



UNIVERSITÀ DEGLI STUDI DI FIRENZE
SCUOLA DI INGEGNERIA - DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Elaborato di Software Engineering



Crowdfunding platform for sustainable projects

David Megli

Anno Accademico 2023-2024

Indice

Introduzione	i
0.1 Descrizione del progetto	i
0.2 Etimologia del nome	i
1 Progettazione	1
1.1 Ambienti e strumenti utilizzati	1
1.2 Struttura e Architettura	2
1.3 Use case diagram	4
1.4 Use case template	5
1.5 Mockups	16
1.6 Class diagram	19
1.7 Entity Relationship diagram	24
1.8 Page Navigation Diagram	25
1.9 Struttura della directory del progetto	26
2 Implementazione	28
2.1 Model	28
2.1.1 User	28
2.1.2 Project	28
2.1.3 Campaign	29
2.1.4 Funding	29
2.1.5 SustainabilityGoal	30
2.1.6 ProjectGoal	30
2.1.7 Reward	30
2.1.8 Category	30
2.2 Repository	30
2.3 Service	31
2.3.1 UserService	31
2.3.2 ProjectService	34

2.3.3	CampaignService	34
2.3.4	FundingService	35
2.4	Controller	36
2.4.1	UserController	36
2.4.2	ProjectController	39
2.4.3	CampaignController	40
2.4.4	FundingController	41
2.4.5	SustainabilityGoalController	42
2.4.6	ProjectGoalController	42
2.4.7	RewardController	43
2.4.8	CategoryController	44
2.5	Database	44
3	Testing	46
3.1	User Tests	46
3.2	Project Tests	47
3.3	Test results	50

Introduzione

L'elaborato è stato sviluppato da David Megli nell'anno accademico 2023-2024 durante periodo di agosto-settembre 2024, per il superamento dell'esame di Ingegneria del Software del corso di Laurea Triennale in Ingegneria Informatica dell'Università degli Studi di Firenze.

Il codice sorgente è disponibile al seguente indirizzo:

<https://github.com/davidmegli/terraegis>

0.1 Descrizione del progetto

Il presente progetto mira a creare la base per una piattaforma di crowdfunding dedicata allo sviluppo e al finanziamento di progetti, startup e attività imprenditoriali che perseguono la sostenibilità del pianeta. Attraverso l'impiego di metodologie e tecnologie innovative, l'obiettivo è di contribuire al raggiungimento dei traguardi stabiliti dalle *Nazioni Unite*, in particolare i 17 Obiettivi di Sviluppo Sostenibile (*Sustainable Development Goals*) stabiliti dalle *Nazioni Unite*.

La piattaforma è progettata per consentire a singoli individui o ad imprese di sviluppare progetti e avviare campagne di raccolta fondi, e di sostenere finanziariamente progetti altrui. Le raccolte fondi possono essere effettuate mediante donazioni, investimenti in capitale o attraverso l'acquisto di premi.

0.2 Etimologia del nome

Il nome TerrAegis unisce le parole latine *Terrae* ed *Aegis*, che affiancate si traducono in "Servitori della Terra".

Altra interpretazione è quella di *Aegis* come l'ègida della mitologia greca, ovvero lo scudo di Zeus e un tipo di protezione indossato da Atena.

Entrambe le interpretazioni condividono il significato comune di protezione del pianeta terra, l'obiettivo ultimo a cui mira questo progetto.

Progettazione

1.1 Ambiente e strumenti utilizzati

Per la progettazione e lo sviluppo del software sono stati utilizzati vari strumenti di prototipazione di interfaccia utente, software di creazione di diagrammi, ambienti di sviluppo e framework. L'ambiente di sviluppo principalmente utilizzato è stato IntelliJ IDEA e il linguaggio di programmazione Java, tramite l'utilizzo del framework Spring e Jakarta Persistence API per la gestione della persistenza dei dati. PostgreSQL è stato il Database Management System scelto e pgAdmin il corrispondente software gestionale.

Di seguito riporto la lista completa degli strumenti utilizzati:

- **Carta e penna:** strumenti base per la progettazione software
- **IntelliJ IDEA:** ambiente di sviluppo Java
- **Spring:** framework java per lo sviluppo di applicazioni Java
- **Jakarta Persistence API:** framework di gestione della persistenza dei dati di DBMS relazionali
- **Hibernate:** piattaforma middleware per sviluppo Java con servizio di Object-Relational Mapping
- **pgAdmin:** software gestionale per database PostgreSQL
- **Postman:** piattaforma per la creazione di APIs
- **StarUML:** software per la creazione di diagrammi UML
- **Draw.io:** software per la creazione di diagrammi
- **Lucidchart:** sito web per la creazione di use case diagrams
- **Photoshop:** software per l'elaborazione di immagini

- **Figma**: sito per il design di UI e mockups
- **GitHub**: piattaforma per il versionamento del codice
- **tree.nathanfriend.io**: sito per la creazione di directory diagram
- **Overleaf**: editor LaTeX
- **Carbon**: sito web per la creazione di code snippets

1.2 Struttura e Architettura

La piattaforma è stata progettata in modo da essere suddivisa in due parti distinte: il lato frontend e il lato backend. Il lato frontend è incaricato di gestire le interazioni con l'utente tramite un'interfaccia grafica (la *User Interface*), e comunica a sua volta con il lato backend tramite chiamate HTTP.

Il lato backend è il cervello della piattaforma, che si occupa di fornire i servizi richiesti dal lato frontend. È stato implementato seguendo un'architettura API REST, fornendo quindi punti di accesso tramite URI e interagendo tramite protocollo HTTP con il lato frontend. Il lato backend realizza quindi la business logic, elabora le richieste, interagisce con il database e risponde fornendo i servizi richiesti.

Il lato frontend non è stato implementato, ma ne vengono forniti un prototipo ed una panoramica funzionale tramite mockups e diagramma di navigazione delle pagine.

L'implementazione del lato backend è stata suddivisa in quattro packages principali, ciascuno con specifiche responsabilità. La suddivisione delle responsabilità ricorda vagamente quella del design pattern MVC (*Model*, *View*, *Controller*), con le differenze che le funzionalità della *View* sono delegate al lato frontend, le funzionalità del *Model* sono ripartite fra *Model* e *Repository*, mentre le funzionalità del *Controller* sono ripartite fra *Controller* e *Service*.

Di seguito fornisco una breve descrizione di tali packages:

- **Model**: contiene i modelli, i quali forniscono una rappresentazione strutturale delle entità. Qui sono dichiarate tutte le classi necessarie alla rappresentazione delle tabelle sulla base dati, compresi i loro attributi, i loro tipi ed i loro vincoli. Tramite l'utilizzo di annotazioni

specifiche del framework JPA, esso è in grado di effettuare il mapping delle entità sul database.

- **Repository:** contiene le repositories, le quali forniscono i metodi per l’interazione fra applicazione backend e entità della base dati, il tutto senza l’utilizzo di query SQL. Le repositories sono le interfacce che estendono le interfacce *JpaRepository* di JPA, per le quali il framework fornisce un’implementazione a runtime. Tali repository interagiscono con l’ORM Hibernate, il quale è responsabile della traduzione delle chiamate JPA in query SQL comprese dal database.
- **Service:** i servizi si occupano di implementare la business logic, tramite l’utilizzo delle repositories per l’effettiva interazione del sistema con la base dati.
- **Controller:** i controller costituiscono l’interfaccia con il lato frontend della piattaforma. Si occupano di gestire le chiamate HTTP e redirezionare verso il servizio richiesto.

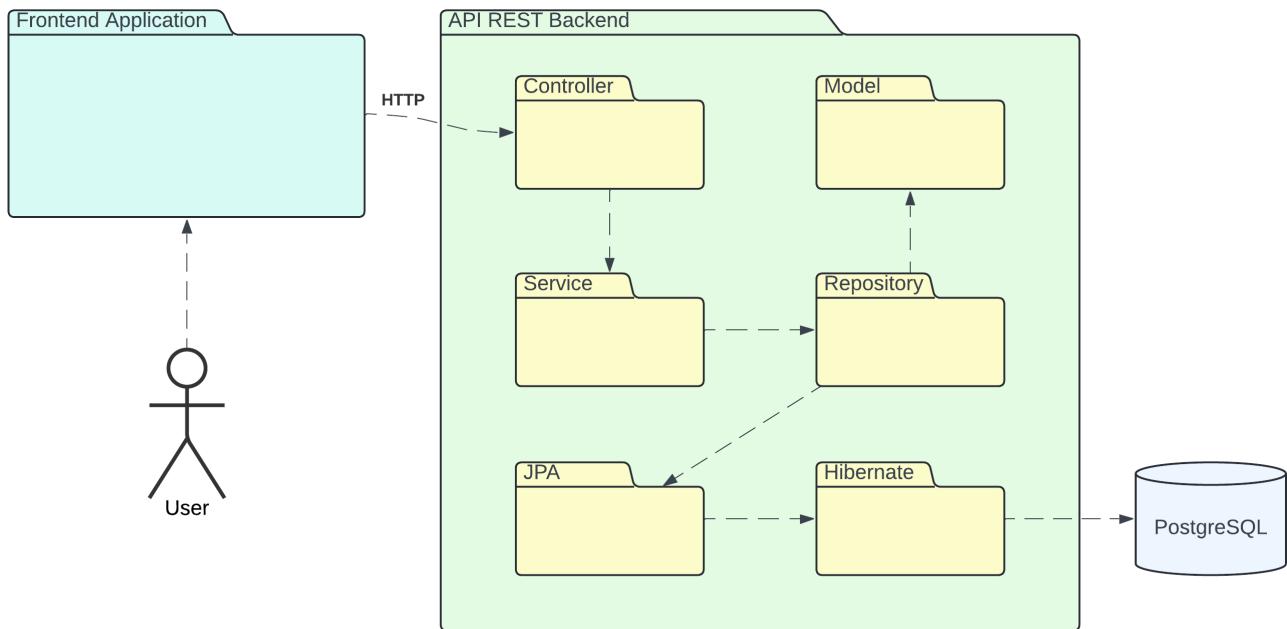


Figura 1.1: Package Dependency Diagram

I principali package utilizzati sono quelli riportati in figura 1.1. L’utente interagisce con l’interfaccia grafica del lato frontend, il quale comunica con il backend tramite richieste HTTP; i controller effettuano il mapping fra le richieste HTTP del lato frontend e i servizi richiesti; i servizi implementano la business logic sfruttando le repositories per la comunicazione con la

base dati; le repositories fanno riferimento ai modelli per conoscere la struttura delle tabelle, i tipi dei dati e i vincoli sul database, e delegano le operazioni CRUD al servizio di ORM (*Object-relational mapping*) di Hibernate.

1.3 Use case diagram

La piattaforma ha come scopo lo sviluppo e il finanziamento di progetti, per questo permette ad ogni utente sia di svolgere il ruolo di finanziatore, che di fondatore (o creatore). Nello use case diagram ho deciso di rappresentare come attore principale un generico utente, piuttosto che suddividere in utente creatore e utente finanziatore, per meglio rispecchiare l'effettiva possibilità fornita dalla piattaforma a un utente di creare progetti che altri utenti possono finanziare e parallelamente finanziare progetti altrui. In figura 1.2 sono rappresentati i principali casi d'uso del sistema, evidenziando le interazioni tra gli attori e le funzionalità principali. Ho rappresentato in blu le azioni relative alla gestione dell'account, in verde quelle relative alla creazione di progetti e campagne di raccolta fondi e in rosso quelle relative al finanziamento di progetti altrui, in grigio l'unico caso d'uso generico in cui un utente visualizza i progetti presenti sulla piattaforma.

I casi d'uso riportanti un identificativo sono descritti più esaustivamente da use case template.

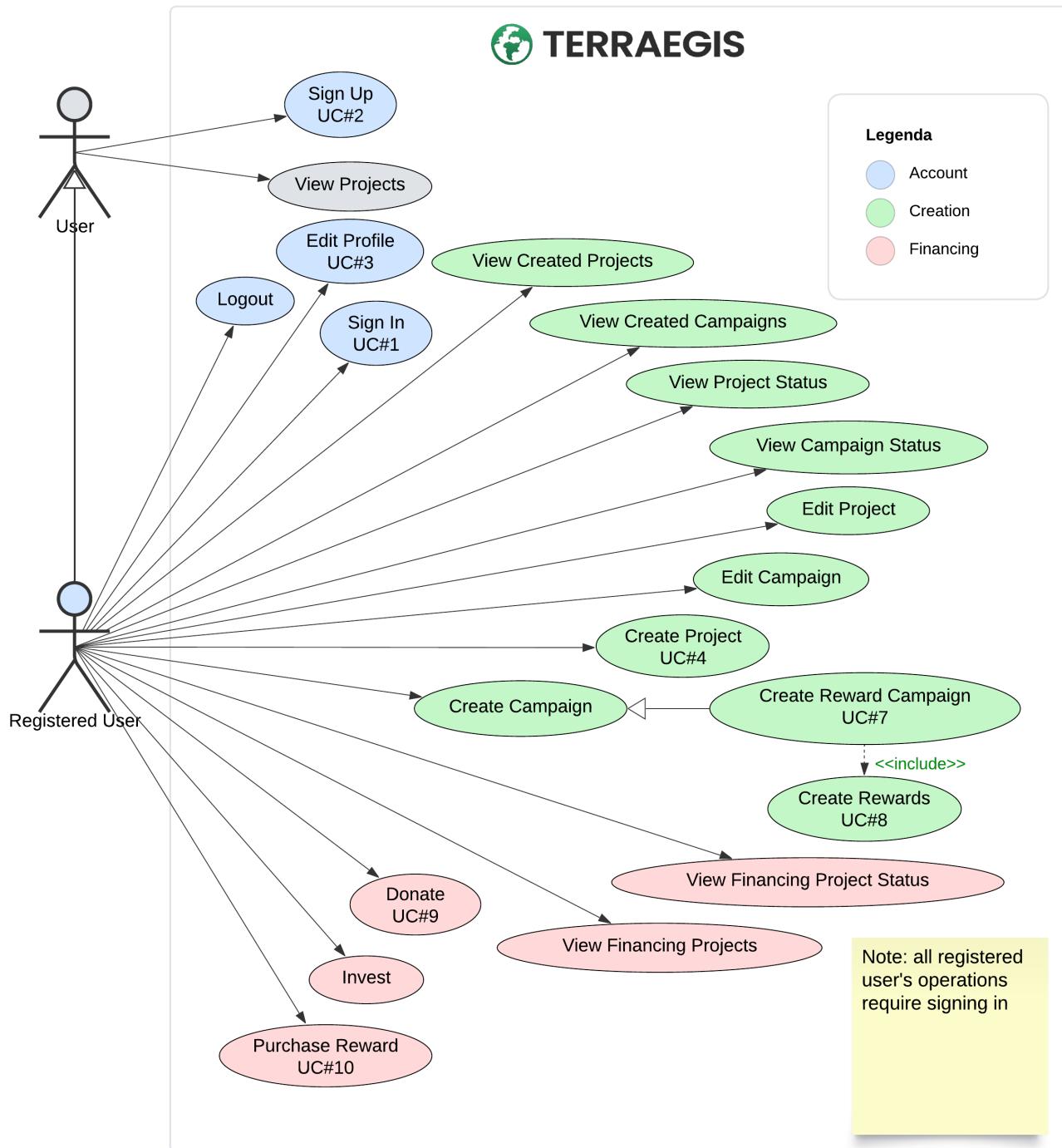


Figura 1.2: Use case diagram

1.4 Use case template

In questa sezione sono riportati 10 use case template relativi ai principali casi d'uso in cui può trovarsi un utente, alcuni dei quali fanno riferimento a casi d'uso riportati nello use case

diagram. Questi casi d'uso riguardano operazioni legate alla gestione dell'account (accesso, registrazione, modifica del profilo), alla creazione di progetti e campagne di raccolta fondi, e al finanziamento di progetti, e presentano una colorazione blu, verde e rossa, rispettivamente.

