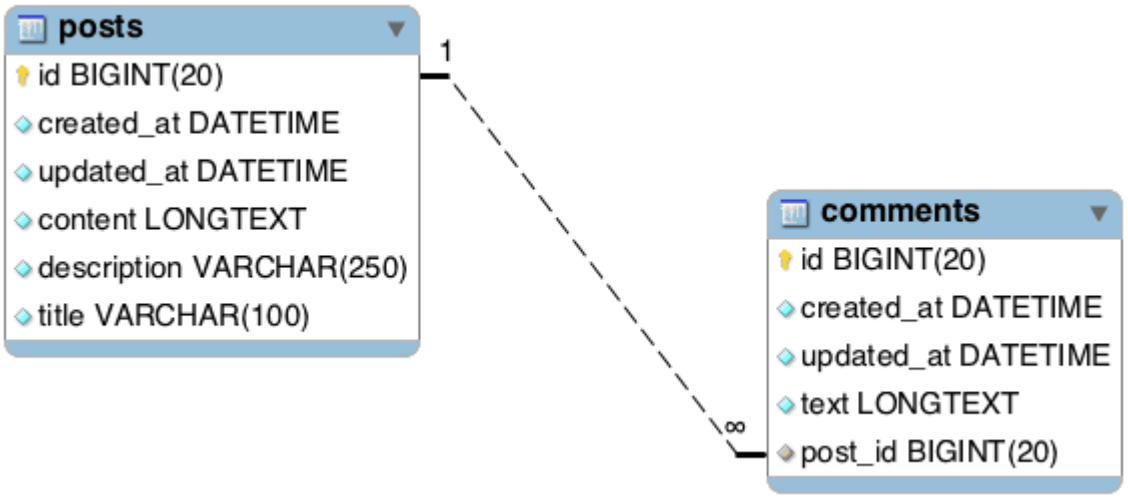
One-to-Many mapping relationship using JPA and Spring Boot

How to map a one-to-many database relationship at the object level using JPA and Hibernate.

Consider the following two tables - `posts` and `comments` of a Blog database schema where the `posts` table has a one-to-many relationship with the `comments` table -



We'll create a project from scratch and learn how to go about implementing such one-to-many relationship at the object level using JPA and hibernate.

We'll also write REST APIs to perform CRUD operations on the entities so that you fully understand how to actually use these relationships in the real world.

Creating the Project

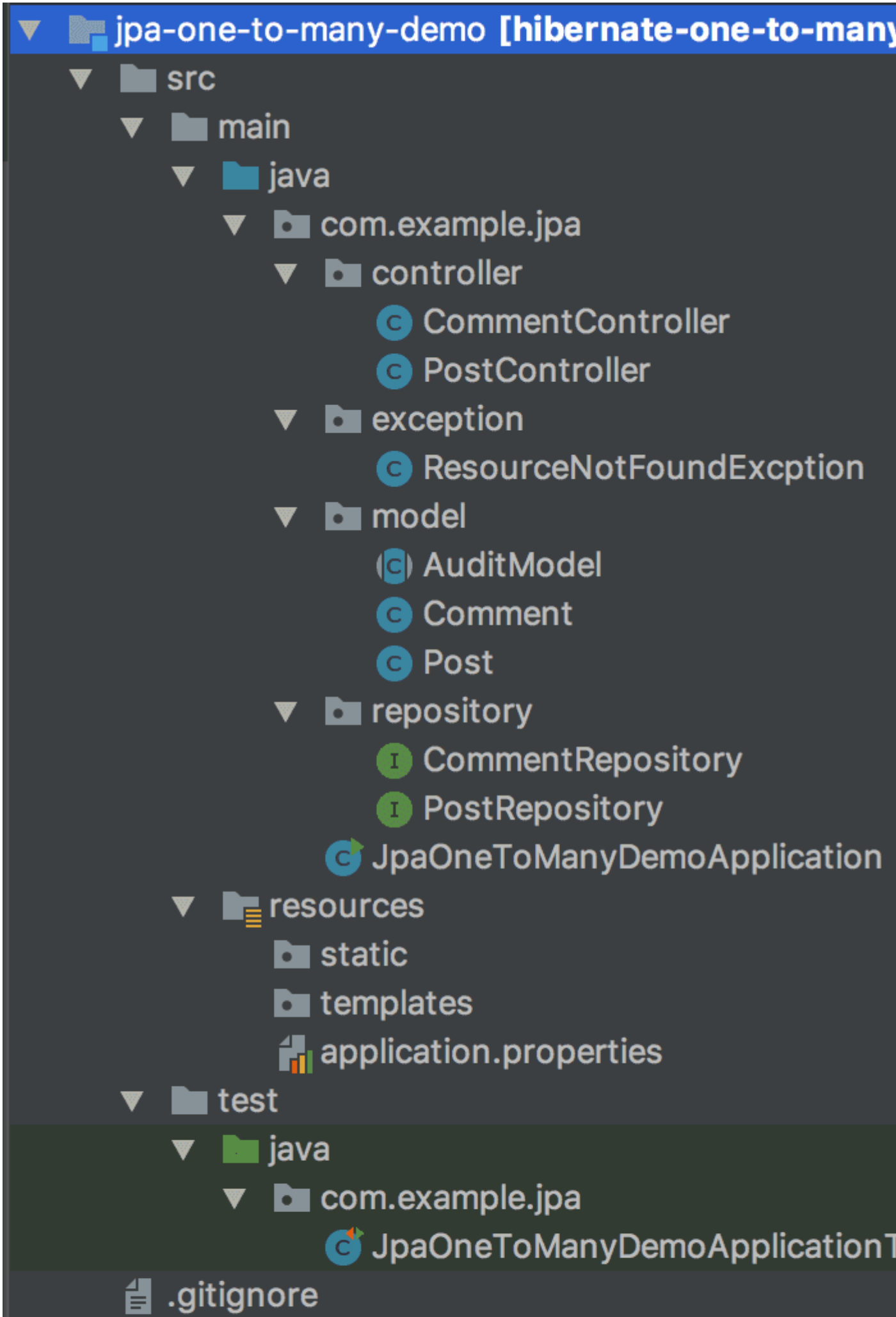
If you have Spring Boot CLI installed, then you can type the following command in your terminal to generate the project -

```
spring init -n=jpa-one-to-many-demo -d=web,jpa,mysql --
package-name=com.example.jpa jpa-one-to-many-demo
```

