Sviluppo di un Backend basato su un GraphDBMS per una app social

Lorenzo Macchiarini

Aprile 2022

1 Introduzione

In questo progetto ho voluto creare Litto: un'applicazione che aiuta l'utente sia a portare a termine i propri obiettivi in modo organizzato e temporizzato, sia a scoprirne di nuovi. L'idea iniziale del progetto era di fornire una possibile soluzione ad un problema frequente: iniziare un "obiettivo" e non rispettare i tempi prestabiliti, o addirittura non portarlo a termine. In questo caso il termine "obiettivo" acquisisce un significato più ampio: tutti quei propositi, azioni, impegni che hanno come scopo quello di essere completati in tempi più o meno definiti.
L'idea dell'applicazione ha quindi preso forma come piattaforma in cui gli utenti potessero creare un proprio percorso di studio, chiamato piano, e fruire di percorsi creati anche da altri utenti. Ogni piano è organizzato in settimane, chiamate steps, in cui sono suddivisi i materiali da consultare, chiamati materials. In questo modo gli utenti hanno a disposizione un'applicazione che, oltre a fornire un servizio simile ad applicazioni ben più note (Udemy, Coursera, ecc), permette di organizzare un proprio obiettivo.
La realizzazione di Litto ha comportato diverse fasi di sviluppo. Nella prima fase sono stati prodotti tutti i blueprint necessari a una comprensione del contesto in prospettiva di implementazione, quali Class Diagram, Use Case Diagram e schema generale dell'Architettura. Nella seconda sono state individuate le tecnologie necessarie ad implementare i casi d'uso specificati. La scelta è ricaduta su JEE che si interfacciasse con un database basato su grafi. Nella terza parte sono state implementate le due soluzioni identificate, basate sull'utilizzo delle tecnologie GraphQL e Neo4j OGM. Infine è stata realizzata l'interfaccia utente identificando le necessità dell'utente, sviluppando i prototipi di tutte le pagine e implementando l'applicazione completa.

2 Contesto dell'applicazione

Dopo aver condotto alcuni colloqui con candidati scelti, sono stati individuati i requisiti che avrebbe dovuto avere l'applicazione e sono stati delineati i casi

d'uso necessari e la struttura del modello di dominio.

2.1 Use Case Diagram

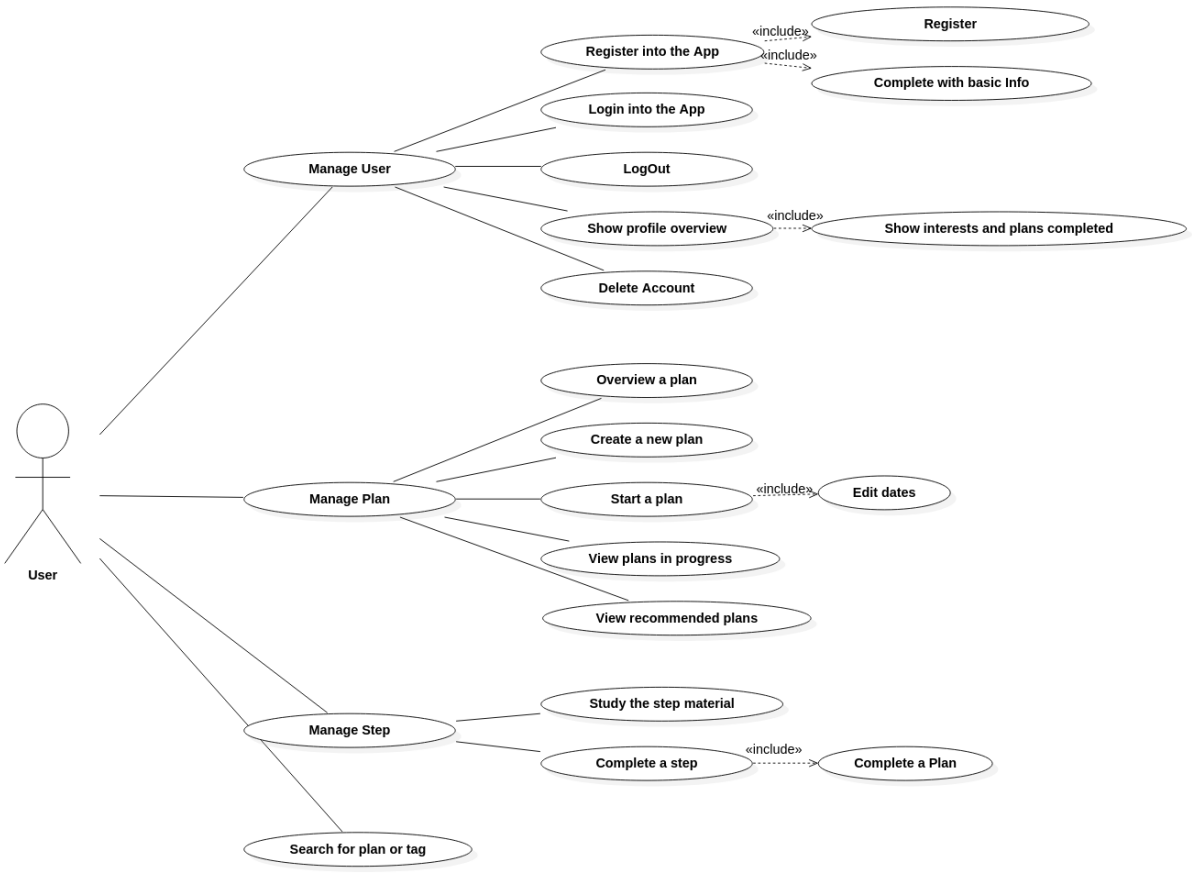

Figura 1: Rappresentazione dei casi d'uso che il backend deve implementare

Con lo Use Case Diagram, visibile in Figura 1, sono stati analizzati i requisiti dell'utente traducendoli in casi d'uso che l'applicazione avrebbe dovuto svolgere. I casi d'uso selezionati sono quelli a più alta priorità fra quelli individuati nei colloqui. Possiamo quindi vedere come l'actor sia lo **User** che può svolgere diverse azioni relative a: gestione del proprio profilo, gestione dei piani, gestione degli step in corso e ricerca. In particolare in una prima fase l'utente può registrarsi o fare login nell'applicazione, senza i quali non può usufruire di tutti gli altri casi d'uso. Quindi può visualizzare il proprio profilo utente, eliminarlo e fare il logout. Nella gestione del piano l'utente può: consultare un piano per decidere se iniziarlo o meno visualizzando tutti gli step che lo compongono e la durata complessiva; può iniziare il piano definendo le date di inizio e fine entro

cui completarlo; vedere i piani che deve ancora completare ed in particolare quale è lo step da completare nella settimana in cui si trova; vedere i piani consigliati secondo il profilo dell'utente; creare un nuovo piano da zero. Quest'ultimo caso d'uso è quello che contiene maggiore carico funzionale in quanto l'utente deve avere già chiaro come suddividere gli step, quale materiale includere all'interno del piano e come disporlo nei vari step. Infine l'utente può consultare uno step, per esempio di un piano iniziato, e completarlo. Se lo step appena completato è l'ultimo del piano iniziato, allora viene completato anche questo piano. Infine viene data la possibilità all'utente di ricercare un piano.

2.2 Class Diagram

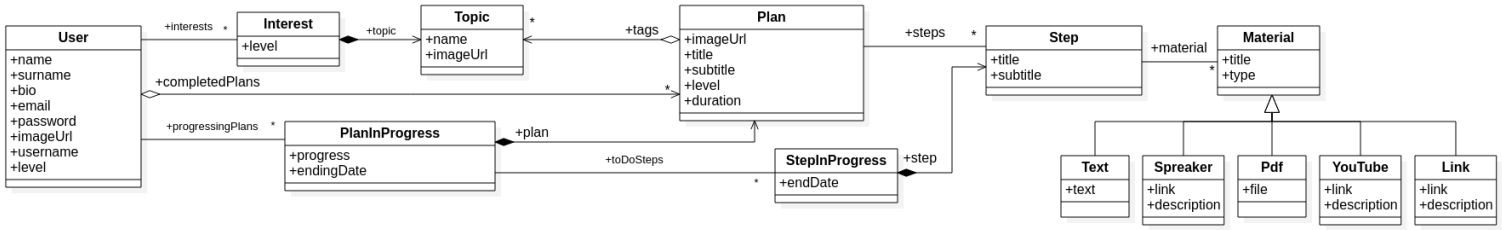

Figura 2: Rappresentazione del diagramma delle classi del Domain Model Anemico

Una volta identificati i casi d'uso è stato possibile sviluppare il Class Diagram Anemico di quello che sarebbe stato il successivo Domain Model. In Figura 2 è riportato il diagramma. Possiamo notare che sono presenti le principali entità descritte fino ad ora: utente, piano, step e materiale. In particolare:

- **User**, oltre ad attributi "semplici" come nome e password, ha riferimenti ad una lista di Plan completati, una lista di **Interest** che sono personali dell'utente e che si riferiscono ognuno ad un Topic, ed infine una lista di **PlanInProgress**, anche questi personali dell'utente;

- **Plan** ha riferimento a una lista di **Step** che a sua volta ha riferimento a una lista di **Material**. Il Material può essere di diverse tipologie e per questo è stato necessario predisporre le subclasses relative seppur mantenendo l'attributo *type* nella classe base. Ogni plan ha anche una lista di **Topic** che rappresentano i suoi tag;

- **PlanInProgress** e **StepInProgress** sono le classi che rappresentano il piano iniziato dall'utente ed i relativi step. In questo caso è stato preferibile usare la composizione anziché il subclassing per permettere una maggiore estendibilità del codice e per rimanere più vicini possibile all'implementazione successiva sul database basato su grafi.

Nel diagramma non è stata esplicitata la presenza di un ID per ogni classe, in quanto questo è richiesto unicamente dal DB utilizzato e verrà quindi mostrato successivamente nello schema del DB stesso. Inoltre è stato scelto un Class Diagram anemico in quanto gran parte delle funzioni della Business Logic saranno svolte dal Controller e quindi, pensando in prospettiva di implementazione, producendo un Domain Model anemico che ha come responsabilità solamente quella di strutturare i dati e non di eseguire funzioni specifiche oltre a setters e getters.

