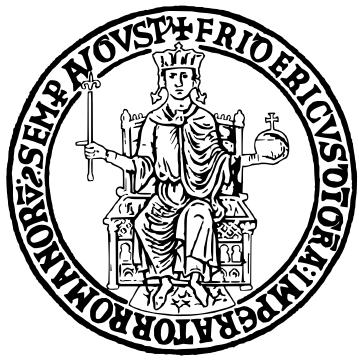


INGEGNERIA DEL SOFTWARE

BugBoard26

Documentazione Progetto



Davide Gargiulo **Francesco Donnarumma**
Matricola: N86004689 *Matricola: N86004658*

Anno Accademico 2024/2025
January 10, 2026

Contents

0 Introduzione	4
0.0.1 Link alla repository GitHub	4
1 Specifica dei Requisiti Software	5
1.1 Glossario	5
1.2 Modellazione di tutti i casi d'uso richiesti tramite Use Case Diagrams	6
1.3 Individuazione e caratterizzazione del target degli utenti, tramite Personas	7
1.3.1 Personas	7
1.3.2 Alexey Kutepov	7
1.3.3 Hideo Kojima	8
1.3.4 Michele Deserto	9
1.3.5 Yan Chernikov	10
1.4 Descrizione dei requisiti non-funzionali e di dominio	12
1.4.1 Requisiti non-funzionali	12
1.4.2 Requisiti di dominio	13
1.5 Dettagli dei casi d'uso specifici	14
1.5.1 Utente: Creazione di una nuova Issue	14
1.5.2 Amministratore: Aggiunta di un nuovo utente	18
1.5.3 Utente: Aggiunta di un commento a una Issue esistente	22
2 Documento di Design del sistema	25
2.1 Obiettivi di Design	25
2.1.1 Analisi delle Caratteristiche di Qualità	25
2.1.2 Tabella riassuntiva delle priorità delle qualità del software	28
2.1.3 Considerazioni sul Tempo di rilascio e Trade-offs	29
2.2 Decomposizione del sistema	30
2.2.1 La Scelta Architetturale: Un "Monolite Modulare con Servizi Esterni"	32
2.2.2 Strutturazione del core monolitico	32
2.2.3 Servizio Cross-Layer Esternalizzato	33

2.2.4	Comunicazione tra parti del sistema	33
2.3	Identificazione dei Threads	34
2.3.1	Parallelismo e Asincronismo	34
2.4	Descrizione e motivazione delle scelte tecnologiche adottate	35
2.4.1	Hardware-Software Mapping	35
2.4.2	Distribuzioni delle Componenti del Sistema	36
3	Documento di Design del Software, documentazione del processo di sviluppo, e artefatti software.	37
3.1	Descrizione dello schema per la persistenza dati (Database)	37
3.1.1	Il Ruolo del Database nel Sistema	38
3.1.2	PostgreSQL come Motore di Persistenza Relazionale	39
3.1.3	Sistema di Tipi e Valutazione: Type Safety nel Database	39
3.2	Evoluzione e Manutenibilità del Sistema	40
3.2.1	Architettura Evolutiva e Modularità	40
3.2.2	Osservabilità e Diagnostica	40
3.2.3	Strategie per scalabilità futura	41
3.3	Descrizione della logica di base (Backend)	42
3.3.1	Gestione delle Dipendenze	42
3.3.2	Gestione della Concorrenza e Integrità Transazionale	43
3.4	Logica di Implementazione del Backend	44
3.4.1	L'adozione di Express.js e la Middleware Pipeline	44
3.4.2	Organizzazione e Routing delle API	45
3.4.3	Gestione Avanzata delle Autorizzazioni e RBAC	45
3.4.4	Implementazione delle Regole di Business	46
3.5	Modello dei Dati Backend	47
3.6	Deployment e Configurazione	48
3.6.1	Configurazione	48
3.6.2	Containerizzazione	48
3.7	Descrizione e motivazione delle scelte di design dell'interfaccia utente adottate (Frontend)	49
3.7.1	Introduzione	49
3.7.2	Architettura e Pattern Fondamentali	49
3.7.3	Organizzazione delle Directory	51

3.8	Evidenza dell'uso di strumenti di versioning	52
3.8.1	Sistema di Versioning e Workflow	52
3.8.2	Continuous Integration e Qualità del Codice	52
3.8.3	Report Statistici	52
3.9	Qualità del Codice	56
3.9.1	Documentazione	56
4	Attività di Testing	57
4.1	Test Plan: Creazione di una Nuova Issue	57
4.1.1	Scopo e Obiettivi	57
4.1.2	Oggetto del Test (Test Items)	57
4.1.3	Strategia di Testing	57
4.1.4	Criteri di Accettazione	57
4.1.5	Casi di Test (Test Cases)	58
4.2	Codice per test di unità automatici	59
4.2.1	CreateIssue.test.ts	59
4.2.2	createComment.test.js	65
4.2.3	jwt.test.js	71
4.2.4	updateIssue.test.js	75
4.3	Documentazione delle strategie di test	83
4.3.1	Strategia per la Creazione Issue (createIssue)	83
4.3.2	Strategia per la Creazione Commento (createComment)	83
4.3.3	Strategia per l'Aggiornamento Issue (updateIssue)	84
4.3.4	Strategia per l'Autenticazione (Middleware JWT)	84

Introduzione

Documentazione del progetto di Ingegneria del Software, BugBoard26.

Università Degli Studi Di Napoli Federico II

Docenti: Sergio Di Martino, Luigi Lucio Libero Starace

0.0.1 | Link alla repository GitHub

BugBoard26-SWE

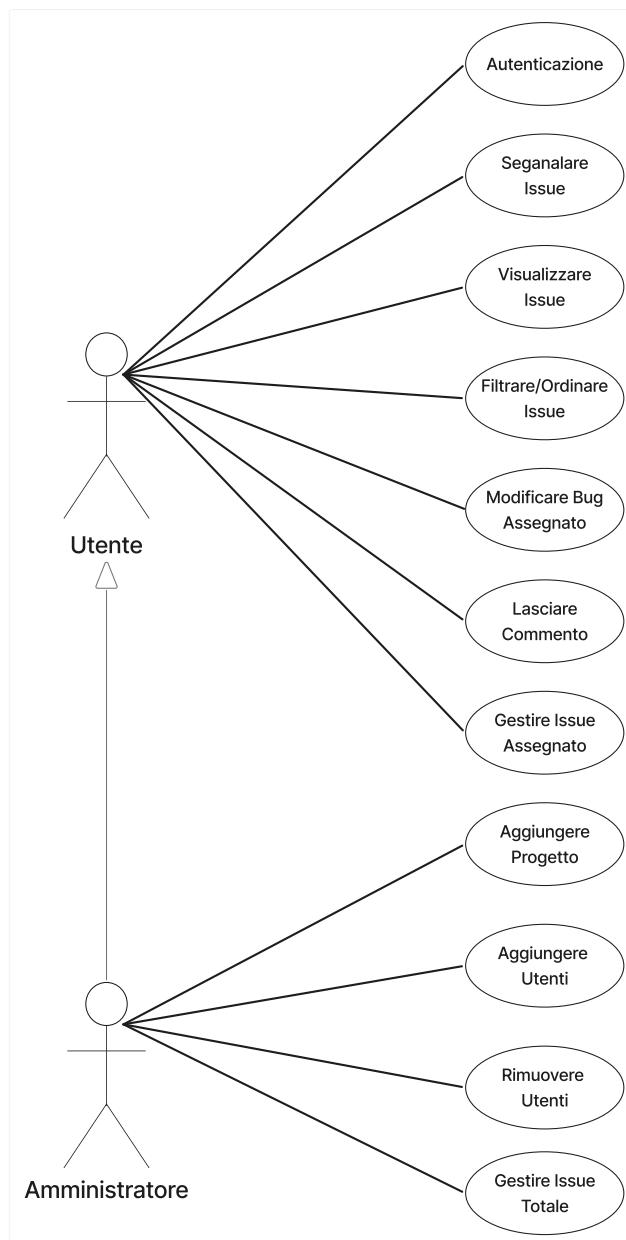
Specifica dei Requisiti Software

1.1

Glossario

- **Amministratore**
Ruolo utente con privilegi completi nel sistema, può gestire progetti, utenti e tutte le issue indipendentemente dall'assegnazione.
- **Bug, Feature, Question, Documentation**
Tipologie di issue che possono esistere nel sistema.
- **Commento**
Testo aggiunto da un utente a una issue esistente per fornire aggiornamenti o discussioni.
- **Dashboard**
Pagina principale dell'applicazione che mostra statistiche e panoramica delle issue.
- **Issue**
Entità centrale del sistema che rappresenta una segnalazione (bug, feature, question, documentation).
- **Priorità**
Attributo di una issue che indica l'urgenza (Alta, Media, Bassa).
- **Progetto**
Contenitore logico che raggruppa issue correlate.
- **Sistema**
Insieme degli artefatti software impiegati nella fornitura dei servizi richiesti dal committente.
- **Stato**
Attributo di una issue che indica la fase del ciclo di vita (TODO, In-Progress, Done).
- **Use Case (UC)**
Tipo di interazione offerto dal sistema che porta un vantaggio a un utente esterno
- **Utente Standard**
Ruolo utente con permessi limitati, può agire solo sulle issue a lui assegnate.

Modellazione di tutti i casi d'uso richiesti tramite Use Case Diagrams

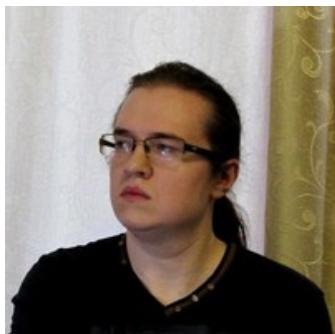


1.3

Individuazione e caratterizzazione del target degli utenti, tramite Personas

1.3.1 | Personas

Alexey Kuteпов (Tsoding)



Età: 35 anni, nato in Russia, attualmente nomade digitale

Residenza: Nomade digitale - Attualmente in Europa dell'Est

Istruzione: Laurea in Computer Science conseguita in Russia nel 2012, con focus su sistemi operativi e programmazione a basso livello

Lavoro: Software Engineer e Content Creator dal 2016. Streamer di programmazione su Twitch e YouTube, specializzato in C, Rust, e sviluppo di linguaggi di programmazione. Creatore di progetti open source e tool di sviluppo. Conosciuto per il suo approccio diretto e senza filtri alla programmazione

Tratti personali: Pragmatico e diretto, con approccio no-nonsense al codice. Perfezionista tecnico che valorizza semplicità ed efficienza. Ironico e sarcastico, ma genuinamente appassionato di condivisione della conoscenza tecnica

Interessi: Appassionato di matematica e teoria dei linguaggi di programmazione. Interessato a minimalismo nel design software e filosofia Unix. Nel tempo libero esplora nuove tecnologie, contribuisce a progetti open source e sperimenta con hardware vintage

Biografia

Alexey Kuteпов, 35 anni, conosciuto online come Tsoding, è un software engineer e content creator nomade specializzato in programmazione di sistema e sviluppo di linguaggi. Dal 2016 condivide il suo processo creativo attraverso live coding su Twitch, dove programma in C e Rust senza script né editing.

Con un approccio pragmatico e diretto, Tsoding è diventato una figura di riferimento per developer che apprezzano la programmazione a basso livello e la filosofia del codice minimalista ed efficiente.

Scopi e obiettivi in riferimento al sistema

- **Tracciare bug nei progetti open source personali:** Gestire efficacemente gli issue dei suoi numerosi repository GitHub, mantenendo organizzazione chiara per la community di contributor.
- **Riprodurre e documentare bug complessi:** Creare report tecnici dettagliati con stack trace, memory dumps e condizioni di riproduzione per problemi di basso livello.
- **Prioritizzare fix in base a impatto tecnico:** Categorizzare issue distinguendo tra crash critici, memory leaks, problemi di performance e feature requests.
- **Integrare bug tracking nel workflow di live coding:** Utilizzare l'applicazione durante le sessioni di streaming per mostrare alla community come affrontare sistematicamente i problemi tecnici.

Hideo Kojima



Età: 61 anni, nato a Tokyo

Residenza: Tokyo, Giappone - Quartiere Shibuya

Istruzione: Laurea in Economia conseguita presso l'Università di Tokyo nel 1986, con studi complementari in cinema e letteratura

Lavoro: Game Director, Producer e Autore dal 1986. Fondatore di Kojima Productions nel 2015. Creatore di franchise iconici come Metal Gear e Death Stranding. Visionario dell'industria videoludica con approccio cinematografico

Tratti personali: Visionario e meticoloso, con attenzione maniacale ai dettagli narrativi e tecnici. Perfezionista, esigente ma collaborativo. Comunicatore carismatico con forte presenza mediatica

Interessi: Cinefilo appassionato che colleziona film e oggetti da collezione. Lettore vorace di letteratura e manga. Interessato a tecnologia, intelligenza artificiale e futuro dell'intrattenimento digitale. Attivo sui social media dove condivide le sue passioni quotidiane

Biografia

Hideo Kojima, 61 anni, è uno dei game director più influenti e riconosciuti a livello mondiale. Con una formazione in economia e una passione viscerale per il cinema, ha rivoluzionato l'industria videoludica dal 1986, creando opere che fondono gameplay innovativo e narrativa cinematografica.

Fondatore di Kojima Productions, è celebre per la sua attenzione maniacale ai dettagli e per la sua visione autoriale che ha ridefinito i confini tra videogiochi, cinema e arte interattiva.

Scopi e obiettivi in riferimento al sistema

- **Mantenere la visione artistica del progetto:** Assicurarsi che ogni bug risolto non comprometta l'esperienza narrativa e l'immersione cinematografica prevista per il gioco.
- **Coordinare team multidisciplinari internazionali:** Utilizzare l'applicazione per sincronizzare il lavoro tra programmati, designer, artisti e tester distribuiti in diverse sedi globali.
- **Prioritizzare issue che impattano l'esperienza emotiva:** Categorizzare i bug in base al loro effetto sulla narrazione, l'atmosfera e il coinvolgimento emotivo del giocatore.
- **Documentare dettagliatamente ogni problema:** Creare report completi con context narrativo, screenshot, video e note precise per guidare il team verso soluzioni che rispettino la visione creativa.
- **Monitorare milestone critiche pre-release:** Tenere traccia dello stato di risoluzione degli issue in vista di demo, eventi stampa e lancio finale, garantendo standard qualitativi eccellenti.

Michele Deserto



Età: 35 anni, nato a Bari, residente a Milano da 8 anni

Residenza: Milano, Italia - Zona Porta Nuova

Istruzione: Diploma in Architettura Moderna conseguito a Bari nel 2009, con specializzazione in design sostenibile

Lavoro: Content Creator dal 2015, specializzato in recensioni videoludiche e tecnologia su YouTube e Twitch. Appassionato di giochi indie, RPG e hardware. Assunto da poco come bug tester dalla FromSoftware

Tratti personali: Flemmatico e riflessivo, con approccio creativo ai problemi. Calmo, empatico e buon ascoltatore con la sua community

Interessi: Oltre ai contenuti digitali, ama viaggiare scoprendo nuove culture e tradizioni. Appassionato di architettura e fotografia urbana. Nel tempo libero cucina e suona la chitarra

Biografia

Michele Deserto, 35 anni, è un content creator milanese specializzato in videogiochi e tecnologia. Con un diploma in architettura moderna e un approccio flemmatico e creativo, dal 2015 produce contenuti su YouTube e Twitch dedicati a giochi indie, RPG e innovazioni tech.

Appassionato viaggiatore, ama scoprire nuove culture e tradizioni, che spesso ispirano i suoi contenuti digitali.

Scopi e obiettivi in riferimento al sistema

- **Diminuire la presenza di bug nei giochi recensiti:** Garantire che i giochi presentino meno bug tecnici, migliorando l'esperienza di gioco per i suoi spettatori.
- **Tracciare efficacemente i bug critici:** Utilizzare l'applicazione per categorizzare e prioritizzare gli issue in base alla loro gravità e impatto sul gameplay.
- **Collaborare con il team di sviluppo:** Condividere report dettagliati e screenshot attraverso l'applicazione per facilitare la comunicazione con i programmati.
- **Monitorare lo stato di risoluzione:** Tenere traccia dei bug segnalati e verificare quali vengono risolti nelle patch successive.
- **Organizzare i test per piattaforme diverse:** Gestire gli issue separando i bug per console, PC e altre piattaforme su cui vengono testati i giochi.

Yan Chernikov (TheCherno)



Età: 29 anni, nato in Australia, residente negli Stati Uniti

Residenza: Los Angeles, California - USA

Istruzione: Laurea in Computer Science conseguita in Australia nel 2017, con specializzazione in computer graphics e game engine development

Lavoro: Software Engineer e Content Creator dal 2012. Creatore della popolare serie YouTube su C++ e game engine development. Lead Engine Programmer con esperienza in AAA studios. Fondatore di Hazel Engine, un game engine educativo open source

Tratti personali: Metodico e didattico, con approccio sistematico all'insegnamento della programmazione. Paziente e chiaro nelle spiegazioni, ma esigente sulla qualità del codice. Entusiasta e motivante con la sua community

Interessi: Appassionato di architettura software e design patterns. Interessato a rendering graphics, fisica e ottimizzazione performance. Nel tempo libero sperimenta con nuove tecnologie grafiche, contribuisce alla community open source e gioca a giochi competitivi

Biografia

Yan Chernikov, 29 anni, conosciuto come TheCherno, è un software engineer e educator specializzato in C++ e game engine development. Dal 2012 produce contenuti educativi di alta qualità su YouTube, rendendo accessibili concetti complessi di programmazione grafica e architettura di game engine.

Con esperienza in studi AAA e una passione genuina per l'insegnamento, TheCherno ha costruito una delle community più rispettate per developer che vogliono comprendere profondamente come funzionano i motori grafici moderni.

Scopi e obiettivi in riferimento al sistema

- **Gestire issue del progetto Hazel Engine:** Organizzare e prioritizzare bug report e feature requests della community per il suo game engine educativo open source.
- **Documentare problemi di rendering e graphics:** Creare report dettagliati per bug complessi relativi a shader, pipeline grafiche e ottimizzazioni rendering con screenshot e frame analysis.
- **Coordinare sviluppo tra tutorial series:** Utilizzare l'applicazione per pianificare quali issue affrontare durante le prossime puntate della serie YouTube, mantenendo coerenza didattica.
- **Categorizzare per subsystem dell'engine:** Organizzare issue per moduli specifici (renderer, physics, ECS, scripting) facilitando la navigazione per contributor e studenti.
- **Tracciare problemi cross-platform:** Monitorare bug specifici per Windows, macOS e Linux, assicurando che Hazel Engine funzioni correttamente su tutte le piattaforme.