Per alcuni di questi casi d'uso sono stati realizzati dei mockup, visibili nella sezione successiva.

Use Case #1	Accesso (Sign In)
Description	L'utente esegue l'accesso alla piattaforma inserendo le proprie credenziali (Mockup #1)
Level	User Goal
Actor	User
Pre-conditions	L'utente dispone di un account
Basic Course	<ol style="list-style-type: none">1. Il caso inizia quando l'utente clicca su "Sign In"2. Il sistema porta l'utente alla schermata di accesso3. L'utente inserisce email e password4. L'utente preme "Sign In"5. Il sistema verifica le credenziali6. Il sistema autentica l'utente
Alternative Course	<ol style="list-style-type: none">3a. L'utente clicca su "Password dimenticata" e viene portato alla pagina di recupero della password5a. Se le credenziali non sono corrette viene mostrato un messaggio di errore
Post-conditions	L'utente è autenticato ed ha accesso alla piattaforma

Tabella 1.1: Use case #1: Accesso (Sign In)

Use Case #2	Registrazione (Sign Up)
Description	L'utente si registra alla piattaforma creando un nuovo account
Level	User Goal
Actor	User
Pre-conditions	L'utente non è registrato o non ha effettuato l'accesso
Basic Course	<ol style="list-style-type: none"> 1. Il caso inizia quando l'utente clicca su "Sign Up" 2. Il sistema porta l'utente alla schermata di creazione di un nuovo account 3. L'utente inserisce i dati richiesti 4. L'utente carica un'immagine profilo 5. L'utente conferma la creazione dell'account 6. Il sistema verifica i dati inseriti 7. Il sistema crea un nuovo account salvando i dati forniti 8. Il sistema porta l'utente alla propria pagina di profilo
Alternative Course	<ol style="list-style-type: none"> 4a. Se la dimensione dell'immagine eccede il limite consentito viene mostrato un messaggio di errore 6a. Se i dati inseriti non sono corretti o l'email inserita è già in uso viene mostrato un messaggio di errore
Post-conditions	L'utente è registrato sul database ed ha accesso alla piattaforma tramite le credenziali appena create

Tabella 1.2: Use case #2: Registrazione (Sign Up)

Use Case #3	Modifica del profilo
Description	L'utente modifica le proprie informazioni di profilo
Level	User Goal
Actor	User
Pre-conditions	L'utente possiede un account ed ha effettuato l'accesso
Basic Course	<ol style="list-style-type: none"> 1. Il caso inizia quando l'utente clicca sulla propria immagine di profilo, e successivamente su "Il mio profilo" nel menù a tendina 2. Il sistema porta l'utente alla schermata di modifica del profilo utente 3. L'utente modifica i propri dati 4. L'utente carica una nuova immagine profilo 5. L'utente conferma la modifica 6. Il sistema verifica i dati inseriti 7. Il sistema salva le modifiche apportate 8. Il sistema riporta l'utente alla propria pagina di profilo
Alternative Course	<ol style="list-style-type: none"> 4a. Se la dimensione dell'immagine eccede il limite consentito viene mostrato un messaggio di errore 6a. Se i dati inseriti non sono corretti viene mostrato un messaggio di errore
Post-conditions	Il profilo dell'utente è aggiornato sul database con le nuove informazioni

Tabella 1.3: Use case #3: Modifica del profilo

Use Case #4	Creazione di un progetto
Description	L'utente crea un nuovo progetto (Mockup #4)
Level	User Goal
Actor	User (Creator)
Pre-conditions	L'utente deve essere registrato ed autenticato
Basic Course	<ol style="list-style-type: none"> 1. Il caso inizia quando l'utente clicca su "Nuovo Progetto" 2. Il sistema porta l'utente alla schermata di creazione di un nuovo progetto 3. L'utente inserisce i dati relativi al nuovo progetto 4. L'utente carica un'immagine per il progetto 5. L'utente sceglie una categoria 6. L'utente conferma la creazione del progetto 7. Il sistema verifica i dati inseriti 8. Il sistema crea un nuovo progetto con i dati forniti 9. Il sistema porta l'utente alla pagina di visualizzazione del progetto appena creato
Alternative Course	<ol style="list-style-type: none"> 4a. Se la dimensione dell'immagine eccede il limite consentito viene mostrato un messaggio di errore 6a. Se i dati inseriti non sono corretti viene mostrato un messaggio di errore
Post-conditions	Il progetto è salvato sul database e possono essere avviate campagne di raccolta fondi relative ad esso

Tabella 1.4: Use case #4: Creazione di un progetto

Use Case #5	Creazione di campagna donazioni
Description	L'utente crea una campagna donazioni per un proprio progetto (Mockup #5)
Level	User Goal
Actor	User (Creator)
Pre-conditions	L'utente deve essere registrato ed autenticato ed avere creato almeno un progetto
Basic Course	<ol style="list-style-type: none"> 1. Il caso inizia quando l'utente clicca su un proprio progetto, e successivamente su "Nuova campagna" 2. Il sistema porta l'utente alla schermata di creazione di una nuova campagna 3. L'utente sceglie la tipologia di campagna "Donazioni" 4. L'utente inserisce i dati richiesti, in particolare l'obiettivo della raccolta fondi e il periodo della campagna 5. L'utente conferma la creazione della campagna 6. Il sistema verifica i dati 7. Il sistema crea una nuova campagna e la salva 8. Il sistema porta l'utente alla pagina di visualizzazione della nuova campagna
Alternative Course	<ol style="list-style-type: none"> 6a. Se i dati inseriti non sono corretti viene mostrato un messaggio di errore
Post-conditions	La campagna relativa al progetto selezionato è salvata sul database e sarà attiva nel periodo specificato, in cui altri utenti potranno offrire donazioni

Tabella 1.5: Use case #5: Creazione di campagna donazioni

Use Case #6	Creazione di campagna di investimenti
Description	L'utente crea una campagna di investimenti per un proprio progetto
Level	User Goal
Actor	User (Creator)
Pre-conditions	L'utente deve essere registrato ed autenticato ed avere creato almeno un progetto
Basic Course	<ol style="list-style-type: none"> 1. Il caso inizia quando l'utente clicca su un proprio progetto, e successivamente su "Nuova campagna" 2. Il sistema porta l'utente alla schermata di creazione di una nuova campagna 3. L'utente sceglie la tipologia di campagna "Investimenti" 4. L'utente inserisce i dati richiesti, in particolare l'obiettivo della raccolta fondi e il periodo della campagna 5. L'utente conferma la creazione della campagna 6. Il sistema verifica i dati 7. Il sistema crea una nuova campagna e la salva 8. Il sistema porta l'utente alla pagina di visualizzazione della nuova campagna
Alternative Course	<ol style="list-style-type: none"> 6a. Se i dati inseriti non sono corretti viene mostrato un messaggio di errore
Post-conditions	La campagna relativa al progetto selezionato è salvata sul database e sarà attiva nel periodo specificato, in cui altri utenti potranno investire in equity del progetto

Tabella 1.6: Use case #6: Creazione di campagna di investimenti

Use Case #7	Creazione di campagna premi
Description	L'utente crea una campagna premi per un proprio progetto
Level	User Goal
Actor	User (Creator)
Pre-conditions	L'utente deve essere registrato ed autenticato ed avere creato almeno un progetto
Basic Course	<ol style="list-style-type: none"> 1. Il caso inizia quando l'utente clicca su un proprio progetto, e successivamente su "Nuova campagna" 2. Il sistema porta l'utente alla schermata di creazione di una nuova campagna 3. L'utente sceglie la tipologia di campagna "Premi" 4. L'utente inserisce i dati richiesti, in particolare l'obiettivo della raccolta fondi e il periodo della campagna 5. L'utente conferma la creazione della campagna 6. Il sistema verifica i dati 7. Il sistema crea una nuova campagna e la salva 8. Il sistema porta l'utente alla pagina di creazione dei premi 9. L'utente aggiunge il numero di premi desiderato e per ognuno inserisce le informazioni richieste 10. L'utente conferma la creazione dei premi 11. Il sistema crea i premi salvandoli sulla base dati 12. Il sistema porta l'utente alla schermata di visualizzazione della campagna
Alternative Course	<p>6a. Se i dati inseriti non sono corretti viene mostrato un messaggio di errore</p>
Post-conditions	La campagna relativa al progetto selezionato è salvata sul database e sarà attiva nel periodo specificato, in cui altri utenti potranno

Tabella 1.7: Use case #7: Creazione di campagna premi

Use Case #8	Creazione di premi
Description	L'utente crea dei premi per una campagna raccolta fondi
Level	User Goal
Actor	User (Creator)
Pre-conditions	L'utente deve essere registrato ed autenticato, ed avere creato almeno un progetto e una campagna premi
Basic Course	<ol style="list-style-type: none"> Il caso inizia quando l'utente seleziona una campagna premi nella pagina di visualizzazione di un proprio progetto e clicca su "Modifica campagna" Il sistema porta l'utente alla schermata di modifica della campagna premi L'utente aggiunge il numero di premi desiderato e inserisce le informazioni richieste L'utente conferma la creazione dei premi Il sistema verifica i dati Il sistema crea i premi e li salva Il sistema porta l'utente alla pagina di visualizzazione della campagna premi
Alternative Course	<ol style="list-style-type: none"> Se i dati inseriti non sono corretti viene mostrato un messaggio di errore
Post-conditions	I premi sono salvati sul database ed associati alla relativa campagna. Nel periodo in cui la campagna è attiva i premi sono acquistabili da altri utenti

Tabella 1.8: Use case #8: Creazione di premi

Use Case #9	Finanziamento di un progetto con donazione
Description	L'utente finanzia la campagna donazioni di un progetto (Mockup #6)
Level	User Goal
Actor	User (Investor)
Pre-conditions	L'utente deve essere registrato ed autenticato
Basic Course	<ol style="list-style-type: none"> 1. Il caso inizia quando l'utente seleziona una campagna donazioni attiva e clicca su "Dona" 2. Il sistema porta l'utente alla schermata di donazione 3. L'utente sceglie l'ammontare della donazione 4. L'utente sceglie il metodo di pagamento e conferma la donazione 5. Il sistema porta l'utente alla pagina del processore del pagamento 6. Il sistema conferma l'avvenuta transazione 7. Il sistema riporta l'utente alla pagina di visualizzazione del progetto
Alternative Course	<ol style="list-style-type: none"> 3a. Se l'ammontare scelto è minore del valore minimo viene mostrato un errore 5a. Se la transazione non è andata a buon fine viene mostrato un errore
Post-conditions	L'utente ha effettuato una donazione per il finanziamento di un progetto, e potrà trovarlo nella schermata di visualizzazione dei progetti finanziati. Le informazioni sono salvate su database. La transazione è registrata.

Tabella 1.9: Use case #9: Finanziamento di un progetto tramite donazione

Use Case #10	Finanziamento di un progetto con premi
Description	L'utente finanzia la campagna premi di un progetto acquistando un premio
Level	User Goal
Actor	User (Investor)
Pre-conditions	L'utente deve essere registrato ed autenticato
Basic Course	<ol style="list-style-type: none"> 1. Il caso inizia quando l'utente seleziona una campagna premi attiva e clicca su "Mostra premi" 2. Il sistema porta l'utente alla schermata di selezione dei premi della campagna 3. L'utente sceglie il premio che intende acquistare 4. Il sistema porta l'utente alla pagina di selezione del metodo di pagamento 5. L'utente sceglie il metodo di pagamento e conferma l'acquisto 6. Il sistema porta l'utente alla pagina del processore del pagamento 7. Il sistema conferma l'avvenuta transazione 8. Il sistema riporta l'utente alla pagina di visualizzazione del progetto
Alternative Course	<ol style="list-style-type: none"> 6a. Se la transazione non è andata a buon fine viene mostrato un errore
Post-conditions	L'utente ha finanziato un progetto tramite l'acquisto di un premio, e potrà visualizzarlo nella pagina di visualizzazione dei premi acquistati, o alternativamente nella pagina di visualizzazione del progetto finanziato. Le informazioni sono salvate su database. La transazione è registrata.

Tabella 1.10: Use case #10: Finanziamento di un progetto tramite acquisto di un premio

1.5 Mockups

In questa sezione sono riportati alcuni Mock-ups relativi all’interfaccia utente di una potenziale web application che potrebbe costituire il lato frontend della piattaforma. Qui sono riportati i principali scenari di interazione dell’utente con l’interfaccia: accesso tramite credenziali da parte dell’utente, esplorazione dei progetti creati dagli utenti, visualizzazione della propria pagina di profilo, creazione di un nuovo progetto, creazione di una nuova campagna di raccolta fondi per un proprio progetto, donazione per il progetto di un altro utente.

