Overview of Spring MVC & Architecture

**Spring MVC (Model-View-Controller)** is a powerful framework within the Spring ecosystem designed to simplify the development of web applications. It follows the MVC design pattern, which separates the application into three interconnected components: Model, View, and Controller. This separation promotes organized code, making it easier to manage and scale applications.

**Key Concepts of Spring MVC**

1. **Model**:
   * The Model represents the data and business logic of the application. It interacts with the database and holds the application's state. In a Spring MVC application, models are often implemented as JavaBeans or POJOs (Plain Old Java Objects).
   * Models may also include validation logic and services that perform business operations.
2. **View**:
   * The View is responsible for rendering the user interface and presenting data to the user. In Spring MVC, views can be implemented using various technologies, including JSP (JavaServer Pages), Thymeleaf, FreeMarker, and others.
   * The View is typically responsible for displaying the data returned by the Controller, allowing users to interact with the application.
3. **Controller**:
   * The Controller acts as an intermediary between the Model and the View. It processes user input, interacts with the Model to retrieve or update data, and returns the appropriate View.
   * Controllers are typically annotated with @Controller in Spring MVC and handle incoming requests using methods annotated with @RequestMapping, @GetMapping, @PostMapping, etc.

**Spring MVC Architecture**

The architecture of Spring MVC consists of several components that work together to handle web requests and responses effectively. Below is a high-level overview of the architecture:

1. **DispatcherServlet**:
   * The central component of Spring MVC is the DispatcherServlet, which acts as the front controller. It receives all incoming HTTP requests and delegates them to appropriate controllers for processing.
   * It handles request routing, view resolution, and response generation.
2. **Handler Mappings**:
   * The DispatcherServlet uses Handler Mappings to determine which Controller should handle a given request. It maps the request URL to the corresponding Controller method based on annotations and configuration.
3. **Controllers**:
   * Once the appropriate Controller is identified, the DispatcherServlet invokes the Controller method. The Controller processes the request, interacts with the Model, and prepares the data to be returned to the View.
4. **View Resolvers**:
   * After the Controller returns the Model data, the DispatcherServlet consults the View Resolvers to determine which View to render. View Resolvers map logical view names to actual View implementations (e.g., JSP files, Thymeleaf templates).
5. **Views**:
   * The View renders the data passed from the Controller and generates the final HTML response to be sent back to the client. The View can also handle user interactions, such as forms and buttons.
6. **Model**:
   * The Model holds the application's data and business logic. It may also include service classes that perform operations on the data, such as retrieving or persisting data to a database.

**Spring MVC Request Handling Flow**

Here’s a simplified flow of how Spring MVC handles a request:

1. **Client Request**: The user makes a request to a specific URL (e.g., http://localhost:8080/myapp/users).
2. **DispatcherServlet**: The request is intercepted by the DispatcherServlet.
3. **Handler Mapping**: The DispatcherServlet uses Handler Mappings to determine the appropriate Controller for the request.
4. **Controller Execution**: The DispatcherServlet invokes the corresponding Controller method, which processes the request and interacts with the Model.
5. **Model Preparation**: The Controller prepares the Model data to be displayed in the View.
6. **View Resolution**: The Controller returns a logical view name to the DispatcherServlet, which then uses View Resolvers to map it to an actual View implementation.
7. **View Rendering**: The View renders the HTML response using the Model data.
8. **Response to Client**: The final response is sent back to the client’s browser.

Controllers and @Controller Annotation

In Spring MVC, **Controllers** play a crucial role in processing incoming requests, interacting with the Model, and returning responses to the client. They act as an intermediary between the View and the Model, handling user inputs and orchestrating the flow of data in the application.

**What is a Controller?**

A Controller in Spring MVC is a class that handles HTTP requests from clients. Each method in the Controller corresponds to a specific request and contains the logic to process that request. Controllers are responsible for:

* Receiving user input via HTTP requests.
* Interacting with the Model to perform business logic and data retrieval.
* Returning a response, typically by specifying a View to be rendered.