Descrizione dei requisiti non-funzionali e di dominio

1.4.1 | Requisiti non-funzionali

Codice	Requisito
SR01	Il Sistema deve essere realizzato in modo distribuito prevedendo almeno due macro-componenti indipendenti (back-end/front-end).
SR02	Il back-end deve esporre interfacce di programmazione accessibili via rete.
SR03	Il back-end dovrebbe essere distribuito utilizzando tecnologie di containerizzazione.
SR04	Il front-end deve essere un'interfaccia utente che si appoggia ai servizi offerti dal back-end esclusivamente attraverso la rete.
SR05	La parte front-end deve essere realizzata come applicazione web app spa.
SR06	La logica applicativa e la persistenza dei dati non devono essere gestite esclusivamente tramite servizi esterni.
SR07	Il Sistema deve essere scritto in un linguaggio di programmazione che supporta il paradigma orientato agli oggetti.
SR08	Il Sistema deve essere conforme al GDPR per la protezione dei dati personali degli utenti.
SR09	Nel caso in cui il Sistema venga distribuito attraverso store di terze parti, deve essere conforme alle policy sui contenuti per sviluppatori di questi ultimi.

1.4.2 | Requisiti di dominio

Codice	Requisito
DR01	Una Issue deve necessariamente contenere un titolo e una descrizione, un tipo e una priorità. Opzionalmente almeno un allegato.
DR02	Un utente normale può agire solo su Issue a lui assegnate, mentre un admin può agire su tutte le Issue disponibili.
DR03	Ogni Issue ha un ciclo di vita definito da stati e transizioni irreversibili. to-do → in progress → done.
DR04	Le password degli utenti devono essere memorizzate in formato hash utilizzando algoritmi crittografici sicuri.
DR05	La priorità di un'Issue deve assumere valori da un insieme predefinito.
DR06	Solo gli amministratori possono creare nuovi account utente nel sistema.
DR06	Un'Issue può essere assegnata a un solo utente alla volta.
DR07	L'account amministratore di default deve essere configurato con credenziali sicure.

1.5

Dettagli dei casi d'uso specifici

1.5.1 Utente: Creazione di una nuova Issue

Mockup

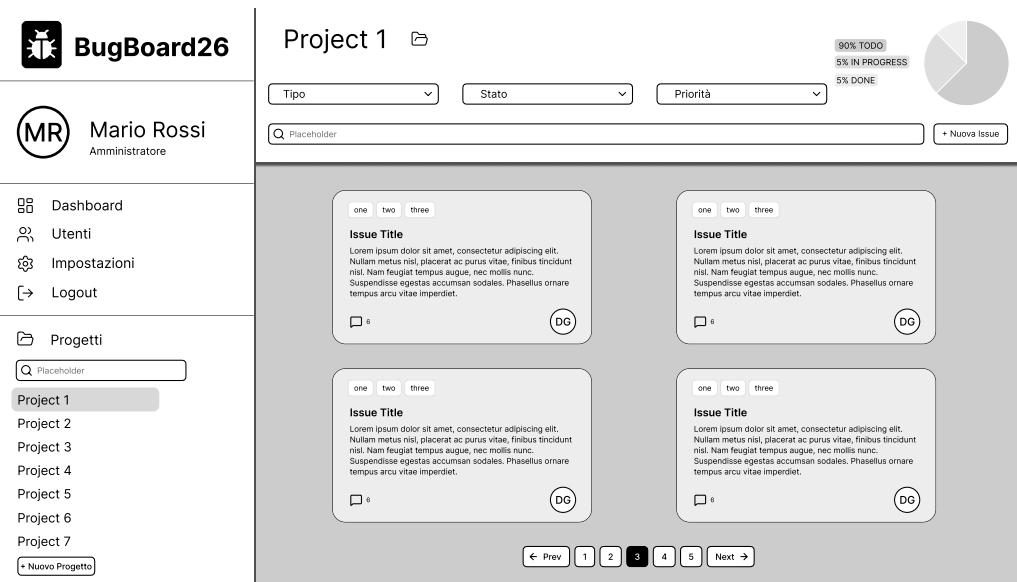


Figure 1.1: UC02_MC01

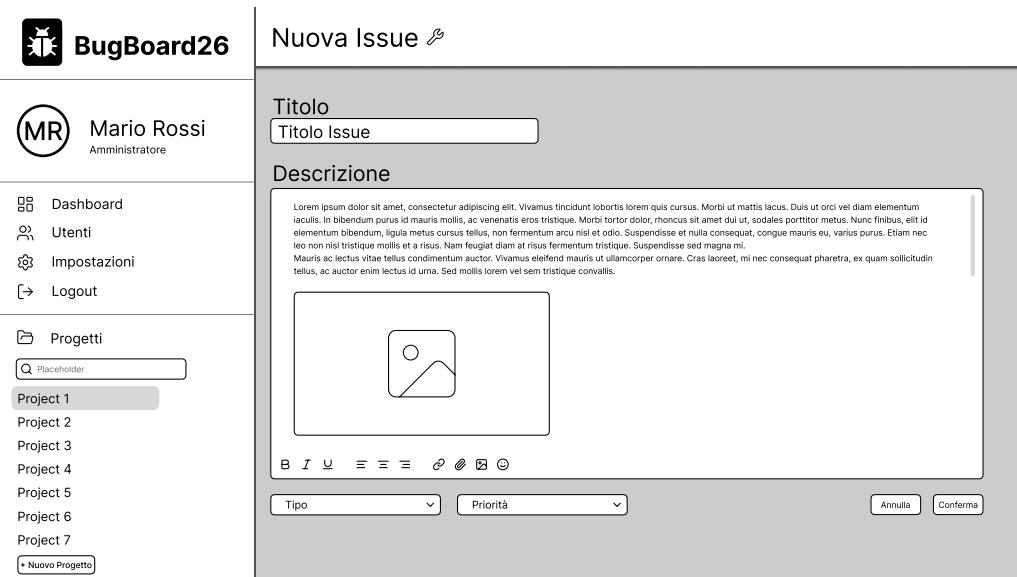


Figure 1.2: UC02_MC02

Figure 1.3: UC02_MC03



BugBoard26



Mario Rossi
Amministratore

-  Dashboard
-  Utenti
-  Impostazioni
-  Logout

-  Progetti

- Project 1
- Project 2
- Project 3
- Project 4
- Project 5
- Project 6
- Project 7

[+ Nuovo Progetto](#)

Nuova Issue

Titolo

Descrizione

... ipsum dolor sit amet, consectetur adipiscing elit. Vivamus tincidunt lobortis lorem quis cursus. Morbi ut mattis lacus. Duis ut orci vel diam elementum ...
... scelerisque. In tincidunt, venenatis id mauris mollis, ac venenatis eros tristique. Morbi tortor dolor, euismod sit amet dui ut, sodales porttitor metus. Nunc finibus, elit id ...
... elementum bibendum, ligula metus cursus tellus, non fermentum arcu nisi et odio. Suspendisse et nulla consequat, congue mauris eu, varius purus. Etiam nec ...
... leo non nisi tristique. Mauris ac lectus vel tellus, ac auctor enim. Sed ...
... sed magna mi.
... ornare. Cras laoreet, mi nec consequat pharetra, ex quam sollicitudin ...

Errore! 

Aimeno uno tra Titolo, Descrizione o Tipo non è compilato

B I U E = ⌂ ⌂ ⌂ ⌂ ⌂ ⌂

Tipo Priorità

Figure 1.4: UC02 MC04

Use Case #02			Crea Nuova Issue
<i>Scopo</i>			Un utente vuole creare una nuova Issue.
<i>Precondizioni</i>			L'utente è autenticato nel sistema.
<i>Condizione finale di successo</i>			La nuova Issue è stata creata con successo nel sistema.
<i>Condizione finale di insuccesso</i>			La creazione della nuova Issue non è riuscita a causa di un errore del sistema o di dati non validi forniti dall'utente.
<i>Attore Principale</i>			Utente generale
<i>Trigger</i>			L'utente clicca sul bottone "Nuova Issue" in UC02_MC01.
<i>Main Scenario</i>	Step n.	Utente generale	Sistema
	01	L'utente clicca sul bottone "Nuova Issue" in UC02_MC01	
	02		Il sistema mostra UC02_MC02
	03	L'utente compila correttamente il form e clicca "Conferma"	
	04		Mostra UC02_MC03
	05	L'utente clicca sul bottone "Conferma"	
	06		Mostra UC02_MC01 e termina UC
<i>Extension #01</i>	Step n.	Utente generale	Sistema
	3.01	L'utente non compila titolo o descrizione e clicca "Conferma"	
	4.01		Mostra UC02_MC04
	5.01	L'utente clicca sul bottone "Conferma"	
	6.01		Riparte dal punto 2
<i>Extension #02</i>	Step n.	Utente generale	Sistema
	3.02	L'utente clicca sul bottone "Annulla"	
	4.02		Mostra UC02_MC01 e termina UC

Codice	Requisito
<i>UR02_01</i>	Il sistema deve permettere il caricamento di immagini o documenti pdf. Per un massimo di tre documenti, ciascuno con dimensione massima 5MB.

1.5.2 | Amministratore: Aggiunta di un nuovo utente

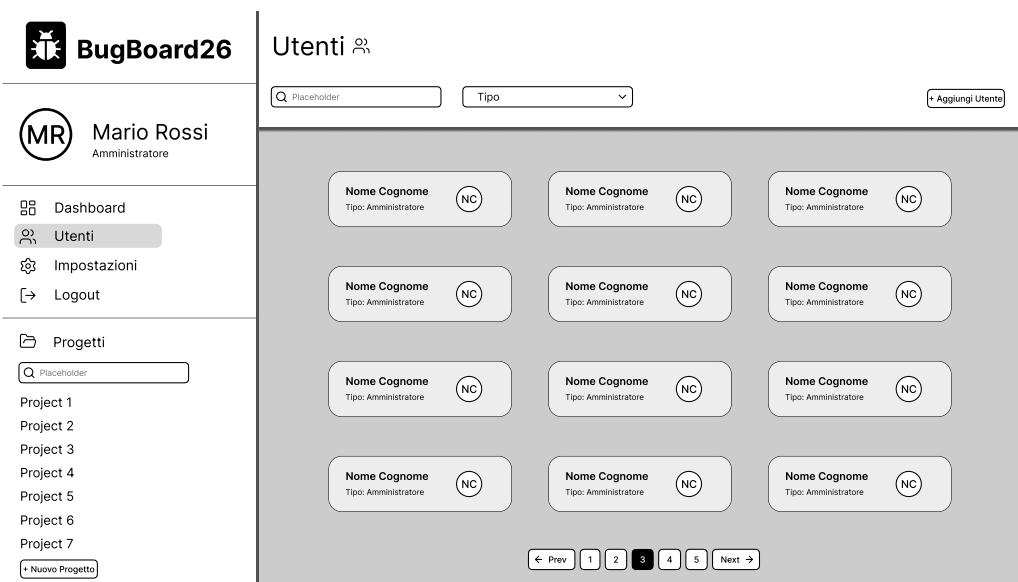


Figure 1.5: UC08_MC01

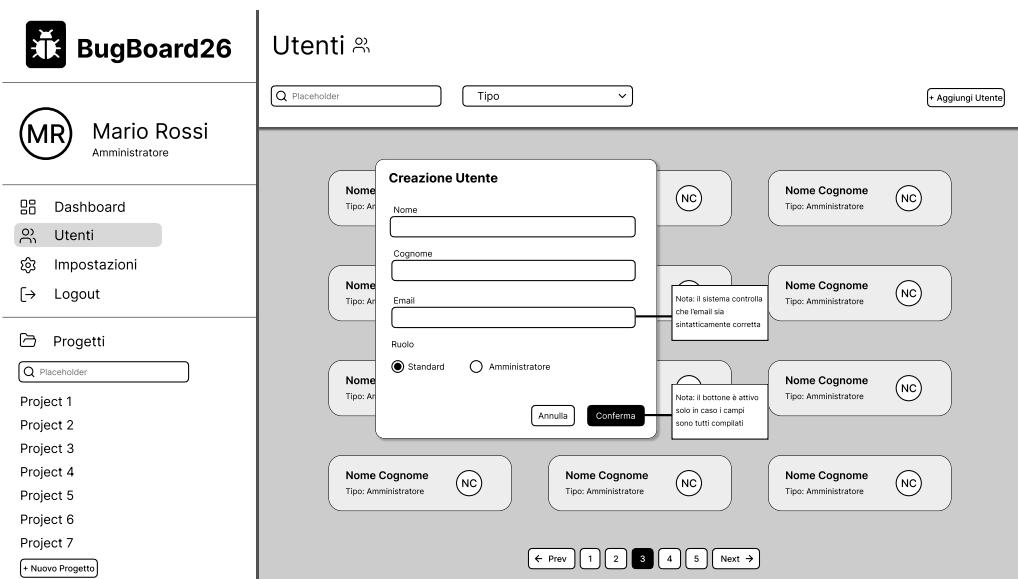


Figure 1.6: UC08_MC02

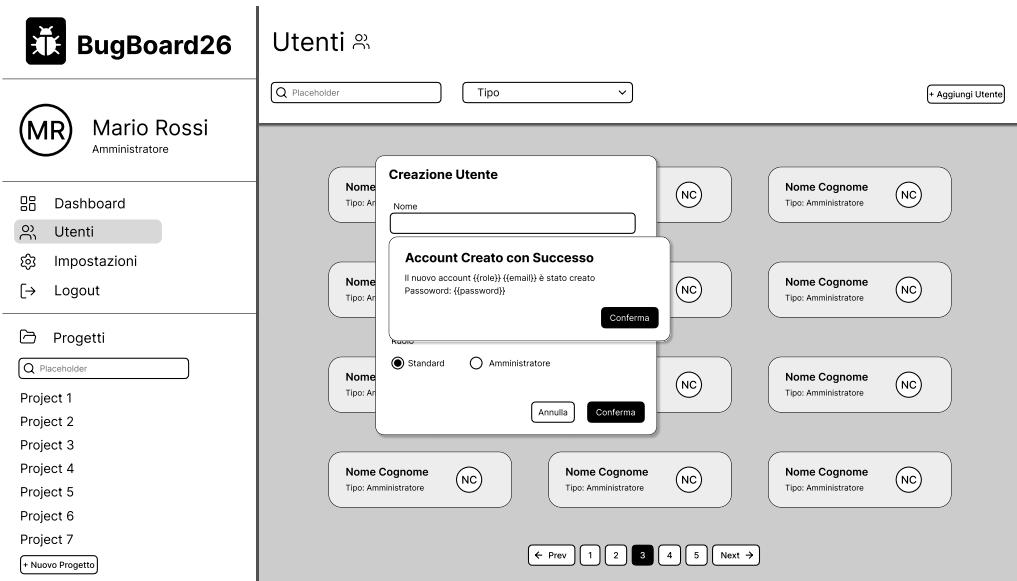


Figure 1.7: UC08_MC03

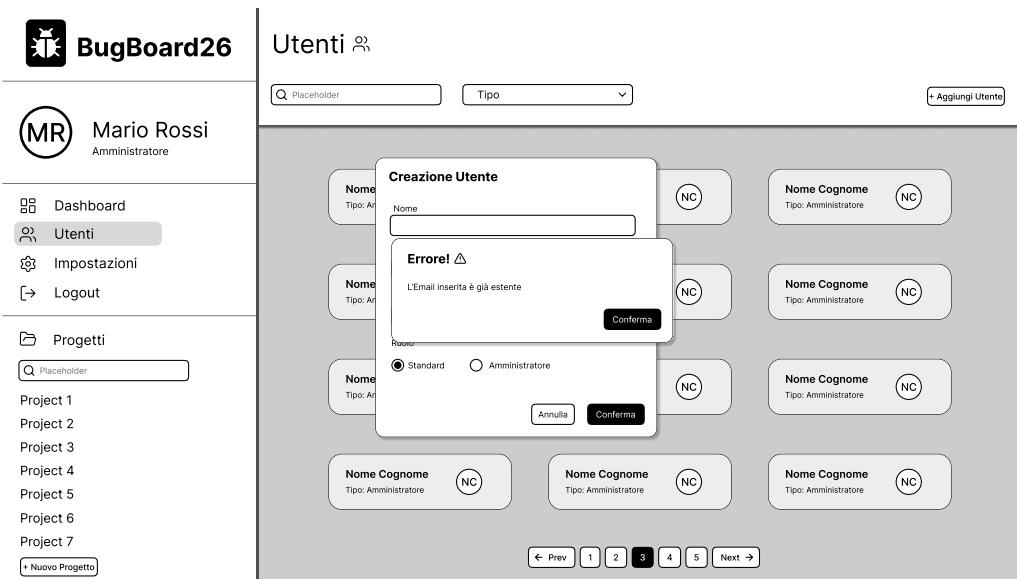


Figure 1.8: UC08_MC04

Use Case #08	Aggiungi Nuovo Utente		
Scopo	L'amministratore vuole aggiungere un nuovo utente.		
Precondizioni	L'amministratore è autenticato nel sistema.		
Condizione finale di successo	Il nuovo utente è stato aggiunto con successo nel sistema.		
Condizione finale di insuccesso	L'aggiunta del nuovo utente non è riuscita a causa di un errore del sistema o di dati non validi forniti dall'amministratore.		
Attore Principale	Amministratore.		
Trigger	L'amministratore clicca sul bottone "Nuovo Utente" in UC08_MC01.		
Main Scenario	Step n.	Amministratore	Sistema
	01	L'amministratore clicca sul bottone "Aggiungi Utente" in UC08_MC01	
	02		Il sistema mostra UC08_MC02
	03	L'amministratore inserisce nome, cognome, ruolo, email e clicca "Conferma"	
	04		Mostra UC08_MC03
	05	L'amministratore clicca sul bottone "Continua"	
	06		Mostra UC08_MC01 e termina UC
Extension #01	Step n.	Amministratore	Sistema
	3.01	L'amministratore clicca sul bottone "Annulla"	
	4.01		Mostra UC08_MC01 e termina UC
Extension #02	Step n.	Amministratore	Sistema
	3.02	L'amministratore compila il form con un'email già esistente e clicca "Conferma"	
	4.02		Mostra UC08_MC04
	5.02	L'amministratore clicca sul bottone "Conferma"	
	6.02		Riparte dal punto 2

Codice	Requisito
<i>UR08_01</i>	Il sistema deve essere in grado di controllare la formattazione dell'email