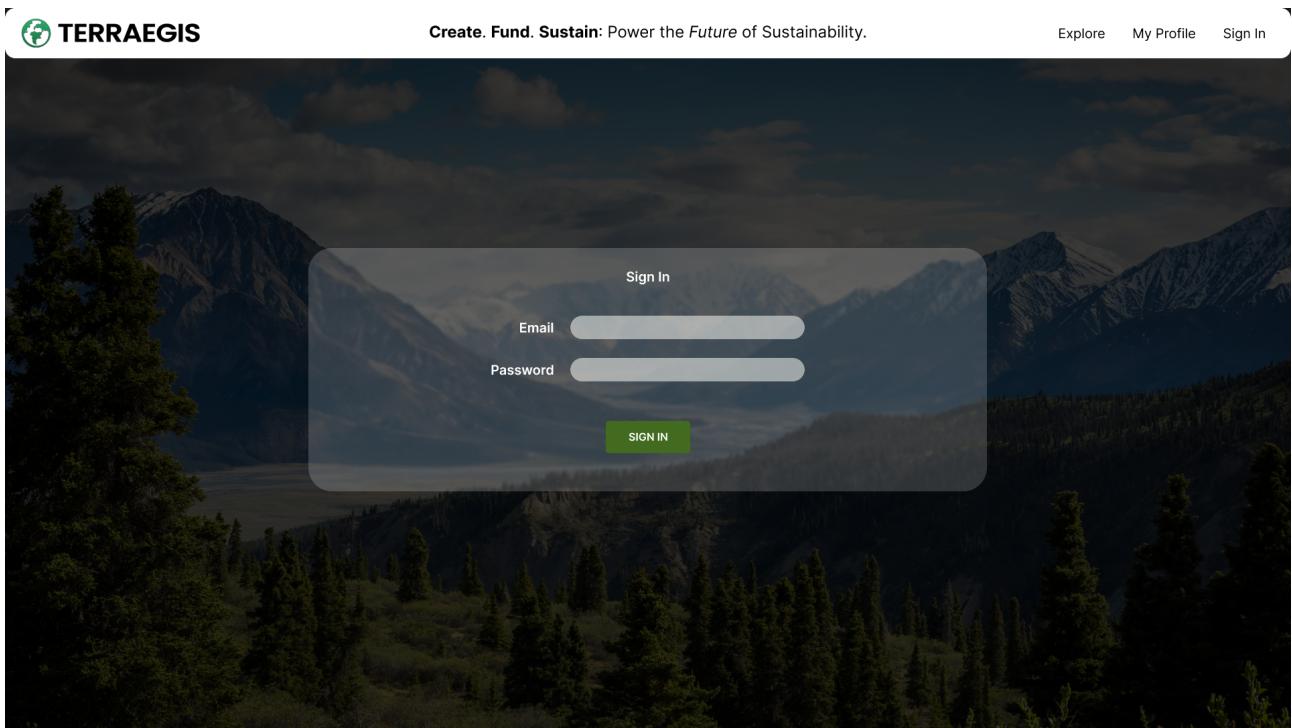


Figura 1.3: Mockup #1: Prototipo di schermata di Sign In

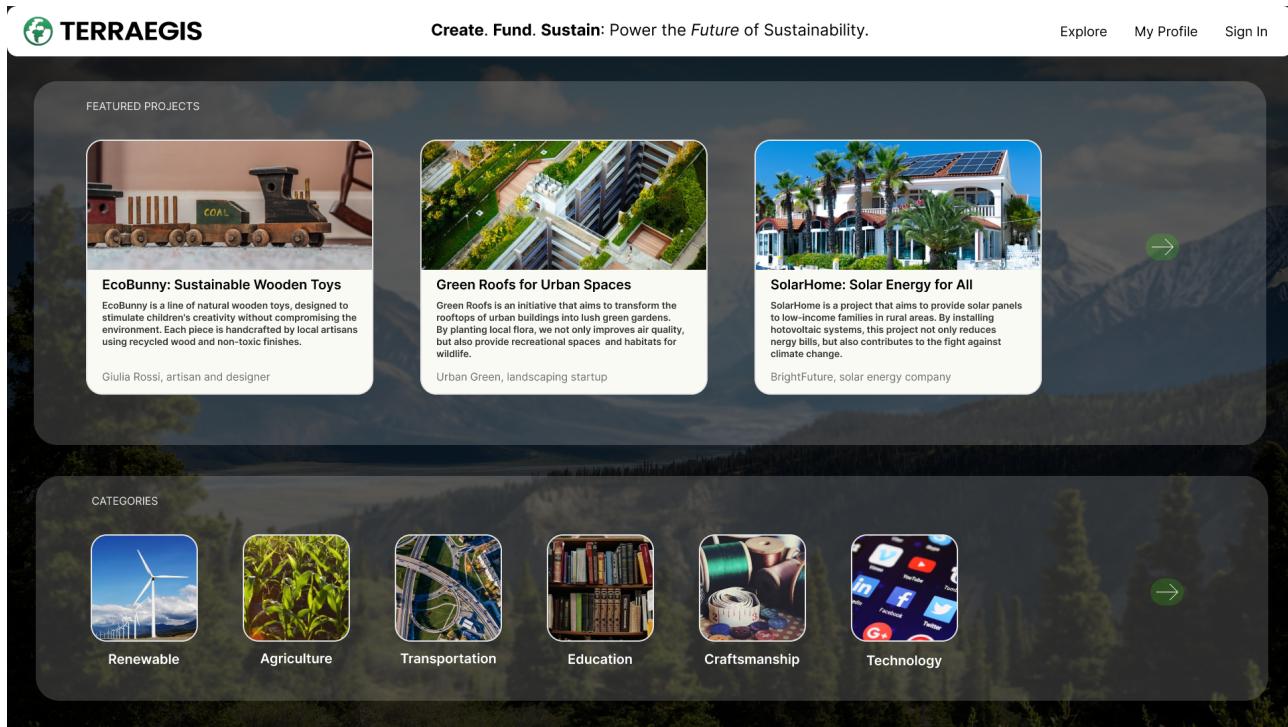


Figura 1.4: Mockup #2: Prototipo di pagina di esplorazione dei progetti

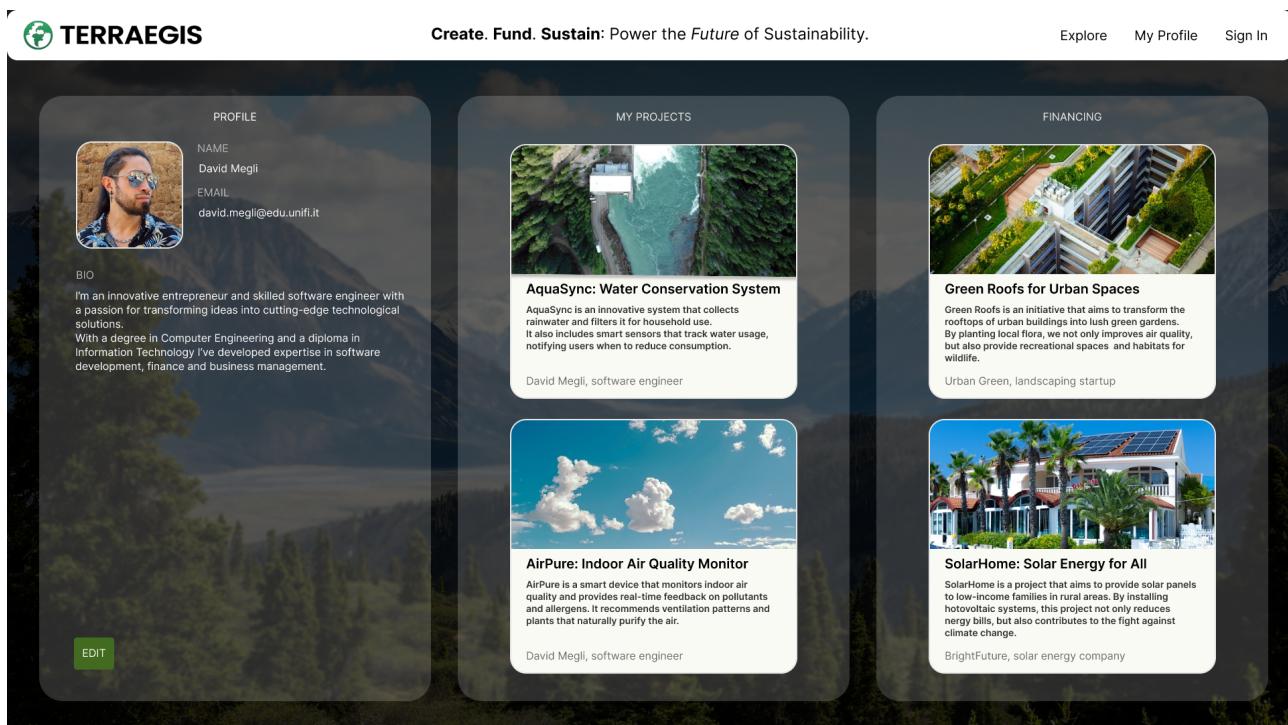


Figura 1.5: Mockup #3: Prototipo di pagina profilo

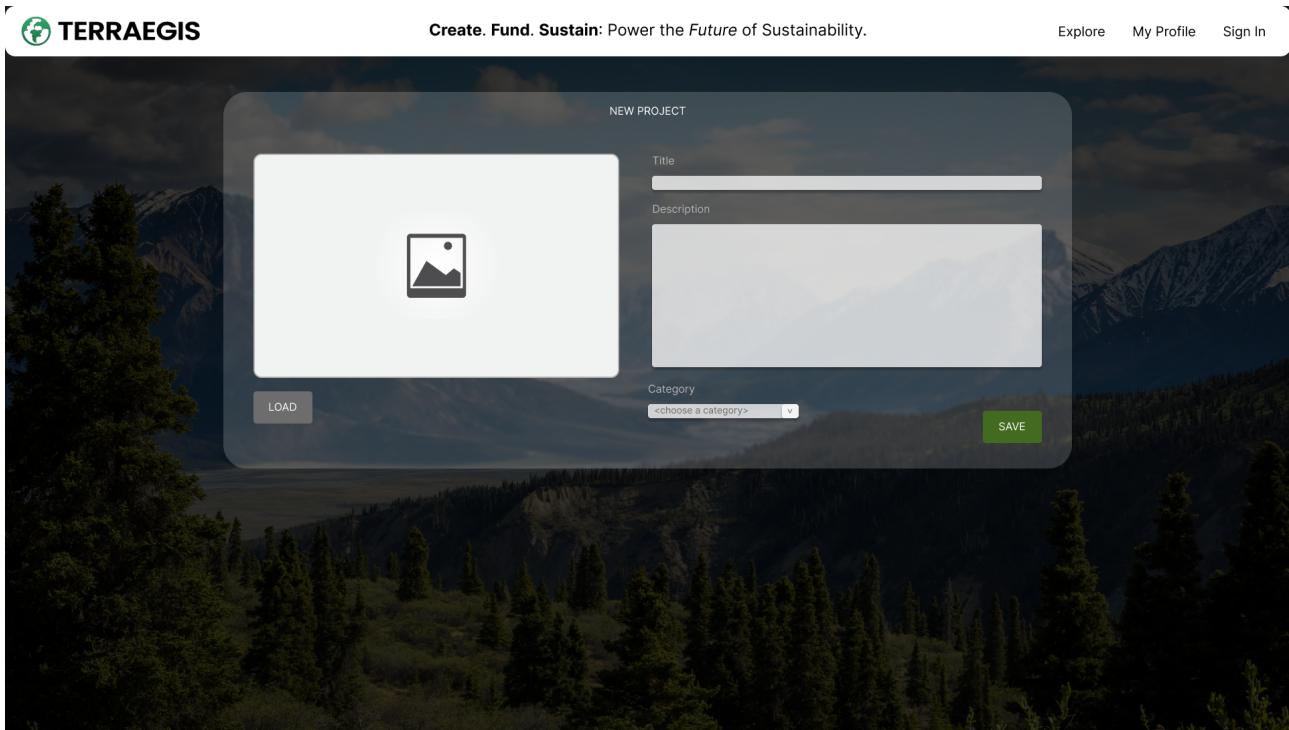


Figura 1.6: Mockup #4: Prototipo di pagina di creazione di un progetto

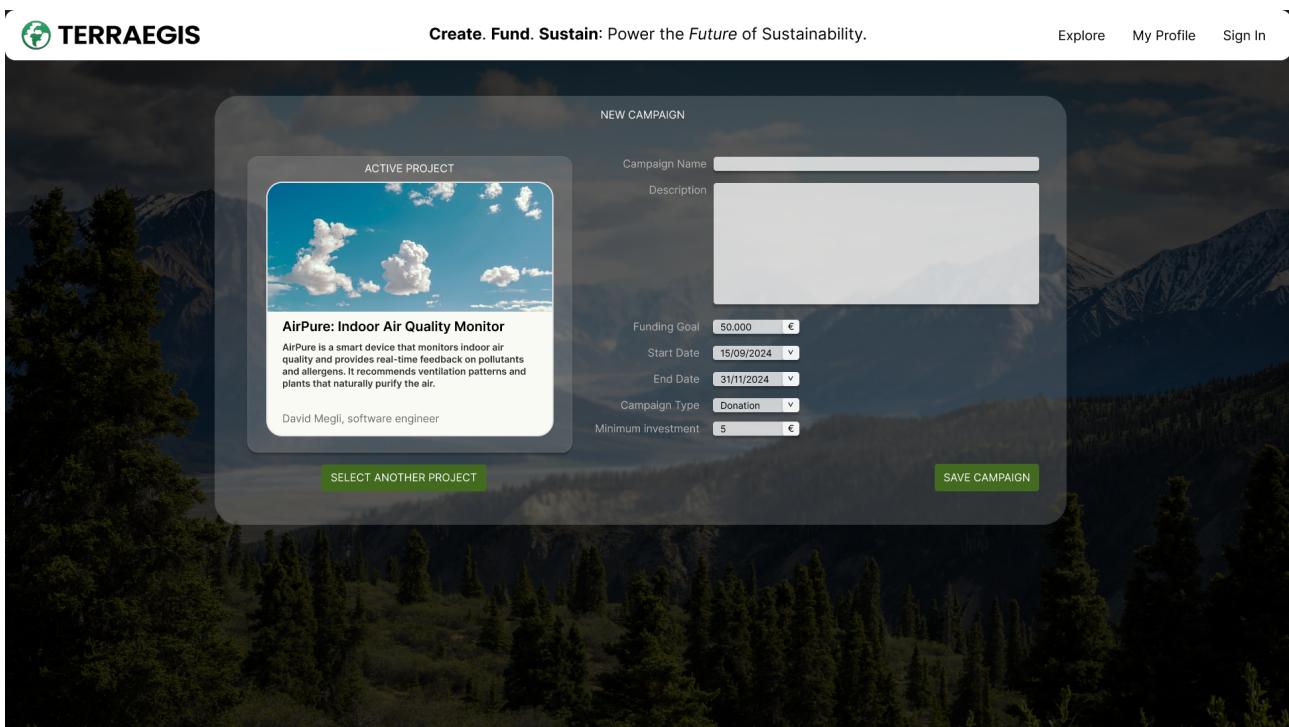


Figura 1.7: Mockup #5: Prototipo di pagina di creazione di una campagna di raccolta fondi

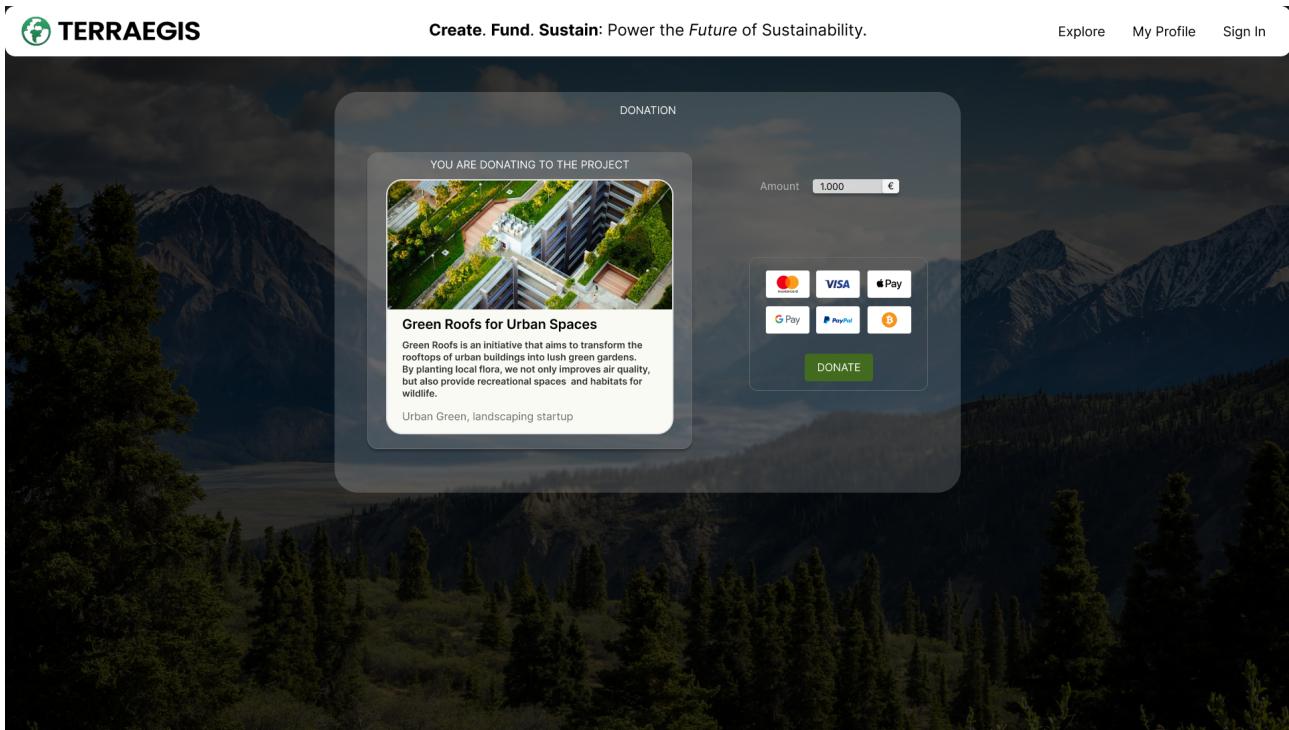


Figura 1.8: Mockup #6: Prototipo di pagina di donazione

1.6 Class diagram

Di seguito sono riportati i diagrammi di classe relativi ai quattro packages principali in cui è strutturato il lato backend della piattaforma, ovvero *Model*, *Repository*, *Service*, *Controller*. I Modelli sono i seguenti:

- **User**: rappresenta un utente, le sue credenziali e le sue informazioni di profilo
- **Project**: rappresenta un progetto e le sue informazioni
- **Category**: rappresenta una categoria a cui un progetto può appartenere
- **SustainabilityGoal**: ognuno rappresenta un obiettivo di sviluppo sostenibile
- **ProjectGoal**: rappresenta l'implementazione di un obiettivo di sviluppo sostenibile da parte di un progetto
- **Campaign**: rappresenta una campagna di raccolta fondi relativa ad un progetto

- **Reward:** rappresenta un premio e può essere associato esclusivamente ad una campagna di tipo "*Reward*"
- **Funding:** rappresenta un singolo finanziamento di un progetto da parte di un utente, se un utente finanzia lo stesso progetto più volte saranno presenti più *Funding*, ognuno per ogni transazione. Ad esso può essere associato un *Reward* se il finanziamento è stato effettuato tramite l'acquisto di un premio di una campagna premi.

