

**Understanding the problem without Maven**

There are many problems that we face during the project development. They are discussed below:

**1) Adding set of Jars in each project:** In case of struts, spring, hibernate frameworks, we need to add set of jar files in each project. It must include all the dependencies of jars also.

**2) Creating the right project structure:** We must create the right project structure in servlet, struts etc, otherwise it will not be executed.

**3) Building and Deploying the project:** We must have to build and deploy the project so that it may work.

What it does?

Maven simplifies the above mentioned problems. It does mainly following tasks.

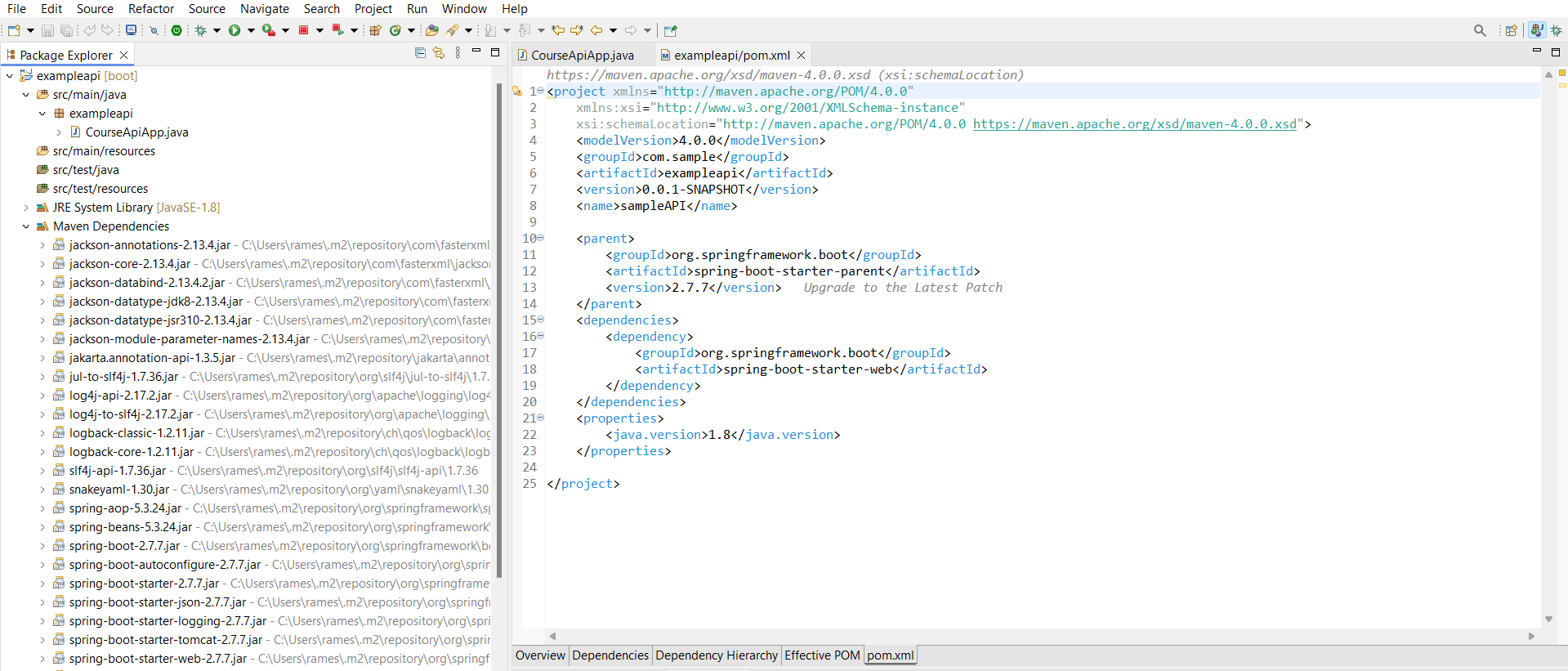
1. It makes a project easy to build
2. It provides uniform build process (maven project can be shared by all the maven projects)
3. It provides project information (log document, cross referenced sources, mailing list, dependency list, unit test reports etc.)
4. It is easy to migrate for new features of Maven

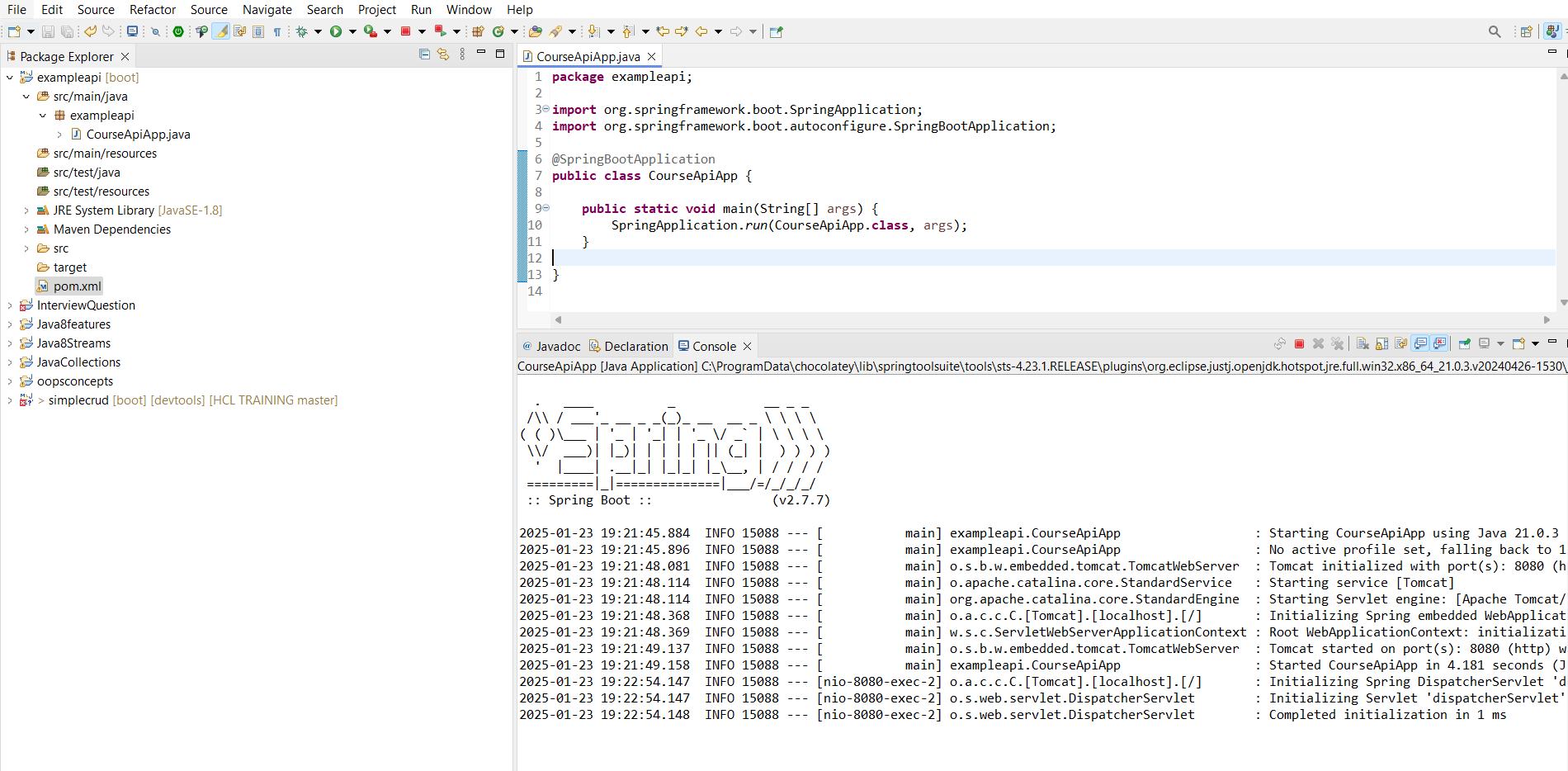
**What Is Maven?**

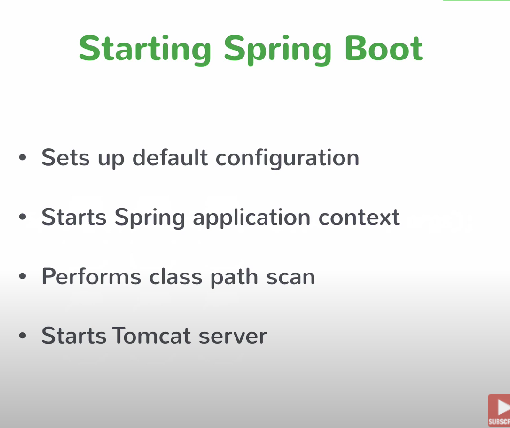
Maven is mostly used for the java projects to build web application packages. Maven provides different features which make it easy to build the web-application packages we manage complex projects easily.

1. **POM Files:** Project Object Model(POM) Files are XML file that contains information related to the project and configuration information such as dependencies, source directory, plugin, goals etc. used by Maven to build the project. When you should execute a maven command you give maven a POM file to execute the commands. Maven reads pom.xml file to accomplish its configuration and operations.
2. **Dependencies and Repositories:**Dependencies are external Java libraries required for Project and repositories are directories of packaged JAR files. The local repository is just a directory on your machine’s hard drive. If the dependencies are not found in the local Maven repository, Maven downloads them from a central Maven repository and puts them in your local repository.
3. **Build Life Cycles, Phases, and Goals:**A build life cycle consists of a sequence of build phases, and each build phase consists of a sequence of goals. Maven command is the name of a build lifecycle, phase, or goal. If a lifecycle is requested executed by giving the maven command, all build phases in that life cycle are executed also. If a build phase is requested executed, all build phases before it in the defined sequence are executed too.
4. **Build Profiles:**Build profiles a set of configuration values that allows you to build your project using different configurations. For example, you may need to build your project for your local computer, for development and test. To enable different builds you can add different build profiles to your POM files using its profiles elements which are triggered in a variety of ways.
5. **Build Plugins:**Build plugins are used to perform a specific goal. you can add a plugin to the POM file. Maven has some standard plugins you can use, and you can also implement your own in Java.
6. **Created sample maven project initially ,added spring related dependencies and achieved spring application. Created main classes and added annotations.**

Maven will download respective jar files based on dependency details will give in pom.xml







Embedded Tomcat Server

Apache Tomcat is an open-source software implementation of the Java Servlet, Java Server Pages (JSP), and WebSocket technologies. It acts as a web server and servlet container, **providing an environment where Java-based web applications can run.**

Key Features of Tomcat Server:

1. Servlet Container:
   * Tomcat provides a runtime environment for Java Servlets, allowing developers to run server-side Java programs that handle HTTP requests and responses dynamically.
2. JSP Engine:
   * It processes JavaServer Pages (JSP), which are HTML pages with embedded Java code, enabling dynamic content generation.
3. WebSocket Support:
   * Tomcat supports the WebSocket protocol for real-time, two-way communication between clients and servers.
4. Lightweight and Flexible:
   * It is lightweight compared to full Java EE application servers (e.g., WebLogic, WildFly) and focuses on web-tier technologies.
5. Configuration and Customization:
   * Offers a highly configurable and customizable runtime environment through its configuration files (server.xml, web.xml, etc.).
6. Cross-Platform Compatibility:
   * It runs on multiple operating systems, such as Windows, Linux, and macOS, due to its Java-based nature.
7. HTTPS Support:
   * Tomcat provides built-in support for HTTPS using Secure Socket Layer (SSL).

