Build a Blog Platform App

Published: March 16, 2025

Getting Started

Welcome

Welcome to the blog platform builder project!

I'll guide you through the process of building blog platform Spring Boot application, step by step. I'll even provide a React application frontend you can use to interact with the Spring Boot application.

Please make full use of the community to ask questions if you get stuck!

Have fun

Aaron

Prerequisites

In the upcoming lessons we'll be focused on building an application rather than discussing the theory in detail, so a foundational knowledge will help you to get the very most out of this project.

To follow along and build the blog platform app you'll need a practical knowledge of Java, Maven, Docker and PostgreSQL.

By "practical knowledge" I mean that I won't be explaining how technologies work in this project, instead I'll assume you're already comfortable with them so we can focus on building our project.

For Spring Boot, you should be comfortable with the basic concepts.

Where this project becomes more challenging is that we'll be using Spring Security and JSON Web Tokens, or JWTs. So a basic knowledge of Spring Security and how JWTs work would be very beneficial, although not strictly needed, as once our security is configured we'll not need to revisit it.

A special note about our project's React frontend. Of course, this project is focused on building a Spring Boot application, however I do offer the same React frontend I use.

We'll be using the Node Package Manager (npm) to install the dependencies for the frontend, so although you don't need to know how Node works, or how to build a React application, it would be beneficial to know what Node is and what is means to use npm. Otherwise, a quick read of the Node website should be enough to understand this.

In the next lesson we'll look to set up our development environment.

Summary

- We will be focused on building rather than deep-diving theory in this project
- You'll need a basic knowledge of Java, Maven, PostgreSQL & Spring Boot to follow along effectively
- Knowledge of Spring Security & JWTs would be beneficial
- I offer a React frontend to help you follow along, though we won't code it in this project.
- We'll be using npm to run the React frontend

Development Environment Setup

Let’s set up our development environment.

Java

To build the blog app you’re going to need Java 21 or later.

You can check your Java version by opening up a terminal or a command prompt and typing:

```
java -version
```

If you see an earlier version then I’d recommend heading over to oracle.com to download JDK 21 or later.

Or if you prefer to use an open source JDK, then you can download it from Adoptium, which is a part of the Eclipse Foundation.

Either JDK should work for this project.

Maven

We’ll be using Apache Maven to manage our project, although you’ll not need to install this on your system as an instance of Maven will come bundled with your skeleton Spring Boot project, but more on that later.

Node

To run the frontend code I’ll provide, you’ll need Node version 20 or later.

You can check your Node version by opening up a terminal or a command prompt and typing:

```
node --version
```

Note it’s two dashes before “version” for node, but only one for Java.

If you don’t have the required version then head over to nodejs.org to download a later version of node.

Docker

To run the PostgreSQL database we’ll be using later you’ll need docker installed on your machine.

To check you have docker installed, open up a terminal or command prompt and type:

```
docker --version
```

You get a version number printed out, otherwise head over to docker.com to download docker.

IDE

I’m going to be using the community version of IntelliJ IDEA as the IDE for this project.

You can use any IDE you like, such as Visual Studio Code, but I recommend IntelliJ as it’s brilliant for Java development.

You can download IntelliJ for free from the JetBrains website.

Summary

- You’ll need JDK 21 or later.
- We’ll be using Maven, but you don’t need to install this.
- You’ll need Node V20 or later to run the frontend.
- You’ll need Docker installed in order to easily run PostgreSQL.
- You’ll need an IDE, I recommend IntelliJ IDEA.

With everything installed, let’s check out the requirements of what we are going to be building.

Project Setup

Create a New Project

Project Generation

Spring Initializr provides a web-based tool for generating Spring Boot projects with the necessary dependencies and configuration.

Navigate to https://start.spring.io in your web browser.

The interface presents various options for customizing your project structure and dependencies.

On the left side, configure the following project settings:

```
Project: Maven
Language: Java
Spring Boot: 3.4.0
Group: com.devtiro
Artifact: blog
Name: blog
Description: A blog platform
Package name: com.devtiro.blog
Packaging: Jar
Java: 21
```

Dependency Selection

Dependencies determine which Spring Boot features and libraries will be available in your project.

Click on the "ADD DEPENDENCIES" button to open the dependency selector.

Search for and select the following dependencies:

```
Spring Web
Spring Data JPA
PostgreSQL Driver
Spring Security
Lombok
Validation
H2
```

Each dependency serves a specific purpose:

- Spring Web: Enables building web applications and RESTful services
- Spring Data JPA: Provides data persistence and ORM capabilities
- PostgreSQL Driver: Allows connection to PostgreSQL databases
- Spring Security: Adds authentication and authorization features
- Lombok: Reduces boilerplate code through annotations
- Validation: Enables data validation using annotations
- H2: An in-memory database we'll use for our test

Project Setup

After generating the project, you need to set it up in your development environment.

Click the "GENERATE" button to download the project as a ZIP file.

Extract the ZIP file to your preferred workspace directory.

Now let's explore the project in our IDE.

Summary

- The Spring Initializr (start.spring.io) provides a web interface for generating Spring Boot projects
- Project configuration includes basic metadata like group ID, artifact ID, and Java version
- Essential dependencies for a blog platform include Web, JPA, Security, and PostgreSQL support
- The generated project contains all necessary configuration files and a basic project structure

Explore the Project

Now, we'll explore the project structure and understand how different components fit together.

Project Dependencies

The pom.xml file defines our project's dependencies and configuration.

Let's examine the key dependencies we included:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

Project Structure

Spring Boot follows a conventional project structure that promotes organization and maintainability.

The main application components are organized under src/main/java/com/devtiro/blog/:

```
src/main/java/com/devtiro/blog/
├── BlogApplication.java
├── controllers/       <= We'll create this later
├── services/          <= We'll create this later
├── repositories/      <= We'll create this later
├── domain/            <= We'll create this later
└── config/
```

The BlogApplication.java file serves as our application's entry point:

```
@SpringBootApplication
public class BlogApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(BlogApplication.class, args);
    }
}
```

Build the Project

To make sure our project is set up correctly, we can use Maven to build and run tests.

Open a terminal in your project directory and run:

```
# Windows
.\mvwn clean install

# *nix
./mvnw clean install
```

Summary

- The pom.xml file manages project dependencies including Spring Web, JPA, and PostgreSQL
- The @SpringBootApplication annotation enables auto-configuration and component scanning and mark our application as a Spring Boot application

Running a Database

Now we'll set up a PostgreSQL database using Docker Compose, which will serve as the persistent data store for our blog.

This containerized approach ensures consistency across development environments and simplifies database management.

Understanding Docker Compose

Docker Compose is a tool that defines and manages multi-container Docker environments.

For our blog platform, we need a PostgreSQL database server and (optionally) a database management interface.

Docker Compose allows us to define these services in a declarative way using a YAML file.

Creating the Docker Compose File

Let's create a new file named docker-compose.yml in the project root directory with the following content:

```
services:
  # Our PostgreSQL database
  db:
    # Using the latest PostgreSQL image
    image: postgres:latest
    ports:
      - "5432:5432"
    restart: always
    environment:
      POSTGRES_PASSWORD: changemeinprod!

  # Database management interface
  adminer:
    image: adminer:latest
    restart: always
    ports:
      - 8888:8080
```

Running the Database

Starting the database requires Docker to be installed and running on your system.

Open a terminal in your project directory and run:

```
docker-compose up
```

This command starts both the PostgreSQL database and Adminer.

Accessing the Database

The PostgreSQL database is now accessible on port 5432.

You can access the Adminer interface at http://localhost:8888 with these credentials:

- System: PostgreSQL
- Server: db
- Username: postgres
- Password: changemeinprod!

Summary

- Docker Compose enables consistent database setup across development environments
- PostgreSQL runs in a container, avoiding direct installation on the host system
- Adminer provides a web interface for database management
- The database automatically restarts if the Docker daemon restarts

Connecting to PostgreSQL

Now we'll configure our Spring Boot application to connect to this database.

Database Configuration

Spring Boot uses the application.properties file to manage database connection settings.

Create or open src/main/resources/application.properties and add these properties:

```
# Database Connection
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=changemeinprod!

# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Understanding the Configuration

The database URL follows a specific format that tells Spring how to connect to our PostgreSQL instance.

The spring.jpa.hibernate.ddl-auto=update setting enables Hibernate to automatically update the database schema based on our entity classes.

The show-sql and format_sql properties help during development by logging SQL statements to the console.

Summary

- Spring Boot auto-configures most database connection properties
- The application.properties file centralizes database configuration
- Hibernate's ddl-auto=update mode automatically manages schema changes

Set up MapStruct

We'll now integrate MapStruct, a code generator that simplifies the conversion between our domain entities and DTOs (Data Transfer Objects).

We'll only be setting up MapStruct in this lesson, we'll cover using MapStruct later on in the course.

Understanding MapStruct

MapStruct is a compile-time annotation processor that generates type-safe mapping code.

The generated code performs direct property mapping, avoiding the performance overhead of reflection-based mapping frameworks.

MapStruct integrates with Lombok and Spring, making it an ideal choice for our blog application, but it will need a little configuring.

Adding Dependencies

First, we need to add MapStruct dependencies to our pom.xml file. Note we're specifying the Lombok version here -- we need to do this to make sure the versions of MapStruct and Lombok we have are compatible.

