Università degli Studi di Firenze

SWAM Exam

Development of a Backend based on a GraphDBMS for an App

Candidate: Lorenzo Macchiarini

Academic Year 2021/2022

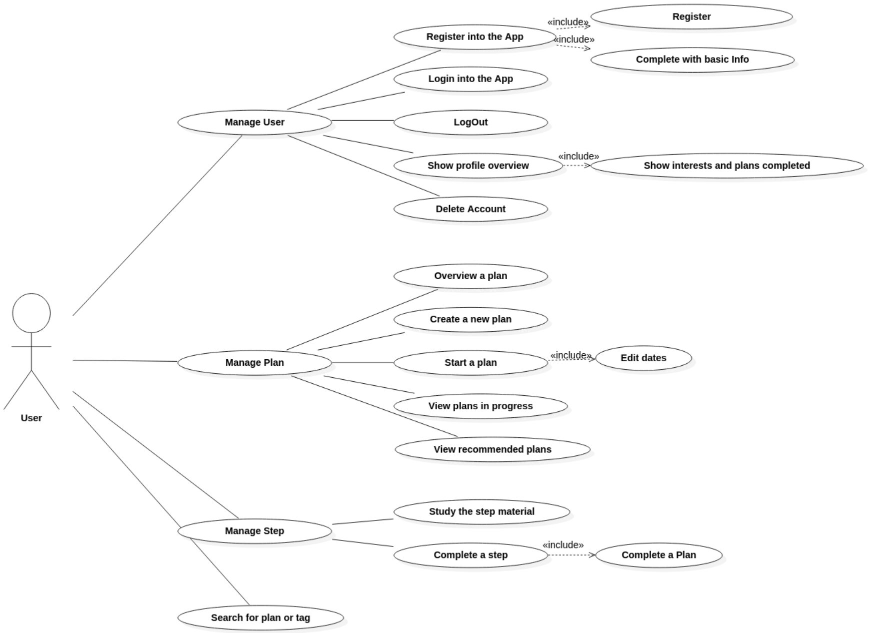
Introduction

Topics covered:

- How to represent data for a learning based app
- Approaches of interfacing with a GraphDB
- How architecture changes to interact with a GraphDB

Context where topics were applied:

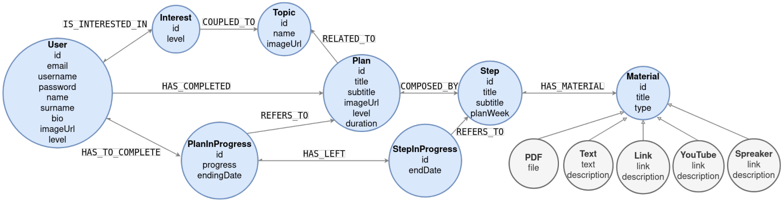
- Social learning App **Litto**, developed within HCI course
- Help user to reach their goals in a organized and timed way and to discover new interests
- Main idea similar to *Coursera* or *Udemy* but Litto allows users to create their goals

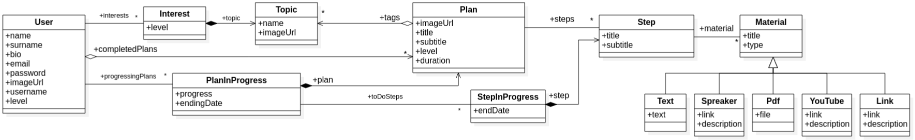

User
Manage User
Register into the App
«include»
Register
«include»
Complete with basic Info
Login into the App
LogOut
Show profile overview
«include»
Show interests and plans completed
Delete Account
Manage Plan
Overview a plan
Create a new plan
Start a plan
«include»
Edit dates
View plans in progress
View recommended plans
Manage Step
Study the step material
Complete a step
«include»
Complete a Plan
Search for plan or tag

DataBase Choice: Graph Data Base Neo4j

Based on the **Labeled Property Graph**[1] concept

- Represents Nodes and Relationship, each with its ID
- Nodes contain key-value properties → *Property*
- Nodes can be grouped together using a label → *Labeled*
- Relationship have a name, a direction, a starting and ending Node