**Embedded Tomcat server**

When you use **spring-boot-starter-web** in a Spring Boot application, it includes **Tomcat** as the default embedded servlet container via the dependency **spring-boot-starter-tomcat**.

**Key Points:**

1. **Default Embedded Server**:
   * Tomcat is embedded by default, meaning you don't need to install or configure an external Tomcat server to run your Spring Boot application.
   * This allows you to package and run your app as a **standalone(application run without internet) JAR** with an embedded server.
2. **Dependency Inheritance**:
   * When you include **spring-boot-starter-web**, it automatically pulls in **spring-boot-starter-tomcat** as a transitive dependency.

Example in Maven:

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

1. **Override Default Server**:
   * If you prefer to use a different embedded server (e.g., **Jetty** or **Undertow**), you can exclude **spring-boot-starter-tomcat** and include the desired server's dependency.

Example for Jetty:

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

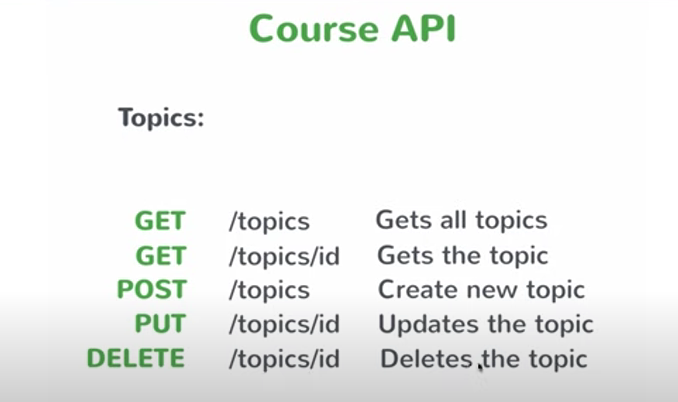
</dependency>

<dependency>

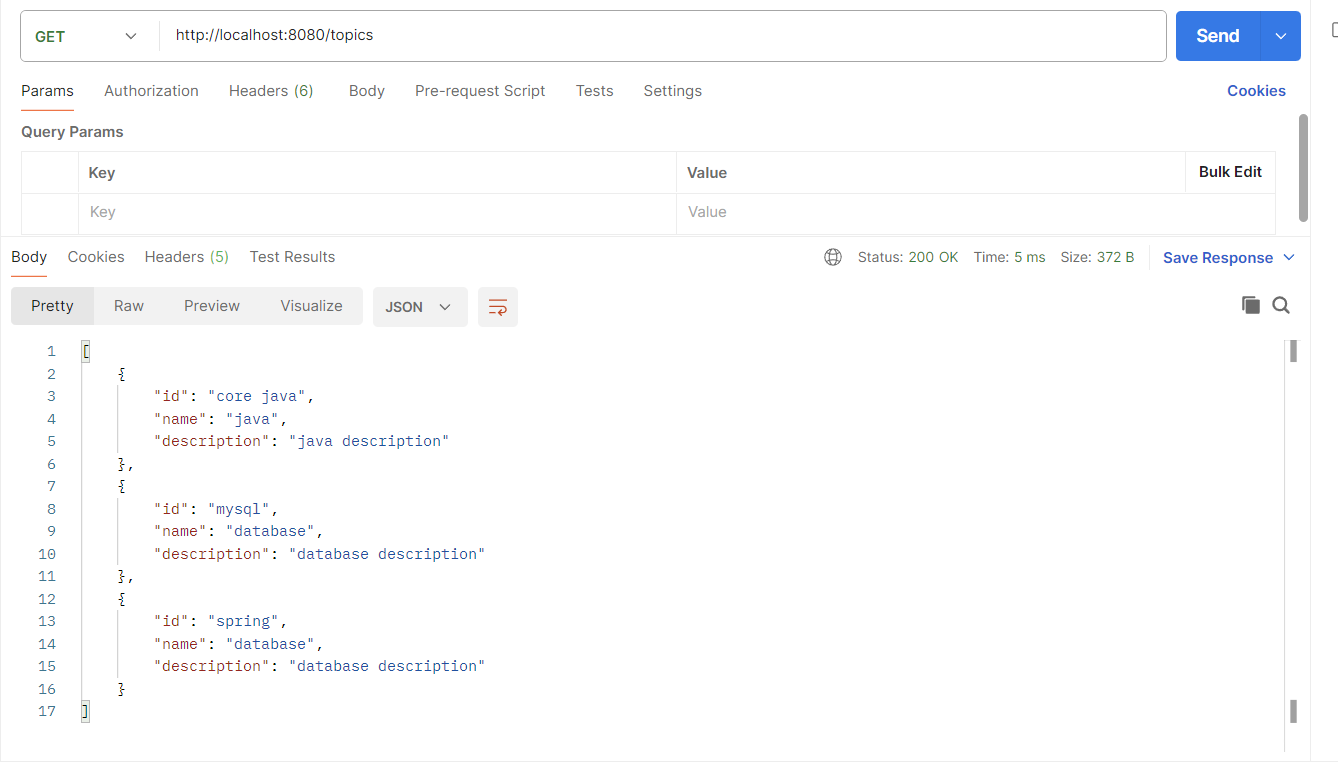
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

