1️⃣ @RestController

- Combines @Controller and @ResponseBody.
- Tells Spring Boot:
  “This class will handle HTTP requests and return data directly in the response body (usually JSON).”
- Without it, your class would not be recognized as a REST endpoint provider.

Example:
When you return a cloudVendor object, Spring automatically converts it into JSON — thanks to @RestController.

2️⃣ @RequestMapping("/cloudvendor")

- Defines the base URL path for all endpoints inside this controller.
- This means every mapping in this class will start with /cloudvendor.

So:

- `GET /cloudvendor/{vendorID}`
- `POST /cloudvendor`
- `PUT /cloudvendor`
- `DELETE /cloudvendor/{vendorID}`

will all map to methods inside this class.

🧺 Inside the class

```
cloudVendor cloudvendor;
```

- This is just a variable (temporary storage).
- It’s not connected to a database — so data resets when you restart the app.

✳️ GET Mapping

```
@GetMapping("/{vendorID}")
public cloudVendor getCloudVendorDetials(@PathVariable String vendorID) {
    return cloudvendor;
}
```

3️⃣ @GetMapping("/{vendorID}")

- Maps HTTP GET requests to this method.
- `/{vendorID}` means the endpoint expects a **path variable** (like `/cloudvendor/C1`).

4️⃣ `@PathVariable`

- Extracts the value from the URL and passes it to the method.
- Example:

```
GET http://localhost:8080/cloudvendor/C1
```

→ Spring sets vendorID = "C1"

Currently, the method just returns the stored cloudvendor object.

✳ POST Mapping

```
@PostMapping
public String createCloudVendorDetials(@RequestBody cloudVendor cloudvendor) {
    this.cloudvendor = cloudvendor;
    return "Cloud Vendor Created Sucessfully";
}
```

5️⃣ @PostMapping

- Maps HTTP POST requests (used for creating resources).
- Example endpoint:

```
POST http://localhost:8080/cloudvendor
```

6️⃣ `@RequestBody`

- Converts incoming JSON data into a Java object automatically.
- Example JSON body:

```
{
  "vendorID": "C1",
  "vendorName": "Nisha",
  "vendorGender": "Female",
  "vendorAge": "30",
  "vendorContact": "8369"
}
```

→ Spring Boot automatically maps it to a cloudVendor Java object.

✳ PUT Mapping

```
@PutMapping
public String updateCloudVendorDetials(@RequestBody cloudVendor cloudvendor) {
    this.cloudvendor = cloudvendor;
    return "Cloud Vendor Updated Sucessfully";
}
```

7️⃣ @PutMapping

- Handles HTTP PUT requests — used for updating existing data.

- Works similar to @PostMapping, but conceptually for updates.

✳ DELETE Mapping

```
@DeleteMapping("/{vendorID}")
public String DeleteCloudVendorDetials(String vendorID) {
    this.cloudvendor = null;
    return "Cloud Vendor Deleted Sucessfully";
}
```

8️⃣ @DeleteMapping("/{vendorID}")

- Handles HTTP DELETE requests.
- Path /cloudvendor/{vendorID} indicates which record to delete.

Note ⚠:
You should also use @PathVariable here:

```
public String DeleteCloudVendorDetials(@PathVariable String vendorID)
```

Otherwise, vendorID won't be populated from the URL.

隣 1️⃣ @Entity

易 What it does:

@Entity marks a Java class as a JPA entity — meaning it represents a table in your database. When you use JPA (Hibernate), it automatically maps your class to a table.

里 Example:

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class CloudVendor {
    @Id
    private String vendorId;
    private String vendorName;
    private String vendorAddress;
    private String vendorPhoneNumber;

    // Getters and Setters
}
```

拓 Explanation:

- The class CloudVendor becomes a database table (for example, cloud_vendor).
- Each field becomes a column.
- `@Id` marks the **`primary key`** of the table.

茶 2 `@Table`

易 What it does:

`@Table` lets you **`customize the table name`** and schema mapping of your entity.

里 Example:

```
import jakarta.persistence.Entity;
import jakarta.persistence.Table;
import jakarta.persistence.Id;

@Entity
@Table(name = "vendors")
public class CloudVendor {
    @Id
    private String vendorId;
    private String vendorName;
}
```

拓 Explanation:

Without @Table, Hibernate would create a table called cloud_vendor by default.
With @Table(name = "vendors"), you explicitly tell Hibernate to use the table name vendors.

⅓ 3 @Id

易 What it does:

@Id marks a field as the primary key of your entity (the unique identifier).

里 Example:

```
@Entity
public class CloudVendor {
    @Id
    private String vendorId;
    private String vendorName;
}
```

拓 Optional — You can generate it automatically:

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

→ Hibernate will auto-generate primary key values (like auto-increment in SQL).

⚙️ 4 @Service

易 What it does:

@Service marks a class as a business logic layer component.
It tells Spring:

> “This is where my service/business logic lives — please create a Spring Bean for it.”

It’s part of Spring’s stereotype annotations, along with:

- `@Component (generic)`
- `@Service (business logic)`
- `@Repository (database access)`
- `@Controller (web layer)`

里 Example:

```
import org.springframework.stereotype.Service;
import java.util.*;

@Service
public class CloudVendorService {
    private Map<String, String> vendorData = new HashMap<>();

    public String getVendorById(String vendorId) {
        return vendorData.get(vendorId);
    }

    public void addVendor(String id, String name) {
        vendorData.put(id, name);
    }
}
```

拓 Explanation:

- Spring will detect this class during component scanning.
- You can inject it into controllers using @Autowired.

```
@Autowired
private CloudVendorService cloudVendorService;
```

就 5 @ControllerAdvice

易 What it does:

@ControllerAdvice is a global exception handler for your Spring MVC controllers.
It lets you handle exceptions in one place instead of writing try-catch blocks in every controller.

里 Example:

```
import org.springframework.http.HttpStatus;
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {

    // Handle a specific exception
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
        return new ResponseEntity<>("❌ " + ex.getMessage(),
HttpStatus.NOT_FOUND);
    }

    // Handle generic exceptions
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception ex) {
        return new ResponseEntity<>("⚠ Something went wrong: " +
ex.getMessage(),
                                    HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

⚠️6 @ExceptionHandler

易 What it does:

@ExceptionHandler is used inside a @ControllerAdvice or directly in a controller to handle specific exceptions.

里 Example:

Inside a controller:

```
@GetMapping("/vendor/{id}")
public ResponseEntity<String> getVendor(@PathVariable String id) {
    if (id.equals("404")) {
        throw new ResourceNotFoundException("Vendor not found!");
    }
    return ResponseEntity.ok("Vendor found!");
}

@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
    return new ResponseEntity<>("❌ " + ex.getMessage(), HttpStatus.NOT_FOUND);
}
```

拓 Explanation:

When ResourceNotFoundException is thrown, this method catches it and returns a custom HTTP 404 response.

里 Combined Example (All Together)

泉 CloudVendor.java

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "cloud_vendor")
public class CloudVendor {
    @Id
    private String vendorId;
    private String vendorName;
    private String vendorAddress;
    private String vendorPhoneNumber;

    // Getters and Setters
}
```

泉 CloudVendorService.java

```
import org.springframework.stereotype.Service;
import java.util.*;

@Service
public class CloudVendorService {
    private Map<String, CloudVendor> vendorList = new HashMap<>();

    public CloudVendor getVendor(String vendorId) {
        CloudVendor vendor = vendorList.get(vendorId);
        if (vendor == null) throw new ResourceNotFoundException("Vendor not
found: " + vendorId);
        return vendor;
    }

    public void addVendor(CloudVendor vendor) {
        vendorList.put(vendor.getVendorId(), vendor);
    }
}
```

泉 GlobalExceptionHandler.java

```
import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }
}
```

易 TL;DR Summary

Annotation	Layer	Purpose	Example
@Entity	Model	Marks a class as DB table	@Entity class CloudVendor
@Table	Model	Customize table name	@Table(name="vendors")
@Id	Model	Primary key	@Id Long id
@Service	Service	Business logic bean	@Service class VendorService
@ControllerAdvice	Global	Centralized exception handling	@ControllerAdvice class GlobalHandler
@ExceptionHandler	Global/Controller	Handle specific exceptions	@ExceptionHandler(Exception.class)

@Component is an annotation used to mark a Java class as a Spring-managed bean.
It tells the Spring IoC (Inversion of Control) container to automatically detect, instantiate, and manage that class.

When you annotate a class with @Component, Spring will:

1. Create an object of that class.
2. Store it in the Spring Application Context (like a container).
3. Allow you to inject it anywhere else using @Autowired.

✔ 1. @PostConstruct

頭 When is it used?

- Runs immediately after bean creation and dependency injection.
- Used for initialization logic.

撒 Example

```
@Component
public class EmailService {

    private Connection connection;

    @PostConstruct
    public void init() {
        System.out.println("Initializing EmailService…");
        connection = Connection.connect();
    }
}
```

✔ What happens?

- Spring creates the bean → injects dependencies → then calls init() automatically.

✅ 2. @PostMapping

🧠 When is it used?

- In Spring MVC (REST API)
- Maps HTTP POST requests to a controller method.

🛠️ Example

```
@RestController
@RequestMapping("/users")
public class UserController {

    @PostMapping("/create")
    public String createUser(@RequestBody User user) {
        return "User created: " + user.getName();
    }
}
```

✔ What happens?

Sending a POST request to /users/create will call this method.

⚠ Note

These are the two commonly used Spring annotations that start with @Post….

📊 Quick Summary Table

Annotation	Module	Purpose
@PostConstruct	Spring Core	Runs after bean initialization
@PostMapping	Spring MVC	Handles HTTP POST requests

IOC- Inversion of control--> principal
Dependency Injection--> design pattern
Sprin framework
--> creation of an object.
Springboot
--> confrigutaion take care of it.
--> opionented framework
-->embedded tomcat(craeate a jar file)
--> depend on use case we choice dependency from the spring intializer
--> SpringApplication.run(RestDemo2Application.class, args);--> run the container
--> if we creating a project without springboot we need to do configruation such a spring.xml and add the required dependency manually from maven site and paste it into pom.xml files

--> from application context we are getting IoC container but not the object. So, if we want the object we will write in the spring.xml. In spring.xml we will write the beans. It wiill craete an

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframewor

    <bean id="dev" class="com.telusko.Dev">


    </bean>
```

object of the class.

-->if a variable is private and we need to access it in another class, we need to create getter and setter for that.
--> <property> tag is used to set the value of variable which is private in another class n need to set

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.spri

    <bean id="dev" class="com.telusko.Dev">
        <property name="age" value="12" />
    </bean>
```

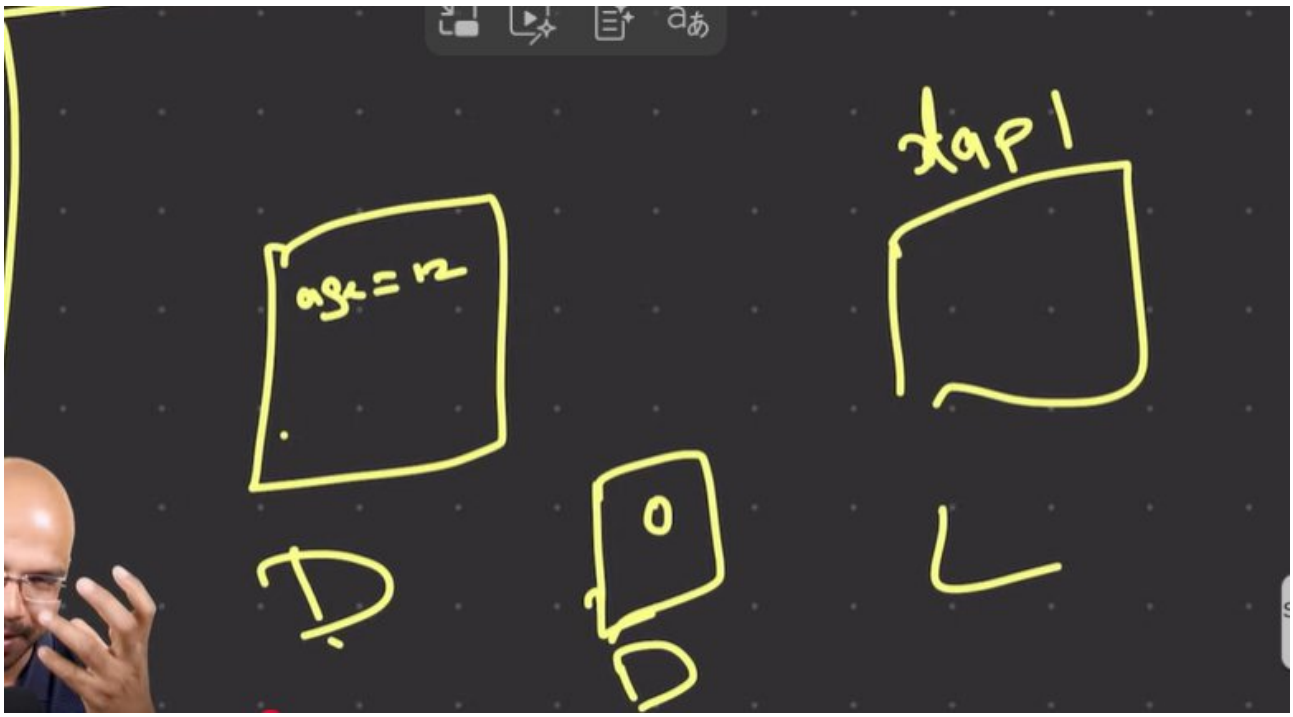
```
public class Dev {

