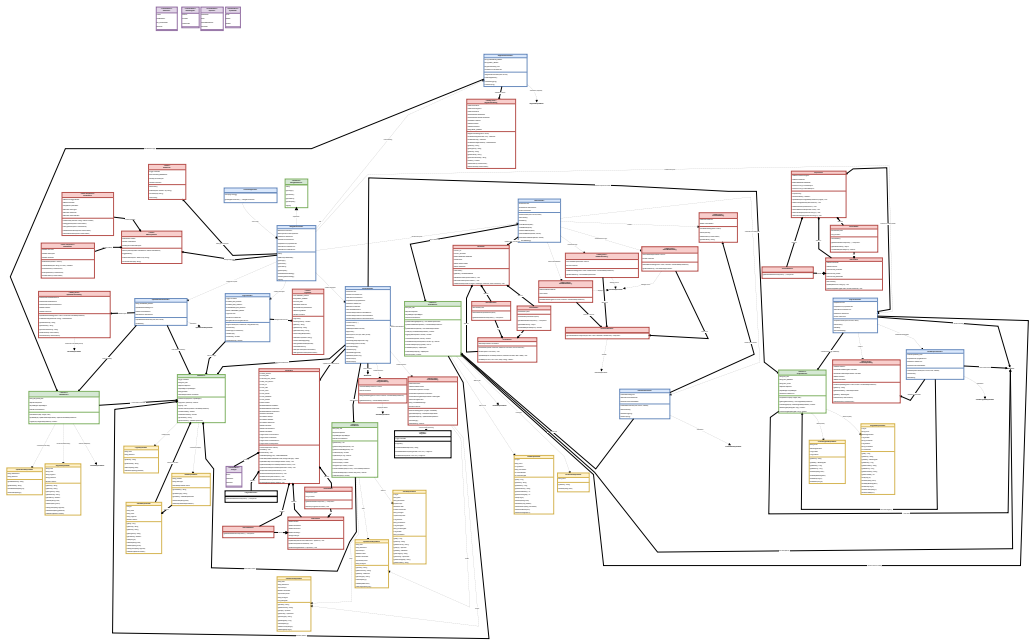


Immagine poco leggibile?
Cliccare qui per scaricare da Github il diagramma Frontend ad alta risoluzione

c. Descrizione e motivazione delle scelte di software design

La progettazione di dettaglio è stata guidata dai principi **SOLID** e dall'applicazione di **Design Pattern** per risolvere problemi ricorrenti in modo standardizzato. Di seguito sono dettagliate le scelte implementative principali.

1. Inversion of Control (IoC) & Dependency Injection

Nel backend, è stato evitato l'uso dell'operatore `new` per istanziare le dipendenze tra le classi di business. La gestione del ciclo di vita degli oggetti è delegata al *Spring IoC Container*.

- **Implementazione:** Le dipendenze (es. `IssueRepository` all'interno di `IssueService`) vengono iniettate tramite costruttore, sfruttando l'annotazione `@RequiredArgsConstructor` di Lombok che genera il codice boilerplate necessario.
- **Motivazione (DIP):** Questa scelta disaccoppia le classi di alto livello dai dettagli implementativi di basso livello. Inoltre, abilita il **Testing Unitario**: durante i test è possibile iniettare dei *Mock* (oggetti simulati) al posto delle implementazioni reali.

2. Pattern DTO e Mapper (Object Mapping)

Il sistema non espone mai le Entità di persistenza (`@Entity`) direttamente attraverso le API REST. Sono stati definiti oggetti **DTO** (Data Transfer Object) specifici per ogni operazione.

- **Implementazione:** Classi dedicate (es. `IssueMapper`, `UtenteMapper`) contengono la logica di conversione *Entity* $\leftrightarrow$ *DTO*. Questo centralizza la logica di trasformazione evitando codice sparso nei Controller.
- **Motivazione:**
  - **Information Hiding:** Previene l'esposizione di dati sensibili (es. password hashate) e dettagli interni del DB.
  - **Disaccoppiamento:** Modifiche allo schema del database non rompono il contratto API verso il client.
  - **Prevenzione Loop:** Evita problemi di ricorsione infinita nella serializzazione JSON di relazioni bidirezionali.

