EECS3311 - Software Design
Course Project

York University
Summer 2023

A1 - Rest API and Neo4j

DESCRIPTION

For this assignment, you will implement the backend for a service that computes six degrees of Kevin Bacon. This problem can be restated as finding the shortest path between Kevin Bacon and a given actor (via shared movies). You will use Neo4j as the database management system.

This project is worth 15% of your final mark, and is due on Fri Aug 11.

For this project you may work alone or in groups of up to four people. If you choose to work with partner(s), you are not allowed to withdraw from partnership. Please choose wisely!

OBJECTIVE

1. Explore NoSQL/Graph Database (Neo4j)
2. To create REST API endpoints that are supported by Neo4j graph databases
3. Practice Software Architecture, in particular Server/Client model
4. Learn how to extend functionality of a software project by adding new features.
5. Practice a build system such as maven.

PROJECT/IDE SETUP AND NEW FEATURES

This assignment requires you to have the following software:

- Java 1.8
- Latest version of Neo4j Desktop
- Maven

If you submit an assignment using the wrong Java version or if you change any of the versions specified in the pom.xml, your code may not work and any remark requests will be rejected.

Command Line:

- Install Maven
- To compile your code simply run `mvn compile` in the root directory (the folder that has pom.xml)
- To run your code run `mvn exec:java`

Eclipse:

- File → Import → Maven → Existing Maven Projects
- Navigate to the project location
- Right click the project → Maven Build…
- Input `compile exec:java` in the Goals input box
- You can now run the project by pressing the green play button

Intellij:

- Import project
- Navigate to the project location
- Import the project from external model → Maven
- Next → Next → Next → Finish
- Add Configuration → (+) Button → Maven
- Name the configurations and input `exec:java` in the Command Line box
- Apply → Ok
- You can now run the project by pressing the green play button.

DESCRIPTION OF THE WORK

Please study this handout in full and complete the setup of your project (per the previous page).

If you are working with a partner, please fill in the team contract (a sample is posted on eclass), and submit it ASAP to the separate submission folder.
If you are working alone please skip this step.

STARTER FILES

Do **not** change any code that is already given to you in App.java (other than adding the 2 specified lines). Doing so may result in a grade of 0. Methods under Utils.java also shouldn't be modified (they are there to help you). You can add more methods to Utils.java if you like. Starter code can be downloaded from eClass.

IMPLEMENTING THE API

Please implement the existing API described in this document.

Additionally, you are asked to implement one new feature to the application. An example of a new feature is the following:

> Add a new property to movies (i.e. genre) and add an endpoint that generates a list of movies for a given genre.

Note: **you are NOT permitted to implement the sample feature above**. Please come up with your own feature for your group.

You may want to add a small document to help describe what your new feature is to the TAs. You should specify the types of endpoint(s) (POST, GET, etc.) and create a description of it, following the examples in this handout. You should also describe how the feature would be tested (see below for how we want the other features to be tested).

The document itself won't be marked, but it will help us understand the **new feature which will be marked on correct implementation + testing**.

Testing your app

You also need to write tests for your endpoints.

You can use the Robot framework for writing a test script, as shown in class. A demo script has been posted to eClass.

It is harder to do the unit tests using Java, but you are also welcome to do so.

You must have **2** tests per endpoint, one that tests the **200** response and one that tests a **4XX** response. These test cases will count for a portion of your grade for this assignment.

