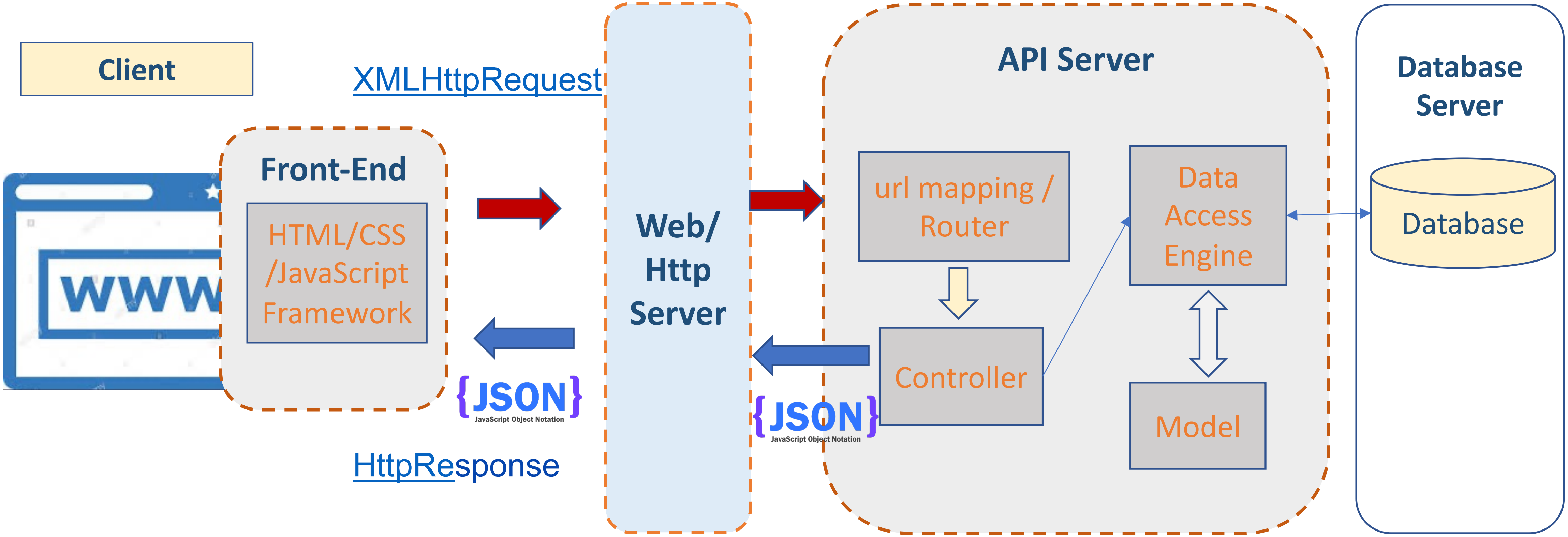
spring
boot

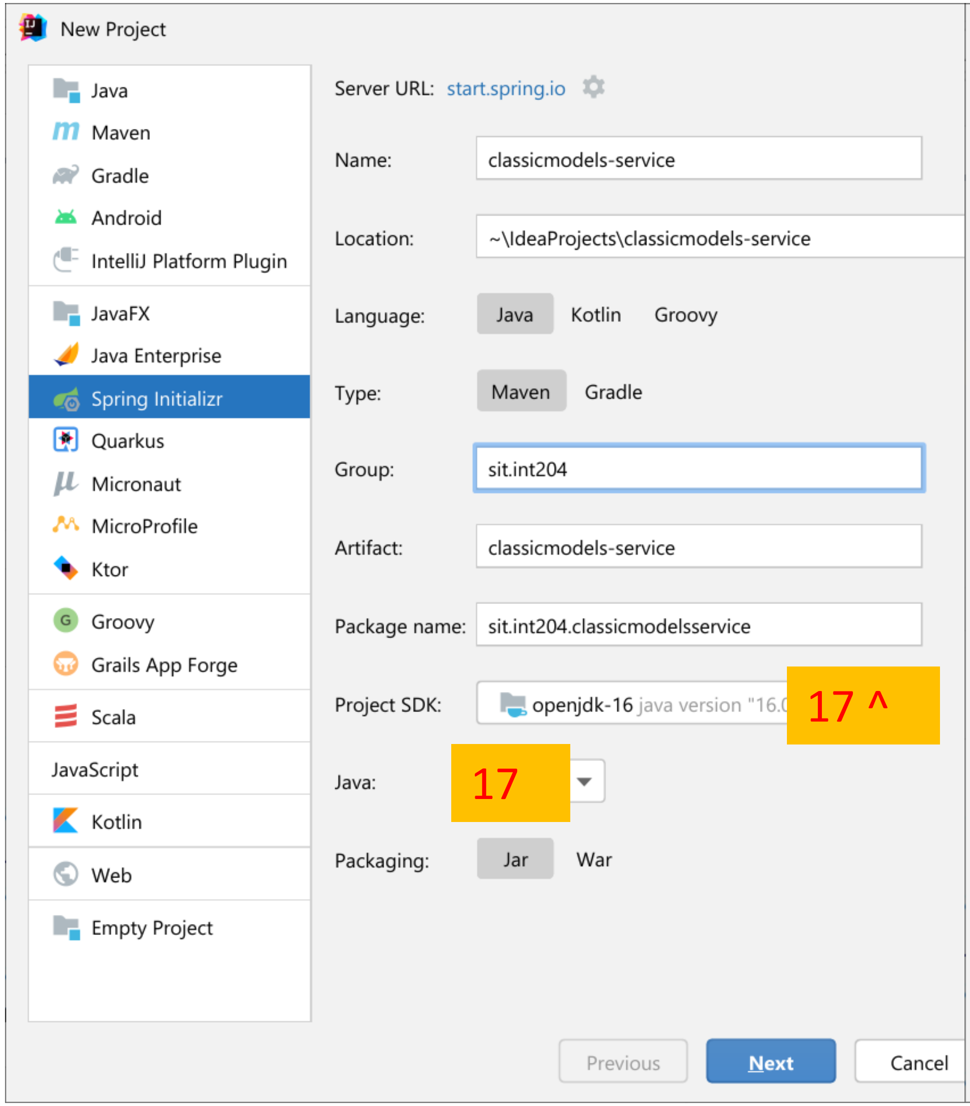
What is RESTful API ?