```
<properties>
    <org.mapstruct.version>1.6.3</org.mapstruct.version>
    <lombok.version>1.18.36</lombok.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct</artifactId>
        <version>${org.mapstruct.version}</version>
    </dependency>
</dependencies>
```

Configuring the Compiler Plugin

The Maven compiler plugin needs special configuration to work with both MapStruct and Lombok:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <version>${lombok.version}</version>
                    </path>
```

```
                    <path>
                        <groupId>org.mapstruct</groupId>
                        <artifactId>mapstruct-processor</artifactId>
                        <version>${org.mapstruct.version}</version>
                    </path>
                    <path>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok-mapstruct-binding</artifactId>
                        <version>0.2.0</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
```

That's all we'll need to do for the moment. We can now use MapStruct to generate DTOs when we need them later on in the build!

Summary

- MapStruct generates efficient bean mapping code at compile-time, eliminating runtime reflection overhead
- Proper configuration of annotation processors is needed for MapStruct to work with Lombok

Run the Frontend

Once we have the code locally, we open up a terminal or command prompt and navigate to directory containing the frontend code.

Once we're here, we should install the necessary dependencies using the following command:

```
npm install
```

Once everything is installed, we'll run the app with the following command:

```
npm run dev
```

All that's left is to open up a browser and visit http://localhost:5173/

And our backend is running, although as we've not yet implemented the necessary endpoints on the backend it's not working completely just yet!

As we build out our backend we should see the frontend starting to resembled the demo earlier in the course.

Summary

- Install dependencies with npm install
- Run the frontend with npm run dev

Domain

Create the User Entity

Now we'll create the User entity, which will represent user accounts in our system.

Understanding JPA Entities

A JPA entity is a persistent domain object that represents a table in our database.

The entity class must be annotated with @Entity and follow certain conventions to work correctly with JPA.

Hibernate (JPA implementation) will use this class to automatically create the corresponding database table and manage the mapping between Java objects and database records.

Implementing the User Entity

The User entity needs to store essential information about each user in our system.

Here's the implementation of our User class:

```
package com.devtiro.blog.domain.entities;

import jakarta.persistence.*;
import lombok.*;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.Objects;
import java.util.Set;
import java.util.UUID;

@Entity
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@Builder
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
```

```
    private LocalDateTime createdAt;

    // We need to implement equals and hashCode explicitly because
    // Lombok's implementation can cause issues with JPA entities
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(id, user.id) &&
               Objects.equals(email, user.email) &&
               Objects.equals(password, user.password) &&
               Objects.equals(name, user.name) &&
               Objects.equals(createdAt, user.createdAt);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, email, password, name, createdAt);
    }

    @PrePersist
    protected void onCreate() {
        createdAt = LocalDateTime.now();
    }
}
```

Entity Design Considerations

Each field in our User entity serves a specific purpose and has constraints.

The @Column annotations ensure data integrity by specifying that email must be unique and that essential fields cannot be null.

We use UUID instead of sequential IDs for better security and scalability.

Lifecycle Callbacks

The @PrePersist annotation helps us manage entity lifecycle events automatically.

When a new user is created, JPA will automatically call our onCreate() method to set the creation timestamp.

This ensures we always have an accurate record of when each user account was created.

Summary

- The User entity forms the foundation for authentication and user management
- JPA annotations map Java objects to database tables automatically
- UUID primary keys provide better security than sequential IDs
- Custom equals and hashCode methods ensure proper entity behavior
- Lifecycle callbacks automate administrative field updates

Create the Category Entity

Now we'll implement the Category entity, which will provide a way to organize blog posts into broad groups.

Understanding Categories in Blog Systems

Blog categories serve as high-level containers that group related content together.

Categories differ from tags in that they represent broad, predefined classifications rather than flexible, user-defined keywords.

Implementing the Category Entity

Here's our initial implementation:

```
package com.devtiro.blog.domain.entities;

import jakarta.persistence.*;
import lombok.*;

import java.util.ArrayList;
import java.util.Objects;
import java.util.Set;
import java.util.UUID;

@Entity
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@Builder
@Table(name = "categories")
public class Category {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(nullable = false, unique = true)
    private String name;

    // Custom equals implementation needed to avoid Lombok's implementation
    // which can cause issues with JPA entity lifecycle management
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Category category = (Category) o;
        return Objects.equals(id, category.id) &&
               Objects.equals(name, category.name);
    }
```

```
    @Override
    public int hashCode() {
        return Objects.hash(id, name);
    }
}
```

Summary

- Categories provide a hierarchical organization structure for blog content
- Custom equals and hashCode methods ensure proper JPA entity behavior
- The unique constraint on category names prevents duplicates
- Using UUID as the primary key supports distributed systems and security

Create the Tag Entity

Now we'll implement the Tag entity, which provides a more flexible way to classify and organize blog posts.

Understanding Tags in Blog Systems

Tags represent keywords or phrases that describe specific aspects of blog content.

This flexibility allows content creators to precisely describe their posts and helps readers find exactly what they're looking for.

Implementing the Tag Entity

Here's our implementation of the Tag entity:

```
package com.devtiro.blog.domain.entities;

import jakarta.persistence.*;
import lombok.*;

import java.util.ArrayList;
import java.util.Objects;
import java.util.Set;
import java.util.UUID;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Table(name = "tags")
public class Tag {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(nullable = false, unique = true)
    private String name;

    // Custom equals/hashCode implementation needed instead of Lombok's
    // to avoid issues with JPA entity lifecycle management
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Tag tag = (Tag) o;
        return Objects.equals(id, tag.id) &&
               Objects.equals(name, tag.name);
    }
```

```
    @Override
    public int hashCode() {
        return Objects.hash(id, name);
    }
}
```

Entity Design Considerations

The unique constraint on the name field ensures we don't have duplicate tags in our system.

Summary

- Tags provide flexible, non-hierarchical content classification
- The unique constraint on tag names prevents duplicates while allowing multiple posts per tag
- Custom equals and hashCode implementations ensure proper JPA entity behavior

Create the Post Entity & Post Status Enum

Now we'll implement the Post entity and its associated PostStatus enum, which together represent the actual blog content in our system.

These classes will enable us to manage blog posts throughout their lifecycle, from draft to publication.

Understanding Post Status

A post status enum helps track the editorial state of each blog post in our system.

Let's create the PostStatus enum in com.devtiro.blog.domain:

```
package com.devtiro.blog.domain;

public enum PostStatus {
    DRAFT,
    PUBLISHED
}
```

Implementing the Post Entity

The Post entity represents individual blog entries and must store all content and metadata.

Here's our initial implementation:

```
package com.devtiro.blog.domain.entities;

import com.devtiro.blog.PostStatus;
import jakarta.persistence.*;
import lombok.*;

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

@Entity
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@Builder
@Table(name = "posts")
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(nullable = false)
    private String title;

    @Column(nullable = false, columnDefinition = "TEXT")
```

```
    private String content;

    @Column(nullable = false)
    @Enumerated(EnumType.STRING)
    private PostStatus status;

    @Column(nullable = false)
    private Integer readingTime;

    @Column(nullable = false)
    private LocalDateTime createdAt;

    @Column(nullable = false)
    private LocalDateTime updatedAt;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Post post = (Post) o;
        return Objects.equals(id, post.id) &&
               Objects.equals(title, post.title) &&
               Objects.equals(content, post.content) &&
               status == post.status &&
               Objects.equals(readingTime, post.readingTime) &&
               Objects.equals(createdAt, post.createdAt) &&
               Objects.equals(updatedAt, post.updatedAt);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, title, content, status, readingTime,
                            createdAt, updatedAt);
    }

    @PrePersist
    protected void onCreate() {
        createdAt = LocalDateTime.now();
        updatedAt = LocalDateTime.now();
    }

    @PreUpdate
    protected void onUpdate() {
        updatedAt = LocalDateTime.now();
    }
}
```

Entity Design Considerations

The columnDefinition = "TEXT" annotation ensures our database can store long-form blog content without size restrictions.

The @Enumerated(EnumType.STRING) annotation stores the post status as a readable string rather than a numeric value.

We include a readingTime field to enhance the user experience by showing estimated reading duration.

Audit Fields Implementation

The createdAt and updatedAt fields track the post's lifecycle.

JPA's @PrePersist and @PreUpdate annotations automatically maintain these timestamps.

This audit trail is valuable for content management and debugging.

Summary

- The PostStatus enum provides a type-safe way to track editorial state
- Post content uses the TEXT column type to handle varying content lengths
- Automatic timestamp management ensures accurate audit trails
- Reading time estimation enhances the user experience
- Custom equals and hashCode methods maintain entity integrity

Configure the User - Post Relationship

Now we'll establish the relationship between users and their blog posts, enabling proper content ownership and management.

Understanding Entity Relationships

A blog post must always have an author, and this relationship needs to be enforced at the database level.

Let's modify our Post entity to include the author relationship:

```
@Entity
@Table(name = "posts")
public class Post {
    // ... existing fields ...

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "author_id", nullable = false)
    private User author;

    // ... rest of the class ...
}
```

Configuring the User Side

The User entity needs to maintain a collection of their posts.

Here's how we update the User entity:

```
@Entity
@Table(name = "users")
public class User {
    // ... existing fields ...

    @OneToMany(mappedBy = "author")
    private List<Post> posts = new ArrayList<>();

    // ... rest of the class ...
}
```

Understanding Fetch Types

The FetchType.LAZY configuration optimizes performance by loading post authors only when explicitly accessed.

This is particularly important for list operations where we might load many posts but not need their author details immediately.

Cascades

Let's now define the cascades for this relationship. Cascades define how changes to one entity affect the other.

As a Post should not be able to change its author, we'll not add cascades to the Post side of the relationship.

