

UNIVERSITÀ DEGLI STUDI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE



CORSO DI LAUREA IN INFORMATICA

INSEGNAMENTO DI INGEGNERIA DEL SOFTWARE

ANNO ACCADEMICO 2025/2026

**Progettazione e sviluppo della piattaforma
BugBoard26**

Indice

1	INTRODUZIONE	2
1.1	Chi siamo	2
2	INGEGNERIA DEI REQUISITI	3
2.1	Casi d'uso	3
2.2	Individuazione delle personas	4
2.3	Requisiti non funzionali e di dominio	5
2.3.1	Requisiti non funzionali	5
2.3.2	Requisiti di dominio	5
2.4	Formalizzazione di un requisito	6
3	SYSTEM DESIGN	8
3.1	Architettura del sistema	8
3.2	Decomposizione del sistema	8
3.3	Scelta delle tecnologie	8
4	SOFTWARE DESIGN	9
4.1	Persistenza dei dati	9
4.2	Strumenti di versioning	10
4.3	Qualità del codice	10
5	TESTING	11
5.1	Test Plan	11
5.1.1	<code>ResponseEntity<IssueResponse> postNewIssue(UUID uuid_project, IssueRequest issueRequest)</code>	11
5.1.2	<code>ResponseEntity<List<IssueResponse>> getIssuesByProject(UUID uuid_project, string type, string priority, string state)</code>	12

1 INTRODUZIONE

1.1 Chi siamo

Benvenuto su **BugBoard26**!

BugBoard26 è una piattaforma di *issues handling* che fornisce una soluzione unica per:

- Dividere in modo facile developer in progetti.
- Segnalare e gestire intuitivamente issue di vario tipo.
- Gestire in modo efficiente tutte le persone coinvolte in un progetto (anche non sviluppatori) tramite una gerarchia di utenze.

Glossario

issue il "problema" identificato all'interno di un progetto che concerne l'utente.

piattaforma Vedi sistema.

sistema BugBoard26.

2 INGEGNERIA DEI REQUISITI

2.1 Casi d'uso

In questa sezione ci interesseremo all'individuazione dei casi d'uso. Come si può evincere dallo use case diagram riportato qui di seguito:

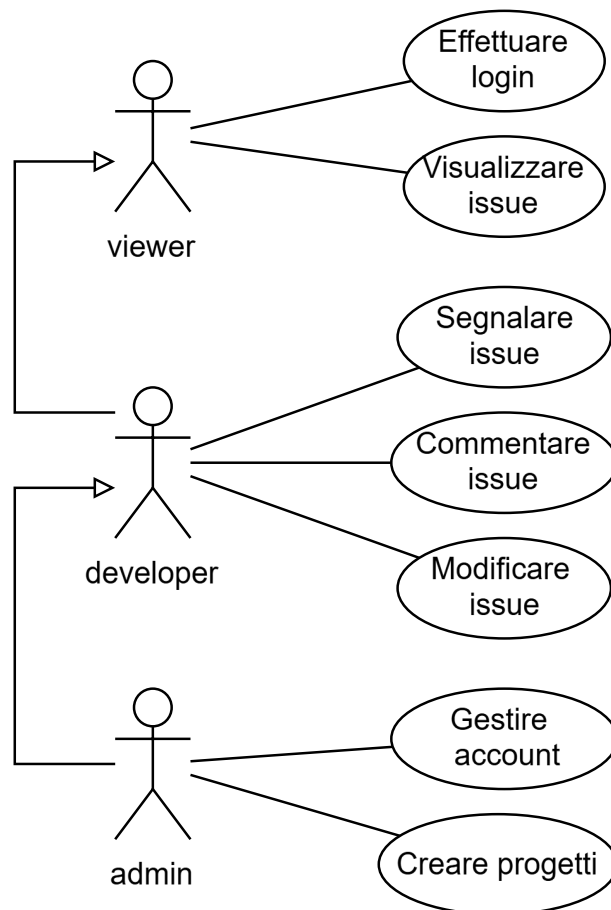


Figura 1: Use Case Diagram

Tutti gli utilizzatori della piattaforma possono essere divisi in tre grandi macrocategorie:

- **Viewer:** individuati anche nella figura di uno stakeholder. Sono utilizzatori, non necessariamente del settore, che hanno comunque interesse a visualizzare le issue legate al progetto senza poterle però aggiungere o modificare.
- **Developer:** rappresentano la stragrande maggioranza di utilizzatori della piattaforma. I developer sono coloro che contribuiscono attivamente all'individuazione e risoluzione delle issue.
- **Admin:** rappresentano l'estensione di un developer con permessi di creazione e gestione di altre utenze.

