



UNIVERSITÀ DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

TaskFlow

Autori:

Botarelli Tommaso

Giovannoni Alberto

Morganti Daniele

ANNO ACCADEMICO 2024/2025

Indice

1	Introduzione	1
1.1	Problema	1
1.2	Obiettivi	2
1.3	Utenti target	3
1.4	Obiettivi ulteriori dell'elaborato	3
2	Progettazione preliminare	5
2.1	Concetti base di un workflow condiviso	5
2.2	Modellazione del dominio applicativo	6
2.3	Analisi Requisiti	9
2.4	Use Cases	11
2.4.1	User	11
2.4.2	Owner	12
3	Progettazione e Architettura	13
3.1	Page Navigation Diagram	13
3.2	Mockups	15
3.2.1	Registration and Login Mockups	15
3.2.2	Organization Mockup	15
3.2.3	Project Mockup	16
3.3	Class Diagram	18
3.3.1	Business Logic	19
3.3.2	DAOs	20
3.3.3	Domain Model	21
3.4	Progettazione Database	24

4	Implementazione	25
4.1	Stack Tecnologico	25
4.1.1	Data Source	26
4.1.2	Tecnologia di Backend	27
4.1.3	Tecnologia di Frontend	29
4.2	Implementazione del Backend	29
4.2.1	Data Access Object (DAO)	30
4.2.2	Mapper e DTO	31
4.2.3	Servizi	33
4.2.4	Controllers	40
4.2.5	API Rest	42
4.3	Sicurezza e Autenticazione	43
4.3.1	Gestione dell'Autenticazione	43
4.3.2	Autorizzazione e Ruoli	44
4.3.3	Integrità delle URI	44
4.3.4	Sicurezza dei Dati	45
4.4	Deployment	45
5	Testing	47
5.1	Unit testing	48
5.1.1	Esempio di Unit Test: Testing su Activity	48
5.2	Integration testing	50
5.2.1	Esempio di Integration Test: Testing su ActivityService	51
5.2.2	Postman	52
6	Conclusione	55
6.1	Sviluppi futuri	55

1

Introduzione

1.1 Problema

La gestione di un flusso di lavoro all'interno di gruppi di persone, sia che si tratti di piccoli team o di grandi organizzazioni, rappresenta una sfida comune e complessa. Il coordinamento delle varie attività, infatti, è fondamentale per garantire che ogni membro del gruppo contribuisca al raggiungimento degli obiettivi. Tuttavia, indipendentemente dalla dimensione del gruppo, emergono problematiche ricorrenti che possono ostacolare il progresso e compromettere l'efficacia del lavoro collaborativo, nonché ridurre la produttività complessiva.

Fra i principali problemi del workflow all'interno di un gruppo di persone si può trovare la sincronizzazione del flusso di lavoro fra i vari membri. Ogni persona, infatti, lavora secondo ritmi e modalità diverse, il che rende difficile mantenere un flusso di lavoro coordinato.

Un altro aspetto critico è la condivisione delle risorse, come i documenti, che spesso accompagnano il processo di sviluppo. In un contesto di collaborazione, è essenziale che tutti abbiano accesso alle informazioni necessarie in modo tempestivo.

Infine, un ulteriore problema è rappresentato dalla capacità di comprendere lo stato di avanzamento del lavoro. Senza un monitoraggio chiaro e trasparente del progetto, diventa difficile per i membri del team comprendere a che punto si trovino nel processo complessivo e quali siano le priorità attuali.

Tutte queste problematiche comportano una perdita di efficienza e produttività indipendente dalla grandezza del gruppo seppur la complessità della gestione del flusso di lavoro tende a crescere con l'aumentare del numero di persone coinvolte. Di conseguenza, trovare soluzioni adeguate a questi problemi è cruciale per il successo di qualsiasi progetto collaborativo.

1.2 Obiettivi

A partire dalle problematiche appena descritte, una possibile soluzione può essere trovata nell'utilizzo di un tool che possa essere sfruttato per garantire coordinamento, condivisione e supervisione dello stato di avanzamento.

Lo scopo di questo progetto è quello di fornire un tool utile sia ai lavoratori, che ad eventuali supervisori, che permetta la facile gestione del lavoro condiviso.

Il tool dovrebbe essere il più possibile adattabile ai più disparati ambiti lavorativi o di sviluppo collaborativo che si possono presentare e dovrebbe essere di facile interfaccia con gli utenti.

Data la necessità di fornire un tool che sia di semplice utilizzo con l'utente finale e che abbia una interfaccia intuitiva, il progetto ha come obiettivo lo sviluppo di una applicazione web flessibile e scalabile per la programmazione del lavoro condiviso. Inoltre deve essere garantito che gli utenti appartenenti ad una certa organizzazione (ad esempio un'azienda o un semplice gruppo di persone) possano visionare lo stato di avanzamento dei progetti ai quali sono assegnati e possano interagire nella creazione e modifica di tasks (passi atomici di sviluppo). L'applicazione deve garantire loro flessibilità nella creazione dei task, in modo che essi siano facilmente personalizzabili, considerando anche che essa deve adattarsi a diversi ambiti lavorativi (non solo all'ambito ingegneristico).

Il prodotto deve permettere, in sostanza, il miglioramento dell'interazione e della comunicazione fra i lavoratori all'interno dell'azienda (o di un qualsiasi team di lavoro), e schematizzare il workflow in modo tale da ottimizzare le fasi lavorative e permettere una maggiore produttività e condivisione.

1.3 Utenti target

Dal momento che i problemi nello sviluppo di un progetto possono sorgere sia in piccoli gruppi di persone che in organizzazioni più ampie, come aziende strutturate con diversi team di lavoro, l'applicazione si rivolge a un vasto pubblico. Il prodotto potrà essere utilizzato infatti da gruppi di persone più o meno numerosi, e diversi anche per la loro struttura interna (presenza di team tra loro separati, o gerarchie all'interno del gruppo).

L'applicazione di management e productivity è rivolta principalmente a professionisti e team aziendali che operano in ambienti dinamici e necessitano di uno strumento efficiente per gestire progetti, attività e flussi di lavoro in modo organizzato. Gli utenti target includono project manager, team leader, e collaboratori, che desiderano ottimizzare la pianificazione e l'esecuzione delle loro attività quotidiane.

Inoltre, l'applicazione si rivolge ad organizzazioni, che cercano soluzioni scalabili per migliorare la produttività e la collaborazione tra i membri del team, facilitando la comunicazione, il monitoraggio dei progressi e l'allocazione delle risorse.

1.4 Obiettivi ulteriori dell'elaborato

Gli obiettivi del tool e le funzionalità che l'applicazione web deve fornire portano ad una naturale sperimentazione di alcune tecnologie che all'interno di questo elaborato sono di fondamentale importanza.

Database non relazionale Nell'ambito del progetto, è stato scelto l'utilizzo di un database non relazionale (NoSQL), in particolare MongoDB, in quanto riesce a garantire una maggior flessibilità di utilizzo rispetto a un classico database SQL. Inoltre, dal punto di vista didattico, abbiamo ritenuto che potesse essere istruttivo usare una tecnologia che nessuno di noi aveva toccato con mano.

Nelle sezioni successive, seguiranno analisi più approfondite delle motivazioni che ci hanno portato a scegliere questa tecnologia.

Spring Considerando gli obiettivi emersi durante la pianificazione del progetto, l'adozione di un framework come Spring si è rivelata una scelta naturale. Spring offre un'integrazione

completa per la gestione dei dati attraverso Spring Data MongoDB, combinata con un sistema di dependency injection. Inoltre Spring consente di personalizzare aspetti di sicurezza in modo semplice e flessibile.

Caricamento e gestione file L'utilizzo di un database NoSQL ha inciso anche sul framework utilizzato nel backend dell'applicativo. Questo ha permesso l'approfondimento del framework Spring Data MongoDB essenziale per la gestione del database MongoDB e utile per la mappatura degli oggetti Java e della loro composizione nel database.

Notifiche La possibilità di ricevere notifiche all'interno di questo contesto risulta una feature imprescindibile. La possibilità di aggiungere dei reminder ad una qualche attività apre quindi al problema della gestione delle notifiche.

2

Progettazione preliminare

In questa sezione verranno definiti alcuni concetti chiave per affrontare con maggiore efficacia le successive fasi di progettazione. Questa operazione è necessaria per suddividere le varie fasi e gli attori coinvolti nel concetto di workflow condiviso in componenti distinti e ben separati. Infatti, senza le giuste indicazioni, tali componenti potrebbero risultare poco chiari nella pratica.

Pertanto, in primo luogo saranno illustrati i concetti fondamentali legati al workflow condiviso, e poi verrà spiegata l'astrazione adottata nell'applicazione.

2.1 Concetti base di un workflow condiviso

Di seguito si riporta una lista di quelli che sono i concetti che rientrano nell'ambito di un flusso di lavoro condiviso.

Organizzazione Un'organizzazione è un'entità che rappresenta una struttura di lavoro come un'azienda o un semplice gruppo di persone. Al suo interno si possono trovare lavoratori e figure di più alto livello che presentano dei privilegi rispetto al semplice lavoratore (come per esempio la possibilità di assegnare dei compiti). Un'organizzazione solitamente ha come obiettivo lo sviluppo e il mantenimento di un qualche prodotto o di un servizio.

Capo dell'organizzazione Il capo dell'organizzazione è colui che ha dei privilegi rispetto ai classici lavoratori. Il capo dell'organizzazione può, ad esempio, creare nuovi progetti o assumere/licenziare i lavoratori.

Membro dell'organizzazione Un lavoratore è una persona appartenente ad una organizzazione che accede al servizio e svolge attività come descritto dai progetti in cui è stato inserito. Ha come obiettivo quello di portare a termine i compiti che gli sono assegnati.

Progetto All'interno di un'organizzazione sono definiti più progetti che costituiscono le attività pianificate per raggiungere un determinato obiettivo. L'insieme dei progetti ha come obiettivo lo sviluppo o il mantenimento del business dell'organizzazione.

Task I task rappresentano le attività specifiche che devono essere eseguite dai lavoratori per portare a termine un progetto. Il task può essere visto come un singolo passo di sviluppo dell'intero progetto. Solitamente un task può coinvolgere più o meno persone dell'organizzazione in base alla complessità per portarlo a termine.

Evento Un evento è un accadimento programmato, che riunisce persone e lavoratori per uno scopo specifico, come attività, riunioni ecc.

2.2 Modellazione del dominio applicativo

In questa sezione, i concetti precedenti sono riportati nell'ottica dell'applicazione. Mentre i concetti di organizzazione e progetto risultano simili a quelli descritti sopra, per i rimanenti si possono trovare alcune variazioni.

ORGANIZATION Un ORGANIZATION rappresenta il concetto di organizzazione descritto in precedenza. All'interno dell'applicativo un ORGANIZATION ha un nome e raggruppa più user. L'ORGANIZATION raggruppa anche più progetti i quali possono essere creati da un capo dell'organizzazione.

PROJECT Analogamente alla precedente descrizione nell'applicativo, un PROJECT rappresenta un obiettivo complesso. Esso ha un nome e raggruppa più attività, per le quali può definire un template dei campi che ne rappresenta uno schema predefinito da compilare.

ACTIVITY L'ACTIVITY racchiude sia il concetto di Task che quello di Evento. Nel contesto della nostra applicazione, un ACTIVITY rappresenta un'entità flessibile e generica, che può assumere la forma di un task (un'azione specifica necessaria per il completamento di un progetto) o di un evento (un accadimento programmato). Questo è possibile grazie a una definizione flessibile dei campi (maggiori dettagli in seguito) che consente di adattare le ACTIVITY alle varie esigenze organizzative, trattando task ed eventi come componenti integrati all'interno dell'architettura del sistema.

Esempio. Un'attività potrebbe rappresentare:

- Un task come “Scrivere il report finale del progetto”, che coinvolge uno o più membri del team con una scadenza definita;
- Un evento come “Riunione di revisione del progetto”, che riunisce il team per discutere l'avanzamento e i prossimi passi.

FIELD Un FIELD di un ACTIVITY è un attributo che è possibile aggiungere per categorizzare, descrivere, o organizzare meglio i task all'interno di un progetto. I FIELD sono usati per arricchire le ACTIVITY con informazioni aggiuntive, facilitando la gestione del flusso di lavoro. Con i FIELD è possibile creare viste filtrando e ordinando le ACTIVITY in base a criteri specifici, consentendo una gestione del progetto più trasparente e strutturata. I FIELD forniscono quindi da una parte un modo per poter filtrare e ordinare le ACTIVITY del lavoratore e dall'altra un modo per aggiungere informazioni utili (ad esempio un documento allegato o una descrizione testuale). All'interno dei progetti è possibile specificare un template dei campi da compilare per le attività del progetto stesso. Questo template viene poi usato come schema di base durante la creazione di una nuova ACTIVITY.