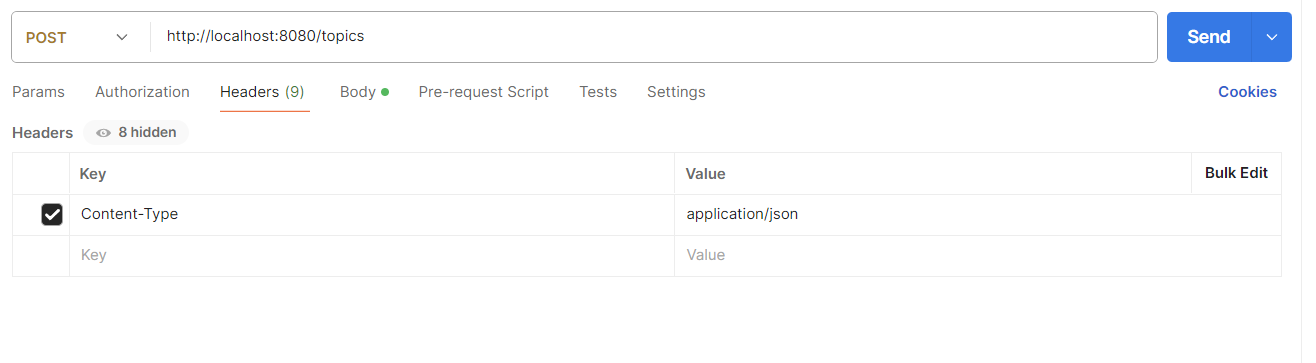
</dependency>

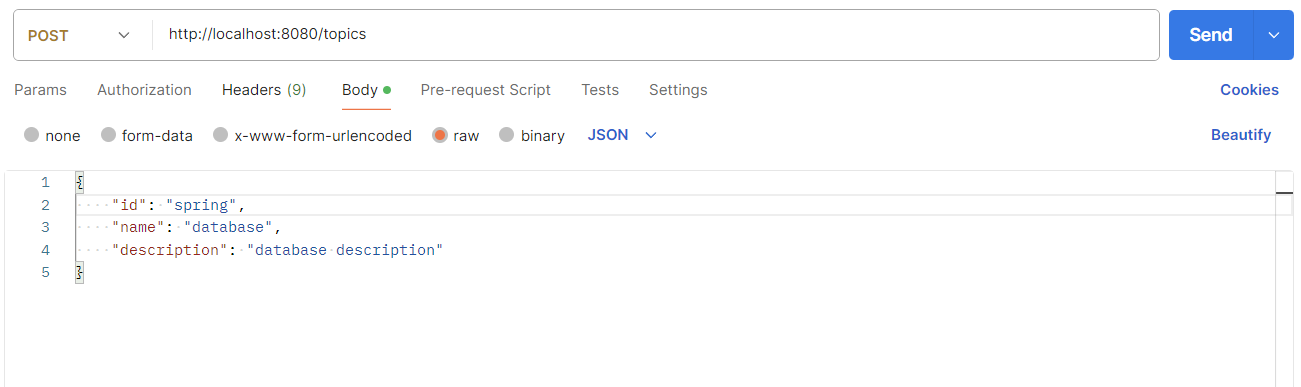


GET

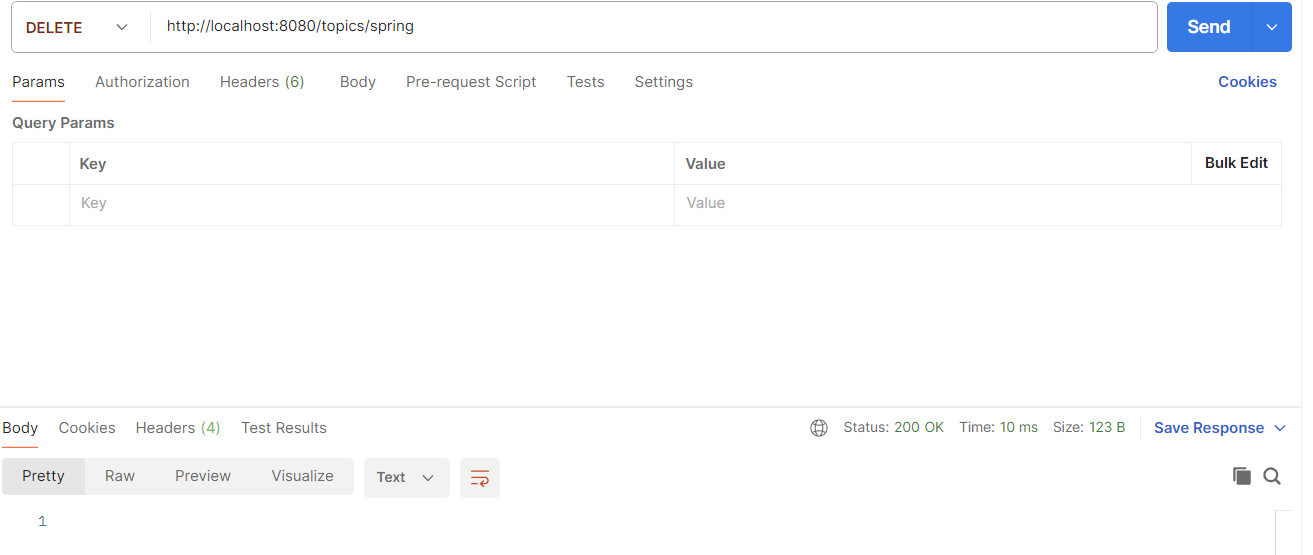


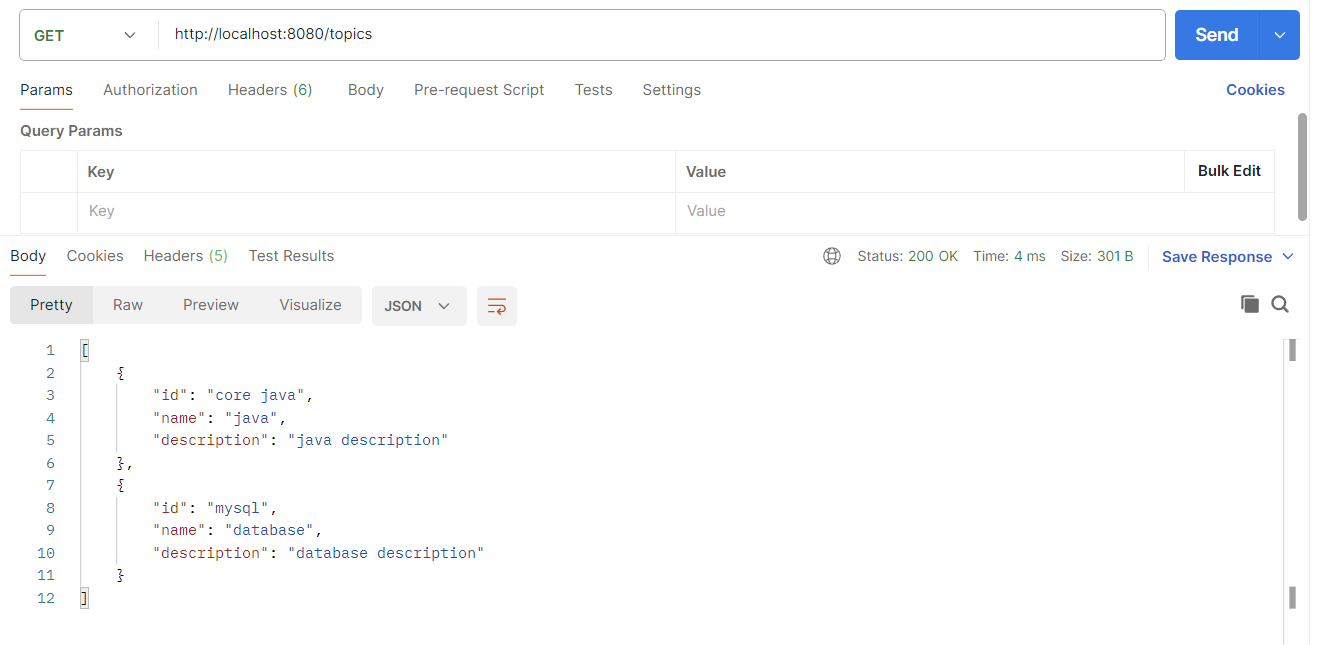
POST



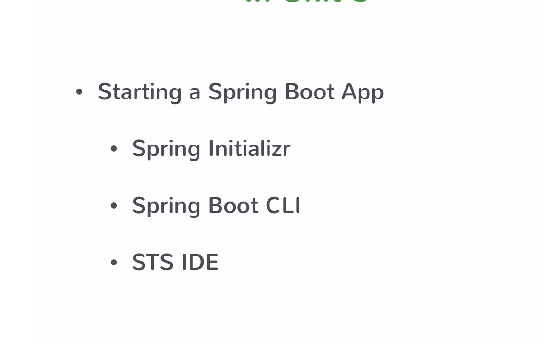


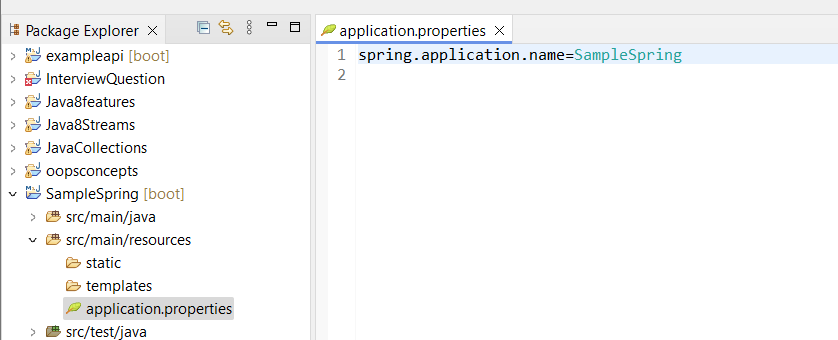
DELETE





1. Now we can create spring boot application using below spring tools.





**application.properties** file is used to write the application-related property into that file. This file contains the different configuration which is required to run the application in a different environment, and each environment will have a different property defined by it. Inside the application properties file, we define every type of property like changing the port, database connectivity, connection to the eureka server, and many more. Now let’s see some examples for better understanding

**Common Application Properties**

<https://docs.spring.io/spring-boot/appendix/application-properties/index.html>

**What is JPA**

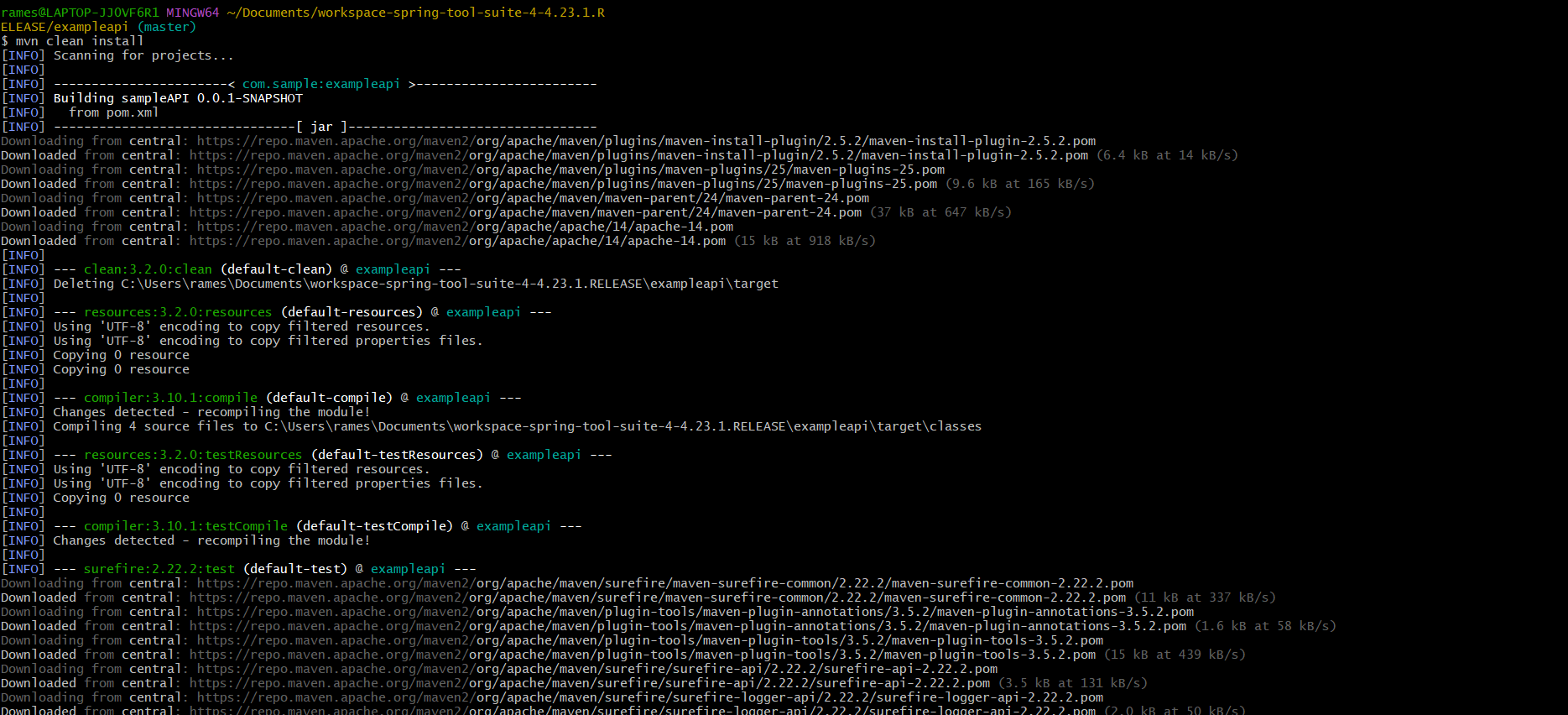
JPA is a Java specification for persisting data between Java objects and relational databases.

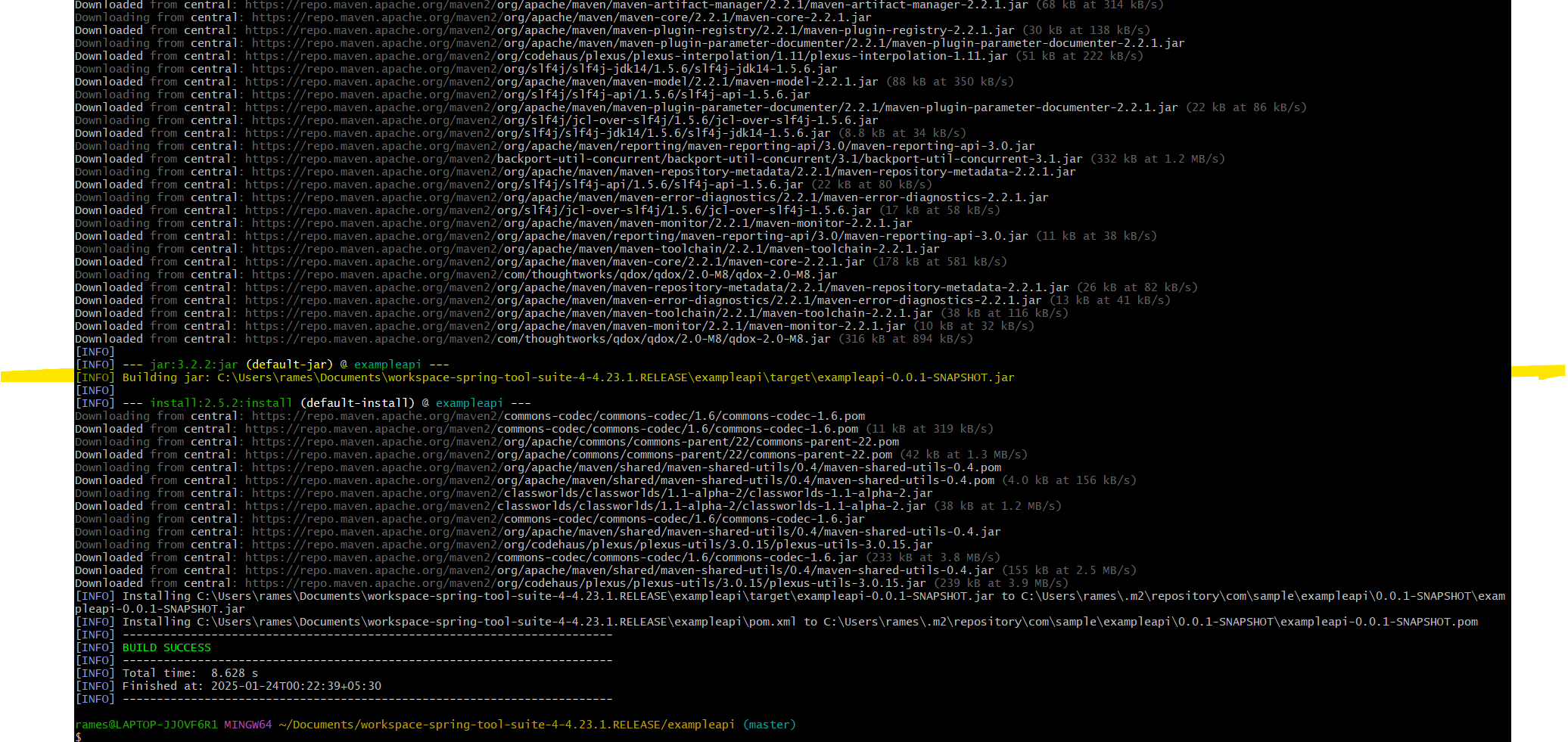
The Java Persistence API (JPA) is a specification of Java. It is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems.