Learning by Doing

MVC Web Application Architectures (spa)



Step 1: Initializing a Spring Boot Project



Server URL: start.spring.io

Name: classicmodels-service

Location: ~\IdeaProjects\classicmodels-service

Language: [✓ Java] Kotlin Groovy

Type: [✓ Maven] Gradle

Group: sit.int204

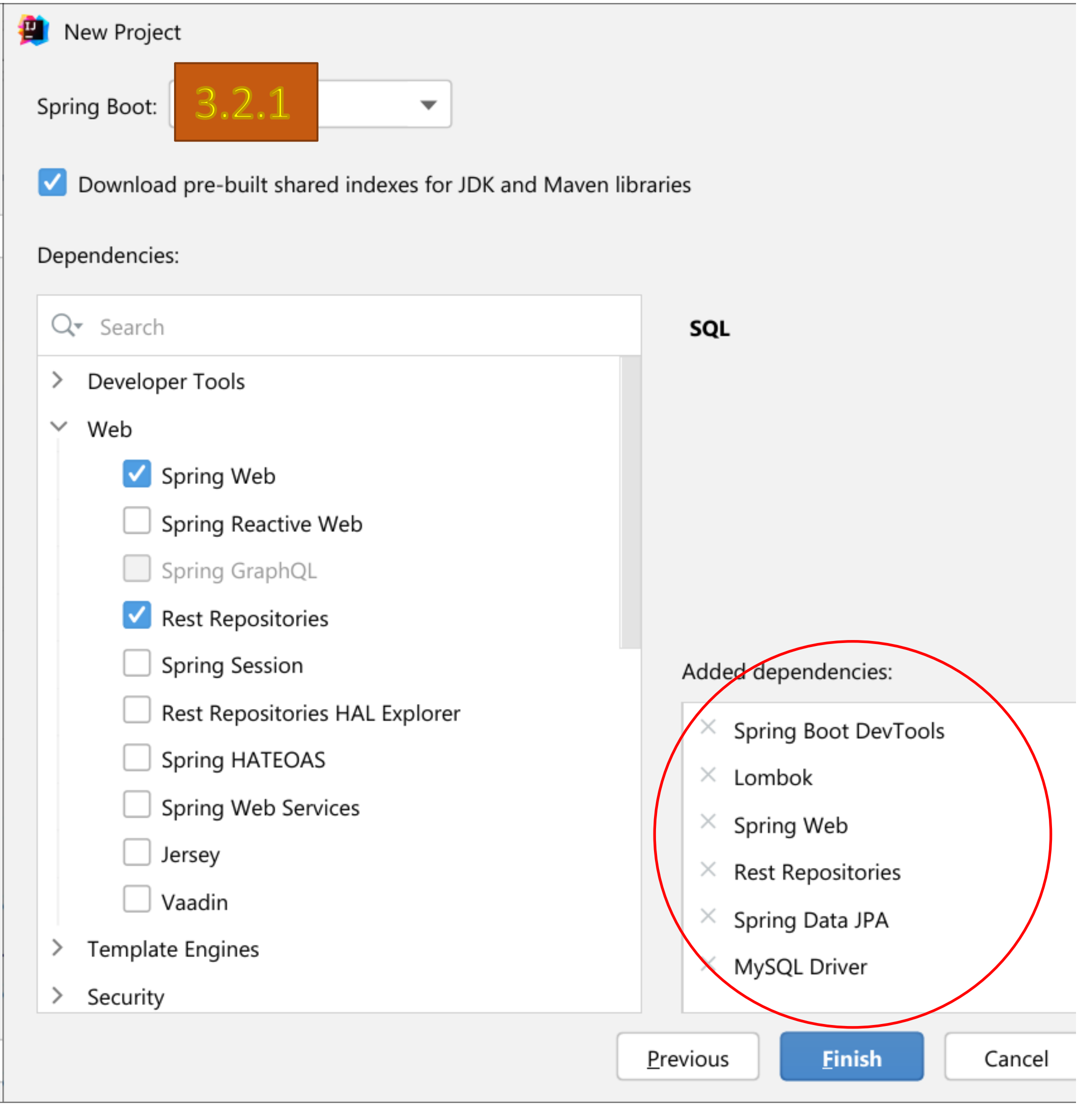
Artifact: classicmodels-service

Package name: sit.int204.classicmodelsservice

Project SDK: openjdk-16 java version "16.0 17 ^

Java: 17

Packaging: [✓ Jar] War



Spring Boot: 3.2.1

☑ Download pre-built shared indexes for JDK and Maven libraries

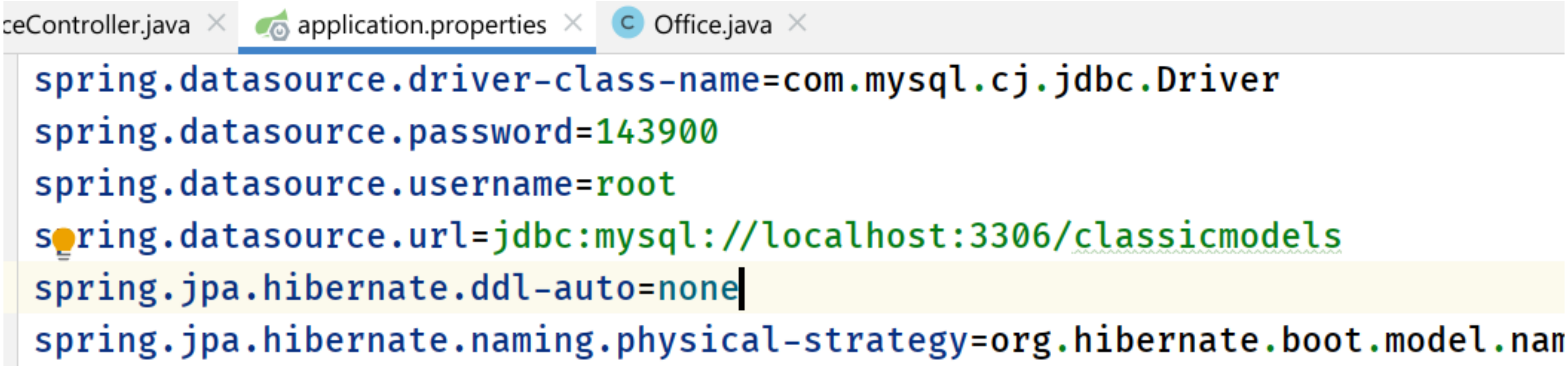
Dependencies:

- ☑ Spring Web
- □ Spring Reactive Web
- □ Spring GraphQL
- ☑ Rest Repositories
- □ Spring Session
- □ Rest Repositories HAL Explorer
- □ Spring HATEOAS
- □ Spring Web Services
- □ Jersey
- □ Vaadin

Added dependencies:

- Spring Boot DevTools
- Lombok
- Spring Web
- Rest Repositories
- Spring Data JPA
- MySQL Driver

Step 2: Connecting Spring Boot to the Database



```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.password=143900
spring.datasource.username=root
spring.datasource.url=jdbc:mysql://localhost:3306/classicmodels
spring.jpa.hibernate.ddl-auto=none
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.nam
```

```
org.hibernate.boot.model.naming.PhysicalNamingStrategy
```

ถ้าไม่กำหนด spring.jpa.hibernate.naming.physical-strategy

จะใช้ convention ดังนี้

การระบุ `@Column(name)` ใน **enitity clase** ต้องพิมพ์เป็นตัวเล็กหมด หรือถ้าระบุ คอลัมน์เป็น **camel case** ชื่อฟิลด์ ใน ตารางต้องแยกคำด้วย ขีดล่าง **(_)**