3 Definizione dell'architettura

Una volta definiti i casi d'uso è stato necessario definire come implementarli, ovvero come fornire all'utente la possibilità di fruirne. Per questo è stato introdotto un frontend con cui l'utente può interfacciarsi, e per suddividere il carico funzionale mantenendo una forte separazione delle responsabilità, è stato predisposto un backend basato su architettura REST. Quindi seguendo il paradigma dell'architettura MVC: il frontend si occupa di gestire l'interazione con l'utente, ovvero esporre l'interfaccia grafica (View) e ricevere gli input dall'utente (Controller), mentre il backend si occupa di eseguire la Business Logic utilizzando anche un DataBase per la gestione del Model. Il frontend si interfaccerà al backend tramite delle API concordate in fase di sviluppo che ricalcano molto precisamente gli Use Case definiti in precedenza. Tale scelta è stata presa per garantire una maggiore semplicità nell'interfacciamento fra frontend e backend e per permettere uno lo sviluppo di entrambe le componenti più naturale e vicino alle specifiche funzionali individuate.
La trattazione della struttura del frontend sarà ripresa più avanti nel paragrafo dedicato, mentre il focus in questo e nei prossimi paragrafi ricadrà sul backend. Quest'ultimo si basa su architettura REST facendo uso delle tecnologie JEE e predisponendo l'utilizzo delle principali componenti di questo paradigma. In Figura 3 è possibile vedere lo schema dell'architettura. In particolare vediamo come:

- Il **Frontend** si interfaccia con il **Backend** utilizzando le REST API esposte. Ogni richiesta che arriva al Backend viene analizzata dai **Filters** che verificano l'autenticazione dell'utente, ed anche ogni risposta che fornisce il Backend viene gestita dai Filters che aggiungono gli headers corretti;

- Una volta che la richiesta è stata analizzata passa ai **Services** che espongono effettivamente l'endpoint della API. Da qua inizia l'esecuzione delle funzioni previste dalla Business Logic. Infatti ogni funzione dei Services richiama la funzione di un **Controller** che ha la responsabilità di orchestrare le operazioni da svolgere per completare la richiesta. Il Controller infatti può mettere in vita istanze di oggetti del **Model**, interfacciarsi con i **DAO** per la gestione dello strato di persistenza ed infine convertire gli oggetti creati in **DTO** tramite i **Mappers** per renderli fruibili dal Frontend;

- Come espresso, i **DAO** si occupano della gestione dello strato di persistenza. I DAO infatti hanno la responsabilità di interfacciarsi con il DataBase svolgendo principalmente le operazioni CRUD. Infatti è stato predisposto un DAO per ogni classe principale del Model.

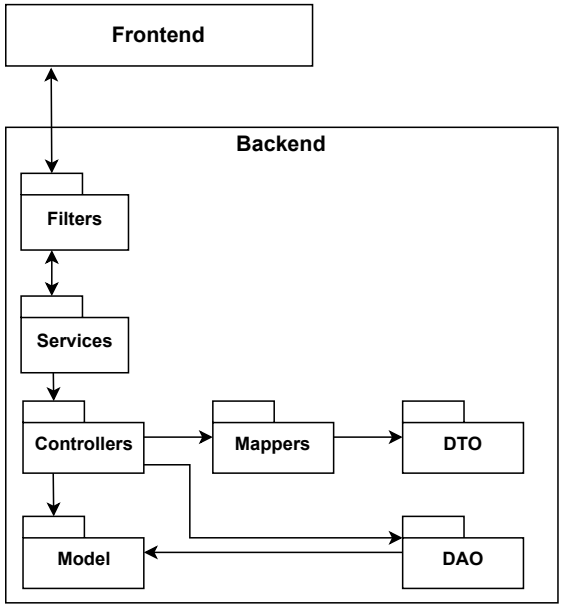

Figura 3: Schema dell'architettura dell'applicazione con particolare enfasi sul Backend

3.1 Scelta della tecnologia del DataBase: Graph-Based

Una volta delineata l'architettura del sistema, è stato necessario definire quale tipologia di DataBase fosse la migliore da utilizzare considerando il modello e le operazioni da eseguire su di esso. Per questo sono stati confrontati **DataBase Relazionali** e **DataBase NoSQL basati su Grafi**, ed in particolare fra questi è stato scelto **Neo4j**.
Questi ultimi sono basati sul concetto di *labeled property graph*, ovvero una struttura in cui:

- sono rappresentati nodi e relazioni;
- i nodi contengono proprietà in forma chiave-valore;
- le relazioni hanno associato un nome e una direzione, ed hanno sempre un nodo di partenza e uno di arrivo.

Questo tipo di DataBase quindi permette di memorizzare le entità sotto forma di grafo in cui l'entità vera e propria viene rappresentata come **Node**, mentre le relazioni fra entità come **Relationship**. La struttura del Domain Model presentata in Figura 2 si presta molto bene ad essere schematizzata su un DataBase simile. In Figura 4 è riportato lo Schema risultante ed è possibile notare come questo sia una trasposizione uno ad uno del Domain Model di Figura 2.

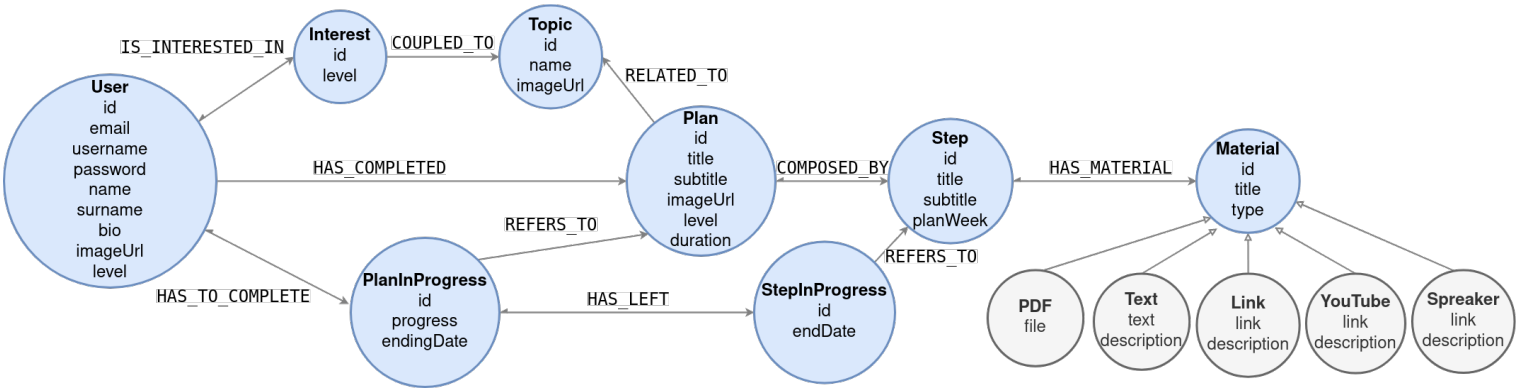

Figura 4: Schema risultante in Neo4j

Oltre a permettere questa facilitazione a livello concettuale i Graph DataBase Management System, ed in particolare Neo4j, permettono di avere performances molto migliori degli RDBMS in casi di utilizzo specifici. Oltre a permettere le basilari operazioni CRUD garantendo integrità nelle transazioni, Neo4j utilizza le tecnologie Native Graph Storage ed Index Free Adjacency. La prima indica come i dati siano memorizzati dal DB sotto forma di grafo effettivo, a differenza di altri GraphDB in cui i dati sono comunque memorizzati su tabelle relazionali. Index Free Adjacency invece indica come non vi sia una tabella degli indici complessiva per tutti i nodi ma che ogni nodo ne ha una propria più piccola che identifica i nodi con cui esso è in relazione. Questo permette di avere performances molto migliori di un RDBMS nel caso in cui si percorrano le relazioni all'interno del DB (*Traversing* del grafo. Nel caso del RDBMS questo implica fare una serie di Join di tabelle potenzialmente dell'ordine di milioni di righe, mentre nel caso del GDBMS implica solamente seguire i riferimenti necessari sotto forma di puntatori. A sostegno di ciò, in [3] viene riportato un esperimento in cui in una struttura Social Network-like conentente 1.000.000 di persone vengono estratti gli amici di amici fino ad una profondità di 5. Considerando che ogni persona ha circa 50 amici, i risultati visibili in Figura 5 confermano quanto precedentemente esposto.

Nel caso specifico dei casi d'uso da implementare è chiaro come sarà necessario attraversare il grafo in gran parte delle query da eseguire (per esempio per trovare gli step che l'utente deve completare nella settimana deve essere attraver-

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

Figura 5: Risultati dei tempi di una query in cui vengono attraversate le relazioni "friend_with" a differenti profondità in un RDBMS e in Neo4j

sata la relazione User-PlanInProgress e quella PlanInProgress-StepInProgress). Quindi, sebbene la poca maturità della tecnologia, e visto l'incremento di performances nel caso specifico, è stato scelto l'utilizzo di Neo4j per lo strato di persistenza del sistema descritto.
Una volta definito il DB di riferimento è stato necessario definire la tecnologia con cui interfacciarsi ad esso. Sono state quindi analizzate ed implementate due diverse scelte: **Neo4j OGM** e **GraphQL**.

3.2 Neo4j OGM

Neo4j OGM è un Object Graph Mapper, ovvero una libreria che mappa i nodi e le relazioni dal Graph DataBase ad oggetti e riferimenti del Domain Model e viceversa. L'OGM è una tecnologia analoga a quanto JPA svolge per gli RDBMS in qualità di Object Relational Mapper. Anche in questo caso infatti l'OGM permette un interfacciamento al DB semplificato. Come per JPA le classi POJO del Domain Model sono annotate per indicare all'OGM come persisterle sul DB. Inoltre anche per eseguire le query e le operazioni CRUD relative alla persistenza, l'OGM fornisce l'utilizzo di apposite classi Java senza la necessità di ricorrere a query in *Cypher*, linguaggio di querying di Neo4j. Queste richieste di persistenza, prima di eseguire effettive query al DB, vengono gestite dall'OGM che ha una struttura interna che traccia i cambiamenti nelle entità persistite in una cache, garantendo tempi di accesso minori e interazione con il DB solamente quando necessario.
Come accennato in precedenza, l'OGM per persistere gli oggetti Java del Domain Model ha bisogno di un *ID* univoco generato dal client Java per ogni oggetto. Per questo ogni oggetto del Domain Model, in questa implementazione, estende la classe astratta **Entity** che provvede un ID univoco e una funzione per generarlo.
Oltre ad annotare le classi del Domain Model è stato necessario integrare le funzioni di persistenza all'interno dei DAO, che quindi si interfacciavano direttamente all'OGM che a sua volta, dopo essere correttamente configurato, si

interfacciava con il DB. In Figura 6 è riportato uno schema che nei paragrafi successivi sarà espanso maggiormente.

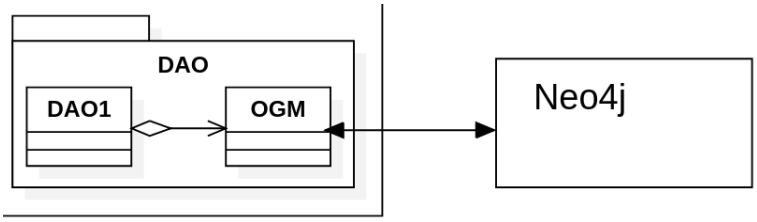

Figura 6: Interfacciamento dei DAO con Neo4j tramite l'OGM nella soluzione Neo4j OGM

3.3 GraphQL

QraphQL è una specifica che permette di standardizzare un query language basato sul concetto di grafo attraverso delle precise API fornite da un middleware. In questo progetto è stata utilizzata la versione che si interfaccia ad un DB Neo4j, ma questo viene nascosto al client fornendo sempre la stessa tipologia di API a prescindere dallo strato di persistenza utilizzato. Il client quindi può interfacciarsi a GraphQL facendo query basate su grafi utilizzando i concetti di nodo e relazione. Per poter uniformare le query è però necessario definire uno **Schema** in cui sono riportati i nodi e le relazioni che caratterizzano il modello a cui poi farà riferimento il client. Come è possibile vedere in Figura 7, questo viene fatto all'interno di un middleware specifico con cui il client si può interfacciare e che interpreta le query GraphQL ed interroga il DB sottostante, che in questo caso è Neo4j.

