UNIVERSITY OF MACEDONIA
MASTER OF SCIENCE
IN APPLIED INFORMATICS

EXPLORATION OF NEO4J GRAPH DATABASE

Thesis by
Penteridou Nikolina

Thessaloniki, June 2020

EXPLORATION OF NEO4J GRAPH DATABASE

Penteridou Nikolina
Bachelor of Applied Informatics

Thesis to partly discharge the
MSc of Applied informatics obligations

Supervisor
Koloniari Georgia

Approved by the three-member committee on 26/06/2020

Koloniari Georgia	Xinogalos Stilianos	Evaggelidis Georgios

Penteridou Nikolina

Abstract

The purpose of this project was the development of a web application that offers users with no specialized knowledge of a Neo4j Graph Database queries the capability of exploring it through an interactive interface. Taking into consideration that the program should be able to function properly with any Neo4j Dataset, the first step I took was to find and then take advantage of those Neo4j queries that shape the fundamental structure of the Database. After the structure definition the entire program runs based on the defined components that are used as parameters for the corresponding functional part.

Keywords: Neo4j, Graph Database

Acknowledgements

I would like to thank my supervisor prof. Koloniari Georgia for her valuable help and the constant support she offered to me in order to complete my thesis.

Table of Contents

Catalog of images

Catalog of tables

Catalog of diagrams

1 Introduction

Data that are presented in a graph form way are able to extract clear image and very important insights for the involved dataset. Neo4j Graph Database is getting more and more popular due to its high potentials and has been inserted in a lot of big companies' implemented technologies like Walmart, eBay, Adobe.

Combining neo4j queries offers accuracy and allows developer to satisfy high level logic requirements although it needs experience in order someone to take advantage of them in the best possible way.

1.2 Aim statement

The main idea of the project is a web application development which will provide the user the possibility to navigate to a Neo4j database through an interactive interface that creates the idea of exploration. Particularly, the user, starting from a point of his or her choice, is able to proceed to the next database information based on his or her option through given "answers", sequentially

The challenging part of the current project is that the information needs to be retrieved from the Neo4j database without the knowledge of the data type and structure. That means that all the retrieved data need to arise through an abstract way during the user navigation with an Socratic method procedure.

Inspired by the Neo4j browser, which offers user the opportunity to check the graph visualization and the result tables, I tried to develop a user interface for Neo4j datasets that not only works via neo4j queries but also gives the capability to users with limited knowledge of neo4j queries to explore a dataset through navigation and interaction with options and buttons.

1.4 Contribution

The said project presents the combination for all the technologies needed in order to develop a full stack application that satisfies the initial aim. In addition, it includes tools used for the technologies and software integration. Moreover, there is summed up information about mandatory Neo4j queries that

conclude to the final result of abstraction according to the initial goal. The program could be used for educational reasons as its functionality is able to enhance Neo4j Datagraph understanding of structure for new users.

1.5 Project structure

Chapter 2 includes a Graph Database general review ending up to the Neo4j definition and relative information sourced by the Neo4j organizational. The project methodology is described in details in chapter 3. The said chapter describes the order that were followed in order the initial idea to be completed. In particular, it introduces the technologies that were used, their integration and the way the domain model was built step by step. Furthermore, it describes the fundamental queries used and the algorithmic logic behind the functionality that is triggered through User Interface as well as the developer guide so that requirements and restrictions to be cleared. The last chapter titled Epilogue sums up the total information including potential future improvements.

2 Bibliography review – theory

2.1 Data Graph Model

Data graph model is the process of "humanizing" the visualization of connected data.

The entire graph model is based on the fundamental units of the given data which are called nodes. All the relationships refer to the connections between those nodes. Adding properties and labels, the connected nodes form the graph data model which represents the database.

Actually, a data graph model considers the relationships of the nodes and emphasizes as a mandatory priority on the need of leveraging those relationships in a direct and easy way.

As Jim Webber & Ian Robinson [1] say in their report for the top Neo4j use cases "Graphs are the future. Not only do graph databases effectively store the relationships between data points, but they're also flexible in adding new kinds of relationships or adapting a data model to new business requirements."

The idea of data graph models was initially inspired by the written whiteboards that people use in order to design the connection between data to a clarified and logical model [2]. The following picture show the hand written graph model that was firstly designed on a white board.

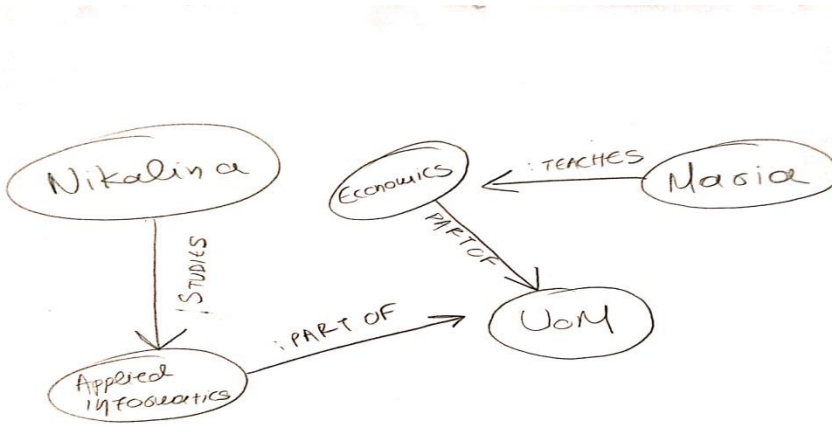

Image 1: White board hand written initial graph model

2.2 Graph Database

The general difference between relational and graph databases lies upon the way each database stores and retrieves data. As Miller said "the requirements of the system should be taken into consideration if there is a need for a dynamic data model that represents highly connected data then a graph database is the best solution. [3]"

Graph databases demonstrate their data with nodes and edges [4]; hence the final structure looks like a mathematical graph. This structure allows the easier handling of the connected data compared to the structure of the tables that are used in the case of the relational database. The relationships are described more clearly through their own properties and this straightforward model is able to focus on the use cases the entire project aims at.

Concerning the volume and the velocity, graph databases are better optimized to handle huge amount of data and to deal with big numbers of read and write executions. Scalability is one of the most important disciplines of project development and graph databases are very efficient due to their simple and clear structure as distinct from the relational table data stores. Last but not least, the cypher statement enables an easy and shorter way of querying the database in order to retrieve data, avoiding the potential need of multiple join in sql statement [2]

2.3 Neo4j Graph Database

Neo4j is a non-relational graph database that provides its own implementation of graph theory concepts. The Labeled Property Graph Model in the Neo4j database consists of the following components:

- Nodes.

Nodes comprise the main data elements of the graph. They are interconnected through relationships and they owe labels and properties. A node can have one or more labels, properties and relationships.

- Relationships.

A relationship represents the type of connection between two nodes. Relationships can have one or more properties and also a node is able to have multiple relationships.

Every node is characterized by one or more labels. This is how nodes are grouped in the same category. A node can have just one or many labels. Labels can be indexed to speed up finding nodes in a graph.

Both nodes and relationships owe properties. They are attributes with a key-value pair form. Properties can be any type of data (String, Integer, Boolean etc.).

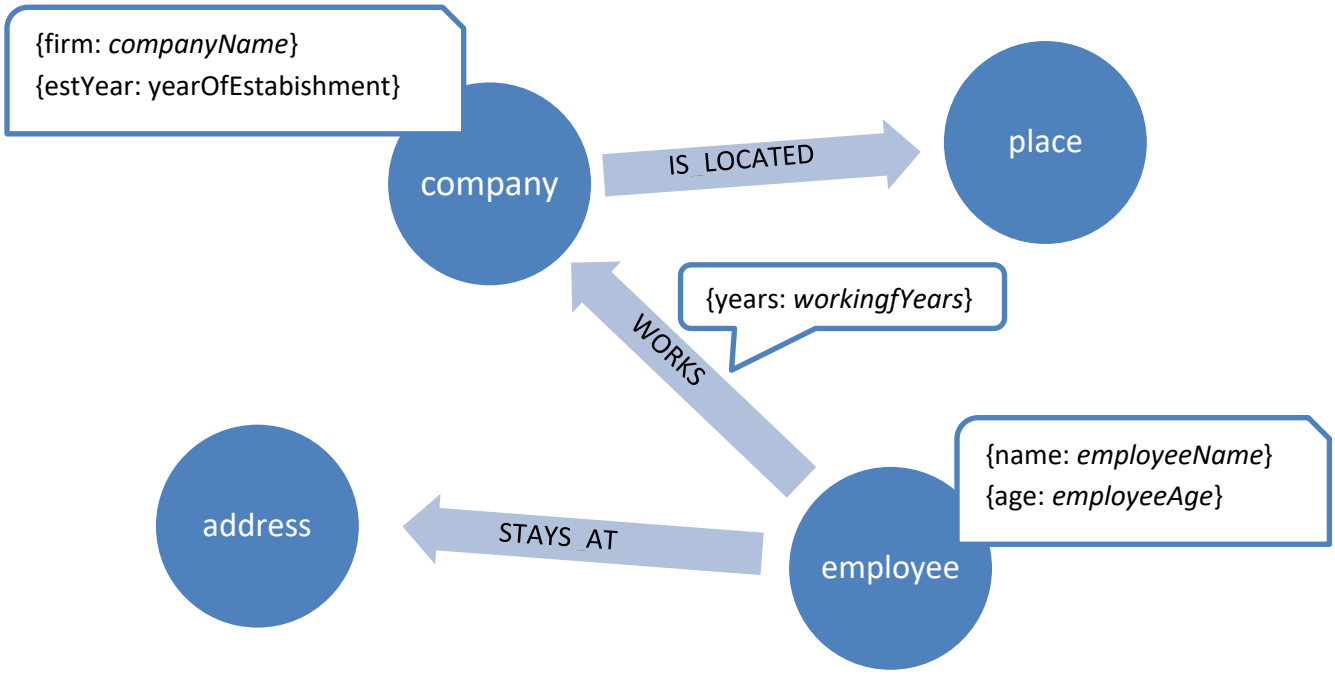

Diagram 1: Neo4j Graph Model

2.4 Why neo4j?

NoSQL databases are simpler and easier to handle over the relational databases. In case of Neo4j graph database though the perquisites also include scalability and flexibility.

Scalability: The clear demonstration of Neo4j database through nodes and relationships allows constant time traversals in big graphs for both depth and breadth.

Flexibility: The structure of neo4j schema enables the efficient adaptation of the database to the requested goal. The user can add and remove nodes, relationships and properties effortlessly in order to adjust the information to his or her business changes and needs.

In addition Neo4j is able to provide a real-time image of the information and form extremely useful insights.

The formal neo4j organization definition is the following :

"Neo4j is an open-source, NoSQL, native graph database that provides an ACID[1]-compliant transactional backend for your applications. Initial development began in 2003, but it has been publicly available since 2007. The source code, written in Java and Scala, is available for free on GitHub or as a user-friendly desktop application download. Neo4j is referred to as a native graph database because it efficiently implements the property graph model down to the storage level. This means that the data is stored exactly as you whiteboard it, and the database uses pointers to navigate and traverse the graph. In contrast to graph processing or in-memory libraries, Neo4j also provides full database characteristics, including ACID transaction compliance, cluster support, and runtime failover – making it suitable to use graphs for data in production scenarios."

Neo4j.com

2.5 Neo4j Use cases

Neo4j top use cases according to the neo4j organization white paper involve:

- Fraud detection

Neo4j present a clear image of node relatedness and consequently increase the possibilities of uncovering patterns designed by the connections between data that imply a potential fraud.

- Real-time Recommendations (Walmart, Adidas)

Neo4j is built to "understand" the strength of connections between the entities. As a result identifying the individual interests eventuates from the node interactions which gives a very meaningful insight into the market needs and trends.

[1] atomicity, consistency, isolation and durability

- Master data Management (Pitney Bowes)

Storing the master data at a graph database demands cheaper resources compared to the relational ones and also it is less complex to relate them and design the final model.

- Network & IT Operations (eBay)

The neo4j graph visualizes the type of nodes and their connections. A lot of visualization tools are available so that IT managers are able to give entities colors , sizes and labels in order to have the image that helps them to separate the database units and report their conclusions.

- Identity & Access Management (Telenor)

A graph database can store complex data connections and provide dynamic structure and environment. This structured data model supports both hierarchical and non-hierarchical structures, while its extensible property model allows for capturing rich metadata regarding every element in the system. [1]

3 Methodology

3.1 Introduction

Starting with my thesis it was a necessity to obtain the mandatory knowledge of Neo4j Graph database queries. Consequently, I began to discover some of the main database potentials. Due to the fact that the project should be based on flexibility and abstraction when it comes to the data schema, my goal was to use the appropriate queries that work efficiency without including specific data information, but only taking into account the main schema structure of any Neo4j database. After being familiar with the Neo4j query syntax, I installed the Neo4j Desktop to build my data. Following up the successful installation , I connected the database to my java code using the Spring Boot in order to simplify the entire procedure of building a web application. My effort to connect the schema to my java code led me to the necessity of using the Object Graph Mapping tool. With this tool I managed the easy and effective mapping of my

classes to the database entities. My next step was to parse the information to the User Interface. In order to do this, I used get request which were called inside JQuery AJAX calls and are displayed after being filtered through some JavaScript logic in the application front end.

3.2 Technologies used

In order to develop a full-stack application project, I discovered and used technologies, tools and libraries that helped me build mappings and connections easier and also to reduce my project size satisfyingly without reducing the optative efficiency. The technologies used for this project development are presented in the following part.