    // private Laptop laptop;

    private int age;

    public Dev() { System.out.println("Dev Constructor"

    public int getAge() {
        return age;
    }
```

the of variable by spring . So, we use property<setter injection>

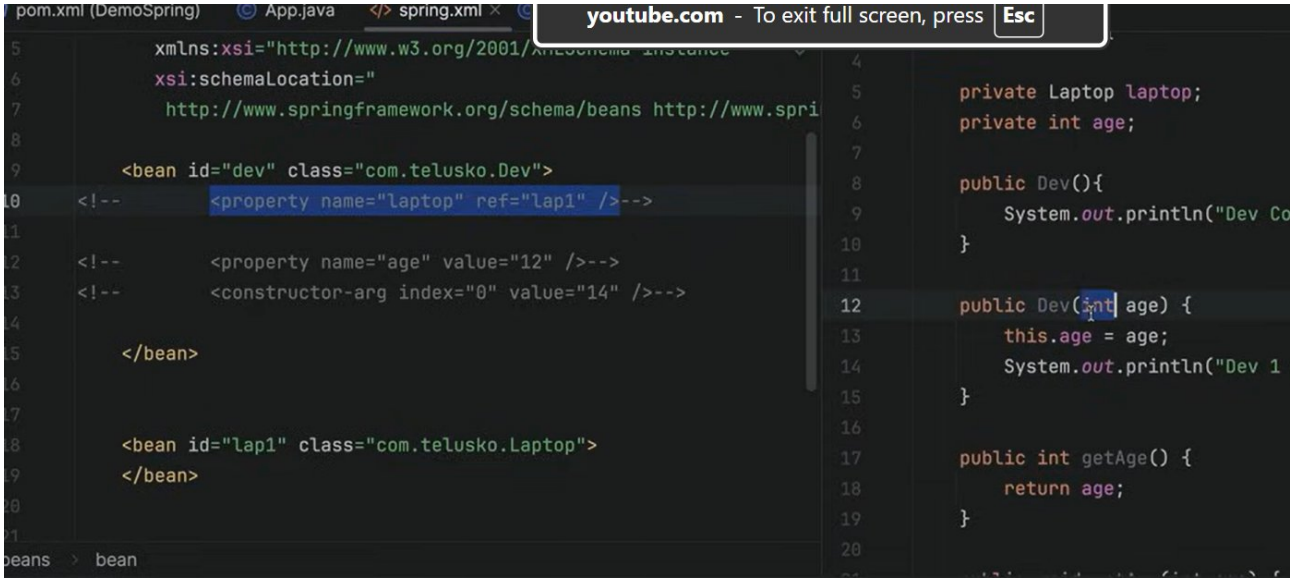
--> constructor injection--> use constructor arg--> injecting the value of variable (age) with constructor

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.spri

    <bean id="dev" class="com.telusko.Dev">
<!--        <property name="age" value="12" />-->
        <constructor-arg value="14" />
    </bean>
```

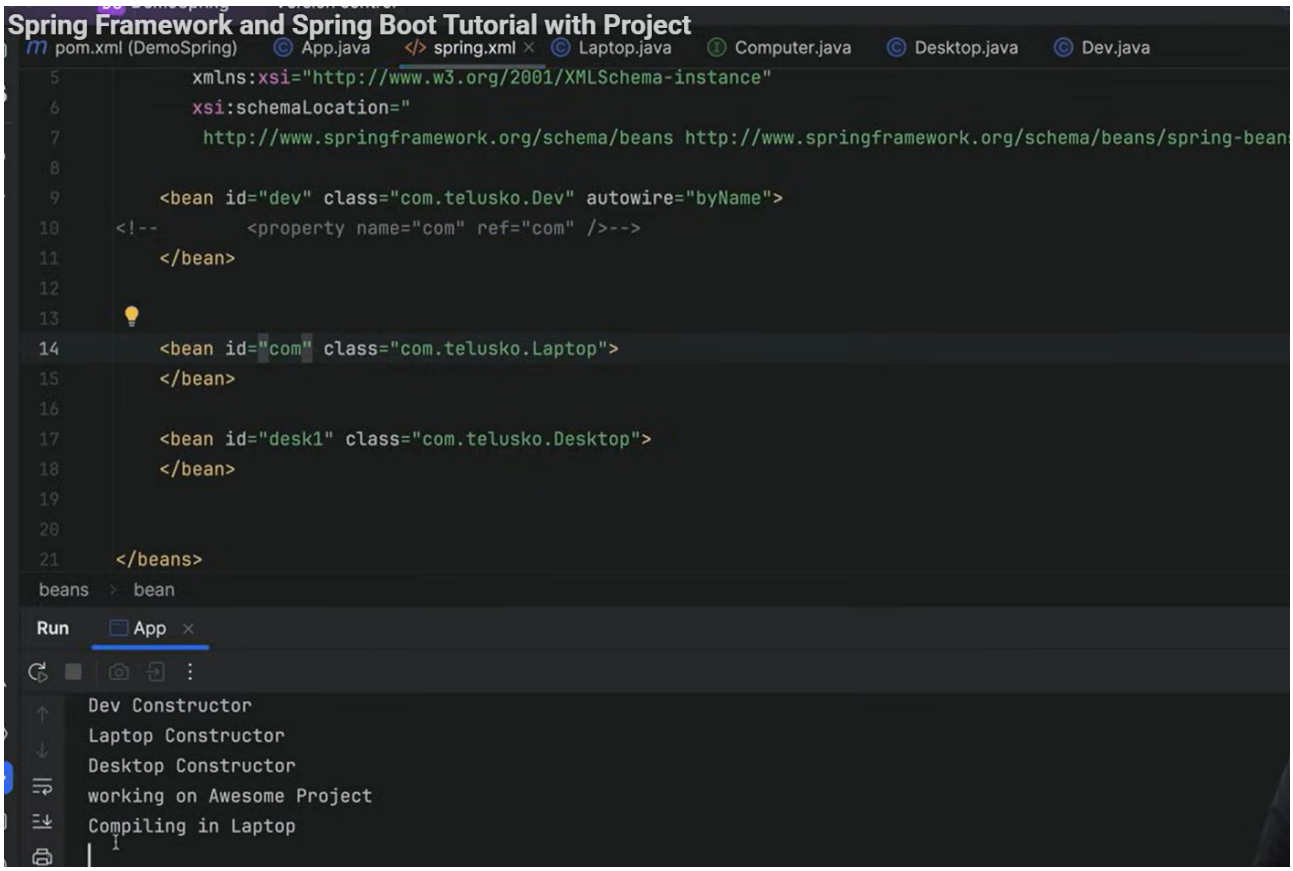
```
public class Dev {

    //private Laptop laptop;
    private int age;

    public Dev(){
        System.out.println("Dev Constructor");
    }

    public Dev(int age) {
        this.age = age;
    }
```

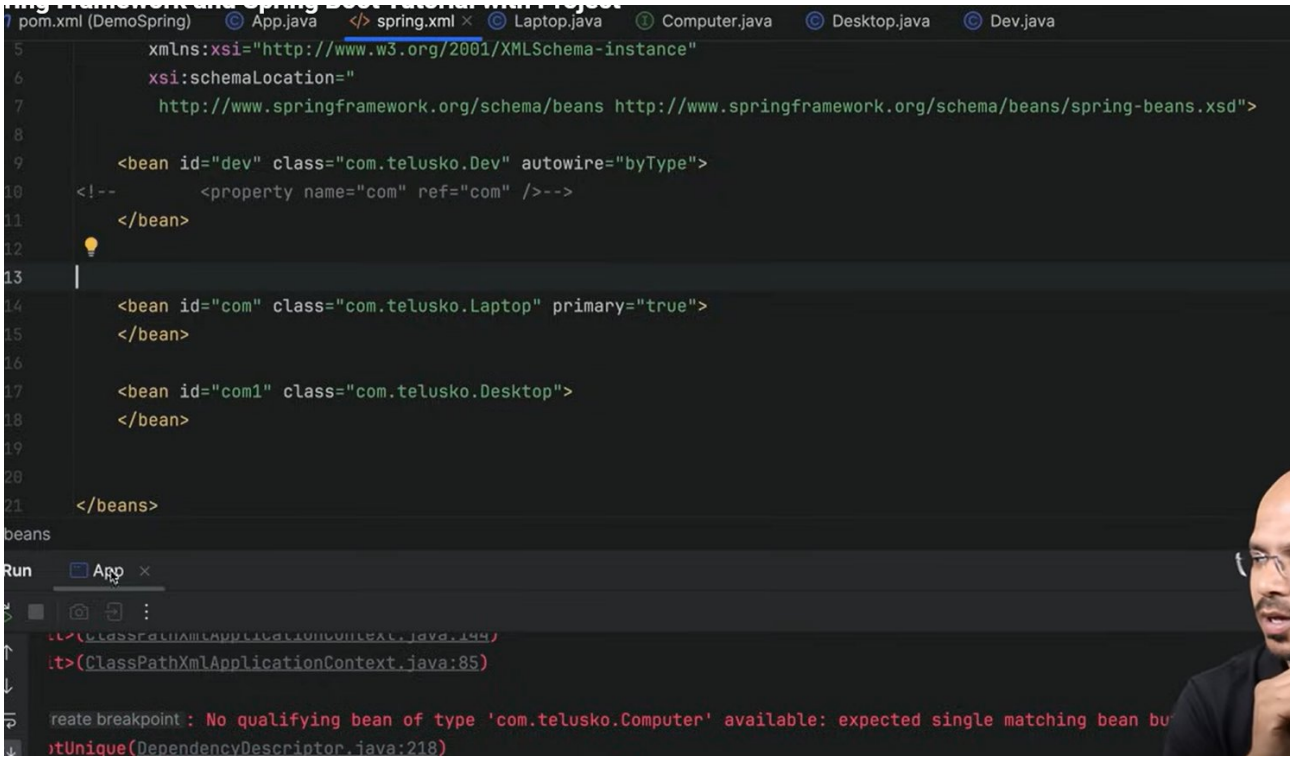


if we want to get an object of the other class that is laptop in dev class. We can say connection two



class with the help of beans. So here we do that





@Autowired is an annotation used by Spring Framework for Dependency Injection (DI).
It tells Spring to automatically inject (supply) the required dependency into a class at runtime.
When Spring Boot starts:

1. It scans all classes with @Component (via Component Scanning).
2. It creates one object (bean) for each.

3. Wherever it sees @Autowired, it injects the matching bean automatically.

For filed injection and setter injection you have you write @Autowired but, for construdtor we have default no need to specify.

It go by type.

JSON- Javascrpit Object Notion

ORM--> object relation mapping

data and behaviour.(productId, ProductName and ProductPrice)

Give object to ORM ,

ORM tool-->create table ,insert the data. Each object is one row.

Hibernate,Esciplse Link

JPA--> Java Persistence API(follow the same standerd)

Spring data JPA

Problem: Multiple Beans of the Same Type

Suppose you have two beans of the same type in Spring:

```
import org.springframework.stereotype.Component;

@Component
public class PetrolEngine implements Engine {
    public void start() {
        System.out.println("Petrol engine started...");
    }
}

@Component
public class DieselEngine implements Engine {
    public void start() {
        System.out.println("Diesel engine started...");
    }
}
```

Now, in another class:

```
@Component
public class Car {

    @Autowired
    private Engine engine; // ✕ Problem — Which engine to inject?
}
```

Spring will be confused because there are two beans of type Engine.
Result → NoUniqueBeanDefinitionException (Spring doesn't know which one to use).

