Introduction to APIs

by BPI

What You'll Learn

1. What is an API and RESTful APIs
2. HTTP
3. JSON and Serialization w/ Jackson
4. Building a REST API

What We'll Discuss

1. Project Structures
2. Building a minimal RESTful API with Spark Java

Project Structure

- In the early 2000s, enterprise projects were getting massive, consequently, teams were also growing in size. Software stability and maintainability was prioritized instead of speed. Productivity slowed down as spaghetti code became an issue. To solve this problem, the industry adapted proper structuring.
- Applications are now separated into layers with each layer having one responsibility.
- Applications became easier to test, to maintain, and to scale.
- There’s no one rule when structuring applications. There are multiple software architectural patterns that exist. One of the most common structure is the N-Tier Layered Architecture

Layered Architecture (N-Layer Architecture)

- A design pattern focused on organizing applications into multiple different layers, with each layer focused on one responsibility.
- Separation of concerns is the key idea.
- It promotes Scalability as it allows developers to upgrade individual components without affecting other components.
- It promotes Maintainability as it becomes easier to navigate the application structure as components are organized and easily manageable.

Other Patterns

- **MVC** pattern or the **Model-View-Controller** pattern is a pattern that introduces separation of concern. The components are grouped into Controllers which handle user input and routing, Models that are concerned with the business logic and the data involved, and the View which presents the model and handles user interaction.
- Another pattern is the **Three-Tier Architecture** which is composed of the Presentation Layer, the Application Layer, and the Data Layer.

Backend Components

- N-Layer architecture is common in RESTful APIs.
- Here are the concepts that you'll see the backend application of RESTful APIs
  - Config
  - Controller
  - Service
  - Repository
  - Model
  - DTO
  - Exception
  - Util

Typical Structure of an N-Layer Architecture

/**config**

Contains configuration like server configuration, database configuration, filters, or any code that would be necessary for the setup of the application.

/**controller**

Contains controller classes that serve as the entry point for HTTP Requests.

/**service**

Contains the business logic for the whole application.

/**repository**

Contains data access logic.

/**model**

Contains the domain objects, or the primary entities in the application

/**dto**

Contains Data Transfer Objects, or objects that are used to represent the application models during API communication.

/**exception**

Contains exception classes.

/**util**

Contains Utility classes for shared use across the application.

Config

- Code that holds application configuration and setup.
- This could include security setup, filters/interceptors, dependency wiring, database configuration, any other configuration or registration required by the framework.
- Keywords: Set up, config

Controller

- Handles HTTP requests or client interactions.
- Declare routes, accept request payload, input validation, returns response.
- Calls the service layer
- It should NOT contain business logic or complex processing.
- Keywords: Routing, API endpoints, Request & Response

Service

- Contains the business logic and application’s core process.
- Calls the repository layer
- Keywords: Business logic, processes, brain of the application

Repository (Data Access Layer)

- Handles the interaction with the data sources.
- Responsible for querying the database and database persistence.
- Keywords: database, repositories, data access layer, DAO (Data Access Object)

Model (Domain / Entity)

- Represents the core business entities.
- The objects or entities used by the application.
- Keywords: models, entities, objects

DTO (Data Transfer Object)