3.2.1 Neo4j Graph Database

Starting from the application back end, neo4j Graph database were used to store and retrieve the information the user ask for. Entities, their properties and the relationships between them are stored in a Neo4j Database which is integrated with the java code using the Object Graph Mapper (OGM)[explained lower].

3.2.2 Java Language

Java programming language used as the back end software of the project implementation. The object- oriented nature of java fosters the database schema representation with classes and properties.

3.2.3 JavaScript / JQuery

JavaScript was the programming language used for the front end project implementation combined with the jQuery Library which made the handling of the document object model (DOM[2]) in HTML file easier.

[2] Defines a tree-like data structure for representing the HTML document regarding the elements it includes.

3.2.4 Spring Framework / Spring Boot

Spring is a java framework ideal for building web applications. It was used to support the restful web services by providing an easy way to handle them with dedicated annotations usage.

Spring Boot is a Spring framework implementation used to provide faster and shorter project development. It needs less dependencies and provides effortless micro services building. Spring Boots annotations used on this project are described later one by one.

3.3 Technologies Integration

Technologies and tools are able to consist a completed web application only when they are connected and combined. To succeed this it is needed to use integration tools and software. The supporting software I used to make the said technologies work and cooperate together is described below.

3.3.1 Maven

Maven is a build automation and management tool. It requires dependencies in order to integrate the requested functionality effortlessly. It simplifies the technologies combination while it manages to make all the necessary information available through a few lines of code. Maven functionality occurs in an xml file that is includes to the project.

Below, you can see the required dependency in order to be able to access data with Neo4j.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-neo4j</artifactId>
</dependency>
```

3.3.2 Object Graph Mapper

Object graph mapper Library used for relating the graph entities to java class in order to build the domain model.

Particularly, it maps nodes and relationships in the graph to objects and references in a domain model using annotations. Object instances are mapped to nodes while object references are mapped using relationships, or serialized to properties.

Neo4j-OGM dependencies consist of neo4j-ogm-core, together with the relevant dependency declarations on the driver you want to use. The driver used in the specific project is the neo4j-ogm-bolt-driver which uses native Bolt protocol to communicate between Neo4j-OGM and a remote Neo4j instance.

In order to use the OGM library, we first need to add the following dependency using Maven.

```
<dependency>
   <groupId>org.neo4j</groupId>
   <artifactId>neo4j-ogm-bolt-driver</artifactId>
   <version>3.2.4</version>
</dependency>
```

3.3.3 Spring MVC

The Spring MVC is a Java framework that provides Model-View-Controller (MVC) architecture and is used for building web applications. It offers ready components that can be used to develop flexible and loosely coupled web applications [5] .

The way Spring MVC is used for the current project implementation is described in chapter 3.5.1 in details.

3.4 Building the domain model

3.4.1 Data Acknowledgment

The data model I worked on demonstrates a book management system which includes books, authors, editors, and users who interact with each other.

The Book Managements system is going to have the following nodes (with attributes in parentheses):

- *Book (title, price)*
- *Author (name)*
- *Genre (type)*

- *Editor (name, address)*
- *User(name, surname, birthday)*

Those nodes represent the node entities of the model and the relationships between them are the following:

- Author :WROTE Book
- Editor :PUBLISHED Book
- Book :HAS_CATEGORY Genre
- User :RATED(rank) Book

The final schema based on nodes and relationships is demonstrated in the diagram below:

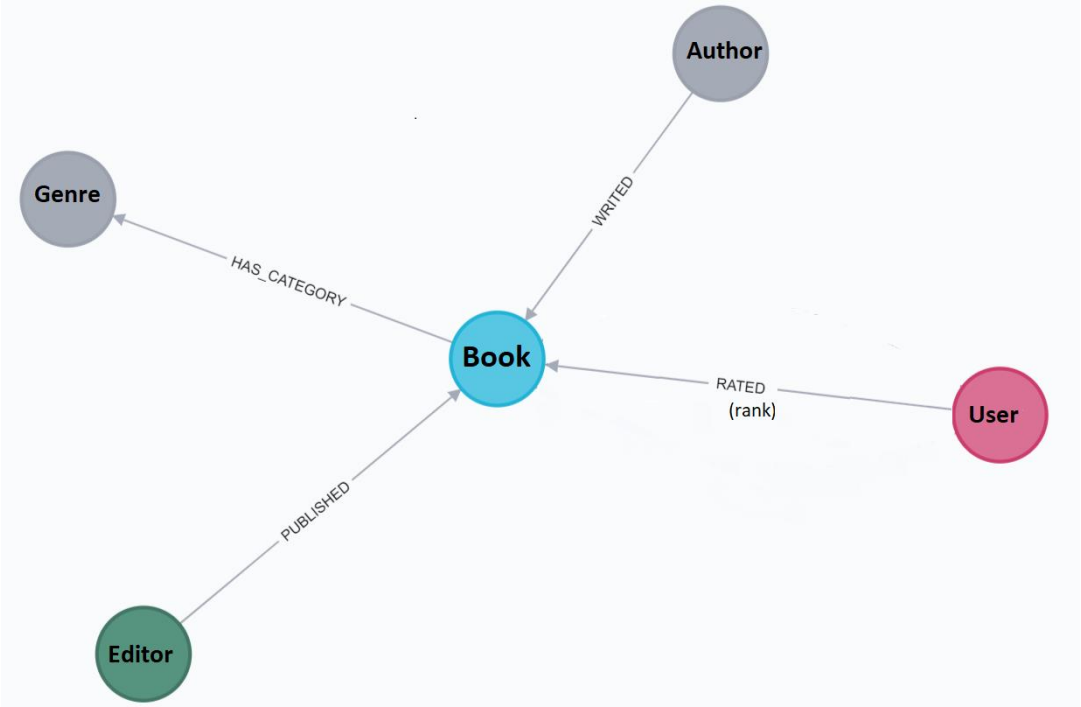

Diagram 2: Schema for Book Management dataset

In order to convert our data graph model into a class centered model, so that we can use object-oriented programming language, we need to adjust the previous nodes to java classes and to their relationships using composition. Before the said step though, from the above information about relationships arises the necessity for two more classes except for the obvious ones which are Book, Author, Genre, Editor and User. Those two classes are the Borrowing class and the Rating class from the "rated" relationship . This relationship class represents the graph "rich" relationships while it owns attributes. Hence, the creation of this

classes is related to the relationship attribute it is followed by which in this case is: rank .

As a consequence, the added relationship entity is:

- *Rating(rank)*

After the last important addition the java classes occur with their attributes and relationships are:

- *Book (title: string, price: double, author: Author.)*
- *Author (name)*
- *Genre (type)*
- *Editor (name, address)*
- *User(name, surname, birthday)*
- *Rating(rank)*

3.4.2 Plain Old Java Objects

The Plain Old Java Objects (POJOs) gives the possibility to manage the properties of graph's relationships. Neo4j-OGM is able to be used in non-annotated objects models as well. It is capable of saving any POJO with no annotations to the graph, as the framework applies conventions to decide what to do. This is useful in cases when there is no or limited control over the classes that we want to persist. The recommended approach from the neo4j org [2], however, is to use annotations wherever possible, since this gives greater control and means that code can be refactored safely without risking breaking changes to the labels and relationships in your graph.

3.4.3 Driver

The driver is considered as main prerequisite in order to allow communication between Neo4j-OGM and a remote Neo4j instance.

For this application building I used the neo4j-ogm-bolt-driver via adding the following dependency for maven:

```
<dependency>
   <groupId>org.neo4j</groupId>
   <artifactId>neo4j-ogm-bolt-driver</artifactId>
```

```
   <version>3.2.4</version>
</dependency>
```

3.4.4 Types of annotations used

There are a lot of annotations used inside the Java code during the project implementation. The next section describes the way I took advantage from each of them.

Spring Boot annotations

@RestController: The class that handles the requests and calls the corresponding function to satisfy them.

@GetMapping: declares a “get” method request.

@PathVariable: declares any parsed variable via URL for the request requirements.

@Repository: annotates a class to define it as storage for the results of queries executions.

@Service: the classes of code that include all the company logic of the program. Each service represent a specific requirements satisfaction within the project goal.

Neo4j-ogm annotations

@NodeEntity: The java classes which represent the graph nodes (Book, Author etc.).

@Query: it is used to annotate a function which triggers the following query when it is executed.

@Property: Nodes and relationships attributes(name, date etc.). They support types of String, Long, Date, any Object etc.

@Relationship: It annotates the composed attributes of the current class which are another class’ objects. For example Author will be annotates with this tag as a

Book’s composed attribute followed by the type of the relationship into a parenthesis (type = "WROTE") and there is the possibility of defining the direction as incoming/outgoing as well. The type of relationship can be represented with an array if we want to create a "to many" relationship.

@RelationshipEntity: It concerns what we call rich relationships. That means relationships that need some more attention while they “carry” important information with their attributes and their values. This tag is followed by the type of the relationship entity into parenthesis(type = "BORROWING").

@StartNode/@EndNode : Define relationship’s direction. These annotations are used inside the class annotated with @RelationshipEntity.

@NotNull: this annotation not allows any property to have null value. The program will throw a nullpointer exception error?

@Id / @GeneratedValue: Id annotation defines the entity id. @GeneratedValue annotation discharges the developer of using any existed ids and fills them in automatically with an increasing by one rate. A good practice that involves the @Id and @GeneratedValue annotation is to include them in an abstract entity class which all of the node entities will inherit.

@JsonIgnoreProperties: Spring Boot initially creates all the existed beans while the program starts running. As a result , composed entities-classes that are connected with relationships throw an error: what error?? By using JsonIgnoreProperties annotation you actually “tell” the system to ignore the current property and keep on initializing the program, so that you can avoid the infinity loop that would make the functionality unable.

@DateLong: this annotation is used to convert the neo4j Long date information to Java Date type.

3.5 Functionality

This chapter demonstrates the code functionality flow that is followed after the triggering by the user that interacts with the program through buttons and

elements on screen. From the user action and the queries execution to the final results display, the process cycle is described in details step by step.

3.5.1 Spring MVC flow

The diagram below shows the spring MVC flow when a request occurs in the present application. The rest controller initially receives the request. Through methods annotated as request mapping or at ones get mapping the URI information passes to the respective service method. Services include the business logic part of the project. That means that all the queries which retrieve data are implemented in methods inside services. In case we want to run a query and store the information we need to use repositories. Regarding to the current project those repositories extends the neo4j repository in order to store the data in a way compatible to the Neo4j schema structure.

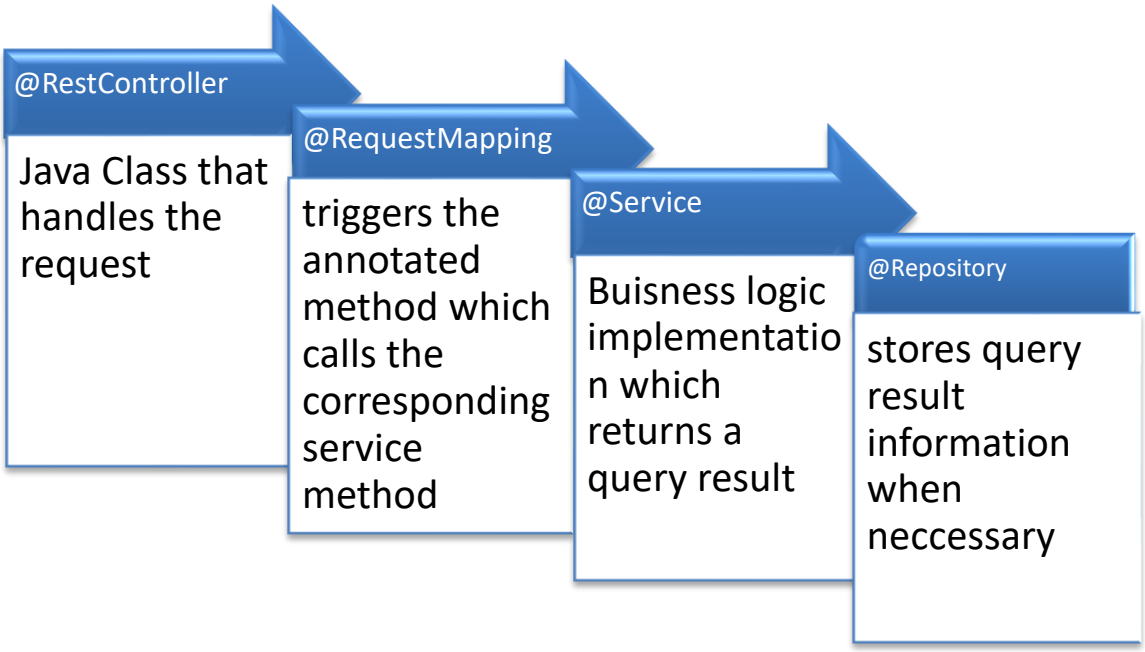

Diagram 3: Spring MVC flow when a request occurs

Below, you can find the source code which represents the mentioned flow process. In the first case the request is about a simple question to the database, whether in the second one we need to store the result information in the database by using a repository annotation.

Case 1: Retrieving data

The Rest controller class receives the request due to the annotated method

```
@RestController
public class NodeController<T> {
[…]
       @GetMapping("/node/count/{node}")
         public Object getNodeCount(@PathVariable String
node){
       return  nodeService.getNodeCount(node);
         }
[…]
}
```

The nodeService Service class contains the method which implements the relative query and returns the result.