2.2 Individuazione delle personas

Ora esamineremo alcune personas che rispecchiano alcuni dei tipi di utilizzatori della nostra piattaforma.

Nome: Mark Party Età: 52 anni Posizione: Product Owner
Obiettivi: <ul style="list-style-type: none">• Sarebbe molto utile poter vedere l'andazzo del team così da sapere in che direzione indirizzarlo e come gestirlo.
Bio: <p>Sono un economista italo-americano di Boston. Nell'arco della mia carriera mi sono ritrovato a gestire diverse start-up e gruppi di lavoro, nonostante non capisca molto di queste diavolerie informatiche, mi ritengo molto più capace a gestire e portare avanti prodotti</p>

Nome: Aleksander Lilia Età: 24 anni Posizione: Developer
Obiettivi: <ul style="list-style-type: none">• Per lavorare in modo efficiente devo sapere quali problemi devo sistemare e magari avere del feedback dai miei colleghi.• Nel caso dovessi trovare dei problemi, vorrei avere un modo comodo per segnalarli in modo dettagliato.• Una volta risolti tali problemi vorrei poter segnalarlo al mio team.
Bio: <p>Sono un developer di Izdebki, dopo essermi laureato all'università di Cracovia mi sono trasferito a Napoli per lavoro e per amore. Sono grande amatore della filosofia "work smarter not harder" che cerco di applicare in ogni modo possibile.</p>

Nome: Pierrelouis Frascout

Età: 37 anni

Posizione: Team leader

Obiettivi:

- Voglio poter gestire i membri del mio team in modo chiaro ed efficiente.
- Voglio poter tenere traccia dei progressi fatti dal mio team e come si sta comportando.
- Voglio condividere con tutte le persone interessate, l'andamento del nostro team.

Bio: Sono un software engineer di Nantes ma ho vissuto buona parte della mia vita a Roma. Sono una persona risolutiva ed estremamente orientata al pratico e questo si riflette nel mio modo di lavorare

2.3 Requisiti non funzionali e di dominio

2.3.1 Requisiti non funzionali

I requisiti non funzionali da noi individuati sono:

- **Permanenza dei dati:** attraverso un database non MBaaS.
- **Utilizzo di linguaggi orientati agli oggetti.**
- **Implementazione di un modello Client-Server.**
- **Elevata manutenibilità.**
- **Efficienza e affidabilità:** non essendo la piattaforma safety-critical, limitazioni di tempo e memoria occupata sono da considerarsi standard e ragionevoli.

2.3.2 Requisiti di dominio

Non sono stati individuati requisiti di dominio particolarmente differenti da quelli forniti nella traccia.

2.4 Formalizzazione di un requisito

Qui di seguito riportiamo la formalizzazione di un requisito quale la visualizzazione di una issue, prima mediante il suo mockup:

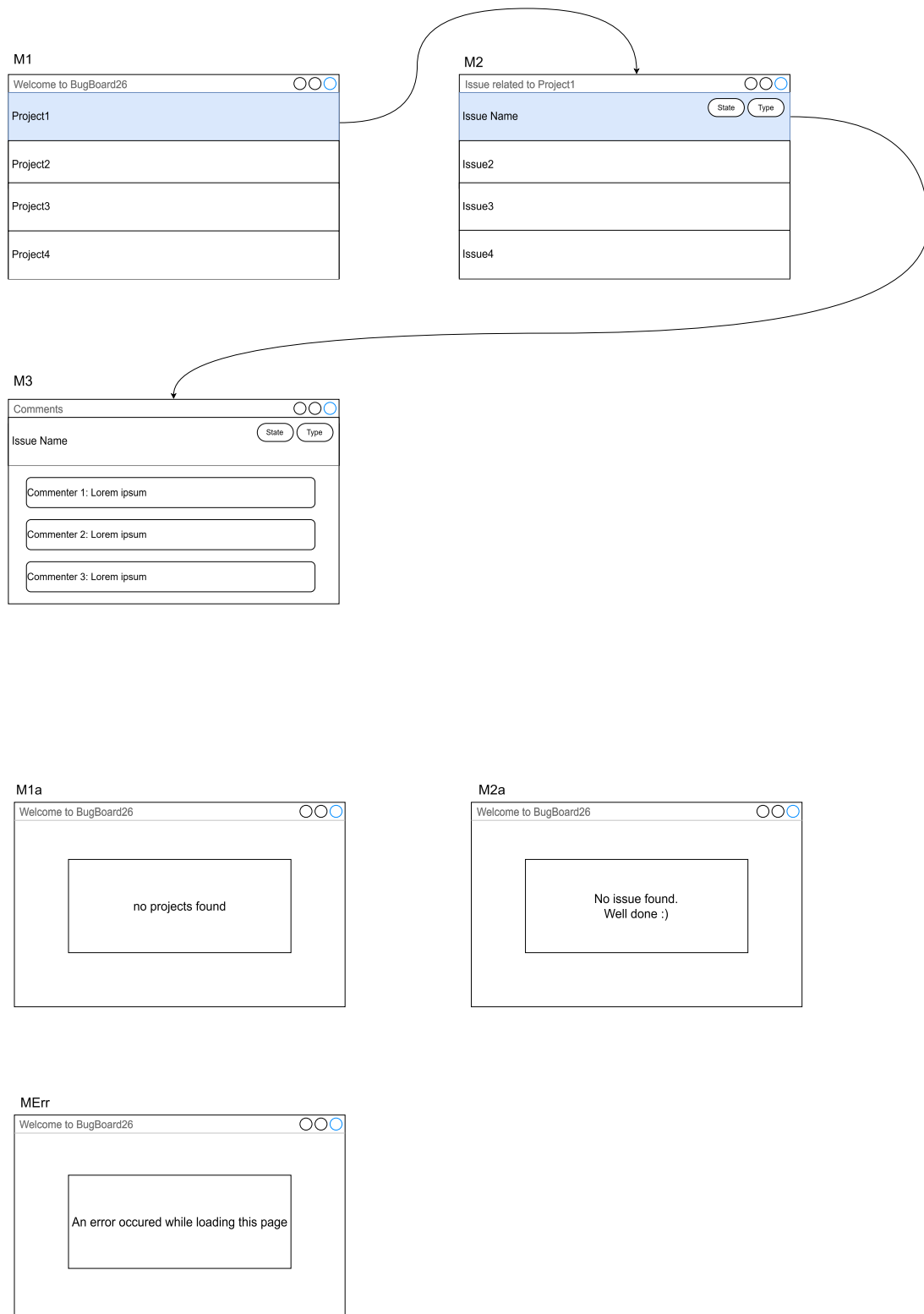


Figura 2: Visualizza issue

E qui di seguito riportiamo l'inerente tabella di Cockburn:

USE CASE	<i>Visualizza issue</i>		
Goal	Un utente vuole visualizzare una issue, le sue proprietà e i commenti.		
Preconditions	L'utente ha un account e si è autenticato.		
Success end conditions	Il sistema mostra la issue e i suoi commenti.		
Failed end conditions	Il sistema mostra una pagina di errore.		
Primary actor	Qualsiasi tipo di user.		
Trigger	L'utente fa accesso.		
Main scenario	Step n.	Utente	Sistema
	1		Mostra M1
	2	Clicca su un progetto	
	3		Mostra M2
	4	Clicca su una issue	
	5		Mostra M3
Extension n° 1 (User has no projects)	1a		Mostra M1a
Extension n° 2 (Project has no issues)	2b		Mostra M2b
Extension n° 3 (Generic error)	1,3,5 err		Mostra MErr

3 SYSTEM DESIGN

3.1 Architettura del sistema

L'architettura del sistema adottata per la realizzazione della piattaforma è di tipo **Client-Server**, più specificatamente a microservizi. Questa scelta, ideale per applicazioni web, disaccoppia nettamente il Frontend (lato utente) dal Backend (logica e dati su server).

I microservizi da noi individuati sono:

- **Servizio di gestione utenze:** si occupa della gestione delle utenze e dell'autenticazione.
- **Servizio di gestione issue:** si occupa della gestione delle issue e dei relativi commenti.