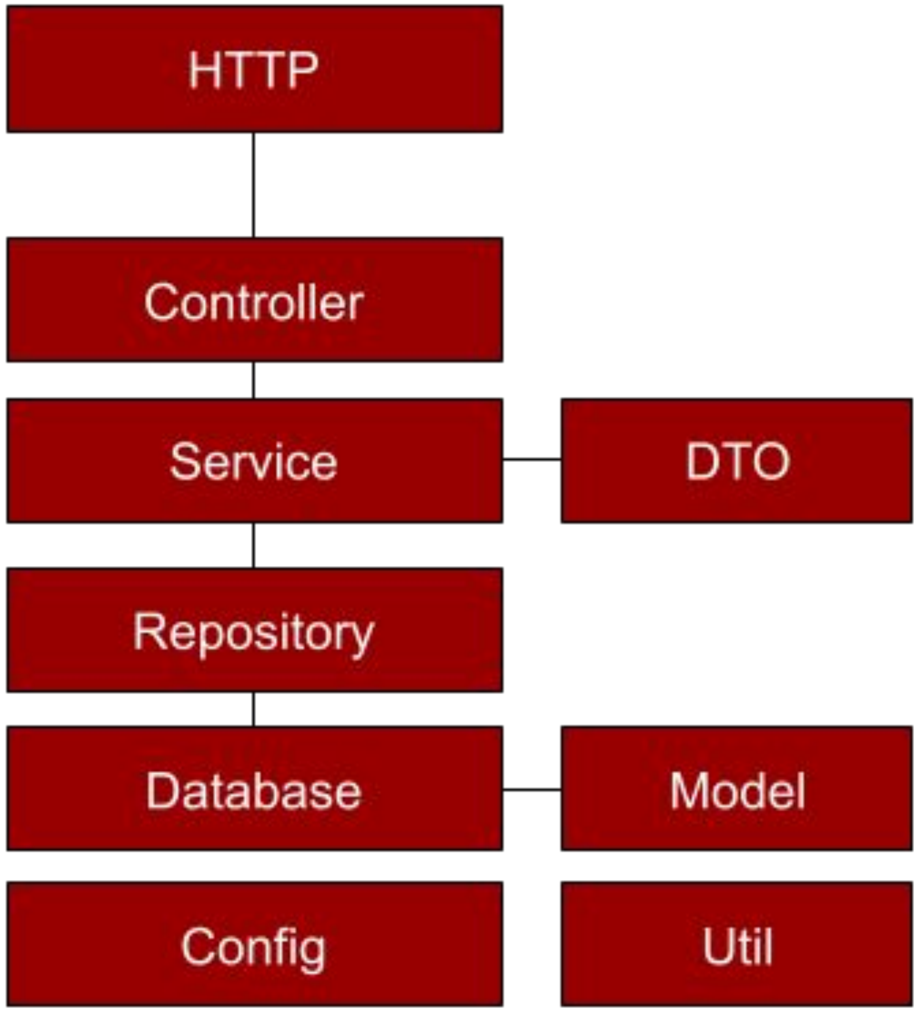
- Represents the data moving to-and-from the application.
- Commonly used as a representation of an entity as the entities themselves should not be passed or exposed to other layers.
- Its primary purpose is to hide sensitive or unnecessary data which in turn saves memory.
- Keywords: carrier, transfers data, representation

Exception

- Contains the application’s custom exceptions and global exception handling.
- Keywords: Exception handling, error handling

Util

- Utility classes that can be used across all layers and contains logic that do not count as “core business” logic.
- It can contain validation, formatting, parsing, file utilities, or any other helper classes.
- Should not be dependent on other layers.
- Keywords: Utility, Helper



*this is only a visualization

Introduction to Spark Java

- A lightweight and expressive framework for creating web applications allowing for fast development and minimal boilerplate code.

Spark Java to Spring Boot

- Spark Java is a framework designed to be like a **Domain-Specific-Language**, a language specialized in solving one problem, for Web applications. It's designed in a way that it needs minimal configuration unlike Spring Boot which, although offers a lot of automatic configuration, still requires a lot of setup even for a minimal web server application.
- In this lecture, we're going to structure our Spark Java application like a Spring Boot application so that the architecture and concepts are carried over even though the framework changes.

Creating a Basic Application (1)

- In this lecture, we're going to structure our Spark Java application like a Spring Boot application as much as possible so that the architecture and concepts are carried over even though the framework changes.
- Let's start with creating a minimal Spark Java Server

