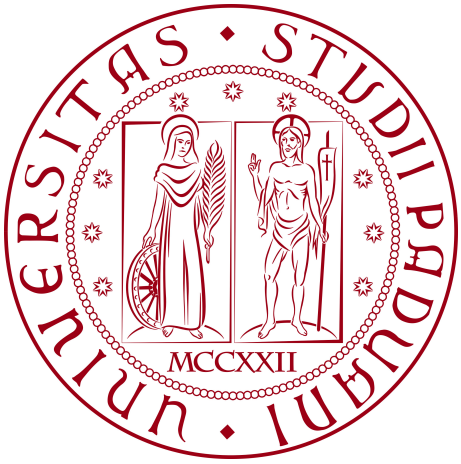
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



RAG/LLM fitness assistant: sviluppo con Spring ed Angular

Tesi di Laurea Triennale

Relatore

Prof. Vardanega Tullio

Laureando

Vedovato Alex

Matricola 2042353

Anno Accademico 2023-2024

"Ogni giorno passa un giorno."

— Vedovato Agostino

Ringraziamenti

Innanzitutto, desidero esprimere la mia profonda gratitudine a tutta la mia famiglia, ed in particolare ai miei nonni ed ai miei genitori, per il loro incessante sostegno e per la loro presenza costante durante tutti i miei anni di studio. Senza il loro incoraggiamento e la loro fiducia, non sarei la persona che sono oggi.

Un sentito ringraziamento va anche a mio fratello Daniel, che è sempre stato una grande fonte di ispirazione per me. La sua dedizione, il suo impegno e la sua passione per ciò che fa mi hanno costantemente spronato a dare il meglio di me stesso. È anche grazie a lui che ho intrapreso questo percorso, seguendo il suo esempio e i suoi consigli preziosi, e di questo non potrò mai ringraziarlo abbastanza.

Desidero poi ringraziare con affetto tutti i miei amici, i quali hanno reso questo viaggio di studio e crescita personale ancora più significativo. Il loro sostegno e i momenti di gioia condivisi mi hanno aiutato a superare le difficoltà e a rimanere motivato. Un grazie speciale va a Michele, Samuele, Nicolò, Andrea ed Alessandro. La vostra amicizia ha reso tutto più leggero e piacevole, e non avrei potuto desiderare compagni migliori in questa avventura.

Inoltre, desidero esprimere la mia gratitudine al professor Vardanega Tullio, mio relatore, per il prezioso aiuto e il costante sostegno che mi ha fornito durante la stesura di questo elaborato. La sua competenza, i suoi consigli saggi e la sua disponibilità hanno rappresentato una guida insostituibile in ogni ~~fase~~ attività del mio lavoro.

Infine voglio ringraziare me stesso, perchè non è nelle stelle che è conservato il nostro destino, ma in noi stessi. Uomini forti destini forti, uomini deboli destini deboli. Non c'è altra strada. Ed io vinco. Sempre.

Padova, Luglio 2024 *Vedovato Alex*

Sommario

Questo documento presenta il lavoro svolto e le conclusioni emerse dallo studio effettuato durante il periodo di *stage*, della durata di circa trecento ore, dal laureando Vedovato Alex presso l'azienda SyncLab.

Ho strutturato il documento in quattro capitoli. Il primo capitolo, **Contesto aziendale**, presenta l'organizzazione, i suoi processi interni, le tecnologie utilizzate, i servizi offerti, i principali clienti e l'approccio all'innovazione dell'azienda. Il secondo capitolo, ***Stage* proposto**, discute i problemi affrontati durante lo *stage*, le attività correlate, il rapporto dell'azienda con gli *stage*, gli obiettivi ed i vincoli del progetto e le ragioni per cui ho scelto questa opportunità. Il terzo capitolo, **Progetto di *stage***, descrive gli elementi essenziali del progetto, il metodo di lavoro, le sfide affrontate e i risultati ottenuti. Infine, il quarto capitolo, **Retrospettiva**, valuta l'esperienza complessiva, il raggiungimento degli obiettivi, la crescita professionale maturata, le conoscenze acquisite e il confronto tra le competenze richieste e quelle fornite dal corso di studi.

Per la stesura del testo di questo documento ho adottato le convenzioni tipografiche riportate qui di seguito. I termini in lingua diversa dall'italiano sono posti in corsivo, per segnalare il loro uso intenzionale. I termini ambigui o di uso non comune sono definiti nel glossario, situato alla fine del documento. La prima occorrenza dei termini riportati nel glossario è contrassegnata con la seguente nomenclatura: *termine*$_{\boldsymbol{G}}$. Infine, ho accompagnato tutte le immagini presenti nel documento con la fonte d'origine, se acquisite da fonti esterne o siti terzi.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici sorgenti

Capitolo 1

Contesto aziendale

1.1 L'azienda

SyncLab è una società italiana con sede a Napoli, fondata nel 2002, che opera nel settore dell'*Information and Communication Technology*$_G$ ed offre servizi di consulenza alle aziende.

Come riportato sul sito ufficiale dell'azienda, in pochi anni SyncLab si è affermata come *system integrator*$_G$ di riferimento, maturando competenze tecnologiche, metodologiche e applicative nel dominio del *software*. Ad oggi, infatti, vanta oltre trecento dipendenti e sei sedi in Italia situate a Napoli, Roma, Milano, Como, Verona ed infine Padova (dove ho avuto il piacere di svolgere il mio *stage* curricolare), riuscendo a servire oltre centocinquanta clienti diretti e finali.

Per esperienza personale posso affermare che SyncLab è un'azienda in continua crescita che guarda al futuro con fiducia, investendo costantemente in innovazione e formazione del personale, per offrire ai propri clienti le soluzioni più avanzate e competitive. Si distingue come un partner IT affidabile e innovativo, in grado di accompagnare le aziende nella loro trasformazione digitale e nel raggiungimento dei loro obiettivi di *business*, e può vantare numerose certificazioni, a testimonianza del suo impegno per la qualità e la sicurezza dei propri servizi, che ho avuto modo di osservare anche in prima persona.

1.2 Servizi offerti e principali clienti

Durante il mio *stage*, ho avuto la possibilità di conversare con il tutor aziendale e diversi dipendenti dell'azienda e ho appreso che SyncLab offre una vasta gamma di servizi che coprono l'intero ciclo di vita del *software*, dall'ideazione alla manutenzione, realizzando prodotti e soluzioni per diversi mercati quali: sanità, industria, energia, telco, finanza e trasporti e logistica.
Tra i principali clienti si possono trovare aziende di tutte le dimensioni, sia pubbliche che private, come ad esempio Sky, Eni, Enel, Tim, Vodafone, Trenitalia, Poste Italiane, UniCredit e moltissime altre ancora.
Quest'ultime usufruiscono principalmente della seguente offerta:

- **Sviluppo *web* e *mobile***: Synclab realizza applicazioni *web* e *mobile* personalizzate per le esigenze specifiche di ogni cliente, utilizzando le più moderne tecnologie e metodologie di sviluppo;
- **Pianificazione delle risorse aziendali**: Synclab implementa soluzioni ERP$_G$ per la gestione efficiente dei processi aziendali digitalizzandoli;
- **Integrazione di applicazioni aziendali**: Synclab integra i diversi sistemi aziendali esistenti, creando un'infrastruttura IT omogenea ed efficiente;
- **Servizi gestiti**: SyncLab offre una vasta gamma di servizi gestiti per aiutare le aziende a migliorare la loro infrastruttura IT e le loro operazioni;
- **Gestione reti**: Synclab progetta e implementa reti IT sicure e performanti per le aziende;
- **Soluzioni e consulenza per il settore marittimo**: Synclab offre soluzioni IT specifiche per le esigenze del settore marittimo, come la gestione delle flotte, la logistica marittima e la sicurezza marittima;
- **Consulenza aziendale**: Synclab affianca i propri clienti nell'elaborazione di strategie IT e nell'ottimizzazione dei processi aziendali;
- **Gestione dei dati**: Synclab aiuta le aziende a gestire i propri dati in modo efficiente e sicuro, traendo valore da essi per il *business*;

- ***E-health***: Synclab sviluppa soluzioni IT per strutture sanitarie pubbliche e private, mettendo al centro i bisogni del paziente e le esigenze di gestione di grandi quantità di dati;
- **Trasporti e logistica**: Synclab offre soluzioni IT per ottimizzare la gestione della catena di approvvigionamento e dei processi logistici;
- **Metaverso e ologrammi**: Synclab sviluppa soluzioni per il metaverso e l'olografia, offrendo alle aziende nuove opportunità di *engagement* e collaborazione;
- ***Blockchain***: Synclab offre servizi di consulenza e sviluppo per l'adozione della tecnologia *blockchain* nelle aziende.

Inoltre, come mi è stato spiegato, nascendo come *software house*$_G$ prima di tramutarsi in *system integrator*, SyncLab offre una serie di prodotti e soluzioni innovative che si distinguono per la loro qualità intrinseca (vedi Figura 1.1).

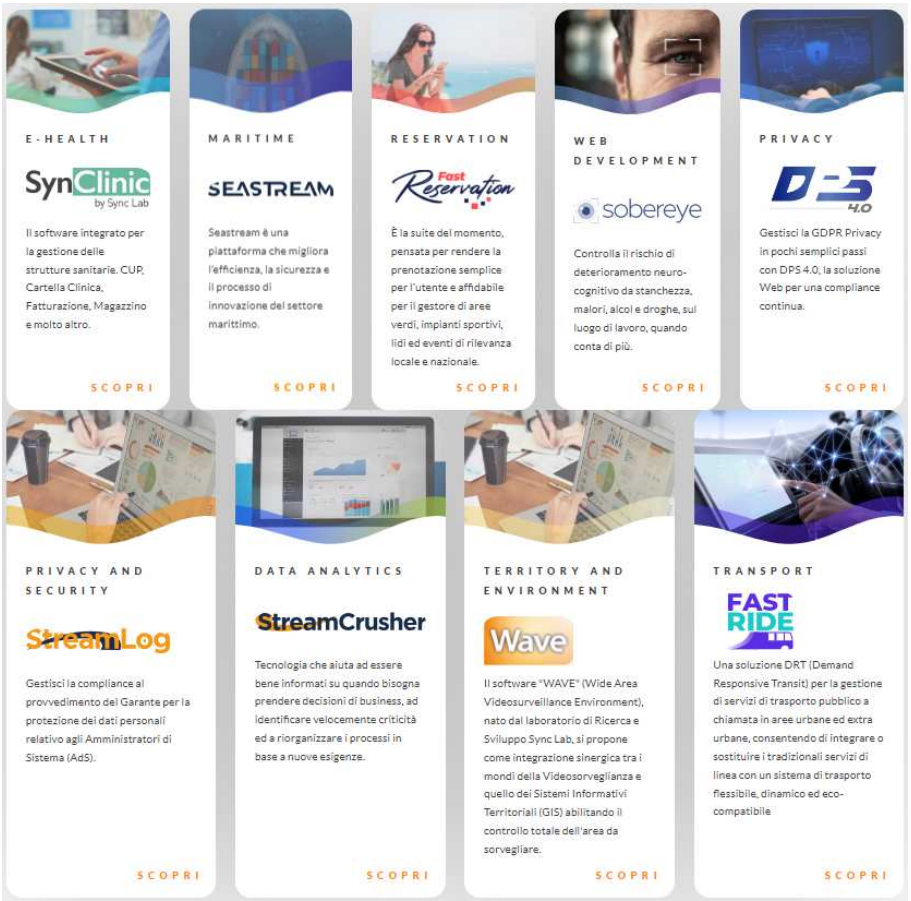


Figura 1.1: Prodotti principali offerti dall'azienda SyncLab

Fonte: synclab.it

1.3 Organizzazione del lavoro

Come ho avuto modo di vedere e vivere, il lavoro in SyncLab è organizzato seguendo il modello di sviluppo *agile*, che enfatizza la flessibilità, la collaborazione e il miglioramento continuo. Si lavora quindi in cicli iterativi e incrementali, chiamati *sprint*, che permettono di adattarsi rapidamente ai cambiamenti e di fornire valore aggiunto in tempi brevi, utilizzando determinati processi e tecnologie che vado a descrivere nel dettaglio in questa sezione.
SyncLab inoltre adotta il *remote working* per la maggior parte della settimana, consentendo ai dipendenti di lavorare dovunque preferiscano. Questa modalità offre numerosi vantaggi, come una maggiore autonomia, una migliore gestione del tempo e un equilibrio ottimale tra vita professionale e personale. Tuttavia, almeno un giorno alla settimana, il *team* si riunisce nella sede a Padova. Questo giorno in presenza è dedicato a incontri di persona, *brainstorming*, attività di *team building* e altre interazioni che beneficiano della comunicazione faccia a faccia.

1.3.1 Processi aziendali e tecnologie utilizzate

1.3.1.1 Gestione di progetto

Per gestire i propri progetti, SyncLab adotta il *framework* Scrum (di cui si può osservare una panoramica nella Figura 1.2) come metodologia *agile* principale. Scrum è particolarmente adatto per progetti complessi in cui i requisiti possono cambiare frequentemente e il *focus* è sulla consegna rapida di valore.
Nel contesto di Scrum, il *team* di lavoro è diviso in ruoli chiave: il *product owner*, il *team* di sviluppo e lo Scrum *master*. Il *product owner* è responsabile di definire le funzionalità del prodotto e di gestire il *backlog*$_G$ del prodotto, che contiene tutte le attività da completare. Il *team* di sviluppo invece è responsabile di sviluppare e consegnare le funzionalità secondo le specifiche definite dal *product owner*. Infine, lo Scrum *master* si assicura che il *team* segua correttamente i principi e le pratiche di Scrum, rimuovendo gli ostacoli e facilitando le riunioni.
Le attività vengono pianificate in *sprint*, periodi di tempo generalmente di durata da una a quattro settimane, durante i quali il *team* si impegna a completare un

insieme specifico di attività. All'inizio di ogni *sprint*, il *team* tiene una riunione di pianificazione dello *sprint*, detta *sprint planning*, per selezionare le attività dal backlog del prodotto e definire gli obiettivi da raggiungere nel periodo di lavoro prefissato. Durante quest'ultimo, il *team* svolge inoltre riunioni giornaliere di stand-up, dette *daily* Scrum, per aggiornarsi sul progresso e identificare eventuali ostacoli. Al termine dello *sprint*, il *team* esegue una riunione di revisione dello *sprint*, detta *sprint review*, per rivedere il lavoro completato e ricevere *feedback* dal *product owner* e dagli *stakeholder*$_G$. Successivamente, tiene una riunione di retrospettiva, detta *sprint retrospective*, per riflettere sulle pratiche di lavoro e identificare eventuali miglioramenti da apportare.

Durante lo *stage* curricolare, il tutor aziendale, Fabio Pallaro, ha assunto i ruoli di *product owner* e Scrum *master*, mentre il *team* di sviluppo era composto da me e da un altro stagista. Tutti i principi esposti in precedenza sono stati rigorosamente seguiti, con *sprint* della durata di una settimana.

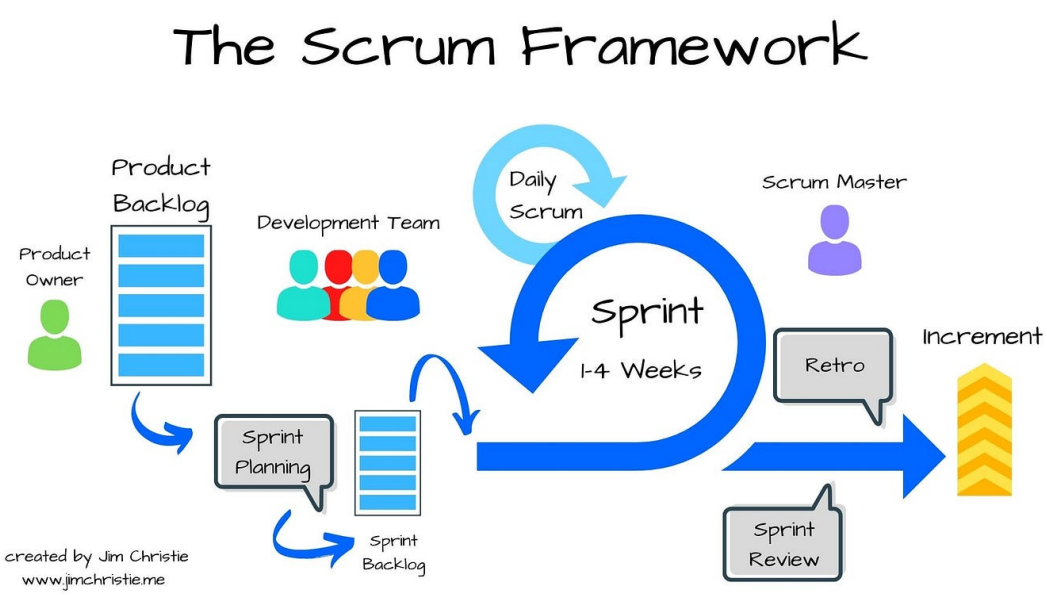


Figura 1.2: Panoramica del *framework* Scrum

Fonte: medium.com

Strumenti e tecnologie utilizzati

Figura 1.3: Strumenti e tecnologie di gestione utilizzati

Trello: uno strumento di gestione dei progetti basato sul *web* che utilizza un sistema di bacheche, liste e *card* per organizzare e tracciare le attività di un progetto. Trello è conosciuto per la sua interfaccia *user-friendly*$_G$ e per la flessibilità che offre nella gestione del lavoro.
All'interno del percorso di *stage* il *team* di lavoro ha utilizzato Trello per la gestione e pianificazione settimanale delle attività, dando prova dello stato di avanzamento raggiunto e consentendo una più facile organizzazione dei compiti da svolgere.

Google Calendar: un'applicazione di calendario *online* che consente agli utenti di creare, condividere e gestire eventi, pianificare appuntamenti e organizzare il proprio tempo in modo efficiente.
All'interno del percorso di *stage* il *team* di lavoro ha utilizzato Google Calendar per segnare i giorni di presenza in sede, facilitando la pianificazione e la coordinazione delle attività settimanali del *team*.

Discord: una piattaforma di comunicazione versatile originariamente sviluppata per i giocatori, ma ormai ampiamente utilizzata da diverse comunità e *team* di lavoro per la sua flessibilità e facilità d'uso. Discord offre una combinazione di funzionalità di *chat* testuale, vocale e video, permettendo una comunicazione efficace in vari contesti.
All'interno del percorso di *stage* il *team* di lavoro ha utilizzato Discord per comunicare nelle giornate di *remote working* e condividere *file* e documenti riguardanti gli argomenti oggetto di studio.

GitHub Issue Tracking System: una funzionalità di GitHub, famosa piattaforma di sviluppo *software*, progettata per gestire e tracciare problemi, *bug*, richieste di funzionalità e altre attività correlate allo sviluppo di *software*. È una parte integrata dei *repository*$_G$ GitHub e offre vari strumenti per facilitare la collaborazione e la gestione del progetto.
All'interno del percorso di *stage* il *team* di lavoro ha utilizzato questa funzionalità di GitHub per segnalare e gestire i *bug* rilevati, nuove funzionalità e altre attività di progetto. Inoltre, abbiamo utilizzato le *milestone* per raggruppare *issue*$_G$ e *pull request*$_G$ correlate a specifici obiettivi del progetto. Quest'ultime hanno permesso di organizzare il lavoro in fasi temporali e di monitorare il progresso verso il completamento di importanti traguardi. Ogni *milestone* rappresentava un insieme di attività da completare entro una determinata scadenza, migliorando la pianificazione e la gestione delle priorità del progetto.

1.3.1.2 Sviluppo *software*

Il processo di sviluppo *software* in SyncLab segue le classiche attività del ciclo di vita del *software*, ovvero le seguenti:

- **Analisi dei requisiti**: in questa attività vengono identificati e analizzati gli obiettivi e i requisiti del *software* in collaborazione con gli *stakeholder*. Questo coinvolge la raccolta di informazioni sulle esigenze degli utenti finali, i requisiti funzionali e non funzionali, e le limitazioni del sistema;

- **Progettazione**: qui vengono definite le specifiche tecniche e funzionali del sistema. Questa attività include la progettazione dell'architettura del *software*, la definizione dell'interfaccia utente e la suddivisione del sistema in moduli o componenti più piccoli;

- **Implementazione**: il codice sorgente del *software* viene effettivamente scritto in base alla progettazione stabilita. Gli sviluppatori traducono le specifiche tecniche in codice utilizzando i linguaggi di programmazione appropriati;

- ***Testing***: attività critica del processo di sviluppo *software*, durante la quale il *software* viene valutato per verificarne il corretto funzionamento e individuare eventuali difetti o *bug*. Questo include *test* unitari, *test* di integrazione e *test* di sistema;

- ***Deployment***: una volta che il *software* è stato testato e validato con successo, viene rilasciato in produzione o reso disponibile agli utenti finali. Questo può coinvolgere l'installazione del *software* su *server* o dispositivi degli utenti, o il rilascio di una nuova versione di un'applicazione;

- **Manutenzione**: dopo il rilascio, il *software* richiede manutenzione continua per correggere eventuali *bug* scoperti dagli utenti, aggiornare le funzionalità esistenti e rispondere alle nuove esigenze degli utenti. Il supporto agli utenti è essenziale per fornire assistenza e risolvere eventuali problemi riscontrati dagli utenti finali;

- **Miglioramento**: il processo di sviluppo *software* è spesso iterativo, il che significa che le attività di analisi, progettazione, implementazione e *test* si ripetono in cicli successivi per migliorare il *software* e rispondere ai *feedback* degli utenti. Questo approccio consente un miglioramento continuo del prodotto nel tempo.

Strumenti e tecnologie utilizzati

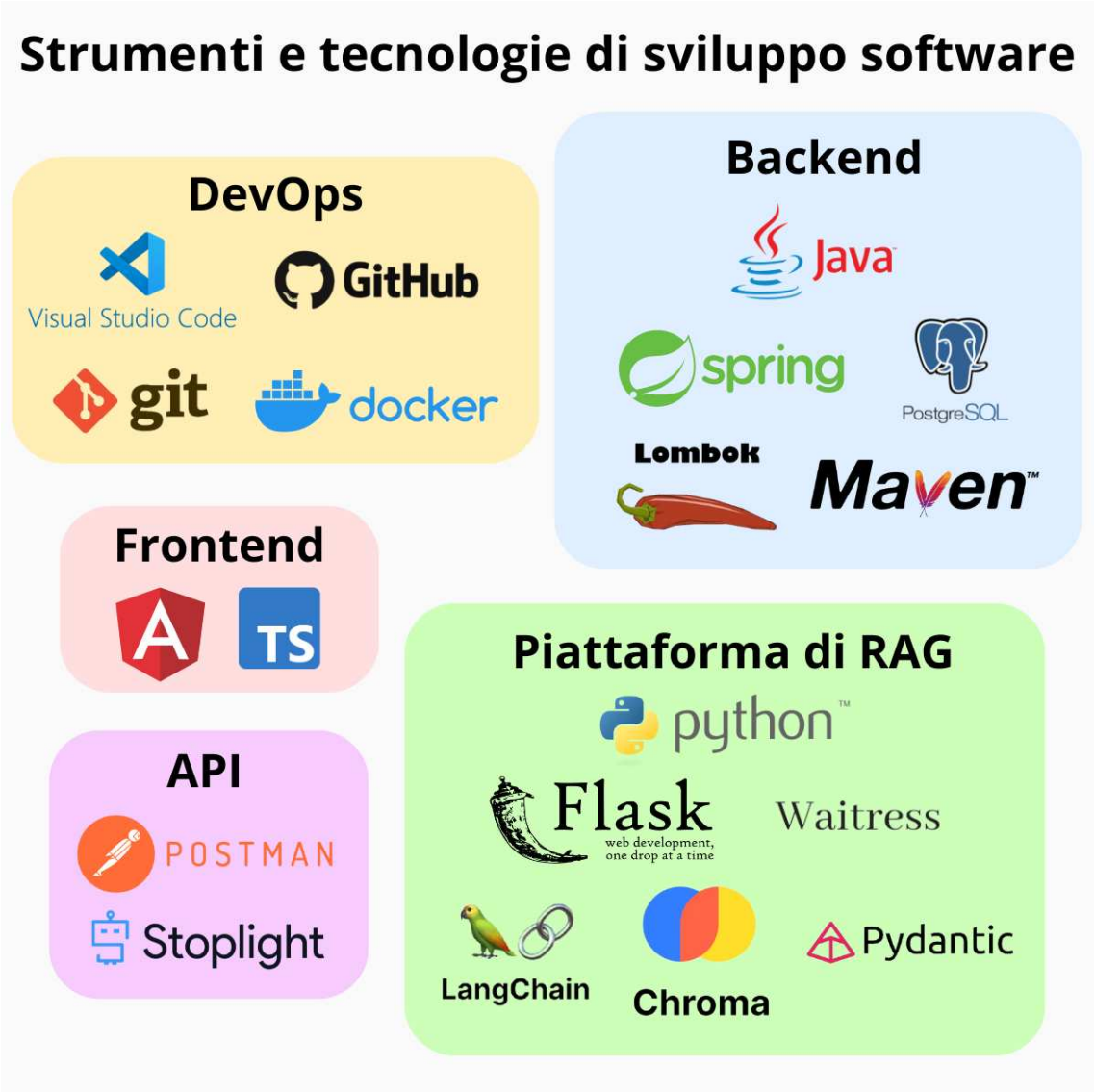


Figura 1.4: Strumenti e tecnologie di sviluppo software utilizzati

Figura 1.5: Strumenti e tecnologie di sviluppo software legati alla parte DevOps

Visual Studio Code: un *editor* di codice sorgente *open-source*$_G$, noto per la sua versatilità, leggerezza e velocità. Una delle sue caratteristiche distintive è la vasta gamma di funzionalità offerte e la possibilità di personalizzazione attraverso estensioni scaricabili dal *marketplace* integrato. Inoltre, grazie alla sua integrazione con Git, consente il controllo di versione direttamente dall'interfaccia dell'*editor*, semplificando il lavoro di sviluppatori e *team*. Dopo ancora, Visual Studio Code supporta una vasta gamma di linguaggi di programmazione e *framework*, rendendolo adatto a molteplici tipi di sviluppo *software*, che vanno dalle applicazioni *web* al *backend*, dalla *data science* alla automazione DevOps. Infine, la collaborazione in tempo reale tramite *live share* consente agli sviluppatori di lavorare insieme su un progetto da posizioni diverse, migliorando la produttività e facilitando la comunicazione.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato questo *editor* per la scrittura dell'intera *code base*$_G$: *frontend*, *backend* e piattaforma di RAG$_G$.

Git: un sistema di controllo di versione distribuito ampiamente utilizzato nel campo dello sviluppo *software* per tracciare e gestire le modifiche al codice sorgente durante lo sviluppo di progetti.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato Git per gestire il codice sorgente efficacemente, collaborare in modo efficiente e garantire l'integrità dei dati nel corso del tempo.

GitHub: una piattaforma di *hosting* per progetti *software* basati su Git. Offre una vasta gamma di funzionalità per consentire agli sviluppatori di collaborare, condividere e gestire progetti *software* in modo efficiente. Tra quest'ultime, le *pull request* rappresentano una caratteristica fondamentale di GitHub, permettendo agli sviluppatori di proporre modifiche al codice sorgente e di richiedere la revisione da parte dei colleghi prima di integrare le modifiche nel *repository* principale. Inoltre, GitHub supporta nativamente *continuous integration* (CI)$_G$ e *continuous deployment* (CD)$_G$, facilitando l'automatizzazione dei processi di *test* e distribuzione del *software*.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato questa piattaforma per creare e gestire un'organizzazione contenente la *repository* di progetto.

Docker: una piattaforma *open-source* che semplifica la creazione, la distribuzione e l'esecuzione di applicazioni in ambienti isolati chiamati *container*. Quest'ultimi includono tutto ciò di cui un'applicazione ha bisogno per funzionare, come codice, librerie e dipendenze. Docker offre un'esperienza consistente e portatile tra ambienti di sviluppo, *test* e produzione, consentendo agli sviluppatori di accelerare il ciclo di sviluppo, garantire la coerenza dell'ambiente e migliorare l'efficienza delle operazioni IT.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato Docker per facilitare lo sviluppo, l'esecuzione ed il *testing* dell'intero prodotto *software*.

