Exploring Graph Database based app using a dynamic model

# Introduction

Creating an application often leads to the following data workflow:

Serialized data 🡪 memory data 🡪 view data🡪 memory data 🡪 serialized data

The application deserializes data then optionally process it in memory then display it in a view.

A user can modify a data through a view, which is optionally processed in memory and then stored

Because the serialized object, the memory objects, and view objects are different entities, the code will have some code like this:

myMemoryObject.memoryPropertyName = mySerializedObject.serializedPropertyName;

myViewObject.viewPropertyName = myMemoryObject.memoryPropertyName;

…

Which means that the synchronization between the different entities are based on a static key value system.

Unfortunately, during the development the system will change a lot for different reasons among:

* The serialization system is changing
* The view components are changed or upgraded
* The structure of the objects has changed, to handle new features, or improvements

When a change occurs, the developer must update the bindings in the code using some find and replace operations and recompile the application.

The current experimentation is trying to leverage on Graph database to have a dynamic binding system.

The system will also put the primitive values at the center of relationships showing how to automatically build subset of interesting queries.

The document will explore a the creation of a simple [Angular](https://angular.io/) web app, displaying data from a [Neo4j](https://neo4j.com/) Graph database.

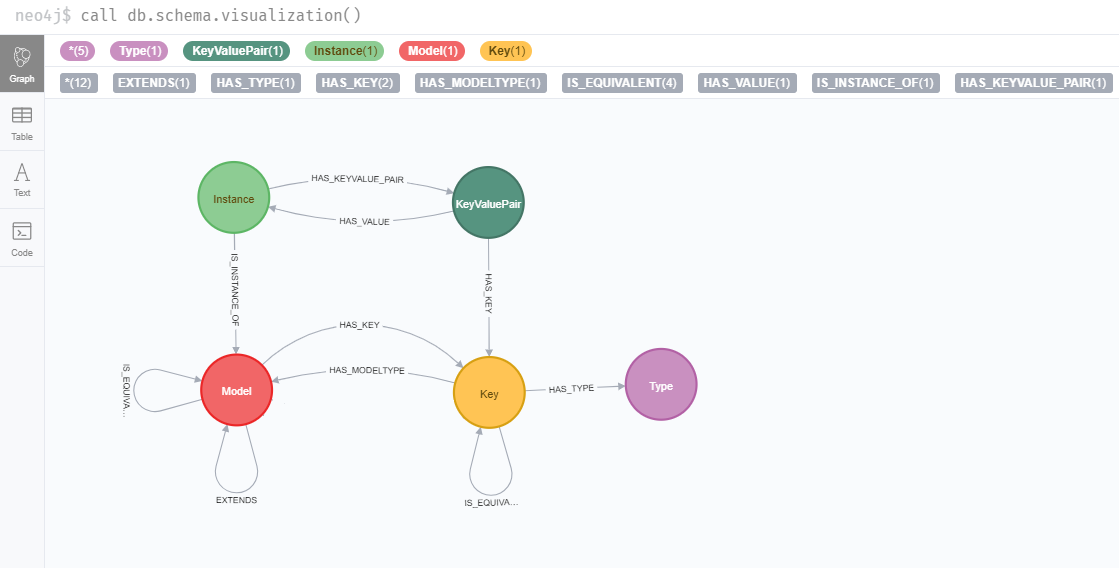
This document is targeting Graph database users or application developers. The sample provided in the documents are very simple and is more focused on the concept that the technology. We will describe the relevant piece of code so that developers can follow the development logic, step by step but this document is not really a tutorial.

***Note:*** I’m very new to Graph database, and I tried through these experiments to understand the reasons of starting to use Graph database in production applications.

# Database dynamic model

Before to begin, we need to define the way we will store data into the database. We decided to consider an abstract model representing data as it is defined from a developer perspective.

Here’s the schema of our graph database:



It contains 5 labels:

* Model (properties: name)
* Key (properties: name, isArray)
* Instance (properties: id)
* KeyValuePair
* Type (properties: name)

And 8 relationships:

* EXTENDS
* HAS\_TYPE
* HAS\_KEY
* HAS\_MODELTYPE
* IS\_EQUIVALENT
* HAS\_VALUE
* IS\_INSTANCE\_OF
* HAS\_KEYVALUE\_PAIR

This describe the following:

* A Model defines keys
* Instances are related to a Model
* Instances define a collection of KeyValuePair
* KeyValuePair is mapping a specific Key to a specific Instance
* A Key has a Type, and can be related to a Model (if the type is an object)
* A Model can extend other models
* A Model can be equivalent to another Model
* A Key of a model can be equivalent to the key of another Model
* Retrieving Instances of a model should give all the instances of the model but also the instances of the extending models and the instances of the equivalent models
* The Instances of a model should expose the keys of the model but also the key of extended models. It should use keys equivalency relationship to provide the appropriate key identifier

The database model is then expressed in typescript as following:

export class DBModel{

    name: string; // Name of the model

    extends?: string[]; // Name of the models extended by the model

    isEquivalentTo?: string[]; // Name of the models that are equivalent to the model

    keys: DBKey[]; // Keys of the model

    instances?: DBInstance[]; // Instances of the model

}

export class DBKey{

    name: string; // Name of the key

    type: string; // Type of the key (should be on one of 'string', 'number', or 'object')

    isEquivalentTo?: string; // Name of a equivalent key

    isArray?: boolean; // Stores the informations that defines if the key is a collection or a single object

    model?: string; // The name of the Model when the key represents instance of another Model

}

export class DBKeyValuePair{

    key: string; // The key name

    values: any[]; // The values which should be number or string representing primitives or instance id

}

export class DBInstance{

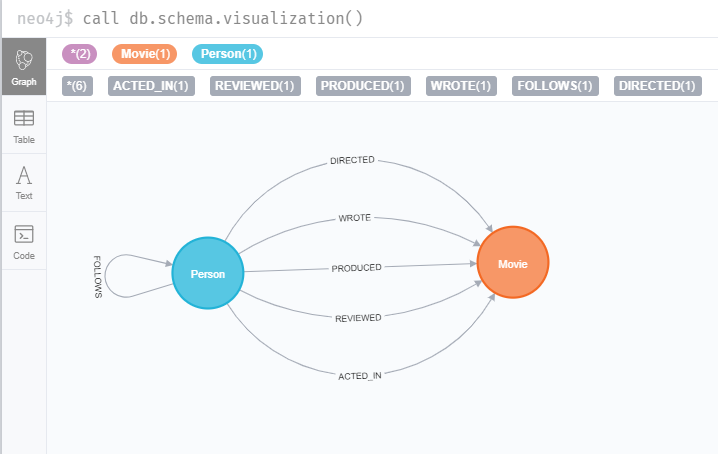
    id: string; // The id of the instance

    properties: DBKeyValuePair[]; // The properties of the instance as a collection of keyvaluepair

}

It is considered, at least in this document, as a dynamic model because the labels of the nodes are representing abstractions and the same model can be used whatever the objects it will store.

In a similar approach, a static model is referring to a model with nodes and relationships based on the nature of stored objects. As an example, here is the schema of the Movie sample database in neo4j:

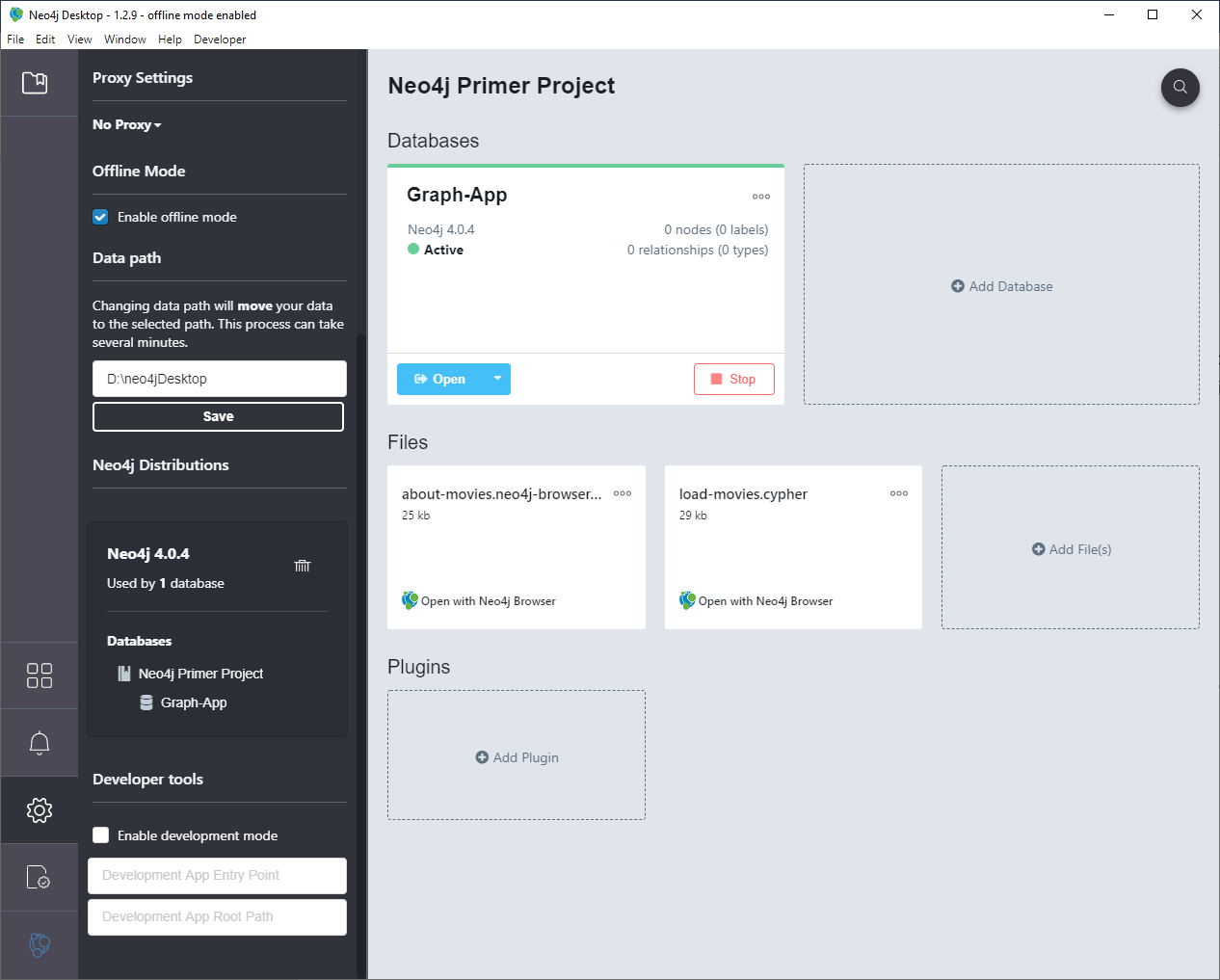


The database is storing ‘Person’ and ‘Movie’ objects which have relationships based on the relationships existing between persons and movies. If we changed some concept in our structure, we will probably have to add, remove, or modify the nodes or the relationships, to fit with the new logic.

