UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli Studi di Firenze

Dipartimento di Ingegneria dell'Informazione

KIT: Kotlin Issue Tracker

Authors:
Leonardo Bessi,
Mirco Caneschi,
Matteo Cardinaletti

Corso principale:
Ingegneria del Software

Docente corso:
Vicario Enrico

Indice

Elenco delle figure

1 Analisi

1.1 Statement

Questo progetto offre un sistema moderno di tracciamento dei problemi, progettato specificamente per semplificare la gestione delle iniziative di sviluppo software. La piattaforma consente agli utenti di organizzare e supervisionare efficacemente più progetti, con la possibilità di assegnare loro uno stato, scegliendo tra "In corso" o "Archiviato".

L'applicazione si basa su un'architettura server centralizzata, gestita da un amministratore, garantendo così un controllo costante e l'integrità dei dati. Per favorire lo sviluppo collaborativo, il sistema include la funzionalità che permette agli utenti di invitare collaboratori ai propri progetti, facilitando il lavoro di squadra e la comunicazione.

Elemento chiave dell'utilità del sistema è la possibilità di tracciare e risolvere i problemi in modo dettagliato. Gli utenti possono creare segnalazioni complete, con titoli descrittivi e descrizioni approfondite. Inoltre, i collaboratori possono inserire commenti sulle segnalazioni, offrendo un canale dedicato per aggiornamenti, discussioni sulle soluzioni e feedback, migliorando così il processo collaborativo di risoluzione dei problemi.

1.2 Casi d'uso

Il presente diagramma dei casi d'uso rappresenta le interazioni principali tra l'attore **Utente** e il sistema di *issue tracking*. Le funzionalità offerte all'utente sono le seguenti:

- **Gestione di un progetto**: l'utente può avviare un nuovo progetto o modificarne uno esistente. Il progetto costituisce un contenitore per le issue da gestire.
- **Gestione di un problema (issue)**: l'utente può segnalare un nuovo problema, associandolo a uno dei progetti esistenti. L'utente ha anche la possibilità di aggiornare lo stato di un problema, indicandone l'avanzamento come 'aperto', 'in risoluzione' o 'chiuso'.
- **Gestione commenti**: L'utente ha la possibilità di commentare le issue, favorendo un filo di discussione strutturato e collaborativo con gli altri partecipanti al progetto.

Il sistema prevede un unico attore, con comportamenti variabili in base al contesto dei progetti (proprietario o collaboratore).

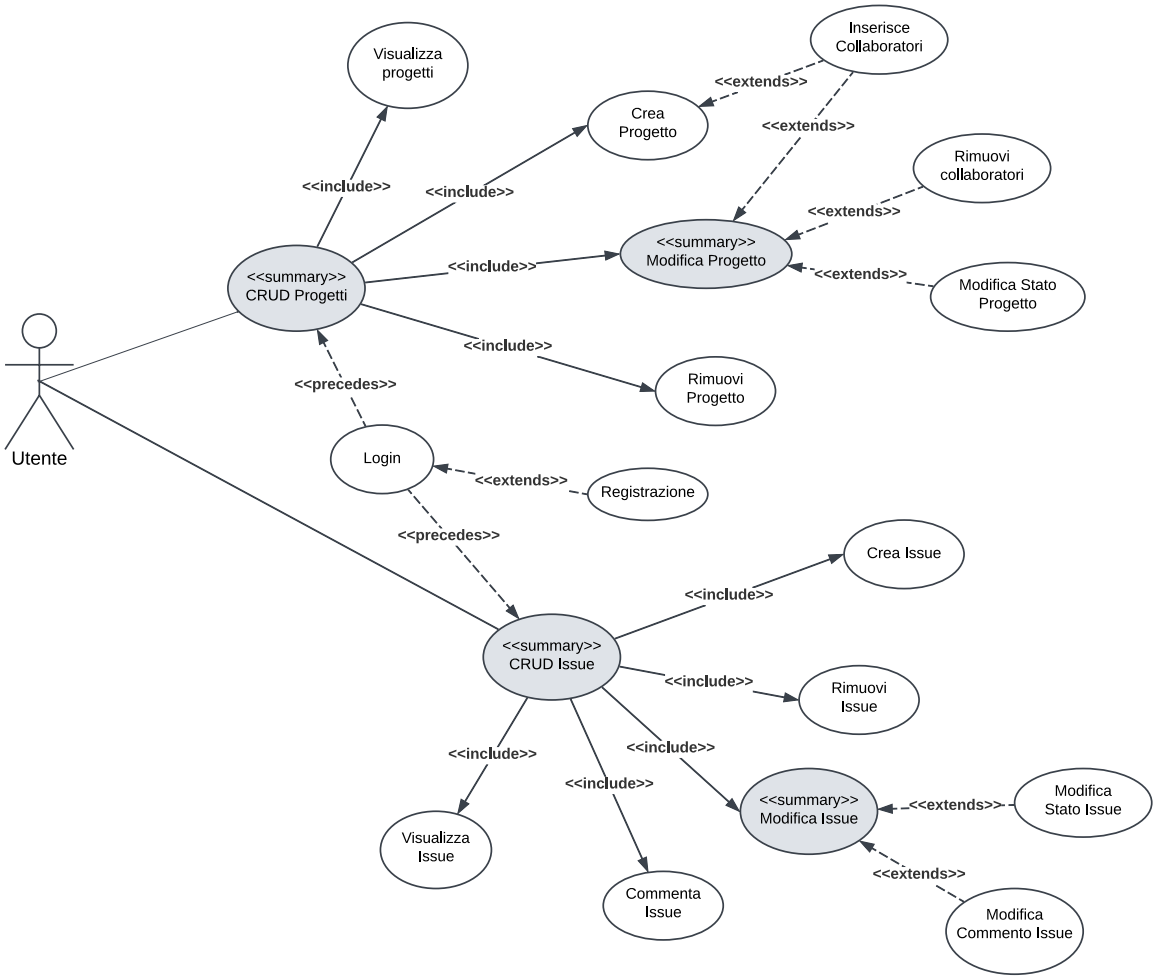


Figura 1: Diagramma UML casi d'uso

1.3 Template dei Casi d'Uso

UC #1: Login	
Livello	Function
Attore Principale	Utente
Descrizione	Permette all'utente di autenticarsi nel sistema per accedere alle funzionalità protette.
Precondizioni	L'utente deve possedere delle credenziali valide (username e password).
Postcondizioni	L'utente è autenticato e ha accesso alle funzionalità del suo ruolo.

Flusso Principale	1. L'utente inserisce username e password. 2. Il sistema verifica la validità delle credenziali. 3. Se le credenziali sono corrette, l'utente viene autenticato.
Estensioni	**2a. Credenziali non valide:** Il sistema mostra un messaggio di errore. **Registrazione (extends):** L'utente può scegliere di avviare il caso d'uso "Registrazione" se non possiede un account.

Tabella 1: Caso d'Uso: Login

UC #2: Registrazione	
Livello	Function
Attore Principale	Utente
Descrizione	Permette a un nuovo utente di creare un account nel sistema. Estende il caso d'uso *Login*.
Precondizioni	L'utente non è autenticato.
Postcondizioni	L'utente ha creato un nuovo account e viene autenticato nel sistema.
Flusso Principale	1. L'utente inserisce i dati richiesti (es. nome, email, password). 2. Il sistema valida i dati e crea un nuovo account utente. 3. L'utente viene registrato e automaticamente autenticato.
Estensioni	**3a. Dati non validi o email già in uso:** Il sistema mostra un messaggio di errore specifico.

Tabella 2: Caso d'Uso: Registrazione

UC #3: Visualizza Progetti	
Livello	User Goal
Attore Principale	Utente
Descrizione	L'utente visualizza la lista dei progetti a cui ha accesso. Questo caso d'uso è incluso in *CRUD Progetti*.
Precondizioni	L'utente deve essere autenticato nel sistema.
Postcondizioni	La lista dei progetti è mostrata all'utente.
Flusso Principale	1. L'utente accede alla sezione "Progetti". 2. Il sistema recupera e mostra l'elenco dei progetti associati all'utente, con informazioni di riepilogo (es. nome, stato).
Estensioni	**2a. Nessun progetto trovato:** Il sistema mostra un elenco vuoto.

Tabella 3: Caso d'Uso: Visualizza Progetti

UC #4: Crea Progetto	
Livello	User Goal
Attore Principale	Utente
Descrizione	L'utente crea un nuovo progetto nel sistema. Questo caso d'uso è incluso in *CRUD Progetti*.
Precondizioni	L'utente deve essere autenticato per creare un progetto.
Postcondizioni	Un nuovo progetto viene creato e salvato nel sistema. L'utente creatore viene assegnato come proprietario.
Flusso Principale	1. L'utente inserisce le informazioni richieste (es. nome del progetto, descrizione). 2. Il sistema valida i dati e crea il nuovo progetto.
Estensioni	

Tabella 4: Caso d'Uso: Crea Progetto

UC #5: Modifica Progetto	
Livello	User Goal
Attore Principale	Utente
Descrizione	L'utente modifica le informazioni di un progetto esistente. Questo caso d'uso è incluso in *CRUD Progetti*.
Precondizioni	L'utente deve essere autenticato e avere i permessi per modificare il progetto selezionato.
Postcondizioni	Le modifiche al progetto sono salvate nel sistema.
Flusso Principale	1. L'utente seleziona un progetto da modificare. 2. L'utente modifica le informazioni desiderate (es. nome, descrizione). 3. Il sistema valida e salva le modifiche.
Estensioni	**Inserisce Collaboratori (extends):** Durante la modifica, l'utente può scegliere di aggiungere nuovi collaboratori al progetto. **Rimuovi Collaboratori (extends):** Durante la modifica, l'utente può scegliere di rimuovere collaboratori esistenti dal progetto. **Modifica Stato Progetto (extends):** Durante la modifica, l'utente può scegliere di archiviare il progetto.