1.5.3

| Utente: Aggiunta di un commento a una Issue esistente

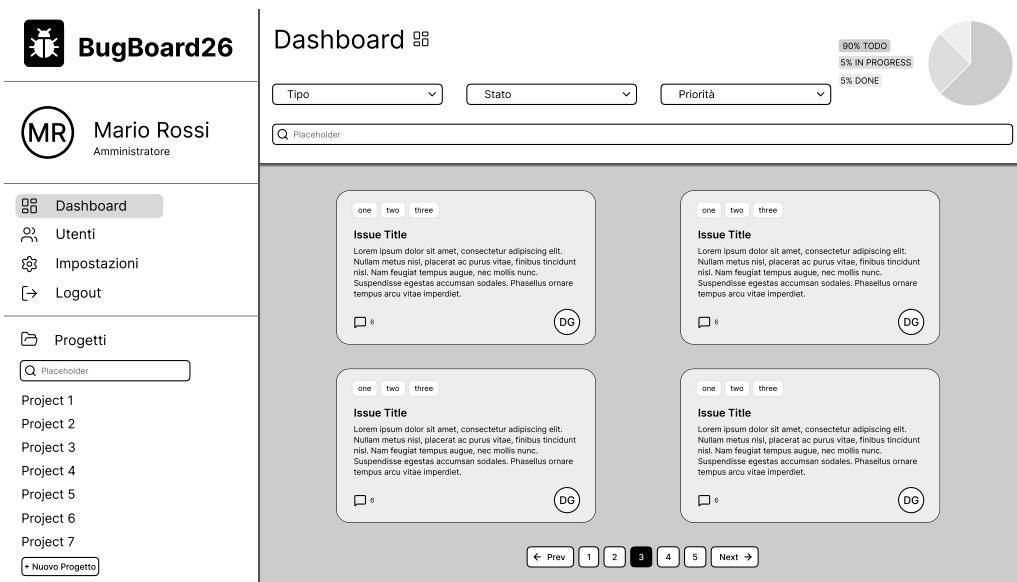


Figure 1.9: UC05_MC01

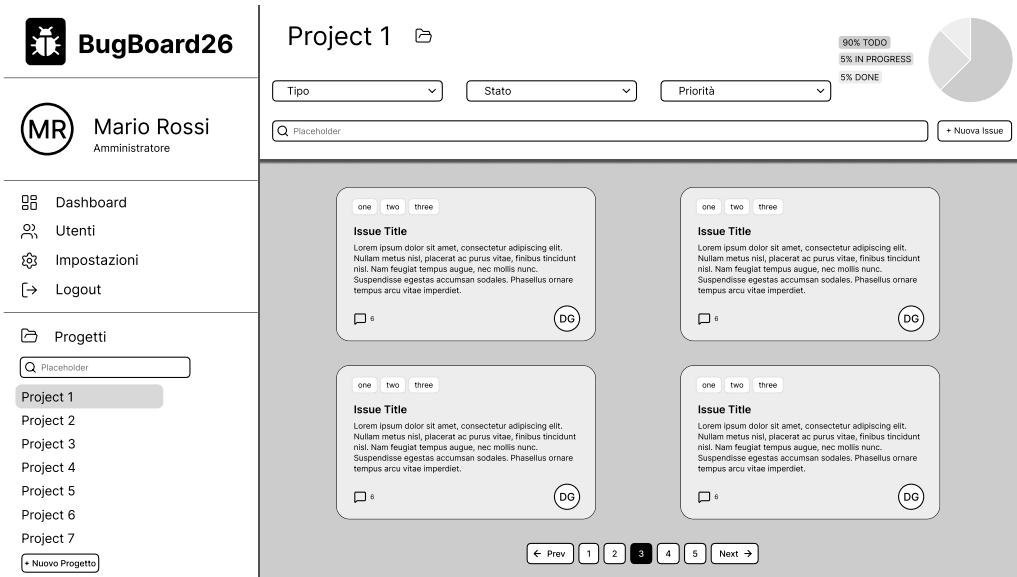


Figure 1.10: UC05_MC02

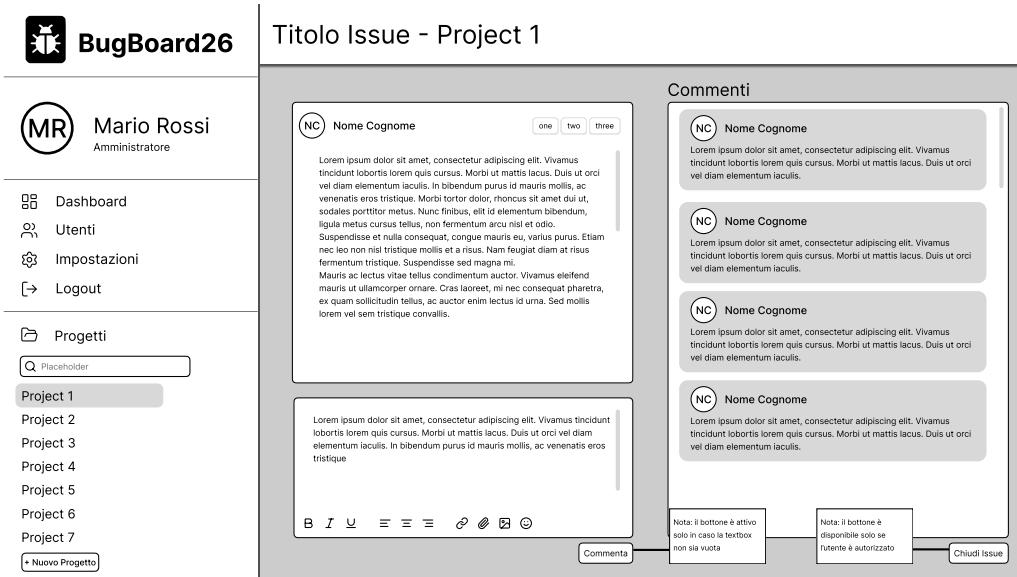


Figure 1.11: UC05_MC03

Use Case #05	Aggiungi Commento a Issue		
Scopo	L'utente vuole aggiungere un commento a una issue esistente.		
Precondizioni	L'utente è autenticato nel sistema.		
Condizione finale di successo	Il commento è stato aggiunto con successo alla issue.		
Condizione finale di insuccesso	L'aggiunta del commento non è riuscita a causa di un errore del sistema o di dati non validi forniti dall'utente.		
Attore Principale	Utente.		
Trigger	L'utente si trova in UC05_MC01 oppure in UC05_MC02 e visualizza un'issue.		
Main Scenario	Step n.	Utente	Sistema
	01	L'utente clicca su una issue per visualizzarla in UC05_MC01 o UC05_MC02	
	02		Il sistema mostra UC05_MC03
	03	L'utente compila la textbox dedicata e clicca "Commenta"	
	04		Mostra il commento in UC05_MC03 e termina UC

Codice	Requisito
UR05_01	Il sistema deve permettere il caricamento di immagini o documenti pdf. Per un massimo di tre documenti, ciascuno con dimensione massima 5MB.

II

Documento di Design del sistema

2.1

Obiettivi di Design

Gli obiettivi di design del sistema si basano sugli attributi di qualità del software definiti nello standard ISO/IEC 25002. Per priorizzare tali obiettivi, è stato adottato un sistema di ponderazione da 1 a 5, dove 5 indica la massima priorità, in funzione dell'impatto specifico di ciascuna caratteristica sulle funzionalità da implementare.

2.1.1 | Analisi delle Caratteristiche di Qualità

- **Functional Suitability (5/5)**

L'idoneità funzionale è centrale per garantire che il flusso di gestione delle anomalie supporti efficacemente il processo di QA aziendale. La **functional completeness (5/5)** è garantita dalla copertura esaustiva dei casi d'uso necessari per il tracking, inclusa la creazione di issue con tipologie diversificate (Bug, Feature, Question), la gestione di allegati e il sistema di commenti per la collaborazione. La **functional correctness (5/5)** è assicurata dalle rigorose validazioni implementate nel backend, che prevengono la creazione di ticket con dati incompleti o incoerenti, e verificata tramite test automatici specifici. La **functional appropriateness (5/5)** si riflette nella segregazione dei compiti tra ruoli "Standard" e "Admin", fornendo a ciascun utente esattamente le funzionalità pertinenti al proprio livello di responsabilità.

- **Reliability (4/5)**

L'affidabilità del sistema è solida, garantendo continuità e robustezza operativa. La **faultlessness (4/5)** è perseguita attraverso l'uso di strumenti di analisi statica e una suite di test automatizzati con **Vitest** che verifica il comportamento corretto delle API, riducendo la presenza di difetti nel codice. L'**Availability (4/5)** è assicurata dall'architettura containerizzata: i servizi critici come Postgres e Keycloak dispongono di healthcheck dedicati che prevengono l'avvio del backend finché le dipendenze non sono operative, massimizzando il tempo di attività del servizio. La **fault tolerance (4/5)** è gestita efficacemente a livello applicativo; i controller utilizzano blocchi try/catch per intercettare eccezioni impreviste senza causare il crash dell'intero servizio, restituendo invece errori strutturati al client. La **recoverability (3/5)** è implementata tramite meccanismi di rollback manuale, come la funzione cleanupFiles che elimina i file caricati se la transazione di creazione della issue fallisce, ripristinando uno stato consistente del sistema.

- **Security (5/5)**

La sicurezza è gestita in modo rigoroso, adottando standard industriali per la protezione del sistema. La **confidentiality (5/5)** è garantita dall'integrazione con Keycloak, che protegge le risorse tramite token JWT cifrati; l'accesso agli endpoint sensibili è regolato da middleware come protect e checkRole, assicurando che solo gli utenti autorizzati visualizzino i dati. L'**integrity (5/5)** è assicurata per gli allegati tramite il calcolo dell'hash SHA256 al momento dell'upload, permettendo di verificare che i file non siano stati alterati. L'**authenticity (5/5)** viene verificata crittograficamente: il backend valida la firma digitale dei token JWT utilizzando le chiavi pubbliche (JWKS) esposte da Keycloak, garantendo con certezza l'identità del mittente. La **non-repudiation (4/5)** è supportata dalla persistenza dei dati: ogni issue e commento creato è indissolubilmente legato all'ID univoco dell'utente nel database, rendendo innegabile la paternità dell'azione all'interno del sistema. Infine, l'**accountability (5/5)** è totale, poiché ogni richiesta che modifica lo stato del sistema passa attraverso il middleware di autenticazione che traccia l'utente responsabile prima di eseguire qualsiasi operazione sui controller.

- **Performance Efficiency (3/5)**

L'efficienza risente della centralizzazione della logica nel layer backend monolitico. Il **time behaviour (3/5)** è penalizzato da alcune implementazioni nel layer di accesso ai dati, come il problema "N+1" nel controller delle issue che esegue una query di conteggio per ogni elemento restituito, rallentando la risposta complessiva del server. La **resource utilization (3/5)** è critica nel layer applicativo, poiché la gestione degli allegati carica interi file in buffer di memoria direttamente nel processo Node.js principale, rischiando di saturare le risorse del container backend in caso di picchi di utilizzo. La **capacity (4/5)** rimane comunque buona grazie alla separazione netta degli strati via Docker Compose, che permette di allocare risorse dedicate o scalare il layer backend indipendentemente dal database e dal frontend.

- **Usability (4/5)**

L'usabilità è progettata per garantire un'interazione intuitiva e accessibile. La **recognizability (5/5)** è eccellente: la presenza di una dashboard iniziale con grafici a torta permette agli utenti di comprendere immediatamente lo scopo del sistema e lo stato corrente delle issue, mentre la navigazione tramite sidebar rende esplicite le funzionalità disponibili. L'**inclusivity (4/5)** è supportata dall'adozione del framework Angular Material, che fornisce componenti nativamente accessibili (come campi di input con etichette chiare e contrasto adeguato), facilitando l'uso anche a utenti con diverse abilità visive. La **learnability (4/5)** è favorita da un'interfaccia coerente e moderna (Tailwind CSS), che riduce la curva di apprendimento per i nuovi membri del team. L'**operability (4/5)** è garantita da flussi di lavoro lineari e dialoghi di conferma (es. toast) che prevengono errori operativi durante azioni critiche come l'eliminazione.

- **Compatibility (5/5)**

La compatibilità è essenziale per l'integrazione in ambienti di sviluppo eterogenei. La **co-existence (5/5)** è eccellente grazie all'uso di Docker Compose, che isola completamente le dipendenze (DB, Keycloak, Backend) evitando conflitti con altro software installato sulla macchina host. L'**interoperability (5/5)** è garantita dall'adozione di standard aperti come REST per le API, JSON per lo scambio dati e OIDC per l'autenticazione, permettendo al sistema di dialogare facilmente con altri strumenti aziendali.

- **Maintainability (5/5)**

La manutenibilità è un punto di forza, strutturata per facilitare evoluzioni future. La **modularity (5/5)** è garantita dalla netta separazione delle responsabilità nel backend tra controller, rotte e modelli dati, e nel frontend tramite l'architettura a componenti di Angular. La **reusability (5/5)** è massimizzata dall'uso di middleware generici per funzioni trasversali come l'autenticazione e la gestione dei file, riutilizzabili su diverse rotte. La **testability (5/5)** è assicurata da un'infrastruttura di testing pronta con l'utilizzo di Vitest. La **modifiability (5/5)** è supportata dall'uso dell'ORM Sequelize, che astrae le query SQL permettendo di modificare lo schema del database o la logica di business con impatti minimi sul resto del codice, e dalla configurazione esternalizzata in file .env che facilita i cambi di ambiente senza ricompilazione.

- **Flexibility (4/5)**

La flessibilità determina la capacità del software di adattarsi a nuovi contesti e tecnologie. L'**adaptability (4/5)** è gestita tramite un ampio uso di variabili d'ambiente che permettono di riconfigurare database, porte e URL di servizi esterni senza modificare il codice sorgente. La **scalability (4/5)** è intrinseca nell'architettura a container, permettendo di scalare orizzontalmente i servizi stateless come il backend e il frontend. L'**installability (3/5)** è buona grazie a Docker Compose, ma il punteggio è ridotto dalla necessità di configurare manualmente il file .env prima dell'avvio, introducendo un passaggio manuale obbligatorio che impedisce un deployment totalmente automatico "out-of-the-box". Infine, la **replaceability (4/5)** è supportata dall'uso dell'ORM Sequelize, che facilita la sostituzione del database PostgreSQL con altri dialetti SQL minimizzando le modifiche al codice.

- **Safety (3/5)**

La sicurezza intesa come assenza di rischi (Safety) riguarda la prevenzione di danni critici. Sebbene in un bug tracker il rischio fisico sia nullo, l'**operational safety (3/5)** è garantita da meccanismi che prevenendo la corruzione dei dati o l'accumulo di "rifiuti" digitali, come la cancellazione automatica dei file caricati se la transazione database fallisce. La **risk identification (3/5)** è mitigata dalle restrizioni sui ruoli, impedendo che utenti inesperti possano compromettere la configurazione dei progetti o degli utenti critici.

2.1.2 | Tabella riassuntiva delle priorità delle qualità del software

Di seguito è presentata una tabella riassuntiva delle valutazioni di priorità assegnate alle qualità del software analizzate.

Qualità	Punteggio
<i>Functional suitability</i>	5
Reliability	4
Security	5
Performance Efficiency	3
Usability	4
Compatibility	5
Maintainability	5
Flexibility	4
Safety	3

2.1.3 | Considerazioni sul Tempo di rilascio e Trade-offs

Il **Time to Release** rappresenta un vincolo critico del progetto, subordinato esclusivamente alla *Functional Suitability* e alla *Security*. La pressione temporale imposta dalle scadenze operative ha determinato trade-offs consapevoli tra qualità del software concorrenti, privilegiando il rilascio rapido di funzionalità essenziali rispetto all'ottimizzazione spinta di attributi secondari.

- **Time to Release vs Code Quality**

L'introduzione controllata di technical debt è stata accettata per rispettare le milestone di progetto, garantendo comunque la correttezza funzionale tramite test automatizzati con Vitest.

- **Time to Release vs Performance Efficiency**

L'adozione dell'ORM Sequelize ha accelerato lo sviluppo del layer di persistenza eliminando la necessità di scrivere query SQL manuali, introducendo però un overhead computazionale fisiologico. L'implementazione diretta di query ottimizzate avrebbe richiesto un dispendio temporale incompatibile con la priorità del rilascio.

- **Time to Release vs Functional Suitability**

La definizione rigorosa del perimetro funzionale ha limitato lo scope alle funzionalità essenziali: creazione, lettura, aggiornamento ed eliminazione di issue con sistema di commenti e allegati.

- **Safety vs Performance Efficiency**

I meccanismi di **operational safety** implementati introducono overhead elaborativo aggiuntivo per ogni transazione. La funzione `cleanupFiles` esegue operazioni di filesystem I/O sincrone per garantire la rimozione degli allegati orfani in caso di rollback del database, rallentando il throughput delle operazioni di creazione issue ma prevenendo la corruzione dello stato del sistema e l'accumulo di risorse non referenziate.

- **Security vs Time to Release**

Nonostante la pressione temporale, la **Security** non è stata compromessa. L'integrazione con Keycloak per la gestione di autenticazione e autorizzazione tramite token JWT, la validazione crittografica delle firme digitali con JWKS, e il calcolo dell'hash SHA256 per l'integrità degli allegati sono stati implementati integralmente sin dalla prima release, riconoscendo il rischio elevato di exploitation di vulnerabilità legate a bug leak o privilege escalation.

Decomposizione del sistema

Per poter comprendere quale architettura fosse adatta per questo progetto, abbiamo analizzato diverse alternative concentrandoci però sugli aspetti chiave di essi:

- **Architetture Standard vs. Architetture Moderne**

L'analisi ha confrontato l'adozione di architetture consolidate, come quella monolitica a strati (layered architecture), con paradigmi più recenti quali microservizi e architetture event-driven. Pur riconoscendo i benefici di approcci distribuiti in termini di scalabilità orizzontale, fault isolation e autonomia dei team, tali soluzioni comportano un overhead significativo nella gestione della comunicazione inter-service, nell'orchestrazione del deployment e nel monitoraggio distribuito. Considerando le caratteristiche del progetto un perimetro funzionale ben definito, un team di sviluppo ridotto e vincoli temporali stringenti un'architettura monolitica a strati si è rivelata la scelta ottimale, bilanciando efficacemente semplicità operativa ed efficienza nello sviluppo. Questa decisione ha consentito di allocare le risorse disponibili sull'implementazione delle funzionalità core del sistema, evitando la complessità architetturale non necessaria per la scala operativa prevista.

- **Persistenza dei Dati**

La natura strutturata delle informazioni gestite dal sistema issue con metadati tipizzati, relazioni tra utenti e progetti, commenti gerarchici e allegati referenziati ha reso necessaria l'adozione di un database relazionale (RDBMS). PostgreSQL è stato selezionato per la sua robustezza nelle operazioni transazionali ACID, essenziali per garantire la consistenza tra la creazione di issue e il caricamento di allegati, e per il supporto nativo a vincoli di integrità referenziale che prevengono la corruzione dei dati in caso di operazioni concorrenti. L'integrazione con Sequelize ORM ha ulteriormente semplificato l'accesso ai dati, astraendo la complessità delle query SQL e massimizzando la **modifiability** del layer di persistenza. Alternative NoSQL, pur offrendo scalabilità orizzontale superiore, avrebbero compromesso la capacità di eseguire query complesse con filtri multipli (per tipo, stato, progetto, assegnatario) e join tra entità correlate, funzionalità core del sistema di ricerca implementato nel frontend.