Un utente può creare un numero qualsiasi di progetti, e finanziare un numero qualsiasi di progetti. Nel secondo caso verrà salvata ogni singola transazione. Ad un progetto è associata una categoria e possono essere associati molteplici obiettivi di sostenibilità. Un utente può avviare o programmare campagne di raccolta fondi per i propri progetti. Le campagne possono essere di tipo *Equity* (campagna investimenti tramite la quale gli utenti possono acquisire una quota di proprietà del progetto), *Donation* (campagna donazioni tramite cui gli utenti possono donare una qualsiasi somma di denaro) o *Reward* (campagna premi per cui il creatore del progetto offre premi che gli utenti possono acquistare al prezzo stabilito dal creatore).

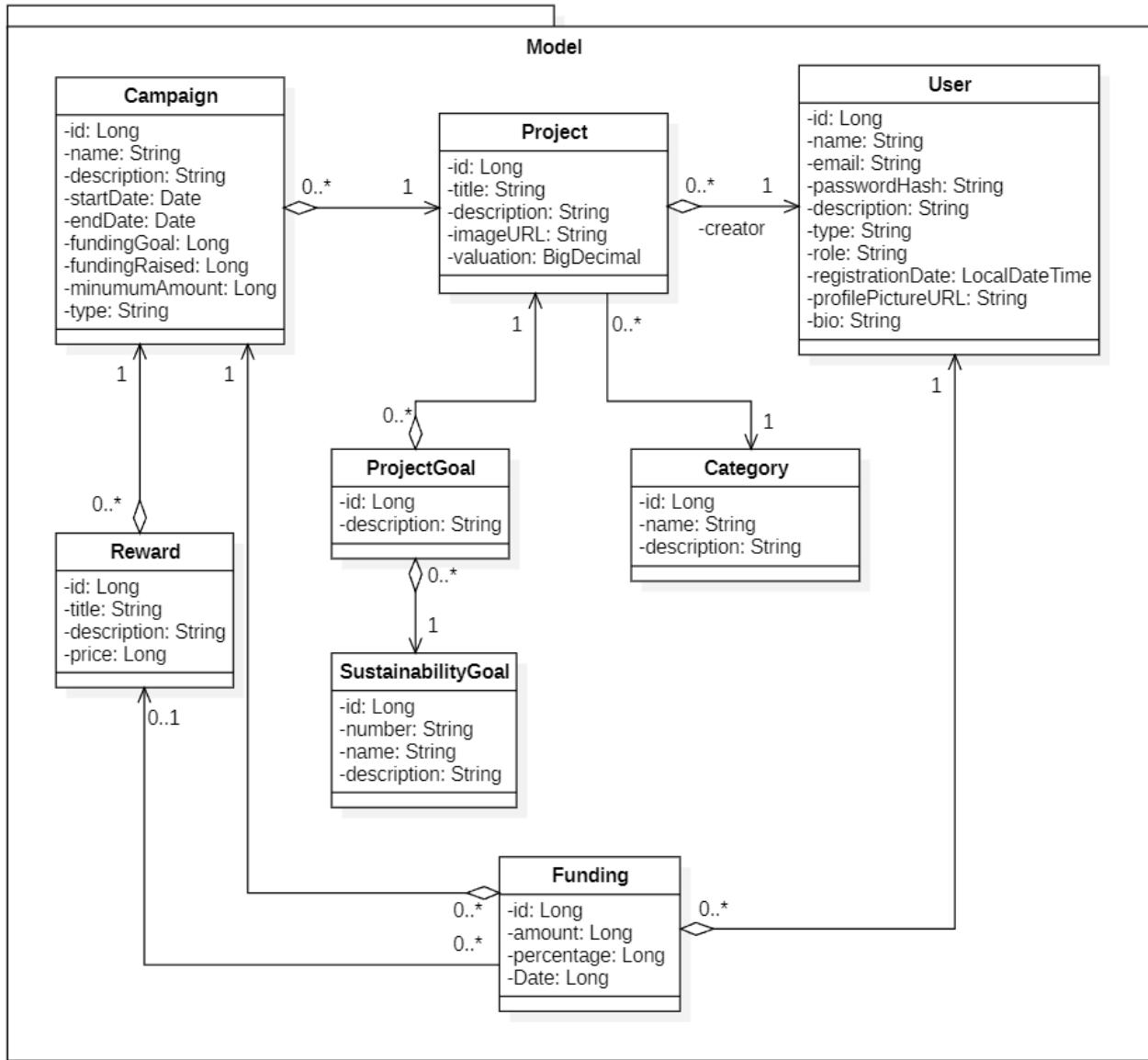


Figura 1.9: Model class diagram

In figura 1.9 sono schematizzate le classi del package *Model* e le loro relazioni.

In figura 1.10 sono schematizzate le classi del package *Repository*. Come si può vedere tutte le *repositories* sono interfacce che estendono l’interfaccia *JpaRepository* del framework JPA, ereditando i suoi metodi predefiniti per le operazioni CRUD (Create, Read, Update, Delete). A partire dai nomi dei metodi dichiarati all’interno delle interfacce dei repository vengono generate automaticamente le query derivate in JPA.

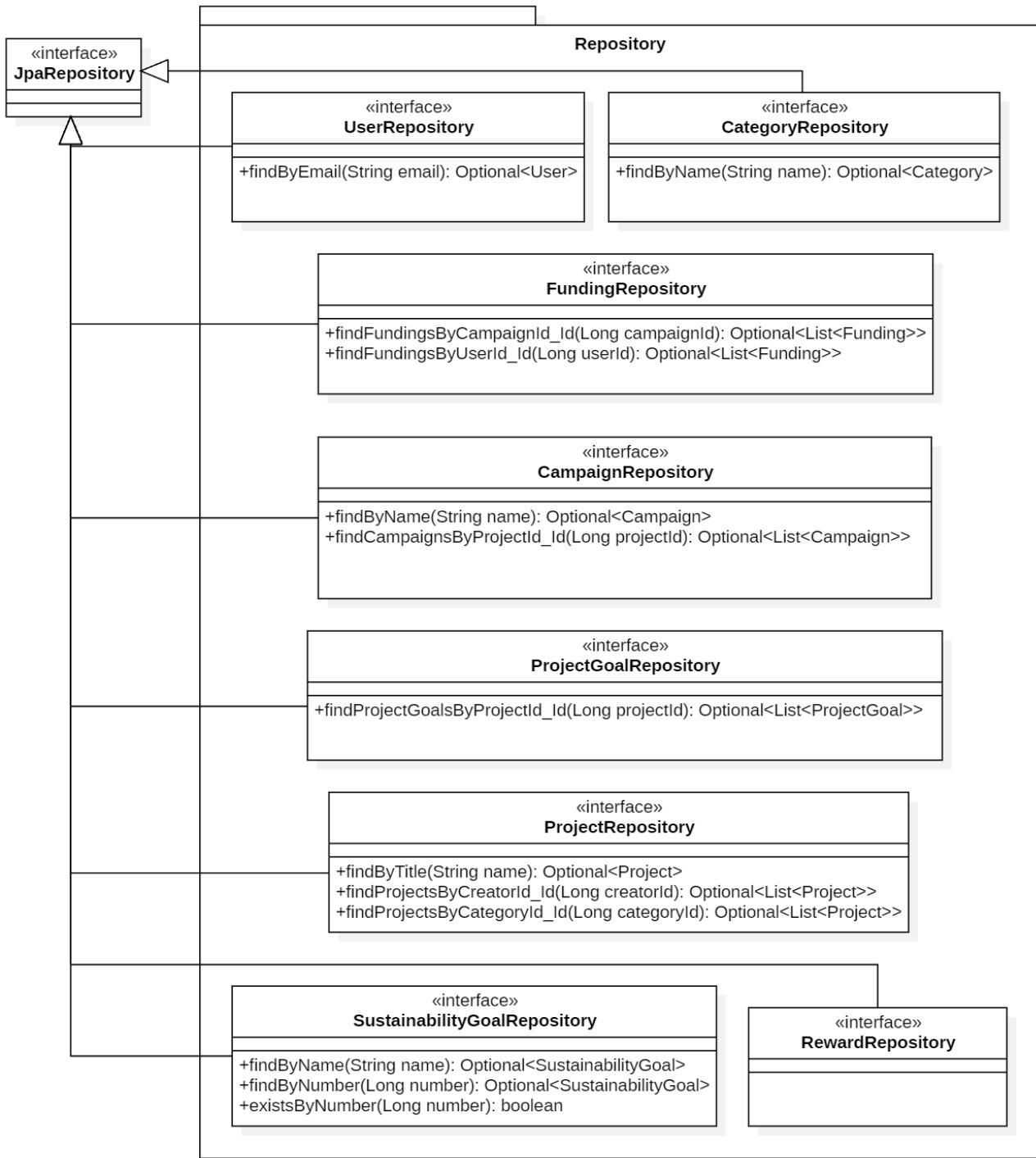


Figura 1.10: Repository class diagram

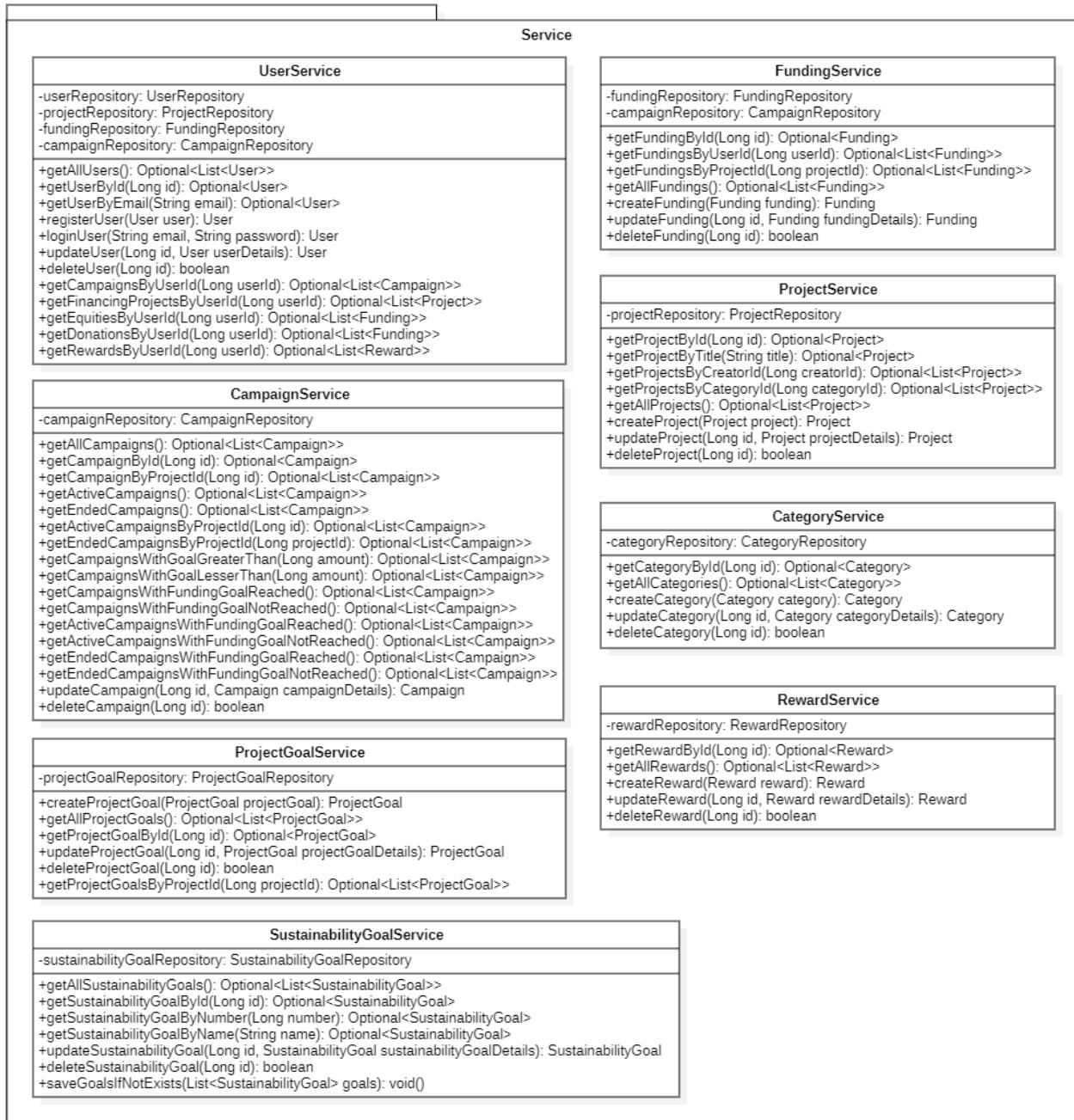


Figura 1.11: Service class diagram

Come nel diagramma 1.11 le classi del package *Service* implementano tutti i metodi necessari a scrivere dati ed effettuare query sul database tramite l'utilizzo delle repository. Questi metodi vengono usati dai controllers, le cui classi sono riportate in figura 1.12, i quali si occupano di effettuare il mapping delle chiamate HTTP provenienti dal lato frontend della piattaforma, in cui è implementata l'interfaccia grafica con cui interagisce l'utente finale. Sulle base delle richieste in arrivo i controllers forniscono il servizio richiesto tramite i services fornendo una risposta al

client (l’interfaccia frontend). Questo meccanismo è implementato secondo l’architettura API REST.