Tabella 5: Caso d'Uso: Modifica Progetto

UC #6: Rimuovi Progetto	
Livello	User Goal
Attore Principale	Utente
Descrizione	L'utente elimina un progetto esistente. Questo caso d'uso è incluso in *CRUD Progetti*.
Precondizioni	L'utente deve essere autenticato e avere i permessi per eliminare il progetto selezionato.
Postcondizioni	Il progetto e tutti i dati associati (es. issue) vengono eliminati dal sistema.

Flusso Principale	1. L'utente seleziona un progetto da eliminare. 2. Il sistema elimina il progetto.
Estensioni	

Tabella 6: Caso d'Uso: Rimuovi Progetto

UC #7: Visualizza Issue	
Livello	User Goal
Attore Principale	Utente
Descrizione	L'utente visualizza le issue associate a un progetto. Questo caso d'uso è incluso in *CRUD Issue*.
Precondizioni	L'utente deve essere autenticato e deve aver selezionato un progetto.
Postcondizioni	La lista delle issue del progetto selezionato è mostrata all'utente.
Flusso Principale	1. L'utente naviga all'interno di un progetto. 2. Il sistema recupera e mostra l'elenco delle issue per quel progetto, con dettagli come titolo, descrizione e stato.
Estensioni	**2a. Nessuna issue trovata:** Il sistema mostra un messaggio che informa l'utente che non ci sono issue per questo progetto.

Tabella 7: Caso d'Uso: Visualizza Issue

UC #8: Crea Issue	
Livello	User Goal
Attore Principale	Utente
Descrizione	L'utente crea una nuova issue all'interno di un progetto. Questo caso d'uso è incluso in *CRUD Issue*.
Precondizioni	L'utente deve essere autenticato e trovarsi all'interno di un progetto.
Postcondizioni	Una nuova issue viene creata e associata al progetto corrente.
Flusso Principale	1. L'utente inserisce le informazioni richieste (titolo, descrizione). 2. Il sistema valida i dati e crea la nuova issue all'interno del progetto.
Estensioni	

Tabella 8: Caso d'Uso: Crea Issue

UC #9: Modifica Issue	
Livello	User Goal
Attore Principale	Utente
Descrizione	L'utente modifica i dettagli di una issue esistente. Questo caso d'uso è incluso in *CRUD Issue*.
Precondizioni	L'utente deve essere autenticato e avere i permessi per modificare la issue selezionata.

Postcondizioni	Le modifiche alla issue sono salvate.
Flusso Principale	1. L'utente seleziona una issue da modificare. 2. L'utente modifica le informazioni desiderate (es. nome, descrizione). 3. Il sistema salva le modifiche.
Estensioni	**Modifica Stato Issue (extends):** L'utente può cambiare lo stato della issue (es. "Aperta", "In corso", "Chiusa"). **Modifica Commento Issue (extends):** L'utente può modificare un commento che ha precedentemente aggiunto alla issue.

Tabella 9: Caso d'Uso: Modifica Issue

UC #10: Commenta Issue	
Livello	User Goal
Attore Principale	Utente
Descrizione	L'utente aggiunge un commento a una issue. Questo caso d'uso è incluso in *CRUD Issue*.
Precondizioni	L'utente deve essere autenticato e aver selezionato una issue.
Postcondizioni	Un nuovo commento viene aggiunto alla issue e reso visibile agli altri membri del progetto.
Flusso Principale	1. L'utente seleziona una issue. 2. L'utente inserisce il corpo del commento. 3. L'utente invia il commento. 4. Il sistema salva il commento e lo associa alla issue.
Estensioni	**3a. Contenuto non valido:** Se il commento è vuoto, il sistema mostra un avviso.

Tabella 10: Caso d'Uso: Commenta Issue

UC #11: Rimuovi Issue	
Livello	User Goal
Attore Principale	Utente
Descrizione	L'utente elimina una issue da un progetto. Questo caso d'uso è incluso in *CRUD Issue*.
Precondizioni	L'utente deve essere autenticato e avere i permessi per eliminare la issue selezionata.
Postcondizioni	La issue viene eliminata dal progetto.
Flusso Principale	1. L'utente seleziona una issue da eliminare. 2. Il sistema elimina la issue.
Estensioni	

Tabella 11: Caso d'Uso: Rimuovi Issue

1.4 Architettura e tecnologie utilizzate

La nostra applicazione, scritta in Kotlin, ha una struttura client-server come imposto dal framework di sviluppo Ktor. La comunicazione dei due programmi avviene attraverso servizi REST, scambiandosi richieste HTTP. Di seguito vengono descritte le varie tecnologie.

1.4.1 Il linguaggio Kotlin

Kotlin è un linguaggio di programmazione moderno e conciso, sviluppato da JetBrains. Progettato per interoperare pienamente con Java. Offre miglioramenti in termini di sintassi, sicurezza del tipo e supporto alla programmazione funzionale. Tra le sue caratteristiche principali vi sono la null-safety, l'inferenza di tipo e l'uso di funzioni estese e lambda, che favoriscono uno stile di scrittura più dichiarativo e meno verboso rispetto a Java.

```kotlin
fun main() {
    // neverNull has String type
    var neverNull: String = "This can't be null"
    // Throws a compiler error
    neverNull = null

    // nullable has nullable String type
    var nullable: String? = "You can keep a null here"
    // This is OK
    nullable = null
}
```

Questo esempio mostra la proprietà di null-safety del linguaggio, mettendo in risalto la particolare attenzione ai tipi di dato che il Kotlin richiede.

1.4.2 Ktor

Ktor è un framework asincrono per la realizzazione di applicazioni web e servizi RESTful, anch'esso sviluppato da JetBrains. Una delle sue principali caratteristiche è l'utilizzo di un DSL (Domain Specific Language) che consente la definizione degli endpoint HTTP in modo chiaro e conciso.

Il framework permette di strutturare un server in poche righe di codice, mantenendo al contempo la modularità e la leggibilità. Segue un esempio tipico di utilizzo, dove viene avviato un server incorporato (embedded) utilizzando il motore Netty sulla porta 8000. All'interno del blocco `routing`, si definisce un endpoint HTTP di tipo GET sulla root `"/"`, che restituisce la stringa "Hello, world!" in risposta. L'approccio DSL adottato rende semplice la definizione delle rotte, offrendo una sintassi espressiva che si integra perfettamente con lo stile idiomatico del linguaggio.

```kotlin
fun main() {
    embeddedServer(Netty, port = 8000) {
        routing {
            get("/") {
                call.respondText("Hello, world!")
            }
        }
    }.start(wait = true)
}
```

1.4.3 JetBrains Exposed

Exposed è un framework ORM (Object-Relational Mapping) per Kotlin, progettato per facilitare l'interazione con database relazionali. Exposed offre due API principali: una basata su DSL SQL e una orientata agli oggetti tramite DAO (Data Access Object). Il DSL SQL consente di scrivere query in uno stile tipicamente Kotlin, mantenendo il controllo sulla struttura e sul comportamento delle interrogazioni.

```
val result: Query = Users.select { Users.id eq 1 }
```

L'API DAO, invece, permette di mappare direttamente le entità del database su oggetti Kotlin, semplificando ulteriormente la gestione dei dati applicativi. Ad esempio, una query per ottenere un'entità con ID specifico usando l'API DAO può essere scritta in questo modo:

```
val user: User = User.findById(1)
```

È richiesto l'utilizzo di entrambe le API per gestire l'accesso al database. Si veda la sezione 3.6.1 per una spiegazione più dettagliata riguardo al loro impiego.

Client Web

Al fine di testare le principali funzionalità della nostra applicazione, viene fornito con essa un client web minimale. Realizzato con HTML, CSS e JavaScript puri seguendo un modello di Single Page Application (SPA), questo approccio privo di ulteriori framework ha favorito l'implementazione rapida degli elementi chiave della nostra applicazione in modo tale da poter testare in modo visivo il suo funzionamento. È possibile trovare alcune schermate relative al progetto del client alla sezione 5.

L'applicazione web non è da considerarsi completa in quanto non copre l'interezza dei casi d'uso analizzati precedentemente. Si lascia pertanto il suo sviluppo come un possibile obiettivo futuro.

2 Progettazione

Sebbene sia comune effettuare la separazione tra **Domain Model**, **DAO** e **Business Logic**, l'utilizzo di Ktor all'interno del nostro progetto ha permesso di definire tutto l'ambito della business logic dentro i costrutti specifici `route` del linguaggio. Si mostrano i diagrammi per i package **domainmodel** e **repository** separatamente.

2.1 Domain Model

Il package contiene tutte le classi, più nello specifico `data class`, che rappresentano le entità base dell'applicazione. Per una spiegazione più dettagliata riguardo all'implementazione, fare riferimento al capitolo successivo.