Step 3: Creating an Office Model (1)

```java
@Getter  @Setter
@Entity @Table(name = "offices")
public class Office {
    @Id
    @Column(name = "officeCode", nullable = false, length = 10)
    private String officeCode;
    @Column(name = "city", nullable = false, length = 50)
    private String city;
    @Column(name = "phone", nullable = false, length = 50)
    private String phone;
    @Column(name = "addressLine1", nullable = false, length = 50)
    private String addressLine1;
```

Step 3: Creating an Office Model (2)

```
    @Column(name = "addressLine2", length = 50)
    private String addressLine2;
    @Column(name = "state", length = 50)
    private String state;
    @Column(name = "country", nullable = false, length = 50)
    private String country;
    @Column(name = "postalCode", nullable = false, length = 15)
    private String postalCode;
    @Column(name = "territory", nullable = false, length = 10)
    private String territory;

    @JsonIgnore
    @OneToMany(mappedBy = "office")
    private Set<Employee> employees = new LinkedHashSet<>();
```

Step 3: Creating an Employee Model (1)

```java
@Getter @Setter
@Entity @Table(name = "employees")
public class Employee {
    @Id
    @Column(name = "employeeNumber", nullable = false)
    private Integer id;

    @JsonIgnore
    @ManyToOne
    @JoinColumn(name = "office")
    private Office office;

    @Column(name = "lastName", nullable = false, length = 50)
```

Step 3: Creating an Employee Model (2)

```
private String lastName;
    @Column(name = "firstName", nullable = false, length = 50)
    private String firstName;
    @Column(name = "extension", nullable = false, length = 10)
    private String extension;
    @Column(name = "email", nullable = false, length = 100)
    private String email;

    @JsonIgnore
    @ManyToOne
    @JoinColumn(name = "reportsTo")
    private Employee employees;

    @Column(name = "jobTitle", nullable = false, length = 50)
    private String jobTitle;
```

Step 4: Creating Repository Interface

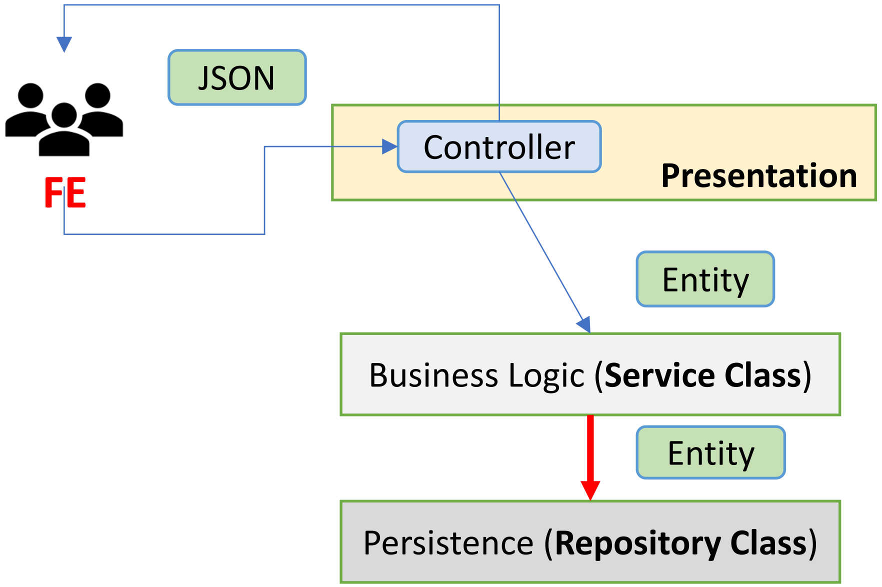
```
public interface OfficeRepository extends JpaRepository<Office, String> {

}
```

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer>
{

}
```

```
public interface CustomerRepository extends JpaRepository<Customer, Integer>
{

}
```

Step 5: Creating Service Class

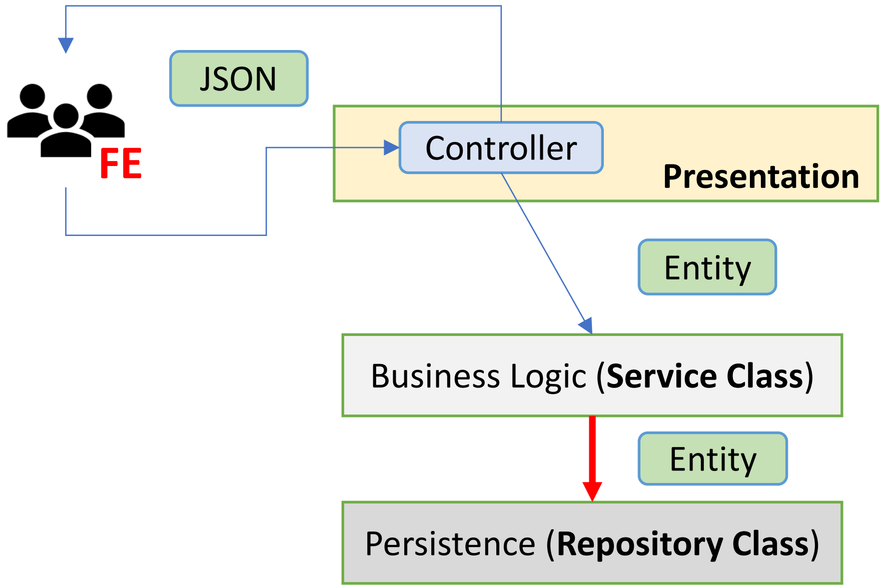
```
@Service

public class OfficeService {
    @Autowired
    private OfficeRepository repository;

    public List<Office> getAllOffice() {
        return repository.findAll();
    }

    public Office getOffice(String officeCode) {
        return repository.findById(officeCode).orElseThrow(
                () -> new HttpClientErrorException(HttpStatus.NOT_FOUND,
                        "Office Id " + officeCode + " DOES NOT EXIST !!!") {
                }
        );
    }