**The @Controller Annotation**

The @Controller annotation is used to mark a class as a Spring MVC Controller. When a class is annotated with @Controller, it indicates to the Spring framework that this class will handle web requests.

**Key Features of @Controller:**

1. **Component Scanning**: Classes annotated with @Controller are automatically detected and registered as Spring beans during component scanning. This means that Spring will create an instance of the Controller and manage its lifecycle.
2. **Request Mapping**: Within a Controller class, you can define methods that handle specific requests using additional annotations like @RequestMapping, @GetMapping, @PostMapping, etc. These annotations map HTTP requests to specific methods.
3. **Returning Views**: Controllers can return a logical view name, which the DispatcherServlet uses to resolve the actual view (e.g., JSP, Thymeleaf) to render the response.

**Example of a Controller**

Here’s a simple example of a Controller in a Spring MVC application:

java

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

import org.springframework.web.bind.annotation.RequestParam;

@Controller

public class UserController {

@GetMapping("/users")

public String getUsers(@RequestParam(name = "name", required = false) String name, Model model) {

// Simulate retrieving user data

String greetingMessage = (name != null) ? "Hello, " + name + "!" : "Hello, Guest!";

// Add data to the model

model.addAttribute("message", greetingMessage);

// Return the logical view name

return "user"; // The view resolver will map this to /WEB-INF/views/user.jsp or equivalent

}

}

**Breakdown of the Example**

1. **Class Definition**:
   * The UserController class is annotated with @Controller, indicating that it is a Spring MVC Controller.
2. **Method Annotation**:
   * The getUsers method is annotated with @GetMapping("/users"), which means it will handle HTTP GET requests to the /users URL.
3. **Request Parameters**:
   * The @RequestParam annotation is used to extract query parameters from the URL. In this case, it looks for a parameter named name. If it is not present, it defaults to null.
4. **Model Object**:
   * The Model object is used to pass data to the View. Here, a greeting message is added to the model, which will be accessible in the View.
5. **Returning a View**:
   * The method returns a logical view name ("user"), which the View Resolver will map to the appropriate View implementation, such as a JSP or Thymeleaf template.

Annotations

Annotations in the Spring Framework are a key feature that allows developers to configure beans, define dependencies, and manage application behavior through metadata. They provide a more concise and readable way to configure the Spring application context compared to XML configuration.

**Common Annotations in Spring**

Here’s an overview of some of the most commonly used annotations in Spring, categorized by their purpose:

**1. Component Scanning and Stereotype Annotations**

* **@Component**:
  + Indicates that a class is a Spring-managed component (a bean). It can be used for any Spring-managed class.
* **@Service**:
  + A specialization of @Component used to indicate that a class is a service. This is typically used in the service layer to encapsulate business logic.
* **@Repository**:
  + A specialization of @Component used for data access logic. It includes exception translation and is typically used in the persistence layer.
* **@Controller**:
  + A specialization of @Component used to define a controller in Spring MVC. It handles HTTP requests and returns views.
* **@RestController**:
  + A convenience annotation that combines @Controller and @ResponseBody, making it easier to create RESTful web services. It automatically serializes responses to JSON or XML.

**2. Dependency Injection Annotations**

* **@Autowired**:
  + Marks a constructor, method, or field for automatic dependency injection by Spring. Spring will resolve the dependency from the application context.
* **@Inject**:
  + Part of the Java Dependency Injection (JSR-330) standard, it can be used as an alternative to @Autowired.
* **@Qualifier**:
  + Used in conjunction with @Autowired to specify which bean should be injected when multiple candidates are available.
* **@Value**:
  + Used to inject values into fields from property files or environment variables.

**3. Configuration Annotations**

* **@Configuration**:
  + Indicates that a class contains Spring bean definitions. Classes annotated with @Configuration can define methods annotated with @Bean, which return the beans to be managed by the Spring container.
* **@Bean**:
  + Indicates that a method produces a bean to be managed by the Spring container. The method’s return value is registered as a bean in the application context.
* **@PropertySource**:
  + Specifies the location of property files to be loaded by the Spring application context.

