Documentazione BugBoard26

Gennaro Iacuaniello, Vincenzo R. Zumbolo

Anno accademico 2025/2026

Contents

1 Specifica dei Requisiti Software

1.1 Introduzione

BugBoard26 è una piattaforma per la gestione collaborativa di issue in progetti software. Il sistema consente a team di sviluppo di segnalare problemi relativi a un progetto, monitorarne lo stato, assegnarli a membri del team e tenere traccia delle attività di risoluzione.

1.2 Glossario

- Mockup: visualazzione grafica abbozzata dell'applicazione.
- Stati dei problemi: stati in cui possono trovarsi i vari problemi segnalati dagli utenti. Essi possono essere:
  - Todo: problema segnalato ma non ancora assegnato a nessuno sviluppatore.
  - Assigned: problema assegnato ad uno sviluppatore che lo risolverà.
  - Solved: problema risolto.
- Tipi di utente:
  - Utente non ancora autenticato.
  - Guest: utente autenticato che ha efffettuato la registrazione autonomamente.
  - Developer: utente registrato da un amministratore, ma non un amministratore esso stesso.
  - Admin: utente amministratore registrato da un altro amministratore
- Tipologie di problemi:
  - Question: richiesta di chiarimenti su qualche aspetto di un progetto.
  - Bug: segnalazione di problemi/malfunzionamenti di un progetto.
  - Documentation: segnalazione di problemi relativi alla documentazione di un dato progetto.
  - Feature: richiesta e/o suggerimento di nuove funzionalità per un dato progetto.
- UCD: "use case diagramm", diagrammi per descrivere i casi d'uso del sistema.

1.3 Diagrammi dei casi d'uso

Lista dei casi d'uso del sistema:

1. Registrarsi come guest (Sign Up As A Guest): registrazione autonoma da parte di un utente che intende segnalare problemi ma non è nè un developer nè un admin.

2. Creare una nuova utenza (Create Developer): possibilità per gli amministratori di creare degli account per la piattaforma per utenti di tipo developer o admin.

3. Effettuare il login (Perform login): modo per autenticare qualsiasi tipo di utente.

4. Segnalare un problema (Report issue): segnalazione di un problema da parte di un utente qualunque. Con questo tipo di segnalazione, è obbligatorio inserire un titolo, una descrizione e un tipo, ed è possibile, se lo si desidera, allegare un'immagine. Il problema appena segnalato viene salvato nello stato "todo".

5. Segnalare un problema come sviluppatore (Report issue developer): segnalazione di un problema da parte di un developer o di un admin. Con questo tipo di segnalazione, è obbligatorio inserire un titolo, una descrizione e un tipo, ed è possibile, se lo si desidera, allegare un'immagine e/o assegnare una priorità. Il problema appena segnalato viene salvato nello stato "todo".

6. Visualizzare i problemi segnalati (View Reported Issues): modo per un qualunque utente di visualizzare i problemi da esso segnalati. Se lo si desidera, è possibile ordinarli e/o filtrarli secondo vari criteri, come come tipologia, stato o priorità.

7. Visualizzare i problemi assegnati (View Assigned Issues): modo per un developer o un admin di visualizzare i problemi ad esso assegnati. Se lo si desidera, è possibile ordinarli e/o filtrarli secondo vari criteri, come come tipologia, stato o priorità.

8. Visualizzare dashboard dei problemi (View Issues DashBoard): visualizzazione, da parte degli amministratori, di una vista riepilogativa dei probelmi, che mostra informazioni aggregate su di essi, come il numero di problemi aperti, quelli assegnati per utente e il tempo medio di risoluzione (aggregato e per utente).

9. Visualizzare report di un team (View Team Report): visualizzazione, da parte degli amministratori, di un report mensile del lavoro di un dato team, che mostra informazioni (aggregate e per singolo utente) come il numero di problemi aperti/gestiti e il tempo medio di risoluzione.

10. Assegnare un problema ad uno sviluppatore (Assign Issue To Developer): assegnazione, da parte di un amministratore, di un problema ad uno sviluppatore che dovrà risolverlo.

11. Assegnazione di un'etichetta (Assign Label): qualunque utente può assegnare un numero qualunque di etichette personalizzate ad un problema da esso segnalato.

1.4 Target di utenti del sistema

BugBoard26 è un'applicazione destinata ad una qualsiasi azienda di sviluppo software. Gli utenti che potranno utilizzzarla sono quindi sia sviluppatori/amministratori delle diverse aziende che la scaricheranno, sia gli utenti dei loro prodotti (che potranno registrarsi come ospiti).

1.5 Requisiti

1.5.1 Requisiti non funzionali

- Sicurezza del sistema: il sistema deve garantire privacy e sicurezza dei dati dei propri utenti, le password saranno criptate.
- Stato iniziale: quando un qualunque utente segnala un nuovo problema, il sistema la salva in uno stato "todo"

1.5.2 Requisiti di sistema

- Gli utenti non ancora registrati devono essere in grado di registrarsi come ospiti.
- Gli utenti registrati devono essere in grado di autenticarsi e di accedere così al sistema.
- Gli amministratori devono essere in grado di creare nuove utenze, specificando se sono di tipo sviluppatore o amministratore, ma non di creare nuovi ospiti.
- Gli utenti devono essere in grado di segnalare un problema, specificando obbligatoriamente titolo, descrizione e tipologia, e, se lo desiderano, possono allegare un'immagine. Inoltre, se l'utente che effettua la segnalazione è uno sviluppatore oppure un amministratore, se lo desidera può assegnare una priorità al problema appena segnalato.
- Tutti gli utenti devono essere in grado di vedere un recap dei problemi da essi segnalati.
- Solo gli sviluppatori e gli amministratori devono essere in grado di vedere un recap dei problemi ad essi assegnati.
- Solo gli amministratori devono essere in grado di visualizzare una dashboard dei problemi, che gli permette di visualizzare informazioni aggregate su di essi, come il numero di problemi aperti, quelli assegnati per utente e il tempo medio di risoluzione (aggregato e per utente).
- Solo gli amministratori devono essere in grado di visualizzzare un report mensile del lavoro di un dato team, che mostra informazioni (aggregate e per singolo utente) come il numero di problemi aperti/gestiti e il tempo medio di risoluzione.
- Solo gli amministratori devono essere in grado di assegnare un problema ad uno sviluppatore che dovrà risolverlo.
- Ogni utente deve poter assegnare un numero qualunque di etichette personalizzate ad un problema da esso segnalato.

1.6 Formalizzazione dei casi d'uso significativi

1.6.1 Descrizione testuale strutturata con il formalismo tabellare di A. Cockburn

USE CASE #4	Segnalazione di una issue da parte di un developer o di un admin	
Goal in Context	Un developer o un admin vuole segnalare una nuova issue	
Preconditions	L'utente è stato autenticato	
Success End Condition	La nuova issue appena segnalata viene aggiunta alla lista delle issue in stato todo	
Failed End Condition	Nessun cambiamento nel sistema	
Primary Actor	Un utente developer o admin	
Trigger	Click del tasto "Segnala un problema"	
Main Scenario		
Step n.	**Developer o admin**	**Sistema**
1	Clicca il tasto "Segnala un problema"	
2		Mostra M1
3	Inserisce i dati obbligatori per segnalare una issue (titolo, descrizione, tipo) e clicca il tasto di conferma	
4		Il Sistema salva la nuova issue, mettendola in stato "todo" e mostra M2
5	Clicca il pulsante ok	
6		Mostra M7 e termina il caso d'uso
Extension A: L'utente non inserisce un titolo		
Step n.	**Developer o admin**	**Sistema**
3a1	Inserisce i dati per segnalare la issue, ma non inserisce un titolo, e clicca il tasto di conferma	
3a2		Mostra M3
3a3	Clicca il pulsante ok	
3a4		Mostra M1 con i dati precedentemente inseriti dall'utente e torna allo step 3 del Main scenario
Extension B: L'utente non inserisce una descrizione		
Step n.	**Developer o admin**	**Sistema**
3b1	Inserisce i dati per segnalare la issue, ma non inserisce una descrizione, e clicca il tasto di conferma	
3b2		Mostra M4
3b3	Clicca il pulsante ok	
3b4		Mostra M1 con i dati precedentemente inseriti dall'utente e torna allo step 3 del Main scenario

Continua nella pagina successiva

Tabella 1 – continua dalla pagina precedente

Step n.	**Developer o admin**	**Sistema**
Extension C: L'utente non inserisce un tipo		
Step n.	**Developer o admin**	**Sistema**
3c1	Inserisce i dati per segnalare la issue, ma non inserisce un tipo, e clicca il tasto di conferma	
3c2		Mostra M5
3c3	Clicca il pulsante ok	
3c4		Mostra M1 con i dati precedentemente inseriti dall'utente e torna allo step 3 del Main scenario
Extension D: L'utente inserisce una priorità non valida		
Step n.	**Developer o admin**	**Sistema**
3d1	Inserisce i dati per segnalare la issue, ma inserisce una priorità non valida, e clicca il tasto di conferma	
3d2		Mostra M6
3d3	Clicca il pulsante ok	
3d4		Mostra M1 con i dati precedentemente inseriti dall'utente (tranne la priorità) e torna allo step 3 del Main scenario
Extension E: L'utente inserisce anche un'immagine		
Step n.	**Developer o admin**	**Sistema**
3e1	Dopo aver inserito i dati per segnalare la issue, clicca il pulsante "allega un'immagine", seleziona l'immagine desiderata e clicca il tasto di conferma	
3e4		Il Sistema salva la nuova issue, mettendola in stato "todo" e salvando anche l'immagine allegata, infine mostra M6 e torna allo step 5 del Main scenario
Extension F: L'utente inserisce anche una priorità		
Step n.	**Developer o admin**	**Sistema**
3f1	Inserisce i dati per segnalare la issue, inserendo anche una priorità, e clicca il tasto di conferma	
3f2		Il Sistema salva la nuova issue, mettendola in stato "todo" e salvando anche la priorità assegnata, infine mostra M6 e torna allo step 5 del Main scenario
Extension G: L'utente clicca il pulsante per tornare indietro		
Step n.	**Developer o admin**	**Sistema**
3g1	Clicca il pulsante per tornare indietro	

