CSC301 Assignment 1

1. Objective

- Coding and teamwork warm-up for the semester
- Reintroducing git
- Explore new technologies (NoSQL/graph database)
- Exposure to new ways of programming (separate backend with REST api)

2. How to submit your work

Your work will be submitted through Github Classrooms

- Ensure your final draft is on your master branch before the assignment deadline.
- Add a team.md under your main branch (your main branch should have the pom.xml, src folder, tsv files and your added team.md)

    RepoName/Team.md example:
    Utorid_1, first_name, last_name
    Utorid_2, first_name, last_name
    …

Note: You should always test your code using the command line and check for errors. Please make sure that it returns no errors when running it on the command line.

Specifically, go to your project folder and run **`mvn compile`** (the project folder that has your pom.xml), run **`mvn exec:java`** and attempt to hit the endpoints.

3. 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).

4. Examples

4.1 George Clooney

Consider the actor George Clooney:

- George Clooney acted in “Good Night, and Good Luck” with Patricia Clarkson
- Patricia Clarkson acted in “Beyond All Boundaries” Kevin Bacon

So we would say that George Clooney has a “Bacon number” of 2.

4.2 Bud Collyer

Consider the actor Bud Collyer:

- Bud Collyer was in “Secret Agent” with Jackson Beck
- Jackson Beck was in “Hysterical History” with Mae Questel
- Mae Questel was in “Who Framed Roger Rabbit” with Frank Welker
- Frank Welker was in “Balto” with Kevin Bacon

So we would say that Bud Collyer has a “Bacon number” of 4.

5. Starter Files

You have been given three starter files. Each file is tab delimited and has the following column titles.

- actors.tsv: List of actors and some attributes

`nconst(actorId) primaryName(name) birthYear deathYear primaryProfession knownForTitle`

- movies.tsv: List of movies and some attributes

`tconst(movieId) titleType primaryTitle(name) originalTitle isAdult startYear endYear`

`runtimeMinutes genres`

- relationships.tsv: List of movies and their actors

`tconst ordering nconst category job characters`

6. Project/IDE Setup

6.1 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`**

6.2 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
- Click apply then run/close
- You can now run the project by pressing the green play button

6.3 Intellij

- Import project
- Navigate to the project location
- Import project from external model → Maven
- Next → Next → Next → Finish
- Add Configuration → + Button → Maven

- Name the configuration and input **`exec:java`** in the Command Line box
- Apply → Ok
- You can now run the project by pressing the green play button

7. API Requirements

You are required to implement the following REST API endpoints. For those unfamiliar with REST a detailed description can be found here (https://www.codecademy.com/articles/what-is-rest).

Your API must run on port 8080 don't change this part of the starter code.

7.1 PUT /api/v1/addActor

Add an actor node to the database.

Body Parameter Types:

```
name : string
actorId: string
```

Body:

```
{
    "name": "Emma Stone",
    "actorId": "9348"
}
```

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 is thrown)

Note: If the same actor is added twice, only one node should exist for that actor, each actor should have a unique actorId.

For example:

Body:

```
{
    "name": "Emma Stone",
    "actorId": "9348"
}
```

And then

```
{
    "name": "Emma Watson",
    "actorId": "9348"
}
```

In this case, you should keep the existing node and return a “400” status code.

7.2 PUT /api/v1/addMovie

Add a movie node to the database.

Body Parameter Types:

```
name : string
moveId: string
```

Body:

```
{
    "name": "Graduation",
    "movieId": "742053"
}
```

Response: (no response body)

- `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 is thrown)

Note: If the same movie is added twice, only one node should exist for that movie, each movie should have a unique movieId.

For example:

Body:

```
{
    "name": "Graduation",
    "movieId": "742053"
}
```

And then

```
{
    "name": "Jumanji",
    "actorId": "742053"
}
```

In this case, you should keep the existing node and return a “400” status code.

7.3 PUT /api/v1/addRelationship

Add a relationship between a movie node and an actor node. We will assume that each movie is acted by at least one actor.

Body Parameter Types:

```
actorId : string
movieId: string
```

Body:

```
{
    "actorId": "9348",
    "movieId": "742053"
}
```

Response: (no response body)

- `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 actors or movies do not exist when adding a relationship
- `500 INTERNAL SERVER ERROR` if save or add was unsuccessful (Java Exception is thrown)

Note: If the same relationship is added twice, only one relationship should exist for that actor and movie.

