Session 04

Implementing RESTful API with JPA

With Java 21 and Spring Boot 3.2

Session 03

- Create a spring boot application **blogger-box-backend**
- Publish project on **Github** Repository
- Expose our first **endpoints**
- Document your endpoints with **Swagger**
- HTTP request **methods** with **conventions**
- Expose all **endpoints** for a blogger platform

Session 04

- **Structure** your code
- Create a **database** remotely
- Connect backend to a **database** via **JPA**
- **Http code**
- **Exception** handling

Architecture and Structure

Dependency Injection is a fundamental aspect of the Spring framework, through which the Spring container “**injects**” objects into other objects or “**dependencies**”.

Simply put, this allows for loose coupling of components and moves the responsibility of managing components onto the container.

Architecture and Structure

Let’s rely on **dependency Injection** to better structure our code.

We will create a service class for each model :

- CategoryService
- PostService

And each service will hold it use cases / functionalities

CategoryService

Annotating with **`@Service`** will allow spring to create a bean for **`CategoryService`**, so we can inject it in another class, like **`CategoryController`**

```java
public interface CategoryService {  5 usages  1 implementation

    List<Category> getAll();  1 usage  1 implementation

    Category getById(UUID id);  2 usages  1 implementation

    Category create(String name);  1 implementation

    Category updateName(UUID id, String name);  1 usage  1 implementation

    boolean deleteById(UUID id);  1 usage  1 implementation
}
```

CategoryServiceImpl.java

```java
package com.dauphine.blogger.services.impl;

import com.dauphine.blogger.models.Category;
import com.dauphine.blogger.services.CategoryService;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

@Service
public class CategoryServiceImpl implements CategoryService {

    private final List<Category> temporaryCategories;  9 usages

    public CategoryServiceImpl() {
        temporaryCategories = new ArrayList<>();
        temporaryCategories.add(new Category(UUID.randomUUID(),  name: "my first category"));
        temporaryCategories.add(new Category(UUID.randomUUID(),  name: "my second category"));
        temporaryCategories.add(new Category(UUID.randomUUID(),  name: "my third category"));
    }

    @Override  no usages
    public List<Category> getAll() { return temporaryCategories; }

    @Override  no usages
    public Category getById(UUID id) {
        return temporaryCategories.stream()  Stream<Category>
                .filter(category -> id.equals(category.getId()))
                .findFirst()  Optional<Category>
                .orElse(  other: null);
    }

    @Override  no usages
    public Category create(String name) {
        Category category = new Category(UUID.randomUUID(), name);
        temporaryCategories.add(category);
        return category;
    }

    @Override  no usages
    public Category update(UUID id, String newName) {
        Category category = temporaryCategories.stream()  Stream<Category>
                .filter(c -> id.equals(c.getId()))
                .findFirst()  Optional<Category>
                .orElse(  other: null);
        if (category != null) {
            category.setName(newName);
        }
        return category;
    }

    @Override  no usages
    public void deleteById(UUID id) { temporaryCategories.removeIf(category -> id.equals(category.getId())); }

}
```

CategoryController

Controller	Management of the REST endpoints
Service	Business Logic Implementations

Inject **CategoryService** in the constructor

The **implementation** and **business logic** of each method will be done at the service layer

CategoryController.java

```java
package com.dauphine.blogger.controllers;

import com.dauphine.blogger.models.Category;
import com.dauphine.blogger.models.Post;
import com.dauphine.blogger.services.CategoryService;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

@RestController
@RequestMapping("/v1/categories")
public class CategoryController {

    private final CategoryService service;  6 usages

    public CategoryController(CategoryService service) {
        this.service = service;
    }

    @GetMapping
    public List<Category> retrieveAllCategories() {
        return service.getAll();
    }

    @GetMapping("{id}")
    public Category retrieveCategoryById(@PathVariable UUID id) {
        return service.getById(id);
    }

    @PostMapping
    public Category createCategory(@RequestBody String name) {
        return service.create(name);
    }

    @PutMapping("{id}")
    public Category updateCategory(@PathVariable UUID id,
                                   @RequestBody String name) {
        return service.update(id, name);
    }

    @DeleteMapping("{id}")
    public UUID deleteCategory(@PathVariable UUID id) {
        return service.deleteById(id);
    }
```

Sync with Github

add `category` service layer

Implement PostService and the changes in the controller

```java
public interface PostService {  8 usages  1 implementation

    List<Post> getAllByCategoryId(UUID categoryId);  1 usage  1 implementation

    List<Post> getAll();  1 usage  1 implementation

    Post getById(UUID id);  2 usages  1 implementation

    Post create(String title, String content, UUID categoryId);  1 implementation

    Post update(UUID id, String title, String content);  1 usage  1 implementation

    boolean deleteById(UUID id);  1 usage  1 implementation

}
```

