**Rest Using Spring Boot**

1. **Rest Using Spring Boot Introduction**

REST (Representational State Transfer) is an architectural style for designing networked applications. It relies on stateless, client-server communication, typically using HTTP methods like GET, POST, PUT, and DELETE to operate on resources. RESTful APIs enable seamless communication between the server and client applications, making it a popular approach for building scalable, maintainable web services.

Spring Boot, an extension of the Spring framework, is widely used to build REST APIs quickly and with minimal configuration. It provides an easy, opinionated way to create stand-alone, production-ready Spring-based applications. With built-in features such as automatic configuration, embedded servers, and a robust ecosystem, Spring Boot simplifies the process of creating RESTful services.

**Key Concepts of REST Using Spring Boot**

* HTTP Methods as CRUD Operations:
  + GET: Retrieve resources from the server.
  + POST: Create new resources on the server.
  + PUT: Update existing resources.
  + DELETE: Remove resources.
* Resources and URIs: In a RESTful API, resources are represented by URIs (Uniform Resource Identifiers). For example, a URL like /api/employees may represent all employee resources, while /api/employees/{id} would represent a single employee.
* Stateless Communication: REST is stateless, meaning that each request from the client to the server must contain all the information needed to understand and process the request. No session state is maintained between requests on the server.

1. **What is Rest**

REST (Representational State Transfer) is an architectural style for designing networked applications. RESTful services use HTTP as their communication protocol and work with resources, typically represented by URLs. REST is based on a stateless client-server communication model, meaning each request from the client to the server must contain all necessary information for the server to understand and respond to it.

**Key Characteristics of REST**:

**Stateless**: Every client request is independent of others; no session information is stored on the server.

**Client-Server**: The client is responsible for the user interface, while the server manages data and resources.

**Uniform Interface**: REST provides a standardized way to communicate using HTTP methods like GET, POST, PUT, DELETE.

**Resource-Based**: Resources are defined by URIs (Uniform Resource Identifiers) and represent different entities, such as users or orders.

REST is used to create scalable, easy-to-maintain APIs that allow interaction between applications over the web. It is widely adopted due to its simplicity, reliability, and use of standard HTTP protocols.

1. **What is Http Method**

**HTTP** methods are standard verbs that represent different operations on resources in RESTful services. They are used to perform CRUD (Create, Read, Update, Delete) operations:

* **GET**: Used to retrieve data from a server. It does not alter any data.
* **POST**: Used to create a new resource on the server.
* **PUT**: Used to update an existing resource or create a new resource if it doesn’t exist.
* **DELETE**: Used to delete a resource on the server.

**Additional HTTP Methods**:

* **PATCH**: Similar to PUT, but it only partially updates a resource.
* **OPTIONS**: Used to describe the communication options for the target resource.

HTTP methods form the backbone of RESTful APIs, allowing clients to interact with server resources in a standardized manner.

1. **Creating a Rest Controller**

A REST Controller is a specialized version of a controller in Spring Boot that is used to expose RESTful endpoints to clients. In Spring Boot, the @RestController annotation is used to create RESTful web services. Let's take an example from the code provided.



**4.1 Explanation**:

**RestController**: This annotation is used to indicate that this class will handle RESTful web requests. It combines @Controller and @ResponseBody, meaning each method will return data directly instead of rendering a view (like HTML).

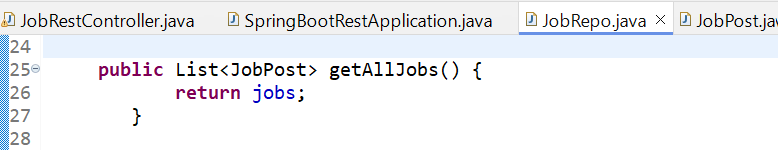
**CrossOrigin**: This annotation enables Cross-Origin Resource Sharing (CORS) for the specified origins. In this case, it's allowing requests from http://localhost:5500. This is useful when you need to enable interaction with the API from different origins.

**Autowired**: This is used to inject the JobService into the controller. Spring automatically wires the dependency so that the controller can access the methods in the service layer.

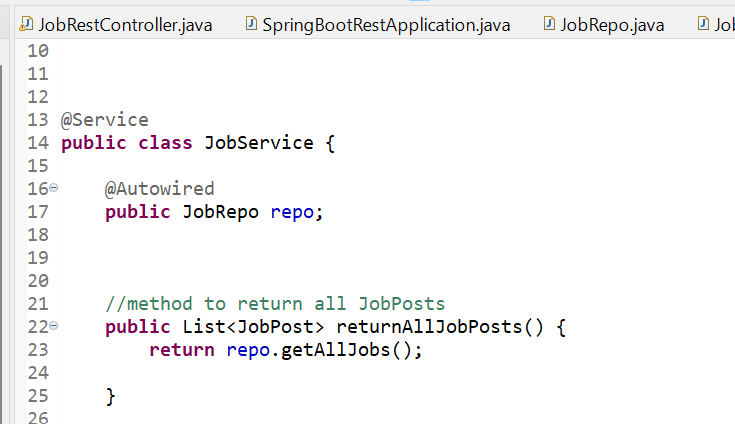
**GetMapping("jobPosts"):** This annotation maps HTTP GET requests to the method. It handles requests coming to /jobPosts and returns a list of job posts using service.returnAllJobPosts().

This is a straightforward REST controller where we expose an endpoint to get all job posts. The data flow happens from the controller to the service and then to the repository, which interacts with the data source.