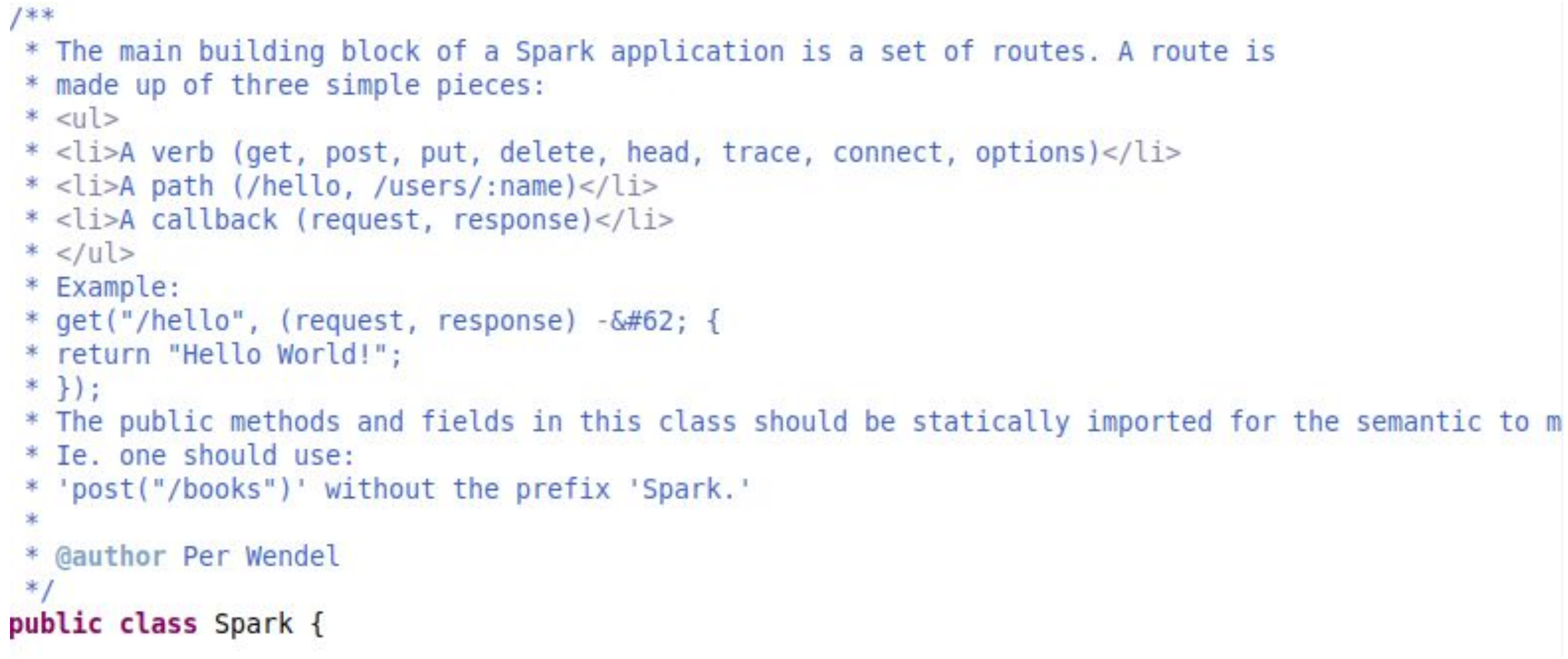
Creating a Basic Application (2)

- Create a basic Maven Project with Spark Java imported. In your Main class, add the following import.
- Note that it is a **static** import. It allows us to use all static methods inside the **Spark** class.

```
import static spark.Spark.*;
```

Creating a Basic Application (3)

- The documentation for the **Spark** class indicates that this is intended as it allows developers to setup the server intuitively.



```
/**
 * The main building block of a Spark application is a set of routes. A route is
 * made up of three simple pieces:
 * <ul>
 * <li>A verb (get, post, put, delete, head, trace, connect, options)</li>
 * <li>A path (/hello, /users/:name)</li>
 * <li>A callback (request, response)</li>
 * </ul>
 * Example:
 * get("/hello", (request, response) -&#62; {
 * return "Hello World!";
 * });
 * The public methods and fields in this class should be statically imported for the semantic to m
 * Ie. one should use:
 * 'post("/books")' without the prefix 'Spark.'
 *
 * @author Per Wendel
 */
public class Spark {
```

Creating a Basic Application (4)

- Let’s declare a **port** for our application; A port is a virtual point in our host that essentially tells where an application is located.
- Let’s create this application at port **4567** by calling the static method: **port(int)**
- The port must be added BEFORE any routes.

```
public class Main {

    private static final Logger logger =  LoggerFactory.getLogger(Main.class);
    private static final ObjectMapper mapper = new ObjectMapper();

   public static void main(String[] args) {

     // Start server on port 4567 (default)
       port(4567);
```

Creating a Basic Application (5)

- Finally, we add **routes** or **endpoints** to our web application. These are paths in our application to which a client application can invoke our API functionalities.
- Let’s create a route that will quickly allow us to know if the application is up and running. Pick a **name** for the path and call the **get(String, Route)** method.

```
// We create a route for checking if the server is active.
get("/check-connection", (req, res) -> {
    res.type("application/json");// define the MIME type of the response
                                 // we're telling the client that the response
                                 // is of JSON format
    // return a map (A map is basically a JSON formatted object)
    Map<String, String> response = new HashMap<>();
    response.put("status", "Server is running");
    return JsonUtil.toJson(response);
});
```