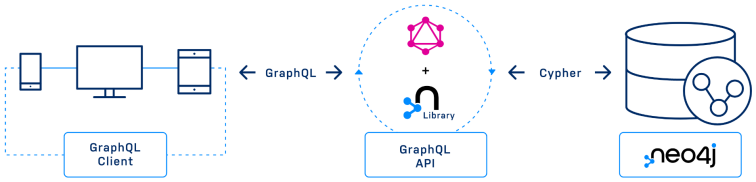

Figura 7: Interfacciamento del client con Neo4j tramite il middleware GraphQL

In GraphQL, oltre alle query, possono essere svolte le *mutations*: le **query** eseguono la funzione di Read mentre le **mutation** eseguono le restanti Create, Update e Delete. Il middleware GraphQL, opportunamente configurato, è in grado di tradurre sia le query sia le mutations in linguaggio Cypher, proprio di Neo4j. Ciò permette a GraphQL di restituire al client solamente i dati necessari rendendo molto più leggera la comunicazione fra middleware e client.

All'interno dell'architettura proposta nel progetto, i DAO si interfacciano al middleware tramite un client adhoc tramite cui compongono le richieste e ricevono le relative risposte, come è possibile vedere in Figura 8.

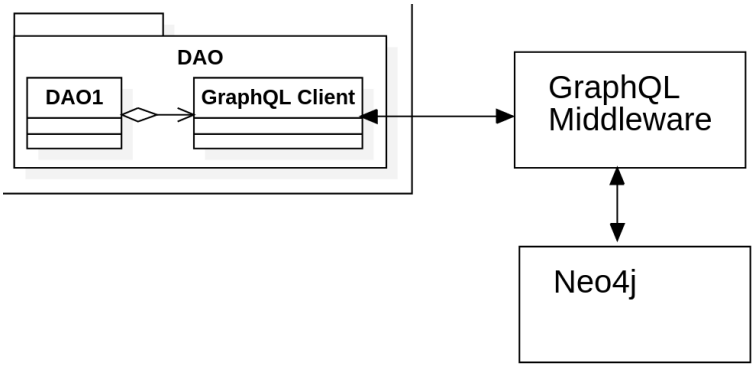

Figura 8: Interfacciamento del DAO con Neo4j tramite il client GraphQL che si interfaccia con il middleware GraphQL

4 Implementazione del Backend

Una volta definita l'architettura in Figura 3 e le tecnologie usate per l'interfacciamento con il DataBase, è possibile descrivere le scelte implementative fatte. Per la realizzazione è stato utilizzato lo stack tecnologico JEE eseguito su un server locale WildFly. L'interfacciamento interno fra JEE e WildFly avviene tramite Jersey. Tutte le dipendenze del progetto sono organizzate all'interno del *pom.xml* e gestite tramite Maven. Nella repository https://github.com/loremacchia/litto-backend può essere consultata tutta l'implementazione di entrambe le versioni. Infatti il progetto è stato suddiviso in due diverse implementazioni a seconda della modalità di interfacciamento con il DataBase, rispettivamente in */litto_backend/ogm* e */litto_backend/gql*. Tale scelta è stata necessaria perché vi erano dipendenze forti dai relativi Model che quindi non potevano essere facilmente riutilizzati. Le classi in cui non fossero presenti tali dipendenze sono state riutilizzate per quanto possibile, per esempio nella cartella */litto_backend/commondto* sono presenti i DTO riutilizzabili.
Per come è stato progettato il sistema, nessuna classe mantiene uno stato all'interno dell'esecuzione dell'applicazione. Ciò è coerente con i principi REST per cui le richieste devono essere stateless, quindi il server sviluppato non deve mantenere un proprio stato ma deve interfacciarsi con lo strato di persistenza ad ogni richiesta.
Nel progetto è stato utilizzato, dove necessario, **CDI** per gestire le istanze delle classi identificate per essere Beans. In particolare tutti i Beans creati (controllers, DAOs e mappers) sono stati definiti come **ApplicationScoped** perché, come appena scritto, nessuno di essi mantiene uno stato all'interno dello scope

e la loro creazione impatta in modo minore sulle performances del sistema rispetto a RequestScoped.
Nei prossimi paragrafi saranno descritte in modo dettagliato le scelte di implementazione prese per ciascun package.

4.1 Filters

I Filters sono fra le poche classi comuni ad entrambi i progetti, dato che non hanno dipendenze dirette da essi. I Filters infatti hanno il compito di gestire le Request e Response con il client ed interfacciarsi con i Services per far eseguire il caso d'uso. Infatti vi sono due diverse classi: *ServiceRequestFilter* e *ServiceResponseFilter*. Entrambe sono annotate come *Provider* per essere integrate all'interno dell'utilizzo di JAX-RS.

- **ServiceRequestFilter** estende *ContainerRequestFilter*, gestisce le *Request* in ingresso con la funzione *filter* e verifica che la richiesta sia in possesso di un token corretto per il tipo di richiesta che viene eseguita. Non tutte le richieste che arrivano devono avere necessariamente avere un token, per esempio la richiesta di login o registrazione dell'utente. Se infatti la Request in ingresso non necessita di token viene considerata corretta ed inoltrata ai Services. In tutti gli altri casi viene verificato che l'header *Authentication* contenga il token corretto. I token in questione sono generati secondo lo standard **JWT**, che permette di autorizzare in modo semplice una richiesta. In particolare viene verificata l'auteniticità controllando se il token è stato emesso dal server e non da altre fonti, come è visibile nel codice sottostante.

```
private boolean verifyToken(String token) {
   try {
       Algorithm algorithm = Algorithm.HMAC256("secret");
       JWTVerifier verifier = JWT.require(algorithm)
           .withIssuer("auth0")
           .build();
       verifier.verify(token);
       return true;
   } catch (JWTVerificationException exception){
       return false;
   }
}
```

- **ServiceResponseFilter** estende *ContainerResponseFilter*, gestisce le *Response* da ritornare al client. Per questo, nella funzione *filter* vengono create Response adeguate a seconda del contenuto della Response stessa: se ha body *null* la Response avrà come codice di stato *400*, mentre avrà *200* altrimenti. In questa funzione vengono aggiunti ad ogni Response anche gli header necessari alla corretta gestione del CORS.

```
public void filter(ContainerRequestContext requestContext,
    ContainerResponseContext responseContext)
     throws IOException {
   if (responseContext.getEntity() != null) {
      responseContext.setStatus(200);
   } else {
      if(requestContext.getMethod().equals("OPTIONS")) {
         responseContext.setStatus(200);
      }
      else {
         responseContext.setStatus(400);
      }
   }
   responseContext.getHeaders().add("Access-Control-Allow-Origin",
        "*");
   responseContext.getHeaders().add("Access-Control-Allow-Credentials",
        "true");
   responseContext.getHeaders().add("Access-Control-Allow-Methods",
        "GET, POST, DELETE, HEAD, OPTIONS");
   responseContext.getHeaders().add("Access-Control-Allow-Headers",
        "Origin, X-Requested-With, Content-Type, Accept,
        authorization");
}
```

Utilizzando queste due classi, i Services possono gestire le richieste come se fossero già validate e corrette, e le response disinteressandosi di come effettivamente verranno inviate.

4.2 Services

I Services sono stati realizzati utilizzando la libreria Jax-Rs grazie a cui le richieste fatte dal client su un certo URI portano all'esecuzione della relativa funzione del Service. Inizialmente è stato correttamente configurato Jersey nel file *webapp/WEB-INF/web.xml*. Quindi ogni classe in Service è stata annotata con **@Path** specificando quale fosse il path corretto a cui quel servizio avrebbe dovuto rispondere. All'interno di ogni Service quindi sono stati definiti i metodi necessari per la corretta gestione degli endpoint. In ogni metodo vengono specificati: tipo di richiesta (se GET, POST o DELETE), il path a cui eseguire la funzione (è un path relativo rispetto a quello della classe) ed infine il tipo di messaggio che arriva in ingresso (*Consumes*) e viene restituito in uscita (*Produces*). Gli URI possono includere anche dei parametri, per esempio lo userId, che vengono forniti in ingresso alla funzione tramite la annotazione *PathParam*. Di particolare interesse è l'utilizzo interno da parte di Jax-Rs della libreria di mapping Jackson che permette di deserializzare un Json fornito in input come un POJO e quindi utilizzarlo normalmente nelle funzioni che vengono richiamate all'interno. Ogni funzione crea la *Response* da inviare poi al client settando

come entity il risultato dell'esecuzione di una funzione del controller relativo. Ogni classe Service differente ha uno specifico controller iniettato tramite CDI sotto forma di Bean.
Viene lasciato il codice di esempio della classe *PlanService* del progetto ogm in cui si ritrovano tutti i concetti appena esposti.

```
@Path("/ogm/plan")
public class PlanService {

   @Inject
   PlanController planController;

   @GET
   @Path("/{id}")
   @Produces({ MediaType.APPLICATION_JSON })
   public Response getPlan(@PathParam("id") String ID) {
      return Response.ok().entity(planController.getPlan(ID)).build();
   }

   @POST
   @Path("/create/{userId}")
   @Consumes({ MediaType.APPLICATION_JSON })
   @Produces({ MediaType.APPLICATION_JSON })
   public Response createPlan(@PathParam("userId") String userID, Plan
       plan) {
      return Response.ok().entity(planController.createPlan(userID,
          plan)).build();
   }
}
```

Di seguito viene mostrata la lista degli endpoint della API. Anaizzando i casi d'uso di Figura 1, si può notare come questi si realizzino per la totalità attraverso le chiamate agli endpoint.

```
// A seconda del progetto scelto
"http://localhost:8080/litto-backend/webapi/" + "ogm" o "gql"
"/user"
    POST // Crea lo user: INPUT UserInitDTO OUTPUT TokenIDDto
    POST "/{{userID}}" // Completa lo user: INPUT UserCompleteDTO OUTPUT
        boolean
    POST "/login" // Login dello user: INPUT UserLoginDto OUTPUT
        TokenIDDto
    GET "/{{userID}}/logout" // Logout dello user: OUTPUT boolean
    GET "/{{userId}}" // Read dello user: OUTPUT UserDto
    DELETE "/{{userId}}" // Rimozione dello user: INPUT ID OUTPUT boolean
    GET "/interests" // Read della lista iniziale degli interessi fra
        cui far scegliere l'utente: OUTPUT List<Topic>
    GET "/{{userId}}/goals" // Read di tutti gli step attivi correnti:
        OUTPUT List<StepDto>
```

```
    GET "/{{userId}}/recommended" // Read dei piani consigliati: OUTPUT
        List<PlanPreviewDto>
    POST "/user/{{userId}}/start/{{planId}}" // Inizio di un piano da
        parte dell'utente: INPUT DateDto OUTPUT boolean
"/search"
    GET "/{{word}}" // Ricerca di piani e topic inerenti alla parola:
        OUTPUT SearchDto
"/plan"
    POST "/create/{{userId}}" // Creazione di un piano da parte
        dell'utente: INPUT Plan OUTPUT ID
    GET "/{{planId}}" // Read del piano: OUTPUT PlanDto
"/step"
    GET "/{{userId}}/{{planId}}" // Read dello step relativo al piano
        avviato dall'utente: OUTPUT StepActiveDto
    GET "/{{userId}}/{{planId}}/next/{{currentPlanWeek}}" // Selezione
        dello step successivo dopo aver completato lo step corrente:
        OUTPUT boolean
```