Respective Repo class



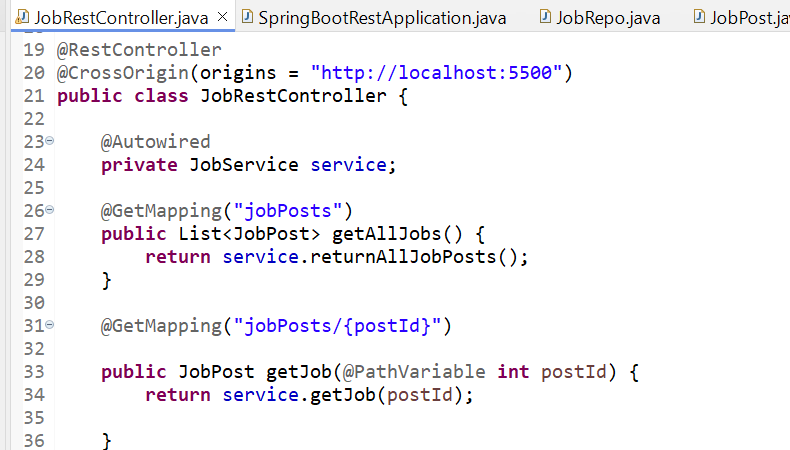
Respective service class



1. **PathVariable**

Sometimes, we want to pass dynamic values to our REST endpoint as part of the URL. Spring Boot allows us to achieve this using the @PathVariable annotation.

Let's modify the existing code to add a new method that returns a specific JobPost by its postId.



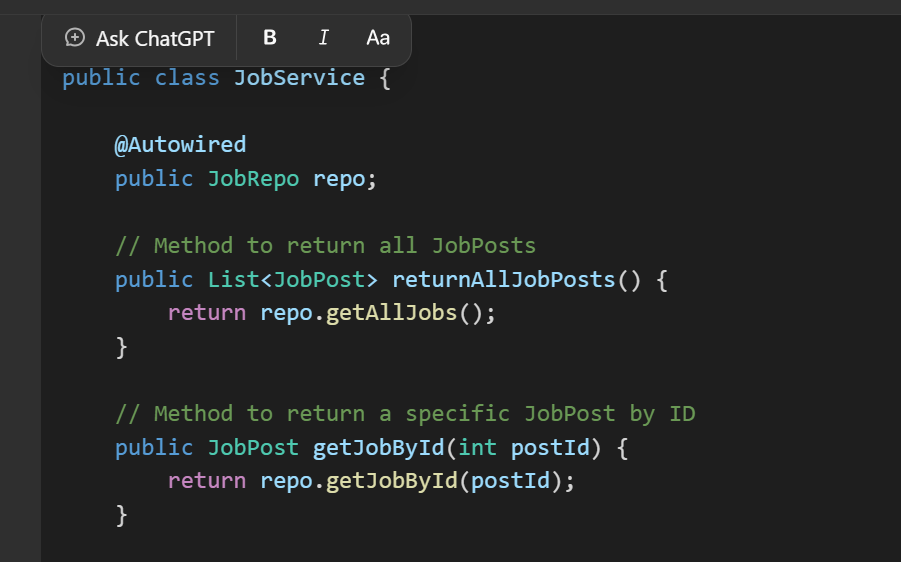
**Explanation of Changes:**

**GetMapping("jobPosts/{postId}"):** This maps an HTTP GET request to a URL like /jobPosts/1, where 1 is the postId. The {postId} part of the URL is dynamic, meaning it can be replaced by any valid postId value.

**PathVariable int postId**: This annotation binds the value from the URL (e.g., 1 in /jobPosts/1) to the postId parameter in the method. This allows us to use this postId to fetch the specific JobPost.

**public JobPost getJobById(@PathVariable int postId):** This method calls the service.getJobById(postId) to get the job post matching the provided ID. For this to work, we also need to add a method in the JobService and JobRepo classes.

**Updated Service Layer**

****

**Updated Repository Layer**

****

**Explanation of Changes in Repository and Service Layers:**

**public JobPost getJobById(int postId):** This method is added in both JobRepo and JobService classes to support fetching a specific JobPost by postId. In the repository layer, the method filters the list of job posts to find the one that matches the given postId.

By using **@PathVariable**, we have made our API more dynamic, enabling clients to access specific resources directly through the URL. This is a key aspect of building scalable and flexible RESTful APIs.

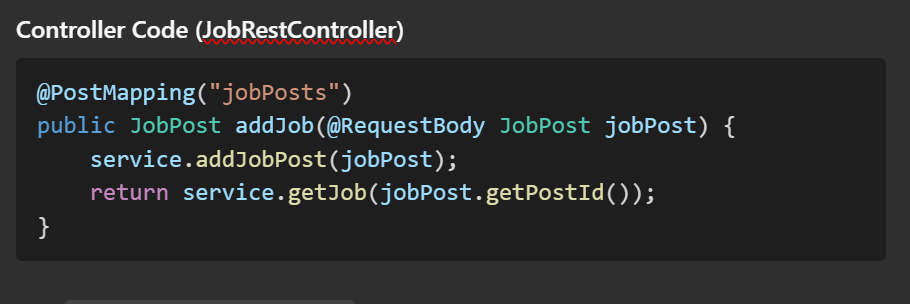
1. **Sending Data and Request Body**

When creating RESTful services, one of the common tasks is to receive data from a client, process it, and store it. This is often done using HTTP POST requests. In Spring Boot, this is achieved using the @PostMapping annotation combined with the @RequestBody parameter to extract the data sent in the request.