As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.

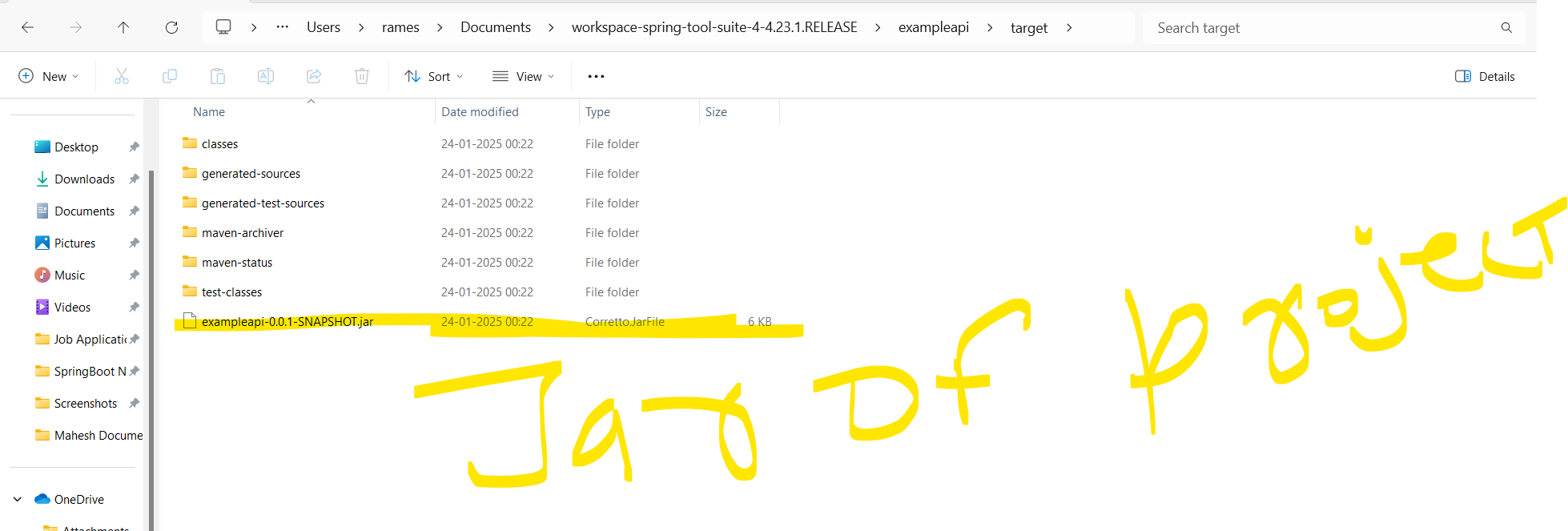
**Packaging and running a Spring Boot app**

give same command from pom.xml file has present.  
has maven build using pom.xml and provide jar/war file  
mvn clean install

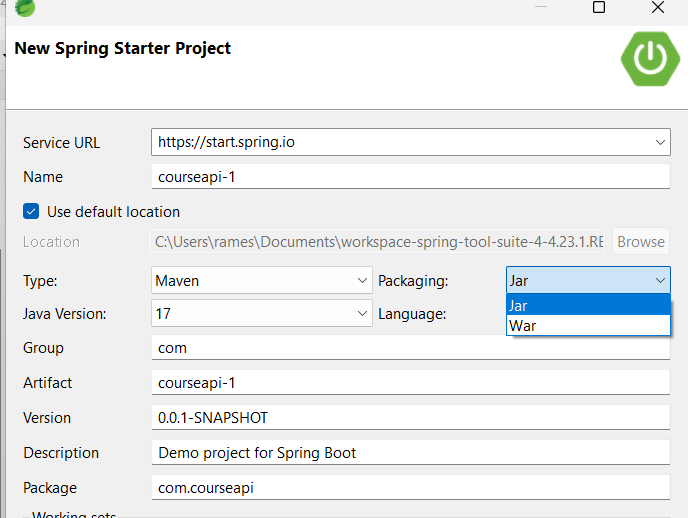




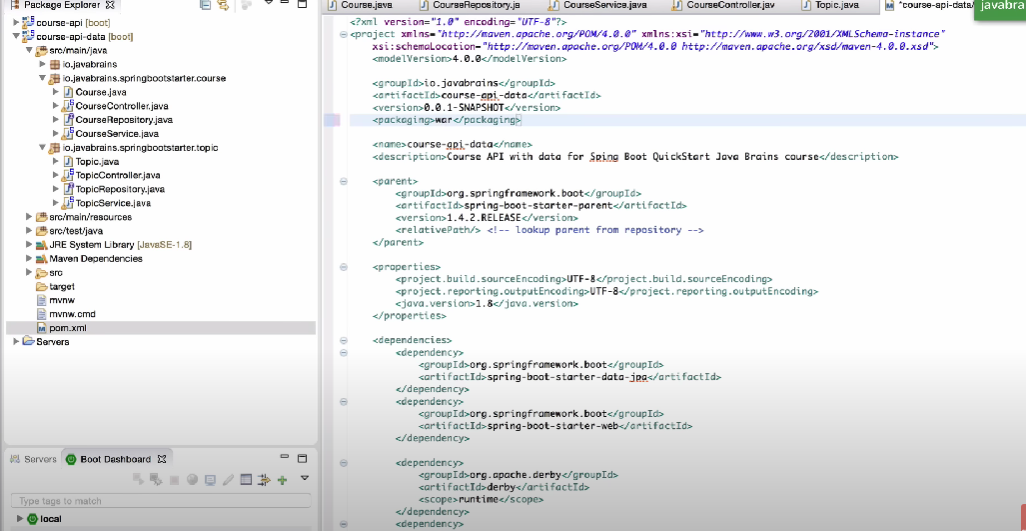
Will see building .jar file location of project also.



For war file , while creating project select war or change innpom.xml also



Or



And give same command from pom.xml file has present.  
has maven build using pom.xml and provide jar/war file  
mvn clean install

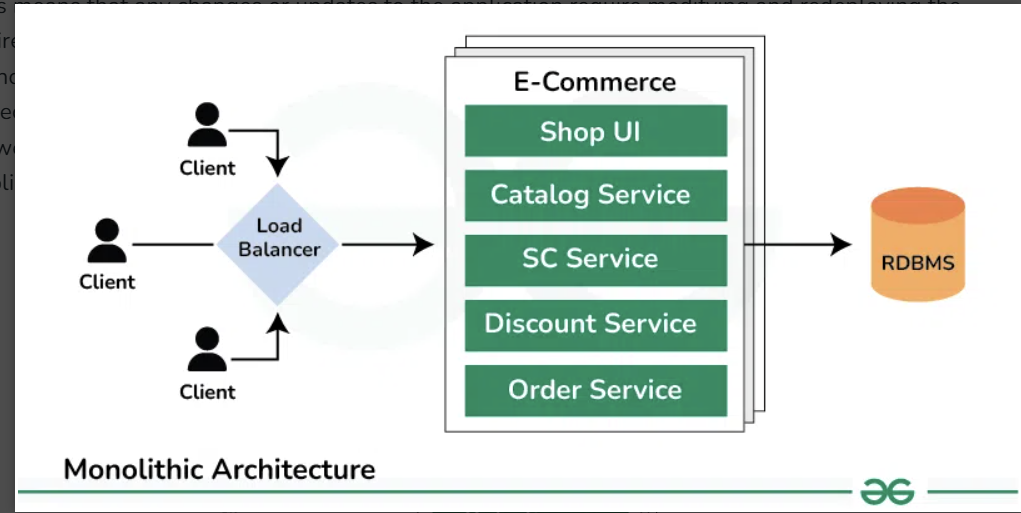
**Java microservices**are a set of small, independent applications written in Java that work together to form a larger system. Each microservice has a specific, limited scope and runs in its own process, communicating with other services using lightweight protocols like HTTP or messaging.

[Microservices](https://www.geeksforgeeks.org/microservices/) is an architectural approach where a large software application is decomposed into smaller, independently deployable services, each responsible for a specific business capability.

**What is a**[Monolithic Architecture](https://www.geeksforgeeks.org/monolithic-architecture-system-design/)**?**

Software is traditionally designed using a monolithic architecture, in which the entire program is constructed as a single, indivisible unit. Every component of the program, including the data access layer, business logic, and user interface, is deployed and integrated tightly together in this design.

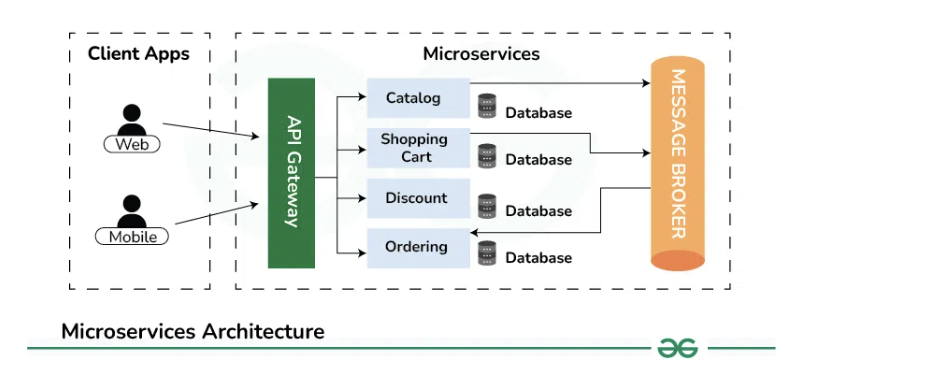
* This means that any changes or updates to the application require modifying and redeploying the entire monolith.
* Monolithic architectures are often characterized by their simplicity and ease of development, especially for small to medium-sized applications.
* However, they can become complex and difficult to maintain as the size and complexity of the application grow.



**What is a**[**Microservices Architecture**](https://www.geeksforgeeks.org/microservices/)**?**

A microservices architecture results in an application designed as a set of small, independent services. Each one represents a business capability in itself. The services loosely couple with one another and communicate over the network, typically making use of lightweight protocols such as HTTP or messaging queues.

* Each service is responsible for a single functionality or feature of the application and can be developed, deployed, and scaled independently.
* The Microservice architecture has a significant impact on the relationship between the application and the database.