For example:

Body:

```
{
    "actorId": "9348",
    "movieId": "742053"
}
```

And then

Body:

```
{
    "actorId": "9348",
    "movieId": "742053"
}
```

In this case, you should keep the existing node and return a “400” status code.

7.4 GET /api/v1/getActor

Get actor id, name, and all the movies they have acted in. We should assume Kevin Bacon is in the database.

Body Parameter Types:

```
actorId : string
```

Response Body Parameter Types:

```
actorId: string
name: string
movies: list of strings
```

Body:

```
{
    "actorId": "9348"
}
```

Response:

- `200 OK` if the actor exists
- `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 is thrown)

Response Body:

```
{
    "actorId": "9348",
    "name": "Emma Stone",
    "movies": [
         "838362",
         "56789765",
         "45678",
         ...
    ]
}
```

Note: If the actor exists but didn't act in any movies, return an empty list in "movies" inside the response body.

For example:

```
{
    "actorId": "9000",
    "name": "Andrew Garfield",
    "movies": []
}
```

7.5 GET /api/v1/getMovie

Get movie id, name, and all actors that acted in it.

Body Parameter Types:

```
movieId: string
```

Response Body Parameter Types:

```
movieId: string
name: string
actors: list of strings
```

Body:

```
{
   "movieId": "284764"
}
```

- `200 OK` if the movie exists
- `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 with the requested movieId
- `500 INTERNAL SERVER ERROR` if save or add was unsuccessful (Java Exception is thrown)

Response Body:

```
{
    "movieId": "284764",
    "name": "Holes",
    "actors": [
         "838334",
         "567823",
         "41394",
         ...
    ]
}
```

7.6 GET /api/v1/hasRelationship

Return the boolean value of if an actor has acted in the movie. true if they have, false if they haven't.

Body Parameter Types:

```
actorId: string
movieId: string
```

Response Body Parameter Types:

```
actorId: string
movieId: string
hasRelationship: boolean
```

Body:

```
{
   "actorId": "93847",
   "movieId": "284764"
}
```

Response:

- `200 OK` if both the actor and movie exist
- `400 BAD REQUEST` if the request body is improperly formatted or missing required information
- `404 BAD REQUEST` if the actorId or movieId doesn't exist in the database
- `500 INTERNAL SERVER ERROR` if save or add was unsuccessful (Java Exception is thrown)

Response Body:

```
{
    "actorId": "93847",
    "movieId": "284764",
    "hasRelationship": true
}
```

7.7 GET /api/v1/computeBaconNumber

Returns Bacon Number of given actor. If you compute the BaconNumber of Kevin Bacon himself, you should return 0.

Body Parameter Type:

```
actorId: string
```

Response Body Parameter Type:

```
baconNumber: string
```

Request Body:

```
{
    "actorId": "9348"
}
```

Response:

- `200 OK` if a path is found
- `400 BAD REQUEST` if the request body is improperly formatted or missing required information or if the actor does not exist
- `404 NOT FOUND` if a path cannot be found
- `500 INTERNAL SERVER ERROR` if save or add was unsuccessful (Java Exception is thrown)

Response Body:

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

7.8 GET /api/v1/computeBaconPath

Kevin Bacon's actorId is nm0000102 from the actor.tsv file. Please use that when computing the BaconPath.

Returns the **shortest** Bacon Path in order from actor given to Kevin Bacon.

If an actor acted in a movie but does not have a path to Kevin Bacon, then return an empty list in “baconPath” inside the response body.

If there is more than 1 baconPath with the same baconNumber, then return a random baconPath.

If you want to compute the Bacon Path of Kevin Bacon himself, you should return 0 as the baconNumber and 1 movie of his at random.

Body Parameter Type:

```
actorId: string
```

Response Body Parameter Type:

```
baconNumber: string
baconPath: list of objects with actorId: string and movieId: string
```

Request Body:

```
{
    "actorId": "9348"
}
```

Response:

- `200 OK` for a successful add
- `400 BAD REQUEST` if the request body is improperly formatted or missing required information or if the actor does not exist
- `404 NOT FOUND` if a path cannot be found
- `500 INTERNAL SERVER ERROR` if save or add was unsuccessful (Java Exception is thrown)

Response Body:

```
{
    "baconNumber": "5"
    "baconPath":[
            {
                "actorId": "9348",
                "movieId": "4079"
            },
            {
                "actorId": "5320",
```

```
                "movieId": "4079"
            },
                ...
            {
                "actorId": "nm0000102",
                "movieId": "4079"
            }

    ]
}
```

Note: The `baconPath` is a list (in order) of the connections that lead from the actor to Kevin Bacon. Make sure that Kevin Bacon is in the path returned in the response body.