Example Overview

In the provided project, we have a REST API that allows users to add new job postings by sending data to the server using a POST request. The implementation of the POST method is illustrated through the following pieces of code:

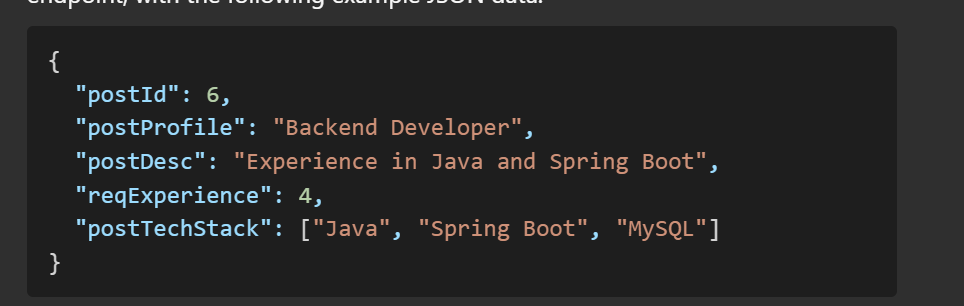
Controller Code (JobRestController)



* **@PostMapping("jobPosts")**: This annotation is used to handle HTTP POST requests made to the /jobPosts endpoint. It indicates that this method is responsible for handling incoming POST requests for adding a new job post.
* **@RequestBody JobPost jobPost**: This annotation is used to map the data coming in the request body to the JobPost object. This means that the client sends data (in JSON format) which is automatically converted into a JobPost instance by Spring Boot. It saves the need for manual parsing of JSON data.
* **Service Layer Interaction**: The method calls service.addJobPost(jobPost). This service method, in turn, calls the addJob() method of the JobRepo repository class to store the new job post in the collection.
* **Return Statement**: After adding the job post, the method returns the newly added job by calling service.getJob(jobPost.getPostId()). This is done to confirm that the new job post was successfully added and to provide a response back to the client.

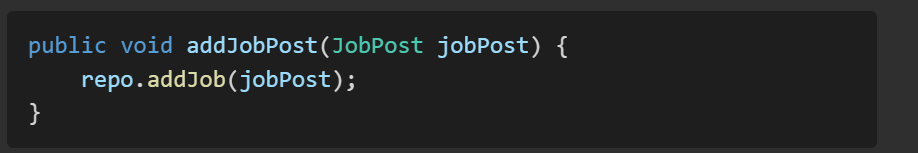
### Example JSON Data for POST Request

When the client wants to add a job post, they send the request to the /jobPosts endpoint, with the following example JSON data:



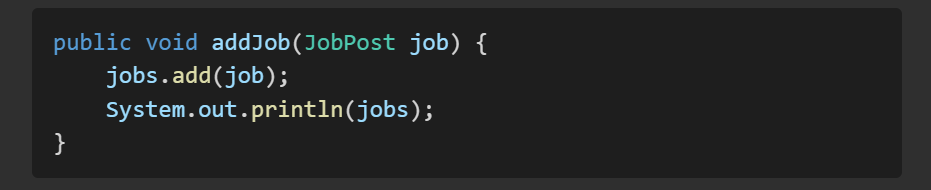
This JSON data represents the details of a new job post and is automatically deserialized by Spring Boot into the JobPost object when received by the endpoint.

**Service Layer (JobService)**



The addJobPost method in JobService calls the addJob() method of JobRepo to add the new job to the list of jobs.

**Repository Layer (JobRepo)**

****

This method adds the incoming job to the jobs list, effectively storing it in memory.

**Flow Explanation**

**Client Sends Data**: The client sends the JSON data representing a new job post to the /jobPosts endpoint using a POST request.

**Controller Receives Data**: The addJob() method in JobRestController handles this request and maps the incoming JSON to a JobPost object using @RequestBody.

**Service Layer:** The controller calls service.addJobPost(jobPost), which passes the new job post to the repository layer.

**Repository Layer**: The repository adds the new job post to the in-memory list.

**Return Response:** The controller returns the newly added job post to the client, providing confirmation.

**Summary**

The @PostMapping and @RequestBody annotations are fundamental when working with HTTP POST requests in Spring Boot. The @RequestBody annotation allows you to bind the incoming JSON data directly to a Java object, making it easy to work with client-provided data. In this example, the POST request adds a new job post to an in-memory list, and the newly added job post is then returned as a response to the client.

1. **Put and Delete mapping**

The PUT HTTP method is used to update an existing resource. It requires a resource identifier (usually part of the URL) and a representation of the updated resource in the request body. In Spring Boot, this is achieved with the @PutMapping annotation.

Code Example: Update a JobPost

In the following example, we add an endpoint to update an existing JobPost. The user must send the updated job details in the request body, and the job with the corresponding ID is updated.

Modified Code for Controller Layer:

****

**Explanation:**

**PutMapping("jobPost"):** Maps the HTTP PUT request to the method, handling requests sent to /jobPost.

**RequestBody JobPost jobPost**: Binds the request body to the jobPost parameter. This allows the updated job post details to be sent as JSON in the body of the request.

**service.updateJob(jobPost):** Calls the service layer to update the job post in the repository.

return service.getJob(jobPost.getPostId()): Returns the updated job post details after the update operation.



**Explanation of Repository Layer**:

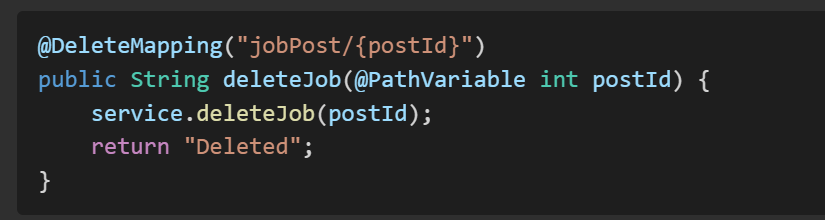
* The method iterates over the list of jobs to find the job post with the given ID and updates its properties with the new values provided in the request body.

**DELETE Mapping**

The **DELETE** HTTP method is used to delete a resource identified by a unique identifier. In Spring Boot, the @DeleteMapping annotation is used to map HTTP DELETE requests to a specific method in the controller.