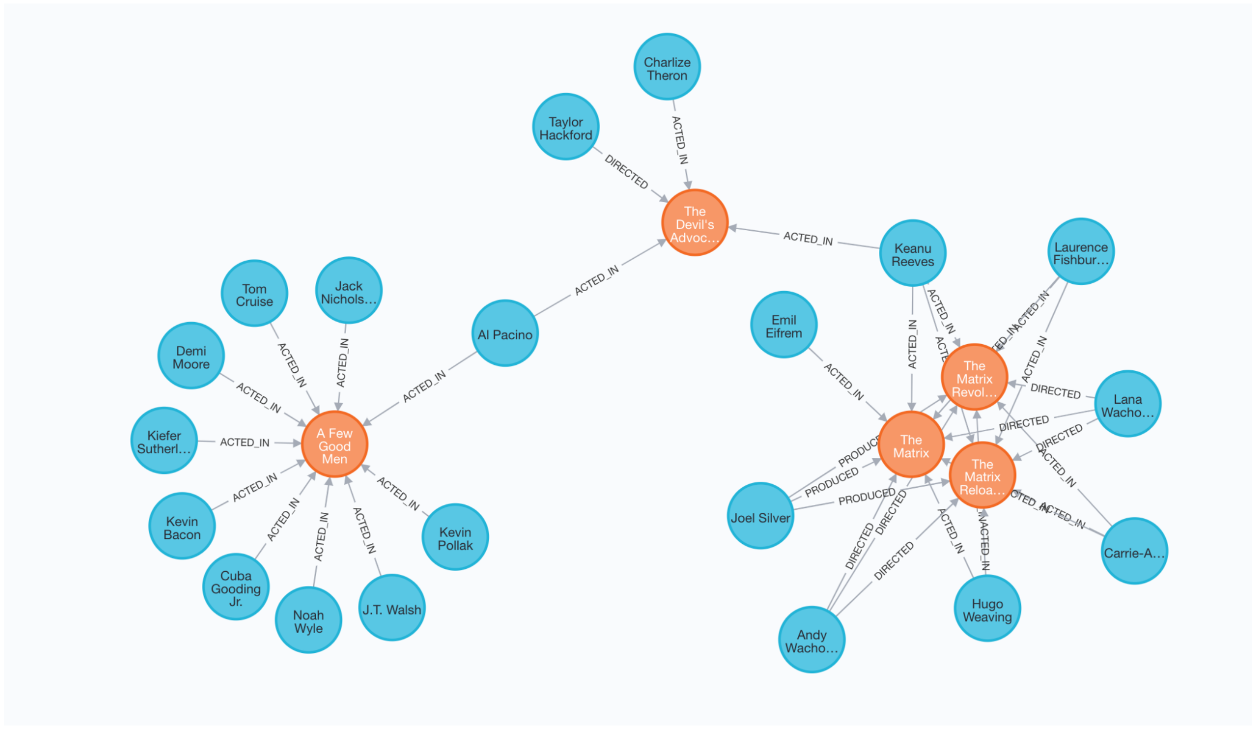
The above number also applies for every new endpoint that you add.
So, a minimum of 16 tests for existing endpoints to implement, and then 2 tests per new endpoint.

Please follow the table on the next page on how to name your tests and what are acceptable statuses to test with each.

NOTE: Failure to follow the naming of these tests exactly may result in you losing marks.

Name of Test	Acceptable Status’
addActorPass	`200`
addActorFail	`400`
addMoviePass	`200`
addMovieFail	`400`
addRelationshipPass	`200`
addRelationshipFail	`400, 404`
getActorPass	`200`
getActorFail	`400, 404`
getMoviePass	`200`
getMovieFail	`400, 404`
hasRelationshipPass	`200`
hasRelationshipFail	`400, 404`
computeBaconNumberPass	`200`
computeBaconNumberFail	`400, 404`
computeBaconPathPass	`200`
computeBaconPathFail	`400, 404`

EXAMPLES



1. Al Pacino has a bacon number of 1. This is because he acted in "A Few Good Men" with Kevin Bacon.
2. Keanu Reeves has a bacon number of 2. This is because he acted in "The Devil's Advocate" with Al Pacino, who acted in a "A Few Good Men" with Kevin Bacon
3. Hugo Weaving has a bacon number of 3. This is because he acted in "The Matrix" trilogy series with Keanu Reeves. Keanu Reeves acted with Al Pacino in "The Devil's Advocate", and Al Pacino acted with Kevin Bacon in "A Few Good Men”
4. You guessed it, KevinBacon himself has bacon number 0.