However, we'll want a user's posts to be deleted when we delete that user, so we'll add cascades to the User side of the relationship:

```
@Entity
@Table(name = "users")
public class User {
    // ... existing fields ...

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Post> posts = new ArrayList<>();

    // ... rest of the class ...
}
```

Adding the cascade type of ALL here will ensure that a user's posts are deleted when that user is deleted.

It will also allow us to save new a Post if we add it to the posts collection and save the User.

Summary

- The @ManyToOne annotation establishes the post-to-author relationship with a non-nullable foreign key
- Lazy loading of author details optimizes performance for post listing operations
- Helper methods ensure relationship consistency on both sides
- Initialization of collections prevents null pointer exceptions

Configure the Category - Post Relationship

Now we'll configure the relationship between categories and posts, which will allow us to organize blog content into thematic groups.

Understanding Category-Post Relationships

Each post in our blog system must belong to exactly one category, establishing a mandatory one-to-many relationship.

Let's modify our Post entity to include the category relationship:

```
@Entity
@Table(name = "posts")
public class Post {
    // ... existing fields ...

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id", nullable = false)
    private Category category;

    // ... rest of the class ...
}
```

Configuring the Category Side

The Category entity needs to maintain a collection of its associated posts.

Here's how we update the Category entity:

```
@Entity
@Table(name = "categories")
public class Category {
    // ... existing fields ...

    @OneToMany(mappedBy = "category")
    private List<Post> posts = new ArrayList<>();

    // ... rest of the class ...
}
```

Understanding Fetch Types and Collections

The FetchType.LAZY configuration on the post side optimizes performance by loading category details only when explicitly accessed.

The initialization of the posts collection with new ArrayList<>() prevents null pointer exceptions when accessing the collection.

Cascade Considerations

We deliberately avoid cascade operations in this relationship because posts and categories have independent lifecycles.

Deleting a category shouldn't automatically delete its posts, as they might need to be reassigned to another category.

Similarly, deleting a post shouldn't affect its category.

Summary

- Posts must belong to exactly one category, enforced by a non-nullable foreign key
- Cascade operations are intentionally omitted to maintain data integrity
- Relationship validation occurs in the service layer rather than the entity level

Configure the Tag - Post Relationship

> **Source Code**

Introduction

In our previous lessons, we established relationships between posts and both users and categories.

Now we'll implement a many-to-many relationship between posts and tags, enabling flexible content classification.

This relationship will allow each post to have multiple tags and each tag to be associated with multiple posts.

Understanding Many-to-Many Relationships

A many-to-many relationship requires a join table in the database to maintain the associations between entities.

Let's modify our Post entity to include the tag relationship:

```
@Entity
@Table(name = "posts")
public class Post {
    // ... existing fields ...

    @ManyToMany
    @JoinTable(
            name = "post_tags",
            joinColumns = @JoinColumn(name = "post_id"),
            inverseJoinColumns = @JoinColumn(name = "tag_id")
    )
    private Set<Tag> tags = new HashSet<>();

    // ... rest of the class ...
}
```

Configuring the Tag Side

The Tag entity needs to maintain its side of the bidirectional relationship.

Here's how we update the Tag entity:

```
@Entity
@Table(name = "tags")
public class Tag {
    // ... existing fields ...

    @ManyToMany(mappedBy = "tags")
    private Set<Post> posts = new HashSet<>();
```

```
    // ... rest of the class ...
}
```

Cascade Considerations

In a many-to-many relationship, we need to carefully consider cascade operations.

Tags and posts have independent lifecycles - deleting a post shouldn't delete its tags, and deleting a tag shouldn't delete associated posts.

This is why we don't specify cascade operations in our relationship mappings.

Collection Type Selection

We use Set instead of List for both sides of the relationship for several reasons.

Sets prevent duplicate associations between posts and tags.

Set operations are generally more efficient when dealing with large collections.

The order of tags or posts in the collections isn't meaningful for our use case.

Summary

- Many-to-many relationships require a join table to maintain associations between posts and tags
- Using Set instead of List prevents duplicates and improves performance
- Omitting cascade operations preserves independent entity lifecycles
- Bidirectional relationships enable efficient navigation in both directions

Persistence Layer

Create the User Repository

In our previous lessons, we created our core domain entities including the User entity.

Now we'll create a repository interface that will handle data access operations for users.

This repository will serve as the foundation for user authentication and management features we'll build later.

Understanding Spring Data JPA Repositories

Spring Data JPA repositories provide a powerful abstraction that eliminates the need to write basic CRUD operations manually.

The JpaRepository interface offers built-in methods for common operations like saving, finding, and deleting entities.

By extending JpaRepository, our custom repository inherits these methods while maintaining the ability to add specialized queries as needed.

Creating the User Repository

Let's create the UserRepository interface in the repositories package:

```
package com.devtiro.blog.repositories;

import com.devtiro.blog.domain.entities.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.UUID;

@Repository
public interface UserRepository extends JpaRepository<User, UUID> {
}
```

Understanding Repository Configuration

The @Repository annotation marks this interface as a Spring Data repository component.

The generic parameters <User, UUID> tell Spring Data that this repository manages User entities with UUID primary keys.

This configuration aligns with our User entity design from the previous lessons.

Repository Benefits

Spring Data JPA will automatically implement this interface at runtime, providing methods like:

- save(User entity)
- findById(UUID id)
- findAll()

- delete(User entity)
- deleteById(UUID id)

These methods handle the underlying SQL generation and execution, significantly reducing boilerplate code.

Summary

- The UserRepository extends JpaRepository to inherit common CRUD operations
- Spring Data JPA automatically implements the repository interface at runtime
- The repository uses UUID as the ID type to match our entity design
- No custom methods are needed at this stage
- The @Repository annotation integrates the repository with Spring's component scanning

Create the Category Repository

Now we'll implement the CategoryRepository to manage the persistence of blog categories.

This repository will enable efficient category management and form the foundation for content organization features.

Understanding Repository Requirements

Spring Data JPA repositories handle the complex task of persisting and retrieving category data.

The repository needs to work with our Category entity, which we created earlier to represent content classifications.

By leveraging Spring Data JPA's features, we can focus on business logic rather than data access implementation.

Creating the Category Repository

Let's create the CategoryRepository interface in the repositories package:

```
package com.devtiro.blog.repositories;

import com.devtiro.blog.domain.entities.Category;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.UUID;

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

Summary

- The CategoryRepository extends JpaRepository to manage category persistence
- Spring automatically implements CRUD operations for category management
- The repository uses UUID as the ID type to align with our entity design
- Built-in methods support both single and batch operations
- No custom methods are needed at this initial stage

Create the Tag Repository

Now we'll implement the TagRepository to manage the persistence of blog tags.

This repository will enable efficient tag management and support the flexible content classification system we're building.

Understanding Repository Requirements

Spring Data JPA provides powerful abstractions for handling tag persistence and retrieval.

The repository needs to work with our Tag entity, which represents the keywords and phrases used to classify blog content.

We'll leverage Spring Data JPA's built-in features to handle common operations like creating, reading, updating, and deleting tags.

Creating the Tag Repository

Let's create the TagRepository interface in the repositories package:

```
package com.devtiro.blog.repositories;

import com.devtiro.blog.domain.entities.Tag;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.UUID;

@Repository
public interface TagRepository extends JpaRepository<Tag, UUID> {
}
```

Summary

- The TagRepository extends JpaRepository to manage tag persistence
- Spring automatically implements essential CRUD operations
- The repository uses UUID as the ID type to match our entity design
- No custom methods are needed at this initial stage
- Built-in methods support both individual and batch operations

Create the Post Repository

Now we'll implement the PostRepository to manage the persistence of blog posts, which form the core content of our platform.

This repository will enable efficient post management and serve as the foundation for our content management features.

Understanding Post Repository Requirements

Spring Data JPA repositories handle the complex task of persisting and retrieving blog post data.

The repository needs to work with our Post entity, which represents the primary content in our blog platform.

By leveraging Spring Data JPA's features, we can focus on implementing business logic rather than data access details.

Creating the Post Repository

Let's create the PostRepository interface in the repositories package:

```
package com.devtiro.blog.repositories;

import com.devtiro.blog.domain.entities.Post;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.UUID;

@Repository
public interface PostRepository extends JpaRepository<Post, UUID> {
}
```

Summary

- The PostRepository extends JpaRepository to manage post persistence
- Built-in methods handle CRUD operations for posts and their relationships
- Spring automatically implements the repository interface at runtime
- The repository uses UUID as the ID type to match our entity design
- No custom methods are needed at this initial stage

Category Endpoints

List Categories Endpoint Part 1

Building upon our entities and repositories from previous lessons, we'll create our first API endpoint that lists blog categories.

We'll implement this functionality layer by layer, starting with the controller and working through to the service implementation.

This endpoint will serve as a foundation for category management and content organization in our blog platform.

Creating the Controller Package

The controllers package will house all our REST API endpoints.

Let's create the package structure:

```
src/main/java/com/devtiro/blog/controllers/
```

Implementing the Category Controller

The CategoryController will handle all category-related HTTP requests.

```
package com.devtiro.blog.controllers;

import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping("/api/v1/categories")
@RequiredArgsConstructor
public class CategoryController {
    private final CategoryService categoryService;

    @GetMapping
    public ResponseEntity<List<CategoryDto>> listCategories() {
        // TODO
    }
}
```

Creating the DTOs Package

DTOs (Data Transfer Objects) separate our API representation from internal domain models.

Let's create the package:

```
src/main/java/com/devtiro/blog/domain/dtos/
```

Implementing Category DTO

The CategoryDto represents the category data that will be sent to clients.

