

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”

CORSO DI LAUREA IN INFORMATICA



Sviluppo API REST di un'applicazione web per la pianificazione delle risorse aziendali

Tesi di laurea

Relatore

Prof. Paolo Baldan

Laureando

Marco Brigo

ANNO ACCADEMICO 2022-2023

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

— Oscar Wilde

Dedicato a ...

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di X ore, dal laureando Marco Brigo presso l'azienda Omicron Consulting Srl di Padova nel periodo che va dal 19 Giugno 2023 a X Agosto 2023.

Il progetto consisteva nello sviluppo di un software di richieste di pianificazioni delle risorse aziendali, verso determinati incarichi e secondo determinati parametri. L'obiettivo dello stage era inserirmi all'interno del progetto, più precisamente nel lato back-end. Il mio lavoro si concentrava sulla progettazione e l'implementazione di un'API REST e sulla creazione e la gestione di nuove tabelle nel database aziendale, utilizzate per la fruizione delle funzionalità da me implementate.

L'attività si è svolta partendo da uno studio preliminare delle tecnologie da utilizzare tramite esercitazioni e materiale fornito, passando per una progettazione dell'architettura e del database.

“Life is really simple, but we insist on making it complicated”

— Confucius

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Nome Cognome, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ho desiderio di ringraziare poi i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.

Padova, Mese AAAA

Nome Cognome

Indice

1	Introduzione	1
1.1	Convenzioni tipografiche nel documento	1
1.2	L'azienda ospitante	1
1.3	Il progetto	2
1.3.1	Presentazione	2
1.3.2	Motivazione del progetto	2
1.3.3	Il mio ruolo nel progetto	2
1.4	Organizzazione del testo	2
2	Analisi dei Requisiti	4
2.1	Scopo del capitolo	4
2.2	Descrizione generale	4
2.3	Semplificazioni adottate nei casi d'uso	4
2.4	Attori	5
2.5	Casi d'uso	5
2.5.1	Primo caso	5
2.6	Tracciamento dei requisiti	5
3	Background tecnologico	6
3.1	Scopo del capitolo	6
3.2	Tecnologie	6
3.3	Strumenti	8
3.3.1	Strumenti di sviluppo	8
3.3.2	Strumenti di supporto	9
4	Progettazione	11
4.1	Scopo del capitolo	11
4.2	Architettura REST	11
4.2.1	Convenzioni di denominazione REST	12
4.3	Spring MVC	12
4.3.1	Model	13
4.3.2	View	13
4.3.3	Controller	13
4.4	Pattern Client-Server	14
4.5	Design Pattern	15
4.6	Progettazione del database	16
4.6.1	Configurazione di Docker	16
4.6.2	Modello dati	17

4.7	Configurazione iniziale del progetto	17
4.7.1	Spring Initializr	17
4.7.2	Contenuto del pacchetto	18
4.7.2.1	Pom.xml	18
4.7.2.2	Application.properties	20
5	Codifica	21
5.1	Scopo del capitolo	21
5.2	Endpoint sviluppati dell'API REST	21
5.3	Organizzazione del codice	22
5.3.1	Common e config	22
5.3.2	Entities	23
5.3.2.1	DTO Entities	24
5.3.2.2	Esempio di un'entità	25
5.3.3	Package dto	27
5.3.3.1	Common	27
5.3.3.2	Requests e Body	28
5.3.3.3	Responses	29
5.3.4	Repository-Service-Controller	29
5.3.4.1	Controller	30
5.3.4.2	Repository	31
5.3.4.3	Service	32
5.3.5	Swagger UI	34
6	Verifica e Validazione	35
6.1	Scopo del capitolo	35
6.2	Processo di verifica	35
6.2.1	Analisi statica	35
6.2.2	Analisi dinamica	35
6.2.2.1	Debugging	36
6.2.2.2	Postman	36
6.3	Testing	37
6.3.1	Esempio di un test effettuato	37
6.4	Validazione	39
7	Deploy dell'applicazione	40
7.1	Scopo del capitolo	40
7.2	Deploy	40
7.2.1	Dockerfile	40
7.2.2	Costruzione dell'immagine e avvio dell'applicazione	41
8	Conclusioni	42
8.1	Consuntivo finale	42
8.2	Raggiungimento degli obiettivi	42
8.3	Conoscenze acquisite	42
8.4	Valutazione personale	42
	Glossario	43
	Bibliografia	44

Elenco delle figure

1.1	Logo Omicron	1
4.1	Schema Model-View-Controller	12
4.2	Client-Server in una REST API	14
4.3	Interfaccia Spring Initializr	17
4.4	Snippet pom.xml generato	19
4.5	Foto application.properties configurato e utilizzato	20
5.1	Package common e config	22
5.2	Package entità	23
5.3	Esempio di classe DTO di Pianificazione	24
5.4	Esempio di mappatura di un'entità	25
5.5	Esempio di mappatura delle relazioni della tabella RichiestaTestata	26
5.6	Subpackage common	27
5.7	Subpackage requests e body	28
5.8	Subpackage responses e dtoresponses	29
5.9	Schema di collegamento tra i componenti Controller, Service e Repository	29
5.10	Controller sviluppati	30
5.11	Firma di un metodo nell'interfaccia di un Controller	31
5.12	Esempio di controllo di validazione	31
5.13	Repository sviluppati	31
5.14	Esempio di una Repository	32
5.15	Service sviluppati	32
5.16	Metodo deleteMilestone() nel Service	33
5.17	Code snippet della creazione della query basata su filtri inseriti	33
6.1	Interfaccia Postman	36
6.2	Configurazione ambiente di testing	37
6.3	Code snippet test di creazione di una nuova Milestone	38
7.1	Configurazione Dockerfile	40

Elenco delle tabelle

5.1 Verbi Standard HTTP utilizzati	22
--	----

Capitolo 1

Introduzione

1.1 Convenzioni tipografiche nel documento

Durante la stesura del testo sono state adottate le seguenti convenzioni:

- la sezione del glossario contiene i termini ritenuti ambigui o non di uso comune, che necessitano quindi di una loro definizione. Il suo scopo è quello di fornire una comprensione comune del linguaggio utilizzato e di evitare confusione o interpretazioni errate;
- per la prima occorrenza di un termine inserito nel glossario viene utilizzata la seguente nomenclatura: termine_g.

1.2 L'azienda ospitante



Figura 1.1: Logo Omicron

L'azienda Omicron Consulting è specializzata nello sviluppo di software gestionali e di revisione di processi aziendali. Essa è presente nel mercato ICT dal 1980, spiccando su vari settori, in cui hanno effettuato importanti implementazioni in area ICT come: Manufacturing, Automotive, Aerospace, Logistics e altre aree. Con particolare

riferimento al settore Manufacturing si sono specializzati nello sviluppo di progetti di trasformazione ERP (acquisendo la certificazione VAR di SAP), stringendo alleanze strategiche con realtà ICT nazionali ed internazionali.

Offrono servizi di gestione e supporto di sistemi ERP, sviluppo di progetti di Business Intelligence e personalizzazione di sistemi software.

Per completare il pacchetto dei servizi offerti, Omicron ha un'esperienza di alto livello negli ambiti Banking, Finance and Insurance, lavorando con le principali istituzioni bancarie e assicurative italiane.

Omicron oltre alle offerte che dedica ai clienti, si occupa anche di garantire un'alta formazione delle proprie risorse, investendo su progetti di ricerca e sviluppo.

1.3 Il progetto

1.3.1 Presentazione

L'applicativo permette la creazione di richieste di pianificazioni di risorse aziendali per svolgere un determinato incarico. In questo progetto per risorse aziendali si fa sempre riferimento alle risorse umane, quindi al personale o alla forza lavoro dell'azienda.

Il Project Manager_g potrà effettuare richieste di pianificazione di risorse aziendali, chiedendo disponibilità di figure professionali con determinate caratteristiche.

In seguito ad una richiesta accettata, il Program Manager_g distribuirà le risorse più adeguate alla richiesta, creando una pianificazione per ogni risorsa richiesta, specificando parametri quali la durata, l'attività da svolgere e molti altri.

1.3.2 Motivazione del progetto

L'approccio adottato per la gestione delle pianificazioni delle risorse e la loro disponibilità veniva gestito attraverso fogli Excel compilati e rivisti dai Program manager. Le nuove richieste di pianificazioni vengono comunicate ai Program Manager tramite posta elettronica, telefono e chat, rendendo arduo tenere traccia di tutto.

Il progetto nasce dunque dall'esigenza di semplificare la gestione delle richieste e delle pianificazioni, garantendo una visione più rapida della disponibilità delle risorse.

1.3.3 Il mio ruolo nel progetto

Le funzionalità da me sviluppate sono due: le richieste e le pianificazioni di risorse aziendali.

Il tutto è stato realizzato creando nuove tabelle da inserire nel database per la fruizione dei servizi dell'applicativo e lo sviluppo dell'API_g REST_g. Gli endpoint_g dell'API permettono operazioni CRUD_g su richieste, pianificazioni e milestone.

1.4 Organizzazione del testo

Questa sezione è dedicata alla spiegazione della struttura del documento, per dare indicazioni su come è organizzato il testo.

Capitolo 1: introduce il progetto, il mio ruolo all'interno del progetto e il profilo aziendale. Questo capitolo è l'unica parte in cui si parlerà del progetto nella sua totalità rispetto ai capitoli successivi che saranno inerenti esclusivamente al mio lavoro svolto;

Il secondo capitolo descrive i casi d'uso individuati ed i relativi requisiti;

Il terzo capitolo approfondisce ...

Il quarto capitolo approfondisce ...

Il quinto capitolo approfondisce ...

Il sesto capitolo approfondisce ...

Nel settimo capitolo descrive ...

Capitolo 2

Analisi dei Requisiti

2.1 Scopo del capitolo

La seguente sezione di Analisi dei Requisiti rappresenta una dettagliata e approfondita esplorazione delle necessità e delle aspettative che guidano la creazione e lo sviluppo del progetto in questione. Questa analisi è stata condotta al fine di definire chiaramente gli obiettivi e le funzionalità del prodotto, fornendo una base solida per la progettazione e l'implementazione del software.

2.2 Descrizione generale

Ogni caso d'uso è stato schematizzato secondo i seguenti punti:

- **attore coinvolto:** in cui si specifica l'attore;
- **descrizione:** offre una spiegazione più dettagliata del caso d'uso;
- **precondizioni:** rappresenta la condizione che deve essere soddisfatta e verificata affinché il caso d'uso possa essere eseguito con successo;
- **postcondizioni:** rappresenta lo stato dell'attore in seguito all'esecuzione con successo del caso d'uso;
- **estensioni:** in cui si specificano le eventuali estensioni collegate;
- **inclusioni:** in cui si specificano le eventuali inclusioni.

Vengono inserite anche delle immagini dell'UML_g per fornire una spiegazione visiva che può aiutare maggiormente la comprensione.

2.3 Semplificazioni adottate nei casi d'uso

All'interno dei casi d'uso è possibile leggere l'abbreviazione "vis." . Il seguente termine è utilizzato per abbreviare la parola "Visualizzazione".

Per agevolare la lettura delle immagini dei casi d'uso non è stato inserito il collegamento tra gli scenari principali e il database. È dato per scontato quindi, che ogni informazione venga recuperata dal database.

2.4 Attori

Gli attori che possiamo trovare all'interno dei casi d'uso rappresentano due risorse aziendali:

Project Manager

Conosciuto in Italia come il "Responsabile di Progetto", si occupa dell'avvio, pianificazione, esecuzione e controllo di un singolo progetto, seguendo tecniche di project management.