Alternatively, You can generate the project from Spring Initializr web tool by following the instructions below -

1. Go to http://start.spring.io
2. Enter **Artifact** as “jpa-one-to-many-demo”
3. Click **Options** dropdown to see all the options related to project metadata.
4. Change **Package Name** to “com.example.jpa”
5. Select **Web**, **JPA** and **Mysql** dependencies.
6. Click **Generate** to download the project.

Following is the directory structure of the project for your reference -


jpa-one-to-many-demo [hibernate-one-to-many
src
main
java
com.example.jpa
controller
CommentController
PostController
exception
ResourceNotFoundExcption
model
AuditModel
Comment
Post
repository
CommentRepository
PostRepository
JpaOneToManyDemoApplication
resources
static
templates
application.properties
test
java
com.example.jpa
JpaOneToManyDemoApplicationT
.gitignore

”*Your bootstrapped project won’t have* `model`, `controller`, `repository` *and* `exception` *packages, and all the classes inside these packages at this point. We’ll create them shortly.*”

Configuring the Database and Logging

Since we’re using MySQL as our database, we need to configure the database URL, username, and password so that Spring can establish a connection with the database on startup.
Open `src/main/resources/application.properties` file and add the following properties to it -

```
# DATASOURCE (DataSourceAutoConfiguration &
DataSourceProperties)

spring.datasource.url=jdbc:mysql://localhost:3306/jpa_one
_to_many_demo?useSSL=false&serverTimezone=UTC&useLegacyDa
tetimeCode=false

spring.datasource.username=root

spring.datasource.password=root


# Hibernate


# The SQL dialect makes Hibernate generate better SQL for
the chosen database

spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQL5InnoDBDialect


# Hibernate ddl auto (create, create-drop, validate,
update)

spring.jpa.hibernate.ddl-auto = update


logging.level.org.hibernate.SQL=DEBUG
```

```
logging.level.org.hibernate.type=TRACE
```

Don't forget to change the `spring.datasource.username` and `spring.datasource.password` as per your MySQL installation. Also, create a database named `jpa_one_to_many_demo` in MySQL before proceeding to the next section.

You don't need to create any tables. The tables will automatically be created by hibernate from the `Post` and `Comment` entities that we will define shortly. This is made possible by the property `spring.jpa.hibernate.ddl-auto = update`.

We have also specified the log levels for hibernate so that we can debug all the SQL statements and learn what hibernate does under the hood.

The best way to model a one-to-many relationship in hibernate

I have been working with hibernate for quite some time and I've realized that **the best way to model a one-to-many relationship is to use just `@ManyToOne` annotation on the child entity**.

The second best way is to define a bidirectional association with a `@OneToMany` annotation on the parent side of the relationship and a `@ManyToOne` annotation on the child side of the relationship. The bidirectional mapping has its pros and cons. I'll demonstrate these pros and cons in the second section of this article. I'll also tell you when a bidirectional mapping is a good fit.

But let's first model our one-to-many relationship in the best way possible.

Defining the Domain Models

In this section, we'll define the domain models of our application - `Post` and `Comment`.