Note: the second parameter of get (and other methods) is a class called **Route** which is what we call a **Functional Interface.** It’s essentially a parameter that is also a **function**. We use it to pass

Creating a Basic Application (5)

- Let’s add a route for a **POST** method.

```
// An example POST method that returns the  payload that you send.
post("/echo", (req, res) -> {
    res.type("application/json");
    // Just return the request body as the response
    return req.body();
});
```

Quick Cheat Sheet for Spark Java

- Below is all you need to know for creating a simple Spark Java Endpoint

```
// This is a PUT endpoint
// put = method
// /persons = path
// :id = uri parameter, this indicates that this is a dynamic value
// req = HTTP request sent by the client
// res = HTTP response to be returned by this path, you can set headers here
put("/persons/:id", (req, res) -> {
    // this is a query param. it looks like this: /persons/1?name=value
    String name = req.queryParams("name");

    // we get the value of :id
    String paramId = req.params("id");

    // we deserialize the JSON into our Java Object
    Object payload = JsonUtil.fromJson(req.body(), Person.class);
    return JsonUtil.toJson(payload);
});
```

Exercise 1 - Individual (M7_ACTIVITY1)

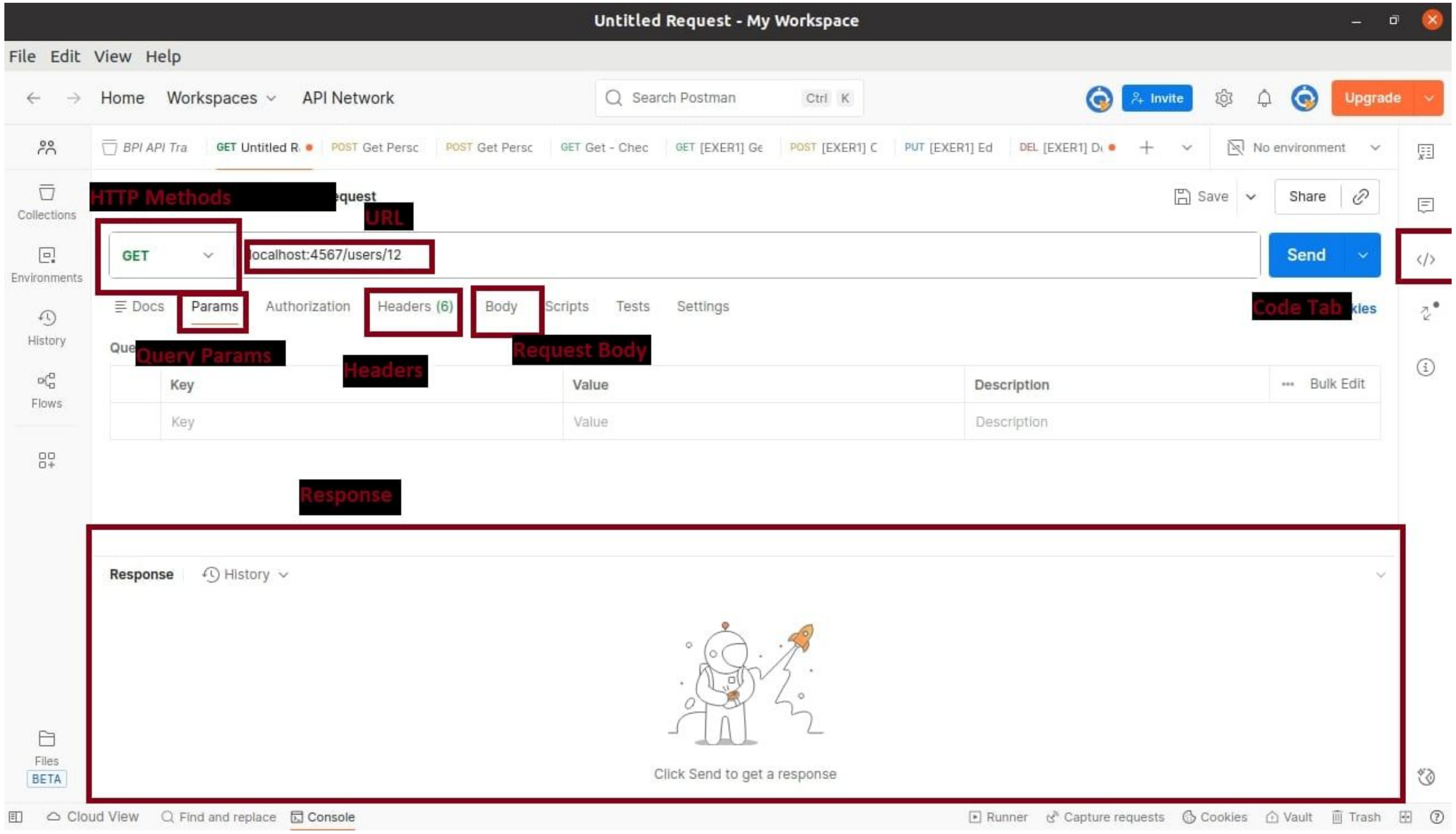
- Clone the repository https://github.com/jpmenguito/api-training (or sync the repository if you already cloned it)
- Import / Run the project named **m7_activity1** using Eclipse or your IDE.
- If you cannot run it, feel free to copy the code into your project, just make sure you have the proper Maven Dependencies. (See **Module 7 Prerequisites.pdf)**
- **Requirements:**
    - Inside **Main.java,** implement a **GET** and **POST** endpoint for the entity Movie
    - **GET** movies should retrieve **all movies**
    - **POST** movies should accept a Movie object from the request and save it inside the repository.