4.3 Model

In **Model** è presente il Domain Model presentato in Figura 2. In particolare, come esposto in precedenza, all'interno di ogni classe vi sono solamente gli attributi e metodi setters e getters senza metodi specifici. Questo perché architetturalmente la logica di dominio viene eseguita dai Controllers che utilizzano il Domain Model, che quindi è "unaware" di come viene utilizzato.
Le due implementazioni differiscono sotto vari aspetti fra cui la presenza di annotazioni e la gestione degli ID:

- in **GraphQL** il Model è composto da classi pure Java e gli ID sono autogenerati dal middleware. In classi come StepInProgress o Material non è necessario che il relativo Controller utilizzi l'informazione dell'ID. Quindi l'ID è stato integrato solamente in quelle classi in cui il Controller avrebbe dovuto utilizzarlo, ovvero in User e Plan;

- in **OGM** le classi invece sono annotate con le annotazioni OGM necessarie al mapping degli oggetti. Infatti tutte le classi sono annotate con *@NodeEntity* che identifica i nodi, mentre gli attributi che rappresentano la composizione con altre classi in UML sono taggati con *@Relationship* per identificare le relazioni fra nodi. Le relazioni possono essere sia Outgoing che Incoming ed in entrambi i casi, se esplicitate, permettono di percorrere la Relationship. In tutte le classi sono comunque necessari gli ID per permettere l'identificazione univoca dallo strato di persistenza. Questo perché per persistere gli oggetti l'OGM non genera automaticamente un ID per ognuno, che quindi deve essere generato prima di eseguire la persistenza. Le classi nel model quindi estendono la classe Entity che ha come attributo l'ID e fornisce un metodo pubblico per generarlo.

```java
public abstract class Entity {

   @Id // Annotazione per indicare l'ID dell'oggetto
   private String id;

   public void generateId() {
      id = UUID.randomUUID().toString();
   }

   // Setters and getters

}
```

Di seguito viene mostrata la differenza di gestione dei due modelli prendendo come esempio la gestione di Plan:

```java
// OGM
@NodeEntity
public class Plan extends Entity {

   public Plan() {}
   private String imageUrl;
   private String title;
   private String subtitle;
   private int level;
   @Relationship(type = "RELATED_TO", direction = Relationship.OUTGOING)
   private List<Topic> tags;
   @Relationship(type = "COMPOSED_BY", direction = Relationship.OUTGOING)
   private List<Step> steps;
   private int duration;

}

// GraphQL
public class Plan {

   public Plan() {}
   private String id;
   private String imageUrl;
   private String title;
   private String subtitle;
   private int level;
   private List<Topic> tags;
   private List<Step> steps;
   private int duration;

}
```

Un'eccezione a tale semplicità di utilizzo è stata il subtyping di Material. Infatti le classi dovevano essere serializzate e deserializzate da Jackson in corrispondenza degli endpoints. Ciò ha imposto di annotare la classe Material in modo da fornire a Jackson la struttura del Json che si sarebbe dovuto aspettare. Infatti a seconda del tipo espresso nell'attributo *type* in Material viene considerata la classe corrispondente. Di seguito viene proposta l'implementazione di Materials e di Link, una sua classe derivata.

```
@JsonTypeInfo(
     use = JsonTypeInfo.Id.NAME,
     include = JsonTypeInfo.As.PROPERTY,
     property = "type")
@JsonSubTypes({
     @Type(value = Link.class, name = "Link"),
     @Type(value = YouTube.class, name = "YouTube"),
     @Type(value = Pdf.class, name = "PDF"),
     @Type(value = Spreaker.class, name = "Spreaker"),
     @Type(value = Text.class, name = "Text")
})
@NodeEntity
public abstract class Material extends Entity{
    // attributi e metodi
}

@JsonTypeName("Link")
public class Link extends Material {

   public Link() {
      super.setType(MaterialType.Link);
   }
   @JsonProperty
   private String link;
   @JsonProperty
   private String description;

    // Setters e getters

}
```

4.4 DAO Neo4j OGM

Come scritto nel Paragrafo 3.2, Neo4j OGM è una libreria Java che permette di persistere in Neo4j oggetti del Domain Model annotati. Nel Paragrafo 4.3 vengono mostrate le annotazioni del Model necessarie all'OGM. In questo paragrafo il focus ricade sul funzionamento interno dell'OGM e come questo viene utilizzato nei DAO.
Per prima cosa per interagire con il DataBase Neo4j è necessaria un'istanza

di **Session** correttamente creata da una **SessionFactory**. Quest'ultima viene configurata inizialmente identificando l'istanza del DB da raggiungere in *ogm.properties* e dove è situato il Model in cui è definita la struttura degli oggetti da persistere. Non è infatti necessario definire lo schema con cui è configurato il DataBase, basta solamente annotare in modo coerente con quest'ultimo il Domain Model. Poiché la configurazione iniziale è onerosa è necessario svolgerla una volta durante tutto il ciclo di vita dell'applicazione. Per questo la SessionFactory espone i necessari metodi statici per fornire le instanze di SessionFactory come singleton e generare nuove Session quando necessario. Di seguito è riportata l'implementazione della SessionFactory.

```
public class SessionFactoryNeo4J {

   private static ClasspathConfigurationSource configurationSource = new
       ClasspathConfigurationSource("ogm.properties");
   private static Configuration configuration = new
       Configuration.Builder(configurationSource).build();
   private static SessionFactory sessionFactory = new
       SessionFactory(configuration,
       "com.macchiarini.lorenzo.litto_backend.ogm.modelOGM");
   private static SessionFactoryNeo4J factory = new
       SessionFactoryNeo4J();

   static SessionFactoryNeo4J getInstance() {
      return factory;
   }

   private SessionFactoryNeo4J() {
   }
    // Metodo per ottenere una nuova sessione
   Session getSession() {
      return sessionFactory.openSession();
   }

}
```

La **Session** quindi è in grado di tener traccia delle modifiche fatte agli oggetti persistiti da lei. Questo grazie all'utilizzo di una cache in cui vengono mantenuti gli oggetti persistiti e che permette di: persistere a sua volta su Neo4j solamente le modifiche ad oggetti precedentemente persistiti e di non inoltrare richieste "get" a Neo4j se vi è già un'istanza in cache. Tutto questo è possibile solamente se le richieste avvengono all'interno dello scopo della stessa Session. Da questo concetto quindi nascono due possibili implementazioni dei DAO: la prima che utilizza una Session sola all'interno dell'application scope del DAO; la seconda che crea una nuova Session per ogni richiesta. La prima permette di evitare svariate richieste inutili a Neo4j, ma i rischi sono che la cache occupi molto spazio dopo un gran numero di query e che lo stato della cache non risulti allineato a quello del DataBase se vengono fatte query da Session diverse (come è normale

che sia nel caso in cui ogni DAO abbia un'istanza di Session). La seconda invece non permette di sfruttare appieno le potenzialità della cache della Session ma evita un appesantimento del server e garantisce risposte sempre coerenti con ciò che si ha in Neo4j. Nei DAO sono state implementate tutte e due le opzioni con il seguente codice in cui viene predisposta una Session comune a tutto il DAO, ma se la variabile booleana *isSessionApplicationScoped* ha valore false allora la Session viene ricreata in ogni richiesta.

```
public class PlanDao {
   private SessionFactoryNeo4J sessionFactory;

   private Session session;
   private boolean isSessionApplicationScoped = true;

   public PlanDao() {
      sessionFactory = SessionFactoryNeo4J.getInstance();
      session = sessionFactory.getSession();
   }

   //...
}
```

Una volta definita la Session, è quindi possibile svolgere le operazioni di persistenza. Tramite la Session si possono eseguire: **save**, **load**, **loadAll** e **delete**. Tutte queste funzioni prendono in ingresso oggetti Java del Domain Model annotato e permettono di esplicitare la profondità con cui eseguire la relativa operazione. La *depth* quindi indica fino a che profondità considerare i nodi nelle relazioni. Se non viene esplicitata la depth, l'operazione verrà svolta su tutti gli oggetti raggiungibili dall'oggetto passato alla funzione stessa. L'utilizzo della depth permette approcci particolari:

- Utilizzare la depth per ottenere una sorta di DTO in base a ciò che effettivamente viene richiesto dal client;
- Salvare o eliminare un oggetto e gli oggetti che stanno in relazione con esso utilizzando una sola funzione. Per il salvataggio delle entity viene applicato il cascading, mentre per l'eliminazione è necessario fare molta più attenzione in quanto verrebbero eliminate tutte le entità in relazione con l'oggetto a una certa profondità;
- Utilizzare il Traversing nei Controllers. Infatti i DAO ritornano quasi sempre entità di base come User o Plan da cui il controller potrà estrarre ed analizzare le relazioni. Non vengono fatte, per scelta, query specifiche.

La semplicità di interfacciamento con la Session ha permesso un'altrettanta semplicità del DAO nei confronti del Controller. Infatti il DAO in questo caso espone pochi altri metodi oltre alle funzioni CRUD basilari. Inoltre per semplificare ulteriormente l'interfacciamento sono stati predisposti metodi che ritornano direttamente entità con profondità specifiche, come se fossero dei DTO.

Per esempio *getUserPreview* ritorna la preview dello User, ovvero l'entità con profondità 0:

```
public User getUser(String ID, int depth) throws Exception {
   if(isSessionApplicationScoped)
      return session.load(User.class, ID, depth);
   return sessionFactory.getSession().load(User.class, ID, depth);
}

public User getUserPreview(String ID) throws Exception {
   return getUser(ID, 0);
}

public User getUserOverview(String ID) throws Exception {
   return getUser(ID, 1);
}
```

Oltre alle funzioni CRUD sono stati implementati metodi specifici che utilizzano *loadAll* in cui è necessario filtrare i risultati ottenuti. Per esempio nelle funzioni di ricerca dell'utente tramite mail e quella tramite email e password risultano:

```
// Utilizzo di filtri semplici
public List<User> searchUserbyEmail(String email) throws Exception {
   if(!isSessionApplicationScoped)
      session = sessionFactory.getSession();
   List<User> users = new ArrayList<User>(session.loadAll(User.class,
       new Filter("email", ComparisonOperator.EQUALS, email)));
   return users;
}

// Utilizzo di filtri concatenati
public User loginUser(String email, String password) throws Exception {
   if(!isSessionApplicationScoped)
      session = sessionFactory.getSession();
   Filter f1 = new Filter("email", ComparisonOperator.EQUALS, email);
   Filter f2 = new Filter("password", ComparisonOperator.EQUALS,
       password);
   Filters f = f1.and(f2);
   List<User> users = new ArrayList<User>(session.loadAll(User.class, f,
       0));
   if(users.size() != 1)
      return null;
   return users.get(0);
}
```