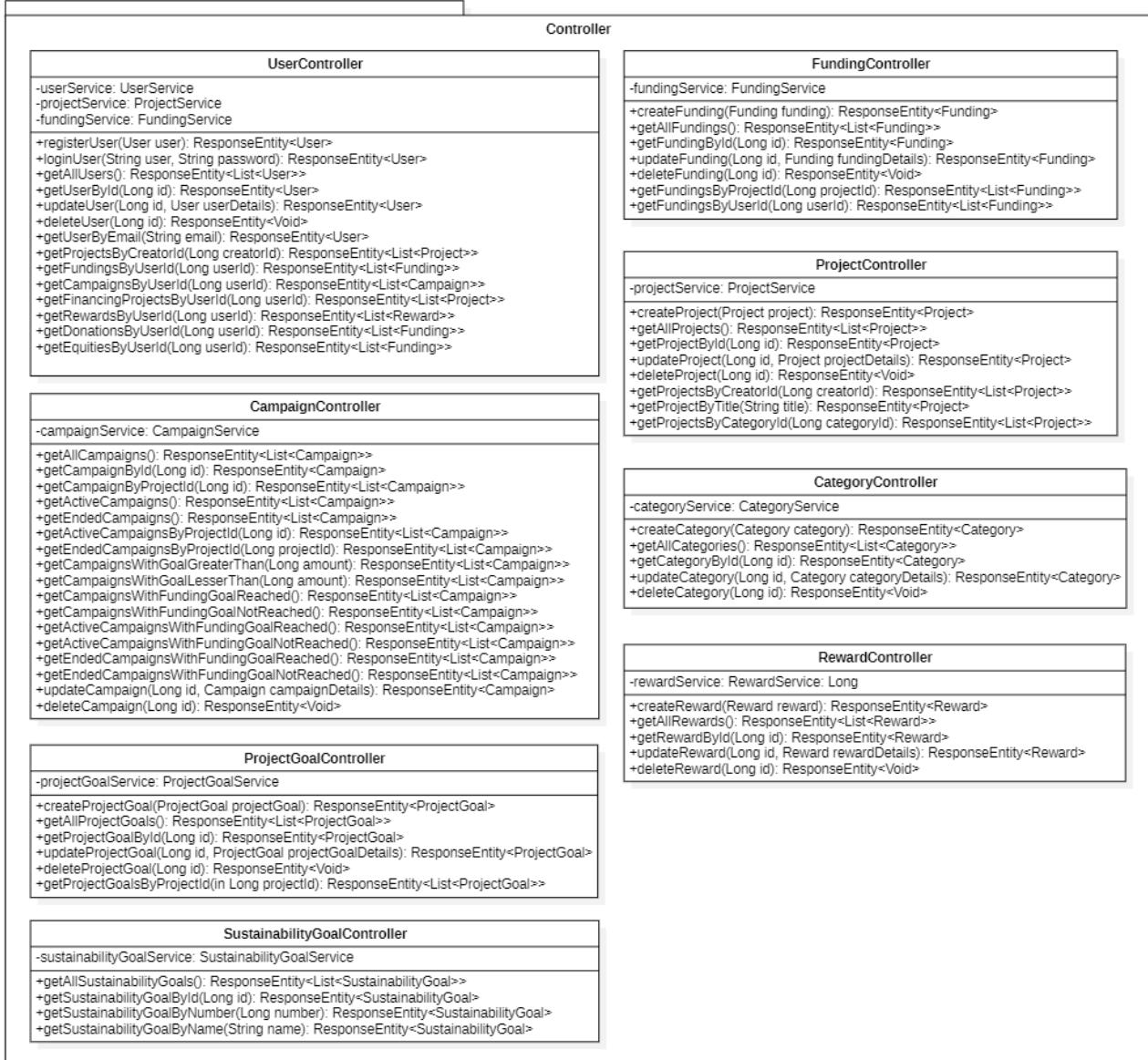


Figura 1.12: Controller class diagram

1.7 Entity Relationship diagram

In figura 1.13 è riportato il diagramma Entity Relationship della base dati. Il database è stato implementato tramite il RDBMS PostgreSQL, e come si può vedere nel Package dependency diagram le operazioni CRUD sono implementate dalla piattaforma tramite il servizio di ORM fornito dal middleware Hibernate. I modelli all’interno del package **Model** rappresentano

le entità tabellari implementate sul database tramite l'utilizzo delle annotazioni del framework JPA.

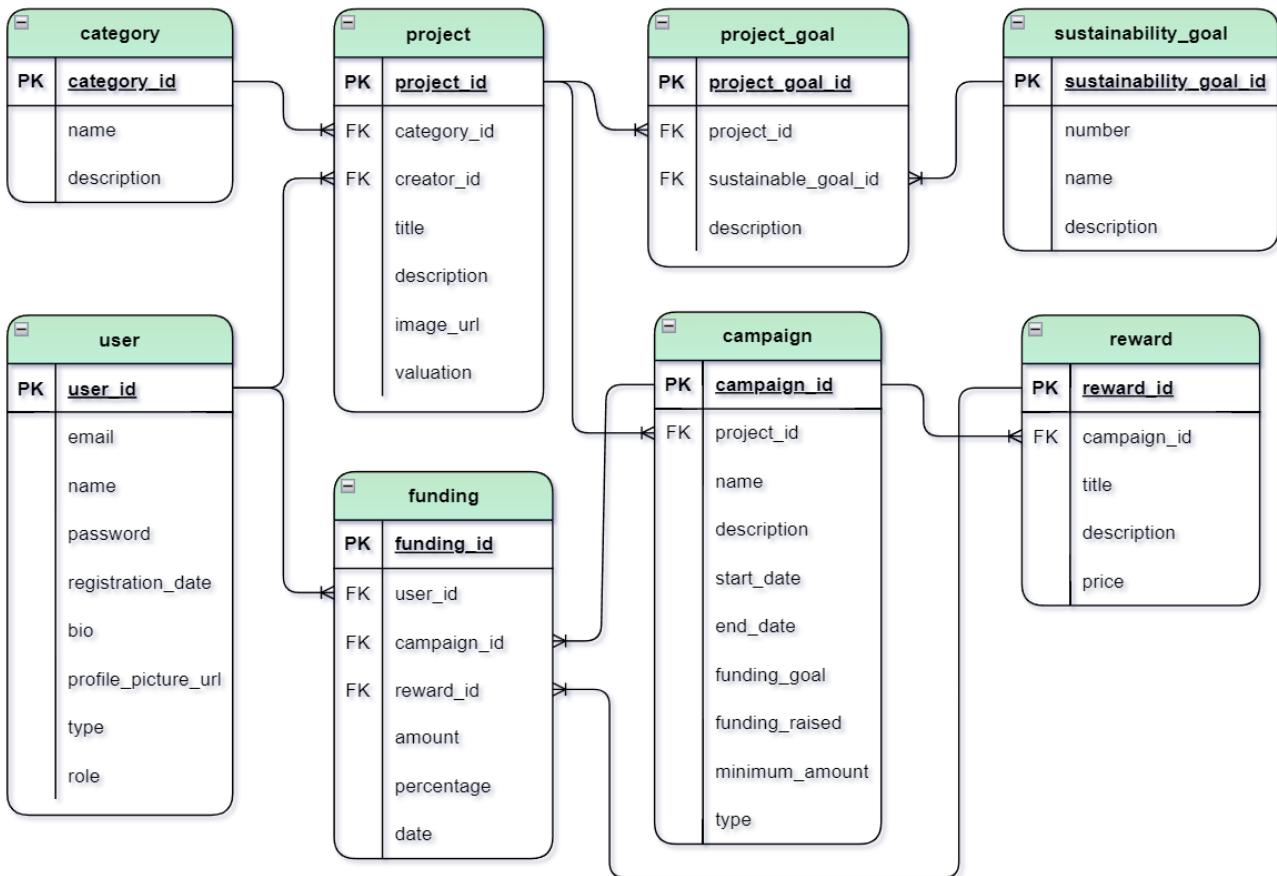


Figura 1.13: Entity Relationship diagram

1.8 Page Navigation Diagram

Il diagramma 1.14 rappresenta il diagramma di navigazione fra le pagine dell'interfaccia grafica della piattaforma. Come si vede, dalla pagina iniziale, l'utente può effettuare l'accesso (sign in) o creare un nuovo account (sign up). Una volta effettuato l'accesso l'utente può visualizzare il proprio profilo, e quindi modificarlo, visualizzare i propri progetti, e da qui crearene di nuovi, o modificare quelli esistenti, dopodiché può procedere alla visualizzazione, modifica e creazione di campagne relative ad un progetto. In alternativa l'utente può visualizzare i progetti presenti sulla piattaforma (questa operazione non richiede l'accesso), visualizzare i progetti che sta finanziando, e da qui visualizzare i dettagli di un singolo progetto e decidere se finanziarne uno tramite donazione, investimento o acquisto di premi. Queste operazioni sono

possibili esclusivamente se sono presenti delle campagne attive relative al progetto selezionato, rispettivamente di tipo donazione, investimento o premi. Una volta scelto il metodo di finanziamento disponibile l'utente può procedere al pagamento.

Di alcune di queste pagine sono stati realizzati riportati i mockup nella sezione 1.5.

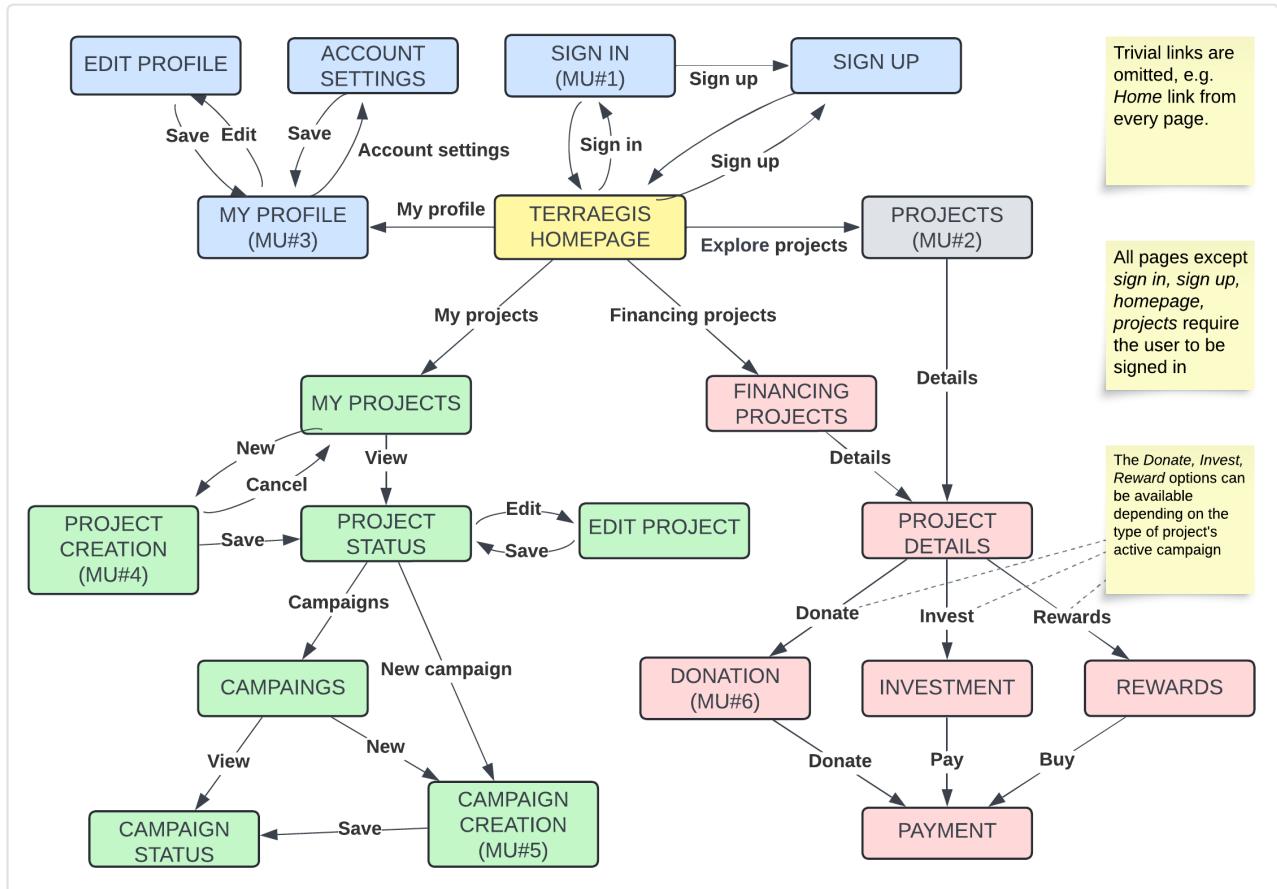


Figura 1.14: Page Navigation Diagram

1.9 Struttura della directory del progetto

Il progetto è strutturato in due sezioni principali: *main* dedicata al codice sorgente e *test* in cui è presente il codice relativo a tutti i test. Il codice è diviso in packages secondo una separazione funzionale. Oltre ai modelli, le repositories, i servizi e i controller, precedentemente discussi, il package *datainitializer* si occupa di inizializzare i dati, come ad esempio quelli relativi ai 17 obiettivi di sviluppo sostenibili, che sono prestabiliti, e il package *securityconfig* si occupa di gestire l'autenticazione.

I test sono stati implementati per controllers, repositories, models e services. Questi sono unit test volti a testare le funzionalità dei loro metodi.

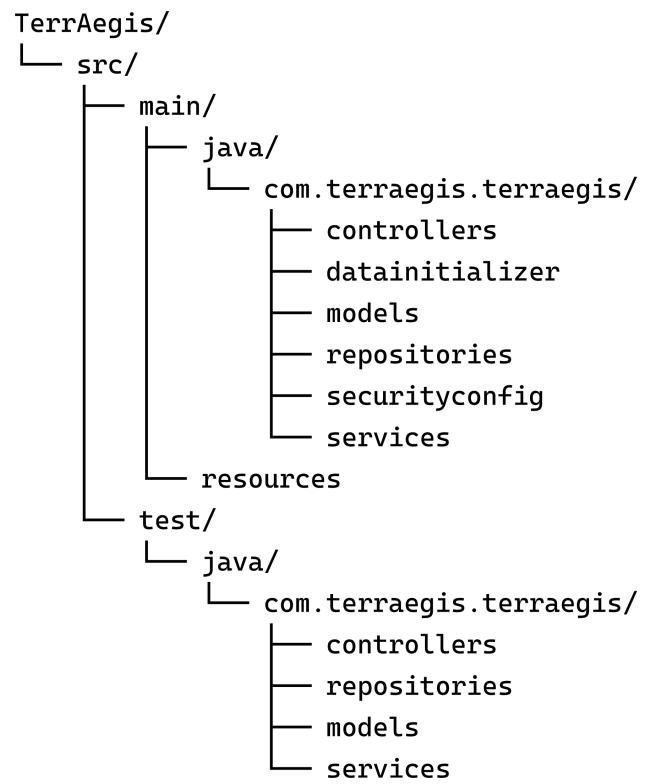


Figura 1.15: Struttura della directory

Implementazione

Il lato backend della piattaforma è stato implementato in Java, su ambiente di sviluppo IntelliJ IDEA, con l'uso dei framework Spring, Jakarta Persistence API e Hibernate. La business logic e la gestione dei dati sono implementati nei package Model, Repository, Service e Controller.

2.1 Model

Il Model rappresenta la struttura dei dati dell'applicazione ed è rappresentato da classi annotate con le annotazioni `@Entity` e `@Table` di JPA. Queste classi rappresentano le tabelle del database e contengono attributi che corrispondono alle colonne, anch'essi dichiarati con l'utilizzo delle annotazioni JPA `@Column` per le colonne, `@ManyToOne` e `@JoinColumn` per le relazioni e altre. Contengono anche metodi `getter` e `setter` e costruttori.

2.1.1 User

La classe ‘User’ rappresenta un utente all’interno del sistema. Ogni utente ha attributi come nome, email, password e un ruolo che definisce i suoi privilegi. Questa classe è essenziale per gestire l’autenticazione e l’autorizzazione degli utenti nella piattaforma.