Exercise 1 - Individual (M7_ACTIVITY1)

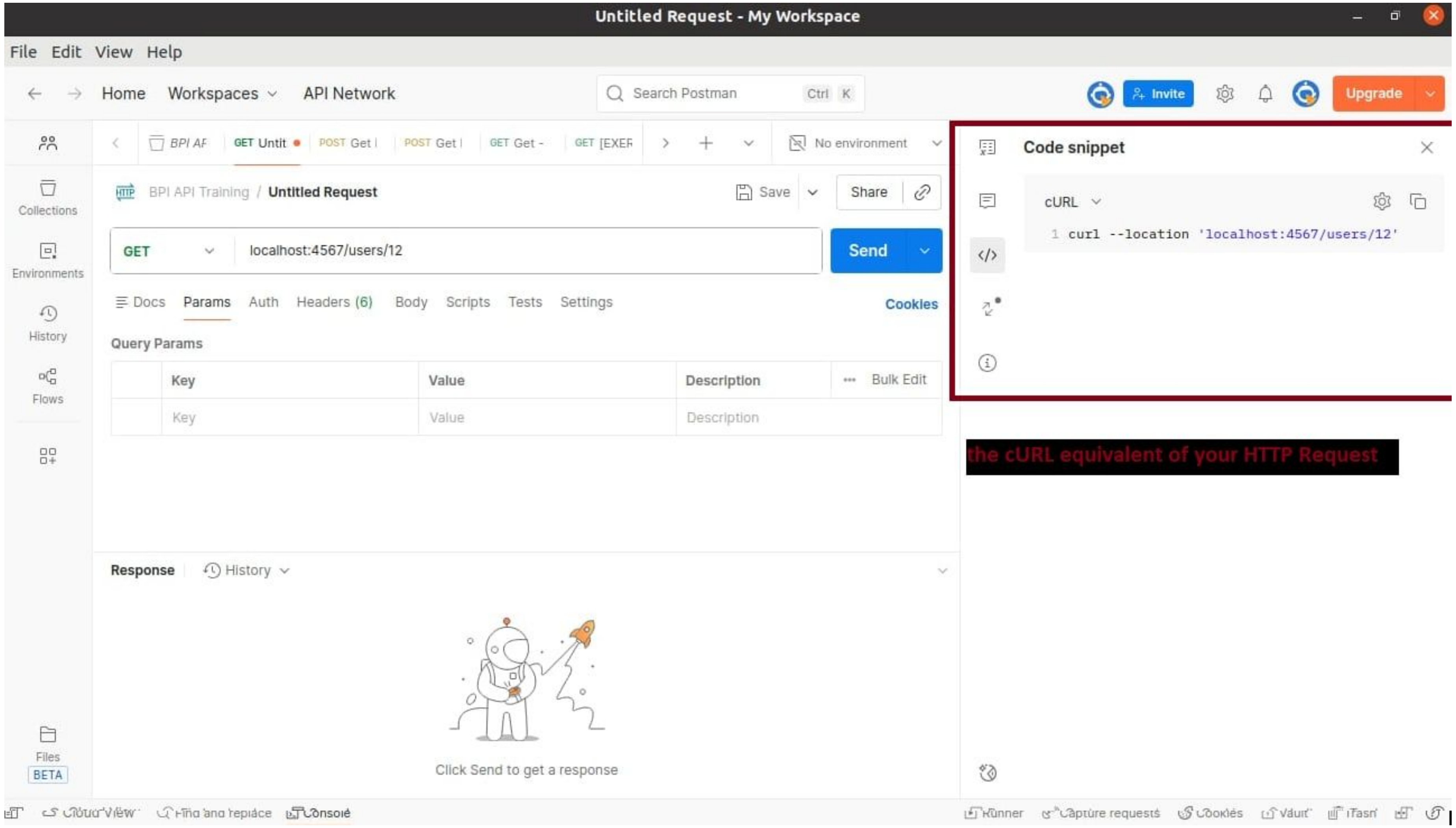
- **m7_activity1** is already configured to have a database with an already existing **Movie** for the model, **MovieRepository** for the DAO.
- Use the functions of **MovieRepository** to satisfy the requirements below.
- **NOTE:** edit /src/resources/META-INF/persistence.xml according to **your computer's database** setup.