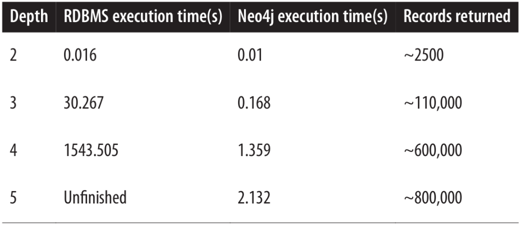




[1] https://neo4j.com/docs/

Graph Data Base Neo4j vs RDBMS

- + **Native Graph Storage**: Data stored as graph structure
- + **Index Free Adjacency**: Each Node has its own index that links all the Nodes in relationship with it
- + **Traversing** the graph is faster than executing Joins of a RDBMS
- - Inefficent when don't have a starting point
- - Nodes cannot store large chunk of data



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

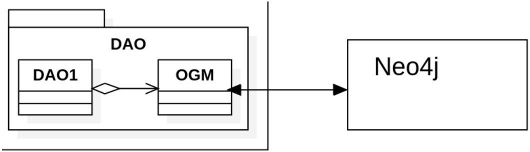
Test[2] with graph with 1 mln users, each with 50 friends.
Retrieving all the friends at a depth from 2 to 5

[2] Robinson, Webber and Eifrem, *Graph Databases: new opportunities for connected data.*

Neo4j OGM Introduction

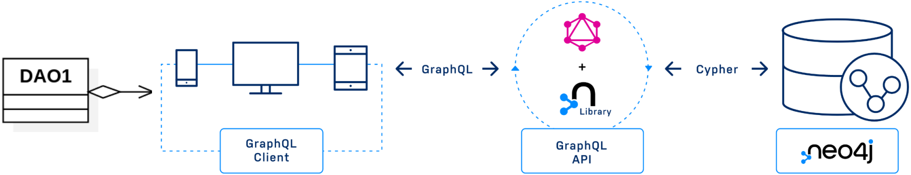
Neo4j OGM is the Neo4j implementation of an **Object Graph Mapper**

- **OGM** is similar to the ORM: persists native Java Objects to the DB into Nodes and Relationships
- The OGM can execute CRUD operations by constructing the query to run against the Neo4j DB in **Cypher** language
- Java Objects need to be annotated
- Clients can easily interact with the Neo4j DB



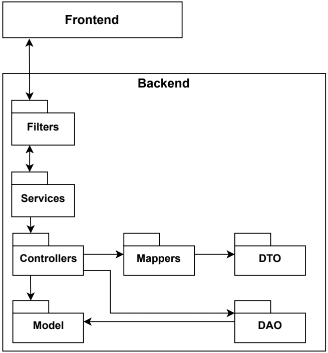
GraphQL Introduction

- **GraphQL** is a specification of query language based on Graph DBs
- Client can execute GraphQL queries instead of native DB language
- Needs a **middleware** that translates queries into Cypher (for Neo4j)
- Client is unaware of how the DB is realized: the middleware will translate it correctly (if configured)



Overview

- Architecture implemented with *JEE* and CDI Beans when needed
- Based on *REST* paradigm and *MVC* → backend implements Use Cases
- Different projects for the OGM and GraphQL implementation: mainly different DAO, Controllers and Model



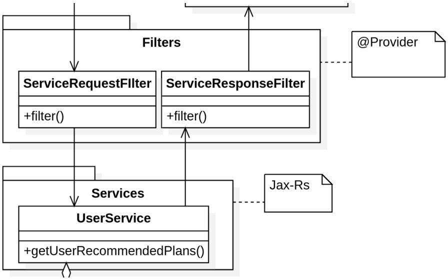
Filters and Services

Filters:

- **ServiceRequestFilter** authenticates with JWT Token
- **ServiceResponseFilter** adds the correct headers to handle CORS

Services exposing endpoints with Jax-Rs:

- One endpoint per Use Case
- Controllers as Beans to invoke the correct methods



Implementation differences between OGM and GraphQL

Following slides will show side by side (when possible) the different implementations of the architecture using **GraphQL** or **Neo4j OGM**

Components with strong differences are:

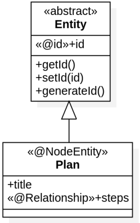
- Model
- DAO and how it interacts with DB using a Client
- Controller

Model

Anemic Domain Model → Business Logic resides in Controllers

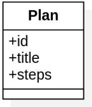
Model for Neo4j OGM:

- Plain Java classes with OGM annotation to map Entity to DB
- ID generated by the Java Server and required by the OGM