NOTE: There will be no **“PRODUCED”** or **“DIRECTED”** relationships in your assignment. This figure is just being used as an example.

REQUIREMENTS

Please implement the following REST API

You must first create an Http server and server context in **`App.java`**. There should only be one context that you create, any more will result in mark deductions. You will then implement the endpoints specified below.

Each API Endpoint must return the **`200`** status code if the operation was successful, otherwise the appropriate error status code must be returned (i.e. **`400`**, **`404`**, **`500`**, etc).

Please make sure that all your PUT requests work before starting the GET requests. If your PUT requests aren't completed it may cause you to lose marks for other endpoints that require objects to already be in the database.

All bodies (PUT requests, and responses to PUT and GET requests) should be in JSON Format as shown in this handout.
For GET requests, you need to use query parameters.

PUT Requests

- **PUT /api/v1/addActor**
  - **Description:** This endpoint is to add an actor node into the database.
  - **Body Parameters:**
    - `name: String`
    - `actorId: String`
  - **Body Example**

```
{
      "name": "Denzel Washington",
      "actorId": "nm1001213"
}
```

  - **Expected Response:**
    - **`200 OK`** – For a successful add
    - **`400 BAD REQUEST`** – If the request body is improperly formatted or missing required information
    - **`500 INTERNAL SERVER ERROR`** – If save or add was unsuccessful (Java Exception Thrown)
  - **Edge cases:**
    - If the same actor is added twice, only one should exist for that actor, the **actorId** is unique.
      Example:
      First:

```
{
      "name": "Denzel Washington",
      "actorId": "nm1001213"
}
```

And then:

```
{
      "name": "Robert Downy jr",
      "actorId": "nm1001213"
}
```

In this case, keep only the first node and return a `400` status code.

- **PUT /api/v1/addMovie**
  - **Description:** This endpoint is to add a movie node into the database.
  - **Body Parameters:**
    - `name: String`
    - `movieId: String`
  - **Body Example**

    ```
    {
          "name": "Parasite",
          "movieId": "nm7001453"
    }
    ```
  -
  - **Expected Response:**
    - `200 OK` – For a successful add
    - `400 BAD REQUEST` – If the request body is improperly formatted or missing required information
    - `500 INTERNAL SERVER ERROR` – If save or add was unsuccessful (Java Exception Thrown)
  - **Edge cases:**
    - If the same movie is added twice, only one should exist for that movie, the **movieId** is unique.
      Example:
      First:

```
{
      "name": "Parasite",
      "movieId": "nm1001213"
}
```

And then:

```
{
      "name": "Iron Man",
      "movieId": "nm1001213"
}
```

In this case, keep only the first node and return a `400` status code.

- **PUT /api/v1/addRelationship**
  - **Description:** This endpoint is to add an **ACTED_IN** relationship between an actor and a movie in the database.
  - **Body Parameters:**
    - `actorId: String`
    - `movieId: String`
  - **Body Example**

    ```
    {
          "actorId": "nm1001231",
          "movieId": "nm7001453"
    }
    ```
  -
  - **Expected Response:**
    - **`200 OK`** – For a successful add
    - **`400 BAD REQUEST`** – If the request body is improperly formatted or missing required information
    - **`404 NOT FOUND`** – If the actor or movie does not exist when adding the relationship.
    - **`500 INTERNAL SERVER ERROR`** – If save or add was unsuccessful (Java Exception Thrown)
  - **Edge cases:**
    - If the same relationship is added twice, only one relationship should exist between the actor and movie.

      Example:

      First:

```
{
      "movieId": "nm7001453",
      "actorId": "nm1001231"
}
```

And then:

```
{
      "movieId": "nm7001453",
      "actorId": "nm1001231"
}
```

In this case, keep only the existing relationship and return a `400` status code.