In a dynamic model, the structure shouldn’t change and still being able to represent a lot of different concepts.

# Database Interaction

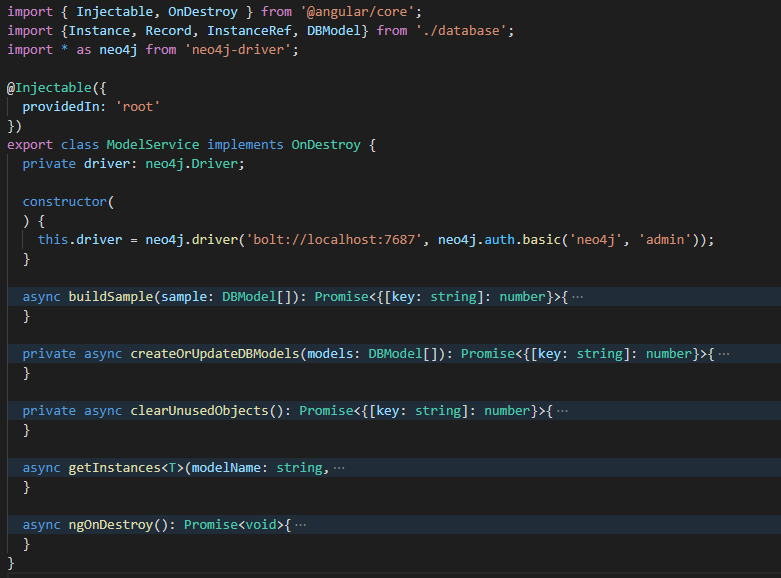
We first, used neo4j Desktop to create an empty database that runs locally and is accessible using bolt protocol:



**IMPORTANT**: To be able to perform a single query, I had to change the following setting in the database (through the neo4j.conf file) and set it to true:

cypher.lenient\_create\_relationship=true

The application presented in the sample is an Angular application that will communicates directly with the graph database through a service named ModelService:



The service is able to:

* Update the database according to a collection of DBModel objects (defined above)
* Retrieve instances of a specific model with optional filtering by id option as a collection of Instance<T> objects where T will be a javascript object containing the key properties of the model

An Instance<T> is defined as following:

export class Instance<T>{

    instance: InstanceRef; // Instance reference

    value: T; // Instance client object

}

export class InstanceRef{

    id: string; // Database instance id

}

Where:

* ‘instance’ property represents the instance database meta data (in the sample we will only use the id, but we can imagine adding other information like a creationDate, lastUpdate, etc…)
* ‘value’ property is a javascript object that will be used by the application

The ModelService class has 2 public methods:

* buildSample: This will update the Database model using the private methods createOrUpdateDBModels and clearUnusedObjects providing a sample that declare the data structure and the instances contained in the database
* getInstances will take a model name and will retrieve the instances for that model

It has also 2 private methods:

* createOrUpdateDBModels will be used to update the data structure of our project using a DBModel collection as an input
* clearUnusedObjects will removed object that have are no more used in the application

The queries were defined once and won’t be updated during the experimentation. They will be detailed at the end of the document.

NOTE: In a real-life project, the client shouldn’t off course communicate directly with the database and the CRUD operations should be handled in a client/server architecture.

## Building the sample

import { Component } from '@angular/core';

import { ModelService } from './model.service';

@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'graph-based-app';

  constructor(private modelService: ModelService){

    this.modelService.buildSample();

  }

}

The application will run the buildSample method once, when launching.

The purpose of the method is to adapt the database using a DBModel collection (referred as sample) representing the Models with their instances. The changes made to the database will be differential: it means that the method will compare the existing database with the sample and create or remove the nodes and relationships to represent exactly the changes between the current version of the database and the sample. In a real project we wouldn’t use such a method, but it was created to check that a single generic query was able to modify dynamically the database.

The buildSample code is detailed below:

  async buildSample(): Promise<{[key: string]: number}>{

    const r1 = await this.createOrUpdateDBModels(sample);

    const r2 = await this.clearUnusedObjects();

    const r = {};

    let hasUpdates = false;

    for (const key in r1) {

      if (Object.prototype.hasOwnProperty.call(r1, key)){

        let val = r1[key];

        if (Object.prototype.hasOwnProperty.call(r2, key)) {

          val += r2[key];

        }

        hasUpdates = hasUpdates || val > 0;

        if (val > 0){

          r[key] = val;

        }

      }

    }

    console.log({changes: hasUpdates ? r : 'No changes'});

    return r;

  }

It calls createOrUpdateDBModels method which will run a cypher query to the database providing the models then it calls clearUnusedObjects that will clean the database.

The changes of the 2 queries are aggregated and logged so we can track what are the changes made to the database.

At the beginning of the project the sample const was defined like this:

import {DBModel} from './database';

export const sample: DBModel[] = [

    ];

This means that de database contains nothing and should remains empty (we started with an empty database) when launching the application.

During the evolution of the project we will modify the sample to declare what we want to find in the database.

The buildsample method will apply that model to the Graph database and log the effective changes in the database.

# Let’s build a simple Movie application

## Displaying the movies

We want to display a list of movies and we created a simple Movie class that represents a movie which is defined by its title.

## Adding Movie model

Let’s first define a Movie model:

export class Movie{

    title: string;

}

We create then a MovieListComponent:

import { Component, OnInit } from '@angular/core';

import {ModelService} from '../model.service';

import {Instance} from '../database';

import {Movie} from '../media';

@Component({

  selector: 'app-movie-list',

  templateUrl: './movie-list.component.html',

  styleUrls: ['./movie-list.component.css']

})

export class MovieListComponent implements OnInit {

  movies: Instance<Movie>[];

  constructor(

    private modelService: ModelService

  ) {

   }

  async ngOnInit(): Promise<void> {

    this.movies = await this.modelService.getInstances('Movie');

  }

}

And its template

<h3>Movies</h3>

<div class="item" \*ngFor="let movie of movies">

    <span>{{movie.instance.id}}</span>

    <span>{{movie.value.title}}</span>

</div>

## Updating the sample

Let’s update the sample like this:

import {DBModel} from './database';

export const sample: DBModel[] = [

      {

        name: 'Movie',

        keys: [{name: 'title', type: 'string'}],

        instances: [

          {id: 'm1', properties: [{key: 'title', values: ['The Matrix']}]},

          {id: 'm2', properties: [{key: 'title', values: ['Harry Potter']}]}

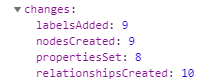
        ]

      }

    ];

This tells the system, that we want a Model named ‘Movie’, containing the key named ‘title’ and having 2 instances ‘m1’ and ‘m2’ configured with ‘The Matrix’ and ‘Harry Potter’ as a title.

When looking into the logs, the first time the application is launched after having updated the sample, we will see :



If we refresh the page or relaunch the application the logs will be:



This is because of the createOrUpdateDBModels method is not incremental but differential (the query will be detailed later).

***Note:*** In the real life we probably never declare DBModel objects manually, but this is convenient for a simple experimentation.

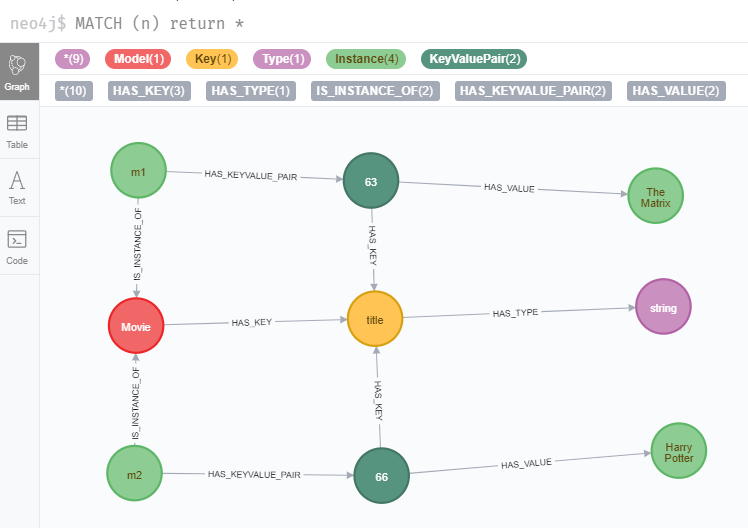
## Watching the changes

When running the application, it will display the movie list:

A picture containing drawing

Description automatically generated

Let’s now have a look at the database:



We can see exactly what we defined in the sample:

* A model, named ‘Movie’ which has a key named ‘title’
* 2 instances ‘m1’ and ‘m2’ having both a key/value pair binding the title to values which are instances having no model meaning that they are primitive
* The key as a type which is a string

Our model is considered dynamic by the fact that ‘Movie’ is just a node labelled as ‘Model’ and ‘title’ a node labelled as ‘Key’. The abstraction model was defined statically but we can work with movies, fruits, or products, the labels and relationships will remain the same, which mean than we can have a generic method based on model name that will dynamically create the desired object according to the specifications defined in the sample.

This way of doing creates more nodes than for a static model, and add complexity to the graph (when displaying the nodes) and also some verbosity to the query (because of the KeyValuePair acting as an hyperedge), but due to the efficiency of Graph database to deal with relationships, it should also have some benefits that we will explore trying to improve the application.

***Important:*** In that model, the primitives are not stored as property of nodes, but as nodes, the id of the node being the primitive. This allows to create relationships between object instances, keys and primitive values. These relationships will create physical subsets of correlations that seems very promising.

# Improving the application

A movie should not only be defined by a title. It has also actors and we want to display the actors of a movie in our app.