- **Cloud vs On-Premise**

Il deployment del sistema è stato progettato per ambienti cloud, sfruttando la containerizzazione tramite Docker Compose per garantire portabilità e isolamento delle dipendenze. Questa scelta elimina la necessità di gestire infrastruttura fisica dedicata e semplificando drasticamente le operazioni di provisioning e scaling. L'architettura containerizzata permette il deployment su qualsiasi provider cloud o su infrastrutture on-premise virtualizzate, mantenendo la **co-existence** con altri servizi aziendali senza conflitti di dipendenze. Un deployment on-premise tradizionale richiederebbe configurazioni manuali per PostgreSQL, Keycloak e Node.js su ciascun ambiente. La soluzione cloud consente inoltre di delegare aspetti infrastrutturali critici come backup automatici, disaster recovery e aggiornamenti di sicurezza al provider, liberando risorse del team per lo sviluppo delle funzionalità core.

- **Buy vs Build**

Nonostante la pressione temporale derivante dal **Time to Release**, la scelta è ricaduta sull'opzione "build", sviluppando il sistema internamente piuttosto che adottare soluzioni commerciali preesistenti come Jira, GitLab Issues o Bugzilla. Questa decisione è stata motivata principalmente da requisiti di **Functional Suitability** specifici per i processi di QA aziendali, che richiedevano workflow personalizzati e integrazioni con sistemi interni non supportate da strumenti generici. Tuttavia, per componenti infrastrutturali critici ma non differenzianti, è stata privilegiata l'integrazione con software open-source maturo: Keycloak in modalità headless come Identity Provider self-hosted garantisce standard industriali di **Security** (autenticazione OIDC, gestione realm con ruoli "Standard" e "Amministratore", validazione JWT tramite JWKS) senza dover implementare da zero logiche complesse di Identity and Access Management. L'infrastruttura cloud è affidata ad Amazon Web Services (AWS), che fornisce compute tramite EC2 per l'esecuzione dei container Docker (frontend Angular, backend Node.js, database PostgreSQL e Keycloak stesso). Questo approccio ibrido ha bilanciato il controllo completo sulle funzionalità core del bug tracker con l'efficienza derivante dall'utilizzo di componenti consolidate per aspetti trasversali, evitando il rischio di reinventare soluzioni già disponibili e battle-tested.

- **Complessità del Sistema e Prospettive Future**

La complessità contenuta del sistema, caratterizzato da un perimetro funzionale ben definito e da incertezze sulla sua evoluzione a lungo termine, ha reso ingiustificata l'adozione prematura di architetture distribuite come i microservizi, che avrebbero introdotto un livello di complessità eccessivo (over-engineering) rispetto alle necessità operative attuali. Tale approccio avrebbe inoltre compromesso significativamente il **Time to Release**, imponendo overhead di gestione, orchestrazione e monitoring sproporzionati alla scala del progetto. Ciononostante, la scelta di un'architettura containerizzata su cloud AWS e l'integrazione con servizi gestiti come Keycloak garantiscono la flessibilità necessaria per una potenziale migrazione verso pattern architetturali più distribuiti qualora le esigenze future del sistema richiedano maggiore scalabilità orizzontale, resilienza o autonomia dei componenti. L'adozione di standard aperti (REST API, OIDC, Docker) e la separazione netta dei layer preservano la **modifiability** necessaria per evolvere l'architettura senza riscrittura sostanziali del codice esistente.

2.2.1 | La Scelta Architetturale: Un "Monolite Modulare con Servizi Esterni"

Dopo una attenta analisi considerando pro e contro di ogni architettura, abbiamo dunque dedotto che la migliore opzione per l'architettura per questo progetto fosse un monolite modulare ma con l'utilizzo di servizi esterni, come per esempio l'identity provider. Tale architettura offre molti vantaggi, ovvero quelli di un'architettura monolitica a strati con l'ausilio di servizi indipendenti, offerti da servizi cloud come Amazon AWS. Questa soluzione è stata la scelta più adeguata data la sua semplicità di implementazione iniziale, ma soprattutto per i suoi vantaggi di scalabilità futura.

Nello specifico, le funzionalità core del sistema, ovvero la logica di business principale (ad esempio la creazione di una issue, commentare una issue etc), saranno implementate all'interno di un'architettura monolitica a strati ben definita. Parallelamente, utilizzeremo servizi esterni per gestire funzionalità trasversali come l'autenticazione.

2.2.2 | Strutturazione del core monolitico

Il core monolitico conterrà le funzionalità principali del sistema. Tale monolite è diviso in 5 layer principali:

- **User Interface;**
- **Presentation Logic;**
- **Application Logic;**
- **Data Access;**
- **Data Storage.**

In particolare ogni layer si occuperà di:

- **User Interface**

La User Interface, comunemente detta UI, si occupa della presentazione visiva e di tutto ciò che riguarda l'interazione dell'utente finale. Esso contiene tutti gli elementi dell'interfaccia e si limita a raccogliere gli input dell'utente e presentare i risultati forniti dal layer di presentation logic.

- **Presentation Logic**

Il layer di presentation è responsabile della formattazione dei dati per la visualizzazione all'utente e alle interazioni dell'utente. In pratica prende le richieste dal layer di user interface, le inoltra al layer di application logic e formatta i risultati ottenuti nell'interfaccia utente.

- **Application Logic**

Il layer di application contiene tutta la logica di business, ovvero tutte le regole e tutti i processi che definiscono il funzionamento del sistema. Fa tutto ciò esponendo gli API (Application programming interface) end-point del backend e si occupa di gestire la logica di business.

- **Data Access**

Il layer di data access si occupa dell'interazione con i sistemi di persistenza dei dati. Fornisce un livello di astrazione maggiore rispetto all'implementazione specifica del database in sé, consentendo al layer di application logic di interagire con i dati non preoccupandosi di query SQL etc.

- **Data Storage**

Il layer data storage rappresenta l'effettiva implementazione della persistenza dei dati. Principalmente include il database relazionale.

2.2.3 | Servizio Cross-Layer Esternalizzato

L'unico servizio esternalizzato è il servizio di **identity provider**.

2.2.4 | Comunicazione tra parti del sistema

L'architettura del sistema è progettata come un'architettura aperta, tenendo conto della presenza del servizio cross-layer. Inoltre, alcuni layer come quello di presentation logic, devono comunicare con più parti del sistema per soddisfare i requisiti funzionali.

Inoltre, considerando esclusivamente il core monolitico dell'architettura, la comunicazione tra i vari layer segue un paradigma di architettura chiusa. Questo approccio è scelto per garantire una chiara separazione delle responsabilità di tutti i layer. Tale separazione porta molti vantaggi tra cui il miglioramento della manutenibilità e la facilità di effettuare eventuali modifiche o evoluzioni del sistema.

Identificazione dei Threads

Nel layer di presentation logic vengono sfruttati dei threads per consentire il caricamento dei dati in parallelo rispetto al caricamento della pagina principale. Questa strategia permette di migliorare significativamente le performance complessive dell'applicazione, evitando che operazioni potenzialmente lunghe blocchino il rendering dell'interfaccia utente. In particolare, i threads vengono utilizzati per eseguire richieste asincrone verso il backend, elaborare dati complessi o effettuare chiamate a servizi esterni, garantendo così un'esperienza utente più fluida e reattiva. L'adozione di questo approccio multi-threading consente inoltre di sfruttare al meglio le risorse hardware disponibili, distribuendo il carico di lavoro su più unità di elaborazione.

2.3.1 | Parallelismo e Asincronismo

L'adozione di meccanismi di parallelismo e asincronismo è fondamentale per migliorare l'efficienza di alcune operazioni, soprattutto quando il carico di lavoro è elevato. Alcuni esempi di caso d'uso significativi di queste tecniche sono:

- **Gestione Concorrente degli Allegati**

Durante la creazione o l'aggiornamento di una issue, il sistema deve gestire l'upload di molteplici file (immagini o PDF). Nel controller `createIssue` e nella funzione helper `createAttachments`, il codice non processa i file in sequenza. Viene invece creato un array di Promises mappando i file in ingresso: per ciascun file vengono avviate simultaneamente le operazioni di lettura dal disco (`fsPromises.readFile`), calcolo dell'hash crittografico (SHA256) e inserimento nel database. L'uso di `Promise.all` attende che tutte queste operazioni parallele siano completate prima di restituire la risposta, riducendo drasticamente il tempo di attesa complessivo per l'utente rispetto a un'esecuzione sequenziale.

- **Aggregazione Dati per le Liste:** Nella funzione ‘`fetchIssuesWithComments`’, utilizzata per recuperare la lista delle issue da mostrare nella dashboard, il sistema deve associare a ogni issue il numero dei suoi commenti. Invece di iterare e attendere sequenzialmente il conteggio per ogni singola issue (che causerebbe un forte rallentamento), il sistema lancia tutte le query di conteggio (‘`countCommentsByIssueId`’) in parallelo utilizzando ‘`Promise.all`’ su una mappa delle issue trovate. Questo permette di sfruttare la capacità del database di gestire connessioni concorrenti.

- **Rollback e Pulizia delle Risorse:** Nel caso in cui la creazione di una issue fallisca (ad esempio per errori di validazione o database), la funzione ‘`cleanupFiles`’ garantisce che i file fisici caricati vengano rimossi per non lasciare "rifiuti". Anche questa operazione sfrutta il parallelismo: viene generata una lista di operazioni di cancellazione (‘`fsPromises.unlink`’) che vengono eseguite simultaneamente tramite ‘`Promise.all`’, minimizzando il tempo impiegato per la gestione dell'errore.

2.4

Descrizione e motivazione delle scelte tecnologiche adottate

2.4.1 | Hardware-Software Mapping

Come discusso nelle sezioni precedenti, l'architettura del sistema fa uso di tecnologie cloud per garantire scalabilità, flessibilità e una gestione semplificata delle risorse in generale. Per rendere l'applicazione accessibile via rete, si utilizzeranno i servizi offerti da Amazon Web Services (AWS). Quindi la struttura sarà organizzata come di seguito

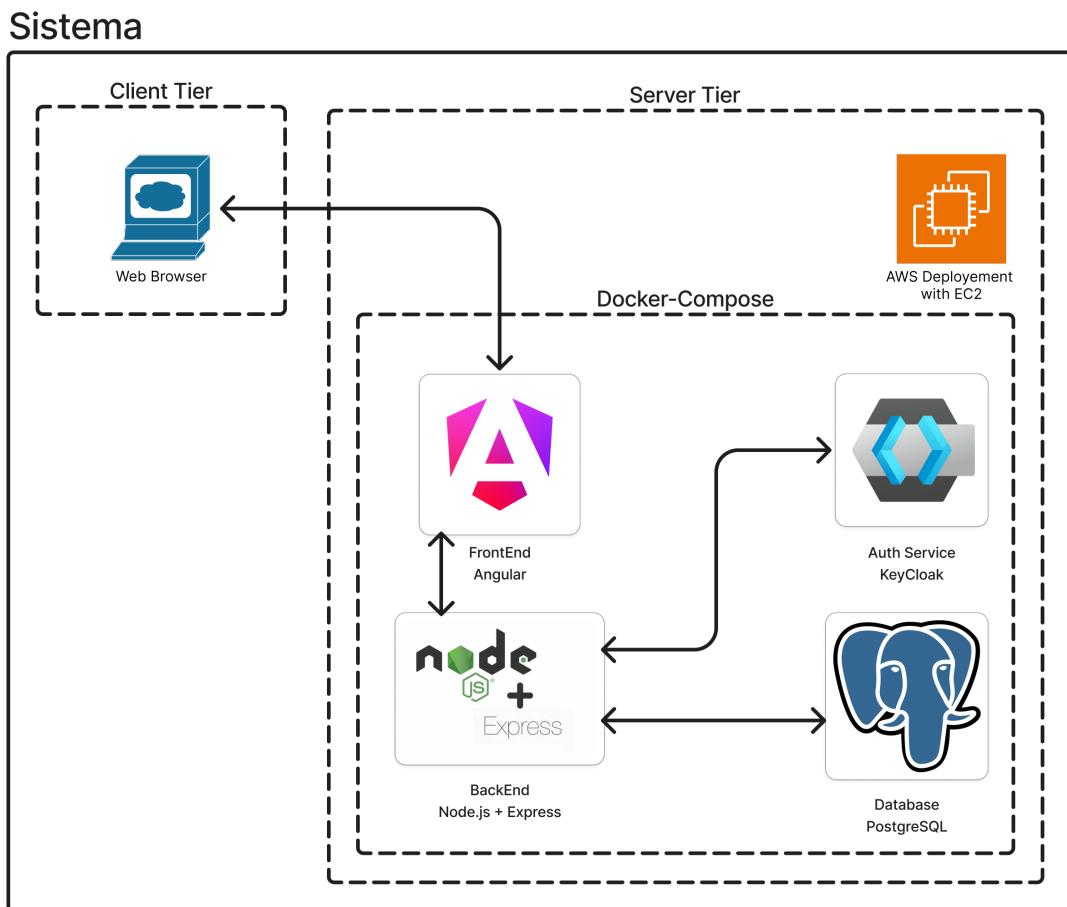


Figure 2.1: Figura rappresentante il sistema

2.4.2 | Distribuzioni delle Componenti del Sistema

2.4.2.1 Layer del Core Monolitico

Le varie parti del sistema sono distribuite tra il client dell'utente e un'infrastruttura cloud basata su container. Ecco i dettagli:

- **User Interface e Presentation Logic Layer**

Questi due layer sono realizzati come **Web Application SPA (Single Page Application)** e vengono eseguiti direttamente all'interno del browser sul dispositivo dell'utente. Il browser scarica il bundle frontend (Angular) fornito dal server e gestisce la logica di presentazione localmente, garantendo un'esperienza utente reattiva senza gravare sul server per il rendering dell'interfaccia.

- **Application Layer e Data Access Layer**

Questi layer costituiscono il cuore del backend e sono incapsulati in container Docker. L'orchestrazione avviene tramite **Docker Compose** su una macchina virtuale **AWS EC2** (Amazon Elastic Compute Cloud). Questa scelta permette di avere un ambiente riproducibile e isolato, dove la logica di business viene eseguita all'interno del proprio container dedicato.

- **Data Storage Layer**

La persistenza dei dati è gestita interamente all'interno dell'ecosistema Docker sulla macchina virtuale, utilizzando volumi per garantire la durata dei dati:

- Il database relazionale PostgreSQL è ospitato in un container dedicato definito nel docker-compose, che sostituisce la necessità di un servizio gestito esterno;
- Gli allegati vengono salvati sul file system locale tramite volumi Docker collegati al container di backend, gestiti direttamente dalle API di upload del sistema.

2.4.2.2 Servizi Cross-Layer

Alcune funzionalità trasversali al sistema sono state esternalizzate a servizi specializzati, ospitati in container dedicati, per ottimizzare la gestione delle identità e garantire la scalabilità:

- **Autenticazione e Gestione Identità**

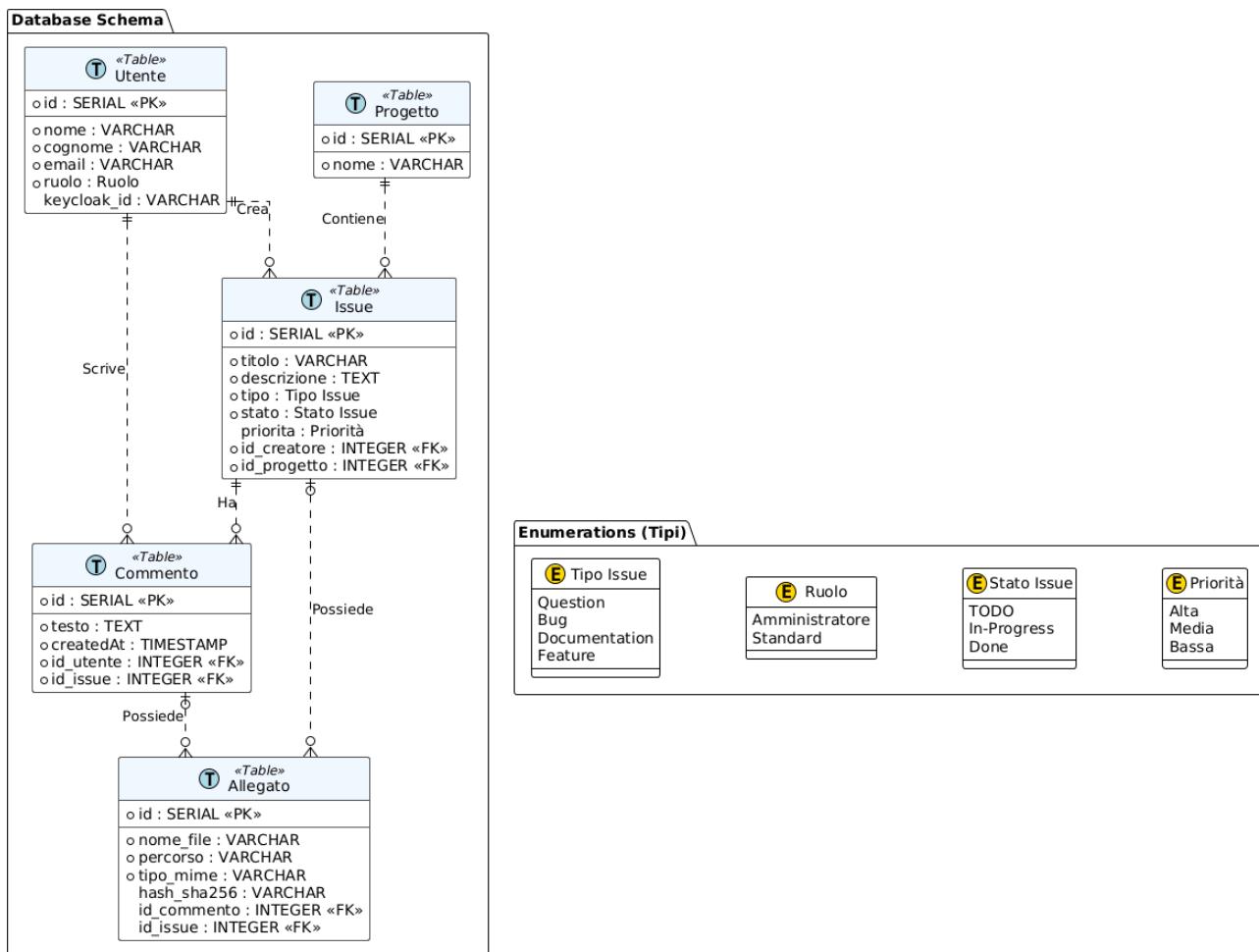
Gestita tramite Keycloak, una soluzione di Identity and Access Management open-source. Per integrarsi perfettamente con l'interfaccia utente personalizzata sviluppata in Angular, Keycloak è utilizzato in modalità "headless": il frontend gestisce i form di accesso e comunica con il backend, che agisce da intermediario sicuro per la validazione delle credenziali e l'emissione dei token JWT, senza reindirizzare l'utente su pagine di login esterne.

III

Documento di Design del Software, documentazione del processo di sviluppo, e artefatti software.