  @Transactional

  public Office createNewOffice(Office office) {
        return repository.save(office);
    }
```



JSON
FE
Controller
Presentation
Entity
Business Logic (Service Class)
Entity
Persistence (Repository Class)

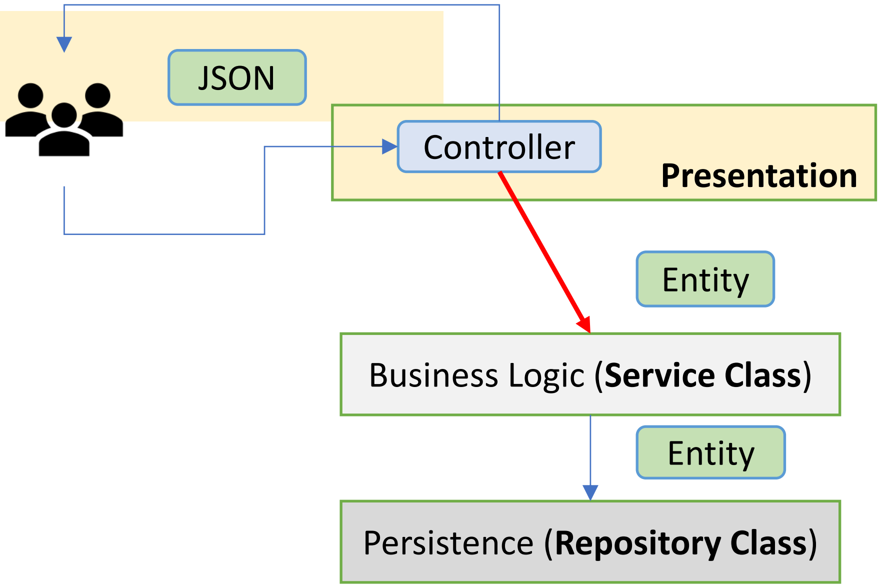
Step 5: Creating Service Class (cont.)

```
@Transactional
public void removeOffice(String officeCode) {
    Office office = repository.findById(officeCode).orElseThrow(
        () -> new HttpClientErrorException(HttpStatus.NOT_FOUND, "Office Id " + officeCode + " DOES NOT EXIST !!!")
    );
    repository.delete(office);
 }

 @Transactional
 public Office updateOffice(String officeCode, Office office) {
    if(office.getOfficeCode()!=null && !office.getOfficeCode().trim().isEmpty()) {
      if (!office.getOfficeCode().equals(officeCode)) {
        throw new HttpClientErrorException(HttpStatus.BAD_REQUEST,
                "Conflict Office code  !!! (" + officeCode +
                " vs " + office.getOfficeCode() + ”)");
      }
    }
    Office existingOffice = repository.findById(officeCode).orElseThrow(
        () -> new HttpClientErrorException(HttpStatus.NOT_FOUND,
                "Office Id " + officeCode + " DOES NOT EXIST !!!"));
    return repository.save(office);
 }
}
```


JSON
FE
Controller
Presentation
Entity
Business Logic (Service Class)
Entity
Persistence (Repository Class)

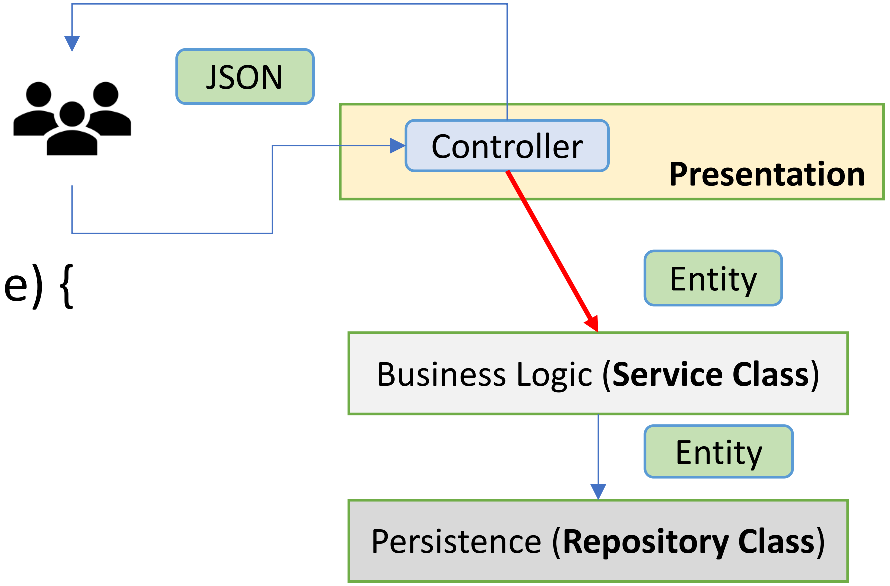
Step 6: Creating Controller

```
@RestController
@RequestMapping("/api/offices")
public class OfficeController {
   @Autowired
   private OfficeService service;

   @GetMapping("")
   public List<Office> getAllOffices() {
      return service.getAllOffice();
   }

   @GetMapping("/{officeCode}")
   public Office getOfficeById(@PathVariable String officeCode) {
      return service.getOffice(officeCode);
   }


