Graph based apps with dynamic models experimentations

# 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 syncronisation 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.

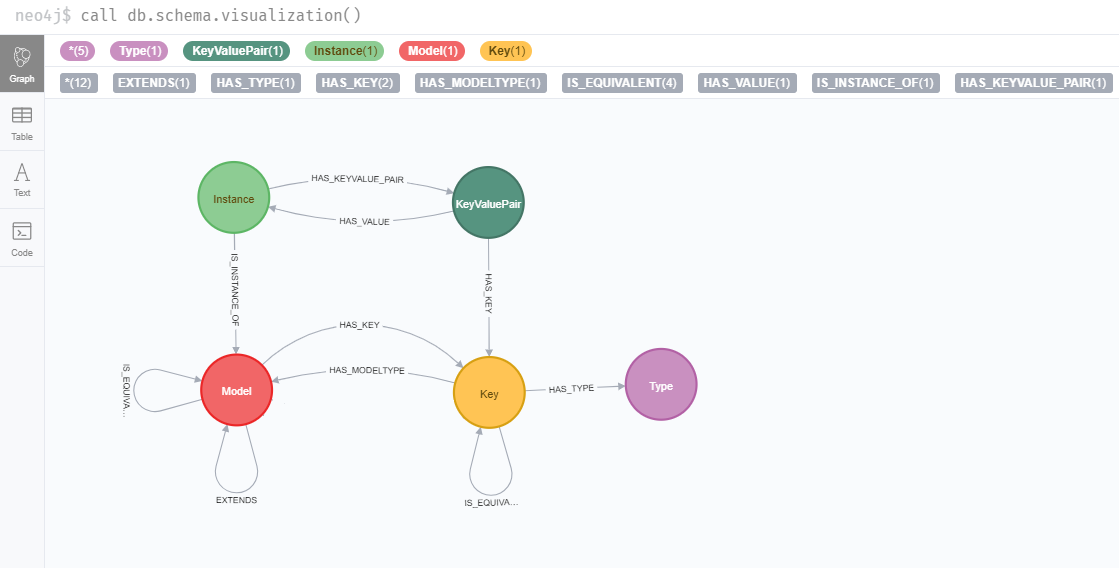
The current experimentation is trying to leverage on Graph database to have a dynamic binding system.

# Application Environment

The current sample will show a simple Angular web app, displaying data from a Neo4j Graph database.

# Database dynamic model

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)

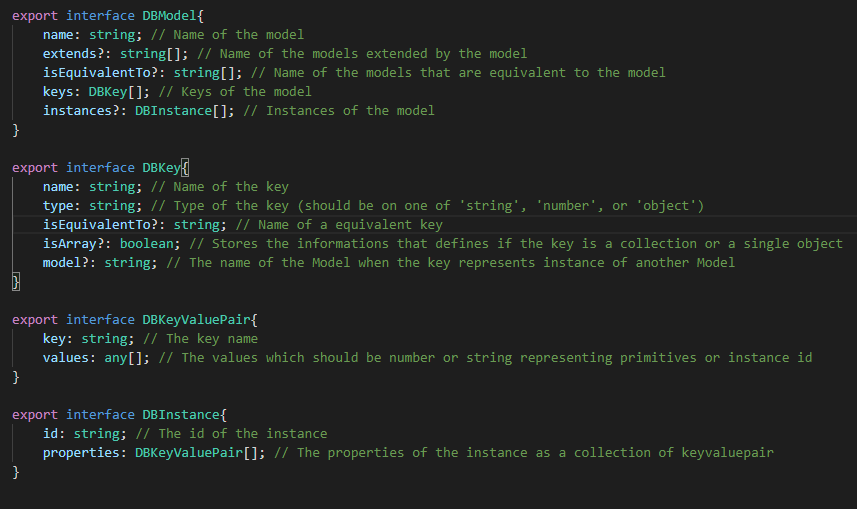
And 8 relationships:

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

This describe the following:

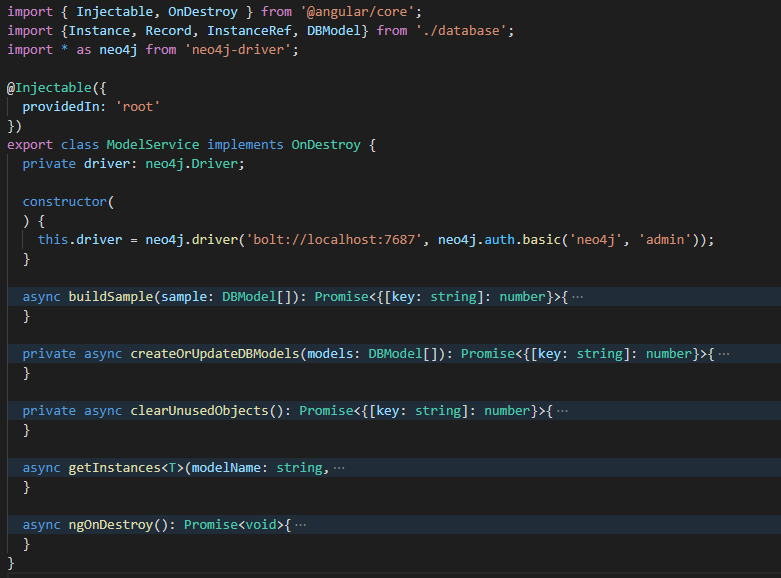
* A Model defines keys
* Instances are related to a Model
* Instances define a collection of KeyValuePair
* KeyValuePair is mapping a specific Key to a specific Instance
* A Key has a Type, and can be related to a Model (if the type is an object)
* A Model can extend other models
* A Model can be equivalent 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 should be expressed in typescript as the following:



# Database Interaction

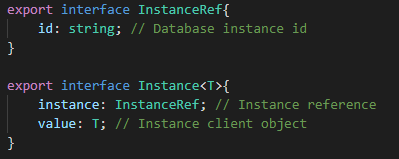
The application will communicate using the ModelService:



That service is able to:

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

An Instance<T> is defined as following:



* 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 java object that will be used by the application

The model service 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 instance
* getInstances will take a model name and will retrieve the instances for that model

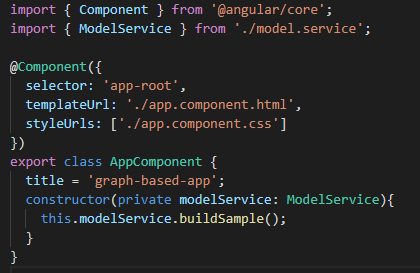
It has also 2 private methods:

* 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 exploration. They will be detailed at the end of the document.