✔ Solution 1: Use @Primary

If you want one bean to be the default when there are multiple options — mark it with @Primary.

```
@Component
@Primary
public class PetrolEngine implements Engine {
    public void start() {
        System.out.println("Petrol engine started...");
    }
}

@Component
public class DieselEngine implements Engine {
    public void start() {
        System.out.println("Diesel engine started...");
    }
}
```

Now, if you use:

```
@Autowired
private Engine engine;
```

Spring will automatically inject the PetrolEngine (the one marked with @Primary).

透 Output:

```
Petrol engine started...
```

✔ Solution 2: Use @Qualifier

@Qualifier lets you explicitly specify which bean to inject.

Example:

```
@Component("petrolEngine")
public class PetrolEngine implements Engine {
    public void start() {
        System.out.println("Petrol engine started...");
    }
}

@Component("dieselEngine")
public class DieselEngine implements Engine {
    public void start() {
        System.out.println("Diesel engine started...");
    }
}
```

Then in your Car class:

```
@Component
public class Car {

    @Autowired
    @Qualifier("dieselEngine")  // 癖 specify exact bean name
    private Engine engine;

    public void drive() {
        engine.start();
    }
}
```

透 Output:

```
Diesel engine started...
```

⚙ Key Difference

Feature	@Primary	@Qualifier
Purpose	Set a default bean when multiple exist	Choose a specific bean explicitly
Where used	On Bean Class or Bean Method	On injection point (@Autowired)
Priority	Acts as fallback	Overrides @Primary if used together
Usage	General default choice	Specific choice per injection

⭐ 1. What is Spring Security? Why do we use it?

✔ Detailed Interview Answer:

Spring Security is a framework that provides authentication and authorization for Java and Spring-based applications.
It acts as a security layer that sits in front of our application and controls who can access what.

We use Spring Security to protect:

- REST APIs
- Web applications
- Endpoints
- Resources like URLs, methods, and data

What it provides internally:

- **Authentication – verifying user identity**
- **Authorization – checking permissions**
- **Password encryption & hashing**
- **Role-based access control (RBAC)**

- **JWT, OAuth2 support**
- **CSRF protection**
- **Session management**

Why we use it:

Implementing security manually is **complex and error-prone**.
Spring Security gives **pre-built, configurable, and industry-standard security**, reducing vulnerabilities and improving maintainability.

☆ 2. Difference between Authentication and Authorization

✔ Authentication:

Authentication answers **“Who are you?”**

It verifies user identity using:

- Username & password
- Token
- OTP
- Biometric

Example:
User logs in using credentials.

✔ Authorization:

Authorization answers “What are you allowed to do?”

It checks permissions after authentication.

Examples:

- Admin → can delete users
- User → can only view profile

In Spring Security:

- **Authentication → handled by `AuthenticationManager`**
- **Authorization → handled by `AccessDecisionManager`**

顯 Authentication happens **first**, authorization happens **after**.

☆ 3. What is the default security provided by Spring Boot?

✔ Detailed Answer:

When Spring Security is added to a Spring Boot project, it **auto-configures security by default**.

It provides:

- A default login page
- A generated username = user
- A random password printed in logs
- **All endpoints are secured**
- **CSRF enabled for form-based apps**

Example log:

```
Using generated security password: 8adf9a2...
```

This default behavior ensures nothing is exposed accidentally.

☆ 4. Explain the Spring Security Filter Chain

✔ Detailed Answer:

Spring Security works internally using a chain of servlet filters.

Every HTTP request passes through this filter chain before reaching the controller.

Important filters:

- `UsernamePasswordAuthenticationFilter`
- `BasicAuthenticationFilter`
- `BearerTokenAuthenticationFilter (JWT)`
- `SecurityContextPersistenceFilter`
- `ExceptionTranslationFilter`

Flow:

1. Request enters application
2. Filters authenticate user
3. Authorization is checked
4. If allowed → controller
5. If denied → 401 or 403

This makes Spring Security centralized and consistent.

⭐ 5. What is SecurityContext and SecurityContextHolder?

✔ SecurityContext:

It stores the Authentication object of the currently logged-in user.

It contains:

- Username
- Roles
- Authentication status

✔ SecurityContextHolder:

A utility class that stores SecurityContext using:

- `ThreadLocal (default)`
- `InheritableThreadLocal`
- `Global mode`

Example:

```
Authentication auth =
 SecurityContextHolder.getContext().getAuthentication();
```

Used to get logged-in user info anywhere in the app.

⭐ 6. What is UserDetails and UserDetailsService?

✔ UserDetails:

A Spring Security interface that represents a security user.

It contains:

- Username
- Password
- Authorities (roles)
- Account status (locked/expired)

✔ UserDetailsService:

Used to load user data from DB during login.

```
public UserDetails loadUserByUsername(String username)
```

Spring Security calls this method automatically during authentication.

⭐ 7. Explain BCryptPasswordEncoder

✔ Detailed Answer:

BCryptPasswordEncoder is used to hash passwords securely.

It:

- Uses salt internally
- Produces different hashes for same password
- Is resistant to brute-force attacks

Example:

```
encoder.encode("password123");
```

BCrypt is one-way hashing, not encryption.

⭐ 8. How to secure REST API URLs?

✔ Detailed Answer:

Using HttpSecurity we define who can access which endpoint.

```
http.authorizeHttpRequests()
    .requestMatchers("/admin/**").hasRole("ADMIN")
    .requestMatchers("/user/**").hasAnyRole("USER","ADMIN")
    .anyRequest().authenticated();
```

This enables role-based access control at URL level.

⭐ 9. Difference between @Secured, @PreAuthorize, @RolesAllowed

✔ @Secured

- Simple role check
- Limited flexibility

```
@Secured("ROLE_ADMIN")
```

✔ @PreAuthorize

- Most powerful
- Uses SpEL
- Supports method arguments

```
@PreAuthorize("hasRole('ADMIN') and #id == authentication.principal.id")
```

✔ @RolesAllowed

- JSR-250 standard
- Simpler syntax

⭐ 10. What is CSRF? Should we disable it for REST APIs?

✔ CSRF:

Cross-Site Request Forgery tricks a logged-in user to perform actions unknowingly.

Spring enables CSRF by default for form-based apps.

✔ For REST APIs:

We disable CSRF because:

- REST APIs are stateless
- They use tokens, not cookies

```
http.csrf().disable();
```

⭐ 11. What is JWT? How does it work?

✔ Detailed Flow:

1. User logs in
2. Server validates credentials
3. Server generates JWT
4. Token sent to client
5. Client sends token in header
6. Server verifies signature

JWT is:

- Stateless
- Fast
- Scalable

Perfect for microservices.

⭐ 12. Stateful vs Stateless Authentication

Stateful:

- Session stored on server
- Server memory dependent
- Used in web apps

Stateless:

- No server session
- Token-based (JWT)
- Used in microservices

⭐ 13. Custom Authentication Provider

✔ Purpose:

Used when default authentication is not enough.

We implement:

`AuthenticationProvider`

It allows custom login logic, DB checks, external services, etc.

⭐ 14. What is CORS? How to enable it?

✔ CORS:

Allows frontend and backend on different domains to communicate.

Example: React (3000) → Spring Boot (8080)

Enabled via:

`http.cors();`

Or via WebMvcConfigurer.

⭐ 15. Allow some APIs publicly

✔ Example:

```
http.authorizeHttpRequests()
    .requestMatchers("/public/**").permitAll()
    .anyRequest().authenticated();
```

Used for:

- Login
- Registration
- Health checks

⭐ 16. What is OncePerRequestFilter?

✔ Detailed Answer:

A filter that executes once per request.

Used in:

- JWT validation
- Token parsing

Ensures no duplicate execution.

⭐ 17. How are roles stored?

Roles are stored using GrantedAuthority.

```
new SimpleGrantedAuthority("ROLE_ADMIN")
```

Returned as part of UserDetails.

⭐ 18. What is AuthenticationManager?

✔ Detailed Answer:

It coordinates authentication.

- Accepts Authentication object
- Delegates to AuthenticationProvider
- Returns authenticated object or throws exception

Used in login APIs.

⭐ 19. What is Basic Authentication?

✔ Detailed Answer:

Uses Base64 encoded credentials in header.

```
Authorization: Basic base64(username:password)
```

Less secure because credentials are sent on every request.

⭐ 20. What is PasswordEncoder? Why do we need it?

✔ Detailed Answer:

PasswordEncoder hashes passwords securely.

Reasons:

- Prevent plain text storage
- Protect against DB breaches
- Meet security compliance

BCrypt is recommended due to:

- Salting
- Adaptive hashing
- One-way encryption

1️⃣ What is Hibernate?

Answer:
Hibernate is a Java ORM framework that maps Java classes to database tables and lets developers perform database operations using Java objects instead of writing SQL manually.

2️⃣ What is ORM (Object Relational Mapping)?

Answer:
ORM is a technique that maps between Java objects and database tables.
Hibernate uses ORM to automatically convert objects to relational data.

3️⃣ What are the advantages of Hibernate over JDBC?

Answer:

- No need for SQL (HQL used instead)
- Automatically generates queries
- Supports caching (better performance)
- Reduces boilerplate code
- Supports lazy loading

- Database independent

---

4 What is SessionFactory and Session?

Answer:

- **SessionFactory:**
  - **Heavy object**
  - **Created once**
  - **Used to create sessions**
- **Session:**
  - **Represents a single database connection**
  - **Used to perform CRUD**

---

5 What is HQL?

Answer:
HQL stands for **Hibernate Query Language**.
It is object-oriented and works on entities, not tables.

Example:

```
FROM Employee WHERE salary > 50000
```

---

6 What is Lazy Loading?

Answer:
Lazy loading means Hibernate loads related data only when needed, not upfront.

Example:

```
@OneToMany(fetch = FetchType.LAZY)
```

---

7 What is Caching in Hibernate?

Answer:

Hibernate provides two types of caching:

✔ Level 1 Cache:

- Default
- Per session

✔ **Level 2 Cache:**

- Optional
- Shared across sessions
- Uses EhCache, Hazelcast, etc.

8️⃣ Difference between save() and persist()?

save()	persist()
Returns generated ID	Returns void
Hibernate-specific	JPA standard
Works outside transaction	Requires transaction

9️⃣ What is the difference between get() and load()?

get()	load()
Returns null if not found	Throws exception
Immediately hits DB	Uses proxy (lazy)
Eager loading	Lazy loading

📛 What is the use of @Entity?

Answer:
Marks a Java class as a database table in Hibernate.

1️⃣1️⃣ Explain 1st level and 2nd level cache.

Answer:

- **1st level cache:**
  - **Enabled by default**
  - **Session-level**
- **2nd level cache:**
  - **Optional**
  - **SessionFactory-level**
  - **Used to reduce DB hits**

1️⃣2️⃣ What are Hibernate states of an object?

1. **Transient → Not associated with DB**

2. **Persistent → Linked with Hibernate session**
3. **Detached → Session closed**
4. **Removed → Deleted**

13️⃣ Explain @OneToMany, @ManyToOne, @OneToOne, @ManyToMany.

These are **Hibernate/JPA annotations** used to map relationships between entities.

- One employee → many tasks
- Many tasks → one employee
- One user → one profile
- Many students ↔ Many courses

⭐ 1. What is Spring MVC?

✔ Definition

Spring MVC is a web framework based on the Model–View–Controller (MVC) design pattern.