```
//COUNT a node instances
public Result getNodeCount(String node){
    Map<String,Object> params = new HashMap<>();
    String query="MATCH (n:" + node + ") RETURN count(n) as
value";
    return
Neo4jSessionFactory.getInstance().getNeo4jSession().query(quer
y,params);
}
```

Case 2: storing the data by using a repository in case of import:

The Rest controller class receives the request due to the annotated method

```
@RestController
public class BookController {
[…]

       @GetMapping("/importbooks")
        public String importBooks() {
        bookImportService.importBooks();
        }

[…]
}
```

The bookImportService Service class which extends the Neo4jRepository contains the method that calls the Repository method importBooks()

```
//import books
public void importBooks(){
     bookRepository.importBooks();

}
```

Annotated by @Query the query result is stored in a Collection of Book objects.

```
@Query("LOAD CSV WITH HEADERS FROM \"file:///books.csv\" AS
Line\n" +
        "MERGE (b:Book {title: Line.title, year:
toInteger(Line.year)  , price: Line.price})" +
        "MERGE (e:Editor {publisher: Line.edition })\n" +
//checked ok --> csv works fine , the problem is the entity
        "MERGE (a:Author {writer: Line.author})\n"+
        "MERGE (a)-[:WROTE]->(b)\n" +
        "merge (e)-[:PUBLISHED]->(b)" )
Collection<Book> importBooks();
```

The image below portraits the general image of the project as a multiple leveled burger from the bottom to the top, like a full stack project that uses the referred technologies and tools.

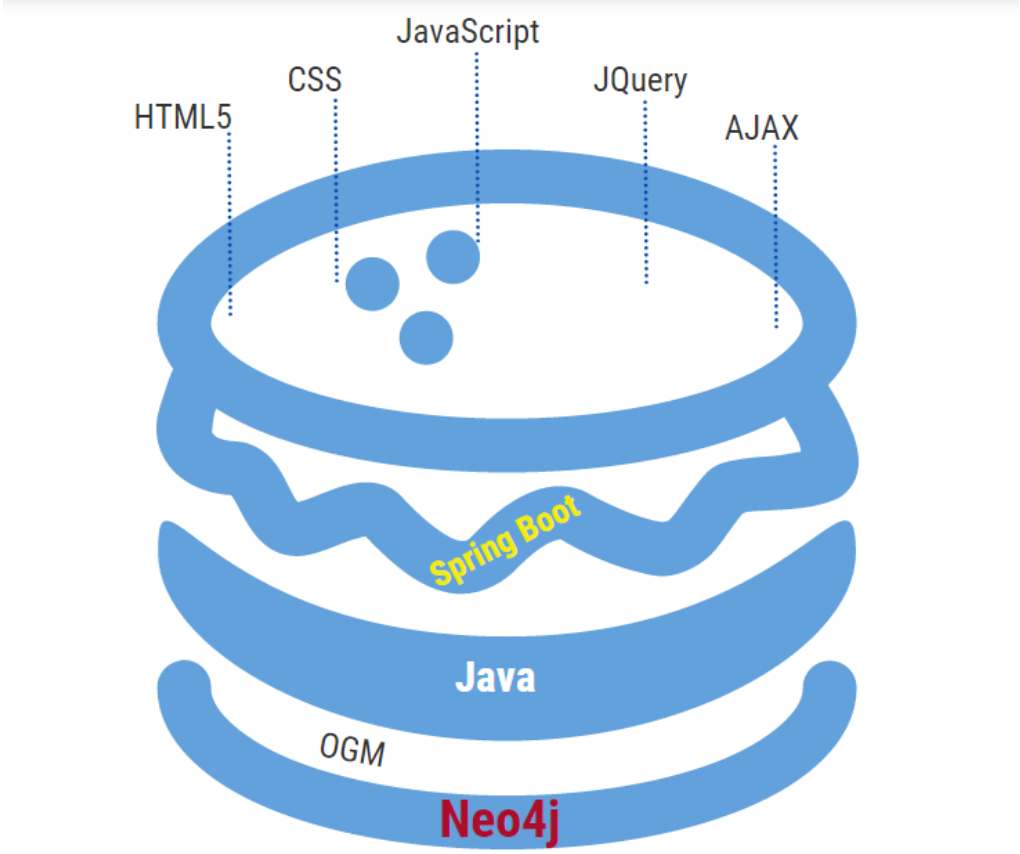

Image 2: The full-stack image of the project

3.5.1 Fundamental queries

Neo4j database provides user a big range of potentials through logical queries. Once the project development started, it was mandatory all the fundamental queries to be collected in order to:

- Provide a better understanding of the data graph structure and
- be combined the way they fulfill the project needs.

The main goal during the neo4j queries studying was those queries which would be used in an abstract way regarding the data to be found. In particular, the effort was focused in queries usage and combination that do not include a specific node or a specific relationship, but only general entities and functions. What is of grave importance is that the entire project is based on flexibility and abstraction.

On the table below there are some fundamental queries and their description.

Table 1: Fundamental queries

Query	Result description
call db.schema()	Get the schema with indexes and constraints
MATCH (n) RETURN distinct labels(n)	Get all nodes' labels
MATCH (n) return collect(distinct labels(n))	Get all nodes' labels **in a table**
match (n) **with keys(n) as nested** **unwind nested as x** **return distinct x**	Get all properties unnested
MATCH p=()-[r:RATED]->() RETURN distinct keys(r)	Get all relationships properties

match (n) **with keys(n) as nested** **unwind nested as x** **with collect(distinct x) as properties_list** **return properties_list**	Get all properties unnested in a list
match (n) **with keys(n) as properties, labels(n) as entities** **unwind properties as prop** **with prop, entities** **unwind entities as ent** **return distinct prop, ent**	Return unwinded properties with the nodes they belong
MATCH (n) RETURN distinct labels(n), count(*)	To get the node count for each label
MATCH (n) RETURN DISTINCT keys(n)	Properties of ALL nodes distinct (no values)
MATCH (n) RETURN DISTINCT keys(n) , labels(n)	Properties of ALL nodes distinct (no values) and the node they belong
MATCH (n)-[r]->(m) RETURN distinct type(r);	Relationship types distinct
MATCH (n)-[r]-(m) RETURN distinct type(r), count(type(r)) order by count(type(r)) desc;	Relationship types and count
MATCH (n)-[r]->(m) RETURN distinct type(r) as relat_type,labels(n)as label1 ,labels(m) as label2, count(*) as num_of_relat order by count(*) desc ;	Relationship types for couples (distinct) , related nodes , and count of types (neighboring nodes) .
MATCH (n) **RETURN DISTINCT labels(n) as Label,keys(n) as properties,size(keys(n))as num_of_prop ,count(*)as count order by count(*) desc;**	See the **properties of each node** , the num of properties and the counter.
MATCH (n:Node) **RETURN DISTINCT keys(n)**	Properties of A node DISTINCT (no values)

MATCH (c:Node) **RETURN DISTINCT keys(c), size(keys(c))** **ORDER BY size(keys(c)) DESC**	Properties of a node once and their num of prop in desc order (no values)
MATCH (c:Node) return ID(c), keys(c), size(keys(c))	see ALL the nodes properties' names and id's with VALUES

3.5.2 Interacting with the elements

The entire functionality of the program is implemented inside functions that Java and JavaScript execute. In this chapter section you can see the entire flow the program follows in order to satisfy any user request triggered through the user interface. The following diagram describes the general flow when a request call occurs:

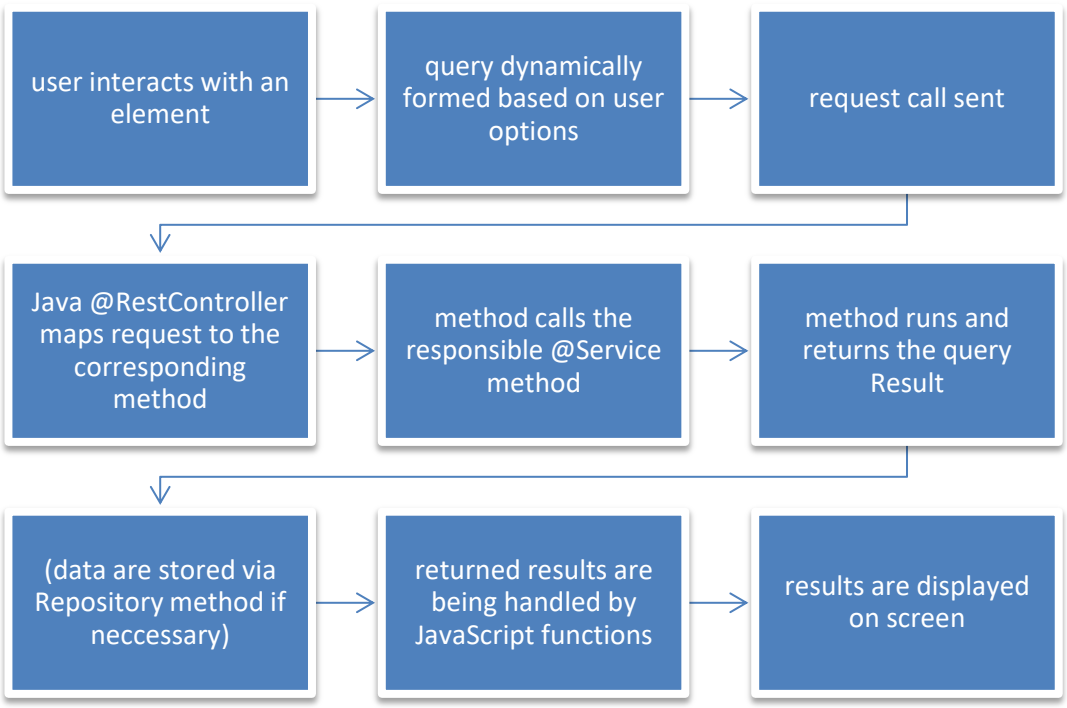

Diagram 4: Request – response call flow

Most of the request handling is being serviced by the NodeController class which is annotated as a @RestController. This class is generic in order to satisfy any type of class imported as an entity. The NodeController functions call the responsible services to run the related code that completes the request as the

diagram described. Next sections analyze the previous flow using more specific cases of requests.

3.5.2.1. Home page

After opening the page the program is ready to accept and response to user actions. The first path of user choice will run some of the referred fundamental queries that will define the schema structure regarding to the nodes, the relationships and their both properties. Those queries are totally abstract and return the main composition of the data set as it is explained later.

In particular, entering the home page, user is able to interact with the following elements:

- A 3.5.2.1 Clear button
- A 3.5.2.2. Data import button
- A 3.5.2.3 Graph Visualization
- A 3.5.2.4 Drop down list with three options : nodes, properties, relationships
- A 3.5.2.5 Search input (up to two keywords inserted)

In general, environment that includes the elements just referred:

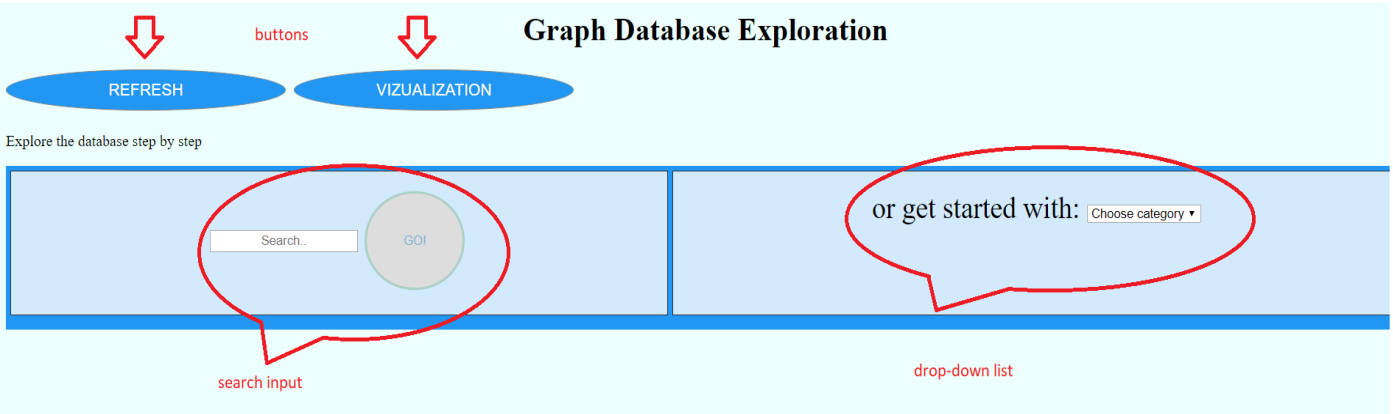

Image 3: home page screenshot

3.5.2.1 Clear button

Starting from the top, the clear button refreshes the page and brings the user back to the initial home screen anytime. It works same way as the browser refresh button. It loads the entire page from the beginning.

3.5.2.2. Data import

The mandatory step in order to use the application is the dataset import. The import button on click starts the function that sends the "/importschema" get request and the code below is being executed in the program back end:

```
@GetMapping("/importschema")
public String importschema() {
    schemaImportService.importSchema();
    return "Schema Imported";
}
```

The function of "schemaImportService" @Service that is called runs another part of code that communicates with a @Repository class' function. This is necessary so that the data to be stored.