**Differences between Monolithic and Microservices Architecture**

Below are the differences the Monolithic and Microservice architecture:

| **Aspect** | **Monolithic Architecture** | **Microservice Architecture** |
| --- | --- | --- |
| **Architecture** | Single-tier architecture | Multi-tier architecture |
| **Size** | Large, all components tightly coupled | Small, loosely coupled components |
| **Deployment** | Deployed as a single unit | Individual services can be deployed independently |
| **Scalability** | Horizontal scaling can be challenging | Easier to scale horizontally |
| **Development** | Development is simpler initially | Complex due to managing multiple services |
| **Technology** | Limited technology choices | Freedom to choose the best technology for each service |
| **Fault Tolerance** | Entire application may fail if a part fails | Individual services can fail without affecting others |
| **Maintenance** | Easier to maintain due to its simplicity | Requires more effort to manage multiple services |
| **Flexibility** | Less flexible as all components are tightly coupled | More flexible as components can be developed, deployed, and scaled independently |
| **Communication** | Communication between components is faster | Communication may be slower due to network call |

**Microservice project example**

Microservices differ based on:

1. **Domain Functionality**:
   * Each microservice should map to a specific business domain. For example, in an e-commerce system:
     + **Order Service**: Handles orders.
     + **Payment Service**: Processes payments.
     + **Inventory Service**: Manages stock levels.
2. **Data Ownership**:
   * Each microservice has its own database to avoid shared state. For example:
     + Order Service → orders database.
     + Inventory Service → inventory database.
3. **Independence**:
   * Microservices can be developed, deployed, and scaled independently.
   * Changes in one microservice do not require redeploying others.
4. **Technology Stack**:
   * Each microservice can use its own language, framework, or database.
5. **Communication**:
   * Microservices communicate using lightweight protocols like HTTP REST APIs or messaging systems (Kafka, RabbitMQ).

**E-Commerce Microservices**

**1. Order Service:**

* REST endpoint: /orders
* Database: orders\_db
* API: Create, update, or retrieve orders.

**2. Payment Service:**

* REST endpoint: /payments
* Database: payments\_db
* API: Process payments, check payment status.

**3. Inventory Service:**

* REST endpoint: /inventory
* Database: inventory\_db
* API: Get stock details, update stock.

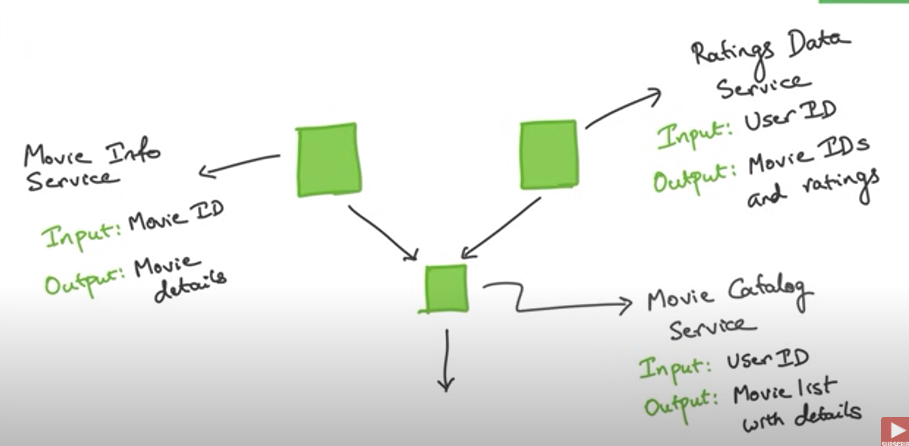
**Key Tools and Technologies for Microservices in Spring Boot**

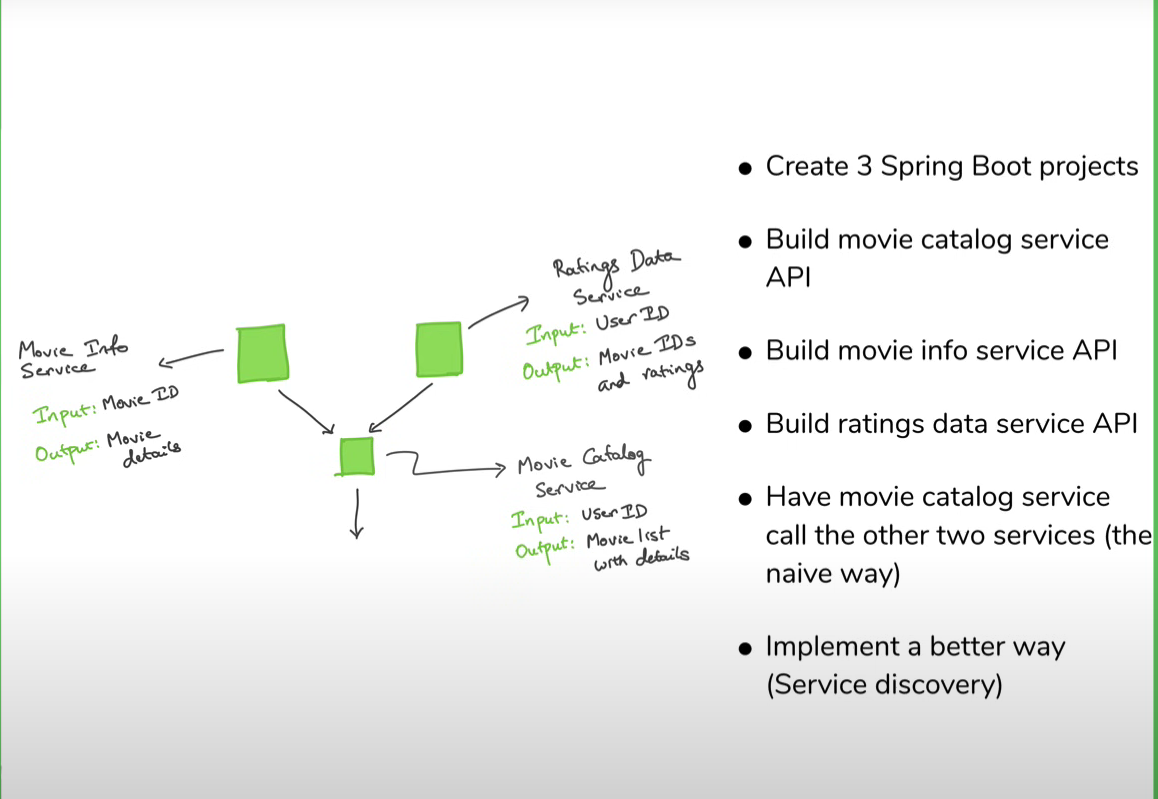
1. **Spring Boot Components**:
   * Spring Cloud: For service discovery (Eureka), configuration (Config Server), and resilience (Hystrix).
   * Spring Data JPA: For database operations.
   * Spring Security: For authentication and authorization.
   * Spring Actuator: For monitoring.

**Communication between microservices**

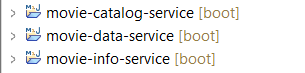
Using **service discovery**

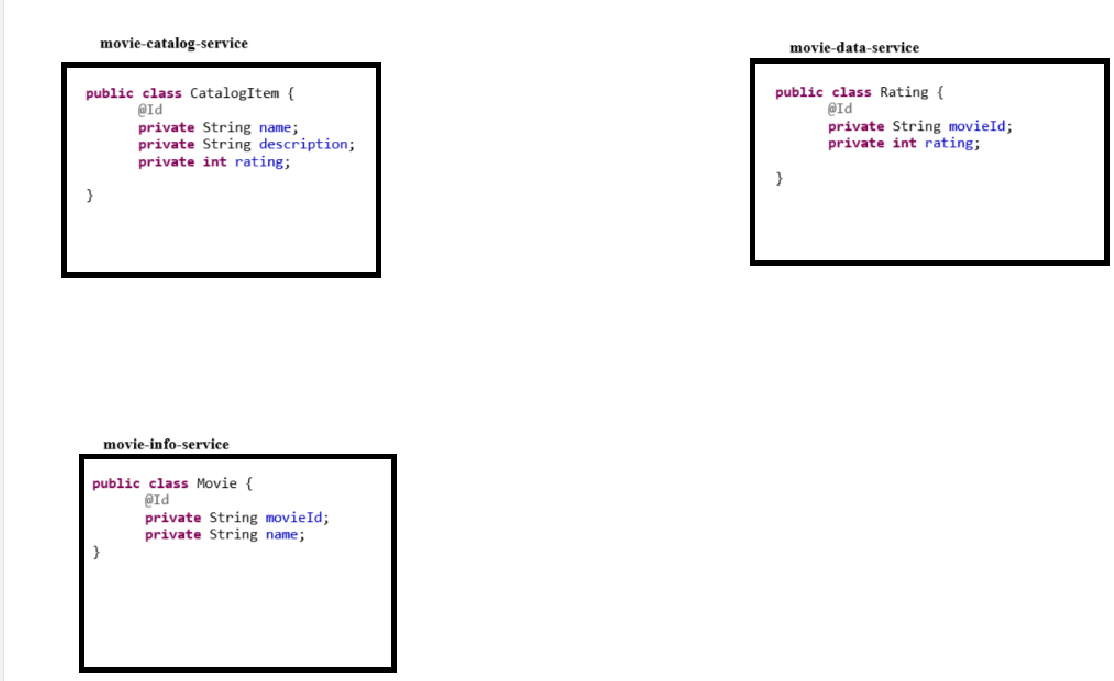
**Ex**

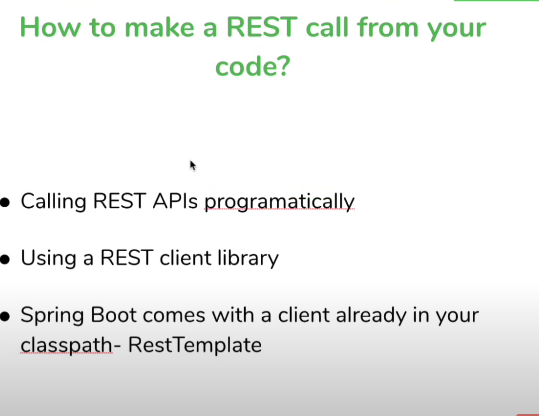




We have to create 3 spring boot project first







RestTemplate to make a REST API call from the Spring controller directly to another microservice and unmarshall the response into an object instance.

the RestTemplate is a utility class provided by Spring Framework to make REST API calls. It allows your application to send HTTP requests to other services and process their responses.

RestTemplate restTemplate = new RestTemplate();

The getForObject method in Spring's RestTemplate is used to make an HTTP GET request and directly retrieve the response body as an object.