It is used to:

- Build web applications
- Handle HTTP requests
- Separate concerns (UI, business logic, data)

Spring MVC Architecture (How it works)

1. Client sends request
2. **DispatcherServlet receives request**
3. **HandlerMapping finds Controller**
4. **Controller calls Service**
5. **Service talks to Repository**
6. **ViewResolver returns View (JSP/Thymeleaf)**

Example: Spring MVC Controller

```
@Controller
public class HelloController {

    @RequestMapping("/hello")
    public String hello(Model model) {
```

```
        model.addAttribute("msg", "Hello Spring MVC");
        return "hello"; // JSP or Thymeleaf view
    }
}
```

Configuration in Spring MVC (IMPORTANT)

Spring MVC requires a lot of configuration:

- web.xml
- DispatcherServlet config
- ViewResolver
- DataSource
- Bean definitions

Example:

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
```

✕ Drawbacks of Spring MVC

- Heavy XML / Java configuration
- Manual dependency management
- External server (Tomcat) required
- Slower setup

⭐ 2. What is Spring Boot?

✔ Definition

Spring Boot is a framework built on top of Spring that makes Spring development:

✔ Faster
✔ Simpler
✔ Production-ready

Spring Boot eliminates boilerplate configuration.

What Spring Boot Provides

- Auto-configuration
- Embedded servers (Tomcat, Jetty)
- Starter dependencies
- No XML configuration
- Production tools (Actuator)

Example: Spring Boot REST Controller

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello Spring Boot";
    }
}
```

顯 No XML, no web.xml, no DispatcherServlet setup

顯 Everything auto-configured

Spring Boot Main Class

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

@SpringBootApplication =

- `@Configuration`
- `@EnableAutoConfiguration`
- `@ComponentScan`

⭐ 3. Key Differences: Spring MVC vs Spring Boot

Feature	Spring MVC	Spring Boot
Type	Web framework	Framework built on Spring
Purpose	Build MVC web apps	Simplify Spring development
Configuration	Heavy (XML/Java)	Minimal / Auto

Feature	Spring MVC	Spring Boot
Server	External Tomcat	Embedded Tomcat
Dependency mgmt	Manual	Starter POMs
Setup time	High	Very low
REST support	Yes	Yes (easier)
Production ready	✖	✔ (Actuator)

⭐ 4. Dependency Difference (Very Important)

Spring MVC (pom.xml)

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
</dependency>
```

You must add many more dependencies manually.

Spring Boot (pom.xml)

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

This single starter includes:

- Spring MVC
- Jackson
- Validation
- Embedded Tomcat

⭐ 5. Real-World Usage Comparison

✔ When to Use Spring MVC

- Legacy applications
- Traditional JSP-based apps
- Applications with strict custom configuration

✔ When to Use Spring Boot

- REST APIs

- Microservices
- Cloud-native apps
- New projects
- Faster development required

🧠 Most modern companies prefer Spring Boot

⭐ 6. Interview-Friendly Explanation (Speak This)

> Spring MVC is a web framework that implements the MVC pattern and requires manual configuration. Spring Boot is built on top of Spring and simplifies Spring development by providing auto-configuration, embedded servers, and starter dependencies. Spring Boot can use Spring MVC internally, but it removes boilerplate configuration and speeds up development.

⭐ 7. Very Common Follow-Up Questions

❓ Can we use Spring MVC without Spring Boot?

✔ Yes

❓ Does Spring Boot replace Spring MVC?

✖ No, it uses Spring MVC internally

❓ Is Spring Boot only for REST?

✖ No, but it’s best suited for REST & microservices

⭐ 8. One-Line Difference (Must Remember)

> Spring MVC is a framework to build web applications, while Spring Boot is a framework that simplifies Spring application development, including Spring MVC, with minimal configuration.

1. What is MVC in Spring?

Answer:
MVC stands for Model–View–Controller.
Spring MVC separates application logic into:

- Model → Data & business logic
- **View → UI (JSP, Thymeleaf)**
- **Controller → Handles request, returns response**

This improves separation of concerns and maintainability.

✔ 2. Explain the Spring MVC Request Flow. (VERY IMPORTANT)

Answer:

1. Client sends request
2. **DispatcherServlet receives it**
3. **DispatcherServlet calls HandlerMapping to find Controller**
4. **Controller method executes the logic**
5. **Returns ModelAndView**
6. **ViewResolver identifies the actual view (HTML/JSP)**
7. **Response is sent back to the client**

This is asked in FIS interviews repeatedly.

✔ 3. What is DispatcherServlet?

Answer:
It is the **front controller** in Spring MVC.
It handles every request coming to the application and routes it to the correct controller.

✔ 4. What is the difference between @Controller and @RestController?

@Controller	@RestController
Returns view (HTML/JSP)	Returns data (JSON/XML)
Used for web UI apps	Used for REST APIs
Needs @ResponseBody	Has @ResponseBody by default

✔ 5. What is @RequestMapping?

Answer:
Annotation used to map incoming HTTP requests to controller methods.

Example:

```
@RequestMapping("/employee")
public String getEmployee() { ... }
```

✔ 6. What is the difference between @RequestMapping and @GetMapping / @PostMapping?

Answer:
@GetMapping and @PostMapping are specialized shortcuts for @RequestMapping(method = GET/POST).

✔ 7. What is Model in Spring MVC?

Answer:
A Model is a map-like structure used to send data from Controller → View.

Example:

```
model.addAttribute("name", "Nisha");
```

✔ 8. What is ModelAndView?

Answer:

- Holds both model data and view name
- Used in traditional Spring MVC (not REST APIs)

Example:

```
ModelAndView mv = new ModelAndView("home");
mv.addObject("user", userObj);
```

✔ 9. What is ViewResolver? Why is it used?

Answer:
ViewResolver maps a logical view name to an actual view file.

Example:
Logical name → "home"
Actual file → /WEB-INF/views/home.jsp

✔ 10. What is InternalResourceViewResolver?

Answer:
It resolves JSP files by adding prefix and suffix.

Example:

```
prefix = "/WEB-INF/views/"
suffix = ".jsp"
```

✔ 11. What is @PathVariable?

Answer:
Used to extract values from the URL.

```
@GetMapping("/user/{id}")
public User getUser(@PathVariable int id) { ... }
```

✔ 12. What is @RequestParam?

Answer:
Used to extract query parameters.

```
/employee?id=100

@RequestParam int id
```

✔ 13. Difference between @PathVariable and @RequestParam?

@PathVariable	@RequestParam
Part of the URL	Query parameter
Mandatory	Optional
/user/10	/user?id=10

✅ 14. What is @ResponseBody in Spring MVC?

Answer:
Converts the return value of a method directly into JSON or XML.

Used in REST APIs.

✅ 15. Explain the role of HandlerMapping.

Answer:
It helps DispatcherServlet find the correct controller method for the incoming request.

✅ 16. What is the difference between Spring MVC and Spring Boot?

Answer:

- Spring MVC requires manual setup
- Spring Boot auto-configures everything
- Boot is faster to develop, uses embedded servers, starters

✅ 17. What is form backing object in Spring MVC?

Answer:
A POJO used to capture form data submitted by the user.

✅ 18. How does validation work in Spring MVC? (@Valid)

Answer:
Use @Valid or @Validated with BindingResult.

```
public String save(@Valid User user, BindingResult result)
```

Error messages are displayed using BindingResult.

✔ 19. What is the role of @ExceptionHandler?

Answer:
Handles exceptions at the controller level.

✔ 20. What is @ControllerAdvice?

Answer:
Handles exceptions globally for all controllers.

灐 Additional FIS Global Real Questions (2–3 Years)

21. How do you handle exceptions in MVC?

→ Using @ExceptionHandler, @ControllerAdvice, ResponseEntityExceptionHandler.

22. How do you return JSON in Spring MVC?

→ Add @ResponseBody OR use @RestController.

23. MVC vs REST?

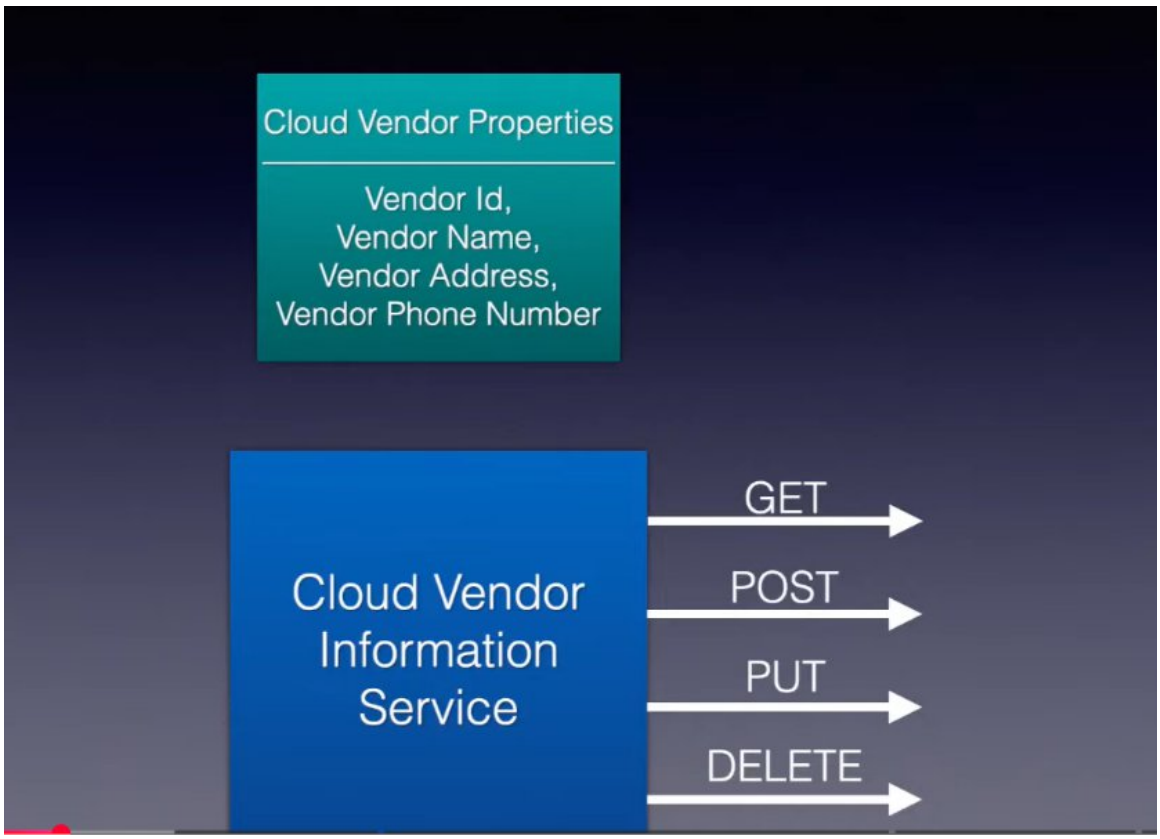
→ MVC returns views; REST returns JSON/XML.

24. What is the difference between Forward and Redirect?

Forward	Redirect
Server-side	Client-side
URL unchanged	URL changes
Faster	Slower
Data retained	Data lost

25. What is the life cycle of a Spring MVC application?

→ Initialization → DispatcherServlet loads → Mapping → Controller → View → Response



RestDemoApplication.java | cloudVendor.java | cloudVendorAPIService.java

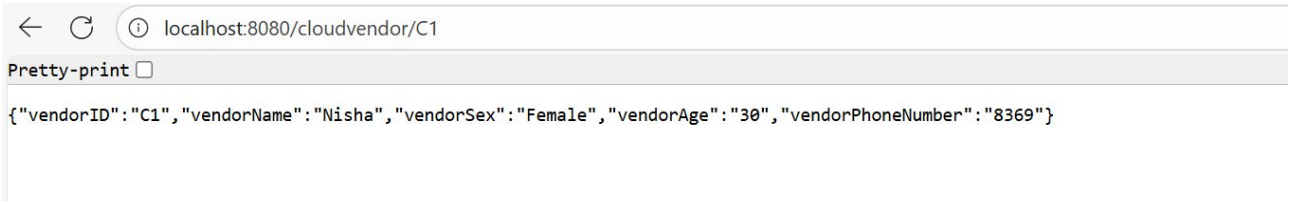
```java
package com.projectcrusoperation.rest_demo.model;