FIELD DEFINITION

Type	Descrizione
<i>Text</i>	Campo in cui puoi inserire informazioni generali in formato testo. Può essere usato per descrizioni brevi, note, o dettagli specifici relativi all'ACTIVITY.
<i>Single Selection</i>	Campo in cui puoi creare un elenco di opzioni predefinite e consentire di selezionare solo una di esse per ciascuna ACTIVITY. Questo è utile per categorizzare le ACTIVITY, ad esempio, per la priorità (Alta, Media, Bassa) o per lo stato (In corso, Completato).
<i>Number</i>	Campo per inserire valori numerici. Potrebbe essere usato per tracciare la stima del tempo necessario per completare un'ACTIVITY, costi, o altre metriche quantitative.
<i>Date</i>	Campo per selezionare una data. Utilizzato comunemente per scadenze, date di inizio o completamento, o per indicare milestone specifiche.
<i>Assignee</i>	Campo per assegnare l'ACTIVITY a uno o più utenti. Permette di vedere chi è responsabile del completamento dell'ACTIVITY.
<i>Document</i>	Campo che consente di allegare un file pdf a una ACTIVITY, utile per documentazione o specifica.

Tabella 2.1: Tabella raffigurante i diversi tipi di FIELD all'interno dell'applicativo

OWNER Con OWNER si intende un user che all'interno dell'organizzazione ha funzionalità aggiuntive. Queste sono: l'aggiunta di un nuovo membro, la possibilità di rendere OWNER un altro user e le operazioni di creazione ed eliminazione di un progetto.

USER Con USER si intende la figura del lavoratore che all'interno dell'organizzazione ha la possibilità di svolgere diverse operazioni, fra le quali la gestione delle Attività (aggiunta/modifica/eliminazione) e la creazione di una nuova organizzazione (di cui diventa OWNER).

2.3 Analisi Requisiti

In questa sezione vengono descritti i requisiti che l'applicazione deve soddisfare. Tali requisiti sono suddivisi in due categorie principali: i requisiti funzionali, che specificano le funzionalità dettagliate che l'applicazione deve offrire, e i requisiti non funzionali, che definiscono i vincoli generali da rispettare, indipendentemente dalle caratteristiche specifiche delle funzionalità.

Requisiti Funzionali

1. Autenticazione e Autorizzazione:

- (a) Un user deve potersi registrare all'applicazione utilizzando una mail, un nome utente e una password;
- (b) Un user iscritto all'applicazione deve poter effettuare login con nome utente e password.
- (c) I nomi utenti e le mail all'interno dell'applicazione devono essere univoci;

2. Gestione delle Organizzazioni

- (a) L'utente deve poter creare una nuova organizzazione e così facendo esso ne diventa owner;
- (b) L'utente owner di una organizzazione deve poter eliminare l'organizzazione;
- (c) L'utente owner di una organizzazione deve poter aggiungere user tramite username;
- (d) L'utente owner deve poter eliminare dei partecipanti da una organizzazione;
- (e) L'utente owner deve poter visualizzare tutti gli utenti all'interno di una organizzazione ed eventualmente elevare gli user ad owner;
- (f) L'utente deve poter visualizzare le organizzazioni a cui partecipa;

3. Gestione dei Progetti:

- (a) Un qualsiasi owner appartenente ad un'organizzazione deve poter creare un nuovo progetto;
- (b) Un qualsiasi user deve poter consultare la lista dei progetti che fanno parte dell'organizzazione a cui partecipa;

4. Gestione delle attività

- (a) Un user deve poter visualizzare le attività inerenti ad un particolare progetto;
- (b) Un qualsiasi user deve poter creare una nuova attività assegnando obbligatoriamente un nome.
- (c) Un user può aggiungere/eliminare/modificare i campi associati alle varie attività di una attività;
- (d) I campi associati alle attività hanno dei tipi specifici rappresentati nella tabella 2.1
- (e) Un qualsiasi user può eliminare una qualsiasi attività all'interno di un progetto;

5. Notifiche e Scadenze

- (a) Quando uno user aggiunge o modifica un campo di tipo date deve avere la possibilità di aggiungere l'invio automatico di una notifica prima di tale data;
- (b) La notifica deve poter essere inviata in modo automatico dall'applicazione e questa deve essere ricevuta dall'utente creatore dell'attività ed eventuali user associati alla stessa attività;

Requisiti Non Funzionali

1. Performance:

- (a) Tempo di risposta rapido per le operazioni comuni;
- (b) Scalabilità per supportare un numero crescente di utenti e dati;

2. Sicurezza:

- (a) Encrypting delle password e dei token;

3. Usabilità:

- (a) Interfaccia utente intuitiva e facile da usare;

2.4 Use Cases

In questa sezione si illustrano gli use cases diagrams che permettono di definire i casi d'uso per i due attori: user (utente semplice dell'applicazione) e owner (utente con privilegi all'interno di una certa organizzazione).

2.4.1 User



Figura 2.1: Use case diagram per User

2.4.2 Owner

L'owner rappresenta un user che ha creato un'organizzazione o che è stato nominato tale da un altro owner.

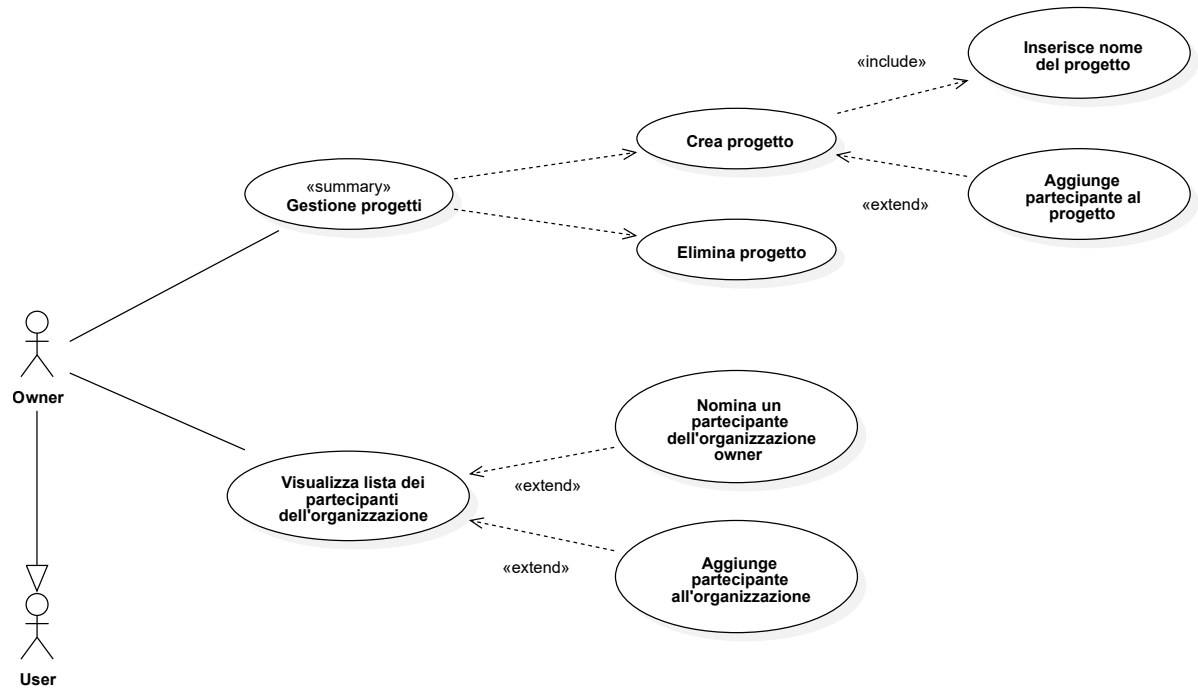


Figura 2.2: Use case diagram per Owner

3

Progettazione e Architettura

In questa sezione vengono illustrate le scelte architetture e progettuali che hanno guidato lo sviluppo dell'applicazione. La progettazione non si limita alla semplice implementazione del codice, ma coinvolge diversi aspetti fondamentali, tra cui la gestione della navigazione tra le pagine, la rappresentazione visiva delle interazioni, la modellazione delle classi e la progettazione del database.

Nelle sezioni seguenti, verranno presentati il *Page Navigation Diagram*, che descrive il flusso di navigazione dell'utente attraverso l'applicazione, i *mockups* delle principali interfacce, il diagramma delle classi (*class diagram*) che delinea la struttura logica dell'applicazione, e infine la progettazione del database.

3.1 Page Navigation Diagram

Il "Page Navigation Diagram" si occupa di descrivere le interazioni e le transizioni tra le diverse pagine dell'interfaccia utente. Rappresenta visivamente il flusso di navigazione dell'utente, mostrando come ciascuna pagina sia collegata alle altre e quali eventi scatenino tali transizioni. Nella fase di progettazione, il Page Navigation Diagram consente anche di visualizzare i percorsi principali e alternativi che l'utente può seguire all'interno del servizio.

Di seguito in figura 3.1 è riportato il page navigation diagram relativo al progetto. Esso è strutturato dall'alto verso il basso, iniziando con la <LoginPage>, il cui mockup è mostrato

in figura 3.3. In caso di fallimento nel login, l'utente riceve un `<ErrorDialog>`, che notifica l'inserimento errato di username o password. Qualora l'utente non possenga già un account, gli viene proposto il `<RegistrationDialog>`, vedi mockup in figura 3.2, con il quale può effettuare la registrazione.

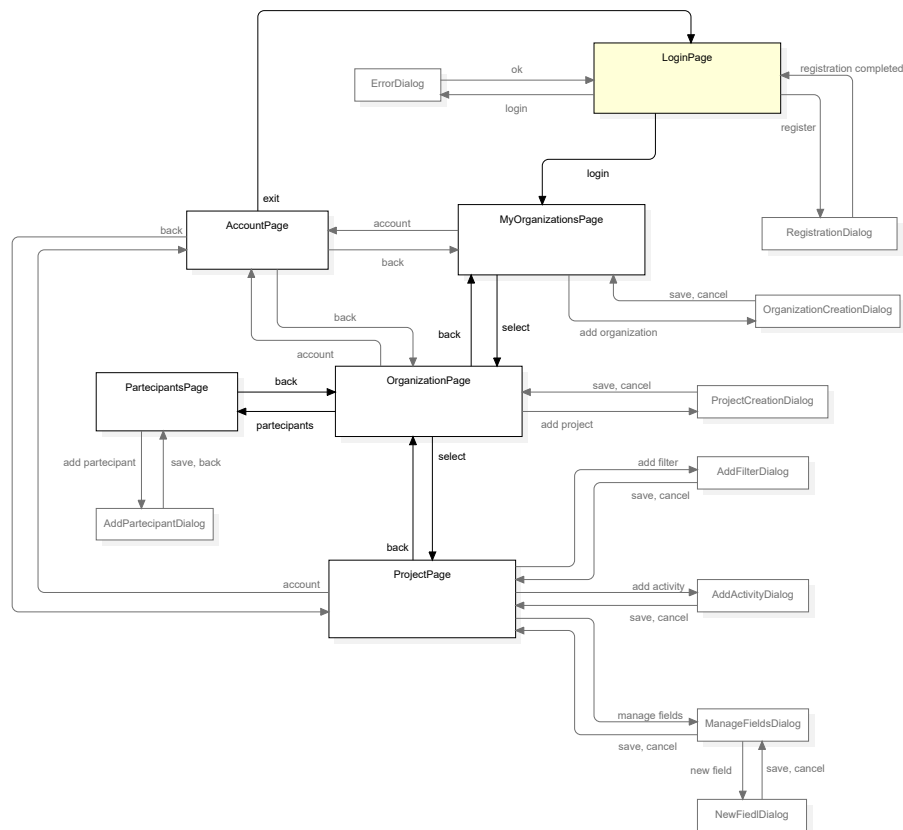


Figura 3.1: Page Navigation Diagram

Una volta effettuato il login, l'utente accede alla schermata `<OrganizationPage>`, vedi mockup in figura 3.4, nella quale sono mostrate tutte le organizzazioni a cui l'utente ha accesso. L'utente all'interno di questa pagina ha la possibilità di selezionare l'organizzazione in cui entrare.

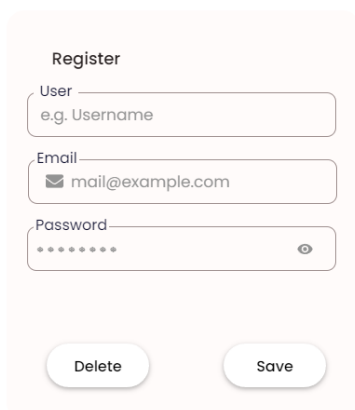
Cliccando su una delle organizzazioni proposte si accede alla schermata `<OrganizationPage>` (mockup 3.5), nella quale vengono mostrati tutti i progetti dell'organizzazione ed in cui è possibile crearne di nuovi attraverso l'interazione con il dialog `<ProjectCreationDialog>` (mockup 3.7).

