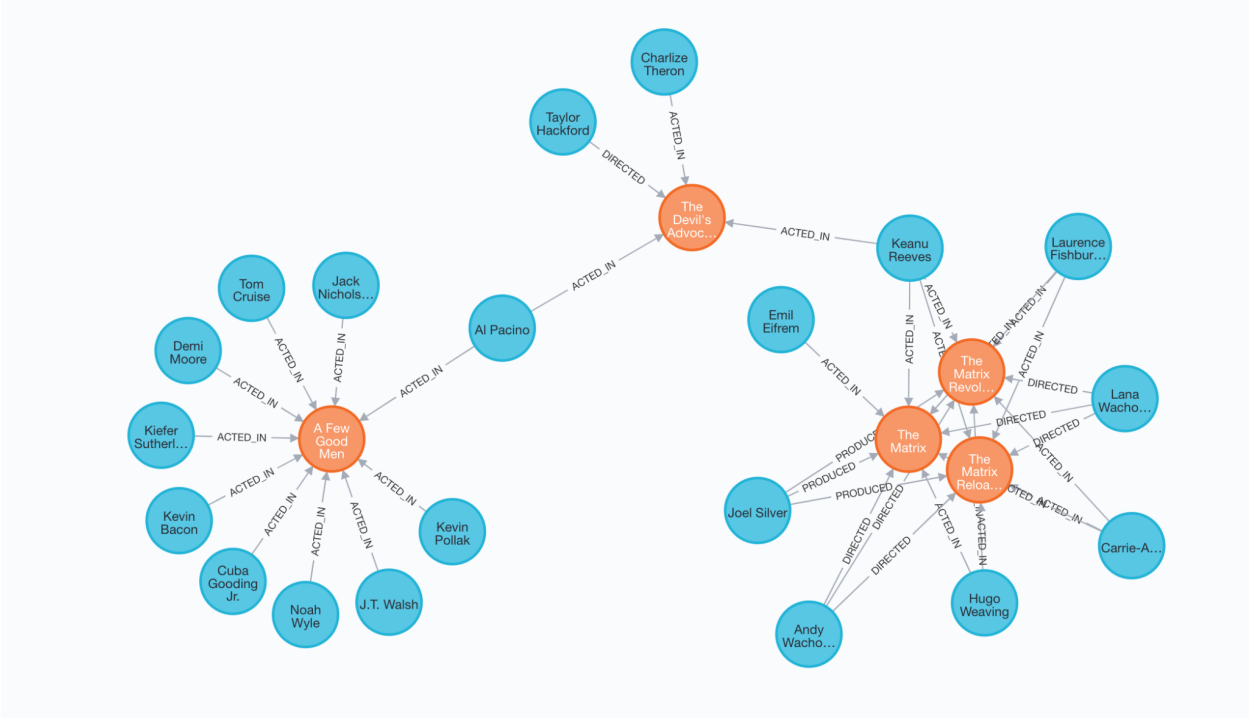
A1 - Six Degrees of Kevin Bacon

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.

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"

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

Objectives

1. Explore NOSQL/Graph Database (Neo4j)
2. To create REST API endpoints that are supported by Neo4j graph databases
3. To utilise Git and Git Flow
4. Practice Enterprise Design Patterns, specifically dependency injection using Dagger 2
5. To learn and apply testing for CI/CD
6. Learn how to use Docker

Project and IDE Setup

This assignment requires you to have the following software:

- Intellij (you may use other IDEs but we recommend Intellij)
- Java: JDK/JRE Version 16.0.1
- Maven version 3.6.3
- Neo4j Desktop
- Docker Desktop

This video is a guide for setting up A1 in case you haven't already installed some of the above software. Make sure to check the description for some corrections.

If you submit an assignment using any of the wrong versions, your code may not work and any remark requests will be rejected.

Submission Information

1. A1 will be submitted through GitHub classrooms, the link can be found on the website
2. You may work alone or in teams of two
3. Make sure your final draft is on your main/master branch before the assignment deadline
4. Make sure your repos file/folder structure is unchanged
5. Make sure your whole assignment runs fully and correctly in Docker
6. Make sure team.md (provided to you in the starter code) is filled out

Starter Files

Do **not** change any code that is already given to you in the starter files (including the pom.xml, dockerfile, docker-compose file, etc) and do not change the file/folder structure of the starter code. We have provided comments in every place where you need to add code. You may also add extra files as you see fit. You may also include more import statements in any of the files where you're writing your code.

