CSCC01H5 - Introduction to Software Engineering
Assignment 1

University of Toronto, Scarborough Campus
Summer 2022
Due: Jun 17, 11:59 pm EST

A1 - Rest API and Dependency Injection

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 be required to use a specific dependency injection framework (Google Dagger2) and you will use Neo4j as the database management system.

OBJECTIVE

1. Explore NoSQL/Graph Database (Neo4j)
2. To create REST API endpoints that are supported by Neo4j graph databases
3. To utilize Git, Git Flow, and code style correctly
4. Practice Enterprise Design Patterns, specifically dependency injection
5. To learn and apply CI/CD

PROJECT/IDE SETUP

We have prepared a video to walk you through the setup. **Please watch it!**

This assignment requires you to have the following software:

- Latest version of Java
  - JDK Version 16.0.1
  - JRE Version 16.0.1
- Latest version of Neo4j Desktop
- Maven version 3.6.3

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.

Docker:

We have provided an optional Docker configuration, allowing easy setup regardless of your platform.

- Install Docker Desktop
- To compile and run your code simply run **`docker compose up --build`** in the root directory (The folder that has docker-compose.yml)
- Note: If you are using a M1 Macbook/Linux based system please comment out the platform line in the docker-compose.yml file.

In the event that you cannot use Docker, we have added code to pull the neo4j address from environment variables (localhost for local development and neo4j when running with Docker). This is to ensure that, even if you code locally, we can mark it later on using Docker. **You are required to incorporate this code into your solution**, and failure to do so will result in a mark reduction.

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.

HOW TO SUBMIT YOUR WORK AND WHAT TO SUBMIT

1. Your work will be submitted through GitHub classrooms on the main branch. Please ensure that your final submissions are merged into the main branch prior to the submission deadline.
2. You may work in teams of 2
3. First team member: please accept the invitation by clicking here, then invite your team mate.
4. Next team member: wait for your other team mate to invite you. Once invited, join the team.
5. Ensure your final draft is on your master branch before the assignment deadline.
6. Specifically, your src folder and pom.xml must be located in your main folder of your repository.
7. In the same place, please add your `team.md` file, listing all team members' utorids separated by commas.
   - Example:

```
smithjon, simpsonb
```

   - **NOTE:** LACK OF TEAM.MD OR INCORRECT FILE/FORMAT WILL RESULT IN NOT RECEIVING YOUR GRADE.

STARTER FILES

Do **not** change any code that is already given to you in the starter files, doing so may result in a grade of 0. Starter code can be found on the course website.

- `App.java`
- `Neo4jDAO.java`
- `ReqHandler.java`
- `ReqHandlerComponent.java`
- `ReqHandlerModule.java`
- `Server.java`
- `ServerComponent.java`
- `ServerModule.java`
- `Utils.java`
- `Apptest.java`

Continuous Integration / Continuous Delivery

You are required to use CI/CD for this assignment as well. Setup the .github/workflows folder and add the provided app-test.yml file **once you have written tests**. Your job will be to write tests for your endpoints. These tests will go in **src/test/ ... /AppTest.java**

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.

Please follow the table below on how to name your tests and what are acceptable status' to test with each.

NOTE: Failure to follow the naming of these tests exactly will 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`

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 methods that you will need to implement the rest of the assignment. If you make any database transactions without using the DAO, you may lose marks.

DEPENDENCY INJECTION

You will be required to use dependency injection for this assignment. We will only be using constructor injection. 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 specific endpoint handlers)

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.

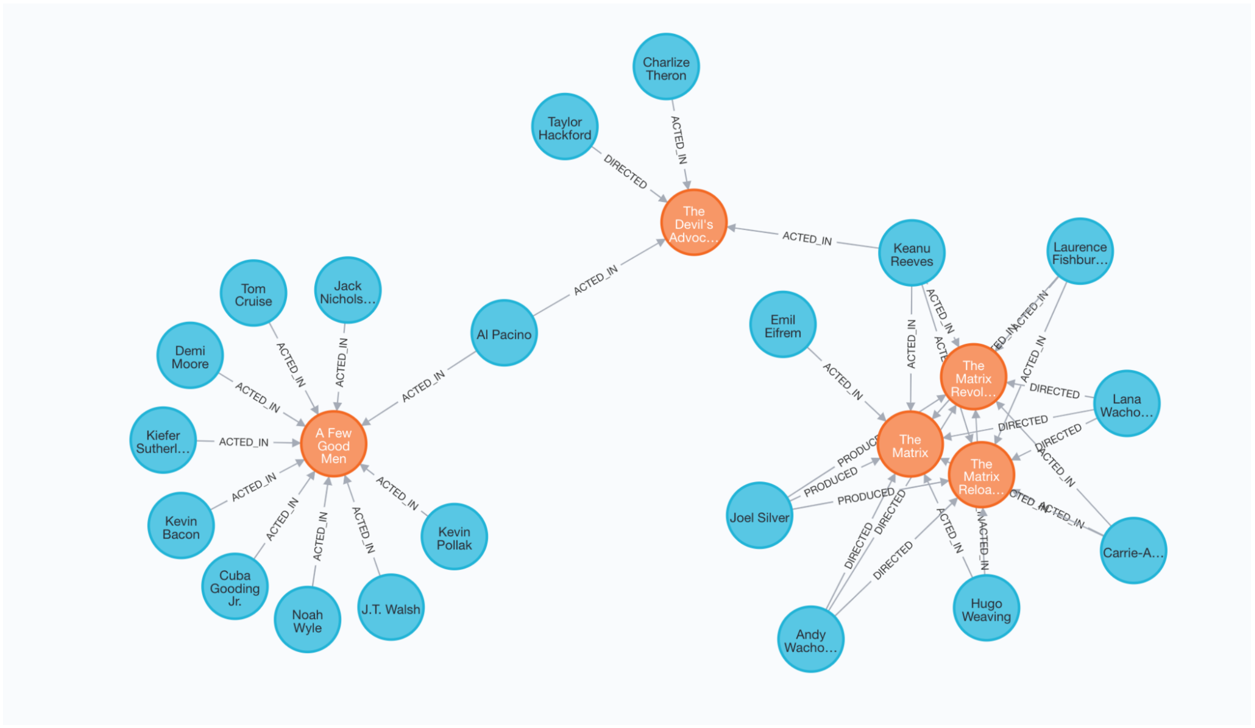
The method names inside your dagger modules must follow this convention: The method name starts with ‘provide’ and ends with the object being returned from that method