```
package com.devtiro.blog.domain.dtos;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.UUID;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class CategoryDto {
    private UUID id;
    private String name;
    private long postCount;
}
```

Creating the Services Package

The services package will contain our business logic interfaces and implementations.

Let's create the package:

```
src/main/java/com/devtiro/blog/services/
```

Defining the Category Service Interface

The CategoryService interface defines the contract for category-related operations.

```
package com.devtiro.blog.services;

import com.devtiro.blog.domain.entities.Category;
import java.util.List;

public interface CategoryService {
    /**
     * Lists all categories with their post counts.
     */
    List<Category> listCategories();
}
```

Implementing the Category Service

The CategoryServiceImpl provides the concrete implementation of our service interface.

```
package com.devtiro.blog.services.impl;
```

```
import com.devtiro.blog.domain.entities.Category;
import com.devtiro.blog.repositories.CategoryRepository;
import com.devtiro.blog.services.CategoryService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class CategoryServiceImpl implements CategoryService {
    private final CategoryRepository categoryRepository;

    @Override
    public List<Category> listCategories() {
        return categoryRepository.findAllWithPostCount();
    }
}
```

Extending the Category Repository

We need to add a method to fetch categories with their post counts.

Update the CategoryRepository interface:

```
package com.devtiro.blog.repositories;

import com.devtiro.blog.domain.entities.Category;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

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

@Repository
public interface CategoryRepository extends JpaRepository<Category, UUID> {
    @Query("SELECT c FROM Category c LEFT JOIN FETCH c.posts")
    List<Category> findAllWithPostCount();
}
```

Now we can get a list of Category objects, but how do we convert them into CategoryDto objects? Let's cover that in the next lesson.

Summary

- Created necessary package structure for controllers, DTOs, and services
- Implemented CategoryController with initial listCategories endpoint
- Created CategoryDto for API response representation
- Defined CategoryService interface with listCategories method
- Implemented CategoryServiceImpl with repository integration

List Categories Endpoint Part 2

In our previous lesson, we created the initial structure for listing categories, but we still need to convert our domain entities to DTOs.

We'll implement a MapStruct mapper to handle this conversion efficiently, completing our category listing endpoint.

This will enable us to properly format our category data for API responses while maintaining clean separation between our domain and API layers.

Creating the Mappers Package

MapStruct mappers require their own dedicated package for organization and clarity.

Let's create a new package at src/main/java/com/devtiro/blog/mappers.

In this package, we'll house all our mapper interfaces that handle entity-to-DTO conversions.

Implementing the Category Mapper

The CategoryMapper interface defines how to convert between Category entities and CategoryDto objects:

```
package com.devtiro.blog.mappers;

import com.devtiro.blog.domain.PostStatus;
import com.devtiro.blog.domain.dtos.CategoryDto;
import com.devtiro.blog.domain.entities.Category;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.Named;
import org.mapstruct.ReportingPolicy;

@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface CategoryMapper {

    @Mapping(target = "postCount", source = "posts", qualifiedByName = "calculatePostCount")
    CategoryDto toDto(Category category);

    @Named("calculatePostCount")
    default long calculatePostCount(java.util.Set<com.devtiro.blog.domain.entities.Post> posts) {
        if (posts == null) {
            return 0;
        }
        return posts.stream()
                .filter(post -> PostStatus.PUBLISHED.equals(post.getStatus()))
                .count();
    }
}
```

Completing the Category Controller

Now we can use our mapper to convert entities to DTOs in the controller:

```
@RestController
@RequestMapping("/api/v1/categories")
@RequiredArgsConstructor
public class CategoryController {
    private final CategoryService categoryService;
    private final CategoryMapper categoryMapper;

    @GetMapping
    public ResponseEntity<List<CategoryDto>> listCategories() {
        List<Category> categories = categoryService.listCategories();
        return ResponseEntity.ok(
                categories.stream().map(categoryMapper::toDto).toList()
        );
    }
}
```

Testing the Endpoint

The categories endpoint is now ready for testing.

Navigate to http://localhost:8080/api/v1/categories in your browser or use a tool like Postman.

You should see a JSON response containing a list of categories with their IDs, names, and post counts.

Summary

- Created dedicated mappers package for entity-to-DTO conversion logic
- Implemented CategoryMapper interface using MapStruct for efficient mapping
- Added post count calculation filtering only published posts
- Completed CategoryController implementation with proper DTO conversion
- Verified endpoint functionality through manual testing

Create Category Endpoint

Building on our list categories endpoint from the previous lesson, we'll now implement the ability to create new categories through our API.

This functionality will enable content organization by allowing users to define new category groupings for blog posts.

The create endpoint will validate inputs and prevent duplicate categories, ensuring data integrity in our blog platform.

Request Object Implementation

The create category request requires validation to ensure data quality and consistency.

Let's create the CreateCategoryRequest class to define the structure and validation rules:

```
package com.devtiro.blog.domain.dtos;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Size;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class CreateCategoryRequest {
    @NotBlank(message = "Category name is required")
    @Size(min = 2, max = 50, message = "Category name must be between {min} and {max} characters")
    @Pattern(regexp = "^[\\w\\s-]+$", message = "Category name can only contain letters, numbers, spac
    private String name;
}
```

Mapper Extension

The CategoryMapper interface needs to handle converting request objects to entities.

Let's add the conversion method to our existing mapper:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface CategoryMapper {
    // ... existing methods ...

    Category toEntity(CreateCategoryRequest createCategoryRequest);
}
```

Service Layer Implementation

The service layer must handle the business logic of creating categories while preventing duplicates.

First, let's add the method to our service interface:

```
public interface CategoryService {
    // ... existing methods ...

    Category createCategory(Category category);
}
```

Now we can implement the creation logic in our service implementation:

```
@Service
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class CategoryServiceImpl implements CategoryService {
    private final CategoryRepository categoryRepository;

    // ... existing methods ...

    @Override
    @Transactional
    public Category createCategory(Category category) {
        // Check if category with same name already exists
        if (categoryRepository.existsByNameIgnoreCase(category.getName())) {
            throw new IllegalArgumentException("Category already exists with name: " + category.getNa
        }

        return categoryRepository.save(category);
    }
}
```

Repository Enhancement

The repository needs a method to check for existing categories by name.

Let's add this method to our repository interface:

```
@Repository
public interface CategoryRepository extends JpaRepository<Category, UUID> {
    // ... existing methods ...

    boolean existsByNameIgnoreCase(String name);
}
```

Controller Implementation

Finally, we can implement the controller method to handle category creation requests:

```
@RestController
@RequestMapping("/api/v1/categories")
@RequiredArgsConstructor
```

```
public class CategoryController {
    private final CategoryService categoryService;
    private final CategoryMapper categoryMapper;

    // ... existing methods ...

    @PostMapping
    public ResponseEntity<CategoryDto> createCategory(
            @Valid @RequestBody CreateCategoryRequest createCategoryRequest) {
        Category category = categoryMapper.toEntity(createCategoryRequest);
        Category savedCategory = categoryService.createCategory(category);
        return new ResponseEntity<>(
                categoryMapper.toDto(savedCategory),
                HttpStatus.CREATED
        );
    }
}
```

Summary

- Implemented category creation with input validation and duplicate prevention
- Extended mapper interface to handle request-to-entity conversion
- Added transactional service method for category creation
- Enhanced repository with case-insensitive name existence check
- Created POST endpoint returning HTTP 201 status for successful creation

Error Handling

In previous lessons, we implemented category management endpoints that could potentially encounter various error conditions.

Now we'll create a centralized error handling system to provide consistent, user-friendly error responses across our API.

This error handling system will ensure our frontend can display meaningful messages to users when things go wrong.

Implementing the Error Controller

Spring's @ControllerAdvice annotation enables global exception handling across all our controllers.

Let's create the ErrorController class in our controllers package:

```
package com.devtiro.blog.controllers;

import com.devtiro.blog.domain.dtos.ApiErrorResponse;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestController;

@RestController
@ControllerAdvice
@Slf4j
public class ErrorController {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ApiErrorResponse> handleException(Exception ex) {
        log.error("Caught exception", ex);
        ApiErrorResponse error = ApiErrorResponse.builder()
                .status(HttpStatus.INTERNAL_SERVER_ERROR.value())
                .message("An unexpected error occurred")
                .build();
        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Creating the Error Response DTO

Our API needs a standardized error response format to ensure consistent error handling across all endpoints.

Let's create the ApiErrorResponse class:

```
package com.devtiro.blog.domain.dtos;

import lombok.AllArgsConstructor;
```

```
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.List;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class ApiErrorResponse {
    private int status;
    private String message;
    private List<FieldError> errors;

    @Data
    @Builder
    @NoArgsConstructor
    @AllArgsConstructor
    public static class FieldError {
        private String field;
        private String message;
    }
}
```

Handling Specific Exceptions

Our category creation endpoint throws IllegalArgumentException when a duplicate category is detected.

Let's add specific handling for this case:

```
@RestController
@ControllerAdvice
@Slf4j
public class ErrorController {
    // ... existing methods ...