Note that both `Post` and `Comment` entities contain some common auditing related fields like `created_at` and `updated_at`.

We'll abstract out these common fields in a separate class called `AuditModel` and extend this class in the `Post` and `Comment` entities.

We'll also use Spring Boot's JPA Auditing feature to automatically populate the `created_at` and `updated_at` fields while persisting the entities.

1. AuditModel

```
package com.example.jpa.model;


import
com.fasterxml.jackson.annotation.JsonIgnoreProperties;

import org.springframework.data.annotation.CreatedDate;

import
org.springframework.data.annotation.LastModifiedDate;

import
org.springframework.data.jpa.domain.support.AuditingEntit
yListener;


import javax.persistence.*;

import javax.validation.constraints.NotNull;

import java.io.Serializable;

import java.util.Date;


@MappedSuperclass

@EntityListeners(AuditingEntityListener.class)

@JsonIgnoreProperties(

        value = {"createdAt", "updatedAt"},

        allowGetters = true
```

```
)
public abstract class AuditModel implements Serializable
{
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_at", nullable = false,
updatable = false)
    @CreatedDate
    private Date createdAt;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "updated_at", nullable = false)
    @LastModifiedDate
    private Date updatedAt;

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
        this.createdAt = createdAt;
    }

    public Date getUpdatedAt() {
        return updatedAt;
    }
```

```
    public void setUpdatedAt(Date updatedAt) {
        this.updatedAt = updatedAt;
    }
}
```

In the above class, we're using Spring Boot's `AuditingEntityListener` to automatically populate the `createdAt` and `updatedAt` fields.

Enabling JPA Auditing

To enable JPA Auditing, you'll need to add `@EnableJpaAuditing` annotation to one of your configuration classes. Open the main class `JpaOneToManyDemoApplication.java` and add the `@EnableJpaAuditing` to the main class like so -

```
package com.example.jpa;


import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplicat
ion;

import
org.springframework.data.jpa.repository.config.EnableJpaA
uditing;


@SpringBootApplication

@EnableJpaAuditing

public class JpaOneToManyDemoApplication {

    public static void main(String[] args) {
```

```
SpringApplication.run(JpaOneToManyDemoApplication.class,
args);
    }
}
```

2. Post model

```
package com.example.jpa.model;

import javax.persistence.*;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Entity
@Table(name = "posts")
public class Post extends AuditModel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Size(max = 100)
    @Column(unique = true)
    private String title;

    @NotNull
```

```
    @Size(max = 250)
    private String description;

    @NotNull
    @Lob
    private String content;

    // Getters and Setters (Omitted for brevity)
}
```

3. Comment model

```
package com.example.jpa.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import javax.persistence.*;
import javax.validation.constraints.NotNull;
import org.hibernate.annotations.OnDelete;
import org.hibernate.annotations.OnDeleteAction;

@Entity
@Table(name = "comments")
public class Comment extends AuditModel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    @NotNull
    @Lob
    private String text;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "post_id", nullable = false)
    @OnDelete(action = OnDeleteAction.CASCADE)
    @JsonIgnore
    private Post post;

    // Getters and Setters (Omitted for brevity)
}
```

The `Comment` model contains the `@ManyToOne` annotation to declare that it has a many-to-one relationship with the `Post` entity. It also uses the `@JoinColumn` annotation to declare the foreign key column.

Defining the Repositories

Next, We'll define the repositories for accessing the data from the database. Create a new package called `repository` inside `com.example.jpa` package and add the following interfaces inside the `repository` package -

1. PostRepository

```
package com.example.jpa.repository;

import com.example.jpa.model.Post;
```

```
import
org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;


@Repository

public interface PostRepository extends
JpaRepository<Post, Long> {


}
```

2. CommentRepository

```
package com.example.jpa.repository;


import com.example.jpa.model.Comment;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import
org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;


import java.util.Optional;


@Repository

public interface CommentRepository extends
JpaRepository<Comment, Long> {

    Page<Comment> findByPostId(Long postId, Pageable
pageable);
```

```
    Optional<Comment> findByIdAndPostId(Long id, Long
postId);

}
```

Writing the REST APIs to perform CRUD operations on the entities

Let's now write the REST APIs to perform CRUD operations on `Post` and `Comment` entities.

All the following controller classes are define inside `com.example.jpa.controller` package.