Figura 1.6: Strumenti e tecnologie di sviluppo software legati alla parte backend

Java: un linguaggio di programmazione ad alto livello, *object-oriented* e multi-piattaforma. Una delle principali caratteristiche di Java è la sua portabilità, che deriva dalla sua capacità di essere eseguito su diverse piattaforme senza necessità di riscrittura del codice sorgente. Questo è reso possibile dal concetto di "*Write Once, Run Anywhere*" (WORA), grazie al quale il codice Java viene compilato in *bytecode* che può essere eseguito su una macchina virtuale Java (JVM) su qualsiasi piattaforma che la supporti.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato Java per la parte *backend* della *web app*.

Maven: uno strumento di gestione e automazione dei progetti *software* particolarmente diffuso nell'ambito Java. Utilizza un modello di progetto dichiarativo descritto in un *file* XML chiamato "pom.xml", il quale centralizza le informazioni di configurazione per la costruzione del progetto. Maven facilita la gestione delle dipendenze, scaricando e integrando automaticamente le librerie necessarie dal *repository* centrale o da altri *repository* configurati. Offre una struttura standardizzata per i progetti e un ecosistema di *plugin* per estendere le sue funzionalità, coprendo fasi come la compilazione, i *test* e il *packaging*.

All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato Maven per gestire le dipendenze e automatizzare il più possibile il ciclo di vita della parte *backend* della *web app*.

Lombok: una libreria Java che si integra con il compilatore per ridurre il *boilerplate code*$_G$, semplificando e accelerando lo sviluppo. Utilizzando annotazioni, Lombok genera automaticamente metodi comuni come *getter*, *setter*, costruttori, *equals*, *hashCode*, e *toString*, oltre a fornire altre funzionalità utili come la generazione di *builder*, la gestione delle eccezioni e l'inizializzazione *lazy*. Questo permette agli sviluppatori di scrivere codice più conciso e leggibile, concentrandosi sulla logica di *business* senza preoccuparsi di dettagli ripetitivi e tediosi. Lombok migliora la produttività e la manutenibilità del codice, riducendo al minimo gli errori comuni associati alla scrittura manuale di questi metodi standardizzati.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato Lombok per evitare la scrittura manuale di metodi ripetitivi e *boilerplate code* nello sviluppo del *backend* della *web app*.

Spring Framework: un *framework* di sviluppo *open-source* per applicazioni Java, progettato per facilitare la creazione di applicazioni *enterprise*$_G$ di alta qualità. Spring fornisce una solida base per la costruzione di applicazioni complesse, grazie alla sua architettura modulare e alla vasta gamma di funzionalità. Il cuore di Spring è il suo IoC$_G$ *container*, che gestisce la creazione, la configurazione e il ciclo di vita degli oggetti Java attraverso l'iniezione delle dipendenze. Questo approccio promuove una maggiore modularità e testabilità del codice. Inoltre, Spring offre supporto per vari aspetti dello sviluppo applicativo tramite i suoi numerosi moduli come Spring Boot, Spring MVC o Spring Data.
All'interno del percorso di *stage* il *team* di sviluppo ha sfruttato questi strumenti per la parte *backend* della *web app*.

Spring MVC: un modulo di Spring *Framework* progettato per costruire applicazioni *web* basate su architettura *Model-View-Controller* (MVC). Fornisce una struttura base per sviluppare applicazioni *web* in modo efficiente, separando la logica di presentazione dalla logica di *business*, utilizzando *controller* per gestire le richieste HTTP$_G$, modelli per rappresentare i dati e viste per renderizzare l'interfaccia

utente. Inoltre supporta anche la gestione delle richieste, la validazione dei dati, e l'integrazione con varie tecnologie di visualizzazione esterne, rendendo lo sviluppo *web* in Java più intuitivo e produttivo.

Spring Boot: un modulo di Spring *Framework* progettato per semplificare la configurazione e lo sviluppo di nuove applicazioni Spring. Introduce un approccio *opinionated*$_G$ che predefinisce le configurazioni di base necessarie, riducendo significativamente il tempo di *setup* e permettendo agli sviluppatori di concentrarsi più rapidamente sulla logica di *business*. Con funzionalità come la configurazione automatica, il *server web* integrato e una vasta gamma di dipendenze preconfigurate, Spring Boot rende lo sviluppo di applicazioni *enterprise* Java rapido, efficiente e meno incline a errori di configurazione, facilitando anche l'implementazione di microservizi.

Sping Data JPA$_G$: un modulo di Spring *Framework* che semplifica lo sviluppo di accesso ai dati basato su JPA. Fornisce un'astrazione di alto livello per interagire con i *database* relazionali utilizzando JPA, riducendo la quantità di codice *boilerplate* necessario per scrivere *query* e operazioni CRUD (*Create, Read, Update, Delete*). Spring Data JPA semplifica il *mapping* tra le entità Java e le tabelle del *database*, offre supporto per *query* personalizzate e operazioni complesse, migliorando la produttività degli sviluppatori e garantendo una maggiore flessibilità nell'accesso ai dati all'interno delle applicazioni Spring.

PostgreSQL Driver: un componente *software* che permette alle applicazioni basate su Spring di interagire con un *database* PostgreSQL. Questo *driver* fornisce le funzionalità necessarie per stabilire una connessione al *database*, eseguire *query*, gestire transazioni e recuperare i risultati all'interno dell'ambiente di sviluppo Spring. Integrando il *driver* PostgreSQL all'interno di un progetto Spring, gli sviluppatori possono sfruttare le potenti funzionalità di Spring Data JPA e altri moduli di Spring per semplificare ulteriormente lo sviluppo dell'applicazione e migliorare l'efficienza nell'accesso e nella gestione dei dati nel *database* PostgreSQL.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato questo componente *software* per far comunicare il *backend* della *web app* con il sottostante *database* PostgreSQL.

Figura 1.7: Strumenti e tecnologie di sviluppo software legati alla parte frontend

Typescript: un linguaggio di programmazione ad alto livello che estende e aggiunge tipizzazione statica al JavaScript *standard*, la quale consente agli sviluppatori di specificare tipi di dati per variabili, parametri di funzione, proprietà degli oggetti e altro ancora. Questo permette di individuare e correggere gli errori nel codice durante le attività di sviluppo, riducendo il rischio di errori e migliorando la qualità del *software*. TypeScript viene poi trasformato in JavaScript *standard* prima dell'esecuzione.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato questo linguaggio per la parte *frontend* della *web app*, visto l'uso di Angular.

Angular: un *framework* di sviluppo *open-source* per applicazioni *web frontend*. Angular è progettato per semplificare lo sviluppo e i *test* di *single page application (SPA)*$_G$ e adotta un approccio a componenti, dove le applicazioni sono suddivise in unità modulari e riutilizzabili, ciascuna con il proprio *template* HTML, stile CSS e logica TypeScript. Angular facilita la gestione dello stato dell'applicazione, il *binding* bidirezionale dei dati e la gestione degli eventi, riducendo il *boilerplate code* e rendendo lo sviluppo più efficiente. Grazie al sistema di *dependency injection*$_G$, Angular permette una facile gestione delle dipendenze tra i vari componenti, migliorando la modularità e la testabilità del codice. Inoltre, offre un robusto sistema di *routing* per la navigazione tra le viste dell'applicazione e strumenti potenti per la gestione delle *form* e delle validazioni.

All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato Angular per la parte *frontend* della *web app*, andando ad utilizzare anche diverse librerie come PrimeNG, Bootstrap, RxJS e Syncfusion.

Figura 1.8: Strumenti e tecnologie di sviluppo software legati alla piattaforma di RAG

Python: un linguaggio di programmazione ad alto livello, interpretato, dinamico e versatile che è diventato estremamente popolare nel campo dello sviluppo *software*, nell'analisi dei dati, nell'intelligenza artificiale e in molte altre aree. Python è progettato per essere facile da imparare e leggere, con una sintassi chiara e intuitiva che favorisce la produttività degli sviluppatori.
All'interno del percorso di *stage* ho utilizzato questo linguaggio per lo sviluppo della piattaforma di RAG.

ChromaDB: un *database* di *embedding*$_G$ *open-source* progettato per semplificare la gestione di documenti di testo, la conversione di testo in *embedding* e l'esecuzione di ricerche per similarità. È particolarmente utile per le applicazioni in ambito intelligenza artificiale le quali si basano sull'utilizzo di *Large Language Model*$_G$.
All'interno del percorso di *stage* ho impiegato ChromaDB come componente fondamentale della piattaforma di RAG sviluppata per gestire l'*embedding* dei documenti.

LangChain: un *framework open-source* progettato per semplificare la creazione di applicazioni basate su *Large Language Model*. LangChain astrae la complessità degli

LLM, rendendoli più accessibili agli sviluppatori con poca o nessuna esperienza di intelligenza artificiale. Inoltre, esso è progettato per essere scalabile e può essere utilizzato per creare applicazioni che gestiscono grandi volumi di dati.
All'interno del percorso di *stage* ho impiegato LangChain come componente fondamentale della piattaforma di RAG sviluppata per gestire l'interfacciamento con gli LLM testati e con ChromaDB.

Pydantic: una libreria Python utilizzata per la validazione e la gestione dei dati. Fornisce un modo semplice ed efficace per definire strutture di dati utilizzando modelli basati su Python, consentendo di validare e convertire i dati in modo automatico. In questa maniera è possibile garantire che i dati rispettino determinati schemi e requisiti, riducendo così la possibilità di errori e migliorando l'affidabilità del codice.
All'interno del percorso di *stage* ho utilizzato Pydantic per validare e gestire i dati generati come *output* da un modello di intelligenza artificiale all'interno della piattaforma di RAG sviluppata.

Flask: un *microframework web* leggero e flessibile per Python, progettato per sviluppare applicazioni *web* semplici e veloci. Con una sintassi chiara e minimalista, Flask offre una vasta gamma di funzionalità per la gestione delle richieste HTTP, il *routing* delle URL, la gestione dei *template* e la manipolazione delle risposte, consentendo agli sviluppatori di creare rapidamente applicazioni *web* senza dover affrontare complessità e *overhead* eccessivi.
All'interno del percorso di *stage* ho utilizzato Flask per esporre gli *endpoint* necessari per consentire al *backend* di interagire con la piattaforma di RAG.

Waitress: un'estensione di terze parti per il *microframework web* Flask che consente la distribuzione di applicazioni Flask in produzione. Esegue un *server* WSGI$_{G}$ integrato per ospitare l'applicazione.

Figura 1.9: Strumenti e tecnologie di sviluppo software legati alle API

Postman: un'applicazione essenziale per gli sviluppatori che semplifica il processo di sviluppo, *testing* e documentazione delle API$_G$. È uno strumento versatile che fornisce un'interfaccia grafica intuitiva per inviare richieste HTTP a un *server* e per esaminare le risposte in modo rapido ed efficiente. Permette agli sviluppatori di creare facilmente richieste personalizzate specificando metodi, *header*, *body* e parametri, e di organizzare le richieste in collezioni per una gestione ordinata e accessibile. Inoltre, Postman facilita la collaborazione tra membri del *team*, consentendo la condivisione di collezioni di richieste, ambienti di lavoro e risultati di *test*.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato Postman per verificare il corretto funzionamento degli *endpoint* che sono stati sviluppati.

Stoplight: una piattaforma completa per la progettazione, lo sviluppo e la gestione delle API, la quale consente ai *team* di sviluppatori di collaborare in modo più efficiente e di ridurre gli errori durante il processo di sviluppo. Con Stoplight, gli utenti possono modellare le API utilizzando un'interfaccia utente intuitiva, generare automaticamente la documentazione e *mock*$_G$ *server* per il *testing*, e monitorare le *performance* delle API in produzione.
All'interno del percorso di *stage* il *team* di sviluppo ha utilizzato Stoplight per facilitare l'API *Driven Development* (ADD), un approccio che mette l'API al centro del processo di sviluppo.

1.4 Propensione all'innovazione

SyncLab è un'azienda rinomata per la sua spiccata propensione all'innovazione. Questo orientamento è evidente nel suo impegno continuo verso la ricerca e sviluppo, che costituisce il cuore della strategia aziendale. L'innovazione non è solo una componente delle attività quotidiane di SyncLab, ma è anche il motore che guida la crescita e la competitività dell'azienda nel settore ICT.

Come ho avuto modo di vedere personalmente, SyncLab investe infatti una quota significativa delle sue risorse in ricerca e sviluppo per esplorare nuove frontiere tecnologiche. L'azienda collabora attivamente con università, centri di ricerca e altre istituzioni accademiche per sfruttare le conoscenze più avanzate e per sviluppare soluzioni innovative che possano rispondere alle esigenze emergenti del mercato. Questi investimenti sono finalizzati non solo allo sviluppo di nuovi prodotti, ma anche al miglioramento continuo delle soluzioni esistenti.

SyncLab offre inoltre programmi di formazione continua e opportunità di sviluppo professionale ai suoi dipendenti per garantire che il proprio *team* sia sempre aggiornato sulle ultime tendenze tecnologiche.

Infine, a dimostrazione della propensione all'innovazione tecnologica dell'azienda, SyncLab vanta numerosi progetti di ricerca e sviluppo. Questi includono innovazioni in *blockchain*, IoT, *cloud computing*, *big data* e *analytics*, oltre a soluzioni avanzate di *cyber security* (vedi Figura 1.10).

Figura 1.10: Principali progetti di ricerca e sviluppo di SyncLab

Fonte: synclab.it

Capitolo 2

Stage proposto

2.1 Gestione aziendale degli *stage*

SyncLab gestisce gli *stage* aziendali attraverso un processo strutturato che coinvolge diverse fasi: la selezione dei candidati, l'inserimento in progetti concreti e formativi e la supervisione da parte del tutor aziendale.
Come già sottolineato nella sezione 1.4, l'azienda infatti offre opportunità di crescita e formazione, anche e soprattutto attraverso gli *stage* proposti, strutturati per lo sviluppo di competenze specifiche nell'ambito dell'integrazione di sistemi e dello sviluppo *software*. Gli *stage* sono pensati per favorire sia l'apprendimento teorico e pratico che l'integrazione nel *team* aziendale.

Selezione dei candidati

La selezione dei candidati per gli *stage* in SyncLab mira a identificare talenti con potenziale, con l'obiettivo di un possibile inserimento in azienda al termine dello *stage*.
Gli interessati possono presentare la loro candidatura attraverso la sezione "Lavora con noi" del sito *web* di SyncLab, inviando il proprio CV, oppure possono entrare in contatto con l'azienda tramite eventi come STAGE-IT, come ho fatto io.
Le candidature vengono poi valutate dall'azienda, che seleziona i profili più idonei in base a competenze ed esperienze. I candidati selezionati partecipano a colloqui individuali, al termine dei quali viene effettuata una valutazione complessiva. I

candidati scelti ricevono un'offerta di *stage* con dettagli sul progetto, il *team* di lavoro e le opportunità di sviluppo professionale in seguito allo *stage*.
Questo approccio assicura che SyncLab selezioni candidati con le giuste competenze e attitudine per contribuire efficacemente ai progetti aziendali e beneficiare al massimo dell'esperienza di *stage*.

Inserimento in progetti concreti e formativi

In SyncLab, gli stagisti vengono coinvolti in progetti concreti e formativi che permettono loro di applicare le conoscenze teoriche in contesti pratici e di sviluppare nuove competenze. Questi progetti sono attentamente selezionati per essere rappresentativi delle sfide reali che l'azienda affronta nei vari settori in cui opera.
Ogni stagista è integrato in un *team* di progetto, dove ha la possibilità di collaborare con professionisti esperti e di essere guidato da mentori dedicati. Questa collaborazione permette agli stagisti di acquisire una comprensione approfondita delle tecnologie e delle metodologie utilizzate da SyncLab, nonché di contribuire attivamente allo sviluppo di soluzioni innovative.
Alcuni esempi di progetti proposti quest'anno a STAGE-IT riguardavano uno studio base di intelligenza artificiale e *machine learning* per la realizzazione di un *chatbot* oppure l'implementazione di meccanismi di *data prediction* all'interno di piattaforme IoT, mentre altri progetti ancora includevano lo sviluppo di applicazioni *web* utilizzando *framework* moderni o la programmazione di architetture *cloud* e *serverless*.
Questo approccio garantisce che gli stagisti non solo migliorino le proprie competenze tecniche, ma sviluppino anche abilità di *problem solving* e lavoro di squadra, preparandoli efficacemente per future opportunità professionali all'interno di SyncLab. Inoltre gli *stage* in SyncLab servono all'azienda anche per conoscere nuove tecnologie e innovarsi.

Supervisione del tutor aziendale

La supervisione del tutor aziendale rappresenta un pilastro fondamentale dell'esperienza degli stagisti presso SyncLab. Ogni settimana quest'ultimo si impegna atti-

vamente a monitorare e guidare il progresso degli stagisti, garantendo un percorso di apprendimento efficace e coinvolgente.

Una delle principali responsabilità del tutor aziendale è, infatti, quella di condurre incontri regolari con gli stagisti al fine di valutare l'avanzamento delle attività assegnate e individuare eventuali difficoltà o aree di miglioramento. Attraverso questo processo di supervisione costante, il tutor è in grado di fornire un supporto personalizzato, offrendo consigli e suggerimenti mirati per superare gli ostacoli e massimizzare il potenziale di ciascuno stagista. Per facilitare questo passaggio, il tutor aziendale utilizza la bacheca di Trello associata allo stagista per monitorare lo stato delle attività che gli sono state assegnate. Nella Figura 2.1 ne è riportato un esempio.

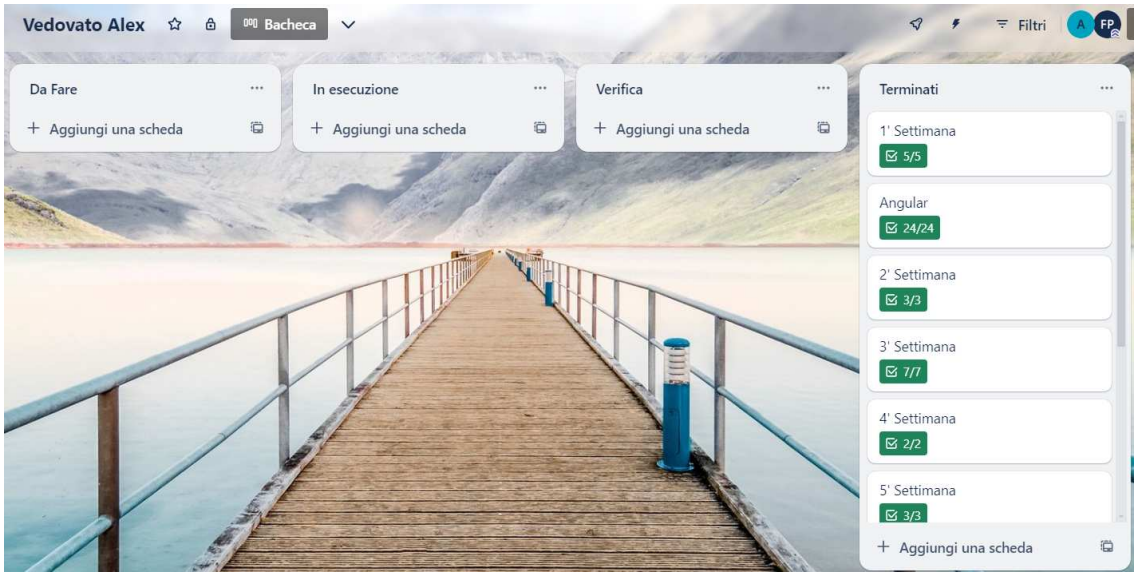


Figura 2.1: Esempio di bacheca di Trello

Infine, il tutor aziendale si impegna a creare un ambiente inclusivo e collaborativo in cui gli stagisti si sentono liberi di esprimere le proprie idee, porre domande e condividere le proprie preoccupazioni. Grazie alla sua esperienza e conoscenza del settore, il tutor offre un'orientamento prezioso e concreto, aiutando gli stagisti a sviluppare competenze professionali e ad acquisire una comprensione più approfondita delle dinamiche aziendali.

2.2 Il mio progetto di *stage*

Nel panorama tecnologico odierno, la crescita esponenziale delle tecnologie in ambito intelligenza artificiale ha trasformato radicalmente il modo in cui le aziende interagiscono con i loro clienti. In particolare, le piattaforme di RAG e gli LLM hanno aperto nuove opportunità per migliorare l'efficacia e l'efficienza delle interazioni automatizzate, permettendo la creazione di *chatbot* avanzati in grado di comprendere e rispondere alle richieste degli utenti in maniera sempre più naturale e contestuale. SyncLab si è resa conto di ciò ed ha voluto approfondire questo mondo attraverso diversi progetti di *stage*, incluso il mio. Durante lo svolgimento di quest'ultimo, infatti, ho avuto l'opportunità di condurre un'analisi dettagliata sulle attuali piattaforme di RAG, su alcuni LLM che rispettassero determinati vincoli e sulla loro integrazione in una *web application* tramite le relative API. L'obiettivo principale del progetto era quello di sviluppare competenze pratiche nell'implementazione di un *chatbot* specializzato, capace di comprendere e rispondere alle richieste degli utenti in contesti specifici, con un *focus* particolare sull'assistenza *fitness*. Inoltre ho anche implementato funzionalità aggiuntive nella *web app* per consentire la creazione automatica di piani di allenamento e piani alimentari personalizzati per gli utenti tramite l'interazione con la piattaforma di RAG sviluppata.

La parte iniziale del progetto ha comportato dunque uno studio approfondito dei *framework* Angular e Spring, necessari rispettivamente per lo sviluppo del *frontend* e del *backend* della *web app*, e del mondo LLM e RAG, necessario per lo sviluppo del *chatbot*. Successivamente, ho condotto un'analisi comparativa di alcune delle opzioni disponibili per lo sviluppo della piattaforma di RAG, processo che ha implicato la valutazione delle *performance* e dei risultati ottenuti utilizzando diversi LLM, al fine di identificare la soluzione più adatta per l'integrazione nell'applicazione *web* finale. Una volta terminate le attività di studio iniziale, corredate da diversi prototipi a dimostrazione delle nuove conoscenze apprese, sono potuto passare alle attività di implementazione pratica del progetto, le quali vengono approfondite all'interno del capitolo 3.

2.3 Obiettivi

Gli obiettivi del progetto di *stage* sono suddivisi in tre categorie: obbligatori, desiderabili e facoltativi. Gli obiettivi obbligatori rappresentano i requisiti minimi indispensabili per il successo del progetto, mentre quelli desiderabili e facoltativi aggiungono valore e miglioramenti ulteriori di diversa importanza, arricchendo l'esperienza dell'utente finale.

2.3.1 Obiettivi obbligatori

- Acquisizione di competenze approfondite sullo stato dell'arte delle piattaforme attuali di RAG/LLM, dimostrabili attraverso il prodotto finale ed i prototipi realizzati;
- Acquisizione di competenze riguardanti le tecnologie moderne per lo sviluppo di applicazioni *web*, in particolare Spring ed Angular, dimostrabili attraverso il prodotto finale ed i prototipi realizzati;
- Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il cronoprogramma;
- Integrazione tramite relative API di una piattaforma di RAG/LLM in una *fitness web app* con lo scopo di realizzare un *chatbot* in grado di fornire risposte adeguate alle richieste degli utenti, considerando anche il loro profilo personale.

2.3.2 Obiettivi facoltativi

- Implementazione di funzionalità aggiuntive nella *web app* per consentire la creazione automatica di piani di allenamento personalizzati per gli utenti tramite l'interazione con una piattaforma di RAG/LLM.

2.3.3 Obiettivi desiderabili

- Implementazione di funzionalità aggiuntive nella *web app* per consentire la creazione automatica di piani alimentari personalizzati per gli utenti tramire l'interazione con una piattaforma di RAG/LLM.

2.4 Vincoli

Durante lo svolgimento dello *stage*, ho dovuto rispettare diversi vincoli per garantire il raggiungimento degli obiettivi fissati e la corretta integrazione delle tecnologie studiate. Questi vincoli sono suddivisi in due categorie principali: tecnologici e di gestione.

2.4.1 Vincoli tecnologici

- **LLM a "costo zero"**: la scelta del miglior LLM da impiegare nella piattaforma di RAG sviluppata doveva avvenire utilizzando soluzioni che non comportassero costi aggiuntivi per l'azienda. Questo vincolo implicava l'utilizzo di versioni gratuite o di prova di LLM le cui API erano disponibili soltanto a pagamento, come ChatGPT o Gemini, oppure l'utilizzo di alternative *open-source*, come Phi-3 o Llama-3. Rimaneva comunque fondamentale scegliere opzioni che garantissero un livello sufficiente di *performance* e funzionalità per lo sviluppo ed il *testing* della piattaforma di RAG;

- ***Framework* e strumenti predefiniti**: ho sviluppato il progetto prevalentemente utilizzando i *framework* e gli strumenti stabiliti dall'azienda all'inizio dello *stage*. Ho impiegato Angular per il *frontend*, in modo da creare un'interfaccia utente dinamica e reattiva, mentre per il *backend* ho utilizzato Spring, il quale ha fornito una solida base per la gestione dei servizi e delle API. L'uso di questi *framework* ha assicurato che lo sviluppo fosse in linea con gli *standard* aziendali e ha facilitato la collaborazione e la revisione del codice da parte del *team* di SyncLab. L'utilizzo di altri *framework* o linguaggi non previsti doveva essere approvato dal tutor aziendale;

- **Ambiente di sviluppo controllato**: ho condotto tutte le attività di sviluppo all'interno di un ambiente controllato. Ho gestito il codice sorgente tramite *repository* centralizzati e sistemi di controllo delle versioni come Git, garantendo così tracciabilità e sicurezza. Ho utilizzato strumenti di gestione e sviluppo del progetto, inclusi *editor* come VS Code e piattaforme come StopLight per la progettazione delle API REST$_G$, sotto la supervisione del tutor aziendale. Questo approccio ha assicurato coerenza, sicurezza e un alto livello di qualità nel prodotto finale.

2.4.2 Vincoli di gestione

- **Rispetto del cronoprogramma**: le attività settimanali dovevano essere completate nei tempi previsti. Ogni settimana aveva infatti obiettivi specifici che dovevano essere raggiunti per garantire il progresso del progetto;

- **Incontri settimanali**: durante lo *stage* era necessario partecipare agli incontri settimanali con il tutor aziendale e gli altri *stakeholder* per discutere lo stato di avanzamento, chiarire eventuali dubbi e, se necessario, aggiornare il piano di lavoro;

- **Autonomia**: durante lo *stage* era richiesto di lavorare in maniera autonoma. Difatti, sebbene le decisioni operative e tecniche siano state prese in collaborazione con il tutor aziendale, ho dovuto mostrare iniziativa proponendo soluzioni e idee innovative. Ho inoltre dedicato una parte significativa del tempo a mia disposizione allo studio indipendente delle tecnologie e dei *framework* necessari per il progetto.

2.5 Motivazioni della scelta

La mia decisione di intraprendere lo *stage* presso SyncLab è motivata da diversi fattori che convergono verso un'unica visione di crescita professionale e personale. Innanzitutto l'azienda ha una reputazione consolidata nel settore dell'innovazione tecnologica e della trasformazione digitale, offrendo un ambiente stimolante e dinamico per lo sviluppo delle competenze. La vasta gamma di progetti e soluzioni proposti da SyncLab rappresentava dunque un'opportunità unica per confrontarsi con sfide stimolanti e per applicare le conoscenze accademiche in contesti reali.
Inoltre il progetto proposto ha suscitato grande interesse in me per la sua rilevanza nel contesto attuale dell'intelligenza artificiale. La possibilità di contribuire allo sviluppo di un *chatbot* specializzato per l'assistenza *fitness* ha rappresentato un'occasione entusiasmante per esplorare nuove tecnologie e acquisire competenze avanzate nel campo dello sviluppo *software* ampiamente spendibili anche nel mondo lavorativo. Per di più, la prospettiva di lavorare su un'applicazione pratica, che potrebbe avere un impatto reale sulla vita delle persone, ha reso questo progetto ancora più intrigante e significativo.
Dopo ancora, la possibilità di continuare a lavorare in azienda e di contribuire al loro sviluppo tecnologico è stato un ulteriore elemento che ha influenzato positivamente la mia scelta, offrendo la prospettiva di un futuro professionale presso SyncLab.
Infine la possibilità di lavorare su un progetto che richiedeva creatività, *problem-solving* e competenze tecniche avanzate è stata vista da me come un'opportunità per mettersi alla prova e crescere professionalmente.
Riassumendo brevemente, quindi, i miei obiettivi personali per il progetto di *stage* sono stati i seguenti:

- migliorare le mie capacità di comunicazione e collaborazione;
- crescere come sviluppatore *software* acquisendo nuove competenze tecniche;
- esplorare nuove tecnologie (Spring, Angular, LLM e RAG);
- contribuire allo sviluppo tecnologico dell'azienda.