Le sue principali responsabilità sono:

- assicurarsi che i progetti siano allineati con gli obiettivi aziendali;
- coordinare le risorse umane, assicurandosi che vengano utilizzate in modo efficiente;
- stabilisce milestone, scadenze e obiettivi, monitorando lo stato di avanzamento dei progetti;
- comunica con stakeholder, team di progetto e altre parti interessate.

Program Manager

È un ruolo di gestione all'interno di un'organizzazione, ed è responsabile della pianificazione complessiva e del controllo di più progetti che compongono il suo programma. Collabora strettamente col Project Manager ed i loro compiti spesso si sovrappongono ma differiscono di portata, in quanto il Program Manager supervisiona gruppi di progetti gestiti singolarmente dai Project Manager.

2.5 Casi d'uso

2.5.1 Primo caso

2.6 Tracciamento dei requisiti

Capitolo 3

Background tecnologico

3.1 Scopo del capitolo

Questo capitolo tratta delle tecnologie e gli strumenti di sviluppo e di supporto utilizzati per la realizzazione del prodotto.

L'apprendimento delle seguenti tecnologie e strumenti è stato affrontato nella prima parte del tirocinio. Questo periodo di formazione è durato circa due settimane in cui il tutor mi ha fornito materiale ed esercitazioni per poter comprendere al meglio il contesto tecnologico.

3.2 Tecnologie

Java

Java è un linguaggio di programmazione ad alto livello, orientato agli oggetti e fortemente tipizzato, sviluppato originariamente da Sun Microsystems.

La sua popolarità deriva dalla sua portabilità, dalla vasta comunità di sviluppatori e dalle numerose risorse disponibili per l'apprendimento e lo sviluppo.

All'interno del mio progetto è stato utilizzato per lo sviluppo del lato back-end del prodotto.

SQL

SQL, acronimo di Structured Query Language, è un linguaggio di programmazione utilizzato per gestire e manipolare dati in un database relazionale. SQL fornisce una serie di comandi standardizzati che consentono agli sviluppatori e agli amministratori di database di eseguire operazioni come l'interrogazione dei dati, l'aggiornamento dei dati, l'inserimento di nuovi dati e la creazione e gestione degli schemi dei database.

Questo noto linguaggio è stato utilizzato nel progetto per i seguenti motivi:

- creare le tabelle o trigger utili alla fruizione dei servizi del progetto;
- eseguire query per inserire, recuperare, eliminare o modificare dati in base alle richieste.

Microsoft SQL Server

SQL Server è un DBMS (Database Management System) relazionale sviluppato da Microsoft. È una piattaforma dati che si utilizza per creare e gestire database, principalmente in ambito aziendale.

Spring

Spring è un framework_g di sviluppo di applicazioni Java che offre un'ampia gamma di strumenti e librerie per semplificare la creazione di applicazioni aziendali robuste, scalabili e di alta qualità. Spring fornisce anche moduli specifici per la gestione dei dati, lo sviluppo web e la sicurezza, rendendolo uno degli strumenti più utilizzati per lo sviluppo Java.

A questo framework sono associati tanti altri progetti, che hanno nomi composti come Spring Boot, Spring Data e molti altri.

All'interno del progetto Spring è stato utilizzato per sviluppare l'API REST.

Spring Boot

Spring Boot è un modulo del framework di sviluppo Java Spring che semplifica la creazione, la configurazione e l'avvio di applicazioni Java. Fornisce un ambiente pronto all'uso per sviluppare rapidamente applicazioni Spring, eliminando gran parte della complessità associata alla configurazione. Spring Boot utilizza convenzioni intelligenti e configurazioni predefinite per accelerare lo sviluppo, consentendo agli sviluppatori di concentrarsi sulle funzionalità dell'applicazione anziché sulla configurazione di base.

Spring Data

Spring Data è un modulo del framework Spring che fornisce un'astrazione per semplificare l'accesso e la gestione dei dati nelle applicazioni Java. Esso offre un'interfaccia unificata per interagire con una varietà di fonti di dati, tra cui database relazionali, database NoSQL e altri servizi di memorizzazione dati.

Questo modulo è stato utile nel progetto per interfacciare l'applicazione con il database.

JSON

JSON, acronimo di JavaScript Object Notation, è un formato leggero di scambio di dati utilizzato comunemente per rappresentare oggetti e strutture di dati. JSON è ampiamente utilizzato per rappresentare dati in applicazioni web, servizi API, scambio di dati tra client e server, configurazioni di applicazioni e molto altro.

All'interno del progetto permette lo scambio di dati tra il lato front-end ed il lato back-end.

3.3 Strumenti

3.3.1 Strumenti di sviluppo

Gli strumenti di sviluppo sono utilizzati direttamente per creare, implementare e testare le funzionalità dell'applicazione. Essi contribuiscono alla realizzazione delle funzionalità dell'applicazione stessa.

IntelliJ IDEA

IntelliJ IDEA è un potente ambiente di sviluppo integrato (IDE) sviluppato da JetBrains, progettato principalmente per la programmazione in linguaggi come Java, Kotlin, Scala e altri. È noto per la sua ricca serie di funzionalità progettate per migliorare l'efficienza degli sviluppatori e semplificare il processo di sviluppo del software. Il seguente IDE è stato utilizzato per la scrittura del codice in Java.

Maven

Maven è uno strumento di gestione delle build e delle dipendenze che fornisce un sistema di automazione per la compilazione, il testing, il packaging delle applicazioni e il download delle dipendenze.

All'interno di questo progetto Maven è stato utilizzato con Spring Boot per semplificare la gestione delle dipendenze e delle versioni.

DBeaver

DBeaver è un'applicazione di amministrazione di database universale e strumento client SQL. È utilizzato per connettersi, esplorare, gestire e interrogare diversi tipi di database.

All'interno del progetto è stato utilizzato per interagire col database.

Hibernate

Hibernate è un framework di mapping oggetto-relazionale (ORM). L'obiettivo principale di Hibernate è semplificare la gestione e l'accesso ai dati in un database relazionale utilizzando oggetti Java anziché scrivere query SQL manualmente.

Postman

Postman è un'applicazione di sviluppo di API (Application Programming Interface) che consente agli sviluppatori di creare, testare, documentare e monitorare le API. Nel corso del progetto è stato uno strumento altamente utilizzato sia come API testing tool.

JUnit

JUnit è un framework di testing per Java utilizzato per la scrittura e l'esecuzione di test d'unità. Questo framework fornisce un ambiente di testing in cui è possibile definire e strutturare test.

Questo framework offre funzionalità come annotazioni e vari metodi che semplificano il processo di scrittura di questi test, automatizzando la verifica che il codice soddisfi i requisiti e produca risultati attesi.

Mockito

Mockito è un framework di testing per Java che si concentra sulla creazione di oggetti simulati (mock) per testare unità di codice. Gli oggetti mock imitano il comportamento di oggetti reali, consentendo ai test di focalizzarsi su parti specifiche del codice, permettendo ai test di non dover interagire con database o sistemi esterni. Permette inoltre di verificare interazioni con i mock e molto altro, al fine di agevolare il processo di testing.

3.3.2 Strumenti di supporto

Gli strumenti di supporto sono utilizzati per attività che sostengono lo sviluppo del progetto. Essi contribuiscono a migliorare la gestione, la qualità e l'efficienza del processo di sviluppo.

Microsoft Teams

Microsoft Teams è una piattaforma di comunicazione e collaborazione aziendale sviluppata da Microsoft. Offre strumenti per la chat, le videoconferenze, la condivisione di documenti e la gestione dei progetti, consentendo ai team di lavorare insieme in modo efficace sia in ufficio che a distanza.

Questa piattaforma è stata utilizzata nel corso del progetto per poter comunicare con il tutor anche quando non era in ufficio o con altri colleghi per determinate situazioni lavorative.

Notion

Notion è un'applicazione di gestione delle informazioni, utilizzata per prendere appunti, creare elenchi di attività, scrivere documenti e molto altro grazie alla sua interfaccia flessibile personalizzabile dagli utenti.

Questa applicazione è stata utilizzata nel corso della mia attività di Stage per prendere appunti o tenere traccia delle tasks che dovevo svolgere.

Microsoft Excel

Microsoft Excel è un'applicazione software di fogli di calcolo sviluppata da Microsoft. È utilizzata per creare, organizzare e analizzare dati in forma di tavole e grafici, offrendo inoltre molte funzionalità per eseguire calcoli.

Questa applicazione è stata utilizzata nel progetto allo scopo di velocizzare la creazione

di dati fittizi per popolare le tabelle del database per poter testare quanto prodotto.

Visual Studio Code

Visual Studio Code è un editor di codice sorgente sviluppato da Microsoft. È progettato per fornire un ambiente di sviluppo leggero, flessibile e personalizzabile per programmatori e sviluppatori.

È stato utilizzato nel progetto scaricando varie estensioni per visualizzare l'API e l'UML del database.

Docker

Docker è una piattaforma di containerizzazione_g che consente di creare, distribuire e gestire container_g virtualizzati_g. Permette di far funzionare le applicazioni in altri ambienti con facilità.

Nel progetto è stato utilizzato per creare un server locale con un'immagine del database MSSQL.

Swagger

Swagger è un insieme di strumenti e specifiche che consentono la documentazione, la progettazione e il test di API. Permette una migliore comunicazione dei propri endpoint dell'API semplificandone la lettura e la comprensione.

Git

Git è un sistema di controllo versione distribuito utilizzato nello sviluppo software. Consente di tenere traccia delle modifiche apportate al codice sorgente e di semplificare la collaborazione tra sviluppatori in progetti di programmazione.

GitLab

GitLab è una piattaforma di sviluppo software basata su web che offre una serie di strumenti per la gestione del ciclo di vita dello sviluppo delle applicazioni. Le principali funzionalità sono: la gestione dei repository Git, la collaborazione tra i membri del team e il monitoraggio delle issue.

All'interno del progetto è stata utilizzata come repository per tenere salvato il progresso del progetto. Nella mia attività di stage ho lavorato su un branch a me dedicatomi.

Git Extensions

Git Extensions è un'interfaccia grafica per il sistema di controllo versione distribuito Git. Questo software fornisce una modalità visuale per interagire con i repository Git.

Capitolo 4

Progettazione

4.1 Scopo del capitolo

In questo capitolo si tratterà dei giorni successivi all'apprendimento del background tecnologico, in cui sono state improntate le basi dell'architettura del progetto, definendo lo schema del database su cui si poggia l'API.

Nei seguenti paragrafi verrà quindi trattata l'architettura e le configurazioni utilizzate che funzionano da base per l'implementazione di quanto sviluppato.

4.2 Architettura REST

L'architettura REST, acronimo di "Representational State Transfer", è un approccio di progettazione per la creazione di servizi web che si basa sui principi dell'HTTP (Hypertext Transfer Protocol).

Le principali caratteristiche di un'architettura REST includono:

- **Sistema client-server**, dove il client è chi fa le richieste e il server fornisce le risposte;
- **Sistema layered**, perchè possono essere composte da più livelli di servizi indipendenti;
- **Stateless**, significa che il server non contiene client state, quindi ogni richiesta ha abbastanza informazione per il server per processarla;
- **Cacheable**, significa che l'architettura può memorizzare le risposte dei server e riutilizzarle;
- **Resource-based**: questo approccio è considerato tale, dato che si concentra sull'identificazione e la gestione delle risorse all'interno di un sistema. Le risorse vengono identificate da degli URI ed esse possono essere create, aggiornate, richieste o eliminate (operazioni CRUD) attraverso operazioni HTTP standard (POST, PUT, GET, DELETE);
- **Manipolazione delle risorse**, poichè le risorse sono diverse dalla loro rappresentazione logica, utilizzando formati come JSON o XML.