Continua nella pagina successiva

Tabella 1 – continua dalla pagina precedente

Step n.	Developer o admin	Sistema
3g2		Mostra M7 e termina il caso d'uso

1.6.2 Prototipazione visuale via Mock-up delle relative interfacce utente

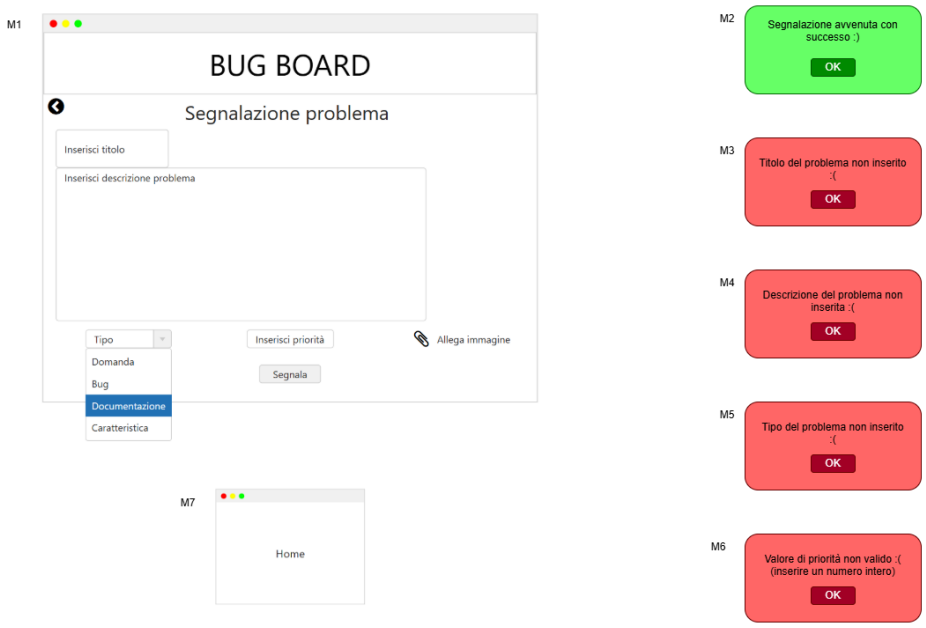

Figure 1: Mockup per lo use case 4

2 Design del sistema

Il sistema è stato sviluppato con un'architettura client-server.

2.1 Backend

Il lato server corrisponde al modulo "Backend" dell'applicazione, ed è stato sviluppato utilizzando il framework Spring Boot e seguendo un'architettura a livelli, in modo da dividere al meglio le responsabilità sia tra i vari livelli, sia tra le varie classi interne ad essi, seguendo così il Single Responsability Principle.

Model

Il primo livello è quello del model. Esso rappresenta i dati seguendo la logica di business del sistema.

Controller

Il secondo livello è quello del controller. Esso riceve le richieste da parte dei client tramite API Rest, e comunica con il livello inferiore (DAO) per gestirle.

DAO

Il terzo livello è quello dei DAO. Esso gestisce la comunicazione con un database. Per generalità del codice, e per favorire eventuali modifiche implementative future, è presente un primo livello di interfacce DAO, implementate poi nello specifico per comunicare con un database online NeonDB, basato su PostgreSQL e che garantisce ottime performance e un'availability di "tre nove" (99.95%).

La comunicazione tra controller e DAO è gestita tramite dei Data Transfer Object (DTO), utilizzati anche nelle chiamate Rest.

Inoltre, per rendere sicure le chiamate Rest, è stata utilizzata l'autenticazione tramite Token, in particolare JSON Web Token. Le uniche API pubbliche sono quelle di Login e Registrazione, per tutte le altre, si controlla che l'utente abbia inserito nella chiamata un header con un token di autenticazione valido.
Sempre per motivi di sicurezza, dati privati (come le credenziali di accesso al database e la chiave segreta per JWT) sono memorizzati in un file ".env", non caricato sulla repository online.

Nel backend è inoltre presente un dockerfile, che permette di creare un'immagine docker con il seguente comando: "docker build -t backend-app .".
Per avviare poi il relativo container, basta eseguire il comando:
"docker run -d -p 8080:8080 –env-file .env –name backend-container backend-app"

2.2 Frontend

Il lato client corrisponde al modulo "Frontend" dell'applicazione, ed è stato sviluppato seguendo un'architettura a livelli, in modo da dividere al meglio le responsabilità sia tra i vari livelli, sia tra le varie classi interne ad essi, seguendo così il Single Responsability Principle.

Graphical user interface

Il primo livello è quello di presentazione, ossia dell'interfaccia grafica che viene mostrata all'utente (GUI). Essa è stata realizzata in Java Swing, utilizzando il "Look & Feel" della libreria FlatLaf.

Controller

Il secondo livello è quello del controller. Esso si occupa principalmente di mandare richieste al server tramite API Rest, per poi comunicare le risposte alla GUI.

La comunicazione tra controller e GUI è gestita tramite dei Data Transfer Object (DTO), utilizzati anche nelle chiamate Rest. Per mappare gli oggetti nei JSON per le chiamate Rest, è stato utilizzato l'object mapper della libreria Jackson.

3 Design del software

3.1 Persistenza dei dati

Per salvare i dati in maniera persistente, è stato scelto l'utilizzo del database online NeonDB, basato su PostgreSQL e che garantisce ottime performance e un'availability di "tre nove" (99.95%).

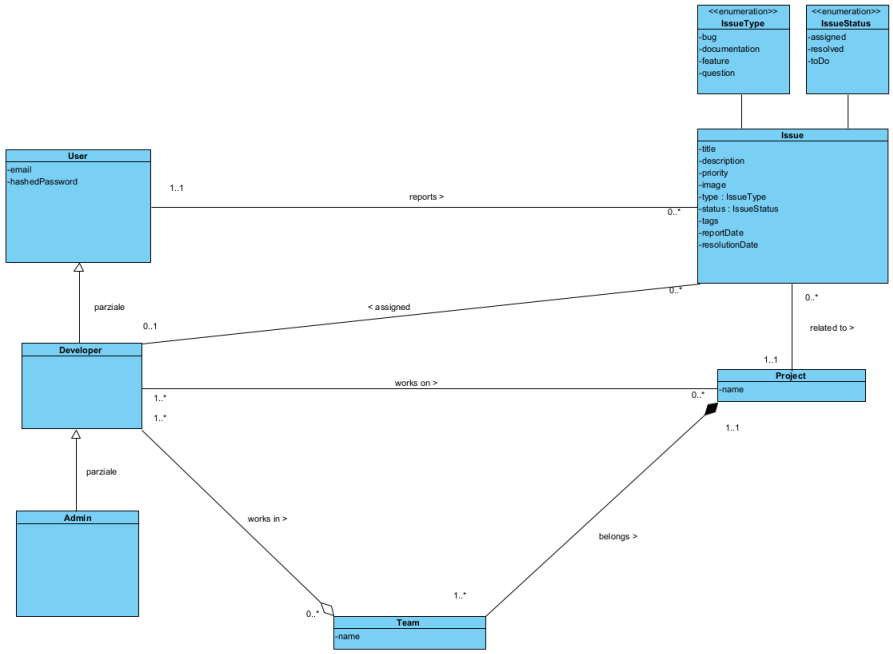

Figure 2: UML iniziale database

L'UML iniziale è poi stato ristrutturato seguendo i seguenti 6 passaggi:

1. Analisi delle ridondanze: si è deciso di mantenere le ridondanze dericanti dalle relazioni tra "Developer", "Team" e "Project", perchè, date le operazioni che verranno effettuate su questi dati, è utile avere, per esempio, sia la lista di tutti i team di un progetto, sia la lista di tutti i developer di un progetto (senza dover passare per ogni team per vedere i developer che ne fanno parte).

2. Eliminazione delle generalizzazioni: si è deciso di "accorpare la figlia nel padre" sia per la generalizzazione tra "Admin" e "Developer", sia per poi tra la nuova classe risultante e "Developer". Per questo, nella classe "User" risultante, sarà aggiunto un nuovo attributo "user_type", che potrà assumere 3 valori: 0 guest, 1 developer, 2 admin.

3. Eliminazione Attributi Multivalore: l'attributo "tags" dell'entità "Issue", che di base può essere una lista di stringhe, verrà trattato come un'unica stringa, in cui i vari tag sono separati dal carattere ";".

4. Eliminazione Attributi Strutturati: non sono presenti attributi strutturati.

5. Partizionamento/accorpamento di entità e associazioni: non sono presenti partizionamenti o accorpamenti da effettuare.

6. Scelta degli identificatori primari: per "User", "Team", "Project" e "Issue", aggiungiamo rispettivamente "user_id", "team_id", "project_id" e "issue_id" come chiavi primarie.