2.1.2 Project

La classe ‘Project’ rappresenta un progetto creato da un utente nel sistema. Include attributi come titolo, descrizione, immagine, categoria, valutazione e un riferimento al creatore. I progetti sono il cuore della piattaforma, permettendo agli utenti di presentare idee e raccogliere fondi. La categorizzazione serve per permettere agli utenti in cerca di progetti da finanziare di filtrare i progetti presenti sulla piattaforma. La valutazione serve nel caso in cui l’utente abbia intenzione di avviare una campagna raccolta fondi basata su investimenti.

```

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "name", nullable = false)
    private String name;
    @Column(name = "description", length = 1000, nullable = false)
    private String description;
    @Column(name = "email", nullable = false, unique = true)
    private String email;
    @Column(name = "password", nullable = false)
    private String passwordHash;
    @Column(name = "type", nullable = false)
    private String type; // Person, Company
    @Column(name = "role", nullable = false) // "user" or "admin"
    private String role;
    @Column(name = "registration_date", nullable = false)
    private LocalDateTime registrationDate = LocalDateTime.now();
    @Column(name = "bio")
    private String bio;
    @Column(name = "profile_picture_url")
    private String profilePictureUrl;
    // constructors, getters, setters ...
}

```

Figura 2.1: Users class

2.1.3 Campaign

La classe ‘Campaign’ rappresenta una campagna di raccolta fondi associata a un progetto. Ogni campagna ha un obiettivo di raccolta, una durata e un progetto correlato, ovvero il progetto che si intende finanziare. Questa classe gestisce la logica di avvio, monitoraggio e completamento delle campagne. Le tipologie di campagna sono *Equity*, *Donation* e *Reward*, la prima tipologia permette l’acquisto di una percentuale del progetto basata sulla sua valutazione, la seconda consiste in semplici donazioni, mentre la terza richiede la creazione di premi associati alla campagna, che saranno acquistabili dagli utenti.

2.1.4 Funding

La classe ‘Funding’ rappresenta un finanziamento dato da un utente a una campagna. Include l’importo del finanziamento e il riferimento all’utente e alla campagna associata. Questa classe traccia i contributi finanziari degli utenti alle campagne relative ad un progetto.

Nel caso di campagne di tipo *Equity*, *percentage* sarà diverso da zero e corrisponderà alla per-

centuale di equity acquisita per la somma *amount*. Nel caso di campagne di tipo *Donation* la percentuale sarà 0. Nelle campagne di tipo *Reward* sarà presente un riferimento al premio acquistato.

2.1.5 SustainabilityGoal

La classe ‘SustainabilityGoal‘ rappresenta un obiettivo di sostenibilità. In fase di boot della piattaforma vengono istanziate 17 istanze corrispondenti ai 17 obiettivi di sviluppo sostenibile.

2.1.6 ProjectGoal

La classe ‘ProjectGoal‘ rappresenta un obiettivo specifico implementato all’interno di un progetto. Questa entità esiste allo scopo di poter associare molteplici obiettivi ad ogni progetto. Include un campo dedicato alla descrizione dettagliata di come il progetto si impegnerà a raggiungere l’obiettivo.

2.1.7 Reward

La classe ‘Reward‘ rappresenta una ricompensa offerta agli utenti che finanziano una campagna di tipo *Reward*. Le ricompense possono variare in base all’importo del finanziamento e possono includere prodotti, riconoscimenti o esperienze. Queste informazioni devono essere inserite nella descrizione.

2.1.8 Category

La classe ‘Category‘ rappresenta una categoria alla quale può appartenere un progetto (e.g. "Technology", "Art", o "Education"). Ogni progetto è associato a una categoria specifica, che aiuta a organizzare e filtrare i progetti all’interno della piattaforma.

2.2 Repository

Il Repository è responsabile dell’interazione diretta con il database. Utilizza JPA per eseguire operazioni CRUD (Create, Read, Update, Delete) e si occupa di fornire metodi per recuperare e manipolare le entità. Ognuna delle repositories è un’interfaccia che estende *JpaRepository* e

utilizza i metodi da essa forniti o ne definisce di nuovi. Di seguito riporto un esempio di Repository.

```
public interface ProjectRepository extends JpaRepository<Project, Long> {  
    Optional<Project> findByName(String name);  
    Optional<List<Project>> findProjectsByCreatorId_Id(Long creatorId);  
    Optional<List<Project>> findProjectsByCategoryId_Id(Long categoryId);  
}
```

Figura 2.2: ProjectRepository class

2.3 Service

Il Service si occupa della logica di business. Qui è implementata la logica di business e il flusso delle operazioni, delegando la persistenza dei dati al Repository. Per ogni classe sono riportati alcuni dei metodi non banali, seguiti da una breve descrizione delle loro implementazione e funzionalità.

2.3.1 UserService

Questa classe implementa i servizi legati all'utente, come quelli per ottenere la lista dei propri progetti, dei progetti che l'utente sta finanziando, i servizi di login, aggiornamento del profilo, disiscrizione ecc... . Questa classe è riportata in figura 2.3. Di seguito sono riportati alcuni dei metodi di questa classe.

- **User registerUser(User user):** Questo metodo si occupa della registrazione di un nuovo utente nel sistema. Riceve un oggetto User contenente le informazioni dell'utente come input. Il metodo convalida i dati forniti, come l'unicità dell'email, e quindi crea un nuovo record utente nel database. Se l'operazione ha successo, restituisce l'oggetto User appena creato.
- **User loginUser(String email, String password):** Questo metodo gestisce il processo di login di un utente. Riceve l'email e la password dell'utente come input e cerca un utente corrispondente nel database. Se viene trovata una corrispondenza e la password è

corretta, il metodo restituisce l'oggetto User corrispondente. In caso contrario, restituisce null o lancia un'eccezione per indicare che le credenziali non sono valide.

- **User updateUser(Long id, User userDetails):** Questo metodo aggiorna i dettagli di un utente esistente. Riceve l'ID dell'utente e un oggetto User contenente i nuovi dettagli come input. Il metodo verifica se l'utente esiste, quindi aggiorna i campi pertinenti (come nome, email, ecc.) nel record utente. Se l'operazione è completata con successo, restituisce l'oggetto User aggiornato; altrimenti, restituisce null o lancia un'eccezione.
- **boolean deleteUser(Long id):** Questo metodo elimina un utente dal sistema. Riceve l'ID dell'utente da eliminare come input. Il metodo verifica se l'utente esiste, quindi rimuove il record dal database. Se l'operazione ha successo, restituisce true; se l'utente non viene trovato o l'eliminazione fallisce, restituisce false.
- **Optional<List<Campaign>> getCampaignsById(Long userId):** Questo metodo recupera tutte le campagne create da un determinato utente. Riceve l'ID dell'utente come input e interroga il database per ottenere l'elenco delle campagne associate a quell'utente. Restituisce un Optional contenente una lista di campagne se trovate; se non ci sono campagne associate all'utente, restituisce un Optional.empty().
- **Optional<List<Project>> getFinancingProjectsById(Long userId):** Questo metodo recupera tutti i progetti che un utente sta finanziando. Riceve l'ID dell'utente come input e cerca nel database i progetti associati ai finanziamenti dell'utente. Restituisce un Optional contenente una lista di progetti se trovati; se non ci sono progetti finanziati dall'utente, restituisce un Optional.empty().
- **Optional<List<Funding>> getDonationsById(Long userId):** Questo metodo recupera tutte le donazioni effettuate da un utente. Riceve l'ID dell'utente come input e interroga il database per ottenere l'elenco delle donazioni associate a quell'utente. Restituisce un Optional contenente una lista di donazioni se trovate; se l'utente non ha effettuato donazioni, restituisce un Optional.empty().

```

@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private ProjectRepository projectRepository;
    @Autowired
    private FundingRepository fundingRepository;
    @Autowired
    private CampaignRepository campaignRepository;
    public Optional<List<Project>> getFinancingProjectsByUserId(Long userId) {
        Optional<List<Funding>> fundingsByUser = fundingRepository.findFundingsByUserId_Id(userId);
        List<Project> projects = new ArrayList<>();
        // get all campaigns ids in fundingsByUser
        List<Long> campaignIds = new ArrayList<>();
        fundingsByUser.ifPresent(fundings -> {
            fundings.forEach(funding -> {
                if (funding.getCampaignId() != null) {
                    campaignIds.add(funding.getCampaignId().getId());
                }
            });
        });
        List<Campaign> campaigns = new ArrayList<>();
        // get all campaigns by campaignIds
        campaignIds.forEach(campaignId -> {
            Optional<Campaign> campaign = campaignRepository.findById(campaignId);
            campaign.ifPresent(campaigns::add);
        });
        // get all projects ids in campaigns
        List<Long> projectIds = new ArrayList<>();
        campaigns.forEach(campaign -> {
            projectIds.add(campaign.getProject().getId());
        });
        // get all projects by projectIds
        projectIds.forEach(projectId -> {
            Optional<Project> project = projectRepository.findById(projectId);
            project.ifPresent(projects::add);
        });
        return Optional.of(projects);
    }
    public Optional<List<Project>> findProjectsByCreatorId(Long creatorId) { ... }
    public Optional<List<User>> getAllUsers() { return Optional.of(userRepository.findAll()); }
    public Optional<User> getUserById(Long id) { return userRepository.findById(id); }
    public Optional<User> getUserByEmail(String email) { return userRepository.findByEmail(email); }
    public User registerUser(User user) { return userRepository.save(user); }
    public User loginUser(String email, String password) {...}
    public User updateUser(Long id, User userDetails) {...}
    public boolean deleteUser(Long id) {...}
    public Optional<List<Funding>> getFundingsByUserId(Long userId) {...}
    public Optional<List<Funding>> getEquitiesByUserId(Long userId) {...}
    public Optional<List<Funding>> getDonationsByUserId(Long userId) {...}
    public Optional<List<Reward>> getRewardsByUserId(Long userId) {...}
}

```

Figura 2.3: UserService

2.3.2 ProjectService

- **Optional<Project> getProjectById(Long id)**: Questo metodo recupera i dettagli di un progetto specifico in base al suo ID. Riceve l'ID del progetto come input e interroga il database per cercare il progetto corrispondente. Se il progetto viene trovato, restituisce un Optional contenente l'oggetto Project; se il progetto non viene trovato, restituisce un Optional.empty().
- **Optional<List<Project>> getProjectsByCreatorId(Long creatorId)**: Questo metodo recupera tutti i progetti creati da un particolare utente, identificato dal suo ID. Riceve l'ID del creatore come input e interroga il database per ottenere la lista dei progetti associati a quell'utente. Restituisce un Optional contenente una lista di progetti se trovati; se l'utente non ha creato progetti, restituisce un Optional.empty().
- **Optional<List<Project>> getProjectsByCategoryId(Long categoryId)**: Questo metodo recupera tutti i progetti associati a una determinata categoria, identificata dal suo ID. Riceve l'ID della categoria come input e cerca nel database i progetti appartenenti a quella categoria. Restituisce un Optional contenente una lista di progetti se trovati; se non ci sono progetti associati a quella categoria, restituisce un Optional.empty().
- **Project createProject(Project project)**: Questo metodo si occupa della creazione di un nuovo progetto nel sistema. Riceve un oggetto Project contenente i dettagli del progetto come input, inclusi titolo, descrizione, creatore, categoria e valutazione. Il metodo salva il nuovo progetto nel database e restituisce l'oggetto Project creato, che include l'ID generato automaticamente.

2.3.3 CampaignService

- **Optional<List<Campaign>> getActiveCampaigns()**: Questo metodo recupera tutte le campagne attive nel sistema, ovvero quelle campagne che non sono ancora terminate. Interroga il database per ottenere la lista delle campagne il cui stato è attivo e che hanno una data di fine futura rispetto alla data corrente. Restituisce un Optional contenente una lista di campagne se trovate; se non ci sono campagne attive, restituisce un Optional.empty().

- **Optional<List<Campaign>> getActiveCampaignsByProjectId(Long projectId)**: Questo metodo recupera tutte le campagne attive associate a un determinato progetto, identificato dal suo ID. Riceve l'ID del progetto come input e cerca nel database le campagne attive collegate a quel progetto specifico. Restituisce un Optional contenente una lista di campagne se trovate; se non ci sono campagne attive per quel progetto, restituisce un Optional.empty().
- **Optional<List<Campaign>> getCampaignsWithFundingGoalReached()**: Questo metodo recupera tutte le campagne che hanno raggiunto il loro obiettivo di finanziamento. Interroga il database per ottenere la lista delle campagne il cui importo totale raccolto è uguale o superiore all'obiettivo di finanziamento prefissato. Restituisce un Optional contenente una lista di campagne se trovate; se nessuna campagna ha raggiunto l'obiettivo di finanziamento, restituisce un Optional.empty().

2.3.4 FundingService

- **Optional<List<Funding>> getFundingsByUserId(Long userId)**: Questo metodo recupera tutti i finanziamenti effettuati da un determinato utente, identificato dal suo ID. Utilizza l'ID dell'utente per interrogare il database e ottenere una lista di finanziamenti associati a quell'utente specifico. Restituisce un Optional contenente una lista di finanziamenti se trovati; se non ci sono finanziamenti per quell'utente, restituisce un Optional.empty().
- **Optional<List<Funding>> getFundingsByProjectId(Long projectId)**: Questo metodo recupera tutti i finanziamenti associati a un determinato progetto, identificato dal suo ID. Utilizza l'ID del progetto per interrogare il database e ottenere una lista di finanziamenti destinati a quel progetto specifico. Restituisce un Optional contenente una lista di finanziamenti se trovati; se non ci sono finanziamenti per quel progetto, restituisce un Optional.empty().
- **Funding createFunding(Funding funding)**: Questo metodo crea un nuovo finanziamento nel sistema. Riceve un oggetto Funding contenente i dettagli del finanziamento, come l'importo, l'utente che lo ha effettuato, e il progetto a cui è destinato. Dopo aver

salvato il finanziamento nel database, restituisce l'oggetto `Funding` appena creato, inclusi eventuali dettagli generati automaticamente come un ID unico.

Quelle appena descritte sono le classi più rilevanti del package `Service`, con i loro principali metodi. Le altre classi e gli altri metodi che non sono qui riportati si occupano delle operazioni CRUD (Create, Read, Update, Delete).

2.4 Controller

Il Controller gestisce le richieste HTTP e coordina le interazioni con il lato frontend. Si occupa di ricevere le richieste provenienti dall'interfaccia utente, chiamare i servizi appropriati e restituire le risposte. Di seguito sono riportati i principali endpoint di ogni classe e uno snippet della classe `UserController`.