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<ApiErrorResponse> handleIllegalArgumentException(
            IllegalArgumentException ex) {
        ApiErrorResponse error = ApiErrorResponse.builder()
                .status(HttpStatus.BAD_REQUEST.value())
                .message(ex.getMessage())
                .build();
        return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);
    }
}
```

Summary

- Created centralized error handling using @ControllerAdvice
- Implemented standardized error response format with ApiErrorResponse

- Added specific handling for IllegalArgumentException
- Enabled detailed error logging for debugging
- Ensured consistent HTTP status codes for different error types

Fix Tests

> Source Code

Understanding Test Database Requirements

Spring Boot provides excellent support for using different databases in different environments.

The H2 database is an excellent choice for testing because it runs in memory and doesn't require external setup.

This approach ensures our tests are fast, reliable, and truly isolated from the production environment.

Observing the Current Problem

If we try running tests with PostgreSQL stopped, we'll see failures:

```
docker-compose down
./mvnw test
```

The test failure occurs because our application tries to connect to PostgreSQL, which isn't available:

```
Connection to localhost:5432 refused
```

Configuring the Test Database

Create a new file src/test/resources/application.properties with the following configuration:

```
# Test Database Configuration
spring.datasource.url=jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver

# JPA Configuration
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Understanding the Configuration

The jdbc:h2:mem:testdb URL specifies an in-memory database that's created fresh for each test run.

The create-drop setting ensures our database schema is created before tests and dropped afterward, providing a clean slate for each test.

Spring Boot automatically prioritizes test-specific properties files during test execution.

Verifying the Setup

Now we can run our tests without PostgreSQL running:

```
./mvnw test
```

The tests should now pass, using the H2 database instead of PostgreSQL.

Summary

- H2 provides a lightweight, in-memory database perfect for testing
- Separate application.properties files enable environment-specific configurations
- The create-drop setting ensures a clean test database for each test run
- Independent test databases enable reliable continuous integration pipelines

The Delete Category Endpoint

Building on our category management endpoints from previous lessons, we'll now implement the ability to delete categories that are no longer needed.

This functionality will maintain data integrity by preventing the deletion of categories that still contain posts.

The delete endpoint ensures our blog platform remains organized while protecting against accidental data loss.

Controller Implementation

The category deletion process begins in our controller layer with a new endpoint definition.

```
@RestController
@RequestMapping("/api/v1/categories")
@RequiredArgsConstructor
public class CategoryController {
    private final CategoryService categoryService;

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteCategory(@PathVariable UUID id) {
        categoryService.deleteCategory(id);
        return ResponseEntity.noContent().build();
    }
}
```

Service Interface

The service layer defines the contract for category deletion through a new method.

```
public interface CategoryService {
    // ... existing methods ...

    void deleteCategory(UUID id);
}
```

Service Implementation

The service implementation enforces our business rules, particularly preventing deletion of categories with associated posts.

```
@Service
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class CategoryServiceImpl implements CategoryService {
    private final CategoryRepository categoryRepository;

    @Override
    @Transactional
    public void deleteCategory(UUID id) {
```

```
        Category category = getCategoryById(id);

        // Check if category has associated posts
        if (!category.getPosts().isEmpty()) {
            throw new IllegalStateException(
                    "Cannot delete category: " + category.getName() + ". It has associated posts.");
        }

        categoryRepository.delete(category);
    }

    private Category getCategoryById(UUID id) {
        return categoryRepository.findById(id)
                .orElseThrow(() -> new EntityNotFoundException("Category not found with id: " + id));
    }
}
```

Error Handling Enhancement

Our error controller needs to handle the new IllegalStateException that could occur during category deletion.

```
@RestController
@ControllerAdvice
@Slf4j
public class ErrorController {
    // ... existing methods ...

    @ExceptionHandler(IllegalStateException.class)
    public ResponseEntity<ApiErrorResponse> handleIllegalStateException(
            IllegalStateException ex) {
        ApiErrorResponse error = ApiErrorResponse.builder()
                .status(HttpStatus.CONFLICT.value())
                .message(ex.getMessage())
                .build();
        return new ResponseEntity<>(error, HttpStatus.CONFLICT);
    }
}
```

Summary

- Implemented DELETE endpoint returning 204 No Content on success
- Added service layer validation preventing deletion of categories with posts
- Enhanced error handling with specific handling for IllegalStateException
- Maintained data integrity through proper validation checks
- Followed RESTful conventions for resource deletion

Authentication

Adding Spring Security

In our previous lessons, we implemented category management with error handling.

Now we'll add security to our blog platform, ensuring that certain operations are protected while keeping our content publicly readable.

This will enable us to implement user authentication and protected endpoints in the following lessons.

Understanding Security Requirements

Let's first ensure we have the necessary dependencies in our pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

Configuring Security Settings

By default, Spring Security requires authentication for all endpoints, which we'll need to customize for our public-facing blog content.

Security configuration in Spring Boot 3.x uses a more modern, functional approach compared to earlier versions.

Let's create a new package com.devtiro.blog.config and implement our intitial SecurityConfig class:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exce
        return config.getAuthenticationManager();
```

```
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers(HttpMethod.GET, "/api/v1/posts/**").permitAll()
                        .requestMatchers(HttpMethod.GET, "/api/v1/categories/**").permitAll()
                        .requestMatchers(HttpMethod.GET, "/api/v1/tags/**").permitAll()
                        .anyRequest().authenticated()
                )
                .csrf(csrf -> csrf.disable())
                .sessionManagement(session ->
                        session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                );

        return http.build();
    }
}
```

Summary

- Added Spring Security and JWT dependencies for authentication support
- Created security configuration enabling public access to GET endpoints
- Configured stateless session management for JWT-based authentication
- Implemented password encoding with the DelegatingPasswordEncoder
- Set up authentication manager for future user authentication implementation

Loading Users

In our previous lesson, we added Spring Security to protect our blog platform's endpoints.

Now we'll implement user loading functionality to enable authentication based on our existing User entity.

This implementation will bridge the gap between Spring Security's user management and our domain model.

Creating the UserDetails Class

The BlogUserDetails class adapts our domain User entity to Spring Security's authentication system:

```
package com.devtiro.blog.security;

import com.devtiro.blog.domain.entities.User;
import lombok.Getter;
import lombok.RequiredArgsConstructor;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import java.util.Collection;
import java.util.List;
import java.util.UUID;

@Getter
@RequiredArgsConstructor
public class BlogUserDetails implements UserDetails {

    private final User user;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority("ROLE_USER"));
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

    @Override
    public String getUsername() {
        return user.getEmail();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }
```

```
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    public UUID getId() {
        return user.getId();
    }
}
```

Extending the User Repository

The UserRepository needs a method to find users by their email addresses as that's what the user will use to log in:

```
@Repository
public interface UserRepository extends JpaRepository<User, UUID> {
    Optional<User> findByEmail(String email);
}
```

Implementing the UserDetails Service

The BlogUserDetailsService loads user data and converts it to Spring Security's UserDetails format:

```
package com.devtiro.blog.services.impl;

import com.devtiro.blog.domain.BlogUserDetails;
import com.devtiro.blog.domain.entities.User;
import com.devtiro.blog.repositories.UserRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@RequiredArgsConstructor
public class BlogUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        User user = userRepository.findByEmail(email)
                .orElseThrow(() -> new UsernameNotFoundException("User not found with email: " + emai
```

```
        return new BlogUserDetails(user);
    }
}
```

Handling Authentication Errors

Let's add handling for authentication failures to our ErrorController:

```
@ExceptionHandler(BadCredentialsException.class)
public ResponseEntity<ApiErrorResponse> handleBadCredentialsException(BadCredentialsException ex) {
    ApiErrorResponse error = ApiErrorResponse.builder()
            .status(HttpStatus.UNAUTHORIZED.value())
            .message("Incorrect username or password")
            .build();
    return new ResponseEntity<>(error, HttpStatus.UNAUTHORIZED);
}
```

Summary

- Created BlogUserDetails to adapt our domain user model to Spring Security
- Extended UserRepository with email-based user lookup
- Implemented BlogUserDetailsService to load users during authentication
- Added error handling for authentication failures

Authenticate Endpoint Part 1

Building on our user loading functionality from the previous lesson, we'll now create the authentication endpoint that clients will use to obtain JWT tokens.

This endpoint will validate user credentials and return a token for subsequent authenticated requests.

This foundation enables secure access to protected resources in our blog platform.

Creating the Request and Response DTOs

The authentication process requires dedicated DTOs to handle login requests and responses securely:

```
package com.devtiro.blog.domain.dtos;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class LoginRequest {
    private String email;
    private String password;
}
```

The response DTO includes the JWT token and its expiration time:

```
package com.devtiro.blog.domain.dtos;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class AuthResponse {
    private String token;
    private long expiresIn;
}
```

Defining the Authentication Service

The authentication service interface defines our core authentication operations:

```
package com.devtiro.blog.services;

import org.springframework.security.core.userdetails.UserDetails;

public interface AuthenticationService {
    UserDetails authenticate(String email, String password);
    String generateToken(UserDetails userDetails);
}
```

Implementing the Authentication Controller

The controller handles incoming authentication requests and produces JWT tokens:

```
package com.devtiro.blog.controllers;

import com.devtiro.blog.domain.dtos.AuthResponse;
import com.devtiro.blog.domain.dtos.LoginRequest;
import com.devtiro.blog.services.AuthenticationService;
import lombok.AllArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/v1/auth")
@AllArgsConstructor
public class AuthController {
    private final AuthenticationService authenticationService;

    @PostMapping("/login")
    public ResponseEntity<AuthResponse> login(@RequestBody LoginRequest loginRequest) {
        UserDetails user = authenticationService.authenticate(
            loginRequest.getEmail(),
            loginRequest.getPassword()
        );