public class cloudVendor {

    private String vendorID;
    private String vendorName;
    private String vendorSex;
    private String vendorAge;
    private String vendorPhoneNumber;
    public cloudVendor(String vendorID, String vendorName, String vendorSex, String vendor
            String vendorPhoneNumber) {
        super();
        this.vendorID = vendorID;
        this.vendorName = vendorName;
        this.vendorSex = vendorSex;
        this.vendorAge = vendorAge;
        this.vendorPhoneNumber = vendorPhoneNumber;
    }
    public cloudVendor() {
        super();
    }
    public String getVendorID() {
        return vendorID;
    }
    public void setVendorID(String vendorID) {
        this.vendorID = vendorID;
```

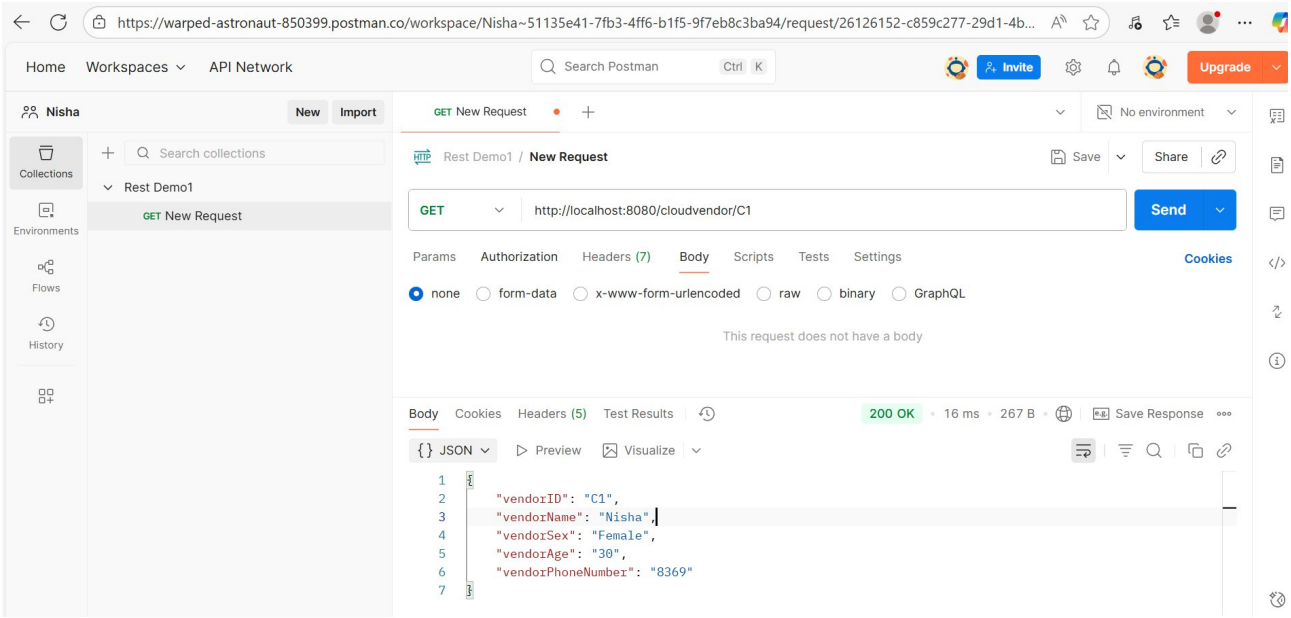
RestDemoApplication.java | cloudVendor.java | cloudVendorAPIService.java ×

```java
package com.projectcrusoperation.rest_demo.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.projectcrusoperation.rest_demo.model.cloudVendor;

@RestController
@RequestMapping("/cloudvendor")
public class cloudVendorAPIService {

    @GetMapping("{vendorID}")//get method
    public cloudVendor getCloudVendorDetials(String vendorID) {

        return new cloudVendor("C1","Nisha","Female","30","8369");
    }

    // when we go to the link -->https://localhost:8080/cloudvendor/C1 -->
    // we will get the detials as "C1","Nisha","Female","30","8369" on the link
}
```

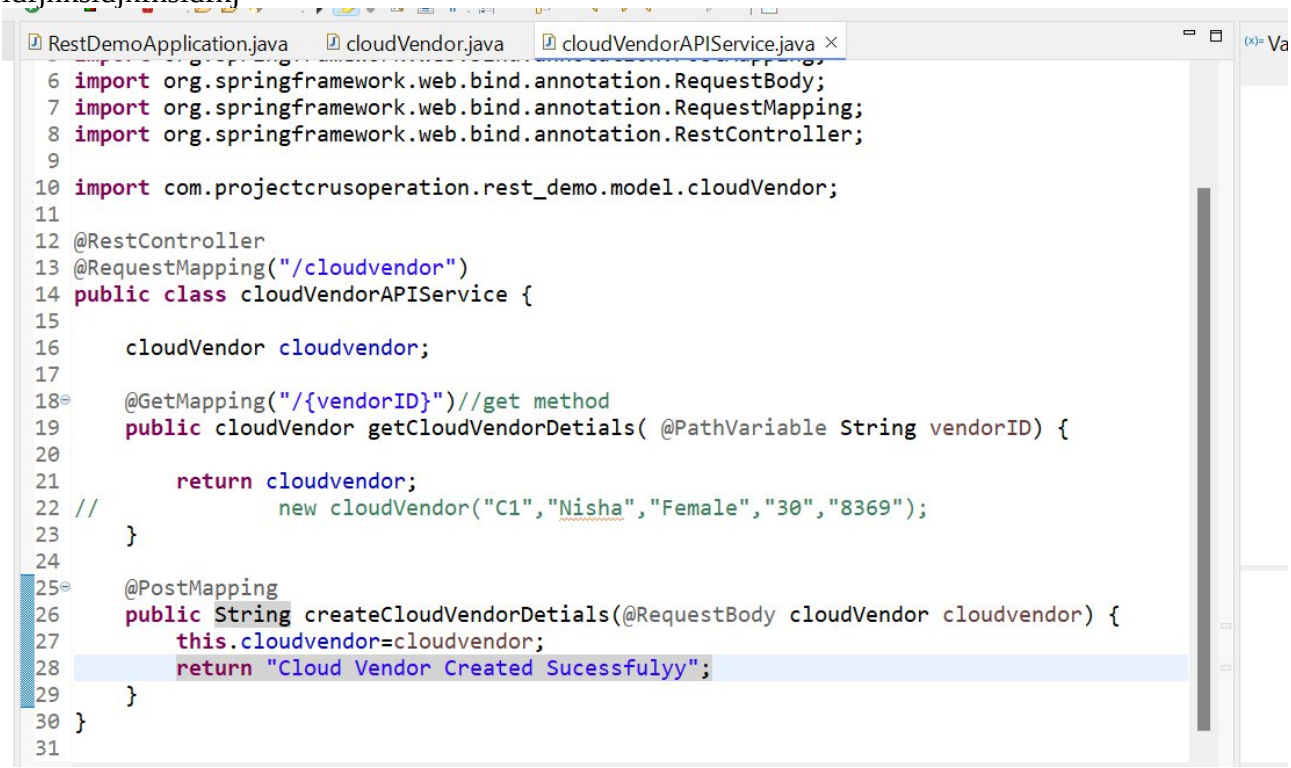
RestDemoApplication.java × | cloudVendor.java | cloudVendorAPIService.java

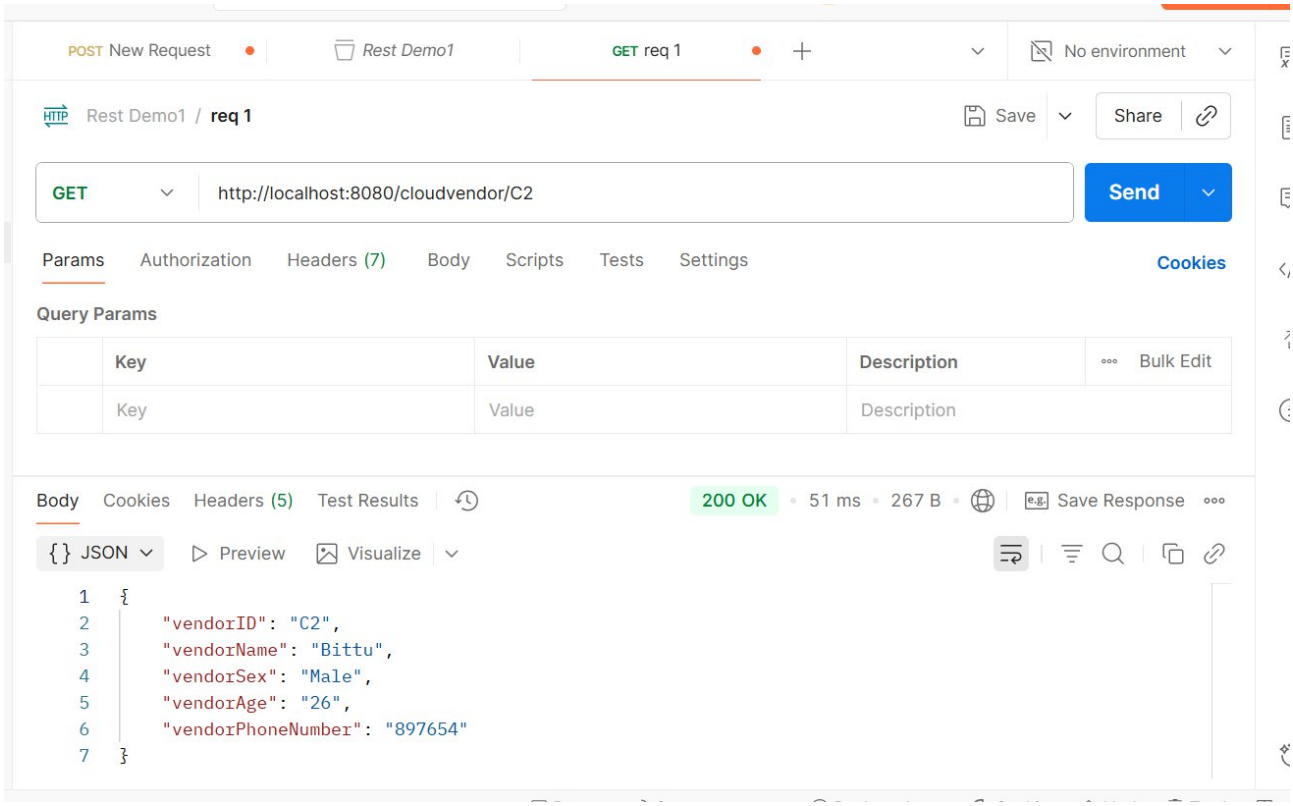
```java
package com.projectcrusoperation.rest_demo;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class RestDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(RestDemoApplication.class, args);
    }

