Here's a clear, practical way to build a **REST API using Spring Boot + MySQL** that stores records with fields: `name`, `city`, `phone`.

---

1. Create Spring Boot Project

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

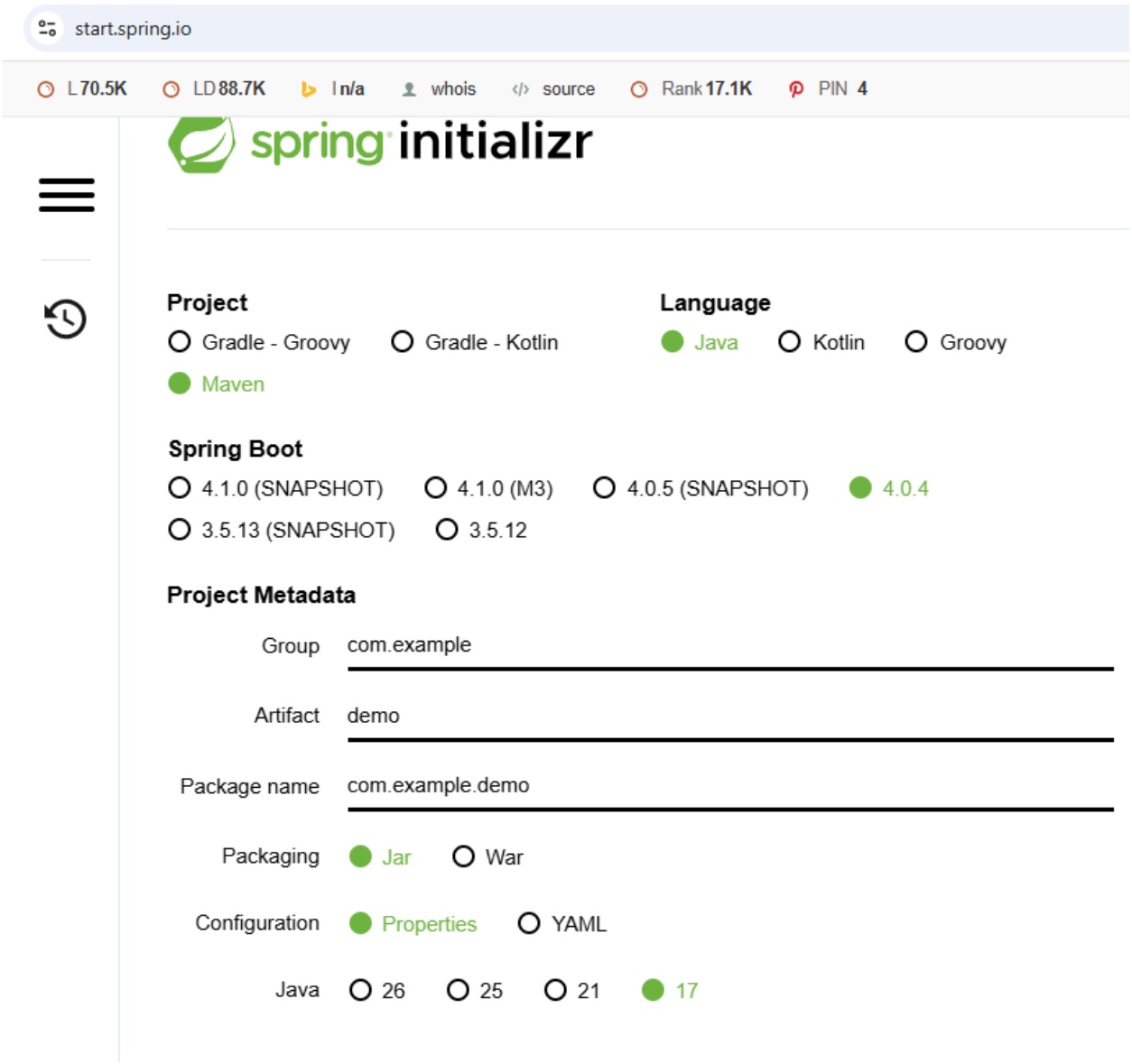
Let's create a **full Java + Maven + Spring Boot CRUD project** that stores records in a **table with columns:** **`id, name, city, phone`**. This will be ready-to-run with **MySQL**.