1. PostController (APIs to create, retrieve, update, and delete Posts)

```
package com.example.jpa.controller;


import
com.example.jpa.exception.ResourceNotFoundException;

import com.example.jpa.model.Post;

import com.example.jpa.repository.PostRepository;

import
org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.*;


import javax.validation.Valid;
```

```java
@RestController
public class PostController {

    @Autowired
    private PostRepository postRepository;

    @GetMapping("/posts")
    public Page<Post> getAllPosts(Pageable pageable) {
        return postRepository.findAll(pageable);
    }

    @PostMapping("/posts")
    public Post createPost(@Valid @RequestBody Post post)
{
        return postRepository.save(post);
    }

    @PutMapping("/posts/{postId}")
    public Post updatePost(@PathVariable Long postId,
@Valid @RequestBody Post postRequest) {
        return postRepository.findById(postId).map(post -
> {
            post.setTitle(postRequest.getTitle());

post.setDescription(postRequest.getDescription());
            post.setContent(postRequest.getContent());
            return postRepository.save(post);
```

```
        }).orElseThrow(() -> new
ResourceNotFoundException("PostId " + postId + " not
found"));

    }




    @DeleteMapping("/posts/{postId}")

    public ResponseEntity<?> deletePost(@PathVariable
Long postId) {

        return postRepository.findById(postId).map(post -
> {

            postRepository.delete(post);

            return ResponseEntity.ok().build();

        }).orElseThrow(() -> new
ResourceNotFoundException("PostId " + postId + " not
found"));

    }


}
```

2. CommentController (APIs to create, retrieve, update, and delete Comments)

```
package com.example.jpa.controller;


import
com.example.jpa.exception.ResourceNotFoundException;

import com.example.jpa.model.Comment;

import com.example.jpa.repository.CommentRepository;
```

```
import com.example.jpa.repository.PostRepository;

import
org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.*;


import javax.validation.Valid;


@RestController

public class CommentController {


    @Autowired

    private CommentRepository commentRepository;


    @Autowired

    private PostRepository postRepository;


    @GetMapping("/posts/{postId}/comments")

    public Page<Comment>
getAllCommentsByPostId(@PathVariable (value = "postId")
Long postId,

                                                Pageable
pageable) {

        return commentRepository.findByPostId(postId,
pageable);

    }
```

```
    @PostMapping("/posts/{postId}/comments")
    public Comment createComment(@PathVariable (value =
"postId") Long postId,
                                 @Valid @RequestBody
Comment comment) {
        return postRepository.findById(postId).map(post -
> {
            comment.setPost(post);
            return commentRepository.save(comment);
        }).orElseThrow(() -> new
ResourceNotFoundException("PostId " + postId + " not
found"));
    }

    @PutMapping("/posts/{postId}/comments/{commentId}")
    public Comment updateComment(@PathVariable (value =
"postId") Long postId,
                                 @PathVariable (value =
"commentId") Long commentId,
                                 @Valid @RequestBody
Comment commentRequest) {
        if(!postRepository.existsById(postId)) {
            throw new ResourceNotFoundException("PostId "
+ postId + " not found");
        }

        return
commentRepository.findById(commentId).map(comment -> {
            comment.setText(commentRequest.getText());
```

```
            return commentRepository.save(comment);

        }).orElseThrow(() -> new
ResourceNotFoundException("CommentId " + commentId + "not
found"));

    }



@DeleteMapping("/posts/{postId}/comments/{commentId}")

    public ResponseEntity<?> deleteComment(@PathVariable
(value = "postId") Long postId,

                                   @PathVariable (value =
"commentId") Long commentId) {

        return
commentRepository.findByIdAndPostId(commentId,
postId).map(comment -> {

            commentRepository.delete(comment);

            return ResponseEntity.ok().build();

        }).orElseThrow(() -> new
ResourceNotFoundException("Comment not found with id " +
commentId + " and postId " + postId));

    }

}
```

The ResourceNotFoundException Class

Both the Post and Comment Rest APIs throw `ResourceNotFoundException` when a post or comment could not be found. Following is the definition of the `ResourceNotFoundException`.

```
package com.example.jpa.exception;
```

```
import org.springframework.http.HttpStatus;

import
org.springframework.web.bind.annotation.ResponseStatus;


@ResponseStatus(HttpStatus.NOT_FOUND)

public class ResourceNotFoundException extends
RuntimeException {

    public ResourceNotFoundException() {

        super();

    }


    public ResourceNotFoundException(String message) {

        super(message);

    }


    public ResourceNotFoundException(String message,
Throwable cause) {

        super(message, cause);

    }

}
```