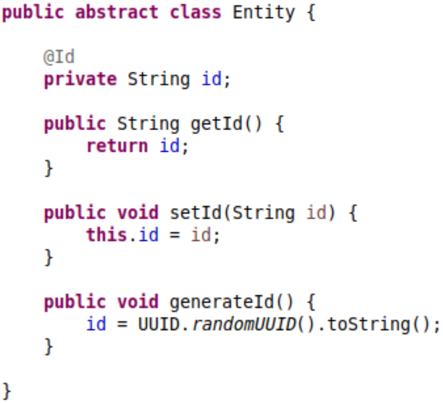
Model for GraphQL:

- Plain Java classes
- ID autogenerated by the Middleware, in Model only for User and Plan



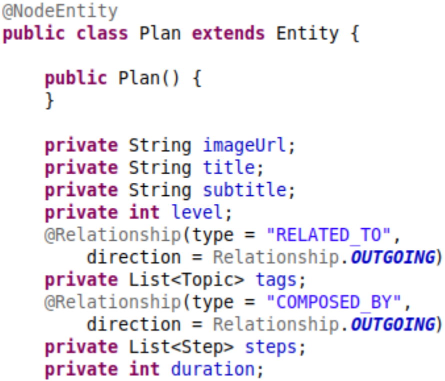
Model for Neo4j OGM

How ID is integrated in Model for Neo4j OGM: inheriting from Entity



```
public abstract class Entity {

    @Id
    private String id;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

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

}
```

Example of using NodeEntity and Relationships in OGM Model



```
@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;
```

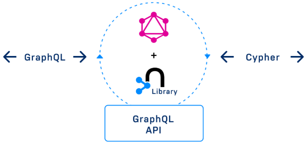
Relationships ∼ *@OneToMany* JPA

Relationships.IN ∼ *@MappedBy* JPA

GraphQL Middleware

Created a Middleware with Neo4jGraphQL JS library and ApolloServer

Defined GraphQL Schema related to Neo4j DB

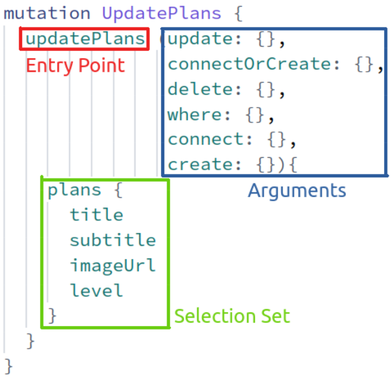


```
const typeDefs = ‘
  type User {
    id: ID @id
    name: String
    surname: String
    bio: String
    email: String @unique
    password: String
    imageUrl: String
    username: String
    level: Int
    interests: [Interest] @relationship(type: "IS_INTERESTED_IN",
        direction: OUT)
 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

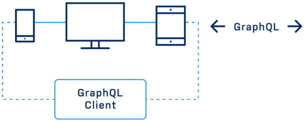
Apollo Server (Middleware)

GraphQL intefacing with Middleware

- Creating queries and mutations from the schema
- Defining precise structure for queries and mutations (entry point, arguments and selection set)
- Middleware translates queries in Cypher, execute against Neo4j
- JSON as response body