```
@Service
public class SchemaImportService {

    @Autowired
    NodeRepository nodeRepository;
    //import books
    public void importSchema(){
        nodeRepository.importSchema();

    }
}
```

Finally, the "nodeRepository" @Repository runs the corresponding function "importScehma()" which is annotated by @Queryand its type is a Collection with EntityNode objects as you can see in the code below:

```
public interface NodeRepository<T> extends Neo4jRepository<T, Long> {

    Optional<T> findById(Long id);
//---------IMPORT SCHEMA-------------------------------------------------
---------

    @Query("LOAD CSV WITH HEADERS FROM \"file:///books.csv\" AS Line\n" +
            "MERGE (b:Book {title: Line.title, year: toInteger(Line.year)
, price: Line.price})" +
```

```
            "MERGE (e:Editor {publisher: Line.edition })\n" +
            "MERGE (a:Author {writer: Line.author})\n"+
            "MERGE (a)-[:WROTE]->(b)\n" +
            "merge (e)-[:PUBLISHED]->(b)" )
  Collection<EntityNode> importSchema();
```

The EntityNode class is used to add abstraction to the code. The main function of this class is described in later chapter.

3.5.2.3 Graph Visualization

The visualization button opens a new page where the user has the opportunity to get a general image of the data structure and information through a graph visualization. The library used inside the JavaScript code for drawing the graph is Neovis.js and the script used is:

```
<script src="https://cdn.neo4jlabs.com/neovis.js/v1.3.0/neovis.js">
<script>
```

The visualization button triggers the function that includes the configuration which is shown in the code below:

```
function(){
console.log("vis script running")
      var config = {
        container_id :"viz",
        server_url:"bolt://localhost:7687",
        server_user:"neo4j",
        server_password:"pass",
        labels: {
        "Genre": {
                                caption: "user_key",
                                size: "pagerank",
                                community: "community"
                            }

        },
        relationships:{
         "HAS_CATEGORY": {
                                caption: "user_key" ,
                                thickness: "count"
                            }

        },
        initial_cypher: "MATCH p=(a)-[r]-(b) return p"
      }

      var viz = new NeoVis.default(config);
      viz.render();
```

```
});
```

As you can see the way the data will be demonstrated and most important the query that extracts the graph are defined in one line inside the configuration and in this case is:

```
"MATCH p=(a)-[r]-(b) return p"
```

Next picture demonstrates the HTML page opened with the final graph visualization based on the Book management data sample.

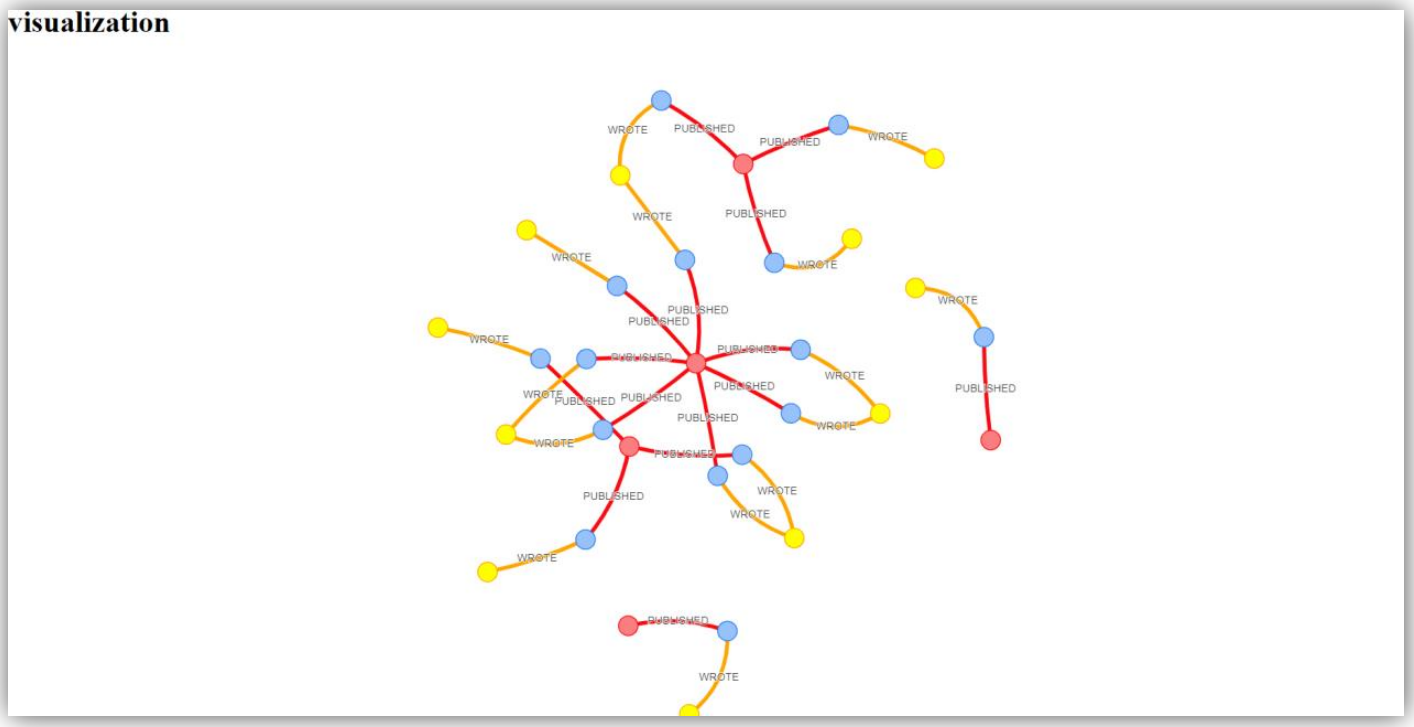

Image 4: visualization page screenshot

3.5.2.4 Drop down list

The drop down list element helps the user to start exploring. The given options of the list are: nodes, relationships and properties. All the returned results are buttons so that the user chooses the next data to proceed. The following images show the screen after choosing nodes, properties and relationships, respectively.

Image 6: Displayed result buttons with “nodes” option selected

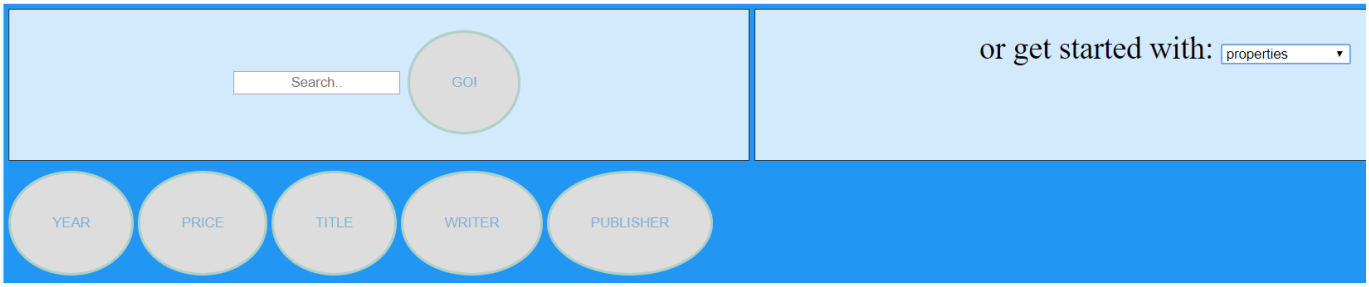

Image 5: Displayed result buttons with “properties” option selected

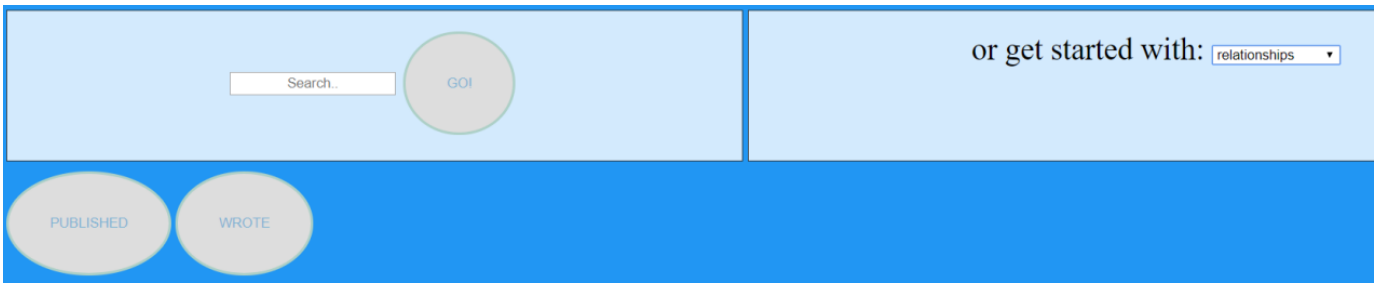

Image 7: Displayed result buttons with “relationships” option selected

The drop down list functionality depends on the user option but the main goal of each function is common. Regarding the option the program executes a function that sends the relative request to the server in order to run the required query so that the correct information from the database to be retrieved. The list options are nodes, relationships and types and the way the program sends the respective request and the data handling is shown it the code below.

```
$(document).ready(function(){
   $( "#mainList" ).change(function() {

       […]

1) assign the selected option’s value to the option variable
       var option = $( "#mainList option:selected" ).val(); //option
values: shownodes, showrelationshiptypes, showproperties,
```

```
            $.ajax({
                type: 'GET',
2) form the request by using the parsed option value as a path
variable
                url: 'http://localhost:8080/' + option,
                dataType : "json",
                contentType:"application/json",
                success: function(data){
                    3)response handling (check next chapter)
                            […]
                        }
                }
            });
   });
});
```

where #mainList id represents the said drop down list.

The "option" variable represents any of the three options and is included in the URL line that defines the request. The "option" variable value is filled in by the selected value of each option as it is defined in the html file:

```
<select id="mainList" class="prop-container">
    <option value="none" selected disabled hidden>Choose category</option>
    <option value="shownodes" id="nodesOption"> nodes </option>
    <option value="showrelationshiptypes" id="relOption">
relationships</option>
    <option value="showproperties" id="propOption"> properties </option>
</select>
```

Thus, the potential ajax requests that can be responded from the Controller for nodes , relationship types, and properties are the following:

```
@GetMapping("/shownodes")
public List<Object> getTheSchemaNodes(){ return (List<Object>)
nodeRepository.getAllNodes();}

@GetMapping("/showrelationshiptypes")
public List<Object> getTheSchemaRelTypes(){ return (List<Object>)
nodeRepository.showRelTypes();}

@GetMapping("/showproperties")
 public List<Object> getTheSchemaProperties(){ return (List<Object>)
nodeRepository.getAllProperties();}
```

Once more the NodeController needs to call the @Repository "NodeRepository" where the data are stored in order to execute the abstract fundamental query

and the information to be retrieved. The following code section includes the implementation of the called functions, respectively:

```
public interface NodeRepository<T> extends Neo4jRepository<T, Long> {

Optional<T> findById(Long id);

[…]

//get node labels
@Query("MATCH (n) \n" +
        "with labels(n) as nested \n" +
        "unwind nested as x \n" +
        "return distinct x")
Collection<Object> getAllNodes();

//get relationship types
@Query("MATCH (a)-[r]-(b) RETURN distinct type(r) as rel_types")
Collection<Object> showRelTypes();

//get properties (names)
@Query("match (n) \n" +
        "with keys(n) as nested\n" +
        "unwind nested as x\n" +
        "return distinct x\n")
Collection<Object> getAllProperties();