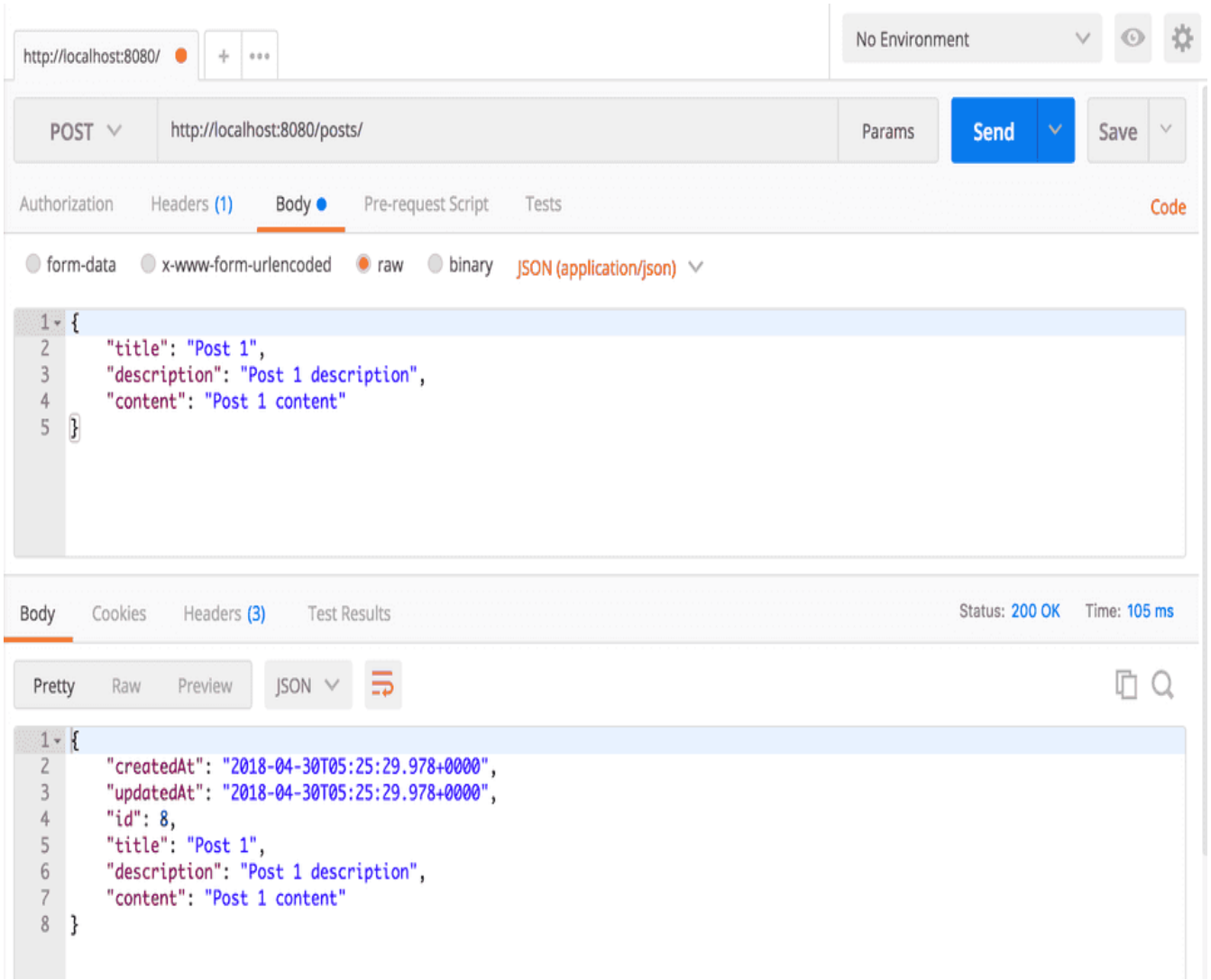
I have added the `@ResponseStatus(HttpStatus.NOT_FOUND)` annotation to the above exception class to tell Spring Boot to respond with a `404` status when this exception is thrown.

