JPA One To Many example

We're gonna create a Spring project from scratch, then we implement JPA/Hibernate One to Many Mapping with `tutorials` and `comments` table as following:

We also write Rest Apis to perform CRUD operations on the Comment entities.

These are APIs that we need to provide:

Methods	Urls	Actions
POST	/api/tutorials/:id/comments	create new Comment for a Tutorial
GET	/api/tutorials/:id/comments	retrieve all Comments of a Tutorial
GET	/api/comments/:id	retrieve a Comment by `:id`
PUT	/api/comments/:id	update a Comment by `:id`
DELETE	/api/comments/:id	delete a Comment by `:id`
DELETE	/api/tutorials/:id	delete a Tutorial (and its Comments) by `:id`
DELETE	/api/tutorials/:id/comments	delete all Comments of a Tutorial

Assume that we've had **tutorials** table like this:

Here are the example requests:

– Create new Comments: POST `/api/tutorials/[:id]/comments`

comments table after that:

– Retrieve all Comments of specific Tutorial: GET `/api/tutorials/[:id]/comments`

– Delete all Comments of specific Tutorial: DELETE `/api/tutorials/[:id]/comments`

Check the **comment** table, all Comments of Tutorial with id=2 were deleted:

– Delete a Tutorial: DELETE `/api/tutorials/[:id]`

All Comments of the Tutorial with id=3 were **CASCADE** deleted automatically.

Let's build our Spring Boot One to Many CRUD example.

Spring Boot One to Many example

Technology

- Java 8
- Spring Boot 2.6.2 (with Spring Web MVC, Spring Data JPA)
- PostgreSQL/MySQL
- Maven 3.8.1

Project Structure

Let me explain it briefly.

– `Tutorial`, `Comment` data model class correspond to entity and table *tutorials*, *comments*.
– `TutorialRepository`, `CommentRepository` are interfaces that extends JpaRepository for CRUD methods and custom finder methods. It will be autowired in `TutorialController`, `CommentController`.
– `TutorialController`, `CommentController` are RestControllers which has request mapping methods for RESTful CRUD API requests.
– Configuration for Spring Datasource, JPA & Hibernate in **application.properties**.
– **pom.xml** contains dependencies for Spring Boot and MySQL/PostgreSQL/H2 database.

– About **exception** package, to keep this post straightforward, I won't explain it. For more details, you can read following tutorial:
@RestControllerAdvice example in Spring Boot

Create & Setup Spring Boot project

Use Spring web tool or your development tool (Spring Tool Suite, Eclipse, Intellij) to create a Spring Boot project.

Then open **pom.xml** and add these 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-web</artifactId>
</dependency>
```

We also need to add one more dependency.
– If you want to use **MySQL**:

```
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
</dependency>
```

– or **PostgreSQL**:

```
<dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
</dependency>
```

– or **H2** (embedded database):

```
<dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
</dependency>
```

Configure Spring Datasource, JPA, Hibernate

Under src/main/resources folder, open application.properties and write these lines.

– For MySQL:

```
spring.datasource.url= jdbc:mysql://localhost:3306/testdb?useSSL=false
spring.datasource.username= root
spring.datasource.password= 123456


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


# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto= update
```

– For PostgreSQL:

```
spring.datasource.url= jdbc:postgresql://localhost:5432/testdb
spring.datasource.username= postgres
spring.datasource.password= 123

spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation= true
spring.jpa.properties.hibernate.dialect=
org.hibernate.dialect.PostgreSQLDialect

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto= update
```

- `spring.datasource.username` & `spring.datasource.password` properties are the same as your database installation.
- Spring Boot uses Hibernate for JPA implementation, we configure `MySQL5InnoDBDialect` for MySQL or `PostgreSQLDialect` for PostgreSQL
- `spring.jpa.hibernate.ddl-auto` is used for database initialization. We set the value to `update` value so that a table will be created in the database automatically corresponding to defined data model. Any change to the model will also trigger an update to the table. For production, this property should be `validate`.

– For H2 database:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto= update

spring.h2.console.enabled=true
# default path: h2-console
spring.h2.console.path=/h2-ui
```