}
```

3.5.2.5 Search input

The search input gives the user the possibility to explore the graph information in a direct way. The user is able to start interacting with the graph based on the keywords of his or her choice. The input field can accept one or two keywords in order to work properly.

In case of one keyword the user can potentially type any general or specific word related to the schema and the data. That means that he or she can search from general schema entities to specific values.

Specifically, typing the keyword “nodes”, “relationships” or “properties” could return all the node labels ,the relationship types or the properties exist ,respectively.

The image below shows the result after inserting the word “nodes” as a keyword. On the bottom darker blue frame Book, Author and Editor label nodes are displayed as buttons to interact with.

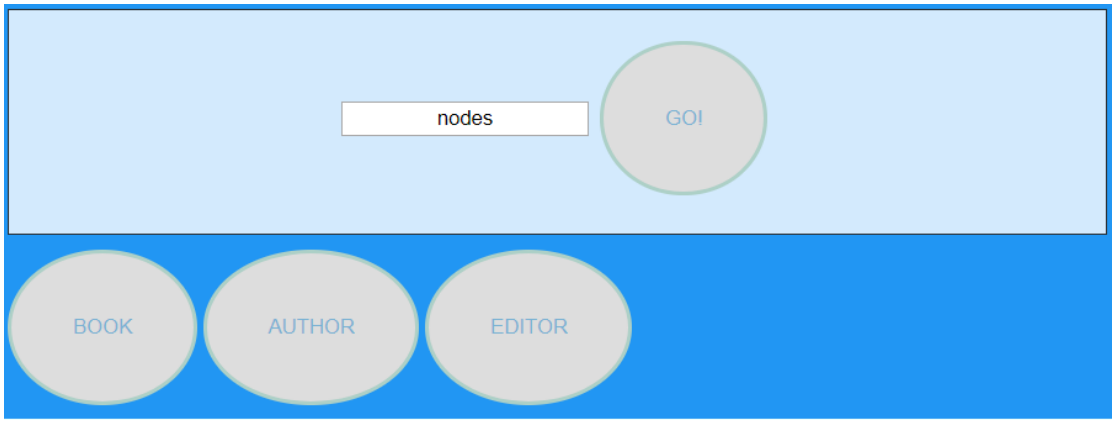

Image 8: search result screenshot with category keyword

On the other hand, the specific label of a node would return the node itself with the available buttons to proceed based on the said node as the picture below displays.

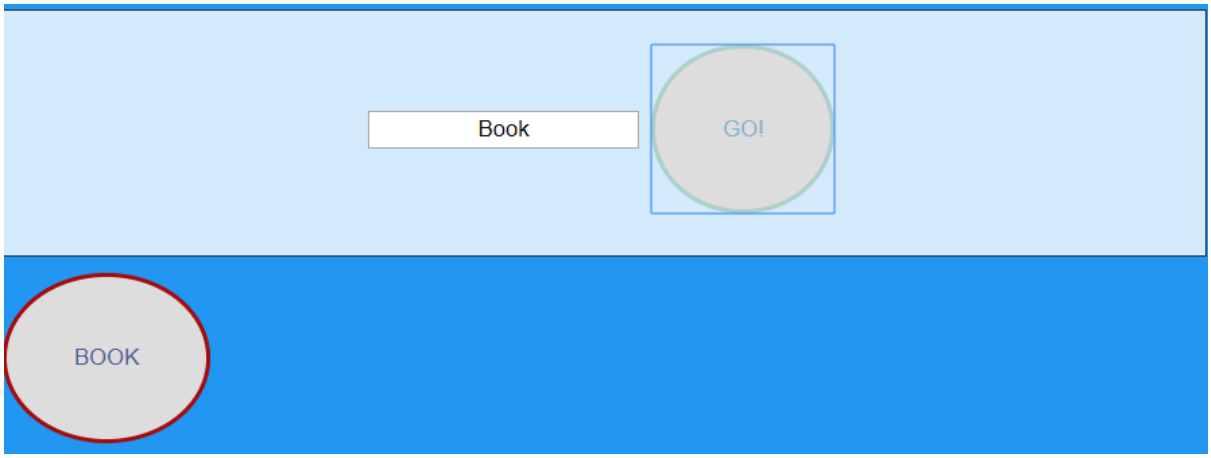

Image 9: Search result screenshot with label keyword

Finally, typing any value of any node attribute would return the entire specific node that includes this value in a table row, like in the following picture after typing the value “Potter”.

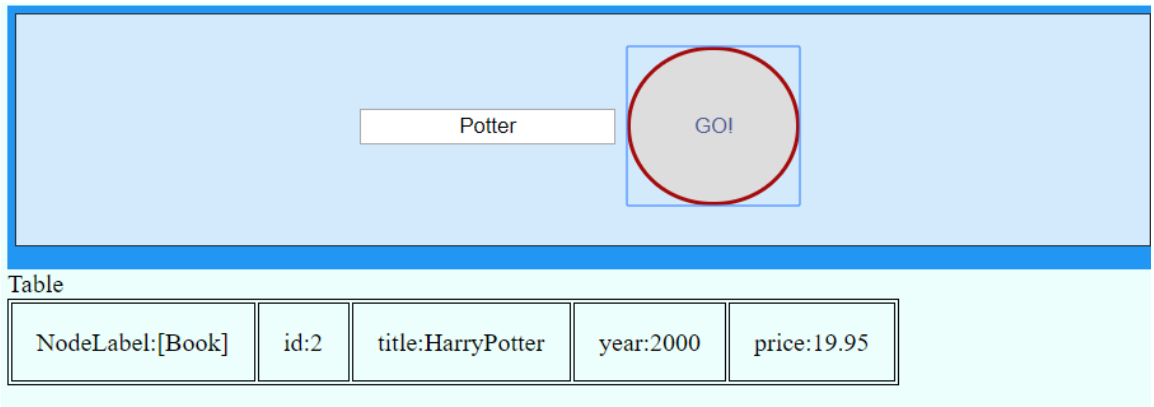

Image 10: search result screenshot with specific value keyword

The following table sums up all the keyword possibilities regarding the case, the expected result described and an example with a represented keyword.

Table 2: Keyword cases

Keyword case	Output info	Example keyword inserted
Node(s)	node labels (buttons) to proceed	*Node(s)*
Relationship(s)	relationship types (buttons) to proceed	*Relationship(s)*
Property(ies)	properties (buttons) to proceed	*Properties/property*
Node label	node labels (buttons) to proceed	*Book*
Relationship type	relationship types (buttons) to proceed	*writer*
Property title	properties (buttons) to proceed	*price*
Property value	entire node (table row) that includes the value	*HarryPotter*

In case of two keywords inserted the types and the combination of them compose the result. If any attribute value is typed the result will be focused on the value inserted regardless of the second keyword. That happens because of

the priority of the value itself as it is the most specific way of search. Hence, the search will work as in the case of one value keyword inserted.

The combination of node label with any property name would return the values of this node property in a table as it is shown in the next picture after using the example of Book price.

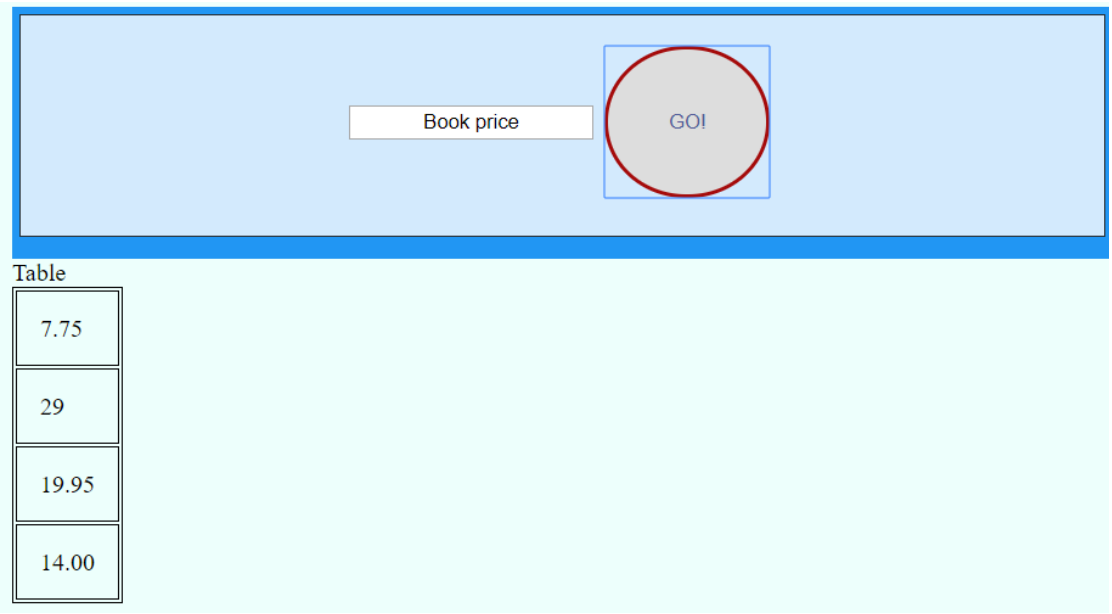

Image 11: search result screenshot with combined keywords including value

Inserting node label with any type of relationship will give the user the chance to see the involved nodes and the direction of the relationship. Moreover, the returned result is displayed in buttons in order to let user continue the exploration. Next picture was shot after inserting “Book WROTE”.

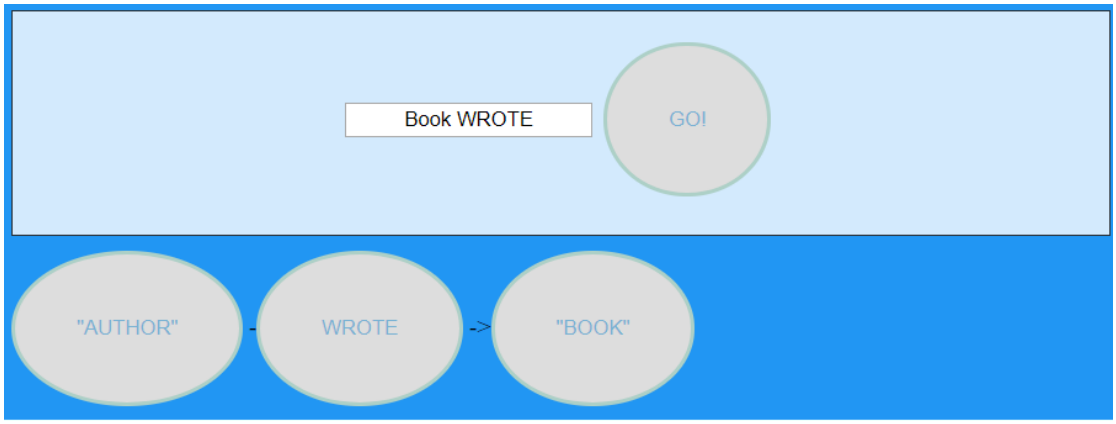

Image 12: search result screenshot with combined keywords including relationship

Another case of combining keywords is about a node label and general schema words as properties or relationships. This would return the obvious information in buttons to let the user continue. The results of this case are displayed in the following pictures.

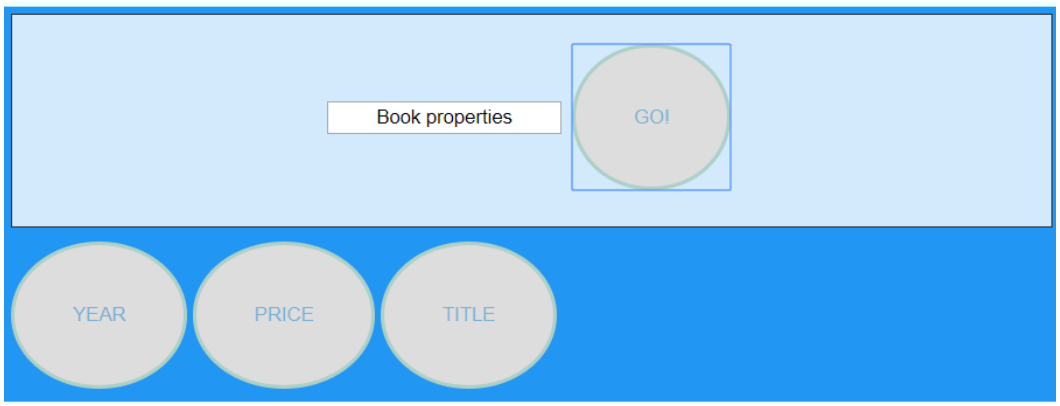

Image 13: search result screenshot with combined keywords including category "properties"

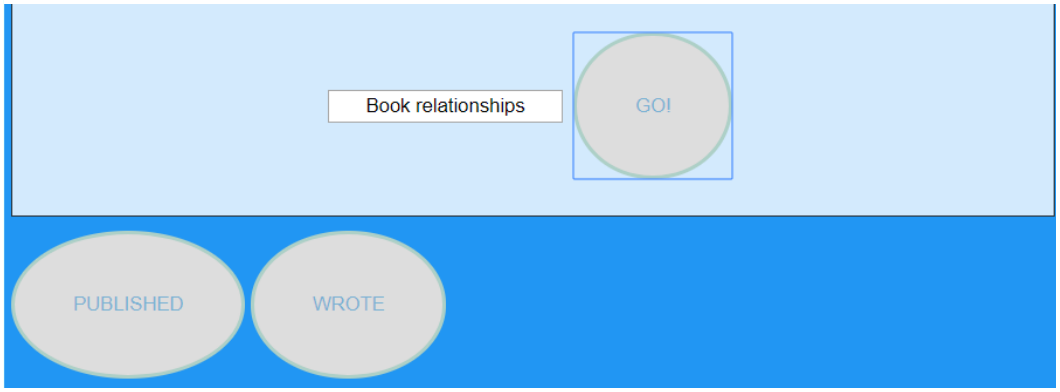

Image 14: search result screenshot with combined keywords including category "relationships"

Last but not least, the user is able to see any potential relationship between two node labels by typing them as keywords together. The result shows which the starter node is and the type of the relationship as it is shown in the picture where Book and Editor node labels used as keywords.

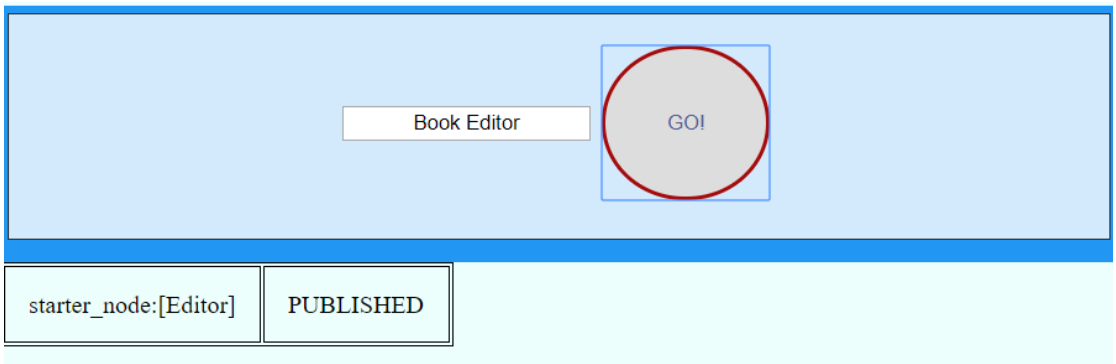

Image 15: search result screenshot with combined node label keywords

Next table gives a completed image of all keyword combinations supported by the application followed by example inputs and result descriptions

Table 3: Combined keywords cases

Keyword 1	Keyword 2	Example input	Result description
value	*		Show entire node (table row) that includes the value
Node label	Specific Property title	*Book price*	Show book price values
Node label	Specific Relationship type	*Book wrote*	The relationship with the Book node involved
Node label	Property(ies)	*Book properties*	The book properties(buttons to interact with)
Node label	Relationship(s)	*Book relationships*	The book relationships(buttons to interact with)
Node	Node	*Book Editor*	Relationship between them and the startnode

Search input functionality is based on the number of keywords input. When the "search" button is triggered the first thing to be done is the number of keywords to be defined as one or two. In both cases the code tries to figure out the type of keyword(s) using some other JavaScript functions in order to send the right request. In addition, the code is able to recognize the null value and show an alert message to the user.

After declaring the number and the type of keywords the program calls the corresponded function that executes the request including the keyword input as path variable.

<u>In case of one keyword</u>

- If the word typed belongs to any category like nodes, relationships or properties, the program declares and sends the same request as in the case of the drop down list described previously:

```
@GetMapping("/shownodes")

[…]
@GetMapping("/showrelationshiptypes")
```

```
[…]
@GetMapping("/showproperties")

[…]
```

- Else If the word typed is any value and not a category as referred just before , the request that is sent includes the said value through “keyword” variable as you can see in the following code :

```
$.ajax({
      type: 'GET',
      url: 'http://localhost:8080/search/' + keyword,
      dataType : "json",
      contentType:"application/json",
      success: function(data){

            […]

        }
});
```

The @RestController NodeController receives the request and runs the function including the keyword transferred as a @PathVariable :

```
@GetMapping("/search/{keyword}")
public Object getObject(@PathVariable String keyword) {
    return objectRecognitionService.setParamsAndSearch(keyword); }
```

The function setParamsAndSearch with keyword as a parameter is implemented inside the @Service ObjectRecognitionService which is the most critical part behind the entire search functionality. Let us have a better look of this @Service function’s code logic by using pseudocode and comments instead of the java commands:

```
Result setParamsAndSearch(keyword){
    //check the type of the keyword

    if(keyword is any property title)){

        1)Get all the properties listed inside a Map collection

        2)Run a query that unwinds these properties one by one and matches
```

```
        the node if any of these properties equals the keyword value and
        finally returns the result of the matched property as
        “showproperties”.

         "unwind {props} as prop" +
         "MATCH (n) WHERE prop  =~ '(?i).*"+keywordInserted+".*' " +
         "RETURN  distinct prop as showproperties"

        3) Return the query Result.

    }else if (keyword is any node title){

        1)Get all the node labels listed inside a Map collection

        2)Run a query that unwinds these node labels one by one and matches
        the node if any of these node labels equals the keyword value and
        finally returns the result of the matched node as “shownodes”.

        "MATCH (n) " +
        "unwind $nodes as node" +
        "match (n) where node=~ '(?i).*"+keywordInserted+".*' " +
        "RETURN distinct node as shownodes "

        3)Return the query Result.

    }else if (keyword is any relationship type title){

        1)Run a query that matches the type of relationship if the second
        one equals to the keyword value and returns the result of the
        matched relationship type as “showrelationshiptypes”

        "MATCH (a)-[r]-(b) " +
        "WHERE type(r)=~ '(?i).*"+keywordInserted+"*'" +
        "RETURN distinct type(r) as showrelationshiptypes "

        2)Return the query Result.

    }else {  //the keyword probably matches any property value

        1)Get all the properties listed inside a Map collection

        2)Run a query that unwinds these properties one by one and matches
        the node if any of these properties values’ equals the keyword
        value and finally returns the result of the matched value as
        “NodeLabel” and the involved node as info.

        "MATCH (n)" +
        "unwind keys(n) as prop" +
        "MATCH (n) WHERE n[prop]  =~ '(?i).*"+keywordInserted+".*'" +
        "RETURN labels(n) as NodeLabel, n as info"

        3)Return the query Result.

    }
```

In case of two keywords

- JavaScript code separates the keywords and puts them in an array in order to handle them one by one.
- Now the code executes similar functions to the case of one inserted word for each keyword. It actually recognizes and defines the kind of keyword.
- The added functionality needed in this case is the handling of keywords combination after receiving the results and types are known. For this handling there is a specific function called keywordHandling inside the JavaScript code that accepts an array with keyword types and an array with the corresponded values in order to proceed.
  You can get an idea of this function code logic from the following section in which the JavaScript commands have been replaced by pseudocode and comments instead to make the flow more clear.

```
function keywordHandling(arrayTypes,arrayValues){
    console.log("inside the handling: ", arrayTypes, arrayValues);
    if( if the combined keywords are : node label & property name )  ){
                                                    //ex. Book price

              1) assign the node label value to the node variable

              2) assign the property name value to the prop variable

              3) make the corresponded request call passing the node and
                 prop values as path variables

                $.ajax({
                      type: 'GET',
                      url:
'http://localhost:8080/property/propertyOfnode/'+ node +'/' +prop,
                      dataType : "json",
                      contentType:"application/json",
                      success: function(data){


                         […]
                      }
                  });
    }else if(if the combined keywords are : node label & relationship
type){                                                  //ex. Book WRITED

               Similar steps to the first "if" case

    }else if(if the combined keywords are : node label & node label){
                                                      //ex. Book Author
               Similar steps to the first "if" case
```

```
    }else if(if the combined keywords are : node label & “nodes”){
                                                        //ex. Book nodes
              1) assign the node label value to the node variable

              2) call the corresponded function (to apply the request)
                 passing the node as parameter


    }else if(if the combined keywords are : node label & “properties”){
                                                      //ex. Book properties
              1) assign the node label value to the node variable

              2) call the corresponded function (to apply the request)
                 passing the node as parameter


    }else if(if the combined keywords are : node label &
“relationships”){){                                 //ex. Book relationships

               1) assign the node label value to the node variable

              2) call the corresponded function (to apply the request)
                 passing the node as parameter

    }
//      >>>>>>>>>>>>> CONTINUE WITH ELSE IF FOR MORE CASES
```

After filtering keywords with “if conditions” the program recognizes in which category they belong and keeps the keyword. In case of search element the category is not defined in the front end of the application. A function needs to be executed in the back end in order to return the category as string instead.

The category types remain the same with one more added:

- Shownodes – if the keyword is a node label
- Showrelationshiptypes – if the keyword is a relationship type
- Showproperties – if the keyword is a property name

and the category that we did not see as an option in the drop down list :

- NodeLabel – it represents the keyword with any property value.

3.5.3. Responses handling

Now that the request formation is described it is the time to be explained the way the results are being handled in order to be functional and help the user to explore the dataset through interaction.

Due to the fact that all the information is being dynamically retrieved according to the user options, the functionality that is invoked by the request responses need to obtain characteristics once the request had being successfully responded. Practically, that means that any data retrieved is immediately connected to a specific functionality by obtaining the ideal class which already has been implemented to satisfy the core functionality. This class is delegated based on the category recognition we mentioned in the previous section.

The response for both search input element and drop down list triggers the function that is responsible for sending the request according to the user option or keyword typed as was described earlier. The corresponding request is formed according to the defined category which can be: shownodes, showrelationshiptypes or showproperties and in case of search input NodeLabel, as well.

Let us see for each of those categories' calls the information returned:

- Shownodes returns the dataset node labels
- Showrelationshiptypes returns the dataset relationship types
- Showproperties returns the dataset properties

The "success" part of the AJAX call includes the function that handles the successfully retrieved data, this function is implemented as below:

```
success: function(data){
    console.log(data);
            for (let nodeTitle in data) {
               $("<button>" + data[nodeTitle] + " </button>"
).addClass(option).appendTo( $("#result"));
            }
}
```

As you can see above, each data retrieved, which in this case is:

- each **node** in case of “shownodes” case
- each **relationship** type in case of “showrelationshiptypes” case
- each **property** in case of “showproperties” case

is wrapped into a new (button) element with text information populated by the current data retrieved. For example, if the data parameter refers to nodes the query returns all the node labels so the button texts will get their names from each node label. Next, the element is assigned to a class in a dynamic way as well, while the addClass() method parameter is a variable instead of a specific value. The potential main classes to be set on the retrieved data depend on the category type once again. Hence, the possibilities of class assignments are:

1. addClass(shownodes)
2. addClass(showrelationshiptypes)
3. addClass(showproperties)

Consequently, when the user selects any drop down list option or type keywords, the program keeps this value to use it dynamically for both request call and response handling when it comes to the class that needs to be set in the front end.

After the class assignment every button element behaves according to the set class’s functions triggered by user event such as click. Below you can see each button class “on click” function with comments and steps for a better understanding of JavaScript and JQuery code.

1. ***“Shownodes” class on click function:***

```
function(){
  1) assign the clicked button’s text to a node variable
       var node = $( this ).text();

  2)store currentNode to the localStorage
      localStorage.setItem("currentNode", node);

                  […]

  3)create 3 new buttons dynamically based on the node variable value

      $("<button>" + node + " node properties</button>"
).addClass("nodeProperties").attr("value", node).appendTo( $("#result2"));
```

```
 $("<button>" + node + " node relationships</button>"
).addClass("nodeRelationships").attr("value", node).appendTo(
$("#result2"));

 $("<button>check all " + node +" nodes</button>"
).addClass("allNodes").attr("value", node).appendTo( $("#result2"));

4)call the function countNode to count and present the number of node
instances
 countNode(node);
  }
```

As you can notice in the code above three new buttons are created using the node variable to form their displayed texts. In this case the classes set to these new buttons are specific:

- nodeProperties class
- nodeRelationships class
- allNodes class

Below you can check the common steps followed for the three said classes when on click function is triggered:

```
function(){
    […]

1) assign the clicked button's text to a variable
         node = $( this ).val();
 //the variable value could be any node label

2) call the responsible function parsing the node variable
         callTheResponsibleFunction(node);
```

```
// callTheResponsibleFunction function implementation

function callTheResponsibleFunction (node) {

     $.ajax({
          type: 'GET',
3) form the request by using the parsed node value as a path variable
          url: 'http://localhost:8080/node/---/' + node,

                […]
4) the corresponding query runs at the back-end function that is being
executed to respond to the request that is sent

          success: function(data){

                […]


                  $.each( data, function( key, val ) {


                5) set the corresponding class on every data retrieved so
```

```
                that the button to obtain this class' functionality and
                form the displayed text according to the returned value

                 });

6) append the elements in order to be displayed
            }
      });
}
//end of callTheResponsibleFunction function implementation


   }
```

In case of allNodes class there is some more functionality to be referenced.

AllNodes class' on click function creates a table in which every row represents a node instance and adds two new button element at the end of each row, dynamically. When the new buttons are created they store as attribute the id of the same row node. In this way their entire functionality is based on the specific instance. Particularly, the buttons are:

- Check relationship button which returns another dynamically created table with two similar buttons for each row and
- Visualize relationship button which draws the graph related to the current node's relationships using neovis.js library.

The user is able to 'jump' from table to table by pressing the check relationship buttons sequentially.

Check relationship button on click function implementation

```
function(){

 1) Get the current (row) node id via button's attribute
       var nodeId = $(this).attr("vid");

 2)Get the current node stored
       var node = localStorage.getItem("currentNode");

 3)Form the query that involves all the relationships of current node
var cypher = "MATCH p=( a:" + node+" )-[r]-(b) where ID(a)=" +nodeId+"
RETURN p"

             var config = {

                [...]
```

```
                labels: {

                },
                relationships:{

                },
4) parse the formed query to draw the graph
                initial_cypher: cypher
              }

              […]

    }
```

Visualize relationship button on click function implementation

```
function(){

                […]

1) Get the current (row) node id via button's attribute
        var nodeId = $(this).attr("id");

2)Get the current node stored
        var node = localStorage.getItem("currentNode");
         $.ajax({
              type: 'GET',
3) form the request by using the parsed node value and node id as path
variables
             url: 'http://localhost:8080/node/relationships/' + node+ '/'+
nodeId ,

                […]

                $.each( data, function( key, val ) {

4) create a table which stores each relationship retrieved (from the query
that ran in the back end) in a row and adds same way two new element
buttons in the end of each row.

