# 🧩 1️⃣ When should you set headers in Postman?

You **set headers** whenever you send data or expect a specific data format.  
In your case (Spring Boot REST API), you must set headers for:

* **POST**
* **PUT**
* **PATCH**

Because in these requests, you are **sending data** (usually JSON) to the backend.

✅ **You do NOT need to set headers** for:

* **GET**
* **DELETE**

since those requests don’t have a body (you’re just fetching or deleting).

# 🧠 2️⃣ Why do we set headers?

Think of headers as **instructions or labels** you attach to your request —  
they tell the server what kind of data you are sending and what response you want back.

In simple words:

“Headers are like the tags on a parcel — they tell the receiver what’s inside and how to handle it.”

# ⚙️ 3️⃣ The Two Most Important Headers for Spring Boot APIs

| **Header Name** | **Value** | **Why It’s Needed** |
| --- | --- | --- |
| Content-Type | application/json | It tells Spring Boot that the data you are sending in the **body** is JSON format. This allows Spring’s **Jackson** library to automatically convert that JSON into a Java object (like Student). |
| Accept | application/json | It tells the backend that you want the **response** also in JSON format. Spring Boot will format the response accordingly. |

# 💡 Simple Answer (in one line):

👉 You set headers **only when your request or response involves a specific data format or authentication**.

That means:

* If you **send JSON data**, you must set headers.
* If you **expect JSON data back**, you should set headers.
* If your API **requires authentication or tokens**, you must set headers.
* For simple GET or DELETE requests (no data body), you usually **don’t need** headers.

# 🧠 Let’s break it by HTTP method

| **HTTP Method** | **Do we set headers?** | **Why** |
| --- | --- | --- |
| **GET** | ❌ Usually not needed | You’re just requesting data, no body is sent. |
| **DELETE** | ❌ Usually not needed | You’re only deleting by ID, no data is sent. |
| **POST** | ✅ Yes, always | You’re sending data (like JSON or form), so you must tell server the format. |
| **PUT** | ✅ Yes, always | You’re sending data to replace an existing record. |
| **PATCH** | ✅ Yes, always | You’re sending partial data to update an existing record. |

# 🧠 1️⃣ For POST / PUT / PATCH

Whenever you send JSON data in the request body,  
add these **two headers**:

| **Header** | **Value** | **Why** |
| --- | --- | --- |
| Content-Type | application/json | Tells backend: “I’m sending JSON data.” |
| Accept | application/json | Tells backend: “I expect JSON data in response.” |

✅ **You must set these in Postman:**

1. Go to the **Headers** tab
2. Add:
3. Content-Type: application/json
4. Accept: application/json
5. Go to **Body → raw → JSON**, and send your data.

# 🧪 Example (POST Request)

**URL:**

http://localhost:8080/students

**Headers:**

| **Key** | **Value** |
| --- | --- |
| Content-Type | application/json |
| Accept | application/json |

**Body (raw JSON):**

{

"id": 101,

"name": "Pavan",

"email": "pavan@gmail.com",

"city": "Hyderabad"

}

✅ This is required because you are **sending JSON** to Spring Boot.

# 🧠 2️⃣ For GET Request

When you just want to **fetch data**, no body is sent.

Example:

GET http://localhost:8080/students/101

👉 You don’t need to set Content-Type, because you’re not sending data.  
But if you want to be consistent, you **can** still set:

Accept: application/json

This tells the backend that you expect a **JSON response**.

# 🧠 3️⃣ For DELETE Request

Example:

DELETE http://localhost:8080/students/101

No request body — so no Content-Type needed.

Optionally:

Accept: application/json

if you expect JSON confirmation.

# 🧠 4️⃣ When Authentication is Involved

If your API requires login, tokens, or API keys,  
you’ll set extra headers like:

| **Header** | **Example Value** | **Purpose** |
| --- | --- | --- |
| Authorization | Bearer eyJhbGciOiJIUzI1Ni... | For JWT authentication |
| x-api-key | 12345-ABCDE | For API key authentication |
| Cookie | JSESSIONID=... | For session-based login |

You’ll learn this once you reach **secured REST APIs**.