3.1

Descrizione dello schema per la persistenza dati (Database)



3.1.1 | Il Ruolo del Database nel Sistema

Contrariamente ai paradigmi Database-Centric che delegano la logica di business a stored procedures complesse, l'architettura di sistema adotta un approccio moderno basato su ORM (Sequelize). Il database PostgreSQL, containerizzato via Docker, agisce come Persistence Layer robusto, responsabile dell'integrità referenziale e della consistenza dei dati (ACID), mentre la logica di business complessa è centralizzata nel Backend. Tuttavia, per garantire la massima coerenza dei dati critici indipendentemente dal flusso applicativo, sono stati implementati mirati meccanismi attivi nel database, come Trigger per l'aggiornamento automatico degli stati e vincoli di integrità (Check Constraints) per la validazione degli allegati.

3.1.2 | PostgreSQL come Motore di Persistenza Relazionale

La scelta di PostgreSQL come foundation tecnologica deriva dalla necessità di garantire la massima integrità e consistenza dei dati in un ambiente multi-utente. Più che una semplice memoria di massa, il database agisce come garante ultimo della qualità del dato.

Le capacità avanzate di PostgreSQL sono sfruttate in modo mirato per rinforzare la logica di dominio:

- **Type Safety e Enum**

L'utilizzo di tipi enumerativi personalizzati per stati, ruoli e priorità assicura che il sistema gestisca solo valori semanticamente validi, prevenendo errori di incoerenza a livello strutturale.

- **Integrità Referenziale Avanzata**

L'architettura fa ampio uso di vincoli FOREIGN KEY con clausole ON DELETE CASCADE per gestire automaticamente la pulizia dei dati correlati (es. eliminazione di issue e commenti quando viene eliminato un progetto), semplificando la logica applicativa.

- **Vincoli Complessi**

Sono implementati CHECK constraints sofisticati, come la logica XOR sulla tabella allegati (che garantisce l'appartenenza esclusiva a issue o commenti), delegando al database la validazione di invarianti critiche che potrebbero sfuggire al livello applicativo.

3.1.3 | Sistema di Tipi e Valutazione: Type Safety nel Database

3.1.3.1 Sistema di Tipi Personalizzati

Per modellare fedelmente il dominio del problema, sono stati definiti tipi enumerativi personalizzati che restringono il campo dei valori ammissibili per attributi critici, eliminando l'ambiguità delle stringhe libere:

- **Ruoli e Permessi**

Il tipo ruolo ('Amministratore', 'Standard') centralizza la definizione della gerarchia utenti.

- **Workflow delle Issue**

I tipi tipo_issue ('Bug', 'Feature', ecc.) e stato_issue ('TODO', 'In-Progress', 'Done') cristallizzano il ciclo di vita delle segnalazioni direttamente nello schema del database, impedendo l'inserimento di stati non riconosciuti dal sistema.

3.1.3.2 Validazione Strutturale e Constraints

Oltre ai vincoli standard di unicità e chiave esterna, sono stati implementati vincoli CHECK complessi per garantire regole di business strutturali:

- **Integrità Polimorfica degli Allegati**

La tabella allegato utilizza un vincolo logico XOR per garantire che un file sia associato esclusivamente a un commento O a una issue, ma mai a entrambi o a nessuno. Questo previene record orfani o relazioni ambigue a livello strutturale.

- **Limiti Fisici**

È stato imposto un vincolo rigido sulla dimensione dei file (dimensione_byte <= 5242880), assicurando che il database rifiuti transazioni che violano i requisiti non funzionali di storage, fungendo da ultima linea di difesa contro errori di validazione nel frontend.

3.2

Evoluzione e Manutenibilità del Sistema

3.2.1 | Architettura Evolutiva e Modularità

Il sistema è stato progettato seguendo un'architettura a livelli (Layered Architecture) che favorisce la manutenibilità e l'evoluzione indipendente dei componenti. La separazione netta tra il livello di presentazione (Frontend Angular), la logica di business (Backend Node.js) e la persistenza dei dati (PostgreSQL) permette modifiche mirate senza effetti a cascata.

3.2.2 | Osservabilità e Diagnostica

Per quanto riguarda osservabilità e diagnostica, ci affidiamo alla robustezza dell'infrastruttura. Il database stesso funge da primo strumento di debug: i suoi vincoli di integrità bloccano sul nascere dati incoerenti, segnalandoci subito eventuali errori logici. Lato logging, Docker ci permette di aggregare tutto in un unico flusso, correlando facilmente l'autenticazione di Keycloak con la logica di business. Infine, abbiamo adottato una nomenclatura semantica che rende lo schema del database auto-esplicativo, riducendo drasticamente la necessità di consultare documentazione esterna.

3.2.3 | Strategie per scalabilità futura

Guardando al futuro, abbiamo progettato il sistema per non essere un sistema chiuso, ma pronto a scalare. La scelta di un backend stateless, basato su token JWT e Keycloak, è fondamentale: ci permetterebbe di aggiungere nuove istanze server in parallelo senza dover modificare una sola riga di codice. Inoltre, abbiamo protetto le prestazioni del database separando fisicamente gli allegati pesanti dai dati testuali e pre-calcolando indici strategici sulle tabelle critiche, garantendo così tempi di risposta rapidi anche qualora il numero di issue dovesse crescere drasticamente.

Descrizione della logica di base (Backend)

Il backend del sistema BugBoard26 implementa le funzionalità server-side di una piattaforma dedicata al Bug Tracking e alla gestione collaborativa delle problematiche software. Il sistema è stato sviluppato utilizzando l'ambiente di runtime Node.js e il framework Express, avvalendosi del driver nativo pg per un'interazione performante e diretta con il database relazionale PostgreSQL.

L'architettura software adotta il pattern Layered Architecture (architettura a livelli), strutturando il codice in strati logici con responsabilità distinte: le Routes per la definizione degli endpoint, i Controllers per l'elaborazione delle richieste e il Data Access Layer per l'interazione con il database. La comunicazione tra client e server è gestita tramite API REST, mentre la sicurezza è delegata all'Identity Provider Keycloak, che gestisce l'autenticazione e l'autorizzazione attraverso standard OpenID Connect e token JWT. L'intera infrastruttura, inclusi database e servizi di supporto, è containerizzata e orchestrata tramite Docker, assicurando coerenza tra gli ambienti di sviluppo e produzione.

3.3.1 | Gestione delle Dipendenze

Moduli ES6 e Pattern Singleton Per mantenere l'architettura snella e performante, il backend evita l'introduzione di complessi container di Inversion of Control (IoC). La gestione delle dipendenze è affidata al sistema nativo di ES6 Modules, implementando il Pattern Singleton per la connessione al database. L'istanza di Sequelize e i relativi modelli vengono inizializzati una singola volta all'avvio (Database.js) e importati staticamente nei controller. Questa scelta riduce drasticamente l'overhead a runtime e semplifica il tracciamento del flusso di esecuzione ("Code Navigation"), pur mantenendo una chiara separazione tra la logica di accesso ai dati e la logica di business.

3.3.2 | Gestione della Concorrenza e Integrità Transazionale

Data la natura collaborativa di BugBoard²⁶, la gestione di accessi concorrenti è critica (es. due utenti che commentano contemporaneamente o un utente che commenta mentre un admin chiude la issue). La concorrenza è gestita delegando l'atomicità al livello più robusto: il Database Relazionale (PostgreSQL).

- **Transazioni Applicative (Sequelize Hooks)**

La logica di validazione critica è incapsulata in Hooks del modello ORM che operano all'interno di transazioni database. Ad esempio, prima di creare un commento, il sistema verifica lo stato della Issue all'interno della stessa transazione SQL dell'inserimento. Questo approccio (Pessimistic Check) previene race conditions in cui un commento potrebbe essere registrato su una issue che è stata chiusa millisecondi prima.

- **Non-Blocking I/O**

A livello di server, l'architettura Event-Driven di Node.js gestisce la concorrenza delle richieste HTTP tramite un singolo thread e un Event Loop non bloccante. Questo permette al sistema di scalare su un alto numero di connessioni simultanee (I/O bound) senza il costoso overhead del context-switching tipico dei modelli thread-per-request.

Logica di Implementazione del Backend

3.4.1 | L'adozione di Express.js e la Middleware Pipeline

Il cuore del layer applicativo è costruito sul framework **Express.js**, scelto per la sua leggerezza e flessibilità nel gestire architetture basate su API REST. L'implementazione segue il pattern della *Chain of Responsibility* attraverso una pipeline di middleware configurata per elaborare le richieste HTTP in ingresso in modo sequenziale e modulare.

Analizzando il punto di ingresso del server, la pipeline è strutturata in fasi distinte per garantire sicurezza e correttezza prima che la richiesta raggiunga la logica di business:

- **Sicurezza degli Header (Helmet)**

Come prima linea di difesa, viene utilizzato il middleware `helmet`. Questo modulo si occupa di impostare vari header HTTP relativi alla sicurezza, disabilitando header potenzialmente rischiosi come `X-Powered-By`, che potrebbero rivelare informazioni sulla tecnologia sottostante agli attaccanti, riducendo la superficie di attacco del server.

- **Gestione CORS (Cross-Origin Resource Sharing)**

Dato che il frontend (Angular) e il backend risiedono su domini o porte differenti (rispettivamente 4200 e 3000 in sviluppo), è stata implementata una configurazione CORS rigorosa. Il sistema accetta richieste solo dall'origine del client specificata nelle variabili d'ambiente (`CLIENT_URL`), permettendo esclusivamente i metodi HTTP necessari (GET, POST, PUT, DELETE) e l'invio di credenziali (cookie/token) tramite l'opzione `credentials: true`.

- **Parsing e Gestione Dati**

Le richieste in ingresso vengono elaborate da `express.json()` per il parsing dei payload JSON e da `cookie-parser` per la lettura dei cookie sicuri, fondamentali per il mantenimento della sessione utente.

- **Integrazione con Keycloak (Auth Middleware)**

Il middleware di autenticazione di Keycloak è posizionato strategicamente prima delle rotte API. Utilizzando una gestione di sessione basata su memoria (`MemoryStore`), il backend intercetta le richieste per verificare la validità del token JWT o la presenza di una sessione attiva, agendo da gatekeeper per tutte le risorse protette.

3.4.2 | Organizzazione e Routing delle API

L'architettura delle rotte segue un approccio RESTful, segregando le funzionalità in moduli distinti per migliorare la manutenibilità e la navigabilità del codice. Le rotte sono prefissate dal path `/api` per distinguere chiaramente le chiamate di servizio dalle risorse statiche o di navigazione.

La struttura delle rotte rispecchia le entità del dominio modellate nel database:

- `/api/auth`: Gestisce i flussi di autenticazione, logout e refresh dei token, interfacciandosi direttamente con l'Identity Provider.
- `/api/users`: Espone endpoint per la gestione dei profili utente e la sincronizzazione dei dati anagrafici tra Keycloak e il database locale.
- `/api/projects`: Gestisce il ciclo di vita dei progetti, inclusa la creazione e l'assegnazione dei membri del team.
- `/api/issues`: Rappresenta il core del bug tracker, permettendo operazioni CRUD sui ticket e gestendo logiche complesse come l'upload degli allegati.
- `/api/comments`: Gestisce l'interazione collaborativa, permettendo l'aggiunta e la visualizzazione dei commenti collegati alle issue.

Infine, il server espone una rotta statica dedicata `/uploads`, servita direttamente da Express, per permettere al frontend di recuperare in modo performante gli asset (immagini o documenti) caricati dagli utenti, mantenendo i file fisici separati dalla logica applicativa ma accessibili via HTTP.

3.4.3 | Gestione Avanzata delle Autorizzazioni e RBAC

Oltre all'autenticazione gestita da Keycloak, il sistema implementa un controllo degli accessi basato sui ruoli, **Role-based access control** (RBAC), direttamente nel layer applicativo tramite middleware personalizzati.

Un esempio critico è il middleware `canModifyIssue`, che regola l'accesso alle operazioni di modifica e cancellazione. La logica implementata distingue dinamicamente i permessi in base al ruolo e alla proprietà della risorsa:

- **Ruolo Amministratore**

Se il token JWT contiene il ruolo *Amministratore* (verificato analizzando sia i ruoli del realm che quelli del client), l'operazione viene sempre consentita, garantendo la supervisione totale del sistema.

- **Utente Standard (Owner)**

Se l'utente non è amministratore, il middleware esegue una query puntuale sul database per verificare se l'ID dell'utente autenticato corrisponde al campo `id_creatore` dell'issue target. In caso negativo, la richiesta viene respinta con un codice 403 (Forbidden), impedendo la modifica di risorse altrui.

3.4.4 | Implementazione delle Regole di Business

I controller non si limitano a operazioni CRUD elementari, ma incapsulano regole di dominio specifiche per garantire l'integrità e la tracciabilità dei dati. Analizzando l'`IssueController`, emergono tre pattern implementativi rilevanti:

- **Integrità e Sicurezza degli Allegati**

Durante l'upload, il sistema non si limita a salvare i file. Per ogni allegato viene calcolato l'hash **SHA256** del contenuto binario tramite il modulo `crypto` di Node.js. Questo hash viene salvato nel database per garantire l'integrità del dato e la verifica di non alterazione nel tempo. Inoltre, viene applicata una validazione pre-upload che impedisce di superare il limite di 3 allegati per issue, interrogando il database per contare gli allegati esistenti prima di accettarne di nuovi.

- **Audit Trail delle Modifiche (Append-Only)**

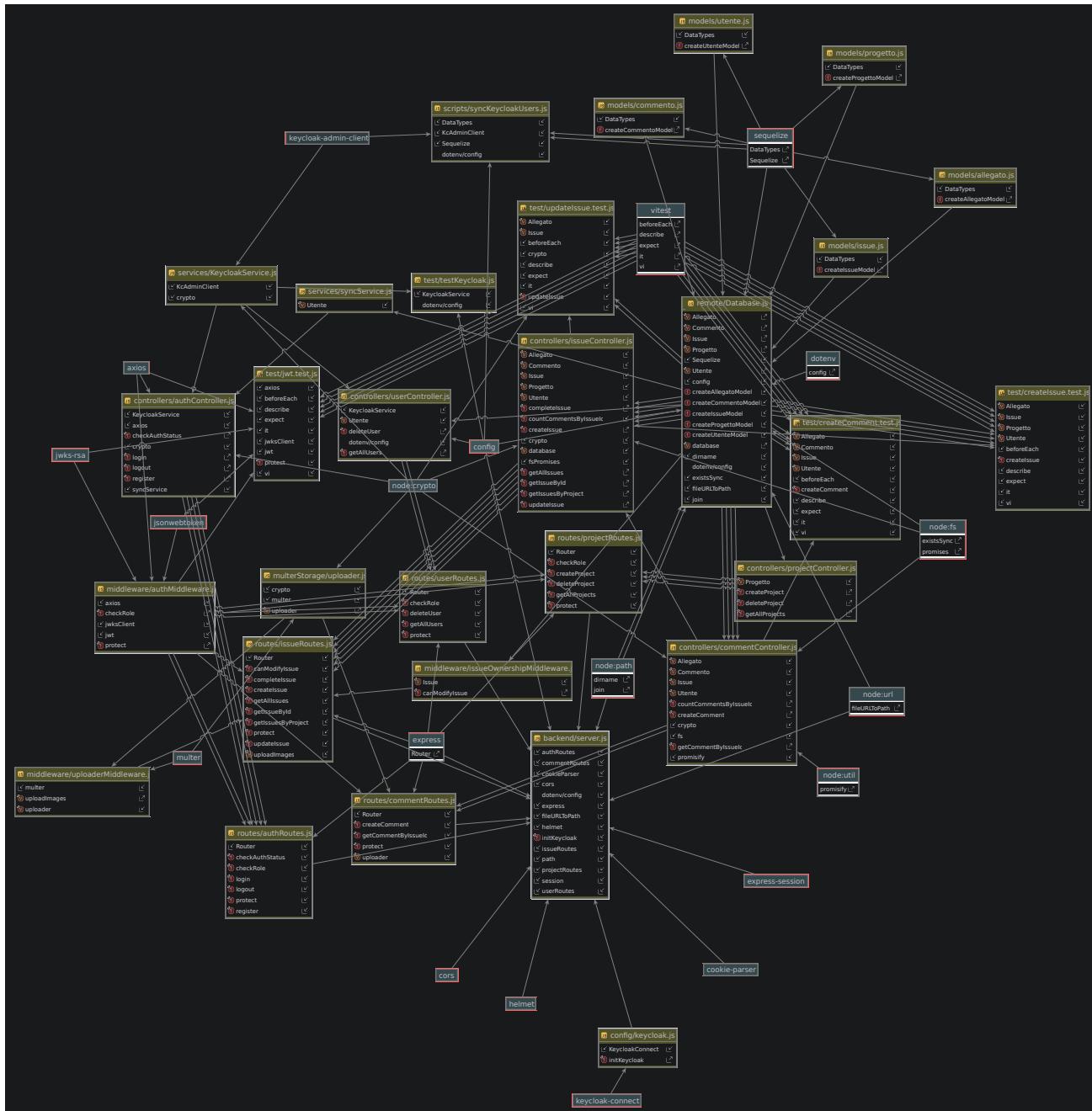
Per preservare la storia delle modifiche, l'aggiornamento della descrizione di una issue non sovrascrive il testo precedente. Il sistema utilizza una logica *append-only*: la nuova descrizione viene accodata a quella esistente, separata da un header temporale generato lato server (timestamp). Questo garantisce che nessuna informazione venga persa e fornisce una cronologia immediata dell'evoluzione del ticket.

- **Transazionalità e Pulizia (Rollback)**

In caso di errori durante la creazione complessa di una issue (es. fallimento della query SQL dopo l'upload fisico dei file), viene attivata una procedura di *cleanup*. Il sistema intercetta l'eccezione e rimuove fisicamente dal filesystem i file appena caricati ma orfani di record nel database, prevenendo l'inconsistenza dello storage.

3.5

Modello dei Dati Backend



Deployment e Configurazione

3.6.1 | Configurazione

Il sistema utilizza file YAML per la configurazione, con supporto per override tramite variabili d'ambiente. Questo approccio permette di mantenere la stessa build per diversi ambienti, cambiando solo la configurazione.

I parametri configurabili includono connessione al database, credenziali Keycloak, varie configurazioni di Keycloak, porte usate.

3.6.2 | Containerizzazione

L'applicazione è containerizzata usando Docker, con un Dockerfile ottimizzato per produrre immagini minimal. Il container espone solo la porta necessaria e include health check per facilitare l'orchestrazione.

Descrizione e motivazione delle scelte di design dell'interfaccia utente adottate (Frontend)

3.7.1 | Introduzione

Il frontend dell'applicazione BugBoard26 è un sistema completo per la gestione di issue legate a dei progetti sviluppato come applicazione web SPA (Single Page Application) utilizzando Angular con l'ausilio di Tailwind. L'applicazione rappresenta un esempio di implementazione moderna che integra pattern architetturali consolidati con tecnologie all'avanguardia per creare un'esperienza utente fluida e manutenibile.

L'obiettivo principale del progetto è fornire una piattaforma che permetta alle software house e singoli, di tenere traccia di ogni bug presente nelle loro applicazioni e per supervisionare l'andamento della risoluzione dei tali.