**Basic Example**

**Fetching a Simple String**

RestTemplate restTemplate = new RestTemplate();

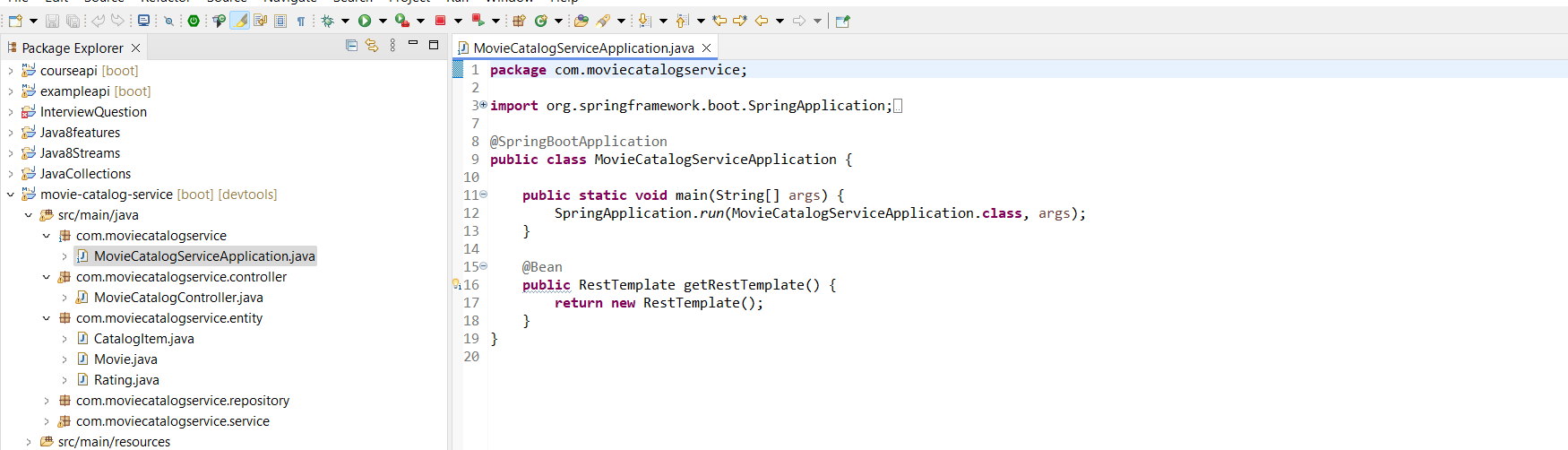
String url = "https://api.example.com/message";

String response = restTemplate.getForObject(url, String.class);

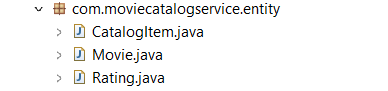
System.out.println("Response: " + response);

@Autowired is like a consumer , it will ask spring like give me something from defined location.

@Bean is like producer , it will tell spring that I have something so execute method and give where it need.



Added





1. **Endpoint**:
   * The @GetMapping("/{userId}") annotation indicates that this method is triggered by a **GET** request to the endpoint /catalog/{userId}.
2. **Input**:
   * The @PathVariable String userId binds the userId from the URL to the userId parameter in the method. The userId is currently not being used in the method, but could be used for filtering or other purposes in a real-world scenario.
3. **Movie ID**:
   * For demonstration, the method sets a fixed movieId = "1" to simulate fetching data for a specific movie.
4. **Movie Info Service Call**:
   * The restTemplate.getForObject("http://localhost:8082/movie/" + movieId, Movie.class) makes a **GET** request to the Movie Info Service running on http://localhost:8082, requesting movie data for the movieId.
   * The result is mapped to the Movie class.
5. **Ratings Data Service Call**:
   * Similarly, restTemplate.getForObject("http://localhost:8083/ratingsdata/" + movieId, Rating.class) makes another **GET** request to the Ratings Data Service at http://localhost:8083 to fetch the movie's rating.
   * The result is mapped to the Rating class.
6. **CatalogItem Creation**:
   * Using the retrieved movie and rating objects, a new CatalogItem is created with the movie's name, a placeholder description ("Description not available"), and the movie's rating.
7. **Return**:
   * The method returns a list containing one CatalogItem object, which is then sent as the response.

**Example Result in Postman**

When you make a **GET** request to http://localhost:8080/catalog/1 in Postman (with userId as 1):

**Response**:

[

{

"name": "rakshadu",

"description": "Description not available",

"rating": 5

}

]

This is a JSON response that includes the movie name (rakshadu), a default description (Description not available), and a rating (5).

**WebClient**

WebClient is a non-blocking, reactive alternative to RestTemplate for making HTTP requests in Spring applications. It was introduced in Spring 5 as part of the **Spring WebFlux** module and is used to make asynchronous, reactive HTTP calls. WebClient provides better performance for high-concurrency applications because it does not block the execution while waiting for HTTP responses.

Here's a comparison of WebClient vs. RestTemplate:

* **RestTemplate** is synchronous and blocking, meaning it waits for a response before moving to the next instruction.
* **WebClient** is asynchronous and non-blocking, allowing for parallel HTTP requests without blocking the thread.

**How to Replace RestTemplate with WebClient**

Let's refactor your controller method using WebClient.

1. **Add WebClient Dependency**: If you're using Spring Boot, make sure to include the spring-boot-starter-webflux dependency:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

1. **Create WebClient Bean**: In your configuration class, you can define a WebClient bean:

@Configuration

public class WebClientConfig {

@Bean

public WebClient.Builder webClientBuilder() {

return WebClient.builder();

}

}

1. **Refactor the Controller to Use WebClient**: Here's how you can refactor your getCatalog method to use WebClient instead of RestTemplate:

@RestController

@RequestMapping("/catalog")

public class CatalogController {

private final WebClient.Builder webClientBuilder;

@Autowired

public CatalogController(WebClient.Builder webClientBuilder) {

this.webClientBuilder = webClientBuilder;

}

@GetMapping("/{userId}")

public Mono<List<CatalogItem>> getCatalog(@PathVariable String userId) {

String movieId = "1"; // Example movieId

WebClient webClient = webClientBuilder.baseUrl("http://localhost:8082").build();

Mono<Movie> movieMono = webClient.get()

.uri("/movie/{movieId}", movieId)

.retrieve()

.bodyToMono(Movie.class);

WebClient ratingsWebClient = webClientBuilder.baseUrl("http://localhost:8083").build();

Mono<Rating> ratingMono = ratingsWebClient.get()

.uri("/ratingsdata/{movieId}", movieId)

.retrieve()

.bodyToMono(Rating.class);

return Mono.zip(movieMono, ratingMono, (movie, rating) -> {

CatalogItem catalogItem = new CatalogItem(movie.getName(), "Description not available", rating.getRating());

return Arrays.asList(catalogItem);

});

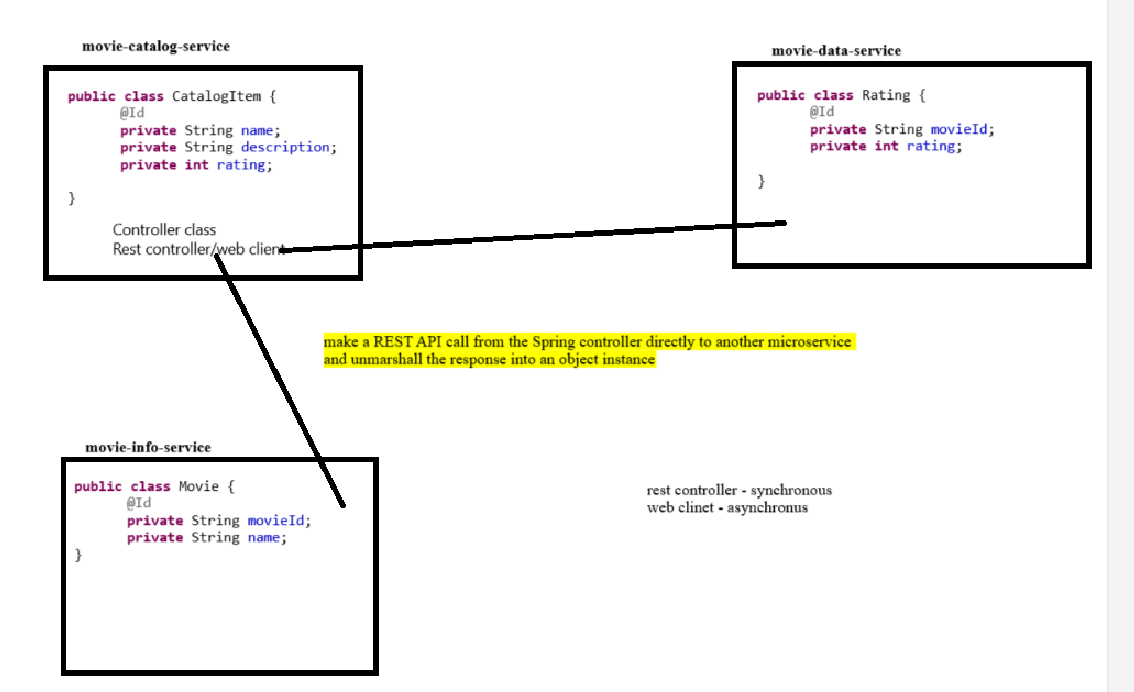
}

}

**Explanation**:

* + WebClient.Builder is injected into the controller and used to configure WebClient instances.
  + Mono<Movie> and Mono<Rating> are reactive wrappers around the result of the WebClient calls. Mono represents a single asynchronous value.
  + Mono.zip is used to combine the results of both WebClient calls (movie and rating) into a single CatalogItem object and return it as a List.
  + The Mono<List<CatalogItem>> return type indicates the response is asynchronous and reactive. It will eventually emit the list of catalog items once the HTTP requests are complete.

1. **Asynchronous Behavior**:
   * This controller method is non-blocking. When you invoke this method, it will not wait for the external service calls to complete before moving to the next request. Instead, it returns a Mono, which will be resolved asynchronously.



Service Discovery

Service discovery is a mechanism used in distributed systems and microservices architectures to automatically detect and connect different services or components in the system. It simplifies communication between services without hardcoding their locations (e.g., IP addresses, ports, or endpoints), making the system more dynamic, scalable, and resilient to changes.

Below is how service discovery typically works:

**1. Service Registration**

When a service instance is started or deployed, it registers itself with a service registry. This registration process includes providing metadata such as the service name, IP address, port number, and health status.

**2. Service Discovery**

Clients looking to interact with a particular service do not need to have prior knowledge of the service's location. Instead, they query the service registry to dynamically discover the available instances of the desired service.

**3.**[Load Balancing](https://www.geeksforgeeks.org/what-is-load-balancer-system-design/)

Service discovery often incorporates load-balancing mechanisms to distribute incoming requests across multiple instances of the same service. This helps in improving scalability, fault tolerance, and resource utilization.

**4. Health Checking**

Service discovery systems continuously monitor the health and availability of service instances. Unhealthy or unavailable instances are automatically removed from the registry, ensuring that clients are directed only to healthy and operational instances.

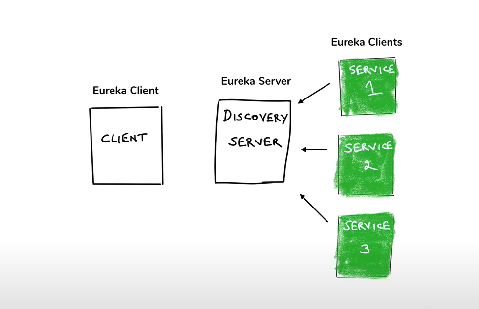
**5. Dynamic Configuration**

Service discovery can also be used for dynamic configuration management, allowing services to discover and retrieve configuration settings or properties from a centralized configuration store.

Overall, service discovery simplifies the complexities of microservices communication by providing a centralized mechanism for service registration, discovery, and resolution. It promotes agility, scalability, and resilience in distributed systems, enabling seamless communication and collaboration between microservices.