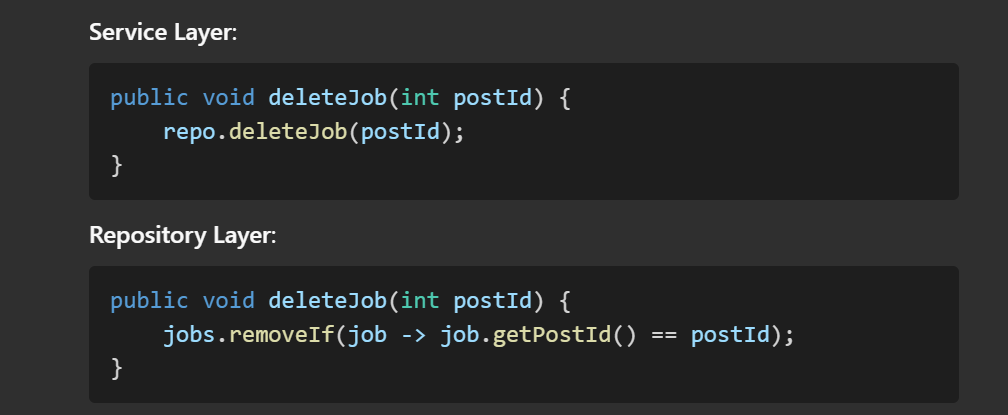
**Code Example: Delete a JobPost**

**Modified Code for Controller Layer**:



**Explanation**:

1. **@DeleteMapping("jobPost/{postId}")**: Maps the HTTP DELETE request to this method, handling requests to /jobPost/{postId} where {postId} is the ID of the job to be deleted.
2. **@PathVariable int postId**: Binds the value from the URL to the postId parameter in the method.
3. **service.deleteJob(postId)**: Calls the service layer to delete the job post from the repository.
4. **return "Deleted"**: Returns a simple message indicating the resource was successfully deleted.

****

**Explanation of Repository Layer:**

* The deleteJob method removes the job from the list if its postId matches the provided ID. This is achieved using the removeIf method, which is a convenient way to filter and remove elements from a list.

**Summary**

* **PUT Mapping (@PutMapping)** is used to update an existing resource. It takes the updated resource as the request body and finds the existing resource to update.
* **DELETE Mapping (@DeleteMapping)** is used to delete a resource based on the provided identifier.
* The Controller delegates the update and delete operations to the Service layer, which in turn calls the Repository layer to modify the data.
* By using RESTful principles, we can easily handle resource management operations like create, read, update, and delete using appropriate HTTP methods.

These annotations and practices make it easy to perform modifications on resources, maintain clear separation of concerns, and ensure each component in the application has a distinct responsibility.

**Flow Execution**

**1. GET Request Example (Fetch All Jobs)**

Endpoint: http://localhost:8080/jobPosts

**Step-by-Step Flow:**

1. **Hitting the Endpoint:**
   * You type http://localhost:8080/jobPosts in the browser or make a request using tools like Postman or curl.
   * This is a GET request.
2. **Request Reaches the Server:**
   * The server receives the request, and the embedded Tomcat server (part of Spring Boot) handles it.
   * Tomcat processes the request and hands it over to the Spring framework to determine how to handle it.
3. **DispatcherServlet:**
   * The DispatcherServlet (part of Spring) is responsible for routing incoming requests to the appropriate controller.
   * DispatcherServlet checks which controller method matches the endpoint /jobPosts.
4. **Routing to the Controller:**
   * The @GetMapping("jobPosts") in your JobRestController class tells Spring that this method should handle GET requests to /jobPosts.
   * Spring then calls the getAllJobs() method in the controller.



* Here, the getAllJobs() method in the controller layer is responsible for getting all job posts.
* The controller uses the service layer (service.returnAllJobPosts()) to handle the business logic.

**Service Layer**:

* The service layer (JobService) contains the actual business logic.
* It calls the returnAllJobPosts() method in the repository (repo.getAllJobs()).

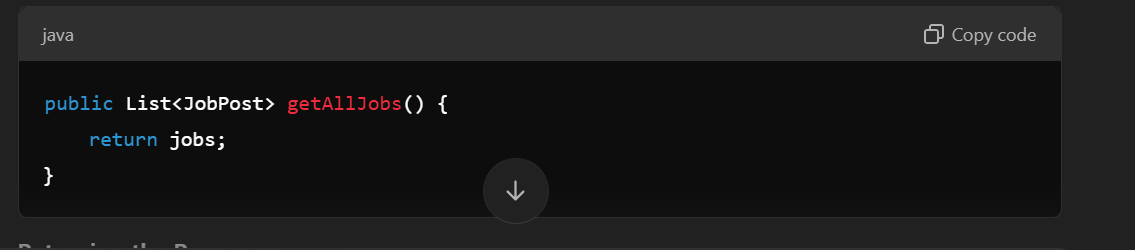
**Code in Service**:



**Repository Layer**:

* The repository layer (JobRepo) interacts with the data (in this case, an in-memory list).
* The getAllJobs() method returns the list of job posts.

**Code in Repository**:



**Returning the Response**:

* + The list of JobPost objects is returned from the repository to the service.
  + The service then returns it to the controller.
  + The controller finally returns the list to the client.
  + Spring automatically converts the list of JobPost objects into a JSON response, which is then sent to the client.