Questo stile architetturale è utilizzato soprattutto per la realizzazione di API poichè l'adozione dei principi standard di HTTP mantiene un'interfaccia uniforme e l'approccio resource-based ne semplifica la progettazione, dato che le risorse vengono identificate da URI_g.

4.2.1 Convenzioni di denominazione REST

Buona prassi nella creazione di un'API REST è il rispetto di convenzioni per la nomenclatura degli endpoint_g. Sebbene non esista un'unica convenzione obbligatoria, esistono delle best practices per garantire una facile lettura e comprensibilità per agevolare gli sviluppatori che adoperano l'API.

Le seguenti convenzioni sono state rispettate nella progettazione dell'API:

- pluralizzare le risorse, per distinguere se si fa riferimento ad una lista di una determinata risorsa o ad una singola risorsa;
- usare lettere minuscole;
- non usare estensioni dei file;
- in caso di nomi composti utilizzare il "-";
- non usare underscore;
- non utilizzare abbreviazioni o slang;
- non utilizzare verbi, poichè l'azione dovrebbe essere indicata dal metodo HTTP utilizzato.

4.3 Spring MVC

Nello sviluppo di una API REST tramite Spring Boot è comune l'utilizzo del pattern architetturale Model-View-Controller(MVC).

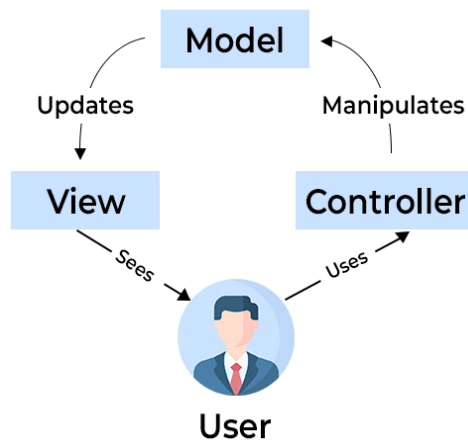


Figura 4.1: Schema Model-View-Controller

4.3.1 Model

Il Modello o Model in inglese, rappresenta i dati e i metodi per accedere ai dati dell'applicazione. Se esso viene aggiornato in seguito ad azioni o eventi, notificherà la View e il Controller del cambiamento.

Nel contesto dello sviluppo di un'API REST, il Model contiene dati che vengono elaborati e restituiti dall'API.

All'interno del progetto il Model è formato dai seguenti elementi.

Entità

Le entità sono risorse rappresentate con classi Java segnate con l'annotazione `@Entity`. Utilizzando questa annotazione si identifica che quella classe è mappata a una tabella del database. Esse dichiarano al loro interno, tramite annotazioni JPA apposite, gli attributi corrispondenti alla tabella a cui fa riferimento e, altre annotazioni per mantenere le relazioni presenti nel database tra le tabelle.

Repository

Le repository implementate nel progetto gestiscono l'accesso ai dati e definiscono metodi per eseguire operazioni di base sui dati, come l'inserimento, la modifica, la cancellazione e la ricerca. Tutto questo estendendo interfacce JPA, che consentono di eseguire operazioni in una base di dati senza scrivere codice SQL o query.

DTO

Data Transfer Object (DTO), oggetti utilizzati per modellare le rappresentazioni dei dati, utilizzati per definire sia i dati inviati dal client nelle requests che quelli che inviati dal server al cliente nelle responses.

4.3.2 View

La Vista o View in inglese, è responsabile di mostrare i dati provenienti dal Model e dell'interazione con l'utente. Essa cattura gli input dell'utente e delega al Controller l'elaborazione.

Dato che il progetto si concentrava sul lato back-end, precisamente sullo sviluppo dell'API REST, non ho quindi sviluppato una vera e propria View, poichè l'output è in formato JSON. In questo contesto l'output prodotto contenente i dati presentati al client in formato JSON, potrebbe essere considerata come "View".

4.3.3 Controller

Il Controller gestisce il flusso dell'applicazione. Esso riceve i comandi dell'utente attraverso la View e reagisce eseguendo operazioni conseguenti. Queste sue operazioni possono o meno coinvolgere il Model, ma portano generalmente sempre ad un cambiamento di stato della View.

All'interno del meccanismo di Spring MVC il Controller gestisce le richieste HTTP in arrivo, selezionando il Controller adeguato e assegnato a quell'endpoint API. Sono quindi responsabili di ricevere le richieste, elaborarle e restituire le risposte corrispondenti.

Per identificare il metodo associato ad una determinata operazione HTTP, vengono utilizzate le seguenti annotazioni *@PostMapping*, *@GetMapping*, *@PutMapping*, *@PatchMapping* e *@DeleteMapping*. All'interno del progetto il Controller è formato dai seguenti elementi.

Controller

I Controller contengono i metodi a cui vengono associati i percorsi delle richieste HTTP. Esso interpreta i parametri che possono provenire dall'URL o dal corpo della richiesta (parametri di query o dati JSON). Internamente a questi metodi viene richiamato il Service appropriato per eseguire operazioni di business logic al risultato finale da restituire.

Per garantire che i dati inseriti dagli utenti o provenienti da richieste siano conformi alle aspettative e non causino errori vengono inseriti dei controlli di validazioni all'interno dei metodi del Controller.

Service

I Service eseguono operazioni complesse ed elaborazioni di dati, gestendo la business logic dell'applicazione. Essi interagiscono con i Repository per recuperare dati dal database e vengono richiamati all'interno dei metodi del Controller.

Exception Handler

Per centralizzare la gestione delle eccezioni è stato creato un handler che garantisce uniformità alle eccezioni che l'applicazione può produrre.

4.4 Pattern Client-Server

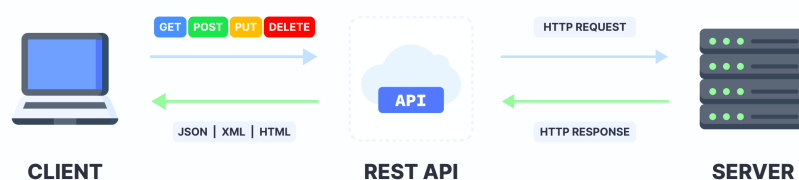


Figura 4.2: Client-Server in una REST API

È un pattern architetturale composto da due componenti: client e server. Il client rappresenta il richiedente del servizio, mentre il server fornisce i contenuti e i servizi ai client, restando in attesa delle loro richieste.

Il pattern client-server si adatta perfettamente ad una API REST e le sue componenti sono identificate in questo modo.

Client

In questo contesto il client è rappresentato da applicazioni o servizi che comunicano con l'API REST, inviando le richieste HTTP al server per ottenere quanto richiesto. Il client invia richieste utilizzando verbi standard HTTP per accedere alle risorse e alle funzionalità che l'API offre.

Server

Il server invece è colui che gestisce le richieste HTTP ricevute e genera le risposte in formato JSON. Esso quindi elabora le richieste ed interagisce col database per recuperare le informazioni.

4.5 Design Pattern

Nel prodotto sviluppato possiamo notare i seguenti Design Pattern. Ad eccezione dell'ultimo Design Pattern adottato, in riferimento all'elenco sottostante, i restanti Design Pattern sono già implementati da Spring e Spring Boot.

Repository pattern

Repository è un pattern ideato per dividere le operazioni di business logic da quelle di persistenza dei dati. Questo pattern fornisce astrazione ai dati nascondendo i dettagli di accesso, facilita il testing, supporta diverse sorgenti di dati e mantiene un codice riutilizzabile in quanto non sarà necessario modificare ampiamente il codice in caso di modifiche alla sorgente dati.

In Spring Boot vengono implementate interfacce annotandole con l'annotazione *@Repository* ed estendendole con interfacce JPA che permettono l'utilizzo di metodi che forniscono query basilari o la possibilità di creare metodi custom per query personalizzate.

Dependency Injection

Dependency Injection è un pattern che inietta una dipendenza in una classe senza sapere l'implementazione effettiva. Questo può avvenire attraverso constructor injection, setter injection o method injection. Gli obiettivi di questo pattern sono: eliminare il forte accoppiamento tra le classi, rendere il codice più manutenibile e più facile da testare. In Spring Boot risulta evidente tramite l'utilizzo di annotazioni come *@Autowired*, iniettando direttamente le dipendenze necessarie, promuovendo il concetto successivo di Inversion of Control;

Inversion of Control

Inversion of Control (IoC) è un design pattern nato sul concetto di invertire il controllo del flusso del sistema nella gestione delle dipendenze e del controllo interno di un'applicazione. Normalmente è l'applicazione che controlla le dipendenze o il flusso di esecuzione, ma questa responsabilità, utilizzando questo pattern, viene trasferita ad un framework. Il framework instanzia oggetti, inietta le dipendenze e coordinerà il flusso di esecuzione.

Spring Boot è costruito proprio su questo concetto chiave. Esso è correlato a IoC per i seguenti motivi:

- Component scan, esegue automaticamente uno scan delle classi individuando quelle contrassegnate con annotazioni come *@Service*, *@Repository*, *@Controller* e altre;
- Semplifica la gestione delle dipendenze, fornendo degli insiemi di dipendenze utili per determinate tecnologie sotto il nome di "starters", velocizzando l'inizializzazione delle dipendenze senza doverle configurare manualmente, fornendo un senso di controllo sulla configurazione iniziale;
- Dependency Injection;
- Application Context, agisce come contenitore per i bean_g dell'applicazione. Esso gestisce i loro cicli di vita e inietta le dipendenze nei punti appropriati.

Front Controller

Il Front Controller è un design pattern che permette di centralizzare la gestione delle richieste all'interno di un'applicazione. In Spring MVC è implementato dal framework per gestire le richieste HTTP verso il Controller adeguato.

Data Transfer Object

Data Transfer Object (DTO) è un pattern utilizzato per gestire il trasferimento dei dati tra client e server. Gli obiettivi principali del pattern sono:

- Sicurezza, in quanto puoi controllare cosa si invia;
- Flessibilità, puoi adattare i DTO in base alle esigenze dell'API;
- Separano la rappresentazione interna da quella esterna dei dati.

Nel contesto di una API REST, i DTO vengono utilizzati sia nella Request che nella Response. Vengono utilizzati in entrambi i punti perchè quando il client invia dei dati magari non necessita degli oggetti completi ma solo di alcune informazioni. Invece quando il server restituisce dei dati il client se li può aspettare sotto una determinata struttura.

4.6 Progettazione del database

4.6.1 Configurazione di Docker

I container Docker offrono un'isolamento completo dell'ambiente, che aiuta a evitare conflitti di dipendenze e interferenze con altre applicazioni o servizi che potrebbero essere presenti sul sistema. Per questo motivo si è deciso di utilizzare l'approccio di sandboxing_g che offre Docker, poichè l'utilizzo di container consente di isolare le applicazioni e i servizi in ambienti virtualizzati, condividendo il kernel del sistema operativo host ma separando le loro risorse e i loro processi.