Facendo click su un progetto, l'utente passa alla schermata `<ProjectPage>` (mockup 3.6),

dove sono mostrate tutte le attività del progetto, con la possibilità di crearne nuove tramite <AddActivityDialog>(mockup 3.8). All'interno di questa schermata è inoltre possibile per l'utente aggiungere ed eliminare nuovi field delle activity tramite <ManageFieldDialog>(mockup 3.9 e 3.10).

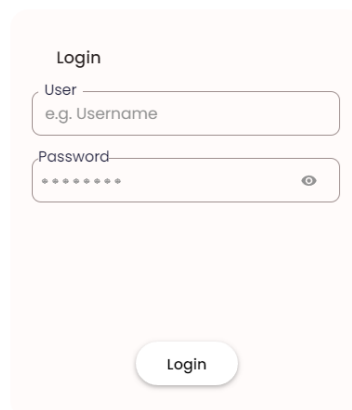
3.2 Mockups

3.2.1 Registration and Login Mockups



A registration dialog mockup with a light pink background. It features three input fields: 'User' with placeholder text 'e.g. Username', 'Email' with placeholder text 'mail@example.com' and an email icon, and 'Password' with placeholder text '.....' and a toggle icon. At the bottom, there are two buttons: 'Delete' and 'Save'.

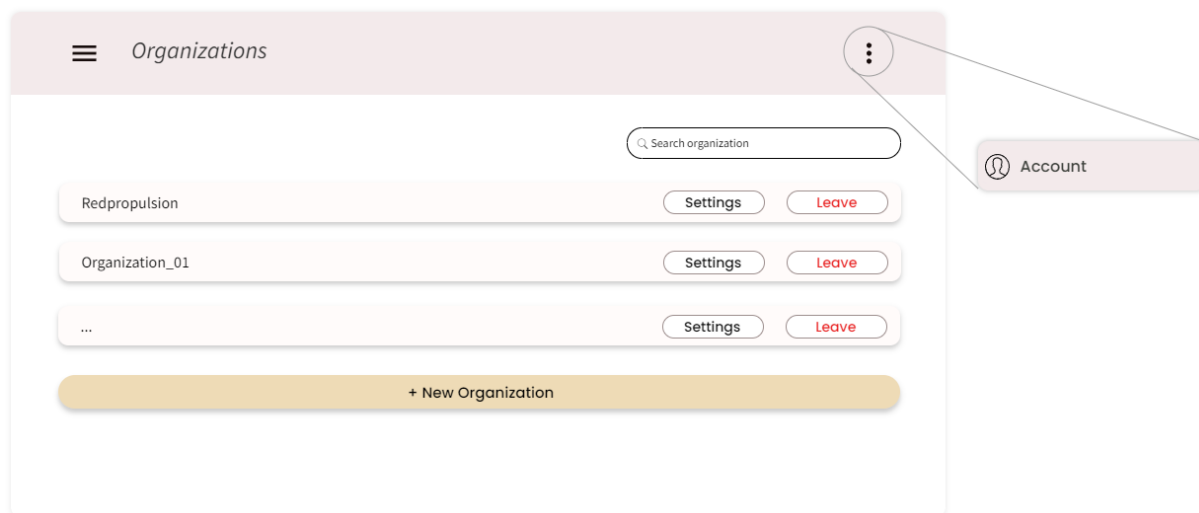
Figura 3.2: Registration dialog mockup



A login dialog mockup with a light pink background. It features two input fields: 'User' with placeholder text 'e.g. Username' and 'Password' with placeholder text '.....' and a toggle icon. At the bottom, there is a single button labeled 'Login'.

Figura 3.3: Login dialog mockup

3.2.2 Organization Mockup



A mockup of a 'My organization' page. The header bar is light pink and contains a hamburger menu icon, the text 'Organizations', and a three-dot menu icon. Below the header, there is a search bar with the placeholder text 'Search organization'. The main content area displays a list of organizations: 'Redpropulsion', 'Organization_01', and an ellipsis. Each organization entry has 'Settings' and 'Leave' buttons. At the bottom, there is a yellow button labeled '+ New Organization'. A callout box on the right, connected by a line to the three-dot menu icon, shows a user profile icon and the text 'Account'.

Figura 3.4: My organization page mockup

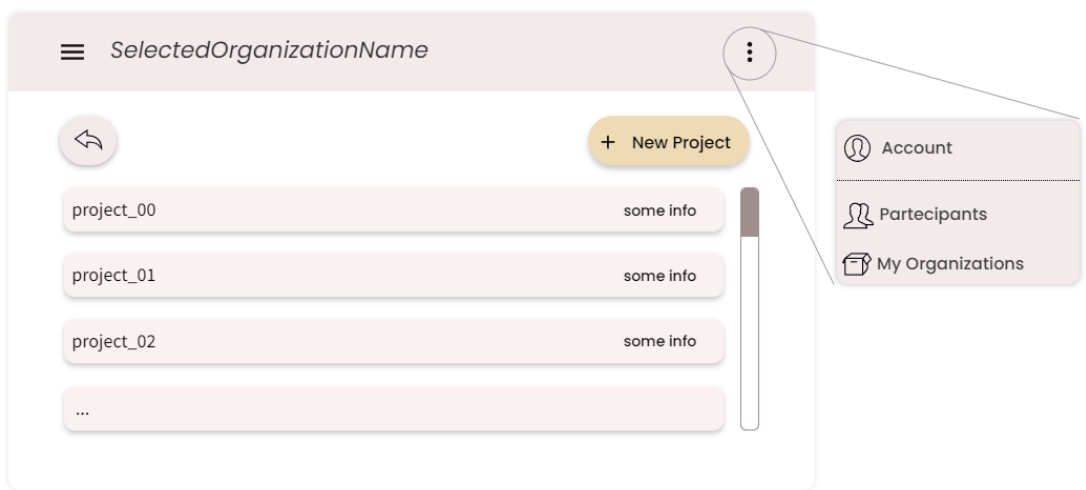


Figura 3.5: Project inside organization list mockup

3.2.3 Project Mockup

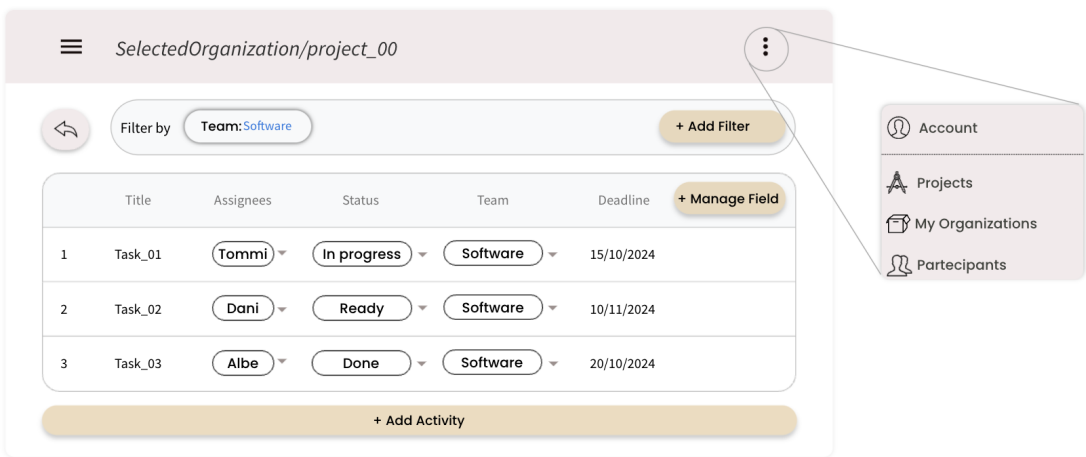


Figura 3.6: Project page mockup

New Project

Title

e.g. Title

Cancel

Save

Figura 3.7: New project dialog mockup

New Activity

Title

e.g. Title

Deadline

e.g. DD/MM/YYYY

☒ Notify

Description

e.g. Text

Partecipants

e.g. @user

Cancel

Save

Figura 3.8: New activity dialog

Manage Field

Assignees

☒

Status

☒

Team

☒

Deadline

☒

Priority

☐

+ New Field

Cancel

Save

Figura 3.9: Manage field dialog mockup

New Field

Field Name

Field Type

Text

Selection

1

Number

Date

Member

Document

Cancel

Save

Figura 3.10: New field dialog mockup

3.3 Class Diagram

Il Class Diagram è la parte fondamentale nella progettazione di una struttura software complessa e articolata. In questo progetto, come si vede in figura 3.11, il software è stato suddiviso in 3 packages principali: *businessLogic*, *domainModel* e *daos*.

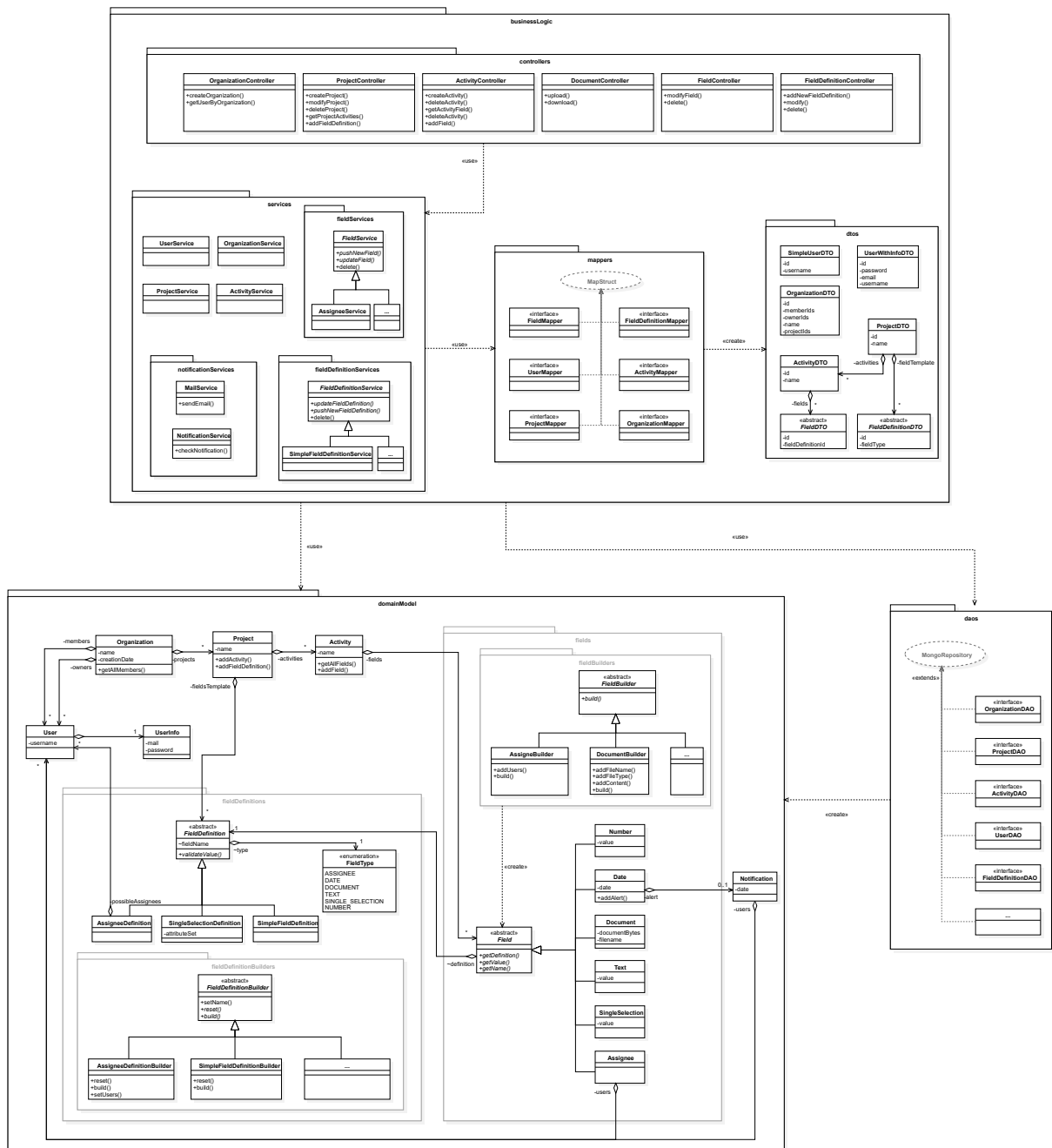


Figura 3.11: Class Diagram

3.3.1 Business Logic

Il package *businessLogic* rispecchia appieno quelle che sono le responsabilità che l'omonima sezione ha in una architettura RESTful. Esso contiene al suo interno diversi *sub-packages* che permettono una maggiore suddivisione delle responsabilità. :

controllers Questo sub-package raggruppa le classi che espongono gli endpoint. La loro funzione è quella di fornire un ingresso al backend tramite le URI e soddisfare le richieste sfruttando i metodi esposti dai services. Ogni classe all'interno di questo sub-package definisce molteplici URI come endpoint che il front end può sfruttare e ha come unica responsabilità quella di controllare che la richiesta sia valida e, in tal caso, chiamare il service adatto a soddisfarla. Come si può vedere dalla figura 3.12 nell'elaborato si sono resi necessari vari tipi di controller. La divisione degli endpoints in più controller ha reso possibile la differenziazione dei vari servizi esposti, ogni controller ha infatti una responsabilità limitata alle entità con cui deve interagire permettendo una maggiore divisione delle responsabilità. Da notare come la suddivisione dei servizi ha reso possibile l'implementazione di un unico controller per i diversi tipi di field da gestire (ad eccezione del field *document* che necessita alcune operazioni aggiuntive);

services Questo sub-package raggruppa le classi che implementano i casi d'uso. I services sono quindi richiamati dai controllers per adempiere alle richieste provenienti dal front end. All'interno di questo elaborato è stata particolarmente delicata la scelta architetturale di queste classi. Infatti, date le diverse responsabilità in gioco, si è resa necessaria una suddivisione molto accurata dei vari servizi esposti.