}
```



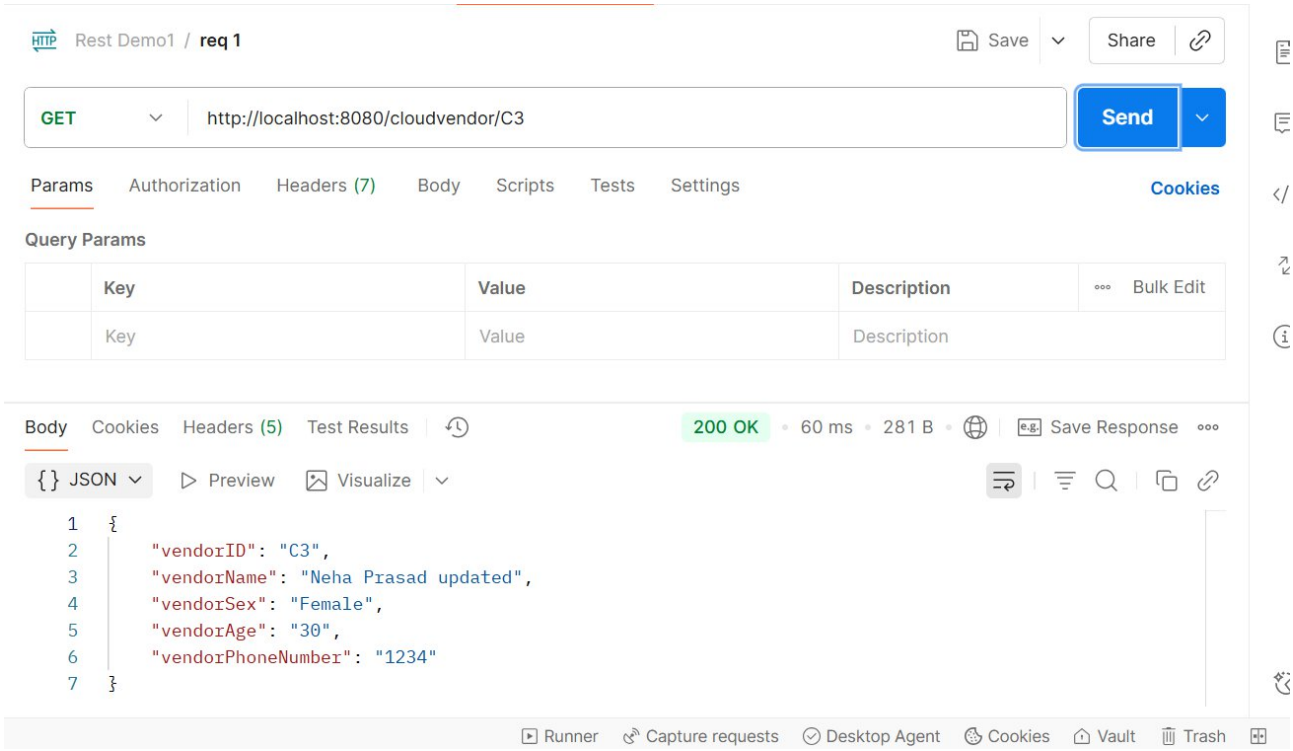


ldfjhksldjkfhsldfkj



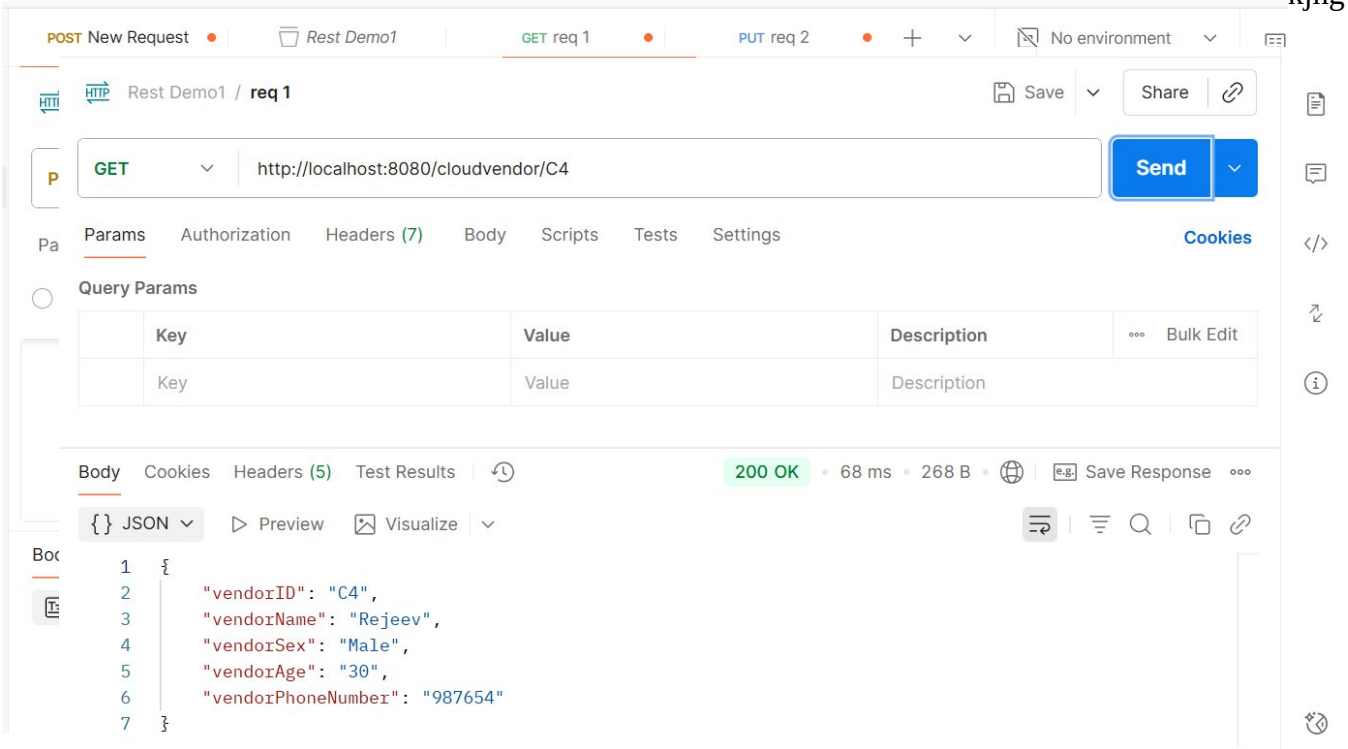


get the request kjfhj

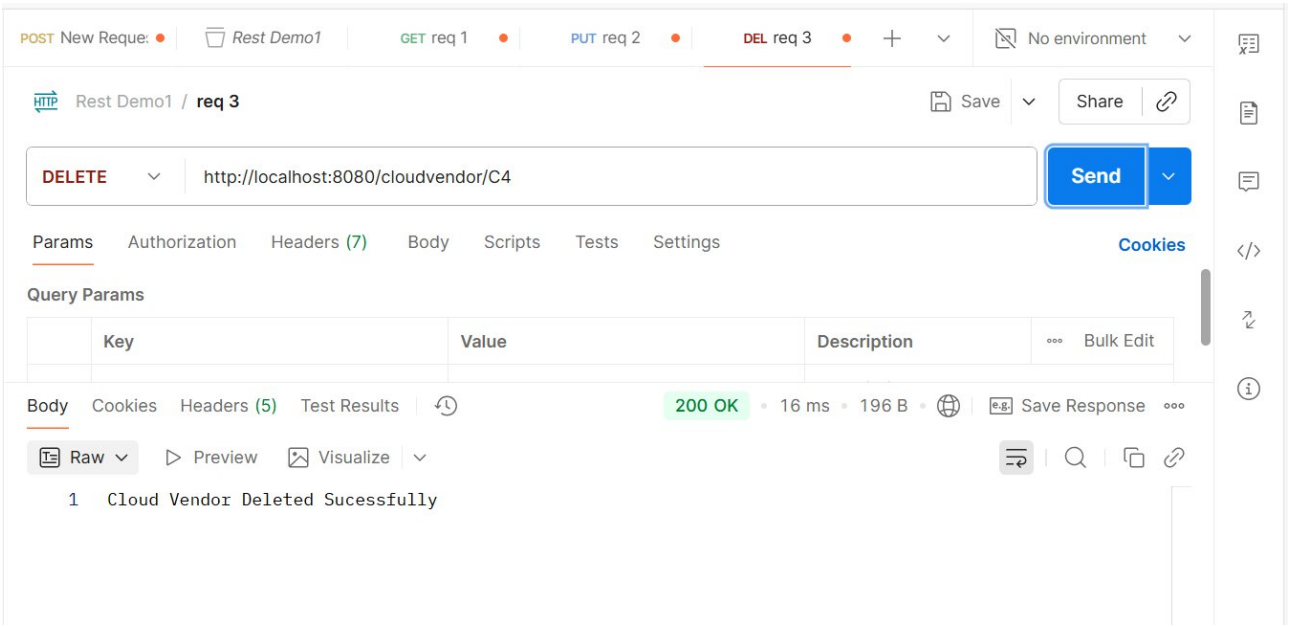


kjhk

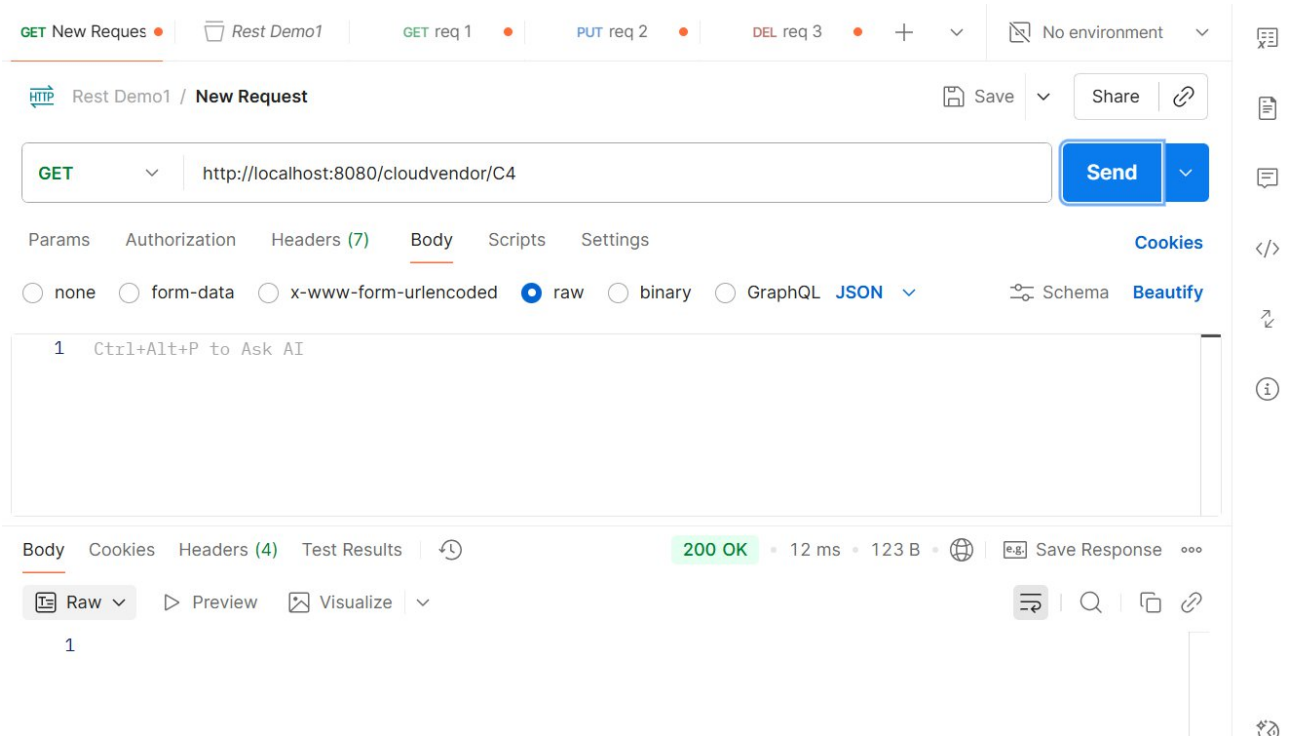
kjhg



lkjh


POST New Reque:
Rest Demo1
GET req 1
PUT req 2
DEL req 3
No environment
Rest Demo1 / req 3
Save
Share
DELETE
http://localhost:8080/cloudvendor/C4
Send
Params
Authorization
Headers (7)
Body
Scripts
Tests
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Body
Cookies
Headers (5)
Test Results
200 OK
16 ms
196 B
Save Response
Raw
Preview
Visualize
1 Cloud Vendor Deleted Sucessfully

kjgh