L'UML ottenuto dalla ristrutturazione è quindi il seguente:

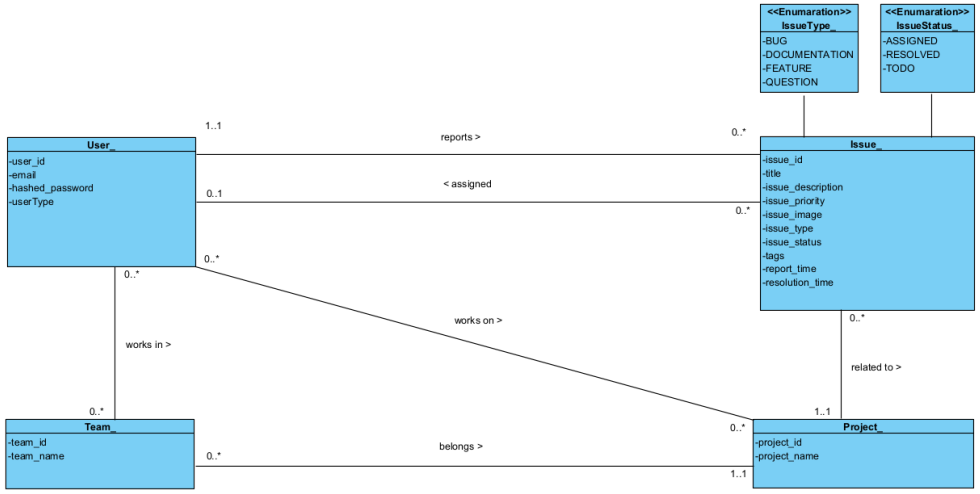

Figure 3: UML ristrutturato database

A questo punto, lo schema logico ricavato a seguito della ristrutturazione è il seguente:

- **User**(<u>user_id</u>, email, hashed_password, user_type)
- **Project**(<u>project_id</u>, project_name)
- **Team**(<u>team_id</u>, name, <u>project_id</u>)
  Chiave esterna: Team.project_id → Project.project_id
- **Issue**(<u>issue_id</u>, title, issue_description, issue_priority, issue_image, issue_type, issue_status, tags, report_time, resolution_time, <u>reporter_id</u>, <u>resolver_id</u>, <u>project_id</u>)
  Chiavi esterne: Issue.reporter_id → User.user_id, Issue.resolver_id → User.user_id, Issue.project_id → Project.project_id
- **Works_on**(<u>project_id</u>, <u>user_id</u>)
  Chiavi esterne: Works_on.project_id → Project.project_id, Works_on.user_id → User.user_id
- **Works_in**(<u>team_id</u>, <u>user_id</u>)
  Chiavi esterne: Works_in.team_id → Team.team_id, Works_in.user_id → User.user_id

Sulla base di quest'ultimo, è stato definito lo script SQL per il database, disponibile a questo link. In esso sono anche stati aggiunti alcuni trigger di utilità:

- Una issue è in relazione "assigned" (resolver_id) con uno user se e solo se type vale 1 o 2.
- Un Project è in relazione "works on" con uno user solo se type vale 1 o 2.
- Un Team è in relazione "works in" con uno user solo se type vale 1 o 2.
- Non si può modificare una issue segnata come 'resolved'.
- Non si possono modificare gli attributi "issue_id", "report_time" e "reporter_id" di una issue.
- Quando una issue viene segnata come "resolved", viene anche impostato il "resolution_time".
- Può essere segnata come "resolved" solo una issue che è stata assegnata (quindi con "resolver_id" not null).
- Quando viene aggiunto (per la precisione, DOPO aver aggiunto) un membro ad un team (nella tabella "Works_in"), si controlla se lo user appena aggiunto fa parte anche del progetto (nella tabella "Works_on"), in caso non sia così, lo si aggiunge.
- Quando viene cancellato un membro da un team (dalla tabella "Works_in"), si controlla se lo user è in altri team del progetto (sempre nella tabella "Works_in"), in caso non sia così, lo si toglie anche dal relativo progetto (dalla tabella "Works_on").

3.2 Classi di design

I diagrammi UML completi, per motivi di grandezza, sono consultabili rispettivamente ai seguenti link: Backend e Frontend. Lì è disponibile anche la JavaDoc per i relativi moduli. Di seguito, analizzeremo in dettaglio le due parti del sistema, procedendo per package.

3.2.1 Backend

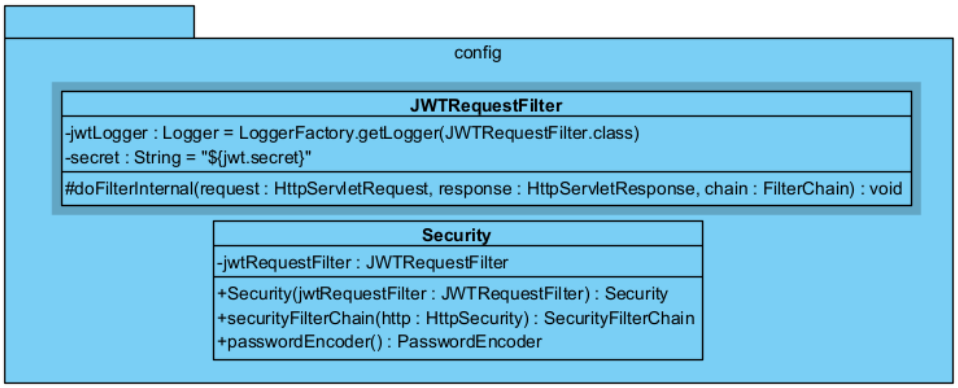

Figure 4: Package config del backend

Config

Il package config contiene le classi necessarie a gestire la sicurezza delle chiamate HTTP con le Rest, in particolare, il filtraggio per le richieste che richiedono JWT e la definizione dell'encoder per le password.

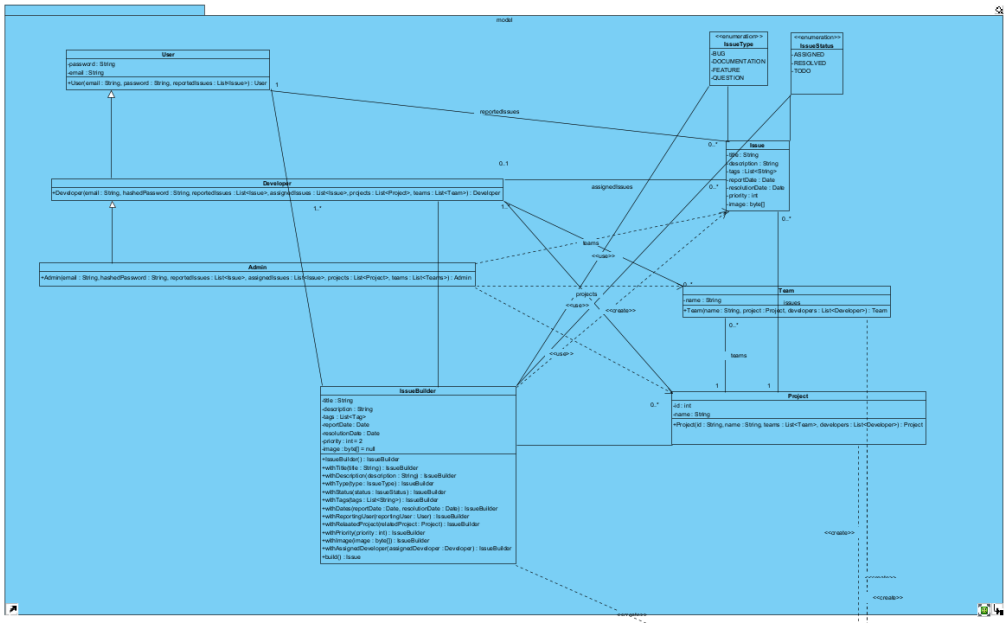

Figure 5: Package model del backend

Model

Il package model contiene le classi della logica di business del sistema, rispettandone i vincoli. In esso, segnaliamo in particolare l'utilizzo del design pattern "builder" per le issue. Questa classe ha molti attributi, alcuni dei quali opzionali, per questo il builder implementa innanzitutto l'opzione di mettere i parametri uno per volta, e poi, nel metodo "build", controlla che quelli obbligatori siano presenti prima di costruire l'oggetto effettivo.

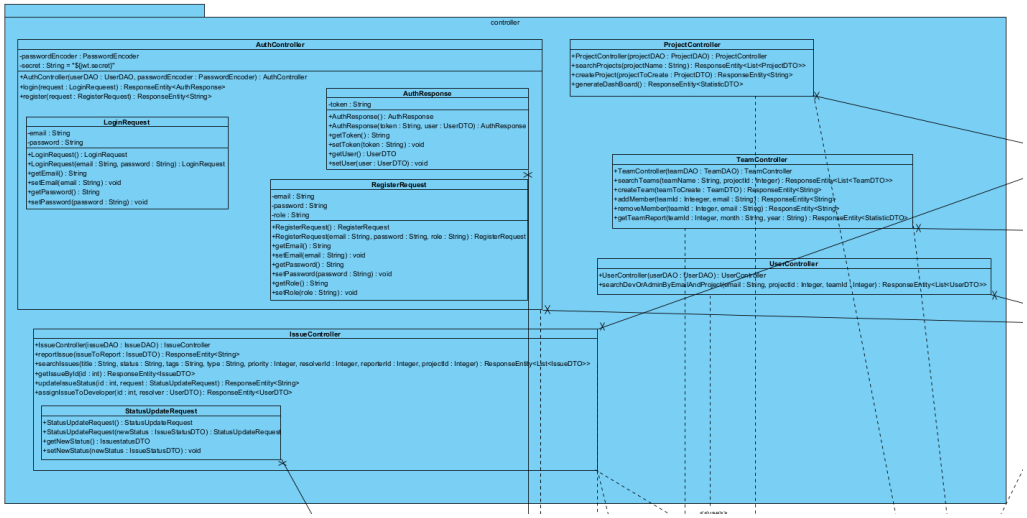

Figure 6: Package controller del backend

Controller

Il package controller contiene tutte le classi che si occupano dell'arrivo e della gestione delle chiamate con le Rest API. I path in queste classi sono le API esposte ai client, e, per ogni chiamata, la classe che la gestisce comunica con opportune classi del livello DAO per gestire i dati coinvolti e costruire la risposta per il frontend. Notare che, per generalità del codice, gli attributi dei DAO in queste classi sono interfacce, l'implementazione effettiva che si desidera utilizzare viene iniettata nel costruttore con Dependency Injection. Seppur l'utilizzo di interfacce introduce dell'accoppiamento, esso è accettato poichè aumenta la generalità del codice.