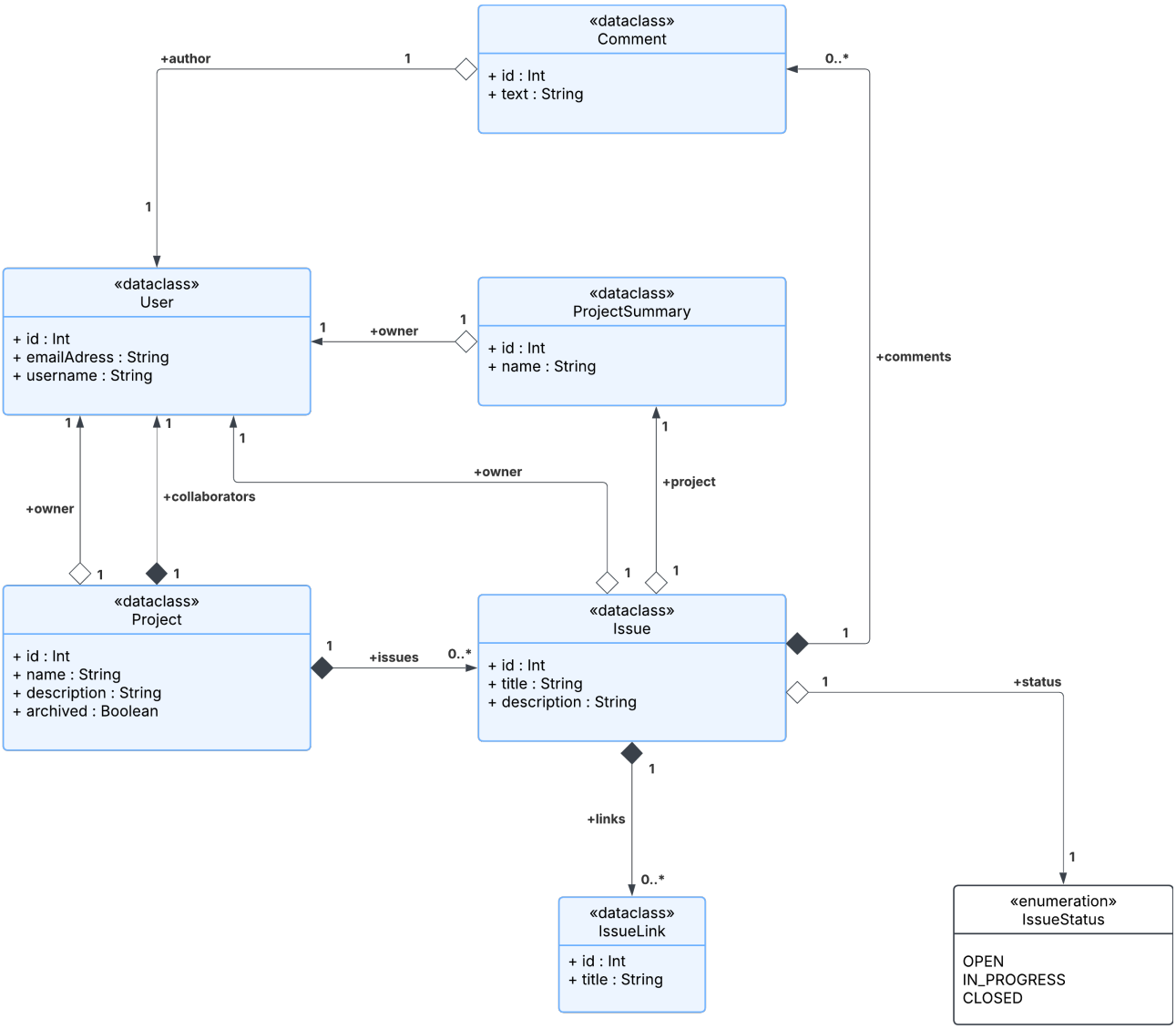


Figura 2: Diagramma UML delle classi: Domain Model

2.2 Repository

Il package prende il nome dall'omonimo design pattern. Tutte le funzionalità di persistenza richieste dall'applicazione sono definite nelle interfacce *Repository* associate alle varie classi.

Per supportare qualsiasi tipo di storage backend, è sufficiente implementare le relative funzioni d'interfaccia.

Nel nostro progetto, abbiamo scelto di utilizzare la libreria **JetBrains Exposed**, progettata appositamente per **Kotlin**. Questa, a sua volta, può utilizzare i driver **JDBC** e **R2DBC** per la connessione ai database.

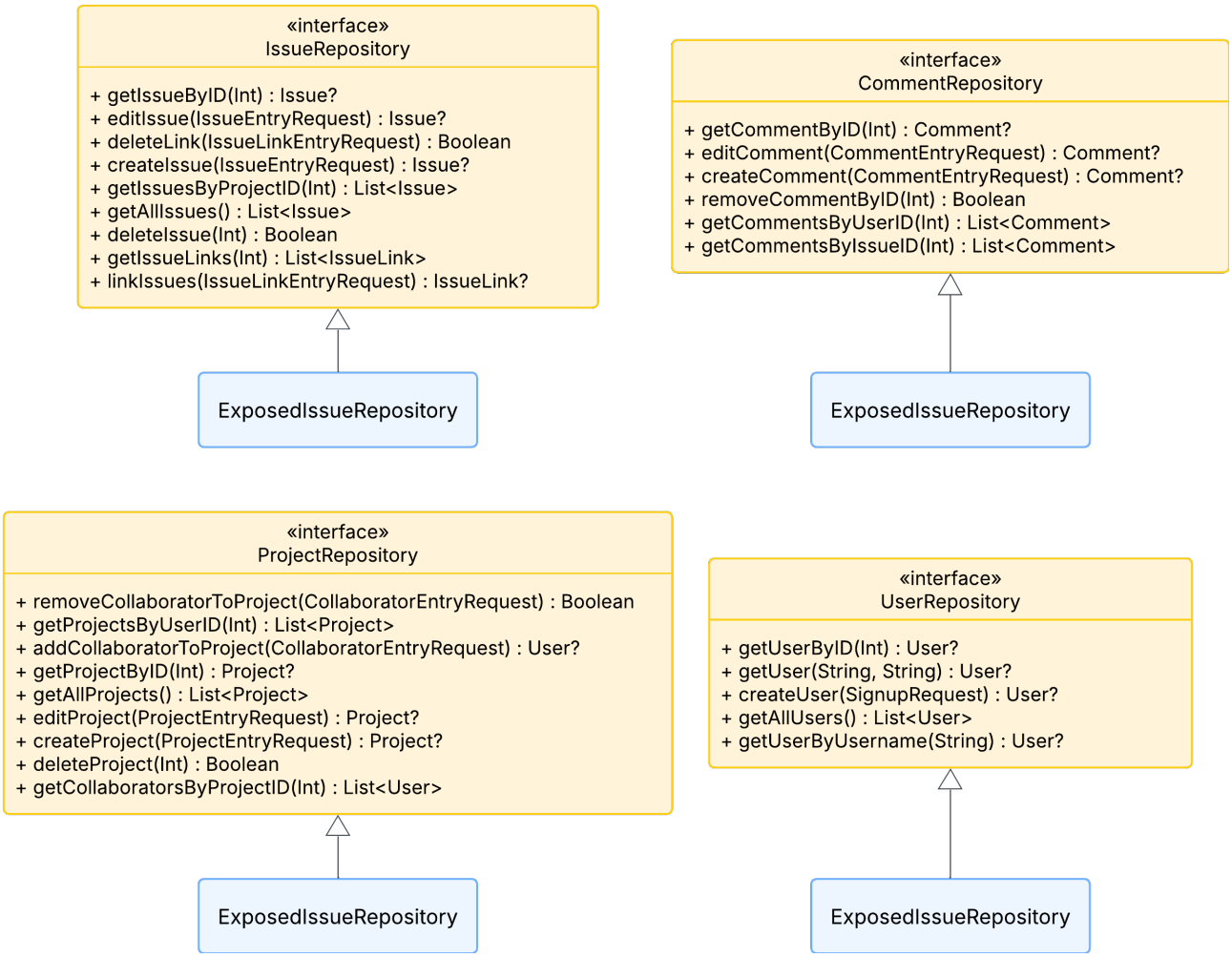


Figura 3: Diagramma UML delle classi: Repository

2.3 Design Pattern

Nel progetto sono stati impiegati diversi design pattern noti, tra cui: **DTO**, **Repository** e, indirettamente, il pattern **Active Record**, utilizzato attraverso l'API DAO di **JetBrains Exposed**.

2.3.1 DTO

Il pattern **Data Transfer Object** (DTO) viene utilizzato per trasferire dati tra i diversi livelli dell'applicazione, in particolare tra la logica di business e la logica di presentazione o di persistenza. I DTO sono oggetti semplici, che contengono esclusivamente dati. Il loro utilizzo permette di disaccoppiare le entità del dominio dalle rappresentazioni esterne, facilitando la serializzazione, il testing e l'integrazione con API REST.

