Unit-5

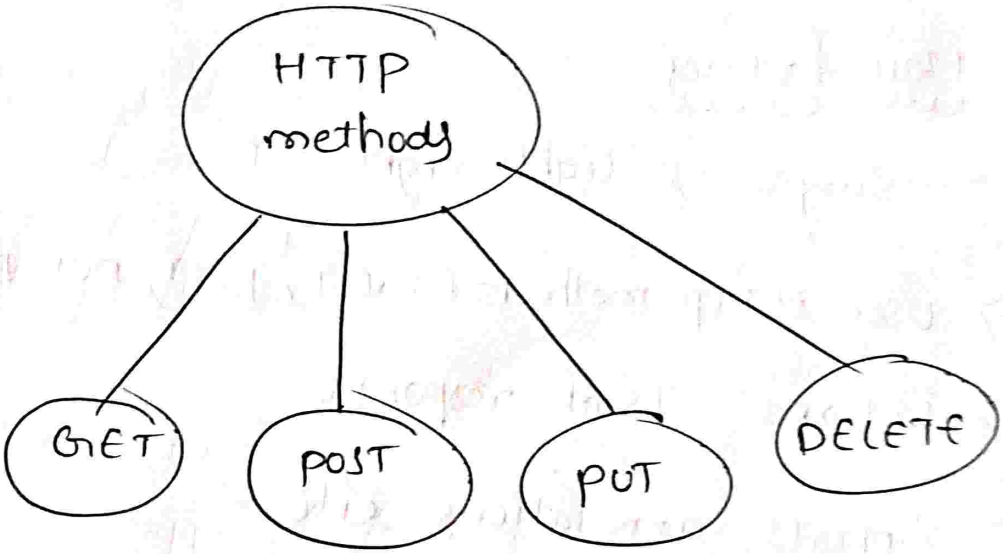
* Introduction to Spring Rest:

G. Paramesh
Asst. professor

REST:

→ REST (Representational State Transfer) is an architectural Style used to design web Services.

→ It allows communication between client and Server using Standard HTTP methods like



GET → Retrieve Data

POST → Create Data

PUT → update Data

DELETE → Delete Data

Spring REST API:

→ Spring REST API is built using Spring Boot and Spring MVC, which helps in developing RESTful web Services. Easily.

→ It allows you to create APIs that return data in JSON (or) XML format.

Main features

→ Simple & lightweight

→ Uses HTTP methods (GET, POST, PUT, DELETE)

→ Returns JSON responses

→ Supports annotations like
- ↳ @RestController
- ↳ @GetMapping
- ↳ @PostMapping

→ Integrates easily with databases and front-end apps.

* Creating a Spring REST Controller:

→ In a Spring Boot application, a REST Controller is used to handle HTTP requests and send HTTP responses.

→ It allows you to create RESTful APIs that can send (or) receive data in JSON (or) XML format.

Steps to create a REST Controller:

Step 1: Create a Spring Boot project

→ Use Spring Intializr (or) your IDE to create a project with:

↳ Spring web dependency

↳ Spring Boot Starter.

Step 2: Create a Model class

Example

```
public class Student
{   private int id;
    private String name;
    private String course;

}
```

Step 3: Create a REST controller

Example

```
package com.example.demo.controller;

import org.springframework.web.bind.annotation.*;
import java.util.*;

@RestController
@RequestMapping("/students")
public class StudentController
{ //Temporary List of Students
private List<String> students = new ArrayList<>(Array.asList(
                                "Aruna", "Paramesh", "Vaibhav"));

    // GET - Get all Students
    @GetMapping
public List<String> getAllStudents()
    {
        return students;
    }
        // POST - Add newStudent
        @PostMapping
public String addStudent(@RequestParam String name)
    {
        students.add(name);
        return "Student added:" + name;
    }
```

```
//DELETE - Remove Student by index
@DeleteMapping("/{index}")
public String deleteStudent(@PathVariable int index)
    {
String removed = Students.remove(index)
    {
return "Student deleted:" + removed;
    }
}
```

* @RequestBody and ResponseEntity:

@RequestBody

→ Annotation in Spring used to map the incoming HTTP request body (like JSON (or) XML) directly into a Java object.

→ It's placed on a method parameter inside a controller.