3.7.2 | Architettura e Pattern Fondamentali

3.7.2.1 Component-Bases Architetture

L'applicazione frontend implementa un'architettura rigorosamente Component-Based, organizzata secondo il pattern "Smart vs Dumb Components". Questa scelta garantisce che la logica di business e di recupero dati sia disaccoppiata dalla logica di presentazione visiva. I componenti "Smart" (o Container) agiscono come orchestratori: gestiscono lo stato della pagina, interagiscono con i servizi e passano i dati verso il basso. I componenti "Dumb" (o Presentational) sono puramente funzionali alla visualizzazione: ricevono dati tramite input, emettono eventi tramite output e rimangono agnostici rispetto al contesto applicativo, massimizzando la riusabilità del codice UI attraverso l'intera applicazione.

3.7.2.2 Programmazione Reattiva con RxJS

Al posto di pattern imperativi, l'applicazione adotta il paradigma della Programmazione Reattiva sfruttando la libreria RxJS. Questo approccio permette di gestire i flussi di dati asincroni (chiamate HTTP, eventi utente, input form) come stream continui (Observables). L'utilizzo degli Observables garantisce una gestione elegante della complessità asincrona: i componenti sottoscrivono i flussi di dati forniti dai servizi e reagiscono automaticamente ai cambiamenti di stato. Questo elimina la necessità di callback nidificate e permette di trasformare, filtrare e combinare i dati in arrivo dal backend prima che questi raggiungano la vista, garantendo che l'interfaccia utente rifletta sempre lo stato più aggiornato dei dati in modo predicable.

3.7.2.3 Service Layer Pattern

Il Service Layer rappresenta l'incarnazione della logica di business nel frontend, centralizzando la comunicazione con le API REST del backend. I servizi Angular encapsulano la complessità delle chiamate di rete, la gestione degli header di autenticazione (tramite Interceptor) e la gestione degli errori. Delegando la logica di interazione dati ai servizi, i componenti (ViewModel) rimangono snelli e focalizzati esclusivamente sulla gestione dello stato della vista. I servizi sono progettati come Singleton, garantendo una gestione efficiente delle risorse e permettendo la condivisione dello stato tra diversi componenti quando necessario, senza duplicazione di logica.

3.7.2.4 Single Responsibility Principle (SRP)

Il principio di singola responsabilità permea l'intera struttura del progetto attraverso una granularità fine dei moduli. Nel frontend, ogni componente UI ha una responsabilità visiva unica (es. una card per le issue, una sidebar di navigazione), mentre la logica è delegata ai servizi. Nel backend, la separazione è garantita dall'uso di middleware per compiti trasversali (autenticazione, upload file, gestione errori) e controller distinti per ogni entità di dominio. Questa separazione meticolosa facilita la manutenzione evolutiva: la modifica della logica di validazione di un form, ad esempio, è isolata nel relativo componente o servizio, senza rischio di regressioni in parti non correlate del sistema.

3.7.2.5 Angular Dependency Injection

L'applicazione sfrutta il potente sistema di Dependency Injection (DI) nativo di Angular per gestire le dipendenze in modo dichiarativo e gerarchico. Questo pattern inverte il controllo della creazione degli oggetti: i componenti non istanziano le proprie dipendenze (come i servizi HTTP), ma le ricevono iniettate dal framework a runtime. L'approccio Type-Safe della DI di Angular non solo riduce l'accoppiamento tra le classi, ma facilita enormemente il testing unitario. È possibile, infatti, sostituire facilmente le implementazioni reali dei servizi con Mock o Stub durante i test, verificando il comportamento dei componenti in isolamento senza dover effettuare reali chiamate di rete.

3.7.3 | Organizzazione delle Directory

L'organizzazione del codice sorgente nel client Angular riflette una chiara distinzione tra le funzionalità di business (Pagine) e le risorse infrastrutturali o condivise. La struttura della directory `src/app` adotta una convenzione visiva immediata basata sull'uso di prefissi:

- **Directory Core e Shared (Prefisso `_`):** Le cartelle che iniziano con underscore raggruppano elementi trasversali e non accessibili direttamente tramite routing.
 - `_services`: Centralizza la logica di comunicazione HTTP e la gestione dello stato, isolandola dai componenti.
 - `_internalComponents`: Contiene i componenti "Dumb" riutilizzabili (es. card, modali, elementi UI) che costituiscono i mattoni dell'interfaccia.
 - `_auth` e `_config`: Custodiscono rispettivamente le guardie di navigazione (Guards) e le configurazioni globali (es. inizializzazione Keycloak).
- **Directory Feature (Pagine):** Le cartelle senza prefisso (es. `dashboard`, `project`, `users`) rappresentano le viste principali dell'applicazione. Ogni directory incapsula tutto il necessario per quella specifica rotta, agendo come un modulo funzionale autonomo.

Evidenza dell'uso di strumenti di versioning

Per la gestione del codice sorgente e il tracciamento delle modifiche, il team ha adottato **Git** come sistema di version control distribuito, appoggiandosi alla piattaforma **GitHub** per l'hosting remoto della repository.

3.8.1 | Sistema di Versioning e Workflow

Il repository del progetto è ospitato pubblicamente su GitHub al seguente indirizzo:

<https://github.com/DavideGargiulo/BugBoard26-SWE>

Dato il team ristretto e la necessità di iterazioni rapide, è stata adottata una strategia di sviluppo centralizzata (*Centralized Workflow*). Il team ha lavorato direttamente sul branch principale (**main**), coordinandosi per evitare conflitti e garantendo che ogni commit rappresentasse un incremento funzionale stabile.

3.8.2 | Continuous Integration e Qualità del Codice

L'integrazione continua (CI) è stata automatizzata tramite **GitHub Actions**. Nello specifico, è stato configurato un workflow automatico che, ad ogni push sul branch **main**, esegue l'analisi statica del codice tramite **SonarQube**. Questo ha permesso di mantenere monitorata la qualità del codice (Quality Gate) durante tutto il ciclo di sviluppo, rilevando tempestivamente "code smells" o vulnerabilità.

3.8.3 | Report Statistici

Di seguito sono riportate le evidenze grafiche dell'attività di sviluppo estratte dagli *Insights* di GitHub, che mostrano la costanza e la distribuzione del lavoro.

3.8.3.1 Frequenza dei Commit

Il grafico seguente mostra l'attività temporale dei commit, evidenziando le fasi di sviluppo più intenso.



Figure 3.1: Cronologia dei commit sul branch main.

3.8.3.2 Frequenza del Codice (Code Frequency)

Il grafico della frequenza del codice illustra la quantità di righe aggiunte ed eliminate nel tempo, testimoniando la crescita del progetto e le fasi di refactoring.

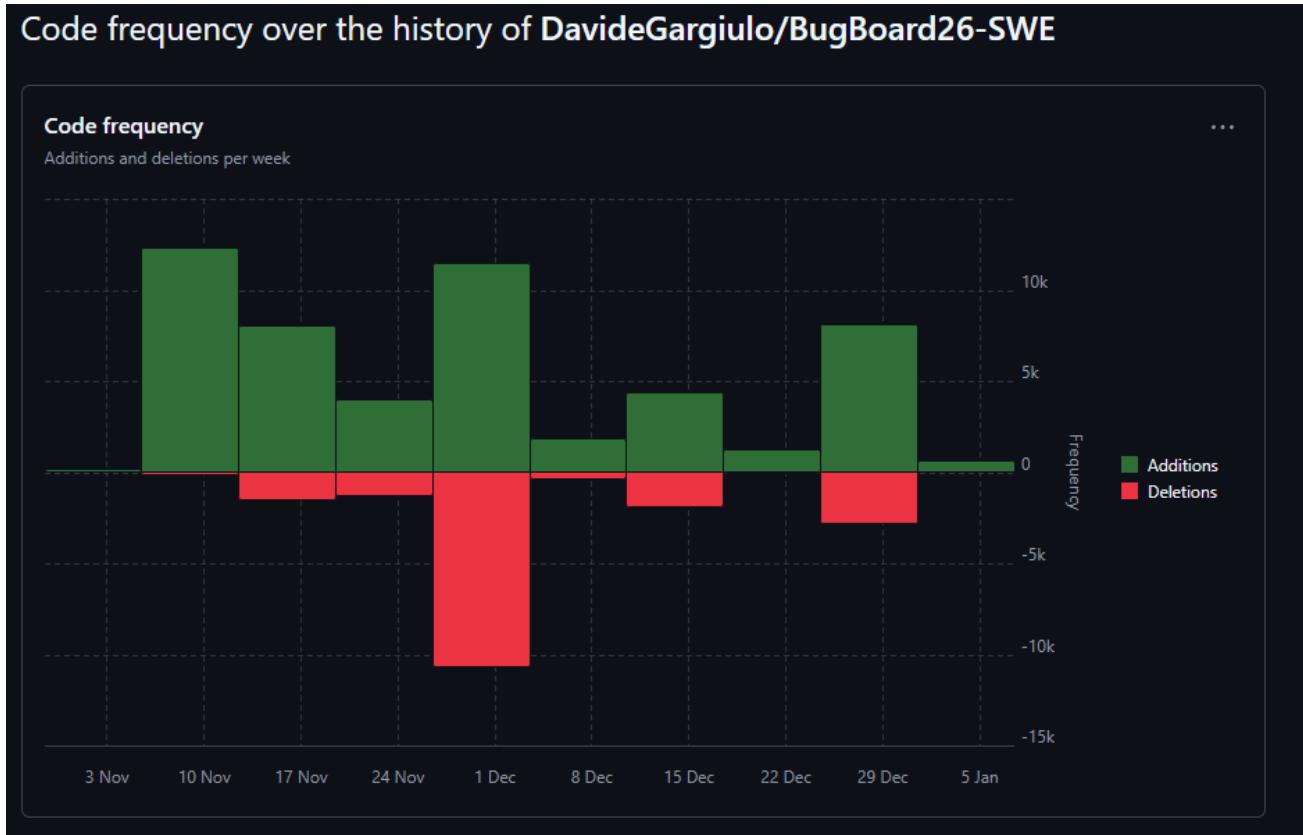


Figure 3.2: Grafico Code Frequency (Righe aggiunte vs cancellate).

3.8.3.3 Contributori

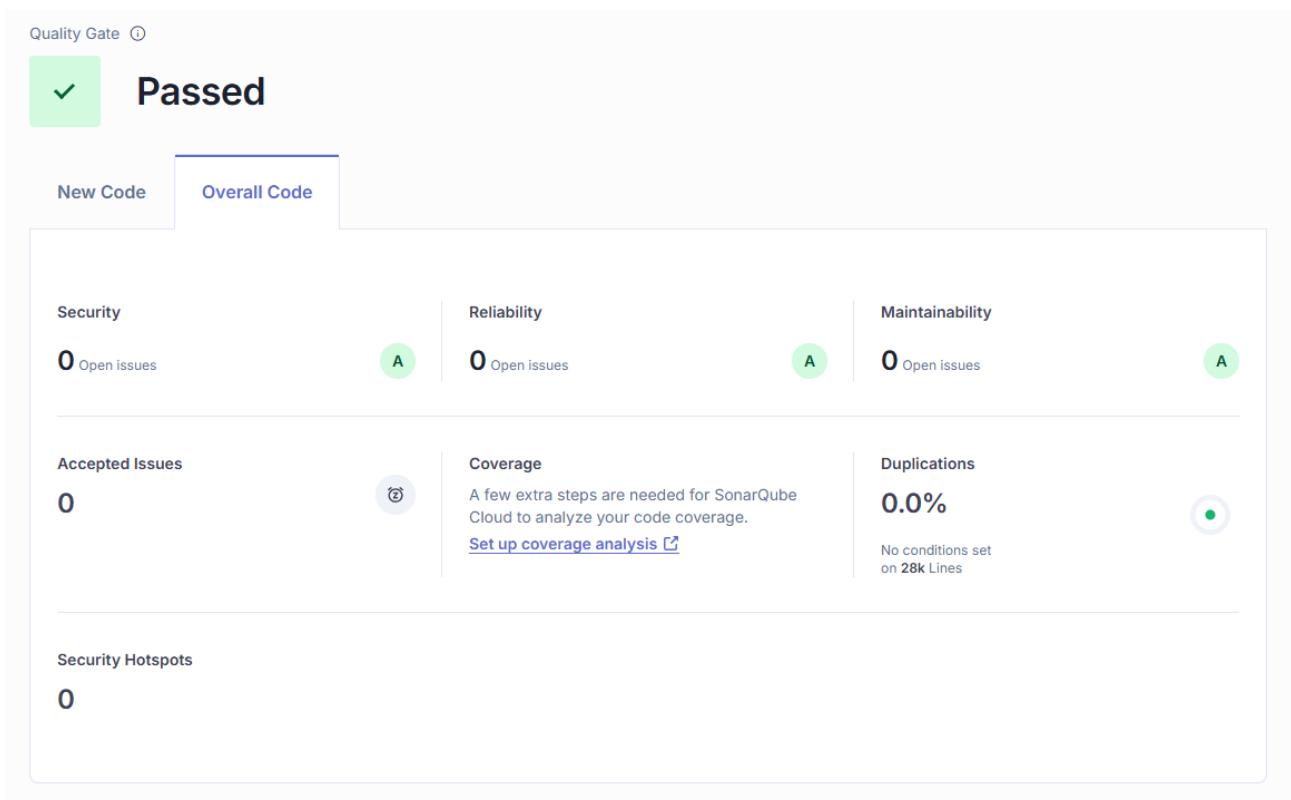
Di seguito è riportato il grafico dei contributori, che attesta la partecipazione attiva dei membri del team allo sviluppo del codice.



Figure 3.3: Statistiche dei contributori.

3.9

Qualità del Codice



3.9.1 | Documentazione

Oltre a questa documentazione tecnica, il codice include commenti dove la logica non è immediatamente ovvia.

IV

Attività di Testing

4.1

Test Plan: Creazione di una Nuova Issue

4.1.1 | Scopo e Obiettivi

L'obiettivo di questo piano di test è verificare la correttezza funzionale e la robustezza della funzionalità di "Creazione di una nuova Issue" (Use Case #02). Il test mira a garantire che il sistema accetti dati validi, gestisca correttamente gli errori di validazione e persista le informazioni nel database relazionale.

4.1.2 | Oggetto del Test (Test Items)

Le componenti software oggetto di test sono:

- **Controller:** `issueController.js` (metodo `createIssue`).
- **Modello Dati:** `Issue` (validazione schema Sequelize).
- **API Endpoint:** `POST /api/issues`.

4.1.3 | Strategia di Testing

Verranno adottate due strategie complementari:

- **Unit Testing (White-box):** Verifica isolata della logica del controller e delle regole di validazione del database.
- **Integration Testing (Black-box):** Verifica della risposta dell'API HTTP simulando richieste client complete.

Strumenti utilizzati:

- **Vitest:** Framework per l'esecuzione dei test runner.
- **Mock della request:** Per simulare le chiamate HTTP e verificare gli status code.

4.1.4 | Criteri di Accettazione

La funzionalità è considerata validata se:

1. Tutte le Issue create con dati validi vengono salvate nel DB con stato iniziale 'TODO'.
2. Tentativi di creazione con campi obbligatori mancanti (titolo) restituiscono errore 400.
3. Tentativi di creazione con dati non conformi ai tipi enumerativi vengono rifiutati.

4.1.5 | Casi di Test (Test Cases)

La seguente tabella definisce i casi di test identificati per coprire le classi di equivalenza dei dati di input.

ID Test	Descrizione Scenario e Risultato Atteso
TC01	Utente non trovato nel DB locale. <i>Input:</i> Token valido ma utente non sincronizzato su DB locale. <i>Output:</i> Status 400 con messaggio "Utente non trovato nel DB locale".
TC02	Validazione Campi Obbligatori. <i>Input:</i> Body della richiesta con campi vuoti (titolo, descrizione, tipo, progetto o priorità). <i>Output:</i> Status 400 con lista dei campi mancanti.
TC03	Progetto inesistente. <i>Input:</i> Tentativo di creare una issue per un progetto ('Progetto Alpha') non presente nel database. <i>Output:</i> Status 404 con messaggio "Progetto non trovato".
TC04	Creazione con successo (Senza Allegati). <i>Input:</i> Dati validi, nessun file caricato. <i>Output:</i> Status 201, Issue salvata con stato 'TODO', contatore allegati a 0.
TC05	Creazione con successo (Con Allegati). <i>Input:</i> Dati validi e file PDF caricato. <i>Output:</i> Status 201, Issue salvata e allegato registrato con hash SHA256 corretto.
TC06	Gestione Errore e Rollback (Cleanup). <i>Input:</i> File caricato fisicamente ma fallimento successivo nell'inserimento DB (Issue.create). <i>Output:</i> Status 500, messaggio di errore e rimozione automatica del file orfano dal disco.
TC07	Resilienza Errore Cleanup. <i>Input:</i> Fallimento DB seguito da fallimento nella rimozione del file. <i>Output:</i> Status 500 (l'errore originale viene restituito correttamente senza causare crash del server).