```
mutation UpdatePlans {
  updatePlans (update: {},
               connectOrCreate: {},
               delete: {},
               where: {},
               connect: {},
               create: {}){
    plans {
      title
      subtitle
      imageUrl
      level
    }
  }
}
```



GraphQL Choosing Client

Available OpenSource GraphQL Clients:

- GraphQL Java by GraphQL
- GraphQL-OGM by GraphQL
- Neo4j-GraphQL by Neo4j
- Nodes by American Express
- DGS framework by Netflix
- Apollo Client Plugin by Apollo
- Manifold by Manifold Systems
- GraphQL-JPA-query by IntroPro Ventures
- Wildfly GraphQL Feature Pack by Wildfly
- Java GraphQL Client by zaibacu
- Java GraphQL Client by kingdevnl

GraphQL Choosing Client

Available OpenSource GraphQL Clients:

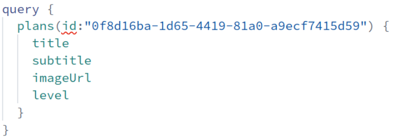
- GraphQL Java by GraphQL
- GraphQL-OGM by GraphQL
- Neo4j-GraphQL by Neo4j
- $\boxed{\text{Nodes by American Express}}$ ⇒ no JS, no Spring, well documented
- DGS framework by Netflix
- Apollo Client Plugin by Apollo
- Manifold by Manifold Systems
- GraphQL-JPA-query by IntroPro Ventures
- Wildfly GraphQL Feature Pack by Wildfly
- Java GraphQL Client by zaibacu
- Java GraphQL Client by kingdevnl

GraphQL Choosing Client

Nodes by American Express

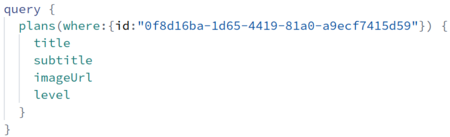
- Annotating POJOs of the Domain Model (@GraphQLArgument, ecc)
- Creating queries setting the arguments step by step

Nodes query (incorrect)



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

Correct query



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

$\longrightarrow$

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

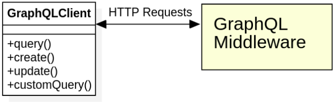
Apollo Server (Middleware)

For architectural design, chosen not to implement this way

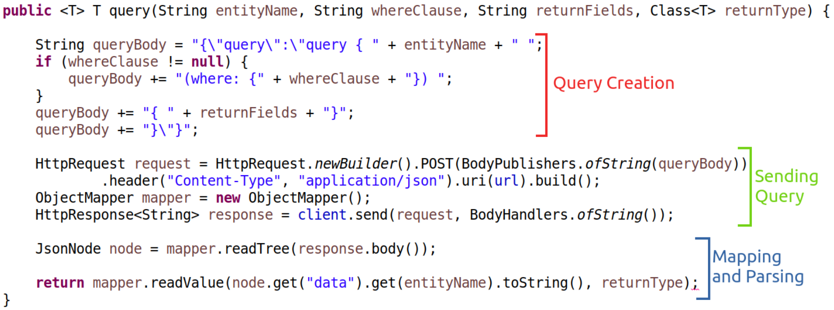
GraphQL Client Creation

Created a custom **GraphQL Client**

- HTTP requests with the query as body
- Jackson to map JSON responses into DTOs
- Methods: **query**, **create**, **update**, **customQuery**



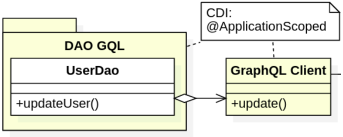
Query creation based on **string concatenation** over defined structure

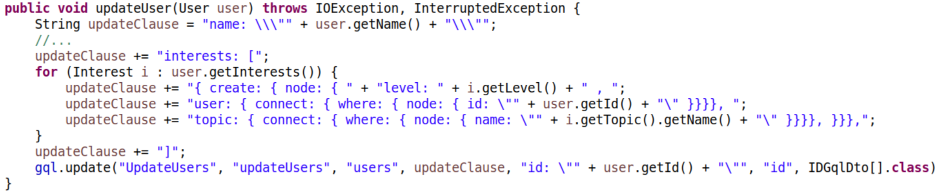


```
public <T> T query(String entityName, String whereClause, String returnFields, Class<T> returnType) {

    String queryBody = "{\"query\":\"query { " + entityName + " ";
    if (whereClause != null) {
        queryBody += "(where: {" + whereClause + "}) ";
    }
    queryBody += "{ " + returnFields + "}";
    queryBody += "}\"}";

    HttpRequest request = HttpRequest.newBuilder().POST(BodyPublishers.ofString(queryBody))
            .header("Content-Type", "application/json").uri(url).build();
    ObjectMapper mapper = new ObjectMapper();
    HttpResponse<String> response = client.send(request, BodyHandlers.ofString());

    JsonNode node = mapper.readTree(response.body());

    return mapper.readValue(node.get("data").get(entityName).toString(), returnType);
}
```

GraphQL DAO

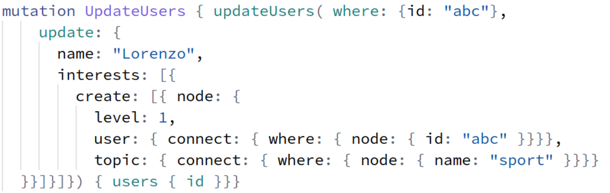
- One DAO for each class
- GraphQL Client as a Bean in each DAO
- Manual mapping to create the query string