Note for M1 Mac Users

If you're using an M1 Mac and your Docker containers aren't working properly, you could try to uncomment the line '`#platform: linux/amd64`' in the docker-compose file and restart your docker containers which might fix the problem.

Docker

You will be required to make sure your whole assignment runs correctly in Docker since all our testing will be done in Docker. We have provided the Dockerfile and docker-compose file that are necessary to run the assignment in Docker. You **do not** need to change these files for this assignment, you are only required to make sure your assignment runs in Docker.

Data Access Object Pattern (DAO)

You will be following the Data Access Object (DAO) Pattern for this assignment as well. This means all interactions with the database must be from a single object. This object class is given to you and you must populate the class with the database methods that you will. If you make any database transactions without using the DAO, you may lose marks.

Neo4j

The username and password for the database are given in the docker-compose file in the starter code which you will use to connect to your database. Note that your server and database must run correctly in Docker so you will not be using localhost as the hostname when you connect to Neo4j. Try to think about why localhost as the hostname in the connection URL wouldn't work to connect from one Docker container to another. In Docker, the hostname you have to use to connect from one container to another is the name of the container you are trying to connect to. Also make sure to use the bolt protocol when connecting to Neo4j.

The .env File

We have provided you a .env file for convenience. You don't have to change this file at all and you don't have to use environment variables if you choose not to. This .env file (if you choose to use it in your code) along with the NEO4J_ADDR environment variable set in docker-compose file under the a1 container, will allow you to connect to a local Neo4j instance or the Docker Neo4j instance depending on which environment you run your code. Meaning if you run your server outside of Docker, the NEO4J_ADDR environment variable will be localhost, allowing you to connect to a local Neo4j Instance and if you run your server in Docker the NEO4J_ADDR will be neo4j allowing you to connect to the neo4j Docker container. This will be convenient for you because you won't have to change the hostname everytime you switch between running the code on your local machine to running the code in Docker. We've included the Java Dotenv dependency in the pom.xml that you can use if you do choose to use the .env file.

Dependency Injection / Dagger

We will be using Dagger 2 constructor injection for this assignment. There are three classes that will need to be injected:

- `ReqHandler`
- `Server`
- `Neo4jDAO`

The dependencies that need to be injected in the constructor of the classes above are as follows:

- A `HttpServer` object should be injected into the constructor of `Server`. The `HttpServer` should be run on localhost port 8080
- A `Driver` (`org.neo4j.driver`) object should be injected into the constructor of `Neo4jDAO`
- A `Neo4jDAO` object should be injected into the constructor of `ReqHandler` (This DAO should then be passed to the specific endpoint handlers in ReqHandler)

Note that you may choose to inject more objects along with the dependencies mentioned above into the constructors of the given classes. You may also add more injections (constructor injections or non-constructor injections) anywhere if you so choose, but make sure the dependency injection requirements above are fulfilled.

You are not allowed to create any more dagger components or dagger modules than the ones given to you.

The method names inside your dagger modules must follow this convention: The method name starts with 'provide' and ends with the object being returned from that method