Dato che le tabelle di mia creazione dovevano integrarsi con il database aziendale, tramite il tool Docker Compose_g e un file di configurazione *docker-compose.yaml*, è

stato possibile avviare un nuovo container contenente un'immagine di Microsoft SQL Server, che forniva un backup del database aziendale con dati e tabelle.

4.6.2 Modello dati

IMMAGINE UML SENZA ATTRIBUTI

Nel seguente disegno si può notare il database su cui poggia l'API. È stata inserita una nomenclatura «esterna» per indicare le tabelle provenienti dal database aziendale.

Di seguito elenco le tabelle da me create, spiegandone l'utilizzo e gli attributi:

TUTTE LE TABELLE MIE

4.7 Configurazione iniziale del progetto

4.7.1 Spring Initializr

The screenshot shows the Spring Initializr web interface. It is divided into several sections for configuring a new Spring Boot project:

- Project:** Options for **Gradle - Groovy**, **Gradle - Kotlin**, and **Maven** (selected).
- Language:** Options for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Version selection including **3.2.0 (SNAPSHOT)**, **3.2.0 (M1)**, **3.1.3 (SNAPSHOT)**, **3.1.2** (selected), **3.0.10 (SNAPSHOT)**, **3.0.9**, **2.7.15 (SNAPSHOT)**, and **2.7.14**.
- Project Metadata:** Fields for **Group** (com.example), **Artifact** (demo), **Name** (demo), **Description** (Demo project for Spring Boot), and **Package name** (com.example.demo).
- Packaging:** Options for **Jar** (selected) and **War**.
- Java:** Version selection including **20**, **17** (selected), **11**, and **8**.
- Dependencies:** A section with a button **ADD DEPENDENCIES... CTRL + B** and three pre-selected dependencies:
 - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Spring Boot DevTools** (DEVELOPER TOOLS): Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
 - Spring Data JPA** (SQL): Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

At the bottom, there are buttons for **GENERATE CTRL + G**, **EXPLORE CTRL + SPACE**, and **SHARE...**

Figura 4.3: Interfaccia Spring Initializr

Spring Initializr è uno strumento online fornito dalla community di Spring Framework, che consente di creare rapidamente un progetto Spring Boot personalizzato, con le dipendenze e le configurazioni preselezionate dall'utente. Questo strumento semplifica notevolmente il processo di inizializzazione di un progetto Spring Boot, permettendo agli sviluppatori di risparmiare tempo e concentrarsi sulla scrittura del codice.

L'utente può selezionare il tipo di progetto di cui ha bisogno, come ad esempio un progetto Maven o Gradle, e specificare il linguaggio e la versione di Spring Boot desiderati. Inoltre, può anche inserire i metadati del progetto, come il nome del progetto e il nome dei packages.

Una volta selezionate le opzioni desiderate, l'utente può scegliere le dipendenze per il progetto. Le dipendenze sono librerie di terze parti che forniscono funzionalità aggiuntive al progetto.

Dopo aver selezionato le dipendenze, l'utente può scaricare il progetto Spring Boot personalizzato in formato ZIP.

4.7.2 Contenuto del pacchetto

Il file ZIP scaricato precedentemente contiene tutti i file necessari per iniziare a lavorare sul progetto. All'interno di questo pacchetto troviamo i seguenti elementi rilevanti:

- file di configurazione *application.properties* e file di build *pom.xml* con le dipendenze selezionate;
- classe principale, rappresenta il punto di ingresso dell'applicazione ed è annotata con *@SpringBootApplication*. Il metodo *main()* al suo interno avvierà l'applicazione Spring;
- una struttura base dei package.

4.7.2.1 Pom.xml

Di seguito riporto uno snippet_g del file di build *pom.xml* che si ottiene creando un progetto con le dipendenze sopra selezionate. La configurazione di Maven avviene proprio tramite questo file.

All'interno di questo file si possono trovare i seguenti tag:

- *<dependencies>*, indica una lista di dipendenze;
- *<build>*, contiene impostazioni di costruzione e compilazione;
- *<plugins>*, contiene plugin di Maven;
- *<properties>*, contiene proprietà definite dall'utente;
- *<groupId>*, organizzazione che ha creato il progetto;
- *<artifactId>*, nome unico del progetto;
- *<version>*, versione del progetto.

Nel file si possono notare le seguenti dipendenze:

- *spring-boot-starter-web*, per utilizzare il framework Spring MVC per la creazione di applicazioni Web;
- *spring-boot-starter-jpa*, per la persistenza dei dati;
- *spring-boot-devtools*, offre tools per migliorare il processo di sviluppo come live reload e rilascio automatico;
- *spring-boot-starter-test*, per includere librerie di testing.

```
<properties>
  <java.version>17</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Figura 4.4: Snippet pom.xml generato

Rispetto al file utilizzato nel progetto mancano le seguenti dipendenze:

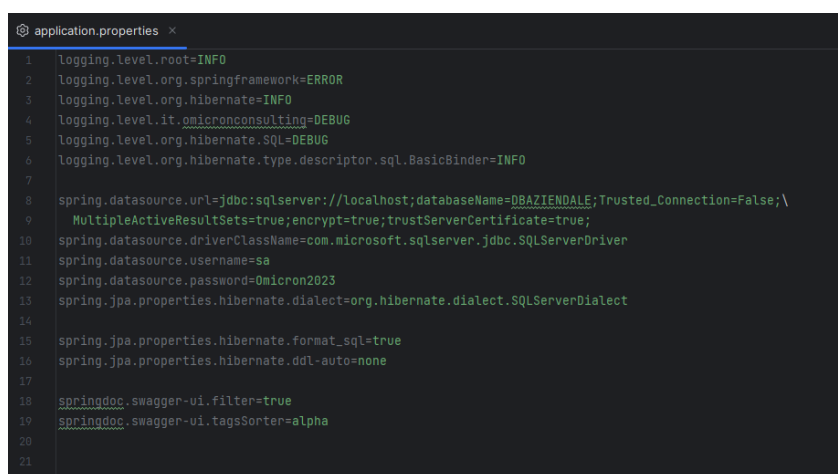
- *spring-boot-starter-jdbc*, per avviare Spring con il supporto JDBC nel progetto;
- *mssql-jdbc*, contenente il driver JDBC di Microsoft SQL Server;
- *springdoc-openapi-starter-webmvc-ui*, dipendenza per l'integrazione di Springdoc OpenAPI con Spring Boot per generare la documentazione dell'API utilizzando l'interfaccia utente WebMvc UI;
- *poi* e *poi-ooxml*, due dipendenze relative ad Apache POI che offre funzionalità per poter lavorare con documenti Microsoft Office;
- *jxls-jexcel*, dipendenza che include la libreria jXLS per creare e manipolare documenti Excel;
- *fastexcel* e *fastexcel-reader*, altre dipendenze inerenti a file Excel che offrono funzionalità per lettura, scrittura e modifica di questi file.

Per il testing sono state aggiunte:

- *junit*, framework usato in Java per scrivere test unitari;
- *hamcrest-library*, libreria utile a scrivere asserzioni più espressive e comprensibili nei test unitari;
- *h2*, si tratta della dipendenza per la libreria H2 Database Engine, che è un database SQL. Utilizzato con scope "test" per crearne un'istanza temporanea e utilizzarlo nei test.

4.7.2.2 Application.properties

Il secondo file qui sotto rappresentato è il file configurazione *application.properties*. Di seguito riporto il file che ho utilizzato nel progetto:



```
1 logging.level.root=INFO
2 logging.level.org.springframework=ERROR
3 logging.level.org.hibernate=INFO
4 logging.level.it.micronconsulting=DEBUG
5 logging.level.org.hibernate.SQL=DEBUG
6 logging.level.org.hibernate.type.descriptor.sql.BasicBinder=INFO
7
8 spring.datasource.url=jdbc:sqlserver://localhost;databaseName=DBAZIENDALE;Trusted_Connection=False;\
9     MultipleActiveResultSets=true;encrypt=true;trustServerCertificate=true;
10 spring.datasource.driverClassName=com.microsoft.sqlserver.jdbc.SQLServerDriver
11 spring.datasource.username=sa
12 spring.datasource.password=0micron2023
13 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.SQLServerDialect
14
15 spring.jpa.properties.hibernate.format_sql=true
16 spring.jpa.properties.hibernate.ddl-auto=none
17
18 springdoc.swagger-ui.filter=true
19 springdoc.swagger-ui.tagsSorter=alpha
20
21
```

Figura 4.5: Foto application.properties configurato e utilizzato

Possiamo notare come il file utilizzi un formato di configurazione basato su chiavi e valori. Le chiavi rappresentano le diverse proprietà di configurazione dell'applicazione, mentre i valori rappresentano le impostazioni specifiche.

Nel file possiamo notare le seguenti chiavi:

- *logging.level*, servono per configurare il livello di dettaglio dei log per diverse classi o package all'interno dell'applicazione;
- *spring.datasource*, servono per collegarsi ad un database, in questo caso locale, inserendo username e password e driver di MSSQL;
- *spring.jpa.properties.hibernate*, servono per configurare le impostazioni di Hibernate, impostando la validazione schema-entità, il dialetto del server SQL e il suo livello di log;
- *springdoc.swagger-ui*, libreria che fornisce integrazione tra Spring Boot e Swagger UI.

Capitolo 5

Codifica

5.1 Scopo del capitolo

Nel seguente capitolo si può osservare il lavoro dei giorni seguenti alla Progettazione fino alla fine del periodo di tirocinio. In questo capitolo viene illustrato lo sviluppo dell'API REST, come è stato organizzato il codice parlando di classi e package, la codifica dei moduli più rilevanti ed infine la validazione.

5.2 Endpoint sviluppati dell'API REST

In questa sezione si trovano le descrizioni di tutti gli endpoint implementati, suddivisi in base all'ambito di interesse del servizio. Sarà fornito anche il verbo HTTP e il percorso necessario per effettuare ciascuna richiesta.

I verbi standard forniti da HTTP utilizzati applicati all'API REST sono i seguenti:

Verbo	Descrizione
POST	Utilizzato per inviare dati al server al fine di creare una nuova risorsa ed aggiungerla all'insieme corrente
GET	Utilizzato per richiedere dati al server in merito ad una o più risorse senza modificarle
DELETE	Utilizzato per eliminare una risorsa specifica dal server
PUT	Utilizzato per aggiornare totalmente una risorsa e viene utilizzato quando si vuole sostituire completamente una risorsa

PATCH	Utilizzato per effettuare aggiornamenti parziali a una risorsa esistente
-------	--

Tabella 5.1: Verbi Standard HTTP utilizzati

Alla fine del progetto questi sono gli endpoint che sono stati sviluppati:

GLI ENDPOINT IN UNA TABELLA CON VERBO + PATH + DESCRIZIONE

5.3 Organizzazione del codice

5.3.1 Common e config

I package contengono i seguenti file .java:

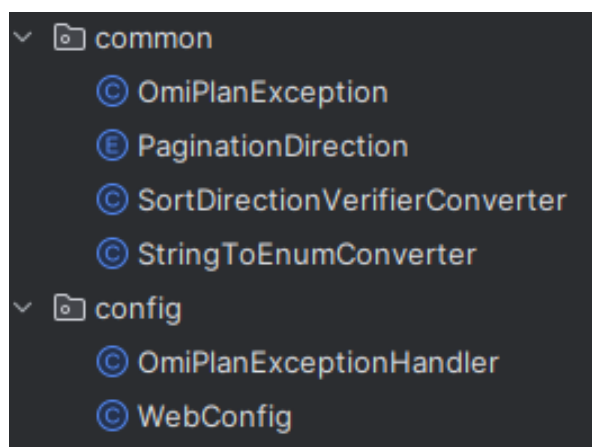


Figura 5.1: Package common e config

Common

- **OmiPlanException**, eccezione custom utilizzata per gestire le *RuntimeException*. Classe formata da un *HttpStatus* per mostrare lo stato di risposta HTTP e dal messaggio fornito al lancio dell'eccezione;
- **PaginationDirection**, classe enum per gestire la direzione della paginazione (ASC o DESC);
- **SortDirectionVerifierConverter**, classe che implementa l'interfaccia *Converter<S,T>* (componente Java utilizzato quando si lavora con strutture dati o oggetti che devono essere trasformati o adattati in tipi diversi) per verificare che la direzione inserita sia ASC o DESC e gestire la *RunTimeException* in caso non sia una variabile enum;

- **StringToEnumConverter**, classe che implementa l'interfaccia *Converter<S,T>* per convertire la direzione Stringa inserita nell'enum della direzione di pagina.

Config

- **OmiPlanExceptionHandler**, handler_g con il compito di gestire le eccezioni runtime e le altre segnandole come "Unexpected error". Questa classe è annotata con *@ControllerAdvice*, poiché definisce una classe che gestisce in maniera centralizzata le eccezioni. Contiene due metodi annotati con *@ExceptionHandler* che gestiscono rispettivamente le *RunTimeException* e le altre eccezioni generali *Exception*;
- **WebConfig**, classe annotata con l'annotazione *@Configuration* indicando che è una classe di configurazione e che contiene definizioni di bean o altre configurazioni necessarie per l'applicazione. Essa infatti estende *WebMvcConfigurer* che consente di modificare le configurazioni predefinite di Spring MVC, in questo caso aggiungendo i converter sopra citati.

5.3.2 Entities

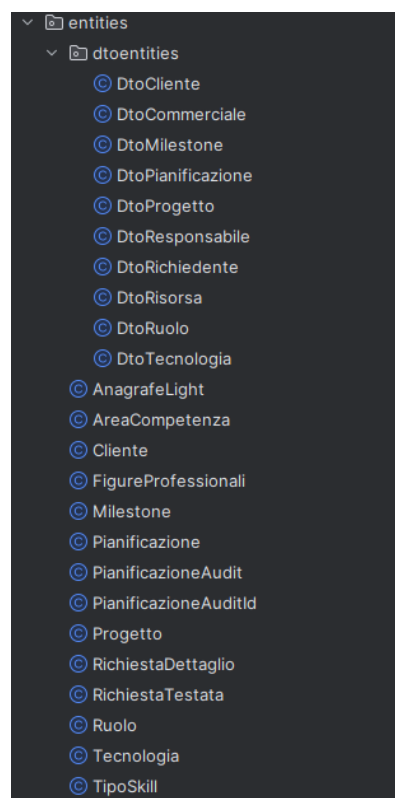


Figura 5.2: Package entità

All'interno del package Entities troviamo le entità. Per entità si intendono tutte quelle classi Java che definiscono i modelli di dati dell'applicazione.

Queste classi vengono annotate con le seguenti annotazioni JPA utili a stabilire come la classe venga associata a una tabella presente nel database relazionale. Ogni istanza di un'entità rappresenta una riga nella tabella relativa.

Di tutte le tabelle da me create è stata mappata ogni relazione tra tabelle e campo, mentre per ogni tabella del database aziendale sono stati mappati tutti i campi e le relazioni ad altre tabelle che potevano tornarmi utili.

In ogni entità troviamo setters e getters, costruttori con e senza argomenti.

5.3.2.1 DTO Entities

All'interno del package "dtoentities" troviamo tutte quelle classi DTO delle entità di cui servivano soltanto determinate informazioni da restituire all'utente, evitando così ridondanza o informazioni superflue. Queste classi non contengono annotazioni, ma sono semplicemente degli oggetti contenenti i campi semplificati delle entità. A loro volta possono contenere altri DTO di altre entità in base alle relazioni che hanno.