**POST Request Example (Add a Job)**

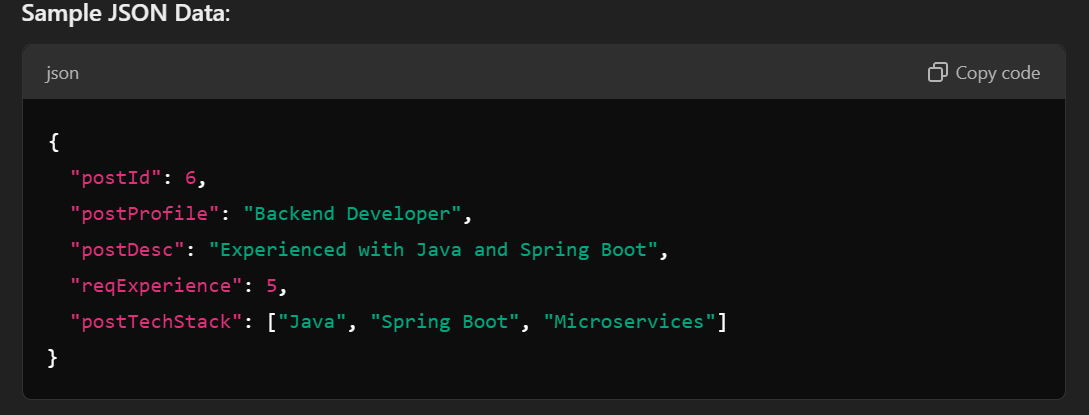
**Endpoint:**http://localhost:8080/jobPosts

**Data is Sent Through**: JSON body in Postman.

**Step-by-Step Flow**:

1. **Hitting the Endpoint**:
   * You make a POST request to http://localhost:8080/jobPosts through Postman with a JSON body containing the job details.

**Sample JSON Data**:



1. **Request Reaches the Server**:
   * The server (Tomcat) processes the POST request and hands it over to Spring's **DispatcherServlet**.
2. **Routing to the Controller**:
   * The @PostMapping("jobPosts") annotation in the controller indicates that this method should handle POST requests to /jobPosts.
   * DispatcherServlet calls the addJob() method.

**Code in Controller**:

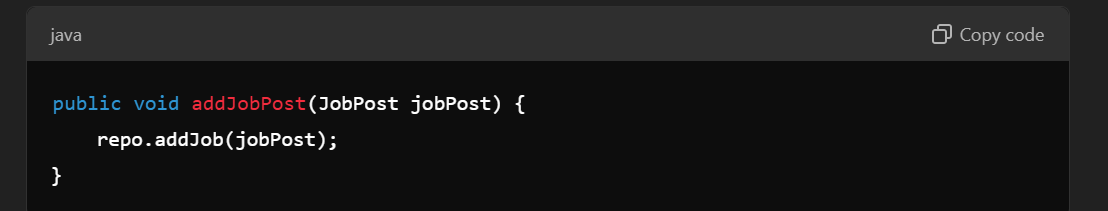


* @RequestBody JobPost jobPost tells Spring to bind the incoming JSON data to a JobPost object.
* The addJob() method receives the JobPost object and then calls the service layer to add the job post.

**Service Layer**:

* The addJobPost() method in the service layer is called.

**Code in Service**:

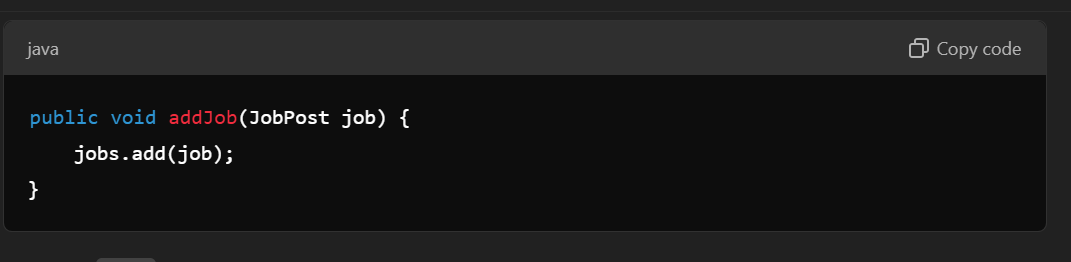


* The service passes the JobPost object to the repository layer to add it to the list.

**Repository Layer**:

* The addJob() method in the repository adds the JobPost object to the list.

**Code in Repository**:



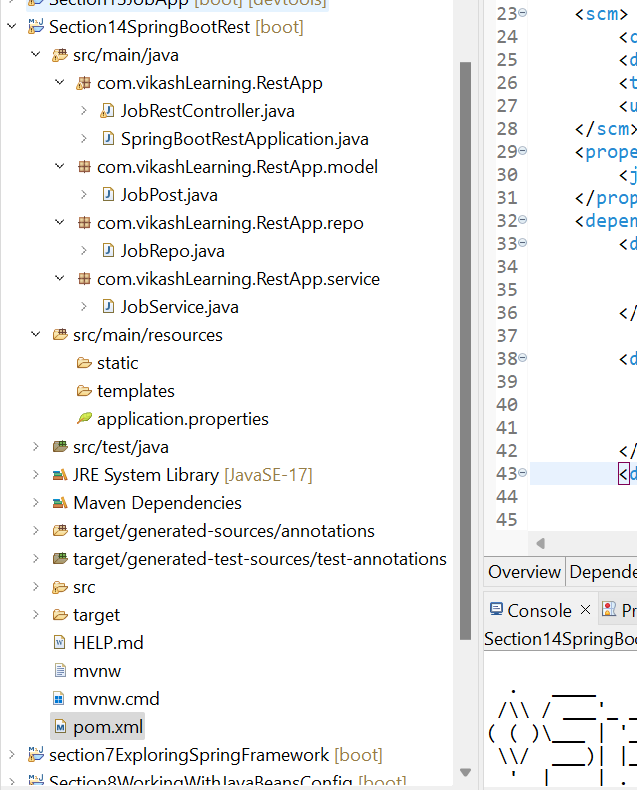
* The jobs list now has the new job post added.