## Creating the Person model

We create first a Person class as following:

export class Person{

    firstName: string;

    surname: string;

}

And a PersonComponent:

import { Component, OnInit, Input } from '@angular/core';

import { Person } from '../person';

@Component({

  selector: 'app-person',

  templateUrl: './person.component.html',

  styleUrls: ['./person.component.css']

})

export class PersonComponent implements OnInit {

@Input() person: Person;

age: number;

  constructor() { }

  ngOnInit(): void {

  }

}

With its template:

<span \*ngIf="person">{{person.firstName}} {{person.surname}}</span>

## Improving the Movie model

The Movie model should now expose an ‘actors’ property which is a Person collection:

export class Movie{

    title: string;

    actors: Person[];

}

We can then improve the MovieListComponent template:

<h3>Movies</h3>

<div class="item" \*ngFor="let movie of movies">

    <span>{{movie.instance.id}}</span>

    <span>{{movie.value.title}}</span>

    <span><div \*ngFor = "let actor of movie.value.actors">

        <app-person [person]="actor"></app-person>

    </div></span>

</div>

## Updating the sample

We will update the sample as following:

export const sample: DBModel[] = [

    {

        name: 'Person',

        keys: [{name: 'firstName', type: 'string'}, {name: 'surname', type: 'string'}],

        instances: [

          {id: 'p1', properties: [{key: 'firstName', values: ['Keanu']}, {key: 'surname', values: ['Reeves']}]},

          {id: 'p2', properties: [{key: 'firstName', values: ['Emma']}, {key: 'surname', values: ['Watson']}]},

          {id: 'p3', properties: [{key: 'firstName', values: ['Daniel']}, {key: 'surname', values: ['Radcliffe']}]},

          {id: 'p4', properties: [{key: 'firstName', values: ['Carie-Anne']}, {key: 'surname', values: ['Moss']}]}

        ]

      },

      {

        name: 'Movie',

        keys: [{name: 'title', type: 'string'}, {name: 'actors', isArray: true, type: 'object', model: 'Person'}],

        instances: [

          {id: 'm1', properties: [{key: 'title', values: ['The Matrix']}, {key: 'actors', values: ['p1', 'p4']}]},

          {id: 'm2', properties: [{key: 'title', values: ['Harry Potter']}, {key: 'actors', values: ['p2', 'p3']}]}

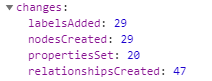
        ]

      }

    ];

We have added the model ‘Person’ and some instances of it and added the ‘actors’ key to the ‘Movie’ model.

And the DB changes log will be:



Because of the separation between models, keys, and instances it is quite easy to update the model differentially, adding keys, and linking the node together.

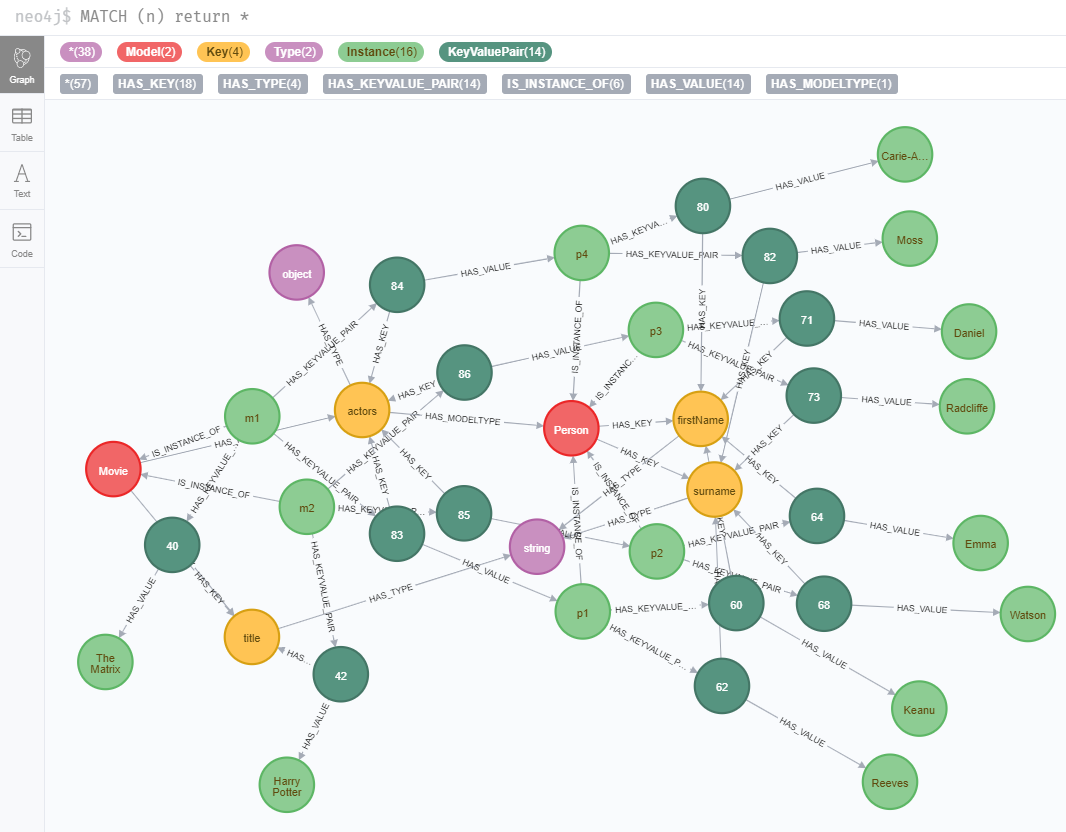
## Watching the changes

The application will now display the following:

A screenshot of a cell phone

Description automatically generated

In the database we have now:



On the right are all the primitives representing the ‘firstName’ and ‘surname’ which are related to the instances of Person. The movies have now new key/value pairs that stores the actors collection.

One relationship was added: HAS\_MODELTYPE which tells to the system that actors are Person

Note that we didn’t have to change the ModelService queries or the MovieListComponent. The changes in the sample has updated the database which has now all the information to return the updated Movie data structure.

What will happen now, if we change the data structure of the sample?

# Changing the data structure

In our application, having the actors of a movie is nice but we should have also the character played in each movie. We need to update our model to reflect that need, creating a ‘Role’ model containing the ‘actor’ and the ‘character’ properties, and changing the ‘actors’ properties to ‘roles’ in the ‘Movie’ model.

## Adding Role model

This is the Role class definition:

export class Role{

    character: string;

    actor: Person;

}

And this is a RoleListComponent:

import { Component, OnInit, Input } from '@angular/core';

import { ModelService } from '../model.service';

import { Movie } from '../media';

import {Role} from '../role';

@Component({

  selector: 'app-role-list',

  templateUrl: './role-list.component.html',

  styleUrls: ['./role-list.component.css']

})

export class RoleListComponent implements OnInit {

  constructor(private modelService: ModelService) { }

  @Input() movie: Movie;

  roles: Role[];

  async ngOnInit(): Promise<void> {

    this.roles = this.movie ? this.movie.roles : [];

  }

}

With its template:

<div \*ngFor="let role of roles">

        <span \*ngIf = "role.actor"><app-person [person]="role.actor"></app-person><span \*ngIf = "role.character"> as "{{role.character}}"</span></span>

</div>

## Updating Movie model

The Movie class is now like this:

export class Movie{

    title: string;

    roles: Role[];

}

And the MovieListComponent template will be:

<h3>Movies</h3>

<div class="item" \*ngFor="let movie of movies">

    <span>{{movie.instance.id}}</span>

    <span>{{movie.value.title}}</span>

    <app-role-list [movie]="movie.value"></app-role-list>

</div>

## Updating the sample

export const sample: DBModel[] = [

    {

        name: 'Person',

        keys: [{name: 'firstName', type: 'string'}, {name: 'surname', type: 'string'}],

        instances: [

          {id: 'p1', properties: [{key: 'firstName', values: ['Keanu']}, {key: 'surname', values: ['Reeves']}]},

          {id: 'p2', properties: [{key: 'firstName', values: ['Emma']}, {key: 'surname', values: ['Watson']}]},

          {id: 'p3', properties: [{key: 'firstName', values: ['Daniel']}, {key: 'surname', values: ['Radcliffe']}]},

          {id: 'p4', properties: [{key: 'firstName', values: ['Carie-Anne']}, {key: 'surname', values: ['Moss']}]}

        ]

      },

      {

        name: 'Role',

        keys: [{name: 'character', type: 'string'}, {name: 'actor', type: 'object', model: 'Person'}],

        instances: [

          {id: 'r1', properties: [{key: 'character', values: ['Neo']}, {key: 'actor', values: ['p1']}]},

          {id: 'r2', properties: [{key: 'character', values: ['Hermione Granger']}, {key: 'actor', values: ['p2']}]},

          {id: 'r3', properties: [{key: 'character', values: ['Harry Potter']}, {key: 'actor', values: ['p3']}]},

          {id: 'r4', properties: [{key: 'character', values: ['Trinity']}, {key: 'actor', values: ['p4']}]},

          {id: 'r5', properties: [{key: 'character', values: ['Kevin Lomax']}, {key: 'actor', values: ['p1']}]}

        ]

      },

      {

        name: 'Movie',

        keys: [{name: 'title', type: 'string'}, {name: 'roles', isArray: true, type: 'object', model: 'Role'}],

        instances: [

          {id: 'm1', properties: [{key: 'title', values: ['The Matrix']}, {key: 'roles', values: ['r1', 'r4']}]},

          {id: 'm2', properties: [{key: 'title', values: ['Harry Potter']}, {key: 'roles', values: ['r2', 'r3']}]},

          {id: 'm3', properties: [{key: 'title', values: ['The Devil"s Advocate']}, {key: 'roles', values: ['r5']}]}

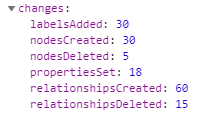
        ]

      }

    ];

The sample reflects the changes in the model. We added the ‘Role’ model and updated the ‘Movie’ model. We have also added a new movie into the database.