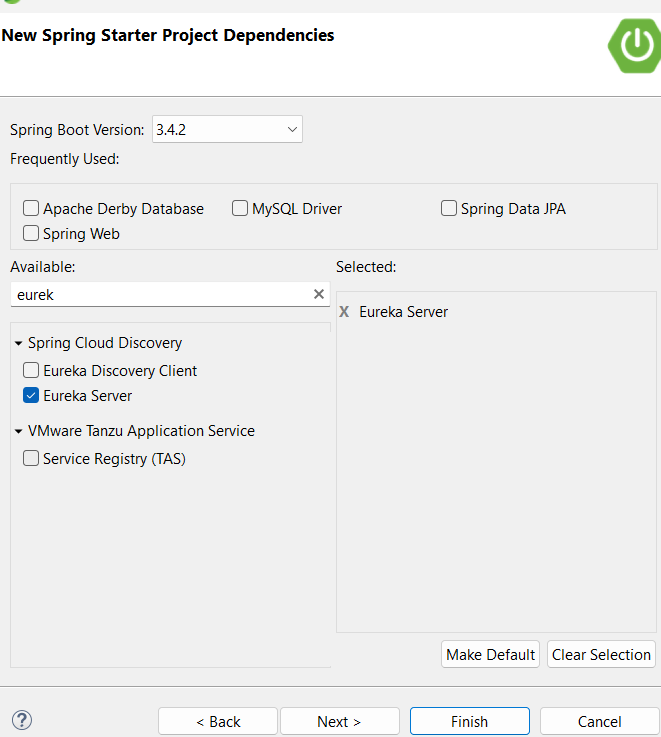
**Types of Service Discovery:**

1. **Client-Side Discovery**:
   * The client queries the service registry to get the location of a service and connects directly to the service instance.
   * Example tools: Netflix Eureka, Consul.
2. **Server-Side Discovery**:
   * The client sends a request to a load balancer or proxy, which queries the service registry to find an appropriate service instance and forwards the request.
   * Example tools: AWS Elastic Load Balancer, Kubernetes.

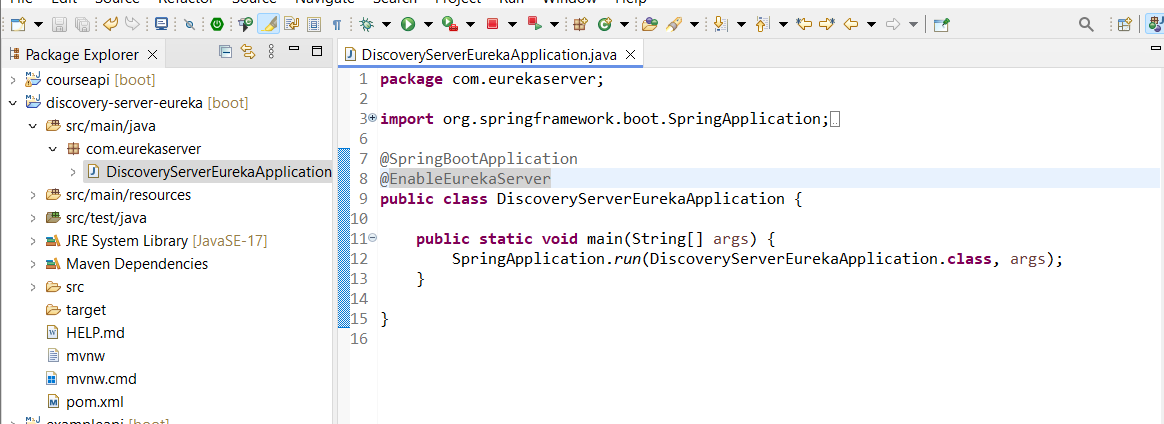


* + 1. **Created spring boot project with Eureka sever dependency**

****

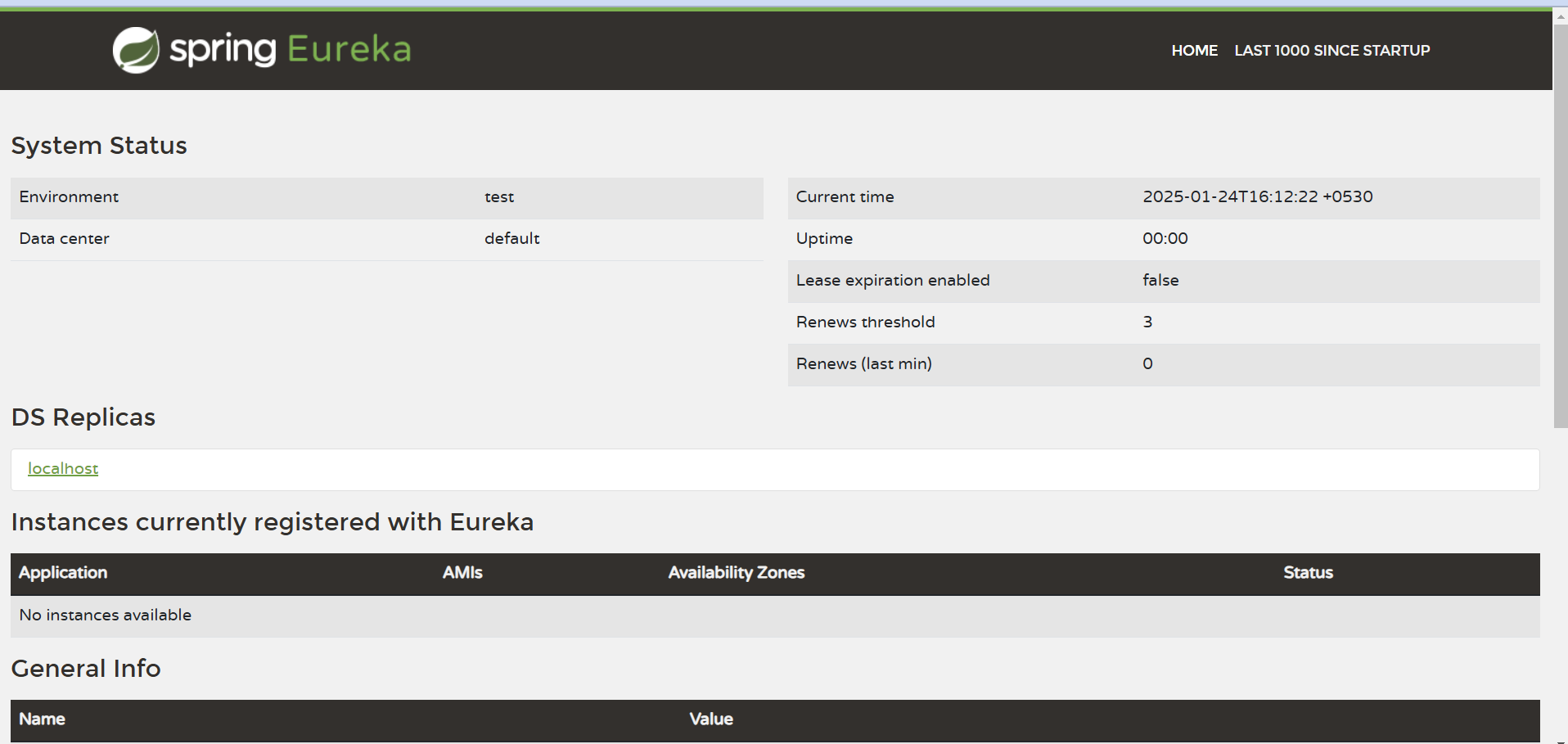


* + 1. Enabale EurekaServer by adding @EnableEurekaServer annotation on main class

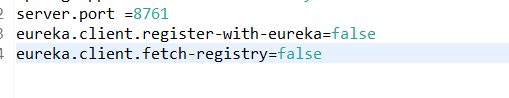


By default Eureka server run on 8761

<http://localhost:8761>



* + 1. Add Common **application.properties** for eureka,



**eureka.client.register-with-eureka=false**

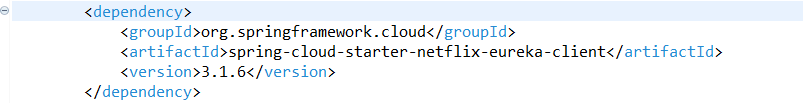
This is correct because the Eureka server shouldn't register itself as a client. This configuration disables the Eureka server from attempting to register as a service.

**eureka.client.fetch-registry=false**

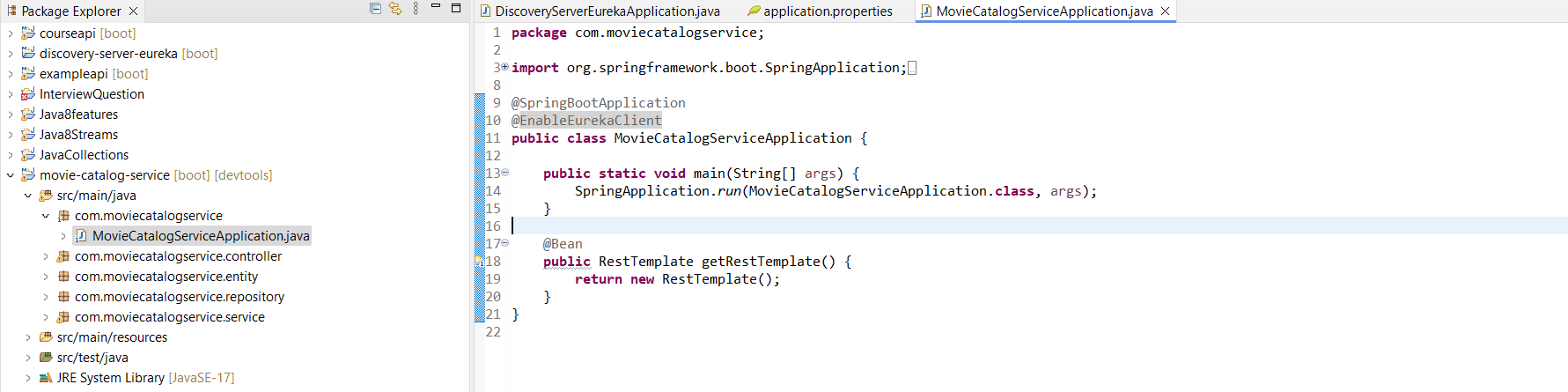
* This disables the fetching of the Eureka registry from the server.
* In this case, the application will not retrieve the list of services from the Eureka server and cannot perform service discovery.