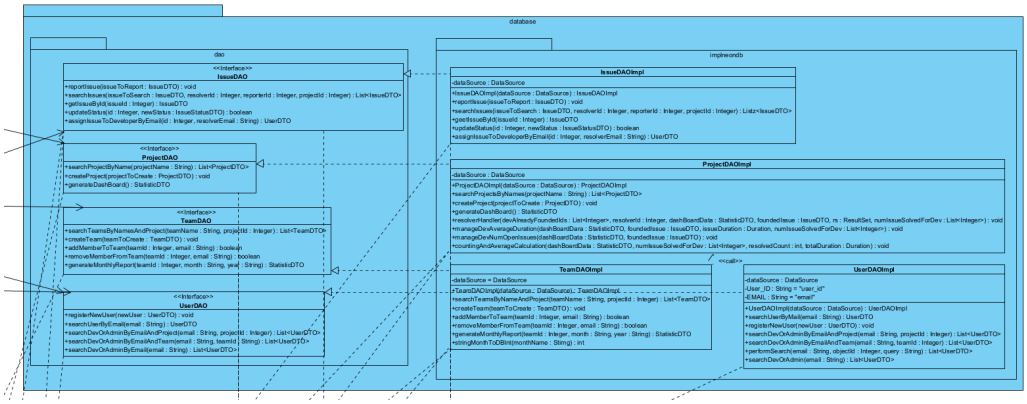

Figure 7: Package database del backend

Database

Il package database contiene al suo interno tutto ciò che è necessario per la comunicazione con il database NeonDB contentente i dati persistenti. È diviso a sua volta in due package:

- DAO: è un package di interfacce, in modo da generalizzare il più possbile il codice. Contiene le firme di tutti i metodi di comunicazione con il database che sono richiesti, ma essi possono essere implementati con un DB qualunque.
- Implementazione con NeonDB (impleneondb): è l'implementazione che abbiamo scelto per questo progetto nello specifico, ma può essere cambiata in qualunque momento senza cambiare nulla nel resto del codice.

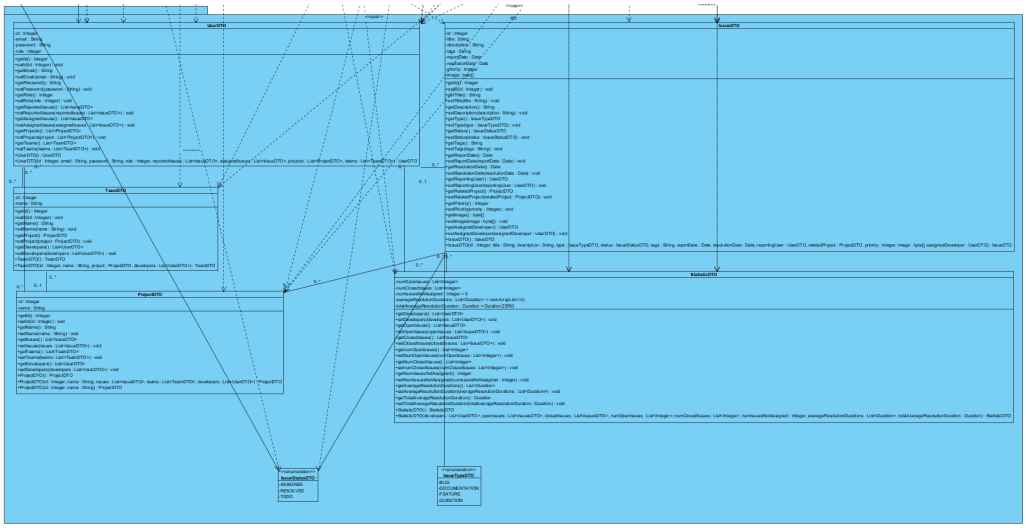

Figure 8: Package DTO del backend

Data transfer object

Il package DTO (Data transfer object) contiene tutte le classi utilizzate per la comunicazione tra diversi componenti del sistema. Essi consentono di avere un codice più snello e leggibile, riducendo di molto il numero di parametri dei metodi. Inoltre, sono molto flessibili, permettendo di costruire oggetti solamente con i parametri che servono davvero in quel momento, alleggerendo il carico di risorse necessario per l'applicazione.

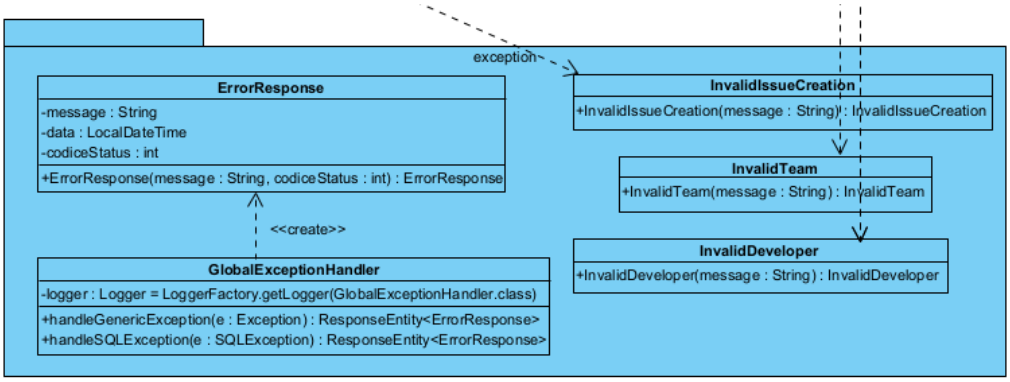

Figure 9: Package exception del backend

Exception

Il package exception contiene tutte le eccezioni personalizzate che l'aaplicazione può lanciare, ed inoltre, grazie al framework Spring Boot, contiene un gestore delle eccezioni globale ("Global Exception Handle"), il quale può rispondere in maniera personalizzata ad eccezioni scelte da noi sviluppatori, oppure in maniera generica per un eccezione non tra quelle precedenti. Il package contiene inoltre anche una classe "Error response", utilizzata dal Global Exception Handler per restituire una risposta di errore con il relativo codice HTTP in caso di problemi.

3.2.2 Frontend

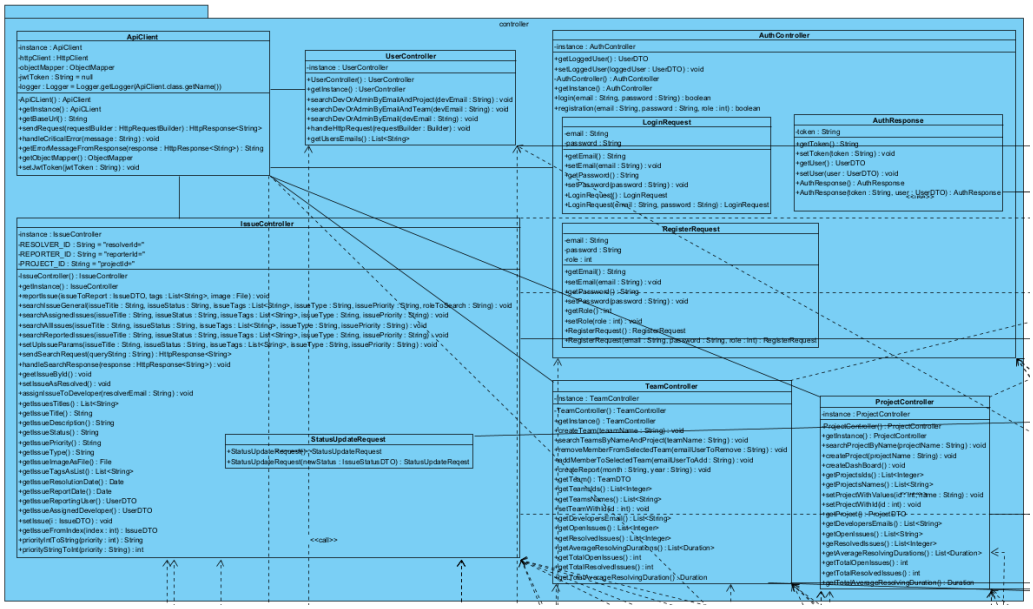

Figure 10: Package controller del frontend

Controller

Il package controller contiene tutte le classi che si occupano delle chiamate al server con le Rest API. Ogni classe si occupa di un ambito del sistema preciso, e comunica attraverso le API relative alla risorsa da essa gestita. Utilizzando i DTO, comunica con la GUI per fornirle i dati/ effettuare i controlli necessari. Ogni classe di questo package è implementata con il pattern singleton, dato che non avrebbe senso istanziarne più di una, dato che esse mantengono, attraverso una cache locale, i dati sulla sessione corrente (es. dati dell'utente loggato, risultati delle ricerche effettuate ecc.). Il mapping degli oggetti nei JSON per le chiamate HTTP è gestito attraverso l'object mapper della libreria "Jackson". In più, in questo package è presente la classe "ApiClient". Essa si occupa sia di mandare effettivamente le richieste al server, sia di gestire eventuali errori imprevisti (ad esempio, riporta l'utente al login se il suo JWT è scaduto oppure se è impossibile comunicare con il server).

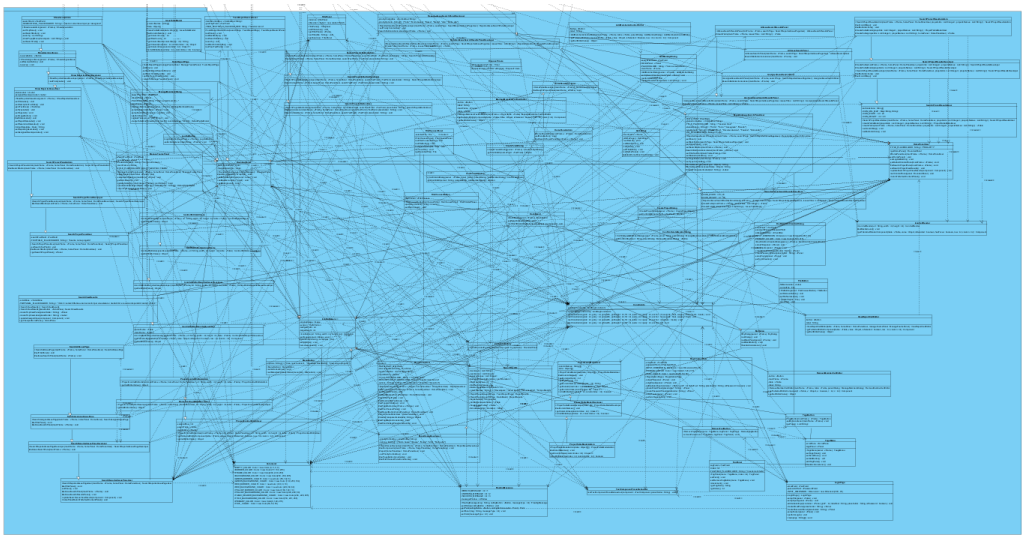

Figure 11: Package GUI del frontend

Graphical user interface

Il package GUI (Graphical user interface) contiene tutte le classi necessarie per l'interfaccia grafica dell'applicazione. Esse comunicano con il controller per richiedergli, sulla base delle azioni dell'utente, i dati necessari dal server. È realizzata con Java Swing, utilizzando il "Look & Feel" della libreria FlatLaf. Ciò che viene mostrato si adatta in base all'utente che sta utilizzando l'applicazione, mostrando solo ciò che è possibile fare in base al ruolo che ha (ad esempio, un guest non vedrà la pagina delle issue ad esso assegnate, che invece vedranno developer e admin).