        AuthResponse authResponse = AuthResponse.builder()
                .token(authenticationService.generateToken(user))
                .expiresIn(86400) // 24 hours in seconds
                .build();

        return ResponseEntity.ok(authResponse);
    }
}
```

Summary

- Created DTOs for handling login requests and responses
- Defined the authentication service interface with core operations
- Implemented the authentication controller with login endpoint

- Set up token generation workflow
- Established 24-hour token expiration time

Authenticate Endpoint Part 2

In our previous lesson, we set up the authentication controller and service interface for our blog platform.

Now we'll implement the authentication service to handle user verification and JWT token generation.

This implementation will enable secure user authentication and stateless session management across our application.

Implementing the Authentication Service

The AuthenticationServiceImpl class forms the core of our authentication logic, integrating with Spring Security's authentication manager:

```
package com.devtiro.blog.services.impl;

import com.devtiro.blog.services.AuthenticationService;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.security.Keys;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.stereotype.Service;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

@Service
@RequiredArgsConstructor
public class AuthenticationServiceImpl implements AuthenticationService {

    private final AuthenticationManager authenticationManager;
    private final UserDetailsService userDetailsService;

    @Value("${jwt.secret}")
    private String secretKey;

    private final Long jwtExpiryMs = 86400000L;

    @Override
    public UserDetails authenticate(String email, String password) {
        authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(email, password)
```

```
        );
        return userDetailsService.loadUserByUsername(email);
    }

    @Override
    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
        return Jwts.builder()
                .setClaims(claims)
                .setSubject(userDetails.getUsername())
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + jwtExpiryMs))
                .signWith(getSigningKey(), SignatureAlgorithm.HS256)
                .compact();
    }

    @Override
    public UserDetails validateToken(String token) {
        String username = extractUsername(token);
        return userDetailsService.loadUserByUsername(username);
    }

    private String extractUsername(String token) {
        Claims claims = Jwts.parserBuilder()
                .setSigningKey(getSigningKey())
                .build()
                .parseClaimsJws(token)
                .getBody();
        return claims.getSubject();
    }

    private Key getSigningKey() {
        byte[] keyBytes = secretKey.getBytes();
        return Keys.hmacShaKeyFor(keyBytes);
    }

}
```

Update Properties

Let's not forget to add a secret to our properties file. It needs to be at least 32 bytes long!

```
jwt.secret=your-256-bit-secret-key-here-make-it-at-least-32-bytes-long
```

Summary

- Implemented AuthenticationServiceImpl with dependency injection for required services
- Created authentication method using Spring Security's AuthenticationManager
- Added token generation logic using JJWT library
- Configured JWT expiration and secret key through properties

JWT Authentication Filter Part 1

Building on our authentication implementation from previous lessons, we'll now create a filter to validate JWT tokens on protected endpoints.

This filter enables stateless authentication by validating tokens and establishing user context for each request.

Authentication Service Enhancement

The AuthenticationService interface needs to validate incoming tokens:

```
public interface AuthenticationService {
    // ... existing methods ...
    UserDetails validateToken(String token);
}
```

Token Validation Implementation

The AuthenticationServiceImpl must decode and validate JWT tokens:

```
    private String extractUsername(String token) {
        return extractAllClaims(token).getSubject();
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parserBuilder()
                .setSigningKey(getSigningKey())
                .build()
                .parseClaimsJws(token)
                .getBody();
    }

    @Override
    public UserDetails validateToken(String token) {
        String username = extractUsername(token);
        return userDetailsService.loadUserByUsername(username);
    }
```

JWT Filter Implementation

The JwtAuthenticationFilter processes each request to validate authentication:

```
@RequiredArgsConstructor
@Slf4j
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final AuthenticationService authenticationService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
```

```
                                    FilterChain filterChain)
            throws ServletException, IOException {

        try {
            String token = extractToken(request);

            if (token != null) {
                UserDetails userDetails = authenticationService.validateToken(token);

                UsernamePasswordAuthenticationToken authentication =
                        new UsernamePasswordAuthenticationToken(
                                userDetails,
                                null,
                                userDetails.getAuthorities()
                        );

                SecurityContextHolder.getContext().setAuthentication(authentication);

                // Add userId to request attributes for controller access
                if (userDetails instanceof BlogUserDetails) {
                    request.setAttribute("userId", ((BlogUserDetails) userDetails).getId());
                }
            }
        } catch (Exception e) {
            // Don't throw exceptions here - just don't authenticate the request
            log.warn("Received invalid auth token");
        }

        filterChain.doFilter(request, response);
    }

    private String extractToken(HttpServletRequest request) {
        String bearerToken = request.getHeader("Authorization");
        if (bearerToken != null && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring(7);
        }
        return null;
    }
}
```

Enabling the Filter

Update the SecurityConfig to use our new filter:

```
    @Bean
    public JwtAuthenticationFilter jwtAuthenticationFilter(
            AuthenticationService authenticationService) {
        return new JwtAuthenticationFilter(authenticationService);
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http,
                                                   JwtAuthenticationFilter jwtAuthenticationFilter)
            throws Exception {
```

```
        http
                // ... existing configuration ...
                .addFilterBefore(jwtAuthenticationFilter,
                        UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
```

Summary

- Created JWT authentication filter to process token-based authentication
- Extended authentication service with token validation capabilities
- Implemented secure token extraction and validation logic
- Added user context to requests for controller access
- Integrated filter into Spring Security chain for automatic processing

JWT Authentication Filter Part 2

Create the User

First, let's declare the UserDetailsService bean in SecurityConfig, but let's also add a check, creating a user if they don't exist:

```
@Configuration
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService(UserRepository userRepository) {
        BlogUserDetailsService blogUserDetailsService = new BlogUserDetailsService(userRepository);

        String email = "user@test.com";
        userRepository.findByEmail(email).orElseGet(() -> {
            User newUser = User.builder()
                    .name("Test User")
                    .email(email)
                    .password(passwordEncoder().encode("password"))
                    .build();
            return userRepository.save(newUser);
        });

        return blogUserDetailsService;
    }
}
```

With this we should now be able to log in to our application using the credentials user@test.com and password.

Summary

- Declared UserDetailsService bean in SecurityConfig
- Created a default user if one doesn't already exist

Tag Endpoints

List Tags Endpoint

Now we'll create an endpoint to list all tags with their associated post counts, providing a foundation for content discovery.

This endpoint will enable users to explore blog content through tag-based navigation and filtering.

Tag Response Structure

A dedicated response DTO ensures consistent representation of tag data in our API:

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class TagResponse {
    private UUID id;
    private String name;
    private Integer postCount;
}
```

Tag Mapping Configuration

The TagMapper interface handles the conversion between our domain entities and response DTOs:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface TagMapper {

    @Mapping(target = "postCount", source = "posts", qualifiedByName = "calculatePostCount")
    TagResponse toTagResponse(Tag tag);

    List<TagResponse> toTagResponseList(List<Tag> tags);

    @Named("calculatePostCount")
    default Integer calculatePostCount(Set<Post> posts) {
        if (posts == null) {
            return 0;
        }
        return (int) posts.stream()
                .filter(post -> PostStatus.PUBLISHED.equals(post.getStatus()))
                .count();
    }
}
```

Repository Enhancement

The TagRepository needs a method to efficiently fetch tags with their post counts:

```
@Repository
public interface TagRepository extends JpaRepository<Tag, UUID> {
    @Query("SELECT t FROM Tag t LEFT JOIN FETCH t.posts")
```

```
    List<Tag> findAllWithPostCount();
}
```

Service Layer Implementation

The TagService interface defines the contract for tag-related operations:

```
public interface TagService {
    List<Tag> getAllTags();
}
```

The TagServiceImpl implements the tag retrieval logic with proper transaction management:

```
@Service
@RequiredArgsConstructor
public class TagServiceImpl implements TagService {
    private final TagRepository tagRepository;

    @Override
    @Transactional(readOnly = true)
    public List<Tag> getAllTags() {
        return tagRepository.findAllWithPostCount();
    }
}
```

Controller Implementation

The TagController handles HTTP requests for tag operations:

```
@RestController
@RequestMapping("/api/v1/tags")
@RequiredArgsConstructor
public class TagController {
    private final TagService tagService;
    private final TagMapper tagMapper;

    @GetMapping
    public ResponseEntity<List<TagResponse>> getAllTags() {
        List<Tag> tags = tagService.getAllTags();
        return ResponseEntity.ok(tagMapper.toTagResponseList(tags));
    }
}
```

Summary

- Created response DTO and mapper for consistent tag representation
- Implemented service layer with transaction management for tag operations
- Added controller endpoint for retrieving all tags with post counts
- Enhanced repository with efficient post count fetching
- Integrated with existing security configuration for public access

Create Tag Endpoint

Request Structure

The CreateTagsRequest DTO enforces validation rules for tag creation:

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class CreateTagsRequest {

    @NotEmpty(message = "At least one tag name is required")
    @Size(max = 10, message = "Maximum {max} tags allowed")
    private Set<@Size(min = 2, max = 30, message = "Tag name must be between {min} and {max} characte
    @Pattern(regexp = "^[\\w\\s-]+$", message = "Tag name can only contain letters, numbers, spaces, a
            String> names;
}
```

Repository Enhancement

The TagRepository needs specialized methods for efficient tag lookup and creation:

```
@Repository
public interface TagRepository extends JpaRepository<Tag, UUID> {
    Set<Tag> findByNameInIgnoreCase(Set<String> names);
}
```

Service Layer Implementation

The TagService interface expands to include tag creation capabilities:

```
public interface TagService {
    List<Tag> createTags(Set<String> tagNames);
}
```

