Chapter 18: Building RESTful APIs Using Java (Spring Boot / Java EE)

Introduction

In today's web-based software systems, RESTful APIs (Application Programming Interfaces) serve as the backbone for enabling communication between client and server applications. REST (Representational State Transfer) is an architectural style that uses HTTP methods for creating scalable, stateless, and platform-independent services.

Java provides powerful frameworks for developing RESTful services, primarily through **Spring Boot** and **Java EE (Jakarta EE)**. In this chapter, we will explore how to build RESTful APIs using both these technologies. You will learn how to design endpoints, manage requests/responses, and perform CRUD operations on resources.

18.1 Overview of REST Architecture

18.1.1 What is REST?

REST stands for Representational State Transfer. It is a web standards-based architecture that uses HTTP protocol to access and manipulate web resources.

Key Concepts of REST:

- **Stateless:** No client context is stored on the server between requests.
- **Client-Server:** Clear separation between client and server roles.
- **Cacheable:** Responses must define themselves as cacheable or not.
- **Uniform Interface:** Use of standard HTTP methods (GET, POST, PUT, DELETE).
- **Resource-based:** Every object is a resource and is accessible via URI.

18.2 HTTP Methods in REST

HTTP Method	Description	Used for
GET	Retrieve a resource	Read operation
POST	Create a new resource	Create operation

HTTP Method	Description	Used for
PUT	Update a resource	Update operation
DELETE	Remove a resource	Delete operation

18.3 RESTful API with Spring Boot

18.3.1 Introduction to Spring Boot

Spring Boot simplifies the development of production-ready Spring applications. It comes with embedded servers, auto-configuration, and starter dependencies.

18.3.2 Creating a REST Controller

Step 1: Add Dependencies (`pom.xml`)

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

Step 2: Create Entity Class

```
javaCopy codepublic class Employee {
    private int id;
    private String name;
    private String department;
    // Getters and setters
}
```

Step 3: Create a Controller Class

```
javaCopy code@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    private List<Employee> employeeList = new ArrayList<>();

    @GetMapping
    public List<Employee> getAllEmployees() {
        return employeeList;
    }

    @PostMapping
    public Employee addEmployee(@RequestBody Employee employee) {
        employeeList.add(employee);
```

```
        return employee;
    }

    @PutMapping("/{id}")
    public Employee updateEmployee(@PathVariable int id, @RequestBody Employe
e updatedEmployee) {
        for (Employee emp : employeeList) {
            if (emp.getId() == id) {
                emp.setName(updatedEmployee.getName());
                emp.setDepartment(updatedEmployee.getDepartment());
                return emp;
            }
        }
        return null;
    }

    @DeleteMapping("/{id}")
    public String deleteEmployee(@PathVariable int id) {
        employeeList.removeIf(emp -> emp.getId() == id);
        return "Employee deleted successfully.";
    }
}
```

18.4 RESTful API with Java EE (Jakarta EE)

18.4.1 Using JAX-RS (Java API for RESTful Web Services)

JAX-RS is the standard API for creating RESTful services in Java EE.

18.4.2 Dependencies (Using Maven)

```
xmlCopy code<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.1</version>
</dependency>
```

18.4.3 Creating a REST Resource

```
javaCopy code@Path("/employees")
public class EmployeeResource {

    private static List<Employee> employees = new ArrayList<>();

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Employee> getEmployees() {
        return employees;
    }
```

```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Employee addEmployee(Employee employee) {
        employees.add(employee);
        return employee;
    }

    @PUT
    @Path("/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Employee updateEmployee(@PathParam("id") int id, Employee employee
) {
        for (Employee e : employees) {
            if (e.getId() == id) {
                e.setName(employee.getName());
                e.setDepartment(employee.getDepartment());
                return e;
            }
        }
        return null;
    }

    @DELETE
    @Path("/{id}")
    public String deleteEmployee(@PathParam("id") int id) {
        employees.removeIf(e -> e.getId() == id);
        return "Employee deleted.";
    }
}
```

18.5 REST API Design Best Practices

- **Use Nouns in URIs:** `/users`, `/orders/123`
- **Use HTTP Status Codes Properly:**
    - `200 OK` – Success
    - `201 Created` – Resource created
    - `204 No Content` – Successfully deleted
    - `400 Bad Request` – Invalid data
    - `404 Not Found` – Resource doesn't exist

- `500 Internal Server Error` – Server failed

- **Use Pagination for Large Data Sets**
- **Include Versioning:** `/api/v1/products`

18.6 Tools for Testing REST APIs

- **Postman** – GUI for testing REST services.
- **curl** – Command-line tool.
- **Swagger/OpenAPI** – API documentation and testing.

18.7 Exception Handling in Spring Boot

```
javaCopy code@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(value = Exception.class)
    public ResponseEntity<String> handleException(Exception e) {
        return new ResponseEntity<>("Error occurred: " + e.getMessage(), Http
Status.INTERNAL_SERVER_ERROR);
    }
}
```

18.8 Deploying REST APIs

- **Spring Boot:** Can be packaged as JAR and run using `java -jar`.
- **Java EE:** Deploy WAR files to a servlet container like Tomcat, WildFly, or GlassFish.

18.9 Security in REST APIs

- **Basic Authentication**
- **Token-based Authentication (JWT)**
- **OAuth 2.0**
- **HTTPS for secure communication**

18.10 Comparison: Spring Boot vs Java EE

Feature	Spring Boot	Java EE (Jakarta EE)
Setup	Simple (auto-configured)	Manual (XML/web.xml config)
Performance	High, lightweight	Slightly heavier
Community	Large, modern	Legacy with enterprise support
Microservice Ready	Yes	Requires customization

Summary

In this chapter, we explored how to build RESTful APIs in Java using both Spring Boot and Java EE. We learned the fundamentals of REST architecture, the use of HTTP methods, and the creation of CRUD-based services. While Spring Boot provides rapid development and microservice readiness, Java EE (now Jakarta EE) offers a more traditional, enterprise-grade solution. Tools like Postman, Swagger, and security mechanisms like JWT and OAuth are crucial for testing and securing APIs.

Understanding and implementing RESTful APIs is an essential skill for Java developers in modern full-stack or backend development.