# 🧭 5️⃣ Summary Table (Most Common Cases)

| **Type of Request** | **Required Headers** | **Example** |
| --- | --- | --- |
| POST | Content-Type: application/json & Accept: application/json | Sending JSON to create record |
| PUT | Same as POST | Sending full record update |
| PATCH | Same as POST | Sending partial update |
| GET | Optional Accept: application/json | Fetching data |
| DELETE | Optional Accept: application/json | Deleting data |
| Auth APIs | Authorization: Bearer <token> | Secured APIs |

# 🎯 Final Takeaway

✅ **Set headers in Postman when:**

1. You send data (POST, PUT, PATCH) → Content-Type
2. You expect data format (like JSON) → Accept
3. You deal with security (tokens, API keys) → Authorization

❌ **Don’t set headers** for simple GET or DELETE unless your API needs them.

**🧩 @DeleteMapping — Complete Explanation (With Real Reasoning)**

**🌱 1️⃣ What is @DeleteMapping?**

**@DeleteMapping** is a Spring Boot annotation that is used to **delete an existing resource (record)** from the backend database through a RESTful API.

It represents the **HTTP DELETE** method in REST architecture.

**💭 2️⃣ Why Do We Need @DeleteMapping?**

Let’s first understand what problem it solves.

**🔹 Problem (Before REST or Without @DeleteMapping)**

Earlier, before REST-style APIs were popular:

* Developers used only POST for *everything* (creating, updating, deleting, etc.).
* Clients had to send **extra instructions** in the body to say what they wanted to do, like:
* {
* "action": "delete",
* "studentId": 101
* }
* This was **confusing**, **non-standard**, and **less secure**, because the meaning of each request wasn’t clear.

Example (Old way):

POST /student

Body: { "action": "delete", "id": 101 }

👉 Problem:  
The same /student URL was used for *create*, *update*, *delete*, which made it hard to maintain and test.

**🔹 Solution (With REST & @DeleteMapping)**

Then REST introduced a **clear separation of actions using HTTP methods**:

| **Action** | **HTTP Method** | **Example Endpoint** |
| --- | --- | --- |
| Create | POST | /students |
| Read | GET | /students/101 |
| Update | PUT/PATCH | /students/101 |
| Delete | DELETE | /students/101 |

✅ Now the **intent is clear** just from the method name — no confusion!

When the client sends a **DELETE** request,  
Spring knows:

“Oh, this means the client wants to remove something permanently.”

So, @DeleteMapping provides a **clean, meaningful, and REST-compliant** way to delete data.

**🌟 3️⃣ Advantages of @DeleteMapping**

| **Advantage** | **Explanation** |
| --- | --- |
| **1. Clarity of Purpose** | The method name DELETE itself shows what the operation does — no confusion like before. |
| **2. RESTful Design** | Follows REST standards where each HTTP method has a defined meaning. |
| **3. Cleaner Code** | You don’t need to use conditional logic in POST methods to handle delete operations. |
| **4. Easier Testing** | Tools like Postman can easily identify CRUD operations based on HTTP methods. |
| **5. Security and Control** | Delete operations can be specifically restricted or logged since they use a separate endpoint. |
| **6. No Request Body Needed** | It needs only an identifier in the URL, making requests lighter and faster. |

**⚙️ 4️⃣ How It Works Internally (Spring View)**

1. When the client sends
2. DELETE /students/101

Spring Boot looks for a controller method annotated with:

@DeleteMapping("/{id}")

1. It extracts the **path variable** (101) and sends it to the method parameter.
2. Inside the method, the record is removed from the database.
3. Spring sends back a response (message or status code).

**🧩 5️⃣ Example — Full Real-Time Demo**

Let’s take a real example of deleting a **Student record**.