e.g. if you’re writing a provided 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.

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.

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.

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.
  - **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",
            "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.
    - **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",
		"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.
    - **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 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.
    - Kevin Bacon has an `actorId` of `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)

- **GET /api/v1/computeBaconPath**

- **Description:** This endpoint returns the **shortest** Bacon Path in order from the actor given to Kevin Bacon.
    - Kevin Bacon has an `actorId` of `nm0000102`
- **Body Parameters:**
    - `actorId: String`
- **Body Example:**

```
{
      "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. Make sure that Kevin Bacon is in the path returned in the response body.

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: 123456

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`

Git Usage

You are required to use Git flow for this assignment, please see tutorial 1 for reference. You will be evaluated on proper usage of Git flow and commit messages.

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 routes and function signatures appropriately.

TESTING YOUR CODE

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.

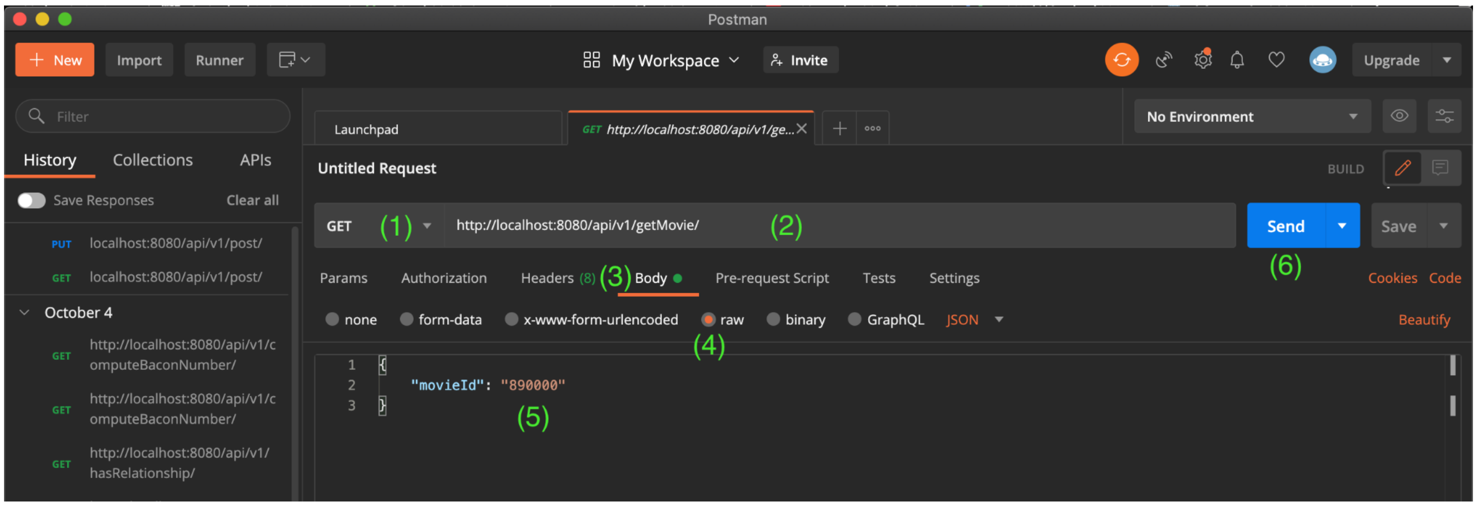
Sending a Request and Reading a Response

- 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 x-www-form-urlencoded (4) if you are sending body parameters
- Your response data will be shown in the body area (5) when you click the Send button

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!

- Setup video



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

- Neo4j Cypher Tutorial
- Dagger2 Tutorial (videos 1, 2, and 5 are very useful)

Good luck! 💯💯