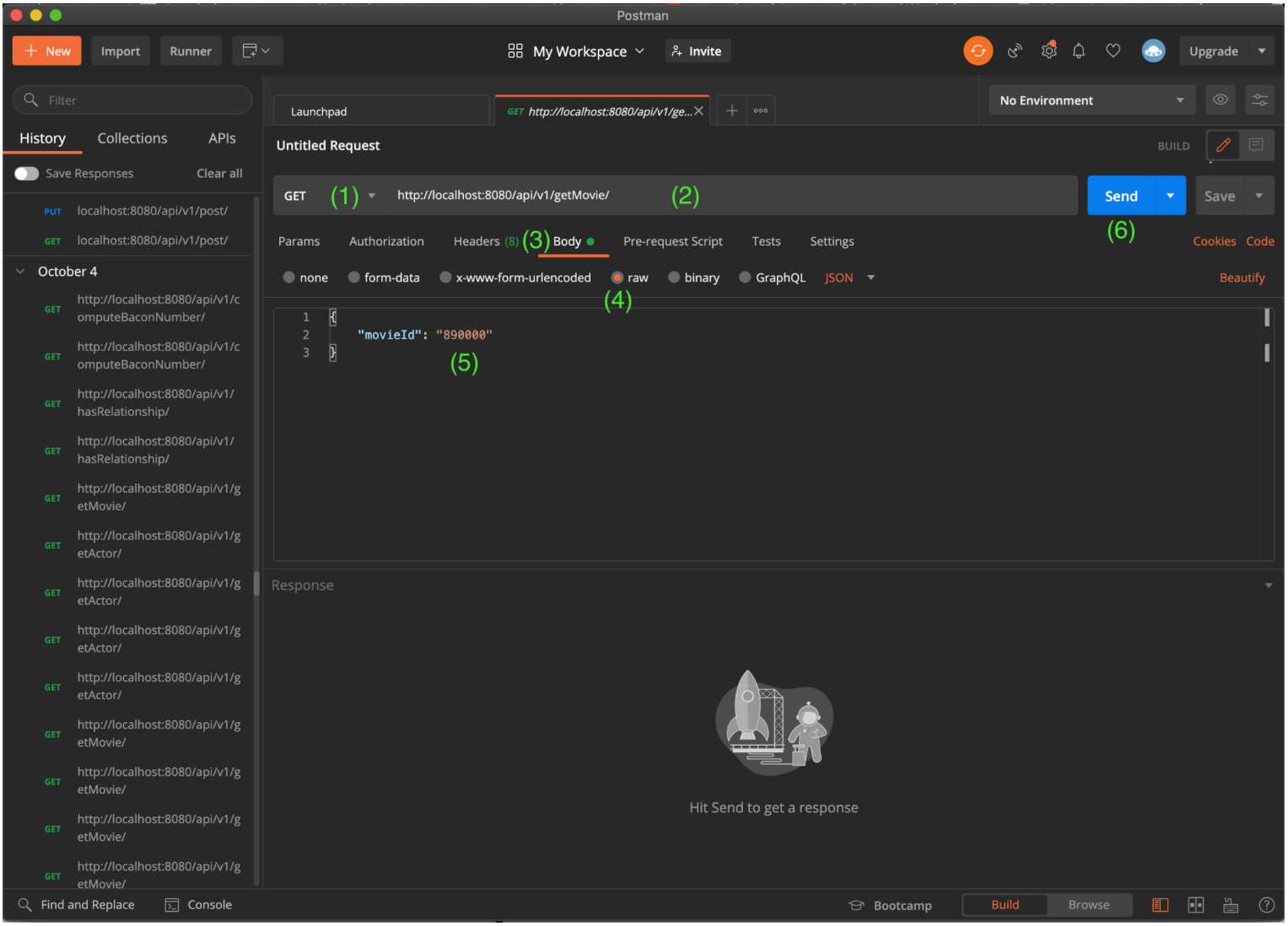
e.g. if you're writing a provides method that provides a `String` object, then the full method name should be '`provideString`'. Note the capitalisation is also copied exactly from the class name of the object being returned so since the `String` class name has an 'S' capitalised the method name must also have the capitalised 'S'.

Any and all methods that you make in the modules must have the `public` access level modifier.

These restrictions are so that we can test your code. Not following any of the guidelines above may result in major mark deductions.

Testing Your Code With Postman

Postman is a helpful tool to test your endpoints. Download Postman then create your workspace to get started. Then (1) select the type of request you want to send. Then (2) enter your request URL. Then (3) click the request body tab and (4) select raw and JSON on the dropdown. Then (5) enter your body parameters and (6) hit send.



Git Usage

You must use Git flow for this assignment and you will be evaluated on correct usage of Git flow. This means you will use a development branch along with feature branches to implement features. You will have to decide what you consider to be a feature. You must also use pull requests any time you merge branches.