Tutte le operazioni svolte dalla Session devono essere all'interno di una **Transaction**. Se nell'utilizzo della Session non viene esplicitata la Transaction, ne viene generata una che svolge un "auto-commit" sulla base dell'operazione richiesta. Nella maggior parte dei casi non è stato necessario utilizzare una Tran-

saction esplicita in quanto l'operazione da svolgere era singola. In casi specifici invece il commit delle richieste sul DataBase era necessario che fosse svolto nello stesso momento. Quindi nelle funzioni *deleteUser* e *completeStep* sono state gestite le transaction in maniera esplicita:

```
// In questa funzione l'avere la Transaction ha permesso di eseguire
// solamente una richiesta al DataBase sincronizzando l'eliminazione
// delle entita'selezionate
public void deleteUser(String userID) throws Exception {
   if(!isSessionApplicationScoped)
      session = sessionFactory.getSession();
   User user = session.load(User.class, userID, 3);
   if(user != null) {
       // Metodo per iniziare una transaction
      Transaction t = session.beginTransaction();
      // Esempio di Cascading manuale in fase di Delete
      for(PlanInProgress p : user.getProgressingPlans()) {
         for(StepInProgress s : p.getToDoSteps()) {
            session.delete(s);
         }
         session.delete(p);
      }
      for(Interest i : user.getInterests()) {
         session.delete(i);
      }
      session.delete(user);
      t.commit(); // Metodo per fare commit
   }
}

// In questo caso avere la transaction ha permesso di aggiornare
// lo User ed eliminare le istanze non piu' necessarie di PlanInProgress
// e degli StepInProgress, tutto nella stessa Transaction
public boolean completeStep(User user, StepInProgress step,
    PlanInProgress pp, Plan p) throws Exception {
   if(!isSessionApplicationScoped)
      session = sessionFactory.getSession();
   if(step != null) {
      Transaction t = session.beginTransaction();
      pp.getToDoSteps().remove(step);
      session.delete(step);
      if(pp.getToDoSteps().isEmpty()) {
         user.addCompletedPlans(p);
         user.getProgressingPlans().remove(pp);
         session.delete(pp);
      }
      session.save(user);
      t.commit();
      return true;
```

```
    }
    return false;
}
```

4.5 DAO GraphQL

Per poter utilizzare GraphQL all'interno dei DAO è stato prima necessario realizzare il middleware con cui interfacciarsi. Seguendo le indicazioni riportate in [1] è stato creato un middleware utilizzando le librerie JavaScript Neo4jGraphQL e ApolloServer. Dopo aver configurato come raggiungere il DataBase, è stato definito lo Schema con cui mappare le richieste su Neo4j. Viene riportata la definizione dello schema per l'entità User:

```
const typeDefs = `
  type User {
    id: ID @id
    name: String
    surname: String
    bio: String
    email: String @unique
    password: String
    imageUrl: String
    username: String
    level: Int
    interests: [Interest] @relationship(type: "IS_INTERESTED_IN",
        direction: OUT)
    completedPlans: [Plan] @relationship(type: "HAS_COMPLETED",
        direction: OUT)
    progressingPlans: [PlanInProgress] @relationship(type:
        "HAS_TO_COMPLETE", direction: OUT)
  }
`;
```

Possiamo notare come:

- l'annotazione *@id* permetta al middleware di generare automaticamente l'ID univoco ed associarlo all'entità;
- sia possibile esprimere il campo come *@unique*;
- con l'annotazione *@relationship* si definisce la relazione fra le entità ed il relativo nome associato. Il tipo dell'entità a cui punta la relazione può essere identificato da parentesi quadre, come [Plan], se la relazione è uno a molti o molti a molti.

Per testare quali query poter fare o meno, è stato utilizzato il servizio Apollo-Sandbox che, analizzando lo schema definito, permette di generare query tramite interfaccia grafica come visibile in Figura 9.

Come si può notare nell'esempio le query in GraphQL includono:

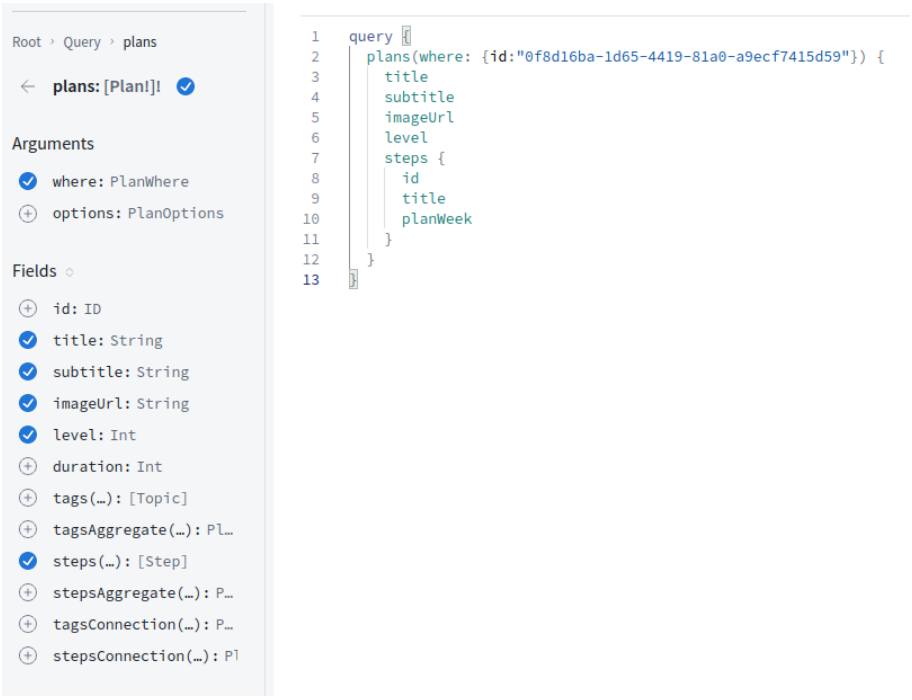

Figura 9: Query composta attraverso il tool ApolloSandbox. La query mostra come filtrare i Plans tramite l'ID e come restituire i campi del piano scelti e i campi scelti degli steps in relazione con il piano.

- un entry point chiamato o *query* o *mutation*;
- degli argomenti in cui vengono apportati filtri come *where*;
- il *selection set* in cui si definiscono i campi dell'entità principale di cui si vuole conoscere il valore.

Le responses ottenute sono in formato JSON e riportano la struttura analoga a quella del selection set, opportunamente ristrutturata per la gestione di liste di elementi, come per esempio nel caso di steps. Per ogni query o mutation eseguita sul middleware, GraphQL genera una query in linguaggio Cypher, proprio di Neo4j, che viene eseguita sul DataBase. Quindi se la richiesta è ben strutturata permette di non incorrere nel problema delle N+1 queries.
Una volta definito il middleware è stato necessario definire l'interfacciamento del "client" Java con esso. Sono state valutate tutte le librerie disponibili open source. La principale e più semplice da utilizzare era Nodes di American Express. Con questa libreria è possibile interfacciarsi in plain Java al middleware annotando le classi POJO del Domain Model. Riportando correttamente i nomi delle proprietà dello schema nel modello era possibile svolgere query. Le annotazioni possibili sono *@GraphQLArgument* e *@GraphQLVariable* che permettono

appunto di identificare il nome della proprietà mappata ed assegnargli un valore. Come riportato in [2] le query e mutations risultanti sono nella forma:

```
query {
  plans(id:"0f8d16ba-1d65-4419-81a0-a9ecf7415d59") {
    title
    subtitle
    imageUrl
    level
  }
}

query {
  plans(where:{id:"0f8d16ba-1d65-4419-81a0-a9ecf7415d59"}) {
    title
    subtitle
    imageUrl
    level
  }
}
```

Questo ha portato a due problemi principali:

- Come mostrato in Figura 9 le query ammesse dal sistema necessitano dell'utilizzo della clausola *where*, mentre in questo caso si rende implicita generando una query non standard. Tale tipologia di query non è contemplata nella documentazione della libreria Neo4j GraphQL;
- Tale tipologia di query necessita di aggiungere nel middleware la specifica di come deve essere interpretata. Questo quindi porta ad aggiungere responsabiltà al middleware che non gli competono. Infatti un caso del genere può essere utile quando il GraphQL client è un'applicazione e questa esegue una query GraphQL disinteressandosi di come fare la query stessa; sarà il middleware a generare correttamente la richiesta. Per chiarezza per rendere eseguibili tali query sarebbe necessario aggiungere un *resolver* che appunto risolve la query e la esegue:

  ```
  const resolvers = {
    Query: {
      plan(parent, args, context, info) {
        return plan.find(plan => plan.id === plan.id);
      }
    }
  }
  ```

Quindi è stato necessario creare un custom GraphQL Client poiché tutti i client presenti eseguivano le query nel medesimo modo. Il custom client creato sfrutta il fatto che le richieste in GraphQL possono essere inoltrate nel body di una

HTTP Request. Quindi l'idea è generare una stringa di testo contenente la query o mutation necessaria, integrarla all'interno di una richiesta HTTP POST e, tramite la libreria di mapping Jackson, ritornare un DTO utilizzabile da classi Java.
È stata creata una classe **GraphQLClient** in cui fosse semplice eseguire le richieste messe a disposizione da GraphQL. La classe predispone i metodi **query**, **create**, **update** e **customQuery**. In questi è necessario dichiarare solamente le parti fondamentali della query, che quindi viene assemblata all'interno dei metodi. I valori di ritorno sono dei Java Generics che hanno lo stesso tipo di quello definito in ingresso alla funzione. Riutilizzando la struttura sempre costante delle query GraphQL, riportata in precedenza, sono create:

- **query**: in cui si può specificare il nome dell'entità principale su cui eseguire la query, la where clause da applicare, quali sono i campi che vogliamo avere come ritorno ed infine in quale classe questi campi devono essere mappati.

```
public <T> T query(String entityName, String whereClause, String
    returnFields, Class<T> returnType)
     throws IOException, InterruptedException {
   // Costruzione della query vera e propria
  String queryBody = "{\"query\":\"query { " + entityName + " ";
  if (whereClause != null) {
     queryBody += "(where: {" + whereClause + "}) ";
  }
  queryBody += "{ " + returnFields + "}";
  queryBody += "}\"}";
   // Esecuzione della query eseguendo una HTTP Request
  HttpRequest request = HttpRequest.newBuilder().POST(
      BodyPublishers.ofString(queryBody))
     .header("Content-Type", "application/json").uri(url).build();
  HttpResponse<String> response;
  ObjectMapper mapper = new ObjectMapper();
  response = client.send(request, BodyHandlers.ofString());
   // Mapping dell'oggetto JSON ritornato in un DTO
  JsonNode node = mapper.readTree(response.body());
  // Legge il JSON, identifica il valore della chiave "entityName",
  // mappa il risultato nel returnType
  return mapper.readValue(node.get("data")
            .get(entityName).toString(), returnType);
}
```

- **create** in cui si può definire il titolo ed il nome della mutation, il nome dell'entità su cui eseguire la mutation, le input e where clauses, ed infine i return fields e il return type come nella query. Non viene riportato tutto il codice perchè è analogo a quello della query per molti aspetti.

```
public <T> T create(String mutationTitle, String mutationName,
    String entityName, String inputClause, String whereClause,
    String returnFields, Class<T> returnType) throws IOException,
    InterruptedException {
   // Creazione richiesta
  String mutationBody = "{\"query\":\"mutation " + mutationTitle +
      "{" + mutationName;
  if (inputClause != null || whereClause != null) {
     mutationBody += "( ";
     if (inputClause != null) {
        mutationBody += "input: {" + inputClause + "}, ";
     }
     if (whereClause != null) {
        mutationBody += "where: {" + whereClause + "} ";
     }
     mutationBody += ") ";
  }

  // ... HTTP Request e strutturazione del mapper

  //Il mapper in questo caso identifica mutationName e entityName
  return mapper.readValue(node.get("data").get(mutationName)
                .get(entityName).toString(), returnType);

}
```

- **update** analogo a create ma con la presenza della clause *update* anziché *input*.