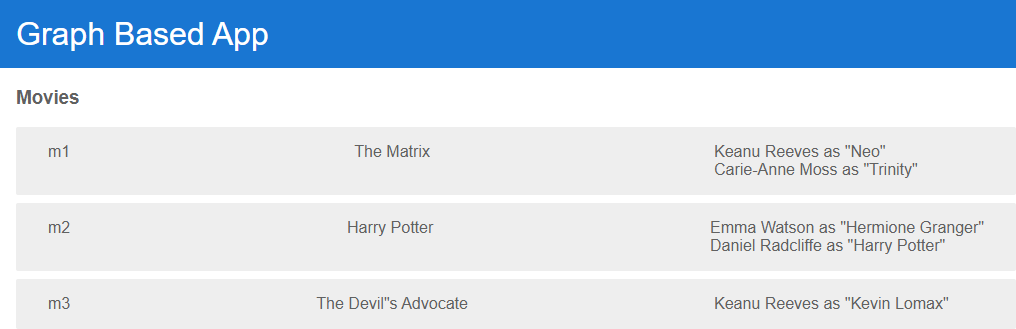
When running the application for the first time after updating the sample the changes in the databases logged by the application are:



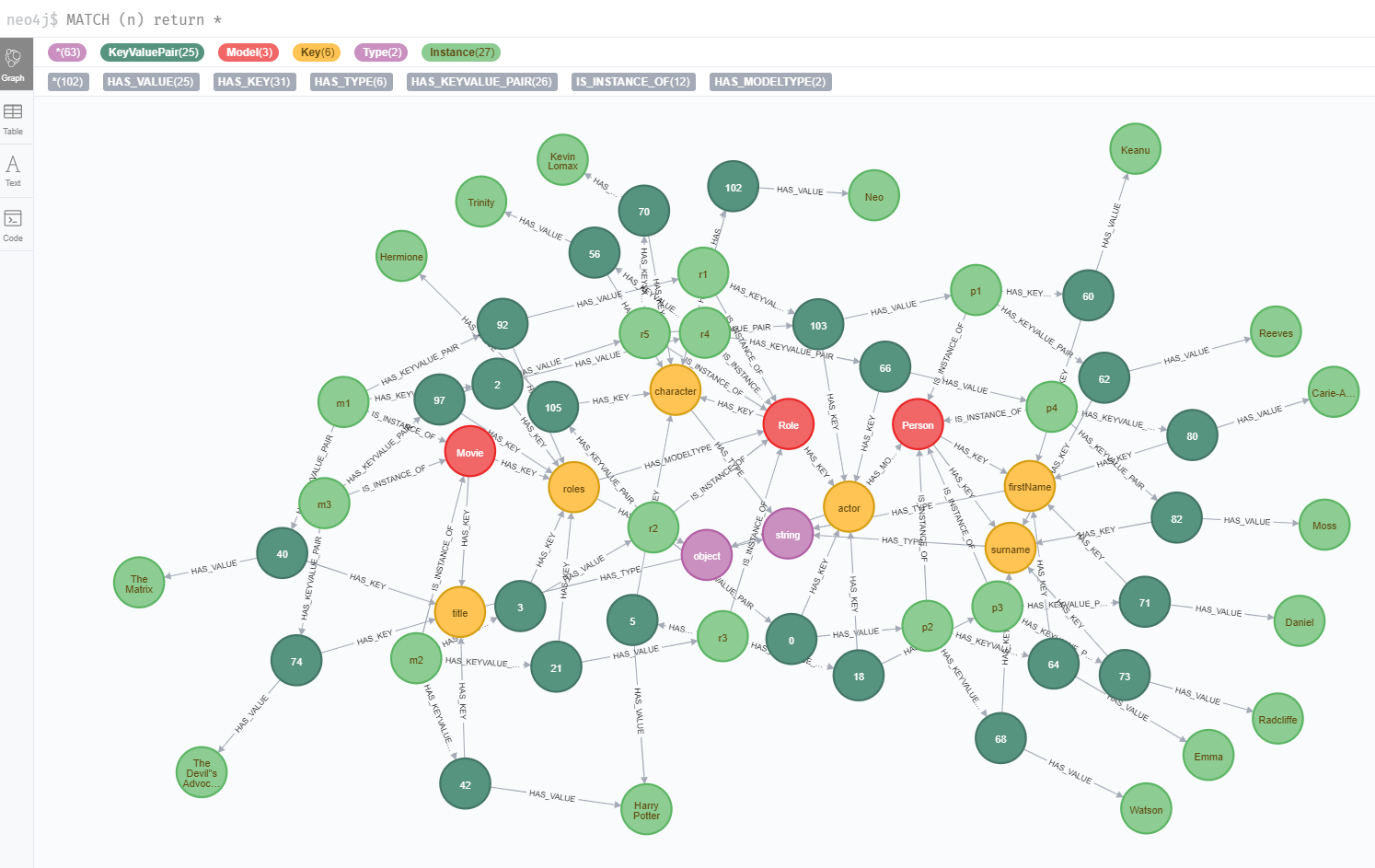
Some nodes and relationships were deleted: because the ‘actors’ property is not used anymore, it was removed with all the key/value pairs related to it.

## Watching the changes

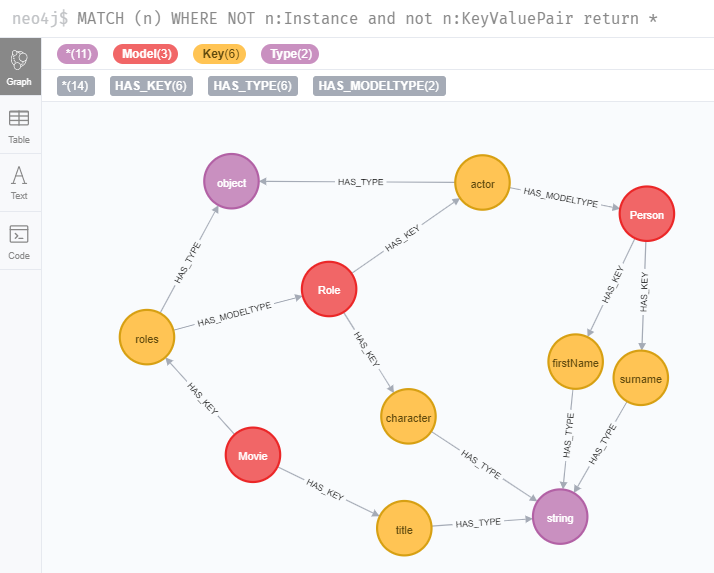
The application will now display:



The database looks now like:



Watching the entire graph beginning to become a little confusing, let’s focus on some nodes of interest:

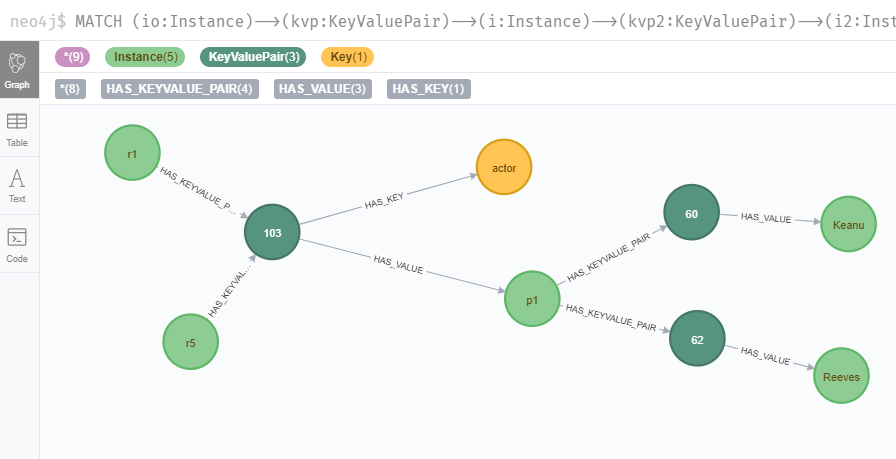


Excluding the instances and the key/value pairs we can focus on the data structure and we can see

That:

* the property ‘actors’ has disappeared from the model ‘Movie’ which has now the property ‘roles’ with ‘Role’ as model type
* The model ‘Role’ was added with 2 keys ‘character’ and ‘actor’ which is a ‘Person’
* The ‘roles’ and ‘actor’ keys have an ‘object’ type
* The other keys are defining primitives of type ‘string

Let’s have a look at Keanu Reeves now, stored by the instance ‘p1’:

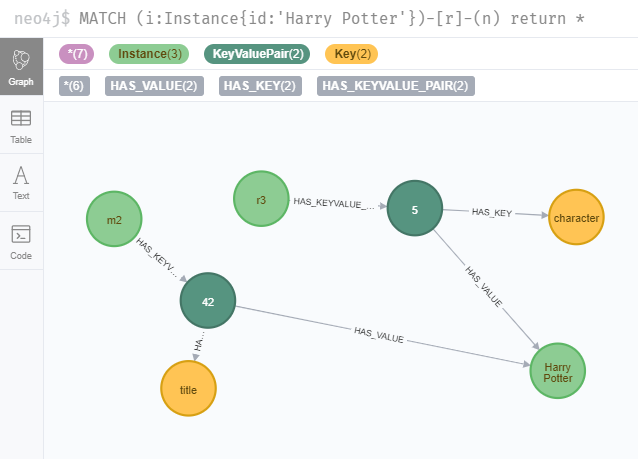


Because Roles ‘r1’ and ‘r5’ have both ‘p1’ as an actor (Keanu Reeves), they are sharing the same key/value pair.

In the system, a KeyValuePair binding a specific key to a specific instance will always be unique.

We can also see that there is a physical connection between ‘p1’ and ‘r1’ and ‘r5’. If a user wants to know which roles are played by Keanu Reeves, we don’t need to iterate through the entire instances of Roles. We just need to search for a KeyValuePair mapping Person ‘p1’ to the key ‘actor’ and all the instances having that KeyValuePair should be the answer to the question.

If we have a look now at ‘Harry Potter’ primitive:



Because ‘r3’ as ‘Harry Potter’ as character and ‘m2’ has ‘Harry Potter’ as title, they have both a key/value pair sharing the same primitive instance.

A primitive instance id being the value of the primitive, primitive Instance with the same value will be unique.

This has a very interesting consequence: it will create relationships (through KeyValuePair) between instances having the same primitive: It means that the system will generates a lot of value-based connections between the instances. This should have a lot of interest when you have to run analytic queries on the database. Starting from a primitive instance node, or a key node you will find all the instances related to that key or value without having to iterate through the entire database.

As an application user, you might be interested by ‘Harry Potter’. When you reach the ‘Harry Potter’ node you can find instances of different models having a key/value pair directly related to it. In our sample, you can learn that the value ‘Harry Potter’ is associated with the key ‘title’ of a model ‘Movie’ and the key ‘character’ of a model ‘Role’. You can also see which instance of ‘Movie’ and ‘Role’ is related to ‘Harry Potter’. The relationship exists physically in the database: you don’t need to iterate through all the objects to find those sharing the ‘Harry Potter’ value. You don’t need neither to know every concept (Key or Model) associated with that value. This will probably be very useful when you don’t know how the users of the application will consume its data.

The Model, Key, KeyValuePair, Instance system allows us to easily modify the entire structure of the data without having to destroy everything and recreate everything. We can destroy some relationships, add new ones, and then easily remove the orphan nodes.

Let’s explore now how we can extend our model.

# Extending the model

We have what we want in our movie list, but there is a new requirement for the application: display movies but also books.