GET Requests

- **GET /api/v1/getActor**
  - **Description:** This endpoint is to check if an actor exists in the database.
  - **Query Parameters:**
    - `actorId: String`
  - **URL Example:**

    ```
    http://localhost:8080/api/v1/getActor?actorId=nm1001231
    ```

  - **Response:**
    - `actorId: String`
    - `name: String`
    - `movies: List of Strings`
  - **Response Body Example:**

```
{
      "actorId": "nm1001231",
      "name": "Ramy Youssef",
      "movies": [
            "nm8911231",
            "nm1991341",
            "nm2005431",
            …
      ]
}
```

- **Expected Response:**
    - `200 OK` – For a successful add
    - `400 BAD REQUEST` – If the request body is improperly formatted or missing required information
    - `404 NOT FOUND` – If there is no actor in the database that exists with that actorId.
    - `500 INTERNAL SERVER ERROR` – If save or add was unsuccessful (Java Exception Thrown)
- **Edge cases:**
    - If the actor exists but didn't act in any movies, return an empty list in “movies” inside the response
      Example:

```
{
      "actorId": "nm1001231",
      "name": "Ramy Youssef",
      "movies": []
}
```

- **GET /api/v1/getMovie**
    - **Description:** This endpoint is to check if a movie exists in the database.
    - **Query Parameters:**
        - `movieId: String`
    - **URL Example:**

```
http://localhost:8080/api/v1/getActor?movieId=nm1111891
```

- **Response:**
    - `movieId: String`
    - `name: String`
    - `actors: List of Strings`
- **Response Body Example:**

```
{
      "movieId": "nm1111891",
      "name": "Groundhog Day",
      "actors": [
            "nm8911231",
            "nm1991341",
            "nm2005431",
            …
      ]
}
```

- **Expected Response:**
  - `200 OK` – For a successful add
  - `400 BAD REQUEST` – If the request body is improperly formatted or missing required information
  - `404 NOT FOUND` – If there is no movie in the database that exists with that movieId.
  - `500 INTERNAL SERVER ERROR` – If save or add was unsuccessful (Java Exception Thrown)
- **Edge cases:**
  - If the movie exists but no one has acted in it, return an empty list in “actors” inside the response

    Example:

```
{
      "actorId": "nm1331231",
      "name": "Our Planet",
      "actors": []
}
```

- **GET /api/v1/hasRelationship**
  - **Description:** This endpoint is to check if there exists a relationship between an actor and a movie.
  - **Query Parameters:**
    - `movieId: String`
    - `actorId: String`
  - **URL Example:**

```
http://localhost:8080/api/v1/hasRelationship?actorId=nm1001231&movieId=nm1251671
```

  - **Response:**
    - `movieId: String`
    - `actorId: String`
    - `hasRelationship: Boolean`
  - **`Response Body Example:`**

```
{
      "actorId": "nm1001231",
      "movieId": "nm1251671",
      "hasRelationship": true
}
```

  -

    - **Expected Response:**
        - `200 OK` – For a successful add
        - `400 BAD REQUEST` – If the request body is improperly formatted or missing required information
        - `404 NOT FOUND` – If there is no movie or actor in the database that exists with that actorId/movieId.
        - `500 INTERNAL SERVER ERROR` – If save or add was unsuccessful (Java Exception Thrown)

- **GET /api/v1/computeBaconNumber**
    - **Description:** This endpoint is to check the bacon number of an actor. **Note** that Kevin Bacon has a BaconNumber of 0.
        - **IMPORTANT!** Kevin Bacon always has an `actorId` of `nm0000102`
    - **Query Parameters:**
        - `actorId: String`
    - **URL Example:**

```
http://localhost:8080/api/v1/computeBaconNumber?actorId=nm1001231
```

    - **Response:**
        - `baconNumber: int`
    - **`Response Body Example:`**

```
{
      "baconNumber": 3
}
```

    - **Expected Response:**
        - `200 OK` – For a successful computation
        - `400 BAD REQUEST` – If the request body is improperly formatted or missing required information
        - `404 NOT FOUND` – If there is no movie or actor in the database that exists with that actorId/movieId or there is no path to Kevin Bacon.
        - `500 INTERNAL SERVER ERROR` – If save or add was unsuccessful (Java Exception Thrown)