Running the Application and Testing the APIs via a Postman

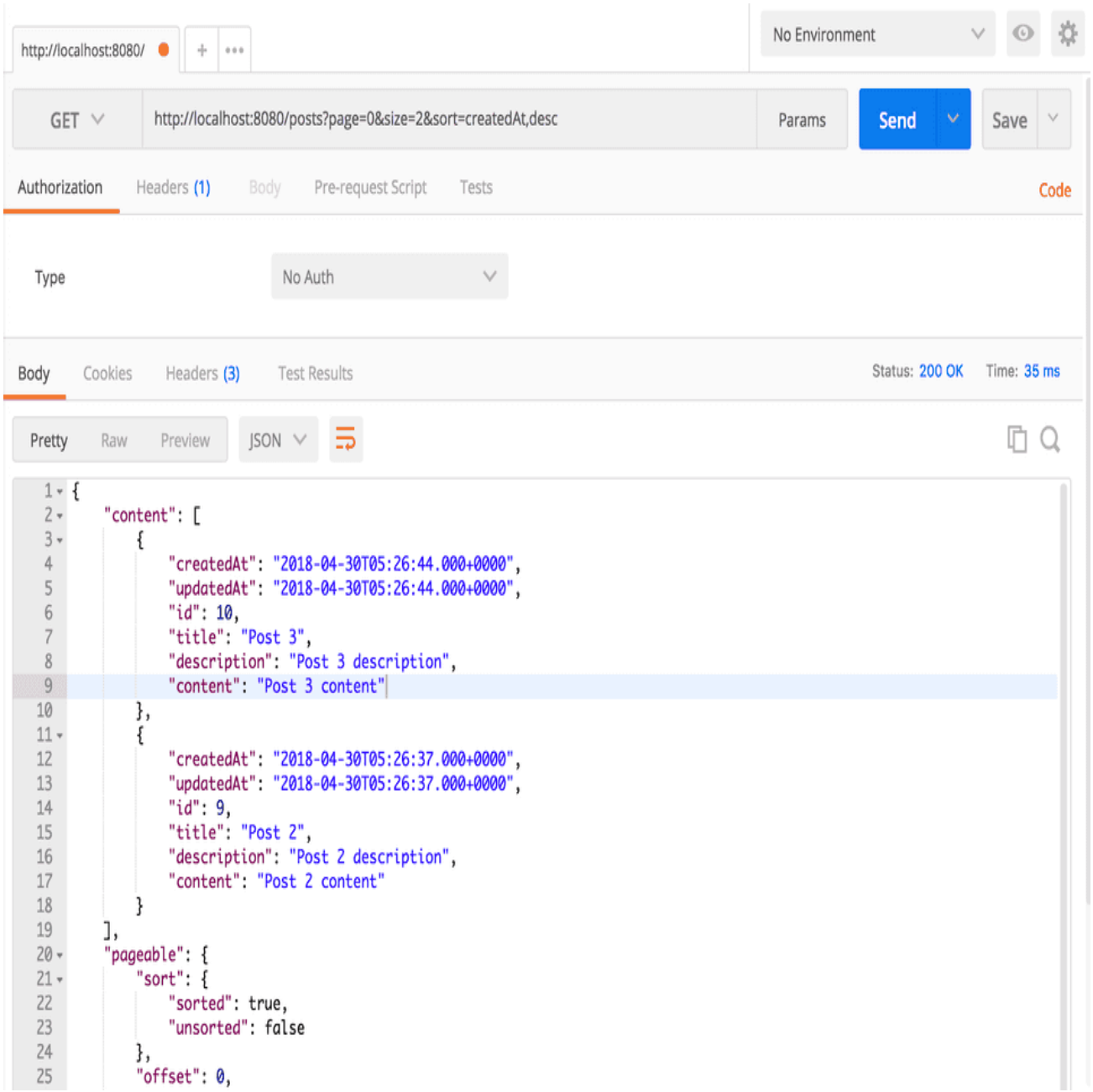
You can run the application by typing the following command in the terminal -

Let’s now test the APIs via Postman.