```java
public void updateUser(User user) throws IOException, InterruptedException {
    String updateClause = "name: \\\"" + user.getName() + "\\\"";
    //...
    updateClause += "interests: [";
    for (Interest i : user.getInterests()) {
        updateClause += "{ create: { node: { " + "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)
}
```

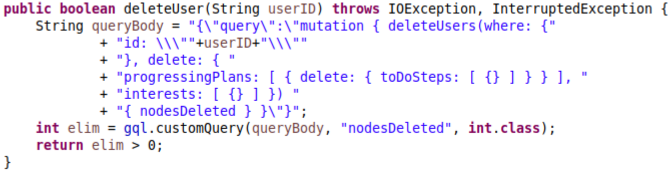
Mutation executed by the DAO



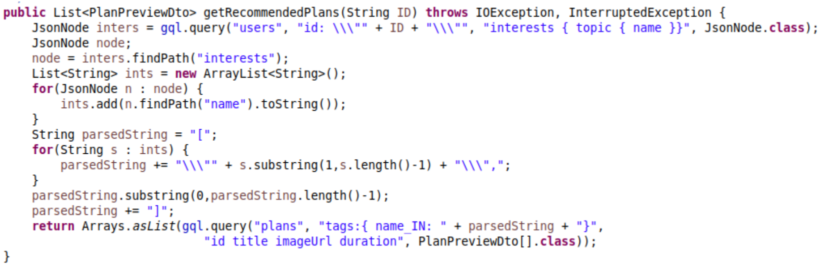
```graphql
mutation UpdateUsers { updateUsers( where: {id: "abc"},
    update: {
      name: "Lorenzo",
      interests: [{
        create: [{ node: {
          level: 1,
          user: { connect: { where: { node: { id: "abc" }}}},
          topic: { connect: { where: { node: { name: "sport" }}}}
  }}]}]}) { users { id }}}
```

GraphQL DAO

- Creating customQueries means no structure from GraphQL Client
- Allows delete that have no cascading
- Custom parsing of object with JsonNode



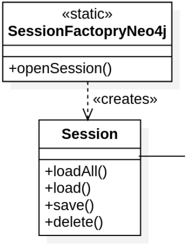
```java
public boolean deleteUser(String userID) throws IOException, InterruptedException {
    String queryBody = "{\"query\":\"mutation { deleteUsers(where: {"
            + "id: \\\""+userID+"\\\""
            + "}, delete: { "
            + "progressingPlans: [ { delete: { toDoSteps: [ {} ] } } ],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



```java
public List<PlanPreviewDto> getRecommendedPlans(String ID) throws IOException, InterruptedException {
    JsonNode inters = gql.query("users", "id: \\\"" + ID + "\\\"", "interests { topic { name }}", JsonNode.class);
    JsonNode node;
    node = inters.findPath("interests");
    List<String> ints = new ArrayList<String>();
    for(JsonNode n : node) {
        ints.add(n.findPath("name").toString());
    }
    String parsedString = "[";
    for(String s : ints) {
        parsedString += "\\\"" + s.substring(1,s.length()-1) + "\\\",";
    }
    parsedString.substring(0,parsedString.length()-1);
    parsedString += "]";
    return Arrays.asList(gql.query("plans", "tags:{ name_IN: " + parsedString + "}",
                            "id title imageUrl duration", PlanPreviewDto[].class));
}
```

Neo4j OGM Client

- Uses **Session** to run queries to DB
- Sessions generated by a **SessionFactory**
- SessionFactory configured once to reach DB
- Session provides also a cache for the persisted entities
- Session ~ *EntityManager* in JPA



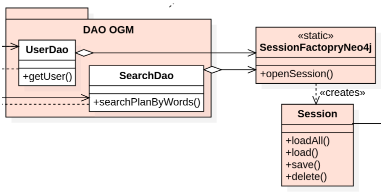
Session can execute queries (CRUD operations) on DB and persist with:

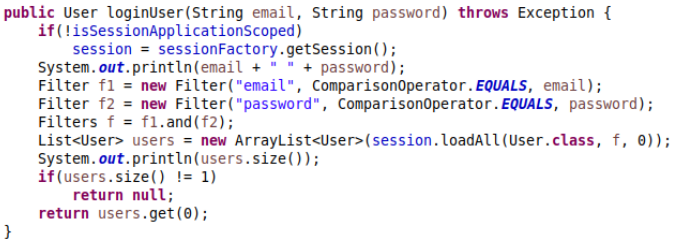
- **save**, **load**, **loadAll**, **delete** ~ *persist, find, remove* in JPA
- functions persist Java objects, no need to map

Queries in Neo4j OGM DAO

Using *depth* to retrive Nodes in relationship with a given Node

- Obtaining part of a Node, like DTO
- Saving objects allows to save its relationships (not in delete)
- Using *Traversing* in controllers