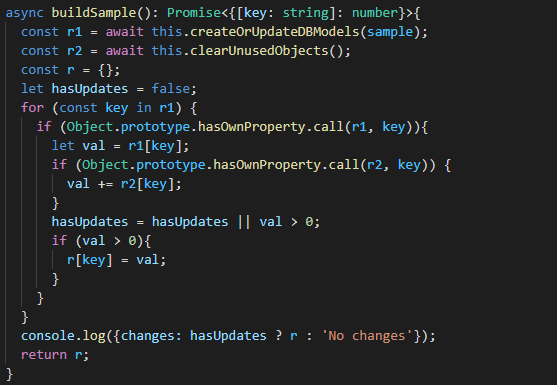
# Let’s build a simple Movie application

## Building the sample



The application will run the ModelService.buildSample() application once, when launching.

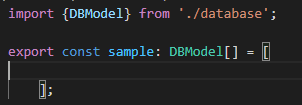
The buildSample code is detailed below:



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



Which means that de database contains nothing and should be empty

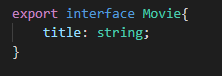
During the evolution of the project we will modify the sample to declare what we want in the database.

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

## Displaying the movies

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

## Adding Movie model



We create then a MovieListComponent:

A screenshot of a cell phone

Description automatically generated

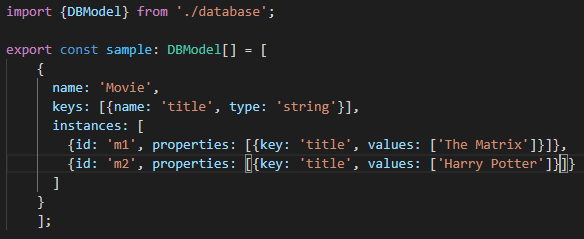
And its template

A close up of a screen

Description automatically generated

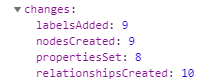
## Updating the sample

Let’s update the sample like this:



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 update the sample, we will see the following logs:



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



This is because of the createOrUpdateDBModels is not incremential but differiental (the query will be detailed later).

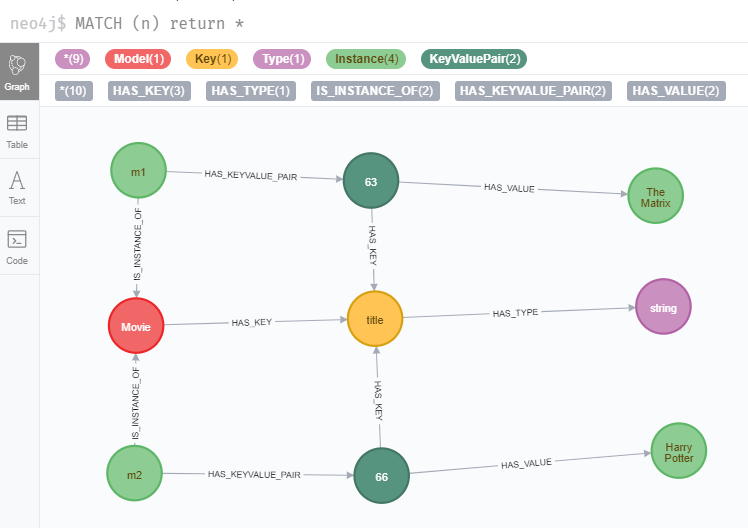
## Watching the changes

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:



If you look to the labels we have:

* Model
* Key
* KeyValuePair
* Instance
* Type

If you look to the relationships, we have:

* IS\_INSTANCE\_OF
* HAS\_KEY
* HAS\_KEYVALUE\_PAIR
* HAS\_VALUE

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

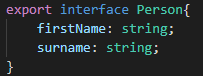
This way of doing creates more nodes than for a static model, and add complexity to the graph and some verbosity to the query, but due to the efficiency of Graph database to deal with relationships, it should also have some benefits that we will explore trying to improve the application.

# 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 an interface Person as following:



And a PersonComponent:

A screen shot of a smart phone

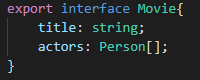
Description automatically generated

With its template:

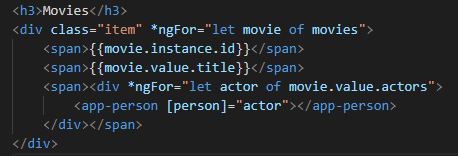


## Improving the Movie model

The movie model should know exposes an ‘actors’ property which is a Person collection:

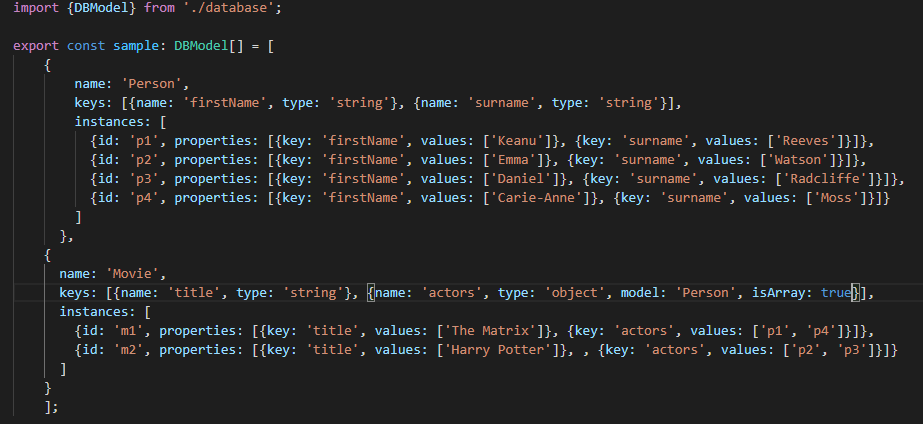


We can then improve the MovieListComponent template:



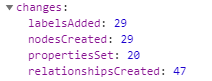
## Updating the sample

We will update the sample as following:



We have added the Model ‘Person’ and some instances of it, and added the ‘actors’ key

And the DB changes log will be:



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

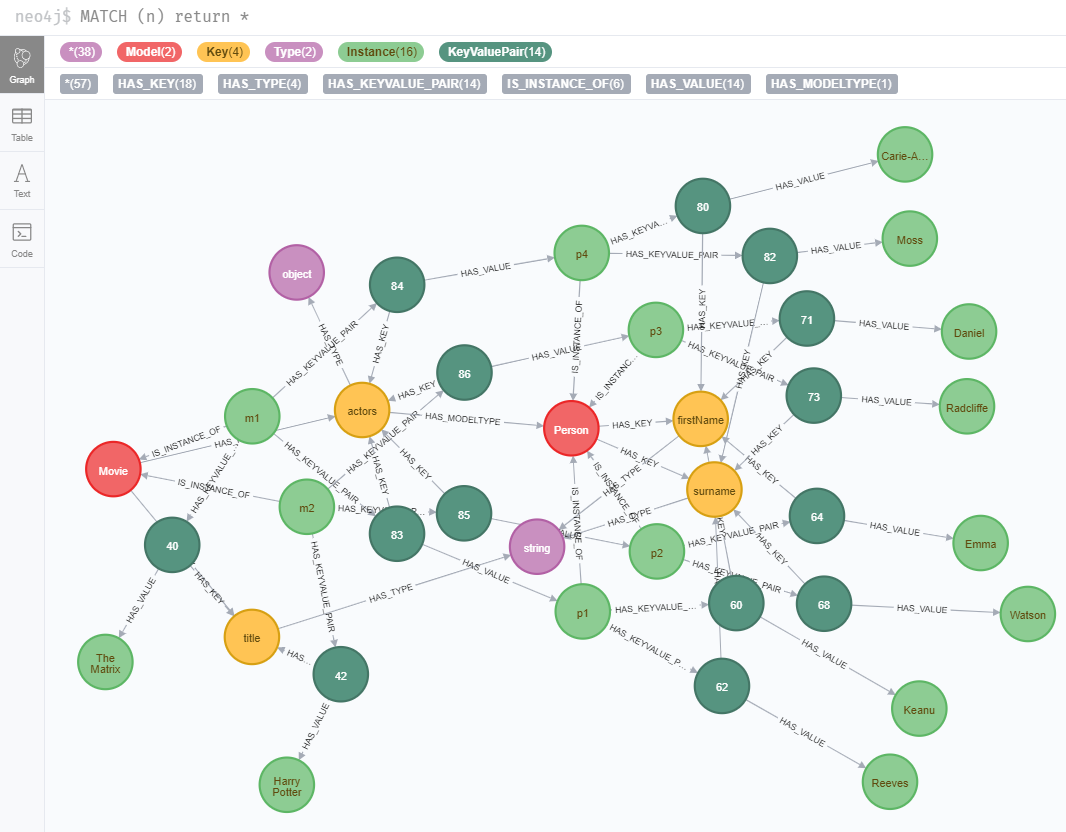
## Watching the changes

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

No new labels were added

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 instances.

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

# Changing the data structure

Well, having the actors, 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’ int the ‘Movie’ model.

## Adding Role model

This is the Role interface definition:

A picture containing sitting, holding, city, person

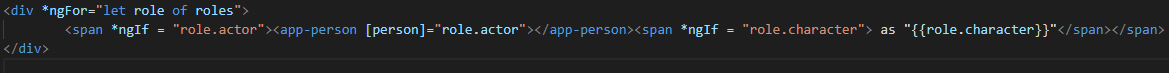
Description automatically generated

With a RoleListComponent:

A screenshot of a cell phone

Description automatically generated

And the template:



## Updating Movie model

The Movie interface is now like this:

A screenshot of a cell phone

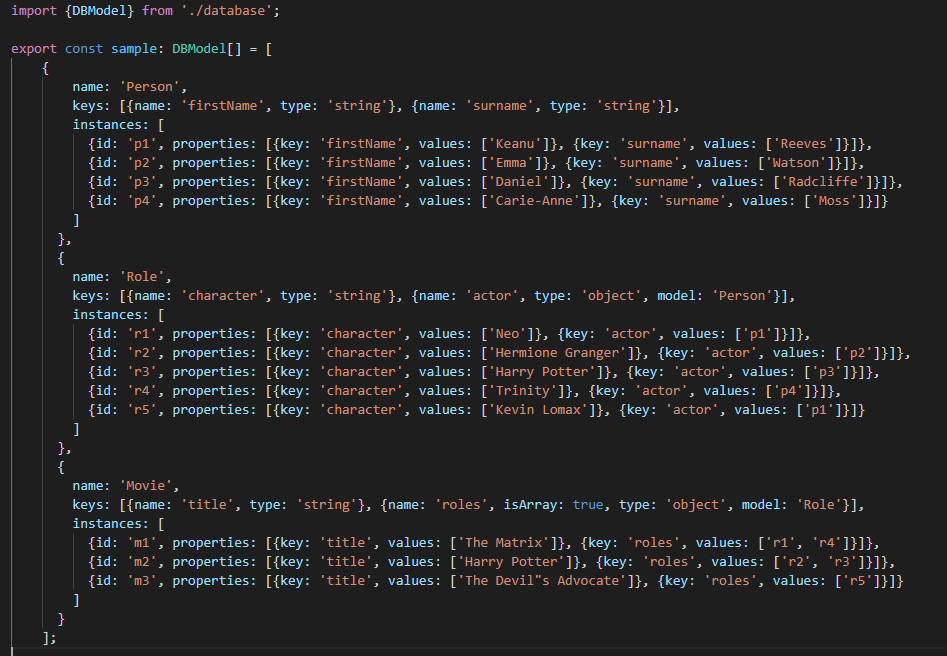
Description automatically generated

And the MovieListComponent template will be:

A screen shot of a person

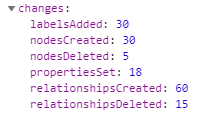
Description automatically generated

## Updating the sample



The sample reflects the changes in the model. We added the Role model, and the Movie update. We have also added a new movie.