```
public class DtoPianificazione {
    5 usages
    private String id;
    5 usages
    private String ambito;
    5 usages
    private String note;
    5 usages
    private String delivery;
    5 usages
    private DtoRisorsa risorsa;
    5 usages
    private DtoResponsabile responsabile;
    5 usages
    private Date dataInizio;
    5 usages
    private Date dataFine;
    5 usages
    private Integer giorni;
    5 usages
    private Integer percentuale;
}

public DtoPianificazione(Pianificazione p){
    this.id=p.getId();
    this.ambito=p.getAmbito();
    this.note=p.getNote();
    this.delivery=p.getDelivery();
    this.dataInizio =p.getDataInizio();
    this.dataFine =p.getDataFine();
    this.giorni=p.getGiorni();
    this.percentuale=p.getPercentuale();
    this.reperibilita=p.isReperibilita();

    if(p.getRuolo() != null){
        this.ruolo=new DtoRuolo(p.getRuolo());
    }

    this.festivi=p.isFestivi();
    this.esperienza=p.getEsperienza();
    this.status=p.getStatus();

    this.risorsa =new DtoRisorsa(p.getRisorsa());
    this.responsabile =new DtoResponsabile(p.getResponsabile());

    this.dataInserimento = p.getDataInserimento();
    this.dataModifica = p.getDataModifica();

    this.utenteInserimento = p.getUtenteInserimento();
    this.utenteModifica = p.getUtenteModifica();
}
```

Figura 5.3: Esempio di classe DTO di Pianificazione

Ogni entità DTO possiede tre costruttori: costruttore senza argomenti e con argomenti e infine un costruttore che velocizzava la conversione da entità ad entità DTO.

5.3.2.2 Esempio di un'entità



```
11
12 @Entity
13 @Table(name = "sched_pianificazione")
14 public class Pianificazione {
15
16     4 usages
17     @Id
18     @Column(name = "id", columnDefinition = "VARCHAR(36)")
19     private String id;
20
21     4 usages
22     @Column(name = "ambito")
23     private String ambito;
24
25     4 usages
26     @Column(name = "note")
27     private String note;
28
29     4 usages
30     @Column(name = "delivery")
31     private String delivery;
32
33     4 usages
34     @Column(name = "data_inizio")
35     @Temporal(TemporalType.DATE)
36     private Date dataInizio;
```

Figura 5.4: Esempio di mappatura di un'entità

Nell'immagine qui sopra possiamo notare uno snippet dell'entità Pianificazione in cui sono state utilizzate le seguenti annotazioni:

- *@Entity*, per mappare le classi Java che rappresentano una tabella in un database si inserisce questa notazione specificando a class level_g.
- *@Table*, nella maggior parte dei casi il nome di una tabella nel database e il nome dell'entità non sono gli stessi. Per questo motivo è stata utilizzata la seguente annotazione per specificare il nome della tabella;
- *@Column*, utilizzata per mappare una campo di una classe a una colonna di una tabella nel database. Questa annotazione possiede parametri come il "nome" per specificare il nome della colonna a cui è associato il campo;
- *@Id*, per identificare un campo all'interno di una classe come chiave primaria in una tabella del database viene utilizzata questa annotazione;
- *@Temporal*, utilizzata per specificare se un campo di tipo Date dovrebbe essere mappato come *TemporalType.DATE*, per una data senza orario, *TemporalType.TIME* per un orario senza data e infine *TemporalType.TIMESTAMP*, utilizzato nella tabella di log di Pianificazione per mappare un campo con data e orario.

```

4 usages
@ManyToOne
@JoinColumn(name = "progetto_id")
private Progetto progetto;

4 usages
@OneToMany(mappedBy = "richiestaTestata", orphanRemoval = true)
private List<RichiestaDettaglio> dettaglioList;

3 usages
@ManyToOne
@JoinColumn(name = "richiedente_id")
private AnagrafeLight richiedente;