L'organizzazione del Backend in microservizi distinti è stata guidata da tre obiettivi principali:

- **Scalabilità Indipendente:** Consente di allocare risorse computazionali in modo mirato. È possibile, ad esempio, scalare orizzontalmente il servizio di gestione delle issue (soggetto a traffico più intenso) senza dover replicare inutilmente il servizio di gestione utenze.
- **Disaccoppiamento e Manutenibilità:** La suddivisione in moduli riduce la complessità del codice del singolo servizio. Questo favorisce uno sviluppo distribuibile, manutenibile, evolvibile e con test più mirati.
- **Tolleranza ai guasti:** Un eventuale errore critico in un microservizio non compromette necessariamente la disponibilità dell'intera piattaforma.

3.2 Decomposizione del sistema

Il sistema è composto dai seguenti elementi:

1. **Frontend:** interfaccia utente accessibile tramite browser web.
 - Login/Registrazione
 - Visualizzazione progetti e issue
 - Creazione/modifica progetti e issue
 - Gestione profili utente (solo per admin)
2. **Backend:** microservizi che gestiscono le funzionalità principali della piattaforma.
 - **Utenze:** gestione utenti e profili
 - **Gestione issue:** gestione progetti e issue
3. **Database:** sistema di gestione dei dati persistenti.
 - **Utenze:** memorizzazione dati utenti e profili
 - **Gestione issue:** memorizzazione dati progetti e issue

3.3 Scelta delle tecnologie

Per la realizzazione della piattaforma BugBoard26, abbiamo scelto le seguenti tecnologie:

- **Frontend:** *Angular.ts* per la costruzione dell'interfaccia utente, grazie alla sua modularità e facilità di integrazione con backend RESTful.
- **Backend:** *Java Spring* per la creazione dei microservizi, grazie alla sua robustezza.
- **Database:** *PostgreSQL* per la gestione dei dati relazionali, grazie alla sua affidabilità e scalabilità.

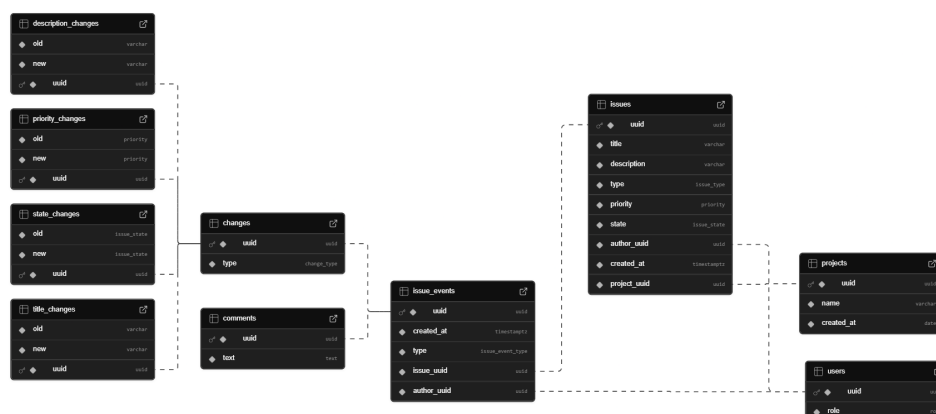
4 SOFTWARE DESIGN

4.1 Persistenza dei dati

Per la parte di persistenza di dati ci siamo affidati a Supabase, non per le funzionalità di MBaaS, ma come provider di database. Come annunciato in precedenza, il backend è diviso in due microservizi, ognuno dei quali fornito del proprio database. Qui di seguito riportiamo lo schema per entrambi:

users		
uuid	uuid	
name	varchar	
surname	varchar	
email	varchar	
password_hash	varchar	
is_admin	bool	
is_active	bool	

Questo è il database dello UsersService. La sua funzione principale è quella di contenere per intero i dati delle varie utenze. Separare le informazioni delle utenze dalle informazioni proprie e caratteristiche di BugBoard26 ci permette una certa scalabilità oltre a impostare BugBoard26 verso quella che potrebbe diventare una famiglia di servizi in un futuro ipotetico.



Questo è il database del BugBoardService, cuore pulsante della piattaforma. È qui che sono conservate tutte le informazioni riguardanti progetti, issue e interazioni degli utenti con queste. Questo possiede un sottoinsieme delle informazioni degli utenti, al fine di semplificare le operazioni intra-database e per permettere il suo corretto funzionamento anche nel caso di down del database dello UsersService.