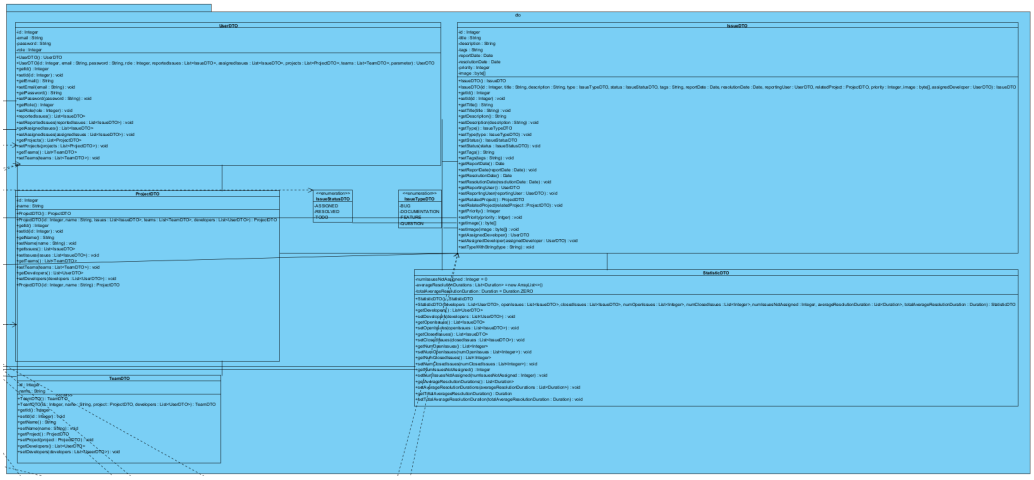

Figure 12: Package DTO del frontend

Data transfer object

Il package DTO (Data transfer object) contiene tutte le classi utilizzate per la comunicazione tra diversi componenti del sistema. Essi consentono di avere un codice più snello e leggibile, riducendo di molto il numero di parametri dei metodi. Inoltre, sono molto flessibili, permettendo di costruire oggetti solamente con i parametri che servono davvero in quel momento, alleggerendo il carico di risorse necessario per l'applicazione.

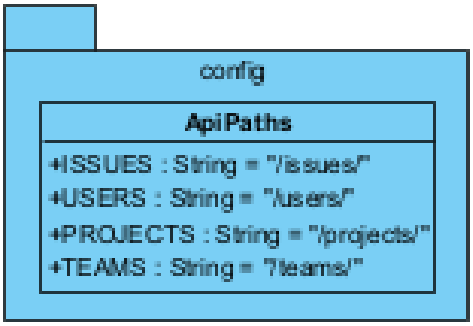

Figure 13: Package config del frontend

Config

Il package config contiene la classe con le costanti per i path basilari delle API, in modo che, in caso di modifiche, essi non debbano essere modificati in giro per l'intero programma.

Exception

Il package exception contiene tutte le eccezioni personalizzate che l'aaplicazione può lanciare.

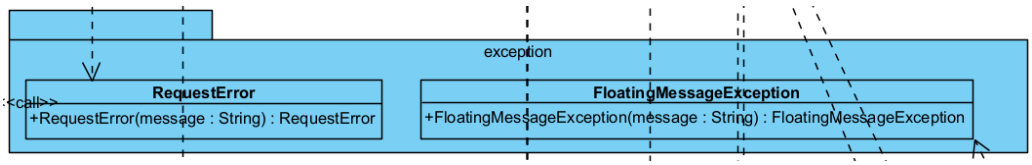

Figure 14: Package exception del frontend

3.2.3 Scelte di design

Come già analizzato, nello sviluppo del software sono stati utilizzati diversi design pattern: Builder (vedere model backend), Singleton (vedere controller frontend), e Dependency Injection (vedere controller backend).
Inoltre, per rispettare il SRP (Single Responsability Principle) ogni classe ha una e una sola responsabilità (per esempio, un controller per ogni entità, un dao per ogni entità del database ecc.), quindi la coesione è molto alta.
Inoltre, per generalizzare il codice e rendere semplici eventuali modifiche in futuro, quando possibile sono state create delle interfacce, implementate poi secondo le nostre scelte attuali, ma che possono essere facilmente cambiate con altre implementazioni in futuro senza cambiare nient'altro nel codice (vedere database backend). L'utilizzo di queste interfacce, seppur aggiunge dell'accoppiamento tra le classi coinvolte, porta molti più vantaggi sul lungo termine.
È stata poi rispettata la legge di Demetra, evitando sempre sequenze di chiamate ripetute su oggetti diversi e sparsi nel programma. I metodi sono stati tenuti il più semplici possibile, sia per lunghezza che per numero di parametri, in modo da rendere il codice più semplice sia da leggere che da comprendere.

3.2.4 Statistiche di GitHub

L'intero progetto è visionabile a questo link.

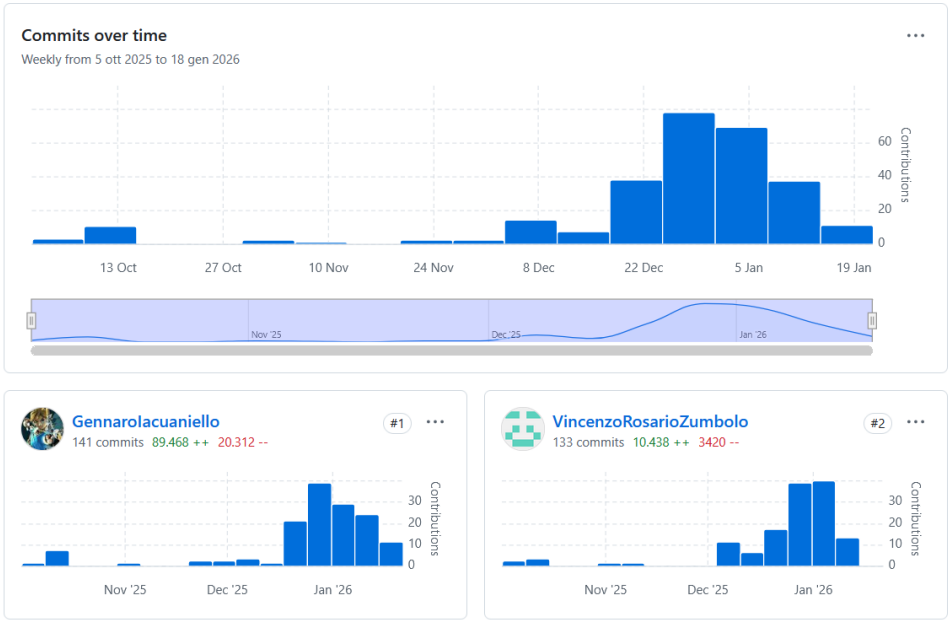

Figure 15: Statistiche di GitHub

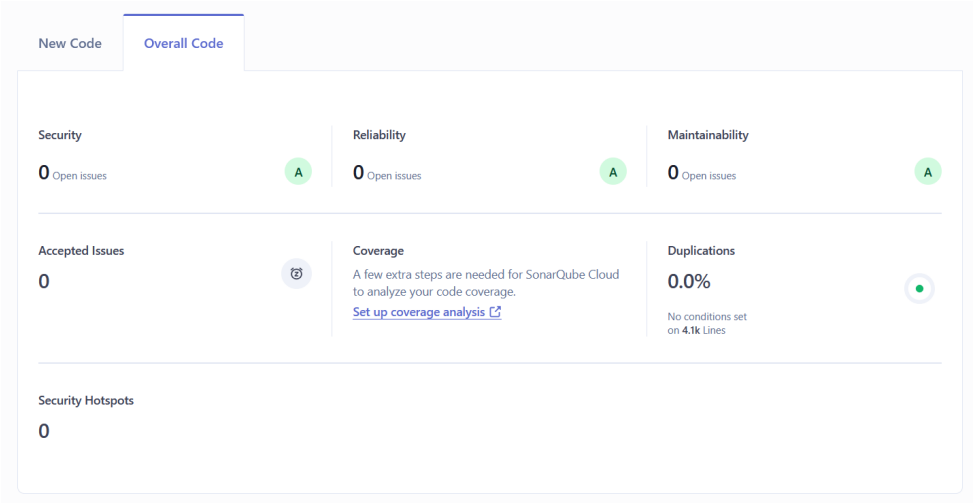

Figure 16: Statistiche di Sonar Cloud

4 Testing

4.1 Test plan per la funzionalità di assegnazione di una issue ad un developer/admin

Per testare completamente la funzionalità di assegnazione di una issue ad un developer/admin, bisogna partire dal testare il comportamento della relativa pagina della GUI.

Durante queste interazioni, vengono chiamati vari metodi dei controller, testati come segue:

ID	Precondizioni	Input	Output Atteso	Postcondizioni
1	Esiste almeno un utente admin, esiste almeno un progetto nel sistema, esiste almeno un team per quel progetto (e quindi c'è anche almeno un developer/admin che lavora su quel progetto), esiste almeno una issue nel sistema relativa a quel progetto e non assegnata a nessun utente.	Aprire l'applicazione, eseguire il login con un utente admin, effettuare una ricerca dei progetti, cliccare su "vedi tutte le issue" del progetto scelto nelle precondizioni, cercare tra le issue, cliccare su "vedi issue" per la issue scelta nelle precondizioni, inserire nella barra di ricerca la mail dell'utente scelto nelle precondizioni, cliccare su di esso nei risultati della ricerca.	La comparsa di un floating message verde di successo.	La issue risulta assegnata, nel database, all'utente scelto, e la schermata è stata aggiornata mostrando la issue come assegnata a quell'utente.