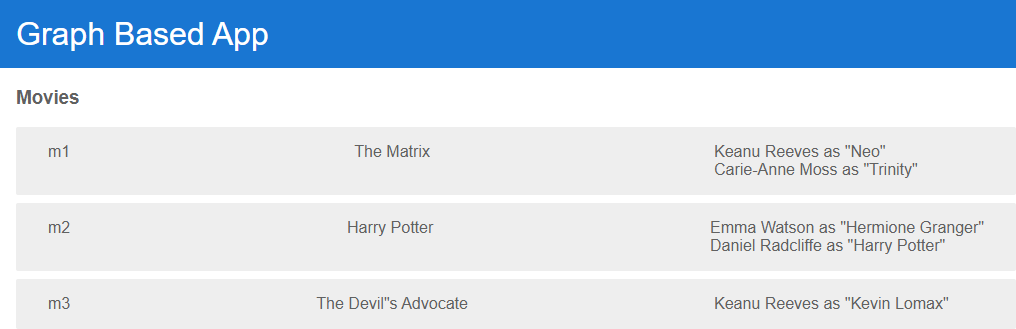
When running the application for the first time after updating the sample the changes in the databases logged by the application are:



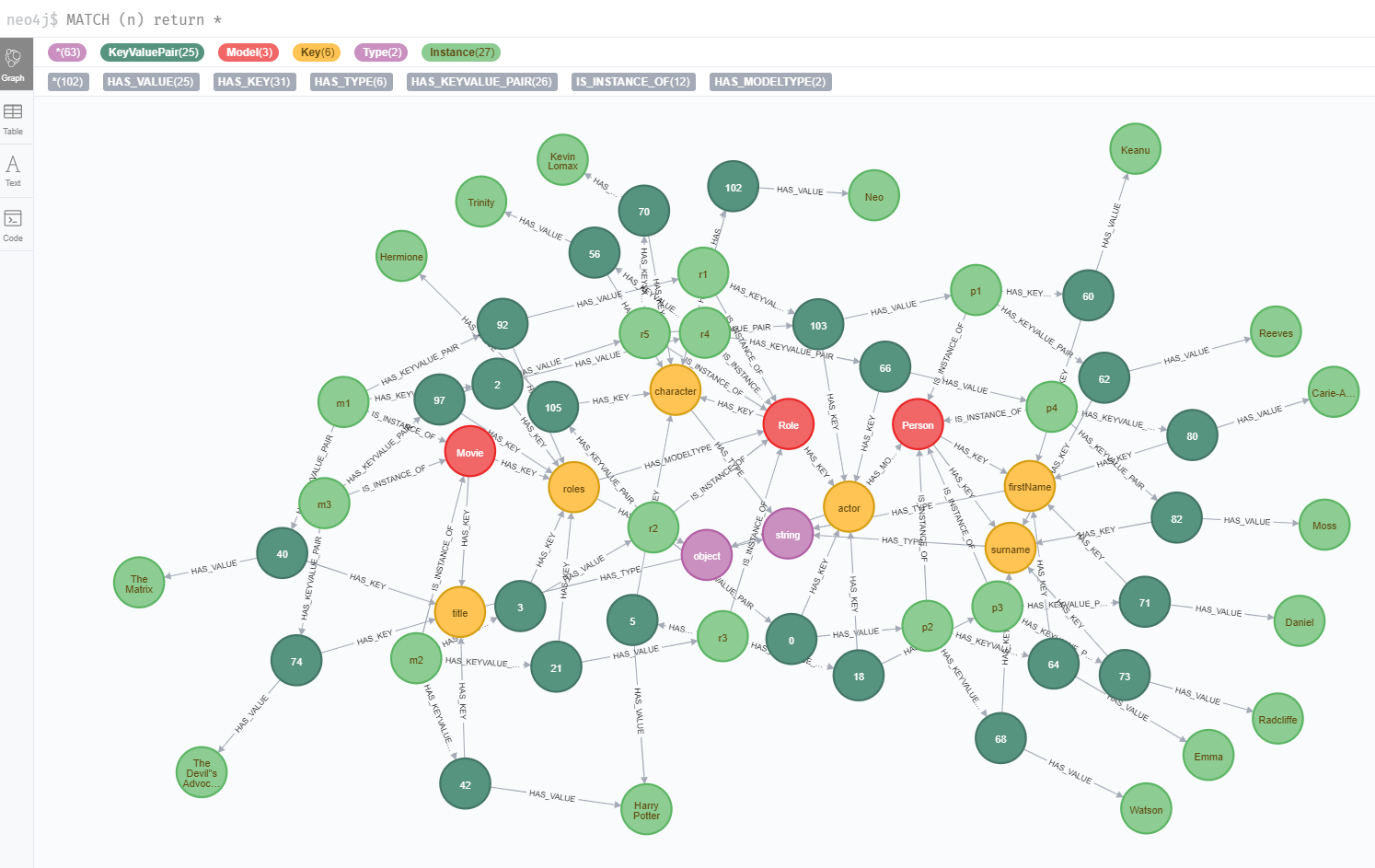
Some nodes and relationships were deleted: because the ‘actors’ properties is not used anymore, it was removed with all the key/value pairs related to it.