→ Spring automatically desirializes the request JSON into the specified Java object.

Example: (Java)

```
@PostMapping("/addEmployee")
Public String addEmployee(@RequestBody Employee employee)
    {
    return "Employee.getName() + "added Successfully";
    }
```

If a client sends: (JSON)

```
{ "id": 1,
"name": John Doe",
"department": "CSE"
} // spring will map it into Employee object
```

@ResponseEntity

→ A class that represents the entire HTTP response.

(i) Body (The actual data we send back)

(ii) Status code (Eg: 200 ok, 201 CREATED, 404 NOT FOUND)

(iii) Headers (optional)

Example: (Java)

```
@PostMapping("/addEmployee")
public ResponseEntity<String> addEmployee(@RequestBody
                                          Employee employee)
    //Logic
{
    return ResponseEntity
            .status(HttpStatus.CREATED)
            .body("Employee " + employee.getName() +
                  " added Successfully!");
}
```

→ O/P

Status code: 201 CREATED

Response Body:

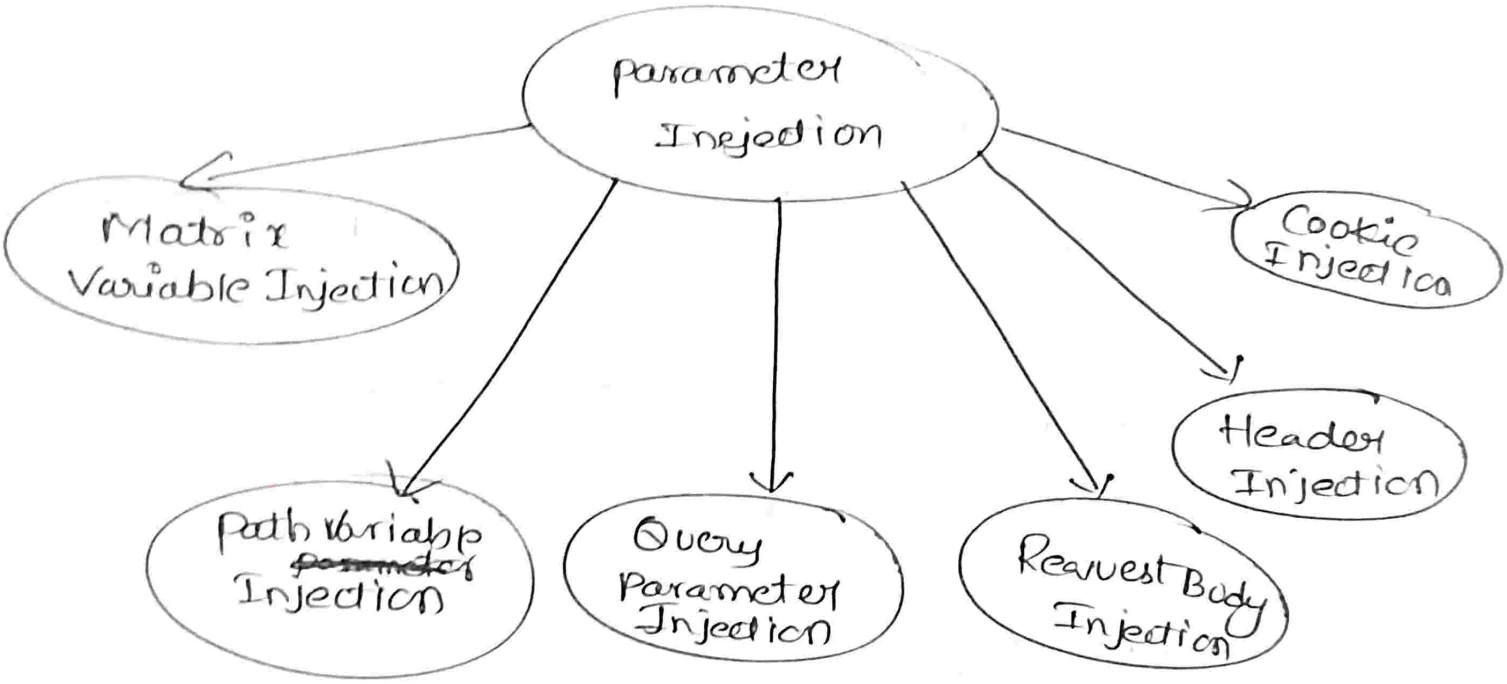
"Employee Vaishnav added Successfully!"