**🧠 Step 1: Model (Student.java)**

package in.orcas.model;

public class Student {

private Integer id;

private String name;

private String email;

private String city;

public Student() {}

public Student(Integer id, String name, String email, String city) {

this.id = id;

this.name = name;

this.email = email;

this.city = city;

}

// Getters and Setters

}

**⚙️ Step 2: Service Layer (StudentService.java)**

package in.orcas.service;

import java.util.HashMap;

import java.util.Map;

import org.springframework.stereotype.Service;

import in.orcas.model.Student;

@Service

public class StudentService {

private Map<Integer, Student> db = new HashMap<>();

// Save student (for testing)

public String saveStudent(Student student) {

db.put(student.getId(), student);

return "Student saved successfully!";

}

// Delete student by ID

public String deleteStudent(Integer id) {

if (db.containsKey(id)) {

db.remove(id);

return "Student deleted successfully with ID: " + id;

} else {

return "Student not found with ID: " + id;

}

}

// View all students (optional)

public Map<Integer, Student> getAll() {

return db;

}

}

**🎯 Step 3: Controller (StudentController.java)**

package in.orcas.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import in.orcas.model.Student;

import in.orcas.service.StudentService;

@RestController

@RequestMapping("/students")

public class StudentController {

@Autowired

private StudentService service;

// POST - Create new student

@PostMapping

public String createStudent(@RequestBody Student student) {

return service.saveStudent(student);

}

// DELETE - Remove student by ID

@DeleteMapping("/{id}")

public String deleteStudent(@PathVariable Integer id) {

return service.deleteStudent(id);

}

// GET - Optional (view all)

@GetMapping

public Object getAll() {

return service.getAll();

}

}

**🧪 Step 4: Testing in Postman**

**1️⃣ Create student (POST)**

**URL:**

http://localhost:8080/students

**Headers:**

Content-Type: application/json

Accept: application/json

**Body:**

{

"id": 101,

"name": "Pavan",

"email": "pavan@gmail.com",

"city": "Hyderabad"

}

✅ Response:

Student saved successfully!

**2️⃣ Delete student (DELETE)**

**URL:**

http://localhost:8080/students/101

**Method:** DELETE  
**No Body Needed**  
**Optional Header:**

Accept: application/json

✅ Response:

Student deleted successfully with ID: 101

If you try again:

Student not found with ID: 101

**🔒 6️⃣ In Real-Time Projects**

In real-world enterprise APIs, delete operations are often handled carefully.

**🔹 Types of Delete:**

1. **Hard Delete:**  
   Permanently removes the record from the database.  
   ➤ Risk: Data can’t be recovered.
2. **Soft Delete:**  
   Instead of removing, mark record as deleted. Example:
3. UPDATE students SET status = 'INACTIVE' WHERE id = 101;

➤ Advantage: You can restore data later.

Spring developers often prefer **soft delete** to avoid accidental data loss.

**🧭 7️⃣ REST Standard & HTTP Status Codes**

When deleting, always send **proper response codes**:

| **Status** | **Meaning** |
| --- | --- |
| 200 OK | Deleted successfully, message returned |
| 204 No Content | Deleted successfully, no message needed |
| 404 Not Found | Record doesn’t exist |
| 400 Bad Request | Invalid request format |

Example with ResponseEntity:

@DeleteMapping("/{id}")

public ResponseEntity<String> deleteStudent(@PathVariable Integer id) {

String result = service.deleteStudent(id);

if (result.contains("not found")) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(result);

} else {

return ResponseEntity.status(HttpStatus.OK).body(result);

}

}

**🧩 8️⃣ Without @DeleteMapping — What Goes Wrong?**

If you try to delete data using POST or GET instead:

* The **meaning of API becomes unclear** (violates REST principles).
* It becomes **hard to maintain** (every action mixed in same endpoint).
* You **cannot use caching, proxy, or security filters** properly — because they rely on HTTP methods.
* Your API **fails REST validation** (bad for third-party integrations).

Example of bad design:

POST /students/delete?id=101

This is not RESTful — because POST should create, not delete.

**🧩 9️⃣ Summary Table**

| **Concept** | **Meaning** |
| --- | --- |
| Annotation | @DeleteMapping |
| Purpose | To delete a resource from backend |
| HTTP Method | DELETE |
| Data Input | Path variable (usually ID) |
| Request Body | ❌ Not needed |
| Headers | Optional (Accept: application/json) |
| Common Response | Text or ResponseEntity with status |
| Real Use | Hard or Soft delete in CRUD apps |
| Benefit | Clear, standard, and RESTful operation |

**✅ Final Takeaway**

@DeleteMapping makes delete operations **clear**, **safe**, and **standardized**.  
Without it, APIs become **messy and non-standard**.  
With it, REST APIs stay **organized, meaningful, and predictable**.

Eg: DeleteMapping