Capitolo 3

Progetto di *stage*

3.1 Attività di studio iniziale

Come indicato precedentemente, la parte iniziale del progetto di *stage* prevedeva uno studio teorico approfondito dei *framework* e delle tecnologie a me sconosciute che avrei dovuto utilizzare successivamente. Questa sezione descrive le attività di studio e formazione svolte, compresi i prototipi sviluppati, e mette in mostra le conoscenze e i risultati ottenuti da queste attività.

3.1.1 Angular

Una volta ripassati i concetti base della gestione di progetto nella prima settimana, tra cui la metodologia *agile*, il *framework* Scrum, e i principi SOLID$_{G}$ e di *clean coding*$_{G}$, la prima nuova tecnologia che ho approfondito nel percorso di *stage* è stata il *framework* Angular.
Il primo passo nello studio è stato il ripasso e l'approfondimento delle basi di JavaScript e TypeScript, infatti essendo Angular costruito su TypeScript è cruciale comprendere le peculiarità di quest'ultimo. Fatto ciò ho potuto affrontare lo studio specifico del *framework* e successivamente realizzare una mini applicazione *web* prototipale utilizzandolo.

Struttura di un'applicazione in Angular

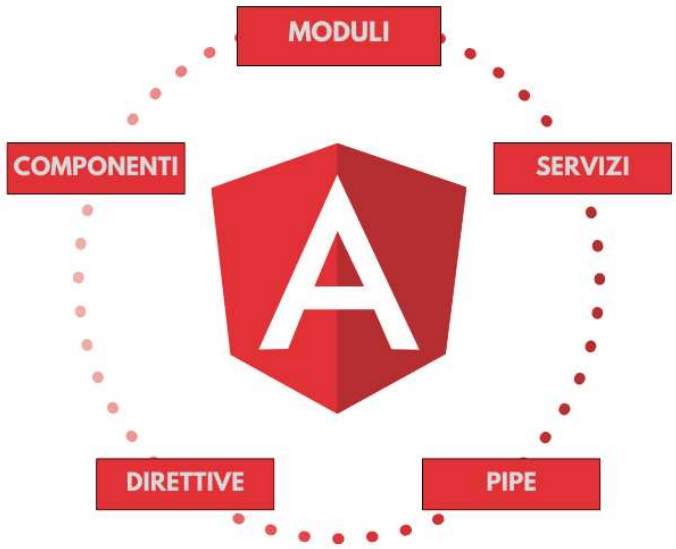


Figura 3.1: Architettura di un'applicazione sviluppata in Angular

Come riportato graficamente nella Figura 3.1, le applicazioni *web* basate su Angular sono progettate seguendo una struttura modulare che facilita l'organizzazione, la manutenzione e la scalabilità del codice. La struttura principale di un'applicazione Angular include i seguenti elementi:

- **Componenti**: sono le unità fondamentali dell'interfaccia utente in Angular. Ogni componente è composto da un *template* HTML, uno *stylesheet* per la definizione degli stili e una classe TypeScript che gestisce la logica del componente. I componenti vengono definiti utilizzando il decoratore `@Component`;
- **Servizi**: sono utilizzati per condividere dati e funzionalità tra diversi componenti. Un servizio è una classe TypeScript, decorata con `@Injectable`, che permette l'uso del *pattern dependency injection* per gestire le sue istanze;
- **Direttive**: sono utilizzate per manipolare il DOM e il comportamento degli elementi del *template* HTML. Ogni direttiva è definita utilizzando un decoratore `@Directive`;

- ***Pipe***: sono utilizzate per trasformare i dati nei *template* HTML. Esse prendono i dati in ingresso e li trasformano in un formato desiderato, senza alterare i dati originali. Ogni *pipe* è definita utilizzando un decoratore `@Pipe`;
- **Moduli**: utilizzati per raggruppare componenti, direttive, *pipe* e servizi correlati. Il modulo principale di un'applicazione Angular è l'`AppModule`, che funge da *entry point*$_G$ per l'applicazione. Ogni modulo è definito utilizzando un decoratore `@NgModule`.

La struttura modulare di Angular consente di mantenere il codice organizzato e scalabile. La separazione in moduli, componenti, servizi, direttive e *pipe* facilita lo sviluppo, la manutenzione e la riusabilità del codice, rendendo Angular un potente *framework* per lo sviluppo di applicazioni *web* moderne e complesse.

Dependency injection

La *dependency injection* è un *design pattern* fondamentale in Angular, che facilita la gestione delle dipendenze tra componenti, servizi e altri oggetti. La DI viene utilizzata per migliorare la modularità e la testabilità delle applicazioni, permettendo di iniettare dipendenze in una classe piuttosto che creare direttamente le istanze al suo interno.

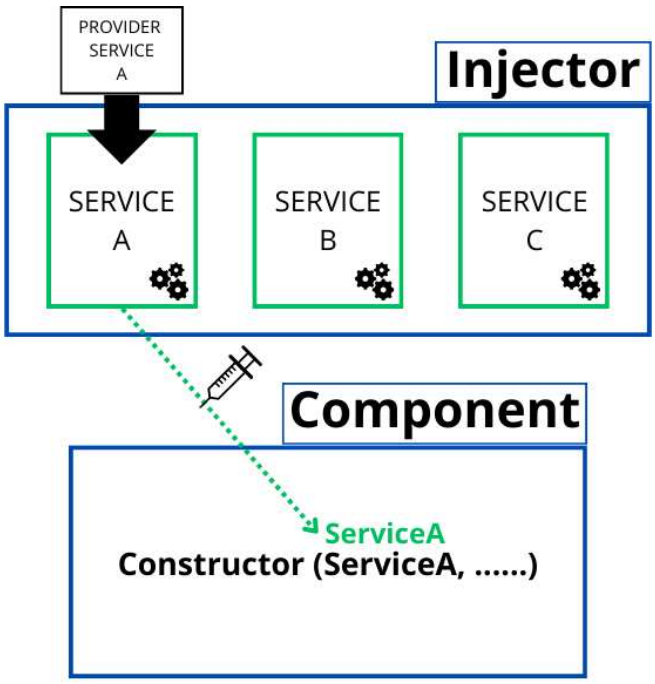


Figura 3.2: Rappresentazione della *dependency injection* in Angular

Come mostrato nella Figura 3.2, in Angular la DI è implementata attraverso un sistema di *provider* e *injector*. I *provider* specificano all'*injector* come creare una dipendenza, mentre gli *injector* sono responsabili della creazione delle istanze e dell'iniezione delle dipendenze richieste. Angular utilizza un sistema gerarchico di *injector*, il che significa che gli *injector* possono essere nidificati. Gli *injector* figli ereditano le dipendenze dagli *injector* genitori, ma possono anche sovrascrivere le dipendenze se necessario.
Per utilizzare un servizio in Angular occorre configurarlo per la DI utilizzando il decoratore `@Injectable`, mentre per iniettare un servizio in un componente basta dichiararlo come dipendenza nel costruttore di quest'ultimo.

Form e validazioni

I *form* sono uno degli elementi fondamentali delle applicazioni *web*, permettendo agli utenti di inserire e inviare dati. Angular offre due approcci principali per la gestione dei *form*: *template-driven form* e *reactive form*. Entrambi offrono potenti funzionalità di validazione, facilitando la gestione e il controllo dei dati degli utenti.
I *template-driven form* sono costruiti principalmente nei *template* HTML, utilizzando direttive Angular, e permettono di adottare un approccio semplice e intuitivo, ideale per *form* di piccole dimensioni e applicazioni meno complesse. Fondamentale è l'uso delle direttive `ngForm`, che crea un controllo di *form*, e `ngModel`, la quale stabilisce un *binding* tra il campo di *input* e il modello di dati. Utilizzandole è possibile eseguire automaticamente la validazione dei campi e mostare eventuali messaggi di errore.
I *reactive form* invece offrono un controllo più dettagliato sulla gestione dei *form* e sono ideali per applicazioni complesse con validazioni avanzate. Per utilizzare i *reactive form* è necessario importare il modulo `ReactiveFormsModule` e costruire un *form* attraverso un'istanza di `FormGroup`, contente al suo interno una serie di `FormControl`, oppure attraverso un'istanza di `FormBuilder`. A questo punto le validazioni vengono eseguite utilizzando i *validator* di Angular associati ai vari `FormControl`. Quest'ultimi possono essere *built-in*$_G$, ovvero definiti di *default*, ma Angular consente anche la creazione di validatori personalizzati per soddisfare requisiti specifici.

Prototipo realizzato

Per poter dare dimostrazione di tutti i concetti elencati, ho applicato le conoscenze teoriche acquisite sviluppando una mini applicazione *web* prototipale utilizzando Angular, di cui ho riportato alcune immagini all'interno delle Figura 3.3. All'interno del prototipo si trovano funzionalità per la gestione degli atleti e dei macchinari di una palestra, con diverse viste per la visualizzazione, la modifica, l'aggiunta e la cancellazione dei dati. Inoltre, in ottica del progetto vero e proprio, ho dedicato una piccola parte del tempo a mia disposizione per esplorare due prototipi di grafica per un *chatbot* in Angular, uno sviluppato con Angular Material, l'altro integrando Bootstrap, entrambi osservabili nella Figura 3.4.

All'interno del prototipo ho utilizzato sia i *reactive form*, per quanto riguarda la gestione dei dati dei macchinari, che i *template-driven form*, per quanto riguarda la gestione dei dati degli atleti. Inoltre, essendo la *web app* sviluppata composta da più viste, ho configurato il *routing* per navigare tra le diverse sezioni dell'applicazione.

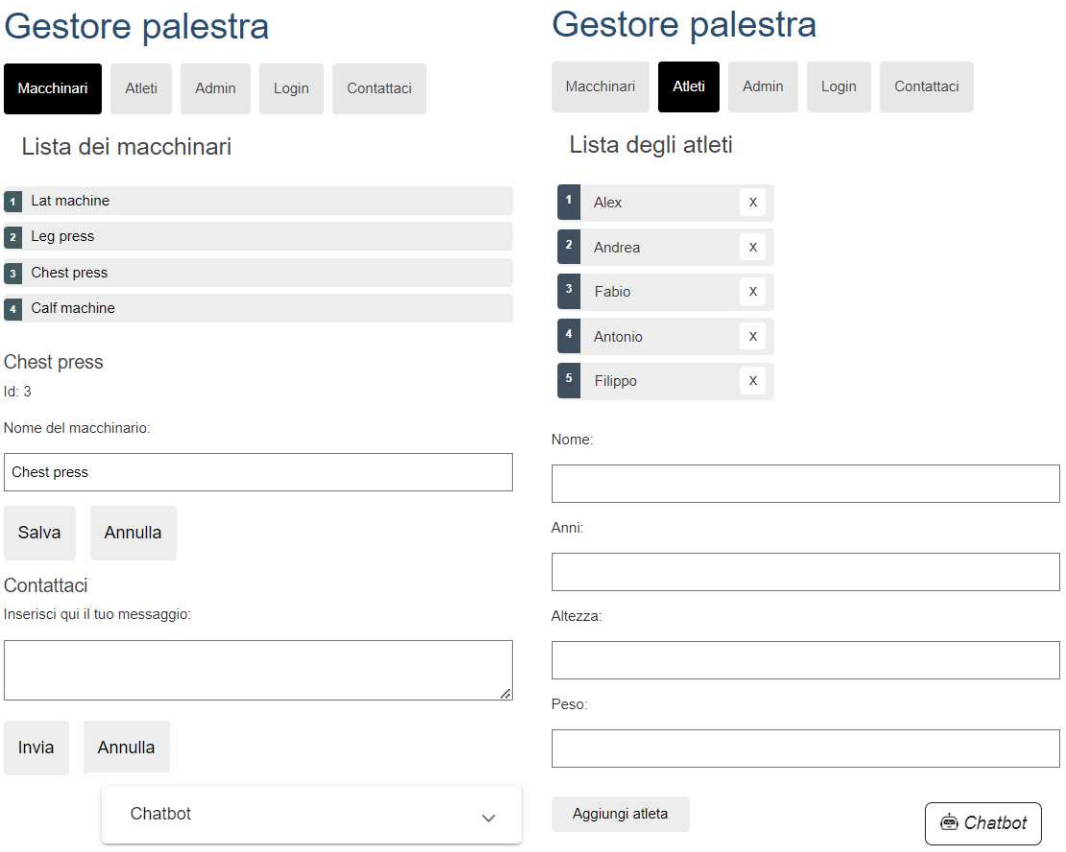


Figura 3.3: Alcune viste del prototipo realizzato in Angular

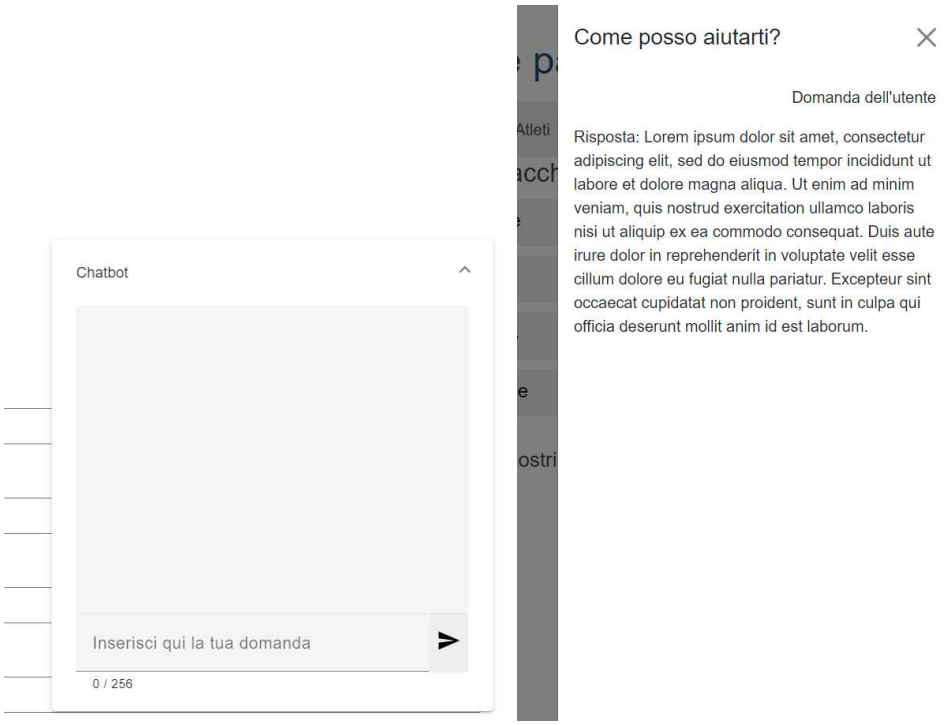


Figura 3.4: I due prototipi di grafica per il *chatbot* sviluppati in Angular

3.1.2 Spring

Terminato lo studio del *framework* Angular, ho successivamente esplorato l'ecosistema Spring, di cui si può osservare una panoramica all'interno della Figura 3.5.

Mentre per Angular mi sono affidato principalmente alla documentazione presente nel sito ufficiale, l'ecosistema Spring, data la sua ampiezza e complessità, ha richiesto un approccio differente. Infatti, per affrontare lo studio in modo efficace, ho utilizzato una combinazione di risorse tra cui la documentazione ufficiale, corsi *online*, video su Youtube e guide pratiche, iniziando con una panoramica generale di Spring, per comprendere le sue componenti fondamentali e come esse interagiscono tra loro, per poi approfondire nel dettaglio i vari moduli che mi sarebbero tornati utili all'interno del progetto.

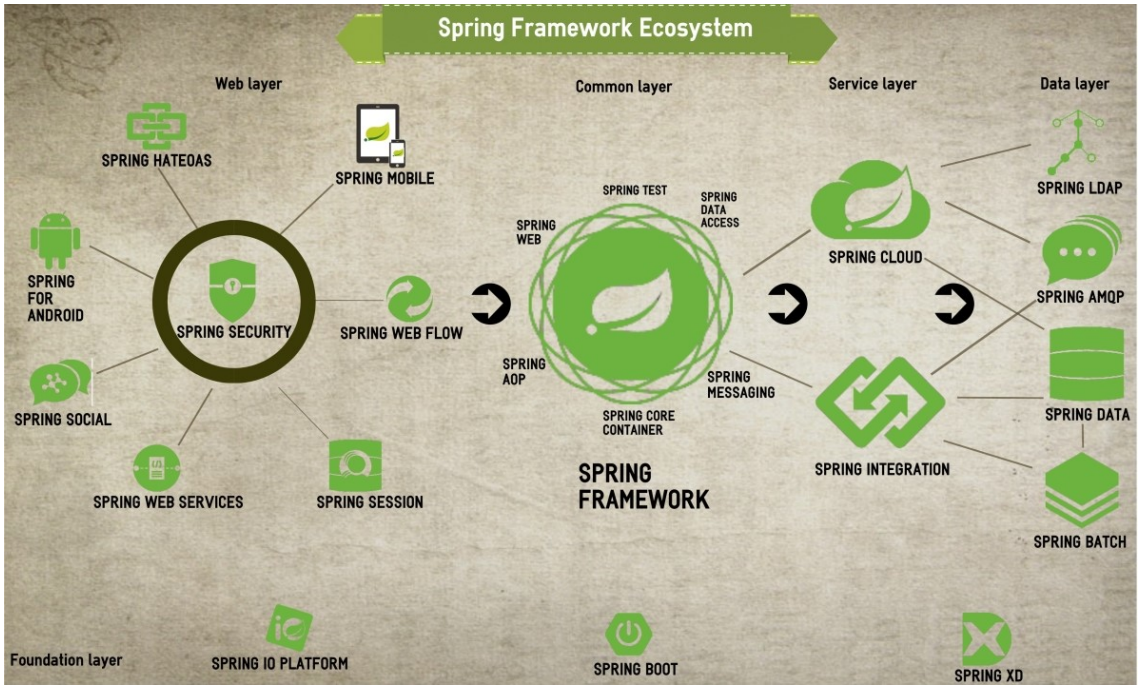


Figura 3.5: Rappresentazione grafica dell'ampio ecosistema Spring

Fonte: springtutorials.com

Spring Core

Il primo passo è stato lo studio di Spring Core, il cuore del *framework*, che fornisce funzionalità essenziali come *dependency injection* e *inversion of control*, concetti fondamentali per capire come Spring facilita la gestione delle dipendenze e la configurazione delle applicazioni.

IoC è implementato attraverso il cosiddetto contenitore IoC, responsabile della creazione, configurazione e gestione del ciclo di vita degli oggetti dell'applicazione, chiamati *bean* all'interno del contesto Spring. Quest'ultimi vengono definiti tramite configurazione XML, annotazioni o JavaConfig, e, una volta fatto ciò, quando l'`ApplicationContext` viene inizializzato, i *bean* definiti vengono creati, configurati e gestiti fino alla loro distruzione.

Le dipendenze tra i *bean* vengono risolte e iniettate automaticamente sempre dal contenitore IoC. Questo è possibile grazie all'uso del *design pattern dependecy injection*, il quale è implementato in Spring secondo le tre classiche modalità, ovvero *constructor injection*, *setter injection* e *field injection*.

Nella Figura 3.6 si può osservare come il contenitore IoC utilizzi i metadati di configurazione e gli oggetti di *business* per creare un'applicazione completamente configurata e pronta per essere utilizzata.

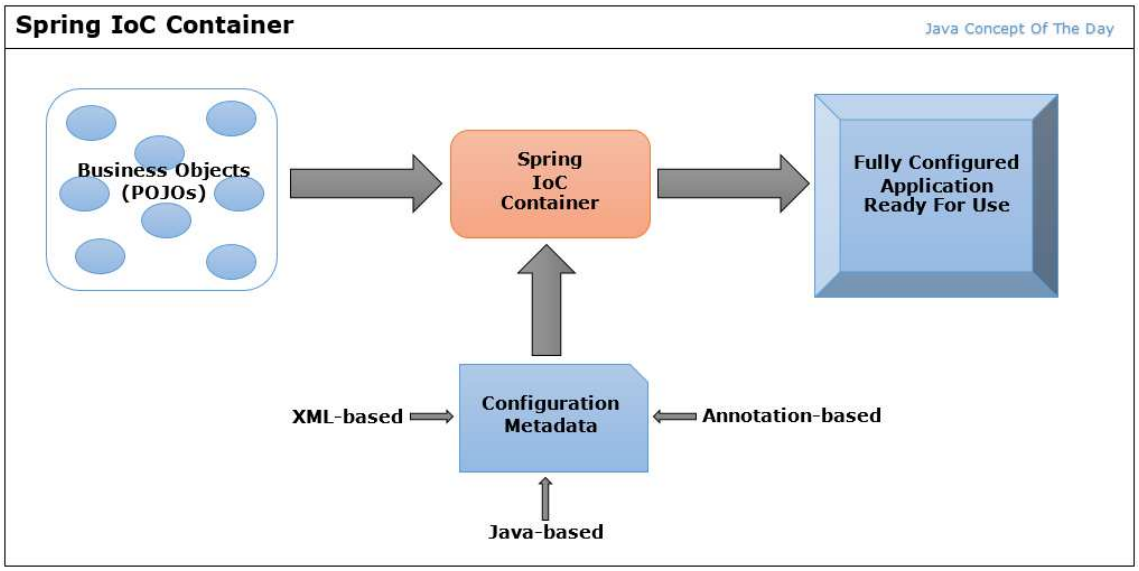


Figura 3.6: Rappresentazione grafica del funzionamento dell'IoC *container* di Spring

Fonte: javaconceptoftheday.com

Spring Boot

Successivamente mi sono concentrato su Spring Boot, un modulo all'interno dell'ecosistema Spring che rende l'uso del *framework* molto più semplice e veloce, eliminando gran parte della configurazione manuale. Esso permette di creare rapidamente applicazioni *standalone*$_G$, *production-ready*$_G$, con una configurazione minima. Spring Boot offre infatti numerose convenzioni predefinite che consentono agli sviluppatori di iniziare a lavorare immediatamente, abbassando il tempo necessario per configurare un'applicazione da zero, dato che riduce la necessità di *file* di configurazione XML o annotazioni complesse. Inoltre permette di eseguire applicazioni *standalone* con un semplice comando, includendo un *server web* integrato come Tomcat, Jetty o Undertow, e offre funzionalità come monitoraggio, *logging*, gestione della configurazione e sicurezza.

Una tipica applicazione Spring Boot è organizzata in un modo che segue le *best practice* del *framework* Spring, con l'aggiunta di alcune convenzioni:

- il *file* `"pom.xml"`, per Maven, o `"build.gradle"`, per Gradle, include le dipendenze necessarie per far funzionare correttamente l'applicazione;
- la classe principale dell'applicazione è annotata con `@SpringBootApplication`, che è una combinazione delle seguenti annotazioni: `@Configuration`, che indica che la classe può essere utilizzata per la configurazione di Spring, `@EnableAutoConfiguration`, la quale specifica a Spring Boot di iniziare ad aggiungere *bean* basati su impostazioni specificate nelle dipendenze, e `@ComponentScan`, che richiede a Spring di scansionare il pacchetto corrente per trovare componenti, configurazioni e servizi;
- i *controller* gestiscono le richieste HTTP e sono annotati con `@RestController`;
- i servizi contengono la logica di *business* e sono annotati con `@Service`;
- le classi *repository* interagiscono con i *database* e sono annotate con `@Repository`;
- i modelli rappresentano le entità del *database* e sono annotati con `@Entity`.

Nella Figura 3.7 si può vedere come le componenti dell'architettura di una tipica applicazione Spring Boot interagiscono tra loro.

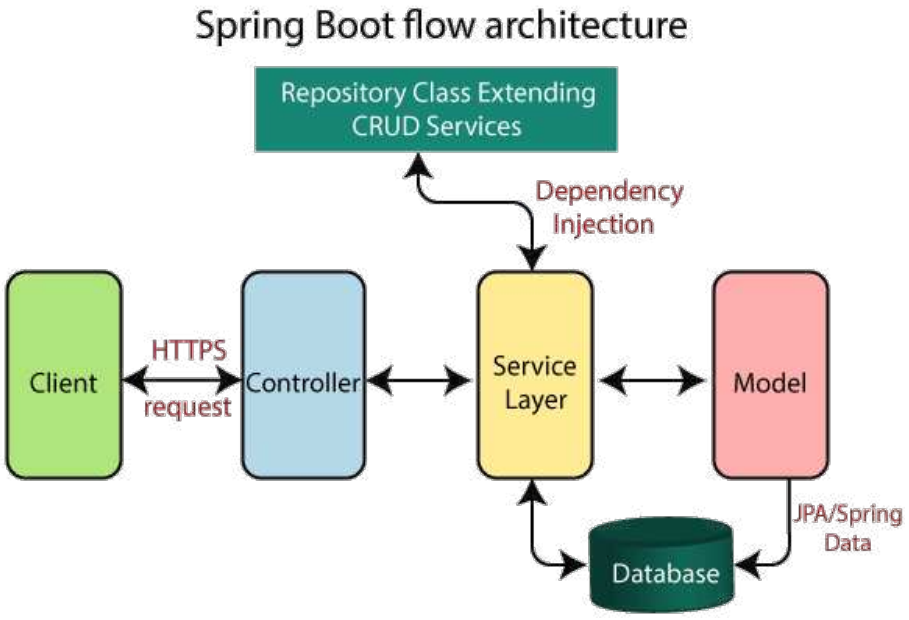


Figura 3.7: Architettura di una tipica applicazione Spring Boot

Fonte: javatpoint.com

Infine Spring Boot utilizza un meccanismo di auto-configurazione per creare automaticamente i *bean* necessari in base alle dipendenze incluse e permette di gestire le proprietà di configurazione dell'applicazione nel *file* `"application.properties"` o `"application.yml"`.

Spring MVC

A seguire ho esaminato i concetti chiave di Spring MVC, modulo per la creazione di applicazioni *web* basate sull'architettura *Model-View-Controller*, consentendo una chiara separazione delle responsabilità all'interno dell'applicazione.
Il cuore di Spring MVC è il `DispatcherServlet`, il quale intercetta le richieste HTTP e le instrada verso i *controller* appropriati per l'elaborazione. Quest'ultimi gestiscono le richieste, elaborano i dati di *input*, interagiscono con il *Model*, il quale contiene i dati dell'applicazione e la logica di *business*, e rispondono alla richieste in maniera appropriata.
Spring MVC offre diverse annotazioni per gestire le richieste HTTP:

- **`@RequestMapping`**: mappa le richieste su metodi specifici di un *controller*;
- **`@GetMapping`**: gestisce le richieste HTTP *GET*;
- **`@PostMapping`**: gestisce le richieste HTTP *POST*;
- **`@PutMapping`**: gestisce le richieste HTTP *PUT*;
- **`@DeleteMapping`**: gestisce le richieste HTTP *DELETE*.

Inoltre Spring MVC supporta la validazione dei dati di *input* utilizzando annotazioni specifiche che possono essere applicate direttamente ai campi delle classi dei modelli per definire le regole di validazione. Alcune delle annotazioni più comuni sono `@NotNull`, che assicura che il campo non sia *null*, `@Size`, che controlla che la lunghezza di una stringa o la dimensione di una collezione sia entro i limiti specificati, `@Min` e `@Max`, che specificano i valori minimo e massimo per un numero, e `@Email`, che controlla che il campo contenga un indirizzo email valido. Oltre a queste e molte altre, è possibile definire anche validazioni *custom* implementando un *validator* personalizzato.

Infine Spring MVC supporta diverse tecnologie di visualizzazione, consentendo agli sviluppatori di scegliere quella più adatta alle loro esigenze. Ad esempio, esso può essere integrato con *framework* di *frontend* moderni come Angular, React e Vue.js per creare applicazioni SPA. In questi casi Spring MVC agisce come un *backend* API che serve dati JSON ai *client frontend*.