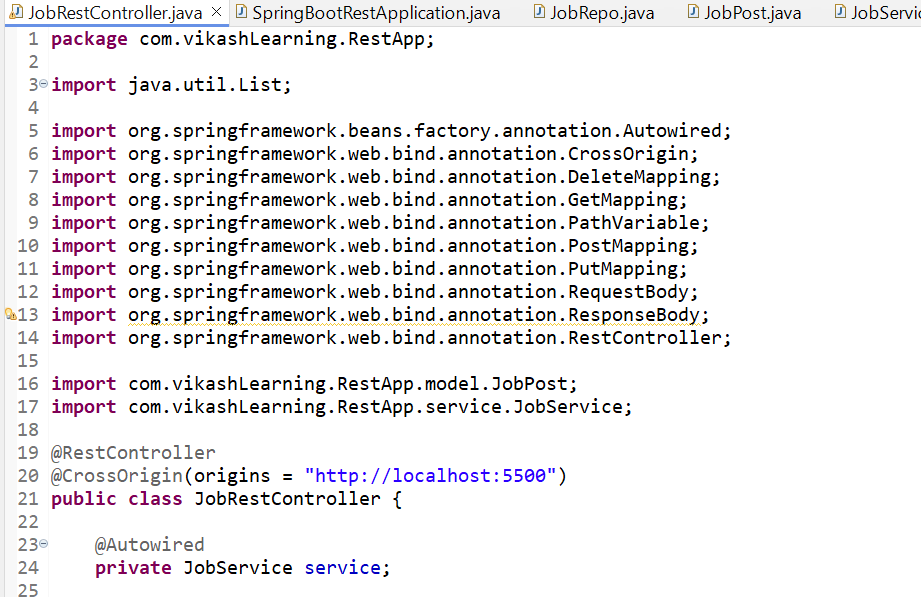
**Returning the Response:**

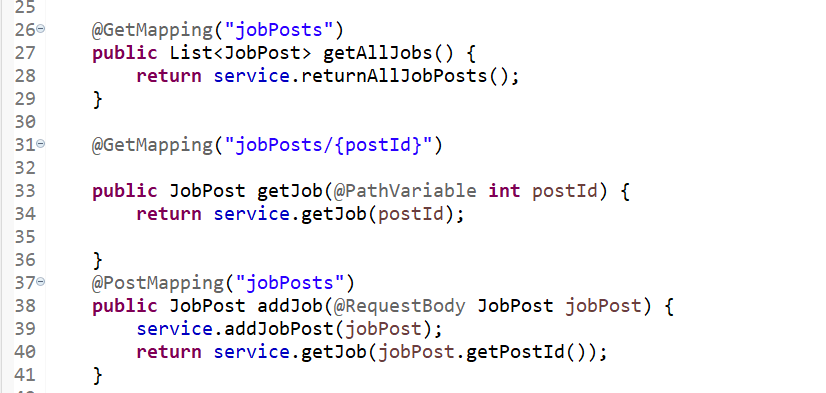
* The response is sent back from the repository to the service.
* The controller returns the newly added job post (service.getJob(jobPost.getPostId())).
* Spring automatically converts the JobPost object to JSON and sends it as a response.