**4. Request Mapping Annotations (Spring MVC)**

* **@RequestMapping**:
  + Used to map HTTP requests to specific handler methods in a Controller. It can specify the request method, URL, and additional parameters.
* **@GetMapping**:
  + A shortcut for @RequestMapping(method = RequestMethod.GET) to handle HTTP GET requests.
* **@PostMapping**:
  + A shortcut for @RequestMapping(method = RequestMethod.POST) to handle HTTP POST requests.
* **@PutMapping**:
  + A shortcut for @RequestMapping(method = RequestMethod.PUT) to handle HTTP PUT requests.
* **@DeleteMapping**:
  + A shortcut for @RequestMapping(method = RequestMethod.DELETE) to handle HTTP DELETE requests.
* **@PatchMapping**:
  + A shortcut for @RequestMapping(method = RequestMethod.PATCH) to handle HTTP PATCH requests.

**5. Aspect-Oriented Programming Annotations**

* **@Aspect**:
  + Indicates that a class is an aspect, which is a module that defines cross-cutting concerns (e.g., logging, security).
* **@Before**:
  + Defines advice that runs before a join point (e.g., a method execution).
* **@After**:
  + Defines advice that runs after a join point.
* **@Around**:
  + Defines advice that runs both before and after a join point, allowing for custom behavior around method execution.

**6. Transactional Annotations**

* **@Transactional**:
  + Indicates that a method or class should be executed within a transaction. Spring manages the transaction boundaries for the annotated methods.

HTTP Method Annotations (@GetMapping, @PostMapping, etc.)

Spring MVC provides a set of specialized annotations for mapping HTTP methods to controller methods. These annotations make it easier to define the HTTP request handling logic in a Spring MVC application. Each HTTP method has its own annotation, which helps clarify the intention of the method and improves code readability.

Here’s a breakdown of the common HTTP method annotations in Spring MVC:

**1. @GetMapping**

* **Purpose**: Used to handle HTTP GET requests, which are typically used for retrieving data from the server.
* **Usage**: This annotation simplifies the mapping of GET requests to controller methods.

**Example**:

java

@GetMapping("/users")

public List<User> getUsers() {

return userService.findAllUsers();

}

* **Description**: The above method will handle GET requests made to the /users URL and return a list of users.

**2. @PostMapping**

* **Purpose**: Used to handle HTTP POST requests, which are commonly used for creating new resources or submitting data to the server.
* **Usage**: This annotation is a shortcut for @RequestMapping(method = RequestMethod.POST).

**Example**:

java

@PostMapping("/users")

public ResponseEntity<User> createUser(@RequestBody User user) {

User savedUser = userService.saveUser(user);

return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);

}

* **Description**: The above method handles POST requests to /users and creates a new user based on the data provided in the request body.

**3. @PutMapping**

* **Purpose**: Used to handle HTTP PUT requests, which are generally used for updating existing resources.
* **Usage**: A shortcut for @RequestMapping(method = RequestMethod.PUT).

**Example**:

java

@PutMapping("/users/{id}")

public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User user) {

User updatedUser = userService.updateUser(id, user);

return ResponseEntity.ok(updatedUser);

}

* **Description**: The above method handles PUT requests to /users/{id} and updates the user with the specified ID.

**4. @DeleteMapping**

* **Purpose**: Used to handle HTTP DELETE requests, which are typically used to delete resources from the server.
* **Usage**: A shortcut for @RequestMapping(method = RequestMethod.DELETE).

**Example**:

java

@DeleteMapping("/users/{id}")

public ResponseEntity<Void> deleteUser(@PathVariable Long id) {

userService.deleteUser(id);

return ResponseEntity.noContent().build();

}

* **Description**: The above method handles DELETE requests to /users/{id} and deletes the user with the specified ID.

**5. @PatchMapping**

* **Purpose**: Used to handle HTTP PATCH requests, which are used for applying partial modifications to a resource.
* **Usage**: A shortcut for @RequestMapping(method = RequestMethod.PATCH).