- **GET /api/v1/computeBaconPath**
  - **Description:** This endpoint returns the **shortest** Bacon Path in order from the actor given to Kevin Bacon.
    - **IMPORTANT!** Kevin Bacon always has an `actorId` of `nm0000102`
  - **Query Parameters:**
    - `actorId: String`
  - **URL Example:**

```
http://localhost:8080/api/v1/computeBaconPath?actorId=nm1991271
```

  - **Response:**
    - baconPath: List of interchanging actors and movies beginning with the inputted `actorId` and ending with Kevin Bacon's `actorId`.
      - `actorId: String`
      - `movieId: String`
      - `...`
  - **`Response Body Example:`**

```
{
      "baconPath": [
            "nm1991271",
            "nm9112231",
            "nm9191136",
            "nm9894331",
            "nm0000102"
      ]
}
```

  - **Expected Response:**
    - **`200 OK`** – For successfully finding a path
    - **`400 BAD REQUEST`** – If the request body is improperly formatted or missing required information
    - **`404 NOT FOUND`** – If there is no movie or actor in the database that exists with that actorId/movieId, or no path exists between actor and Kevin Bacon.
    - **`500 INTERNAL SERVER ERROR`** – If save or add was unsuccessful (Java Exception Thrown)

- **Edge Cases**
  - If an actor acted in a movie but does not have a path to Kevin Bacon, then return a status of 404 and nothing else
  - If there is more than one baconPath with the same baconNumber, just return one of the baconPaths.
  - If you want to compute the baconPath of Kevin Bacon himself, you should return a list with just his actorId in it.

Note: The bacon path is a list (in order) of the connection that leads from the actor to Kevin Bacon (i.e. Kevin Bacon's ID is always the last one in the list). Make sure that Kevin Bacon is in the path returned in the response body.

Hint: for a JSON object, you can have a value of type JSONArray. JSONArrays are sorted (like an ArrayList).
JSON objects themselves are not sorted alphabetically by key names (like a Java Map or Set)

EXTRA PARAMETERS?

Remember to filter out any extra parameters that shouldn't be in a payload as soon as you have parsed it (or even before).

For example:

```
{
     "actorId": "nm1001231",
     "name": "Clint Eastwood",
     "extraStuff": "Good luck on your assignment!"
}
```

If it is possible to understand the request and act safely and accordingly on it. You don't need to send a `4XX` status code.

DATABASE REQUIREMENTS

You are required to use Neo4j for this assignment. The username and password for an instance of the database:

Username: neo4j

Password: 12345678

Local DBMS version needs to be 4.4.23 or lower in order to work with the application. Version 5+ does NOT work as the Neo4j driver for Java that we use was discontinued.

The port specified in your Java files for the DB should be 7687

Please note, **you are NOT submitting your database**; rather, your Java code creates the necessary entries through Cypher queries, written by you.

NODE REQUIREMENTS

- Actor
  - Must have the node label **`actor`**
  - Must have the following properties
    - **`id`**
    - **name**
- Movie
  - Must have the node label **`movie`**
  - Must have the following properties
    - **`id`**
    - **name**

RELATIONSHIP REQUIREMENTS

- Acted In
  - Must have the relationship label **`ACTED_IN`**

Allowed imports are:

- whatever is already in App.java
- com.sun.net.httpserver.*
- org.neo4j.driver.v1.*
- org.json.*
- java.net.* (including java.net.HttpUrlConnection which some of you might use to do unit testing within Java)
- java.util.*
- java.io.*
- java.lang.* (perhaps you needed some Math functions to make your proposed features work)

Code Style/Documentation

You will be required to use appropriate variable naming & file naming conventions throughout the whole assignment. You will also be required to comment your function signatures appropriately.