L'utilizzo dell'ereditarietà è stato fondamentale per i servizi di *Field* e *FieldDefinition* perché ha permesso la non replicazione di codice ed una facile espandibilità alla futura introduzione di nuovi *Field*.

Per queste classi si è reso necessario l'utilizzo della classe *FieldServiceManager* che permette l'istanziamento del corretto service per il corrispondente tipo. Questa classe ha quindi la sola responsabilità di restituire il tipo di service adatto rispetto al tipo in ingresso.

La classe *FieldService* infine fornisce una interfaccia per gli attuali *Field* (ed implementa le operazioni che invece sono comuni fra i *Field*) ed eventuali *Field* futuri permettendo una facile estensione (questo in modo analogo anche per *FieldDefinition*).

dtos I dtos permettono di comunicare con il front end tramite versioni semplificate delle classi del domain model. Data la struttura altamente annidata delle varie classi anche lato front end (si pensi per esempio alla visualizzazione di una attività la cui visualizzazione dei relativi field diventa imprescindibile) in questo elaborato è stato scelto di considerare anche relazioni fra le varie classi dto. Attraverso la composizione è quindi possibile passare oggetti più complessi fra back-end e front-end permettendo tramite una singola richiesta di ottenere tutte le informazioni necessarie.

mappers Questo sub-package contiene le classi che permettono la traduzione da oggetti del domain model alle relative versioni semplificate dei dtos e viceversa.

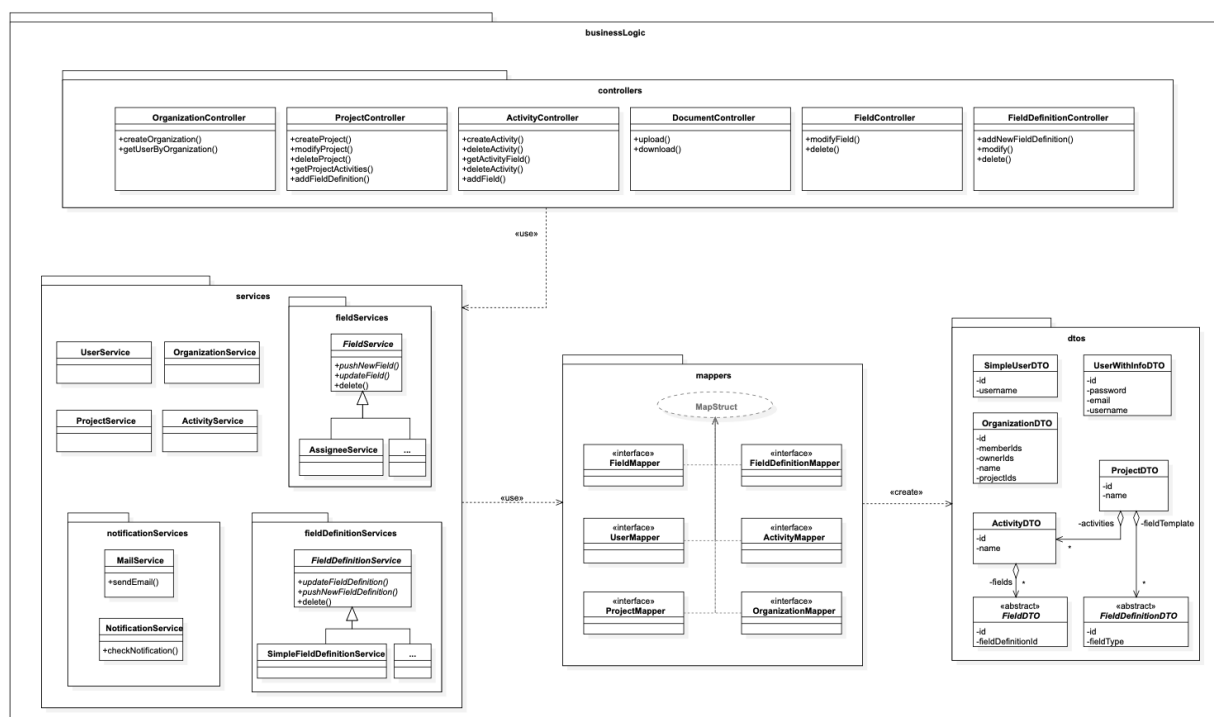


Figura 3.12: Business Logic

3.3.2 DAOs

Il pattern Data Access Object (DAO) è stato implementato per gestire l'interazione con il database in modo strutturato e separato dalla logica di business. I DAOs fungono da intermediari tra l'applicazione e il database, permettendo di eseguire operazioni di lettura e scrittura sui dati senza esporre direttamente la logica di accesso al database. In questo progetto, i DAO sono stati utilizzati per eseguire query e operazioni CRUD (Create, Read, Update, Delete) su diverse

entità del domain model, come utenti, organizzazioni, progetti e attività. Questa struttura favorisce una maggiore manutenibilità e testabilità, consentendo di aggiornare la logica di persistenza senza influenzare altre parti dell'applicazione.

L'approccio adottato con i DAO offre anche una maggiore flessibilità nel gestire il backend, permettendo di astrarre i dettagli specifici del database utilizzato. Questo consente, in futuro, di cambiare il tipo di database o ottimizzare le query senza dover modificare il resto dell'applicazione.

In figura 3.13 sono rappresentate una parte delle classi che sono state implementate nel progetto. In generale si è reso necessario introdurre una classe DAO per ogni entità rilevante del progetto (vedi sezione 4.2.1).

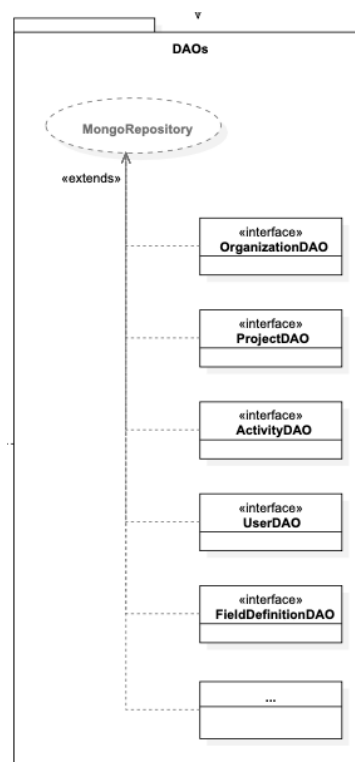


Figura 3.13: DAOs

3.3.3 Domain Model

Il domain model ha l'obiettivo di astrarre il contesto che il software vuole modellare, creando una rappresentazione chiara e semplificata degli elementi fondamentali che lo compongono. Ogni elemento del modello rappresenta un concetto o un'entità del mondo reale, trasportandolo nel software con caratteristiche specifiche e relazioni che rispecchiano le dinamiche operative

contenute in classi diverse: Project presenta una lista di FieldDefinition, mentre activity presenta una lista di Field. Questo perchè ogni Activity all'interno di un progetto deve rispettare il template di campi definito in Project, e che quindi non è relativo alla classe Activity ma bensì alla classe Project. Al contrario un Field deve essere relativo ad una Activity, e deve inoltre rispettare un FieldDefinition, motivo per cui questa classe presenta un riferimento ad un FieldDefinition. Field possiede tutte le sottoclassi concrete descritte nel dominio applicativo, esse specificano solamente il tipo di valore che il Field contiene (e.g. la data di un giorno nel caso di Date). FieldDefinition invece non possiede tutte le sottoclassi relative a Field, poichè la responsabilità di questa classe è definire un pool di valori ammissibile che poi un oggetto Field potrà possedere (e.g. l'elenco degli User che possono essere inseriti in un Assignee). Nel caso di alcune sottoclassi di Field, il pool di valori dalla quale scegliere non esiste (pensiamo ad esempio al caso del field 'Text'). Per questo motivo è presente una sottoclasse SimpleFieldDefinition che comprende tutti questi casi (Text, Number, Date).

FieldBuilder e FieldDefinitionBuilder Dato che le classi Field e FieldDefinition presentano una certa complessità nei loro sottotipi, sono state pensate delle relative classi Builder che incapsulano la creazione di questi oggetti. Questo è pensato per un uso più semplice ed affidabile da parte di un codice client esterno.

4

Implementazione

4.1 Stack Tecnologico

In questa sezione si vuole illustrare lo stack utilizzato per lo sviluppo dell'applicazione web. Le scelte effettuate sono il risultato di un'attenta analisi di quelli che sono i requisiti funzionali e non.

L'architettura di base dell'applicativo risulta essere della tipologia RESTful. La scelta di questa tipologia di stack può essere ricercata sia nella natura del tool che si intende sviluppare, sia nei suoi obiettivi.

L'uso di questa tecnologia permette di rendere facilmente estendibile l'applicazione web con nuove funzionalità. Questo perché lo schema RESTful si presta bene all'aggiunta di molte nuove funzionalità, che con il tempo potrebbero rendersi necessarie.

L'architettura RESTful permette infatti di sviluppare l'applicativo permettendo una facile e veloce implementazione di future funzionalità che possono rendersi necessarie.

Inoltre, grazie al decoupling tra client e server, le modifiche o le estensioni del server possono essere implementate senza richiedere aggiornamenti immediati ai client, consentendo una maggiore flessibilità nello sviluppo e nel rilascio di nuove versioni. Questo approccio modulare e organizzato semplifica anche il testing e la documentazione delle API, migliorando la manutenibilità a lungo termine e riducendo il rischio di introdurre bug o regressioni durante

l'implementazione di nuove funzionalità.

Dal momento che l'applicazione web potrebbe essere utilizzata da un numero sempre crescente di utenti, l'adozione di una tecnologia RESTful permette una scalabilità migliore grazie alla sua architettura stateless, che non richiede al server di mantenere lo stato della sessione tra le richieste. Questo significa che ogni richiesta HTTP contiene tutte le informazioni necessarie per essere elaborata indipendentemente, permettendo ai server di gestire ogni richiesta in modo isolato e di distribuire il carico di lavoro. Questa caratteristica facilita la scalabilità orizzontale, poiché nuovi server possono essere aggiunti o rimossi senza dover replicare lo stato della sessione.

4.1.1 Data Source

Il database utilizzato all'interno dell'applicazione è MongoDB. MongoDB è un database No-SQL di tipo document-oriented che memorizza i dati in documenti BSON (un formato binario simile a JSON), permettendo una grande flessibilità nella struttura dei dati. A differenza dei database relazionali (SQL), che richiedono schemi rigidi e normalizzati in tabelle, MongoDB consente di memorizzare documenti con campi variabili e nidificati, facilitando così l'eventuale modifica del modello dati senza necessità di refactoring pesante. Un altro vantaggio nell'utilizzo di MongoDB rispetto ai database SQL è la sua capacità di scalare orizzontalmente. MongoDB supporta lo sharding, una tecnica di distribuzione dei dati su più server, che consente di gestire grandi volumi di dati e di traffico senza compromettere le prestazioni. Questo nell'ottica dell'applicazione risulta un requisito fondamentale dato che il numero di utenti al suo interno potrebbe variare nel tempo. Infine, la natura schemaless di MongoDB permette di manipolare i dati in modo più dinamico e adattabile rispetto ai modelli tabellari rigidi dei database SQL. Questi ultimi, infatti, comportano per query complesse molte operazioni tra più tabelle, che possono compromettere la comprensione da parte degli sviluppatori. Nel nostro problema questa caratteristica si rende estremamente utile, dato che, le unità fondamentali modellate (le attività) non hanno uno schema di campi preimpostato. L'utente ha quindi la possibilità di personalizzare queste entità in modo più o meno complesso, secondo la sua visione del progetto.

4.1.2 Tecnologia di Backend

Il linguaggio di programmazione utilizzato per il backend dell'applicazione è Java accoppiato con l'utilizzo del framework Spring, del quale è stato utilizzato il modulo Spring Data MongoDB per interagire con il database.

Nonostante i vantaggi che l'utilizzo di MongoDB porti rispetto ad un database relazione, l'interazione con esso può risultare complessa, soprattutto quando si tratta di gestire query avanzate, operazioni CRUD (Create, Read, Update, Delete) e altre attività comuni di accesso ai dati. Per questa parte quindi risulta essenziale Spring Data MongoDB, che fornisce un'interfaccia di alto livello per interagire con MongoDB, nascondendo la complessità della gestione manuale del database.