- **customQuery** in cui vengono richiesti in ingresso il query body, la final entity e il tipo da ritornare. Questa query particolare permette al DAO di eseguire una query senza sottostare alle limitazioni imposte dai precedenti metodi. In questo caso il DAO dovrà creare in modo preciso la query da inviare. Se il mapping richiede maggiore complessità che un semplice parsing di un JSON, in questo caso il DAO potrà far ritornare come tipo un JsonNode con finalEntity = null e fare il parsing nel DAO stesso. La funzione risultante quindi è:

```
public <T> T customQuery(String queryBody, String finalEntity,
    Class<T> returnType) throws IOException, InterruptedException {
   // non viene creato il body perche' si presume che sia pronto
  HttpRequest request =
      HttpRequest.newBuilder().POST(BodyPublishers.ofString(queryBody))
        .header("Content-Type",
            "application/json").uri(url).build();
  HttpResponse<String> response;

  ObjectMapper mapper = new ObjectMapper();
```

```
        response = client.send(request, BodyHandlers.ofString());
        // mapping
        JsonNode node = mapper.readTree(response.body());
        if (finalEntity != null) {
           System.out.println(node.findPath(finalEntity).toString());
           return
                mapper.readValue(node.findPath(finalEntity).toString(),
                returnType);
        }
        return mapper.readValue(node.get("data").toString(), returnType);
     }
```

La classe GraphQL query ottenuta quindi è:

```
@ApplicationScoped // Annotazione CDI per essere usata come Bean dai DAO
public class GraphQLClient {

   private URI url;
   private HttpClient client;

   public GraphQLClient() {
      url = URI.create("http://localhost:4000/");
      client = HttpClient.newHttpClient();
   }

   public <T> T query(String entityName, String whereClause, String
        returnFields, Class<T> returnType) throws IOException,
        InterruptedException {...}

   public <T> T customQuery(String queryBody, String finalEntity,
        Class<T> returnType) throws IOException, InterruptedException
        {...}

   public <T> T create(String mutationTitle, String mutationName, String
        entityName, String inputClause, String whereClause, String
        returnFields, Class<T> returnType) throws IOException,
        InterruptedException {...}

   public <T> T update(String mutationTitle, String mutationName, String
        entityName, String updateClause,String whereClause, String
        returnFields, Class<T> returnType) throws IOException,
        InterruptedException {...}
}
```

Una volta definito il client con cui interfacciarsi al middleware, sono stati definiti i DAO. In particolare è stato predisposto un DAO per ogni classe del Domain Model che dovesse essere persistita sul DataBase e che non potesse essere persistita attraverso relazioni di un'altra classe. Ogni DAO ha come unico attributo il Bean del GraphQLClient e a sua volta anche il DAO è un Bean per i controller.

Ogni funzione dei DAO si occupa di gestire l'oggetto in ingresso occupandosi di deserializzarlo nella richiesta e di specificare quali valori di ritorno necessita. Un chiaro esempio di questo è updateUser in cui viene deserializzato lo User e i suoi interessi e, dopo aver chiamato update, ritorna l'ID dell'utente aggiornato.

```
public void updateUser(User user) throws IOException,
    InterruptedException {
   String updateClause = "name: \\\"" + user.getName() + "\\\"";
   updateClause += "surname: \\\"" + user.getSurname() + "\\\"";
   updateClause += "bio: \\\"" + user.getBio() + "\\\"";
   updateClause += "imageUrl: \\\"" + user.getImageUrl() + "\\\"";
   updateClause += "interests: [";
   for (Interest i : user.getInterests()) {
      updateClause += "{ create: { node: { ";
      updateClause += "level: " + i.getLevel() + " , ";
      updateClause += "user: { connect: { where: { node:
             { id: \\\"" + user.getId() + "\\\" }}}}, ";
      updateClause += "topic: { connect: { where: { node:
             { name: \\\"" + i.getTopic().getName()
           + "\\\" }}}}, }}},";
   }
   updateClause += "]";
   gql.update("UpdateUsers", "updateUsers", "users", updateClause, "id:
       \\\"" + user.getId() + "\\\"", "id", IDGqlDto[].class);
}
```

Un esempio di come viene svolta una query e di come i DAO effettivamente abbiano maggior carico computazionale, può essere visibile in getRecommendedPlans in cui viene svolta prima una query per ottenere gli interessi dell'utente e poi una query per ricercare quali piani hanno nel proprio titolo il dato interesse. Nella funzione viene gestito manualmente anche il valore di ritorno di query, utilizzando il tipo di ritorno JsonNode. Questo permette di fare il parsing personalizzato del Json e ottenere i nomi di tutti gli interessi trovati senza dover creare una classe DTO apposita. In output dall'ultima query invece possiamo vedere come venga mappato il Json in una lista di PlanPreviewDto tramite Jackson.

```
public List<PlanPreviewDto> getRecommendedPlans(String ID) throws
    IOException, InterruptedException {
   // Query per ottenere gli interessi dell'utente
   JsonNode inters = gql.query("users", "id: \\\"" + ID + "\\\"",
       "interests { topic { name }}", JsonNode.class);
   // Estrazione dei nomi di ogni interesse
   JsonNode node;
   node = inters.findPath("interests");
   List<String> ints = new ArrayList<String>();
   for(JsonNode n : node) {
      ints.add(n.findPath("name").toString());
   }
```

```
   // Creazione della stringa per il body della query
   String parsedString = "[";
   for(String s : ints) {
      parsedString += "\\\"" + s.substring(1,s.length()-1) + "\\\",";
   }
   parsedString.substring(0,parsedString.length()-1);
   parsedString += "]";
   // Esecuzione della query e ritorno del valore di tipo corretto
   return Arrays.asList(gql.query("plans", "tags:{ name_IN: " +
        parsedString + "}", "id title imageUrl duration",
        PlanPreviewDto[].class));
}
```

Un altro esempio di interesse è la rimozione dell'utente. Qui possiamo vedere come non vi sia cascading nelle operazioni in GraphQL ma che ogni richiesta debba essere esplicitata. In particolare, partendo dal nodo User e ripercorrendo le relazioni, è possibile identificare i nodi da eliminare. Anche in questo caso, vista la differenza sostanziale con le altre query, è stato necessario utilizzare una customQuery.

```
public boolean deleteUser(String userID) throws IOException,
    InterruptedException {
   String queryBody = "{\"query\":\"mutation { deleteUsers(where: {"
        + "id: \\\""+userID+"\\\""
        + "}, delete: { "
        + "progressingPlans: [ { delete: { toDoSteps: [ {} ] } } ], "
        + "interests: [ {} ] }) "
        + "{ nodesDeleted } }\"}";
   int elim = gql.customQuery(queryBody, "nodesDeleted", int.class);
   return elim > 0;
}
```

Altro caso di interesse è la funzione di creazione del piano che, poiché i materiali vengono creati dall'utente in modo standard, deve serializzare la richiesta analizzando di che tipo è ogni Material. Visto che la query sarebbe risultata estremamente complicata, è stato deciso di suddividerla in due parti distinte: la prima in cui viene creato il piano e vengono settati i field iniziali (titolo, tags, ecc); nella seconda vengono creati gli Steps e i relativi Material.

```
public String createPlan(Plan plan) throws IOException,
    InterruptedException {

   List<Topic> topics = plan.getTags();

   String queryBody = "{\"query\":\"mutation {createPlans(input: {"
        + "title: \\\"" + plan.getTitle() + "\\\","
        + "subtitle: \\\"" + plan.getSubtitle() + "\\\","
        + "imageUrl: \\\"" + plan.getImageUrl() + "\\\","
        + "level: " + plan.getLevel() + ","
```

```
     + "duration: " + plan.getDuration() + ",";
queryBody += "tags: { connectOrCreate: [";
for (int i = 0; i < topics.size(); i++) {
   queryBody += "{ onCreate: {node: { name: \\\"" +
       topics.get(i).getName() + "\\\"}}, "
           + "where: {node:{name: \\\"" + topics.get(i).getName() +
               "\\\""";
}
queryBody += "}}}]}}) {plans {id}}}\"}";
String planId = gql.customQuery(queryBody, "plans",
    IDGqlDto[].class)[0].getId();


String secondQueryBody = "{\"query\":\"mutation UpdatePlans {
    updatePlans(where: "
          + "{id:\\\"" + planId + "\\\"}, " + "create: {steps: [";
for (Step s : plan.getSteps()) {
   secondQueryBody += "{node: {" + "title: \\\"" + s.getTitle() +
       "\\\","
                + "subtitle: \\\"" + s.getSubtitle()+ "\\\","
                + "planWeek: " + s.getPlanWeek() + ","
                + "plan: {connect: {where: {node: {id: \\\"" + planId
                    + "\\\"}}}},"
                + "materials: {create: [";
   for (Material m : s.getMaterials()) {
      switch (m.getType()) {
      case PDF:
         secondQueryBody += "{node: {title: \\\"" + m.getTitle() +
             "\\\","
                   + "type: \\\"" + m.getType() + "\\\","
                   + "file: \\\"" + m.getFile() + "\\\"}}";
         break;
      case Text:
         secondQueryBody += "{node: {title: \\\"" + m.getTitle() +
             "\\\","
                   + "type: \\\"" + m.getType() + "\\\","
                   + "text: \\\"" + m.getText() + "\\\"}}";
         break;
      case YouTube:
         secondQueryBody += "{node: {title: \\\"" + m.getTitle() +
             "\\\","
                   + "type: \\\"" + m.getType() + "\\\","
                   + "description: \\\"" + m.getDescription() +
                       "\\\","
                   + "link: \\\"" + m.getLink() + "\\\"}}";
         break;
      case Link:
         secondQueryBody += "{node: {title: \\\"" + m.getTitle() +
             "\\\","
                   + "type: \\\"" + m.getType() + "\\\","
```

```
                        + "description: \\\"" + m.getDescription() +
                            "\\\","
                        + "link: \\\"" + m.getLink() + "\\\"}}";
            break;
         case Spreaker:
            secondQueryBody += "{node: {title: \\\"" + m.getTitle() +
                "\\\","
                        + "type: \\\"" + m.getType() + "\\\","
                        + "description: \\\"" + m.getDescription() +
                            "\\\","
                        + "link: \\\"" + m.getLink() + "\\\"}}";
            break;
         default:
            break;
         }
      }
      secondQueryBody += "]}}}";
   }
   secondQueryBody += "]}) {plans { id }}}\"}";
   gql.customQuery(secondQueryBody, "plans", IDGqlDto[].class);
   return planId;
}
```