**Example**:

java

@PatchMapping("/users/{id}")

public ResponseEntity<User> updateUserPartial(@PathVariable Long id, @RequestBody Map<String, Object> updates) {

User updatedUser = userService.updateUserPartially(id, updates);

return ResponseEntity.ok(updatedUser);

}

* **Description**: The above method handles PATCH requests to /users/{id} and updates specific fields of the user with the given ID.

Request Parameters and Path Variables

In Spring MVC, handling incoming data from HTTP requests is crucial for building interactive web applications. Two common ways to extract data from requests are through **request parameters** and **path variables**. Here’s a detailed look at both concepts:

**1. Request Parameters**

**Request Parameters** are key-value pairs sent as part of the query string in the URL or in the body of a POST request. They are typically used for filtering, searching, or sending data from the client to the server.

**How to Access Request Parameters**

You can access request parameters in a controller method using the @RequestParam annotation.

**Example**:

java

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

import org.springframework.web.bind.annotation.RequestParam;

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

@RestController

public class UserController {

@GetMapping("/users")

public String getUserById(@RequestParam(name = "id") Long userId) {

return "User ID: " + userId;

}

}

**Breakdown of the Example:**

* **@RequestParam**:
  + The @RequestParam annotation is used to bind a request parameter to a method parameter. In this case, name = "id" indicates that the request should contain a parameter named id.
* **Accessing Data**:
  + When a client makes a request like /users?id=123, the userId variable in the method will be set to 123.

**Optional Parameters**

You can make request parameters optional by setting the required attribute to false.

**Example**:

java

@GetMapping("/users")

public String getUserByName(@RequestParam(name = "name", required = false) String name) {

return name != null ? "User name: " + name : "No name provided.";

}

**2. Path Variables**

**Path Variables** are used to capture values from the URI path. They are particularly useful for accessing resource identifiers in a RESTful API.

**How to Access Path Variables**

You can access path variables in a controller method using the @PathVariable annotation.

**Example**:

java

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

import org.springframework.web.bind.annotation.PathVariable;

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

@RestController

public class UserController {

@GetMapping("/users/{id}")

public String getUserById(@PathVariable Long id) {

return "User ID: " + id;

}

}

**Breakdown of the Example:**

* **@PathVariable**:
  + The @PathVariable annotation binds a URI template variable to a method parameter. In this case, {id} in the URL is matched to the id parameter in the method.
* **Accessing Data**:
  + When a client makes a request like /users/123, the id variable in the method will be set to 123.

**Summary of Differences**

| **Feature** | **Request Parameters** | **Path Variables** |
| --- | --- | --- |
| Definition | Key-value pairs in the query string or body | Variables in the URI path |
| Access Method | @RequestParam | @PathVariable |
| URL Example | /users?id=123 | /users/123 |
| Usage Context | Suitable for optional filtering and parameters | Suitable for identifying resources |
| Required by Default | Yes (unless specified as optional) | Always required unless specified as optional |

Request Body and @RequestBody Annotation

In Spring MVC, the **request body** is used to send data from the client to the server as part of an HTTP request, typically in formats like JSON or XML. This is especially common in RESTful APIs, where clients often send data to create or update resources. The @RequestBody annotation in Spring simplifies the process of mapping this incoming data to a Java object.

**What is @RequestBody?**

The @RequestBody annotation is used to bind the HTTP request body to a method parameter in a controller. It allows you to automatically deserialize the incoming data into a Java object, making it easy to work with complex data structures.

**How to Use @RequestBody**

1. **Define a Model Class**: First, you need a Java class that represents the structure of the data you expect to receive in the request body.

**Example Model Class**:

java

public class User {

private Long id;

private String name;

private String email;

// Getters and Setters

}

1. **Controller Method**: Use the @RequestBody annotation in the controller method to receive the data.