Codice per test di unità automatici

4.2.1 | CreateIssue.test.ts

```

1 import { describe, it, expect, vi, beforeEach } from 'vitest';
2 import { createIssue } from '../controllers/issueController.js';
3 import { Utente, Issue, Allegato, Progetto } from '../data/remote/Database.js';
4
5 const fsMocks = vi.hoisted(() => ({
6   unlink: vi.fn(),
7   readFile: vi.fn()
8 }));
9
10 vi.mock('node:fs', () => ({
11   promises: fsMocks,
12   default: { ...fsMocks, promises: fsMocks }
13 }));
14
15 vi.mock('fs/promises', () => ({
16   ...fsMocks,
17   default: fsMocks
18 }));
19
20 vi.mock('../data/remote/Database.js', () => ({
21   Utente: { findOne: vi.fn() },
22   Progetto: { findOne: vi.fn() },
23   Issue: { create: vi.fn() },
24   Allegato: { create: vi.fn() }
25 }));
26
27 vi.mock('crypto', () => ({
28   default: {
29     createHash: vi.fn().mockReturnThis(),
30     update: vi.fn().mockReturnThis(),
31     digest: vi.fn().mockReturnValue('mocked-hash-123')
32   },
33   createHash: vi.fn().mockReturnThis()
34 }));
35
36 describe('createIssue', () => {
37   let req, res;
38
39   beforeEach(() => {
40     vi.clearAllMocks();

```

```
41
42     req = {
43         user: { sub: 'keycloak-uuid-123' },
44         body: {
45             titolo: 'Titolo Issue',
46             descrizione: 'Descrizione Issue',
47             tipo: 'Bug',
48             progetto: 'Progetto Alpha',
49             priorita: 'Alta'
50         },
51         files: []
52     };
53
54     res = {
55         status: vi.fn().mockReturnThis(),
56         json: vi.fn().mockReturnThis()
57     };
58 });
59
60 it('dovrebbe restituire 400 se l\'utente non esiste nel DB locale',
61     async () => {
62     Utente.findOne.mockResolvedValue(null);
63     await createIssue(req, res);
64     expect(Utente.findOne).toHaveBeenCalledWith({
65         where: { keycloak_id: 'keycloak-uuid-123' } });
66     expect(res.status).toHaveBeenCalledWith(400);
67     expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
68         message: "Utente non trovato nel DB locale"
69     }));
70 });
71
72 it('dovrebbe restituire 400 se manca il titolo obbligatorio', async () => {
73     Utente.findOne.mockResolvedValue({ id: 1 });
74     req.body.titolo = '';
75     await createIssue(req, res);
76
77     expect(res.status).toHaveBeenCalledWith(400);
78     expect(res.json).toHaveBeenCalledWith({
79         message: 'Campi obbligatori mancanti',
80         required: ['titolo', 'descrizione', 'tipo', 'priorita', 'progetto']
81     });
82 });
83
84 it('dovrebbe restituire 400 se manca la descrizione obbligatoria',
85     async () => {
86     Utente.findOne.mockResolvedValue({ id: 1 });
```

```
87 req.body.descrizione = '';
88 await createIssue(req, res);
89
90 expect(res.status).toHaveBeenCalledWith(400);
91 expect(res.json).toHaveBeenCalledWith({
92   message: 'Campi obbligatori mancanti',
93   required: ['titolo', 'descrizione', 'tipo', 'priorita', 'progetto']
94 });
95 });
96
97 it('dovrebbe restituire 400 se manca il tipo obbligatorio', async () => {
98   Utente.findOne.mockResolvedValue({ id: 1 });
99
100  req.body.tipo = '';
101
102  await createIssue(req, res);
103
104  expect(res.status).toHaveBeenCalledWith(400);
105  expect(res.json).toHaveBeenCalledWith({
106    message: 'Campi obbligatori mancanti',
107    required: ['titolo', 'descrizione', 'tipo', 'priorita', 'progetto']
108  });
109 });
110
111 it('dovrebbe restituire 400 se manca il progetto obbligatorio', async () => {
112   Utente.findOne.mockResolvedValue({ id: 1 });
113
114  req.body.progetto = '';
115
116  await createIssue(req, res);
117
118  expect(res.status).toHaveBeenCalledWith(400);
119  expect(res.json).toHaveBeenCalledWith({
120    message: 'Campi obbligatori mancanti',
121    required: ['titolo', 'descrizione', 'tipo', 'priorita', 'progetto']
122  });
123 });
124
125 it('dovrebbe restituire 400 se manca la priorita obbligatoria', async () => {
126   Utente.findOne.mockResolvedValue({ id: 1 });
127
128  req.body.priorita = '';
129
130  await createIssue(req, res);
131
132  expect(res.status).toHaveBeenCalledWith(400);
```

```
133 expect(res.json).toHaveBeenCalledWith({
134   message: 'Campi obbligatori mancanti',
135   required: ['titolo', 'descrizione', 'tipo', 'priorita', 'progetto']
136 });
137 });
138
139 it('dovrebbe restituire 404 se il progetto non viene trovato', async () => {
140   Utente.findOne.mockResolvedValue({ id: 1 });
141   Progetto.findOne.mockResolvedValue(null);
142
143   await createIssue(req, res);
144
145   expect(Progetto.findOne).toHaveBeenCalledWith({
146     where: { nome: 'Progetto Alpha' } });
147   expect(res.status).toHaveBeenCalledWith(404);
148   expect(res.json).toHaveBeenCalledWith({ message: 'Progetto non trovato' });
149 });
150
151 it('dovrebbe creare una issue con successo senza allegati', async () => {
152   const mockUser = { id: 10 };
153   const mockProject = { id: 50, nome: 'Progetto Alpha' };
154   const mockCreatedIssue = {
155     id: 99,
156     titolo: req.body.titolo,
157     toJSON: () => ({ id: 99 })
158   };
159
160   Utente.findOne.mockResolvedValue(mockUser);
161   Progetto.findOne.mockResolvedValue(mockProject);
162   Issue.create.mockResolvedValue(mockCreatedIssue);
163
164   await createIssue(req, res);
165
166   expect(Issue.create).toHaveBeenCalledWith({
167     titolo: 'Titolo Issue',
168     descrizione: 'Descrizione Issue',
169     tipo: 'Bug',
170     stato: 'TODO',
171     priorita: 'Alta',
172     id_creatore: 10,
173     id_progetto: 50
174   });
175
176   expect(res.status).toHaveBeenCalledWith(201);
177   expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
178     message: 'Issue creata con successo',
```

```
179     issue: mockCreatedIssue,
180     allegati: 0
181   });
182 });
183
184 it('dovrebbe creare una issue con successo con allegati', async () => {
185   const mockUser = { id: 10 };
186   const mockProject = { id: 50 };
187   const mockCreatedIssue = { id: 99 };
188
189   req.files = [
190     {
191       originalname: 'test.pdf',
192       filename: 'storage-123.pdf',
193       path: 'uploads/storage-123.pdf',
194       mimetype: 'application/pdf',
195       size: 500
196     }
197   ];
198
199   Utente.findOne.mockResolvedValue(mockUser);
200   Progetto.findOne.mockResolvedValue(mockProject);
201   Issue.create.mockResolvedValue(mockCreatedIssue);
202   fsMocks.readFile.mockResolvedValue(Buffer.from('file-content'));
203   Allegato.create.mockImplementation(data =>
204     Promise.resolve({ ...data, id: 777 }));
205
206   await createIssue(req, res);
207   expect(fsMocks.readFile).toHaveBeenCalledWith('uploads/storage-123.pdf');
208
209   expect(Allegato.create).toHaveBeenCalledWith(expect.objectContaining({
210     nome_file_originale: 'test.pdf',
211     nome_file_storage: 'storage-123.pdf',
212     hash_sha256: 'mocked-hash-123',
213     id_issue: 99
214   ));
215
216   expect(res.status).toHaveBeenCalledWith(201);
217   expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
218     allegati: 1,
219     allegatiDettagli: expect.arrayContaining([
220       expect.objectContaining({ id: 777, nome_originale: 'test.pdf' })
221     ])
222   ));
223 });

224
```

```

225 it('dovrebbe restituire 500 ed eliminare i file se si verifica un errore
226   durante la creazione', async () => {
227   req.files = [{ path: 'uploads/temp.jpg' }];
228
229   Utente.findOne.mockResolvedValue({ id: 1 });
230   Progetto.findOne.mockResolvedValue({ id: 1 });
231   Issue.create.mockRejectedValue(new Error('DB Insert Failed'));
232
233   fsMocks.unlink.mockResolvedValue();
234   const consoleSpy = vi.spyOn(console, 'error').mockImplementation(() => {});
235   await createIssue(req, res);
236
237   expect(fsMocks.unlink).toHaveBeenCalledWith('uploads/temp.jpg');
238   expect(res.status).toHaveBeenCalledWith(500);
239   expect(res.json).toHaveBeenCalledWith({
240     message: 'Errore nella creazione della issue',
241     error: 'DB Insert Failed'
242   });
243
244   consoleSpy.mockRestore();
245 });
246
247 it('dovrebbe gestire il fallimento della pulizia dei file (unlink)
248   senza crashare', async () => {
249   req.files = [{ path: 'uploads/temp.jpg' }];
250
251   Utente.findOne.mockResolvedValue({ id: 1 });
252   Progetto.findOne.mockResolvedValue({ id: 1 });
253   Issue.create.mockRejectedValue(new Error('Errore Iniziale'));
254
255   fsMocks.unlink.mockRejectedValue(new Error('Impossibile eliminare file'));
256   const consoleSpy = vi.spyOn(console, 'error').mockImplementation(() => {});
257   await createIssue(req, res);
258
259   // Deve comunque restituire 500 per l'errore originale
260   expect(res.status).toHaveBeenCalledWith(500);
261   expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
262     message: 'Errore nella creazione della issue',
263     error: 'Errore Iniziale'
264   }));
265   consoleSpy.mockRestore();
266 });
267 });

```

Listing 4.1: ES6 (ECMAScript-2015) Listing

4.2.2 | createComment.test.js

```

1 import { describe, it, expect, vi, beforeEach } from 'vitest';
2 import { createComment } from '../controllers/commentController.js';
3 import { Commento, Allegato, Utente, Issue } from '../data/remote/Database.js';
4
5 // 1. Setup Mock per FS
6 const fsMocks = vi.hoisted(() => ({
7   unlink: vi.fn(),
8   readFile: vi.fn()
9 }));
10
11 // Mock 'node:fs'
12 vi.mock('node:fs', () => ({
13   default: fsMocks
14 }));
15
16 // Mock 'node:util' per bypassare la logica dei callback di promisify
17 vi.mock('node:util', () => ({
18   promisify: (fn) => fn
19 }));
20
21 // 2. Mock Database
22 vi.mock('../data/remote/Database.js', () => ({
23   Utente: { findOne: vi.fn() },
24   Issue: { findByPk: vi.fn() },
25   Commento: { create: vi.fn() },
26   Allegato: { create: vi.fn() }
27 }));
28
29 // 3. Mock Crypto
30 vi.mock('crypto', () => ({
31   default: {
32     createHash: vi.fn().mockReturnThis(),
33     update: vi.fn().mockReturnThis(),
34     digest: vi.fn().mockReturnValue('mocked-hash-comment')
35   },
36   createHash: vi.fn().mockReturnThis()
37 }));
38
39 describe('createComment', () => {
40   let req, res;
41
42   beforeEach(() => {
43     vi.clearAllMocks();
44

```

```
45 // Setup base request
46 req = {
47   user: { sub: 'keycloak-uuid-123' },
48   body: {
49     testo: 'Questo è un commento di test',
50     id_issue: '10'
51   },
52   files: []
53 };
54
55 // Setup response
56 res = {
57   status: vi.fn().mockReturnThis(),
58   json: vi.fn().mockReturnThis()
59 };
60 );
61
62 // --- Test di Validazione e Autenticazione ---
63 it('dovrebbe restituire 500 se l\'utente non è autenticato
64   (req.user mancante)', async () => {
65   req.user = null;
66
67   await createComment(req, res);
68
69   expect(res.status).toHaveBeenCalledWith(500);
70   expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
71     success: false,
72     message: expect.stringContaining('Utente non autenticato')
73   }));
74 );
75
76 it('dovrebbe restituire 500 se mancano campi obbligatori (testo)',
77   async () => {
78   req.body.testo = '';
79   await createComment(req, res);
80
81   expect(res.status).toHaveBeenCalledWith(500);
82   expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
83     success: false,
84     message: expect.stringContaining("Campi obbligatori mancanti")
85   }));
86 );
87
88 it('dovrebbe restituire 500 se mancano campi obbligatori (id_issue)',
89   async () => {
90   req.body.id_issue = '';
```

```
91  await createComment(req, res);
92
93  expect(res.status).toHaveBeenCalledWith(500);
94  expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
95    success: false,
96    message: expect.stringContaining("Campi obbligatori mancanti")
97  }));
98}
99
100 // --- Test Logica di Business (Utente/Issue non trovati) ---
101
102 it('dovrebbe restituire 500 se l\'utente non viene trovato nel DB locale',
103   async () => {
104   Utente.findOne.mockResolvedValue(null);
105   await createComment(req, res);
106
107   expect(Utente.findOne).toHaveBeenCalled({
108     where: { keycloak_id: 'keycloak-uuid-123' } });
109   expect(res.status).toHaveBeenCalledWith(500);
110   expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
111     message: "Utente non trovato nel database locale."
112   }));
113 }
114
115 it('dovrebbe restituire 500 se l\'issue non viene trovata', async () => {
116   Utente.findOne.mockResolvedValue({ id: 1 });
117   Issue.findByPk.mockResolvedValue(null);
118   await createComment(req, res);
119
120   expect(Issue.findByPk).toHaveBeenCalledWith('10');
121   expect(res.status).toHaveBeenCalledWith(500);
122   expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
123     message: "Issue non trovata."
124   }));
125 }
126
127 // --- Test Successo ---
128
129 it('dovrebbe creare un commento con successo senza allegati', async () => {
130   const mockUser = { id: 5 };
131   const mockIssue = { id: 10 };
132   const mockComment = {
133     id: 100,
134     testo: 'Testo',
135     toJSON: () => ({ id: 100, testo: 'Testo' })
136   };

```

```
137
138 Utente.findOne.mockResolvedValue(mockUser);
139 Issue.findByPk.mockResolvedValue(mockIssue);
140 Commento.create.mockResolvedValue(mockComment);
141 await createComment(req, res);
142
143 expect(Commento.create).toHaveBeenCalledWith({
144   testo: 'Questo è un commento di test',
145   id_utente: 5,
146   id_issue: 10 // Verifica conversione parseInt
147 });
148
149 expect(res.status).toHaveBeenCalledWith(201);
150 expect(res.json).toHaveBeenCalledWith({
151   success: true,
152   message: 'Commento creato con successo',
153   data: {
154     id: 100,
155     testo: 'Testo',
156     allegati: []
157   }
158 });
159 });
160
161 it('dovrebbe creare un commento con successo con allegati', async () => {
162   const mockUser = { id: 5 };
163   const mockIssue = { id: 10 };
164   const mockComment = { id: 100, toJSON: () => ({ id: 100 }) };
165
166   req.files = [
167     {
168       originalname: 'img.png',
169       filename: 'hash123.png',
170       path: 'uploads/hash123.png',
171       mimetype: 'image/png',
172       size: 1024
173     }
174   ];
175
176   Utente.findOne.mockResolvedValue(mockUser);
177   Issue.findByPk.mockResolvedValue(mockIssue);
178   Commento.create.mockResolvedValue(mockComment);
179
180   // Poiché abbiamo mockato promises per ritornare la funzione stessa,
181   // mockiamo readFile come una funzione che ritorna una Promise (async)
182   fsMocks.readFile.mockResolvedValue(Buffer.from('fake-image-data'));
```

```
183
184 Allegato.create.mockImplementation(data => Promise.resolve({
185   ...data, id: 500 }));
186
187 await createComment(req, res);
188
189 // Verifiche
190 expect(fsMocks.readFile).toHaveBeenCalledWith('uploads/hash123.png');
191 expect(Allegato.create).toHaveBeenCalledWith(expect.objectContaining({
192   nome_file_originale: 'img.png',
193   hash_sha256: 'mocked-hash-comment',
194   id_commento: 100,
195   id_issue: null
196 }));
197
198 expect(res.status).toHaveBeenCalledWith(201);
199 expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
200   success: true,
201   data: expect.objectContaining({
202     allegati: expect.arrayContaining([expect.objectContaining({
203       id: 500 })])
204   })
205 }));
206 });
207
208 // --- Test Gestione Errori e Pulizia (Cleanup) ---
209
210 it('dovrebbe eliminare i file caricati se si verifica un errore
211   (es. DB fallisce)', async () => {
212   req.files = [{ path: 'uploads/to-delete.jpg' }];
213
214   Utente.findOne.mockResolvedValue({ id: 5 });
215   Issue.findByPk.mockResolvedValue({ id: 10 });
216
217   // Simuliamo errore durante la creazione del commento
218   Commento.create.mockRejectedValue(new Error('DB Error'));
219   fsMocks.unlink.mockResolvedValue(); // Unlink ok
220
221   // Silenzia console.error
222   const consoleSpy = vi.spyOn(console, 'error').mockImplementation(() => {});
223   await createComment(req, res);
224   expect(fsMocks.unlink).toHaveBeenCalledWith('uploads/to-delete.jpg');
225
226   expect(res.status).toHaveBeenCalledWith(500);
227   expect(res.json).toHaveBeenCalledWith({
228     success: false,
```

```

229     message: 'DB Error'
230   );
231
232   consoleSpy.mockRestore();
233 });
234
235 it('dovrebbe gestire l\'errore di req.files undefined', async () => {
236   req.files = undefined; // Simuliamo caso limite
237
238   Utente.findOne.mockResolvedValue({ id: 5 });
239   Issue.findByPk.mockResolvedValue({ id: 10 });
240   const mockComment = { id: 100, toJSON: () => ({} ) };
241   Commento.create.mockResolvedValue(mockComment);
242
243   await createComment(req, res);
244
245   expect(res.status).toHaveBeenCalledWith(201);
246   expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
247     success: true
248   }));
249 });
250
251 it('dovrebbe non crashare se la pulizia (unlink) fallisce', async () => {
252   req.files = [{ path: 'uploads/fail-delete.jpg' }];
253   Utente.findOne.mockRejectedValue(new Error('Errore Iniziale'));
254
255   // Unlink fallisce
256   fsMocks.unlink.mockRejectedValue(new Error('Unlink fallito'));
257   const consoleSpy = vi.spyOn(console, 'error').mockImplementation(() => {});
258   await createComment(req, res);
259
260   // Deve loggare l'errore di pulizia ma rispondere con l'errore originale
261   expect(res.status).toHaveBeenCalledWith(500);
262   expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
263     message: 'Errore Iniziale'
264   }));
265
266   // Verifica che console.error sia stato chiamato per l'unlink
267   expect(consoleSpy).toHaveBeenCalledWith('Errore pulizia file:',
268     expect.any(Error));
269
270   consoleSpy.mockRestore();
271 });
272 });