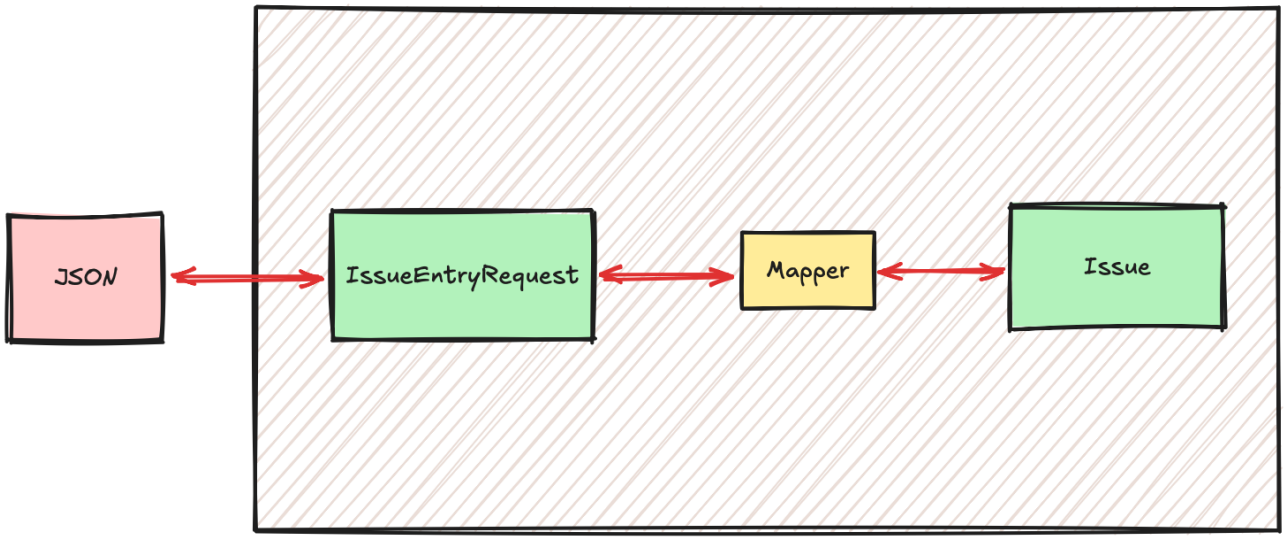


Figura 4: Diagramma pattern DTO

2.3.2 Repository

Il pattern **Repository** fornisce un'interfaccia astratta per accedere ai dati da una fonte esterna (ad esempio un database), nascondendo i dettagli dell'implementazione concreta. Nel nostro progetto, per ogni entità è definita un'interfaccia Repository che specifica le operazioni CRUD richieste. È poi possibile fornire diverse implementazioni per vari backend. Questo approccio migliora la manutenibilità e facilita il testing, permettendo l'uso di mock o repository in-memory.

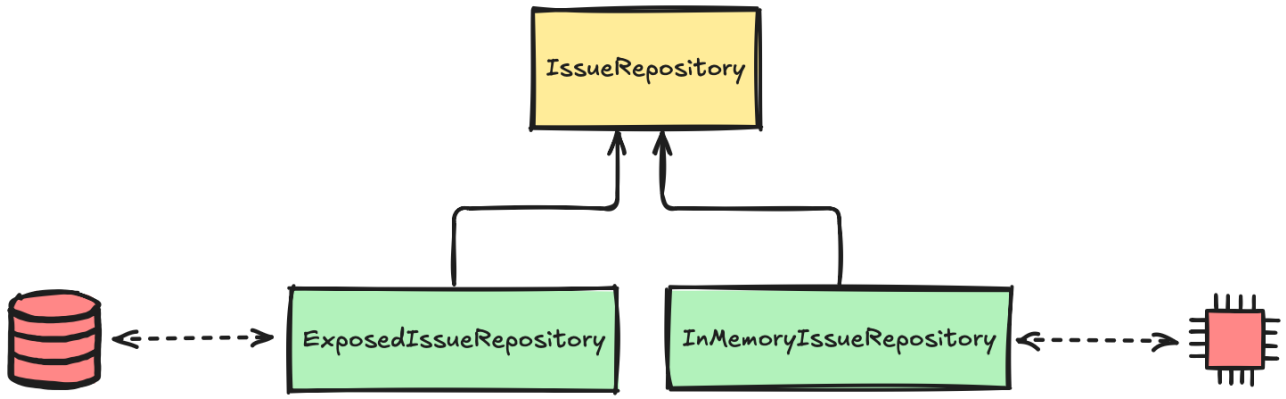


Figura 5: Diagramma pattern Repository

2.3.3 Active Record, DAO API JetBrains Exposed

Il pattern **Active Record** associa ciascuna entità del dominio a una riga di una tabella del database, incorporando in essa sia i dati che i metodi per accedervi e modificarli.

L'API DAO fornita da **JetBrains Exposed** segue questo approccio: ogni classe DAO estende `IntEntity` (o `Entity`), rappresentando una singola riga di una tabella definita tramite `IntIdTable`. Le operazioni CRUD sono disponibili come metodi dell'oggetto stesso o tramite la `companion object`, ad esempio:

```
val user = User.findById(1)
user?.age = 30
user?.flush()
```

Questa struttura consente una gestione dei dati semplice e diretta, tipica di questo pattern, pur richiedendo l'esecuzione delle operazioni all'interno di un blocco `transaction`.

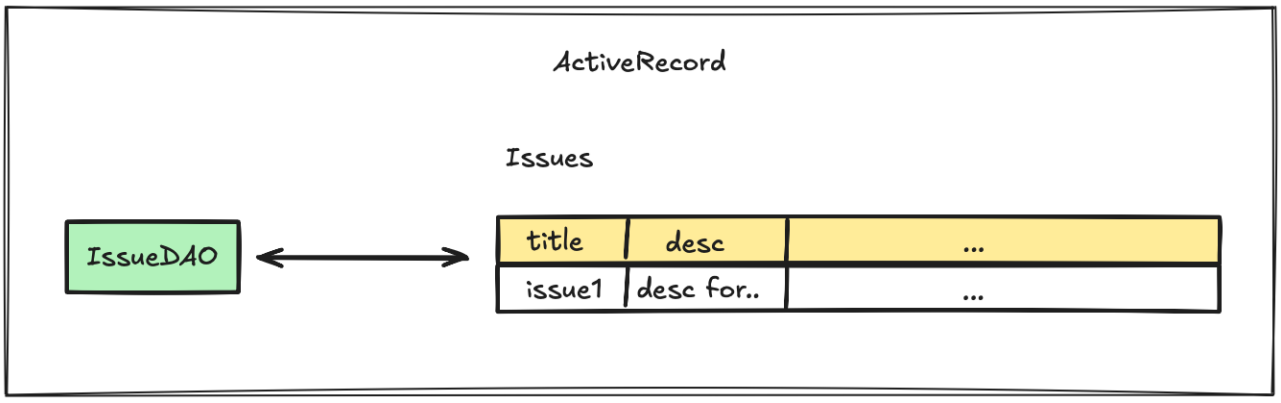


Figura 6: Diagramma pattern Active Record

3 Implementazione

Il progetto è stato sviluppato e strutturato secondo la documentazione ufficiale di **Ktor**.

3.1 Domain model

L'implementazione del domain model usa delle classi speciali di Kotlin chiamate `data class`, molto simili ai `record` presenti a partire da Java 14. Mostriamo ad esempio come è realizzata la classe che rappresenta una issue:

```kotlin
package edu.kitt.domainmodel

import kotlinx.serialization.Serializable

@Serializable
data class Issue(
    val id: Int,
    val title: String,
    val description: String,
    val status: IssueStatus,
    val project: ProjectSummary,
    val createdBy: User,
    val comments: MutableList<Comment>,
    val links: MutableList<IssueLink>
)
```

Listing 1: Data Class della Issue

Come si può notare, le data class sono un meccanismo semplice per rappresentare entità sulle quali non devono essere definite operazioni complesse. Il compilatore genera automaticamente alcuni metodi quali ad esempio `equals`, `toString` e `copy` che dovrebbero essere altrimenti scritti per una classe normale. È comunque possibile attribuire dei metodi a questo particolare tipo di classi, tuttavia è più consigliato il loro utilizzo solo quando si vuole descrivere un tipo di dato.

Un altro elemento fondamentale è l'annotazione `@Serializable` della libreria `kotlinx.serialization`. Questa annotazione istruisce il compilatore a generare automaticamente il codice per la serializzazione e deserializzazione della data class. Questo processo permette di convertire, quando necessario, un'istanza della classe in un formato per lo scambio di dati, come JSON, usato appunto per la comunicazione con il client. *Si osservi che il codice viene generato in fase di compilazione.*

3.1.1 Classi del Domain Model

- **Comment**: Rappresenta un commento associato a un issue. Contiene autore (User), testo e identificativo univoco.
- **Issue**: Elemento centrale che rappresenta un problema o task all'interno di un progetto. Include titolo, descrizione, stato, proprietario, commenti e collegamenti ad altri issue.
- **IssueLink**: Definisce una relazione tra issue diversi. Contiene un identificativo e un titolo descrittivo per il collegamento.

- **IssueStatus**: Enum che definisce gli stati possibili di un issue: OPEN (aperto), IN_PROGRESS (in lavorazione), CLOSED (chiuso).
- **Project**: Entità principale che rappresenta un progetto completo. Include proprietario, collaboratori, lista issue e metadati come nome e descrizione.
- **ProjectSummary**: Versione semplificata di un progetto, utilizzata per visualizzazioni concise. Include solo id, nome e proprietario.
- **User**: Rappresenta un utente del sistema con informazioni base come username, email e identificativo univoco.

3.2 Request Pipeline

L'entry point dell'applicazione si trova nel file *Application.kt*. La funzione `main` si riduce a una singola chiamata che avvia il server. Da questo punto, **Ktor** invoca il metodo `module` sulla classe *io.ktor.server.application.Application*.

L'implementazione di questo metodo non richiede la creazione di una classe figlia, poiché viene sfruttata la **funzione di estensione** delle classi, parte di Kotlin. Questa scelta di design permette di mantenere un'architettura più pulita e modulare.

All'interno del metodo `module` vengono chiamati i metodi di `Application` che costruiscono la **pipeline**, una catena di elaborazione che ogni richiesta seguirà.

Gli stadi di questa pipeline di Ktor sono:

1. **Setup**: Prepara la chiamata.
2. **Monitoring**: Traccia le chiamate, gestisce il logging e raccoglie metriche.
3. **Plugins**: Fase in cui i plugin, come ad esempio `Authentication` e `ContentNegotiation`, vengono eseguiti.
4. **Call**: Qui la richiesta viene gestita effettivamente, ad esempio tramite il sistema di routing.
5. **Fallback**: Gestisce le richieste che non sono state elaborate nelle fasi precedenti.