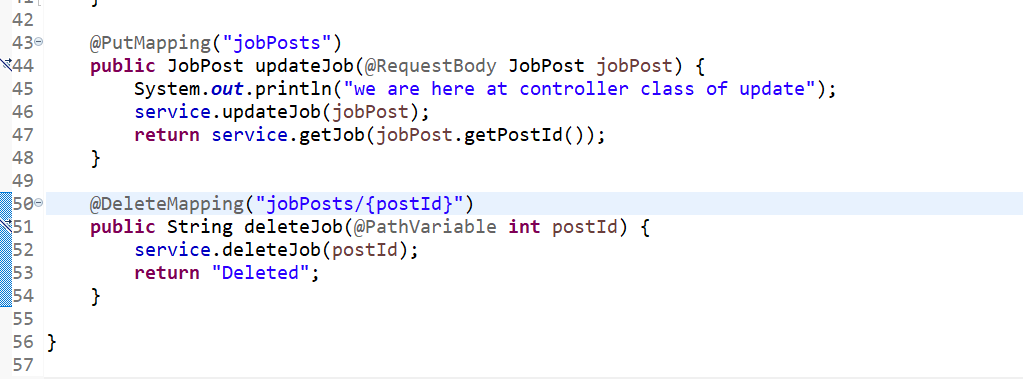
**Project Codes**

**Project Structure**

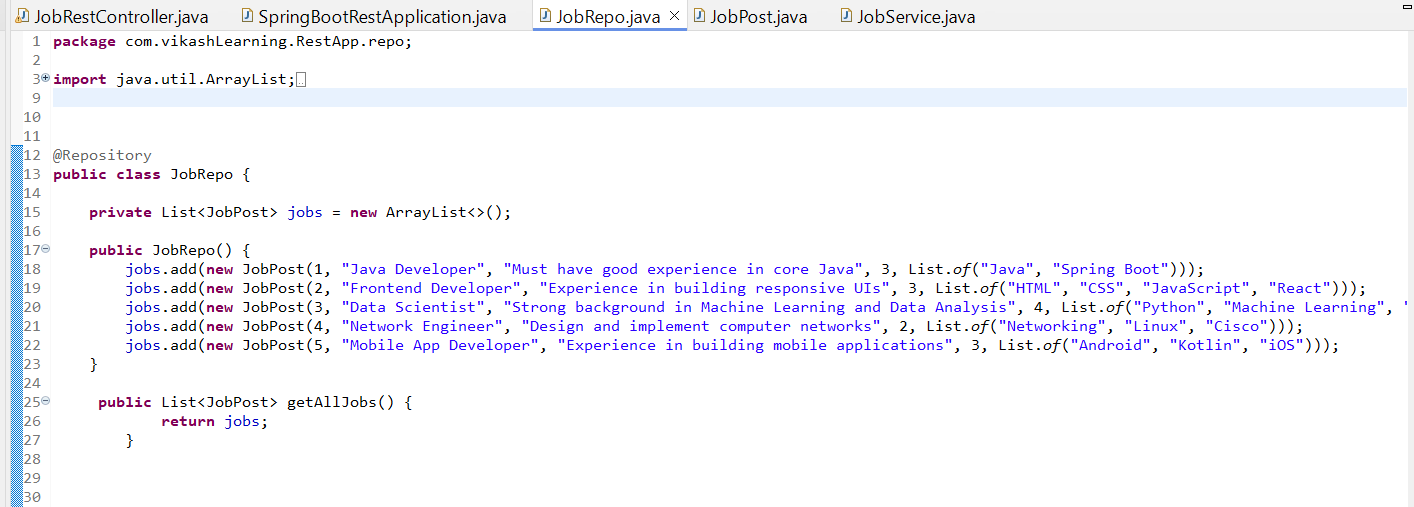
****

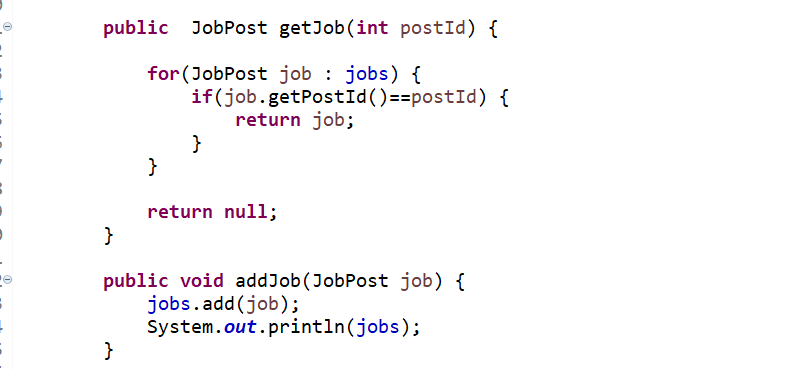
****

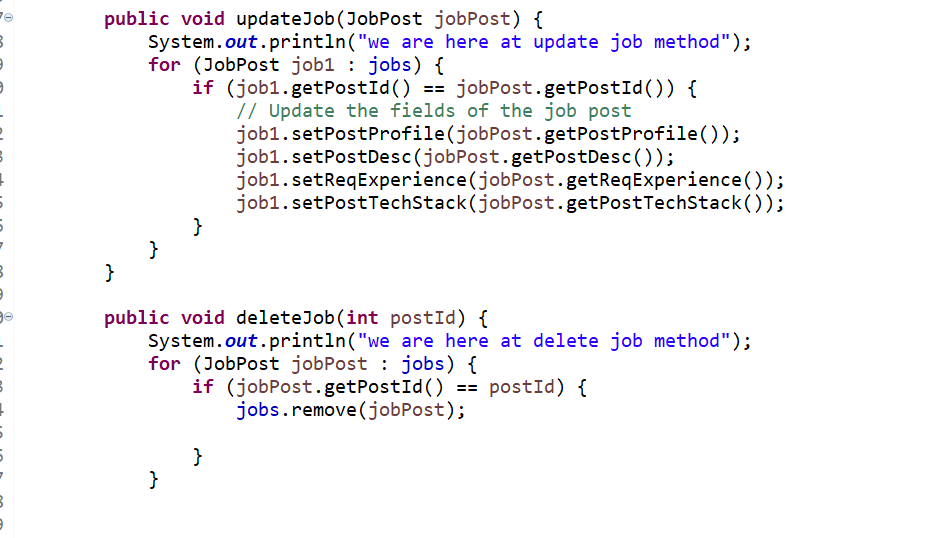
****

****

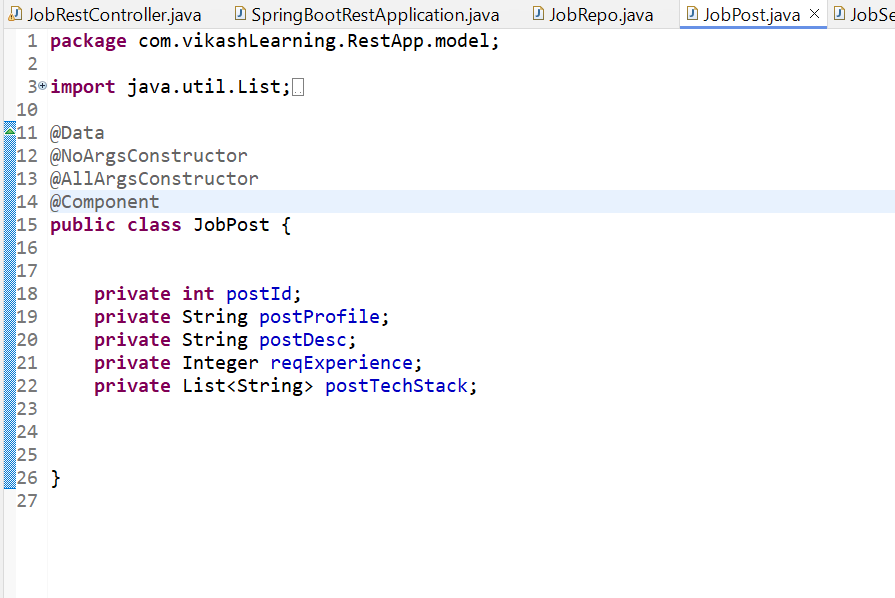
**JopRepo Class**

****

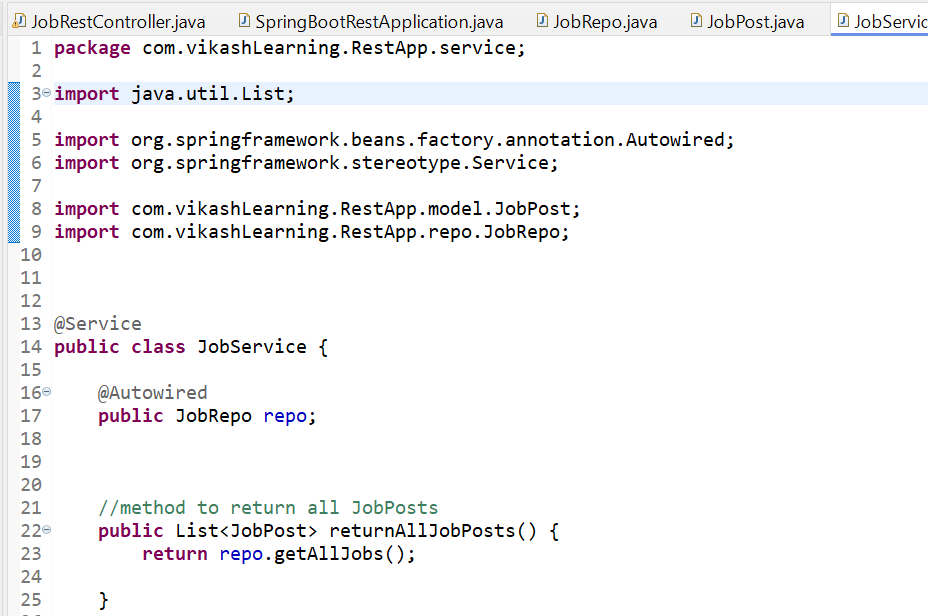
****

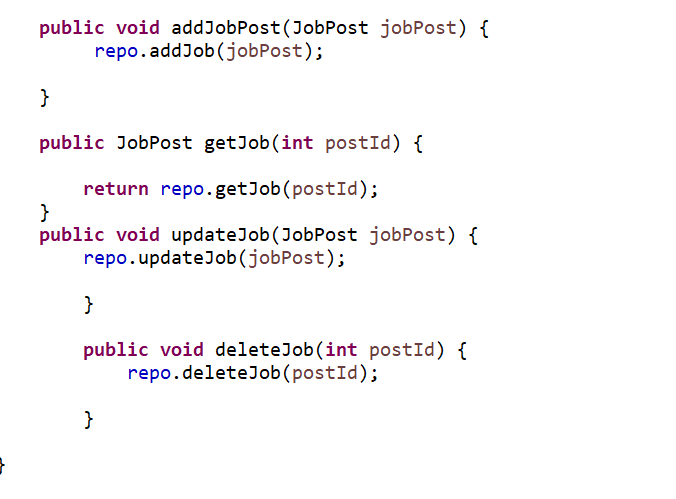
****

**Job Post class**

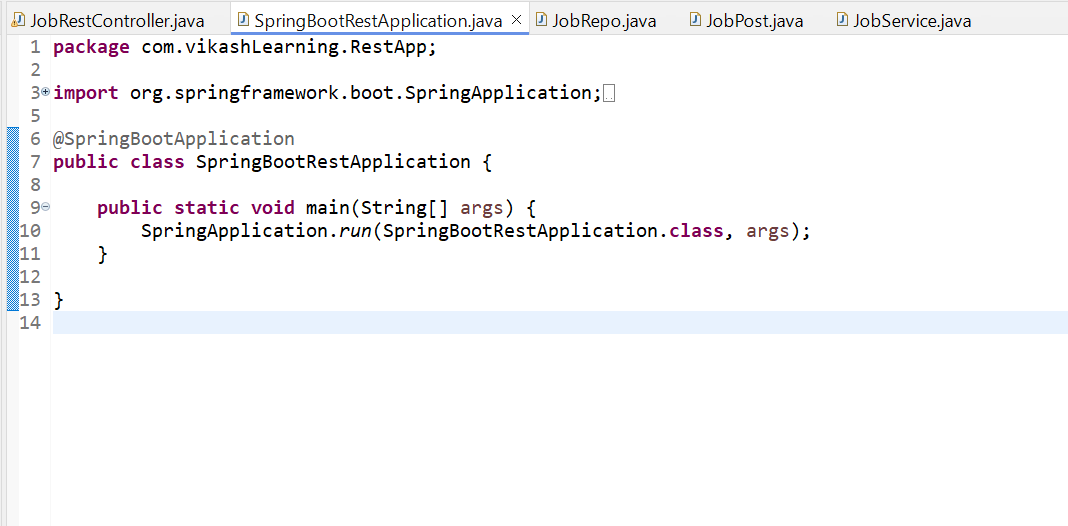
****

**Job Service classs**

****

****

**SpringBootApplication**

****

**POM>XML**

**<?xml version="1.0" encoding="UTF-8"?>**

**<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">**

**<modelVersion>4.0.0</modelVersion>**

**<parent>**

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

**<artifactId>spring-boot-starter-parent</artifactId>**

**<version>3.3.5</version>**

**<relativePath/> <!-- lookup parent from repository -->**

**</parent>**

**<groupId>com.vikashLearning</groupId>**

**<artifactId>Spring-boot-rest</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**<name>spring-boot-rest</name>**

**<description>Demo project for Spring Boot</description>**

**<url/>**

**<licenses>**

**<license/>**

**</licenses>**

**<developers>**

**<developer/>**

**</developers>**

**<scm>**

**<connection/>**

**<developerConnection/>**

**<tag/>**

**<url/>**

**</scm>**

**<properties>**

**<java.version>17</java.version>**

**</properties>**

**<dependencies>**

**<dependency>**

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

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

**</dependency>**

**<dependency>**

**<groupId>org.projectlombok</groupId>**

**<artifactId>lombok</artifactId>**

**<optional>true</optional>**

**</dependency>**

**<dependency>**

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

**<artifactId>spring-boot-starter-test</artifactId>**

**<scope>test</scope>**

**</dependency>**

**</dependencies>**

**<build>**

**<plugins>**

**<plugin>**

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

**<artifactId>spring-boot-maven-plugin</artifactId>**

**<configuration>**

**<excludes>**

**<exclude>**

**<groupId>org.projectlombok</groupId>**

**<artifactId>lombok</artifactId>**

**</exclude>**

**</excludes>**

**</configuration>**

**</plugin>**

**</plugins>**

**</build>**

**</project>**