Table 2: Test Case ID 1: Assegnazione Issue GUI tutto ok

ID	Precondizioni	Input	Output Atteso	Postcondizioni
2	Esiste almeno un utente admin, esiste almeno un progetto nel sistema, esiste almeno un team per quel progetto (e quindi c'è anche almeno un developer/admin che lavora su quel progetto), esiste almeno una issue nel sistema relativa a quel progetto e non assegnata a nessun utente.	Aprire l'applicazione, eseguire il login con un utente admin, effettuare una ricerca dei progetti, cliccare su "vedi tutte le issue" del progetto scelto nelle precondizioni, cercare tra le issue, cliccare su "vedi issue" per la issue scelta nelle precondizioni, poi l'utente clicca indietro.	Si ritorna alla pagina di ricerca issue.	Non cambia nulla nel sistema.

Table 3: Test Case ID 2: L'utente annulla l'assegnazione prima del passaggio finale

ID	Precondizioni	Input	Output Atteso	Postcondizioni
3	Esiste almeno un utente admin, esiste almeno un progetto nel sistema, esiste almeno un team per quel progetto (e quindi c'è anche almeno un developer/admin che lavora su quel progetto), esiste almeno una issue nel sistema relativa a quel progetto e non assegnata a nessun utente.	Aprire l'applicazione, eseguire il login con un utente admin, effettuare una ricerca dei progetti, cliccare su "vedi tutte le issue" del progetto scelto nelle precondizioni, poi l'utente clicca indietro.	Si ritorna alla pagina di ricerca progetti.	Non cambia nulla nel sistema.

Table 4: Test Case ID 3: L'utente annulla l'assegnazione prima ancora di selezionare la issue

ID	Precondizioni	Input	Output Atteso	Postcondizioni
4	L'utente ha inserito almeno una lettera nella casella di ricerca dei progetti.	String projectName = la stringa inserita dall'utente nelle precondizioni.	True.	Viene riempita l'ArrayList di ProjectDTO projects in ProjectController con i risultati della ricerca, e la tabella nella pagina di ricerca dei progetti viene riempita con essi.

Table 5: Test Case ID 4: Test metodo searchProjectsByName di ProjectController del frontend con un parametro di ricerca

ID	Precondizioni	Input	Output Atteso	Postcondizioni
5	L'utente non ha inserito nulla nella casella di ricerca dei progetti.	String projectName = stringa vuota.	True.	Viene riempita l'ArrayList di ProjectDTO projects in ProjectController con tutti i progetti del sistema, e la tabella nella pagina di ricerca dei progetti viene riempita con essi.

Table 6: Test Case ID 5: Test metodo searchProjectsByName di ProjectController del frontend senza parametro di ricerca

ID	Precondizioni	Input	Output Atteso	Postcondizioni
6	L'utente ha inserito o meno almeno una lettera nella casella di ricerca dei progetti, ma c'è un errore nella connessione al server/-database.	String projectName = la stringa inserita dall'utente nelle precondizioni oppure stringa vuota.	False.	L'ArrayList di ProjectDTO projects in ProjectController diviene un'ArrayList vuota, e l'applicazione mostra un messaggio di errore e l'utente viene riportato alla pagina di login.

Table 7: Test Case ID 6: Test metodo searchProjectsByName di ProjectController del frontend con errore di connessione al server/database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
7	L'utente ha inserito almeno una lettera nella casella di ricerca delle issue, e/o uno stato dalla dropDownList, e/o dei tag, e/o un tipo dalla dropDownList, e/o una priorità dalla dropDownList.	`String issueTitle` = la stringa inserita dall'utente, `String issueStatus` = lo stato inserito, `List<String> issueTags` = la lista di tag inseriti, `String issueType` = il tipo inserito, `String issuePriority` = la priorità inserita.	True.	Viene riempita l'ArrayList di `IssueDTO issues` in `IssueController` con i risultati della ricerca, e la tabella nella pagina di ricerca delle issue viene riempita con esse.

Table 8: Test Case ID 7: Test metodo searchAllIssues di IssueController del frontend con i vari parametri di ricerca

ID	Precondizioni	Input	Output Atteso	Postcondizioni
8	L'utente non ha inserito alcuna lettera nella casella di ricerca delle issue, e non ha inserito nessun altro parametro.	`String issueTitle` = stringa vuota, `String issueStatus` = stringa vuota, `List<String> issueTags` = lista vuota, `String issueType` = stringa vuota, `String issuePriority` = stringa vuota.	True.	Viene riempita l'ArrayList di `IssueDTO issues` in `IssueController` con tutte le issue del progetto selezionato, e la tabella nella pagina di ricerca delle issue viene riempita con esse.

Table 9: Test Case ID 8: Test metodo searchAllIssues di IssueController del frontend con nessun parametro di ricerca

ID	Precondizioni	Input	Output Atteso	Postcondizioni
9	L'utente ha inserito o meno almeno una lettera nella casella di ricerca delle issue, e/o uno stato dalla dropDownList, e/o dei tag, e/o un tipo dalla dropDownList, e/o una priorità dalla dropDownList. C'è però un errore nella connessione al server/database.	`String issueTitle` = la stringa inserita oppure stringa vuota, `String issueStatus` = lo stato inserito oppure stringa vuota, `List<String> issueTags` = la lista di tag inseriti oppure lista vuota, `String issueType` = il tipo inserito oppure stringa vuota, `String issuePriority` = la priorità inserita oppure stringa vuota.	False.	L'ArrayList di `IssueDTO issues` in `IssueController` diviene una lista vuota, e l'applicazione mostra un messaggio di errore e l'utente viene riportato alla pagina di login.

Table 10: Test Case ID 9: Test metodo searchAllIssues di IssueController del frontend con errore di connessione al server/database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
10	L'utente clicca su "vedi issue" nella tabella di risultati ricerca issue.	Nulla.	True.	Il parametro `issue` di `IssueController` viene impostato alla issue selezionata dall'utente con tutti i dati presenti nel database, e l'applicazione mostra questi ultimi in una schermata.

Table 11: Test Case ID 10: Test metodo getIssueById di IssueController del frontend

ID	Precondizioni	Input	Output Atteso	Postcondizioni
11	L'utente clicca su "vedi issue" nella tabella di risultati ricerca issue, ma c'è un errore nella connessione al server/database.	Nulla.	False.	Il parametro `issue` di `IssueController` viene impostato a `null` e l'applicazione mostra un messaggio di errore e l'utente viene riportato alla pagina di login.

Table 12: Test Case ID 11: Test metodo getIssueById di IssueController del frontend con errore di connessione al server/database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
12	L'utente ha inserito almeno una lettera nella casella di ricerca delle mail dei developer/admin.	`String devEmail` = la stringa inserita dall'utente nelle precondizioni.	True.	Viene riempita l'ArrayList di `UserDTO users` in `UserController` con i risultati della ricerca sulla base degli utenti che lavorano sul progetto della issue selezionata, e i risultati della ricerca vengono mostrati in una lista di utenti tra cui è possibile scegliere.

Table 13: Test Case ID 12: Test metodo searchDevOrAdminByEmailAndProject di UserController del frontend con un parametro di ricerca

ID	Precondizioni	Input	Output Atteso	Postcondizioni
13	L'utente non ha inserito nulla nella casella di ricerca delle mail dei developer/admin.	`String devEmail` = stringa vuota.	True.	Viene riempita l'ArrayList di `UserDTO users` in `UserController` con tutti gli utenti che lavorano sul progetto della issue selezionata e questi vengono mostrati in una lista di utenti tra cui è possibile scegliere.

Table 14: Test Case ID 13: Test metodo searchDevOrAdminByEmailAndProject di UserController del frontend senza parametro di ricerca

ID	Precondizioni	Input	Output Atteso	Postcondizioni
14	L'utente ha inserito o meno almeno una lettera nella casella di ricerca delle mail dei developer/admin, ma c'è un errore nella connessione al server/-database.	`String devEmail` = la stringa inserita dall'utente nelle precondizioni oppure stringa vuota.	False.	L'ArrayList di `UserDTO users` in `UserController` diviene un'ArrayList vuota e l'applicazione mostra un messaggio di errore e l'utente viene riportato alla pagina di login.

Table 15: Test Case ID 14: Test metodo searchDevOrAdminByEmailAndProject di UserController del frontend con errore di connessione al server/database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
15	L'utente ha cliccato su un'email di un developer/admin che lavora al progetto della issue selezionata.	`String resolverEmail` = l'email su cui l'utente ha cliccato.	`True`.	L'attributo `assignedDeveloper` della issue selezionata viene settato all'utente desiderato e l'applicazione mostra che ora la issue è assegnata a quell'utente.

Table 16: Test Case ID 15: Test metodo assignIssueToDeveloper di IssueController del frontend

ID	Precondizioni	Input	Output Atteso	Postcondizioni
16	L'utente ha cliccato su un'email di un developer/admin che lavora al progetto della issue selezionata, ma c'è un errore di connessione al server/database.	`String resolverEmail` = l'email su cui l'utente ha cliccato.	`False`.	La issue resta come in partenza, l'applicazione mostra un messaggio di errore e l'utente viene riportato alla pagina di login.

Table 17: Test Case ID 16: Test metodo assignIssueToDeveloper di IssueController del frontend con errore di connessione al server/database

Ora si passa invece ai metodi del backend coinvolti:

ID	Precondizioni	Input	Output Atteso	Postcondizioni
17	Viene effettuata una chiamata al server con verbo GET e path /projects/search con parametro "name" una stringa con almeno una lettera.	String projectName = il parametro name presente nel path.	Viene restituita al client una lista dei ProjectDTO risultanti dalla ricerca, e la risposta ha status HTTP 200.	Per il server tutto resta invariato.

Table 18: Test Case ID 17: Test metodo searchProjects di ProjectController del backend con un parametro di ricerca non vuoto

ID	Precondizioni	Input	Output Atteso	Postcondizioni
18	Viene effettuata una chiamata al server con verbo GET e path /projects/search con parametro "name" una stringa che non è presente nel nome di nessun progetto del sistema.	String projectName = il parametro name presente nel path.	Viene restituita al client una lista di ProjectDTO vuota, e la risposta ha status HTTP 204.	Per il server tutto resta invariato.