---

1 Project Setup on Spring Initializr

Settings:

Setting	Recommendation
Project	Maven Project
Language	Java
Spring Boot	Latest stable (e.g., 3.2.x)
Group	`com.example`
Artifact	`person-crud`
Packaging	Jar
Java	17 or 20



Dependencies:

• **Spring Web** → REST API
• **Spring Data JPA** → Database access
• **MySQL Driver** → MySQL connection
• **Lombok** (optional) → reduces getters/setters
• **Spring Boot DevTools** (optional) → auto-restart during development

Dependencies ADD DEPENDENCIES... CTRL + B

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver SQL

MySQL JDBC driver.

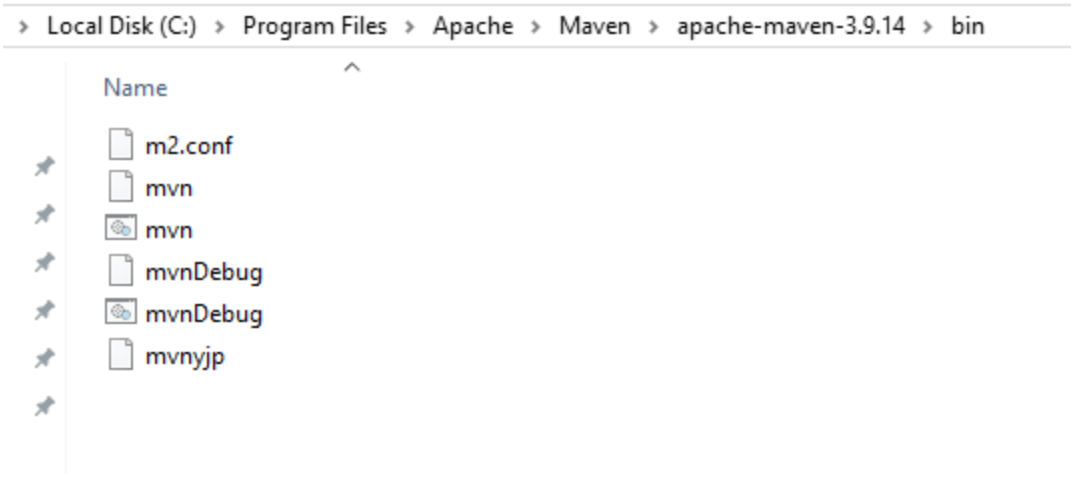
Lombok DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

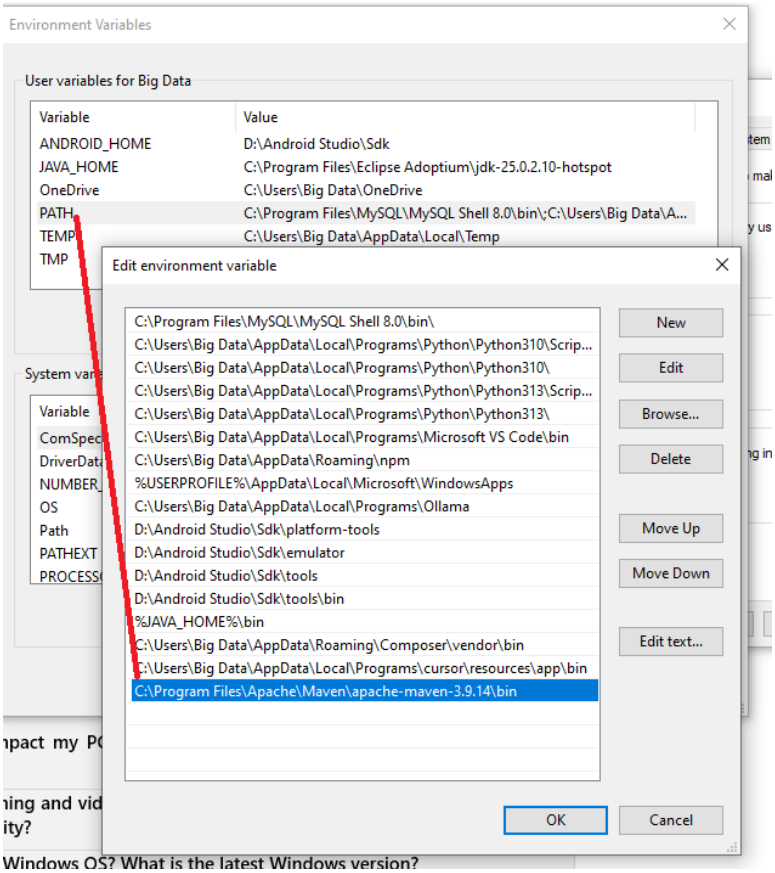
Spring Boot DevTools DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Note:- on your windows pc download Maven(Binary zip archive) from https://maven.apache.org/download.cgi and extract it inside Program Fiels/Apache/Maven as shown below.