                });

                […]

              }
          });
    }
```

2. ***"showrelationshiptypes" class on click function:***

```
function(){
1) assign the clicked button's text to a rel variable
     var rel = $(this).text()
```

```
        […]

2) create 2 new buttons dynamically based on the rel variable value

      $("<button>" + rel + " properties</button>"
).addClass("relProperties").appendTo( $("#result2"));
      $("<button>" + rel+ " involved nodes</button>"
).addClass("relativeNodes").appendTo( $("#result2"));

3) call the function countNode to count and present the number of
relationships type instances

        countRel(rel);
  }
```

The code above shows that 2 new buttons are created using the rel variable to form their displayed texts. In this case the classes set to these new buttons are specific:

- relProperties class
- relativeNodes class

Below you can check the common steps for the two said classes on click functions:

- ***buttons created by "showrelationshiptypes" classes' on click common function:***

```
function(){
       […]
1) assign the clicked button's text to a variable
      var rel = $( this ).text().replace("properties", "").replace(/[^a-zA-
                                                          Z ]/g, "");
        //the variable value could be any relationship type

2) call the responsible function parsing the node variable
         callTheResponsibleFunction(rel);
```

```
// callTheResponsibleFunction function implementation

function callTheResponsibleFunction (rel) {

      $.ajax({
           type: 'GET',
3) form the request by using the parsed rel value as a path variable
           url: 'http://localhost:8080/relationship/---/' + rel,

                […]
```

```
4) the corresponding query runs at the back-end function that is being
executed to respond to the request that is sent

          success: function(data){

                […]


                  $.each( data, function( key, val ) {


                5) set the corresponding class on every data retrieved so
                that the button to obtain this class' functionality and
                form the displayed text according to the returned value

                 });

6) append the elements in order to be displayed
          }
      });
}
// end of callTheResponsibleFunction function implementation


}
```

3. *"showproperties" class on click function:*

```
function(){

1) keep the clicked button's text as prop variable
        var prop =  $(this).text();

2)store currentProperty to the localStorage
         localStorage.setItem("currentProperty", prop);

3) create 1 new button dynamically based on the prop variable value


    $("<button>" + prop + " property involved nodes</button>"
).addClass("propNodes").appendTo( $("#result2"));

4) create another new button dynamically based on the prop variable value
and relate it to the currentNode if exists or to the



    if(localStorage.getItem("currentNode")!=""){
       $("<button>" + prop+ " property of
"+localStorage.getItem("currentNode")+" node</button>"
).addClass("propOfCurrentNode").appendTo( $("#result2"));
    }
}
```

The code just presented creates 2 new buttons using the prop variable to form their displayed texts. In this case the classes set to these new buttons are specific:

- propNodes class
- propOfCurrentNode class

- ***buttons created by "showproperties" classes' on click common function:***

```
function(){
       […]
 1) assign the clicked button's text to a variable
  prop = $( this ).text().replace("property involved nodes",
"").replace(/[^a-zA-Z ]/g, "");

            $.ajax({
                 type: 'GET',
2) form the request by using the parsed prop value as a path variable
                 url: 'http://localhost:8080/property/---/' + prop,
                […]

3) the corresponding query runs at the back-end function that is being
executed to respond to the request that is sent

          success: function(data){

                […]


                  $.each( data, function( key, val ) {


                4) set the corresponding class on every data retrieved so
                that the button to obtain this class' functionality and
                form the displayed text according to the returned value
                  });

5) append the elements in order to be displayed
          }
      });
}
```

3.5.3.1 Types of results

This chapter describes the application functionality from the side of User experience. All the functionality flow that were described in the previous is triggered by the user interface elements. There are two types of results returned categorized by the capability of interacting with them .

When the results regard to specific values of the dataset they have an information form and display the data. The user can learn the desired information but cannot interact with the result itself.

On the contrary, when the results include more general and category like information (node labels, relationship types and properties) the user can keep interacting with the graph data in order to retrieve more information about it. In this chapter the user possibilities are presented and explained step by step.

3.5.3.2 Interacting with node buttons

After selecting to interact with the node labels, each result button pressed offers the user three options through three buttons to check the node properties, relationships or instances, as well as the information about the existed instances of the node, like in the picture:

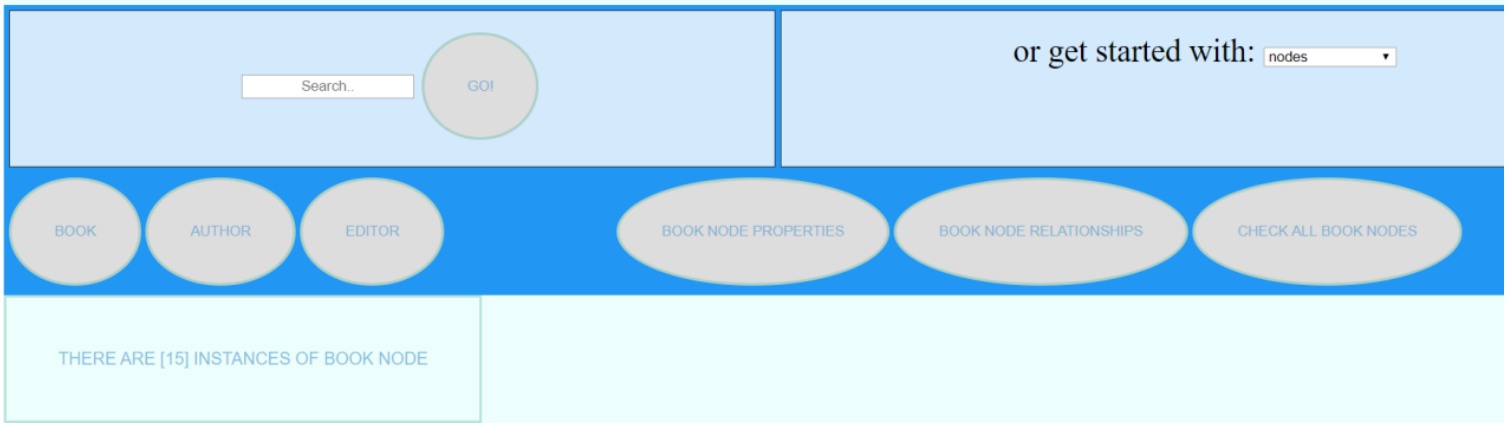

Image 16: Displayed result buttons after interacting with node result buttons

The user is able to proceed the way he or she prefers. In case of " _ node properties" and "_ node relationships" the application will show the requested information in interactive buttons to let the user choose next steps again.

If the "check all _ nodes" button be pressed, the result will be a table with all the instances of the node displayed. In addition, two more buttons will be created for each table row that represents an instance. Those buttons are the "Check Relationship" button and the "Visualize" button. You can have an image of the above description due to the following picture.

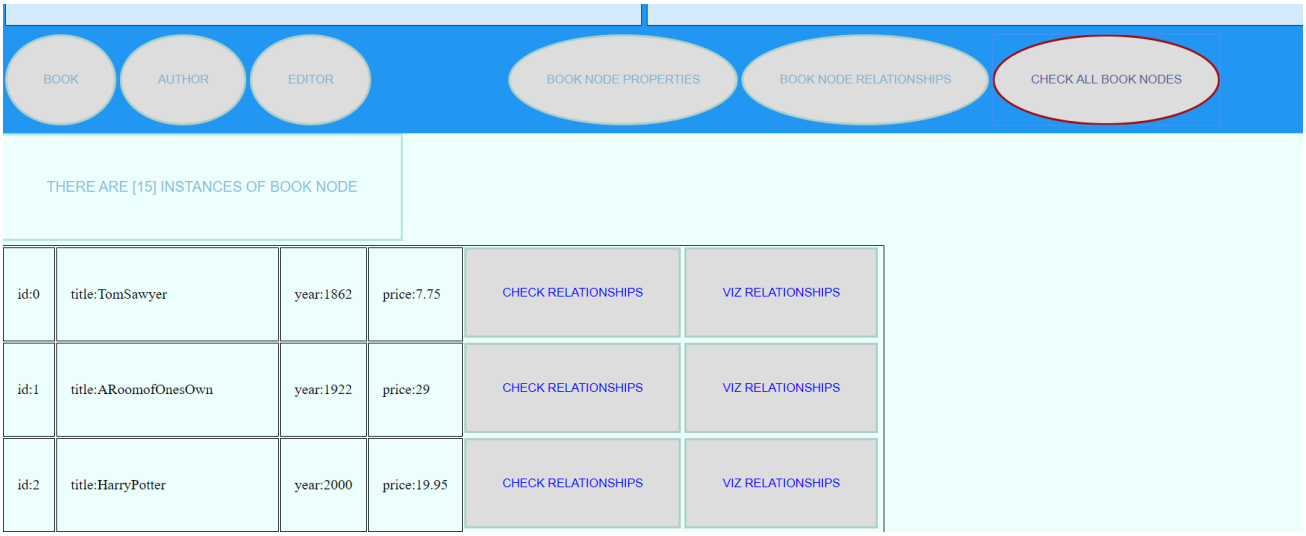

Image 17: Displayed result table after clicking the "check all nodes" button

3.5.3.3 Interacting with the table buttons

Every pair of the "check relationships" and "viz relationships" buttons represents each node instance.

The first button leads to another table that includes all the information about nodes that have any type of connection with the selected row node. An example of the "check relationships" result table of the "Tom Sawyer" node involved is shown below:

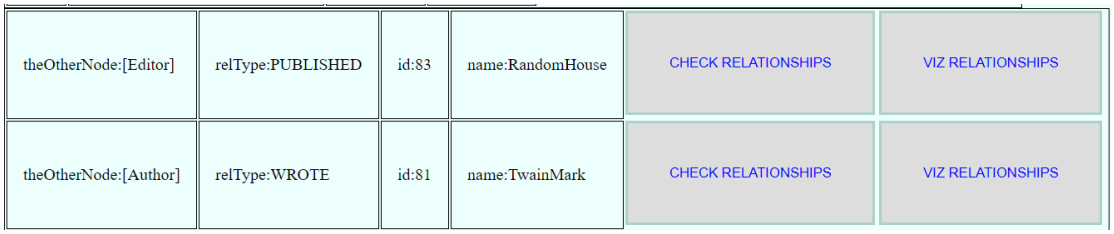

Image 18: Displayed result table after clicking the "check relationships" row button

The same pair of buttons keeps being created for every row and lets the user keep interacting sequentially.

The second button visualizes the involved relationships of the selected node row. The next graph picture was drawn after clicking the “Tom Sawyer” node‘s “visualize relationships” button.

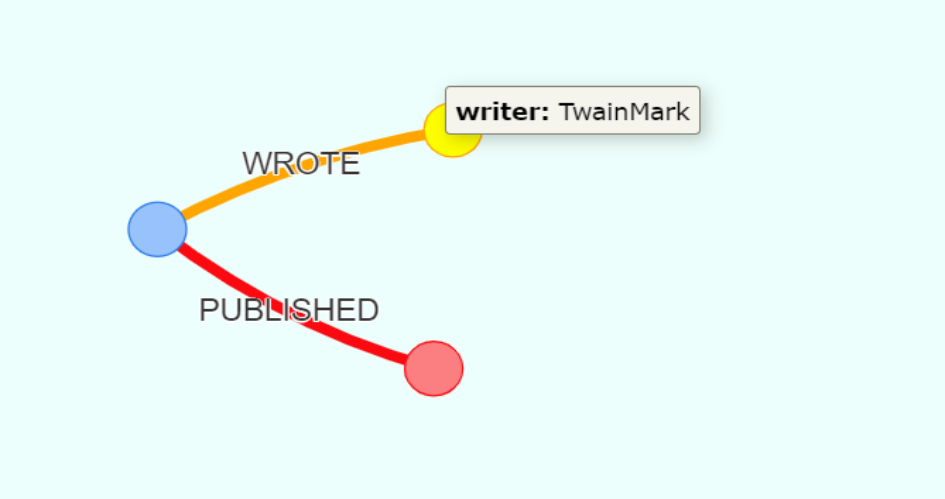

Image 19: Displayed result table after clicking the “check relationships” row button

3.5.5.4 Interacting with property buttons

Selecting any property button consequences to a button “[*property*] property involved nodes” display that when pressed shows the node labels that are related to the property the user chose. Then, the user can pick the node he or she wants in order to proceed and interact with the nodes buttons. The button as displayed in the screen are the following:

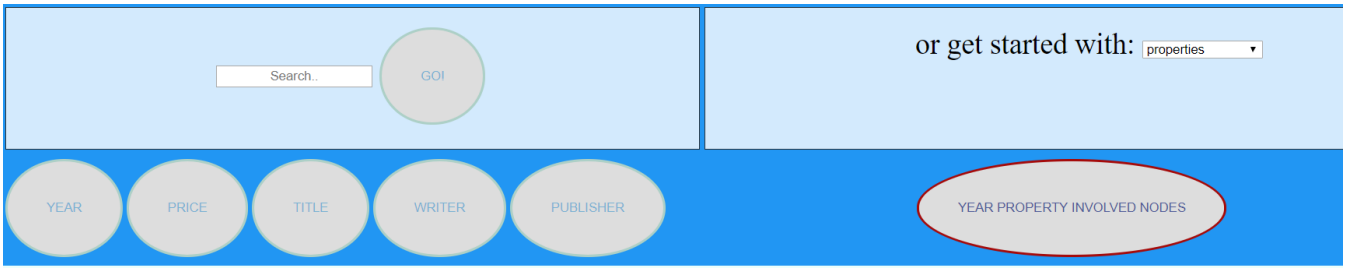

Image 20: Displayed result buttons after interacting with property result button

3.5.5.5 Interacting with relationship type buttons

When a relationship type button is pressed the user has two options available through the buttons created as in the picture below, and also the number relationship instances as information on bottom:

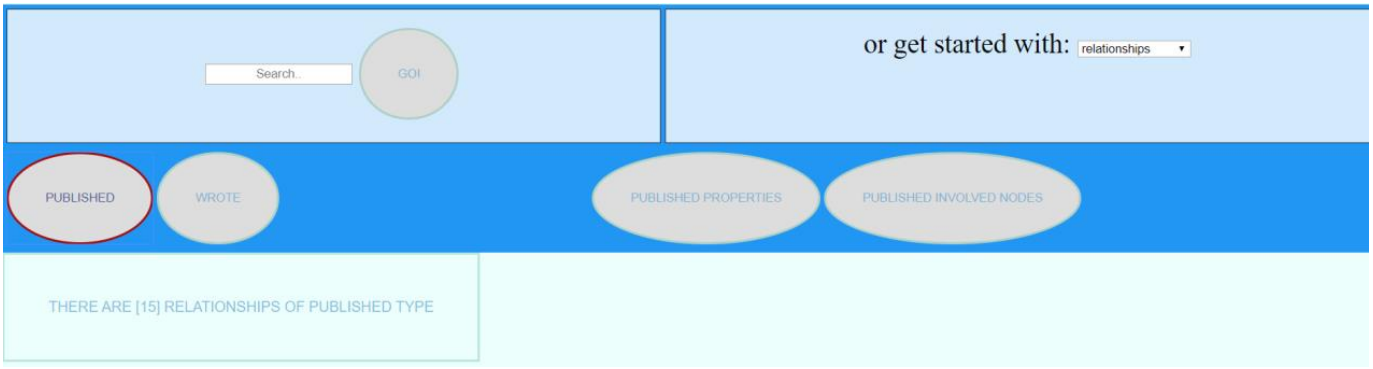

Image 21: Displayed result buttons after interacting with relationship result button

The "[relationship] properties" button shows the relationship properties in interactive buttons in the described regular way . The "[relationship] involved nodes" button displays the relationship using symbols with both the involved nodes and the relationship type as buttons to interact with. Next picture helps to the said description better understanding:

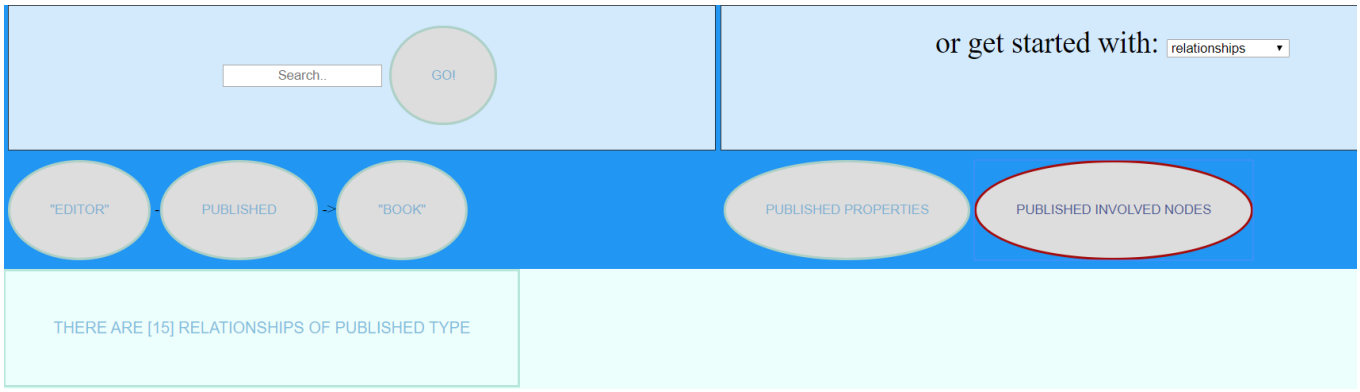

Image 22: Displayed result table after clicking the "involved nodes" button

3.5.5.6 The general navigation

The following diagram describes the flow of interaction through user interface elements and user's options. Starting from the initial screen, the user has two different ways to begin exploring as it was analyzed in the previous chapters. Each element gives user the chance to proceed the way he or she wants. The main displayed frame describes the available interaction sources while the white field description gives the details about user's choices.

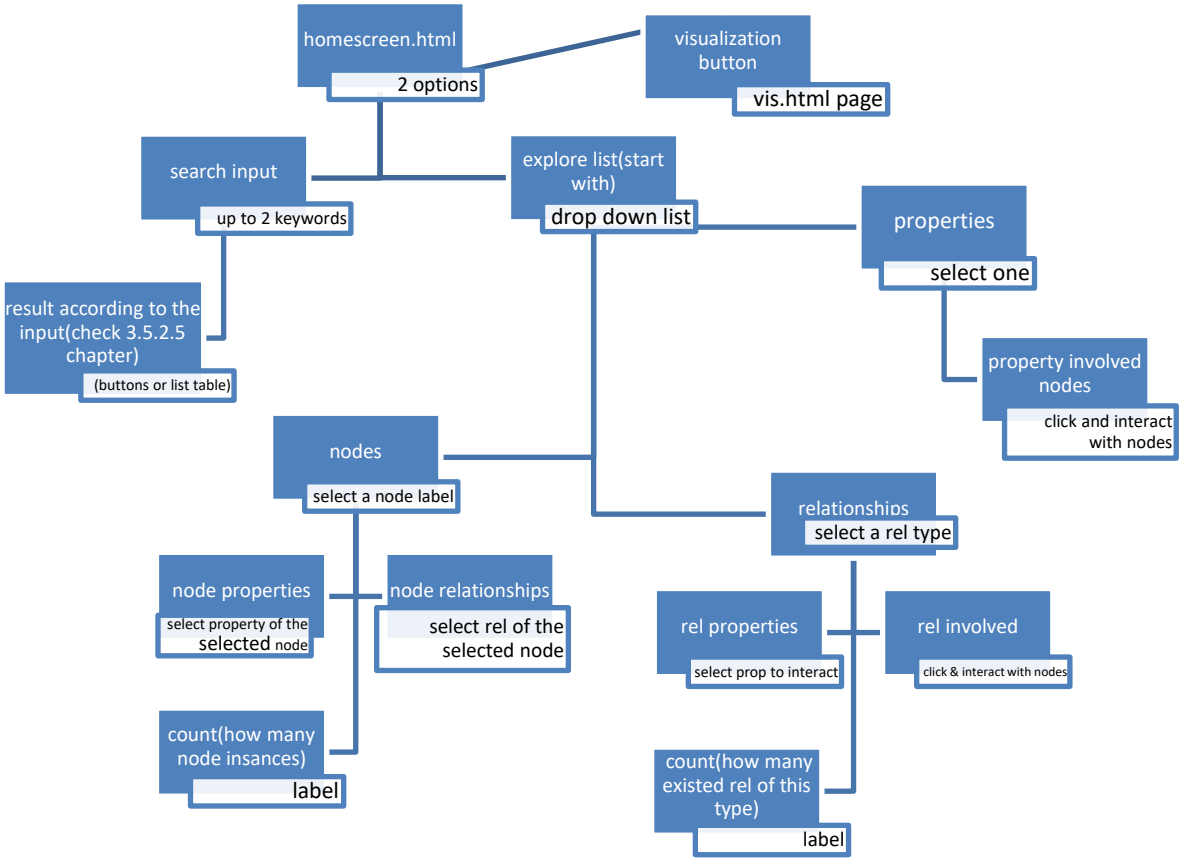

Diagram 5: User interface interaction flow

3.5.4 Developer guide

In order to use this project source code as developer for implementing another neo4j database sample we need to take into consideration some necessities and restrictions.

3.5.4.1 Prerequisites

First of all, we have to transfer neo4j schema could Java objects. After figuring out the final domain model, the next Step is to create all the necessary classes that will be used as node entities and their respective controllers and repositories. All the node entities should extend the EntityNode existed class which ensures the correct id usage. Inside the node entity annotated classes we define the object properties. The object properties need to have same name as the import variables so that they will be mapped correctly during the import phase. It is preferred not to use any existed id. If it is inevitable the class should not extend the EntityNode and the id should be Long type.

Last but most important is to implement very detailed query that implies all the needed information inside their respective import service.

3.5.4.2 Restrictions

There are some restrictions while implementing another database sample inside the project. There are several reserved words that could cause functionality breakdown. This happens because these words are used as variables in both front and back end. Consequently, the program will crush or provide a wrong result to the user. You can see these words that are involved at the source code and should not be used as variables, entities or values inside the imported data in the following table.

RESERVED WORDS: value, result, nodelabel, details, node, property, relationship

3.5.4.3 Simplicity

The Rest API used only for retrieving data from the database and not for creating new information or updating them. This helps the fact that the imported data is not necessary to be mapped to the java objects in order to be used. The requests for rendering data use the “GET” method which has been implemented combined with the java functions that return “Result” type after executing the queries that also only retrieve database information (except for the import

cases). The following chapter is about a small sample of data to be imported with one by one mapping to POJO's. The entire process is explained step by step.

Testing

The database schema includes Songs, Artists and Categories. You can see the schema in the graph below:

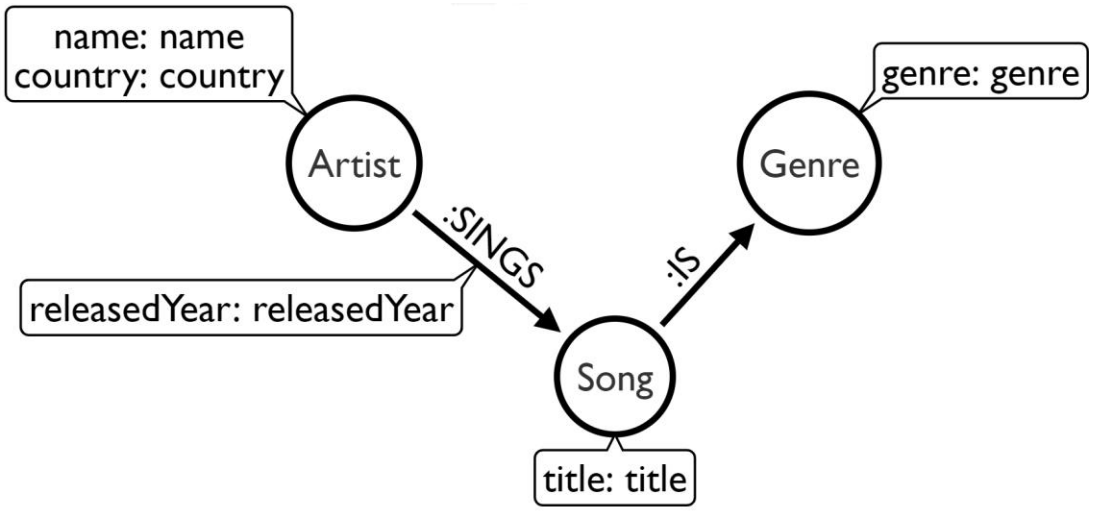

Diagram 6: Schema for testing dataset

The first step is to fill in the import neo4j query to the corresponding function. The function is inside the generic NodeRepository interface which extends the Neo4jRepository. The Collection stores EntityNode objects so the imports entities obtain a unique id due to this class' functionality.