## Watching the changes

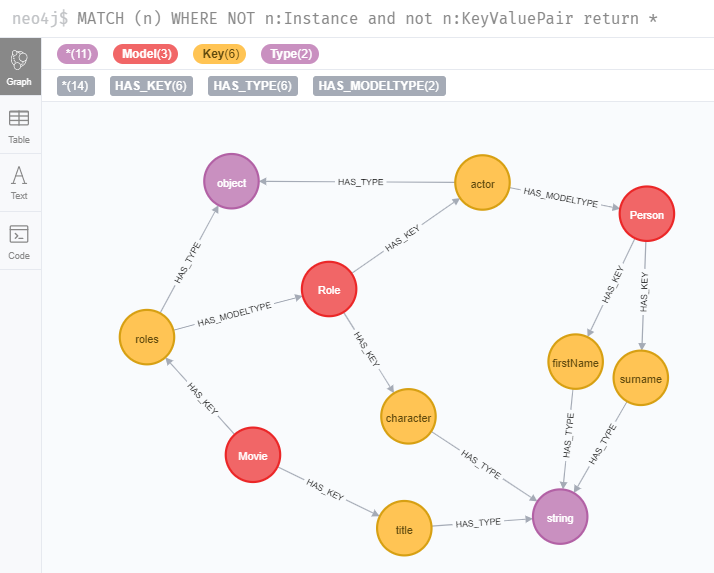
The application will now display:



The database looks now like:



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

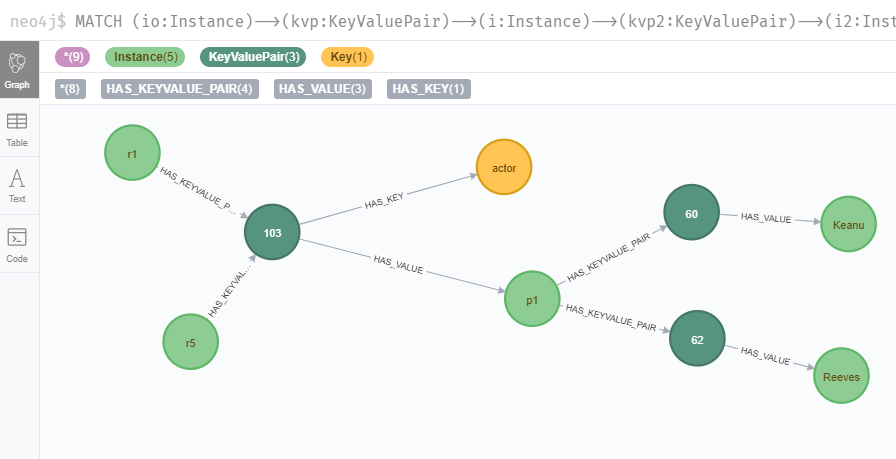


Excluding the instances and the key/value pairs we can focus on the data structure and we can see

That:

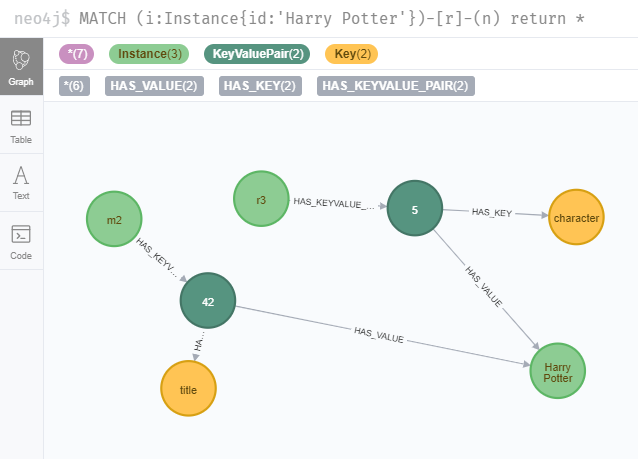
* the property ‘actors’ has disappeared from the model ‘Movie’ which has now the property ‘roles’ with ‘Role’ as model type
* The model ‘Role’ was added with 2 keys ‘character’ and ‘actor’ which is a ‘Person’
* The ‘roles’ and ‘actor’ keys have an ‘object’ type
* The other keys are defining primitives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

Let’s have a look at Keanu Reeves now:



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

If we have a look 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

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 remove the nodes without specific relationships.

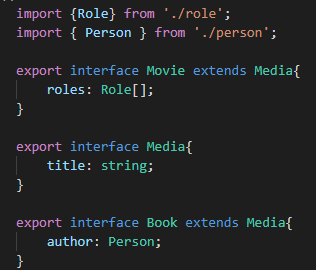
Let’s explore now how we can extend our model.

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



This will allow us to create a BookListComponent:

A screenshot of a cell phone

Description automatically generated

With its template:

A picture containing holding, sitting, person, orange

Description automatically generated

And a MediaListComponent:



With its template:

A screen shot of a person

Description automatically generated

## Adding navigation system

We added some buttons in the TopBarComponent template with different routes:

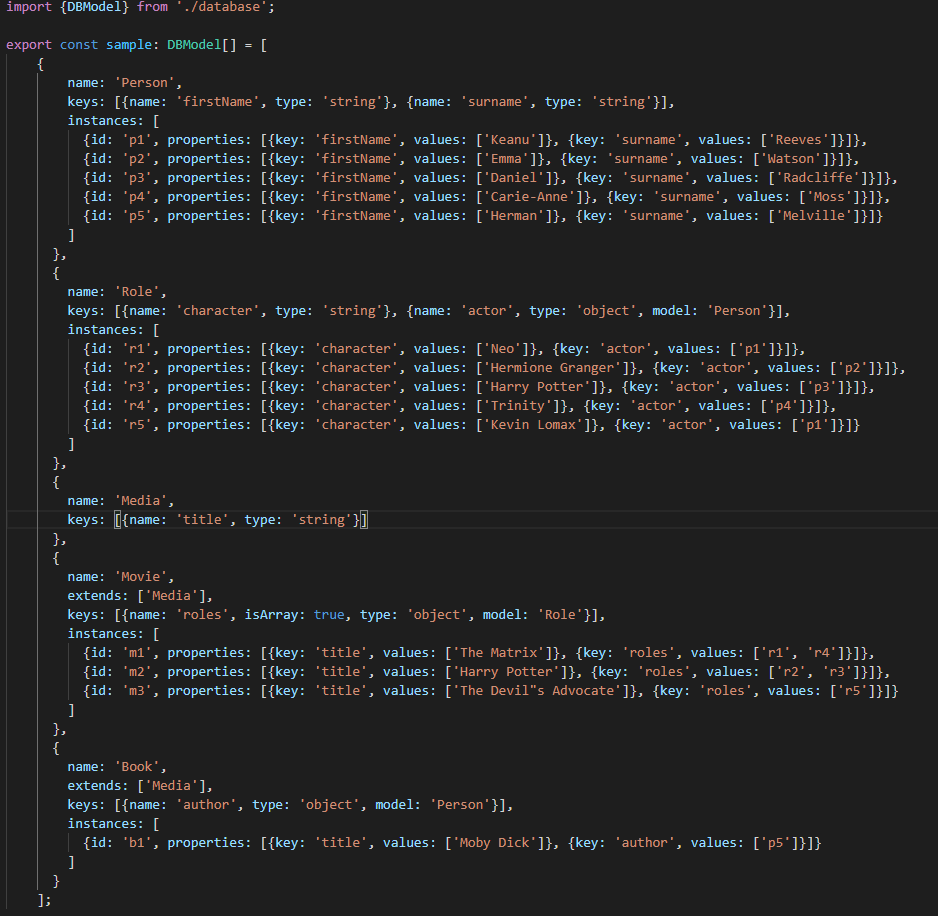
A close up of a sign

Description automatically generated

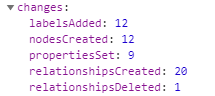
Our app having a routing module describing the following routes:



## Updating the sample

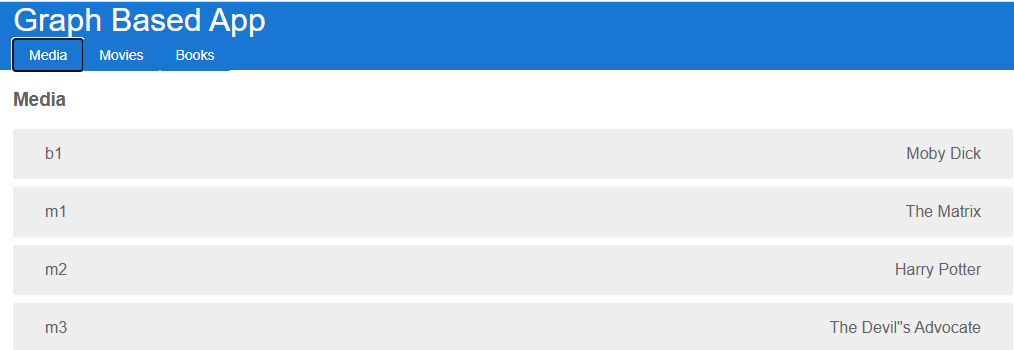


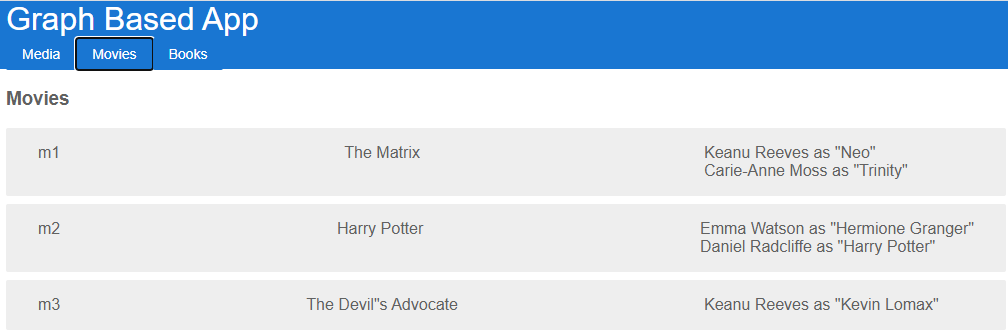
We added ‘Media’ and ‘Book’, added a ‘Person’ instance and a ‘Book’ instance and it gives us the following changes

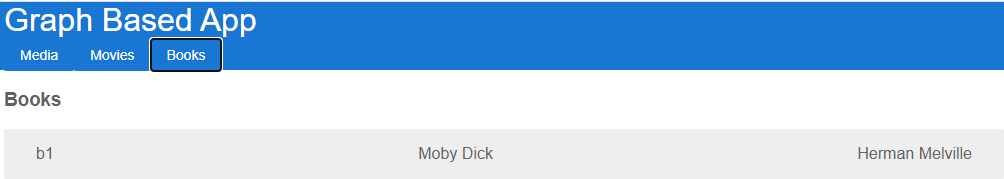


## Watching the changes

The app will display the following:

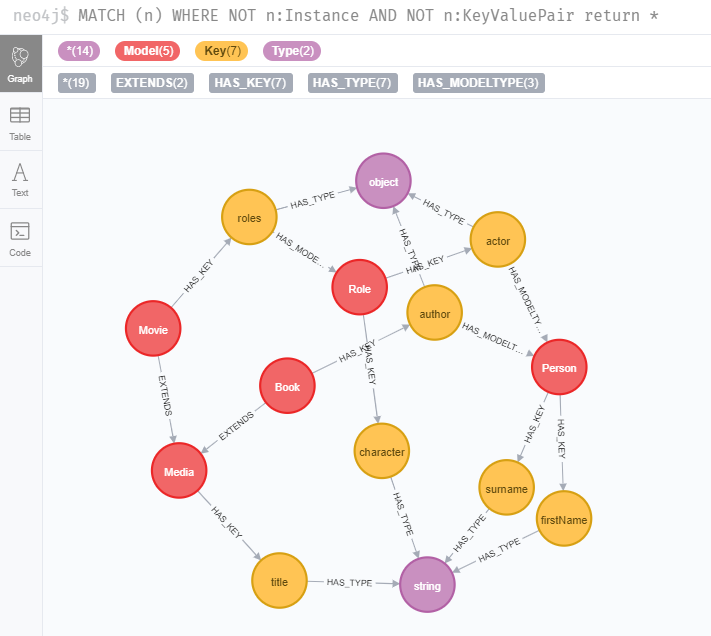






Note that the objects have been retrieved with the query getInstances, and the unique change in the query was the name of the model. Because the relation between models are described in the database the method can retrieve the appropriate objects for each model just knowing the model name.