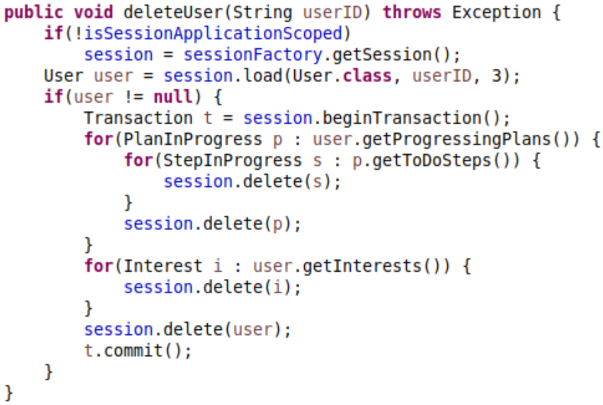




```java
public User loginUser(String email, String password) throws Exception {
    if(!isSessionApplicationScoped)
        session = sessionFactory.getSession();
    System.out.println(email + " " + password);
    Filter f1 = new Filter("email", ComparisonOperator.EQUALS, email);
    Filter f2 = new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

Transactions in Neo4j OGM DAO

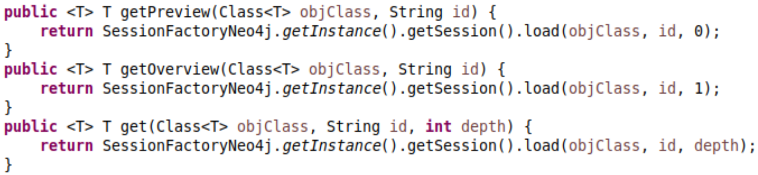
- Transactions are autocommitted on Session operations
- Sometimes need to explicitate it



```
public void deleteUser(String userID) throws Exception {
    if(!isSessionApplicationScoped)
        session = sessionFactory.getSession();
    User user = session.load(User.class, userID, 3);
    if(user != null) {
        Transaction t = session.beginTransaction();
        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();
    }
}
```

Generic DAO

- Simple queries could be generalized in a Generic DAO
- DAO exposing simple CRUD function at different level of depth
- Generalizes the operation using Generics
- Predefined depth levels



```
public <T> T getPreview(Class<T> objClass, String id) {
    return SessionFactoryNeo4j.getInstance().getSession().load(objClass, id, 0);
}
public <T> T getOverview(Class<T> objClass, String id) {
    return SessionFactoryNeo4j.getInstance().getSession().load(objClass, id, 1);
}
public <T> T get(Class<T> objClass, String id, int depth) {
    return SessionFactoryNeo4j.getInstance().getSession().load(objClass, id, depth);
}
```

- Controllers have more responsibility
- Too much uniformity means no clear responsibility partition

Controllers

Different controllers for different DAO:

GraphQL

- No mapping operations since DAOs can retrive needed DTOs
- Less responsibility

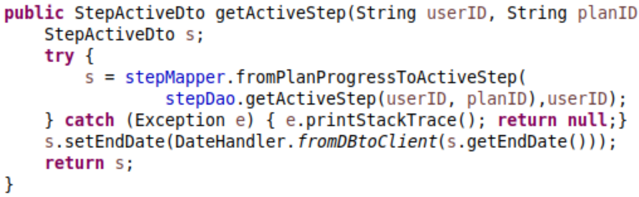
Neo4j OGM

- ID generation for each persisted object
- Mapping responsibility
- Has to do *Traversing* to obtain the needed object
- More responsibility

Both have to:

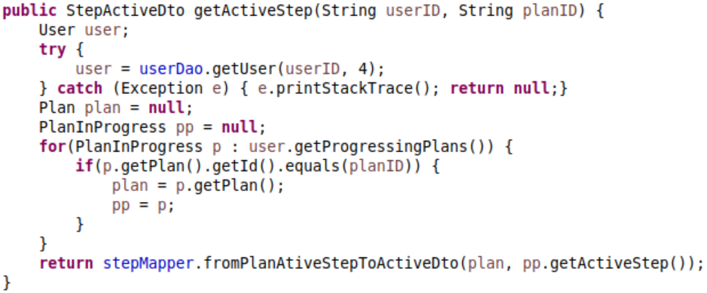
- Handle JWT Token creation
- Handle exceptions thrown by DAOs

Controllers



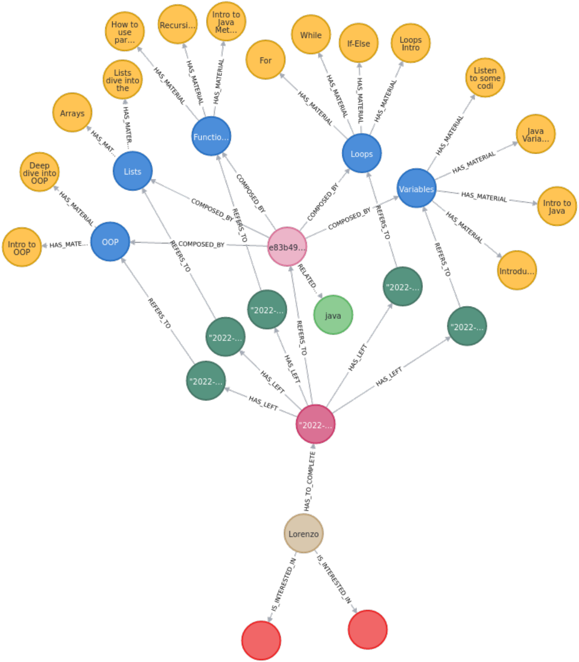
```
public StepActiveDto getActiveStep(String userID, String planID) {
    StepActiveDto s;
    try {
        s = stepMapper.fromPlanProgressToActiveStep(
                stepDao.getActiveStep(userID, planID),userID);
    } catch (Exception e) { e.printStackTrace(); return null;}
    s.setEndDate(DateHandler.fromDBtoClient(s.getEndDate()));
    return s;
}
```

GraphQL