4.6 Controllers

Nei Controllers risiede la parte principale della computazione della Business Logic. Per questo essi hanno il ruolo di "orchestratori", ovvero mettono in vita gli oggetti necessari, richiamano le funzioni dei DAO se necessitano di persistere un oggetto, richiamano metodi dei mapper a seconda di come DataBase o client si aspettano il dato in ingresso, ecc. Per limitare il carico dei Controllers è stato necessario suddividere in modo abbastanza netto le responsabilità con le varie componenti citate. Infatti ogni DAO ha la responsabilità di persistere un ben specifico tipo di oggetto e permette al controller di interfacciarsi ad esso come ad una blackbox. Ad esempio inizialmente i Controllers dell'OGM, vista la semplicità di persistere oggetti mostrata nel paragrafo precedente, si interfacciavano con un GenericDAO. Questo particolare DAO esponeva metodi generici per interfacciarsi con il DataBase, per esempio:

```
// Esempio di metodo esposto da GenericDAO
public <T> T getPreview(Class<T> objClass, String id) {
   return sessionFactory.getSession().load(objClass, id, 0);
}
```

in cui, per svolgere funzioni CRUD, bastava indicare quale era l'oggetto in questione, di che tipo era ed eventualmente la profondità a cui svolgere l'operazione. Il controller quindi poteva interfacciarsi direttamente ad esso per gran parte delle richieste di persistenza, almeno per quelle che non richiedevano particolari

modifiche alla query. Questo però caricava ancora di più il Controller richiedendogli la responsabilità di dover sapere come fare la richiesta per ottenere un certo oggetto e quando fare query generiche e quando no. Per questo è stato scelto di uniformare i DAO creandone uno per ogni classe da persistere, implementando al loro interno le richieste sebbene semplici e standardizzate. Questo ha permesso al controller di interfacciarsi sempre con lo stesso DAO per le richieste inerenti ad uno stesso oggetto, per esempio a UserDAO per le richieste relative a User, astraendo il modo in cui fare la richiesta.
Vista la differenza sostanziale dei DAO fra GraphQL e OGM, sono stati creati due packages di Controllers differenti. In entrambi vengono iniettati i beans necessari per la corretta esecuzione della Business Logic ed in entrambi i metodi esposti, richiamati dai Services, sono identici. Ciò che cambia è cosa il controller si aspetta dal DAO:

- in GraphQL viene sfruttata la capacità del DAO di ritornare esattamente ciò che serve, quindi nella maggior parte dei casi il controller evita di svolgere operazioni di mapping e quelle funzioni che non richiedono particolare logica. Come esposto in precedenza infatti in questo caso sul DAO ricade molta più responsabilità;

- in OGM invece si sfrutta l'estrema semplicità di utilizzo del DAO. Infatti ogni interfacciamento con esso implica il mappare un DTO nell'entità base e generare l'ID. Questo implica che il controller svolga anche le operazioni di mapping semplice, ma toglie responsabilità al DAO che deve persistere l'oggetto ad una data profondità. Un'altra differenza sostanziale nelle due implementazioni è il fatto che nell'OGM sia sempre necessario ottenere l'entità dal DataBase prima di svolgere ogni operazione CRUD. In particolare l'aggiornamento di campi o relazioni deve essere svolto sull'istanza ottenuta dal DataBase, poi persistita nuovamente dal DAO. Ciò permette di sfruttare la cache della Session come espresso in precedenza. Infine viene utilizzato molto il concetto di Traversing, ovvero ottenere dal DAO l'oggetto principale e da esso, nel controller, ricavare gli oggetti con cui è in relazione.

Entrambi i controller gestiscono nello stesso modo la creazione del token tramite JWT. Questo viene creato aggiungendo come claims o userID e l'email e cifrandolo tramite l'algoritmo SHA256. La funzione eseguita in fase di creazione utente e login è:

```
public String createToken(String email,String userID) throws
    JWTCreationException, Exception {
  Algorithm algorithm = Algorithm.HMAC256("secret");
  String token = JWT.create()
      .withIssuer("auth0")
      .withClaim("userID", userID)
      .withClaim("email", email)
      .sign(algorithm);
  return token;
```

```
}
```

Nelle funzioni dei controllers vengono gestite anche le eccezioni lanciate dai DAO nella fase di interfacciamento con il DataBase. I valori di ritorno in quei casi sono *null* o *false*, che devono a loro volta essere correttamente gestiti dal client.
Di seguito sono riportati alcuni esempi di differenti strutture nei controller nei due casi OGM e GraphQL.

```
// OGM
public String createPlan(String userId, Plan plan) {
// In questo caso si mostra come sia necessario avere un ID per ogni
// oggetto persistito. Questo deve essere generato dal controller.
    plan.generateId();
   for(Step s : plan.getSteps()) {
      s.generateId();
      for(Material m : s.getMaterials()) {
         m.generateId();
      }
   }
   try {
      planDao.createPlan(plan);
   } catch (Exception e) {
      System.err.println("ERROR: Cannot create the plan");
      e.printStackTrace();
      return null;
   }
   return "\""+plan.getId()+"\"";
}

// GraphQL
public String createPlan(String userId, Plan plan) {
// In questo caso il middleware a genera l'ID
   try {
      plan.setId(planDao.createPlan(plan));
   } catch (Exception e) {
      System.err.println("ERROR: cannot create the plan");
      e.printStackTrace();
      return null;
   }
   return "\""+plan.getId()+"\"";
}
```

```
// OGM
public StepActiveDto getActiveStep(String userID, String planID) {
// Esempio di come utilizzando il Traversing si possano ottenere gli
// oggetti con cui lo user e' in relazione a partire dallo user stesso
   User user;
```

```
   try {
      user = userDao.getUser(userID, 4);
   } catch (Exception e) {
      System.err.println("ERROR: Cannot retrieve the user");
      e.printStackTrace();
      return null;
   }
   // Necessario mappare manualmente particolari DTO
   Plan plan = null;
   PlanInProgress pp = null;
   for(PlanInProgress p : user.getProgressingPlans()) {
      if(p.getPlan().getId().equals(planID)) {
         plan = p.getPlan();
         pp = p;
      }
   }
   return stepMapper.fromPlanAtiveStepToActiveDto(plan,
       pp.getActiveStep());
}

// GraphQL
public StepActiveDto getActiveStep(String userID, String planID) {
// Richiesta al DAO che ritorna direttamente un oggetto mappabile
   StepActiveDto s;
   try {
      s = stepMapper.fromPlanProgressToActiveStep(
                  stepDao.getActiveStep(userID, planID),
                  userID);
   } catch (Exception e) {
      System.err.println("ERROR: cannot retrieve the current step of the
          plan for the user");
      e.printStackTrace();
      return null;
   }
   s.setEndDate(DateHandler.fromDBtoClient(s.getEndDate()));
   return s;
}
```

4.7 Mappers, DTOs e DateHandler

I Mapper ed i DTO permettono un corretto interfacciamento con le risorse esterne al server, in particolare con il client ed il DataBase. Per questo, come mostrato, sono stati predisposti DTO specifici per gestire le Responses del client GraphQL e le Request del client.
Infine per gestire anche lo scambio di date fra client-server-DB è stato predisposto un DateHandler che converte le Date in oggetti Java e permette anche il calcolo dell'incremento delle stesse.

5 Comparazione delle scelte GraphQL e Neo4j OGM

In entrambi i casi per fare il deploy del DataBase Neo4j è stata scelta un'istanza AuraDB in cui gratuitamente è possibile ottenere un DB Neo4j preconfigurato ed esposto via HTTP. AuraDB inoltre permette di visualizzare lo stato del DataBase in modo grafico ed intuitivo. Un esempio è visibile in Figura 10 in cui viene mostrata parte del grafo che connette entità distinte.

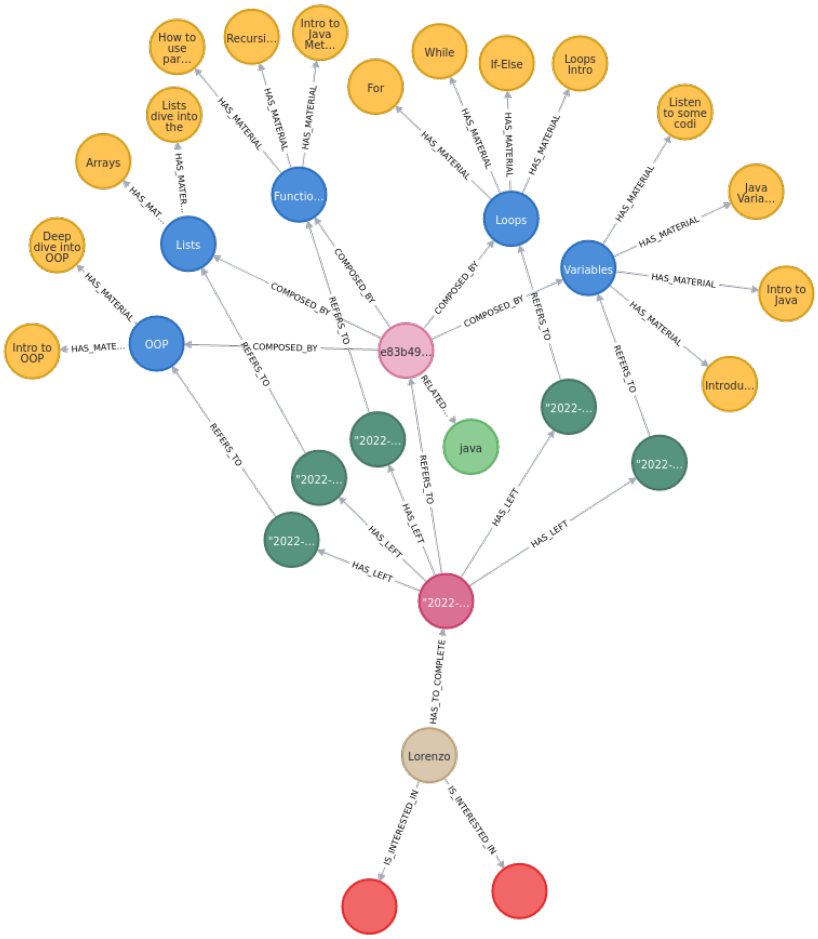

Figura 10: Esempio di grafo relativo allo User che deve completare un Plan. Viene riportato il PlanInProgress che relazionato al Plan mostra tutti gli Step rimanenti ed i Materials relativi

Prima di mostrare i test svolti è necessario fare un confronto fra le due soluzioni attuate e cosa ci aspettiamo dai test di confronto.

Neo4j OGM:

- **pro** poter eseguire query e relative funzioni in modo molto semplice e in plain Java;
- **pro** è una tecnologia sviluppata dalla stessa organizzazione che ha realizzato il DataBase, quindi ha maggiore supporto, soprattutto in Java;
- **pro** interfacciarsi direttamente con il server AuraDB;
- **pro** possibilità di utilizzare la cache per diminuire i tempi di risposta ed evitare richieste inutili al DB;
- **contro** possibilità di incoerenza fra i dati in cache e i dati persistiti sul DB;
- **contro** dati ottenuti dalla query spesso non del tutto necessari.

GraphQL Client

- **pro** poter ottenere dalla query esattamente i dati richiesti;
- **pro** poter sviluppare query in GraphQL che possono essere tradotte dal middleware per funzionare, potenzialmente, con qualsiasi tipo di DataBase;
- **pro** estrema customizzabilità della richiesta;
- **pro/contro** presenza di un middleware che permette di togliere carico dal DataBase, ma che genera un aumento dei tempi di risposta delle richieste;
- **contro** assenza di un client Java sviluppato per il caso specifico: richieste molto onerose da creare da parte del DAO dal punto di vista di serializzazione dei dati e completa mancanza di supporto;
- **contro** assenza di gestione della cache.

Da quanto scritto possiamo immaginare che :

- le richieste GraphQL inviino un numero minore di dati rispetto a quelle dell'OGM, cosa ci si aspetta;
- le richieste OGM dovrebbero essere più veloci in quanto non passano dal middleware;
- le richieste OGM con un uso sostanziale della cache dovrebbero essere ancora più veloci.