Continuous Integration / Continuous Delivery Testing

For the CI/CD testing component of this assignment, your job will be to write tests that test your REST API endpoints. These tests will go in **src/test/... /AppTest.java.** You will write two tests for each endpoint, one to test the OK response (200) and one to test a failing response (400, 404 or 500). The test API requests must originate from outside of Docker and go to your server running inside Docker, meaning you will run your tests from outside the Docker containers to test your server running inside the Docker container. Your tests only have to check the response code and response body and you don't have to test if the database is updated correctly. The tests are marked for completion only, if you haven't implemented all the endpoints and some tests fail, you won't lose marks as long as your tests are written correctly.

API Requirements

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

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.

Also 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!"
}
```

Here, it's possible to understand the request and act safely and accordingly on it so you don't need to send a failing status code for this example.

PUT Requests

- **PUT /api/v1/addActor**
  - **Description:** 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 an actor with the same actorId already exists in the database, keep the actor that was originally in the database and return a `400` status code. This behaviour applies regardless of whether the names are equal or not.
    - Two actors in the database can have the same name as long as their actorIds are different.
    - You don't have to handle cases where name/actorId are empty strings.
    - You don't have to worry about the format of the Ids.

- **PUT /api/v1/addMovie**
  - **Description:** 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 a movie with the same movieId already exists in the database, keep the movie that was originally in the database and return a `400` status code. This behaviour applies regardless of whether the names are equal or not.
    - Two movies in the database can have the same name as long as their movieIds are different.
    - You don't have to handle cases where name/movieId are empty strings.
    - You don't have to worry about the format of the Ids.

- **PUT /api/v1/addRelationship**
  - **Description:** Add an **ACTED_IN** relationship from an actor to a movie.
  - **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 relationship already exists in the database, return `400` status code.
    - You don't have to worry about the format of the Ids.
    - You don't have to handle the case where Ids are empty strings.

GET Requests

- **GET /api/v1/getActor**
  - **Description:** Get an actor's information from the database
  - **Body Parameters:**
    - `actorId: String`
  - **Body Example:**

```
{
     "actorId": "nm1001231"
}
```

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

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

  - **Expected Response:**
    - `200 OK` - For a successful get
    - `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 the given 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 for "movies" inside the response.
    - You don't have to worry about the format of the Ids.
    - You don't have to handle the case where actorId is an empty string.

- **GET /api/v1/getMovie**
  - **Description:** Get a movie's information from the database.
  - **Body Parameters:**
    - `movieId: String`
  - **Body Example:**

```
{
     "movieId": "nm1111891"
}
```

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

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

  - **Expected Response:**
    - `200 OK` - For a successful get
    - `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 the given 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.
    - You don't have to worry about the format of the Ids.
    - You don't have to handle the case where movieId is an empty string.

- **GET /api/v1/hasRelationship**
  - **Description:** Check if there exists a relationship between an actor and a movie.
  - **Body Parameters:**
    - `movieId: String`
    - `actorId: String`
  - **Body Example:**

```
{
     "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 check
    - `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)
  - **Edge cases:**
    - You don't have to worry about the format of the Ids.
    - You don't have to handle the case where Ids are empty strings.

- **GET /api/v1/computeBaconNumber**
  - **Description:** Get the bacon number of an actor. You can assume Kevin Bacon will always be in the database with actorId=`nm0000102`
  - **Body Parameters:**
    - `actorId: String`
  - **Body Example:**

```
{
     "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)
  - **Edge cases:**
    - You don't have to worry about the format of the Ids.
    - You don't have to handle the case where Ids are empty strings.

- **GET /api/v1/computeBaconPath**
  - **Description:** Get a path (in order) from a given actor to Kevin Bacon. You can assume Kevin Bacon will always be in the database with actorId=`nm0000102`
  - **Body Parameters:**
    - `actorId: String`
  - **Body Example:**

```
{
      "actorId": "actor1991271"
}
```

  - **Response:**
    - baconPath: List/Path of interchanging actorIds/movieIds (Strings) beginning with the given `actorId` and ending with Kevin Bacon's `actorId`.
  - **`Response Body Example:`**

```
{
      "baconPath": [
            "actor1991271",
            "movie9112231",
            "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's no actor in the database with the given actorId, 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 there's multiple baconPaths, return any one.
    - If you want to compute the baconPath of Kevin Bacon himself, you should return a list with just his actorId in it.
    - You don't have to worry about the format of the Ids.
    - You don't have to handle the case where Ids are empty strings.