Di seguito viene fornita una possibile configurazione di questi stadi (Listing 2).

```kotlin
fun Application.module() {
    // Plugin per stampare le richieste sui log.
    // Si inserisce nello stadio "Monitoring".
    install(CallLogging)

    // Plugin per la serializzazione/deserializzazione JSON.
    // Si inserisce nello stadio "Plugins".
    install(ContentNegotiation) {
        json()
    }

    // Configurazione del routing.
    // Si inserisce nello stadio "Call".
    routing {
        // http[s]://serveraddress:port/ -> "Hello, world!"
        get("/") {
            call.respondText("Hello, world!")
        }
    }
}
```

Listing 2: Configurazione pipeline

3.3 Routing

Per una maggiore leggibilità e per organizzare meglio il codice, i blocchi `routing` possono essere suddivisi in file separati. Nel nostro caso nel file *Routing.kt* vengono definiti gli endpoint dell'API utilizzando le **funzioni di estensione** della classe **Route**.

Queste vengono poi richiamate nel metodo `module` (Listing 3), come mostrato sotto.

```kotlin
fun Application.module() {
    ...
    val repos = initExposedRepositories()
    ...
    authenticate("auth-jwt") {
        route("/api") {
            projectRoutes(repos)
            collaboratorRoutes(repos)
            issueRoutes(repos)
            commentRoutes(repos)
            linkRoutes(repos)
            userRoutes(repos)
        }
    }
    ...
}
```

Listing 3: Inclusione di file di routing esterni

In questo esempio, gli endpoint definiti all'interno del blocco `route("/api")` sono protetti dal meccanismo di autenticazione `auth-jwt`, rendendo l'accesso possibile solo agli utenti autenticati.

3.4 Autenticazione

Di seguito viene mostrato il flusso di autenticazione per gli utenti previsto dalla nostra applicazione (Figura 7).

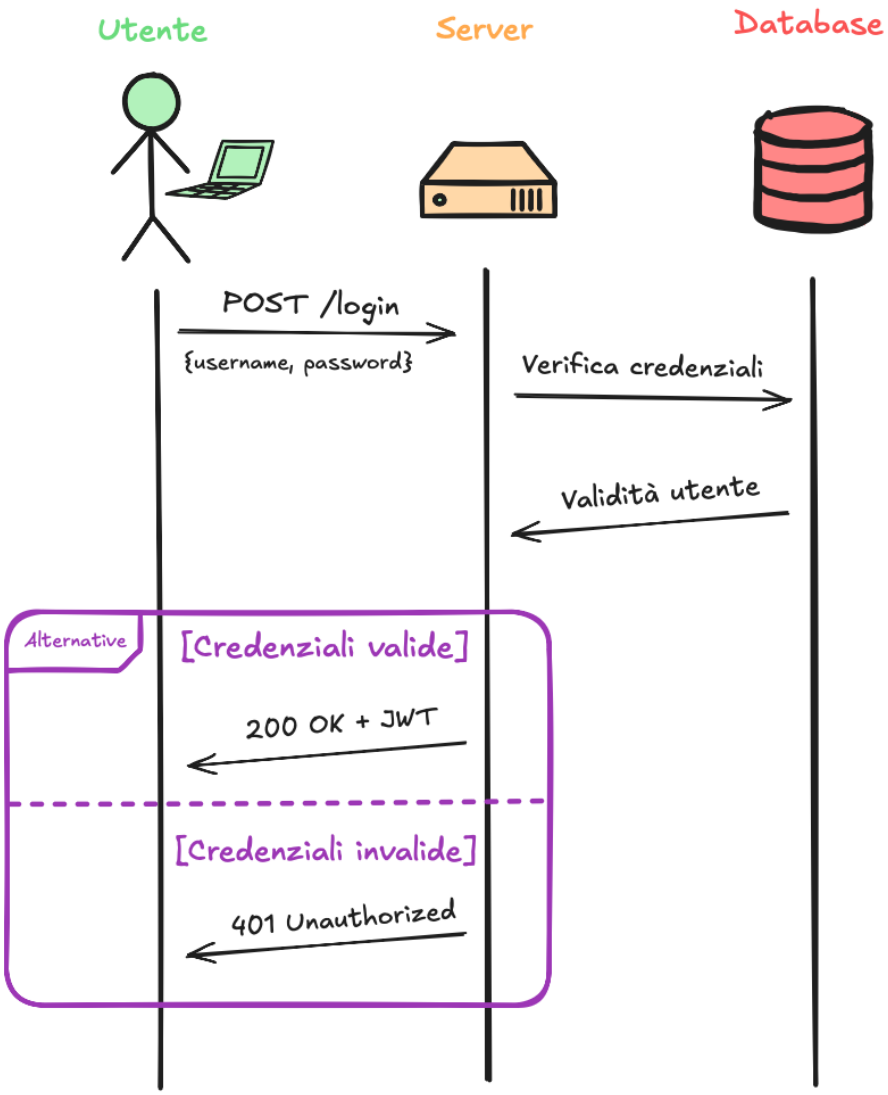


Figura 7: Sequence Diagram: Flusso di autenticazione

Il client invia le credenziali al server che, dopo averle verificate, genera un token JWT. Per tutte le richieste successive il client dovrà usare questo token, altrimenti non potrà accedere agli endpoint protetti.

La verifica sul nostro server può avvenire in due modi, attraverso i cookie o attraverso gli header della richiesta. Questa duplice opzione è stata prevista sia per semplificare la scrittura dei test, che non devono quindi prevedere l'utilizzo di cookie, sia per eventuali client dove questi non sono purtroppo configurabili. Il token JWT fornito all'utente sarà generato a partire dal suo username e dal suo relativo id.

3.5 DTO

Per rendere più flessibile e semplice la realizzazione dei repository e dell'API dell'applicazione è stato utilizzato il pattern DTO (Sezione 2.3.1). All'interno del package `repository.requests` sono presenti le seguenti *data class:*

- `CollaboratorEntryRequest`: Operazioni CRUD per Collaboratori
- `CommentEntryRequest`: Operazioni CRUD per Commenti
- `IssueEntryRequest`: Operazioni CRUD per Issue
- `IssueLinkEntryRequest`: Operazioni CRUD per Link
- `ProjectEntryRequest`: Operazioni CRUD per Progetti
- `UserEntryRequest`: Operazioni CRUD per Utenti
- `LoginRequest`: Richiesta di accesso
- `SignupRequest`: Richiesta di registrazione

L'operazione da svolgere è determinata dal metodo della richiesta, seguendo l'ideologia delle API REST. Una IssueEntryRequest inviata con metodo POST richiederà al server di creare la issue con i campi definiti all'interno del DTO. Nei blocchi route ottenere questo oggetto è molto semplice (Listing 4):

```
fun Route.issueRoutes(repos: Repositories) {
    ...
    post {
        // deserializzazione automatica da json (definito nel plugin ContentNegotiation)
        val issue = call.receive<IssueEntryRequest>()
        ...
        // verifica permessi utente
        ...
        val created = repos.issueRepository.createIssue(issue)
        ...
        // gestione errori
        ...
        call.respond(HttpStatusCode.Created, created)
    }
    ...
}
```

Listing 4: Esempio DTO Issue

3.6 Repository Exposed

Per la realizzazione di un repository che potesse interagire con database relazionali è stata utilizzata la libreria JetBrains Exposed (Sezione 1.4.3). All'interno del package `repository.exposed` è presente l'implementazione e il file che definisce il mapping sul database.

3.6.1 Mapping database

Exposed fornisce una API che permette di definire tabelle direttamente in linguaggio Kotlin (Listing 5).

Successivamente alla definizione delle tabelle, è necessario utilizzare `SchemaUtils` per generare le query responsabili della creazione del database.

```kotlin
object Users : IntIdTable() {
    val userName = varchar("username", 50).uniqueIndex()
    val emailAddress = varchar("email_address", 255)
    val passwordHash = varchar("password_hash", 255)
}
```

Listing 5: Tabella utenti

```
SchemaUtils.create(Users, Projects, Issues, IssueLinks, Comments, Collaborators)
```

Solo con questo è già possibile effettuare query sulla tabella `Users` attraverso il linguaggio DSL SQL (Listing 6).

```kotlin
transaction {
    val user = Users.select { Users.userName eq "john.doe" }.firstOrNull()
    user?.let {
        println("User found: ${it[Users.userName]}")
    }
}
```

Listing 6: Selezione utente per username

Per ogni tabella, è possibile associare un **DAO** (Data Access Object) (Listing 7), che permette di semplificare le query attraverso l'utilizzo di costrutti proprietari del Kotlin (Listing 8).

Per ogni **DAO** è stata inoltre definita una mappa che lo associa alla relativa entità nel **domain model** (Listing 9)

```kotlin
class UserDAO(id: EntityID<Int>) : IntEntity(id) {
    companion object: IntEntityClass<UserDAO>(Users)

    var userName by Users.userName
    var emailAddress by Users.emailAddress
    var passwordHash by Users.passwordHash
}
```

Listing 7: DAO Utenti

```kotlin
transaction {
    val user = UserDAO.find { Users.userName eq "jane.doe" }.firstOrNull()
    user?.let { user ->
        println("User found: ${user.userName}, Email: ${user.emailAddress}")
    }
}
```

Listing 8: Selezione utenti per username con DAO

```kotlin
fun mapUserDAOtoUser(userDAO: UserDAO) = User(
    id = userDAO.id.value,
    username = userDAO.userName,
    emailAddress = userDAO.emailAddress
)
```

Listing 9: Mappatura da DAO a Domain Model

SQL DSL vs DAO

L'utilizzo del pattern DAO in JetBrains Exposed fornisce un livello di astrazione orientato agli oggetti rispetto all'accesso diretto alle tabelle tramite query SQL o DSL. Con i DAO, le entità del database vengono mappate a oggetti Kotlin con proprietà e metodi, permettendo di interagire con i dati in maniera tipizzata e più vicina alla logica di dominio. Ciò riduce la necessità di scrivere manualmente join e conversioni da ResultRow a oggetti, facilitando la navigazione tra relazioni (grazie a proprietà come project.issues invece di query esplicite), e centralizza la logica di accesso ai dati in un unico punto. Al contrario, l'uso diretto delle tabelle tramite la DSL di Exposed, pur essendo più flessibile per query complesse, espone più dettagli implementativi e richiede maggiore codice "collante" per trasformare i risultati in oggetti di dominio. In sintesi, il DAO è preferibile quando si vuole un approccio più dichiarativo, orientato agli oggetti e con meno codice ripetitivo nella gestione delle entità.