7.9 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:

```
{
      "name": "Emma Stone",
      "actorId": "9348"
      "extraData": "sneaked into this body"
}
```

However, if you can understand the request and act on it safely and accordingly, you don't need to send a 4xx status code.

8. Database Requirements

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

Username: neo4j

Password: 1234

8.1 Node Requirements

8.1.1 Actor

- must have the node label `actor`
- must have the following properties
    - `id`
    - `Name`

8.1.2 Movie

- must have the node label `movie`
- must have the following properties
    - `id`
    - `Name`

8.2 Relationship Requirements

8.2.1 Acted In

- must have the relationship label `ACTED_IN`

9. Testing Your Code

9.1 Run Your Code

Refer to Section 5 to run your code in your IDE of preference.

Once you've run your code, you'll have a local url (http://localhost:8080). Your endpoints will be relative paths of this url. You can assume that you are testing with existing endpoints.

Ex. http://localhost:8080/api/v1/addActor

9.2 Hitting your Endpoints

9.2.1 CLI

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

```
curl -X POST http://localhost:8080/api/v1/endpoint/ --data \
     '{ "key": "value", "other": "thing" }'
```

- The `-X` flag specifies the REST action you wish to use.
- And the `--data` flag specifies the body of the request

9.2.2 Python

You can use python to programatically call your endpoints if you so desire.

```
import requests
d = { "key": "value" }
requests.put('http://localhost:8080/api/v1/endpoint', json=d)
```

- The dictionary is the json body you want to send.
- There are also methods for `requests.get(...)` and `requests.post(...)`.