Sync with Github

add `post` service layer

Create database remotely

Create and **Connect** to a remotely **PostgreSQL** database via **Elephant SQL**

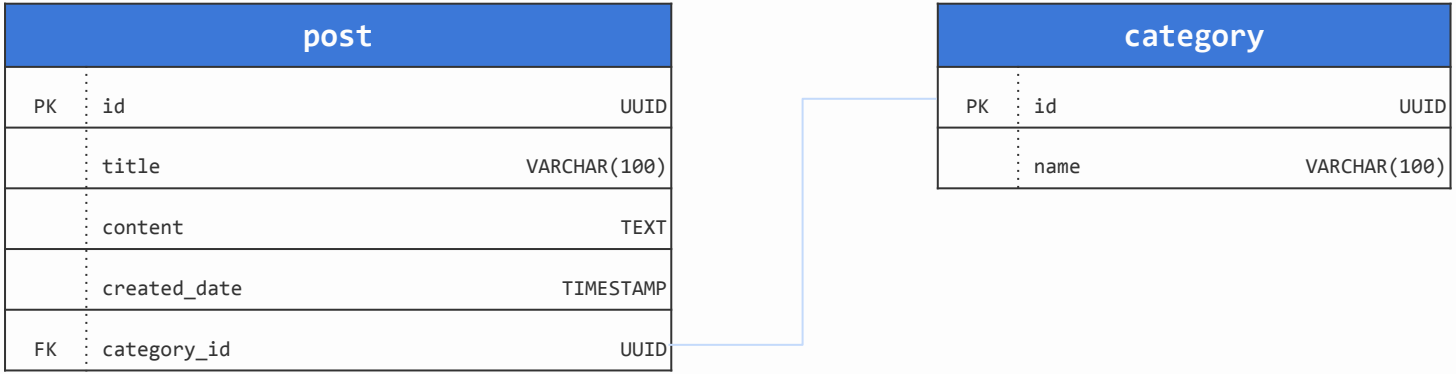
Elephant SQL

PostgreSQL

Alternatives to Elephant SQL (since it will be shut down in 2025):

- *https://neon.tech*
- *https://heroku.com*
- *https://supabase.com*

Model


post
PK id UUID
title VARCHAR(100)
content TEXT
created_date TIMESTAMP
FK category_id UUID
category
PK id UUID
name VARCHAR(100)

Category Model

Category.java

```java
package com.dauphine.blogger.models;

import java.util.UUID;

public class Category {  12 usages

    private UUID id;  2 usages
    private String name;  2 usages

    public UUID getId() {  3 usages
        return id;
    }

    public void setId(UUID id) {  no usages
        this.id = id;
    }

    public String getName() {  no usages
        return name;
    }

    public void setName(String name) {  1 usage
        this.name = name;
    }
}
```

category		
PK	id	UUID
	name	VARCHAR(100)

Post model

post		
PK	id	UUID
	title	VARCHAR(100)
	content	TEXT
	created_date	TIMESTAMP
FK	category_id	UUID

category		
PK	id	UUID
	name	VARCHAR(100)

Post.java

```java
package com.dauphine.blogger.models;

import java.time.LocalDateTime;
import java.util.UUID;

public class Post {  6 usages

    private UUID id;  2 usages
    private String title;  2 usages
    private String content;  2 usages
    private LocalDateTime createdDate;  2 usages
    private Category category;  2 usages

    public UUID getId() {  no usages
        return id;
    }

    public void setId(UUID id) {  no usages
        this.id = id;
    }

    public String getTitle() {  no usages
        return title;
    }

    public void setTitle(String title) {  no usages
        this.title = title;
    }

    public String getContent() {  no usages
        return content;
    }

    public void setContent(String content) {  no usages
        this.content = content;
    }

    public LocalDateTime getCreatedDate() {  no usages
        return createdDate;
    }

    public void setCreatedDate(LocalDateTime createdDate) {
        this.createdDate = createdDate;
    }

    public Category getCategory() {  no usages
        return category;
    }

    public void setCategory(Category category) {  no usages
        this.category = category;
    }
}
```

Populate database

Create tables (Category and Post)

Add some data into the created table

check slide 33 from session 01…

Connect to the database

In order to connect to the database, let’s add a new dependency Data JPA and Postgres , which will allow to persist into a database

```
<dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
</dependency>
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

```
<dependencies> Edit Starters...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
        <version>${swagger.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
```

Configure database in properties

application.properties

```
spring.application.name=Blogger box backend

spring.datasource.url=jdbc:postgresql://REPLACE_WITH_HOST:5432/REPLACE_WITH_NAME
spring.datasource.username=REPLACE_WITH_USERNAME
spring.datasource.password=REPLACE_WITH_PASSWORD
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.database=POSTGRESQL
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=validate
```

Database connections

Specify database

Show executed queries in log

Deactivate JPA hibernate auto generation

Change our model classes into entities

Category.java