- **Requirements:**
  - Inside **Main.java,** implement a **GET** and **POST** endpoint for the entity Movie
  - **GET** movies should retrieve **all movies**
  - **POST** movies should accept a Movie object from the request and save it inside the repository.

Postman Quick Guide



Postman - Code Tab



GET localhost:4567/users/12

	Key	Value	Description
	Key	Value	Description

```
curl --location 'localhost:4567/users/12'
```

the cURL equivalent of your HTTP Request

cURL (Client for URLs)

- A powerful open-source command-line tool for API testing (among other things).
- It's similar to Postman and Bruno.

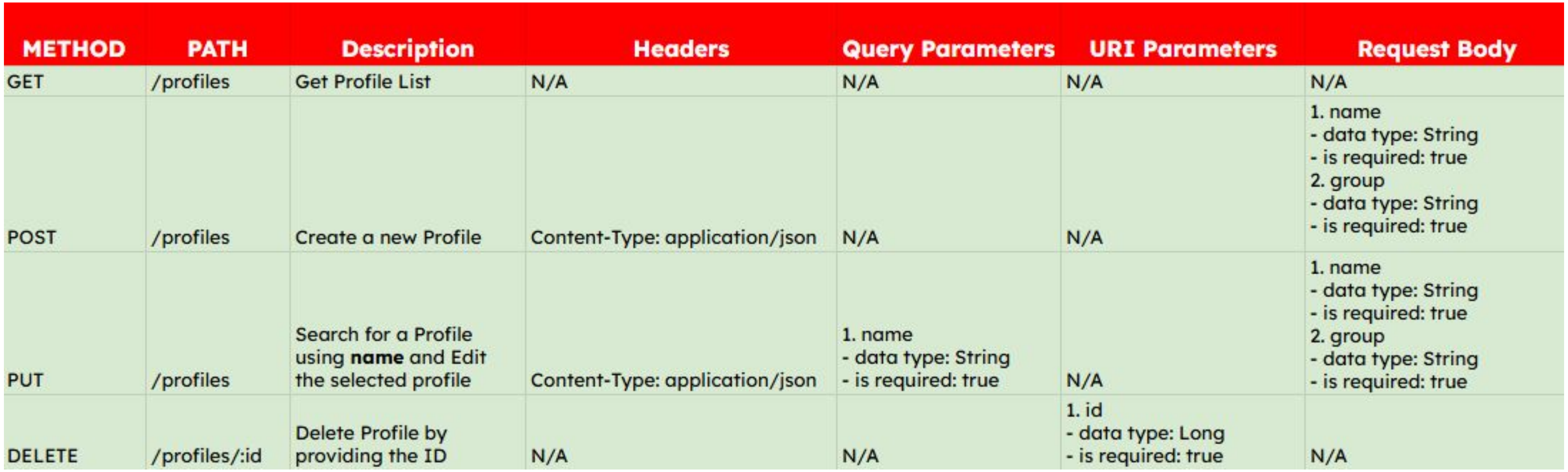
```
curl -X POST https://api.example.com/data -H "Content-Type: application/json" -d '{"key": "value"}
```