Rispetto ad una gestione manuale del database, Spring Data MongoDB fornisce molti vantaggi:

1. **Astrazione delle Operazioni CRUD:** Con Spring Data MongoDB, le operazioni CRUD sono semplici e intuitive. Il framework fornisce repository predefiniti che permettono di eseguire queste operazioni con poche righe di codice, eliminando la necessità di scrivere query MongoDB manuali;
2. **Query Derivation:** Una delle caratteristiche più potenti di Spring Data MongoDB è la query derivation, che permette di generare automaticamente query MongoDB basate sulla convenzione di denominazione dei metodi all'interno dei repository;
3. **Integrazione Seamless con il Framework Spring:** Spring Data MongoDB si integra perfettamente con l'intero ecosistema Spring. Questo include il supporto per la dependency injection, la gestione delle transazioni (anche se in modo limitato in contesti NoSQL), e l'integrazione con altri moduli come Spring Boot;
4. **Mapping e Conversione Automatizzata:** Spring Data MongoDB gestisce automaticamente il mapping degli oggetti Java (POJO) ai documenti MongoDB e viceversa. Questo significa che è possibile lavorare direttamente con oggetti Java senza preoccuparsi di come questi siano rappresentati a livello di database;
5. **Supporto per le Operazioni Aggregation:** Le operazioni di aggregazione in MongoDB, che possono essere complicate e verbose, sono semplificate in Spring Data MongoDB

grazie all'API di alto livello che permette di costruire pipeline di aggregazione utilizzando un approccio programmatico e tipizzato;

6. **Estendibilità e Customizzazione:** Nonostante l'alta astrazione, Spring Data MongoDB permette ancora un alto grado di customizzazione. Si possono definire metodi di query personalizzati, scrivere implementazioni native MongoDB dove necessario, e utilizzare le annotazioni per configurare i dettagli di mapping.

Si può concludere quindi dicendo che l'utilizzo di Spring Data MongoDB rappresenta una scelta strategica per facilitare l'interazione con MongoDB, migliorando la produttività nella scrittura del codice. Questo approccio riduce la complessità e consente di concentrarsi sulla logica di business piuttosto che sulla gestione dei dettagli del database.

Librerie e Dipendenze

Spring Security è una libreria di autenticazione e autorizzazione utilizzata per proteggere le applicazioni Java, in particolare quelle basate su Spring Framework, permettendo di proteggere facilmente le applicazioni web, API o microservizi. Una delle caratteristiche principali di Spring Security è la flessibilità nella definizione delle regole di accesso alle risorse, che possono variare da endpoint pubblici, accessibili a tutti, a risorse protette, disponibili solo a utenti autenticati o con ruoli specifici.

MapStruct è un framework di mappatura Java che facilita la conversione tra DTO (Data Transfer Object) e oggetti del Domain Model, riducendo al minimo il codice e migliorandone la leggibilità. Uno dei principali vantaggi di MapStruct è la capacità di mappare automaticamente gli attributi che sono identici fra le due versioni dell'oggetto, generando il codice di mapping in fase di compilazione. Questo riduce significativamente gli errori e il tempo di sviluppo rispetto a soluzioni manuali. Gli sviluppatori devono occuparsi solo di mappare manualmente gli attributi più complessi o quelli che richiedono una logica personalizzata.

Spring Mail è una componente del framework Spring che semplifica l'invio di email in applicazioni Java. Integrato con JavaMail, fornisce un'interfaccia per configurare e inviare email in modo asincrono o sincrono. Con Spring Mail, è possibile inviare email di testo o HTML, allegare file e gestire eventi di posta elettronica. Nel contesto di applicazioni aziendali o servizi

online, Spring Mail è spesso utilizzato per inviare notifiche automatiche, conferme d'ordine, avvisi di sicurezza o altre comunicazioni con gli utenti..

4.1.3 Tecnologia di Frontend

Per la parte di frontend è stato scelto di utilizzare Angular in combinazione con Material Design 3, per garantire la scalabilità, la manutenibilità e l'efficienza del prodotto finale. Angular, nella sua versione più recente, introduce significative ottimizzazioni delle prestazioni, una gestione del rendering ancora più efficiente grazie a Ivy, e il supporto per l'adozione di TypeScript 4.x, offrendo così un'esperienza di sviluppo all'avanguardia. Queste caratteristiche ci permettono di costruire applicazioni reattive e performanti, semplificando al contempo la gestione della complessità del codice.

L'adozione di Angular Material, inoltre, consente di seguire le linee guida di Material Design 3, che sono state aggiornate per offrire componenti UI moderni, accessibili e altamente personalizzabili. Questo non solo velocizza il processo di sviluppo, ma assicura anche un'interfaccia utente coerente, accattivante e user-friendly, capace di adattarsi facilmente a dispositivi e contesti diversi. Integrando queste tecnologie, si può garantire un front-end all'avanguardia, in grado di soddisfare sia i requisiti tecnici del progetto che le aspettative degli utenti finali.

4.2 Implementazione del Backend

In questa sezione, vengono presentate le principali componenti implementate, come i Data Access Object (DAO), i mapper, i Data Transfer Object (DTO) i servizi per la gestione dei dati e delle notifiche.

4.2.1 Data Access Object (DAO)

Spring consente di istanziare i DAO in modo automatico, come si vede nel codice 4.1, generando automaticamente i metodi per le operazioni CRUD.

```
public interface FieldDAO extends MongoRepository<Field, String> {  
    void deleteFieldByFieldDefinition(FieldDefinition fieldDefinition);  
    List<Field> findFieldByFieldDefinition(FieldDefinition fieldDefinition);  
    List<Date> findByNotification(Notification notification);  
}
```

Codice 4.1: DAO di esempio, FieldDAO

Le classi che rappresentano i DAO devono essere *interface* e quindi non si possono implementare operazioni personalizzate (le interfacce non permettono la definizione dei metodi). Tuttavia per alcuni DAO si è reso necessario implementare metodi aggiuntivi. Alcuni metodi semplici che si basano su operazioni *by attribute* nei documenti sono implementati direttamente dal framework Spring (introducendo il metodo nell'interfaccia e seguendo una convenzione nella dicitura di esso è possibile far generare a Spring il codice relativo alla query).

Per altre operazioni più complesse dove è necessario introdurre una logica più elaborata è invece necessario seguire una certa convenzione imposta da Spring che implica la creazione di un'interfaccia *custom* e di una classe concreta che implementa tale interfaccia (codice) con le operazioni personalizzate. Dopodiché l'interfaccia del DAO classica estenderà oltre alla classe *MongoRepository* anche tale implementazione permettendo l'utilizzo delle operazioni custom introdotte.

```
// Definizione di una interfaccia custom
public interface CustomOrganizationDAO {
    // Il seguente metodo richiede una logica complessa
    ArrayList<Organization> getOrganizationByUserId(String userId);
}
...
// Creazione classe concreta che implementa il metodo
public class CustomOrganizationDAOImpl implements CustomOrganizationDAO{
    @Override
    public ArrayList<Organization> getOrganizationByUser(String userId) {
        // Implementazione del metodo
    }
}
...
// L'interfaccia base estende anche la nuova implementazione
public interface OrganizationDAO extends MongoRepository<Organization, String>,
    CustomOrganizationDAO {
}
```

Codice 4.2: Implementazione di operazioni complesse all'interno di un DAO in Spring, esempio di OrganizationDAO

4.2.2 Mapper e DTO

Nel contesto REST, i DTO fungono da oggetti intermedi che rappresentano i dati trasferiti tra il client e il server. L'uso dei DTO consente la riduzione del numero di campi non necessari trasmessi, una separazione tra modello di dominio e modello di visualizzazione e una maggiore flessibilità nell'adattare i dati esposti tramite le API senza modificare le entità di dominio sottostanti.

Per convertire i dati tra entità di dominio (oggetti Entity) e DTO, si utilizzano i mapper.

Implementazione di un Mapper con MapStruct In questo progetto, è stata utilizzata la libreria MapStruct per automatizzare la creazione dei mapper. MapStruct genera automaticamente il codice di mapping durante la compilazione, riducendo così il lavoro manuale e minimizzando gli errori. Per attributi semplici che posseggono nome e tipo uguale nel DTO e nell'entity la mappatura avviene in automatico, mentre per attributi più complessi è necessario implementare il mapping con metodi appositi, referenziati con apposite stringhe nei tag.

Un esempio di mapper implementato è il seguente, il quale mappa oggetti della classe Activity alla loro corrispondente rappresentazione DTO, ovvero ActivityDTO, e viceversa.

```
@Mapper(componentModel = "spring")
@Component
public interface ActivityMapper {

    @Mapping(source = "fields", target = "fields", ignore = true)
    Activity toEntity(ActivityDTO dto);

    @Mapping(source = "fields", target = "fields", qualifiedByName = "
        mapFieldsToFieldDTO")
    ActivityDTO toDto(Activity user);

    @Named("mapFieldsToFieldDTO")
    default ArrayList<FieldDTO> mapFieldsToFieldDTO(ArrayList<Field> fields) {
        ArrayList<FieldDTO> fieldsDTO = new ArrayList<FieldDTO>();
        fieldsDTO = fieldToFieldDto(fields);

        return fieldsDTO;
    }

    default ArrayList<FieldDTO> fieldToFieldDto(ArrayList<Field> fields) {
        return Mappers.getMapper(FieldMapper.class).fieldToFieldDto(fields);
    }

    default ArrayList<ActivityDTO> activityToActivityDto(ArrayList<Activity>
        activities){
        ArrayList<ActivityDTO> activityDtoList = new ArrayList<ActivityDTO>();

        for(Activity activity : activities){
            activityDtoList.add(this.toDto(activity));
        }

        return activityDtoList;
    }
}
```

Codice 4.3: Mapper di esempio, ActivityMapper

L'interfaccia è definita con l'annotazione `@Mapper`, che indica a MapStruct di generare automaticamente l'implementazione del mapper. Il `componentModel` impostato su *"spring"* permette di integrare questo mapper come un bean di Spring, facilitando l'iniezione del mapper nei componenti del servizio.

Il metodo `toEntity()` mappa un oggetto DTO in un'entità, gli attributi `source` e `target` dentro l'annotazione rappresentano la coppia di attributi da mappare, tutti gli attributi che posseggono nome identico vengono mappati automaticamente senza una specifica dentro l'annotazione. Nel

caso dell'esempio, e in molti altri, il mapping di un campo (fields) è ignorato, questo perchè l'oggetto DTO presenta solo i riferimenti del database di tali entità, è quindi responsabilità del servizio che chiama il mapper accedere tramite DAO al database e reperire tutte le entità annidate da inserire all'interno dell'entity.

Il metodo *toDto()* esegue il mapping inverso. Qui viene utilizzato un metodo qualificato (*mapFieldsToFieldDTO*) per mappare il campo fields. Nella maggior parte dei mapper implementati, questo metodo usa un mapping personalizzato; nell'esempio questo è stato fatto per convertire una lista di Field in una lista di FieldDTO. A differenza di prima questa mappatura è possibile poichè tutte le informazioni necessarie per la costruzione di un DTO sono sicuramente presenti nella relativa entità.

Il metodo di mappatura personalizzato è annotato con *@Named*, che consente di referenziare un metodo personalizzato per gestire la trasformazione di una entity in un DTO (nell'esempio della lista fields in una lista di FieldDTO). Questo metodo può invocare a sua volta un mapper per le classi annidate, che delega la conversione effettiva ad un mapper specifico (nell'esempio per la conversione degli oggetti Field).

4.2.3 Servizi

In questa sezione sono presentati i principali servizi implementati nel programma. Attraverso i servizi, l'applicazione può coordinare l'interazione tra i vari componenti, come i DAO e gli oggetti del dominio. Nel progetto, i servizi sono stati sviluppati per gestire diverse funzionalità, tra cui l'inserimento, la modifica e la cancellazione di risorse, nonché la gestione delle notifiche, l'autenticazione e l'autorizzazione degli utenti.

Inserimento di Documenti nelle Attività

Gli utenti hanno la possibilità di caricare documenti (PDF o altri formati) in allegato alle Attività, come un campo vero e proprio. Questa funzionalità è stata implementata attraverso un servizio che consente agli utenti di caricare i documenti, i quali vengono serializzati e associati al campo desiderato lato frontend. Risulta poi possibile inviare il documento serializzato tramite una richiesta HTTP POST. Il servizio preposto effettuerà il salvataggio del documento nel database. Per quanto riguarda il recupero del PDF, viene fatto tramite una API GET che consente di scaricare il PDF desiderato nel proprio File System.