## Adding Media and Book model

We will create a Media class that will store the title. The Book class will have an ‘author’ property which is a Person. The Movie class will be updated to remove the ‘title’ property and both Book and Movie will extend the Media class

import {Role} from './role';

import { Person } from './person';

export class Media{

    title: string;

}

export class Movie extends Media{

    roles: Role[];

}

export class Book extends Media{

    author: Person;

}

This will allow us to create a BookListComponent:

import { Component, OnInit } from '@angular/core';

import { ModelService } from '../model.service';

import { Book} from '../media';

import { Instance} from '../database';

@Component({

  selector: 'app-book-list',

  templateUrl: './book-list.component.html',

  styleUrls: ['./book-list.component.css']

})

export class BookListComponent implements OnInit {

  books: Instance<Book>[];

  constructor(

    private modelService: ModelService

  ) { }

  async ngOnInit(): Promise<void> {

    this.books = await this.modelService.getInstances<Book>('Book');

  }

}

With its template:

<h3>Books</h3>

<div class="item" \*ngFor="let book of books">

    <span>{{book.instance.id}}</span>

    <span>{{book.value.title}}</span>

    <app-person [person]="book.value.author"></app-person>

</div>

And a MediaListComponent :

import { Component, OnInit } from '@angular/core';

import { Instance } from '../database';

import { Book } from '../media';

import { ModelService } from '../model.service';

@Component({

  selector: 'app-media-list',

  templateUrl: './media-list.component.html',

  styleUrls: ['./media-list.component.css']

})

export class MediaListComponent implements OnInit {

  constructor(private modelService: ModelService) { }

  medialist: Instance<Book>[];

  async ngOnInit(): Promise<void> {

    this.medialist = await this.modelService.getInstances('Media');

  }

}

With its template:

<h3>Media</h3>

<div class="item" \*ngFor="let media of medialist">

    <span>{{media.instance.id}}</span>

    <span>{{media.value.title}}</span>

</div>

## Adding navigation system

We will now add some buttons in the TopBarComponent template with different routes:

<a >

    <h1>Graph Based App</h1>

    <button [routerLink]="['/media']">Media</button>

    <button [routerLink]="['/movies']">Movies</button>

    <button [routerLink]="['/books']">Books</button>

</a>

Our app having a routing module describing the following routes:

import { NgModule } from '@angular/core';

import { Routes, RouterModule } from '@angular/router';

import { MovieListComponent } from './movie-list/movie-list.component';

import { MediaListComponent } from './media-list/media-list.component';

import { BookListComponent } from './book-list/book-list.component';

const routes: Routes = [

    {path: '', component: MovieListComponent},

    {path: 'movies', component: MovieListComponent},

    {path: 'media', component: MediaListComponent},

    {path: 'books', component: BookListComponent}

  ];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

## Updating the sample

We will now have to update the sample:

export const sample: DBModel[] = [

    {

        name: 'Person',

        keys: [{name: 'firstName', type: 'string'}, {name: 'surname', type: 'string'}],

        instances: [

          {id: 'p1', properties: [{key: 'firstName', values: ['Keanu']}, {key: 'surname', values: ['Reeves']}]},

          {id: 'p2', properties: [{key: 'firstName', values: ['Emma']}, {key: 'surname', values: ['Watson']}]},

          {id: 'p3', properties: [{key: 'firstName', values: ['Daniel']}, {key: 'surname', values: ['Radcliffe']}]},

          {id: 'p4', properties: [{key: 'firstName', values: ['Carie-Anne']}, {key: 'surname', values: ['Moss']}]},

          {id: 'p5', properties: [{key: 'firstName', values: ['Herman']}, {key: 'surname', values: ['Melville']}]}

        ]

      },

      {

        name: 'Role',

        keys: [{name: 'character', type: 'string'}, {name: 'actor', type: 'object', model: 'Person'}],

        instances: [

          {id: 'r1', properties: [{key: 'character', values: ['Neo']}, {key: 'actor', values: ['p1']}]},

          {id: 'r2', properties: [{key: 'character', values: ['Hermione Granger']}, {key: 'actor', values: ['p2']}]},

          {id: 'r3', properties: [{key: 'character', values: ['Harry Potter']}, {key: 'actor', values: ['p3']}]},

          {id: 'r4', properties: [{key: 'character', values: ['Trinity']}, {key: 'actor', values: ['p4']}]},

          {id: 'r5', properties: [{key: 'character', values: ['Kevin Lomax']}, {key: 'actor', values: ['p1']}]}

        ]

      },

      {

        name: 'Media',

        keys: [{name: 'title', type: 'string'}]

      },

      {

        name: 'Movie',

        extends: ['Media'],

        keys: [{name: 'roles', isArray: true, type: 'object', model: 'Role'}],

        instances: [

          {id: 'm1', properties: [{key: 'title', values: ['The Matrix']}, {key: 'roles', values: ['r1', 'r4']}]},

          {id: 'm2', properties: [{key: 'title', values: ['Harry Potter']}, {key: 'roles', values: ['r2', 'r3']}]},

          {id: 'm3', properties: [{key: 'title', values: ['The Devil"s Advocate']}, {key: 'roles', values: ['r5']}]}

        ]

      },

      {

        name: 'Book',

        extends: ['Media'],

        keys: [{name: 'author', type: 'object', model: 'Person'}],

        instances: [

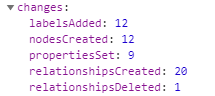
          {id: 'b1', properties: [{key: 'title', values: ['Moby Dick']}, {key: 'author', values: ['p5']}]}

        ]

      }

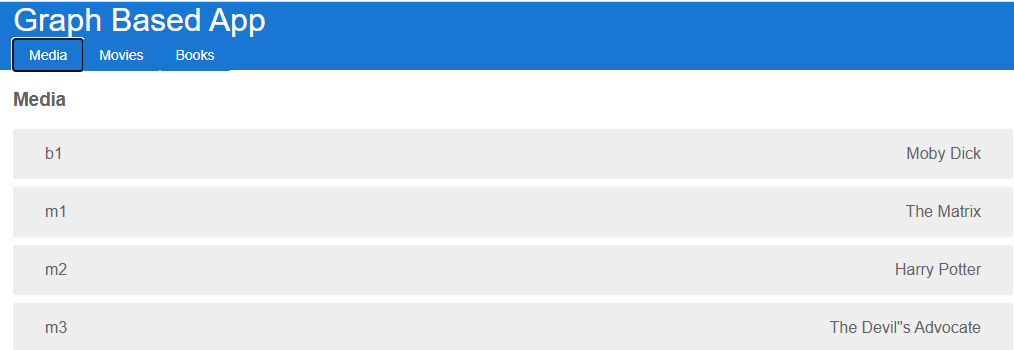
    ];

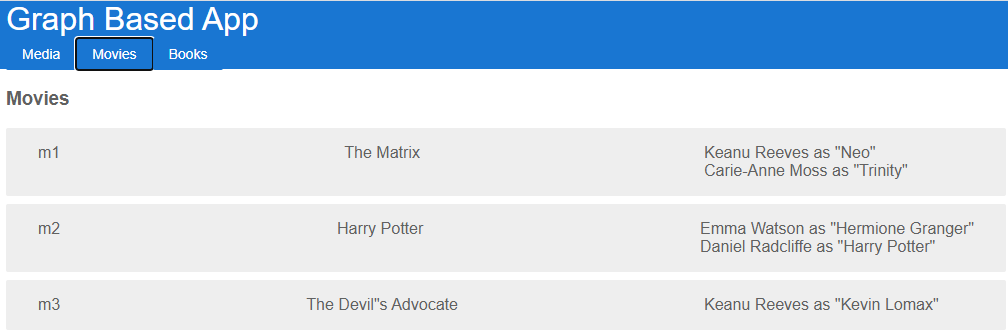
We added ‘Media’ and ‘Book’, added a ‘Person’ instance and a ‘Book’ instance and it gives us the following changes

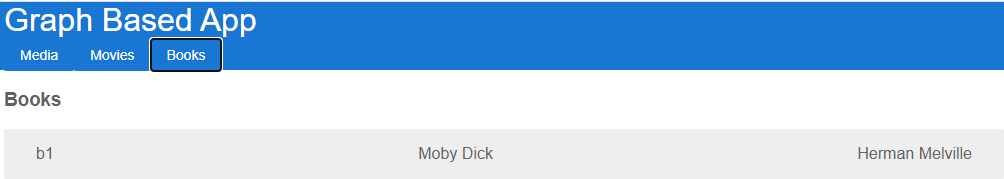


## Watching the changes

The app will display the following:

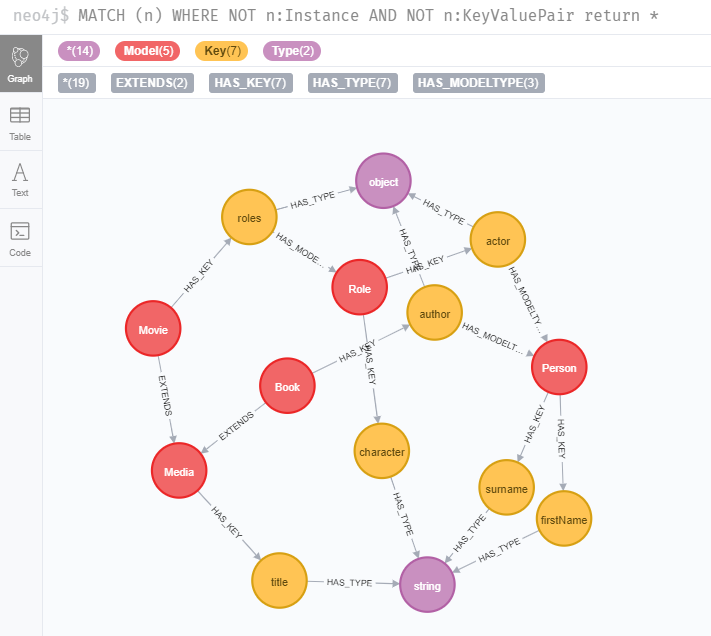






Note that the objects have been retrieved with the getInstances method, and the unique change in the query was the name of the model. Because the relation between models are described in the database the method can retrieve the appropriate objects for each model just knowing the model name.

The database model structure looks now like this:



A new relationship type was created:

* EXTENDS: tells the system how some models can extend other sharing their attributes.

The system is extensible, but what happens when introducing a new feature, that has a different model but is like existing ones.

# Merging new concepts

Let’s say that we have another app, displaying some ‘Items’ that are defined by their ‘name’.

We want to add these items to the application and be able to consider the ‘Media’ instances as ‘Items’.