Exercise 2 - Individual (M7_ACTIVITY2)

- Clone the repository https://github.com/jpmenguito/api-training (or sync the repository if you already cloned it)
- Import / Run the project named **m7_activity2** using Eclipse or your IDE.
- If you cannot run it, feel free to copy the code into your project, just make sure you have the proper Maven Dependencies. (See **Module 7 Prerequisites.pdf)**

Exercise 2 - Individual (M7_ACTIVITY2)

- Refer to the API specifications below:



METHOD	PATH	Description	Headers	Query Parameters	URI Parameters	Request Body
GET	/profiles	Get Profile List	N/A	N/A	N/A	N/A
POST	/profiles	Create a new Profile	Content-Type: application/json	N/A	N/A	1. name - data type: String - is required: true 2. group - data type: String - is required: true
PUT	/profiles	Search for a Profile using **name** and Edit the selected profile	Content-Type: application/json	1. name - data type: String - is required: true	N/A	1. name - data type: String - is required: true 2. group - data type: String - is required: true
DELETE	/profiles/:id	Delete Profile by providing the ID	N/A	N/A	1. id - data type: Long - is required: true	N/A

Exercise 2 - Individual (M7_ACTIVITY2)

- Given the API specifications, test the API using Postman or Bruno.
- Do the steps outlined below.
- **AFTER each step, copy the cURL command equivalent of your request and paste it in a .txt file.**

1. Get Profile List.
2. **Create** a profile using your **name** and **group**.
3. Edit the profile of **Emilio Aguinaldo** and update the name to **Antonio Luna** and the group name to **Illustrados.**
4. Delete the profile of **Andres Bonifacio.**
5. Copy and paste the **Response** of Step #4.

Adding Structure to our API

- Let’s improve our basic API by refactoring our code and separating them into different classes.
- A structured RESTful API must at least have a Controller class, a Service class, and a Repository class and a Model class.
- For now, let’s create **PersonController, PersonService**

Controller

- Create a PersonController class with a dependency PersonService. Create a constructor initializing the dependency.

```
public class PersonController {

    private final PersonService personService;

    // initialize dependencies
    public PersonController (PersonService personService) {

        this.personService = personService;

    }
```

Controller

- add a method called **registerRoutes()** which should contain all the routes we previously added in our Main class

```
public void registerRoutes() {

        // We create a route for checking if the server is active.
       get("/check-connection", (req, res) -> {
           res.type("application/json");    // define the MIME type of the response
                                           // we're telling the client that the response
                                           // is of JSON format
           // return a map (A map is basically a JSON formatted object)
           Map<String, String> response = new HashMap<>();
           response.put("status", "Server is running");

           return JsonUtil.toJson(response);
       });
```

Controller

- let’s add a **GET** and **POST** route that calls the **PersonService**

```
get("/persons", (req, res) -> {

          ResponseDTO<List<PersonDTO> > response = new ResponseDTO<>();
          res.type("application/json");
          response.setStatus(ResponseStatus.SUCCESS);
          response.setData(this.personService.listPersons() );

      return JsonUtil.toJson(response);
      });
```

Controller

- let’s add a **GET** and **POST** route that calls the **PersonService**

```
post("/persons", (req, res) -> {
        ResponseDTO<PersonDTO>  response = new ResponseDTO<>();

    if(req.body() == null || req.body().isBlank() ) {
        response.setStatus(ResponseStatus.ERROR);
        response.setMessage("Payload cannot be blank!");
        return JsonUtil.toJson(response);
    }

    PersonDTO personDTO = JsonUtil.fromJson(req.body(), PersonDTO.class);
    this.personService.addPerson(personDTO);

    response.setStatus(ResponseStatus.SUCCESS);
    response.setData(this.personService.getPersonById(personDTO.getId() ) );
    return JsonUtil.toJson(response);
   });
```

DTO (Data Transfer Object)

- In our routes, we used two DTOs called **ResponseDTO** and **PersonDTO**. DTOs are simple objects used to transfer/hold data.
- We use ResponseDTO as the standard response representation of our API. PersonDTO is the equivalent of our model.