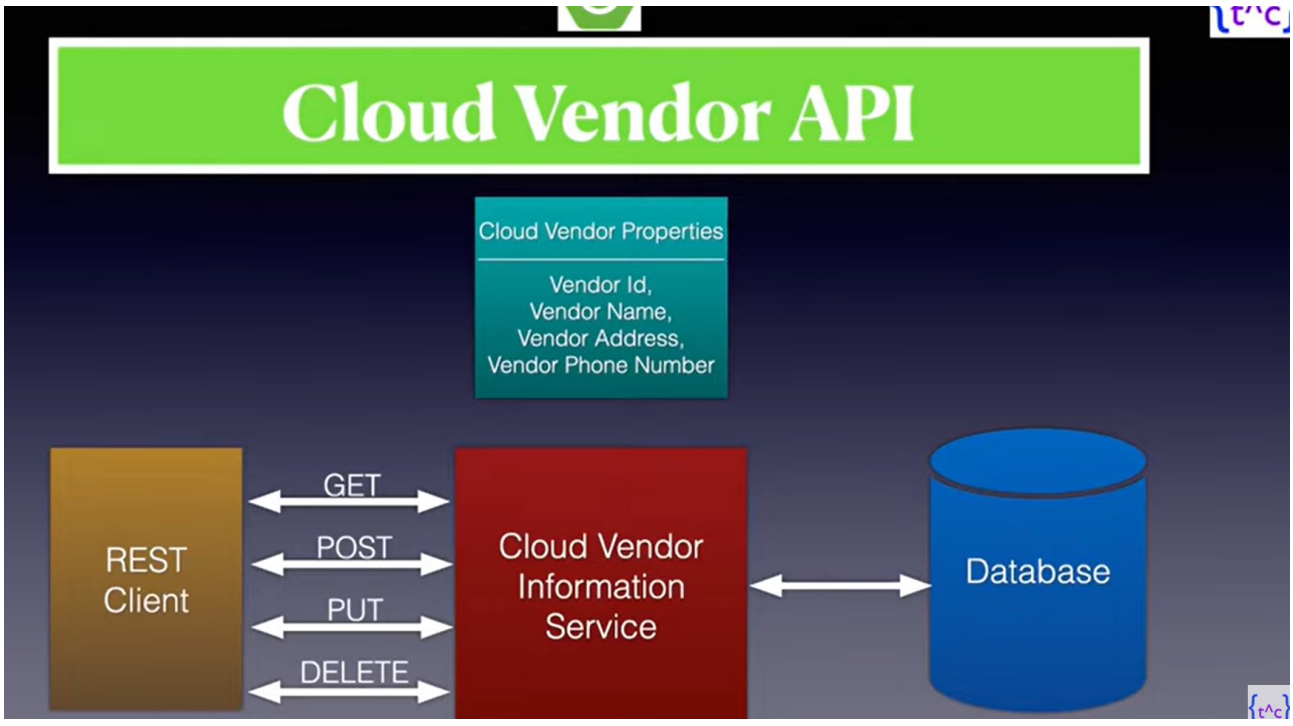

GET New Reques
Rest Demo1
GET req 1
PUT req 2
DEL req 3
No environment
Rest Demo1 / New Request
Save
Share
GET
http://localhost:8080/cloudvendor/C4
Send
Params
Authorization
Headers (7)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
1 Ctrl+Alt+P to Ask AI
Body
Cookies
Headers (4)
Test Results
200 OK
12 ms
123 B
Save Response
Raw
Preview
Visualize
1

kjhljk

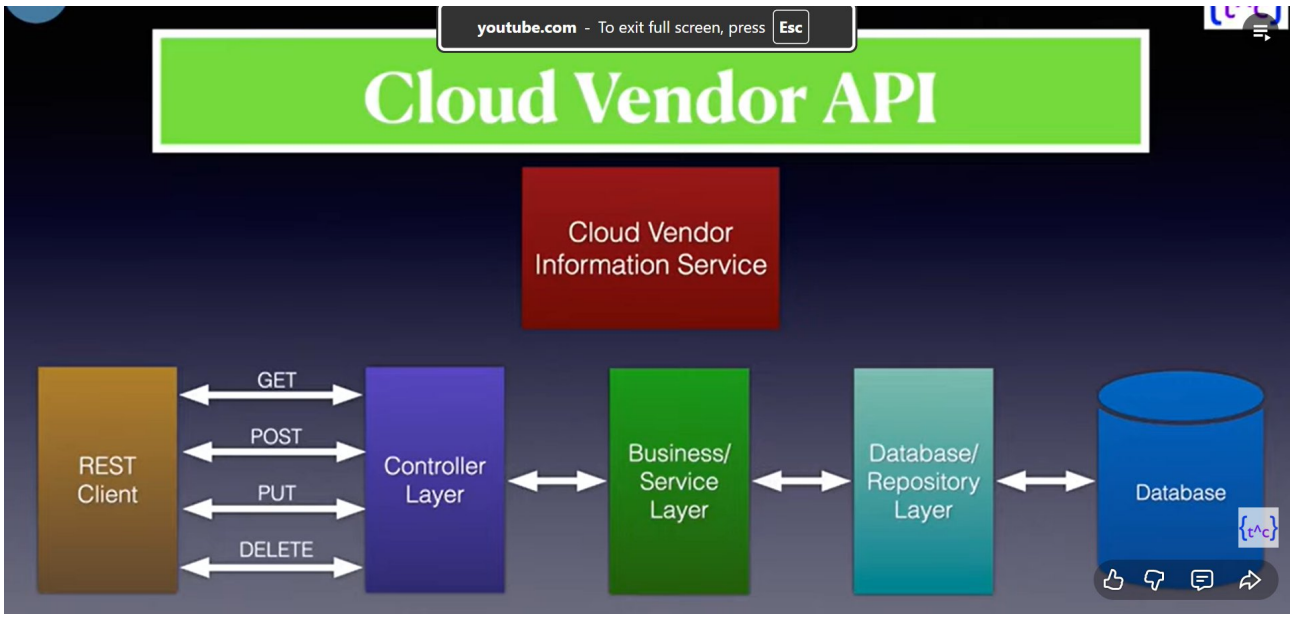
RestDemoApplication.java | cloudVendor.java | cloudVendorAPIService.java

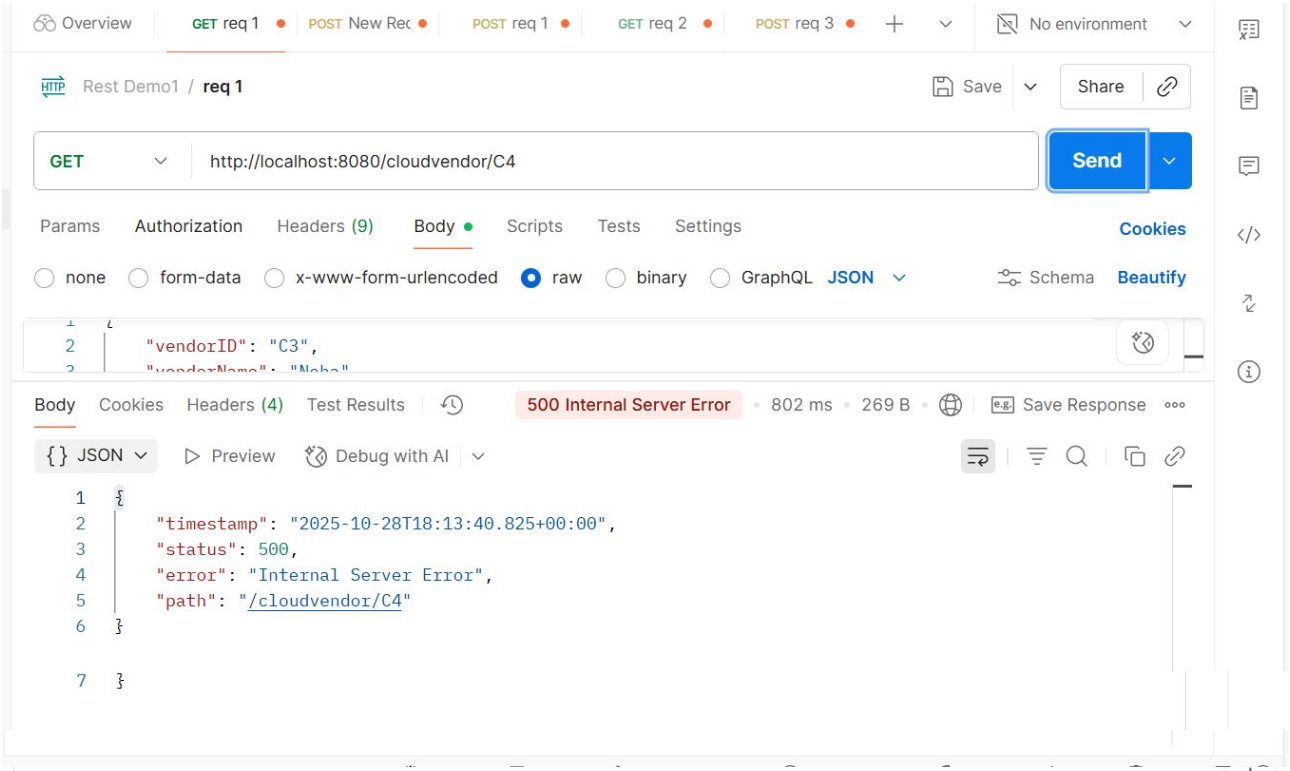
```java
@RestController
@RequestMapping("/cloudvendor")
public class cloudVendorAPIService {