## Adding Item model

First, we create the Item class:

export class Item{

    name: string;

}

With an ItemListComponent:

import { Component, OnInit } from '@angular/core';

import { ModelService } from '../model.service';

import {Item} from '../item';

import { Instance } from '../database';

@Component({

  selector: 'app-item-list',

  templateUrl: './item-list.component.html',

  styleUrls: ['./item-list.component.css']

})

export class ItemListComponent implements OnInit {

  constructor(private modelService: ModelService) { }

  items: Instance<Item>[];

  async ngOnInit(): Promise<void> {

    this.items = await this.modelService.getInstances('Item');

  }

}

And its template:

<h3>Items</h3>

<div class="item" \*ngFor="let item of items">

    <span>{{item.instance.id}}</span>

    <span>{{item.value.name}}</span>

</div>

## Handling navigation

Then we add a navigation button to the TopBarComponent template:

<a >

    <h1>Graph Based App</h1>

    <button [routerLink]="['/items']">Items</button>

    <button [routerLink]="['/media']">Media</button>

    <button [routerLink]="['/movies']">Movies</button>

    <button [routerLink]="['/books']">Books</button>

</a>

With the according route in the Routing module:

const routes: Routes = [

    {path: '', component: MovieListComponent},

    {path: 'movies', component: MovieListComponent},

    {path: 'media', component: MediaListComponent},

    {path: 'books', component: BookListComponent},

    {path: 'items', component: ItemListComponent}

  ];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

## Updating the sample

As there is no change in the other models, we just need to add the ‘Item’ concept in our DBModel collection

{

        name: 'Item',

        isEquivalentTo: ['Media'],

        keys: [{name: 'name', isEquivalentTo: 'title', type: 'string'}],

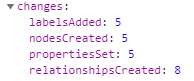
        instances: [

          {id: 'i1', properties: [{key: 'name', values: ['Item 1']}]}

        ]

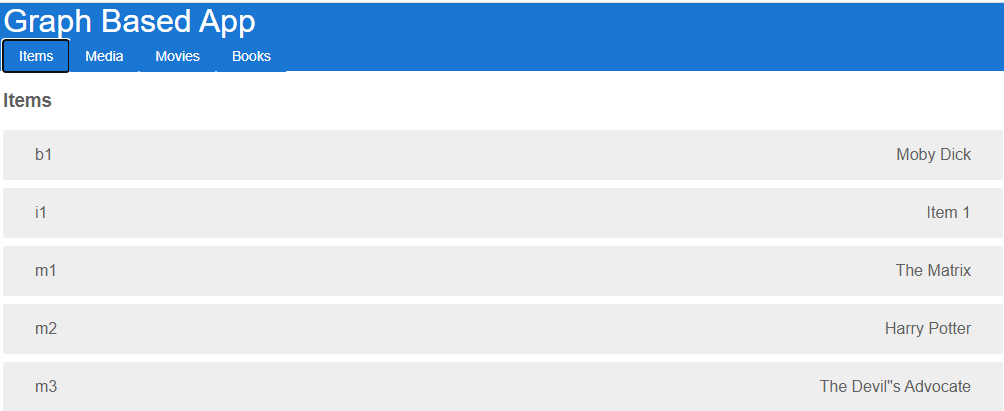
      }

When saving the sample, the following changes appear in the logs:

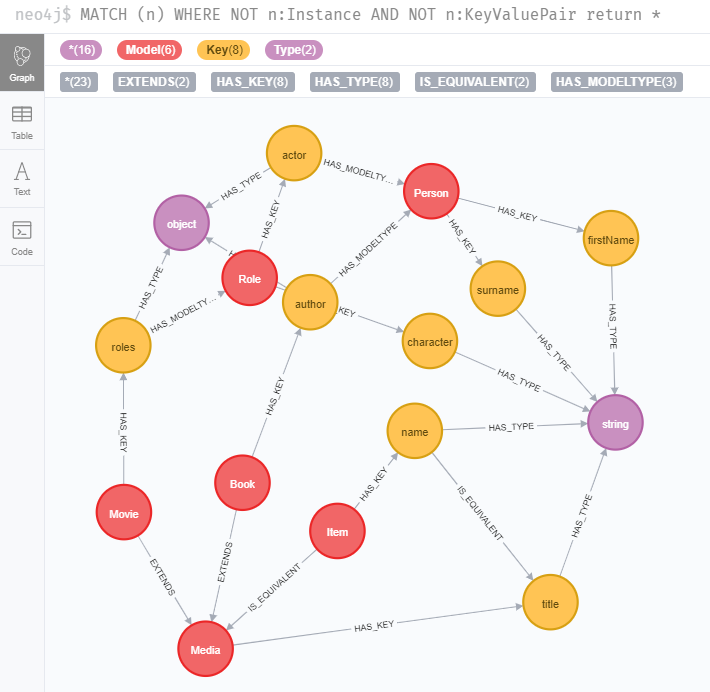


## Watching the changes

The application has now an Items panel displaying all the items and media objects



In the database, looking at the model structure:



We can see that a new relationship type was created:

* IS\_EQUIVALENT: tells the system that a model should consider the instance of other models as equivalent or that a key should be considered as equivalent to another

# Considerations about primitives

Primitives are instances without defined model. They will store number, or string. Having those primitives as nodes can be useful to retrieve transversal relationships by value between the different instances.

By example, Harry Potter is a movie title or a character.

This can also be interesting with dates.

Let’s create a Date model containing the keys ‘year’, ‘month’, ‘day’.

## Adding Date model

We will create first a DBDate model:

export class DBDate{

    year: number;

    month: number;

    day: number;

}

And add a ‘dob’ property to Person model and ‘releaseDate’ to Media model:

import { DBDate } from './dbDate';

export class Person{

    firstName: string;

    surname: string;

    dob?: DBDate;

}

export class Media{

    title: string;

    releaseDate?: DBDate;

}

We will then create a MediaComponent:

import { Component, OnInit, Input } from '@angular/core';

import { Media } from '../media';

import \* as moment from 'moment';

@Component({

  selector: 'app-media',

  templateUrl: './media.component.html',

  styleUrls: ['./media.component.css']

})

export class MediaComponent implements OnInit {

  moment = moment;

  constructor() { }

  @Input() media:Media;

  ngOnInit(): void {

  }

}

With its template:

<span \*ngIf="media">{{media.title}}<span \*ngIf="media.releaseDate">({{moment(media.releaseDate).year()}})</span></span>

And use the template in the MediaListComponent, BookListComponent and MovieListComponent templates:

<h3>Books</h3>

<div class="item" \*ngFor="let book of books">

    <span>{{book.instance.id}}</span>

    <app-media [media]="book.value"></app-media>

    <app-person [person]="book.value.author"></app-person>

</div>

<h3>Media</h3>

<div class="item" \*ngFor="let media of medialist">

    <span>{{media.instance.id}}</span>

    <app-media [media]="media.value"></app-media>

</div>

<h3>Movies</h3>

<div class="item" \*ngFor="let movie of movies">

    <span>{{movie.instance.id}}</span>

    <app-media [media]="movie.value"></app-media>

    <app-role-list [movie]="movie.value"></app-role-list>

</div>

We will also update the PersonComponent:

import { Component, OnInit, Input } from '@angular/core';

import { Person } from '../person';

import \* as moment from 'moment';

@Component({

  selector: 'app-person',

  templateUrl: './person.component.html',

  styleUrls: ['./person.component.css']

})

export class PersonComponent implements OnInit {

@Input() person: Person;

age: number;

  constructor() { }

  ngOnInit(): void {

    if (this.person.dob){

      this.age = moment(Date.now()).diff(moment(this.person.dob), 'years');

    }

  }

}

And its template:

<span \*ngIf="person">{{person.firstName}} {{person.surname}}<span \*ngIf="age">({{age}})</span></span>

## Updating the sample

The sample is updated with some date instances:

export const sample: DBModel[] = [

    {

        name: 'Date',

        keys: [{name: 'year', type: 'number'}, {name: 'month', type: 'number'}, {name: 'day', type: 'number'}],

        instances: [

          {id: '1964-9-2', properties: [{key: 'year', values: [1964]}, {key: 'month', values: [9]}, {key: 'day', values: [2]}]},

          {id: '1964-8-27', properties: [{key: 'year', values: [1964]}, {key: 'month', values: [8]}, {key: 'day', values: [27]}]}

        ]

    },

    {

        name: 'Person',

        keys: [{name: 'firstName', type: 'string'}, {name: 'surname', type: 'string'}, {name: 'dob', type: 'object', model: 'Date'}],

        instances: [

          {id: 'p1', properties: [{key: 'firstName', values: ['Keanu']}, {key: 'surname', values: ['Reeves']}, {key: 'dob', values: ['1964-9-2']}]},

          // ...

        ]

      },

      // ...

      {

        name: 'Media',

        keys: [{name: 'title', type: 'string'}, {name: 'releaseDate', type: 'object', model: 'Date'}]

      },

      {

        name: 'Movie',

        extends: ['Media'],

        keys: [{name: 'roles', isArray: true, type: 'object', model: 'Role'}],

        instances: [

          // ...