In figura 4.1 è riportato un esempio di un documento PDF salvato sul database. Come si può vedere, il campo *base64* contiene il codice serializzato del documento, che consente di recuperare e scaricare il contenuto del documento.

```

{
  "_id": {
    "$oid": "6712299d7728094263f63a8b"
  },
  "fileName": "test",
  "fileType": "pdf",
  "content": {
    "binary": {
      "base64": "JVBERi0xLjUKJb/3ov4KNjMgMCDvYmoKPDwgL0xpbmVhcmL6ZWQgMSAvTCAxMDkyNzk4IC9IIFsgODk2IDE4OSBdIC9PIDY3IC9FIDExMTIyNSAvTiA3IC9U
      "subType": "00"
    }
  },
  "fieldDefinition": {
    "sref": "fieldDefinition",
    "id": {
      "$oid": "6712299d7728094263f63a8a"
    }
  },
  "uuid": "494619da-dee2-4d05-a45c-9d6844889b81",
  "_class": "com.example.taskflow.DomainModel.FieldPackage.Document"
}

```

Figura 4.1: Esempio di un documento pdf salvato nel Database

La serializzazione del documento lato client permette di ricevere in ingresso un qualsiasi tipo di file, lasciando quindi spazio all'utilizzo del campo *Document* per allegare un qualsiasi tipo di file inerente l'attività.

Notifiche Mail

Quando un utente crea un campo di tipo "Date", si presenta la possibilità di pianificare una notifica, come *reminder* di tale attività programmata. In questa notifica è possibile specificare: un messaggio, una data di ricezione e gli utenti ai quali inviare la notifica. Grazie a Spring Mail, è stato possibile implementare un sistema di notifiche via email, consentendo l'invio automatico di comunicazioni agli utenti. Il template della mail è riportato in figura 4.2.

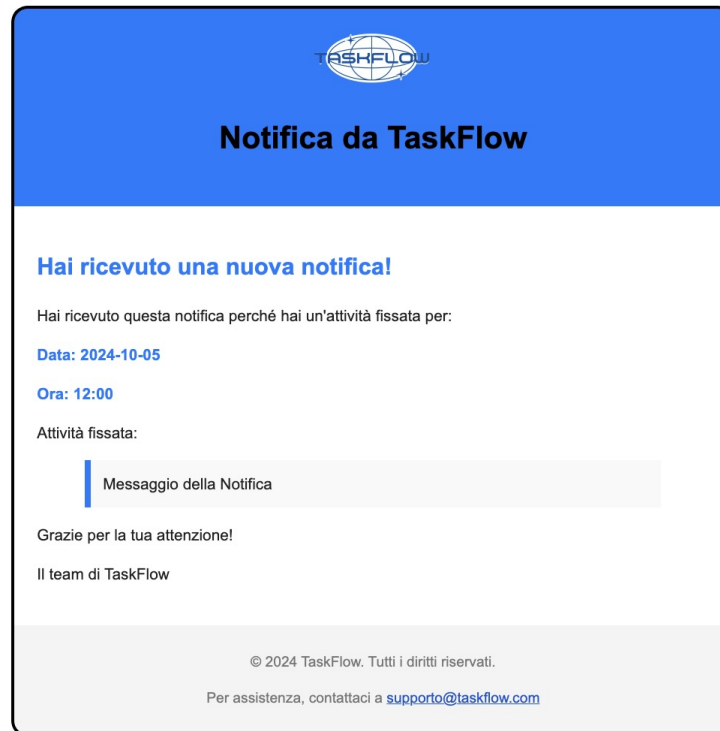


Figura 4.2: Notifica ricevuta via email

Per garantire l'invio delle notifiche viene usato un processo schedato che ogni minuto controlla se ci sono notifiche che devono essere inviate, e provvede a raccogliere le email degli utenti specificati nella notifica e a inviare loro una mail con le specifiche. Vedi figura 4.4.

```
@Service
public class NotificationService {
    @Autowired
    private NotificationDAO notificationDAO;
    @Autowired
    private FieldDAO fieldDAO;
    @Autowired
    private MailService mailService;

    @Scheduled(fixedRate = 60000) // ogni minuto
    public void checkForExpiredNotifications() throws MessagingException {
        ZonedDateTime nowInRome = ZonedDateTime.now(ZoneId.of("Europe/Rome"));
        LocalDateTime now = nowInRome.toLocalDateTime();
        LocalDateTime nextMinute = now.plusMinutes(1);

        // Trova solo le notifiche che scadono nel prossimo minuto
        List<Notification> notifications = notificationDAO.
            findExpiringNotifications(now, nextMinute);
    }
}
```



```
        for (Notification notification : notifications) {
            sendNotificationEmail(notification);
        }
    }

    private void sendNotificationEmail(Notification notification) throws
        MessagingException {
        for (User receiver : notification.getReceivers()) {

            List<Date> date = this.fieldDAO.findByNotification(notification);

            if (date == null || date.isEmpty()) {
                System.out.println("No dates found for notification: " +
                    notification.getId());
            } else {
                System.out.println("Found dates: " + date);
            }

            this.mailService.sendEmail(receiver.getEmail(), "Notifica da
                TaskFlow", notification.getMessage(), date.get(0));
        }
    }
}
```

Codice 4.4: NotificationService

Come si vede dalla figura 4.4 l'introduzione di un servizio schedulato con una certa frequenza è estremamente semplificato in Spring. Si rende necessario il solo utilizzo dell'annotazione `@Scheduled(fixedRate = n)` dove n indica la frequenza di scheduling espressa in millisecondi.

Servizi di inserimento Gli utenti hanno la possibilità di inserire attributi nelle varie entità modellate. Per garantire queste funzionalità sono stati predisposti appositi metodi all'interno dei servizi. Questi metodi prendono in ingresso oggetti di tipo DTO, li mappano nelle relative entità tramite gli appositi mappers (nell'evenienza aggiungendo tutti gli attributi ignorati nel mapper) ed infine li salvano nel database utilizzando i servizi di DAO.

```
public Activity pushNewActivity(ActivityDTO activityDTO) {

    ArrayList<FieldDTO> fieldsDto = new ArrayList<FieldDTO>();

    if (activityDTO.getFields().size() != 0) {
        fieldsDto = activityDTO.getFields();
    }
}
```

```
ArrayList<Field> fields = new ArrayList<Field>();

for (FieldDTO movingFieldDto : fieldsDto) {
    fields.add(
        this.fieldServiceManager
            .getFieldService(movingFieldDto)
            .pushNewField(movingFieldDto));
}

Activity activity = EntityFactory.getActivity();
activity.setName(activityDTO.getName());
activity.setFields(fields);

activity = this.activityDao.save(activity);

return activity;
}
```

Codice 4.5: Aggiunta di una nuova activity

Nel codice 4.5 è possibile vedere un esempio di inserimento: inizialmente vengono istanziati gli attributi annidati dentro activity che la rappresentazione DTO presenta solo come riferimenti nel database, l'oggetto DTO viene poi convertito nella sua versione entity, nella quale vengono aggiunti gli attributi precedentemente mappati (fields). Infine tramite l'uso del metodo save del DAO l'entità viene salvata nel database.

Servizi di modifica

In modo simile all’inserimento, un utente ha la capacità di modificare un attributo di una entità già presente. Se l’attributo è un metadato, viene modificato direttamente, se invece è un entità del progetto, viene richiamato il servizio di modifica della stessa.

```
// abstract class per i Field services
@Service
public abstract class FieldService {
    //...
    abstract public Field updateField(FieldDTO fieldDto);
    //...
}

// esempio di implementazione per il caso di field type Single Selection
@Service
public class SingleSelectionService extends FieldService {
    @Override
    public Field updateField(FieldDTO fieldDto) {

        SingleSelection field = (SingleSelection) this.fieldDao.findById(
            fieldDto.getId()).orElse(null);

        if (field == null){
            throw new IllegalArgumentException("Single selection not found");
        }

        SingleSelectionDTO singleSelectionDTO = (SingleSelectionDTO) fieldDto;

        field.setValue(singleSelectionDTO.getValue());

        this.fieldDao.save(field);

        return field;
    }
}
```

Codice 4.6: FieldService abstract class e implementazione di esempio per la modifica del tipo SingleSelection.

Il codice 4.6 fornisce un esempio di implementazione di una modifica ad una entità del domain model. In questo caso si deve vedere a questo metodo come un metodo utilizzato nel caso di grosse variazioni ad un *Field* (questo spiega l’utilizzo di un *FieldDTO* in ingresso).

Un servizio di modifica più semplice che avviene attraverso un parametro ingresso è invece implementato in modo analogo a quanto si vede nel codice 4.7.

```
@Service
public class ActivityService {
    //...
    public Activity renameActivity(String activityId, String newName) {
        Activity activity = this.activityDao.findById(activityId).orElse(null);

        if (activity == null) {
            throw new IllegalArgumentException("Activity not found");
        }

        activity.setName(newName);

        return this.activityDao.save(activity);
    }
    //...
}
```

Codice 4.7: Metodo per la modifica del nome di una activity.

Servizi di lettura Per poter mostrare lato frontend tutte le risorse di cui un utente ha bisogno, vengono esposti lato backend diversi servizi di lettura. Tali servizi prendono in ingresso una stringa contenente l'Id dell'entità da leggere, che tramite relativo DAO viene reperita dal database, mappata in un oggetto DTO, ed infine restituita come valore di ritorno.

```
public ActivityDTO getActivityById(String activityId) {
    Activity activity = this.activityDao.findById(activityId).orElse(null);

    if (activity == null){
        throw new IllegalArgumentException("Activity not found");
    }
    return this.activityMapper.toDto(activity);
}
```

Codice 4.8: get di una attività

Analoghe funzioni vengono esposte per altri servizi fornendo quindi tutte le possibili operazioni di lettura. Per esempio le API di lettura delle organizzazioni di un utente, dei progetti all'interno di un organizzazione e delle attività all'interno di un progetto.

Servizi di cancellazione Gli ultimi servizi necessari sono quelli di cancellazione. Questi prendono in ingresso una stringa contenente l'Id dell'entità da cancellare, provvedono a instanziarla reperendola dal database, con essa cancellano tutte le eventuali entità annidate, ed infine cancellano la stessa entità tramite DAO.

```
public void deleteActivityAndFields(String activityId) {
    Activity activity = this.activityDao.findById(activityId).orElse(null);

    if (activity == null){
        throw new IllegalArgumentException("Activity not found");
    }

    this.fieldDao.deleteAll(activity.getFields());

    Project project = this.projectDao.findProjectByActivity(activityId);
    project.deleteActivity(activity);

    this.projectDao.save(project);

    this.activityDao.delete(activity);
}
```

Codice 4.9: delete di una attività

L'esempio 4.9 mostra la cancellazione di una attività. L'attività presa in considerazione viene istanziata tramite DAO e le vengono tolte tutte le entità annidate (in questo caso i field), questo si rende necessario poichè i DAO implementati da Spring non supportano il cascading in fase di cancellazione. In questo esempio si rende necessario anche la rimozione di tale attività dal project, che altrimenti presenterebbe un riferimento ad un oggetto non più esistente. L'oggetto viene infine eliminato tramite l'apposito metodo del DAO.

4.2.4 Controllers

Il compito dei controller è quello di validare i dati che vengono inviati tramite le API, in modo da passarli poi ai servizi, in caso essi siano corretti. Per definire un controller in Spring si utilizza l'annotazione `@RestController`. Questa permette di utilizzare varie altre annotazioni per la definizione del verbo dell'endpoint esposto.

Attraverso l'annotazione `@RequestMapping("")` è possibile specificare l'URL di base del controller in modo tale non dover ripetere una stessa base per ogni metodo.

All'interno del progetto i controller prevedono anche la validazione degli argomenti in ingresso e per questo viene utilizzato il tag `@Valid`, come si vede nell'esempio 4.10. Questo tag si occupa di controllare che tutti i campi necessari siano riempiti correttamente.

```
@RestController
@RequestMapping("/api/user")
public class FieldController {
    //...
    @PreAuthorize("@checkUriService.check(authentication, #userId, #
        organizationId, #projectId, #activityId, #fieldId)")
    @PatchMapping("/{userId}/myOrganization/{organizationId}/projects/{
        projectId}/activities/{activityId}/fields/{fieldId}")
    public ResponseEntity<?> updateField(
        @PathVariable String fieldId,
        @PathVariable String userId,
        @PathVariable String organizationId,
        @PathVariable String projectId,
        @PathVariable String activityId,
        @RequestBody FieldDTO fieldDto)
    {
        try {
            return ResponseEntity.status(HttpStatus.OK)
                .body(this.fieldServiceManager.getFieldService(fieldId).
                    updateField(fieldDto));
        } catch (Exception exception) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(exception.
                getMessage());
        }
    }
    //...
}
```

Codice 4.10: Esempio di controller, implementazione metodo di update di un Field. Si noti come l'ereditarietà per i FieldDTO e per i FieldService permetta di sfruttare il FieldServiceManager per eseguire il metodo di update sul giusto servizio per il tipo in ingresso.

Come si vede nel codice 4.10 l'annotazione *@PathMapping* permette di specificare l'URL completo per l'end-point e di specificare l'operazione (in questo esempio operazione di *PATCH*).