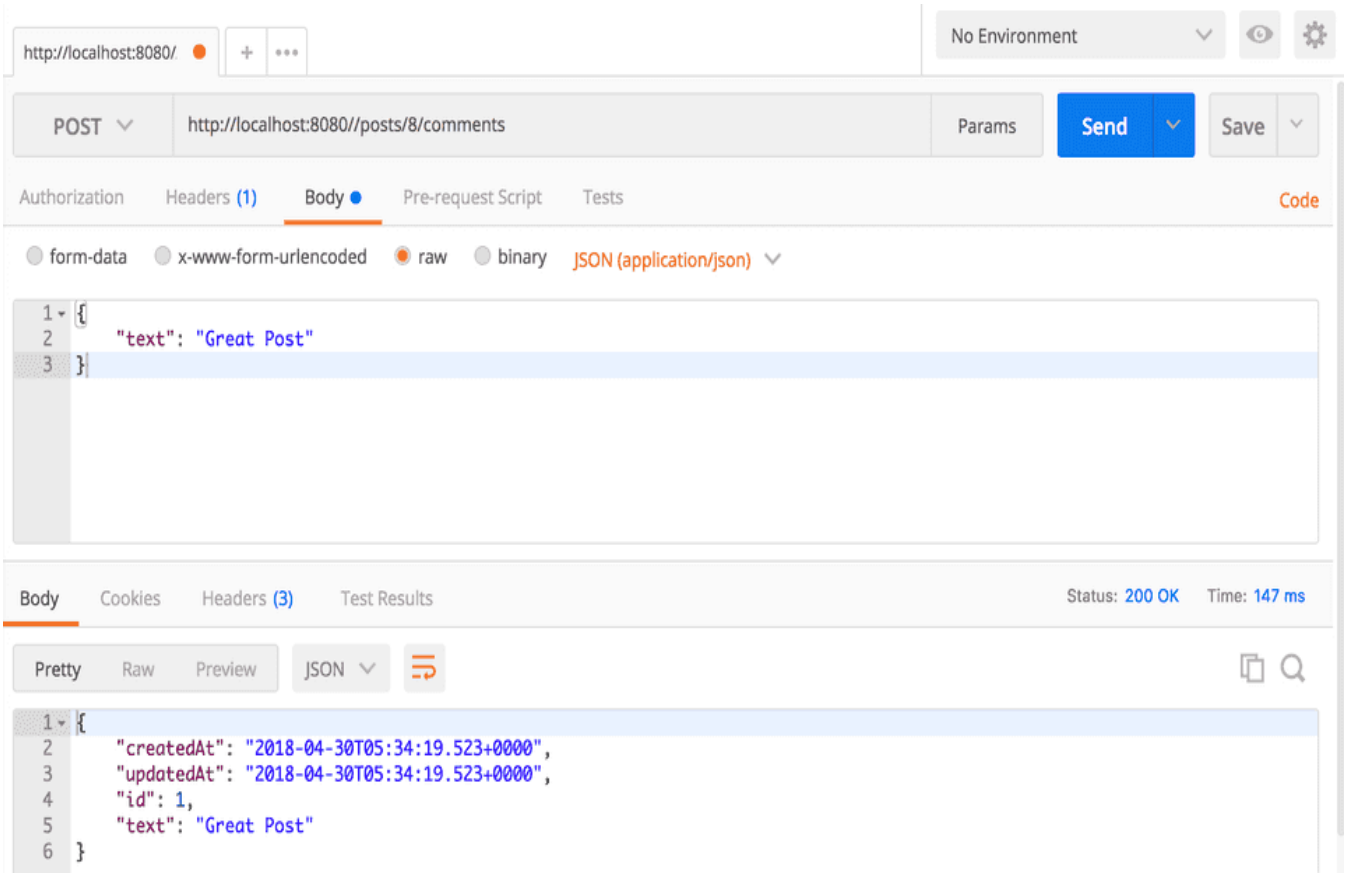
Create Post `POST /posts`



Get paginated Posts `GET /posts?page=0&size=2&sort=createdAt,desc`

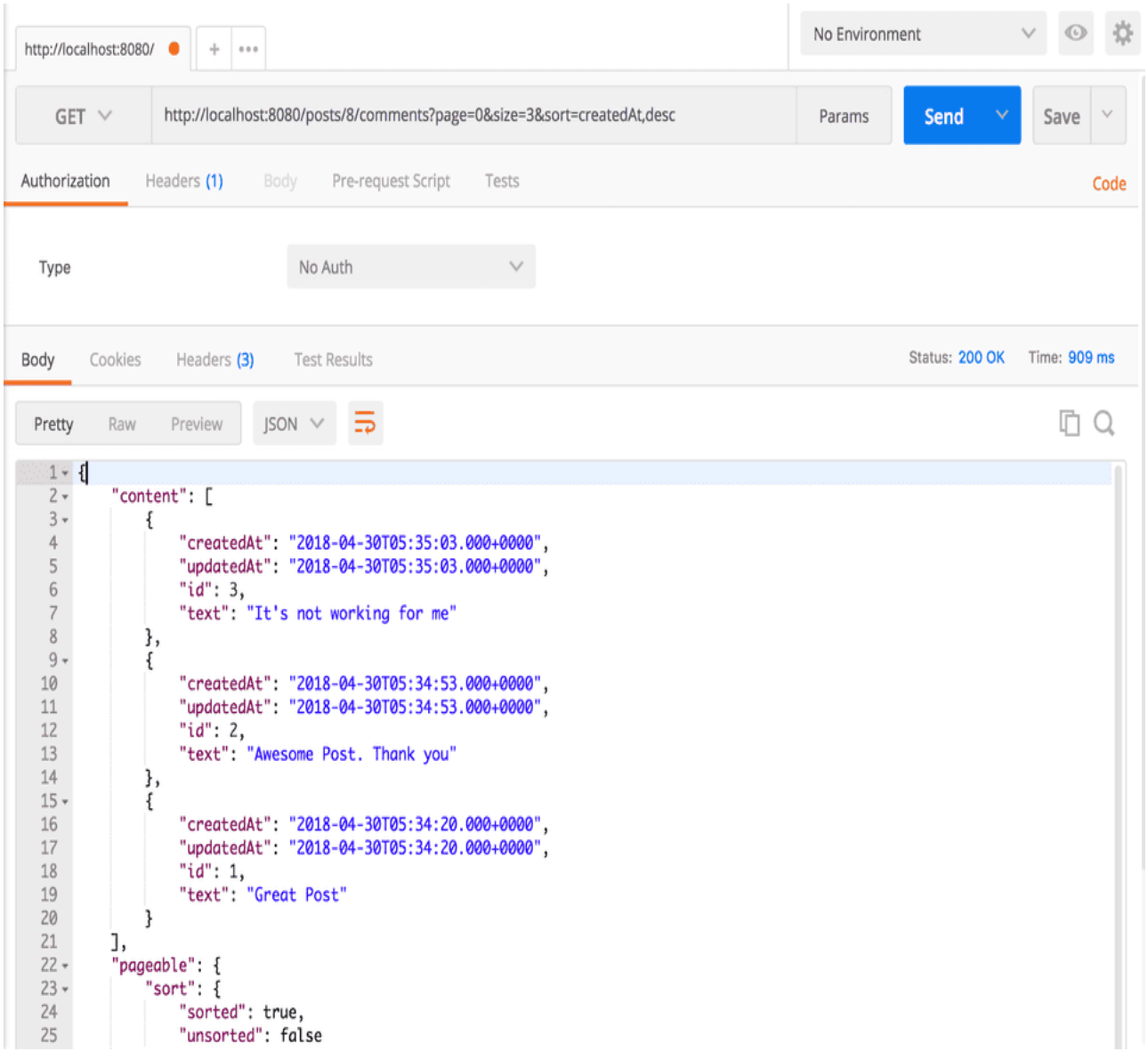


Create Comment `POST /posts/{postId}/comments`



Get paginated comments `GET`

`/posts/{postId}/comments?page=0&size=3&sort=createdAt,desc`



You can test other APIs in the same way.

How to define a bidirectional one-to-many mapping and when should you use it

The Internet is flooded with examples of bidirectional one-to-many mapping. But it's not the best and the most efficient way to model a one-to-many relationship.

Here is the bidirectional version of the one-to-many relationship between the `Post` and `Comment` entities -

Post Entity

```java
package com.example.jpa.model;

import javax.persistence.*;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import java.util.*;

@Entity
@Table(name = "posts")
public class Post extends AuditModel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Size(max = 100)
    @Column(unique = true)
    private String title;

    @NotNull
    @Size(max = 250)
    private String description;

    @NotNull
    @Lob
```

```
    private String content;

    @OneToMany(cascade = CascadeType.ALL,
            fetch = FetchType.LAZY,
            mappedBy = "post")
    private Set<Comment> comments = new HashSet<>();

    // Getters and Setters (Omitted for brevity)
}
```

Comment Entity

```
package com.example.jpa.model;

import javax.persistence.*;
import javax.validation.constraints.NotNull;

@Entity
@Table(name = "comments")
public class Comment extends AuditModel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Lob
```

```
    private String text;


    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id", nullable = false)
    private Post post;


    // Getters and Setters (Omitted for brevity)
}
```

The idea with bidirectional one-to-many association is to allow you to keep a collection of child entities in the parent, and enable you to persist and retrieve the child entities via the parent entity. This is made possible via Hibernate's entity state transitions and dirty checking mechanism.

For example, here is how you could persist comments via `post` entity in the bidirectional mapping -

```
// Create a Post
Post post = new Post("post title", "post description",
"post content");


// Create Comments