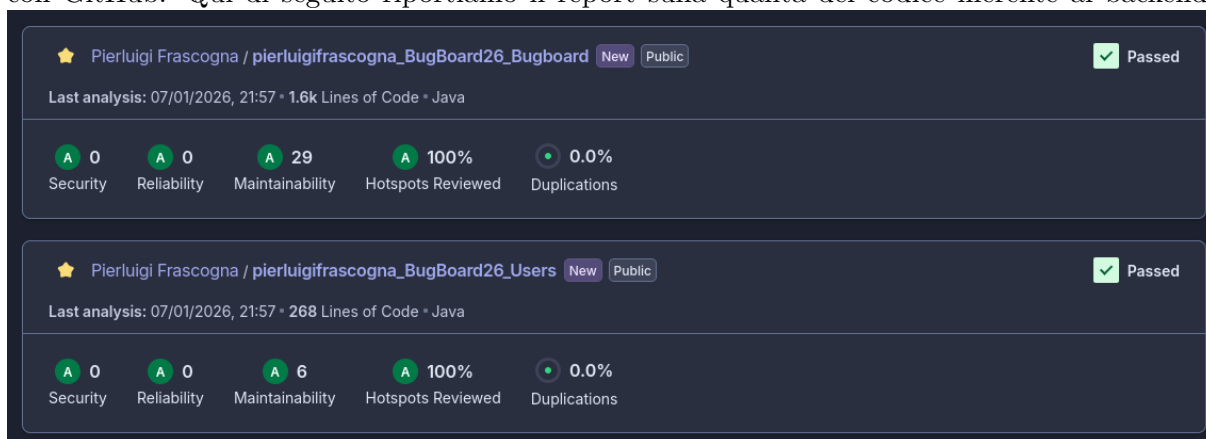
4.2 Strumenti di versioning

Lo strumento di versioning da noi utilizzato è stato GitHub, scelta facile e dettata dalla sua diffusione e semplicità d'uso. È possibile visualizzare la repo di BugBoard26 al seguente link: <https://github.com/PierluigiFrascogna/BugBoard26>

Ovviamente, qui è anche possibile visualizzare ogni possibile statistica relativa a software e contributors, in modo molto più esaustivo di quanto potremmo mai offrire noi.

4.3 Qualità del codice

Per la generazione di report sulla qualità del codice la nostra scelta è ricaduta su SonarQube. Questa scelta è stata dettata dalla sua estensività, semplicità d'uso e la profonda integrazione con GitHub. Qui di seguito riportiamo il report sulla qualità del codice inerente al backend:



5 TESTING

5.1 Test Plan

La funzionalità che abbiamo deciso di testare è quella inerente alle issue in quanto considerata da noi fondamentale per la piattaforma stessa.

Nello specifico testeremo i metodi:

- `ResponseEntity<IssueResponse> postNewIssue(UUID uuid_project, IssueRequest issueRequest)` della classe `IssueAPI`;
- `ResponseEntity<List<IssueResponse>> getIssuesByProject(UUID uuid_project, String type, String priority, String state)` della classe `IssueService`.

La metodologia utilizzata per testare entrambi i metodi è quella black-box, nello specifico R-WECT, in quanto giusto compromesso tra mole di test ed esaustività degli stessi. Ora illustreremo le informazioni relative al testing di ognuno dei metodi.

5.1.1 `ResponseEntity<IssueResponse> postNewIssue(UUID uuid_project, IssueRequest issueRequest)`

Questo metodo, come si può intuire dal nome, si occupa di creare nuove issue e racchiuderle nel corretto formato JSON da spedire al frontend. Al fine del testing sono state individuate le seguenti classi di equivalenza (al fine di una maggiore chiarezza, le classi di equivalenza per parametri diversi di uno stesso metodo saranno in ordine crescente):

Tabella 1: Classi di equivalenza per il parametro `UUID uuid_project`

Classi di equivalenza	Descrizione	Valido
CE1	uuid valido, ossia conforme allo standard UUID e presente all'interno del database	SI
CE2	uuid non valido, ossia non conforme allo standard UUID	NO
CE3	uuid valido, ossia conforme allo standard UUID ma non presente all'interno del database	NO

Tabella 2: Classi di equivalenza per il parametro `issueRequest`

Classi di equivalenza	Descrizione	Valido
CE4	un oggetto di tipo <code>issueRequest</code> valido, ossia correttamente allocabile a partire da un JSON	SI
CE5	un oggetto di tipo <code>issueRequest</code> non valido, ossia impossibile da allocare a partire da un JSON	NO