And into your Environment Path as shown below:-

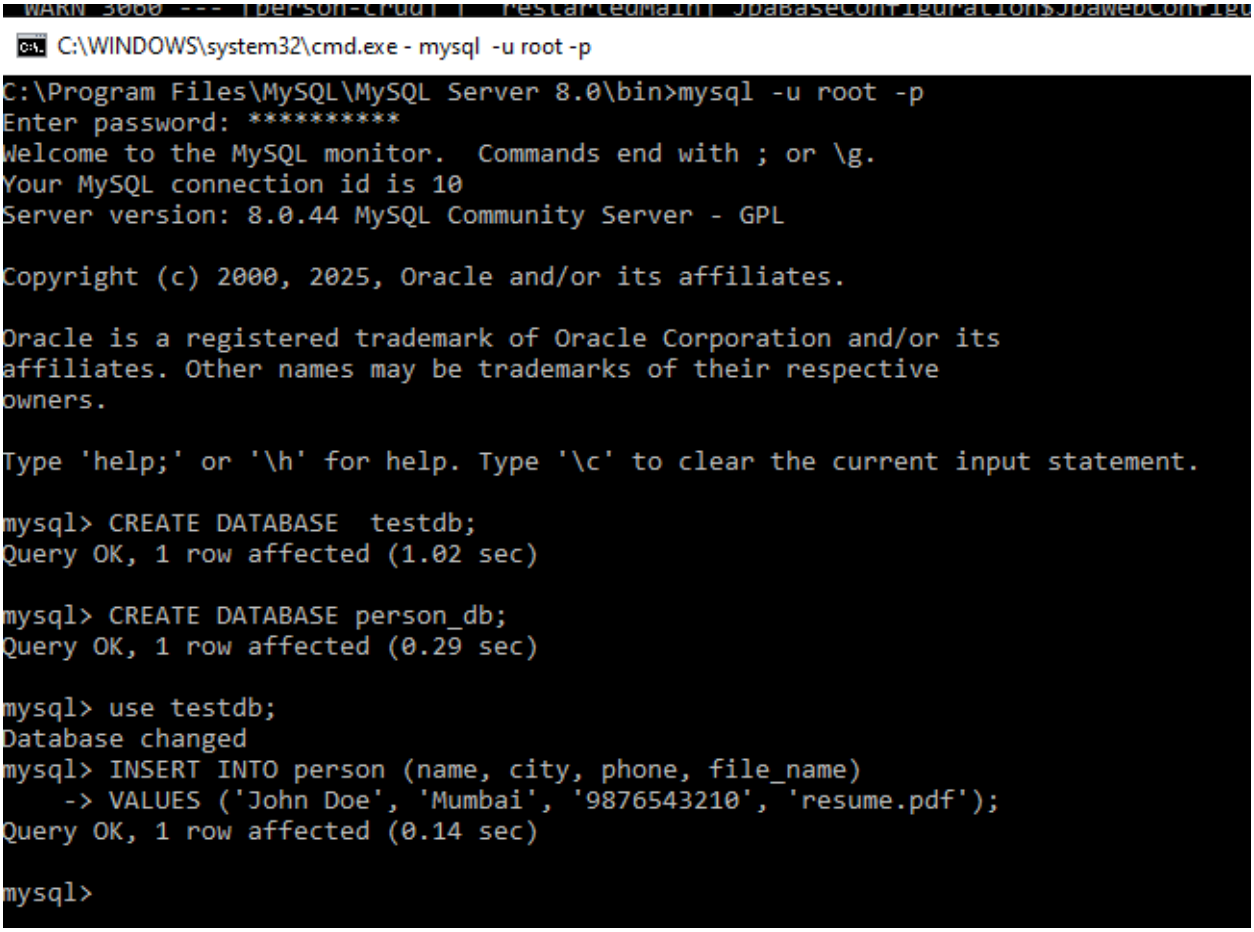


□ 2. Configure MySQL in

application.properties

```
# Application name

spring.application.name=person-crud


# MySQL datasource

spring.datasource.url=jdbc:mysql://localhost:3306/testdb?useSSL=false&serverTimezone=UTC

spring.datasource.username=root

spring.datasource.password=yourpassword

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver


# JPA / Hibernate

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect


# Server port

server.port=8080
```

□ Make sure database `testdb` exists in MySQL.