Table 19: Test Case ID 18: Test metodo searchProjects di ProjectController del backend con parametro di ricerca non presente nel database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
19	Viene effettuata una chiamata al server con verbo GET e path /projects/search con parametro "name"qualunque, ma c'è un errore di connessione al database.	String projectName = il parametro name presente nel path.	Viene restituita al client una ErrorResponse con status HTTP 500.	Per il server tutto resta invariato.

Table 20: Test Case ID 19: Test metodo searchProjects di ProjectController del backend con errore di connessione al database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
20	Viene effettuata una chiamata al server con verbo GET e path /projects/search con parametri: title, status, tags, type, priority, resolverID, reporterId, projectId.	`String title` = param title, `String status` = param status, `String tags` = param tags, `String type` = param type, `String priority` = param priority, `Integer resolverId` = param resolverId, `Integer reporterId` = param reporterId, `Integer projectId` = param projectId.	Viene restituita al client una lista di IssueDTO risultanti dalla ricerca, e la risposta ha status HTTP 200.	Per il server tutto resta invariato.

Table 21: Test Case ID 20: Test metodo searchIssues di IssueController del backend con i vari parametri di ricerca

ID	Precondizioni	Input	Output Atteso	Postcondizioni
21	Viene effettuata una chiamata al server con verbo GET e path /projects/search con parametri: title, status, tags, type, priority, resolverID, reporterId, projectId, ma la combinazione specifica non è presente nel database.	`String title` = param title, `String status` = param status, `String tags` = param tags, `String type` = param type, `String priority` = param priority, `Integer resolverId` = param resolverId, `Integer reporterId` = param reporterId, `Integer projectId` = param projectId.	Viene restituita al client una lista di IssueDTO vuota, e la risposta ha status HTTP 200.	Per il server tutto resta invariato.

Table 22: Test Case ID 21: Test metodo searchIssues di IssueController del backend con i vari parametri di ricerca, ma la combinazione specifica non è presente nel database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
22	Viene effettuata una chiamata al server con verbo GET e path /projects/search con parametri: title, status, tags, type, priority, resolverID, reporterId, projectId, ma projectId è null.	`String title` = param title, `String status` = param status, `String tags` = param tags, `String type` = param type, `String priority` = param priority, `Integer resolverId` = param resolverId, `Integer reporterId` = param reporterId, `Integer projectId` = null.	Viene restituita al client una ErrorResponse con status HTTP 500, perchè projectId è un parametro obbligatorio.	Per il server tutto resta invariato.

Table 23: Test Case ID 22: Test metodo searchIssues di IssueController del backend con i vari parametri di ricerca, ma projectId è null

ID	Precondizioni	Input	Output Atteso	Postcondizioni
23	Viene effettuata una chiamata al server con verbo GET e path /projects/search con parametri: title, status, tags, type, priority, resolverID, reporterId, projectId, ma c'è un errore di connessione al database.	`String title` = param title, `String status` = param status, `String tags` = param tags, `String type` = param type, `String priority` = param priority, `Integer resolverId` = param resolverId, `Integer reporterId` = param reporterId, `Integer projectId` = param projectId.	Viene restituita al client una ErrorResponse con status HTTP 500.	Per il server tutto resta invariato.

Table 24: Test Case ID 23: Test metodo searchIssues di IssueController del backend con i vari parametri di ricerca, ma c'è un errore di connessione al database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
24	Viene effettuata una chiamata al server con verbo GET e path /issues/id, con un id valido.	`int id` = param id.	Viene restituita al client una IssueDTO con tutti i dati della issue con id dato, e la risposta ha status HTTP 200.	Per il server tutto resta invariato.

Table 25: Test Case ID 24: Test metodo getIssueById di IssueController del backend con id valido

ID	Precondizioni	Input	Output Atteso	Postcondizioni
25	Viene effettuata una chiamata al server con verbo GET e path /issues/id, ma con un id non valido.	`int id` = param id.	Viene restituita al client risposta HTTP con status 404.	Per il server tutto resta invariato.

Table 26: Test Case ID 25: Test metodo getIssueById di IssueController del backend con id non valido

ID	Precondizioni	Input	Output Atteso	Postcondizioni
26	Viene effettuata una chiamata al server con verbo GET e path /issues/id, ma c'è un errore di connessione al database.	`int id` = param id.	Viene restituita al client una ErrorResponse con status HTTP 500.	Per il server tutto resta invariato.

Table 27: Test Case ID 26: Test metodo getIssueById di IssueController del backend, ma c'è un errore di connessione al database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
27	Viene effettuata una chiamata al server con verbo GET e path /users/developers/search con parametri: email, projectId, teamId.	`String email` = param email, `Integer projectId` = param projectId, `Integer teamId` = param teamId.	Viene restituita al client una lista di UserDTO risultanti dalla ricerca, e la risposta ha status HTTP 200.	Per il server tutto resta invariato.

Table 28: Test Case ID 27: Test metodo searchDevOrAdminByEmailAndProject di UserController del backend con i vari parametri di ricerca

ID	Precondizioni	Input	Output Atteso	Postcondizioni
28	Viene effettuata una chiamata al server con verbo GET e path /users/developers/search con parametri: email, projectId, teamId, ma la combinazione specifica non è presente nel database.	`String email` = param email, `Integer projectId` = param projectId, `Integer teamId` = param teamId.	Viene restituita al client una lista di UserDTO vuota, e la risposta ha status HTTP 204.	Per il server tutto resta invariato.

Table 29: Test Case ID 28: Test metodo searchDevOrAdminByEmailAndProject di UserController del backend con i vari parametri di ricerca, ma la combinazione specifica non è presente nel database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
29	Viene effettuata una chiamata al server con verbo GET e path /users/developers/search con parametri: email, projectId, teamId, ma email è null.	`String email` = null, `Integer projectId` = param projectId, `Integer teamId` = param teamId.	Viene restituita al client una risposta HTTP con status 400.	Per il server tutto resta invariato.

Table 30: Test Case ID 29: Test metodo searchDevOrAdminByEmailAndProject di UserController del backend con i vari parametri di ricerca, ma email è null

ID	Precondizioni	Input	Output Atteso	Postcondizioni
30	Viene effettuata una chiamata al server con verbo GET e path /users/developers/search con parametri: email, projectId, teamId, ma c'è un errore di connessione al database.	`String email` = param email, `Integer projectId` = param projectId, `Integer teamId` = param teamId.	Viene restituita al client una ErrorResponse con status HTTP 500.	Per il server tutto resta invariato.

Table 31: Test Case ID 30: Test metodo searchDevOrAdminByEmailAndProject di UserController del backend con i vari parametri di ricerca, ma c'è un errore di connessione al database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
31	Viene effettuata una chiamata al server con verbo GET e path /issues/id/resolver con parametri: id, resolver.	`int id` = param id, `UserDTO resolver` = param resolver.	Viene restituito al client lo UserDTO a cui è stata assegnata la issue, e la risposta ha status HTTP 200.	Nel server, la issue risulta assegnata all'utente desiderato.

Table 32: Test Case ID 31: Test metodo assignIssueToDeveloper di UserController del backend con dati validi

ID	Precondizioni	Input	Output Atteso	Postcondizioni
32	Viene effettuata una chiamata al server con verbo GET e path /issues/id/resolver con parametri: id, resolver, ma l'email del risolver è null o vuota.	`int id` = param id, `UserDTO resolver` = null o stringa vuota.	Viene restituita al client una ErrorResponse con status HTTP 400.	Per il server tutto resta invariato.

Table 33: Test Case ID 32: Test metodo assignIssueToDeveloper di UserController del backend con email del risolver è null o vuota

ID	Precondizioni	Input	Output Atteso	Postcondizioni
33	Viene effettuata una chiamata al server con verbo GET e path /issues/id/resolver con parametri: id, resolver, ma l'email del non esiste nel database.	`int id` = param id, `UserDTO resolver` = param resolver.	Viene restituita al client una ErrorResponse con status HTTP 404.	Per il server tutto resta invariato.

Table 34: Test Case ID 33: Test metodo assignIssueToDeveloper di UserController del backend con email del risolver che non esiste nel database

ID	Precondizioni	Input	Output Atteso	Postcondizioni
33	Viene effettuata una chiamata al server con verbo GET e path /issues/id/resolver con parametri: id, resolver, ma c'è un errore di connessione al database.	`int id` = param id, `UserDTO resolver` = param resolver.	Viene restituita al client una ErrorResponse con status HTTP 500.	Per il server tutto resta invariato.

Table 35: Test Case ID 33: Test metodo assignIssueToDeveloper di UserController del backend, ma c'è un errore di connessione al database