```java
package com.dauphine.blogger.models;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

import java.util.UUID;

@Entity  25 usages
@Table(name = "category")
public class Category {

    @Id
    @Column(name = "id")
    private UUID id;

    @Column(name = "name")  3 usages
    private String name;

    public Category(UUID id, String name) {  4 usages
        this.id = id;
        this.name = name;
    }

    public Category() {
    }

    public UUID getId() { return id; }

    public void setId(UUID id) { this.id = id; }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }
}
```

Table and **column** map

Very important to have an **empty constructor** (add one in case you have created a new one that takes parameter) and **getters** and **setters** for each property

Change our model classes into entities

```java
@Entity  28 usages
@Table(name = "post")
public class Post {

    @Id
    @Column(name = "id")
    private UUID id;

    @Column(name = "title")  3 usages
    private String title;

    @Column(name = "content")  3 usages
    private String content;

    @Column(name = "created_date")  3 usages
    private LocalDateTime createdDate;

    @ManyToOne  3 usages
    @JoinColumn(name = "category_id")
    private Category category;

    public Post() {
    }

    public Post(String title,  no usages
                String content,
                Category category) {
        this.id = UUID.randomUUID();
        this.title = title;
        this.content = content;
        this.createdDate = LocalDateTime.now();
        this.category = category;
    }
```

Table &
Column
mapping

Many to one relation in Post, as a post have only 1 category but the categories can have many posts, hence the ManyToOne annotation

Repository layer

Controller	Management of the REST endpoints
Service	Business Logic Implementations
Repository	Storage of the models/entities

CategoryRepository.java

```java
package com.dauphine.blogger.repositories;

import com.dauphine.blogger.models.Category;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.UUID;

public interface CategoryRepository extends JpaRepository<Category, UUID> {
}
```

JPA repository allows to generate data storage operation (SELECT, UPDATE, DELETE, …) queries without the need to write any …

Entity model class

Primary key type

Inject Repositrory in service

```
@Service
public class CategoryServiceImpl implements CategoryService {

    private final CategoryRepository repository;  6 usages

    public CategoryServiceImpl(CategoryRepository repository) {
        this.repository = repository;
    }
```

Retrieve all existing categories

We will use **`findAll`** method already implemented in **`JpaRepository`**

```
20      @Override  1 usage
21      public List<Category> getAll() {
22          return repository.findAll();
23      }
```

Retrieve an existing category by id

We will use **`findById`** method already implemented in **`JpaRepository`**

```
25      @Override  2 usages
26      public Category getById(UUID id) {
27          return repository.findById(id)
28                  .orElse( other: null);
29      }
```

Create new category

We will use **`save`** method already implemented in **`JpaRepository`** which will create the row in database if it doesn't exists

```
31      @Override
32      public Category create(String name) {
33          Category category = new Category(name);
34          return repository.save(category);
35      }
```

Update

We will use **`save`** method already implemented in **`JpaRepository`** which will update the row in database if it does exists

```
37      @Override  1 usage
38      public Category updateName(UUID id, String name) {
39          Category category = getById(id);
40          if (category == null) {
41              return null;
42          }
43          category.setName(name);
44          return repository.save(category);
45      }
```

Delete

We will use **`deleteById`** method already implemented in **`JpaRepository`**

```
47      @Override  1 usage
48      public boolean deleteById(UUID id) {
49          repository.deleteById(id);
50          return true;
51      }
```

Sync with Github

add `category` repository layer with JPA

Add repository layer for Post

Add repository layer for Post and complete service layer

Sync with Github

add `post` repository layer with JPA

Exposing a new endpoint that finds all categories by name

We can write custom queries the repository layer, by default, the query definition uses **JPQL**.

Here an example on how to find all categories by name

```
public interface CategoryRepository extends JpaRepository<Category, UUID> {

    @Query("""  1 usage  Elie Daher
        SELECT category
        FROM Category category
        WHERE UPPER(category.name) LIKE UPPER(CONCAT('%', :name ,'%'))
    """)
    List<Category> findAllLikeName(@Param("name") String name);

}
```

Exposing a new endpoint that find all categories by name

And we will modify the following endpoint

```
@GetMapping  Elie Daher
@Operation(
        summary = "Get all categories",
        description = "Retrieve all categories or filter like name"
)
public List<Category> getAll(@RequestParam(required = false) String name) {
    List<Category> categories = name == null || name.isBlank()
            ? categoryService.getAll()
            : categoryService.getAllLikeName(name);
    return categories;
}
```

Get all categories

```
GET /v1/categories
```

Get all categories by name

```
GET /v1/categories?name=Child
```

Expose an endpoint that find all posts by title or content

Get all topics by title or content

```
GET /v1/topics?value=...
```