### Creating REST APIs with Spring Boot (`@RestController`, `@RequestMapping`)

Spring Boot makes it easy to create RESTful web services by providing built-in support for developing REST APIs. It simplifies the process of handling HTTP requests and responses, working with JSON data, and more. Two key annotations for building REST APIs in Spring Boot are `@RestController` and `@RequestMapping`.

---

### Key Annotations for Building REST APIs

#### 1. \*\*`@RestController`\*\*

The `@RestController` annotation is used to create RESTful web services in Spring Boot. It’s a specialization of the `@Controller` annotation, which simplifies creating RESTful services by combining `@Controller` and `@ResponseBody`. Every method in a class annotated with `@RestController` will return JSON, XML, or another custom response format by default.

- \*\*`@Controller`\*\* is for web applications where HTML views are returned.

- \*\*`@RestController`\*\* is for REST APIs where JSON (or other formats) is returned as a response body.

#### Example:

```java

@RestController

public class MyRestController {

@GetMapping("/hello")

public String sayHello() {

return "Hello, World!";

}

}

```

In this example, Spring Boot will return `"Hello, World!"` as the response when a `GET` request is made to the `/hello` endpoint.

---

#### 2. \*\*`@RequestMapping`\*\*

`@RequestMapping` is a versatile annotation that maps HTTP requests to specific handler methods or classes in your Spring Boot application. It can be used at both the \*\*class level\*\* and \*\*method level\*\*. It supports various HTTP request methods like `GET`, `POST`, `PUT`, `DELETE`, and more.

- At the \*\*class level\*\*, it defines the base URL for all methods in the controller.

- At the \*\*method level\*\*, it defines the specific path for the handler method, along with the HTTP method (e.g., GET, POST, etc.).

#### Example:

```java

@RestController

@RequestMapping("/api")

public class MyRestController {

@RequestMapping(value = "/greet", method = RequestMethod.GET)

public String greet() {

return "Greetings from Spring Boot!";

}

}

```

In this example, when a `GET` request is made to `/api/greet`, the method `greet()` will be invoked, and the response will be `"Greetings from Spring Boot!"`.

---

### Simplifying HTTP Request Mappings

In addition to `@RequestMapping`, you can use method-specific annotations for different HTTP request methods:

- \*\*`@GetMapping`\*\*: Shortcut for `@RequestMapping(method = RequestMethod.GET)`

- \*\*`@PostMapping`\*\*: Shortcut for `@RequestMapping(method = RequestMethod.POST)`

- \*\*`@PutMapping`\*\*: Shortcut for `@RequestMapping(method = RequestMethod.PUT)`

- \*\*`@DeleteMapping`\*\*: Shortcut for `@RequestMapping(method = RequestMethod.DELETE)`

- \*\*`@PatchMapping`\*\*: Shortcut for `@RequestMapping(method = RequestMethod.PATCH)`

#### Example:

```java

@RestController

@RequestMapping("/api")

public class MyRestController {

@GetMapping("/users")

public List<String> getUsers() {

return List.of("User1", "User2", "User3");

}

@PostMapping("/users")

public String createUser(@RequestBody String user) {

// Logic to save the user

return "User created: " + user;

}

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

public String updateUser(@PathVariable String id, @RequestBody String user) {

// Logic to update the user

return "User updated: " + id;

}

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

public String deleteUser(@PathVariable String id) {

// Logic to delete the user

return "User deleted: " + id;

}

}

```

Here’s how these mappings work:

- \*\*`GET /api/users`\*\* returns a list of users.

- \*\*`POST /api/users`\*\* creates a new user by reading the request body.

- \*\*`PUT /api/users/{id}`\*\* updates an existing user using the path variable `id`.

- \*\*`DELETE /api/users/{id}`\*\* deletes a user based on the `id`.

---

### Handling Path Variables and Request Parameters

#### 1. \*\*`@PathVariable`\*\*

This annotation is used to extract values from the URL itself. It binds a method parameter to a URI template variable.

#### Example:

```java

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

public String getUserById(@PathVariable String id) {

return "User ID: " + id;

}

```

If a request is made to `/users/123`, the response will be `"User ID: 123"`.

#### 2. \*\*`@RequestParam`\*\*

This is used to extract query parameters from the request URL.

#### Example:

```java

@GetMapping("/search")

public String searchUser(@RequestParam String name) {

return "Searching for user: " + name;

}

```

For a request like `/search?name=John`, the response will be `"Searching for user: John"`.

---

### Working with JSON Request and Response

Spring Boot, by default, uses \*\*Jackson\*\* for JSON serialization and deserialization. You can map incoming JSON data to Java objects using the `@RequestBody` annotation and automatically return JSON responses from Java objects.

#### Example:

```java

@RestController

@RequestMapping("/api")

public class UserController {

@PostMapping("/users")

public User createUser(@RequestBody User user) {

// Save user logic

return user; // Returns the user object as JSON

}

}

class User {

private String name;

private String email;

// Getters and setters

}

```

- The `@RequestBody` annotation is used to map the incoming JSON request to the `User` object.

- The `User` object is returned as JSON in the response.

For example, a `POST` request with this body:

```json

{

"name": "John Doe",

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

}

```

would result in the `User` object being created, and the response would be the same JSON.

---

### Returning Different HTTP Status Codes

You can customize the HTTP status codes returned by your APIs using the `ResponseEntity` class, which allows you to set both the body and the status of the response.

#### Example:

```java

@PostMapping("/users")

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

if (user.getName() == null) {

return ResponseEntity.status(HttpStatus.BAD\_REQUEST).body("Invalid user data");

}

// Save user logic

return ResponseEntity.status(HttpStatus.CREATED).body("User created successfully");

}

```

In this example:

- If the `User` object is missing required data (like `name`), it returns a `400 Bad Request` with an error message.

- If the user is created successfully, it returns a `201 Created` status with a success message.

---

### Full Example: Simple REST API

Here’s an example of a basic RESTful service with multiple endpoints:

```java

@RestController

@RequestMapping("/api/users")

public class UserController {

private Map<String, User> users = new HashMap<>();

@GetMapping

public List<User> getAllUsers() {

return new ArrayList<>(users.values());

}

@GetMapping("/{id}")

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

User user = users.get(id);

if (user == null) {

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

}

return ResponseEntity.ok(user);

}

@PostMapping

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

users.put(user.getId(), user);

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

}

@PutMapping("/{id}")

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

if (!users.containsKey(id)) {

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

}

users.put(id, user);

return ResponseEntity.ok(user);

}

@DeleteMapping("/{id}")

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

users.remove(id);

return ResponseEntity.noContent().build();

}

}

class User {

private String id;

private String name;

private String email;

// Getters and setters

}

```

### Key Endpoints:

1. `GET /api/users` - Retrieves all users.

2. `GET /api/users/{id}` - Retrieves a specific user by ID.

3. `POST /api/users` - Creates a new user.

4. `PUT /api/users/{id}` - Updates an existing user by ID.

5. `DELETE /api/users/{id}` - Deletes a user by ID.

---

### Conclusion

Building REST APIs with Spring Boot is simplified using annotations like `@RestController` and `@RequestMapping`. These annotations, combined with JSON support via Jackson, make it easy to develop RESTful services quickly. With features like path variables, query parameters, and response entities, you can build flexible, scalable APIs with ease.