# What is Spring Data JPA?

Last Updated : 04 Sep, 2025

Spring Data JPA is a framework that simplifies database access in Spring Boot applications by providing an abstraction layer over the Java Persistence API (JPA). It enables seamless integration with relational databases using Object-Relational Mapping (ORM), eliminating the need for boilerplate SQL queries.

With Spring Data JPA, developers no longer need to:

- Manually write DAO implementations.
- Manage EntityManager directly.
- Write repetitive CRUD queries

## Key Features of Spring Data JPA

- Eliminates most boilerplate code for data access.
- Provides built-in CRUD methods through JpaRepository.
- Supports derived query methods (e.g., findByName).
- Offers pagination and sorting out of the box.
- Works with any JPA provider (commonly Hibernate).

## JPA vs Spring Data JPA

It's important to understand that Spring Data JPA builds on top of JPA.

- JPA is just a specification (a set of interfaces and rules).
- Spring Data JPA is a framework that provides higher-level abstractions and auto-implemented repositories.

## Building a Spring Boot Application with JPA

### Step 1. Create the project

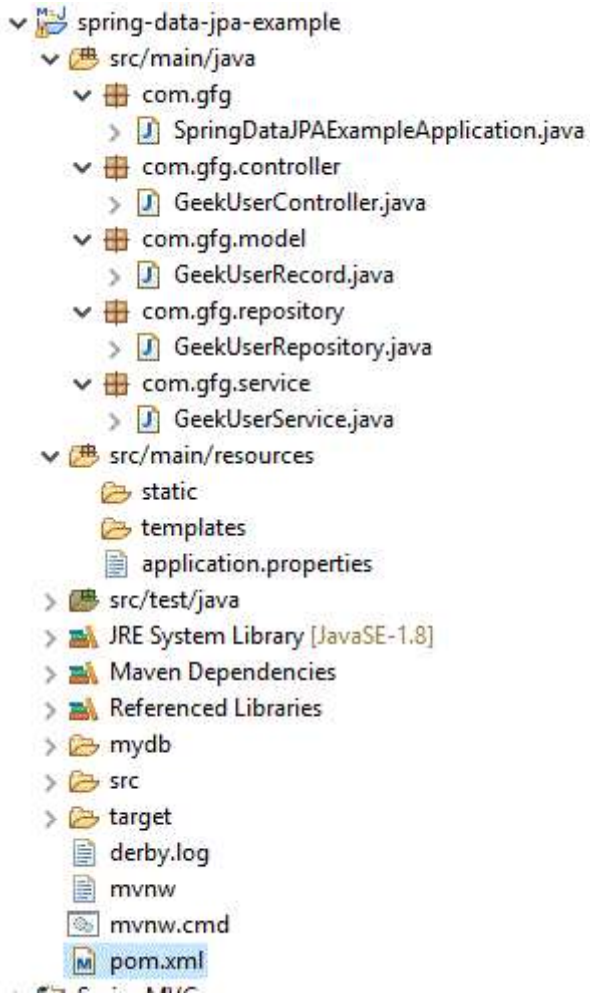
Use Spring Initializr (https://start.spring.io) or your IDE:

**Project:** Maven
**Language**: Java
**Spring Boot:** 3.x (any modern 3.x release)
**Java:** 17 (or 21)
**Dependencies:**

- Spring Web
- Spring Data JPA
- H2 Database (for quick testing)

**Project Structure:**

After creating the project, A dd the following classes then the folder structure will be like below:




## Step 2. pom.xml

If using Maven, ensure these dependencies (Initializr will generate them). Example snippet (dependencies only)

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>

  <dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

## Step 3. Configure datasource (H2 quick setup)

Configure database datasource inside **src/main/resources/application.properties:**

**application.properties:**

```
# H2 config (in-memory)
spring.datasource.url=jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver

# JPA / Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# H2 console (optional)
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

## Step 4. Defining the Entity Class

Let us see the key important files in the project. Starting with POJO class

**GeekUserRecord.java:**

```
package com.gfg.model;

import jakarta.persistence.*;

@Entity
@Table(name = "geek_user")
public class GeekUserRecord {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    private String name;
    private String email;
    private String gender;
    private Integer numberOfPosts;

    public GeekUserRecord() {}

    // Getters & setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    public String getGender() { return gender; }
    public void setGender(String gender) { this.gender = gender; }

    public Integer getNumberOfPosts() { return numberOfPosts; }
    public void setNumberOfPosts(Integer numberOfPosts) { this.numberOfPosts
= numberOfPosts; }
}
```

## Step 5. REST Controller

Let us see the controller file now.

### GeekUserController.java:

```
package com.gfg.controller;

import com.gfg.model.GeekUserRecord;
import com.gfg.service.GeekUserService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")                // base path
public class GeekUserController {

    private final GeekUserService userService;

    public GeekUserController(GeekUserService userService) {
        this.userService = userService;
    }
```

```
    // GET /api/users
    @GetMapping("/users")
    public List<GeekUserRecord> getAllUser() {
        return userService.getAllGeekUsers();
    }

    // POST /api/users
    @PostMapping("/users")
    public ResponseEntity<GeekUserRecord> addUser(@RequestBody
GeekUserRecord userRecord) {
        GeekUserRecord saved = userService.addGeekUser(userRecord);
        return ResponseEntity.status(HttpStatus.CREATED).body(saved);
    }
}
```

## Step 6. Service layer

Let us see the service file.

### GeekUserService.java:

```
package com.gfg.service;

import com.gfg.model.GeekUserRecord;
import com.gfg.repository.GeekUserRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
@Transactional
public class GeekUserService {

    private final GeekUserRepository repository;

    public GeekUserService(GeekUserRepository repository) {
        this.repository = repository;
    }

    public List<GeekUserRecord> getAllGeekUsers() {
        return repository.findAll();
    }

    public GeekUserRecord addGeekUser(GeekUserRecord userRecord) {
        return repository.save(userRecord);
    }
}
```

## Step 7. Creating the Repository Interface

Now we need to add a repository interface and it should extend CrudRepository.

**GeekUserRepository.java:**

```
package com.gfg.repository;

import com.gfg.model.GeekUserRecord;
import org.springframework.data.jpa.repository.JpaRepository;

public interface GeekUserRepository extends JpaRepository<GeekUserRecord,
Long> {
    // add custom finder methods if needed, e.g.List<GeekUserRecord>
findByName(String name);
}
```

## Step 8. Main application class

Now, we need to execute this program and check the output. For that, we need to run the below file

**SpringDataJPAExampleApplication.java:**

```
package com.gfg;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringDataJPAExampleApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringDataJPAExampleApplication.class, args);
    }
}
```

## Running and Testing the Application

Right-click on the main class and run the file as a Java application, we can see the output in the console.

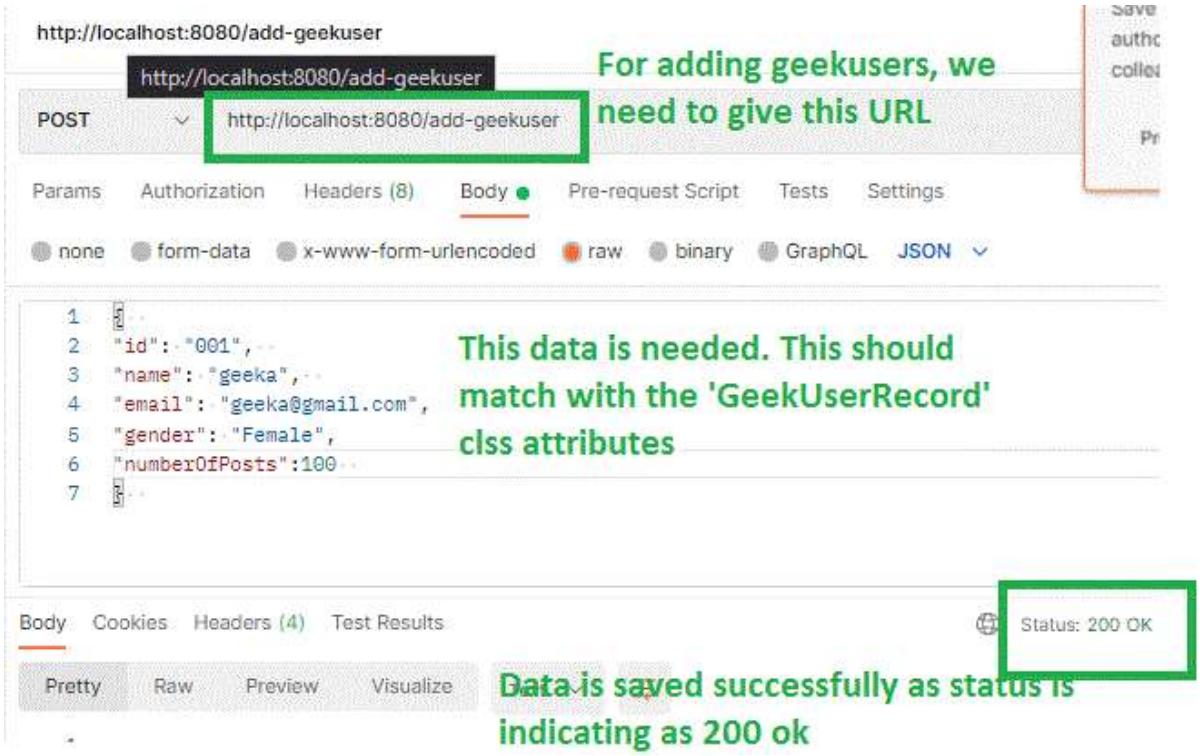
**Output:**

```
task-1] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal
task-1] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
  main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
  main] DeferredRepositoryInitializationListener : Triggering deferred initialization of Spring Data repositories...
  main] DeferredRepositoryInitializationListener : Spring Data repositories initialized!
  main] com.gfg.SpringDataJPAExampleApplication  : Started SpringDataJPAExampleApplication in 7.338 seconds (JVM running for 8.043)
8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 12 ms
```



*Running and Testing the Application*

Initially as there are no records, when we hit **http://localhost:8080**, we won't be seeing any data. Let's add the data by adding via the Postman client. Postman client has to be installed for doing this operation. URL to add users: **http://localhost:8080/add-geekuser** (Remember that this URL matches the controller file requestmapping).




Now a user is added. Hence we can verify the same by using **http://localhost:8080**


## JPA vs Hibernate

| JPA | Hibernate |
|---|---|
| It is a Java specification for mapping relational data in Java application. It is not a framework | Hibernate is an ORM framework and in that way data persistence is possible. |
| In JPA, no implementation classes are provided. | In Hibernate, implementation classes are provided. |
| Main advantage is It uses JPQL (Java Persistence Query Language) and it is platform-independent query language. | Here it is using HQL (Hibernate Query Language). |
| It is available under javax.persistence package. | It is available under org.hibernate package. |
| In Hibernate, EclipseLink, etc. we can see its implementation. | Hibernate is the provider of JPA. |
| Persistence of data is handled by EntityManager. | Persistence of data is handled by Session. |