Comment comment1 = new Comment("Great Post!");
comment1.setPost(post);
Comment comment2 = new Comment("Really helpful Post.
Thanks a lot!");
comment2.setPost(post);


// Add comments in the Post
```

```
post.getComments().add(comment1);

post.getComments().add(comment2);


// Save Post and Comments via the Post entity

postRepository.save(post);
```

Hibernate automatically issues insert statements and saves the comments added to the post.

Similarly, you could fetch comments via the `post` entity like so -

```
// Retrieve Post

Post post = postRepository.findById(postId)


// Get all the comments

Set<Comment> comments = post.getComments()
```

When you write `post.getComments()`, hibernate loads all the comments from the database if they are not already loaded.

Problems with bidirectional one-to-many mapping

- A bidirectional mapping tightly couples the many-side of the relationship to the one-side.
- In our example, If you load comments via the `post` entity, you won't be able to limit the number of comments loaded. That essentially means that you won't be able to paginate.
- If you load comments via the `post` entity, you won't be able to sort them based on different properties. You can define a default sorting order using `@OrderColumn` annotation but that will have performance implications.
- You'll find yourself banging your head around something called a `LazyInitializationException`.

When can I use a bidirectional one-to-many mapping

A bidirectional one-to-many mapping might be a good idea if the number of child entities is limited.

Moreover, A bidirectional mapping tightly couples the many-side of the relationship to the one-side. *Many times, this tight coupling is desired.*

For example, Consider a Survey application with a `Question` and an `Option` entity exhibiting a one-to-many relationship between each other.

In the survey app, A `Question` can have a set of `Options`. Also, every `Question` is tightly coupled with its `Options`. When you create a `Question`, you'll also provide a set of `Options`. And, when you retrieve a `Question`, you will also need to fetch the `Options`.

Moreover, A `Question` can have at max 4 or 5 `Options`. These kind of cases are perfect for bi-directional mappings.

So, To decide between bidirectional and unidirectional mappings, you should think whether the entities have a tight coupling or not.