Tabella 3: Oracolo per test del metodo `ResponseEntity<IssueResponse> postNewIssue(UUID uuid_project, IssueRequest issueRequest)`

Caso di test	Classe di equivalenza	Esito atteso
TC_InvalidProjectUUID	CE2, CE4	Codice 400: Bad Request
TC_NotFoundProjectUUID	CE3, CE4	Codice 404: Not Found
TC_InvalidIssueRequest	CE1, CE5	Codice 400: Bad Request
TC1	CE1, CE4	Codice 201: Created & Issue-Response corretta, con tutti i dati della issue creata

5.1.2 `ResponseEntity<List<IssueResponse>> getIssuesByProject(UUID uuid_project, string type, string priority, string state)`

Questo metodo, come si può intuire dal nome, si occupa di creare nuove istanze della classe Issue.

Tabella 4: Classi di equivalenza per il parametro `uuid_project`

Classi di equivalenza	Descrizione	Valido
CE1	uuid valido, ossia conforme allo standard UUID e presente all'interno del database	SI
CE2	uuid non valido, ossia non conforme allo standard UUID	NO
CE3	uuid valido, ossia conforme allo standard UUID ma non presente all'interno del database	NO

Tabella 5: Classi di equivalenza per il parametro `type`

Classi di equivalenza	Descrizione	Valido
CE_BUG	stringa "BUG"	SI
CE_FEATURE	stringa "FEATURE"	SI
CE_DOCUMENTATION	stringa "DOCUMENTATION"	SI
CE_QUESTION	stringa "QUESTION"	SI
CE_TOTHERS	tutte le altre combinazioni di stringhe possibili	NO
CE_TNULL	"null"	SI

Tabella 6: Classi di equivalenza per il parametro priority

Classi di equivalenza	Descrizione	Valido
CE.LOW	stringa "LOW"	SI
CE.MEDIUM	stringa "MEDIUM"	SI
CE.HIGH	stringa "HIGH"	SI
CE.POTHERS	tutte le altre combinazioni di stringhe possibili	NO
CE.PNULL	"null"	SI

Tabella 7: Classi di equivalenza per il parametro state

Classi di equivalenza	Descrizione	Valido
CE.TODO	stringa "TODO"	SI
CE.PENDING	stringa "PENDING"	SI
CE.DONE	stringa "DONE"	SI
CE.SOTHERS	tutte le altre combinazioni di stringhe possibili	NO
CE.SNULL	"null"	SI

Tabella 8: Oracolo per test del metodo `ResponseEntity<List<IssueResponse>> getIssuesByProject(UUID uuid_project, string type, string priority, string state)`

Caso di test	Classe di equivalenza	Esito atteso
TC_InvalidProjectUUID	CE2, CE_TNULL, CE_PNULL, CE_SNULL	Codice 400: Bad Request
TC_NotFoundProjectUUID	CE3, CE_BUG, CE_LOW, CE_PENDING	Codice 404: Not Found
TC_InvalidType	CE1, CE_TOTHERS, CE_LOW, CE_SNULL	Codice 400: Bad Request
TC_InvalidPriority	CE1, CE_BUG, CE_POTHERS, CE_SNULL	Codice 400: Bad Request
TC_InvalidState	CE1, CE_BUG, CE_LOW, CE_SOTHERS	Codice 400: Bad Request
TC1	CE1, CE_BUG, CE_LOW, CE_TODO	Codice 200: Ok & Lista di IssueResponse corretta, con tutte le issue che rispettano i filtri
TC2	CE1, CE_QUESTION, CE_MEDIUM, CE_PENDING	Codice 200: Ok & Lista di IssueResponse corretta, con tutte le issue che rispettano i filtri
TC3	CE1, CE_FEATURE, CE_HIGH, CE_DONE	Codice 200: Ok & Lista di IssueResponse corretta, con tutte le issue che rispettano i filtri
TC4	CE1, CE_DOCUMENTATION, CE_PNULL, CE_SNULL	Codice 200: Ok & Lista di IssueResponse corretta, con tutte le issue che rispettano i filtri
TC5	CE1, CE_TNULL, CE_PNULL, CE_SNULL	Codice 200: Ok & Lista di IssueResponse corretta, con tutte le issue che rispettano i filtri