Spring Data JPA

Dopo ancora, un'altra tecnologia chiave dell'ecosistema che ho avuto modo di approfondire è stata Spring Data JPA, che semplifica l'interazione con i *database*. Il principale obiettivo di questo modulo è ridurre il numero di righe di codice necessarie per implementare le operazioni di accesso ai dati e fornire un'infrastruttura robusta e facilmente estendibile per la persistenza dei dati. Esso automatizza la generazione del codice necessario per le operazioni CRUD di base e permette di definire *query* personalizzate utilizzando il linguaggio JPQL (Java *Persistence Query Language*)$_G$ o direttamente in SQL.

Per utilizzare Spring Data JPA è necessario includere le dipendenze appropriate nel *file* `"pom.xml"`, per Maven, o `"build.gradle"`, per Gradle, e configurare i dettagli del *database* nel *file* `"application.properties"` o `"application.yml"`.

Una caratteristica che contraddistingue Spring Data JPA è la gerarchia dei *repository* (vedi Figura 3.8), la quale è costruita su diverse interfacce che estendono funzionalità specifiche: l'interfaccia di base, `CrudRepository`, fornisce metodi di CRUD di base, `PagingAndSortingRepository` estende `CrudRepository` e aggiunge metodi per la paginazione e l'ordinamento, mentre `JpaRepository` estende `PagingAndSortingRepository` e aggiunge metodi specifici di JPA. Per utilizzare questi *repository* è sufficiente estendere l'interfaccia appropriata nell'applicazione sviluppata.

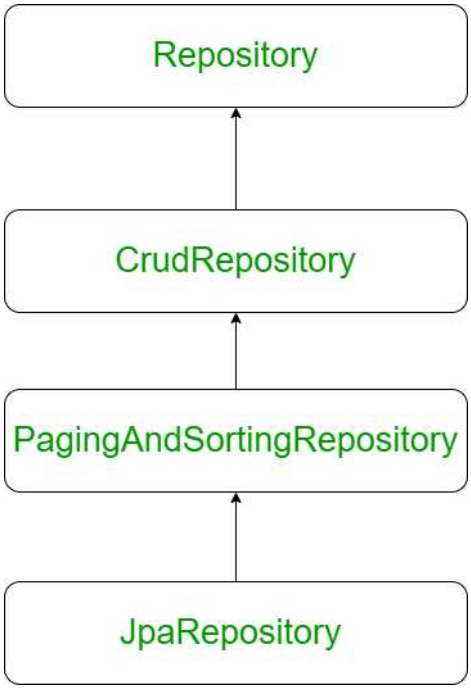


Figura 3.8: Gerarchia dei *repository* di Spring Data JPA

Fonte: geeksforgeeks.org

Infine Spring Data JPA permette di definire *query methods*$_G$ seguendo convenzioni di denominazione specifiche: il nome del metodo determina la *query* che verrà eseguita.

Prototipo realizzato

A seguito dell'apprendimento teorico, per dimostrare la mia comprensione dei concetti trattati e sperimentare, ho sviluppato un prototipo di servizio REST completo di metodi *GET*, *POST*, *PUT* e *DELETE*. Questo prototipo utilizza tutte le conoscenze acquisite per implementare una serie di *endpoint* che consentono la gestione dei dati relativi a un gruppo di atleti.

Il servizio REST sviluppato permette di recuperare l'elenco completo degli atleti, ottenere i dettagli di un singolo atleta tramite il suo ID, aggiungere un nuovo atleta, aggiornare le informazioni di un atleta esistente e eliminare un atleta dall'elenco. Ho implementato ogni operazione seguendo le *best practice* di progettazione REST e utilizzando le annotazioni proprie di Spring.

Questo esercizio pratico mi ha permesso non solo di consolidare le mie conoscenze teoriche, ma anche di acquisire esperienza concreta nello sviluppo di servizi REST utilizzando tecnologie moderne e metodologie robuste.

3.1.3 LLM e RAG

Completato l'approfondimento relativo ai *framework* necessari per lo sviluppo della *web app*, il passo successivo per concludere le attività iniziali di studio dello *stage* è stato approfondire il fulcro del progetto: l'integrazione e l'utilizzo di LLM e RAG. Ho iniziato con una rapida revisione teorica basata su diverse fonti e successivamente ho condotto un'analisi più approfondita delle principali piattaforme disponibili e delle loro API. Infine, ho applicato le conoscenze acquisite attraverso sperimentazioni pratiche per testare e consolidare le competenze apprese.

***Large Language Model*: la base**

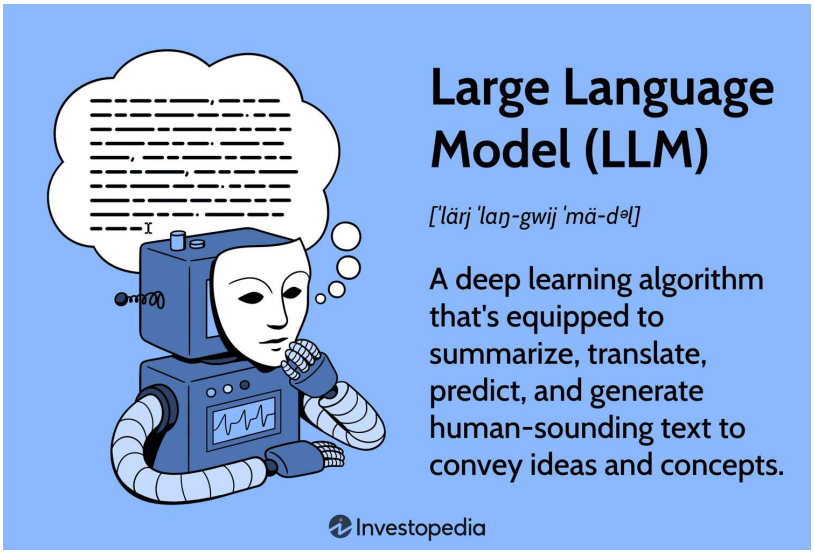


Figura 3.9: Definizione di *Large Language Model*
Fonte: investopedia.com

Come suggerito nella Figura 3.9, i *Large Language Model* sono modelli di *machine learning* addestrati su enormi quantità di dati testuali con lo scopo di comprendere e generare linguaggio naturale. Questi modelli utilizzano tecniche avanzate di *deep learning*, in particolare le reti neurali trasformative, per prevedere la probabilità delle sequenze di parole, rispondere a domande, tradurre testi, riassumere documenti e svolgere altre attività di elaborazione del linguaggio naturale.
L'addestramento dei LLM richiede enormi risorse computazionali e grandi quantità di dati, con costi di conseguenza molto elevati. I modelli vengono pre-addestrati su una vasta gamma di testi, tra cui libri, articoli, siti *web* e altro, per acquisire una

comprensione generale del linguaggio, e successivamente, possono essere ulteriormente addestrati (*fine-tuned*$_G$) su *dataset* specifici per compiti particolari, migliorando le loro prestazioni in contesti specifici.

Retrieval-Augmented Generation: l'evoluzione

Per migliorare le prestazioni dei LLM e superare numerose limitazioni, si può sfruttare la *Retrieval-Augmented Generation*. Essa è una tecnica avanzata che combina il recupero di informazioni con la generazione di testo per migliorare la qualità e la pertinenza delle risposte prodotte dai LLM. A differenza dei tradizionali modelli linguistici, i quali si basano esclusivamente sui dati di addestramento, la RAG integra un componente di recupero di informazioni che consente al modello di accedere a fonti di conoscenza contenute in un *database* esterno durante la fase di generazione. L'architettura di una piattaforma di RAG, come si può osservare nella Figura 3.10, si compone solitamente di due parti logiche principali: *retriever* e *generator*. Il *retriever* è la parte del sistema che si occupa di cercare le informazioni rilevanti all'interno di un vasto *corpus* di dati in risposta a una richiesta, cosa che viene fatta affidandosi a tecniche di ricerca basate su similarità semantica per identificare le informazioni più pertinenti, mentre il *generator* elabora il testo recuperato e produce una risposta coerente.

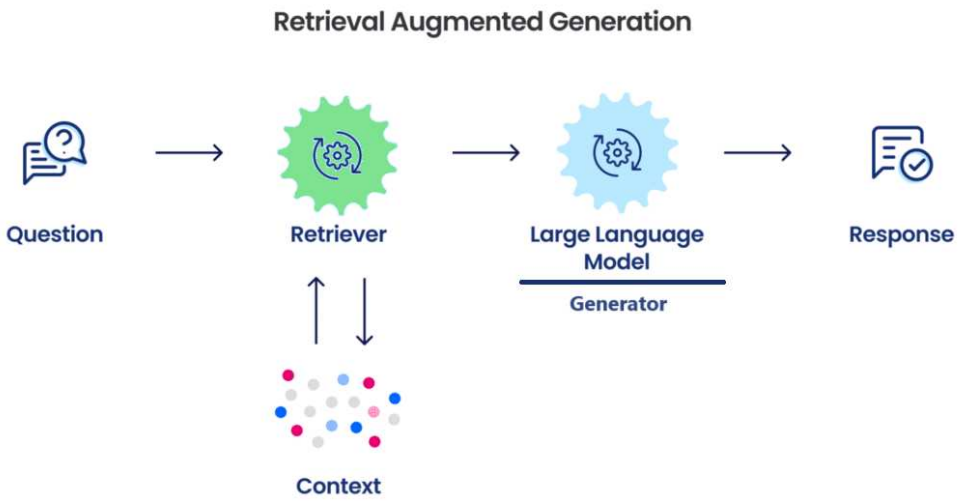


Figura 3.10: Architettura logica di un sistema di RAG

Funzionamento della RAG

Il funzionamento della RAG può essere suddiviso in più fasi:

- **Reperimento di dati esterni**: per prima cosa è fondamentale raccogliere dati rilevanti da fonti esterne o produrli autonomamente, in modo da poter arricchire il modello linguistico con la conoscenza desiderata. Questi dati possono includere articoli, documenti, siti *web*, *file* di testo, e altri ancora;
- **Inserimento dei dati in un *database* vettoriale**: i dati raccolti vengono suddivisi in unità più piccole e facilmente gestibili, dette *chunk*, cercando di fare in modo che abbiano senso compiuto. I *chunk* vengono poi convertiti in rappresentazioni vettoriali utilizzando tecniche di *embedding* e memorizzati in un *database* vettoriale. Questo processo facilita in seguito il recupero rapido delle informazioni rilevanti;
- **Recupero di informazioni pertinenti**: quando viene effettuata una *query* da parte dell'utente, il sistema utilizza il *database* vettoriale per cercare e recuperare i dati più pertinenti rispetto alla richiesta. Questo passaggio coinvolge l'uso di tecniche di *nearest neighbor search*$_G$ per trovare i vettori nel *database* che sono più simili al vettore della *query*. In questa maniera le informazioni recuperate sono strettamente correlate al contesto della *query*, migliorando la pertinenza e l'accuratezza delle risposte;
- **Creazione del *prompt* finale**: le informazioni recuperate dal *database* vettoriale vengono poi combinate con la *query* iniziale dell'utente per creare un *prompt* finale. Questo è progettato per fornire al LLM un contesto ricco e informativo per generare una risposta accurata;
- **Ottenimento della risposta**: il *prompt* finale viene inviato al LLM, che genera una risposta utilizzando sia le informazioni recuperate che il contesto della *query* originale. Questa risposta viene poi presentata all'utente.

Vantaggi della RAG rispetto all'uso diretto dei LLM

Sebbene i LLM siano incredibilmente potenti e capaci di generare testo in modo naturale e convincente, l'integrazione con la RAG offre numerosi vantaggi che ne fanno una soluzione superiore per molte applicazioni pratiche.
Innanzitutto, la RAG migliora l'accuratezza delle risposte. I LLM, nonostante la loro vastissima capacità di generare testo, sono limitati alle informazioni disponibili al momento del loro addestramento e possono quindi generare risposte basate su dati obsoleti o imprecisi, mentre la RAG permette di integrare dati aggiornati e specifici al contesto, migliorando notevolmente l'accuratezza delle risposte fornite.
Inoltre, la RAG riduce significativamente il problema della allucinazioni, ovvero la generazione di informazioni plausibili ma completamente inventate. Questo accade perché i LLM cercano di prevedere la sequenza di parole più probabile, senza necessariamente avere accesso a dati fattuali, ma la RAG riduce questo problema recuperando informazioni concrete e verificabili da fonti esterne, migliorando l'affidabilità delle risposte.
Dopo ancora, mentre l'addestramento e l'aggiornamento di un LLM possono richiedere risorse computazionali significative, la RAG consente di mantenere un *database* esterno aggiornato senza necessità di riaddestrare continuamente il modello. Questo approccio riduce i costi e facilita l'aggiornamento delle informazioni disponibili.
Infine con la RAG è possibile personalizzare le risposte e utilizzare e proteggere meglio i dati sensibili, recuperando informazioni da *database* sicuri e controllati, senza necessità di includere tali dati direttamente nel processo di addestramento del LLM.

Stato dell'arte

Il campo riguardante LLM e RAG è in continua evoluzione, con nuove ricerche e applicazioni che emergono continuamente. Anche solo durante il corso dello *stage* questo è mutato notevolmente e ho potuto osservare diversi progressi significativi e nuove implementazioni che hanno arricchito le possibilità offerte da queste tecnologie.
I LLM continuano a migliorare in termini di capacità e *fine-tuning*, con modelli sempre più grandi e complessi, con un numero sempre crescente di parametri, i

quali riescono a comprendere e generare linguaggio naturale in maniera sempre più accurata. Ad esempio, nuove versioni di modelli esistenti come GPT-4o, così come modelli sviluppati da altre organizzazioni, hanno spinto ulteriormente i confini di ciò che è possibile fare con l'IA nel campo del linguaggio naturale.
Parallelamente, la ricerca sulla RAG ha portato a tecniche sempre più sofisticate per combinare il recupero di informazioni con la generazione di testo. Ma soprattutto, le piattaforme di intelligenza artificiale hanno iniziato a integrare questa tecnologia nei loro servizi, rendendo più facile per gli sviluppatori implementare soluzioni avanzate senza dover partire da zero, come invece ho fatto io.

Principali piattaforme esplorate a confronto

Durante le attività di studio, ho esaminato diverse piattaforme di intelligenza artificiale che rispettassero i vincoli imposti dal progetto. Poiché lo scopo dello *stage* non era quello di effettuare un confronto esaustivo tra numerosi LLM, mi sono concentrato esclusivamente sulla prova di alcuni modelli selezionati, sia locali che remoti, individuandone debolezze e punti di forza, in modo da poter poi scegliere la soluzione più adatta tra quelle sperimentate ed integrarla nel prodotto finale. Per questo motivo, ho scelto di approfondire lo studio delle seguenti piattaforme:

- **Gemini 1.0 Pro**: modello multimodale sviluppato da Google, appartenente alla famiglia Gemini. Accessibile tramite API, esegue remotamente e si distingue per la sua versatilità e capacità di eccellere in un'ampia gamma di attività di elaborazione del linguaggio naturale, specialmente per quanto riguarda la generazione di testo coerente e fluido e una buona gestione del contesto. Tuttavia, presenta tempi di risposta medi, che lo rendono meno adatto per applicazioni che richiedono bassa latenza. Non è *open-source* ma si può utilizzare gratuitamente tramite il credito di prova di Google Cloud Platform$_G$;

- **Gemini 1.5 Flash**: una versione aggiornata e migliorata di Gemini 1.0 Pro, con prestazioni significativamente più elevate. Possiede le stesse caratterische principali del modello precedentemente trattato, ma eccelle nella gestione di conversazioni complesse e nella generazione di testi dettagliati e accurati, battendo Gemini 1.0 Pro in quasi tutti i *benchmark* pubblici. Inoltre, la velocità di risposta è stata notevolmente migliorata, rendendola ideale per applicazioni in tempo reale, e la dimensione della finestra di contesto è stata aumentata enormemente, raggiungendo un milione di *token*$_G$, la più grande nel mercato dei LLM;

- **Phi-3 Mini**: modello sviluppato da Meta con 3,8 miliardi di parametri e una finestra di contesto di 128 mila *token*, appartenente alla famiglia Phi-3. Nonostante le sue dimensioni ridotte, Phi-3 Mini si confronta favorevolmente con LLM più grandi in termini di accuratezza e prestazioni e richiede una potenza di calcolo ridotta rispetto ad altri modelli, rendendolo più accessibile e adatto per l'utilizzo su dispositivi con risorse limitate. Esso esegue infatti localmente ed è utilizzabile gratuitamente essendo *open-source*;

- **Llama-3**: creato da Meta, prevede due modelli di intelligenza artificiale *open-source*: uno con 8 miliardi di parametri e l'altro con 70. Entrambi hanno una finestra di contesto abbastanza piccola, ovvero soltanto di 8 mila *token*, ma questo non impedisce comunque a Llama 3 di posizionarsi come uno dei LLM più capaci e accessibili disponibili pubblicamente. Offre prestazioni all'avanguardia, nonostante esegua localmente, ma richiede una maggiore potenza di calcolo rispetto a modelli più piccoli come Phi-3 Mini;

- **All-minilm:l6-v2**: modello di *embedding open-source* sviluppato da Hugging Face, noto per la sua efficienza computazionale e prestazioni elevate in diverse attività di elaborazione del linguaggio naturale, nonostante le sue dimensioni ridotte. Esegue localmente ma richiede poca memoria e potenza di calcolo, rendendolo ideale per dispositivi con risorse limitate. Può elaborare il testo rapidamente, consentendo un'interazione fluida e reattiva, e, sebbene i suoi risultati non siano chiaramente i migliori nel mercato, è un ottimo compromesso;

- **Textembedding-gecko@003**: modello specializzato nell'*embedding* di testo, offerto da Google attraverso Vertex AI, una piattaforma che fornisce un'ampia gamma di strumenti e servizi di intelligenza artificiale. Accessibile tramite API, questo modello opera in remoto ed è particolarmente efficace nelle attività di ricerca semantica. Grazie alle sue prestazioni elevate, Textembedding-gecko@003 garantisce *embedding* di alta qualità in tempi molto rapidi, rappresentando una soluzione efficiente e accessibile. Non è *open-source* ma si può utilizzare gratuitamente tramite il credito di prova di Google Cloud Platform.

Va sottolineato che tutti i modelli da me esplorati sono compatibili con LangChain, garantendo una facilità d'uso uniforme. È inoltre cruciale ricordare che nessun LLM è perfetto: ciascun modello possiede punti di forza e debolezza unici. Pertanto, la scelta del modello più adatto dipende in gran parte dalle esigenze specifiche dell'applicazione. Per questo motivo, allo scopo di individuare la soluzione ottimale per il progetto tra quelle citate precedentemente, nel prototipo realizzato ho definito una serie di *test* mirati a confrontare i migliori LLM selezionati, al fine di valutare le loro prestazioni in contesti specifici e identificare quello più adatto alle esigenze del progetto.

Prototipo realizzato

Dopo aver acquisito una solida base teorica, ho avviato una serie di esperimenti pratici mediante la definizione di piccoli *script* di prova finalizzati all'invocazione diretta delle API di vari LLM, tra cui Gemini 1.0 Pro, Gemini 1.5 Flash e Phi-3 Mini, quest'ultimo installato localmente sulla mia macchina tramite Ollama$_G$. Fin dai primi *test* è emersa la superiorità dei modelli di Google, evidenziata sia nelle *performance* che nella qualità delle risposte generate. Tuttavia, volendo approfondire ulteriormente le potenzialità di questi modelli, ho scelto di utilizzare LangChain per sviluppare un prototipo di sistema di RAG, variando poi i LLM alla base di quest'ultimo e verificando con dei *test* pratici l'efficacia e l'efficienza del sistema di RAG risultante. Quest'ultimo, indipendentemente dai modelli impiegati, utilizza dei documenti PDF di prova, effettua un *chunking* a dimensione fissa dei contenuti di quest'ultimi e immagazzina gli *embedding* generati dal LLM scelto in

un *database* vettoriale ChromaDB. Una volta fatto ciò, permette poi l'esecuzione di una serie di *query* da parte dell'utente, le quali vengono elaborate seguendo le classiche fasi di recupero delle informazioni pertinenti, creazione del *prompt* finale e ottenimento della risposta utilizzando uno specifico LLM.
Il sistema, per quanto prototipale, mi ha permesso di eseguire una serie di confronti mantenendo la stessa semplice metodologia di *chunking*, di recupero delle informazioni pertinenti e di creazione del *prompt* finale, in modo da garantire una base uniforme per valutare le prestazioni dei diversi modelli. Inoltre, ho testato ogni LLM con le stesse *query* per assicurare la coerenza dei risultati e permettere un confronto equo.
I *test* che ho eseguito e i relativi risultati comprendono i seguenti aspetti:

- **Velocità di elaborazione**: ho eseguito *test* mirati a valutare il tempo richiesto per l'*embedding*, la risposta all'utente oppure l'esecuzione dell'intera catena. Come ci si poteva facilmente aspettare, i modelli remoti si sono dimostrati nettamente più rapidi rispetto a quelli eseguiti localmente, grazie alle infrastrutture di calcolo avanzate disponibili nei *server cloud*. I modelli di Google richiedevano soltanto pochi secondi per eseguire le varie operazioni, mentre quelli *open-source* locali necessitavano di diversi minuti, anche per via della limitata potenza di calcolo della mia macchina;

- **Correttezza e qualità delle risposte ottenute**: per valutare le risposte generate dai vari LLM a specifiche *query* ho utilizzato due tecniche differenti. Innanzitutto, ho impiegato Gemini 1.5 Flash come giudice, essendo il più rapido tra i LLM testati, sfruttando una metodologia di *testing* chiamata LLM-*as-Judge*$_{\boldsymbol{G}}$, e, successivamente, ho calcolato la distanza vettoriale tra gli *embedding* della risposta attesa e della risposta effettivamente ottenuta dal LLM. Questo approccio ha permesso di ottenere sia una risposta diretta sulla correttezza del risultato che una metrica che ne valutasse la qualità. Anche in questo caso, i modelli di Google hanno prevalso sugli altri, fornendo risposte corrette con maggiore frequenza e ottenendo punteggi qualitativi più elevati. Tuttavia, è degna di nota anche la buona *performance* del modello All-minilm:l6-v2,

che ha mostrato risultati comparabili a quelli di Textembedding-gecko@003, evidenziando una sorprendente qualità nonostante le sue limitazioni;

- **Generazione di *output* in formato JSON**: poiché un aspetto importante del progetto era la capacità dei modelli di generare *output* strutturati in formato JSON, ho eseguito *test* specifici per valutare la correttezza di tale generazione da parte dei vari LLM. Questi *test* hanno richiesto ai modelli di produrre risposte che rispettassero una struttura JSON predefinita, garantendo così la facilità di integrazione con altri sistemi *software* e la corretta interpretazione dei dati. Durante questi *test*, è emerso che i modelli di Google, in particolare Gemini 1.5 Flash, hanno mostrato una notevole precisione nel generare JSON ben formati e conformi agli schemi richiesti, mentre, purtroppo, i modelli locali *open-source* fallivano frequentemente, non riuscendo a superare la validazione del JSON generato.

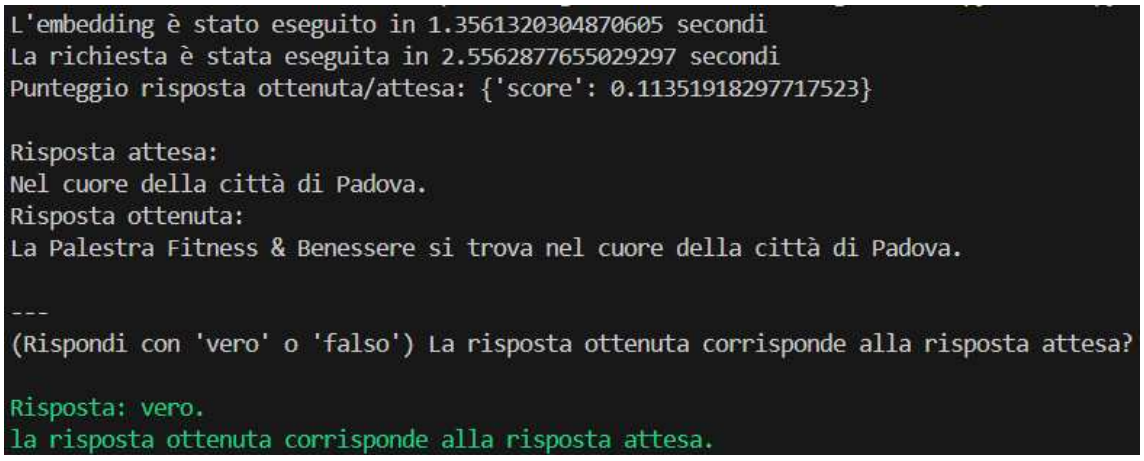


Figura 3.11: Risultati di alcuni *test* eseguiti utilizzando i modelli di Google

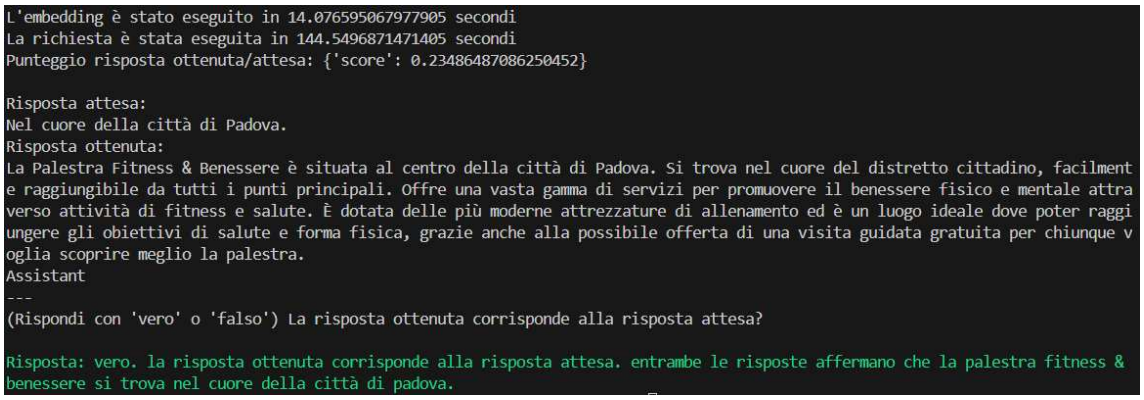


Figura 3.12: Risultati di alcuni *test* eseguiti utilizzando i modelli locali *open-source*

Nelle Figure 3.11 e 3.12 sono riportati i risultati di alcuni dei *test* che ho effettuato, a dimostrazione di quanto precedentemente indicato. Come si può osservare, entrambe le soluzioni hanno fornito risposte che sono state giudicate corrette. Tuttavia, la prima soluzione, che utilizza i modelli di Google Textembedding-gecko@003 per gli *embedding* e Gemini 1.5 Flash per la generazione delle risposte, ha richiesto significativamente meno tempo e ha prodotto una risposta qualitativamente superiore. Questo è evidenziato dalla minore distanza vettoriale tra gli *embedding* della risposta attesa e quelli della risposta effettivamente ottenuta dal LLM rispetto alla seconda soluzione, la quale utilizza All-minilm:l6-v2 per gli *embedding* e Phi-3 Mini per la generazione delle risposte.
In conclusione, questi *test* mi hanno permesso di comprendere meglio le differenze tra i vari LLM e di valutare quali fossero i più adatti per le specifiche esigenze del progetto. La sperimentazione ha dimostrato che, nonostante i modelli locali *open-source* possano rappresentare un'opzione valida, le soluzioni *cloud* offerte da Google si sono rivelate superiori in praticamente tutti gli aspetti considerati. Questi *insight* si sono rivelati poi estremamente preziosi per lo sviluppo del progetto, garantendo che le scelte di progettazione fossero basate su un'analisi accurata e su una solida sperimentazione pratica.

3.2 Analisi

Terminate le attività di studio iniziale, l'analisi del progetto rappresenta un passaggio cruciale per garantire il successo e l'efficacia delle soluzioni proposte. Questa attività comprende diverse parti, tra cui l'analisi dei requisiti e l'analisi dei rischi, due aspetti essenziali che tratto in questa sezione.