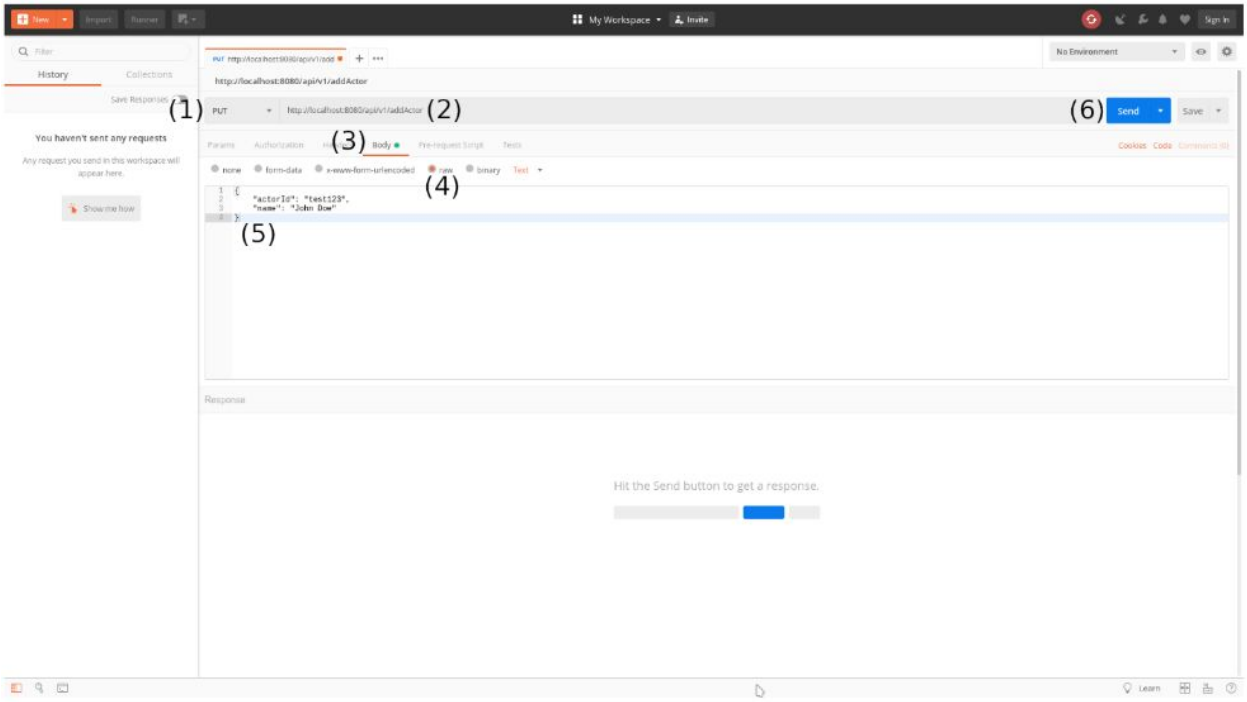
9.2.3 Postman

Postman is a nice tool to use to test your endpoints. It can be downloaded from:

https://www.getpostman.com/products

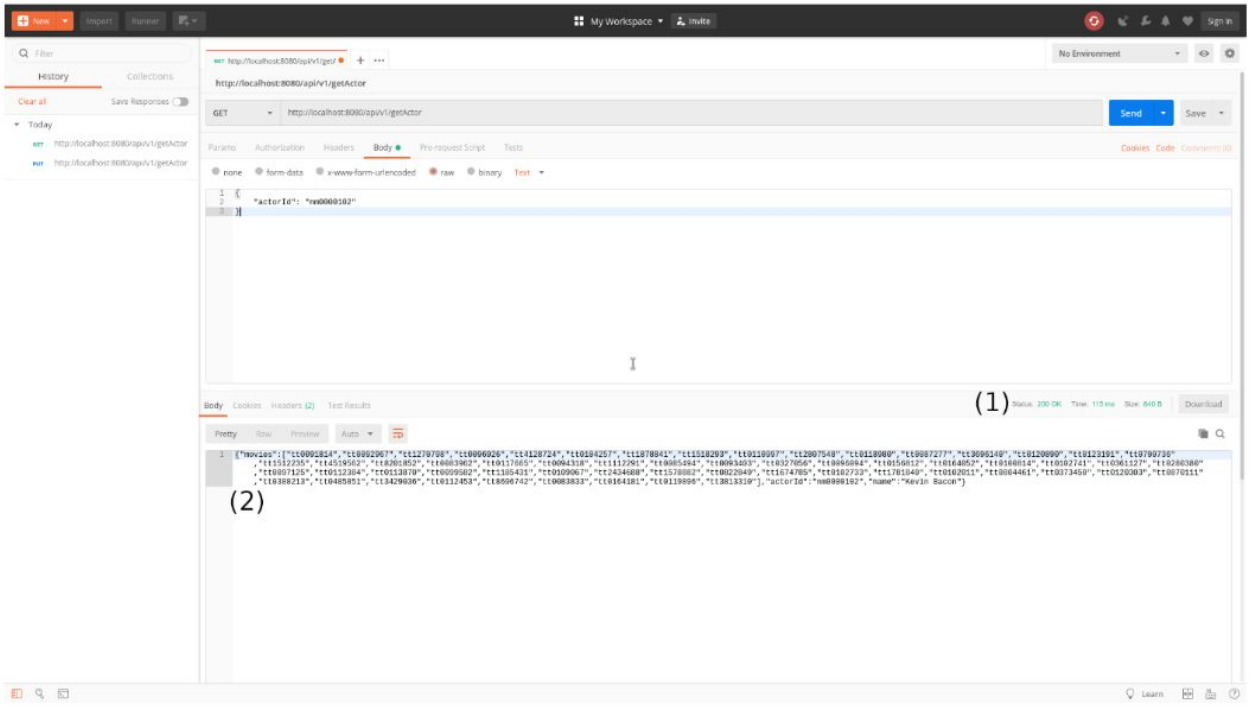
Sending a Request

- Open up Postman and select the type of request you would like to send (1)
- Enter in your request url (2)
- Open up the request body tab (3) and select **raw**(4)
- Enter in your request body data into the test box (5)
- Click **Send** (6)



Reading the Response

- You can see the status of your response (1)
- You can see the response body (2)
- Ensure that responses that should not have a body, do not have a body


http://localhost:8080/api/v1/getActor
GET
http://localhost:8080/api/v1/getActor
Send
Save
Body
raw
{
"actorId": "nm0000102"
}
(1)
(2)