```

Listing 4.2: ES6 (ECMAScript-2015) Listing

4.2.3 | jwt.test.js

```
1 import { describe, it, expect, vi, beforeEach } from "vitest";
2 import { protect } from "../middleware/authMiddleware.js";
3
4 // Mock delle dipendenze
5 vi.mock('jsonwebtoken');
6 vi.mock('jwks-rsa');
7 vi.mock('axios');
8
9 import jwt from 'jsonwebtoken';
10 import jwksClient from 'jwks-rsa';
11 import axios from 'axios';
12
13 describe("Controllo token di sessione", () => {
14     let req, res, next;
15
16     beforeEach(() => {
17         // Reset dei mock
18         vi.clearAllMocks();
19
20         // Setup request/response mock
21         req = {
22             headers: {},
23             cookies: {},
24             accessToken: null
25         };
26
27         res = {
28             status: vi.fn().mockReturnThis(),
29             json: vi.fn().mockReturnThis(),
30             cookie: vi.fn().mockReturnThis(),
31             clearCookie: vi.fn().mockReturnThis()
32         };
33
34         next = vi.fn();
35
36         // Mock jwksClient
37         jwksClient.mockReturnValue({
38             getSigningKey: vi.fn((kid, callback) => {
39                 callback(null, {
40                     getPublicKey: () => 'mocked-public-key'
41                 });
42             })
43         });
44     });
45 }
```

```
45
46 it("access_token presente e valido", async () => {
47   const mockDecoded = {
48     sub: 'user-123',
49     email: 'test@example.com',
50     preferred_username: 'testuser',
51     name: 'Test User',
52     exp: Math.floor(Date.now() / 1000) + 3600,
53     realm_access: { roles: ['user'] }
54   };
55
56   // Mock jwt.verify per restituire un token valido
57   jwt.verify.mockImplementation((token, getKey, options, callback) => {
58     callback(null, mockDecoded);
59   });
60
61   req.cookies['access_token'] = 'valid-token-here';
62
63   await protect(req, res, next);
64
65   expect(next).toHaveBeenCalled();
66   expect(req.user).toEqual(expect.objectContaining({
67     id: 'user-123',
68     email: 'test@example.com',
69     username: 'testuser'
70   }));
71 });
72
73 it("access_token mancante, refresh_token presente", async () => {
74   const mockDecoded = {
75     sub: 'user-456',
76     email: 'refresh@example.com',
77     preferred_username: 'refreshuser',
78     exp: Math.floor(Date.now() / 1000) + 3600,
79     realm_access: { roles: ['user'] }
80   };
81
82   // Mock refresh token success
83   axios.post.mockResolvedValueOnce({
84     data: {
85       access_token: 'new-access-token',
86       refresh_token: 'new-refresh-token',
87       expires_in: 300
88     }
89   });
90 });
```

```
91 // Mock jwt.verify per il nuovo token
92 jwt.verify.mockImplementation((token, getKey, options, callback) => {
93   callback(null, mockDecoded);
94 });
95
96 req.cookies['refresh_token'] = 'valid-refresh-token';
97
98 await protect(req, res, next);
99
100 expect(axios.post).toHaveBeenCalled();
101 expect(res.cookie).toHaveBeenCalledWith('access_token', 'new-access-token',
102   expect.any(Object));
103 expect(next).toHaveBeenCalled();
104 expect(req.user.id).toBe('user-456');
105 });
106
107 it("access_token mancante, refresh_token mancante", async () => {
108   // Nessun token nei cookies
109   req.cookies = {};
110
111   await protect(req, res, next);
112
113   expect(res.status).toHaveBeenCalledWith(401);
114   expect(res.json).toHaveBeenCalledWith({
115     error: 'Autenticazione richiesta' });
116   expect(next).not.toHaveBeenCalled();
117 });
118
119 it("access_token scaduto, refresh_token valido", async () => {
120   const expiredError = new Error('Token expired');
121   expiredError.name = 'TokenExpiredError';
122
123   const mockDecoded = {
124     sub: 'user-789',
125     email: 'expired@example.com',
126     preferred_username: 'expireduser',
127     exp: Math.floor(Date.now() / 1000) + 3600,
128     realm_access: { roles: ['user'] }
129   };
130
131   // Prima chiamata: token scaduto
132   jwt.verify.mockImplementationOnce((token, getKey, options, callback) => {
133     callback(expiredError, null);
134   });
135
136   // Seconda chiamata (dopo refresh): token valido
```

```
137  jwt.verify.mockImplementationOnce((token, getKey, options, callback) => {
138    callback(null, mockDecoded);
139  });
140
141  // Mock refresh success
142  axios.post.mockResolvedValueOnce({
143    data: {
144      access_token: 'refreshed-token',
145      refresh_token: 'new-refresh-token',
146      expires_in: 300
147    }
148  });
149
150  req.cookies['access_token'] = 'expired-token';
151  req.cookies['refresh_token'] = 'valid-refresh-token';
152
153  await protect(req, res, next);
154
155  expect(axios.post).toHaveBeenCalled();
156  expect(next).toHaveBeenCalled();
157  expect(req.user.id).toBe('user-789');
158 });
159});
```

Listing 4.3: ES6 (ECMAScript-2015) Listing

4.2.4 | updateIssue.test.js

```
1 import { describe, it, expect, vi, beforeEach } from 'vitest';
2 import { updateIssue } from '../controllers/issueController.js';
3 import { Issue, Allegato } from '../data/remote/Database.js';
4 import crypto from 'node:crypto';
5
6 const fsMocks = vi.hoisted(() => ({
7   unlink: vi.fn(),
8   readFile: vi.fn()
9 }));
10
11 vi.mock('node:fs', () => ({
12   promises: fsMocks,
13   default: {
14     ...fsMocks,
15     promises: fsMocks
16   }
17 }));
18
19 vi.mock('fs/promises', () => ({
20   ...fsMocks,
21   default: fsMocks
22 }));
23
24 vi.mock('../data/remote/Database.js', () => ({
25   Issue: {
26     findByPk: vi.fn()
27   },
28   Allegato: {
29     count: vi.fn(),
30     create: vi.fn()
31   }
32 }));
33
34 vi.mock('crypto', () => ({
35   default: {
36     createHash: vi.fn().mockReturnThis(),
37     update: vi.fn().mockReturnThis(),
38     digest: vi.fn().mockReturnValue('mocked-hash')
39   },
40   createHash: vi.fn().mockReturnThis()
41 }));
42
43 describe('Test metodo UpdateIssue', () => {
44   let req, res;
```

```
45
46 beforeEach(() => {
47   vi.clearAllMocks();
48
49   req = {
50     params: { id: '1' },
51     body: {},
52     files: null
53   };
54
55   res = {
56     status: vi.fn().mockReturnThis(),
57     json: vi.fn().mockReturnThis()
58   };
59 });
60
61 it('dovrebbe restituire errore 400 se la descrizione è vuota', async () => {
62   req.body = { descrizione: '' };
63
64   await updateIssue(req, res);
65
66   expect(res.status).toHaveBeenCalledWith(400);
67   expect(res.json).toHaveBeenCalledWith({
68     message: "Devi fornire almeno una descrizione per aggiornare l'issue"
69   });
70 });
71
72 it('dovrebbe restituire errore 400 se la descrizione contiene solo spazi', async () => {
73   req.body = { descrizione: ' ' };
74
75   await updateIssue(req, res);
76
77   expect(res.status).toHaveBeenCalledWith(400);
78   expect(res.json).toHaveBeenCalledWith({
79     message: "Devi fornire almeno una descrizione per aggiornare l'issue"
80   });
81 });
82
83 it('dovrebbe restituire errore 404 se l\'issue non viene trovata', async () => {
84   req.body = { descrizione: 'Nuova descrizione' };
85   Issue.findByPk.mockResolvedValue(null);
86
87   await updateIssue(req, res);
88
89
90
```

```
91  expect(Issue.findByPk).toHaveBeenCalledWith('1');
92  expect(res.status).toHaveBeenCalledWith(404);
93  expect(res.json).toHaveBeenCalledWith({ message: 'Issue non trovata' });
94 });
95
96 it('dovrebbe eliminare i file caricati se l\'issue non viene trovata',
97   async () => {
98   req.body = { descrizione: 'Nuova descrizione' };
99   req.files = [
100     { path: '/uploads/file1.jpg' },
101     { path: '/uploads/file2.jpg' }
102   ];
103
104   Issue.findByPk.mockResolvedValue(null);
105   fsMocks.unlink.mockResolvedValue();
106
107   await updateIssue(req, res);
108
109   expect(fsMocks.unlink).toHaveBeenCalledTimes(2);
110   expect(fsMocks.unlink).toHaveBeenCalledWith('/uploads/file1.jpg');
111   expect(fsMocks.unlink).toHaveBeenCalledWith('/uploads/file2.jpg');
112 });
113
114 it('dovrebbe restituire errore 400 se si supera il limite di 3 allegati',
115   async () => {
116   req.body = { descrizione: 'Nuova descrizione' };
117   req.files = [
118     { path: '/uploads/file1.jpg' },
119     { path: '/uploads/file2.jpg' }
120   ];
121
122   const mockIssue = { id: 1 };
123   Issue.findByPk.mockResolvedValue(mockIssue);
124   Allegato.count.mockResolvedValue(2);
125   fsMocks.unlink.mockResolvedValue();
126
127   await updateIssue(req, res);
128
129   expect(Allegato.count).toHaveBeenCalledWith({
130     where: {
131       id_issue: '1',
132       id_commento: null
133     }
134   });
135   expect(res.status).toHaveBeenCalledWith(400);
136   expect(res.json).toHaveBeenCalledWith({
```

```
137     message: "Limite superato. L'issue ha già 2 allegati e  
138     ne stai inviando 2. Il massimo totale consentito è 3."  
139   );  
140   expect(fsMocks.unlink).toHaveBeenCalledTimes(2);  
141 };  
142  
143 it('dovrebbe aggiornare l\'issue con successo con solo descrizione',  
144   async () => {  
145   req.body = { descrizione: 'Nuova descrizione' };  
146  
147   const mockIssue = {  
148     id: 1,  
149     descrizione: 'Descrizione esistente',  
150     update: vi.fn().mockResolvedValue()  
151   };  
152  
153   const mockUpdatedIssue = {  
154     id: 1,  
155     descrizione: 'Descrizione esistente<br><br><strong>22/12/2024,  
156       10:30:</strong><br>Nuova descrizione',  
157     allegati: []  
158   };  
159  
160   Issue.findByPk  
161     .mockResolvedValueOnce(mockIssue)  
162     .mockResolvedValueOnce(mockUpdatedIssue);  
163  
164   Allegato.count.mockResolvedValue(0);  
165  
166   vi.useFakeTimers();  
167   vi.setSystemTime(new Date('2024-12-22T10:30:00'));  
168  
169   await updateIssue(req, res);  
170  
171   expect(mockIssue.update).toHaveBeenCalled();  
172   expect(res.status).toHaveBeenCalledWith(200);  
173   expect(res.json).toHaveBeenCalledWith({  
174     message: 'Issue aggiornata con successo',  
175     issue: mockUpdatedIssue  
176   });  
177  
178   vi.useRealTimers();  
179 };  
180  
181 it('dovrebbe aggiornare l\'issue con descrizione e allegati', async () => {  
182   req.body = { descrizione: 'Nuova descrizione' };
```

```
183 req.files = [
184   {
185     originalname: 'test.jpg',
186     filename: 'test-123.jpg',
187     path: '/uploads/test-123.jpg',
188     mimetype: 'image/jpeg',
189     size: 1024
190   }
191 ];
192
193 const mockIssue = {
194   id: 1,
195   descrizione: 'Descrizione esistente',
196   update: vi.fn().mockResolvedValue()
197 };
198
199 const mockUpdatedIssue = {
200   id: 1,
201   descrizione: 'Descrizione aggiornata',
202   allegati: [{ id: 1, nome_file_originale: 'test.jpg' }]
203 };
204
205 Issue.findByPk
206   .mockResolvedValueOnce(mockIssue)
207   .mockResolvedValueOnce(mockUpdatedIssue);
208
209 Allegato.count.mockResolvedValue(0);
210 Allegato.create.mockResolvedValue({ id: 1 });
211
212 fsMocks.readFile.mockResolvedValue(Buffer.from('test'));
213
214 const mockHash = {
215   update: vi.fn().mockReturnThis(),
216   digest: vi.fn().mockReturnValue('abc123')
217 };
218 crypto.createHash.mockReturnValue(mockHash);
219
220 await updateIssue(req, res);
221
222 expect(Allegato.create).toHaveBeenCalledWith({
223   nome_file_originale: 'test.jpg',
224   nome_file_storage: 'test-123.jpg',
225   percorso_relativo: '/uploads/test-123.jpg',
226   tipo_mime: 'image/jpeg',
227   dimensione_byte: 1024,
228   hash_sha256: 'abc123',
```

```
229     id_issue: '1',
230     id_commento: null
231   );
232
233   expect(res.status).toHaveBeenCalledWith(200);
234   expect(res.json).toHaveBeenCalledWith({
235     message: 'Issue aggiornata con successo',
236     issue: mockUpdatedIssue
237   });
238 });
239
240 it('dovrebbe aggiungere l\'header con timestamp quando la descrizione
241   è vuota', async () => {
242   req.body = { descrizione: 'Prima descrizione' };
243
244   const mockIssue = {
245     id: 1,
246     descrizione: '',
247     update: vi.fn().mockResolvedValue()
248   };
249
250   Issue.findByPk.mockResolvedValue(mockIssue);
251   Allegato.count.mockResolvedValue(0);
252
253   vi.useFakeTimers();
254   vi.setSystemTime(new Date('2024-12-22T10:30:00'));
255
256   await updateIssue(req, res);
257
258   const expectedDescription = '<br><br><strong>22/12/2024,
259   10:30:</strong><br>Prima descrizione';
260   expect(mockIssue.update).toHaveBeenCalled();
261   expect(mockIssue.update).toHaveBeenCalledWith(
262     descrizione: expectedDescription
263   );
264
265   vi.useRealTimers();
266 });
267
268 it('dovrebbe gestire errori generici e pulire i file caricati', async () => {
269   const consoleSpy = vi.spyOn(console, 'error').mockImplementation(() => {});
270
271   req.body = { descrizione: 'Nuova descrizione' };
272   req.files = [{ path: '/uploads/file1.jpg' }];
273
274   Issue.findByPk.mockRejectedValue(new Error('Database error'));
275   fsMocks.unlink.mockResolvedValue();
```

```
275     await updateIssue(req, res);
276
277     expect(fsMocks.unlink).toHaveBeenCalledWith('/uploads/file1.jpg');
278     expect(res.status).toHaveBeenCalledWith(500);
279
280     consoleSpy.mockRestore();
281 });
282
283 it('dovrebbe gestire errori durante l\'eliminazione dei file', async () => {
284   req.body = { descrizione: 'Nuova descrizione' };
285   req.files = [{ path: '/uploads/file1.jpg' }];
286
287   Issue.findByPk.mockResolvedValue(null);
288   fsMocks.unlink.mockRejectedValue(new Error('Unlink error'));
289
290   await updateIssue(req, res);
291
292   expect(res.status).toHaveBeenCalledWith(404);
293 });
294
295 it('dovrebbe processare multipli file contemporaneamente', async () => {
296   req.body = { descrizione: 'Nuova descrizione' };
297   req.files = [
298     {
299       originalname: 'test1.jpg',
300       filename: 'test1-123.jpg',
301       path: '/uploads/test1-123.jpg',
302       mimetype: 'image/jpeg',
303       size: 1024
304     },
305     {
306       originalname: 'test2.pdf',
307       filename: 'test2-456.pdf',
308       path: '/uploads/test2-456.pdf',
309       mimetype: 'application/pdf',
310       size: 2048
311     }
312   ];
313
314   const mockIssue = {
315     id: 1,
316     descrizione: '',
317     update: vi.fn().mockResolvedValue()
318   };
319
320   Issue.findByPk.mockResolvedValue(mockIssue);
```

```
321 Allegato.count.mockResolvedValue(0);
322 Allegato.create.mockResolvedValue({ id: 1 });
323
324 fsMocks.readFile.mockResolvedValue(Buffer.from('test'));
325
326 const mockHash = {
327   update: vi.fn().mockReturnThis(),
328   digest: vi.fn().mockReturnValue('abc123')
329 };
330 crypto.createHash.mockReturnValue(mockHash);
331
332 await updateIssue(req, res);
333
334 expect(Allegato.create).toHaveBeenCalledTimes(2);
335 expect(fsMocks.readFile).toHaveBeenCalledTimes(2);
336 expect(res.status).toHaveBeenCalledWith(200);
337 });
338});
```

Listing 4.4: ES6 (ECMAScript-2015) Listing

Documentazione delle strategie di test

In questa sezione vengono dettagliate le metodologie, le classi di equivalenza e i criteri di copertura adottati per ciascuno dei moduli sottoposti a test unitario (descritti nella Sezione 4.2). La strategia generale ha privilegiato un approccio *White-box* per garantire la massima copertura dei rami decisionali (*Branch Coverage*), simulando (mocking) le dipendenze esterne come il Database e il File System per isolare la logica di business.

4.3.1 | Strategia per la Creazione Issue (createIssue)

Per il controller di creazione issue, la strategia si è focalizzata sulla validazione degli input e sulla gestione atomica delle risorse (file).

- **Classi di Equivalenza Individuate:**
 - *Contesto Utente*: Utente non sincronizzato su DB locale (Caso d'errore specifico).
 - *Input Obbligatori*: Set completo vs Set parziale (Test *Fail-Fast*).
 - *Integrità Referenziale*: Riferimento a Progetto inesistente.
 - *Gestione Allegati*: Nessun allegato (Caso base) vs Allegati presenti (Caso complesso).
- **Scenari Critici Coperti:** È stata posta particolare enfasi sulla verifica del meccanismo di **Rollback manuale** (Cleanup). I test TC06 e TC07 verificano che, in caso di errore logico dopo l'upload fisico del file, il sistema rimuova il file orfano, garantendo l'*Operational Safety* e prevenendo l'inconsistenza dello storage.

4.3.2 | Strategia per la Creazione Commento (createComment)

Il test della creazione commenti ha richiesto una strategia specifica per gestire la catena di dipendenze (Utente → Issue → Commento).

- **Classi di Equivalenza Individuate:**
 - *Autenticazione*: Richiesta priva di oggetto **user** (Middleware failure simulation).
 - *Relazioni*: Issue target inesistente vs Esistente.
 - *Payload*: Commento solo testo vs Commento con allegati (verifica polimorfismo tabella Allegati).
- **Criteri di Copertura:** Sono stati coperti i rami di errore annidati: fallimento ricerca utente, fallimento ricerca issue, e fallimento creazione commento. Anche qui, la strategia di mock del modulo **fs** ha permesso di verificare la pulizia dei file in caso di fallimento a cascata.

4.3.3 | Strategia per l'Aggiornamento Issue (updateIssue)

La logica di aggiornamento è intrinsecamente più complessa della creazione a causa della gestione dello stato preesistente e dei limiti di dominio.

- **Classi di Equivalenza Individuate:**
 - *Validità Descrizione:* Stringa vuota/spazi (Invalida) vs Testo valido.
 - *Stato Precedente:* Issue non trovata (404).
 - *Limiti di Business:* Numero allegati totali (Esistenti + Nuovi) > 3 vs ≤ 3 .
- **Logica di Business Testata:**
 - **Append-Only Logic:** È stato verificato che l'aggiornamento non sovrascriva la descrizione ma la accodi. La strategia di test ha incluso l'uso di *Fake Timers* (`vi.useFakeTimers`) per garantire che il timestamp generato nell'header della modifica fosse deterministico e verificabile.
 - **Vincoli Quantitativi:** Il test simula specificamente il conteggio degli allegati preesistenti per forzare l'errore di "superamento limite", coprendo un requisito funzionale specifico non testabile nella semplice creazione.

4.3.4 | Strategia per l'Autenticazione (Middleware JWT)

Per il middleware di autenticazione (`jwt.test.js`), la strategia è passata da test di logica di business a test di sicurezza e gestione stati.

- **Classi di Equivalenza Individuate:**
 - *Stato Token:* Access Token valido vs Scaduto vs Mancante.
 - *Meccanismo di Refresh:* Refresh Token valido (con Access scaduto) vs Refresh Token mancante.
- **Integrazione Simulata:** A differenza dei controller, qui è stato necessario mockare chiamate HTTP esterne (verso l'Identity Provider Keycloak tramite `axios`). La strategia ha coperto il flusso di "Auto-Refresh":
 1. Rilevamento token scaduto.
 2. Chiamata all'endpoint di refresh.
 3. Aggiornamento trasparente dei cookie nella risposta.

Questo garantisce che la sessione utente rimanga attiva senza richiedere interventi manuali, coprendo un requisito non funzionale di *Usabilità* e *Sicurezza*.