3.6.2 Implementazione del repository

Una volta definiti i mapping per le tabelle e i relativi DAO, è possibile procedere con l'implementazione del repository. Questo strato di astrazione conterrà la logica per accedere ai dati, coordinando le operazioni tra il modello di dominio e la sorgente dati.

Ad esempio, il metodo `getUserByID` viene implementato come mostrato nel Listing 10.

```kotlin
override suspend fun getUserByID(uid: Int): User? = withContext(Dispatchers.IO) {
    transaction {
        UserDAO.findById(uid)?.let(::mapUserDAOtoUser)
    }
}
```

Listing 10: Implementazione di getUserByID nel repository

Dall'analisi dello snippet si possono fare le seguenti osservazioni:

- La funzione è marcata con la keyword **suspend**. Le operazioni di accesso al database sono di natura bloccante (I/O). Per evitare di bloccare il thread di esecuzione in attesa della risposta, l'operazione viene eseguita all'interno di una coroutine. La funzione `withContext(Dispatchers.IO)` sposta l'esecuzione del suo blocco su un thread-pool ottimizzato per questo tipo di operazioni. Poiché le richieste in Ktor vengono gestite nativamente tramite coroutine, l'integrazione risulta naturale e performante.

- Viene sfruttata la *null safety* di Kotlin insieme alla scope function `let()`. Il metodo `UserDAO.findById(uid)` restituisce un `UserDAO?`, ovvero un risultato che può essere nullo. L'operatore di chiamata sicura (`?.`) esegue il blocco `let` solo se il risultato non è nullo. In caso contrario, l'intera espressione restituisce `null`, in coerenza con la firma del metodo. Se l'utente viene trovato, la funzione `let` passa il `UserDAO` non nullo alla funzione di mapping `::mapUserDAOtoUser`, che si occuperà di convertirlo nell'oggetto `User` presente nel Domain Model.

4 Test

Uno degli aspetti fondamentali nella realizzazione di un API REST è la possibiltà di testare tutti gli endpoint per verificarne la correttezza, sia funzionale che logica.

4.1 Ktor test suite

Ktor fornisce un sistema basato su JUnit per testare in maniera semplice e compatta gli endpoint. Il Listing 11 mostra un test effettuato sulla configurazione fatta nel listing 2.

```
1  class ApplicationTest {
2      @Test
3      // chiama la funzione testApplication passando il blocco sottostante
4      fun testRoot() = testApplication {
5          // Indica che Application.module() è l'entrypoint della nostra applicazione
6          application {
7              module()
8          }
9          // Test vero e proprio
10         val response = client.get("/")
11         assertEquals(HttpStatusCode.OK, response.status)
12         assertEquals("Hello, world!", response.bodyAsText())
13     }
14 }
```

Listing 11: Esempio Classe di test

È sufficiente quindi definire una classe i cui metodi saranno annotati con *@Test* per controllare l'effettivo funzionamento dei vari endpoint. Si noti la presenza di un blocco `testApplication` utilizzato per contenere il codice relativo al test.

Il blocco `application` interno indica quale sia il metodo entry-point da cercare sotto la classe `Application`. Per maggiori dettagli si veda la Sezione 3.2.

4.1.1 Configurazione specifica per test

Per funzionare, l'applicazione necessita di un file di configurazione *application.yaml* al cui interno si possono trovare informazioni riguardo a database, parametri JWT, etc... In fase di test, poiché ogni test utilizza un database volatile, è necessario fornire valori differenti per evitare problemi di sicurezza.

Per automatizzare il caricamento della configurazione, è stata realizzata una funzione helper (Listing 12).

Il vantaggio di questa soluzione lo si può osservare nella scrittura dei test che richiedono accesso a dati persistenti, mantenendo quindi il codice all'interno del blocco identico ai test normali (Listing 13).

```kotlin
fun testApplicationWithConfig(block: suspend ApplicationTestBuilder.() -> Unit)
= testApplication {
    environment {
        config = ApplicationConfig("application.yaml")
    }
    block()
}
```

Listing 12: Funzione helper test

```kotlin
    @Test
    fun testRoot() = testApplicationWithConfig {
        application { module() }
        val response = client.get("/")
        assertEquals(HttpStatusCode.OK, response.status)
    }
```

Listing 13: Test utilizzando la configurazione

4.2 Descrizione dei test

Si riportano di seguito le descrizioni per i test effettuati.

CollaboratorRoutesTest.kt

Test	Descrizione
test add collaborator as project owner	Verifica che il proprietario di un progetto possa aggiungere collaboratori e che questi possano accedere al progetto
test add collaborator unauthorized as non-owner	Verifica che un collaboratore non proprietario non possa aggiungere altri collaboratori

CommentRoutesTest.kt

Test	Descrizione
test create comment as project collaborator	Verifica che un collaboratore possa creare commenti
test create comment unauthorized	Verifica che utenti non autorizzati non possano creare commenti
test edit comment as author	Verifica che l'autore possa modificare il proprio commento
test edit comment unauthorized as non-author	Verifica che utenti non autorizzati non possano modificare commenti
test delete comment as author	Verifica che l'autore possa eliminare il proprio commento
test delete comment unauthorized as non-author	Verifica che utenti non autorizzati non possano eliminare commenti

IssueRoutesTest.kt

Test	Descrizione
test create issue as project collaborator	Verifica che i collaboratori possano creare issue
test create issue unauthorized	Verifica che utenti non autorizzati non possano creare issue
test get issue as project owner	Verifica l'accesso agli issue da parte del proprietario
test edit issue as issue creator	Verifica che il creatore possa modificare l'issue
test edit issue as project owner (not creator)	Verifica che il proprietario del progetto possa modificare issue
test edit issue unauthorized as collaborator	Verifica che collaboratori non autorizzati non possano modificare issue
test delete issue as project owner	Verifica che il proprietario possa eliminare issue

ProjectRoutesTest.kt

Test	Descrizione
test get projects when not authenticated	Verifica il blocco accesso a progetti senza autenticazione
test get projects when authenticated	Verifica l'accesso alla lista progetti per utenti autenticati
test get single project when authenticated	Verifica l'accesso a singoli progetti
test get non-existent project	Verifica la gestione di richieste per progetti inesistenti
test create new project	Verifica la creazione di nuovi progetti
test get single project as collaborator	Verifica l'accesso ai progetti da parte dei collaboratori
test get single project unauthorized	Verifica il blocco accesso a progetti non autorizzati
test edit project successfully as owner	Verifica la modifica progetti da parte del proprietario
test edit project unauthorized as non-owner	Verifica il blocco modifiche da parte di non proprietari
test delete project successfully as owner	Verifica l'eliminazione progetti da parte del proprietario
test delete project unauthorized as non-owner	Verifica il blocco eliminazione da parte di non proprietari

5 Schermate

Di seguito vengono fornite una serie di immagini prese dal client web fornito insieme all'applicazione KIT sviluppata.