```
public interface NodeRepository<T> extends Neo4jRepository<T, Long> {

[…]

 @Query("LOAD CSV WITH HEADERS FROM 'file:///songs.csv' AS Line\n" +
        "merge (s:Song{title:Line.title})\n" +
        "merge (a:Artist{name:Line.artist})\n" +
        "merge (g:Genre{category:Line.genre})\n" +
        "create (a)-[:SINGS{ releasedYear: Line.releasedYear }]->(s)-[:IS]-
>(g)" )
 Collection<EntityNode> importSchema();

[…]


}
```

Next, we need to create the POJOs that will be mapped to the entities to be imported via OGM. We can adapt relationship types with @Relationship annotation and add or change attributes names. For simplicity reasons in this example the only added functionality after importing the data is in regard to the id as mentioned earlier.

```
@NodeEntity
public class Song extends EntityNode {

    @Property("title")
    private String title;
    public Song() {
    }
    public String getTitle() {
        return title;
    }
}
```

```
@NodeEntity
public class Artist extends EntityNode {

    @Property("name")
    private String name;
    public Artist(){
    }
    public String getName() {
        return name;
    }
}
```

```
@NodeEntity
public class Category extends EntityNode {

    @Property("category")
    private String category;
    public Category() {
    }
    public String getCategory() {
        return category;
    }
}
```

After this process we can run the import function in order to make data available for exploration by pressing the “import” button on home screen.

4 Epilogue

4.1 Summary and Conclusion

This project implementation combined Neo4J Graph Database with a lot of popular technologies in a simple and direct way. There is plenty of available tools for neo4j integration and usage. OGM and Spring MVC provide enormous flexibility considering to Neo4j Database integration. Neo4j cypher potentials are huge and clear for any kind of data retrieving.

4.2 Future improvements

There is a plenty of future improvements that could make this project better regarding the user interface, the functionality and the code clarity for both developer and user side.

In general, the program is difficult to run with a large volume of data and it slows down importantly when the information to be retrieved is a lot. The data rendering could be enhanced by using software engineering tools, libraries and best practices.

From the side of developer, the limitations should be less in order to make the integration easier and faster. The keyword field can be improved so that it will accept more keywords at the same time and convert the total input into a “query like” result. Also, the input could use auto complete in order to help user with suggested words options and recognize any invalid value.

On the other hand, user should have more descriptions available to understand the entire logic of the program even if he or she is not familiar with graph databases. Furthermore, the results could be interactive in any case. Table results could give more interaction via more buttons. An import form could be placed to give the user the possibility to import datasets the way he or she wants.

Bibliography

[1] J. Webber και I. Robinson, «The Top 5 Use Cases - Unlocking New Possibilities with Connected Data,» neo4j.com, 2017.

[2] Neo4j, «https://neo4j.com/,» [Ηλεκτρονικό]. Available: https://neo4j.com/. [Πρόσβαση 20 06 2020].

[3] J. J. Miller, «Graph Database Applications and Concepts with Neo4j,» Atlanta US, 2013.

[4] M. Rodriguez και P. Neubauer, «“Constructions from Dots and Lines” Bulletin of the American Society for,» *American Society for Information Science and Technology,,* τόμ. 36, pp. 35-41, 2010.

[5] S. org, «https://docs.spring.io,» [Ηλεκτρονικό]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html. [Πρόσβαση 2 6 2020].

Appendix

Github link for the project code:

https://github.com/Penteridou/neo4jexploration

The end