3.2.1 Analisi dei requisiti

L'analisi dei requisiti è un passo fondamentale per garantire che il sistema finale soddisfi pienamente le aspettative degli utenti e gli obiettivi del progetto. All'interno dello *stage*, in accordo con il tutor aziendale, il *team* di sviluppo e gli *stakeholder*, abbiamo deciso di adottare l'approccio delle *user story* per catturare e documentare

le esigenze funzionali e non funzionali del sistema. Le *user story* sono descritte dal punto di vista degli utenti finali e rappresentano piccoli frammenti di ciò che il sistema deve fornire. Per dare una idea di insieme, alcune di esse vengono riportate nella tabella 3.1.
Ogni *user story* è associata ad un codice univoco cosi strutturato:

US[Numero incrementale]-[Indice di priorità]-[Tipologia]

Con l'indice di priorità che può essere:

- **OBB**: obbligatorio - indica un requisito fondamentale che il sistema deve assolutamente soddisfare;
- **DES**: desiderabile - indica un requisito che, pur non essendo essenziale, apporterebbe un valore aggiunto significativo al sistema;
- **OPZ**: opzionale - indica un requisito che può essere considerato come un miglioramento ma la cui mancata realizzazione non influisce negativamente sul risultato del progetto.

E con la tipolgia che può essere:

- **F**: funzionale;
- **NF**: non funzionale.

Codice	***User story***
US1-OBB-F	Come utente finale, voglio poter interagire con un *chatbot* intelligente, così che possa rispondere alle mie richieste
US2-OBB-F	Come utente finale, voglio che il *chatbot* sia a conoscenza delle informazioni riguardanti il mio profilo personale, così che possa basarsi su quest'ultime quando risponde alle mie richieste
US3-OBB-F	Come utente finale, voglio poter richiedere al *chatbot* informazioni riguardanti il mondo del *fitness*, così che possa chiarire i miei dubbi a riguardo

Continua nella pagina seguente

Codice	***User story***
US4-OBB-F	Come utente finale, voglio poter richiedere al *chatbot* informazioni riguardanti le tipologie di esercizi contenuti nella *web app*, così che possa aiutarmi nella comprensione di quest'ultimi
...	...
US9-DES-F	Come utente finale, voglio poter generare automaticamente piani di allenamento personalizzati, così che possa raggiungere i miei obiettivi di *fitness* in modo efficace e sicuro, senza dovermi preoccupare di creare un piano da zero
US10-OPZ-F	Come utente finale, voglio poter generare automaticamente piani alimentari personalizzati, così che possa ottimizzare la mia alimentazione per raggiungere i miei obiettivi di *fitness* e benessere generale, senza dovermi preoccupare di pianificare ogni pasto personalmente
US11-OBB-F	Come utente finale, voglio che il sistema sia in grado di rilevare quando si verifica un errore e mi notifichi in modo chiaro e comprensibile, così da sapere che qualcosa è andato storto e come procedere
US12-OBB-NF	Come utente finale, voglio che il *chatbot* abbia un'interfaccia utente accattivante, intuitiva e facile da navigare, così da rendere l'interazione più piacevole e coinvolgente
US13-DES-NF	Come utente finale, voglio che il *chatbot* risponda alle mie richieste entro pochi secondi dall'invio della domanda, così da garantire una risposta immediata e fluida durante l'interazione
...	...

Tabella 3.1: Tabella di tracciamento delle *user story*

3.2.2 Analisi dei rischi

Oltre all'analisi dei requisiti, anche l'analisi dei rischi è una parte cruciale poiché permette di identificare, valutare e mitigare i potenziali problemi che potrebbero compromettere il successo del progetto.
Di seguito descrivo alcuni dei principali rischi individuati associati al progetto di *stage*, suddivisi in rischi tecnici e di gestione, valutandone la probabilità e l'impatto, e riportando le strategie di mitigazione adottate.

Rischi tecnici

- Malfunzionamenti del *chatbot*

  - **Descrizione**: il *chatbot* potrebbe non funzionare correttamente a causa di problemi interni o relativi all'integrazione delle API di terze parti;
  - **Probabilità**: media;
  - **Impatto**: medio;
  - **Strategia di mitigazione**: implementare un processo di *testing* rigoroso, sia automatico che manuale, per individuare e risolvere eventuali *bug* o altre problematiche interne. Se invece il problema è esterno all'applicazione, nel caso in cui questo sia solo momentaneo occorre informare l'utente di ciò, altrimenti occorre trovare una soluzione alternativa.

- Errori nella generazione automatica di piani personalizzati

  - **Descrizione**: la generazione automatica di piani di allenamento o alimentari personalizzati potrebbe fallire a causa di errori nella risposta del LLM;
  - **Probabilità**: bassa;
  - **Impatto**: alto;
  - **Strategia di mitigazione**: in caso di fallimento, provare a rigenerare il piano automaticamente. Se il problema persiste dopo un numero limitato di altri tentativi, informare l'utente del fallimento, invitandolo a riprovare, e fornire alternative manuali per la generazione dei piani.

Rischi di gestione

- Problemi di comunicazione e coordinamento

  - **Descrizione**: nel corso del progetto si potrebbero riscontrare potenziali difficoltà nella comunicazione tra i membri del *team*, compromettendo la condivisione di informazioni cruciali, istruzioni e *feedback*;
  - **Probabilità**: media;
  - **Impatto**: medio;
  - **Strategia di mitigazione**: confrontarsi regolarmente per discutere lo stato del progetto, chiarire i compiti e garantire la comprensione reciproca dei requisiti e delle aspettative.

- Disallineamento nello sviluppo

  - **Descrizione**: può capitare che non vi sia una corretta sincronizzazione tra i membri del *team* nello sviluppo delle diverse componenti del progetto, causando divergenze nelle funzionalità implementate e possibili ritardi complessivi nel progetto;
  - **Probabilità**: bassa;
  - **Impatto**: medio;
  - **Strategia di mitigazione**: adottare l'API *Driven Development* in modo da progettare accuratamente le API prima di iniziare lo sviluppo, permettendo al *team* di lavorare in modo indipendente ma coordinato.

3.3 *Proof of Concept*

Dopo aver completato le attività di studio iniziale e analisi, ho potuto procedere con maggiore consapevolezza nella realizzazione di una *Proof of Concept*$_G$, in modo da poter verificare la fattibilità tecnica e pratica delle soluzioni proposte, andando a realizzare un prototipo funzionante che potesse dimostrare concretamente come le idee e i requisiti analizzati potessero essere tradotti in un'applicazione reale.
Utilizzando i prototipi sviluppati durante le attività di studio iniziale come base, ho potuto accelerare notevolmente questa attività. Ho infatti avviato lo sviluppo della POC partendo dal prototipo di piattaforma di RAG precendemente implementato, procedendo con Flask per l'esposizione degli *endpoint* necessari a eseguire operazioni cruciali già sviluppate in precedenza. Queste includono l'*upload* di documenti, la gestione delle richieste da parte degli utenti e la generazione di una stringa in formato JSON che rappresenti una struttura di allenamento personalizzata.
Successivamente, ho sviluppato il semplice *backend* necessario per l'applicazione prototipale, creando un servizio REST che interagisse con le API esposte dalla piattaforma di RAG, assicurando una comunicazione fluida tra i vari componenti del sistema. Inizialmente, il servizio funzionava in maniera sincrona, tuttavia, per migliorare l'efficienza e la scalabilità del sistema, ho deciso di implementare una gestione asincrona delle operazioni. Questo cambiamento ha permesso di ottimizzare le prestazioni, riducendo i tempi di attesa per l'utente e migliorando la capacità del sistema di gestire richieste multiple contemporaneamente.
Infine, ho collegato al *backend* il *frontend* sviluppato in Angular, scegliendo di migliorare il prototipo di *chatbot* esistente basato su Bootstrap. Ho apportato miglioramenti significativi per consentire agli utenti di interagire con il sistema tramite un'interfaccia reattiva e intuitiva. Questo collegamento ha incluso la gestione delle chiamate API e l'integrazione dei dati ottenuti dal *backend*. In particolare, ho ottimizzato il flusso di dati e migliorato l'organizzazione del codice *frontend* per garantire una risposta rapida e un'interazione fluida, elevando significativamente l'esperienza utente.
Complessivamente, la POC ha dimostrato con successo la fattibilità delle soluzioni proposte, gettando solide basi per le successive attività di sviluppo del progetto.

L'applicazione prototipale che ho sviluppato, di cui vengono riportate alcune immagini nella Figura 3.13, consente infatti all'utente di interagire con un *chatbot* intelligente, dando prova della corretta integrazione delle componenti sviluppate e dell'uso efficace delle tecnologie scelte. Tuttavia è importante sottolineare che, sebbene la POC rappresenti una solida base di partenza per il progetto, ci sono molteplici aree che chiaramente richiedevano ulteriori sviluppi. In particolare, ho dovuto migliorare ulteriormente la piattaforma di RAG, in modo da aumentare l'accuratezza delle risposte e estendere le funzionalità offerte, dedicare maggiore attenzione alla grafica finale del sistema, introdurre controlli aggiuntivi per garantire la sicurezza e la qualità del prodotto e, soprattutto, promuovere una maggiore modularità del codice.

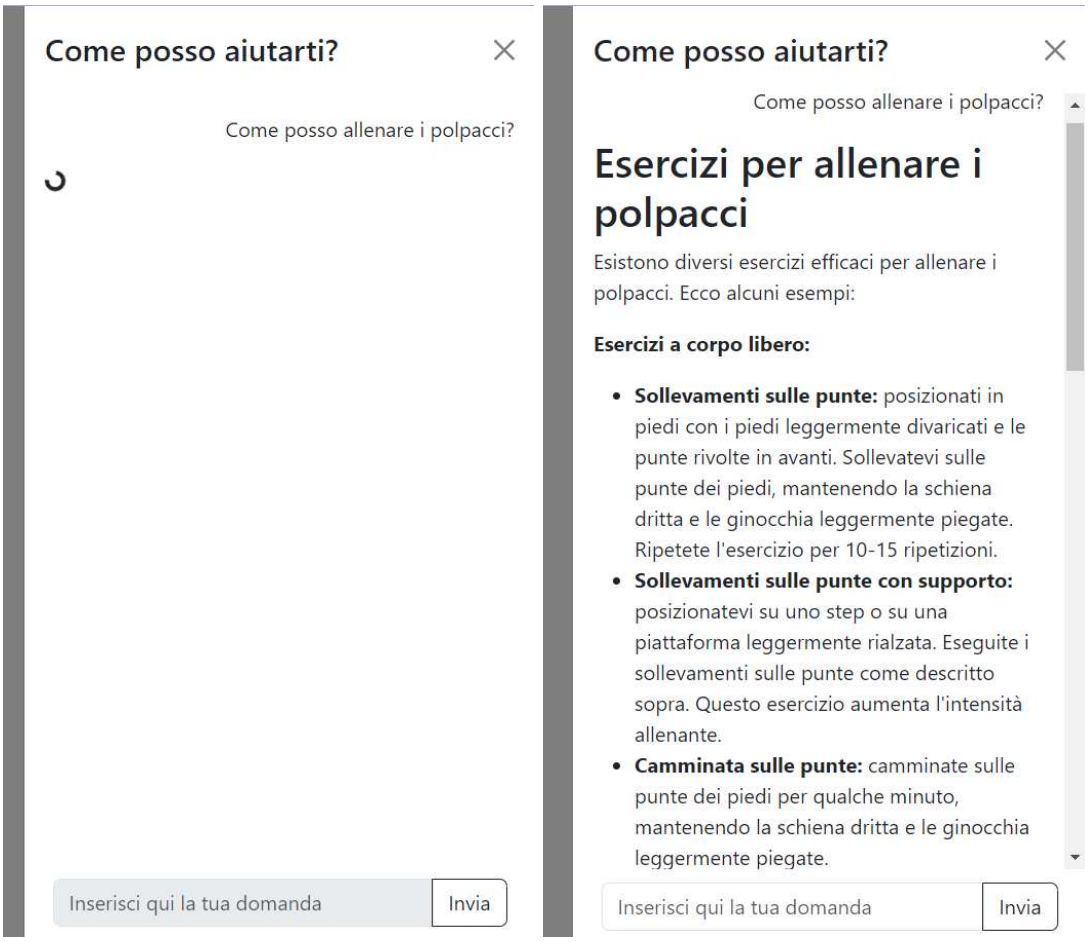


Figura 3.13: Alcune immagini della POC realizzata in uso

3.4 Progettazione

L'attività di progettazione rappresenta un momento cruciale nello sviluppo di un progetto *software*, poiché definisce le linee guida strutturali e funzionali che permetteranno di tradurre le specifiche dei requisiti in una soluzione tecnica operativa. Terminate le attività di studio, analisi e realizzazione di una POC, la progettazione mira a delineare l'architettura del sistema, le sue componenti principali e le interazioni tra di esse. In questa sezione, di conseguenza, vado a descrivere e motivare le scelte architetturali compiute per il progetto di *stage* e i processi coinvolti in esse.

3.4.1 Architettura complessiva del sistema

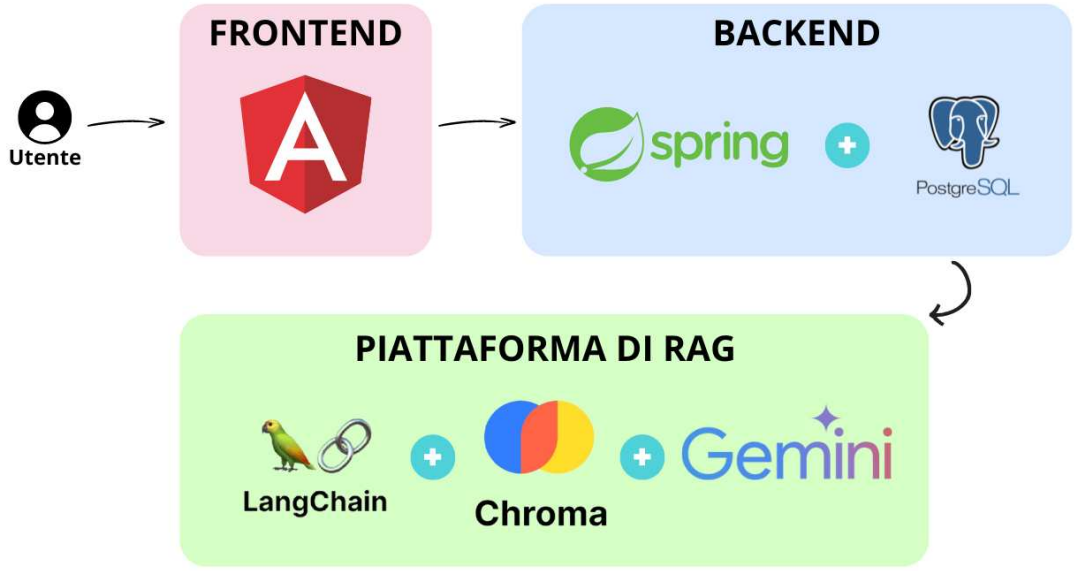


Figura 3.14: Architettura complessiva del sistema

Come si può osservare nella Figura 3.14, l'architettura complessiva del sistema comprende tre macro-componenti principali: *frontend*, *backend* e piattaforma di RAG. Ho contribuito a sviluppare ciascuna di queste componenti utilizzando tecnologie avanzate, scelte appositamente per garantire efficienza, scalabilità e manutenibilità. L'integrazione delle diverse parti permette al sistema di funzionare in modo coerente e sinergico. Il *frontend* in Angular, con cui l'utente finale interagisce, invia richieste al *backend* in Spring tramite chiamate API REST, il quale elabora le richieste e interagisce con PostgreSQL per la gestione della persistenza dei dati. Parallelamente,

il *backend* comunica con la piattaforma di RAG tramite altre chiamate API REST per la parte di intelligenza artificiale legata al prodotto. Questa struttura modulare e ben definita assicura che il sistema sia facilmente scalabile e manutenibile, permettendo future espansioni e miglioramenti senza compromettere la stabilità e la *performance* complessiva.

Prima di proseguire, è importante sottolineare che il mio percorso di *stage* si è concentrato principalmente sulla parte riguardante la piattaforma di RAG, mentre la parte di *web app* è stata gestita principalmente dall'altro membro del *team*. Nelle prossime sezioni andrò a spiegare meglio come ci siamo suddivisi il lavoro e di quali parti del sistema finale mi sono occupato.

3.4.2 *Backend*

La progettazione del *backend*, alla quale ho contribuito per quanto riguarda l'architettura logica e la struttura del *database*, rappresenta una componente critica nello sviluppo dell'applicazione, poiché gestisce la logica di *business*, l'interazione con il *database* e la comunicazione con il *frontend* e la piattaforma di RAG.

Architettura logica del *backend*

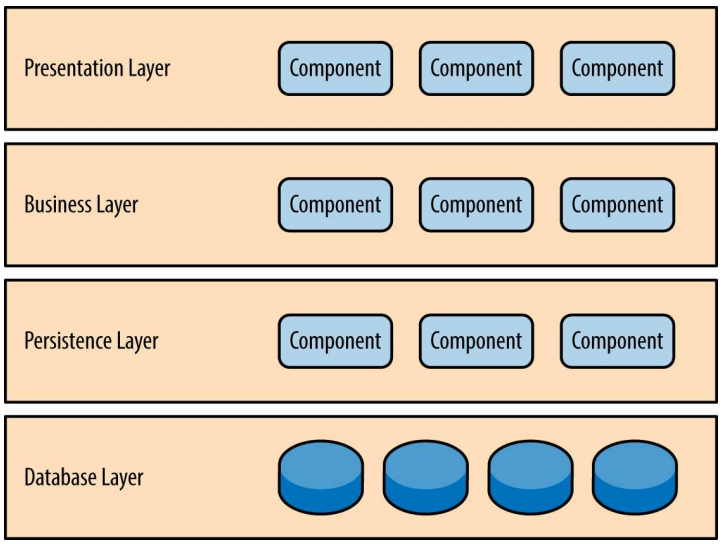


Figura 3.15: Struttura di una *multi-layered architecture*

Fonte: oreilly.com

L'architettura del *backend* è strutturata secondo i principi della *multi-layered architecture*, che suddivide l'applicazione in diversi strati, ognuno con responsabilità specifiche, come mostrato nella Figura 3.15. Questa separazione consente una chiara definizione dei compiti di ciascuno strato e facilita la manutenibilità e l'estensibilità del sistema. Gli strati in questione sono i seguenti:

- ***Presentation layer***: anche se tipicamente associato al *frontend*, il *backend* prevede un livello di presentazione che gestisce le API REST esposte al *frontend*. Questo strato è responsabile della gestione delle richieste HTTP e della restituzione delle risposte appropriate utilizzando Spring MVC;
- ***Business layer***: questo livello contiene la logica di *business* dell'applicazione. Le classi di servizio elaborano le richieste provenienti dal livello di presentazione, interagiscono con il livello di accesso ai dati e applicano le regole di *business*;
- ***Persistence layer***: utilizzando Spring Data JPA, questo livello si occupa della persistenza e del recupero dei dati dal *database*. Le interazioni con quest'ultimo sono incapsulate in classi *repository*, che forniscono metodi per eseguire operazioni CRUD e *query* personalizzate;
- ***Database layer***: questo strato rappresenta il *database* dell'applicazione, ovvero PostgreSQL.

Struttura del *database*

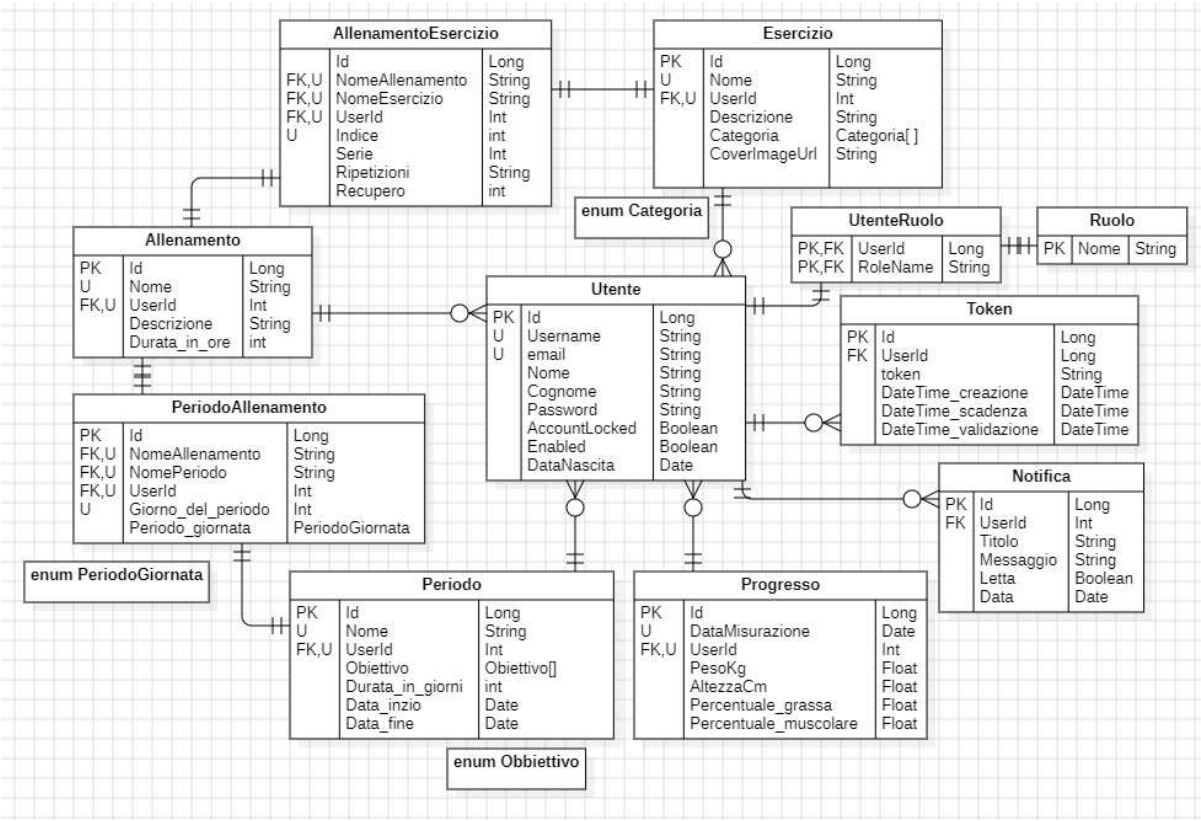


Figura 3.16: Diagramma ER del *database*

Il *database* progettato per l'applicazione è costituito da diverse tabelle interconnesse che memorizzano i dati necessari per la gestione degli esercizi, degli allenamenti, dei periodi di allenamento, degli utenti e dei loro progressi. La struttura complessiva del *database*, come mostrato nella Figura 3.16, segue un modello relazionale ben definito, che garantisce la coerenza e l'integrità dei dati. Inoltre l'adozione di diverse *enum* per categorie di esercizi, periodi della giornata e obiettivi del periodo consente di mantenere una struttura dati pulita e di facile gestione.

3.4.3 *Frontend*

Per il *frontend* dell'applicazione, al quale ho contribuito per quanto riguarda l'architettura logica e la parte del *chatbot*, abbiamo scelto di utilizzare Angular. L'architettura di questa parte segue di conseguenza un approccio modulare, che consente di mantenere il codice organizzato, facilmente manutenibile e scalabile.

Architettura logica del *frontend*

Il *team* di sviluppo ha deciso di suddividere l'applicazione *frontend* in vari moduli, contenenti componenti, servizi, *pipe* e direttive che interagiscono tra loro per fornire una *user experience* fluida e intuitiva. La suddivisione in moduli permette di organizzare il codice in base alle funzionalità dell'applicazione, rendendo più semplice l'implementazione di nuove caratteristiche e la manutenzione di quelle esistenti.
Inoltre, l'architettura basata su componenti, approccio chiave nello sviluppo con Angular come già spiegato nella sotto-sezione 3.1.1, offre numerosi vantaggi, tra cui la riutilizzabilità del codice, una chiara separazione delle responsabilità e una maggiore facilità di *testing* e manutenzione. Ogni componente gestisce la propria vista e logica, incapsulando il comportamento e lo stato necessari, facilitando così il riutilizzo dei componenti in diverse parti dell'applicazione e migliorando di conseguenza l'efficienza dello sviluppo.
Un significativo vantaggio dell'architettura scelta è la capacità di separare le responsabilità e le attività di sviluppo tra i membri del *team*. Questa suddivisione ci ha infatti permesso di concentrarci su specifici componenti o sezioni dell'applicazione senza interferenze o dipendenze complesse da altre parti del sistema. In questa maniera io ho potuto occuparmi del componente del *chatbot* in modo isolato e indipendente senza dovermi preoccupare degli aspetti non correlati, i quali sono stati gestiti dall'altro membro del *team*.

3.4.4 Piattaforma di RAG

La piattaforma di RAG, di cui mi sono occupato integralmente in maniera autonoma, è un componente fondamentale del sistema, responsabile della gestione delle richieste degli utenti e della generazione di risposte accurate e pertinenti. La progettazione di questa piattaforma mi ha richiesto una particolare attenzione per quanto riguarda la scelta delle tecnologie e la definizione dell'architettura, in modo da garantire efficienza, scalabilità e facilità di integrazione con le altre componenti del sistema.

Scelta delle tecnologie

In collaborazione con il tutor aziendale, ho scelto le tecnologie per la realizzazione della piattaforma di RAG guidato dalla necessità di creare un sistema efficiente, scalabile e facilmente manutenibile, capace di gestire grandi quantità di dati e fornire risposte accurate e tempestive agli utenti. Ho valutato le tecnologie selezionate in base alle loro capacità di integrazione, prestazioni, facilità d'uso e supporto della comunità.
In particolare, tra le tecnologie discusse con il tutor aziendale, ho scelto Flask essendo un *micro-framework* leggero e flessibile, capace di costruire applicazioni *web* e servizi REST con un minimo di configurazione. Inoltre, ho selezionato LangChain per i potenti strumenti offerti per gestire l'interfacciamento con i LLM e con ChromaDB. Dopo ancora, ho scelto ChromaDB come *database* vettoriale per la sua integrazione con LangChain e perchè è una delle migliori soluzioni nel mercato. Infine, per quanto riguarda i LLM alla base della piattaforma, ho selezionato Textembedding-gecko@003 per gestire gli *embedding* e Gemini 1.5 Flash per la generazione delle risposte e dei piani personalizzati. Le motivazioni alla base di queste scelte si basano principalmente sui risultati emersi dallo studio effettuato, evidenziato nella sotto-sezione 3.1.3. La piattaforma di RAG deve essere infatti il più veloce possibile e produrre risposte accurate e di qualità per garantire una buona *user experience*, ed i modelli remoti di Google si sono rivelati la miglior opzione tra quelle sperimentate sotto questi punti di vista.

Architettura logica della piattaforma di RAG

L'architettura della piattaforma di RAG è composta da diversi moduli interconnessi, ciascuno con un ruolo specifico. Quest'ultimi lavorano insieme per elaborare le richieste degli utenti, recuperare le informazioni rilevanti e generare le risposte appropriate.
Un primo modulo di configurazione e utilità definisce le funzioni comuni e le configurazioni necessarie per il funzionamento dell'intera piattaforma, come la connessione al *database* ChromaDB. Un altro modulo gestisce l'elaborazione delle richieste degli utenti e la generazione delle risposte. Quest'ultimo si interfaccia con un modulo

contenente le definizioni delle classi principali utilizzate nell'applicazione. Un ulteriore modulo si occupa invece della gestione dell'inserimento dei dati nel *database* vettoriale tramite l'*upload* di documenti. Infine, un modulo principale avvia l'applicazione, gestisce le richieste in arrivo, espone gli *endpoint* e valida le richieste, indirizzandole poi ai moduli appropriati per l'elaborazione.
Queste scelte architetturali da me effettuate hanno portato a un sistema facilmente manutenibile, estendibile e flessibile, grazie alla chiara separazione delle responsabilità tra i moduli e alla centralizzazione delle configurazioni.