**Client**

* + 1. **Add spring cloud and Eureka-client dependencies in microservices(clients) which we want to register with Eureka server**

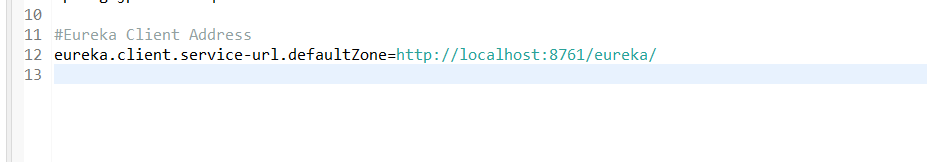
****

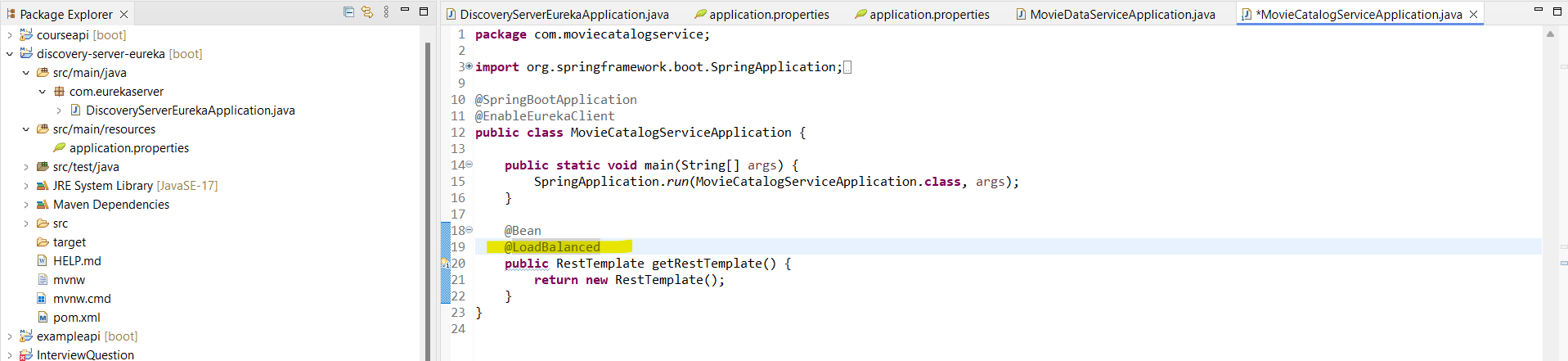
* + 1. **Enable Eurika client adding annotation @EnableEurekaClient in microservices main class**



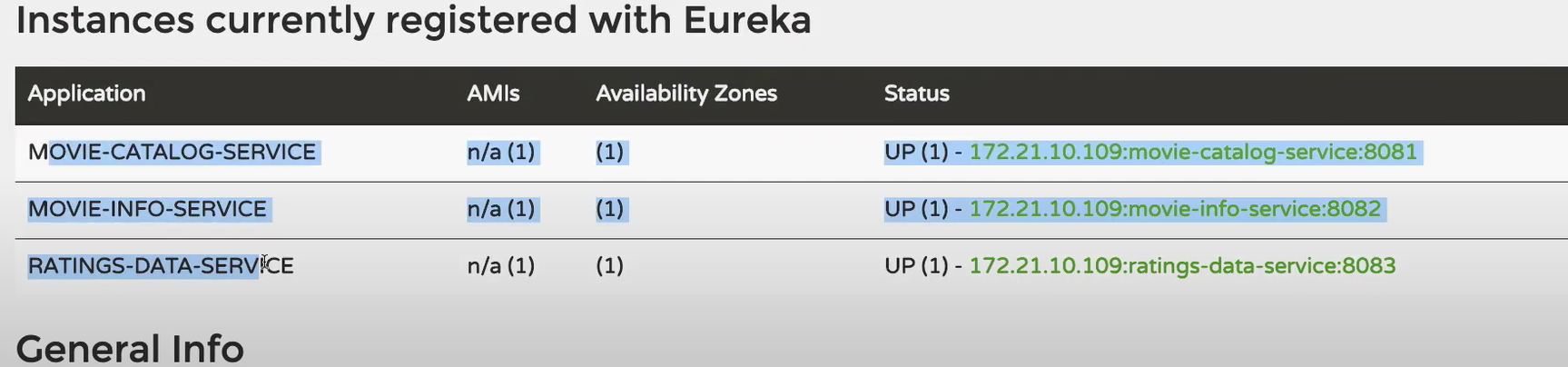
* + 1. **Application Properties (for Client):** You need to ensure the Eureka client is configured to point to the Eureka server. Typically, the following configuration is added to the client’s application.properties or application.yml:

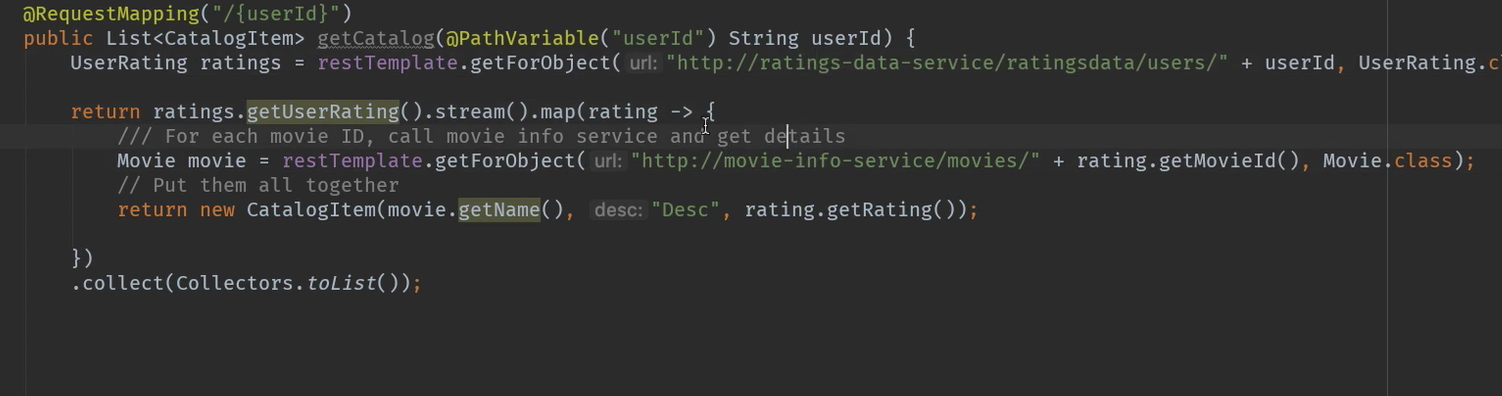
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/





@LoadBalanced : It will tell to rest template that don’t go service directly and I am giving you a sample url but it is not the original url , it will give hint what actually the service does /discover .





**API Gateway**

An API Gateway acts as a single entry point for client applications to communicate with backend services. It simplifies interactions between clients and microservices by handling various cross-cutting concerns.

**Key Responsibilities:**

1. **Request Routing:** Routes incoming client requests to the appropriate microservices.
2. **Authentication and Authorization:** Validates user credentials and ensures secure access to backend services.
3. **Load Balancing:** Distributes requests among service instances to ensure optimal performance.
4. **Rate Limiting and Throttling:** Controls the rate of incoming requests to prevent service overload.
5. **Data Transformation:** Transforms request and response formats to suit client needs.
6. **Caching:** Stores frequent responses to reduce backend load and improve response times.
7. **Monitoring and Logging:** Tracks requests and responses for debugging and analytics.

**Popular API Gateway Tools:**

* **Kong**
* **AWS API Gateway**
* **NGINX**
* **Spring Cloud Gateway**
* **Apigee**

**Service Discovery**

Service Discovery ensures that services within the architecture can dynamically locate and communicate with each other, especially in environments where service instances frequently scale up, down, or move.

**Key Concepts:**

1. **Service Registry:** A centralized directory where service instances register their availability and metadata (e.g., host, port, protocol).
2. **Service Providers:** Microservices that register themselves with the service registry when they start and deregister when they shut down.
3. **Service Consumers:** Microservices or API Gateways that query the registry to find service instances they need to communicate with.

**Types of Service Discovery:**

* **Client-Side Discovery:**
  + The client queries the service registry directly to find available service instances and sends requests to one of them.
  + Example Tool: Netflix Eureka
* **Server-Side Discovery:**
  + The client sends a request to the API Gateway or Load Balancer, which queries the service registry and routes the request to an appropriate service instance.
  + Example Tools: AWS Elastic Load Balancing, Kubernetes Services

**Popular Service Discovery Tools:**

* **Netflix Eureka**
* **Consul**
* **Zookeeper**
* **Etcd**
* **Kubernetes DNS**

**Technical Stack Example**

1. **Service Registry**: **Eureka Server**
   * Microservices (Product, Order, Payment) register themselves with Eureka.
   * Eureka handles service lookup.
2. **Service Discovery**: Spring Boot’s **Eureka Client**
   * Microservices use Spring Cloud Netflix Eureka for service discovery.
3. **API Gateway**: **Spring Cloud Gateway**
   * Manages routing, authentication, and load balancing for client requests.

**How API Gateway and Service Discovery Work Together**

1. When a client makes a request, the **API Gateway** acts as the entry point.
2. The API Gateway queries the **Service Registry** to identify the available instances of the target microservice.
3. The Gateway routes the request to a selected service instance, applying any additional logic (e.g., authentication, transformation).
4. Responses are sent back to the client via the API Gateway.

Bean

**Simple example** of the **Spring Bean Life Cycle** using annotations:

**1. Bean Class with Lifecycle Hooks**

import jakarta.annotation.PostConstruct;

import jakarta.annotation.PreDestroy;

import org.springframework.stereotype.Component;

@Component

public class MyBean {

public MyBean() {

System.out.println("Constructor: Bean is instantiated");

}

@PostConstruct

public void init() {

System.out.println("@PostConstruct: Bean is initialized");

}

public void doSomething() {

System.out.println("Business Logic: Bean is doing work");

}

@PreDestroy

public void cleanup() {

System.out.println("PreDestroy: Bean is about to be destroyed");

}

}

**2. Spring Configuration & Runner**

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

@SpringBootApplication

public class BeanLifecycleApp {

public static void main(String[] args) {

SpringApplication.run(BeanLifecycleApp.class, args);

}

@Bean

public CommandLineRunner runner(MyBean myBean) {

return new CommandLineRunner() {

@Override

public void run(String... args) {

myBean.doSomething();

}

};

}

**🖨️ Output on Console:**

Constructor: Bean is instantiated

@PostConstruct: Bean is initialized

Business Logic: Bean is doing work

PreDestroy: Bean is about to be destroyed

break down the example step by step so you can clearly see how the **Spring Bean life cycle** works in action:

**🔹 MyBean Class Breakdown**

@Component

public class MyBean {

* @Component tells Spring that this class is a **bean** and should be **automatically detected** and managed by the Spring container (thanks to component scanning).

public MyBean() {

System.out.println("1️⃣ Constructor: Bean is instantiated");

}

* This is the **constructor**.
* It's the **first step** in the bean life cycle where Spring **creates the object** of MyBean.

@PostConstruct

public void init() {

System.out.println("2️⃣ @PostConstruct: Bean is initialized");

}

* @PostConstruct is called **after the bean is constructed and dependencies are injected**.
* This is where you can write **initialization logic** (e.g., connecting to a DB, loading config, etc.).

public void doSomething() {

System.out.println("💼 Business Logic: Bean is doing work");

}

* This method simulates **actual work** the bean does in the application — your normal business logic.

@PreDestroy

public void cleanup() {

System.out.println("3️⃣ @PreDestroy: Bean is about to be destroyed");

}

* @PreDestroy is called **before the bean is destroyed** (typically when the Spring context is shutting down).
* Here you write **cleanup logic**, like releasing resources, closing files, or connections.

**🔹 BeanLifecycleApp (Main Spring Boot App)**

@SpringBootApplication

public class BeanLifecycleApp {

* This is your main Spring Boot application.
* @SpringBootApplication enables component scanning and auto-configuration.

@Bean

CommandLineRunner runner(MyBean myBean) {

return args -> {

myBean.doSomething();

};

}

* CommandLineRunner runs **immediately after the app starts**.
* We inject MyBean into it, and call the doSomething() method to simulate usage of the bean.

**💡 Flow of Execution**

When the app runs:

1. ✅ **Spring creates the bean** → constructor runs.
2. 🛠️ **Spring injects dependencies** → then calls @PostConstruct.
3. 🚀 The doSomething() method is triggered by the runner.
4. 🧹 Before shutdown, Spring calls @PreDestroy for cleanup.