3 usages
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(name = "sched_richiesta_testata_tipo_skill",
    joinColumns = @JoinColumn(name = "sched_richiesta_testata_id"),
    inverseJoinColumns = @JoinColumn(name = "tipo_skill_id"))
private List<TipoSkill> skills;

```

Figura 5.5: Esempio di mappatura delle relazioni della tabella RichiestaTestata

Per mantenere le relazioni tra le tabelle sono state utilizzate le annotazioni come nello snippet qui sopra raffigurante una porzione della classe Java che mappa la tabella RichiestaTestata.

In casi in cui entrambi i lati della relazione necessitavano per motivi implementativi, come la visualizzazione da entrambe le parti dell'informazione, di mappare la relazione opposta, veniva dichiarata una relazione bidirezionale, in cui entrambi i lati mappavano la relazione e diventava importante gestire correttamente la sincronizzazione tra le due entità.

- **uno-a-molti**, relazione in cui il campo associato sarà una lista di oggetti della classe opposta (in relazione all'esempio, una RichiestaTestata è associata a più RichiesteDettaglio), mentre nella relazione molti-a-uno sarà un oggetto singolo della classe opposta annotato con `@JoinColumn` con `name` uguale a quello del campo nella tabella del database (in relazione all'esempio, una RichiestaTestata ha un solo Richiedente).

Nell'esempio è stato utilizzato il parametro `mappedBy` per mappare la relazione opposta e `orphanRemoval` per poter implementare la cancellazione di una RichiestaTestata garantendo l'eliminazione di tutte le RichiesteDettaglio associate al momento della rimozione.

- **uno-a-uno**, relazione in cui un record della tabella è associato ad un solo record dell'altra tabella. Anche in questo caso se si vuole mappare da entrambi le parti la relazione bisogna utilizzare lo stesso principio dichiarato nella relazione `@OneToMany`, solo che da entrambi le parti avranno come campo un oggetto della classe opposta.

Non sono state identificate relazioni uno-a-uno nel database.

- **molti-a-molti**, relazione in cui viene mappata la tabella di join presente nel database tra le due tabelle, utilizzando il codice che si vede in esempio. Per gestire operazioni di rimozione tra le due tabelle è stato utilizzato un metodo

annotato con `@PreRemove` nella classe `TipoSkill`, che entra in azione quando una `RichiestaTestata` viene eliminata, assicurando che la disconnessione avvenga in modo appropriato.

5.3.3 Package dto

Per la composizione delle richieste che il client invia o delle risposte che il server manda, sono state create vari tipi di requests e responses.

Ogni oggetto di risposta è composto da un oggetto data e un oggetto metadata, mentre gli oggetti di richiesta possono variare. Se la richiesta è fatta per un endpoint che restituisce una lista di risultati, possiamo avere tre casistiche:

- `bodyg` formato da un oggetto data, contenente informazioni utili alla risposta, e metadata che contiene i dati di paginazione_g;
- nessun body, ma soltanto i dati di paginazione passati come parametri di query_g;
- `path variableg` che solitamente fa riferimento ad un ID, nei parametri di query e dati di paginazione.

Se invece la richiesta è costruita per un endpoint che restituisce un singolo oggetto nell'oggetto data, non viene utilizzato alcun body o parametri di query, ma solo una path variable.

Questa suddivisione data e metadata è un modello comune nella progettazione API per separare i dati effettivi o dati che forniscono informazioni per ottenere un tipo di risposta, dalle informazioni aggiuntive che descrivono il risultato o aggiungono caratteristiche alla richiesta.

Questo package è formato da vari subpackage_g.

5.3.3.1 Common

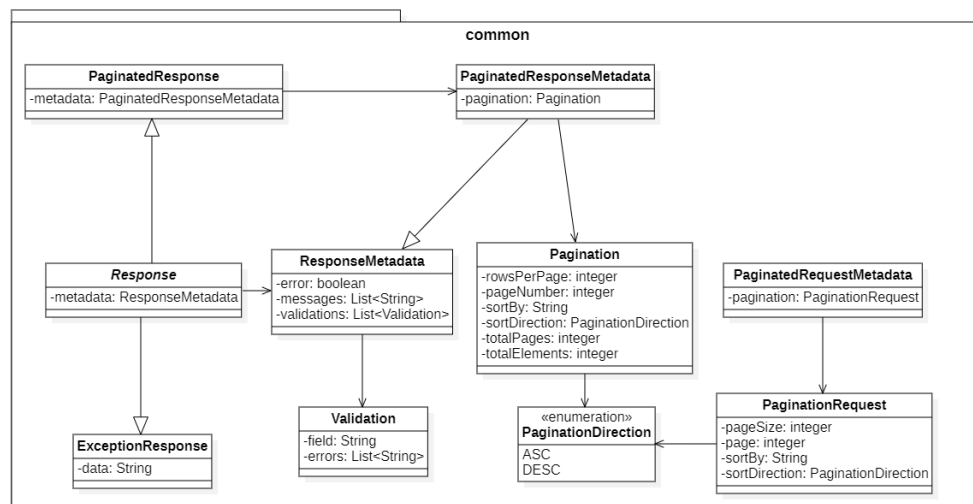


Figura 5.6: Subpackage common

Questo subpackage contiene la response di base, response paginata, la response utilizzata nell'eccezione personalizzata, una request per la paginazione e i vari metadata utilizzati

nelle responses. Ogni metadata contiene un boolean che indica se si è andati in errore (true) o meno (false), il messaggio di errore e una lista di **validation**, contenente informazioni di validazione dei dati.

È stata creata una classe **Pagination** per avere un controllo personalizzato sulla Paginazione.

5.3.3.2 Requests e Body

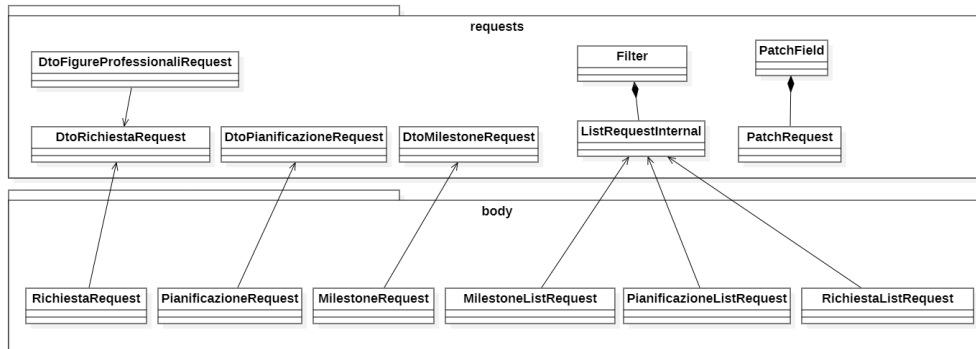


Figura 5.7: Subpackage requests e body

Questi due subpackages sono correlati tra di loro dato che ogni body ha come attributo una request presa dal subpackage requests. Esistono due tipi di body:

- ogni **Request** singola contiene una request DTO formata dai campi utili che l'utente deve inserire per eseguire un determinato endpoint;
- ogni **ListRequest** è formata da un oggetto data di tipo **ListRequestInternal**, che contiene tre parametri: filters, lista di oggetti **Filter** formati da campo-valore, una stringa q (quicksearch) utile nella query personalizzata per cercare in determinati campi quello che l'utente inserisce e infine andOperator per impostare i filtri in And o in Or nella query.

All'interno del subpackage requests troviamo anche **PatchRequest**, contenente un unico campo corrispondente ad una lista di **PatchField**. Questa richiesta viene utilizzata nella richieste PATCH, in cui è possibile inserire uno o più valori in base a cosa si va a modificare.

5.3.3.3 Responses

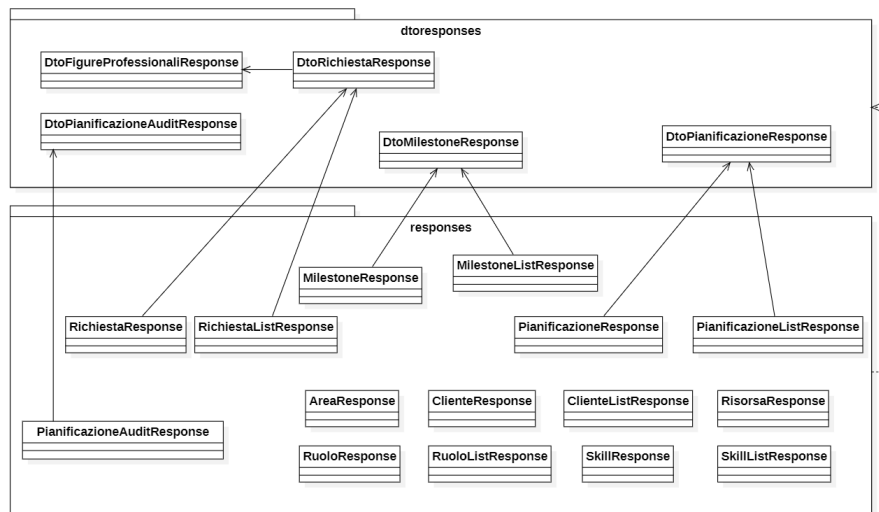


Figura 5.8: Subpackage responses e dtoreponses

Il package responses contiene tutte le response per entità. Ogni response eredita la classe astratta **response** se è una risposta singola, altrimenti eredita la classe **PaginatedResponse** se è una lista di risposte. Questo permette all'utente di visualizzare la risposta dopo aver interagito con un'endpoint, formata da data e metadata. Ogni response contiene un solo attributo che corrisponde all'oggetto data e può essere una responses DTO del subpackage dtoreponses o un'entità DTO del subpackage delle entità. Se si è di fronte ad una **ListResponse** l'attributo sarà una lista, altrimenti un oggetto singolo.

5.3.4 Repository-Service-Controller

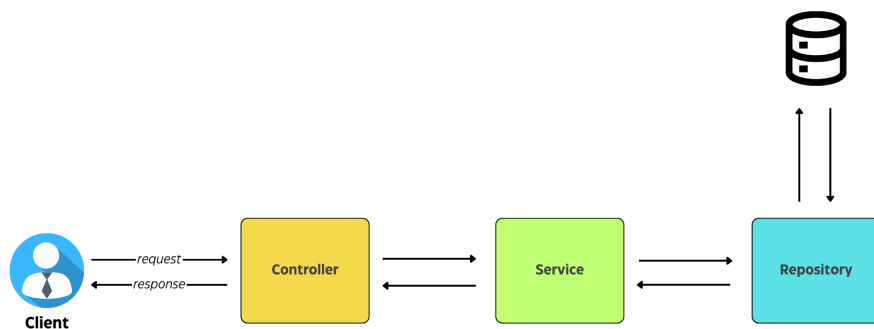


Figura 5.9: Schema di collegamento tra i componenti Controller, Service e Repository

In un'applicazione Spring Boot che implementa una REST API l'interazione tra questi tre componenti è fondamentale per gestire le richieste HTTP, elaborare la business

logic e interagire con il database.

Come descritto nell'immagine l'interazione tra i componenti segue questo flusso:

1. il client invia una richiesta HTTP e arriva al Controller tramite URL;
2. il controller estrae i dati della richiesta, effettua controlli di validazione e chiama il metodo del service appropriato;
3. il metodo del service invocato conterrà la business logic per eseguire la richiesta e ulteriori controlli di validazione;
4. la/le repository coinvolta/e ottengono i dati richiesti dal database che vengono restituiti al service che li rielaborerà, restituendo il risultato al controller;
5. il controller crea una risposta appropriata in formato JSON e la restituisce al client.

5.3.4.1 Controller

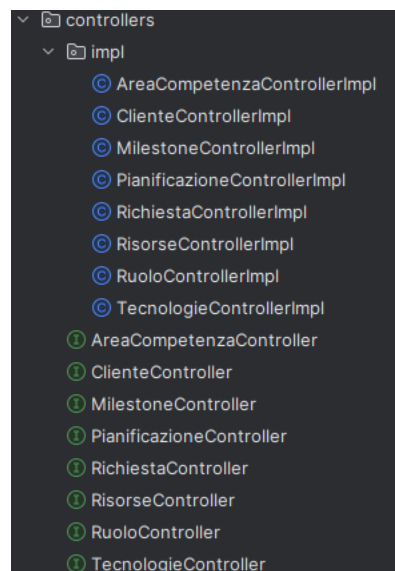


Figura 5.10: Controller sviluppati

Il Controller è punto di instradamento per le richieste HTTP. Riceve le richieste dal client, estrae i dati dai parametri della richiesta o dal corpo (body), esegue appropriati controlli di validazione ed infine inoltra le operazioni al Service. I controlli di validazione comprendono: validazione degli UUID, validazione del formato delle date inserite, validazione che un dato obbligatorio non sia nullo e controllo che i filtri inseriti non siano non valorizzati.

Ogni Controller è formato dalla sua interfaccia, contenente la firma dei metodi associati agli endpoint, e dalla sua implementazione. Ognuno di essi è annotato con *@RestController*.

```
@PostMapping("/milestones/list")
MilestoneListResponse getMoreMilestone(@RequestBody MilestoneListRequest request);
```

Figura 5.11: Firma di un metodo nell'interfaccia di un Controller

All'interno dell'interfaccia del Controller troviamo tutti i metodi associati agli endpoint. Per identificare a quale endpoint appartiene vengono utilizzate annotazioni come: *@PostMapping*, *@GetMapping*, *@DeleteMapping*, *@PutMapping* e *@PatchMapping*. Tramite l'annotazione *@RequestBody* Spring deserializza_g automaticamente il JSON in un tipo Java, in questo caso *MilestoneListRequest*. Come libreria standard utilizzata per mappare i campi JSON inseriti nella request dall'utente in un oggetto Java è la libreria Jackson_g.

```
if(dtoMilestoneRequest.getProgetto() != null) {
    try {
        UUID.fromString(dtoMilestoneRequest.getProgetto());
    } catch (IllegalArgumentException e) {
        throw new OmiPlanException(HttpStatus.BAD_REQUEST,
            String.format("Sintassi errata del Progetto Id: %s",
                dtoMilestoneRequest.getProgetto()));
    }
}

return schedMilestoneService.postMilestone(milestoneRequest);
```

Figura 5.12: Esempio di controllo di validazione

Nelle classi *ControllerImpl* troviamo l'implementazione dei metodi dell'interfaccia, controlli di validazione che possono lanciare eccezioni e il metodo del Service utilizzato. In questo code snippet vediamo il controllo di validazione su l'Id del Progetto associato ad una Milestone. Se passerà il controllo, allora verrà inoltrata la richiesta al Service.

5.3.4.2 Repository

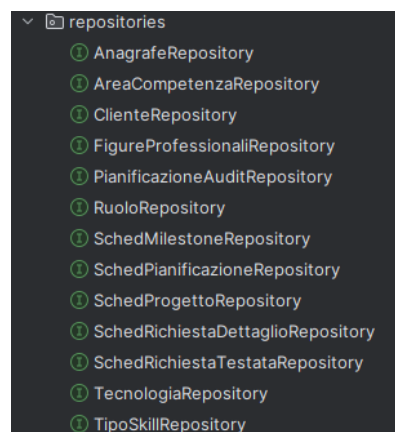


Figura 5.13: Repository sviluppati

Un Repository in Spring Boot fornisce un modo per interagire con una fonte di dati, in questo caso un database. Permette di eseguire operazioni CRUD tramite Spring Data JPA che genera automaticamente query SQL in base ai metodi che l'interfaccia del Repository contiene.

Le Repository da me create estendono, per poter eseguire le query, estendono l'interfaccia `JpaRepository` o `PagingAndSortingRepository` per risposte paginate. Ogni Repository deve essere annotato con `@Repository`.