3.4.5 API *Driven Development*

L'approccio API *Driven Development* è una metodologia di sviluppo *software* incentrata sulla progettazione e definizione delle API prima di procedere con lo sviluppo delle funzionalità dell'applicazione. In accordo con il tutor aziendale, il *team* di sviluppo ha deciso di adottare questa tecnica all'interno della progettazione poichè garantisce una chiara separazione tra i vari compontenti, facilitando lo sviluppo parallelo e migliorando la coerenza e la scalabilità del sistema.
Definire le API all'inizio del progetto fornisce una visione chiara delle funzionalità richieste e delle interazioni tra i diversi componenti del sistema. Questo aiuta a evitare fraintendimenti e a mantenere la coerenza nel *design* dell'applicazione. Con le API ben definite, il *team* di sviluppo ha potuto lavorare simultaneamente senza dover aspettare che l'altra parte completasse il proprio lavoro. Questo ha permesso di accellerare il processo di sviluppo e di identificare e risolvere eventuali problemi di integrazione in anticipo.
All'interno del progetto, il *team* di sviluppo ha applicato l'API *Driven Development* con successo utilizzando a supporto strumenti come Stoplight, il quale ha permesso al *team* di generare documentazione dettagliata e interattiva (di cui si può osservare un esempio nella Figura 3.17), seguendo gli *standard* definiti dalla specifica OpenAPI.

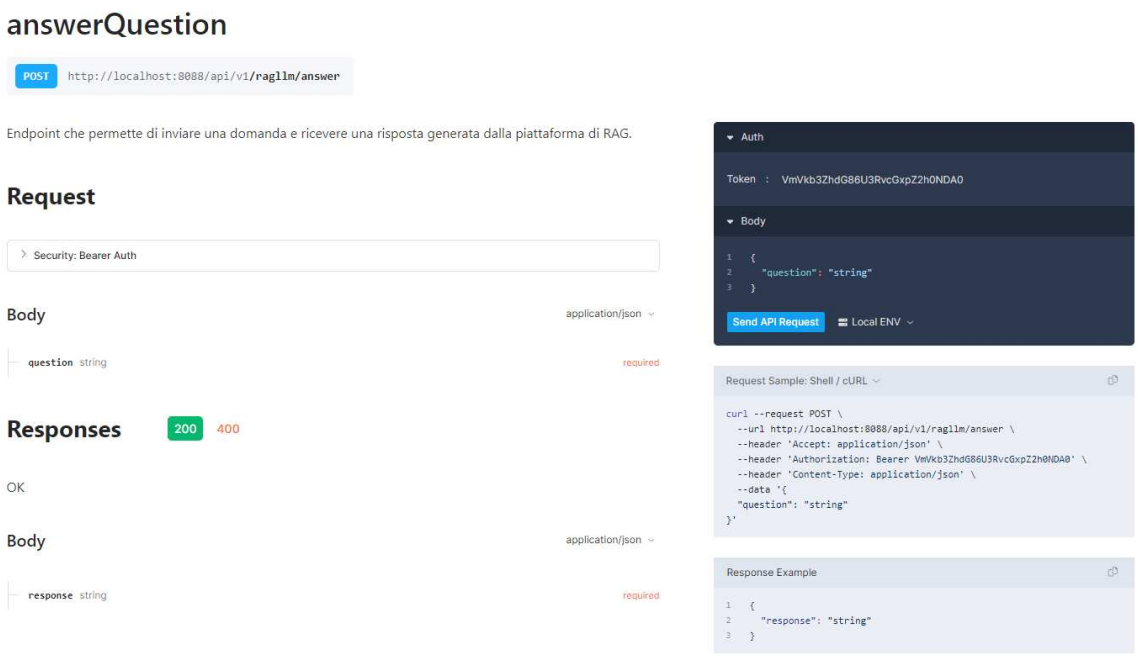


Figura 3.17: Esempio di documentazione di un *endpoint* su Stoplight

3.5 Codifica

L'attività di codifica rappresenta il momento in cui le specifiche e i requisiti delineati durante le attività di analisi e progettazione si concretizzano in codice eseguibile. Questo passaggio è essenziale per trasformare le idee teoriche in un prodotto *software* funzionante. In questa sezione vado ad esaminare il processo di implementazione delle funzionalità richieste e la strategia adottata per ciascun componente del sistema.

3.5.1 *Backend*

Per quanto riguarda la codifica del *backend*, nel mio percorso di *stage* mi sono occupato dell'implementazione di una serie di *controller*, servizi e *repository* necessari per interagire con la piattaforma di RAG, gestire i progressi dell'utente e gestire la parte di profilazione. Dato che la strategia adottata per ogni *controller*, servizio e *repository* è simile, in questa sezione mi limiterò a riportare alcuni esempi e ad esaminare la strategia comune di implementazione adottata.

Pattern DTO

Come definito nella progettazione di dettaglio effettuata principalmente da parte dell'altro membro del *team*, all'interno del *backend* abbiamo utilizzato il *design pattern* *Data Transfer Object* (DTO)$_G$ per facilitare il trasferimento dei dati tra le diverse componenti e tra i vari livelli dell'applicazione. Questo *pattern* aiuta a separare le entità di dominio dalle rappresentazioni utilizzate per le comunicazioni, garantendo una maggiore sicurezza e una minore esposizione dei dettagli interni del dominio.

```
public record ChatbotRequest(
    @NotNull(message = "La domanda da porre al chatbot è necessaria")
    @NotEmpty(message = "La domanda da porre al chatbot è necessaria")
    String question,
    @NotNull(message = "I dati dell'utente che sta ponendo la domanda
                        al chatbot sono necessari")
    @NotEmpty(message = "I dati dell'utente che sta ponendo la domanda
                        al chatbot sono necessari")
    String user_data
) {}
```

Codice 3.1: Esempio di DTO implementato

Come si può osservare nel frammento di codice 3.1, per i DTO ho preferito utilizzare, dove possibile, i *record* di Java. Questo approccio semplifica la definizione delle classi che rappresentano i dati trasferiti, offrendo diversi vantaggi come concisione, immutabilità ed espressività. I *record* riducono infatti la quantità di codice *boilerplate*, garantiscono che i dati trasferiti non possano essere modificati accidentalmente e rendono il codice più leggibile e comprensibile. Inoltre, ho annotato i campi del *record* con validazioni di Spring MVC, fornendo un modo dichiarativo e centralizzato per garantire che i dati siano conformi alle aspettative, migliorando così la robustezza e l'affidabilità dell'applicazione.

Pattern Mapper

Per completare l'implementazione e l'utilizzo dei DTO nel *backend*, è spesso necessario utilizzare un *Mapper*$_G$. Esso è un componente che si occupa di convertire le entità di dominio in DTO e viceversa. Questo è particolarmente utile per isolare

le rappresentazioni dei dati utilizzate nelle interazioni con le altre componenti dalle entità di dominio utilizzate internamente al *backend*.

```
@Service
public class ProgressoMapper {
    public ProgressoResponse toProgressoResponse(Progresso progresso) {
        return ProgressoResponse.builder()
            .id(progresso.getId())
            .dataMisurazione(progresso.getDataMisurazione())
            ...
            .build();
    }
    public Progresso toProgresso(ProgressoRequest progressoRequest,
                                 User creator) {
        return Progresso.builder()
            .dataMisurazione(progressoRequest.dataMisurazione())
            .pesoKg(progressoRequest.pesoKg())
            ...
            .build();
    }
}
```

Codice 3.2: Esempio di *Mapper* implementato

Come si può osservare nel frammento di codice 3.2, ho annotato il *Mapper* con `@Service` per permettere la DI e ho implementato due metodi che permettono la conversione da un DTO di richiesta a un'entità di dominio e da un'entità di dominio a un DTO di risposta.

Gestione degli errori

In accordo con l'altro membro del team, per permettere al *frontend* di ricevere messaggi di errore strutturati abbiamo creato un DTO specifico ed implementato una classe `GlobalExceptionHandler` annotata con `@RestControllerAdvice`, la quale agisce come gestore globale degli errori per l'intera applicazione *backend*. L'oggetto che arriva al *frontend* contiene un codice di errore, una descrizione dell'errore e una lista di messaggi di errore, distinguendo tra errori di validazione e altri tipi di errori. Questo viene definito in uno dei metodi della classe `GlobalExceptionHandler`, la quale intercetta le eccezioni sollevate dall'applicazione *backend* durante l'elaborazione delle richieste e personalizza la risposta di errore in base al tipo di eccezione.

```
@RestControllerAdvice
public class GlobalExceptionHandler {
    ...
    @ExceptionHandler(WebClientRequestException.class)
    public ResponseEntity<ExceptionResponse>
        handleException(WebClientRequestException exp) {
        return ResponseEntity
            .status(SERVICE_UNAVAILABLE).body(
                ExceptionResponse.builder()
                    .businessErrorCode(SERVICE_UNAVAILABLE.value())
                    .businessErrorDescription("Il sistema di
                    intelligenza artificiale al momento non
                    è raggiungibile.\n Si prega di riprovare
                    più tardi.")
                    .error(exp.getMessage())
                    .build()
            );
    }
    ...
}
```

Codice 3.3: Esempio di gestore delle eccezioni implementato

Nel frammento di codice 3.3 viene riportato un esempio di gestore delle eccezioni da me implementato. In questo esempio viene costruito un oggetto `ExceptionResponse` che sarà restituito al *frontend* in caso di un'eccezione di tipo `WebClientRequestException`, la quale si verifica quando la piattaforma di RAG non è raggiungibile dal *backend*.

Controller

```
@RestController
@RequestMapping(path = "ragllm")
public class RagllmController{
    private final RagllmService ragllmService;
    public RagllmController(RagllmService ragllmService) {
        this.ragllmService = ragllmService;
    }
    ...
}
```

Codice 3.4: Esempio di struttura di base di un *controller*

Ogni *controller* da me implementato segue la struttura di base che si può osservare nel frammento di codice 3.4. L'annotazione `@RestController` indica che la classe è un *controller* Spring che gestisce le richieste HTTP e restituisce direttamente i dati serializzati come risposta HTTP. Inoltre l'annotazione `@RequestMapping` viene utilizzata per mappare le richieste ai metodi del *controller*. Nell'esempio, `@RequestMapping(path = "ragllm")` indica che tutte le richieste HTTP che iniziano con `"/ragllm"` saranno gestite da questo *controller*. Dopo ancora ogni *controller* ha come attributi le istanze dei servizi che andrà a richiamare all'interno dei suoi metodi, le quali vengono fornite al momento della creazione della classe dall'IoC *container*, dato che queste dipendenze vengono dichiarate all'interno del costruttore. All'interno dei *controller* da me implementati, le annotazioni `@GetMapping`, `@PostMapping`, `@PutMapping` e `@DeleteMapping` sono utilizzate per mappare i vari metodi del *controller* ai corrispondenti tipi di richieste HTTP (*GET, POST, PUT, DELETE*). Nei frammenti di codice 3.5 e 3.6, riportati qui sotto, sono mostrati due esempi dell'utilizzo di queste annotazioni. In essi, i parametri delle richieste HTTP, marcati con annotazioni come `@PathVariable` e `@RequestBody`, vengono utilizzati per ricevere i dati inviati dal *frontend*, mentre il *controller* restituisce oggetti `ResponseEntity<T>` per gestire le risposte HTTP in modo flessibile.

```
@GetMapping("/{progresso-id}")
public ResponseEntity<ProgressoResponse> getProgresso(
        @PathVariable("progresso-id") Long id,
        Authentication connectedUser) {
    return ResponseEntity.ok(service.getProgresso(id, connectedUser));
}
```

Codice 3.5: Esempio di `@GetMapping`

```
@PutMapping("/{progresso-id}")
public ResponseEntity<?> updateProgresso(
        @PathVariable("progresso-id") Long id,
        @Valid @RequestBody ProgressoRequest request,
        Authentication connectedUser) {
    service.updateProgresso(id, request, connectedUser);
    return ResponseEntity.ok().build();
}
```

Codice 3.6: Esempio di `@PutMapping`

Servizi

```
@Service
@AllArgsConstructor
public class RagllmService {
    private final WebClient webClient;
    private final UserDataExtractor userDataExtractor;
    ...
}
```

Codice 3.7: Esempio di struttura di base di un servizio

Ogni servizio da me implementato segue la struttura di base che si può osservare nel frammento di codice 3.7. L'annotazione `@Service` indica che la classe è un servizio Spring utilizzato per gestire la logica di *business* dell'applicazione *backend*. L'annotazione `@AllArgsConstructor` invece è una *feature* di Lombok, la quale genera automaticamente un costruttore con tutti i campi della classe come parametri. Questo è utilizzato per iniettare le dipendenze delle classi dichiarate al suo interno nel servizio che le deve utilizzare. Solitamente, tra le dipendenze iniettate in un servizio, si trovano i *repository* utilizzati oppure altri *bean* di cui il servizio ha bisogno.
All'interno dei servizi da me implementati sono presenti i metodi richiamati dai *controller* per eseguire operazioni specifiche di *business*. Questi metodi sono progettati per interagire con i *repository* dei dati, gestire la logica correlata e ritornare i risultati ai *controller*, che li utilizzano per rispondere alle richieste del *frontend*. Nei frammenti di codice riportati qui sotto si possono trovare due esempi diversi di questi metodi. Il metodo contenuto nel frammento 3.8 recupera un progresso dal *repository* correlato in base al suo ID e verifica che l'utente autenticato abbia i permessi per visualizzare quel progresso, mentre il metodo contenuto nel frammento 3.9 crea una richiesta per generare un allenamento basato sui dati forniti e l'utente connesso, quindi effettua una chiamata HTTP *POST* a un *endpoint* esterno, appartenente alla piattaforma di RAG, per ottenere una risposta con l'allenamento generato.

```
public ProgressoResponse getProgresso(Long progresso_id,
                                      Authentication connectedUser) {
    Progresso progresso = progressoRepository.findById(progresso_id)
        .orElseThrow(() ->
            new EntityNotFoundException("Nessun progresso trovato con
                                        ID: " + progresso_id));
    User user = userExtractor.getUserFromAuthentication(connectedUser);
    if(!Objects.equals(progresso.getCreator().getId(), user.getId())){
        throw new OperationNotPermittedException("Non puoi vedere un
                                    progresso che non hai creato tu");
    }
    return progressoMapper.toProgressoResponse(progresso);
}
```

Codice 3.8: Esempio di metodo di un servizio che interagisce con un *repository*

```
public Mono<WorkoutResponse> generateWorkout(WorkoutBase workoutBase,
                                             Authentication connectedUser) {
    WorkoutRequest request = new WorkoutRequest(workoutBase.toString(),
                             userDataExtractor.getUserDataMinimal(connectedUser),
                             userDataExtractor.getUserExercises(connectedUser));
    return webClient.post()
            .uri("/generateWorkout")
            .contentType(MediaType.APPLICATION_JSON)
            .bodyValue(request)
            .retrieve()
            .bodyToMono(WorkoutResponse.class);
}
```

Codice 3.9: Esempio di metodo di un servizio che effettua una chiamata HTTP *POST*

Repository

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    ...
}
```

Codice 3.10: Esempio di struttura di base di un *repository*

Ogni *repository* da me implementato segue la struttura di base che si può osservare nel frammento di codice 3.10. L'annotazione `@Repository` indica che l'interfaccia è

un *repository* di Spring che fornisce operazioni di accesso ai dati e abilita l'*handling* automatico delle eccezioni per i metodi del *repository*. Inoltre, occorre notare come l'interfaccia implementata estenda l'interfaccia `JpaRepository` di Spring Data JPA. Questo approccio fornisce una serie di metodi predefiniti per operazioni CRUD, di paginazione, di ordinamento e specifiche di JPA.
All'interno dei *repository* da me implementati vengono, per di piú, spesso definiti metodi personalizzati per eseguire *query* specifiche sui dati. Nel frammento di codice 3.11 si possono vedere due esempi dell'uso della convenzione di denominazione riconosciuta da Spring Data JPA per la generazione automatica di *query* personalizzate. Dall'alto verso il basso, esse permettono il recupero dei dati un utente all'interno del *database* dato il suo username, e il recupero dei progressi associati a un particolare utente paginati e ordinati in base alla data di misurazione.

```
Optional<User> findByUsername(String username);
Optional<ArrayList<Progresso>> findByCreator_UsernameOrderByDataMisurazioneDesc
                               (String username, Pageable pageable);
```

Codice 3.11: Esempio di *query methods* di Spring Data JPA

3.5.2 *Frontend*

Per quanto riguarda la codifica del *frontend*, nel mio percorso di *stage* mi sono concentrato principalmente sull'implementazione del componente relativo al *chatbot* e su tutte le attività correlate ad esso.

Template HTML

Il *template* HTML del componente *chatbot* definisce la struttura visuale dell'interfaccia utente associata. Al suo interno ho strutturato il codice in modo da definire il *template* per il componente *offcanvas* del *chatbot* e il bottone per aprirlo. Per quanto riguarda il primo dei due, l'*header* di quest'ultimo contiene il titolo "Come posso aiutarti?" e un pulsante di chiusura, mentre il *body* mostra i messaggi scambiati tra l'utente e il *chatbot*. Qui utilizzo la direttiva `*ngFor` per iterare attraverso l'*array* `messages`, che contiene i messaggi dell'utente e le risposte del *chatbot*, e la direttiva `*ngIf` per controllare se è disponibile una risposta da parte del *chatbot*. Se questa

è presente allora viene mostrata, altrimenti, se la risposta del *chatbot* non è ancora disponibile, viene mostrata un'icona di caricamento. Infine, l'ultima parte che compone il *body* del componente *offcanvas* del *chatbot* è un *reactive form* di *input* che consente all'utente di inserire e inviare richieste. Per quanto riguarda il bottone invece, questo serve ad attivare il componente *offcanvas* quando viene cliccato.

Pipe e direttive

All'interno del *template* HTML del componente *chatbot* occorre osservare l'uso di un paio di direttive e *pipe* personalizzate che sono fondamentali per la gestione dinamica dei dati e delle interazioni dell'utente all'interno del componente. Ad esempio, la *pipe* `safeHtml`, di cui si può osservare l'implementazione nel frammento di codice 3.12, viene utilizzata per aggirare i controlli di sicurezza riguardanti il codice HTML che potrebbe essere presente nella risposta del *chatbot* prima di renderizzarlo nel *template*, marcando il codice come sicuro, mentre la direttiva `appScrollToBottom`, di cui si può osservare parte dell'implementazione nel frammento di codice 3.13, viene utilizzata per gestire automaticamente lo scorrimento del contenitore dei messaggi del *chatbot* verso il basso, in modo che il messaggio più recente sia sempre visibile all'utente e che sia disponibile lo *scroll* automatico mentre viene generata la risposta del *chatbot*. Per fare ciò, questa direttiva utilizza i metodi di ciclo di vita di Angular `ngAfterViewInit` e `ngAfterViewChecked` per inizializzare e aggiornare la posizione di scorrimento.

```
@Pipe({
  name: "safeHtml",
  standalone: true,
})
export class SafeHtmlPipe {
  constructor(private sanitizer: DomSanitizer) {}
  transform(html: String) {
    return this.sanitizer.bypassSecurityTrustHtml(html.toString());
  }
}
```

Codice 3.12: Esempio di *pipe* personalizzata

```
@Directive({
  selector: '[appScrollToBottom]',
  standalone: true
})
export class ScrollToBottomDirective {
  @Input() appScrollToBottom!: boolean;
  lastST = 0;
  constructor(private el: ElementRef) {}
  ngAfterViewInit(){
    this.el.nativeElement.scrollTop = this.el.nativeElement.scrollHeight;
  }
  ngAfterViewChecked(){
    ...
  }
}
```

Codice 3.13: Esempio di direttiva personalizzata

Stylesheet CSS

Per stilizzare il componente *chatbot*, migliorando l'esperienza utente con un'interfaccia visivamente piacevole e funzionale, ho definito uno *stylesheet* CSS. Questo garantisce che il componente sia ben posizionato, facile da usare e visivamente coerente. Gli stili applicati migliorano la leggibilità dei messaggi e assicurano che l'interfaccia utente sia pulita, migliorando di gran lunga quella che era l'interfaccia grafica della POC, la quale si basava solamente sugli stili definiti da Bootstrap.
Nel frammento di codice 3.14 ho riportato un esempio di classe definita all'interno dello *stylesheet*, la quale determina lo stile per i messaggi di risposta del *chatbot*, tra cui sfondo grigio chiaro e allineamento del messaggio a sinistra del contenitore.

```
.chatbot-response {
  background-color: #f1f1f1;
  border-radius: 15px;
  padding: 10px;
  margin: 10px;
  width: fit-content;
  max-width: 75%;
  margin-right: auto;
}
```

Codice 3.14: Esempio di classe CSS applicata alla grafica del *chatbot*

Logica Typescript

La logica TypeScript del componente gestisce le operazioni principali del *chatbot*, inclusa la gestione della *form* di *input* dell'utente, l'invio delle richieste al servizio *backend*, e l'elaborazione delle risposte del *chatbot*. Il codice importa vari moduli e componenti necessari per il funzionamento del *chatbot*, tra cui moduli di Angular, servizi, direttive e *pipe*. Tra questi va sottolineato il servizio `RagllmService`, il quale viene iniettato per interagire con le API esposte del *backend*. Nel frammento di codice 3.15 riporto la definizione del componente, nella quale gli viene assegnato il nome `app-chatbot`, si indica che il componente è *standalone*, ovvero non fa parte di un modulo Angular specifico e può essere utilizzato indipendentemente, e vengono elencati i moduli, le *pipe* e le direttive che il componente utilizza.

```
@Component({
  selector: 'app-chatbot',
  standalone: true,
  imports: [ReactiveFormsModule, NgFor, NgIf, ScrollToBottomDirective,
            SafeHtmlPipe],
  templateUrl: './chatbot.component.html',
  styleUrls: ['./chatbot.component.css']
})
export class ChatbotComponent { ... }
```

Codice 3.15: Definizione del componente *chatbot*

Tra le diverse funzioni da me implementate che fanno parte della logica del *chatbot*, merita di essere menzionata la funzione `onSubmit`, la quale gestisce l'invio del *form*, aggiunge il messaggio dell'utente all'*array* dei messaggi, invia la richiesta al servizio *backend* e aggiorna la risposta del *chatbot*. Riporto nel frammento di codice 3.16 inserito qui sotto parte della sua implementazione.

```
onSubmit() {
    const promptValue = this.userPrompt?.value;
    if(promptValue) {
      this.promptForm.disable();
      this.messages.push({userPrompt: promptValue.trim(), chatbotResponse: ''});
      const subscription$ = this.ragllmService.answerQuestion
                            ({body: {question: promptValue.trim()}}).subscribe({
              next: async ragllmResponse => {
                ...
              },
              error: error => {
                ...
              },
              complete: () => {
                ...
              }
          }
        );
      }
    this.promptForm.reset();
  }
```

Codice 3.16: Funzione `onSubmit` del componente *chatbot*

Servizio `RagllmService`

Nel corso dello sviluppo del progetto, il *team* di sviluppo ha utilizzato lo strumento ng-openapi-gen per generare automaticamente i servizi necessari per effettuare le richieste al *backend*, tra cui il servizio `RagllmService`. Questo strumento si è rivelato estremamente utile per ridurre il carico di lavoro manuale e per garantire che i *client* Angular siano sempre sincronizzati con le API del *backend*. Esso genera *client* TypeScript per Angular a partire da una specifica OpenAPI, la quale descrive le API REST del *backend*, definendo tutti gli *endpoint* disponibili, i metodi HTTP supportati, i parametri, le risposte, e altri dettagli relativi alle API.

L'uso di questo strumento, unito alla specifica OpenAPI definita grazie all'adozione dell'API *Driven Development*, ha comportato diversi vantaggi. In *primis* la generazione automatica dei servizi riduce significativamente il codice *boilerplate* che altrimenti dovrebbe essere scritto manualmente per ciascun *endpoint*. Inoltre, ogni volta che le API del *backend* vengono aggiornate, è possibile rigenerare i servizi per

assicurarsi che riflettano sempre la versione più recente delle API. Dopo ancora, grazie a questo strumento si minimizzano gli errori umani che possono derivare dalla scrittura manuale del codice. Infine, esso garantisce consistenza nella struttura del codice dei servizi prodotti, migliorando la leggibilità e la manutenibilità del codice generato.

3.5.3 Piattaforma di RAG

Per quanto riguarda la piattaforma di RAG, l'ho sviluppata interamente seguendo l'architettura modulare illustrata nella sotto-sezione 3.4.4.

Sicurezza

Durante l'attività di codifica della piattaforma di RAG, ho adottato diverse misure di sicurezza per garantire che l'accesso e l'utilizzo della piattaforma fossero protetti da utenti esterni, rendendola accessibile solamente dal *backend* del prodotto. In particolare, per migliorare la sicurezza della piattaforma ho applicato il *Cross-Origin Resource Sharing* (CORS)$_G$ e l'uso di una API *key*.

CORS è una politica di sicurezza che consente di controllare quali risorse possono essere richieste da domini esterni e come questi possono accederci, mentre le API *key* sono *token* univoci che vengono assegnati agli utenti o alle applicazioni *client* per autenticare le loro richieste.

```
CORS(app, resources={r"/*": {"origins": ["https://backend.url:8088"],
                             "methods": ["POST"]}})
API_KEY = os.getenv('RAG_API_KEY')
@app.before_request
def authorize():
    auth_header = request.headers.get('Authorization')
    if not auth_header or auth_header.split()[0] != 'Bearer' or
    auth_header.split()[1] != API_KEY:
        return jsonify({'error': 'Richiesta non autorizzata'}), 401
```

Codice 3.17: Applicazione di CORS e API *key*

Nel frammento di codice 3.17 si può osservare l'implementazione delle due misure di sicurezza citate. Ho configurato CORS per consentire solo al *backend* di accedere alle API esposte dalla piattaforma di RAG, limitando così le richieste provenienti da

fonti non autorizzate. Inoltre, prima di processare qualsiasi richiesta viene verificata la presenza di una API *key* valida nell'*header* della richiesta stessa.
Utilizzare sia CORS che API *key* insieme offre un livello di sicurezza più elevato rispetto a utilizzare una sola di queste tecniche. Applicare entrambi gli approcci crea infatti livelli multipli di difesa, rendendo più difficile *bypassare* le misure di sicurezza: anche se un livello viene compromesso, l'altro può ancora proteggere il sistema. Inoltre CORS e API *key* affrontano minacce diverse, seppur sempre controllando l'accesso limitato alla piattaforma, ed insieme forniscono quindi una copertura più completa.
In conclusione, l'adozione di CORS e API *key* ha significativamente migliorato la sicurezza della piattaforma di RAG, garantendo un accesso controllato e protetto alle sue funzionalità.

Esposizione delle API

Utilizzando Flask e Waitress, la piattaforma di RAG espone diverse API per consentire l'interazione da parte del *backend* con le funzionalità implementate.