Parameter Injection :

→ process of passing client request data into API controller methods.

→ Spring Boot automatically maps request values (query params, path variables, headers, body) to method parameters using annotations.

Types of parameter Injection :



(1) Path variable Injection (@pathVariable)

→ Extracts values from URL path.

Ex: @GetMapping("/employee/{id}")

```
public String getEmployee @pathVariable int id)
{
    return "Employee ID: "+id;  }
```

(2) Query Parameter Injection: (@RequestParam)

Extracts key-value pairs from URL Query String

Ex

```
@GetMapping("/search")
public String search(@RequestParam String name,
                     @RequestParam(defaultValue="10")
                     int limit)
{
    return "search for:"+name+"Limit:"+limit
}
```

(3) Request Body Injection: (@RequestBody)

→ Maps JSON/XML body to Java object.

Ex: @PostMapping("/add")

```
public String addEmployee(@RequestBody Employee emp
{
    return "Added:"+emp.getName();
}
```

(4) Header Injection (@RequestHeader)

→ Extracts value from HTTP headers.

Ex:

```
@GetMapping("/secure")
public String secure(@RequestHeader("Authorization")
                     String token)
{
    return "Token: " + token;
}
```

(5) Cookie Injection (@CookieValue)

→ Read cookies sent by client

Ex:

```
@GetMapping("/welcome")
public String welcome(@CookieValue("sessionId") String sessi
{
    return "Session: " + sessionId;
}
```

(6) Matrix Variable Injection (@MatrixVariable)

→ values embedded in path segment.

Ex

```
@GetMapping("/cars/{brand}")
public String getCars(@PathVariable String brand
        @MatrixVariable int year)
{
return brand + "cars of year " + year;
}
```

* Usage of @PathVariable:

-> @PathVariable is used in Spring REST controllers to extract values from the URL path.

-> It binds the URI template variable to a method parameter in your controller.

Example:

-> We want to fetch a specific note by StudentID from the URL

Example (1): Get Note by ID

```
@RestController
@RequestMapping("/api/notes")
public class StudentNotesController
{
//Example: http://Localhost:8080/api/notes/101
@GetMapping("/{noteId}")
public String getNoteById(@PathVariable("noteId") int id)
{
return "Fetching note details for Note ID:" + id;
}
}
```

→ @GetMapping("/{noteId}") → declares a path variable named noteId.

→ @PathVariable("noteId") → binds that part of the URL to the Java method parameter id.

Example ②

→ Multiple path variables

// Example: http://localhost:8080/api/students/notes/10

```
@GetMapping("/students/{studentId}/notes/{noteId}")
public String getStudentNote(@PathVariable int studentId,
                             @PathVariable int noteId)
{
return "Note ID" + noteId + " for Student ID " + studentId;
}
```

* @RequestParam and @MatrixVariable:

@RequestParam:

→ @RequestParam is used in Spring to extract values from the Query String of a URL and pass them as method parameters in your REST API.

→ It helps you read data sent by the client through the URL parameters.

Example URL

http://localhost:8080/api/Student?id=101&name=vaibhav

Program:

```
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/api")
public class StudentController
{
@GetMapping("/Student")
public String getStudent(@RequestParam String name)
{
return "Student ID:" + id + ", Name:" + name;
}
}
```

Output

Student ID:101, Name:Vaibhav

@MatrixVariable :

→ @MatrixVariable is used in Spring REST to extract values from the URI path segment - not from the Query String.

→ Matrix variables are written after a Semicolon(;) in the URL.

Example URL

→ http://localhost:8080/api/student;id=101;name=Ravi

Here id=101
name = vaishnav
are matrix variables.

Program

```
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/api")
public class StudentMatrixController
{
@GetMapping("/student")
public String getStudent(@MatrixVariable int id,
@MatrixVariable String name)
{
return "Student ID:" + id + ", Name:" + name;
}
}
```

O/P: Student ID: 101 Name: vaishnav

* Data validation:

→ Data validation means checking whether the input data given to an API is correct, complet and safe before processing.

→ It protects the application from wrong data, errors and security risks.