```
@Repository
public interface SchedPianificazioneRepository extends JpaRepository<Pianificazione,String> {

    2 usages
    Optional<Pianificazione> findByRisorsaAndRuolo(AnagrafeLight risorsa,Ruolo ruolo);

}
```

Figura 5.14: Esempio di una Repository

È possibile creare delle query personalizzate SQL tramite parole chiave di Spring Data JPA. In questo caso nell'immagine la query corrisponderà alla ricerca della Pianificazione che ha la stessa Risorsa e lo stesso Ruolo.

5.3.4.3 Service

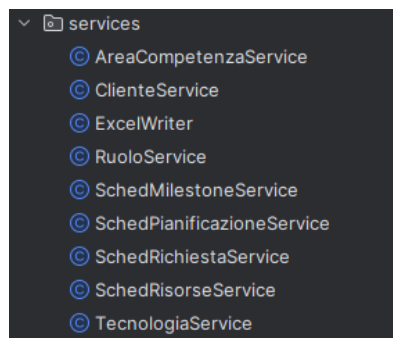


Figura 5.15: Service sviluppati

Il Service è un componente che contiene elaborazioni di dati, chiamate a metodi nei Repository per accedere al database e restituisce i risultati al Controller. Ogni Service è annotato con `@Service`.

```

public MilestoneResponse deleteMilestone(String id) {

    Optional<Milestone> milestoneOptional = milestoneRepository.findById(id);
    if (milestoneOptional.isEmpty()) {
        throw new OmiPlanException(HttpStatus.NOT_FOUND, String.format("L'id %s inserito non è nel DB!", id));
    }

    if (!milestoneOptional.get().getPianificazioneList().isEmpty()) {
        throw new OmiPlanException(HttpStatus.CONFLICT, String.format("La milestone %s è collegata ad una Pianificazione",
            milestoneOptional.get().getId()));
    }

    milestoneRepository.deleteById(id);

    ResponseMetadata responseMetadata = new ResponseMetadata();

    return new MilestoneResponse(responseMetadata, new DtoMilestoneResponse(milestoneOptional.get()));
}

```

Figura 5.16: Metodo deleteMilestone() nel Service

In questo code snippet possiamo osservare come funziona il metodo `deleteMilestone()` all'interno del `MilestoneService`. Viene controllato, tramite un metodo fornito dal repository della Milestone se l'ID è presente nel database e in caso positivo viene sollevata un'eccezione che interrompe la cancellazione della Milestone. Il controllo successivo verifica che la Milestone non sia collegata ad una Pianificazione ed in caso di esito positivo interrompere la cancellazione.

Se tutti questi controlli risultano negativi allora avverrà la cancellazione nel database, tramite il metodo fornito dal repository della Milestone, della Milestone associata a quell'ID. Viene infine generata la response da ritornare al Controller formata da un oggetto metadata vuoto, dato che non ci sono stati errori, e una nuova `MilestoneResponse`.

Un altro metodo rilevante utilizzato nei Service è quello utilizzato per la generazione della query personalizzata in base ai filtri inseriti dall'utente.

Dato che la query con Spring Data JPA sarebbe stata troppo lunga e poco leggibile, è stato deciso di crearla tramite `CriteriaBuilder` e `CriteriaQuery` che permettono di creare query dinamiche in modo programmatico.

```

CriteriaBuilder builder = entityManager.getCriteriaBuilder();
CriteriaQuery<Milestone> criteriaQuery = builder.createQuery(Milestone.class);
Root<Milestone> root = criteriaQuery.from(Milestone.class);
criteriaQuery.select(root).distinct(true);

Join<Milestone, Progetto> MilestoneProgetto = root.join(attributeName: "progetto", JoinType.LEFT);

List<Predicate> finalPredicateBase = new ArrayList<>();
criteriaQuery.where(finalPredicateBase.toArray(new Predicate[0]));

TypedQuery<Milestone> typedQuery = entityManager.createQuery(criteriaQuery);
typedQuery.setFirstResult(pagination.getPage()*pagination.getPageSize());
typedQuery.setMaxResults(pagination.getPageSize());
List<Milestone> result = typedQuery.getResultList();

```

Figura 5.17: Code snippet della creazione della query basata su filtri inseriti

In questo frammento di codice, ho semplificato la creazione della query dinamica, lasciando i punti chiave che spiegano il funzionamento.

Nel primo blocco di istruzioni inizializziamo `CriteriaBuilder`, una nuova `CriteriaQuery` e selezioniamo tutti i campi dell'entità `Milestone`.

Nelle successive istruzioni andiamo a creare tutte le operazioni di *Join*<> tra le entità, inseriamo dei predicati (condizioni logiche, utili quando la query è dinamica) nella clausola **where()** della query e infine generiamo la query simulando la paginazione recuperando la lista di risultati in una Lista di **Milestone**.

5.3.5 Swagger UI

Spiegazione come è stata creata la grafica per gli endpoint con screen di un'interfaccia Controller.

Capitolo 6

Verifica e Validazione

6.1 Scopo del capitolo

Questo capitolo descrive come è stata attuata la verifica e la validazione all'interno del progetto. Grazie al processo di verifica garantiamo che ogni attività dei processi svolti non introduca errori nel prodotto e che soddisfi i requisiti. Mentre con il processo di validazione viene determinato in maniera oggettiva che il prodotto sia conforme ai requisiti richiesti.

6.2 Processo di verifica

Durante tutto il periodo di tirocinio, ad ogni avanzamento di funzionalità e quindi cambio di requisito da soddisfare, con il tutor Antonio Fasolato è stato verificato che il codice sviluppato fino a quel momento fosse conforme con quanto aspettato e producesse output veritieri. Questo era aiutato inoltre dai controlli di validazione effettuati nei vari Controller e ulteriori controlli inseriti nei Service per garantire l'utilizzo di valori corretti.

Svolgendo le seguenti analisi regolarmente è stato possibile identificare e correggere errori evitando la loro propagazione nel corso della Codifica.

6.2.1 Analisi statica

Veniva effettuata un'analisi statica che non richiede l'esecuzione del software, sul codice prodotto, eseguendo prima un inspection, una lettura mirata e focalizzata sugli errori più noti e probabili, andando verso errori più specifici in base alla funzionalità implementata.

6.2.2 Analisi dinamica

L'analisi dinamica consiste nell'effettuare test sul prodotto in esecuzione. In caso di errore veniva eseguita un'ulteriore verifica successivamente alla mia correzione dell'errore che si era presentato.

6.2.2.1 Debugging

Per identificare al meglio un errore a run-time veniva usata la tecnica di debugging: consiste nell'individuazione e la correzione di errori che provocano anomalie rilevate durante l'esecuzione del programma. Questa tecnica è possibile grazie allo strumento di debugging che offre IntelliJ che permette di inserire dei breakpoint, cioè dei punti di interruzione nel codice, che non appena raggiunti durante l'esecuzione, sospende l'esecuzione ed è possibile ispezionare lo stato delle variabili, delle strutture dati e lo stack trace delle chiamate.

6.2.2.2 Postman

Per testare l'API veniva utilizzato il software Postman. Esso permetteva di osservare la response dell'endpoint in analisi per verificare che fosse conforme alle attese.

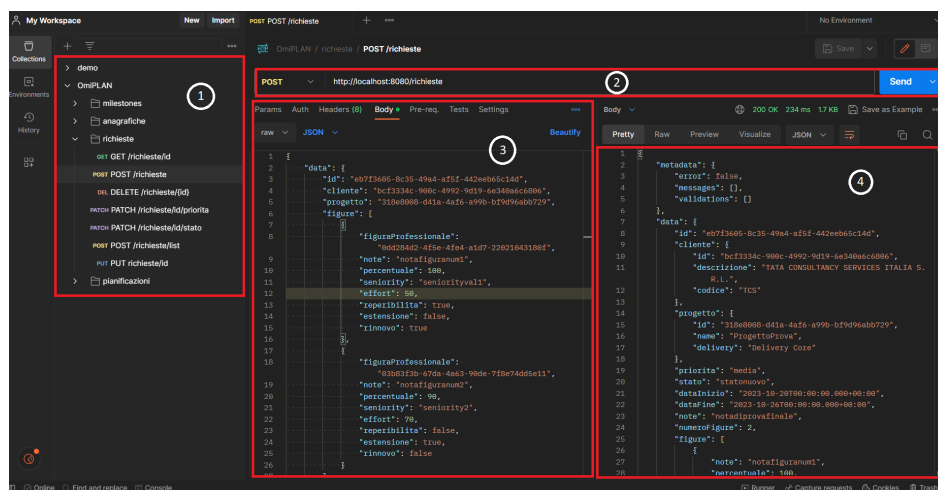


Figura 6.1: Interfaccia Postman

Per testare un endpoint si andava a configurare Postman nelle seguenti sezioni:

1. sezione in cui è possibile creare una nuova richiesta all'interno di una Collezione;
2. sezione in cui è possibile inserire l'URL (Uniform Resource Locator) che indica l'indirizzo del server e il percorso dell'endpoint a cui si desidera accedere, si seleziona il verbo HTTP e si aggiungono eventuali parametri di query. I parametri di query sono parte dell'URL che consente all'utente di filtrare, paginare o personalizzare i risultati;
3. sezione in cui è possibile inserire il body della request. Sopra al body si possono notare varie opzioni, di cui è stata utilizzata principalmente la sezione "Params", che permette di scrivere i parametri di query in modo più strutturato;
4. sezione in cui si potrà visualizzare la risposta in seguito al click sul pulsante "Send". La risposta sarà conforme alle specifiche selezionate.

6.3 Testing

Per garantire la sicurezza e l'affidabilità, negli ultimi giorni di tirocinio, sono stati introdotti i test di unità.

Questo processo è iniziato con lo studio dei framework JUnit e Mockito.

I test di unità verificano il funzionamento corretto di singole unità di codice, come metodi o classi. Questi test vengono sviluppati isolandosi dalle dipendenze esterne permettendo di verificare che il codice funzioni correttamente e che produca risultati attesi.

È stato ritenuto ragionevole testare ciò che io avevo prodotto come metodi o classi. Componenti introdotti da librerie o framework non hanno necessitato di test, dando per assodato il loro funzionamento.

Dato il poco tempo rimanente, per mettere in pratica quanto appreso, si è deciso di limitare lo svolgimento dei test su un *@Service* semplice, *MilestoneService*, andando a testare il corretto funzionamento dei suoi metodi.

6.3.1 Esempio di un test effettuato

```
@ExtendWith(MockitoExtension.class)
public class MilestoneServiceTests {

    14 usages
    @InjectMocks
    SchedMilestoneService milestoneService;

    11 usages
    @Mock
    SchedMilestoneRepository milestoneRepository;

    1 usage
    @Mock
    SchedProgettoRepository progettoRepository;

    1 usage
    @Mock
    SchedPianificazioneRepository pianificazioneRepository;

    @BeforeEach
    public void init() { MockitoAnnotations.openMocks( testClass: this); }
```

Figura 6.2: Configurazione ambiente di testing

Per poter eseguire dei test su oggetti simulati (mock) è stata utilizzata l'annotazione *@Mock* che ha permesso di creare i componenti mock da iniettare all'interno del Service sotto test, tramite l'annotazione *@InjectMocks*. Tramite il metodo *init()*, annotato con *@BeforeEach*, inizializziamo i componenti che vogliamo mockare permettendo l'utilizzo delle annotazioni che Mockito fornisce.

Qui riporto un code snippet dei controlli eseguiti nel test relativo al metodo di creazione di una nuova Milestone. Per garantire il corretto funzionamento della creazione della Milestone, è stata testata ogni parte delicata che avrebbe compromesso il funzionamento del metodo, controllando l'eccezione sollevata, al momento dell'errore, assicurando

che fosse quella corretta.

