Xideral

Back End Academy

# REST APIs with Spring Data JPA

**Project By:**

**Kenneth Rodríguez García**

# REST APIs with Spring JPA

**Summary**

This project is a continuation and evolution of the previous REST API. Using the Spring Data JPA framework on top of Spring Web, allowing for more secure data persistence and a more maintainable codebase.

**Research**

**Spring Data JPA**

The Spring Data JPA is a framework that simplifies data access across layers in an application. As its name implies, this framework builds on top of the Java Persistence API, or JPA, which provides a consistent model for interaction with a database.

By using Spring Data JPA, we can define a repository interface with only the specific data access methods we desire, while the implementations for the methods are generated at runtime.

However, possibly the best advantage of using this framework is that it belongs to the Spring ecosystem. This provides developers with the most compatibility and ease of use, and the extensive documentation is also a great benefit.

### Design

For comparison purposes, our database will be the same as the previous one: a VideoGame database of different video game titles.

```
▷ Run | New Tab | 🔒 Active Connection
CREATE DATABASE IF NOT EXISTS VideoGameDB;

▷ Run | New Tab
USE VideoGameDB;

▷ Run | New Tab
DROP TABLE IF EXISTS videogames;

▷ Run | New Tab | Copy
CREATE TABLE videogames (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(64) DEFAULT NULL,
    genre VARCHAR(64) DEFAULT NULL,
    platform VARCHAR(64) DEFAULT NULL,
    release_year INT DEFAULT 0,
    developer VARCHAR(64) DEFAULT NULL,
    publisher VARCHAR(64) DEFAULT NULL,
    score DOUBLE DEFAULT 0
) AUTO_INCREMENT=1;
```

The classes in the project fall into 4 categories: Entity, Data Access Objects, Services, and a Rest Controller.

### Entity - VideoGame

The VideoGame class is the entity, which means it is a single object where data is encapsulated. It is used to properly map the data to and from the database.

```java
@Entity
@Table (name="videogames")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class VideoGame{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="title")
    private String title;
    @Column(name="genre")
    private String genre;
    @Column(name="platform")
    private String platform;
    @Column(name="release_year")
    private int releaseYear;
    @Column(name="developer")
    private String developer;
    @Column(name="publisher")
    private String publisher;
    @Column(name="score")
    private double score;

    @Override
    public String toString() {…
}
```

There is virtually no change between this project's Entity and the previous one.

**Data Access Object - VideoGameRepository**

The Data Access Object presents the first major change: by using the JpaRepository interface available in the Data JPA framework, we can create an "empty" interface, which will provide other components with much extended functionality.

```java
import org.springframework.data.jpa.repository.JpaRepository;

public interface VideoGameRepository extends JpaRepository<VideoGame, Integer>{}
```

In the previous DAO implementation, we had to manually use the JPA to define all of our desired methods. But here we can see that simply by using an external method we have abstracted lots of boiler-plate code and have saved development time which can now be allocated to other meaningful tasks.