TESTING YOUR ENDPOINTS MANUALLY

Run your services. See *Environment Setup* for how to run each service

CLI

From the command line the best way to test your endpoints is using curl.

```
curl -X POST http://localhost:PORT/routeNameHere/ --data \
     '{ "key": "value", "other": "thing" }'
```

- The -X flag specifies the REST action you wish to use
- The --data flag specifies the body of the request if one is required

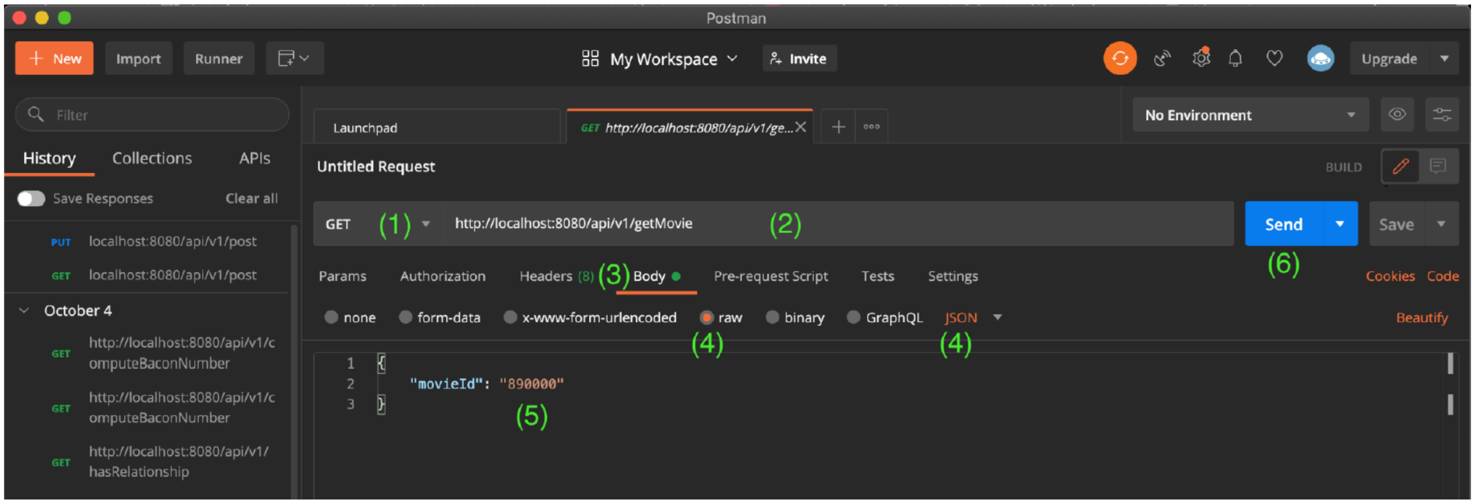
Postman

Postman is a very helpful tool to use to test your endpoints for basic behaviour.

This tool is highly recommended for beginners working with REST API testing.

Sending a Request and Reading a Response

- Open up Postman and select the type of request you would like to send (1)
- Enter your request URL (2)
- Open up the request body tab (3) and select raw, then select JSON on the drop-down to the right (4) if you are sending body parameters
- Your response data will be shown in the body area (5) when you click the Send button



```
{
    "movieId": "890000"
}
```

Useful Videos 🙂

If you've made it this far in the assignment handout, here are some good videos for you to watch that will help you with the assignment!

- Neo4j Cypher Tutorial

HOW TO SUBMIT YOUR WORK AND WHAT TO SUBMIT

1. Your work will be submitted through eclass.
2. Please submit a zipped copy of your complete project folder.
3. Remember, if you work in groups, only one person submits on behalf of all members.

Good luck! 💯💯

Submission checklist:

- App.java
- All Java files for your classes to support the application
- pom.xml
- Your Robot testing file (please place in the root folder of submission)
- If you need to add any notes to the marking TA, please put any txt or document files in the root folder of your submission for visibility.

It is best to just zip your whole project (after adding the Robot file), then submit that.