   @PostMapping("")
   public Office addNewOffice(@RequestBody Office office) {
       return service.createNewOffice(office);
  }
```

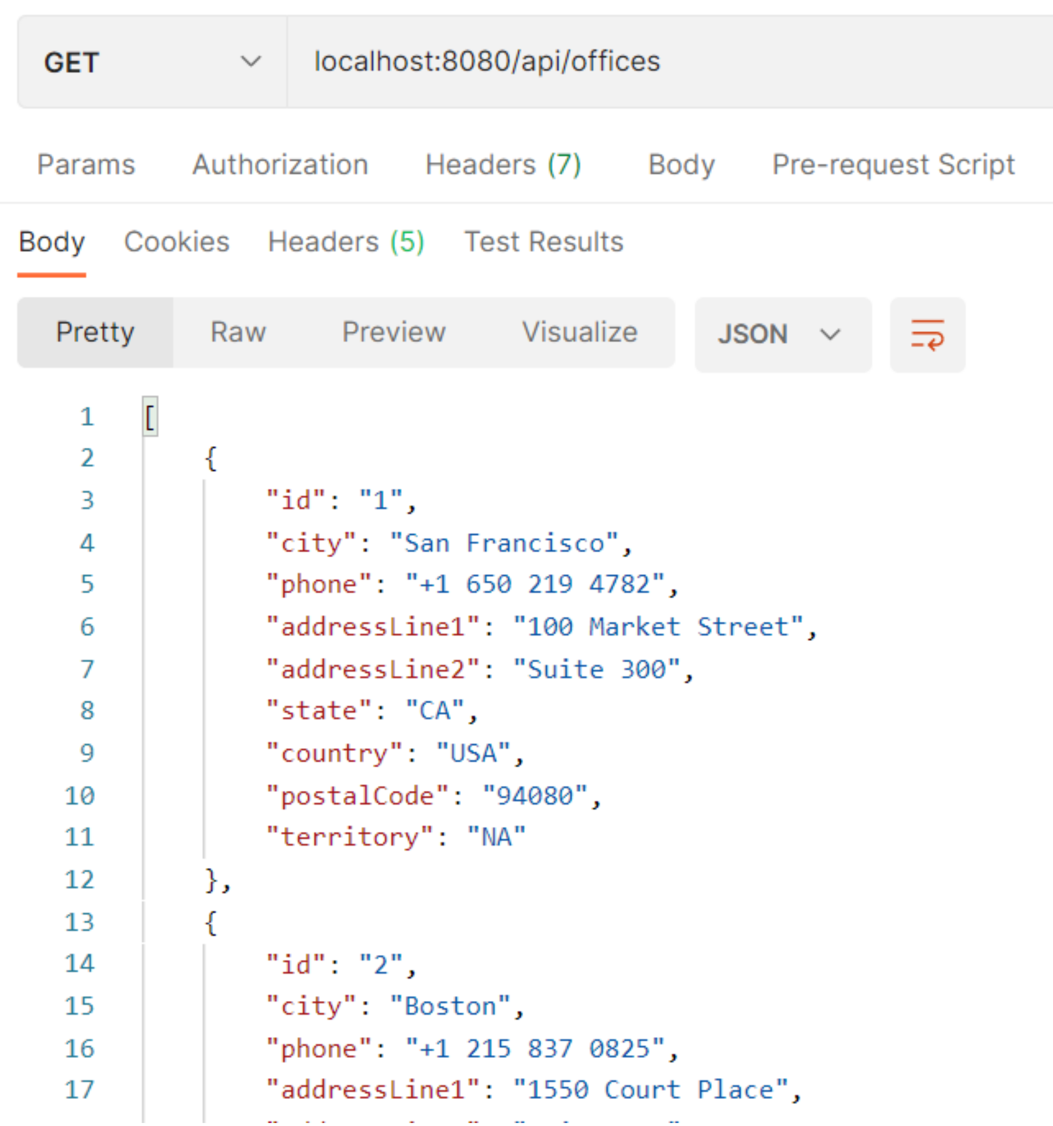

JSON
Controller
Presentation
Entity
Business Logic (**Service Class**)
Entity
Persistence (**Repository Class**)

Step 6: Creating Controller (cont.)

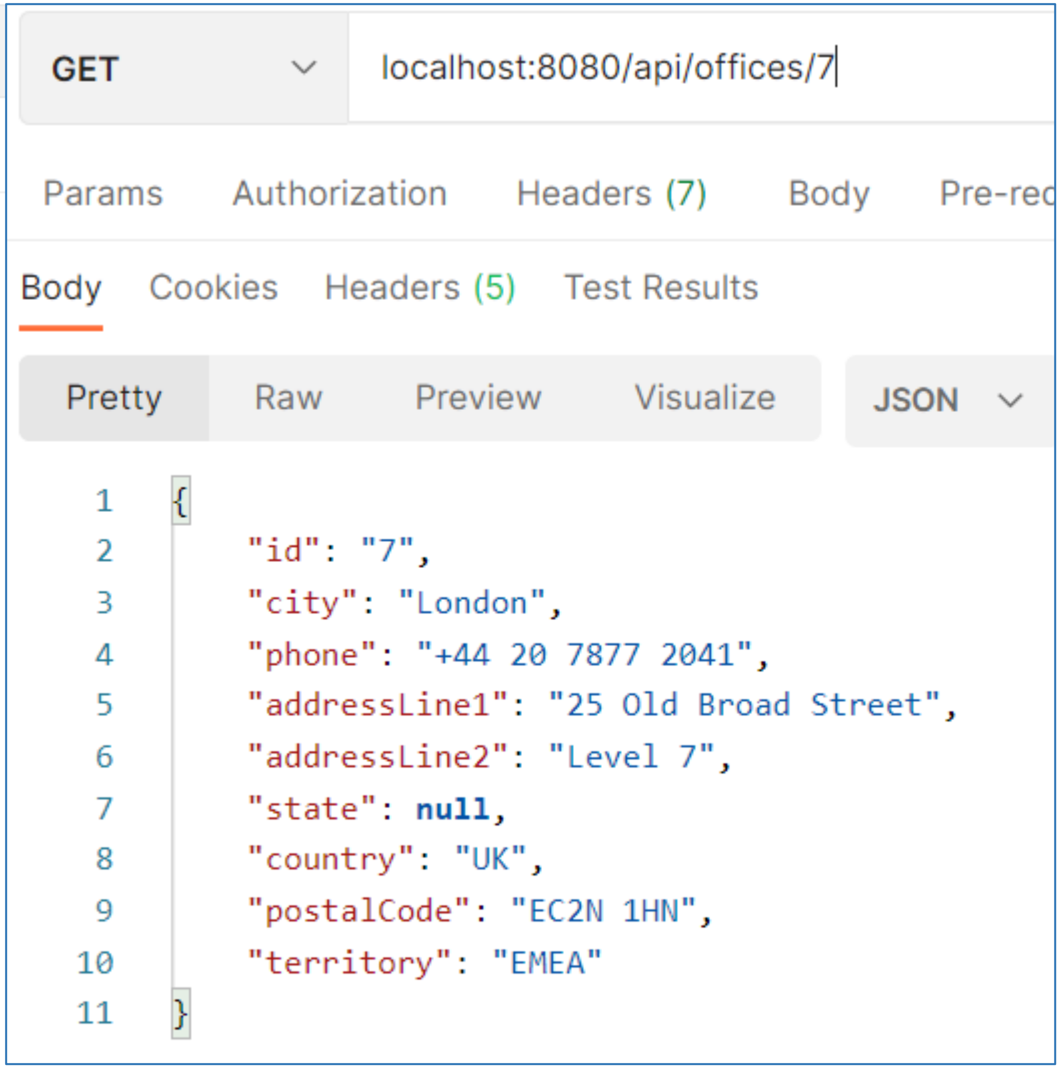
```
@PutMapping("/{officeCode}")
public Office updateOffice(@RequestBody Office office, @PathVariable String officeCode) {
   return service.updateOffice(officeCode, office);
}