L'utilizzo della libreria *jakarta.validation* permette inoltre di utilizzare l'annotazione *@Valid* che in molti controller risulta fondamentale per verificare la correttezza dei campi ricevuti.

```
public abstract class FieldDTO {
    String id;
    String uuid;
    FieldType type;
    @NotBlank(message = "FieldDefinition non puo' essere vuoto")
    String fieldDefinitionId;
}

public class ActivityDTO {

    private String id;
    @NotNull
    private String name;
    @Valid
    private ArrayList<FieldDTO> fields;
    private String uuid;
}
```

Codice 4.11: FieldDTO

Nel codice 4.11 si può vedere come la validazione del campo *fieldDefinitionId* possa essere effettuata usando la combinazione dell'annotazione *@Valid* nel controller e *@NotBlank* nella definizione del *FieldDTO* (o una qualsiasi altra annotazione della stessa libreria).

4.2.5 API Rest

Nel backend dell'applicazione implementata, sono state sviluppate numerose API REST per gestire le risorse e interagire con i servizi del sistema. Come illustrato in precedenza, Spring Boot è stato usato come framework per lo sviluppo delle API, in quanto fornisce una gestione efficace delle richieste HTTP e dei servizi RESTful.

Un'API REST è composta da vari elementi, come i metodi HTTP (GET, POST, PUT, DELETE), URL delle risorse, e le autorizzazioni necessarie. Ogni endpoint gestisce richieste specifiche relative a un'entità o a un'azione all'interno del sistema.

In questo progetto, le API REST utilizzano annotazioni di Spring come *@PostMapping*, *@GetMapping*, *@PutMapping*, *@DeleteMapping* per definire i metodi HTTP e gli URL associati. Inoltre, gestiamo la sicurezza e i permessi tramite annotazioni come *@PreAuthorize*, utilizzando un sistema di controllo delle autorizzazioni basato sui ruoli.

Struttura delle URI

Le URI delle API sono state strutturate per rispecchiare la gerarchia delle pagine del Page Navigation Diagram. Di seguito è riportato un esempio di una URI:

```
http://endpoint/api/user/67010fd07999bd3345d9ecd4/myOrganization/67010
fd27999bd3345d9edc3/projects/67010fd27999bd3345d9ed3e/activities/67010
fd27999bd3345d9ed07/field
```

Come si può vedere, ogni URI inizia con "user" e l'ID dell'utente ottenuto in seguito al login. In seguito si hanno: myOrganization, projects, activities e field; sempre alternati con i relativi ID. Questa struttura consente un routing chiaro e permette di mantenere un controllo dettagliato sulle risorse a cui si accede.

4.3 Sicurezza e Autenticazione

In questa sezione vengono illustrate le principali strategie di sicurezza e autenticazione adottate nel progetto, con l'obiettivo di proteggere i dati degli utenti e garantire l'accesso sicuro alle risorse sensibili.

4.3.1 Gestione dell'Autenticazione

Nel progetto, *Spring Security* è stato utilizzato per garantire la protezione dell'applicazione attraverso l'autenticazione e l'autorizzazione degli utenti. È stato implementato un sistema che gestisce l'accesso alle risorse, permettendo di definire quali API sono liberamente accessibili e quali richiedono un'autenticazione. Questa struttura consente di separare le risorse pubbliche da quelle riservate agli utenti autenticati, migliorando la sicurezza dell'applicazione.

Per la gestione degli utenti, il sistema sfrutta un meccanismo di autenticazione basato su *BasicAuth*, che consente di caricare i dettagli degli utenti registrati e di verificarne le credenziali in fase di autenticazione. Questo permette di associare agli utenti i permessi necessari per accedere alle risorse protette, migliorando il controllo e la gestione delle autorizzazioni all'interno dell'applicazione.

4.3.2 Autorizzazione e Ruoli

Nel progetto sono stati implementati due livelli di autorizzazione: *user* e *owner*, che garantiscono una gestione precisa degli accessi alle risorse e alle funzionalità dell'applicazione. Le API pubbliche, come la registrazione degli utenti, sono liberamente accessibili senza autenticazione, permettendo agli utenti non registrati di creare un nuovo account.

Per quanto riguarda i servizi più sensibili riservati agli *owner*, sono state implementate logiche di controllo aggiuntive per verificare che l'utente sia effettivamente un *owner* dell'organizzazione associata. Questo controllo viene eseguito prima di consentire l'esecuzione di operazioni privilegiate, come la modifica dei dati delle organizzazioni, la gestione dei progetti e la gestione degli utenti.

4.3.3 Integrità delle URI

Un altro layer di sicurezza e allo stesso momento che permette integrità delle URI è portato dalle logiche di controllo che avvengono per la totalità delle API esposte dai *controllers*.

All'interno del progetto è stato implementato infatti una classe dedicata che fornisce il servizio di *check* della URI per i vari livelli.

La classe *CheckUriService* permette infatti di eseguire il controllo che la URI faccia riferimento alle corrette risorse all'interno dell'applicativo e che, tali risorse, siano in una struttura valida.

Infatti facendo riferimento al codice di esempio 4.12 si vede come sia possibile controllare che un certo *project* sia effettivamente all'interno dei progetti della rispettiva *organization* fornita in *organizationId*.

```
public boolean check(Authentication authentication, String userId, String
    organizationId, String projectId){
    this.check(authentication, userId, organizationId);

    Project project = this.projectDao.findById(projectId).orElse(null);
    this.checkNotNullOrThrow(project, projectId);

    if (!(this.organization.getProjects().contains(project))){
        throw new IllegalArgumentException("Project " + projectId + " isn't in
            organization " + organizationId);
    }

    this.project = project;
```

```
    return true;  
}
```

Codice 4.12: Metodo di *check* a livello *project*. Si noti come la funzione *check* richiami l'omonima funzione per il livello precedente permettendo una concatenazione delle chiamate ed una facile estensione a controlli ulteriori.

4.3.4 Sicurezza dei Dati

Per la gestione delle password degli utenti, è stata adottata una soluzione che sfrutta Bcrypt per la crittografia delle password. Bcrypt è un algoritmo di hashing progettato appositamente per la protezione delle password, caratterizzato da una funzione di derivazione delle chiavi che utilizza il salting. Grazie ad esso, Bcrypt genera un hash unico per ogni password, anche se due utenti inseriscono la stessa password, garantendo così una maggiore protezione e rendendo difficile identificare password uguali.

4.4 Deployment

Ambiente di produzione L'ambiente di produzione del progetto è stato realizzato utilizzando una configurazione basata su container Docker. Tre container principali compongono l'architettura: uno per il backend (Spring), uno per il frontend (Angular), e uno per il database (MongoDB). Questi container sono coordinati attraverso Docker Compose, il quale consente una gestione efficiente e automatizzata del ciclo di vita dell'applicazione, garantendo l'isolamento tra le varie componenti e la scalabilità dell'infrastruttura.

Il backend Spring Boot è configurato per girare su una porta esposta tramite il container dedicato, collegato al database MongoDB per gestire la persistenza dei dati, mentre il frontend Angular è ospitato su un container separato. MongoDB, come database, risiede in un container autonomo, gestito tramite un volume Docker per garantire la persistenza dei dati anche in caso di riavvio o aggiornamento dei container.

Configurazione del Deployment La configurazione del deployment è gestita tramite un file *docker-compose.yml*, che definisce i tre container e le loro dipendenze. Il container del backend è configurato per avviarsi utilizzando un comando Maven.

Il container MongoDB, ha a disposizione un volume dedicato ai dati, in modo da garantire che le informazioni rimangano persistenti tra le sessioni di esecuzione.

```
services:
  backend:
    build:
      context: ./TaskFlow_serverSide
      dockerfile: Dockerfile
    ports:
      - 3000:8080
    depends_on:
      - mongodb
    environment:
      - SPRING_DATA_MONGODB_URI=mongodb://mongodb:27017/taskflowdb
    volumes:
      - ./TaskFlow_serverSide:/app
    command: mvn spring-boot:run

  frontend:
    build:
      context: ./TaskFlow_clientSide
      dockerfile: Dockerfile
    ports:
      - 4200:4200
    volumes:
      - ./TaskFlow_clientSide:/app
      - /app/node_modules

  mongodb:
    image: mongo:latest
    ports:
      - 27017:27017
    volumes:
      - mongodb_data:/data/db
```

Codice 4.13: File *docker-compose.yml*

Integrazione con Docker Compose Docker Compose è utilizzato per orchestrare il deployment dell'applicazione, definendo la configurazione per ogni container e le loro interazioni. Il servizio del backend è strettamente legato a MongoDB tramite la variabile d'ambiente *SPRING_DATA_MONGODB_URI*, che garantisce una corretta connessione tra il servizio Spring Boot e il database MongoDB. Ogni container ha un proprio volume dedicato per garantire la persistenza dei dati e delle dipendenze (vedi codice 4.13)

5

Testing

Il processo di testing è una fase cruciale nello sviluppo di software, in quanto permette di validare il corretto funzionamento del codice e di identificare potenziali bug. In questo progetto, è stato adottato un approccio di unit testing e integration testing per garantire la qualità del codice e la sua conformità ai requisiti.

Gli unit test verificano il comportamento di singoli metodi o classi. Nel nostro progetto, i test sono focalizzati su singole unità funzionali, come le entità del domain model, per assicurarci che ogni operazione sia eseguita correttamente. Gli integration test verificano che i diversi componenti del sistema (ad esempio DAO e servizi) interagiscano correttamente tra di loro.

Il principale strumento utilizzato per effettuare i test è *Spring Boot Test*, un framework all'interno di Spring Boot progettato per facilitare il testing delle applicazioni *Spring*. In modo simile a *JUnit* Spring Boot test permette l'utilizzo di annotazioni utili per il testing di applicazioni in Spring.

All'interno delle classi di test sono state utilizzate varie annotazioni le cui principali sono le seguenti:

@Test Questa annotazione indica che un metodo è un test case e dovrebbe essere eseguito come parte di una suite di test. Ha l'obiettivo di definire i metodi di test veri e propri per

verificare il comportamento di componenti o funzionalità (`@Test void myTest() // logica del test`).

@Autowired Utilizzata per iniettare automaticamente le dipendenze nei componenti Spring (bean) gestiti dal contesto di Spring. Spring risolve e inietta le dipendenze dove è annotato. Solitamente serve per iniettare un servizio, repository o altro componente (`@Autowired private MyService myService;`).

Le applicazioni spesso hanno dipendenze complesse. `@Autowired` aiuta a gestire queste dipendenze in modo automatico, garantendo che il contesto contenga le istanze corrette delle classi necessarie, questo è particolarmente utile nei test, dove le dipendenze devono essere facilmente disponibili per i componenti testati.

5.1 Unit testing

Nell'elaborato è stato effettuato unit testing delle principali funzioni delle classi nel *domain model* e nelle classi all'interno del package *business logic*.

Lo unit testing ha permesso di sviluppare l'intero progetto con la possibilità di testare in modo continuativo il software e di velocizzare il processo di sviluppo.

Dal momento che molto spesso le operazioni del domain model si limitano a *getter/setter* è stato deciso di implementare a questo livello i vari test sulla correttezza della persistenza dei dati.

Per l'architettura di test è stato utile la parallela implementazione di una classe apposita che permettesse operazioni di supporto ai test. Dal momento che le varie entità possono avere varie dipendenze fra di loro la classe *TestUtil* contiene al suo interno varie funzioni che hanno permesso di eliminare la complessità della creazione delle varie entità in tutti i test effettuati.