C:\WINDOWS\system32\cmd.exe - mysql -u root -p
C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql -u root -p
Enter password: **********
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 10
Server version: 8.0.44 MySQL Community Server - GPL
Copyright (c) 2000, 2025, Oracle and/or its affiliates.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql> CREATE DATABASE testdb;
Query OK, 1 row affected (1.02 sec)
mysql> CREATE DATABASE person_db;
Query OK, 1 row affected (0.29 sec)
mysql> use testdb;
Database changed
mysql> INSERT INTO person (name, city, phone, file_name)
-> VALUES ('John Doe', 'Mumbai', '9876543210', 'resume.pdf');
Query OK, 1 row affected (0.14 sec)
mysql>

1 Person Entity

```
package com.example.person_crud.entity;

import jakarta.persistence.*;

@Entity

@Table(name = "person")

public class Person {

 @Id

 @GeneratedValue(strategy = GenerationType.IDENTITY)

 private Long id;

 private String name;

 private String city;

 private String phone;

 private String fileName; // new field for file name

 // Getters and 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 getCity() { return city; }

 public void setCity(String city) { this.city = city; }

 public String getPhone() { return phone; }

 public void setPhone(String phone) { this.phone = phone; }

 public String getFileName() { return fileName; }

 public void setFileName(String fileName) { this.fileName = fileName; }

}
```

2 PersonRepository

```
package com.example.person_crud.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import com.example.person_crud.entity.Person;
public interface PersonRepository extends JpaRepository<Person, Long> {
}
```

---

3 PersonService (Full CRUD with fileName)

```
package com.example.person_crud.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import com.example.person_crud.entity.Person;
import com.example.person_crud.repository.PersonRepository;

@Service
public class PersonService {

    @Autowired
    private PersonRepository repository;

    // CREATE
    public Person savePerson(Person person) {
        return repository.save(person);
    }

    // READ ALL
    public List<Person> getAllPersons() {
        return repository.findAll();
    }

    // READ BY ID
    public Person getPersonById(Long id) {
        return repository.findById(id)
                .orElseThrow(() -> new RuntimeException("Person not found
with id: " + id));
    }

    // UPDATE
    public Person updatePerson(Long id, Person updatedPerson) {
        Person existing = getPersonById(id);
        existing.setName(updatedPerson.getName());
        existing.setCity(updatedPerson.getCity());
        existing.setPhone(updatedPerson.getPhone());
        existing.setFileName(updatedPerson.getFileName()); // update file
name
        return repository.save(existing);
```

```
    }

    // DELETE
    public void deletePerson(Long id) {
        Person existing = getPersonById(id);
        repository.delete(existing);
    }
}
```

---

4 PersonController

```
package com.example.person_crud.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.example.person_crud.service.PersonService;
import com.example.person_crud.entity.Person;
@RestController
@RequestMapping("/api/persons")
public class PersonController {

    @Autowired
    private PersonService service;

    // CREATE
    @PostMapping
    public Person addPerson(@RequestBody Person person) {
        return service.savePerson(person);
    }

    // READ ALL
    @GetMapping
    public List<Person> getAllPersons() {
        return service.getAllPersons();
    }

    // READ BY ID
    @GetMapping("/{id}")
    public Person getPersonById(@PathVariable Long id) {
        return service.getPersonById(id);
    }

    // UPDATE
    @PutMapping("/{id}")
    public Person updatePerson(@PathVariable Long id, @RequestBody Person
person) {
        return service.updatePerson(id, person);
    }

    // DELETE
    @DeleteMapping("/{id}")
    public String deletePerson(@PathVariable Long id) {
        service.deletePerson(id);
        return "Person deleted successfully!";
    }
}
```

5️⃣ Example JSON to Create / Update

```
{
  "name": "Rahul Sharma",
  "city": "Mumbai",
  "phone": "9876543210",
  "fileName": "resume.pdf"
}
```

- The `fileName` field will now be stored along with the other fields.
- Works for **create**, **update**, and you can retrieve it with **GET** methods.

✅ This is a**complete working CRUD with file name support**.