    cloudVendor cloudvendor;

    @GetMapping("/{vendorID}")//get method
    public cloudVendor getCloudVendorDetials( @PathVariable String vendorID) {

        return cloudvendor;
//              new cloudVendor("C1","Nisha","Female","30","8369");
    }

    @PostMapping//create a new data and pu it into data base.
    public String createCloudVendorDetials(@RequestBody cloudVendor cloudvendor) {
        this.cloudvendor=cloudvendor;
        return "Cloud Vendor Created Sucessfully";
    }



    @PutMapping//to update a existing data.
    public String updateCloudVendorDetials(@RequestBody cloudVendor cloudvendor) {
        this.cloudvendor=cloudvendor;
        return "Cloud Vendor Updated Sucessfully";
    }

    @DeleteMapping("/{vendorID}")
    public String DeleteCloudVendorDetials(String vendorID) {
        this.cloudvendor=null;
        return "Cloud Vendor Deleted Sucessfully";
    }
```

kjhk

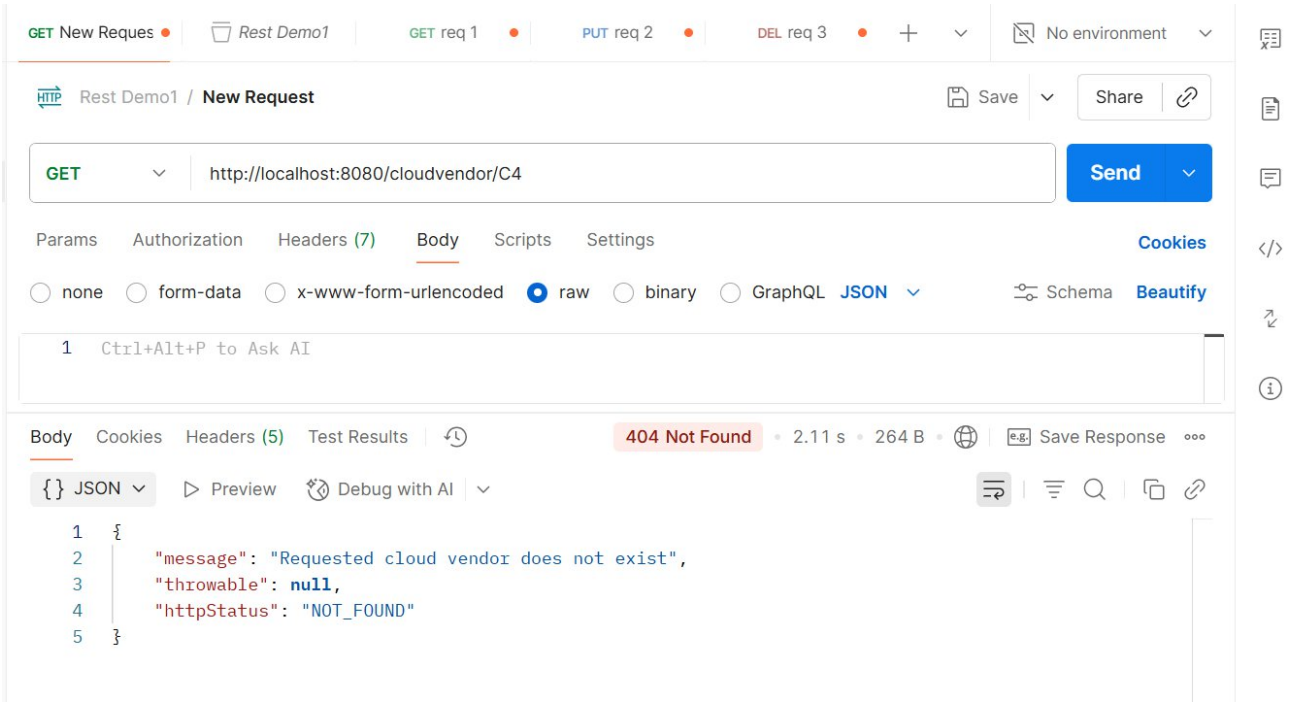


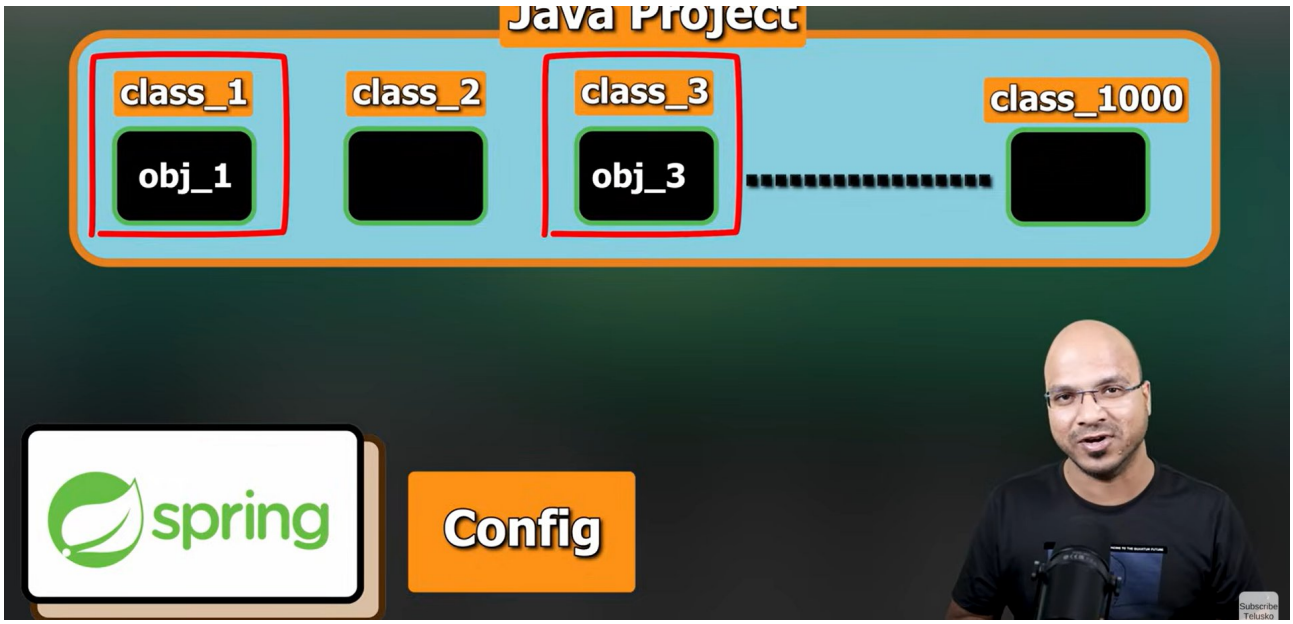
jhk

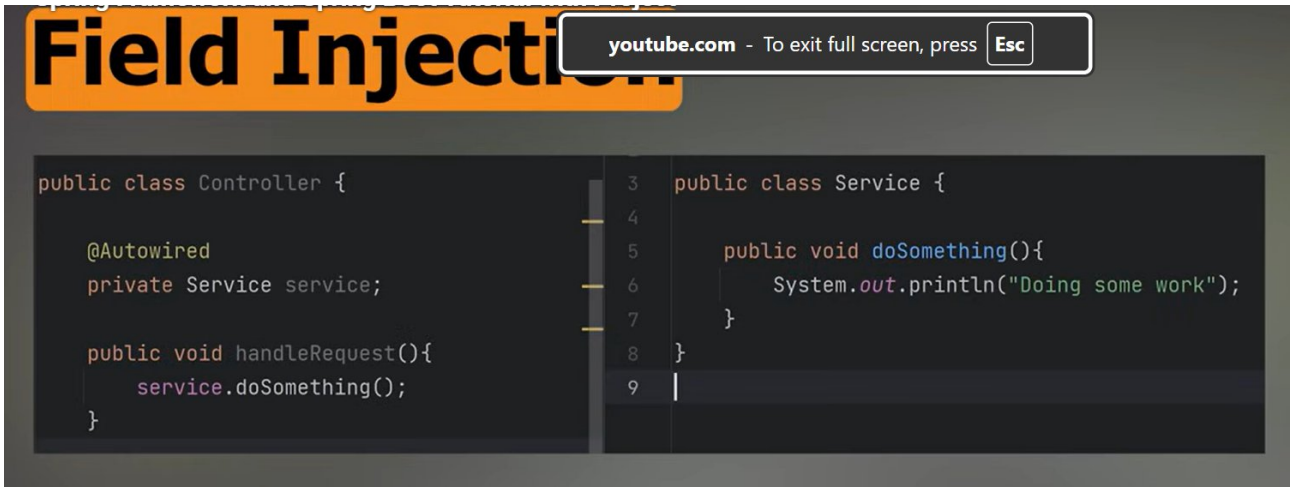

youtube.com - To exit full screen, press Esc
Cloud Vendor API
Cloud Vendor Information Service
REST Client
GET
POST
PUT
DELETE
Controller Layer
Business/ Service Layer
Database/ Repository Layer
Database
lkhrlfsk

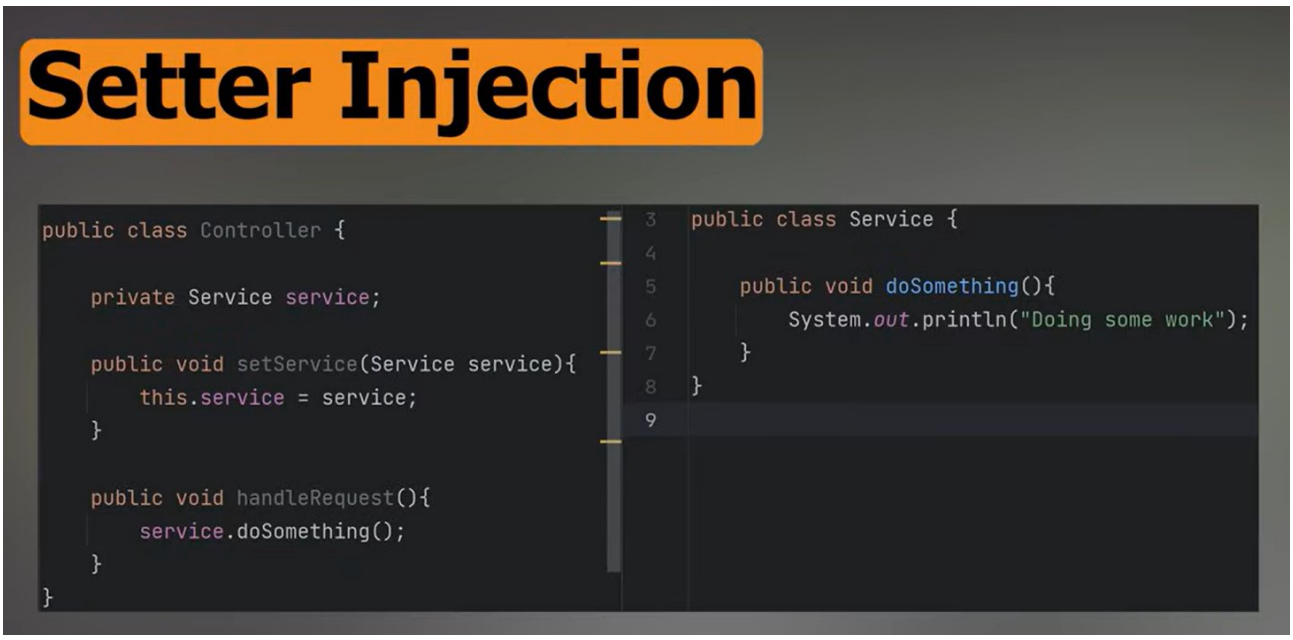

Overview
GET req 1
POST New Rec
POST req 1
GET req 2
POST req 3
No environment
Rest Demo1 / req 1
Save
Share
GET
http://localhost:8080/cloudvendor/C4
Send
Params
Authorization
Headers (9)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
"vendorID": "C3",
Body
Cookies
Headers (4)
Test Results
500 Internal Server Error
802 ms
269 B
Save Response
JSON
Preview
Debug with AI
{
"timestamp": "2025-10-28T18:13:40.825+00:00",
"status": 500,
"error": "Internal Server Error",
"path": "/cloudvendor/C4"
}
}

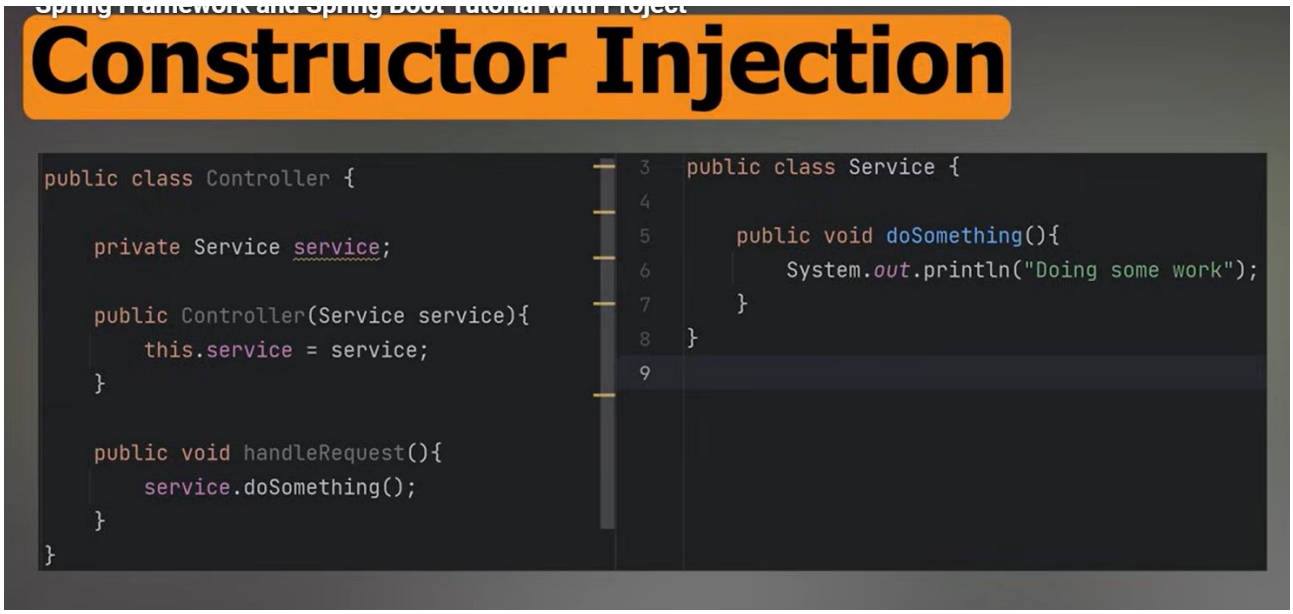
kjgkjh


GET New Reques
Rest Demo1
GET req 1
PUT req 2
DEL req 3
No environment
Rest Demo1 / New Request
Save
Share
GET
http://localhost:8080/cloudvendor/C4
Send
Params
Authorization
Headers (7)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
Ctrl+Alt+P to Ask AI
Body
Cookies
Headers (5)
Test Results
404 Not Found
2.11 s
264 B
Save Response
JSON
Preview
Debug with AI
{
"message": "Requested cloud vendor does not exist",
"throwable": null,
"httpStatus": "NOT_FOUND"
}

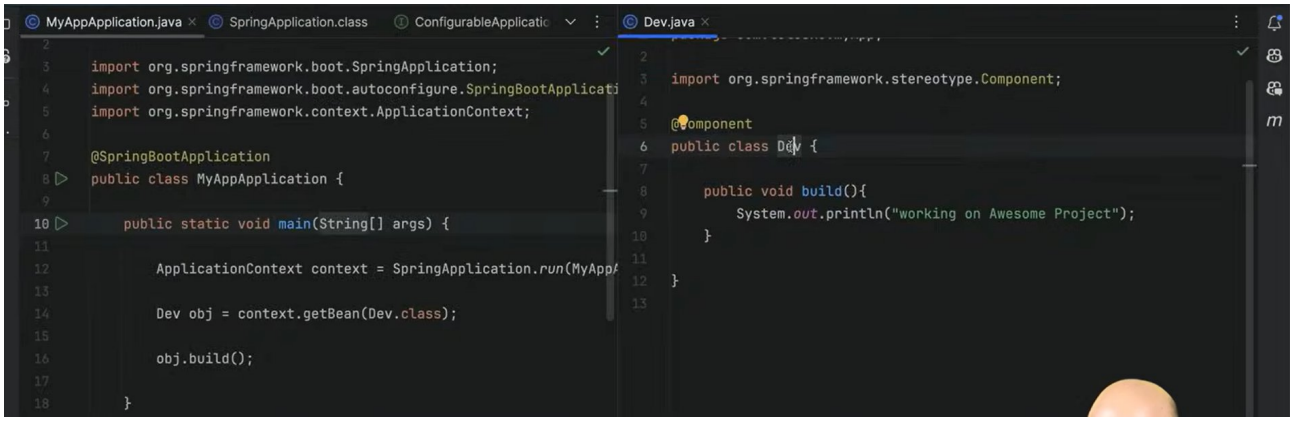

Java Project
class_1
obj_1
class_2
class_3
obj_3
class_1000
spring
Config


Field Injection
youtube.com - To exit full screen, press Esc
public class Controller {
@Autowired
private Service service;
public void handleRequest(){
service.doSomething();
}
public class Service {
public void doSomething(){
System.out.println("Doing some work");
}
}


Setter Injection
public class Controller {
private Service service;
public void setService(Service service){
this.service = service;
}
public void handleRequest(){
service.doSomething();
}
}
public class Service {
public void doSomething(){
System.out.println("Doing some work");
}
}



klj



@autowired