5.1.1 Esempio di Unit Test: Testing su Activity

Come si può vedere dal codice 5.1 data la semplicità dei metodi che le classi del domain model espongono un singolo metodo della classe *ActivityTest* permette di testare varie funzionalità esposte dalla classe *Activity*.

```
@DataMongoTest
@ActiveProfiles("test")
@ComponentScan(basePackages = "com.example.taskflow")
```

```
public class ActivityTest {
    // @Autowired necessari omissi per semplicità'
    // activity utilizzata nei test introdotta nel database dal metodo
    @BeforeEach
    private Activity activity;

    ...

    @Test
    public void testFieldMethods(){
        // creazione di field casuali
        FieldDefinition fieldDef1 = this.testUtil.getFieldDefinition(FieldType.
            TEXT);
        FieldDefinition fieldDef2 = this.testUtil.getFieldDefinition(FieldType.
            NUMBER);
        Field field1 = this.testUtil.getField(fieldDef1);
        Field field2 = this.testUtil.getField(fieldDef2);

        // test su ordinamento dei fields
        this.activity.addField(field1);
        this.activity.addField(field2);
        assertEquals(this.activity.getFields().get(0), field1);
        assertEquals(this.activity.getFields().get(1), field2);

        // test di rimozione field
        this.activity.removeField(field1);
        this.activity.removeField(field2);
        assertTrue(this.activity.getFields().isEmpty());

        // test di aggiunta con metodo addFields()
        ArrayList<Field> fields = new ArrayList<Field>();
        fields.add(field1);
        fields.add(field2);
        this.activity.addFields(fields);
        assertEquals(this.activity.getFields().get(0), field1);
        assertEquals(this.activity.getFields().get(1), field2);
    }

    ...
}
```

Codice 5.1: Classe *ActivityTest* e metodo *testFieldMethods*

Lo unit test è stato eseguito anche per verificare la correttezza delle operazioni di persistenza delle modifiche. Il codice 5.2 mostra come sia stato effettuato all'interno della stessa classe di esempio *ActivityTest* una verifica dell'integrità dei dati una volta salvati nel database.

```
@Test
public void testActivityFieldsModificationDB(){
```

```
// creazione di field casuali e persistenza
FieldDefinition fieldDefinition1 = this.testUtil.
    pushGetFieldDefinitionToDatabase(FieldType.TEXT);
FieldDefinition fieldDefinition2 = this.testUtil.
    pushGetFieldDefinitionToDatabase(FieldType.NUMBER);
Field field1 = this.testUtil.getField(fieldDefinition1);
Field field2 = this.testUtil.getField(fieldDefinition2);
field1 = this.fieldDao.save(field1);
field2 = this.fieldDao.save(field2);

// aggiunta dei field all'activity e persistenza
this.activity.addField(field1);
this.activity.addField(field2);
this.activityDAO.save(this.activity);

// test della persistenza dell'aggiunta dei fields
Activity activitySearchedInDB = activityDAO.findById(this.activity.getId())
    .orElse(null);
assertEquals(activitySearchedInDB.getFields().get(0), field1);
assertEquals(activitySearchedInDB.getFields().get(1), field2);

// rimozione di un field dall'activity e persistenza
this.activity.removeField(field2);
this.activityDAO.save(activity);

// test della persistenza della rimozione del field
activitySearchedInDB = activityDAO.findById(this.activity.getId()).orElse(
    null);
assertEquals(this.activity.getFields().size(), 1);
assertEquals(this.activity.getFields().get(0), field1);
}
```

Codice 5.2: Metodo di testing della persistenza delle informazioni di una *Activity*

5.2 Integration testing

Successivamente alla parte di unit testing, sono stati implementati casi di test anche a livello di integrazione, concentrandosi sulle classi della business logic, principalmente tra services e controllers. L'obiettivo principale di questi test è stato verificare che le varie componenti del sistema funzionino correttamente quando interagiscono tra loro, simulando scenari più realistici rispetto ai singoli test unitari.

Per facilitare e velocizzare lo sviluppo dei test di integrazione, è stata sviluppata una piccola libreria chiamata TestUtil. Questa libreria fornisce un'API progettata per la popolazione del database con entità generate dinamicamente, gli attributi di tali entità sono creati in mo-

do randomico. L'uso di questa utility consente di evitare la duplicazione di codice e ridurre i tempi necessari alla preparazione dei casi di test. Ogni test può così concentrarsi sulla logica applicativa e sui flussi da verificare, senza preoccuparsi della predisposizione manuale del database.

5.2.1 Esempio di Integration Test: Testing su ActivityService

Effettuare operazioni come quelle viste nei codici 5.1 e 5.2 ha permesso di avere una solida base di test per le successive operazioni più complesse. A scopo di esempio si mostra nel codice 5.4 il testing del metodo *delete()* esposto dal servizio *ActivityService*.

Appare chiaro come il testing di un service possa essere già considerato una forma di **integration testing** dal momento che la verifica della correttezza di metodi esposti dai service tocca vari aspetti del software.

Inoltre il test dei metodi dei services risulta essere una operazione che necessariamente deve essere effettuata in seguito alla fase di unit testing. Infatti le verifiche effettuate dallo unit testing permettono di concentrarsi, in questa fase, nel test più oculato dei metodi esposti nei service, dando per scontato che molte operazioni basilari siano state già verificate in precedenza.

```
@Test
public void testDelete(){
    // esempio di utilizzo della generazione automatica dell'intero database
    Organization organization = this.testUtil.getEntireDatabaseMockup(1, 10, 5,
        30).get(0);

    ArrayList<Project> projects = organization.getProjects();

    for (Project project : projects){
        ArrayList<Activity> activities = project.getActivities();

        for (Activity activity : activities){
            ArrayList<Field> fields = activity.getFields();

            this.activityService.deleteActivityAndFields(activity.getId());
            assertFalse(this.activityDao.findById(activity.getId()).isPresent())
                ;
            for (Field field : fields){
                assertFalse(this.fieldDao.findById(field.getId()).isPresent());
            }

            Project projectFromDb = this.projectDao.findById(project.getId()).
                orElseThrow();
            assertFalse(projectFromDb.getActivities().contains(activity));
        }
    }
}
```



```
    }  
  }  
}
```

Codice 5.3: Codice per il testing dell'operazione di eliminazione di una attività (lato service)

Il codice 5.4 mostra come, soprattutto per il testing sui services, si sia resa necessaria l'implementazione di una classe *TestUtil* che permetta l'instaurazione di tutte quelle connessioni fra le varie entità per effettuare una verifica approfondita per queste operazioni.

Un metodo molto utile infatti risulta essere *getEntireDatabaseMockup()* che permette la generazione parametrica dell'intero database e che ha snellito il codice di testing per i vari services.

```
public ArrayList<Organization> getEntireDatabaseMockup(int nOrganization, int  
    nProjectForOrganization, int nActivitiesForProject, int nUsers) {  
    // pulizia database  
    this.cleanDatabase();  
    // generazione di tutte le entita' in modo randomico seguendo le linee  
        indicate dai parametri in ingresso  
    ...  
}
```

Codice 5.4: Funzione di generazione del database della classe *TestUtil*

5.2.2 Postman

Sempre per verificare la corretta integrazione e funzionamento dei componenti del sistema, sono stati condotti test sulle API esposte dai *controller*. Questo è stato possibile grazie all'uso di *Postman*.

Come si vede in figura 5.1, è riportato l'esempio di registrazione di un nuovo utente. Vengono infatti specificati username, email e password, e in risposta si ottiene l'*userDTO* dell'utente appena creato.

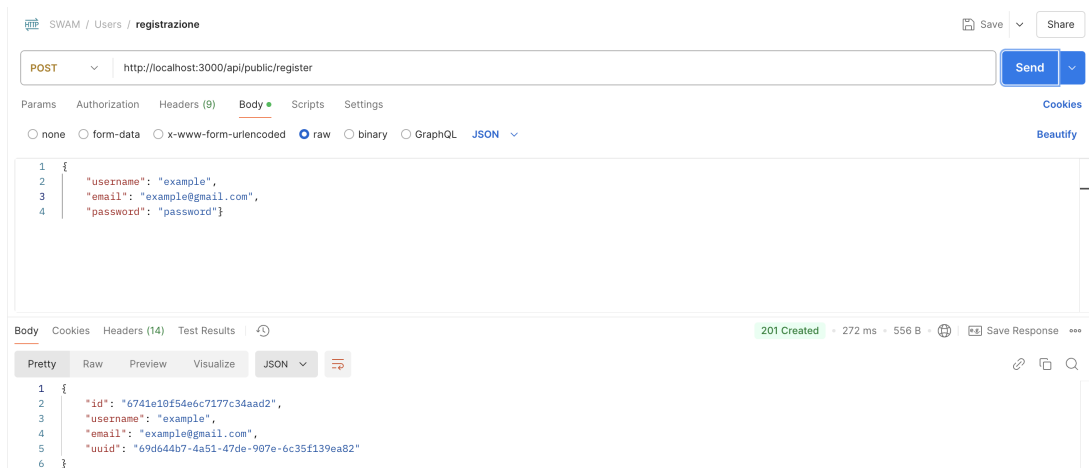


Figura 5.1: Query di registrazione utente

In figura 5.2 è invece riportato l'esempio di creazione di una organizzazione. In questo caso viene fornito il nome dell'organizzazione e in risposta si ottiene un'organizzazione con nessun progetto, nessun membro e l'utente che ha eseguito la query come Owner.

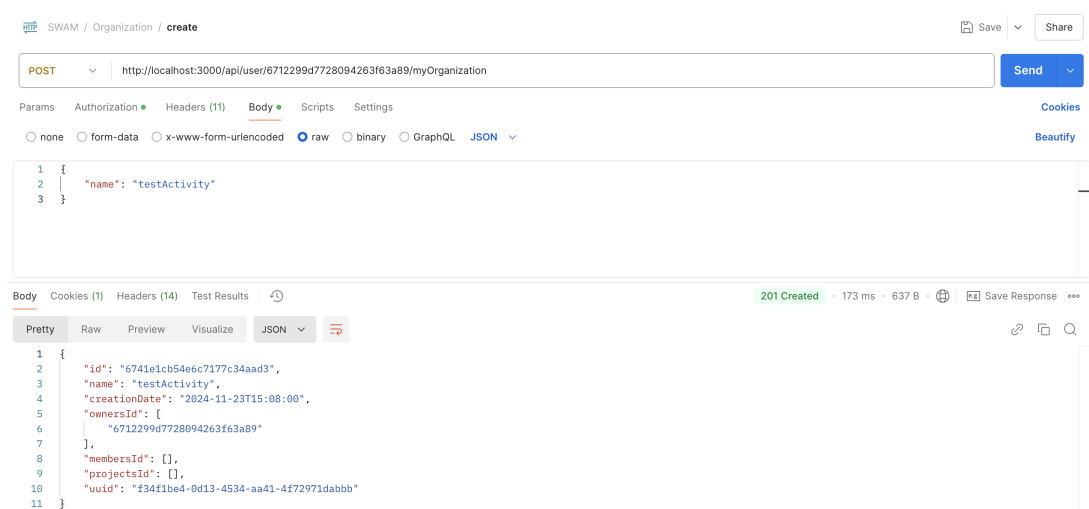


Figura 5.2: Query di creazione di una organizzazione

Infine in figura 5.3 si trova la creazione di una nuova attività. Nel body viene incluso il nome dell'activity e un field di tipo TEXT, e in risposta si ottiene il resoconto della nuova activity contenente il field.

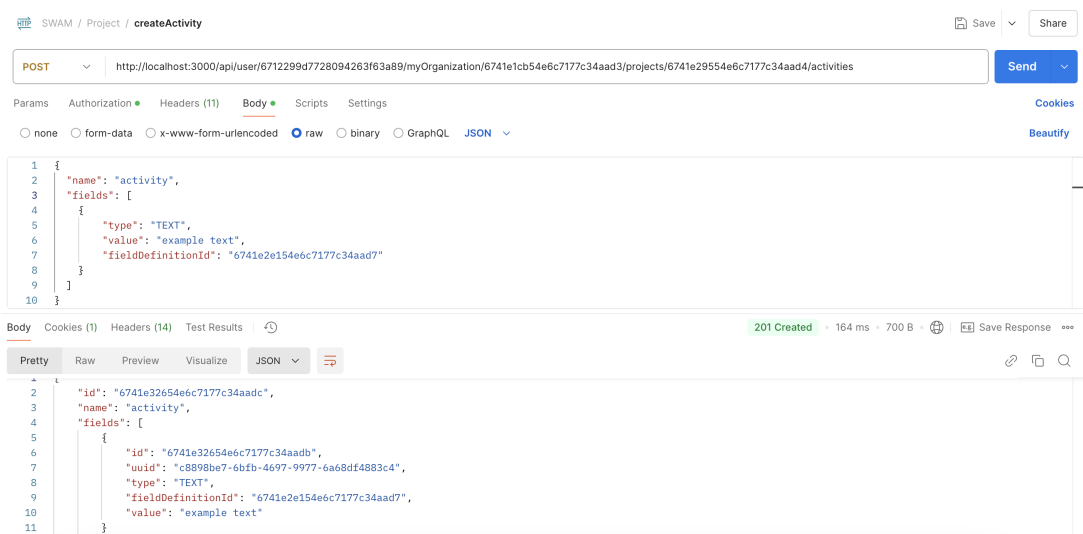


Figura 5.3: Query di creazione di una activity

6

Conclusione

6.1 Sviluppi futuri

Nella fase di sviluppi futuri, sarà necessario implementare la logica del frontend, che è stata delineata nei dettagli all'interno di questa relazione. Attualmente, l'interfaccia utente è rappresentata solo da mockup, che forniscono una visualizzazione di come sarà strutturata l'applicazione una volta completata. Sebbene il frontend non sia stato ancora completamente sviluppato, è stato predisposto un container Docker che include un'implementazione base di Angular, correttamente collegata al backend tramite le API sviluppate.

Questo ambiente di sviluppo fornisce una base su cui costruire, consentendo di testare l'integrazione tra il backend e il frontend.

La natura del tool permette inoltre l'aggiunta di varie funzionalità in futuro. Alcuni esempi possono essere:

- *Chat testuali/vocali.* La possibilità di effettuare chat testuali e/o vocali fra gli utenti permetterebbe una migliore esperienza utente.
- *Single sign on.* Permettere la registrazione all'applicativo tramite piattaforme quali Facebook, Google account, LinkedIn. Questa funzionalità fornirebbe un servizio di registrazione più immediato e di conseguenza una esperienza utente migliore.