```
public StepActiveDto getActiveStep(String userID, String planID) {
    User user;
    try {
        user = userDao.getUser(userID, 4);
    } catch (Exception e) { e.printStackTrace(); return null;}
    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());
}
```

Neo4j OGM


How to use par...
Recursi...
Intro to Java Met...
While
If-Else
Loops Intro
For
Lists dive into the
Arrays
Listen to some codi
Java Varia...
Intro to Java
Introdu...
Deep dive into OOP
Intro to OOP
Functio...
Loops
Lists
Variables
OOP
e83b49...
java
"2022-...
"2022-...
"2022-...
"2022-...
"2022-...
"2022-...
"2022-...
Lorenzo
HAS_MATERIAL
HAS_MATERIAL
HAS_MATERIAL
HAS_MATERIAL
HAS_MATERIAL
HAS_MATERIAL
HAS_MATERIAL
HAS_MAT...
HAS_MATER...
HAS_MATERIAL
HAS_MATERIAL
HAS_MATERIAL
HAS_MATERIAL
HAS_MATERIAL
HAS_MATE...
COMPOSED_BY
COMPOSED_BY
COMPOSED_BY
COMPOSED_BY
COMPOSED_BY
RELATED...
REFERS_TO
REFERS_TO
REFERS_TO
REFERS_TO
REFERS_TO
REFERS_TO
HAS_LEFT
HAS_LEFT
HAS_LEFT
HAS_LEFT
HAS_LEFT
HAS_TO_COMPLETE
IS_INTERESTED_IN
IS_INTERESTED_IN

Comparison Neo4j OGM - GraphQL

Neo4j OGM

- + supported Java client
- + certified security
- + easy and plain Java queries
- + direct interface with DB
- + caching - data incoherence
- - need to have Neo4j as DB
- - retrieved unnecessary data

GraphQL

- - no Java client
- - prone to SQL injections
- - no Java query but GraphQL like
- - middleware slows times
- - no caching
- + queries unaware of used DB
- + retrieved only necessary data

Expectations:

- GraphQL responses lighter than Neo4j OGM ones
- Neo4j OGM responses faster than GraphQL ones, faster using cache

Comparison Neo4j OGM - GraphQL Execution Times

Endpoint Execution Times (ms) between *OGM* with session having wider scope (*application*) and smaller scope (*request*) and *GraphQL*

Endpoint	**OGM Session Application scope**	**OGM Session Request scope**	**GraphQL**
*GET user/id *1*	**86**	87	186
*GET user/id *5*	**98**	108	241
POST user/id	230	317	**185**
GET plan/id	89	**87**	183
GET search/word	**182**	187	245
POST plan/create/id	**309**	330	381

- Tests executed 1000 times
- User in *1 has one started plan, in *5 has five

Comparison Neo4j OGM - GraphQL Response Dimension

Query Response Bytes

Endpoint	**OGM**	**GraphQL**
*GET /user/id *1*	785	**424**
*GET /user/id *5*	1098	**553**
GET /plan/id	1216	**696**
GET /search/word	105210	**82492**