```
@app.route('/answer', methods=['POST'])
def answer_user_question():
    try:
        jsonRequest = request.get_json()
    except:
        return jsonify({'error': 'Formato della richiesta non valido'}), 400
    if 'question' not in jsonRequest:
        return jsonify({'error': 'Nella richiesta non è presente il campo
                        +'"question" relativo alla domanda'}), 400
    if 'user_data' not in jsonRequest:
        return jsonify({'error': 'Nella richiesta non è presente il campo
                        +'"user_data" relativo ai dati dell\'utente'}), 400
    question = jsonRequest.get("question")
    user_data = jsonRequest.get("user_data")
    return jsonify({'response': answer.answer_question(question,
                                                       user_data)}), 200
```

Codice 3.18: Funzione associata ad un *endpoint* esposto dalla piattaforma di RAG

Nel frammento di codice 3.18 si può osservare l'implementazione di uno degli *endpoint* disponibili, il quale consente di ottenere risposte personalizzate e dettagliate a

domande specifiche poste dagli utenti. Quando il *backend* invia una richiesta HTTP *POST* a `"/answer"`, contenente un corpo JSON con la domanda formulata dall'utente e dati aggiuntivi dell'utente necessari per contestualizzare e personalizzare la risposta, allora il *server* prima autentica la richiesta, poi verifica che la richiesta abbia il formato corretto e contenga entrambi i campi obbligatori, ed infine utilizza il modulo `answer` per elaborare la domanda ricevuta, restituendo la risposta generata in formato JSON.
In generale, gli *endpoint* esposti della piattaforma RAG possono restituire diverse risposte in base all'esito dell'elaborazione delle richieste effettuate. Di seguito sono descritti i possibili tipi di risposte che gli *endpoint* possono generare:

- **200 OK**: questo tipo di risposta indica che la richiesta è stata elaborata correttamente e ha prodotto un risultato valido;
- **400 *Bad Request***: questa risposta viene restituita quando la richiesta inviata dal *backend* non può essere elaborata a causa di problemi nel formato o nella struttura della richiesta stessa;
- **401 *Unauthorized***: questa risposta indica che la richiesta non è stata autorizzata poichè non ha superato le misure di sicurezza imposte;
- **500 *Internal Server Error***: questo tipo di risposta viene restituito quando si verifica un errore durante l'elaborazione della richiesta sul lato del *server*. Può essere causato da problemi temporanei come l'impossibilità di interagire con API di terze parti.

Chunking

Come precedentemente accennato nella sotto-sezione 3.1.3, una fase fondamentale nel processo di caricamento della conoscenza attraverso una piattaforma di RAG è il *chunking*. Prima di integrare i dati recuperati nel *database* vettoriale, è infatti essenziale suddividerli in unità più gestibili, denominate *chunk*, che mantengano un significato compiuto. Successivamente, questi *chunk* vengono convertiti in rappresentazioni vettoriali utilizzando tecniche di *embedding* e memorizzati nel *database*.

Esistono diversi approcci al *chunking* del testo, ognuno con i suoi vantaggi e svantaggi. All'interno del prototipo di piattaforma di RAG sviluppato, per esempio, ho utilizzato il *chunking* a dimensione fissa, caratterizzato dall'estrema semplicità di implementazione. Questo approccio prevede la suddivisione del testo in *chunk* di dimensioni predeterminate, come ad esempio 1024 caratteri. Ciò facilita notevolmente la gestione e l'elaborazione dei dati durante il caricamento nel *database* vettoriale, tuttavia l'uso di *chunk* di dimensioni fisse può portare a una gestione inefficiente dei dati quando la lunghezza dei testi varia notevolmente. Suddividere il testo in *chunk* senza tener conto del significato di ciascuno può portare infatti alla perdita del contesto che potrebbe essere cruciale per l'analisi o l'elaborazione successiva dei dati.

Proprio per questo motivo, quando ho implementato il modulo `upload`, responsabile della gestione del caricamento di *file* PDF all'interno della piattaforma di RAG, ho dovuto considerare attentamente come applicare il *chunking*.

```
def make_chuncks(pages):
    text_splitter = SemanticChunker(common.embeddings_model)
    return text_splitter.split_documents(pages) #chunk
```

Codice 3.19: Funzione per effettuare il *chunking* delle pagine dei documenti PDF caricati

Nel frammento di codice 3.19 si può osservare l'approccio da me scelto. Esso non prevede l'implementazione diretta del *chunking*, ma si basa sull'utilizzo di un modulo di LangChain che se ne occupa al posto mio. Questa scelta è motivata dal desiderio di affidarsi a una soluzione già ben ottimizzata per gestire l'operazione di *chunking* in modo efficace e preciso. Il modulo `SemanticChunker` si occupa di suddividere i documenti in *chunk*, ciascuno dei quali rappresenta un'unità di testo significativa, e per farlo sfrutta un modello di *embedding* comune che suddivide i *chunk* in base al loro significato e non alla loro lunghezza. Questo approccio non solo semplifica il processo di sviluppo, riducendo il rischio di errori nell'implementazione del *chunking*, ma sfrutta anche le migliori pratiche e le tecniche avanzate integrate in Langchain.

Recupero di informazioni pertinenti

Quando la piattaforma di RAG deve andare a generare una risposta di qualsiasi tipo, il recupero di informazioni pertinenti è un processo cruciale per garantire che le risposte generate dal sistema siano accurate, rilevanti e tempestive.

```
def create_context(query):
    docs_chroma = get_loaded_DB().similarity_search_with_score(query, k=10)
    return "\n\n".join([doc.page_content for doc in docs_chroma]) #context
```

Codice 3.20: Funzione per creare il contesto attinente ad una *query*

Nel frammento di codice 3.20 viene mostrato il processo di creazione del contesto rilevante per una data *query* utilizzando il *database* vettoriale scelto. La funzione riceve come *input* la domanda o il tema per cui occorre ottenere informazioni rilevanti e, attraverso un'instanza del *database* vettoriale caricato, esegue una ricerca di similarità, cercando i 10 *chunk* più rilevanti, ovvero più vicini rispetto al vettore della *query*. LangChain, per impostazione predefinita, utilizza la metrica di distanza coseno per misurare la similarità tra vettori, la quale consente di trovare i *chunk* di testo nel *database* che sono più simili alla *query* passata come argomento. Dopo aver recuperato i *chunk* rilevanti dal *database*, il contesto è costruito concatenando i contenuti di quest'ultimi in una singola stringa di testo, che viene poi utilizzata all'interno del *prompt* finale per generare una risposta completa alla richiesta dell'utente.

Creazione del *prompt* finale

La creazione del *prompt* finale rappresenta l'ultimo passaggio fondamentale prima di interrogare il modello linguistico nel processo di generazione delle risposte da parte della piattaforma, poiché permette di combinare la *query* dell'utente con il contesto rilevante recuperato dal *database*. Questo processo garantisce che il LLM possa fornire risposte precise e pertinenti, sfruttando sia le informazioni fornite dall'utente sia quelle contenute nel *database*.

```
def create_prompt_from_template_to_answer(context_text, query, user_data):
    PROMPT_TEMPLATE = """
    I dati dell'utente a cui stai rispondendo sono i seguenti:
    {user_data}
    Rispondi alla sua domanda basandoti sul seguente contesto:
    {context}
    La domanda a cui devi rispondere basandoti sul contesto è la seguente:
    {question}
    Fornisci una risposta dettagliata.
    Non giustificare la tua risposta.
    Rispondi in italiano.
    Evita frasi che citino il fatto che tu stia usando un contesto.
    Non inserire note.
    """
    prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
    return prompt_template.format(context=context_text, question=query,
                                  user_data=user_data) #prompt
```

Codice 3.21: Esempio di creazione del *prompt* finale

Nel frammento di codice 3.21 si può osservare un esempio di creazione del *prompt* finale da me implementato, il quale combina il contesto rilevante, la *query* dell'utente e i dati dell'utente in un formato predefinito. Questo *prompt* viene poi utilizzato dal modello linguistico per generare una risposta dettagliata e pertinente alla domanda posta dall'utente attraverso il *chatbot*.
Come avviene nell'esempio riportato, il *template* deve essere strutturato per fornire tutte le informazioni necessarie al LLM in un formato chiaro e conciso e può inoltre imporre anche delle regole specifiche per la risposta, come ad esempio il fatto di rispondere senza inserire note.

Generazione di *output* in formato JSON

Per poter generare piani di allenamento o alimentari personalizzati per l'utente, ho dovuto implementare delle funzioni che interrogassero il modello linguistico e richiedessero risposte strutturate in formato JSON. Questo approccio permette di ottenere dati organizzati in una struttura facilmente interpretabile e manipolabile dal sistema, facilitando così l'integrazione con altre componenti e migliorando l'esperienza utente complessiva.