(1) Data validation important (why):

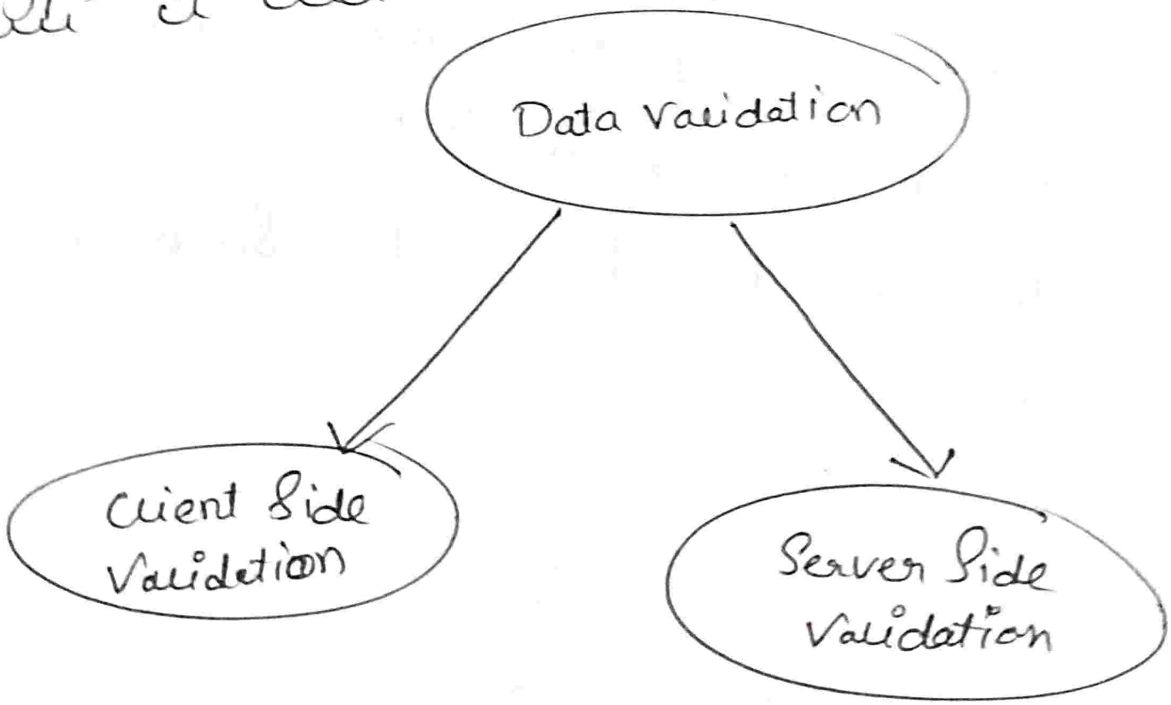
→ prevents invalid data from entering the systems

→ Reduces runtime errors in applications.

→ protects against security attacks (SQL Injection)

→ Ensures data consistency and reliability

Types of Data validation:



(i) Client-Side validation:

→ Done on the frontend (browser, mobile app)

Example:

→ Checking if email format is correct before sending request.

→ Advantage: Quick feedback to user.

→ Limitation: Can be bypassed, So Server-Side validation is still needed.

(ii) Server Side validation:

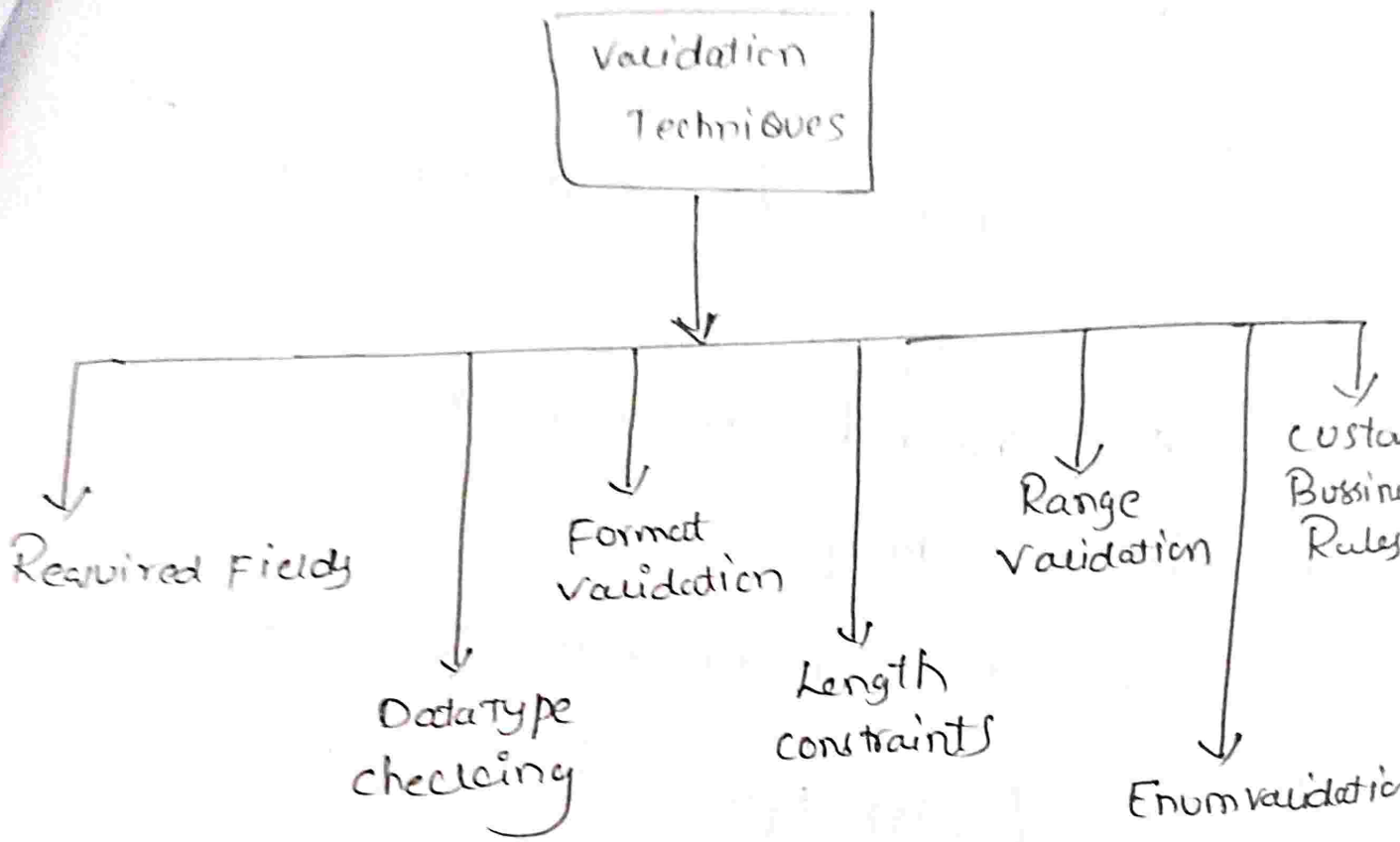
→ Done on ~~background~~ Backend (API).

→ Example:

checking if phone number has 10 digits before saving in DB.

→ More reliable because the Server controls the Logic.

Validation Techniques in APIs:



① Required Fields:

→ Make sure important fields (like Username, password) are not empty.

② Data Type checking:

→ Numbers should be numbers, dates should be valid dates.

③ Format validation:

→ Email, phone, ZIP code formats must be correct

④ Length Constraints:

→ password must be atleast 8 characters, etc

⑤ Range validation:

→ Age should be between 18-60 etc.

⑥ Enum validation:

→ only allow valid values (Ex: Gender = Male/Female/Other)

⑦ Custom Business Rules:

Example — Salary must be greater than minimum wage.

Data validation in Spring Boot API :

→ Spring Boot provides Hibernate Validator.

We use annotations in model classes.

Example

```
import jakarta.validation.constraints;

public class User
{
@NotNull(message = "Name cannot be null")
  private String name;
@Email(message="Invalid Email format")
  private String email;
@Size(min=8, message="password must be atleast 8 char
  private String password;

  @Min(16)
  @Max(80)
  private in age;
}
```

* Creating a REST client in API:

→ A REST client is an application (or) component that sends requests to a RESTful API and receives responses.