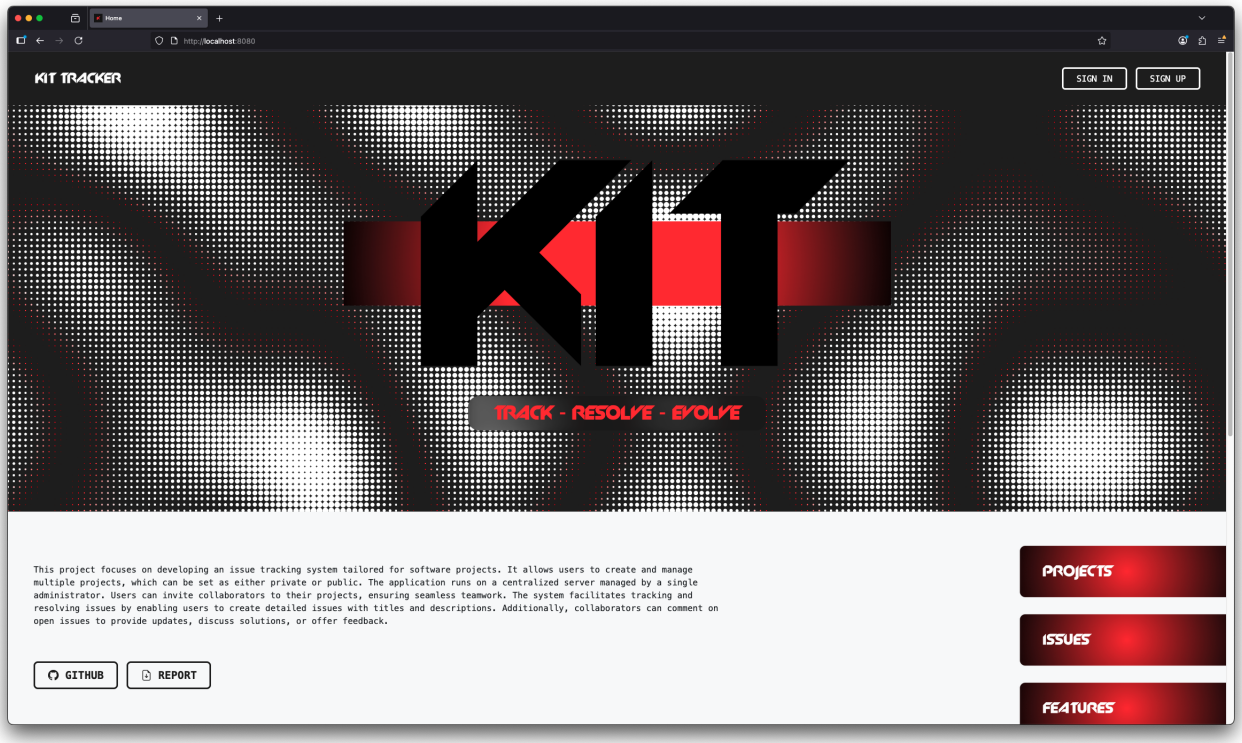
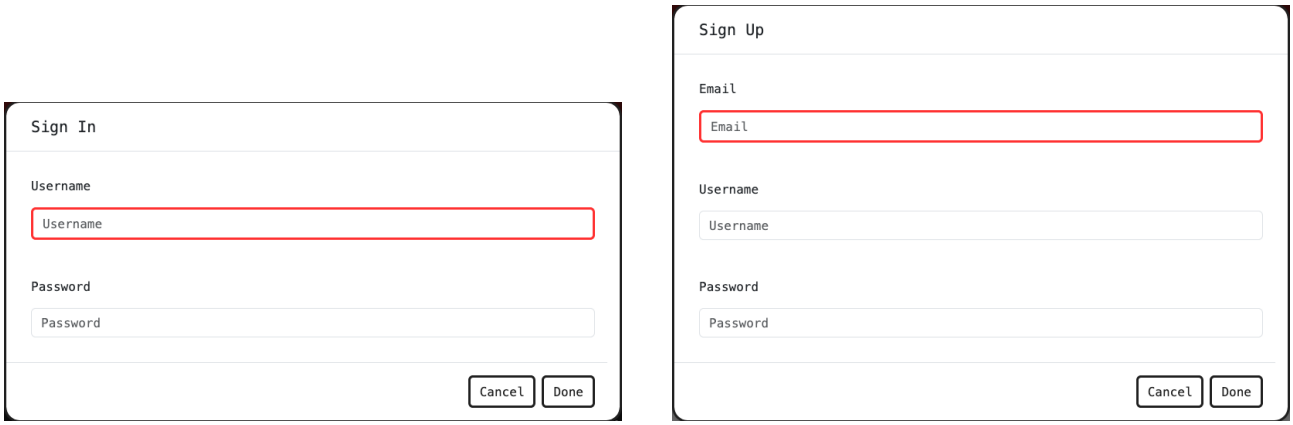