ResponseDTO

- Create a ResponseDTO that can hold any data type for the field data
- Add getters and setters.

```
public class ResponseDTO<T> {

    private ResponseStatus status;
    private String message;
    private T data;             // this means that this class
                                // can hold ANY "data" of type "T"
                                // if we use ResponseDTO<String> res = new ResponseDTO<String>();
                                // res.getData() will be type String
```

ResponseStatus (Enum)

- Create a simple Enum for the status

```java
public enum ResponseStatus {
    SUCCESS,ERROR
}
```

PersonDTO

- Create a PersonDTO that contains all the fields (from **Person**) that we want to expose in our API. Add constructors, getters and setters, and a special method we use to convert PersonDTO to Person

```
public class PersonDTO {

    private Long id;
    private String name;
    private Integer age;
    private Boolean isAwesome;

    public PersonDTO() {}
    public PersonDTO(Long id, String name, Integer age, Boolean isAwesome) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.isAwesome = isAwesome;
    }
```

PersonDTO

```
public PersonDTO(Person personEntity) {
    this.setId(personEntity.getId() );
    this.setName(personEntity.getName() );
    this.setAge(personEntity.getAge() );
    this.setIsAwesome(personEntity.getIsAwesome() );
}


// we call this method to return an instance of Person
public Person toEntity() {
    Person person = new Person();
    person.setId(this.getId() );
    person.setName(this.getName() );
    person.setAge(this.getAge() );
    person.setIsAwesome(this.getIsAwesome() );
    return person;
}
```

PersonService

- Create the PersonService class

```
public class PersonService {

    // this is just a holder of all the Persons that we create
    // ideally this should just be a repository
    private final List<Person> personList = new ArrayList<>();

    // GET ALL Persons
    public List<PersonDTO> listPersons() {
        // return a list of PersonDTO
        return personList.stream()
                  .map(x -> new PersonDTO(x) )
                  .collect(Collectors.toList() );
    }
    // GET One Person by id
    public PersonDTO getPersonById(Long id) {
        return personList.stream()
                  .filter(x -> x.getId().equals(id) )
                  .findFirst()
                  .map(x -> new PersonDTO(x) )
                  .orElse(null);
    }
```

PersonService

```
// CREATE Person
public List<PersonDTO> addPerson(PersonDTO personDTO) {
    // add person
    personList.add(personDTO.toEntity() );

    return personList.stream()
            .map(x -> new PersonDTO(x) )
            .collect(Collectors.toList() );
}
```

Main.java

- Modify Main.java and only instantiate the Service and Controller classes.

```
public static void main(String[] args) throws Exception {

 // Start server on port 4567 (default)
  port(4568);


  // instantiate the service
  PersonService personService = new PersonService();
  // instantiate the controller while plugging in the personService
  PersonController personController = new PersonController(personService);
  // call the registerRoutes to register all declared routes
  personController.registerRoutes();

  logger.info("Server started at port {}", port() );
}
```

Exercise 3 - Individual (M7_ACTIVITY3)

- Modify your **M7_ACTIVITY1** and refactor the contents of **Main.java**.
- Create a **MovieController, MovieService** and put them into different **packages**.
- **MovieController** has a dependency on **MovieService** and should call the service methods for business logic. It should also have the method **void registerRoutes()** containing all the routes.
- **MovieService** should call the **MovieRepository** methods. (And instantiate the EntityManager)
- Your Main.java should just contain the **port** and the instantiation of **MovieService and MovieController** only.
- Call MovieController.registerRoutes() in Main.java.

Group Project Requirements - M7

1. Convert your M6 project into a RESTful API using Spark Java
2. Create a **BookController** class.
3. Create a route that utilizes/represents all the M6 functionalities:
   a. Display All Books
   b. Display Available Books
   c. Display All Borrowed Books
   d. Borrow Book
   e. Return Book
   f. Add Book
   g. Remove Book
   h. Update Book
4. All interactions (input and output) must be made through the API.
5. Create an API specification for your API. It must at least indicate the **HTTP Method, Resource Path, Query Parameters, URI Parameters, Request Body, and the Response Body**.
6. Bonus Points:
   a. +1 for Clever Design
   b. +1 for Validation
   c. +1 for Proper Use of HTTP Statuses