Spring Boot REST API with JPA: CRUD Tutorial (Teacher's Notes)

Below is a step-by-step guide to build a REST API in Java using Spring Boot and Spring Data JPA that performs basic CRUD (Create, Read, Update, Delete) operations. It includes the project structure, all key annotations, code snippets, and the reasoning behind each step.

What You'll Build

- A REST API for a simple "User" resource with endpoints:
  - POST /api/users — create
  - GET /api/users — read all
  - GET /api/users/{id} — read one
  - PUT /api/users/{id} — update
  - DELETE /api/users/{id} — delete [1] [2]

Prerequisites

- JDK 17+ (or 11 as needed)
- Maven or Gradle
- An RDBMS (H2 for quick start, or MySQL/PostgreSQL)
- IDE (IntelliJ IDEA or similar)

> For a reference end-to-end example with JPA and CRUD endpoints, see community tutorials and guides that demonstrate similar endpoints and layering. For foundational JPA usage, see the official Spring guide. For repository interface choices, see an overview of CrudRepository and JpaRepository. [3] [4] [5] [2] [1]

1) Create Project

- Use Spring Initializr:
  - Dependencies: Spring Web, Spring Data JPA, H2 (or MySQL), Validation (optional), Lombok (optional).

> Most tutorials start this way and then add Web, Data JPA, and a database driver dependency. [4] [2] [3]

2) Add Dependencies (Maven)

In pom.xml, ensure dependencies like:

- spring-boot-starter-web
- spring-boot-starter-data-jpa
- h2 (or mysql/postgresql driver)

> Spring Web exposes REST endpoints; Spring Data JPA implements repositories; the database driver provides connectivity.[2] [3] [4]

3) Configure application.properties

For H2 (in-memory, quick start):

- spring.datasource.url=jdbc:h2:mem:testdb
- spring.jpa.hibernate.ddl-auto=update
- spring.h2.console.enabled=true

For MySQL (example):

- spring.datasource.url=jdbc:mysql://localhost:3306/demo
- spring.datasource.username=root
- spring.datasource.password=yourpassword
- spring.jpa.hibernate.ddl-auto=update
- spring.jpa.show-sql=true

> Configuration enables JPA to auto-manage schema and connect to the DB as shown in CRUD examples using MySQL/PostgreSQL.[3] [1] [2]

4) Create the Entity

What it is:

- A POJO mapped to a database table.

Key annotations:

- @Entity: marks the class as a JPA entity (table mapped).[6]
- @Table(name="users"): optional, to specify table name.[6]
- @Id: marks primary key.[6]
- @GeneratedValue(strategy=GenerationType.IDENTITY): auto-increment PK.[6]

Example:

```
import jakarta.persistence.*;
```

```
@Entity                    // JPA entity mapped to a table [^4]
@Table(name = "users")     // Optional custom table name [^4]
public class User {
    @Id                                    // Primary key [^4]
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-generated ID [^4]
    private Long id;

    @Column(nullable = false)              // Not null column
    private String firstName;

    @Column(nullable = false)
    private String lastName;

    @Column(unique = true, nullable = false)
    private String email;

    // getters and setters
}
```

> These annotations are standard JPA mappings also shown in CRUD tutorials and JPA guides.[4] [6]

5) Create the Repository

What it is:

- Interface extending Spring Data repository to get CRUD methods out-of-the-box.

Options:

- CrudRepository<T,ID>: basic CRUD.[5]
- JpaRepository<T,ID>: includes CRUD plus paging/sorting and JPA-specific features.[5]

Choose JpaRepository for convenience:

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

public interface UserRepository extends JpaRepository<User, Long> {
    // Optional custom finders: e.g., Optional<User> findByEmail(String email);
}
```

> JpaRepository builds on CrudRepository and provides common operations (save, findById, findAll, deleteById) used in CRUD APIs. Tutorials rely on JpaRepository for simplicity.[1] [2] [3] [5]

6) Create the Service Layer (Optional but Recommended)

Why:

- Encapsulates business logic; controller remains thin.

Example:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
@Transactional
public class UserService {
    private final UserRepository repo;

    public UserService(UserRepository repo) { this.repo = repo; }

    public User createUser(User u) { return repo.save(u); } // create/update [^7]
    public User getUserById(Long id) {
        return repo.findById(id).orElseThrow(() -> new RuntimeException("User not found")
    }
    public List<User> getAllUsers() { return repo.findAll(); } // read all [^7]
    public User updateUser(Long id, User u) {
        User existing = getUserById(id);
        existing.setFirstName(u.getFirstName());
        existing.setLastName(u.getLastName());
        existing.setEmail(u.getEmail());
        return repo.save(existing); // save updates [^7]
    }
    public void deleteUser(Long id) { repo.deleteById(id); } // delete [^7]
}
```

> Service methods wrap repository calls like save(), findById(), findAll(), deleteById() as shown in practical guides and enabled by Crud/JpaRepository.[2] [1] [5]

7) Create the REST Controller

What it is:

- Exposes HTTP endpoints.