          {id: 'm4', properties: [{key: 'title', values: ['Mary Poppins']}, {key: 'roles', values: ['r6']}, {key: 'releaseDate', values: ['1964-8-27']}]}

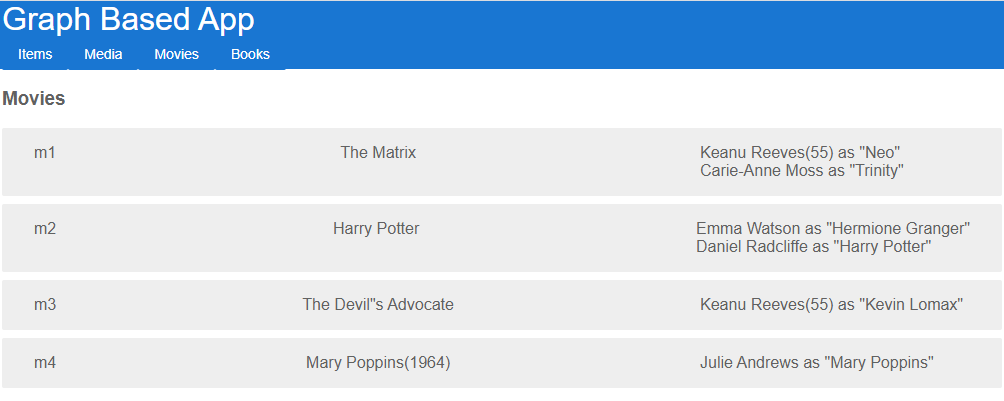
        ]

      }

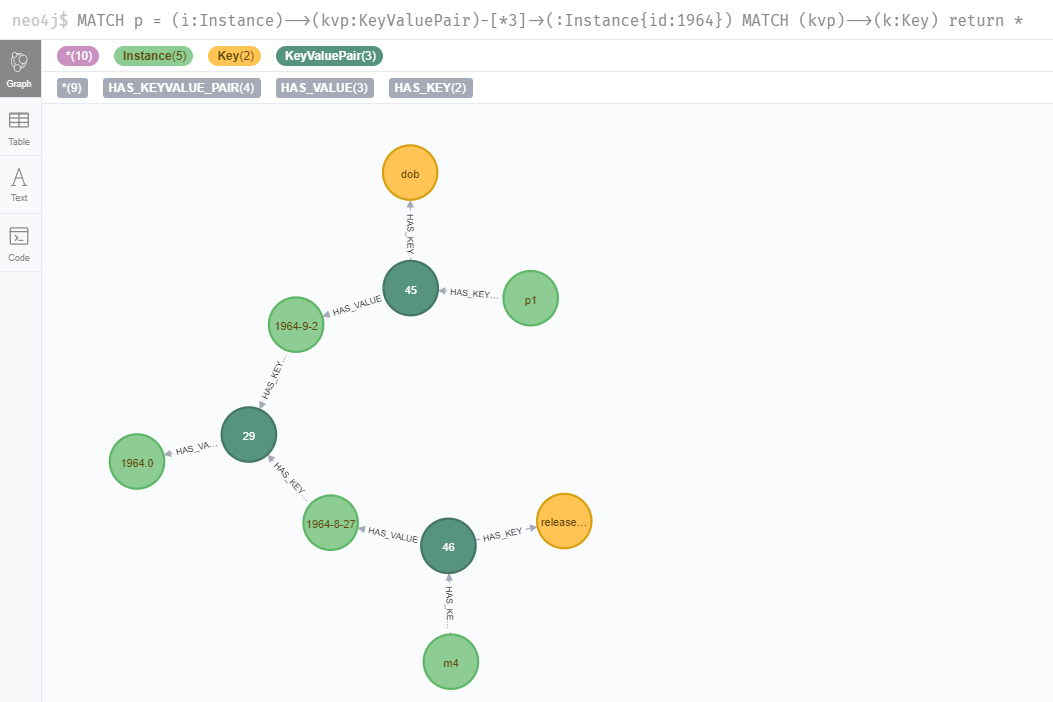
    ];

## Watching the changes

The application is now displaying actor’s age (for Keanu Reeves) or media release date (for Mary Poppins):



In the database if we focus now on ‘1964’ primitive:



We can find a correlation between a day of birth and the release of a movie. Once again, if a user want to know which movie was released on a specific year, the query won’t have to look for all the instances matching the condition: they will be linked to the specific year primitive through Date instances. Looking for the instances of ‘Movie’ having a key/value pair mapping the key ‘releaseDate’ and a ‘Date’ instance that have a year primitive being the desired year, we will be able to answer to the user without exploring the entire ‘Movie’ database.

The good thing having primitives as node instead as properties is that the objects of the database will define relationships by value that will be very useful when you’ll want to analyze or search your database. You can start from a Key, a Model, or an Instance, you’ll always find all the object related to the node at a few hops (the number of hops depending of you data structure that is also defined in the database).

Now that we have explored some typical application development use cases, let’s have a closer look at the cypher queries behind the ModelService.

# Queries in details

## Instances retrieval query

The ModelService has a method getInstances that will:

* Query all the instances of a specific model using a cypher query that will return json objects. The query can optionally be filtered by instance id
* Transform json objects into Instances object that will contain the instance meta data and its object representation

### Cypher query

Here’s the code of the query:

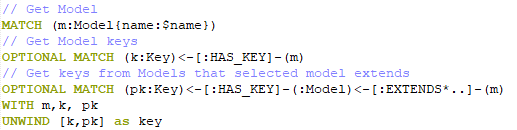


#### Getting the keys

The query first tries to reach the model by its name (the parameter of the query)

It looks then for the keys defined by the model and the keys defined by the parent models (if they exist) and uses those keys as a unique collection named ‘key’.

Example: A ‘Book’ model will have ‘author’ key from book and ‘title’ key from the ‘Media’ model because ‘Book’ extends ‘Media’.

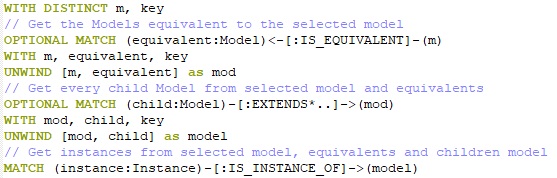


#### Getting the instances

With the model and all its keys, the query search for eventual equivalent model then for all the models extending them. The model, the equivalents and their children are gathered in a single collection a as ‘model’.

Example: an ‘Item’ model is equivalent to ‘Media’ and ‘Media’ is extended by ‘Book’ and ‘Movie’. These 4 models will be considered now as ‘model’.

The query can then get all the instances of the models:

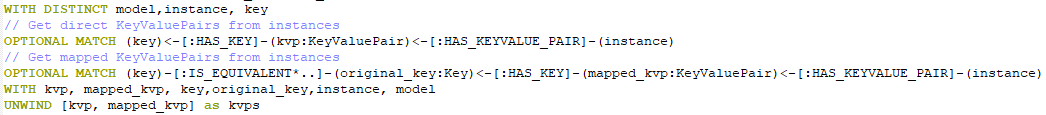


#### Getting the key/value pairs

The query will search for the KeyValuePair nodes of each instances that are related to a model key. It will also search for the KeyValuePair that might be equivalent to a model’s key.

Example: ‘Item’ model will have instances from ‘Book’, ‘Movie’, ‘Media’ and ‘Item’. ‘Item’ has the key ‘name’, but ‘name’ being equivalent to ‘title’, the query will gather all the KeyValuePair from the instances of ‘Book’, ‘Movie’, ‘Media’, or ‘Item’ that are related to ‘name’ or ‘title’

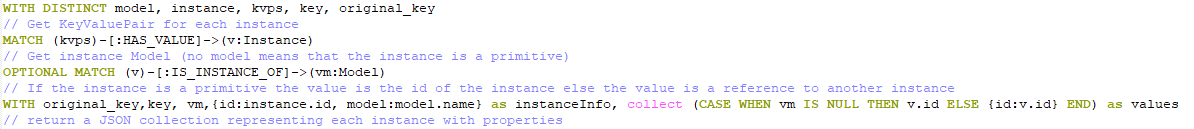
The KeyValuePair will be gathered as ‘kvps’:



#### Getting the values

Now, the query can find the values associated to each KeyValuePair. It will also try to find the model associated with each value. If there is a model, the instance should be considered as a reference otherwise as primitive. An instance reference will be an object defining the instance id. A primitive will return its id property as a value.

Example: A ‘Book’ has a ‘title’ and an ‘author’. The instance ‘b1’ has a value for ‘title’ which not related to any model so it will return the id of the instance which is ‘Moby Dick’. It has also a value for ‘author’ which is related to the model ‘Person’ so it will return {id: ‘p1’} which is a reference to the instance ‘p1’.

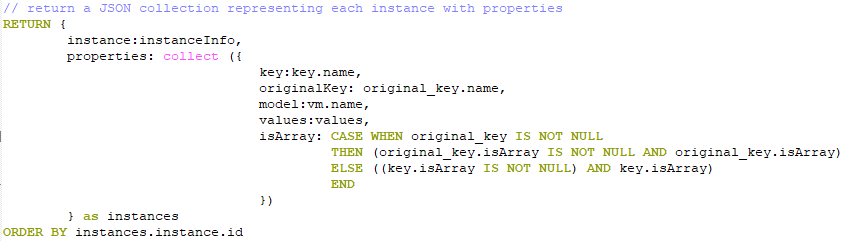


#### Formatting the result

The instances will be return as an array of json object containing:

* The instance meta data (containing the id and the model name of the instance)
* The instance properties: an array of object containing:
  + The key identifier
  + The original key identifier in case of key mapping between equivalent models
  + The model in case the values are instance references
  + The values
  + The information if the key is an array or a single value

Here’s the code



In the application the result of the query is defined as collection of Record described as following:

export class InstanceRef{

    id: string; // Database instance id

}

export class Record{ // Db query result record

    instance: InstanceRef; // Instance reference

    properties: KeyValuePair[]; // Instance properties as keyvaluepair collection

}

export class KeyValuePair{ // Db query result keyvaluepair

    key: string; // Key name

    isArray?: boolean; // Stores the informations that defines if the key is a collection or a single object

    model?: string; // Model name if values are instances references

    values: any[]; // The values which should be number or string representing primitives or instance id

    originalKey?: string; // The original key name if the instance came from on an equivalent model with an equivalent key

}

### getInstances method

In the ModelService, the method is defined as following:

async getInstances<T>(modelName: string,

                        filterInstances?: InstanceRef[],

                        recursiveDepth?: number,

                        currentDepth?: number): Promise<Instance<T>[]>{

    currentDepth = currentDepth || 0;

    const session = this.driver.session();

    try {

      const filter = (filterInstances && filterInstances.length) ? filterInstances.map(value => value.id) : [];

      const query = '...'; // cypher query defined here

      const result = await session.run(

        query,

        {name: modelName, filter}

      );

      const instances = result.records.map(async (value) =>

        {

            const i = value.get(0) as Record;

            const v = {};

            const r: Instance<T> = {instance: null, value: null};

            if (i){

              r.instance = i.instance;

              if (typeof(recursiveDepth) === 'undefined' || recursiveDepth === null){

                recursiveDepth = 0;

              }

              if (i.properties && i.properties.length){

                  for (const element of i.properties) {

                  if (!element.model ||

                     recursiveDepth < 0 ||

                      (recursiveDepth > 0 && currentDepth >= recursiveDepth) ) {

                  v[element.key] = element.isArray ? element.values : element.values[0];

                  }

                  else{

                    const recursiveInstances = await this.getInstances<any>(element.model,

                       element.values,

                        recursiveDepth,

                         currentDepth + 1);

                    const recursiveValues = recursiveInstances.map(inst => inst.value);

                    v[element.key] = element.isArray ? recursiveValues : recursiveValues[0];

                  }

                }

              }

              r.value = v as T;

            }

            return  r;

        } );

      return Promise.all(instances);

    } finally {

      await session.close();

    }

  }

The method takes 4 arguments:

* The model’s name
* An optional InstanceRef collection that can be used as a filter
* An optional recursive index that defines how many levels of instances should be extracted (0 means all). Example: A Movie is composed by a ‘title’ and ‘roles’. The ‘roles’ property has instances of ‘Role’ which have instance of ‘Person’ in ‘actor’ property, which has an instance of ‘Date’ as ‘dob’ property. If the recursiveDepth is equal to 0, the method will recursively call itself until it reaches only primitives. If it is equal to 1, it will only return the Instances references for the ‘roles’ and not the Role objects. If it is equal to 2, it will return the Movie with the Roles objects, but the actor will be an InstanceRef.
* The current recursive level

It calls the instances Record objects using the query described above adding optionally a filter for the instances with that line:



Then for each Record, it creates an instance object setting the properties of the object dynamically using the key and values properties of the KeyValuePair collection of the Record.properties.