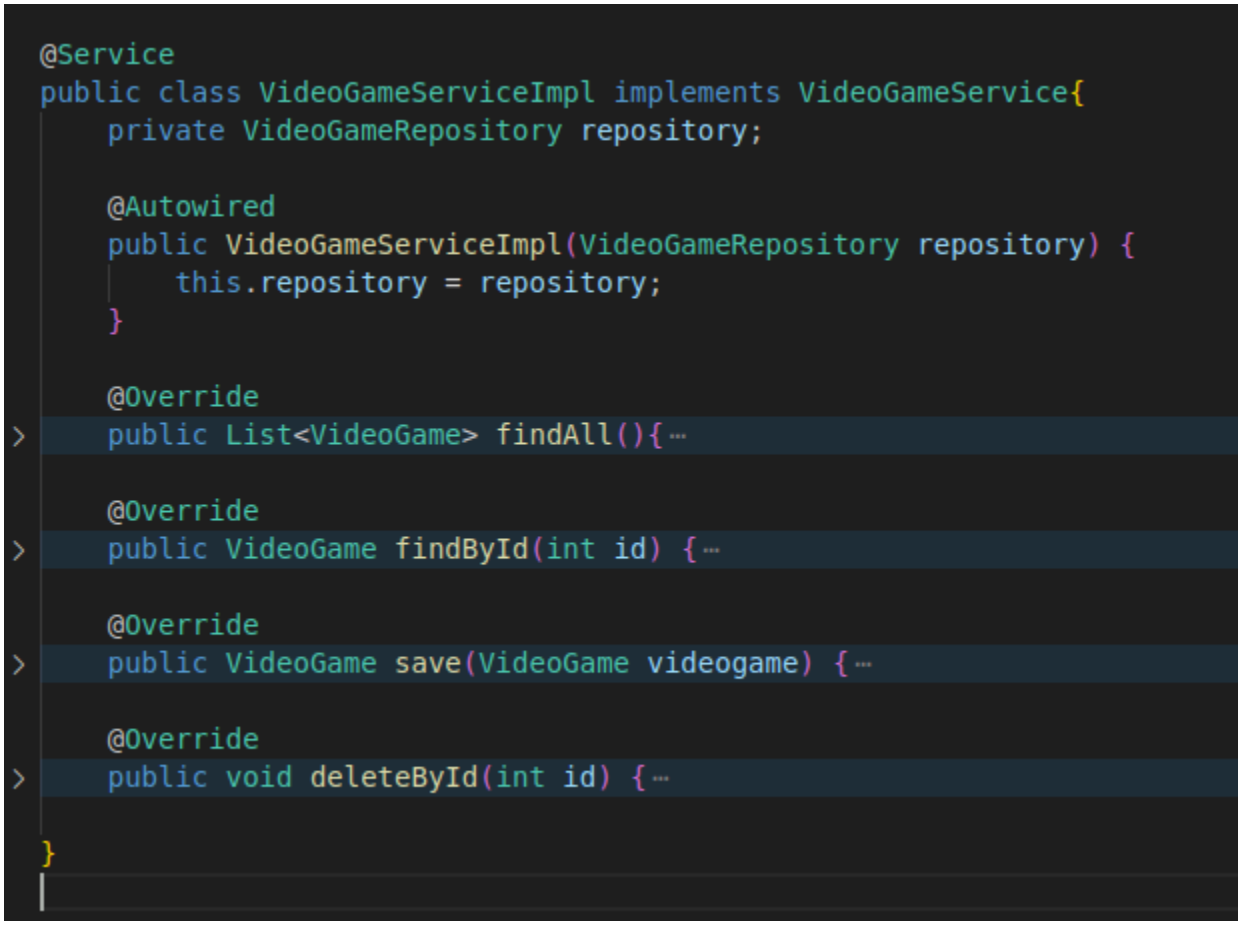
**Service - VideoGameService and VideoGameServiceImpl**

These classes have also changed, although less than the DAOs.

The VideoGameService interface (and VideoGameServiceImpl by extension) no longer has the “update” method. The reason for this is that the Data JPA framework abstracts the behavior of the **POST** and **PUT** methods, essentially merging them. The POST and PUT methods can still have individual routes in the API, but their implementation will be very similar.

```
public interface VideoGameService {
    List<VideoGame> findAll();
    VideoGame findById(int id);
    VideoGame save(VideoGame videogame);
    void deleteById(int id);
}
```

In VideoGameServiceImpl we can observe a special annotation above the class declaration: “@Service”.




This annotation is used to inject the service into a Controller that will consume it. Similarly, the “@Autowired” annotation initializes the constructor of this class and injects a VideoGameRepository object.

**Rest Controller - VideoGameRestController**

The Rest Controller is the class that handles the HTTP requests received by the API. It defines the routes used by the API, as well as each of the methods associated with it.