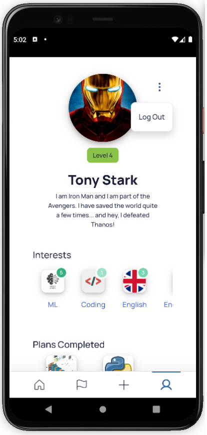
GraphQL has always smaller responses from the Middleware

In execution times we have:

- Similar behavior on GET requests for the two OGM implementations
- Application scoped cache faster when need to update user
- GraphQL always slower, except when update user

Frontend Implementation

- Angular + Taiga UI + Ionic
- State in frontend: localStorage keeps userID and Token
- Services to send requests to server (has an Interceptor to handle CORS)
- Components to handle the View and Controller responsibilities
- Angular Model as DTOs



5:02

Log Out

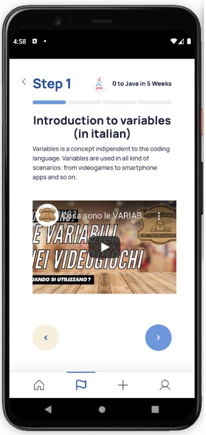
Level 4

Tony Stark

I am Iron Man and I am part of the Avengers. I have saved the world quite a few times... and hey, I defeated Thanos!

Interests

Plans Completed



4:58

Step 1

0 to Java in 5 Weeks

Introduction to variables (in italian)

Variables is a concept indipendent to the coding language. Variables are used in all kind of scenarios: from videogames to smartphone apps and so on.

Conclusions

UseCases fully implemented by the backend instead of the frontend

Pro and cons led me to choose Neo4j OGM at the moment

- Is able also to run Cypher queries
- Client developed natively by Neo4j is more stable and easier to use

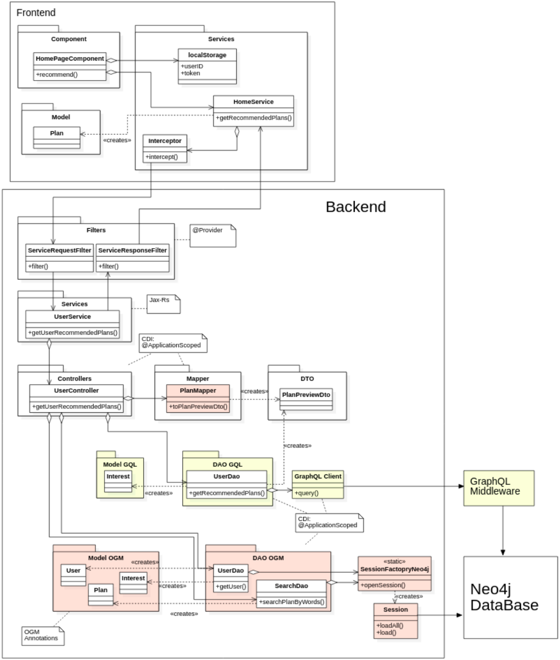
Future Works:

- Native GraphQL backend implemented in middleware
- Cypher queries in OGM to retrieve less data
- More Use Cases of the app

Thank you for your attention

Litto
Learn It Together

Complete Architecture


Frontend
Component
HomePageComponent
+recommend()
Services
localStorage
+userID
+token
HomeService
+getRecommendedPlans()
Model
Plan
«creates»
Interceptor
+intercept()
Backend
Filters
ServiceRequestFilter
+filter()
ServiceResponseFilter
+filter()
@Provider
Services
UserService
+getUserRecommendedPlans()
Jax-Rs
CDI:
@ApplicationScoped
Controllers
UserController
+getUserRecommendedPlans()
Mapper
PlanMapper
+toPlanPreviewDto()
«creates»
DTO
PlanPreviewDto
«creates»
Model GQL
Interest
«creates»
DAO GQL
UserDao
+getRecommendedPlans()
GraphQL Client
+query()
CDI:
@ApplicationScoped
GraphQL
Middleware
Model OGM
User
Interest
Plan
«creates»
«creates»
«creates»
DAO OGM
UserDao
+getUser()
SearchDao
+searchPlanByWords()
«static»
SessionFactopryNeo4j
+openSession()
«creates»
Session
+loadAll()
+load()
OGM
Annotations
Neo4j
DataBase