The TagServiceImpl provides the creation logic with duplicate handling:

```
@Transactional
@Override
public List<Tag> createTags(Set<String> tagNames) {
    List<Tag> existingTags = tagRepository.findByNameIn(tagNames);

    Set<String> existingTagNames = existingTags.stream()
            .map(Tag::getName)
            .collect(Collectors.toSet());

    List<Tag> newTags = tagNames.stream()
            .filter(name -> !existingTagNames.contains(name))
            .map(name -> Tag.builder()
                    .name(name)
                    .posts(new HashSet<>())
```

```
                    .build())
            .toList();

    List<Tag> savedTags = new ArrayList<>();
    if(!newTags.isEmpty()) {
        savedTags = tagRepository.saveAll(newTags);
    }

    savedTags.addAll(existingTags);

    return savedTags;
}
```

Controller Implementation

The TagController handles the HTTP POST requests for tag creation:

```
@RestController
@RequestMapping("/api/v1/tags")
@RequiredArgsConstructor
public class TagController {
    private final TagService tagService;
    private final TagMapper tagMapper;

    @PostMapping
    public ResponseEntity<List<TagDto>> createTags(@RequestBody CreateTagsRequest createTagsRequest) {
        List<Tag> savedTags = tagService.createTags(createTagsRequest.getNames());
        List<TagDto> createdTagRespons = savedTags.stream().map(tagMapper::toTagResponse).toList();
        return new ResponseEntity<>(
                createdTagRespons,
                HttpStatus.CREATED
        );
    }
}
```

Summary

- Implemented tag creation with validation and duplicate prevention
- Created transactional service method for atomic tag operations
- Enhanced repository with efficient name-based tag lookup

Delete Tag Endpoint

This endpoint will enable content organizers to remove unused tags while preventing the deletion of tags that are still associated with posts.

Service Layer Enhancement

The tag deletion process requires careful validation to prevent orphaned content:

```
public interface TagService {
    // ...
    void deleteTag(UUID id);
}
```

```
    @Transactional
    @Override
    public void deleteTag(UUID id) {
        tagRepository.findById(id).ifPresent(tag -> {
            if(!tag.getPosts().isEmpty()) {
                throw new IllegalStateException("Cannot delete tag with posts");
            }
            tagRepository.deleteById(id);
        });
    }
```

Controller Implementation

The TagController handles HTTP DELETE requests with proper response codes:

```
@DeleteMapping(path = "/{id}")
public ResponseEntity<Void> deleteTag(@PathVariable UUID id) {
    tagService.deleteTag(id);
    return ResponseEntity.noContent().build();
}
```

Summary

- Implemented tag deletion with validation
- Created DELETE endpoint returning appropriate HTTP status codes

Post Endpoints

List Posts Endpoint

This endpoint will support filtering by categories and tags while ensuring only published content is accessible to users.

The feature enables content discovery and forms the foundation for our blog's public interface.

Data Transfer Object Creation

The PostDto encapsulates all necessary post information for API responses:

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class PostDto {
    private UUID id;
    private String title;
    private String content;
    private AuthorDto author;
    private CategoryDto category;
    private Set<TagDto> tags;
    private Integer readingTime;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
    private PostStatus status;
}
```

> ⚠ **Warning**
> I use postStatus as the instance variable for this class, which causes some troubleshooting later! To avoid this use status.

Repository Implementation

The PostRepository provides flexible query methods for post retrieval:

```
@Repository
public interface PostRepository extends JpaRepository<Post, UUID> {
    List<Post> findAllByStatusAndCategoryAndTagsContaining(PostStatus status, Category category, Tag t
    List<Post> findAllByStatusAndCategory(PostStatus status, Category category);
    List<Post> findAllByStatusAndTagsContaining(PostStatus status, Tag tag);
    List<Post> findAllByStatus(PostStatus status);
}
```

Service Layer Definition

The PostService interface defines our core post retrieval functionality:

```
public interface PostService {
    List<Post> getAllPosts(UUID categoryId, UUID tagId);
}
```

Service Implementation

The PostServiceImpl orchestrates post retrieval with proper filtering:

```
@Service
@RequiredArgsConstructor
public class PostServiceImpl implements PostService {

    private final PostRepository postRepository;
    private final CategoryService categoryService;
    private final TagService tagService;

    @Override
    @Transactional(readOnly = true)
    public List<Post> getAllPosts(UUID categoryId, UUID tagId) {
        if(categoryId != null && tagId != null) {
            Category category = categoryService.getCategoryById(categoryId);
            Tag tag = tagService.getTagById(tagId);
            return postRepository.findAllByStatusAndCategoryAndTagsContaining(
                    PostStatus.PUBLISHED,
                    category,
                    tag
            );
        }

        if(categoryId != null) {
            Category category = categoryService.getCategoryById(categoryId);
            return postRepository.findAllByStatusAndCategory(
                    PostStatus.PUBLISHED,
                    category
            );
        }

        if(tagId != null) {
            Tag tag = tagService.getTagById(tagId);
            return postRepository.findAllByStatusAndTagsContaining(
                    PostStatus.PUBLISHED,
                    tag
            );
        }

        return postRepository.findAllByStatus(PostStatus.PUBLISHED);
    }
}
```

Mapper Configuration

The PostMapper interface handles entity-to-DTO conversion:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface PostMapper {
    @Mapping(target = "author", source = "author")
    @Mapping(target = "category", source = "category")
    @Mapping(target = "tags", source = "tags")
```

```
    PostDto toDto(Post post);
}
```

Post Controller Setup

The PostController serves as the entry point for all post-related operations:

```
@RestController
@RequestMapping(path = "/api/v1/posts")
@RequiredArgsConstructor
public class PostController {

    private final PostService postService;
    private final PostMapper postMapper;
    private final UserService userService;

    @GetMapping
    public ResponseEntity<List<PostDto>> getAllPosts(
            @RequestParam(required = false) UUID categoryId,
            @RequestParam(required = false) UUID tagId) {
        List<Post> posts = postService.getAllPosts(categoryId, tagId);
        List<PostDto> postDtos = posts.stream().map(postMapper::toDto).toList();
        return ResponseEntity.ok(postDtos);
    }

}
```

Exception Handling

Let's also add a block to handle EntityNotFoundException in the ErrorController:

```
@RestController
@ControllerAdvice
@Slf4j
public class ErrorController {
    // ...
    @ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity<ApiErrorResponse> handleEntityNotFoundException(EntityNotFoundException ex)
        ApiErrorResponse error = ApiErrorResponse.builder()
                .status(HttpStatus.NOT_FOUND.value())
                .message(ex.getMessage())
                .build();
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }
}
```

Summary

- Created flexible post retrieval endpoint with optional category and tag filtering
- Implemented comprehensive DTO structure for post representation
- Added repository methods for efficient post querying
- Established service layer with proper transaction management
- Configured MapStruct mapper for clean entity-to-DTO conversion

List Draft Posts Endpoint

This feature enables content creators to manage their works in progress effectively through the blog platform.

Repository Query Definition

The PostRepository requires a specialized query method to fetch user-specific drafts:

```
@Repository
public interface PostRepository extends JpaRepository<Post, UUID> {
    // ... existing methods ...
    List<Post> findAllByAuthorAndStatus(User author, PostStatus status);
}
```

Service Interface Enhancement

The PostService interface expands to support draft post retrieval:

```
public interface PostService {
    // ...
    List<Post> getDraftPosts(User user);
}
```

Service Implementation

The PostServiceImpl implements draft retrieval with proper user context:

```
@Service
@RequiredArgsConstructor
public class PostServiceImpl implements PostService {
    //...
    private final PostRepository postRepository;


    @Override
    public List<Post> getDraftPosts(User user) {
        return postRepository.findAllByAuthorAndStatus(user, PostStatus.DRAFT);
    }
}
```

Post Controller Method Declaration

The draft posts endpoint requires careful handling of user authentication to maintain content security:

```
@GetMapping(path = "/drafts")
public ResponseEntity<List<PostDto>> getDrafts(@RequestAttribute UUID userId) {
    User loggedInUser = userService.getUserById(userId);
    List<Post> draftPosts = postService.getDraftPosts(loggedInUser);
    List<PostDto> postDtos = draftPosts.stream().map(postMapper::toDto).toList();
```

```
    return ResponseEntity.ok(postDtos);
}
```

Security Filter Chain Updates

Let's update the SecurityFilterChain to secure the drafts endpoint:

```
 // ...
public SecurityFilterChain securityFilterChain(
        HttpSecurity http,
        JwtAuthenticationFilter jwtAuthenticationFilter) throws Exception {
    http
            .authorizeHttpRequests(auth -> auth
                    .requestMatchers(HttpMethod.POST, "/api/v1/auth/login").permitAll()
                    .requestMatchers(HttpMethod.GET, "/api/v1/posts/drafts").authenticated()
                    .requestMatchers(HttpMethod.GET, "/api/v1/posts/**").permitAll()
                    .requestMatchers(HttpMethod.GET, "/api/v1/categories/**").permitAll()
                    .requestMatchers(HttpMethod.GET, "/api/v1/tags/**").permitAll()
                    .anyRequest().authenticated()
            )
            .csrf(csrf -> csrf.disable())
            .sessionManagement(session ->
                    session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            ).addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

Summary

- Implemented secure draft post retrieval with user context validation
- Added service layer support for user-specific draft access
- Created repository query method for efficient draft filtering

Create Post Endpoint

This endpoint will enable writers to compose and save both draft and published content with proper categorization and tagging.

Request Data Transfer Object

The CreatePostRequestDto class defines the structure and validation rules for incoming post creation requests:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class CreatePostRequestDto {

    @NotBlank(message = "Title is required")
    @Size(min = 3, max = 200, message = "Title must be between {min} and {max} characters")
    private String title;

    @NotBlank(message = "Content is required")
    @Size(min = 10, max = 50000, message = "Content must be between {min} and {max} characters")
    private String content;

    @NotNull(message = "Category ID is required")
    private UUID categoryId;

    @Builder.Default
    @Size(max = 10, message = "Maximum {max} tags allowed")
    private Set<UUID> tagIds = new HashSet<>();

    @NotNull(message = "Status is required")
    private PostStatus status;
}
```