3. Repository Pattern (Data Access)

L'accesso ai dati è stato astratto tramite interfacce che estendono `JpaRepository`.

- **Implementazione:** Interfacce come `IssueRepository` definiscono i metodi di query, lasciando a Spring Data JPA l'implementazione concreta delle query SQL a runtime.
- **Motivazione:** Separa la logica di business (Service Layer) dalla logica di accesso ai dati. Permette di cambiare la tecnologia di persistenza o ottimizzare le query senza modificare il codice di business.

4. Builder Pattern (Creational Pattern)

Utilizzato estensivamente per la creazione di oggetti complessi, sia DTO che Entità del dominio.

- **Implementazione:** Tramite l'annotazione `@Builder` di Lombok.
- **Motivazione:** Migliora drasticamente la leggibilità del codice client e riduce la possibilità di errori nello scambio dei parametri nei costruttori.

5. Observer Pattern (Event Handling)

Nel Frontend Java Swing, l'interazione utente è gestita tramite il pattern Observer.

- **Implementazione:** I componenti della View (es. `JButton`) agiscono da *Subject*, mentre i metodi nei Controller (implementati tramite listener come `ActionListener`) agiscono da *Observer*.
- **Motivazione:** Disaccoppia la componente grafica dalla logica di risposta all'evento. La View non sa cosa succederà quando il bottone viene premuto, si limita a notificare l'evento.

6. Strategy Pattern (Global Exception Handler)

La gestione delle eccezioni è centralizzata utilizzando l'Advice di Spring.

- **Implementazione:** La classe `GlobalExceptionHandler` annotata con `@ControllerAdvice` intercetta le eccezioni lanciate in qualsiasi punto dello stack.
- **Motivazione:** Implementa il principio **DRY** e garantisce che il client riceva sempre risposte JSON strutturate in modo coerente (`ErrorDTO`), indipendentemente dall'origine dell'errore.

7. Chain of Responsibility (Security Layer)

La sicurezza è implementata come catena di filtri middleware.

- **Implementazione:** Il filtro `JwtAuthenticationFilter` intercetta le richieste, valida il token e passa il controllo al prossimo filtro nella `SecurityFilterChain`.
- **Motivazione:** Gestisce l'autenticazione come un aspetto trasversale, evitando di duplicare i controlli in ogni metodo di business.

8. Concorrenza nel Frontend (Worker Thread)

- **Problema:** Swing opera su un singolo thread. Eseguire chiamate API sull'EDT bloccherebbe la UI.
- **Soluzione:** I Controller delegano le operazioni di I/O a **Worker Threads** separati, aggiornando la UI solo al termine tramite `SwingUtilities.invokeLater`.
- **Motivazione:** Garantisce un'applicazione sempre reattiva.

d. Evidenza dell'uso di strumenti di versioning

La gestione della configurazione software è stata affidata a **Git**, con il repository remoto ospitato sulla piattaforma **GitHub**.

Statistiche del Repository

- **Repository URL:** `github.com/kiyx/progetto-swe-2026`
- **Branching Strategy:** Sviluppo diretto sul ramo principale `main`, garantendo la Continuous Integration.
- **Totale Commit:** ∼110

Figura 5: Grafico della frequenza dei commit (GitHub Insights)

e. Report di qualità del codice (Backend)

Il controllo della qualità del codice è stato integrato nel ciclo di sviluppo utilizzando **SonarCloud** per l'analisi statica (SAST), **SonarQube IDE** e **JaCoCo** per la verifica della copertura dei test. Il progetto rispetta il **Quality Gate** definito (Rating A su tutte le metriche principali).