- `POST /api/persons` → create record with file name

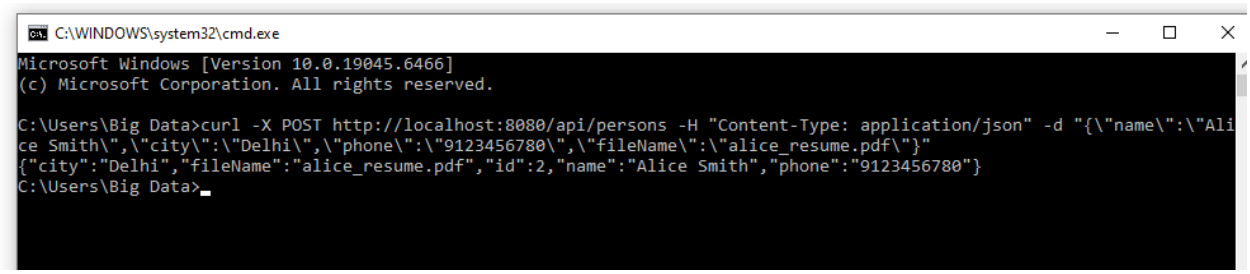
`curl` command in Windows CMD that inserts a record via your Spring Boot API with **different values**. Here's an example:

Example 1️⃣ – Insert a new person

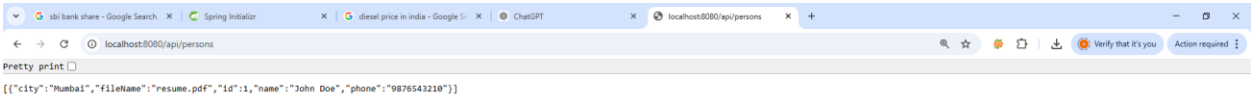
```
curl -X POST http://localhost:8080/api/persons -H "Content-Type:
application/json" -d "{\"name\":\"Alice
Smith\",\"city\":\"Delhi\",\"phone\":\"9123456780\",\"fileName\":\"alice_resu
me.pdf\"}"
```

Values changed:

- `name`: `"Alice Smith"`
- `city`: `"Delhi"`
- `phone`: `"9123456780"`
- `fileName`: `"alice_resume.pdf"`



- `GET /api/persons` → read all records



- `GET /api/persons/{id}` → read one record
- `PUT /api/persons/{id}` → update record and file name

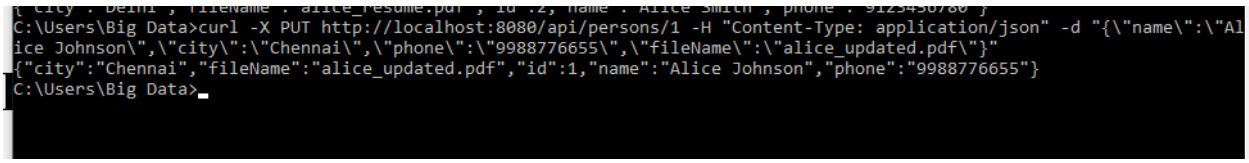
Example – Update a person

Assume you want to update **person with ID = 1**:

```
curl -X PUT http://localhost:8080/api/persons/1 -H "Content-Type:
application/json" -d "{\"name\":\"Alice
Johnson\",\"city\":\"Chennai\",\"phone\":\"9988776655\",\"fileName\":\"alice_
updated.pdf\"}"
```

Values changed:

- `name`: `"Alice Johnson"`
- `city`: `"Chennai"`
- `phone`: `"9988776655"`
- `fileName`: `"alice_updated.pdf"`



- `DELETE /api/persons/{id}` → delete record

Example – Delete a person

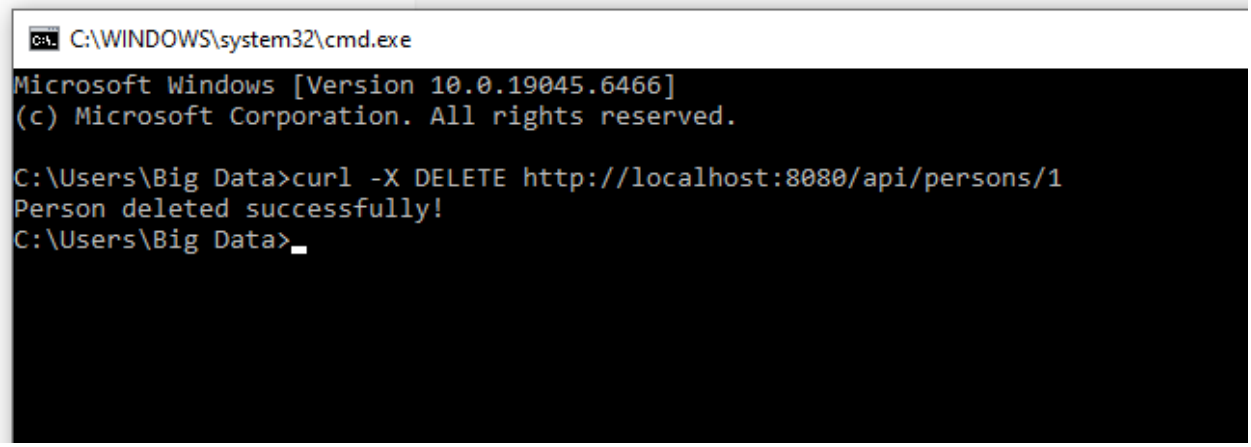
Assume you want to delete **person with ID = 1**:

```
curl -X DELETE http://localhost:8080/api/persons/1
```