Domain Request Model

The CreatePostRequest class serves as our internal domain model for post creation:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class CreatePostRequest {
    private String title;

    private String content;

    private UUID categoryId;

    @Builder.Default
    private Set<UUID> tagIds = new HashSet<>();
```

```
    private PostStatus status;
}
```

Mapper Configuration

The PostMapper interface handles the conversion between DTOs and domain objects:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface PostMapper {
    // ...
    CreatePostRequest toCreatePostRequest(CreatePostRequestDto dto);
}
```

Service Layer Definition

The PostService interface defines our post creation contract:

```
public interface PostService {
    // ...
    Post createPost(User user, CreatePostRequest createPostRequest);
}
```

Service Implementation

The PostServiceImpl handles the core logic of post creation:

```
@Override
@Transactional
public Post createPost(User user, CreatePostRequest createPostRequest) {
    Post newPost = new Post();
    newPost.setTitle(createPostRequest.getTitle());
    newPost.setContent(createPostRequest.getContent());
    newPost.setStatus(createPostRequest.getStatus());
    newPost.setAuthor(user);
    newPost.setReadingTime(calculateReadingTime(createPostRequest.getContent()));

    Category category = categoryService.getCategoryById(createPostRequest.getCategoryId());
    newPost.setCategory(category);

    Set<UUID> tagIds = createPostRequest.getTagIds();
    List<Tag> tags = tagService.getTagByIds(tagIds);
    newPost.setTags(new HashSet<>(tags));

    return postRepository.save(newPost);
}
```

Controller Implementation

The PostController handles HTTP POST requests for creating new posts:

```
@PostMapping
public ResponseEntity<PostDto> createPost(
```

```
        @Valid @RequestBody CreatePostRequestDto createPostRequestDto,
        @RequestAttribute UUID userId) {
    User loggedInUser = userService.getUserById(userId);
    CreatePostRequest createPostRequest = postMapper.toCreatePostRequest(createPostRequestDto);
    Post createdPost = postService.createPost(loggedInUser, createPostRequest);
    PostDto createdPostDto = postMapper.toDto(createdPost);
    return new ResponseEntity<>(createdPostDto, HttpStatus.CREATED);
}
```

Summary

- Implemented post creation with validation
- Added reading time calculation

Update Post Endpoint

This endpoint will enable content creators to modify post content, change categories and tags, and toggle between draft and published states.

Update Request DTO

The UpdatePostRequestDto defines the structure and validation rules for post updates:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class UpdatePostRequestDto {

    @NotNull(message = "Post ID is required")
    private UUID id;

    @NotBlank(message = "Title is required")
    @Size(min = 3, max = 200, message = "Title must be between {min} and {max} characters")
    private String title;

    @NotBlank(message = "Content is required")
    @Size(min = 10, max = 50000, message = "Content must be between {min} and {max} characters")
    private String content;

    @NotNull(message = "Category ID is required")
    private UUID categoryId;

    @Builder.Default
    @Size(max = 10, message = "Maximum {max} tags allowed")
    private Set<UUID> tagIds = new HashSet<>();

    @NotNull(message = "Status is required")
    private PostStatus status;
}
```

Domain Model Implementation

The UpdatePostRequest class serves as our internal model for post updates:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class UpdatePostRequest {

    private UUID id;

    private String title;

    private String content;
```

```
    private UUID categoryId;

    @Builder.Default
    private Set<UUID> tagIds = new HashSet<>();

    private PostStatus status;
}
```

Mapper Implementation

Let's implement the necessary methods on the PostMapper interface:

```
@Mapper(componentModel = "spring", unmappedTargetPolicy = ReportingPolicy.IGNORE)
public interface PostMapper {
    // ...
    UpdatePostRequest toUpdatePostRequest(UpdatePostRequestDto dto);
}
```

Service Layer Enhancement

The PostService interface expands to include post update capability:

```
public interface PostService {
    // ... existing methods ...
    Post updatePost(UUID id, UpdatePostRequest updatePostRequest);
}
```

Update Logic Implementation

The PostServiceImpl handles the complex logic of updating posts with proper validation:

```
@Override
@Transactional
public Post updatePost(UUID id, UpdatePostRequest updatePostRequest) {
    Post existingPost = postRepository.findById(id)
            .orElseThrow(() -> new EntityNotFoundException("Post does not exist with id " + id));

    existingPost.setTitle(updatePostRequest.getTitle());
    String postContent = updatePostRequest.getContent();
    existingPost.setContent(postContent);
    existingPost.setStatus(updatePostRequest.getStatus());
    existingPost.setReadingTime(calculateReadingTime(postContent));

    UUID updatePostRequestCategoryId = updatePostRequest.getCategoryId();
    if(!existingPost.getCategory().getId().equals(updatePostRequestCategoryId)) {
        Category newCategory = categoryService.getCategoryById(updatePostRequestCategoryId);
        existingPost.setCategory(newCategory);
    }

    Set<UUID> existingTagIds = existingPost.getTags().stream().map(Tag::getId).collect(Collectors.toSe
    Set<UUID> updatePostRequestTagIds = updatePostRequest.getTagIds();
    if(!existingTagIds.equals(updatePostRequestTagIds)) {
```

```
        List<Tag> newTags = tagService.getTagByIds(updatePostRequestTagIds);
        existingPost.setTags(new HashSet<>(newTags));
    }

    return postRepository.save(existingPost);
}
```

Controller Implementation

The PostController handles the HTTP PUT requests for post updates:

```
@PutMapping(path = "/{id}")
public ResponseEntity<PostDto> updatePost(
        @PathVariable UUID id,
        @Valid @RequestBody UpdatePostRequestDto updatePostRequestDto) {
    UpdatePostRequest updatePostRequest = postMapper.toUpdatePostRequest(updatePostRequestDto);
    Post updatedPost = postService.updatePost(id, updatePostRequest);
    PostDto updatedPostDto = postMapper.toDto(updatedPost);
    return ResponseEntity.ok(updatedPostDto);
}
```

Summary

- Implemented post update functionality with validation
- Created separate DTOs and domain models for clean separation of concerns

Get Post Endpoint

This endpoint will enable allow us to retrieve a post, in order to read the post, update or delete it.

Service Interface Definition

The PostService interface needs to define the contract for post retrieval:

```
public interface PostService {
    // ...
    Post getPost(UUID id);
}
```

Service Implementation

Now let's implement getPost in PostServiceImpl:

```
@Override
public Post getPost(UUID id) {
    return postRepository.findById(id)
            .orElseThrow(() -> new EntityNotFoundException("Post does not exist with ID " + id));
}
```

Controller Method Declaration

Finally we declare the controller endpoint:

```
@GetMapping(path = "/{id}")
public ResponseEntity<PostDto> getPost(
        @PathVariable UUID id
) {
    Post post = postService.getPost(id);
    PostDto postDto = postMapper.toDto(post);
    return ResponseEntity.ok(postDto);
}
```

Summary

- Implemented post retrieval

Delete Post Endpoint

This endpoint will enable content creators to permanently delete posts that are no longer needed, whether they are drafts or published content.

The deletion process ensures proper cleanup of relationships while maintaining data integrity across the platform.

Service Interface Definition

The PostService interface needs to define the contract for post deletion:

```
public interface PostService {
    // ... existing methods ...
    void deletePost(UUID id);
}
```

Service Implementation

The PostServiceImpl handles the core logic of post deletion with proper validation:

```
@Transactional
@Override
public void deletePost(UUID id) {
    Post post = getPost(id);
    postRepository.delete(post);
}
```

> ⚠ **Warning**
> The video misses out the @Transactional annotation on deletePost. Please note that this method would benefit @Transactional, so it's worth adding!

Controller Method Declaration

The post deletion endpoint requires user authentication and proper error handling to maintain security:

```
@DeleteMapping(path = "/{id}")
public ResponseEntity<Void> deletePost(@PathVariable UUID id) {
    postService.deletePost(id);
    return ResponseEntity.noContent().build();
}
```

Summary

- Implemented post deletion endpoint
- Added service layer support for delete operations

Review the Build

Great work getting this far! You've built a working blog platform from scratch.

Let's take a moment to review what we've covered and talk about ways to make it even better.

What We've Built

Our blog platform includes the core features needed for content management:

- A category system for organizing posts
- A flexible tagging system that supports multiple tags per post
- Post creation with draft and published states
- Filtering functionality for both categories and tags

Next Steps

While our platform has the basics, there are some key areas to improve on:

Security

The authentication system could use refresh tokens to improve user session handling.

Adding CSRF protection would help prevent cross-site request forgery attacks.

User Experience

The validation system works at a basic level.

Better error messages would help users understand and fix issues more quickly.

Under the Hood

Some key improvements would include:

- Adding comprehensive tests
- Optimizing database queries for better performance
- Refining features like the reading time calculation

Summary

- You've built a solid foundation with the essential features of a blog platform
- The next steps focus on moving from basic functionality to production readiness