It is using recursivity to optionnaly retrieve the complete objects.

It returns the Instance collection that can then be used directly by the application components.

## Data Base update query

### Cypher query

The cypher query takes a DBModel array as parameter and update the database accordingly.

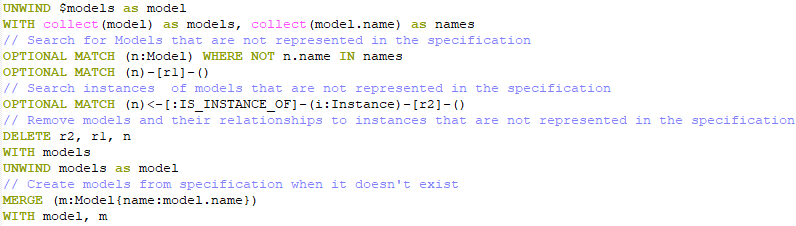
It removes the Model nodes and the relationships existing in the database and not in the DBModel objects and create the nodes and relationships that doesn’t exist in the database.

The query is quite long because it does a lot of thing. It was created more to be able how the structure can be updated using queries. I discovered cypher creating this project and I have to admit that I was amazed how the language is intuitive and allows to perform complex tasks in a single query.

***Note***: In the real life, we won’t probably never use such a query, but it’s convenient for a POC.

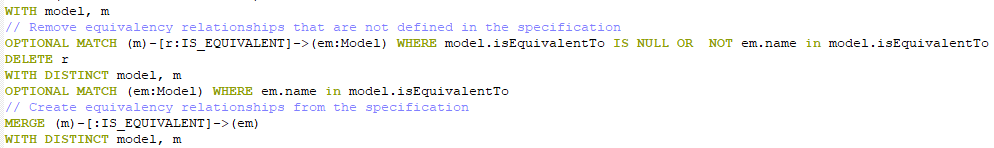
#### Removing or Creating the Models

The query first looks for the Model objects existing in the database and not in the DBModel collection and remove them. Then it merges the Model objects using the name property of the node.



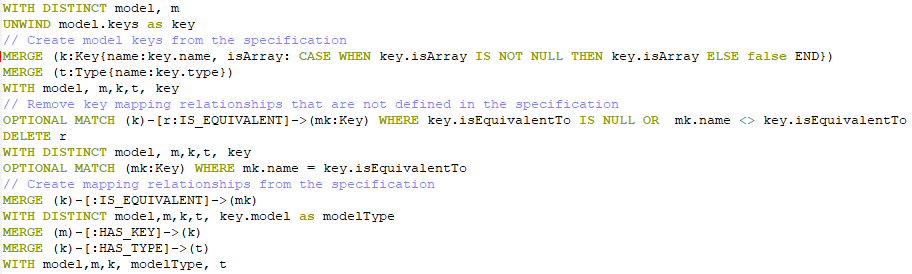
#### Removing or Creating Equivalency relationships

The query then looks for the relationships that are not defined in the specification (the DBModelcollection) and remove those relationships int the database. It will then create the [:IS\_EQUIVALENT] relationship between the Model objects according to the specification:



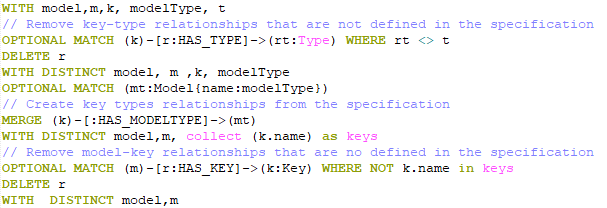
#### Creating Keys, Types and Removing or Creating Equivalency relationships between Keys

The query creates then the Key nodes related to a model. It merges the Type objects for each key and remove or merge the [:IS\_EQUIVALENT] relationships according to the specification. It creates then [:HAS\_KEY] and [:HAS\_TYPE] relationships that links models to keys and keys to types:



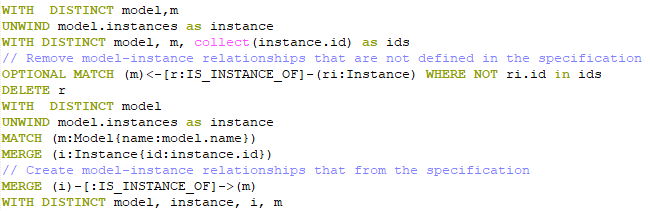
#### Updating Key and Model relationships

The query delete then the relationships that are not defined in the specification and create optionally the [:HAS\_MODELTYPE] relationships between a key and its model when a key is related to another object. It removes then all the [:HAS\_KEY] relationships between model and keys that are not defined in the specification:



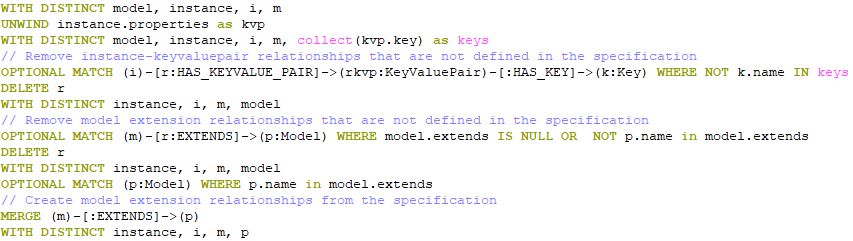
#### Merge Instance and Update Instance and Model relationships

The query then removes the relationships [:IS\_INSTANCE\_OF] that are not defined in the specification. Then merge Instance defined for each model and create the relationship between them.



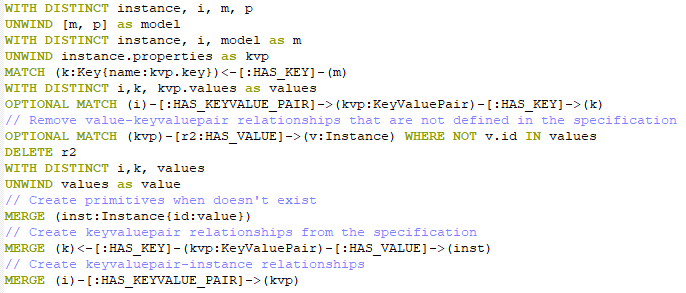
#### Updating Instance/KeyValuePair relationships and extends relationships between Model objects

The query continues removing unspecified relationships between the Instance and KeyValuePair nodes then update the [:EXTENDS] relationships between Model objects according to the specification



#### Merging Value Instances, KeyValuePair and updating relationships

Finally, the KeyValuePair / Value relationships are updated, merging the Instance by id.



### createOrUpdateDBModels method

The method simply the cypher query described above and return the update information from the server (number of nodes, labels, relationships that were created or deleted).

private async createOrUpdateDBModels(models: DBModel[]): Promise<{[key: string]: number}>{

    if (models){

      const session = this.driver.session();

      try{

        const query = '...'; // cypher query defined here

        const result = await session.run(

          query,

          {name: 'models', models}

        );

        return result.summary.counters.updates();

      }

      finally{

        await session.close();

      }

    }

  }

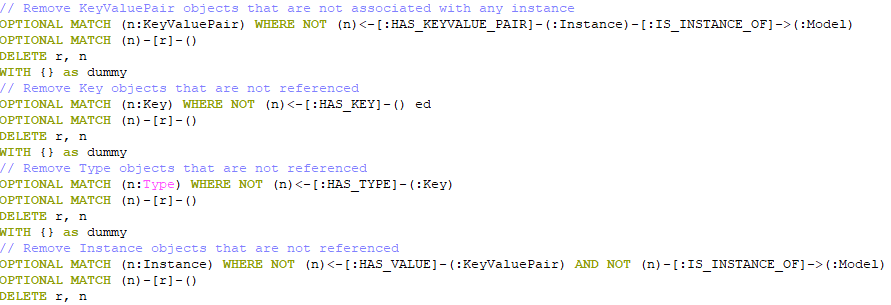
## Remove unused objects query

### Cypher query

The previous query might have removed some relationships and the database will have nodes that have no more meaning. That query ensures that every unwanted object is removed from the database.

The query removes:

* The KeyValuePair nodes that are not associated with any instance
* The Key nodes that are not referenced
* The Type nodes that are not referenced
* The Instance nodes that are not referenced



### clearUnusedObjects method

In the ModelService, the method is simply calling the cypher query described above returning the counters.

private async clearUnusedObjects(): Promise<{[key: string]: number}>{

    const session = this.driver.session();

    try{

      const query = '...'; // cypher query defined here

      const result = await session.run(

        query

      );

      return result.summary.counters.updates();

    }

    finally{

      await session.close();

    }

  }

# Edition and Versionning considerations

You probably noticed that there is a ‘Type’ node and it was never used during the experiments.

The type will be important to handle generic editor that won’t be covered in that sample.

We can easily imagine adding some edition information on types or keys such as editors, validation rules (a title should contain 100 character at max, etc..). This should allow us to specify the edition rules and components directly to the database to ensure edition requirements flexibility.

Another concept that might be interesting: Version. Adding the label ‘Version’ with the relationships [:FROM\_VERSION] and [:TO\_VERSION] will allows us to track the different versions of our models in the database. A client query could be able then to retrieve a specific data structure for a specific version. Developers would be also able to track the history of the application which can be very interesting.

# Conclusion

We have discovered during that "little" experiment that a graph database could be used to store primitives as node to create physical relationships between different objects and concepts without have to anticipate them. The system should generate a lot of subsets of objects linked by value that should allow queries to avoid looking into the entire database.

The Model, Key, KeyValuePair, Instance approach also allows to dynamically generate the way these primitives should be returned without having to change the query.

This will of course have a cost on the size, and there will requires the improve this system for productions project, but I'm really excited by the opportunities that Graph database can offer to application development.