4.2 Codice di test per i metodi updateIssueStatus e assignIssueToDeveloper di IssueController

Di seguito viene riportato il codice dei test di unità automatici dei metodi updateIssueStatus e assignIssueToDeveloper di IssueController.

```
@SpringBootTest
@AutoConfigureMockMvc // Configures MockMvc for HTTP requests
@WithMockUser(username = "admin", roles = {"ADMIN", "DEVELOPER"})
class IssueControllerTests {

    static {
        try {
            Dotenv dotenv = Dotenv.configure()
                    .directory("./")
                    .ignoreIfMissing()
                    .load();

            dotenv.entries().forEach(entry ->
                    System.setProperty(entry.getKey(), entry.getValue())
            );

        } catch (Exception e) {
            System.err.println(".env not found: " + e.getMessage());
        }
    }

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private IssueDAO issueDAO;

    @Autowired
    private ProjectDAO projectDAO;

    @Autowired
    private UserDAO userDAO;

    @Autowired
    private DataSource dataSource;

    private final ObjectMapper objectMapper = new ObjectMapper().
    findAndRegisterModules();

    private int testIssueId;

    private int testProjectId;

    private UserDTO adminForTesting;

    @BeforeEach
    void setup() throws Exception {

        IssueDTO testIssue = new IssueDTO();
        testIssue.setTitle("TestIssue_Test");
        testIssue.setDescription("Temporary issue for testing.");
        testIssue.setStatus(IssueStatusDTO.TODO);
```

```
52          testIssue.setType(IssueTypeDTO.BUG);
53          testIssue.setPriority(1);
54
55          testIssue.setReportDate(Date.from(Instant.now()));
56
57          adminForTesting = new UserDTO();
58          adminForTesting.setId(0);
59          adminForTesting.setEmail("admin@admin.admin");
60          testIssue.setReportingUser(adminForTesting);
61
62          ProjectDTO relatedProjectTest = new ProjectDTO();
63          relatedProjectTest.setName("RelatedProjectTest_Test");
64
65          projectDAO.createProject(relatedProjectTest);
66
67          List<ProjectDTO> projectJustCreatedForId = projectDAO.searchProjectsByName("
        RelatedProjectTest_Test");
68
69          relatedProjectTest.setId(projectJustCreatedForId.get(0).getId());
70
71          this.testProjectId = relatedProjectTest.getId();
72
73          testIssue.setRelatedProject(relatedProjectTest);
74
75          issueDAO.reportIssue(testIssue);
76
77          List<IssueDTO> issueJustCreatedForId = issueDAO.searchIssues(testIssue, null
        , 0, relatedProjectTest.getId());
78
79          if (!issueJustCreatedForId.isEmpty()) {
80              testIssue.setId(issueJustCreatedForId.get(0).getId());
81              this.testIssueId = testIssue.getId();
82              System.out.println("Test Issue created with ID: " + this.testIssueId);
83          } else {
84              throw new RuntimeException("Impossible found created test issue!");
85          }
86      }
87
88      @Test
89      void testUpdateStatusAssignedOk() throws Exception {
90
91          mockMvc.perform(put("/issues/" + testIssueId + "/resolver")
92                          .contentType(MediaType.APPLICATION_JSON)
93                          .content(objectMapper.writeValueAsString(adminForTesting)))
94                  .andExpect(status().isOk());
95
96          IssueController.StatusUpdateRequest request = new IssueController.
        StatusUpdateRequest(IssueStatusDTO.ASSIGNED);
97
98          mockMvc.perform(put("/issues/" + testIssueId + "/status")
99                          .contentType(MediaType.APPLICATION_JSON)
100                         .content(objectMapper.writeValueAsString(request)))
101                 .andExpect(status().isOk())
102                 .andExpect(content().string("Status update success"));
103
104     }
105
106     @Test
```

```
107     void testUpdateStatusResolvedOk() throws Exception {
108
109         mockMvc.perform(put("/issues/" + testIssueId + "/resolver")
110                         .contentType(MediaType.APPLICATION_JSON)
111                         .content(objectMapper.writeValueAsString(adminForTesting)))
112                 .andExpect(status().isOk());
113
114         IssueController.StatusUpdateRequest statusRequest = new IssueController.
    StatusUpdateRequest(IssueStatusDTO.RESOLVED);
115
116         mockMvc.perform(put("/issues/" + testIssueId + "/status")
117                         .contentType(MediaType.APPLICATION_JSON)
118                         .content(objectMapper.writeValueAsString(statusRequest)))
119                 .andExpect(status().isOk())
120                 .andExpect(content().string("Status update success"));
121
122     }
123
124     @Test
125     void testUpdateStatusResolvedWhenNotAssigned() throws Exception {
126
127         IssueController.StatusUpdateRequest statusRequest = new IssueController.
    StatusUpdateRequest(IssueStatusDTO.RESOLVED);
128
129         mockMvc.perform(put("/issues/" + testIssueId + "/status")
130                         .contentType(MediaType.APPLICATION_JSON)
131                         .content(objectMapper.writeValueAsString(statusRequest)))
132                 .andExpect(status().isInternalServerError())
133                 .andExpect(jsonPath("$.message").value("Database error"));
134     }
135
136     @Test
137     void testUpdateStatusWithNull() throws Exception {
138
139         IssueController.StatusUpdateRequest statusRequest = new IssueController.
    StatusUpdateRequest(null);
140
141         mockMvc.perform(put("/issues/" + testIssueId + "/status")
142                         .contentType(MediaType.APPLICATION_JSON)
143                         .content(objectMapper.writeValueAsString(statusRequest)))
144                 .andExpect(status().isBadRequest())
145                 .andExpect(content().string("Status error: missing"));
146     }
147
148     @Test
149     void testUpdateStatusOfAResolvedIssue() throws Exception {
150
151         mockMvc.perform(put("/issues/" + testIssueId + "/resolver")
152                         .contentType(MediaType.APPLICATION_JSON)
153                         .content(objectMapper.writeValueAsString(adminForTesting)))
154                 .andExpect(status().isOk());
155
156         IssueController.StatusUpdateRequest statusRequest = new IssueController.
    StatusUpdateRequest(IssueStatusDTO.RESOLVED);
157
158         mockMvc.perform(put("/issues/" + testIssueId + "/status")
159                         .contentType(MediaType.APPLICATION_JSON)
160                         .content(objectMapper.writeValueAsString(statusRequest)))
```

```
                .andExpect(status().isOk())
                .andExpect(content().string("Status update success"));

        statusRequest = new IssueController.StatusUpdateRequest(IssueStatusDTO.
    ASSIGNED);

        mockMvc.perform(put("/issues/" + testIssueId + "/status")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(statusRequest)))
                .andExpect(status().isInternalServerError())
                .andExpect(jsonPath("$.message").value("Database error"));
    }

    @Test
    void testUpdateStatusInvalidId() throws Exception {

        int invalidIssueId = -1;

        IssueController.StatusUpdateRequest request =
                new IssueController.StatusUpdateRequest(IssueStatusDTO.RESOLVED);

        mockMvc.perform(put("/issues/" + invalidIssueId + "/status")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isNotFound())
                .andExpect(content().string("Issue not found"));

    }

    @Test
    void testAssignDeveloperEmptyEmail() throws Exception {

        UserDTO invalidResolver = new UserDTO();
        invalidResolver.setEmail("");

        mockMvc.perform(put("/issues/" + testIssueId + "/resolver")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(invalidResolver)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.message").value("Assigning error: missing
    email"));

    }

    @Test
    void testAssignDeveloperNullEmail() throws Exception {

        UserDTO invalidResolver = new UserDTO();
        invalidResolver.setEmail(null);

        mockMvc.perform(put("/issues/" + testIssueId + "/resolver")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(invalidResolver)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.message").value("Assigning error: missing
    email"));

    }
```

```
216
217     @Test
218     void testAssignDeveloperNotExistingEmail() throws Exception {
219
220         UserDTO invalidResolver = new UserDTO();
221         invalidResolver.setEmail("a");
222
223         mockMvc.perform(put("/issues/" + testIssueId + "/resolver")
224                         .contentType(MediaType.APPLICATION_JSON)
225                         .content(objectMapper.writeValueAsString(invalidResolver)))
226                 .andExpect(status().isNotFound())
227                 .andExpect(jsonPath("$.message").value("User with specified email
    not found or issue does not exist"));
228
229     }
230
231     @Test
232     void testAssignDeveloperNotExistingIssue() throws Exception {
233
234         int invalidIssueId = -1;
235
236         mockMvc.perform(put("/issues/" + invalidIssueId + "/resolver")
237                         .contentType(MediaType.APPLICATION_JSON)
238                         .content(objectMapper.writeValueAsString(adminForTesting)))
239                 .andExpect(status().isNotFound())
240                 .andExpect(jsonPath("$.message").value("User with specified email
    not found or issue does not exist"));
241
242     }
243
244     @Test
245     void testAssignDeveloperOk() throws Exception {
246
247         UserDTO dev = new UserDTO();
248         dev.setEmail("admin@admin.admin");
249
250         mockMvc.perform(put("/issues/" + testIssueId + "/resolver")
251                         .contentType(MediaType.APPLICATION_JSON)
252                         .content(objectMapper.writeValueAsString(dev)))
253                 .andExpect(status().isOk());
254     }
255
256     @Test
257     void testAssignGuest() throws Exception{
258
259         UserDTO testGuest = new UserDTO();
260         testGuest.setEmail("email@prova.test_test");
261         testGuest.setPassword("test_test");
262         testGuest.setRole(0);
263
264         userDAO.registerNewUser(testGuest);
265
266         try {
267
268             mockMvc.perform(put("/issues/" + testIssueId + "/resolver")
269                             .contentType(MediaType.APPLICATION_JSON)
270                             .content(objectMapper.writeValueAsString(testGuest)))
271                     .andExpect(status().isInternalServerError())
```

```
                    .andExpect(jsonPath("$.message").value("Database error"));

        } finally {

            String query = "DELETE FROM User_ WHERE email = ?";

            try (Connection conn = dataSource.getConnection();
                 PreparedStatement ps = conn.prepareStatement(query)) {
                ps.setString(1, testGuest.getEmail());
                ps.executeUpdate();
                System.out.println("Deletion complete: Guest user eliminated.");
            }
        }

    }

    @AfterEach
    void tearDown() throws Exception {

        String query = "DELETE FROM Issue WHERE issue_id = ?";

        try (Connection conn = dataSource.getConnection();
             PreparedStatement ps = conn.prepareStatement(query)) {
            ps.setInt(1, testIssueId);
            ps.executeUpdate();
            System.out.println("Deletion complete: Issue " + testIssueId + "
    eliminated.");
        }

        query = "DELETE FROM Project WHERE project_id = ?";

        try (Connection conn = dataSource.getConnection();
             PreparedStatement ps = conn.prepareStatement(query)) {
            ps.setInt(1, testProjectId);
            ps.executeUpdate();
            System.out.println("Deletion complete: Project " + testProjectId + "
    eliminated.");
        }

    }

}
```

Listing 1: Codice di test: IssueControllerTests.java

4.3 Analisi e descrizione dei test effettuati

I test di unità precedentemente riportati sono stati effettuati utilizzando JUnit ed il tool "MockMVC" di Spring Boot, il quale ci ha permesso di creare un client mock che simuli le richieste al server e simuli anche di essere loggato come un utente amministratore. Detto ciò, i test sono stati strutturati seguendo una logica white-box, con lo scopo sia di raggiungere la massima copertura del codice dei metodi testati, sia di testare tutte le possibili classi di equivalenza dei dati (es stringhe qualunque, vuote e null).