I primi test svolti riguardano i tempi di esecuzione delle query. Di seguito sono riportati i risultati in millisecondi ottenuti svolgendo le query necessarie a ritornare il corretto user e il corretto plan sulle tre versioni di DAO. Come previsto le richieste GraphQL sono più lente di quelle svolte con l'OGM. Inaspettato il

HTTP Request	**OGM Session req.**	**OGM Session app.**	**GraphQL**
GET /user/id	106	108	201
GET /plan/id	100	109	191

Tabella 1: Tempi di esecuzione su query uguali dei tre metodi

tempo di risposta maggiore da parte dell'OGM con Session application scoped rispetto a quello con Session request scoped.
Il secondo test riguarda la quantità di dati ottenuta dal DataBase o dal middleware. Le richieste testate sono le stesse del test precedente in quanto rappresentano un buon esempio del caso generico. In questo caso è stato rilevato il numero di bytes che compongono la risposta del DataBase e sono stati testati solamente l'OGM generico e GraphQL. In questo caso il risultato conferma le

HTTP Request	**OGM**	**GraphQL**
GET /user/id	787	424
GET /plan/id	1216	696

Tabella 2: Numero di bytes ottenuti dal DataBase tramite la query relativa alla richiesta svolta

aspettative rivelando una forte differenza fra le due implementazioni. Infatti in questo caso il client richiede che il risultato della Request abbia: campi basilari dello User, informazioni sugli Interest (a profondità 1) e sui piani completati (a profondità 1 anch'essi). Quindi l'OGM DAO deve svolgere la query a profondità 1 andando anche a trovare i piani in progress che possono essere anche svariati. Cosa analoga accade per i Plans in cui la query deve essere a profondità 1 a vengono ottenuti anche i tag e le informazioni sui materials.

6 Implementazione del Frontend

Una volta implementato il backend è stata sviluppata l'applicazione. Partendo da prototipi realizzati è stata sviluppata l'applicazione in Angular con la libreria di componenti Taiga UI. Poiché non è possibile compilare un'applicazione mobile nativamente in Angular, è stato necessario integrare il progetto in Ionic che, utilizzando Capacitor, ha permesso di compilare l'applicazione sia per Android che per IOs. Il codice del progetto può essere consultato nella pagina GitHub https://github.com/loremacchia/litto-ui, in particolare nella cartella src. L'applicazione creata è una Single Page Application, ovvero la pagina mostrata all'utente è sempre la stessa ma dinamicamente vengono modificate le componenti al suo interno, questo è reso possibile da Angular utilizzando tags HTML specifici . Lo schema dell'applicazione quindi si basa su "components" contenenti sia codice HTML da mostrare all'interno della pagina renderizzata, sia codice TypeScript per interfacciarsi con la vista e gestire input dall'utente ed interazioni con il backend. Queste ultime erano svolte utilizzando i "servi-

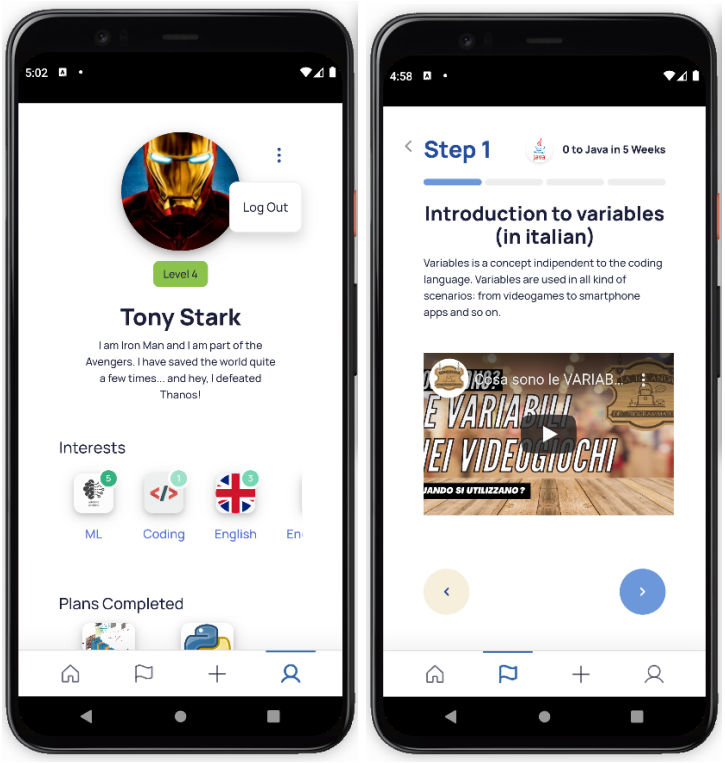

Figura 11: Schermate di riepilogo dell'utente e di fruizione di un piano

ces", classi TypeScript predisposte solamente a fare richieste HTTP al server e ritornare le risposte secondo gli standard attesi dalla componente. Per fare ciò sono stati introdotti anche dei "model", oggetti che rappresentano dei Data Transfer Object e che permettono di gestire un dato di ritorno come un oggetto appartenente ad una classe definita e non come semplice file Json. Avendo una panoramica del funzionamento, è possibile vedere come l'applicazione sia stata sviluppata seguendo il paradigma Model View Controller, come esposto in precedenza: in ogni componente è presente sia la componente View (HTML+CSS) sia la componente Controller (TypeScript), mentre il Model è rappresentato dal backend che gestisce i dati ricevuti e risponde alle richieste del controller. Per minimizzare le richieste HTTP e per soddisfare i requisiti REST, in cui il backend è stateless, è stato usato il localStorage fornito da Angular. In questo modo vengono salvate in memoria locale le informazioni intermedie quando l'utente sta creando un piano o il suo profilo e nello stesso modo viene salvato l'ID dell'utente connesso e il suo token.

7 Conclusioni

Considerando tutta l'architettura mostrata è possibile vedere come tutti i casi d'uso necessari siano stati realizzati dal server JEE. Con la suddivisione delle responsabilità è stato possibile far svolgere molte delle operazioni direttamente dal Controller o dal DAO nel server anziché caricare il client.

In Figura 12 viene mostrato il diagramma completo dell'architettura integrando sia Frontend sia Backend. Per rendere la visualizzazione più fruibile è stato proposto solamente il caso d'uso "get recommended plans" dell'utente. Partendo dal component del frontend è quindi possibile capire come la richiesta venga gestita dal sistema e dalle sue componenti. Sono state distinte le componenti dei DAO in GraphQL e OGM rispettivamente in giallo e arancione.

Inoltre i confronti svolti fra le due implementazioni dei DAO permettono di delineare quali sviluppi futuri poter considerare. Uno sviluppo interessante potrebbe quello di creare un server GraphQL evitando l'implementazione di uno in Java e aumentando le responsabilità del client che non dovrebbe usare più API rest ma GraphQL, quindi query già predisposte. Un altro test potrebbe essere utilizzare le query Cypher nell'OGM per compensare il fatto che il DataBase invii una grande quantità di dati non rilevanti in ogni Request. Infine un altro sviluppo futuro percorribile è l'ampliare i casi d'uso dell'applicazione integrando vari tipologie di utente, integrando una recommendation più puntuale,ecc.

Riferimenti bibliografici

[1] [Online]. Available: https://neo4j.com/docs/graphql-manual/current/getting-started/

[2] [Online]. Available: https://github.com/jefrajames/evalnodes

[3] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: new opportunities for connected data*. [Online]. Available: https://neo4j.com/graph-databases-book/

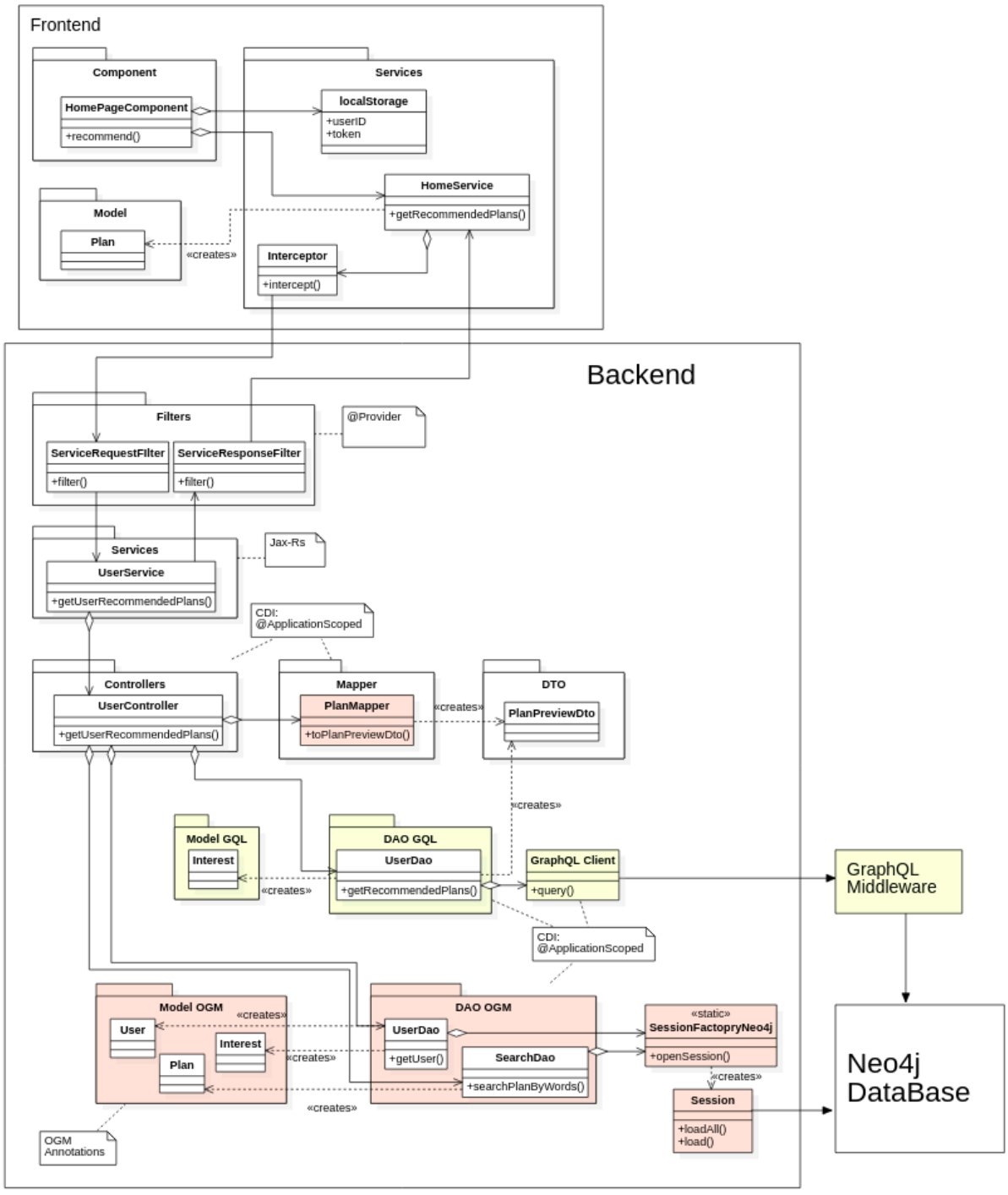

Figura 12: Diagramma che mostra l'architettura completa del sistema. Per mostrarlo è stato presentato il caso d'uso della recommendation di piani.