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’ interface:

A picture containing object, clock, dark, sitting

Description automatically generated

With an ItemListComponent:

A screen shot of a social media post

Description automatically generated

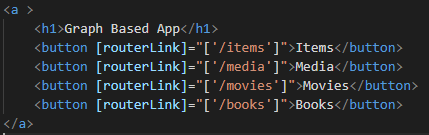
And its template:

A close up of a screen

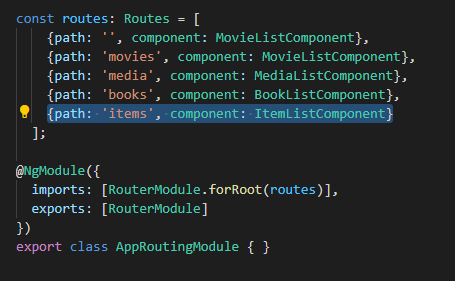
Description automatically generated

## Handling navigation

Then we add a navigation button to the TopBar:

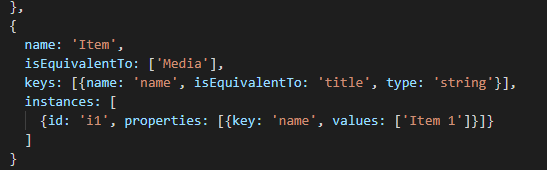


With the according route in the Routing module:

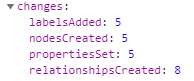


## Updating the sample

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

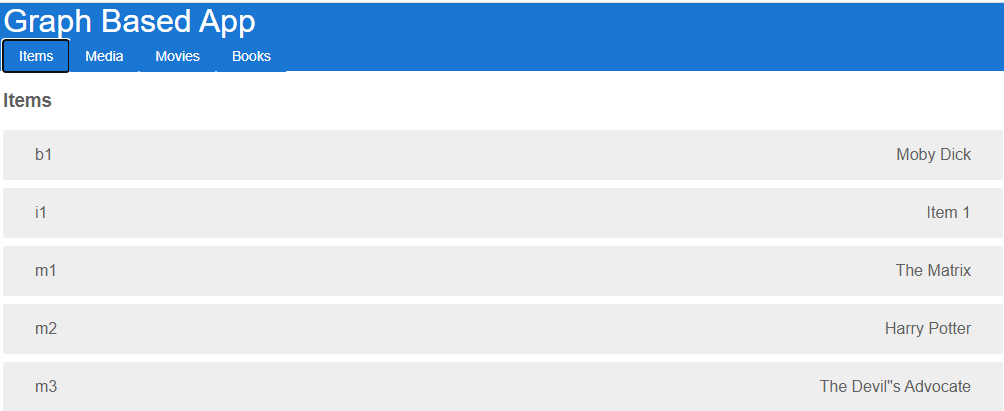


When saving the sample, the following changes appear in the logs:

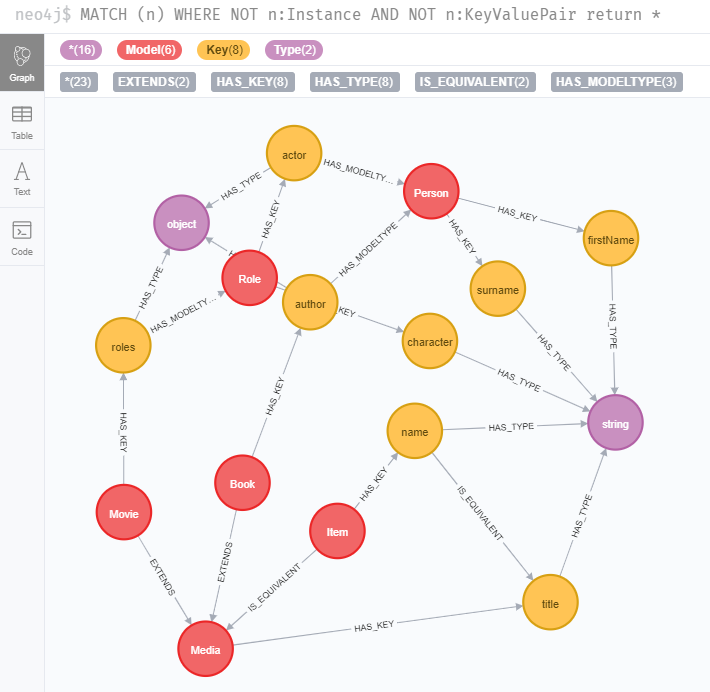


## Watching the changes

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.

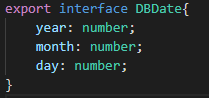
By exemple Harry Potter is a movie title or a character.

This can also interesting with dates.

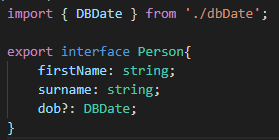
Let’s create a Date model containing the keys ‘year’, ‘month’, ‘day’

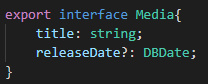
## Adding Date model

We will create first an interface DBDate model:



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





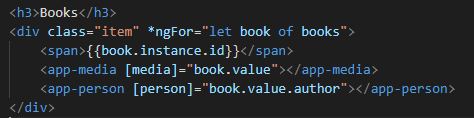
We will then create a MediaComponent:

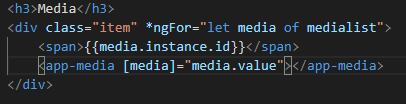


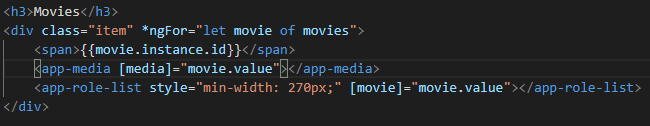
With its template:



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







We will also update the PersonComponent:

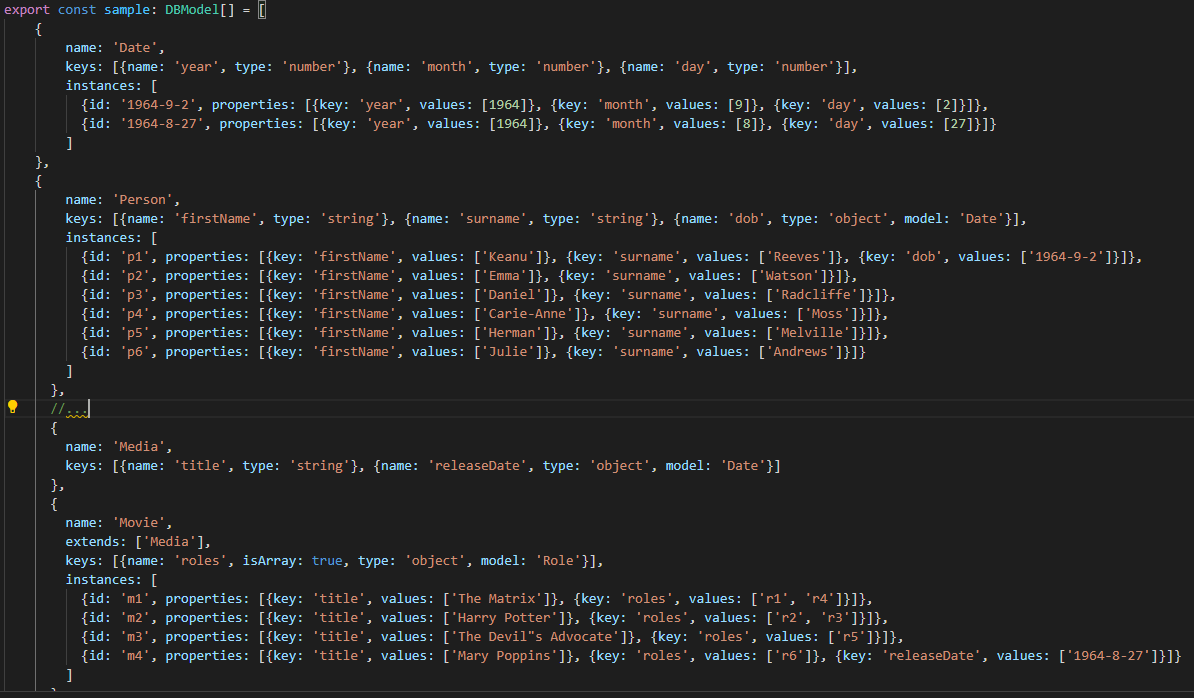


And its template:



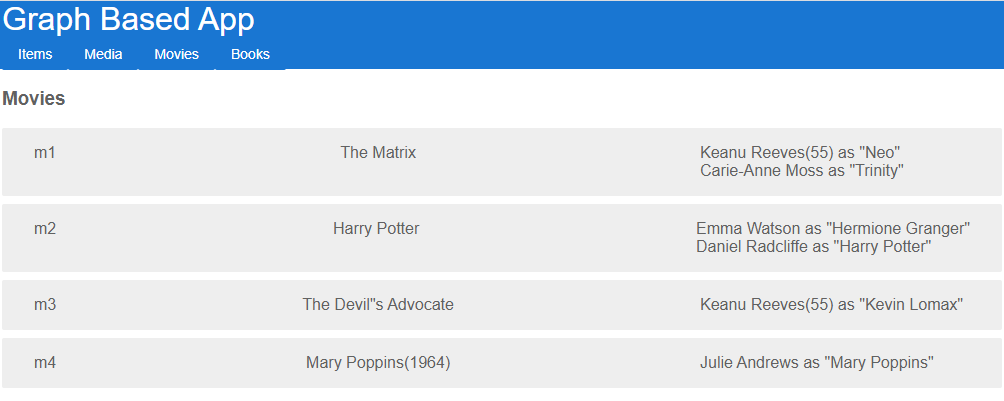
## Updating the sample

The sample is updated with some date instances:

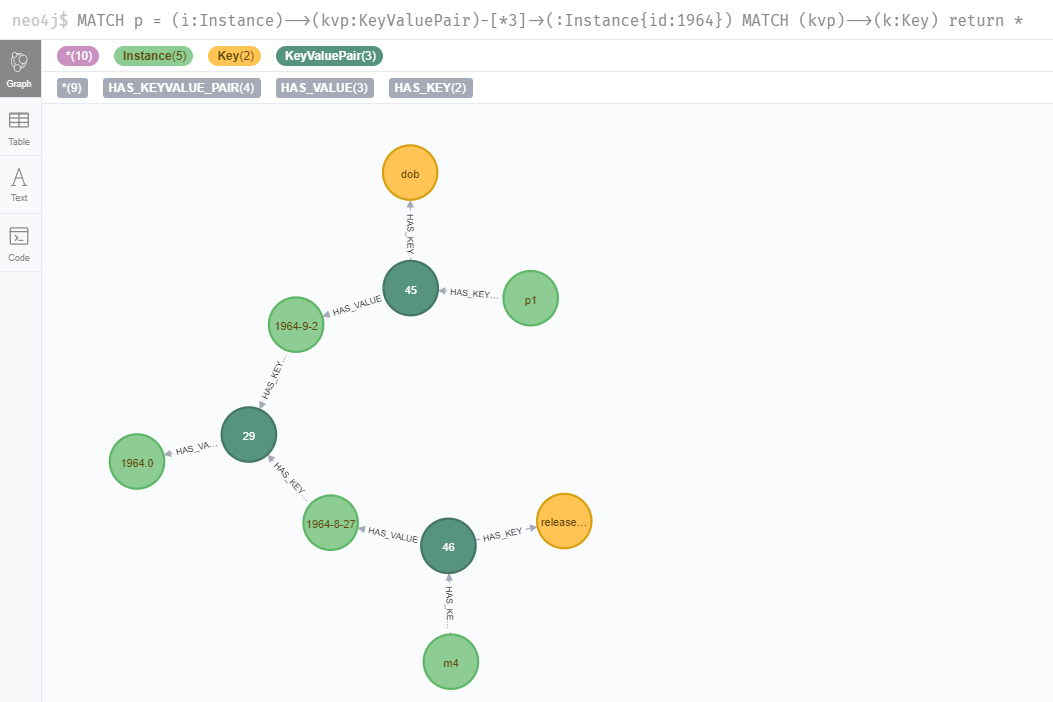


## Watching the changes

And the app is now displaying actor’s age or media release date:



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.