```
def generate_diet_json(diet_data, user_data):
    context_text = create_context(diet_data)
    parser = JsonOutputParser(pydantic_object=classes.PianoAlimentare)
    prompt = create_prompt_from_template_to_generate_diet_json(context_text,
                user_data, diet_data, parser.get_format_instructions())
    for attempt in range(MAX_RETRIES):
        try:
            answer = generate_answer(prompt)
            parsed_answer = parser.parse(answer)
            if check_for_null_fields(parsed_answer):
                return True, parsed_answer
            else:
                continue
        except:
            continue
    return False, "Non è stato possibile generare il piano alimentare "
                  + "richiesto.\n Si prega di riprovare più tardi."
```

Codice 3.22: Esempio di funzione per la generazione di *output* in formato JSON

Nel frammento di codice 3.22 ho riportato la funzione `generate_diet_json`, la quale sfrutta diverse altre funzioni per creare il contesto, generare il *prompt* finale, ottenere una risposta dal modello linguistico e infine parsare e validare questa risposta. All'interno delle varie operazioni svolte, va sottolineato l'uso di `JsonOutputParser`, il quale utilizza un oggetto Pydantic (`classes.PianoAlimentare`) per definire la struttura JSON del piano alimentare. Inizialmente, questo *parser* fornisce le istruzioni sul formato della risposta, le quali vengono incorporate nel *prompt* finale. Successivamente, il *parser* viene utilizzato per validare e trasformare la risposta del modello linguistico in un formato JSON strutturato, assicurando che i dati restituiti siano conformi alle specifiche previste. Inoltre, occorre notare come sia presente un ciclo che tenti più volte di ottenere una risposta valida dal modello linguistico. Ad ogni iterazione, la funzione prova a parsare la risposta in un oggetto JSON utilizzando il *parser* definito in precedenza e controlla se ci sono campi nulli nella risposta parsata. Se tutti i campi richiesti sono presenti e non nulli, la funzione restituisce il piano alimentare, altrimenti, se si verifica un errore durante la generazione o il *parsing* della risposta, il ciclo continua fino a esaurire il numero massimo di tentativi, per poi restituire un messaggio di errore indicando che non è stato possibile generare il piano alimentare richiesto e invitando l'utente a riprovare più tardi.

3.6 Verifica e validazione

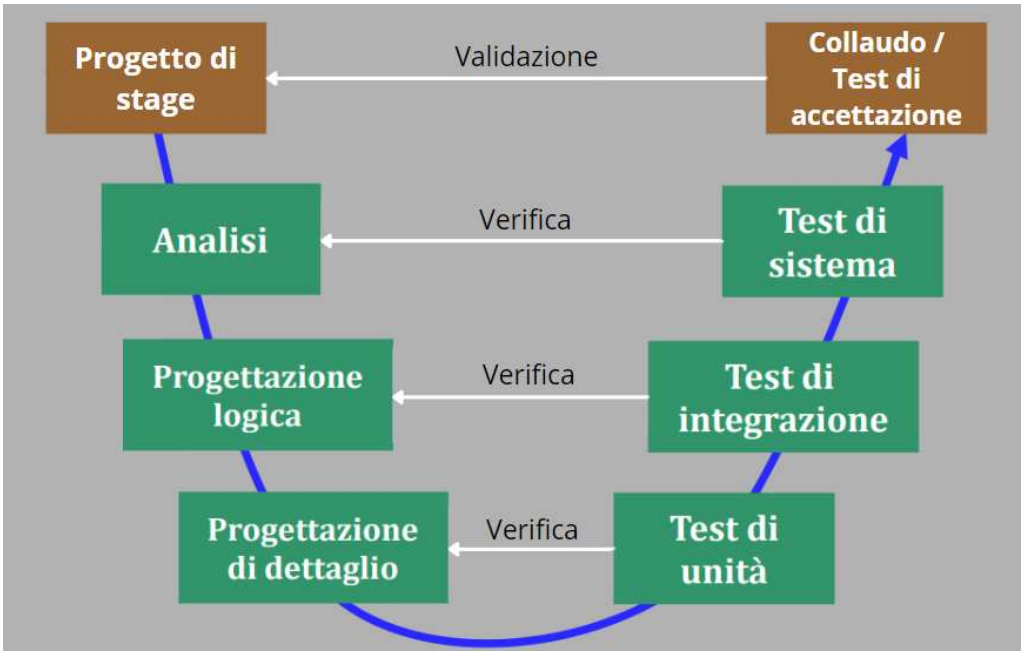


Figura 3.18: Modello a V delle tipologie di *test* eseguiti

Le attività di verifica e validazione sono di fondamentale importanza nel ciclo di vita dello sviluppo *software*, poiché garantiscono che il prodotto finale sia conforme ai requisiti specificati e sia privo di errori significativi. In questa sezione, descrivo i *test* e gli strumenti utilizzati per verificare e validare le diverse componenti del prodotto finale, includendo il *backend*, il *frontend* e la piattaforma di RAG.

Test di unità

I *test* di unità rappresentano un componente essenziale della strategia di verifica e validazione, poiché consentono di assicurare il corretto funzionamento delle singole componenti del *software* in isolamento. Nel progetto, ho applicato tali *test* a tutte le componenti del sistema, garantendo che ciascuna parte dell'architettura implementata soddisfacesse i requisiti specificati e funzionasse come previsto.
Il *backend* ha beneficiato dell'uso degli strumenti JUnit e Mockito per implementare i *test* di unità. JUnit ha fornito un *framework* robusto per eseguire *test* automatizzati, mentre Mockito ha permesso di simulare le dipendenze, rendendo possibile

il *test* dei metodi delle diverse classi in condizioni controllate. Questa combinazione ha garantito che le unità di codice fossero testate in isolamento, identificando e risolvendo rapidamente eventuali problemi.
Per il *frontend* ho impiegato Jasmine e Karma per testare unità su componenti, servizi, direttive e *pipe*. Jasmine, un *framework* di *testing* per JavaScript, ha fornito una sintassi semplice e intuitiva per scrivere i *test*, mentre Karma ha automatizzato l'esecuzione di quest'ultimi. Questo approccio ha assicurato che ogni parte dell'interfaccia utente funzionasse correttamente in isolamento, migliorando la qualità del codice e la *user experience*.
Infine, per la piattaforma di RAG ho utilizzato il *framework* di *testing* unittest, insieme a unittest.mock per effettuare il *mock* delle dipendenze, e il *framework* flask-testing per testare l'applicazione Flask. Unittest ha fornito una struttura per organizzare e eseguire i *test*, mentre unittest.mock ha permesso di isolare le unità di codice simulando le dipendenze esterne. Flask-testing, invece, ha facilitato il *testing* delle funzionalità specifiche di Flask, come la gestione delle richieste HTTP. Questo approccio ha garantito che la piattaforma rispondesse correttamente alle diverse richieste e scenari di utilizzo.

Test di integrazione

Oltre ai *test* di unità, anche i *test* di integrazione rappresentano una parte cruciale della strategia di verifica e validazione, garantendo che tutti i componenti interagiscano tra loro correttamente come un sistema integrato.
Visto il poco tempo a disposizione, il *team* di sviluppo ha deciso di focalizzarsi principalmente sul *testing* dell'interazione tra il *backend* e la piattaforma di RAG, poichè questa comunicazione critica tra le due componenti dell'architettura assicura il corretto funzionamento delle funzionalità principali. Concentrandoci su questa area, abbiamo potuto garantire che il flusso di dati tra il *backend* e la piattaforma di RAG avvenisse senza problemi.
Gran parte dei *test* di integrazione implementati si concentra sulla verifica delle API e dei servizi offerti dall'applicazione. Per farlo, abbiamo utilizzato PyTest come *framework* di *test*, il quale permette di automatizzare l'esecuzione dei *test* e di verificare che le diverse componenti del sistema interagiscano correttamente.

Questo approccio non solo ci ha aiutato a identificare e risolvere tempestivamente eventuali problemi di comunicazione o di integrazione tra le componenti, ma ci ha anche permesso di assicurarci che l'applicazione funzioni in modo coeso e conforme ai requisiti specificati.

Test di sistema e di accettazione

Infine, a completare le tipologie di *test* effettuati e riportati nella Figura 3.18, troviamo i *test* di sistema e di accettazione, mirati a garantire che il prodotto finale sia conforme ai requisiti definiti e che soddisfi le aspettative degli utenti finali. Abbiamo eseguito questi *test* per verificare che l'intero sistema, inclusi tutti i suoi componenti, funzioni come previsto e risponda ai requisiti specificati.
Questi *test*, svolti alla presenza del tutor aziendale, sono stati necessari per la validazione del prodotto finale, convalidando la corretta integrazione tra le diverse parti del sistema, accertandosi che tutte le funzionalità chiave siano testate attraverso scenari di utilizzo realistici e verificando che l'applicazione soddisfi i criteri di accettazione stabiliti dagli *stakeholder*.

3.7 Risultati raggiunti

In questa sezione espongo i risultati ottenuti al termine del progetto di *stage*, suddividendoli in due dimensioni principali: qualitativa e quantitativa.

3.7.1 Piano qualitativo

Dal punto di vista qualitativo, il sistema finale prodotto offre una soluzione integrata per l'assistenza *fitness* attraverso una *web app* completa e *user-friendly*. Al suo interno l'utente può registrarsi, tracciare i propri progressi, gestire esercizi, allenamenti e periodi di allenamento, ma soprattutto può interagire con un *chatbot* intelligente, in grado di comprendere e rispondere efficacemente alle sue richieste in ambito *fitness*. Inoltre, all'utente finale viene offerta pure la possibilità di generare automaticamente allenamenti e piani alimentari personalizzati, creati in base al suo profilo, ai suoi progressi, ai suoi obiettivi e alla descrizione fornita nel momento della generazione.

Tutte queste funzionalità sono offerte tramite un'interfaccia grafica progettata per essere intuitiva e facilmente navigabile. La UI è responsiva e garantisce una fruizione ottimale su diversi dispositivi, inclusi *desktop*, *tablet* e cellulari. Ogni elemento dell'interfaccia è studiato per massimizzare l'usabilità, con un *design* pulito e accessibile che facilita l'interazione dell'utente con l'applicazione.

Infine, grazie all'uso di tecnologie moderne e avanzate, abbiamo costruito la *web app* per garantire *performance* elevate, sicurezza e scalabilità.

Nelle Figure 3.19, 3.20 e 3.21 si possono osservare alcune immagini del prodotto finale, a dimostrazione delle funzionalità descritte e della qualità dell'interfaccia utente.

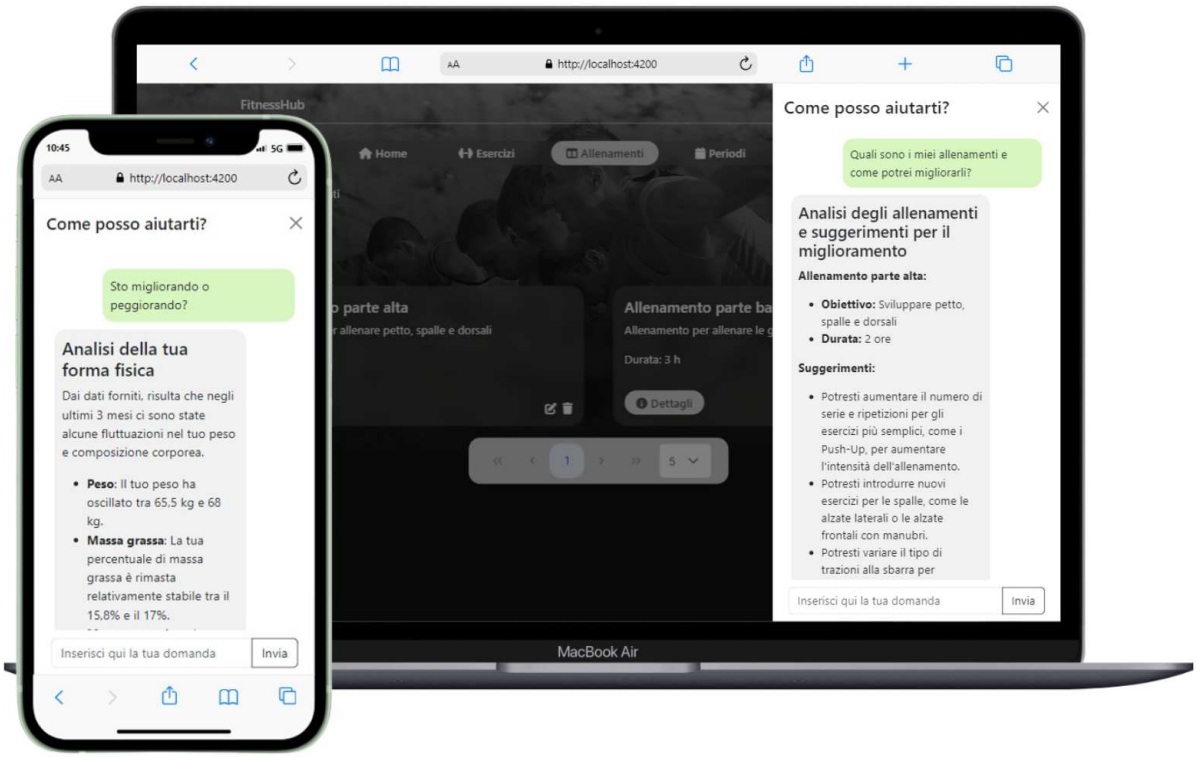


Figura 3.19: Immagini del *chatbot* in uso

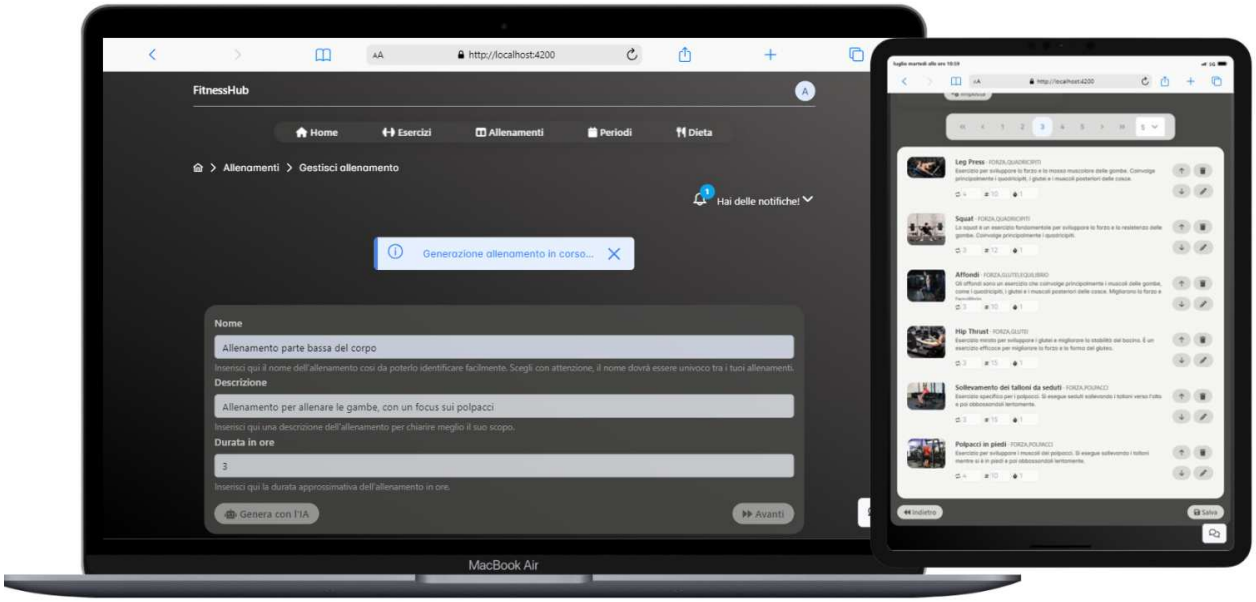


Figura 3.20: Immagini della generazione di un piano di allenamento personalizzato

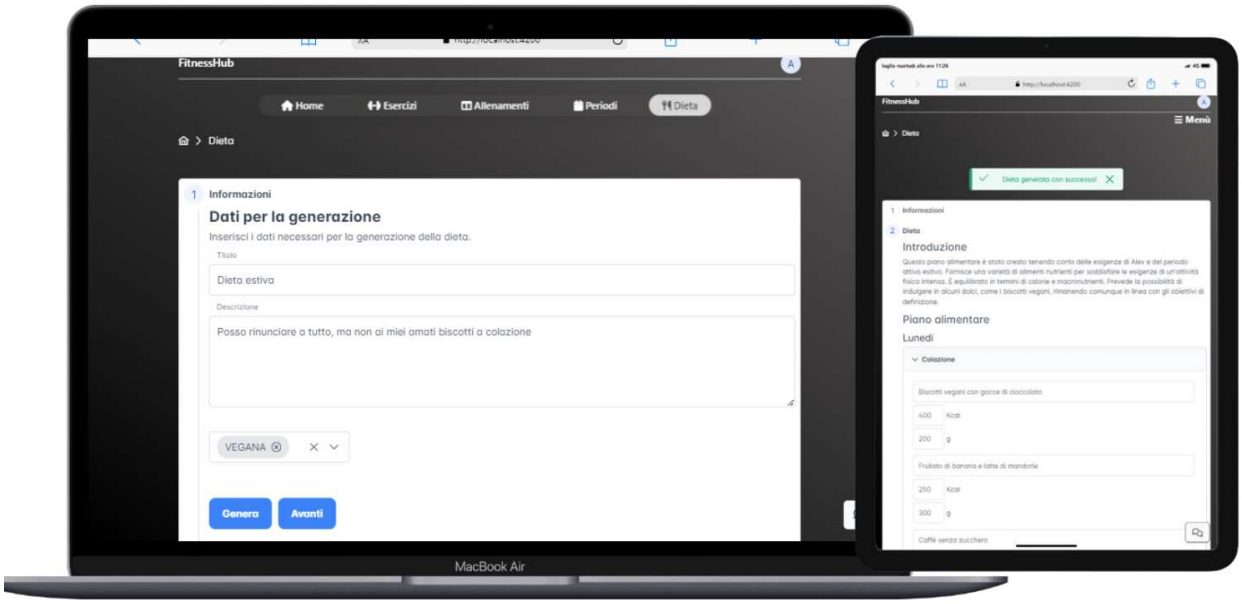


Figura 3.21: Immagini della generazione di un piano alimentare personalizzato

In sintesi, il sistema finale rappresenta una soluzione innovativa e all'avanguardia nel campo dell'assistenza *fitness*, combinando funzionalità avanzate, un'interfaccia *user-friendly* e un'architettura tecnica solida. Questo progetto non solo risponde alle esigenze attuali degli utenti, ma pone anche le basi per futuri sviluppi e miglioramenti.

3.7.2 Piano quantitativo

Sul piano quantitativo, i risultati sono misurati attraverso la copertura dei requisiti, la copertura dei *test* e la quantità di prodotti realizzati.

Copertura dei requisiti e dei *test*

Al termine del progetto, posso affermare di aver coperto con successo tutti i requisiti, funzionali e non funzionali, identificati nell'attività di analisi.
Inoltre ho contribuito a scrivere numerosi *test* di unità ed alcuni *test* di integrazione, assicurandomi che ogni singola parte del sistema funzionasse come previsto in isolamento e non solo. Grazie a questi *test*, è stato possibile raggiungere una copertura del codice media attorno al 95%, garantendo che tutte le funzionalità principali fossero adeguatamente verificate. Nella Figura 3.22 si mostra come esempio il *report* di copertura del codice derivante dai *test* che ho effettuato sulla piattaforma di RAG, la quale raggiunge addirittura una copertura pari al 100%.

File ▲	statements	missing	excluded	coverage
src/__init__.py	0	0	0	100%
src/answer.py	72	0	0	100%
src/classes.py	28	0	0	100%
src/common.py	5	0	0	100%
src/main.py	74	0	2	100%
src/upload.py	16	0	0	100%
Total	**195**	**0**	**2**	**100%**

Figura 3.22: *Report* di copertura del codice dei moduli che compongono la piattaforma di RAG

Questa elevata copertura ci ha permesso di identificare e correggere tempestivamente eventuali *bug* e problemi, assicurando inoltre che il sistema soddisfacesse tutti i requisiti funzionali e non funzionali.
In conclusione, la copertura dei requisiti e dei *test* è stata completa e approfondita, assicurando che ogni aspetto del sistema fosse accuratamente verificato e conforme alle richieste specificate. Questo approccio ha garantito un prodotto finale di alta qualità, affidabile e pronto per l'uso da parte degli utenti.

Quantità di prodotti realizzati

Durante lo sviluppo del progetto, ho contribuito a realizzare una quantità significativa di prodotti. Il codice sorgente scritto per coprire tutte le funzionalità del sistema, dal *backend* al *frontend*, passando per la piattaforma di RAG, corrisponde all'incirca a dieci mila linee di codice totali, andando a contare anche i *test* definiti, i quali sono circa un centinaio o poco più. Inoltre ho realizzato diversi documenti, circa una decina, da fornire alla piattaforma di RAG in modo da poter potenziare il *chatbot* per soddisfare i requisiti individuati.
In sintesi, la quantità di prodotti generati durante lo sviluppo del progetto è stata considerevole, riflettendo l'ampiezza e la complessità del sistema realizzato. Questi prodotti, combinati tra loro, hanno contribuito a creare una soluzione robusta e di alta qualità.

Capitolo 4

Retrospettiva

4.1 Raggiungimento degli obiettivi

Nel corso del progetto di *stage* ho perguito con impegno e metodo il raggiungimento dei diversi obiettivi prefissati, sia quelli dell'organizzazione che quelli personali. In questa sezione presento una valutazione dettagliata e ragionata dei risultati conseguiti in relazione agli obiettivi stabiliti, evidenziando come ciascuno di essi abbia contribuito al successo complessivo del progetto.

4.1.1 Obiettivi aziendali

4.1.1.1 Obiettivi obbligatori

- **Obiettivo**: Acquisizione di competenze approfondite sullo stato dell'arte delle piattaforme attuali di RAG/LLM, dimostrabili attraverso il prodotto finale ed i prototipi realizzati;

  Risultato: Ho pienamente raggiunto questo obiettivo. Infatti, come già spiegato in precedenza, durante il progetto ho esplorato e valutato diversi LLM per lo sviluppo finale della piattaforma di RAG, acquisendo una conoscenza approfondita delle loro funzionalità, limitazioni e potenzialità. Ho poi dimostrato chiaramente le competenze acquisite nel prodotto finale e nei prototipi sviluppati, i quali integrano le migliori pratiche e soluzioni tecnologiche disponibili tenendo conto dei vincoli imposti dal progetto;

- **Obiettivo**: Acquisizione di competenze riguardanti le tecnologie moderne per lo sviluppo di applicazioni *web*, in particolare Spring ed Angular, dimostrabili attraverso il prodotto finale ed i prototipi realizzati;

  Risultato: Ho raggiunto anche questo obiettivo con successo. L'uso dei *framework* Spring ed Angular nel progetto ha dimostrato una solida padronanza di entrambe le tecnologie. Il prodotto finale e i prototipi sviluppati sono infatti testimonianze tangibili dell'acquisizione e dell'applicazione efficace di queste competenze tecniche;

- **Obiettivo**: Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il cronoprogramma;

  Risultato: Ho pienamente raggiunto l'obiettivo. Ho infatti completato il progetto, compreso di obiettivi non obbligatori, rispettando il cronoprogramma stabilito, dimostrando un'efficace gestione del tempo e delle risorse, nonché una significativa capacità di lavorare in autonomia;

- **Obiettivo**: Integrazione tramite relative API di una piattaforma di RAG/LLM in una *fitness web app* con lo scopo di realizzare un *chatbot* in grado di fornire risposte adeguate alle richieste degli utenti, considerando anche il loro profilo personale;

  Risultato: Ho raggiunto con successo anche quest'ultimo obiettivo obbligatorio. Come già evidenziato, la *web app* che ho contribuito a sviluppare contiene infatti un *chatbot* in grado di fornire risposte appropriate alle richieste degli utenti, tenendo conto anche del loro profilo personale. Questa funzionalità è stata testata e validata, confermando la sua efficacia nel migliorare l'esperienza degli utenti all'interno della *fitness web app*.

4.1.1.2 Obiettivi facoltativi

- **Obiettivo**: Implementazione di funzionalità aggiuntive nella *web app* per consentire la creazione automatica di piani di allenamento personalizzati per gli utenti tramite l'interazione con una piattaforma di RAG/LLM;

Risultato: Ho pienamente raggiunto pure questo obiettivo. Ho infatti sviluppato e integrato con successo nella *web app* una funzione per la generazione automatica di piani di allenamento personalizzati tramite interazione con la piattaforma di RAG sviluppata. Questa funzionalità è operativa e offre un significativo valore aggiunto all'utente finale.

4.1.1.3 Obiettivi desiderabili

- **Obiettivo**: Implementazione di funzionalità aggiuntive nella *web app* per consentire la creazione automatica di piani alimentari personalizzati per gli utenti tramire l'interazione con una piattaforma di RAG/LLM;

  Risultato: Ho raggiunto anche questo obiettivo, seppur desiderabile. Ho infatti sviluppato e integrato con successo nella *web app* la funzionalità per la creazione automatica di piani alimentari personalizzati. Questa aggiunta arricchisce ulteriormente l'esperienza dell'utente, offrendo una soluzione completa e personalizzata per le esigenze di *fitness* e nutrizione.

4.1.2 Obiettivi personali

- **Obiettivo**: Migliorare le mie capacità di comunicazione e collaborazione;

  Risultato: Ho pienamente raggiunto l'obiettivo. Difatti, come sottolineato a fine *stage* dall'altro membro del *team* e dal tutor aziendale, ho migliorato notevolemente le mie capacità di comunicazione e collaborazione attraverso le riunioni periodiche con entrambi, dove ho dimostrato un significativo miglioramento nella chiarezza delle mie comunicazioni e nella capacità di dare e ricevere *feedback* costruttivi;

- **Obiettivo**: Crescere come sviluppatore *software* acquisendo nuove competenze tecniche;

  Risultato: Ho raggiunto con successo l'obiettivo, e le sezioni 3.1 e 4.2 ne sono la dimostrazione. Attraverso l'uso di diverse e molteplici fonti, la realizzazione di diversi prototipi e dell'applicazione finale, e le revisioni effettuate con il

tutor aziendale, ho acquisito moltissime nuove competenze che mi hanno reso uno sviluppatore migliore e mi hanno permesso di crescere professionalmente;

- **Obiettivo**: Esplorare nuove tecnologie (Spring, Angular, LLM e RAG);

  Risultato: Ho pienamente raggiunto pure questo obiettivo, e la sezione 3.1 ne è la prova, riportando i punti chiave appresi per ciascuna delle tecnologie di interesse. Per di più, i prototipi realizzati ed il prodotto finale sono un'ulteriore dimostrazione del mio percorso di esplorazione;

- **Obiettivo**: Contribuire allo sviluppo tecnologico dell'azienda.

  Risultato: Ho raggiunto anche quest'ultimo ambizioso obiettivo. SyncLab era infatti molto interessata ad affacciarsi al mondo LLM e RAG, ed il mio percorso di *stage* è stato un ottimo campo di prova per l'azienda per poter valutare l'adozione di queste tecnologie all'interno dei loro prodotti. I risultati conseguiti, evidenziando come queste tecnologie possano migliorare il servizio clienti automatizzato, riducendo significativamente i tempi di risposta e aumentando la soddisfazione dei clienti, hanno convinto l'organizzazione ad investire in questo nuovo campo, basandosi anche sulle conoscenze che ho potuto trasmettergli attraverso il mio *stage*.

4.1.3 Conclusioni

In sintesi, ho raggiunto con successo tutti gli obiettivi del progetto, sia quelli aziendali che quelli personali. Gli obiettivi obbligatori hanno dimostrato le capacità di acquisire nuove competenze, sviluppare soluzioni tecnologiche avanzate e rispettare le tempistiche stabilite, mentre gli obiettivi facoltativi e desiderabili hanno contribuito a una significativa estensione delle funzionalità offerte. Gli obiettivi personali, invece, mi hanno permesso di crescere. Il progetto, quindi, ha non solo soddisfatto i requisiti minimi indispensabili, ma ha anche gettato ottime basi per il futuro.

4.2 Crescita professionale

Il completamento del percorso di *stage* ha rappresentato un significativo punto di svolta nel mio percorso professionale. Infatti, durante lo *stage* ho sviluppato una maggiore autonomia nella gestione delle attività lavorative quotidiane e dei progetti a lungo termine. Questa autonomia mi ha insegnato l'importanza della pianificazione e della gestione efficace del tempo, rendendomi più abile nel fissare priorità e nel rispettare le scadenze, qualità che sono essenziali per mantenere alta la produttività in qualsiasi contesto lavorativo.

Inoltre, l'affrontare le sfide tecniche e logistiche del progetto mi ha reso più competente nel *problem solving* e più adattabile ai cambiamenti. Ho imparato a valutare rapidamente le situazioni, a identificare le possibili soluzioni e a implementarle efficacemente. Questa esperienza mi ha reso più resiliente e pronto ad affrontare situazioni inaspettate, caratteristiche fondamentali in un ambiente di lavoro dinamico.

Dopo ancora, la necessità di collaborare con colleghi e tutor aziendale durante lo *stage* mi ha aiutato a migliorare le mie capacità di comunicazione e di lavoro in *team*. Questa esperienza ha affinato le mie capacità di interazione professionale, rendendomi un membro di *team* migliore e contribuendo a un ambiente di lavoro più produttivo e armonioso.

Allo stesso tempo, le responsabilità assunte durante lo *stage* mi hanno permesso di sviluppare capacità di *leadership*. Ho imparato a prendere decisioni, a guidare, almeno in parte, progetti e a ispirare gli altri a raggiungere i loro obiettivi. Queste competenze di *leadership* sono fondamentali per il mio sviluppo professionale e mi preparano per ruoli di maggiore responsabilità in futuro.

Infine, lo *stage* mi ha offerto l'opportunità di ampliare il mio *network* professionale, incontrando e collaborando con professionisti del settore. Questo *network* non solo mi ha fornito risorse e supporto, ma rappresenta anche una piattaforma per future collaborazioni e opportunità di carriera, costruendo preziose relazioni per il mio sviluppo professionale a lungo termine.

In conclusione, l'esperienza di *stage* ha avuto un impatto profondo sulla mia crescita professionale, affinando le mie capacità di gestione del tempo, *problem solving*, comunicazione e *leadership*, e preparandomi per affrontare con successo le sfide future.

Questa crescita non solo ha arricchito il mio percorso professionale, ma ha anche consolidato la mia determinazione a continuare a migliorarmi e a eccellere nel mio campo. Le lezioni apprese e le relazioni costruite durante lo *stage* rappresentano una solida base per il mio futuro professionale, offrendo numerose opportunità di crescita e successo.

4.3 Analisi critica finale

Durante lo *stage* ho avuto modo di confrontare le competenze richieste per lo svolgimento del lavoro con quelle acquisite durante il mio percorso di studi. Questo confronto mi ha permesso di identificare sia i punti di forza della mia preparazione, sia le lacune che sarebbe utile colmare per affrontare meglio il mondo del lavoro e le sfide professionali future.

Il mio *stage* ha evidenziato una buona corrispondenza tra le competenze fornite dal corso di studi e quelle necessarie per affrontare con successo il mondo del lavoro. La formazione accademica mi ha dotato infatti di una solida base teorica e pratica, e soprattutto della capacità di autoapprendere, un'abilità cruciale per adattarmi e crescere professionalmente. Tuttavia, nonostante l'eccellente preparazione fornita dal corso di studi, per colmare completamente il divario tra formazione accademica e contesto lavorativo, ritengo che sarebbe utile integrare maggiormente tecnologie più contemporanee nel *curriculum* degli studi.

Difatti, sebbene l'autoapprendimento si sia rivelato particolarmente utile durante lo *stage*, permettendomi di colmare eventuali lacune per lo svolgimento del lavoro, penso che, ad esempio, l'integrazione di tecnologie come React o Angular nel *curriculum* accademico potrebbe preparare meglio gli studenti alle sfide del mercato del lavoro contemporaneo.

Glossario

API Un'*Application Programming Interface* è un insieme di regole e protocolli che permette a diverse applicazioni *software* di comunicare tra loro. Le API definiscono i metodi e i dati che le applicazioni possono utilizzare per interagire con altri *software*, servizi o componenti, facilitando l'integrazione e l'interoperabilità tra sistemi diversi. 18

Applicazione *enterprise* Le applicazioni *enterprise* sono *software* progettati e sviluppati per soddisfare le esigenze di grandi organizzazioni o aziende, spesso con complessi requisiti di *business* e un alto volume di utenti e dati. Queste applicazioni sono caratterizzate da funzionalità avanzate, scalabilità, sicurezza e integrazione con altri sistemi aziendali. 13

Backlog Il *backlog* è una lista prioritaria di tutte le attività, funzionalità e requisiti che devono essere completati in un progetto, utilizzata principalmente nelle metodologie di sviluppo *agile* per organizzare e gestire il lavoro da svolgere. 4

Boilerplate code Il *boilerplate code* è codice *standard* e ripetitivo che viene utilizzato frequentemente in molteplici progetti o componenti *software* per eseguire funzioni comuni e di base. 13

Built-in Il termine *built-in* si riferisce a funzioni o caratteristiche integrate direttamente nel linguaggio di programmazione o nell'ambiente di esecuzione, disponibili per l'uso immediato senza la necessità di importare moduli o librerie aggiuntive o di definirle personalmente. 32

CD Il *continuous deployment* è una pratica di sviluppo *software* in cui le modifiche al codice, una volta superati i *test* automatizzati, vengono automaticamente

distribuite negli ambienti di produzione. Questo processo riduce al minimo l'intervento manuale, accelerando la consegna del *software* e garantendo che le nuove funzionalità e correzioni siano immediatamente disponibili agli utenti finali. 11

CI La *continuous integration* è una pratica di sviluppo *software* in cui gli sviluppatori integrano frequentemente il loro codice in un *repository* condiviso, idealmente più volte al giorno. Ogni integrazione viene verificata automaticamente mediante la costruzione del progetto e l'esecuzione di *test* automatizzati, permettendo di rilevare e risolvere rapidamente errori e conflitti nel codice. 11

Clean coding Un insieme di pratiche e principi per scrivere codice in modo leggibile, comprensibile e manutenibile. Promuove l'uso di nomi significativi, la semplicità delle funzioni, la riduzione della complessità e l'adesione a *standard* di formattazione coerenti. L'obiettivo è facilitare la comprensione e la manutenzione del codice da parte di altri sviluppatori. 29

Code base Una *code base* è l'insieme completo del codice sorgente di un *software*, inclusi tutti i file, le librerie e le risorse necessarie per costruire, compilare e gestire il progetto. 10

CORS *Cross-Origin Resource Sharing* è una politica di sicurezza implementata dai *browser* per permettere il caricamento di risorse da domini diversi rispetto a quello della pagina *web*. Consente ai *server* di dichiarare quali domini esterni possono accedere alle risorse, utilizzando intestazioni HTTP specifiche. Questo meccanismo aiuta a prevenire attacchi di tipo *Cross-Site Request Forgery* (CSRF) e *Cross-Site Scripting* (XSS), ovvero due tipi di vulnerabilità di sicurezza *web*. 76

DI La *dependency injection* è un principio di progettazione *software* che promuove la separazione delle dipendenze tra i componenti del sistema. Invece di creare le dipendenze all'interno di un oggetto stesso, esse vengono passate dall'esterno, solitamente tramite costruttori o metodi, consentendo una maggiore modularità e flessibilità del codice. L'obiettivo è quello di ridurre l'accoppia-

mento tra i componenti del *software*, semplificare la gestione delle dipendenze e favorire la testabilità e la manutenibilità del codice. 15

DTO Un *Data Transfer Object* è un oggetto utilizzato per trasferire dati tra le diverse parti di un'applicazione, in particolare tra il livello di presentazione (*controller*) e il livello di servizio. I DTO sono utili per incapsulare i dati e per separare le entità di dominio dalle rappresentazioni utilizzate nelle interazioni con il *client*. 65

Embedding Un *embedding* è una rappresentazione numerica vettoriale di oggetti, come parole o immagini, in uno spazio multidimensionale. Gli *embedding* permettono di catturare le relazioni semantiche tra gli oggetti, rendendo possibile il confronto e l'analisi di somiglianze in modo più efficiente. È comunemente usato nel *machine learning* e nel processamento del linguaggio naturale per trasformare parole, frasi o altri dati discreti in formati che gli algoritmi possono elaborare più facilmente. Gli *embedding* permettono di mantenere le relazioni semantiche tra i dati, facilitando compiti come classificazione, ricerca e *clustering*. 16

Entry point Un *entry point* è il punto di ingresso principale di un'applicazione o sistema, dove inizia l'esecuzione del codice. In un programma *software*, l'*entry point* è solitamente una funzione o un metodo specifico che viene eseguito per primo quando l'applicazione viene avviata. 31

ERP L'*Enterprise Resource Planning* è un sistema integrato di gestione aziendale che centralizza e coordina le diverse funzioni e processi di un'organizzazione. Un ERP consente di automatizzare e ottimizzare attività come contabilità, gestione delle risorse umane, *supply chain*, produzione, vendite e assistenza clienti. 2

Fine-tuned Si parla di modello *fine-tuned* quando un LLM generico viene esposto a un *set* di dati mirato per eseguire un compito associato ad un determinato dominio. In questa maniera il modello rifinisce i propri parametri interni

basandosi su questi dati specifici, migliorando la sua abilità in quel particolare dominio. 42

Google Cloud Platform *Google Cloud Platform* (GCP) è una *suite* di servizi di *cloud computing* offerti da Google, che fornisce infrastruttura, piattaforme e strumenti per l'elaborazione, l'archiviazione e l'analisi dei dati. GCP consente alle aziende di sviluppare, distribuire e scalare applicazioni e servizi su un'infrastruttura globale gestita da Google. 45

HTTP *HyperText Transfer Protocol* è un protocollo di comunicazione usato per trasferire dati su reti informatiche, principalmente nel contesto del *World Wide Web*. HTTP definisce come i messaggi vengono formattati e trasmessi, e come i *server* e i *browser* dovrebbero rispondere alle varie richieste. È la base per il funzionamento dei siti *web*, permettendo il trasferimento di ipertesti, immagini, video e altri tipi di contenuti. 13

ICT *Information and Communication Technology* è un termine che si riferisce all'insieme delle tecnologie utilizzate per gestire e comunicare informazioni. ICT include l'*hardware*, come *computer* e dispositivi di rete, e il *software* che li gestisce, nonché le infrastrutture di rete e i servizi di telecomunicazione che permettono la trasmissione e la ricezione dei dati. Questa tecnologia è fondamentale per la gestione efficiente delle informazioni e per la comunicazione in vari settori, facilitando la trasformazione digitale e l'innovazione. 1

IoC L'*Inversion of Control* è un principio di progettazione *software* in cui il controllo del flusso di esecuzione di un'applicazione viene spostato da un componente centrale a un *framework* o a un contenitore di inversione di controllo. Questo permette di separare le responsabilità e favorire una migliore modularità e manutenibilità del codice. In pratica, significa che anziché un componente chiamare direttamente un altro componente, il controllo di come e quando viene chiamato viene delegato a un'entità esterna. La *dependency injection* è una delle tecniche più comuni utilizzate per implementare l'IoC. 13

Issue Una *issue* è un'istanza registrata in un sistema di tracciamento, utilizzata per segnalare, discutere e risolvere problemi, *bug*, richieste di funzionalità o altre attività correlate a un progetto. Le *issue* forniscono un modo strutturato per tenere traccia dei problemi e delle attività da completare, consentendo ai membri del *team* di collaborare, assegnare, monitorare lo stato e commentare su ciascun problema. 7

JPA Java *Persistence* API è una specifica Java per la gestione della persistenza dei dati tra applicazioni Java e *database* relazionali. Permette di mappare oggetti Java a tabelle del *database*, facilitando le operazioni CRUD (*Create, Read, Update, Delete*) tramite un'API *standard*. 14

JPQL Java *Persistence Query Language* è un linguaggio di *query* orientato agli oggetti utilizzato per eseguire *query* su entità gestite dal contesto di persistenza in applicazioni Java. Simile a SQL, ma operante su entità Java invece che su tabelle del *database*, JPQL consente di scrivere *query* per recuperare dati da un *database* in modo indipendente dalla specifica implementazione del *database* stesso. 39

LLM Un *Large Language Model* è un modello di intelligenza artificiale addestrato su grandi quantità di testo per comprendere e generare linguaggio naturale. Questi modelli utilizzano reti neurali profonde con miliardi di parametri per eseguire una varietà di compiti linguistici, tra cui traduzione, riassunto, completamento di testo e conversazione. Gli LLM sono progettati per cogliere le complessità del linguaggio umano, fornendo risposte coerenti e contestualmente pertinenti alle domande e ai *prompt* degli utenti. 16

LLM-*as-Judge* LLM-*as-Judge* (LLM come giudice) si riferisce all'uso di un LLM per valutare o giudicare la qualità, la correttezza o l'adeguatezza di un testo generato, una traduzione, una risposta a una domanda o un altro *output* linguistico. Il modello scelto analizza il contenuto basandosi su criteri specifici e fornisce un giudizio o una valutazione, che può essere utilizzata per migliorare l'accuratezza del modello, per l'apprendimento supervisionato, o per automatizzare compiti di revisione e verifica. 48

Mapper Il *pattern mapper* è un *design pattern* utilizzato per trasformare oggetti di un tipo in oggetti di un altro tipo, solitamente per convertire tra entità di dominio e DTO. Questo *pattern* separa la logica di *mapping* dal resto dell'applicazione, facilitando la manutenzione, migliorando la leggibilità del codice e garantendo che le conversioni siano centralizzate e gestite in modo coerente. 65

Mock Un *mock* è un oggetto simulato utilizzato nei *test* per imitare il comportamento di componenti reali del *software*. Viene impiegato per isolare l'unità di codice in *test*, consentendo di verificare il suo funzionamento senza dipendere da altri componenti o risorse esterne come *database*, servizi *web* o *filesystem*. I *mock* possono essere programmati per restituire risposte specifiche, verificare chiamate a metodi e controllare le interazioni, facilitando la creazione di *test* precisi e affidabili. 18

Nearest neighbor search La *nearest neighbor search* (ricerca del vicino più prossimo) è una tecnica utilizzata per trovare il punto più vicino o i punti più vicini a un dato punto all'interno di un insieme di punti. È comunemente usata in ambito di intelligenza artificiale, *machine learning*, e sistemi di raccomandazione. Questa ricerca può essere implementata utilizzando vari algoritmi, i quali permettono di eseguire la ricerca in modo più efficiente rispetto a una semplice ricerca lineare. 43

Ollama Una piattaforma *open-source* che consente di eseguire e sperimentare modelli di intelligenza artificiale generativa in locale su *computer* con diversi sistemi operativi. 47

Open-source *Open-source* è un termine che si riferisce a *software* la cui licenza permette agli utenti di accedere, modificare e distribuire il codice sorgente liberamente. Questo modello promuove la collaborazione, l'innovazione e la trasparenza, consentendo a chiunque di contribuire e migliorare il *software*. 10

Opinionated *Opinionated* è un termine utilizzato per descrivere *software* che offrono una serie di convenzioni e configurazioni predefinite, indirizzando gli

sviluppatori verso un modo specifico di costruire e configurare applicazioni. Questo approccio semplifica e accelera il processo di sviluppo, riducendo la quantità di decisioni configurative dettagliate da parte degli sviluppatori. 14

POC Una *Proof of Concept* è una dimostrazione preliminare realizzata per verificare la fattibilità tecnica di un'idea o progetto, identificare potenziali problemi e valutare le soluzioni prima di cimentarsi nello sviluppo completo del prodotto finale. 55

Production-ready Si definisce *production-ready* un *software*, sistema o componente sufficientemente stabile, sicuro e ottimizzato per essere utilizzato in un ambiente di produzione reale. Questi sistemi soddisfano *standard* di qualità, affidabilità e prestazioni richiesti per l'uso operativo reale. 36

Pull request Una *pull request* è una richiesta per incorporare le modifiche di un ramo di sviluppo nel ramo principale di un *repository* Git. È utilizzato per la revisione e l'integrazione delle modifiche nel progetto. 7

Query methods I *query methods* sono metodi definiti nelle interfacce di *repository*, tipicamente nelle applicazioni che utilizzano il *framework* Spring Data, che permettono di eseguire *query* sul *database* in modo automatico. Questi metodi vengono costruiti seguendo una convenzione di denominazione basata sui nomi degli attributi delle entità, consentendo di eseguire operazioni di ricerca, aggiornamento, eliminazione e conteggio senza dover scrivere *query* SQL manualmente. 40

RAG La *Retrieval-Augmented Generation* costituisce un paradigma emergente per potenziare l'*output* dei LLM nell'ambito della generazione di testo. La RAG integra gli LLM con un sistema di recupero delle informazioni, consentendo loro di attingere a basi di conoscenza esterne al *corpus* di addestramento originale. Questo approccio permette agli LLM di generare risposte più accurate e contestualmente rilevanti, superando le limitazioni dei dati statici impiegati nel *training*. 10

Repository Un *repository* è un archivio centralizzato dove vengono memorizzati e gestiti i dati, i *file* o il codice sorgente di un progetto. Nei contesti di sviluppo *software*, un *repository* (spesso abbreviato in "repo") contiene la storia delle modifiche al codice, permettendo ai *team* di collaborare, tracciare versioni e gestire il controllo delle versioni del progetto. Piattaforme come GitHub, GitLab e Bitbucket sono esempi comuni di *repository* utilizzati per la gestione del codice sorgente. 7

REST *Representational State Transfer* è un'architettura per la progettazione di servizi *web* che utilizza i metodi HTTP *standard*, come GET, POST, PUT e DELETE, per operare su risorse rappresentate da URL. REST è ampiamente utilizzato per creare API ed è noto per la sua semplicità, scalabilità e indipendenza tra *client* e *server*. 27

***Server* WSGI** Un *server* WSGI (*Web Server Gateway Interface*) è un *server* che implementa il protocollo WSGI per servire applicazioni *web* scritte in Python. WSGI è uno *standard* Python che definisce come un *server web* comunica con applicazioni *web* o *framework*, permettendo la separazione tra il *server* e l'applicazione. 17

Software house Una *software house* è un'azienda specializzata nello sviluppo, nella progettazione e nella manutenzione di *software*. Queste aziende creano applicazioni, sistemi operativi, giochi, strumenti di produttività e altri tipi di *software* destinati a soddisfare esigenze specifiche di utenti, aziende o mercati. Le *software house* possono sviluppare prodotti *software* personalizzati per clienti specifici o produrre *software* standardizzati per il mercato di massa. 3

SOLID I principi SOLID sono un insieme di linee guida per lo sviluppo *software*, che mirano a rendere il codice più comprensibile, flessibile e manutenibile. Questi comprendono il *Single Responsibility Principle* (SRP), l'*Open/Closed Principle* (OCP), il *Liskov Substitution Principle* (LSP), l'*Interface Segregation Principle* (ISP) e il *Dependency Inversion Principle* (DIP). 29

SPA Una *Single Page Application* è un'applicazione *web* che carica una singola pagina HTML e aggiorna dinamicamente il contenuto man mano che l'uten-

te interagisce con l'app, senza ricaricare completamente la pagina. Questo approccio offre un'esperienza utente più fluida e veloce, simile a quella delle applicazioni *desktop*, grazie all'uso intensivo di JavaScript e AJAX per la gestione dei dati e degli aggiornamenti di interfaccia. 15

Stakeholder Uno *stakeholder* è una persona, gruppo o organizzazione che ha interesse o influenza in un progetto o azienda, e che può essere influenzato dai risultati del progetto o delle attività aziendali. Gli *stakeholder* possono includere clienti, dipendenti, fornitori, investitori, comunità locali e altri enti governativi. 5

Standalone Il termine *standalone* si riferisce a un'applicazione o sistema che può funzionare autonomamente, senza dipendere da altri *software* o risorse esterne. Questi programmi sono auto-sufficienti e possono essere eseguiti indipendentemente su un dispositivo. 36

System integrator Un *system integrator* è un'azienda o un professionista che specializza nell'unificare e far funzionare insieme diversi sistemi informatici e *software*, creando una soluzione integrata che risponda alle esigenze tecnologiche e operative di un'azienda. Il loro compito è analizzare le necessità del cliente, progettare, implementare e testare l'integrazione, garantendo che tutti i componenti funzionino armoniosamente, migliorando così l'efficienza e riducendo la complessità operativa. 1

Token In ambito LLM, un *token* è una singola unità di testo che il modello elabora. I *token* possono essere parole, parti di parole o anche caratteri, a seconda di come è stato addestrato il modello. Essi sono fondamentali per la comprensione e la generazione del linguaggio da parte del LLM, poiché esso lavora su sequenze di *token* per eseguire compiti come la traduzione, il completamento del testo e la risposta a domande. 46

User-friendly *User-friendly* significa facile da usare e intuitivo per l'utente. Si riferisce a prodotti, interfacce o sistemi progettati in modo che gli utenti possano interagire con essi senza difficoltà, comprendendo facilmente le funzioni e

le operazioni, riducendo la curva di apprendimento e migliorando l'esperienza complessiva. 6