2.4.1 UserController

- POST `/api/users/login`: Autentica un utente. Questo metodo riceve le credenziali dell'utente (username e password) in formato JSON e verifica la validità tramite il servizio di autenticazione. Se le credenziali sono corrette, restituisce un token di autenticazione e uno stato HTTP 200; in caso contrario, restituisce un errore 401 (Unauthorized).
- POST `/api/users/register`: Registra un nuovo utente. Questo metodo riceve i dettagli del nuovo utente (come username, email, password) in formato JSON e li passa al servizio di registrazione. Se la registrazione è completata con successo, restituisce i dettagli dell'utente appena creato con uno stato HTTP 201; altrimenti, restituisce un errore in caso di fallimento (es. 400 Bad Request se i dati sono invalidi).
- GET `/api/users/<id>/projects`: Recupera tutti i progetti creati da un utente specifico. Il controller utilizza l'ID dell'utente per interrogare il servizio e ottenere l'elenco dei progetti associati. Restituisce l'elenco dei progetti con uno stato HTTP 200 se i progetti vengono trovati; altrimenti, restituisce un errore 404 se l'utente non esiste o non ha progetti.

```

@RestController
@RequestMapping("/api/users")
public class UserController {
    @Autowired
    private UserService userService;
    @Autowired
    private ProjectService projectService;
    @Autowired
    private FundingService fundingService;
    // create a new user
    @PostMapping("/register")
    public ResponseEntity<User> registerUser(@RequestBody User user) {
        User createdUser = userService.registerUser(user);
        return ResponseEntity.ok(createdUser);
    }
    // login a user
    @PostMapping("/login")
    public ResponseEntity<User> loginUser(@RequestBody String user, @RequestBody String password) {
        User loggedInUser = userService.loginUser(user, password);
        if (loggedInUser != null) {
            return ResponseEntity.ok(loggedInUser);
        } else {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
        }
    }
    // update a user
    @PutMapping("/update/{id}")
    public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User userDetails) {
        User updatedUser = userService.updateUser(id, userDetails);
        if (updatedUser != null) {
            return ResponseEntity.ok(updatedUser);
        } else {
            return ResponseEntity.notFound().build();
        }
    }
    // get a user by email
    @GetMapping("/email/{email}")
    public ResponseEntity<User> getUserByEmail(@PathVariable String email) {
        Optional<User> user = userService.getUserByEmail(email);
        return user.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());
    }
    // get all projects created by a user
    @GetMapping("/{id}/projects")
    public ResponseEntity<List<Project>> getProjectsByCreatorId(@PathVariable Long creatorId) {
        Optional<List<Project>> projects = projectService.getProjectsByCreatorId(creatorId);
        return projects.map(ResponseEntity::ok).orElseGet(() ->
            ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build());
    }
    // get all fundings by a user
    @GetMapping("/{id}/fundings")
    public ResponseEntity<List<Funding>> getFundingsByUserId(@PathVariable Long userId) {
        Optional<List<Funding>> fundings = fundingService.getFundingsByUserId(userId);
        return fundings.map(ResponseEntity::ok).orElseGet(() ->
            ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build());
    }
    //...
}

```

Figura 2.4: UserController

- GET /api/users/<**id**>/**fundings**: Recupera tutte le operazioni di finanziamento fatte da un utente specifico. Questo metodo utilizza l'ID dell'utente per ottenere l'elenco dei finanziamenti effettuati dall'utente, restituendo l'elenco con uno stato HTTP 200 se i dati vengono trovati; altrimenti, restituisce un errore 404 se l'utente non esiste o non ha finanziamenti.
- GET /api/users/<**id**>/**campaigns**: Recupera tutte le campagne create o supportate da un utente specifico. Utilizzando l'ID dell'utente, il controller interroga il servizio per ottenere l'elenco delle campagne associate. Restituisce l'elenco con uno stato HTTP 200 se i dati vengono trovati; altrimenti, restituisce un errore 404 se l'utente non esiste o non ha campagne.
- GET /api/users/<**id**>/**financing-projects**: Recupera tutti i progetti finanziati da un utente specifico. Il controller utilizza l'ID dell'utente per ottenere l'elenco dei progetti che l'utente ha finanziato. Se vengono trovati progetti, l'elenco viene restituito con uno stato HTTP 200; altrimenti, restituisce un errore 404 se l'utente non esiste o non ha finanziato alcun progetto.
- POST /api/users/**create**: Crea un nuovo utente. Questo metodo riceve i dettagli dell'utente come input (in formato JSON), delega la creazione dell'utente a un servizio, e restituisce l'utente appena creato con uno stato HTTP 200 se l'operazione ha successo.
- GET /api/users/<**id**>: Recupera i dettagli di un utente specifico in base al suo ID. Il controller verifica l'esistenza dell'utente e restituisce le informazioni, oppure un errore 404 se l'utente non viene trovato.
- PUT /api/users/**update**/<**id**>: Aggiorna i dettagli di un utente esistente. Il metodo riceve i nuovi dati dell'utente, valida l'input, e invia i dati al servizio per l'aggiornamento. Se l'operazione è completata con successo, restituisce l'utente aggiornato.
- DELETE /api/users/**delete**/<**id**>: Elimina un utente. Se l'utente esiste, viene eliminato dal sistema, altrimenti il controller restituisce un errore 404.

- GET /api/users/**email**/**<email>**: Recupera un utente in base al suo indirizzo email. Il controller interroga il servizio per ottenere l'utente e restituisce i dettagli, oppure restituisce un errore 404 se l'utente non viene trovato.

2.4.2 ProjectController

- GET /api/projects/**creator**/**creatorId**: Recupera tutti i progetti creati da un particolare utente (creatore). Questo endpoint utilizza l'ID del creatore per interrogare il servizio e ottenere l'elenco dei progetti che l'utente ha creato. Se i progetti vengono trovati, l'elenco viene restituito con uno stato HTTP 200. In caso contrario, se l'utente non esiste o non ha creato alcun progetto, restituisce un errore 404.
- GET /api/projects/**category**/**categoryId**: Recupera tutti i progetti che appartengono a una determinata categoria. Il controller utilizza l'ID della categoria per interrogare il servizio e ottenere l'elenco dei progetti associati a quella categoria. Se vengono trovati progetti appartenenti a quella categoria, l'elenco viene restituito con uno stato HTTP 200. Altrimenti, restituisce un errore 404 se la categoria non esiste o non ha progetti associati.
- POST /api/projects/**create**: Crea un nuovo progetto. Questo metodo riceve i dettagli del progetto come input (in formato JSON), delega la creazione del progetto a un servizio, e restituisce il progetto appena creato con uno stato HTTP 200 se l'operazione ha successo.
- GET /api/projects/**<id>**: Recupera i dettagli di un progetto specifico in base al suo ID. Il controller verifica l'esistenza del progetto e restituisce le informazioni, oppure un errore 404 se il progetto non viene trovato.
- PUT /api/projects/**update**/**<id>**: Aggiorna i dettagli di un progetto esistente. Il metodo riceve i nuovi dati del progetto, valida l'input, e invia i dati al servizio per l'aggiornamento. Se l'operazione è completata con successo, restituisce il progetto aggiornato.
- DELETE /api/projects/**delete**/**<id>**: Elimina un progetto. Se il progetto esiste, viene eliminato dal sistema, altrimenti il controller restituisce un errore 404.

- GET /api/projects/**creator**/**<creatorId>**: Recupera tutti i progetti creati da un particolare utente. Il controller interroga il servizio per ottenere i progetti e li restituisce all’utente.
- GET /api/projects/**title**/**<title>**: Recupera un progetto in base al suo titolo. Il controller interroga il servizio per ottenere il progetto e restituisce i dettagli, oppure restituisce un errore 404 se il progetto non viene trovato.
- GET /api/projects/**category**/**<categoryId>**: Recupera tutti i progetti associati a una particolare categoria. Il controller interroga il servizio per ottenere i progetti e li restituisce all’utente.

2.4.3 CampaignController

- GET /api/campaigns/**active**: Recupera tutte le campagne attualmente attive. Questo endpoint interroga il servizio per ottenere l’elenco delle campagne che non sono ancora terminate e che stanno ancora raccogliendo fondi. Se vengono trovate campagne attive, l’elenco viene restituito con uno stato HTTP 200. Se non ci sono campagne attive, restituisce un elenco vuoto con uno stato HTTP 200.
- GET /api/campaigns/**ended**: Recupera tutte le campagne che sono già terminate. Il controller utilizza questo endpoint per ottenere l’elenco delle campagne che hanno superato la data di fine prevista. Se vengono trovate campagne terminate, l’elenco viene restituito con uno stato HTTP 200. Se non ci sono campagne terminate, restituisce un elenco vuoto con uno stato HTTP 200.
- GET /api/campaigns/**funding-reached**: Recupera tutte le campagne che hanno raggiunto o superato il loro obiettivo di finanziamento. Questo endpoint interroga il servizio per ottenere le campagne che hanno raccolto fondi sufficienti per soddisfare o superare l’importo richiesto. Se vengono trovate campagne con obiettivo di finanziamento raggiunto, l’elenco viene restituito con uno stato HTTP 200. Se non ci sono campagne che hanno raggiunto l’obiettivo, restituisce un elenco vuoto con uno stato HTTP 200.
- POST /api/campaigns/**create**: Crea una nuova campagna. Questo metodo riceve i dettagli della campagna come input (in formato JSON), delega la creazione della campagna

gna a un servizio, e restituisce la campagna appena creata con uno stato HTTP 200 se l'operazione ha successo.

- GET /api/campaigns/<**id**>: Recupera i dettagli di una campagna specifica in base al suo ID. Il controller verifica l'esistenza della campagna e restituisce le informazioni, oppure un errore 404 se la campagna non viene trovata.
- PUT /api/campaigns/**update**/<**id**>: Aggiorna i dettagli di una campagna esistente. Il metodo riceve i nuovi dati della campagna, valida l'input, e invia i dati al servizio per l'aggiornamento. Se l'operazione è completata con successo, restituisce la campagna aggiornata.
- DELETE /api/campaigns/**delete**/<**id**>: Elimina una campagna. Se la campagna esiste, viene eliminata dal sistema, altrimenti il controller restituisce un errore 404.
- GET /api/campaigns/**creator**/<**creatorId**>: Recupera tutte le campagne create da un particolare utente. Il controller interroga il servizio per ottenere le campagne e le restituisce all'utente.

2.4.4 FundingController

- POST /api/fundings/create: Crea un nuovo finanziamento. Questo metodo riceve i dettagli del finanziamento come input (in formato JSON), delega la creazione del finanziamento a un servizio, e restituisce il finanziamento appena creato con uno stato HTTP 200 se l'operazione ha successo.
- GET /api/fundings/<**id**>: Recupera i dettagli di un finanziamento specifico in base al suo ID. Il controller verifica l'esistenza del finanziamento e restituisce le informazioni, oppure un errore 404 se il finanziamento non viene trovato.
- PUT /api/fundings/**update**/<**id**>: Aggiorna i dettagli di un finanziamento esistente. Il metodo riceve i nuovi dati del finanziamento, valida l'input, e invia i dati al servizio per l'aggiornamento. Se l'operazione è completata con successo, restituisce il finanziamento aggiornato.

- **DELETE /api/fundings/**delete**/**<id>**:** Elimina un finanziamento. Se il finanziamento esiste, viene eliminato dal sistema, altrimenti il controller restituisce un errore 404.
- **GET /api/fundings/**project**/**<projectId>**:** Recupera tutti i finanziamenti associati a un progetto specifico. Il controller interroga il servizio per ottenere i finanziamenti e li restituisce all'utente.

2.4.5 SustainabilityGoalController

- **POST /api/sustainability-goals/**create**:** Crea un nuovo obiettivo di sostenibilità. Questo metodo riceve i dettagli dell'obiettivo come input (in formato JSON), delega la creazione dell'obiettivo a un servizio, e restituisce l'obiettivo appena creato con uno stato HTTP 200 se l'operazione ha successo.
- **GET /api/sustainability-goals/**<id>**:** Recupera i dettagli di un obiettivo di sostenibilità specifico in base al suo ID. Il controller verifica l'esistenza dell'obiettivo e restituisce le informazioni, oppure un errore 404 se l'obiettivo non viene trovato.
- **PUT /api/sustainability-goals/**update**/**<id>**:** Aggiorna i dettagli di un obiettivo di sostenibilità esistente. Il metodo riceve i nuovi dati dell'obiettivo, valida l'input, e invia i dati al servizio per l'aggiornamento. Se l'operazione è completata con successo, restituisce l'obiettivo aggiornato.
- **DELETE /api/sustainability-goals/**delete**/**<id>**:** Elimina un obiettivo di sostenibilità. Se l'obiettivo esiste, viene eliminato dal sistema, altrimenti il controller restituisce un errore 404.
- **GET /api/sustainability-goals/**project**/**<projectId>**:** Recupera tutti gli obiettivi di sostenibilità associati a un progetto specifico. Il controller interroga il servizio per ottenere gli obiettivi e li restituisce all'utente.

2.4.6 ProjectGoalController

- **POST /api/project-goals/**create**:** Crea un nuovo obiettivo di progetto. Questo metodo riceve i dettagli dell'obiettivo come input (in formato JSON), delega la creazione dell'o-

biettivo a un servizio, e restituisce l'obiettivo appena creato con uno stato HTTP 200 se l'operazione ha successo.

- GET /api/project-goals/<**id**>: Recupera i dettagli di un obiettivo di progetto specifico in base al suo ID. Il controller verifica l'esistenza dell'obiettivo e restituisce le informazioni, oppure un errore 404 se l'obiettivo non viene trovato.
- PUT /api/project-goals/**update**/<**id**>: Aggiorna i dettagli di un obiettivo di progetto esistente. Il metodo riceve i nuovi dati dell'obiettivo, valida l'input, e invia i dati al servizio per l'aggiornamento. Se l'operazione è completata con successo, restituisce l'obiettivo aggiornato.
- DELETE /api/project-goals/**delete**/<**id**>: Elimina un obiettivo di progetto. Se l'obiettivo esiste, viene eliminato dal sistema, altrimenti il controller restituisce un errore 404.
- GET /api/project-goals/**project**/<**projectId**>: Recupera tutti gli obiettivi di progetto associati a un progetto specifico. Il controller interroga il servizio per ottenere gli obiettivi e li restituisce all'utente.

2.4.7 RewardController

- POST /api/rewards/**create**: Crea una nuova ricompensa. Questo metodo riceve i dettagli della ricompensa come input (in formato JSON), delega la creazione della ricompensa a un servizio, e restituisce la ricompensa appena creata con uno stato HTTP 200 se l'operazione ha successo.
- GET /api/rewards/<**id**>: Recupera i dettagli di una ricompensa specifica in base al suo ID. Il controller verifica l'esistenza della ricompensa e restituisce le informazioni, oppure un errore 404 se la ricompensa non viene trovata.
- PUT /api/rewards/**update**/<**id**>: Aggiorna i dettagli di una ricompensa esistente. Il metodo riceve i nuovi dati della ricompensa, valida l'input, e invia i dati al servizio per l'aggiornamento. Se l'operazione è completata con successo, restituisce la ricompensa aggiornata.

- **DELETE /api/rewards/**delete**/**<id>**:** Elimina una ricompensa. Se la ricompensa esiste, viene eliminata dal sistema, altrimenti il controller restituisce un errore 404.
- **GET /api/rewards/**project**/**<projectId>**:** Recupera tutte le ricompense associate a un progetto specifico. Il controller interroga il servizio per ottenere le ricompense e le restituisce all'utente.

2.4.8 CategoryController

- **POST /api/categories/**create**:** Crea una nuova categoria. Questo metodo riceve i dettagli della categoria come input (in formato JSON), delega la creazione della categoria a un servizio, e restituisce la categoria appena creata con uno stato HTTP 200 se l'operazione ha successo.
- **GET /api/categories/**<id>**:** Recupera i dettagli di una categoria specifica in base al suo ID. Il controller verifica l'esistenza della categoria e restituisce le informazioni, oppure un errore 404 se la categoria non viene trovata.
- **PUT /api/categories/**update**/**<id>**:** Aggiorna i dettagli di una categoria esistente. Il metodo riceve i nuovi dati della categoria, valida l'input, e invia i dati al servizio per l'aggiornamento. Se l'operazione è completata con successo, restituisce la categoria aggiornata.
- **DELETE /api/categories/**delete**/**<id>**:** Elimina una categoria. Se la categoria esiste, viene eliminata dal sistema, altrimenti il controller restituisce un errore 404.
- **GET /api/categories/**project**/**<projectId>**:** Recupera tutte le categorie associate a un progetto specifico. Il controller interroga il servizio per ottenere le categorie e le restituisce all'utente.

2.5 Database

Lo schema del database è stato prima progettato su carta, poi trascritto nel diagramma Entity-Relationship di 1.13, successivamente implementato in Java con le entità di JPA,

è realizzato tramite RDBMS PostgreSQL. Il database è stato creato al primo avvio dell'applicazione tramite il comando `spring.jpa.hibernate.ddl-auto=create` all'interno del file `application.properties`.