Notes:

1. `-X DELETE` → tells the API to remove the resource.
2. URL must include the **person ID** (`/api/persons/1`).
3. The server will respond with a message from your controller, e.g.:

```
"Person deleted successfully!"
```



Step 4: Run the Application

Option 1: Using IDE

- Right-click on your main class (`DemoApplication.java`) → **Run 'DemoApplication'**
- You should see logs like:

```
Tomcat started on port(s): 8080
Started DemoApplication in X seconds
```

Option 2: Using Command Prompt

1. Navigate to project folder:

```
cd C:\path\to\your\project
```

2. Run with Maven:

```
mvn spring-boot:run
```

Step 5: Test the API

Use **Postman** or browser:

- **Create a record (POST)**

```
POST http://localhost:8080/api/persons
Content-Type: application/json
Body:
{
  "name": "Rahul Sharma",
  "city": "Mumbai",
  "phone": "9876543210"
}
```

- **Get all records (GET)**

```
GET http://localhost:8080/api/persons
```

- **Get record by ID (GET)**

```
GET http://localhost:8080/api/persons/1
```

- **Update record (PUT)**

```
PUT http://localhost:8080/api/persons/1
Body:
{
  "name": "Rahul Updated",
  "city": "Pune",
  "phone": "9999999999"
}
```

- **Delete record (DELETE)**

```
DELETE http://localhost:8080/api/persons/1
```

Step 6: Check MySQL

- Open MySQL Workbench or command line:

```
USE person_db;
SELECT * FROM person;
```

You should see the records created via your API.

What is JPA?

JPA (Java Persistence API) is:

- A **standard specification** in Java for **object-relational mapping (ORM)**.
- It defines **interfaces and annotations** to map Java objects (entities) to database tables.
- It allows you to **perform CRUD operations** without writing SQL manually (mostly).

Key points in your project:

- `@Entity` → tells JPA this class (`Person`) is mapped to a DB table.
- `@Id`, `@GeneratedValue` → JPA annotations for primary key and auto-increment.
- `PersonRepository extends JpaRepository<Person, Long>` → JPA repository interface gives you methods like `save()`, `findAll()`, `deleteById()`.

Think of JPA as **the contract**: "Here's how Java objects map to tables, and here's how you can query them."

---

2 What is Hibernate?

Hibernate is:

- A **popular implementation of JPA** (the actual library that does the work).
- It takes your JPA entities and **generates SQL queries** for MySQL.
- It manages **connections, transactions, caching, and schema updates**.

In your project:

- `spring.jpa.hibernate.ddl-auto=update` → Hibernate automatically **updates your MySQL tables** to match your entity classes.
- `spring.jpa.show-sql=true` → Hibernate shows all SQL it executes.
- `spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect` → Hibernate knows how to write SQL **optimized for MySQL**.

Think of Hibernate as **the engine** that actually talks to the database, while JPA is the **blueprint**.

---

3 How they work together in your project

1. You define your **Person entity** using JPA annotations.
2. Spring Boot creates a **JpaRepository** bean (interface) that gives CRUD methods.
3. When you call `service.savePerson(person)`:
    - **JPA** defines what "save" means.
    - **Hibernate** generates the SQL `INSERT` statement and executes it on MySQL.
4. When you call `service.getAllPersons()`:
    - JPA defines "findAll" behavior.
    - Hibernate translates it to `SELECT * FROM person` and fetches results.
5. Hibernate handles **transactions, caching, and connection pooling** automatically.

Summary

Concept	Role in Your Project
JPA	Standard specification for mapping Java objects to DB tables and defining repository methods.
Hibernate	Implementation of JPA that executes SQL, manages connections, and synchronizes schema.