@DeleteMapping("/{officeCode}")
 public void removeOffice(@PathVariable String officeCode) {
     service.removeOffice(officeCode);
 }
}
```


JSON
Controller
Presentation
Entity
Business Logic (Service Class)
Entity
Persistence (Repository Class)

Step 6: Testing the APIs (GET)

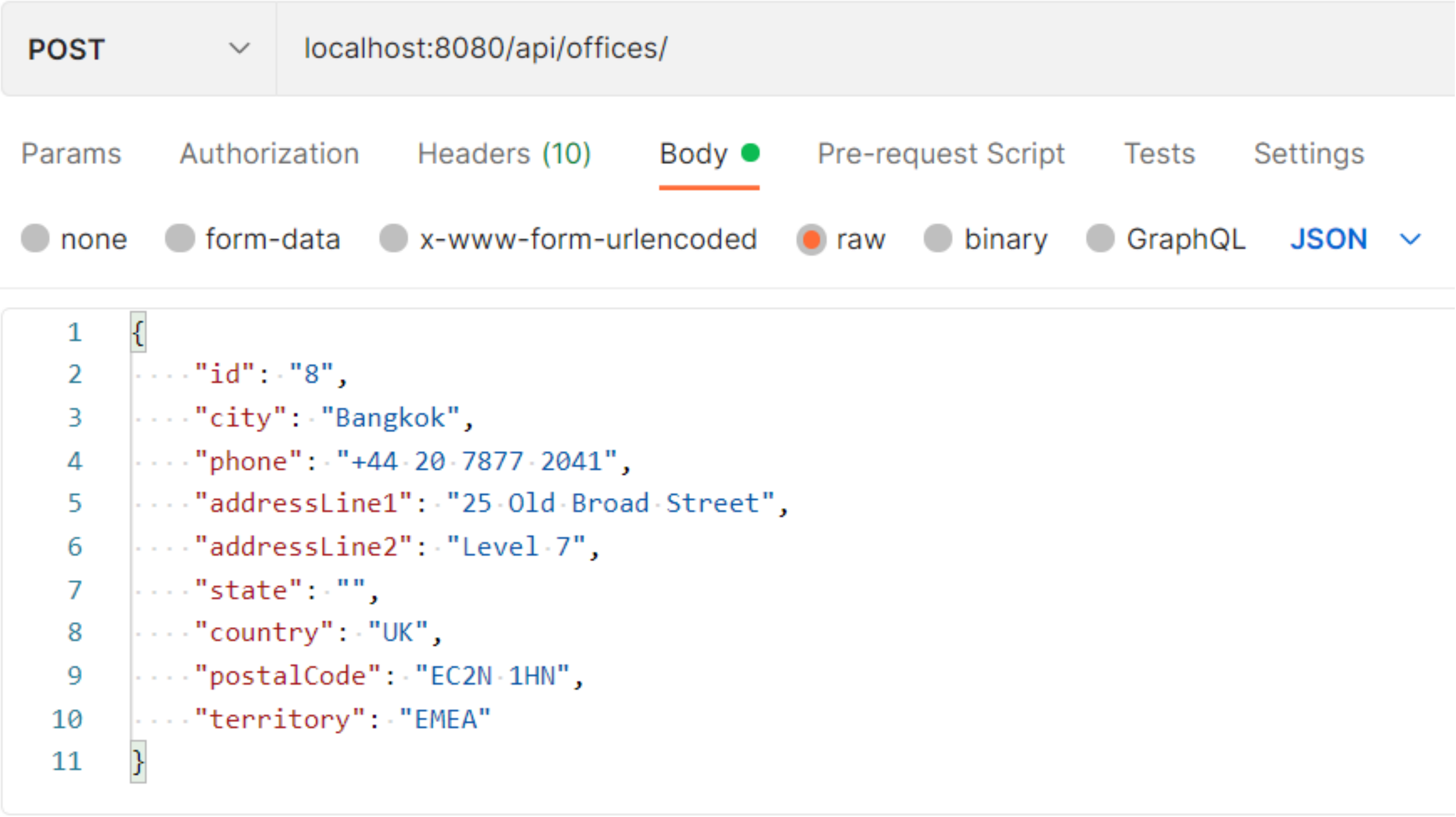


```
[
    {
        "id": "1",
        "city": "San Francisco",
        "phone": "+1 650 219 4782",
        "addressLine1": "100 Market Street",
        "addressLine2": "Suite 300",
        "state": "CA",
        "country": "USA",
        "postalCode": "94080",
        "territory": "NA"
    },
    {
        "id": "2",
        "city": "Boston",
        "phone": "+1 215 837 0825",
        "addressLine1": "1550 Court Place",
```



```
{
    "id": "7",
    "city": "London",
    "phone": "+44 20 7877 2041",
    "addressLine1": "25 Old Broad Street",
    "addressLine2": "Level 7",
    "state": null,
    "country": "UK",
    "postalCode": "EC2N 1HN",
    "territory": "EMEA"
}
```

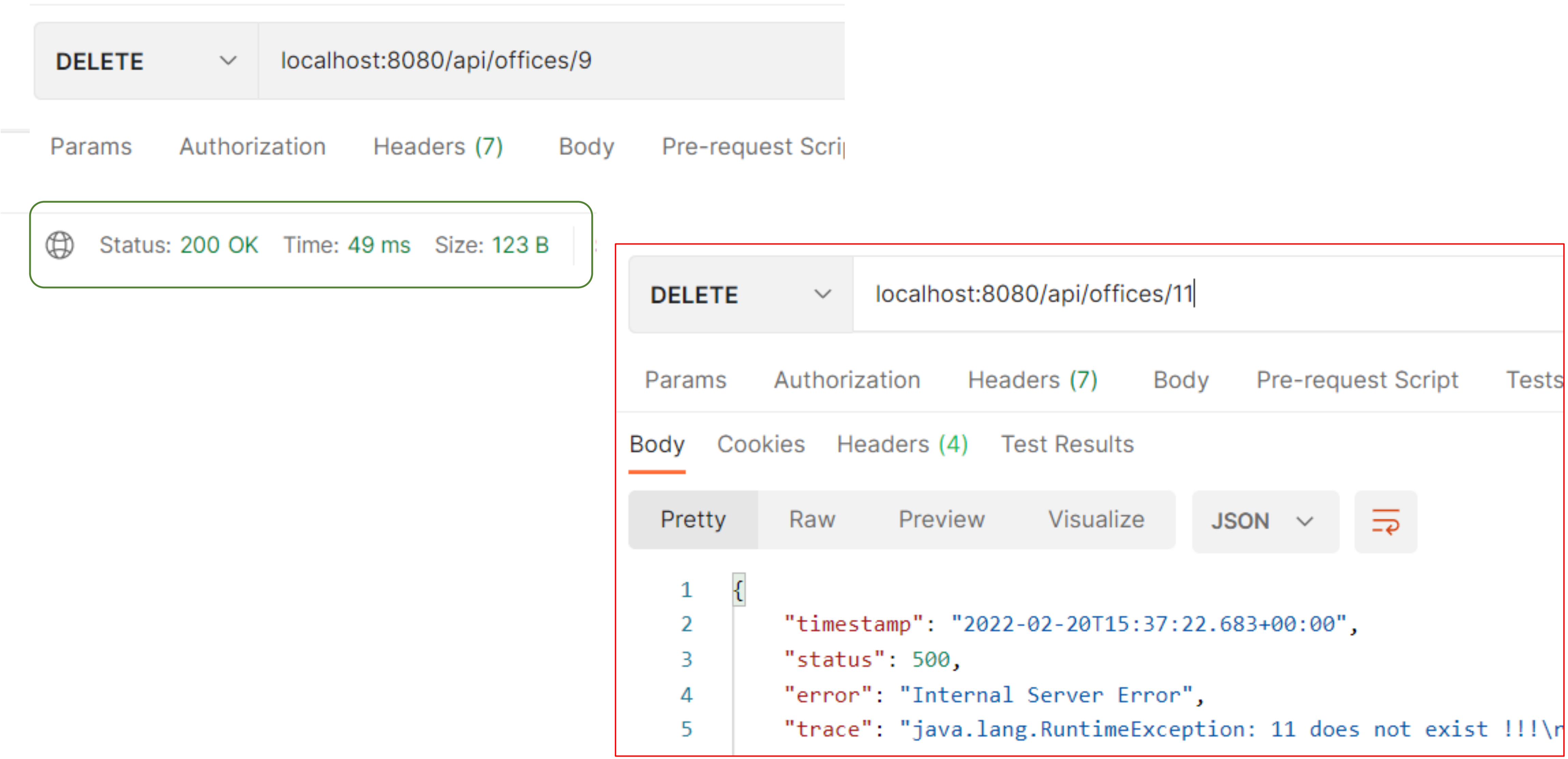
Step 7: Testing the APIs (POST)



POST localhost:8080/api/offices/