**Example Controller Method**:

java

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

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

@RestController

public class UserController {

@PostMapping("/users")

public ResponseEntity<User> createUser(@RequestBody User user) {

// Process the user object (e.g., save to database)

return ResponseEntity.status(HttpStatus.CREATED).body(user);

}

}

**Breakdown of the Example:**

* **Request Mapping**: The method is mapped to handle POST requests to the /users endpoint.
* **Using @RequestBody**: The @RequestBody annotation indicates that the method parameter user should be populated with the data from the request body.
* **Automatic Deserialization**: When a client sends a JSON request body like the following:

json

{

"id": 1,

"name": "John Doe",

"email": "john@example.com"

}

Spring will automatically convert this JSON into a User object and populate its fields.

**Content-Type**

The client must specify the content type of the request body in the HTTP headers. For JSON, it should be:

bash

Content-Type: application/json

**Error Handling**

If the request body cannot be deserialized into the specified object (for example, due to missing fields or incorrect types), Spring will throw a HttpMessageNotReadableException, resulting in a 400 Bad Request response.

HTTP Status Code & Exception Handling with @ExceptionHandler

HTTP status codes are standard response codes used by web servers to indicate the outcome of a client's request. Properly handling these codes is essential for building robust and user-friendly web applications. In Spring MVC, you can handle exceptions gracefully using the @ExceptionHandler annotation, which allows you to specify how to respond to specific exceptions thrown during the processing of requests.

**HTTP Status Codes Overview**

HTTP status codes are divided into five categories:

* **1xx**: Informational responses (e.g., 100 Continue)
* **2xx**: Success responses (e.g., 200 OK, 201 Created)
* **3xx**: Redirection messages (e.g., 301 Moved Permanently, 302 Found)
* **4xx**: Client error responses (e.g., 400 Bad Request, 404 Not Found, 401 Unauthorized)
* **5xx**: Server error responses (e.g., 500 Internal Server Error, 503 Service Unavailable)

**Commonly Used HTTP Status Codes**

* **200 OK**: The request was successful.
* **201 Created**: The request was successful, and a resource was created.
* **400 Bad Request**: The server could not understand the request due to invalid syntax.
* **401 Unauthorized**: The request requires user authentication.
* **403 Forbidden**: The server understood the request but refuses to authorize it.
* **404 Not Found**: The server cannot find the requested resource.
* **500 Internal Server Error**: The server encountered a situation it doesn't know how to handle.

**Exception Handling with @ExceptionHandler**

In Spring MVC, exceptions can occur during request processing, and you can handle them elegantly using the @ExceptionHandler annotation. This annotation allows you to define a method that will be called when a specific exception is thrown.

**How to Use @ExceptionHandler**

1. **Define the Exception Class**: If you have a custom exception, create an exception class.

**Example**:

java

public class UserNotFoundException extends RuntimeException {

public UserNotFoundException(String message) {

super(message);

}

}

1. **Controller with Exception Handling**: Use @ExceptionHandler in a controller to handle specific exceptions.

**Example**:

java

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.ExceptionHandler;

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

import org.springframework.web.bind.annotation.PathVariable;

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

@RestController

public class UserController {

@GetMapping("/users/{id}")

public User getUserById(@PathVariable Long id) {

User user = userService.findUserById(id);

if (user == null) {

throw new UserNotFoundException("User not found with ID: " + id);

}

return user;

}

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<String> handleUserNotFoundException(UserNotFoundException ex) {

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

}

}

**Breakdown of the Example:**

* **Controller Method**: The getUserById method fetches a user based on the provided ID. If the user is not found, it throws a UserNotFoundException.
* **Exception Handler**: The handleUserNotFoundException method is annotated with @ExceptionHandler(UserNotFoundException.class), which means it will be invoked whenever a UserNotFoundException is thrown in this controller.
* **ResponseEntity**: The handler method returns a ResponseEntity with a 404 NOT FOUND status and the exception message in the response body.

**Handling Multiple Exceptions**

You can handle multiple exceptions in a single method by specifying them as an array in the @ExceptionHandler annotation.

**Example**:

java

@ExceptionHandler({ UserNotFoundException.class, SomeOtherException.class })

public ResponseEntity<String> handleUserExceptions(Exception ex) {

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

}

**Global Exception Handling**

For more complex applications, you may want to handle exceptions globally across all controllers. You can do this by using @ControllerAdvice.