Figura 8: Home page del client



(a) Modale login (b) Modale registrazione

Figura 9: Modali login e registrazione

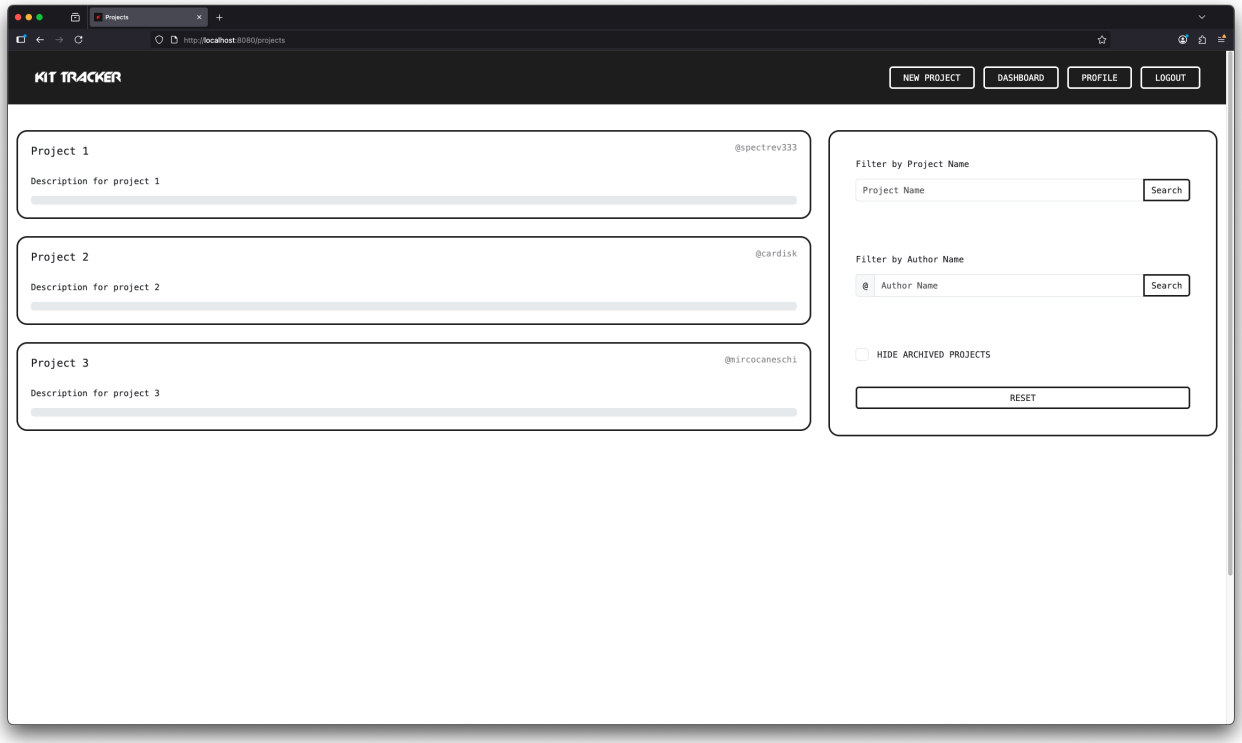


Figura 10: Dashboard

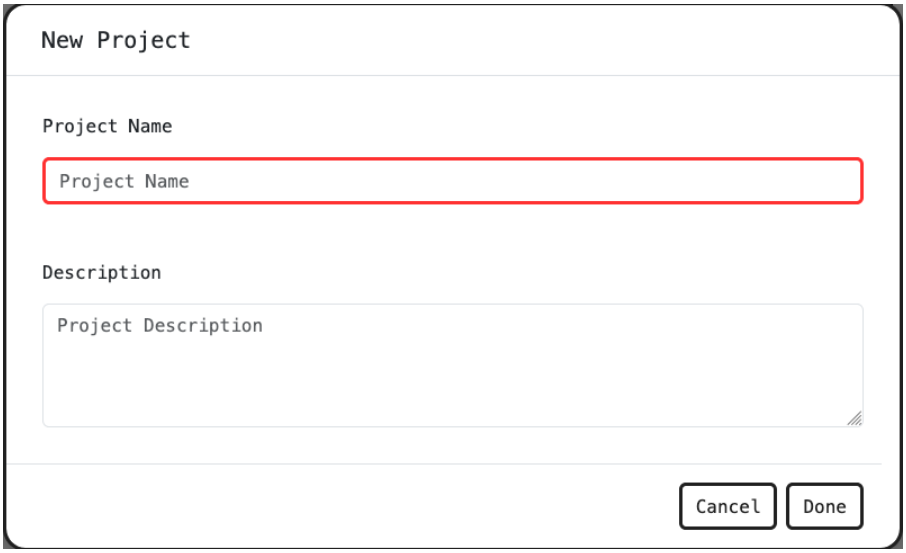


Figura 11: Modale nuovo progetto

Figura 12: Dettagli progetto

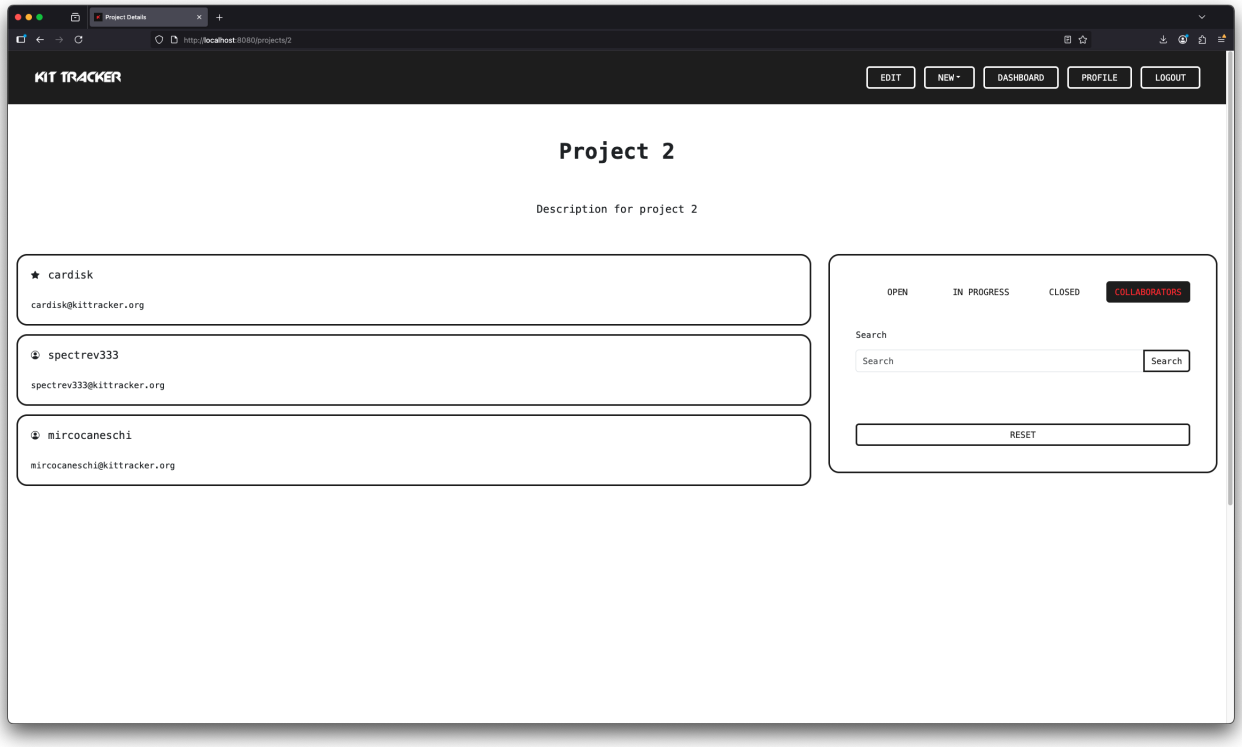


Figura 13: Dettagli collaboratori

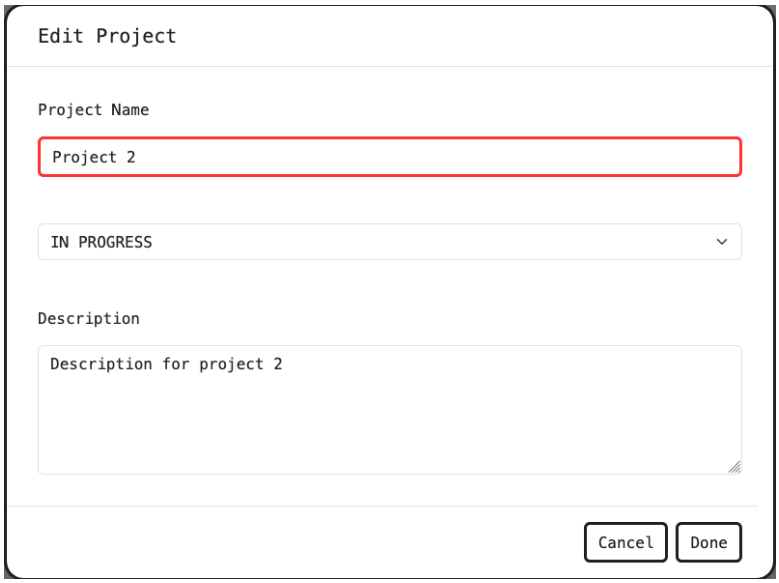
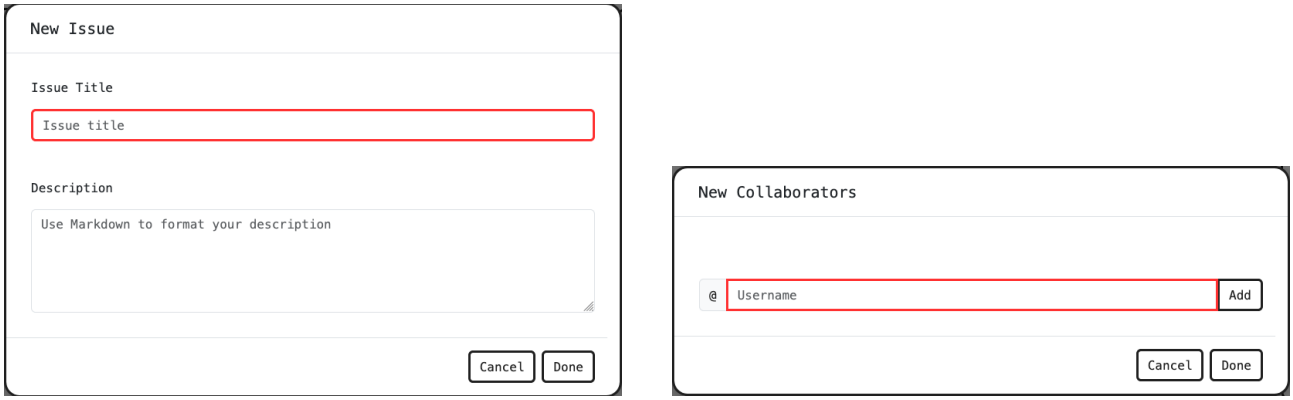


Figura 14: Modale modifica progetto

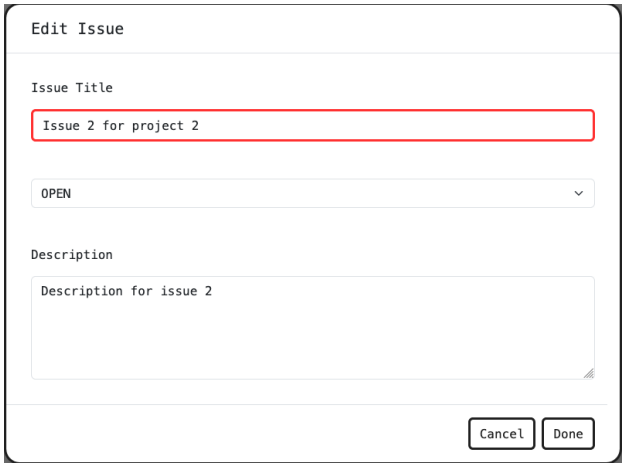


(a) Modale nuovo issue

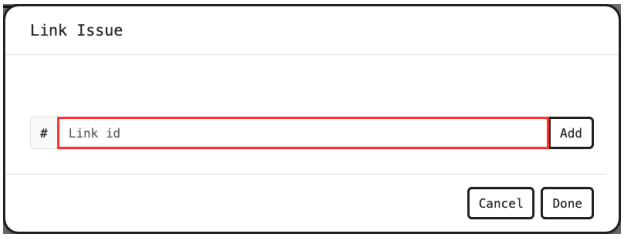
(b) Modale nuovo collaboratore

Figura 15: Modali issue e collaboratori

Figura 16: Dettagli issue



(a) Modale modifica issue



(b) Modale nuovo link

Figura 17: Modali modifica issue e link

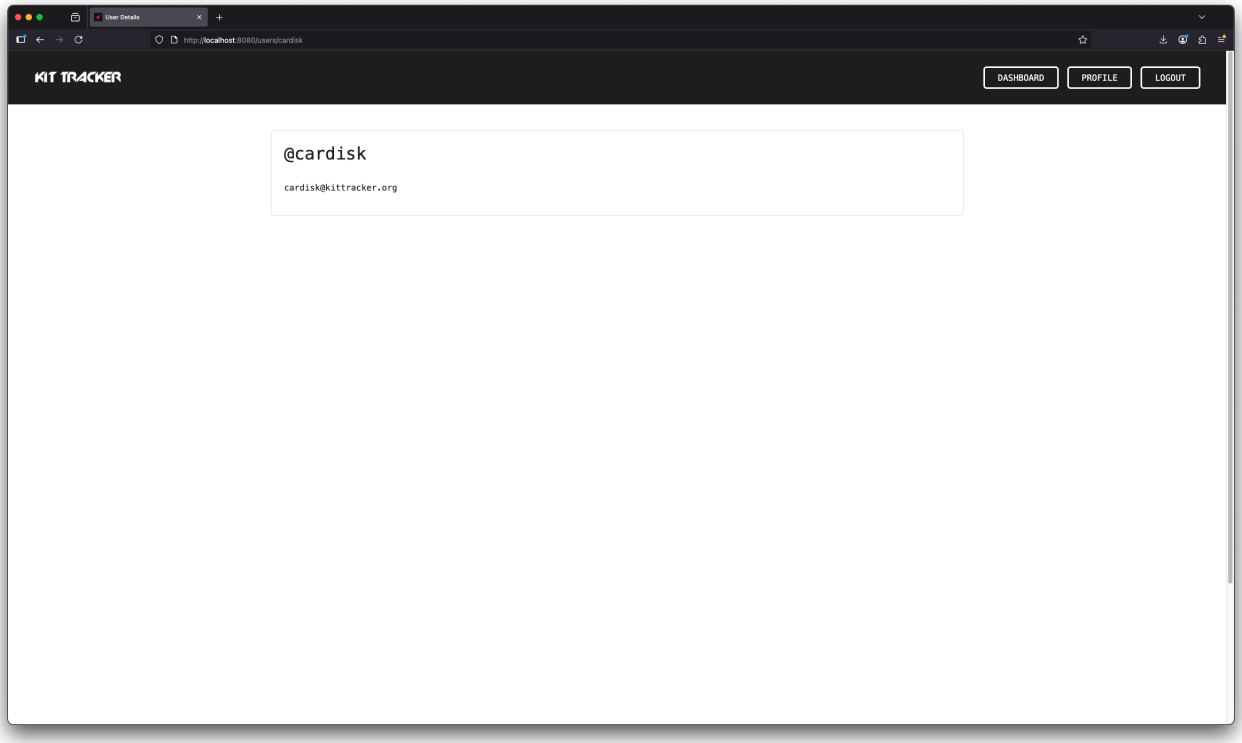


Figura 18: Dettagli profilo

Riferimenti bibliografici

JetBrains. Ktor documentation, 2025a. URL https://ktor.io/docs/welcome.html.

JetBrains. Kotlin programming language, 2020. URL https://kotlinlang.org/.

JetBrains. Exposed, 2025b. URL https://www.jetbrains.com/exposed/.