Basic Steps to create a REST client:

1. Understand the REST API Endpoint

→ Example:

https://api.example.com/students

→ Supports HTTP methods like

① GET → Fetch data
② POST → Add data
③ PUT → update data
④ DELETE → Remove data

2. Choose a REST client Tool (or) Library

→ Tools: postman, curl

→ Java libraries: Rest Template, web client, okHTTP

→ phyton: requests

→ Javascript: fetch, axios

* Example:

```
import org.springframework.web.client.RestTemplate;

public class RestClientExample
{
    psvm (String args[])
    {
String url = "https://api.example.com/students";

RestTemplate restTemplate = new RestTemplate();

    // Get request

String response = restTemplate.getForObject(url, String.cl

    System.out.println (response);

    }
}
```

Table:

HTTP Method	purpose	Example URL
GET	Read data	/students
POST	Create new record	/students
PUT	update existing	/students/1
DELETE	Remove record	/students/1

→ A Rest client is like a messenger that asks for or Sends information to another System using HTTP method Such as GET (or) POST.

* Versioning a Spring REST Endpoint:

→ API versioning means maintaining multiple versions of the same API so that older clients continue to work even after new changes are added.

Example

→ If you have /api/students, a new version might be /api/v2/students.

Why versioning is Needed:

Reason	Explanation
- Backward Compatibility	- old clients can still use old versions.
- Smooth updates	- you can release new features without breaking old code.
- Flexibility	- Developers can test new API versions safely.

Example

→ URI Versioning (Most common)

Version is part of the URL

```
@RestController
@RequestMapping("api/v1/Students")
Public Class StudentControllerV1
  {
   @GetMapping
  public String getStudentsV1()
    {
     return "Student API -version 1";
    }
  }
    @RestController
    @RequestMapping("api/V2/Students")
   public class StudentControllerV2
      {  @GetMapping
       public String getStudentsV2()
       {
return "Student API -version 2 (with extra fieldy)";
       }  }
```

output:

/api/v1/students → old version

/api/v2/students → New version

* Enabling CORS in Spring REST:

→ CORS (Cross-origin Resource sharing) allows a web application running on one domain (Eg: http://localhost:3000) to access a REST API running on another domain (eg: http://localhost:8080)

→ By default browser that it's safe to share data between different origins.

Why Enable CORS

Reason	Explanation
– Security	– Allows safe sharing of data across domains.
– Integration	– Needed when frontend (React, Angular) and backend (Spring Boot) are on different servers.
– Development	– Common in microservices and APIs used by multiple apps.

Example

→ Enable CORS at Controller Level

Use @CrossOrigin annotation in your Controller class (or) method.

```
@RestController
@RequestMapping("/api/Students")
@CrossOrigin(origins = "http://localhost:3000")
public class StudentController
{
    @GetMapping
    public String getStudents()
    {
    return "CORS Enabled for localhost:3000";
    }
}
```

→ @CrossOrigin - Enables CORS for that endpoint

→ origins defining which frontend URL can access this API.

Example // Define users (In-Memory Authentication Example)

```
@Bean
public UserDetailsService users()
    {
    UserDetails admin = user.buider()
            .username("admin")
            .password ("{noop} password ")
            .roles ("ADMIN")
            .build ();

    UserDetails User = User.builder()
            .username("user")
            .password ("{noop} 1234 ")
            .roles ("USER")
            .build ();

return new InMemoryDetailManager(admin, user);
    }
```

* Security Spring rest endpoints:

→ REST API's expose Sensitive data (like User details, Payments, etc)

→ without Security, anyone can misuse them.

(1) Common Security Threats in API's:

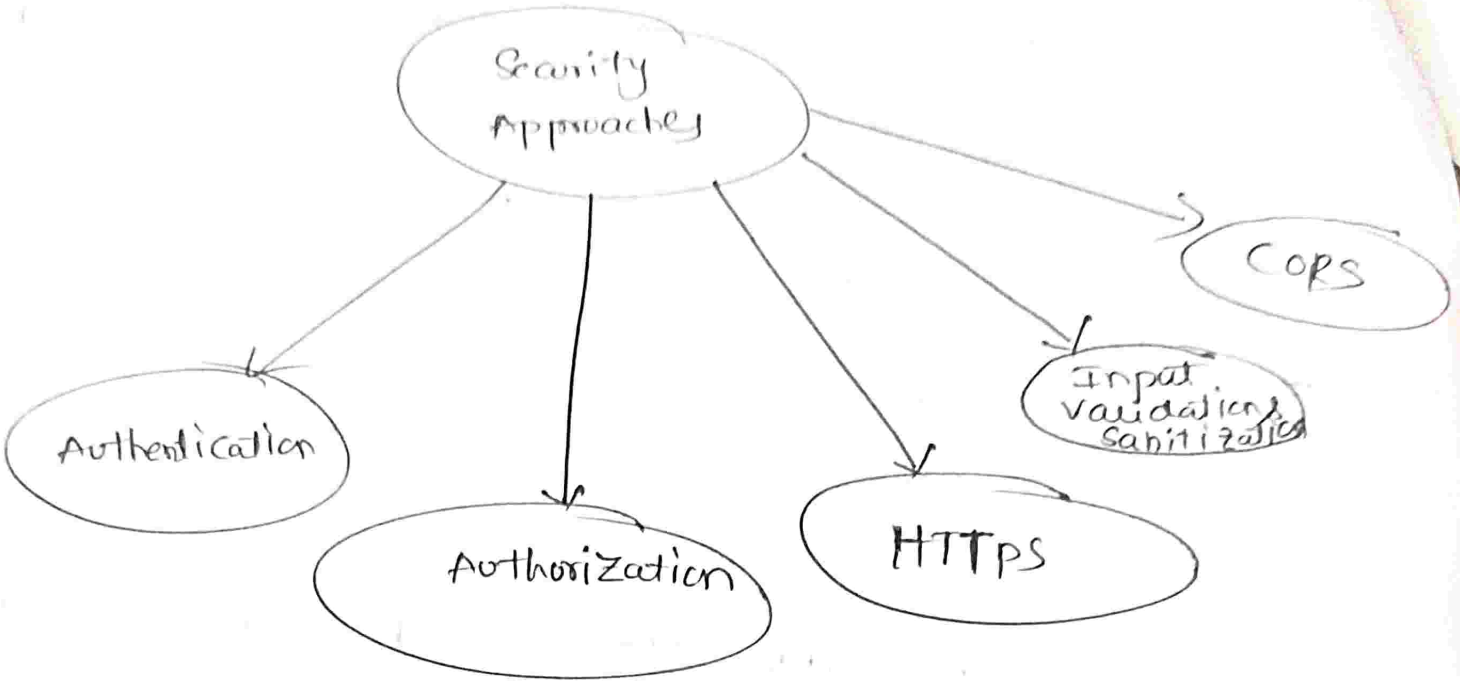
→ Unauthorized Access - attackers access data without log in.

→ Man-in-the-Middle (MITM) - data intercepted between client & Server.

→ Injection Attacks - SQL/Script injection in API requests.

→ Data Leakage - exposing unnecessary data in responses.

(2) Security Approaches in Spring Boot REST API's: (2)



(1) Authentication:

Verifies who the user is

→ Basic Authentication - Username & password Sent in header (Less Secure)

→ JWT (JSON Web Token) - Commonly used for Stateless authentication.

→ OAuth2 - Secure protocol for delegated ~~fosing~~ access (used in Google, Facebook login

(2) Authorization:

Checks what the User can do

① Role-based access (Example: ADMIN vs USER)

② Method level Security with Annotations

(1) @PreAuthorize("hasRole('ADMIN')")

(2) @Secured("ROLE USER")

(3) HTTPS (Transport Security)

→ Use HTTPS instead of HTTP to encrypt Communication.

(4) Input validation & Sanitization:

→ validate API requests data before processing.

→ prevents SQL injection, XSS.

(5) CORS (Cross-Origin Resource Sharing)

→ Restrict which frontend/domains can access the API

(3) Spring Security Features:

→ Spring Security provides:

↳ Authentication & Authorization

↳ password encoding (eg: BCryptpassword Encoder)

↳ Security filters for requests.

Example:

```
import java.lang.*;

@EnableWebSecurity

Public class SecurityConfig extends WebSecurityConfigurerAdapter
{
    @override

protected void Configure(HttpSecurity http) throws Exception
        {
            http
            .csrf().disable()
            .authorizeRequests()
            .antMatchers("api/public/**").permitAll()
            .antMatchers("api/admin/**").hasRole
                                        ("ADMIN")
            .HttpBasic();
        }
}
```