**Example**:

java

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 {

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<String> handleUserNotFoundException(UserNotFoundException ex) {

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

}

// Handle other exceptions...

}

RESTful API Development with @RestController Annotation

The @RestController annotation in Spring MVC is a specialized version of the @Controller annotation that is used to create RESTful web services. It combines the functionality of @Controller and @ResponseBody, allowing you to develop APIs that return JSON or XML responses without needing to annotate each method with @ResponseBody.

**Overview of RESTful APIs**

**REST** (Representational State Transfer) is an architectural style for designing networked applications. RESTful APIs follow specific principles:

* **Stateless**: Each request from the client contains all the information the server needs to fulfill that request.
* **Resource-Based**: APIs are designed around resources (e.g., users, products) that can be accessed through URIs.
* **Standard HTTP Methods**: RESTful APIs typically use standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources.

**Creating a RESTful API Using @RestController**

**1. Define the Model Class**

First, define a model class that represents the resource.

**Example Model Class**:

java

public class User {

private Long id;

private String name;

private String email;

// Getters and Setters

}

**2. Create the REST Controller**

Use the @RestController annotation to create a controller class that handles API requests.

**Example Controller**:

java

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

import java.util.ArrayList;

import java.util.List;

@RestController

@RequestMapping("/api/users")

public class UserController {

private List<User> userList = new ArrayList<>();

// Create a new user

@PostMapping

public ResponseEntity<User> createUser(@RequestBody User user) {

userList.add(user);

return ResponseEntity.status(HttpStatus.CREATED).body(user);

}

// Get all users

@GetMapping

public ResponseEntity<List<User>> getAllUsers() {

return ResponseEntity.ok(userList);

}

// Get a user by ID

@GetMapping("/{id}")

public ResponseEntity<User> getUserById(@PathVariable Long id) {

return userList.stream()

.filter(user -> user.getId().equals(id))

.findFirst()

.map(ResponseEntity::ok)

.orElse(ResponseEntity.status(HttpStatus.NOT\_FOUND).build());

}

// Update a user by ID

@PutMapping("/{id}")

public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User user) {

for (int i = 0; i < userList.size(); i++) {

User existingUser = userList.get(i);

if (existingUser.getId().equals(id)) {

userList.set(i, user);

return ResponseEntity.ok(user);

}

}

return ResponseEntity.status(HttpStatus.NOT\_FOUND).build();

}

// Delete a user by ID

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteUser(@PathVariable Long id) {

userList.removeIf(user -> user.getId().equals(id));

return ResponseEntity.noContent().build();

}

}

**Breakdown of the Example:**

* **@RestController**: This annotation indicates that the class is a controller where every method returns a domain object instead of a view. It automatically serializes the return value to JSON or XML based on the request's Accept header.
* **@RequestMapping("/api/users")**: This annotation specifies the base URI for all the endpoints in this controller.
* **HTTP Method Annotations**:
  + **@PostMapping**: Handles the creation of a new user.
  + **@GetMapping**:
    - One method retrieves all users.
    - Another method retrieves a user by ID using the path variable.
  + **@PutMapping**: Updates an existing user.
  + **@DeleteMapping**: Deletes a user by ID.

**Example Requests**

Here are some example requests to interact with this API:

1. **Create a User**:

http

POST /api/users

Content-Type: application/json

{

"id": 1,

"name": "John Doe",

"email": "john@example.com"

}

1. **Get All Users**:

http

GET /api/users

1. **Get User by ID**:

http

GET /api/users/1

1. **Update User**:

http

PUT /api/users/1

Content-Type: application/json

{

"id": 1,

"name": "John Doe",

"email": "john.doe@example.com"

}

1. **Delete User**:

http

DELETE /api/users/1

**Benefits of Using @RestController**

* **Simplified Response Handling**: Automatically converts Java objects to JSON/XML without needing to annotate every method with @ResponseBody.
* **Clearer API Structure**: Makes it easier to build RESTful APIs with a clean separation of resources.
* **Built-in Error Handling**: Can easily integrate with exception handling mechanisms to provide meaningful error responses.