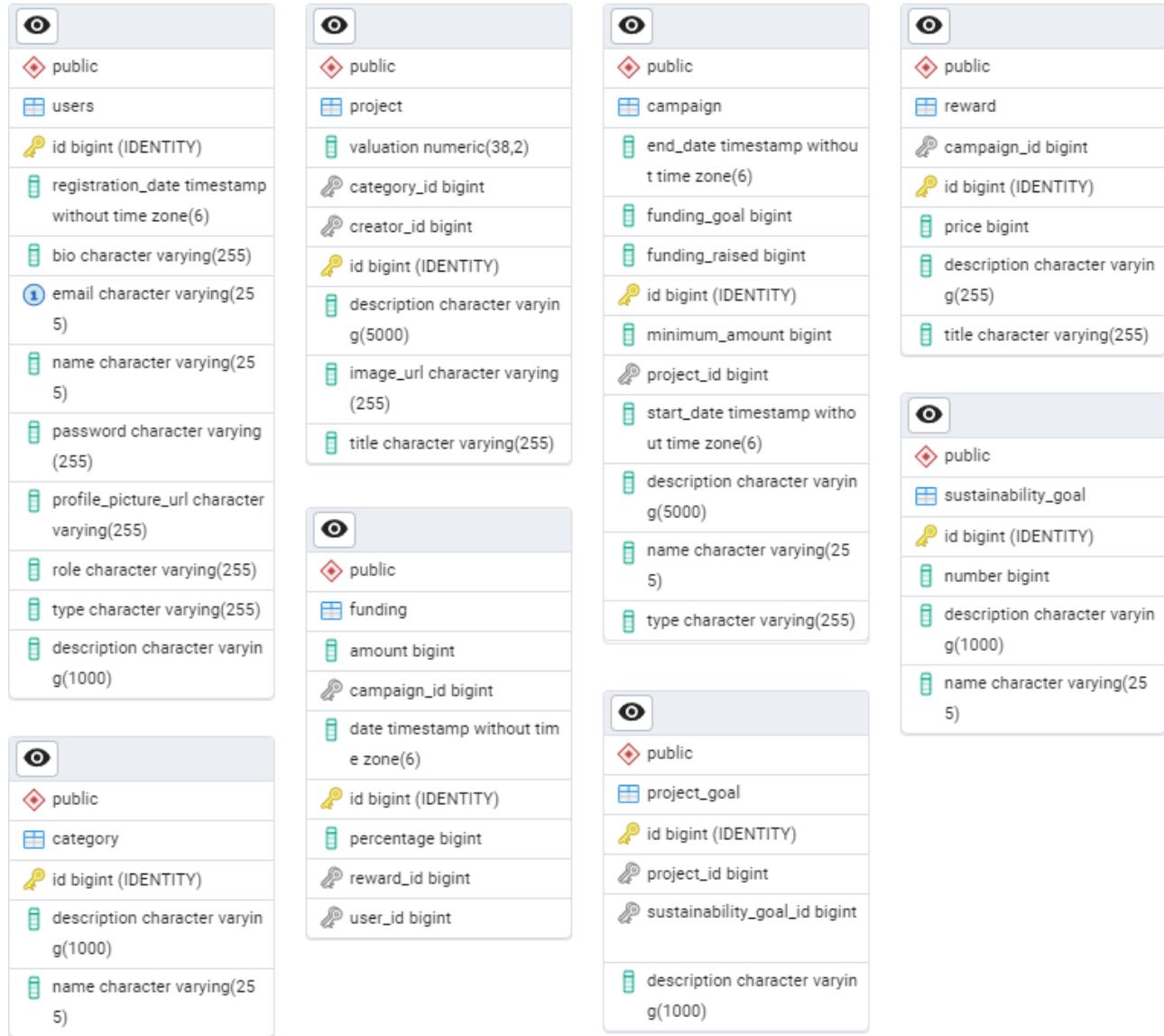


Figura 2.5: PostgreSQL database entities

Testing

Sono stati realizzati test unitari di ogni classe dei package *Model*, *Service* e *Controller* con l'utilizzo del testing framework JUnit, più alcuni test funzionali e di integrazione. Allo stato attuale sono stati implementati solo alcuni dei test unitari, mentre degli altri ne viene fornita una definizione per implementazioni future. Sono state utilizzate funzioni di setup per inizializzare gli oggetti prima dell'esecuzione di ogni test. Negli unit test relativi ai servizi, sono stati utilizzati oggetti mock delle classi repository in modo da simularne il comportamento senza effettuare operazioni CRUD sul database.

Sono stati inoltre realizzati dei test funzionali con il software Postman, tramite l'esecuzione di chiamate HTTP volte a testare la correttezza delle risposte e dei servizi forniti dalla piattaforma. Questi test sono stati realizzati in locale e senza l'utilizzo di mock, in modo da verificare la completa funzionalità del lato backend, a partire dal mapping delle chiamate, fino alle operazioni CRUD su database PostgreSQL.

3.1 User Tests

Di seguito riporto alcuni test riguardanti le classi utente. *UserTest* è un'unità di test per il modello *User*. È progettata per verificare che i getter e i setter della classe *User* funzionino correttamente, e per assicurarsi che lo stato dell'oggetto *User* possa essere correttamente gestito. *UserControllerTest* Verifica che le API esposte dal controller funzionino correttamente.

UserServiceTest verifica che la business logic implementata nella classe *UserService* funzioni correttamente, nello snippet sono mostrate le funzioni che testano la registrazione e il login dell'utente.

```

class UserControllerTest {

    private User user;
    private final Long id = 1L;
    private final String name = "David Megli";
    private final String description = "Software Engineer";
    private final String email = "david@example.com";
    private final String passwordHash = "hashedpassword123";
    private final String type = "Person";
    private final String role = "user";
    private final LocalDateTime registrationDate = LocalDateTime.now();
    private final String bio = "Experienced software developer with passion for entrepreneurship.";
    private final String profilePictureUrl = "http://example.com/profile.jpg";
    @Mock
    private UserService userService;
    @InjectMocks
    private UserController userController;
    private MockMvc mockMvc;
    @Test
    void testGetUserById() throws Exception {
        when(userService.getUserById(1L)).thenReturn(Optional.of(user));

        mockMvc.perform(get("/api/users/1")
                        .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.name").value(name));
    }
    @Test
    void testRegisterUser() throws Exception {
        when(userService.registerUser(user)).thenReturn(user);
        MvcResult result = mockMvc.perform(post("/api/users/register")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("{\"name\":\"David Megli\",\"description\":\"Software
Engineer\",\"email\":\"david@example.com\",\"passwordHash\":\"hashedpassword123\",\"type\":\"Person\",\"role\":\"use
r\",\"registrationDate\":\"2021-09-01T00:00:00\",\"bio\":\"Experienced software developer with passion for
entrepreneurship.\",\"profilePictureUrl\":\"http://example.com/profile.jpg\"}"))
                .andExpect(status().isOk())
                .andReturn();
    }
    @BeforeEach
    void setUp() {
        userController = new UserController();
        userService = new UserService();
        MockitoAnnotations.openMocks(this);
        mockMvc = MockMvcBuilders.standaloneSetup(userController).build();
        user = new User(name, description, email, passwordHash, type, role, registrationDate, bio,
                        profilePictureUrl);
    }
    //...
}

```

Figura 3.1: Snippet of UserControllerTest

3.2 Project Tests

La classe ProjectTest è una classe di test unitario per il modello Project che testa i metodi getter e setter della classe Project per verificare che i valori delle proprietà possano essere correttamente impostati e recuperati. Ad esempio, viene testato se i metodi getId(), getTitle(),

`setTitle(String title)` e simili funzionano correttamente. Questo assicura che l'oggetto `Project` possa gestire i suoi dati interni in modo coerente.

La classe `ProjectControllerTest` esegue test unitari e di integrazione per il controller *ProjectController*, simulando richieste HTTP per verificare le risposte del controller. Viene utilizzato `MockMvc` per simulare queste richieste e verificare che il controller risponda correttamente. Testa la creazione di un progetto (POST /api/projects/create), l'aggiornamento di un progetto (PUT /api/projects/update/<id>), l'eliminazione di un progetto (DELETE /api/projects/- delete/<id>), e il recupero dei progetti per ID, creatore, e categoria. In particolare, il test di aggiornamento (*updateProject*) controlla che il titolo di un progetto possa essere aggiornato correttamente e che il valore restituito corrisponda a quello previsto.

La classe `ProjectServiceTest` testa le operazioni principali fornite dalla classe *ProjectService*, che funge da livello intermedio tra il controller e il repository. Esta i metodi di creazione, aggiornamento, cancellazione, e recupero dei progetti. Utilizza `Mockito` per simulare le interazioni con il repository *ProjectRepository*, verificando che i metodi del servizio si comportino correttamente quando si interfaccia con i dati sottostanti. I test verificano anche i casi in cui un progetto potrebbe non essere trovato (`Optional.empty()`) e assicurano che i metodi gestiscano questi scenari in modo appropriato.

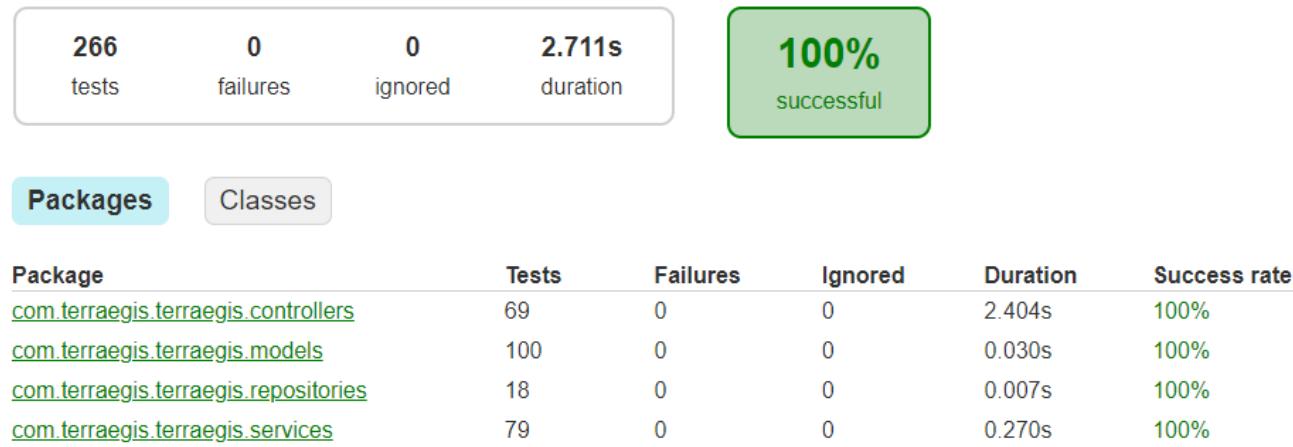
```
class ProjectTest {  
  
    private Project project;  
    private User creator;  
    private Category category;  
  
    @BeforeEach  
    void setUp() {  
        creator = new User();  
        creator.setId(1L);  
        creator.setName("David Megli");  
        category = new Category();  
        category.setId(1L);  
        category.setName("Technology");  
        project = new Project();  
        project.setId(1L);  
        project.setTitle("Project Title");  
        project.setDescription("Project Description");  
        project.setImageUrl("http://example.com/image.jpg");  
        project.setCreator(creator);  
        project.setCategory(category);  
        project.setValuation(new BigDecimal("100000.00"));  
    }  
  
    @Test  
    void getTitle() {  
        assertEquals("Project Title", project.getTitle());  
    }  
  
    @Test  
    void setTitle() {  
        project.setTitle("New Title");  
        assertEquals("New Title", project.getTitle());  
    }  
    @Test  
    void setCreator() {  
        User newCreator = new User();  
        newCreator.setId(2L);  
        project.setCreator(newCreator);  
        assertEquals(newCreator, project.getCreator());  
    }  
    @Test  
    void getCategory() {  
        assertEquals(category, project.getCategory());  
    }  
    @Test  
    void setCategory() {  
        Category newCategory = new Category();  
        newCategory.setId(2L);  
        project.setCategory(newCategory);  
        assertEquals(newCategory, project.getCategory());  
    }  
    @Test  
    void getValuation() {  
        assertEquals(new BigDecimal("100000.00"), project.getValuation());  
    }  
    @Test  
    void setValuation() {  
        project.setValuation(new BigDecimal("200000.00"));  
        assertEquals(new BigDecimal("200000.00"), project.getValuation());  
    }  
    //...  
}
```

Figura 3.2: Snippet of ProjectTest

3.3 Test results

In questa sezione sono riportati gli esiti di tutti i test unitari, raggruppati per package e raggruppati per classi.

Test Summary



Generated by [Gradle 8.8](#) at 26 ago 2024, 13:09:20

Figura 3.3: Tests results: packages

Test Summary

266 tests	0 failures	0 ignored	2.711s duration	100% successful	
Packages Classes					
Class					
Class	Tests	Failures	Ignored	Duration	Success rate
com.terraegis.terraegis.controllers.CampaignControllerTest	18	0	0	0.028s	100%
com.terraegis.terraegis.controllers.CategoryControllerTest	5	0	0	0s	100%
com.terraegis.terraegis.controllers.FundingControllerTest	7	0	0	0.003s	100%
com.terraegis.terraegis.controllers.ProjectControllerTest	8	0	0	0.004s	100%
com.terraegis.terraegis.controllers.ProjectGoalControllerTest	6	0	0	0.002s	100%
com.terraegis.terraegis.controllers.RewardControllerTest	5	0	0	0.002s	100%
com.terraegis.terraegis.controllers.SustainabilityGoalControllerTest	4	0	0	0.001s	100%
com.terraegis.terraegis.controllers.UserControllerTest	16	0	0	2.364s	100%
com.terraegis.terraegis.models.CampaignTest	20	0	0	0.010s	100%
com.terraegis.terraegis.models.CategoryTest	6	0	0	0.002s	100%
com.terraegis.terraegis.models.FundingTest	14	0	0	0.002s	100%
com.terraegis.terraegis.models.ProjectGoalTest	8	0	0	0.002s	100%
com.terraegis.terraegis.models.ProjectTest	14	0	0	0.004s	100%
com.terraegis.terraegis.models.RewardTest	10	0	0	0.002s	100%
com.terraegis.terraegis.models.SustainabilityGoalTest	8	0	0	0.001s	100%
com.terraegis.terraegis.models.UserTest	20	0	0	0.007s	100%
com.terraegis.terraegis.repositories.CampaignRepositoryTest	4	0	0	0.003s	100%
com.terraegis.terraegis.repositories.CategoryRepositoryTest	2	0	0	0s	100%
com.terraegis.terraegis.repositories.FundingRepositoryTest	4	0	0	0.001s	100%
com.terraegis.terraegis.repositories.ProjectGoalRepositoryTest	1	0	0	0.001s	100%
com.terraegis.terraegis.repositories.ProjectRepositoryTest	3	0	0	0.001s	100%
com.terraegis.terraegis.repositories.SustainabilityGoalRepositoryTest	3	0	0	0.001s	100%
com.terraegis.terraegis.repositories.UserRepositoryTest	1	0	0	0s	100%
com.terraegis.terraegis.services.CampaignServiceTest	20	0	0	0.005s	100%
com.terraegis.terraegis.services.CategoryServiceTest	6	0	0	0.002s	100%
com.terraegis.terraegis.services.FundingServiceTest	8	0	0	0.002s	100%
com.terraegis.terraegis.services.ProjectGoalServiceTest	7	0	0	0.002s	100%
com.terraegis.terraegis.services.ProjectServiceTest	10	0	0	0.175s	100%
com.terraegis.terraegis.services.RewardServiceTest	5	0	0	0.001s	100%
com.terraegis.terraegis.services.SustainabilityGoalServiceTest	7	0	0	0.001s	100%
com.terraegis.terraegis.services.UserServiceTest	16	0	0	0.082s	100%

Generated by Gradle 8.8 at 26 ago 2024, 13:09:20

Figura 3.4: Tests results: classes