Metriche Rilevate da SonarCloud

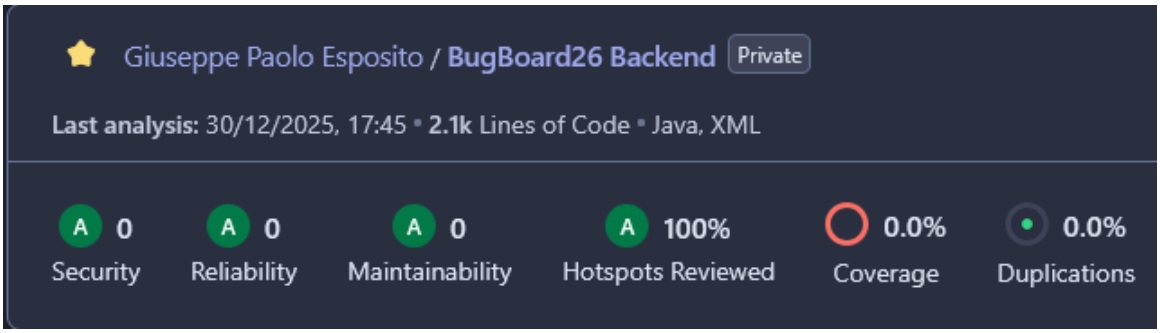


Figura 6: Dashboard di Qualità SonarCloud

Analisi dei Risultati:

- **Affidabilità (Reliability): Rating A.** Sono stati rilevati **0 Bug** critici o bloccanti.
- **Sicurezza (Security): Rating A. 0 Vulnerabilità.**
  Nota: Un "Security Hotspot" relativo alla disabilitazione della protezione CSRF è stato analizzato manualmente e marcato come **Safe**. Questa scelta è architetturalmente corretta e intenzionale poiché il sistema utilizza un'autenticazione Stateless basata su Token JWT (che non sfrutta i cookie di sessione), rendendo l'attacco CSRF non applicabile in questo contesto.
- **Manutenibilità (Maintainability): Rating A.** Il *Debito Tecnico* è assente. L'uso estensivo della libreria **Lombok** ha contribuito significativamente a questo risultato, eliminando centinaia di righe di codice ripetitivo (getter, setter, costruttori) che avrebbero altrimenti inflazionato la complessità ciclomatica e ridotto la leggibilità.
- **Code Coverage: 0.0%.**
  L'analisi riporta una copertura del codice nulla in quanto, al momento dello snapshot, non sono stati integrati test di unità automatici (JUnit). L'implementazione di una suite di test automatici completa è demandata a una fase successiva del ciclo di vita del software (consegna successiva)

f. Codice Sorgente sviluppato e Artefatti

Il progetto è organizzato come un **Monorepo Maven Multi-Modulo**. Questa struttura centralizzata permette di gestire sia il frontend che il backend in modo coerente. Per facilitare la valutazione, l'archivio compresso (`.zip`) consegnato include tutti i file di configurazione necessari (incluso il file `.env` con le credenziali per il database cloud), rendendo il sistema pronto all'uso.

Struttura della Directory

```
progetto-swe-2026/
 backend/                 # Modulo Server (Spring Boot)
    .env                  # Configurazione DB (Pre-caricata nel .zip)
    Dockerfile            # Configurazione Multi-Stage Build
    docker-compose.yml    # Orchestrazione container (Backend)
    src/                  # Codice sorgente Java
 frontend/                # Modulo Client (Java Swing)
    mvnw.cmd              # Maven Wrapper (per Windows)
    src/                  # Codice sorgente Java
 .github/workflows/       # Pipeline CI/CD (GitHub Actions)
 README.md                # Documentazione introduttiva
```

Istruzioni per Build ed Esecuzione

1. Avvio del Backend (Docker)
Il backend viene eseguito in un container Docker. Non è necessario avviare un DB locale in quanto il sistema si connette al database Cloud (Neon).

```
cd backend
docker-compose up --build
```

2. Avvio del Frontend (Maven)
Aprire un nuovo terminale ed eseguire il client Desktop:

```
cd ../frontend
mvn clean compile exec:java
```

Strategia di Deployment e CI

- **Docker Multi-Stage Build:** Il `Dockerfile` utilizza una strategia a due stadi (Build vs Runtime) per produrre un'immagine sicura e leggera basata su `eclipse-temurin:21-jre-alpine`.
- **CI Pipeline:** Configuratata su GitHub Actions (`build.yml`) per eseguire compilazione, test e analisi SonarCloud ad ogni push sul branch principale.