The API route is “/api”, but these by itself won’t consume any service. The proper use is to use one of the associated routes, such as “/api/videogames”.

```java
@RestController
@RequestMapping("/api")
public class VideoGameRestController {
    private VideoGameService service;

    @Autowired
    public VideoGameRestController(VideoGameService service) {…

    @GetMapping("/videogames")
    public List<VideoGame> findAll(){…

    @GetMapping("/videogames/{videoGameId}")
    public VideoGame getVideoGame(@PathVariable int videoGameId) {…

    @PostMapping("/videogames")
    public VideoGame addVideoGame(@RequestBody VideoGame videoGame) {…

    @PutMapping("/videogames")
    public VideoGame updateVideoGame(@RequestBody VideoGame videoGame) {…

    @DeleteMapping("/videogames/{videoGameId}")
    public void deleteVideoGame(@PathVariable int videoGameId) {…
}
```

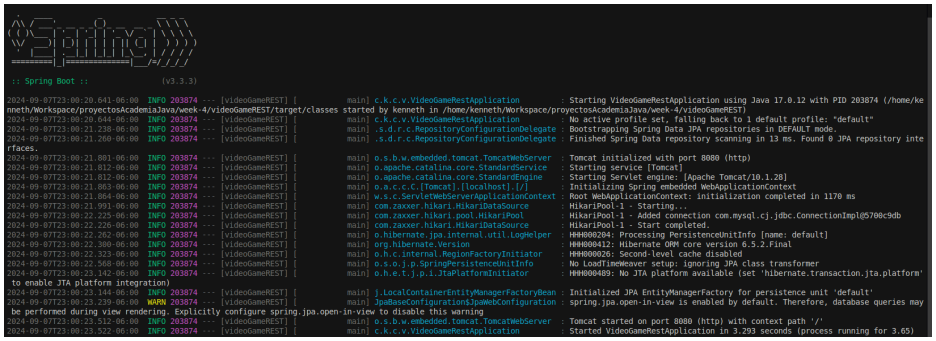
The Rest Controller provides 5 different methods

- findAll(): requests the full list of all VideoGames in the database.
- getVideoGame(): this method uses a parameter (retrieved from the path) to search for a video game with the same ID.
- postVideoGame(): this method inserts the data of a VideoGame object into the database
- putVIdeoGame(): this method updates the data of a VideoGame object already in the database. To identify which entry must be updated, the ID of the object serves as the identifying key.
- deleteVideoGame(): this method retrieves a VideoGame object using the id parameter obtained from the path. If one such object is found, the corresponding entry is removed from the database.

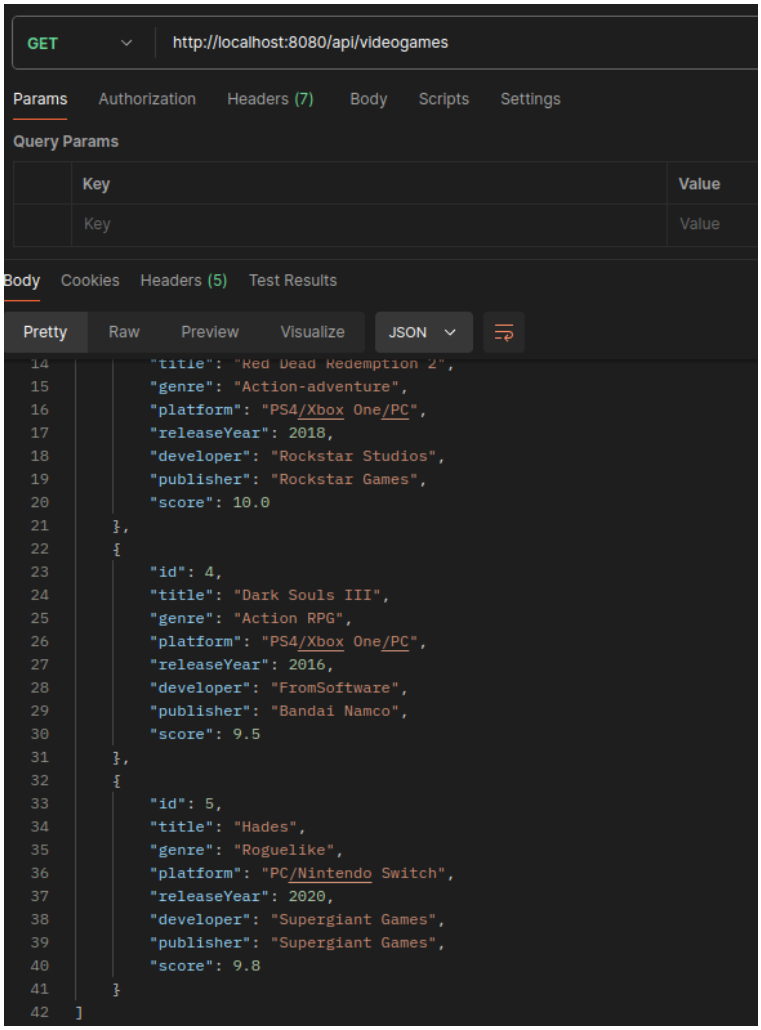
We can observe that our Rest Controller has barely changed from its previous implementation. This is because, while the Data JPA framework abstracts behavior on the Model side of the MVC architecture, the View and Controller don't have to change much. This displays one of the main benefits of the MVC architecture, because despite changing the implementation details of the API, the other components have barely even required modification.