```
{
    "id": "8",
    "city": "Bangkok",
    "phone": "+44 20 7877 2041",
    "addressLine1": "25 Old Broad Street",
    "addressLine2": "Level 7",
    "state": "",
    "country": "UK",
    "postalCode": "EC2N 1HN",
    "territory": "EMEA"
}
```

Step 8: Testing the APIs (DELETE)



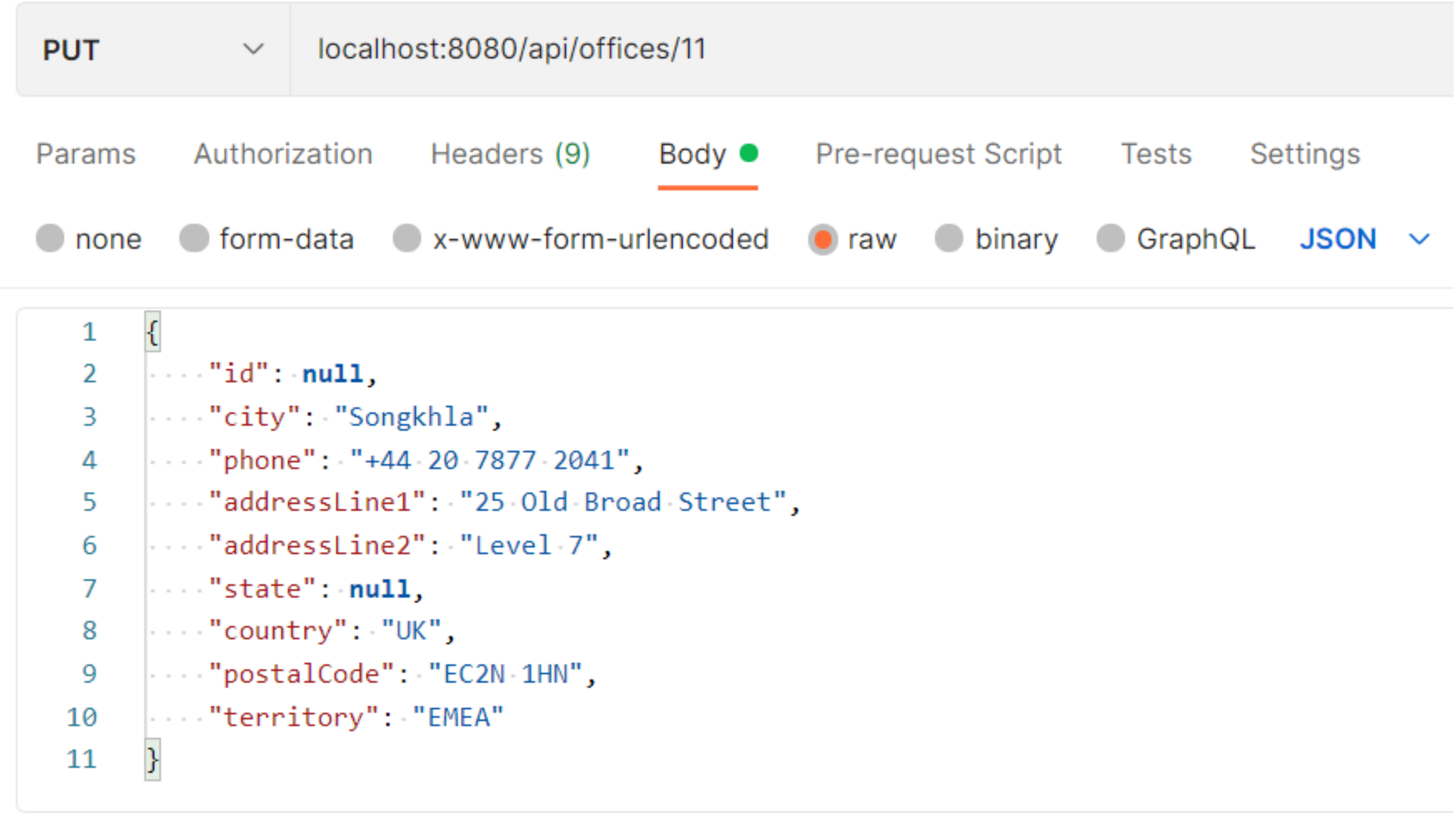
DELETE localhost:8080/api/offices/9

Status: 200 OK Time: 49 ms Size: 123 B

DELETE localhost:8080/api/offices/11

```
{
    "timestamp": "2022-02-20T15:37:22.683+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "trace": "java.lang.RuntimeException: 11 does not exist !!!\r
```

Step 8: Testing the APIs (PUT)



PUT localhost:8080/api/offices/11