Key annotations:

- @RestController: REST controller responding with JSON.[3] [1] [2]
- @RequestMapping("api/users"): base path.[1] [2]
- @PostMapping, @GetMapping, @PutMapping, @DeleteMapping: HTTP verb mappings.[2] [3] [1]
- @PathVariable: binds path parameters.[1] [2]
- @RequestBody: binds JSON request body to POJO.[2] [1]

- ResponseEntity<T>: control status codes.[1] [2]

Example:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController                                    // REST controller [^6][^9]
@RequestMapping("api/users")                       // Base path [^6][^9]
public class UserController {

    private final UserService service;

    public UserController(UserService service) { this.service = service; }

    @PostMapping                                   // Create [^6][^9]
    public ResponseEntity<User> create(@RequestBody User user) {
        User saved = service.createUser(user);
        return new ResponseEntity<>(saved, HttpStatus.CREATED);
    }

    @GetMapping                                    // Read all [^6][^9]
    public ResponseEntity<List<User>> findAll() {
        return ResponseEntity.ok(service.getAllUsers());
    }

    @GetMapping("{id}")                            // Read one [^6][^9]
    public ResponseEntity<User> findOne(@PathVariable("id") Long id) {
        return ResponseEntity.ok(service.getUserById(id));
    }

    @PutMapping("{id}")                            // Update [^6][^9]
    public ResponseEntity<User> update(@PathVariable("id") Long id,
                                       @RequestBody User user) {
        return ResponseEntity.ok(service.updateUser(id, user));
    }

    @DeleteMapping("{id}")                         // Delete [^6][^9]
    public ResponseEntity<String> delete(@PathVariable("id") Long id) {
        service.deleteUser(id);
        return ResponseEntity.ok("User successfully deleted!");
    }
}
```

> This structure mirrors widely used CRUD controller patterns, including status control and path mapping , consistent with typical REST endpoints listed in similar examples.[3] [2] [1]

8) Run and Test

- Run application:
  - mvn spring-boot:run
- Test with curl or Postman:
  - POST /api/users with JSON body
  - GET /api/users
  - GET /api/users/{id}
  - PUT /api/users/{id}
  - DELETE /api/users/{id}

> Running via Maven is standard in tutorials; example commands reference this workflow. The endpoints align with CRUD API definitions shown in step-by-step guides.[3] [2] [1]

9) Common Annotations Explained

- @SpringBootApplication: main entry point (auto-configuration + component scan).
- @Entity: marks class as JPA entity mapped to a table.[6]
- @Table: optional table name configuration.[6]
- @Id: primary key field.[6]
- @GeneratedValue: configures PK generation strategy.[6]
- @Column: column constraints like nullable, unique.[6]
- @Repository (implicit with interface extension): JPA repository proxy bean.[5]
- JpaRepository / CrudRepository: exposes CRUD methods (save, findById, findAll, deleteById).[5]
- @Service: business logic layer bean.
- @Transactional: ensures DB operations execute within transactions.
- @RestController: controller returning JSON responses.[2] [1]
- @RequestMapping / @GetMapping / @PostMapping / @PutMapping / @DeleteMapping: route mapping.[3] [1] [2]
- @PathVariable: bind URL path segment.[1] [2]
- @RequestBody: bind request body JSON to an object.[2] [1]
- ResponseEntity: return body + HTTP status.[1] [2]

> The roles of these annotations and interfaces are consistent with Spring Data JPA and REST tutorials and documentation.[4] [5] [3] [2] [1] [6]

10) Optional Enhancements

- Validation: add Jakarta Bean Validation annotations (@NotBlank, @Email) and @Valid in controller.
- Exception Handling: @ControllerAdvice with @ExceptionHandler for cleaner error responses.
- DTOs and Mappers: separate API models from entities.
- Pagination/Sorting: extend JpaRepository and use Pageable in GET endpoints.[5]
- Database profile switching: use application-{profile}.properties.

> These are common improvements built on top of the same repository and controller foundations used by CRUD tutorials and repository capabilities.[5] [3] [2] [1]

Minimal Project Structure

- src/main/java/com/example/demo
  - DemoApplication.java (@SpringBootApplication)
  - entity/User.java (@Entity)[6]
  - repository/UserRepository.java (extends JpaRepository)[5]
  - service/UserService.java (business logic using repo)[2] [1]
  - controller/UserController.java (REST endpoints)[1] [2]
- src/main/resources/application.properties (DB + JPA config)[3] [2]

> This layering is the standard, clean separation of concerns found in example projects and guides.[3] [2] [1]

Quick Endpoint Reference

- POST /api/users — create a user[2] [3] [1]
- GET /api/users — list users[3] [1] [2]
- GET /api/users/{id} — get a user[1] [2] [3]
- PUT /api/users/{id} — update a user[2] [3] [1]
- DELETE /api/users/{id} — delete a user[3] [1] [2]

These match typical REST CRUD operations demonstrated in comprehensive tutorials and are powered by JPA repository methods like save, findById, findAll, and deleteById.[5] [1] [2] [3]

If code for the main class or specific configurations is needed, it can be added next.

⁂

1. https://www.javaguides.net/2022/09/spring-boot-rest-api-crud-example-with-mysql-database.html
2. https://dzone.com/articles/build-crud-restful-api-using-spring-boot-3
3. https://www.bezkoder.com/spring-boot-jpa-crud-rest-api/

4. https://spring.io/guides/gs/accessing-data-jpa
5. https://www.geeksforgeeks.org/advance-java/crudrepository-jparepository-and-pagingandsortingrepository-in-spring-data/
6. https://www.javaguides.net/2022/01/spring-data-jpa-crud-example-tutorial.html