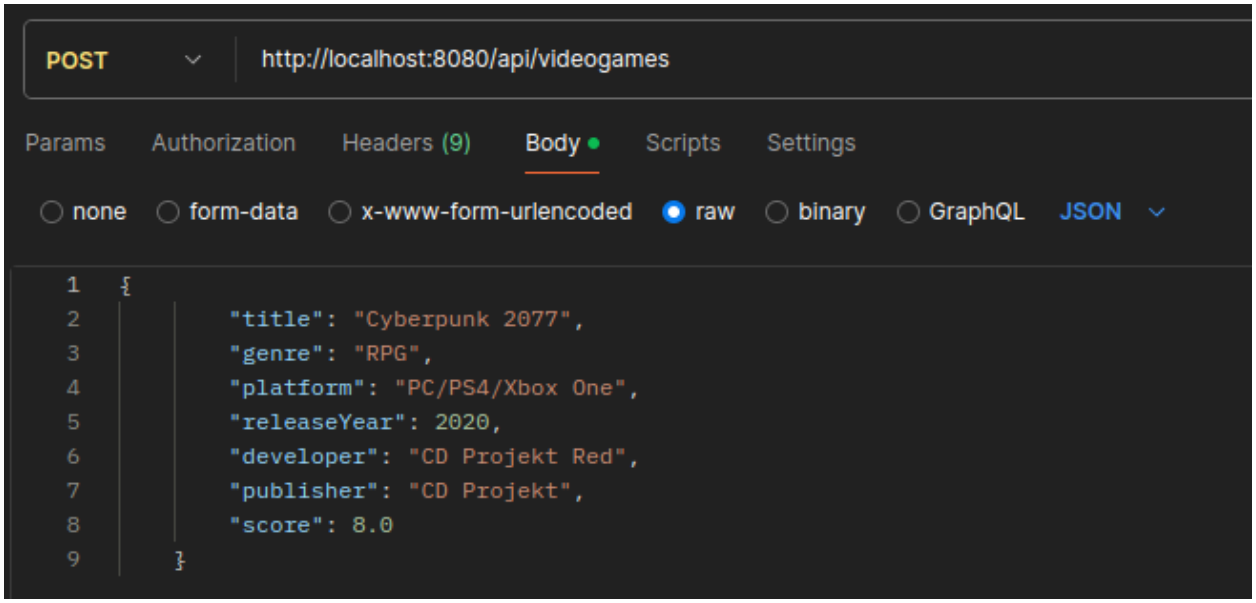
### Execution

When running the application, we will only be greeted by the Spring banner.




By using **Postman**, we can consume the API and verify that its execution produces the same behavior as the previous API.






By thoroughly testing our API we can not only confirm that our latest API's behavior is the same as the previous one, but that the previous changes made are persistent.

**Conclusion**

I am very much a "build-your-own-x" kind of person, but I must admit that using the various Spring frameworks is definitely an eye-opening moment, because they abstract the repetitive behavior and code and instead allow developers to focus and developing the critical aspects of a software project.

This project has made me reconsider my priorities when developing my own software project, so I think I will continue to use it for a while.