```json
{
    "id": null,
    "city": "Songkhla",
    "phone": "+44 20 7877 2041",
    "addressLine1": "25 Old Broad Street",
    "addressLine2": "Level 7",
    "state": null,
    "country": "UK",
    "postalCode": "EC2N 1HN",
    "territory": "EMEA"
}
```

Summary

JSON : JavaSrcipt Object Notation

- A lightweight data-interchange format.
- Easy for humans to read and write.
- Easy for machines to parse and generate.
- Based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999.
- JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.

https://www.json.org/json-en.html

https://education.launchcode.org/js-independent-track/chapters/fetch-json/index.html

RESTful Resource

URI	HTTP Verb	Description
api/offices	GET	Get all office
api/offices/1	GET	Get an office with id = 1
api/offices/1/employees	GET	Get all employee for office id = 1
api/offices	POST	Add new office
api/offices/1	PUT	Update an office with id = 1
api/offices/1	DELETE	Delete an office with id = 1

REST API (also known as RESTful API)

- REST stands for **RE**presentational **S**tate **T**ransfer and was created by computer scientist Roy Fielding.
- An application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services.
- In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the **resources**.
- Each **resource** is **identified** by URIs/ global IDs.
- REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

Rules of REST API

- There are certain rules which should be kept in mind while creating REST **API endpoints**.
  - REST is based on the resource or **noun** instead of action or verb based. It means that a URI of a REST API should always end with a noun. Example: **/api/users** is a good example.
  - HTTP verbs are used to identify the action. Some of the HTTP verbs are – GET, PUT, POST, DELETE, PATCH.
  - A web application should be organized into resources like users and then uses HTTP verbs like – GET, PUT, POST, DELETE to modify those resources. And as a developer it should be clear that what needs to be done just by looking at the endpoint and HTTP method used.

RESTful Resource Example

URI	HTTP verb	Description
api/users	GET	Get all users
api/users/new	GET	Show form for adding new user
api/users	POST	Add a user
api/users/1	PUT	Update a user with id = 1
api/users/1/edit	GET	Show edit form for user with id = 1
api/users/1	DELETE	Delete a user with id = 1
api/users/1	GET	Get a user with id = 1

Always use plurals in URL to keep an API URI consistent throughout the application.
Send a proper HTTP code to indicate a success or error status.

RESTFul Principles and Constraints

- RESTFul Client-Server
  - Server will have a RESTful web service which would provide the required functionality to the client.
  - The client send's a request to the web service on the server. The server would either reject the request or comply and provide an adequate response to the client.
- Stateless
  - Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.
  - The server cannot take advantage of any previously stored context information on the server.
  - For this reason, the client application must entirely keep the session state.

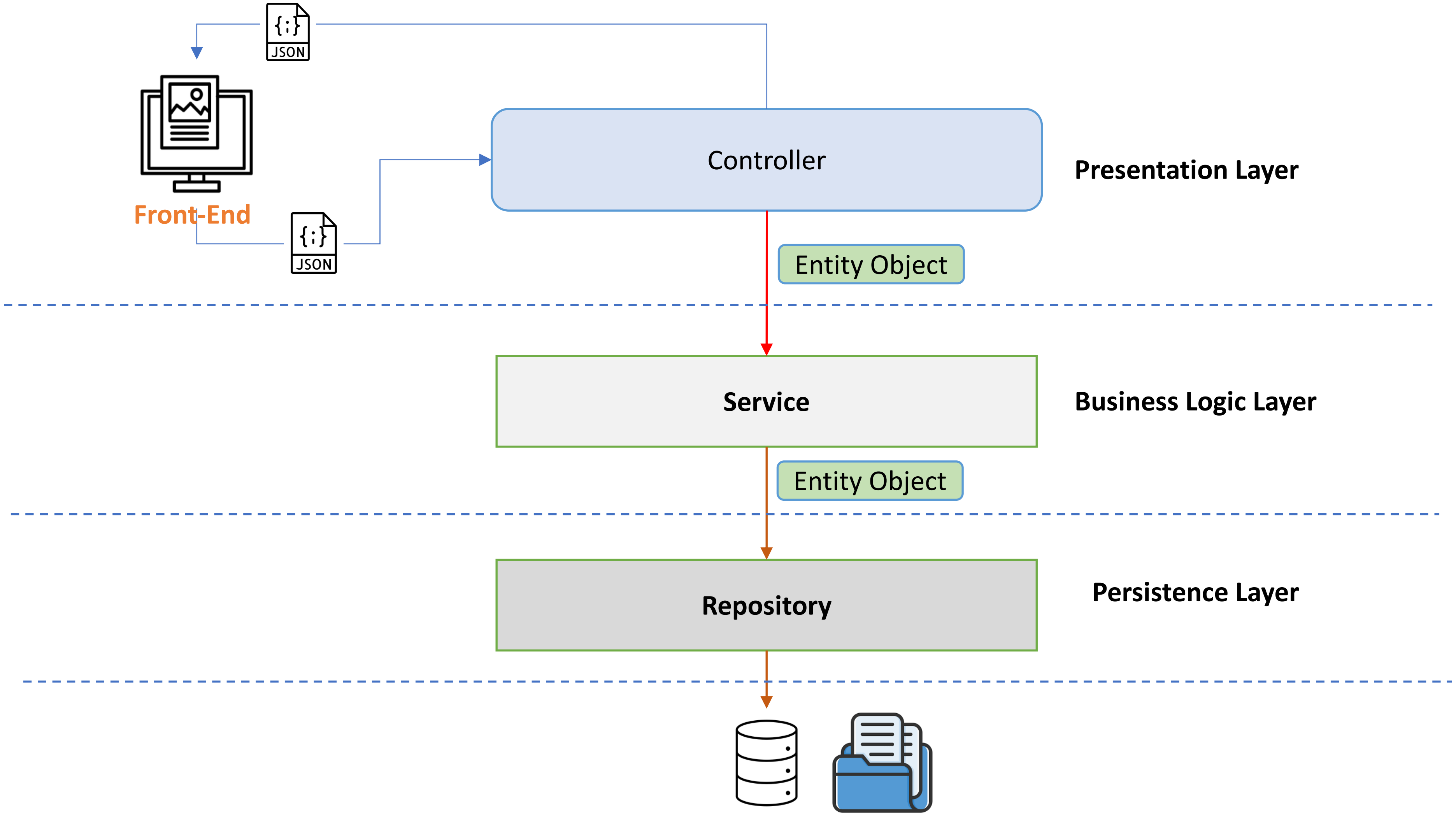
RESTFul Principles and Constraints (2)

- Interface/Uniform Contract
  - This is the underlying technique of how RESTful web services should work. RESTful basically works on the HTTP web layer and uses the below key verbs to work with resources on the server.
    - POST – To create a resource on the server
    - GET – To retrieve a resource from the server
    - PUT – To change the state of a resource or to update it
    - DELETE – To remove or delete a resource from the server
- Code on demand (optional)
  - REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.
  - Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

RESTFul Principles and Constraints (3)

- Cacheable
  - The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.
  - If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.
- Layered system
  - The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior.
  - For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with.

Layered System



Controller

```
@RestController
@RequestMapping("/api/offices")
public class OfficeController {
  @Autowired
  private OfficeService service;

  @GetMapping("")
  public List<Office> getAllOffices() {
    return service.getAllOffice();
  }

  @GetMapping("/{officeCode}")
  public Office getOfficeById(@PathVariable String officeCode) {
    return service.getOffice(officeCode);
  }


  @PostMapping("")
  public Office addNewOffice(@RequestBody Office office) {
     return service.createNewOffice(office);
 }
```

In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. These components are identified by the @RestController annotation, the data returned by each method will be written straight into the response body instead of rendering a template.

You can use the @RequestMapping annotation to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. You can use it at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

An annotation used in Spring Boot to enable **automatic dependency injection**. It allows the Spring container to provide an instance of a required dependency when a bean is created. This annotation can be used on fields, constructors, and methods to have Spring provide the dependencies automatically

What is a Postman

- Postman is an API platform for building and using APIs. Postman simplifies each step of the API lifecycle and streamlines collaboration so you can create better APIs—faster.

Creating a Spring Boot REST API Project

- Step 1: Initializing a Spring Boot Project
- Step 2: Connecting Spring Boot to the Database
- Step 3: Creating a User Model
- Step 4: Creating Repository Interface (Persistence Layer)
- Step 5: Creating Service Classes (Business Layer)
- **Step 6: Creating a Rest Controller** (Presentation Layer)
- Step 7: Compile, Build and Run
- **Step 8: Testing the APIs** POSTMAN