```
//Forcing the exception "Milestone già esistente"
when(milestoneRepository.findById(milestoneAlreadyPresent)).thenReturn(Optional.of(new Milestone()));
try {
    response = milestoneService.postMilestone(milestoneRequest);
    fail("The method should have raised an OmiPlanException");
} catch (OmiPlanException e) {
    // We expect the exception
    if(!e.getMessage().equalsIgnoreCase(String.format("L'id %s esiste già nel DB", milestoneAlreadyPresent))) {
        fail(String.format("Wrong exception raised (%s)", e.getMessage()));
    }
    // Correct exception raised
}

//Reset the Milestone Id to a new one
request.setId(UUID.randomUUID().toString());

when(progettoRepository.findById(projectId)).thenReturn(Optional.of(progetto));

when(pianificazioneRepository.findById(pianificazioneId)).thenReturn(Optional.of(pianificazione));

when(milestoneRepository.save(any(Milestone.class))).thenReturn(new Milestone());

response = milestoneService.postMilestone(milestoneRequest);

ArgumentCaptor<Milestone> requestArgumentCaptor = ArgumentCaptor.forClass(Milestone.class);
verify(milestoneRepository, times( wantedNumberOfInvocations: 1)).save(requestArgumentCaptor.capture());

Milestone capturedRequest = requestArgumentCaptor.getValue();

Assertions.assertThat(response).isNotNull();
assertEquals(capturedRequest.getId(), request.getId());
```

Figura 6.3: Code snippet test di creazione di una nuova Milestone

In questo snippet possiamo osservare come è stata forzata il lancio dell'eccezione quando si va a creare una Milestone con Id già esistente, controllando che sia stata proprio quella l'eccezione lanciata dal metodo.

Mockito offre strumenti, come il metodo `when()`, che consentono di effettuare `stubbing` dei metodi, simulando il loro comportamento. Questo permette di definire come tali metodi dovrebbero rispondere quando vengono invocati, restituendo risultati specifici anziché eseguire il codice reale.

Un ulteriore metodo fornito da Mockito è `verify()`, utilizzato per controllare che il metodo `save()` della repository della Milestone venga chiamato una sola volta. Con `ArgumentCaptor<>` (che permette di raccogliere i parametri passati in una funzione) andiamo a "catturare" la Milestone salvata col metodo `save()`.

Per accertare che questa parte di test abbia avuto successo è stato verificato che la response ricevuta dal metodo non fosse vuota e, dato che il DTO restituito differisce per alcune caratteristiche dall'istanza completa di una Milestone, non è stato possibile un confronto intero tra i due oggetti, ed è stato ritenuto esaustivo garantire l'uguaglianza dei due Id. Queste verifiche sono state effettuate tramite metodi di asserzione forniti da JUnit. Con l'ausilio di queste metodologie è stato possibile simulare e controllare la corretta creazione di una Milestone.

6.4 Validazione

Nell'ultimo giorno di tirocinio si è tenuto un incontro conclusivo insieme al tutor Antonio Fasolato e al Senior Technical Leader Giovanni Incammicia, a conferma definitiva della conformità alle specifiche richieste del prodotto finale. Durante questa riunione, è stata effettuata una revisione dell'Analisi dei Requisiti fornita all'inizio del progetto, con particolare attenzione ai requisiti di mia competenza, al fine di verificare il loro soddisfacimento. Tuttavia, per scrivere il codice dell'API, è stato utilizzato principalmente un mock dell'API e l'Analisi dei Requisiti fornita è stata utilizzata per aiutare la comprensione di alcuni endpoint. Questo approccio ha comportato alcune discrepanze tra le funzionalità implementate e i requisiti specificati. Rispetto all'Analisi dei Requisiti mancavano delle informazioni che il Front-end necessitava o alcune funzionalità che necessitavano implementazioni particolari o di collegamenti a servizi esterni di cui non c'era tempo a sufficienza per poter studiare. Nonostante queste mancanze, analizzando le funzionalità implementate e le risposte fornite, è stato ritenuto sufficiente per il soddisfacimento dei requisiti di cui gli endpoint sviluppati facevano parte.

A causa delle problematiche da me sopra menzionate, non è stato possibile condurre una fase di Collaudo effettiva per valutare l'interazione tra il Front-end e il Back-end.

Capitolo 7

Deploy dell'applicazione

7.1 Scopo del capitolo

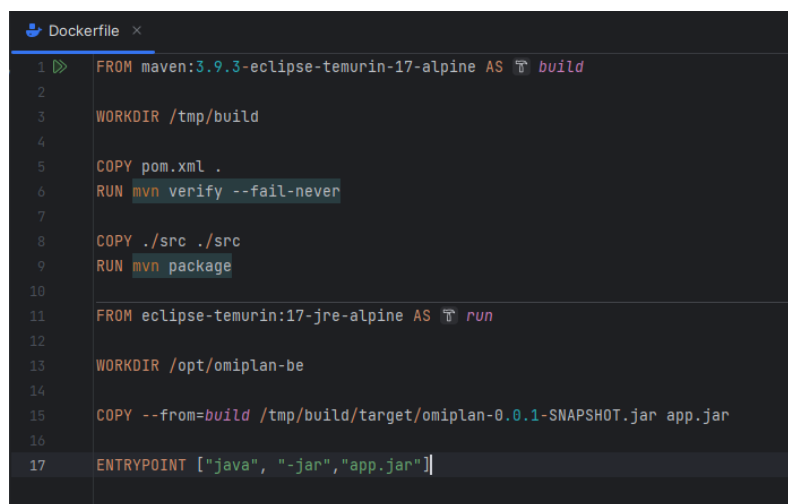
Al termine dell'attività di stage, insieme al tutor, ho potuto osservare come avviene il deploy di una Spring Boot app su Docker.

Per deployment del software si intende la distribuzione o avviamento, con relativa installazione e configurazione, di un'applicazione. È considerabile come fase finale del ciclo di vita di un software, fase che conclude lo sviluppo e il collaudo, dando inizio alla manutenzione.

7.2 Deploy

7.2.1 Dockerfile

Per cominciare è stato creato un `Dockerfile` nella root del progetto. Questo file definirà come Docker dovrebbe costruire l'immagine.

A screenshot of a code editor showing a Dockerfile. The file is titled 'Dockerfile' with a close button. The content is as follows:

```
1 FROM maven:3.9.3-eclipse-temurin-17-alpine AS build
2
3 WORKDIR /tmp/build
4
5 COPY pom.xml .
6 RUN mvn verify --fail-never
7
8 COPY ./src ./src
9 RUN mvn package
10
11 FROM eclipse-temurin:17-jre-alpine AS run
12
13 WORKDIR /opt/omiplan-be
14
15 COPY --from=build /tmp/build/target/omiplan-0.0.1-SNAPSHOT.jar app.jar
16
17 ENTRYPOINT ["java", "-jar", "app.jar"]
```

Figura 7.1: Configurazione Dockerfile

All'interno del file, possiamo notare due stage separati, quello di build e quello di run (esecuzione).

Stage Build

Nello stage di "build" viene indicato che si sta costruendo un'immagine di Docker basata su Maven 3.9.3 e Java 17. Vengono poi eseguite operazioni relative alla compilazione dell'applicazione, come la copia del codice sorgente, la verifica delle dipendenze tramite il tool di build Maven e la creazione del pacchetto `.jar`.

Stage Run

Lo stage di "run" ha come obiettivo l'esecuzione dell'applicazione. In questo stage viene utilizzata nuovamente la stessa immagine di Java e vengono copiati i risultati della prima fase. L'istruzione finale `ENTRYPOINT` definisce il comando che verrà eseguito quando il container basato su questa immagine verrà avviato.

7.2.2 Costruzione dell'immagine e avvio dell'applicazione

In seguito alla creazione del `Dockerfile` si apre il terminale nella directory in cui si trova questo file.

Seguono poi i seguenti passi:

1. viene eseguito il comando `docker build -t nome_immagine` per creare l'immagine Docker con le specifiche selezionate;
2. una volta costruita l'immagine si può avviare il container tramite il comando `docker run -ti nome_immagine`;
3. per accedere all'applicazione è possibile utilizzare un browser o un'applicazione da riga di comando come `cURL`, digitando il seguente indirizzo per interagire con l'applicazione: `http://localhost:porta_scelta`.

Capitolo 8

Conclusioni

8.1 Consuntivo finale

8.2 Raggiungimento degli obiettivi

8.3 Conoscenze acquisite

8.4 Valutazione personale

Glossario

Bibliografia

- [https://it.wikipedia.org/wiki/Java_\(linguaggio_di_programmazione\)](https://it.wikipedia.org/wiki/Java_(linguaggio_di_programmazione))
- <https://docs.spring.io/spring-framework/reference/overview.html>
- <https://spring.io/projects/spring-boot>
- <https://spring.io/projects/spring-data>
- https://it.wikipedia.org/wiki/Structured_Query_Language
- <https://www.json.org/json-it.html>
- https://it.wikipedia.org/wiki/IntelliJ_IDEA
- <https://www.baeldung.com/maven>
- <https://en.wikipedia.org/wiki/DBeaver>
- https://it.wikipedia.org/wiki/Microsoft_SQL_Server
- <https://www.baeldung.com/spring-boot-hibernate>
- <https://learning.postman.com/docs/introduction/overview/>
- https://it.wikipedia.org/wiki/Microsoft_Teams
- [https://en.wikipedia.org/wiki/Notion_\(productivity_software\)](https://en.wikipedia.org/wiki/Notion_(productivity_software))
- https://it.wikipedia.org/wiki/Microsoft_Excel
- https://it.wikipedia.org/wiki/Visual_Studio_Code
- <https://it.wikipedia.org/wiki/Docker>
- <https://swagger.io/docs/specification/about/>
- <https://git-scm.com/docs/git>
- <https://it.wikipedia.org/wiki/GitLab>
- <http://gitextensions.github.io/>
- https://it.wikipedia.org/wiki/Front_Controller_pattern
- <https://www.baeldung.com/jpa-entities>
- <https://www.adecco.it/il-lavoro-che-cambia/program-manager>

- <https://www.adecco.it/il-lavoro-che-cambia/project-manager>
- <https://site.mockito.org/>
- <https://it.wikipedia.org/wiki/JUnit>
- <https://it.wikipedia.org/wiki/Deployment>
- https://it.wikipedia.org/wiki/Sistema_client/server
- <https://www.baeldung.com/spring-component-repository-service>