- `spring.datasource.url`: `jdbc:h2:mem:[database-name]` for In-memory database and `jdbc:h2:file:[path/database-name]` for disk-based database.
- We configure `H2Dialect` for H2 Database
- `spring.h2.console.enabled=true` tells the Spring to start H2 Database administration tool and you can access this tool on the browser: `http://localhost:8080/h2-console`.
- `spring.h2.console.path=/h2-ui` is for H2 console's url, so the default url `http://localhost:8080/h2-console` will change to `http://localhost:8080/h2-ui`.

Define Data Model for JPA One to Many mapping

In **model** package, we define `Tutorial` and `Comment` class.

Tutorial has four fields: id, title, description, published.

model/*Tutorial.java*

```
package com.bezkoder.spring.hibernate.onetomany.model;

import javax.persistence.*;

@Entity
@Table(name = "tutorials")
public class Tutorial {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"tutorial_generator")
  private long id;

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

  @Column(name = "description")
  private String description;

  @Column(name = "published")
  private boolean published;

  public Tutorial() {

  }

  public Tutorial(String title, String description, boolean published) {
    this.title = title;
    this.description = description;
    this.published = published;
  }

  // getters and setters
}
```

– `@Entity` annotation indicates that the class is a persistent Java class.
– `@Table` annotation provides the table that maps this entity.

– `@Id` annotation is for the primary key.
– `@GeneratedValue` annotation is used to define generation strategy for the primary key. `GenerationType.SEQUENCE` means using database sequence to generate unique values.
We also indicate the name of the primary key `generator`. If you don't give it the name, id value will be generated with **hibernate_sequence** table (supplied by persistence provider, for all entities) by default.

– `@Column` annotation is used to define the column in database that maps annotated field.

The `Comment` class has the `@ManyToOne` annotation for many-to-one relationship with the `Tutorial` entity. `optional` element is set to `false` for non-null relationship.

model/*Comment.java*

```
package com.bezkoder.spring.hibernate.onetomany.model;


import javax.persistence.*;


import org.hibernate.annotations.OnDelete;
import org.hibernate.annotations.OnDeleteAction;


import com.fasterxml.jackson.annotation.JsonIgnore;


@Entity
@Table(name = "comments")
public class Comment {
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"comment_generator")
  private Long id;


  @Lob
  private String content;


  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "tutorial_id", nullable = false)
  @OnDelete(action = OnDeleteAction.CASCADE)
  @JsonIgnore
  private Tutorial tutorial;
```

```
  // getters and setters
}
```

We also use the @JoinColumn annotation to specify the foreign key column (`tutorial_id`). If you don't provide the `JoinColumn` name, the name will be set automatically.

`@JsonIgnore` is used to ignore the logical property used in serialization and deserialization.

We also implement cascade delete capabilities of the foreign-key with `@OnDelete(action = OnDeleteAction.CASCADE)`.

We set the `@ManyToOne` with `FetchType.LAZY` for `fetch` type:

By default, the `@ManyToOne` association uses `FetchType.EAGER` for `fetch` type but it is bad for performance:

```
public class Comment {
  ...

  @ManyToOne(fetch = FetchType.EAGER, optional = false)
  @JoinColumn(name = "tutorial_id", nullable = false)
  @OnDelete(action = OnDeleteAction.CASCADE)
  private Tutorial tutorial;

  ...
}
```

Create Repository Interfaces for One To Many mapping

Let's create a repository to interact with database.
In **repository** package, create `TutorialRepository` and `CommentRepository` interfaces that extend `JpaRepository`.

repository/*TutorialRepository.java*

```
package com.bezkoder.spring.hibernate.onetomany.repository;

import java.util.List;

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

import com.bezkoder.spring.hibernate.onetomany.model.Tutorial;

public interface TutorialRepository extends JpaRepository<Tutorial, Long> {
```

```
  List<Tutorial> findByPublished(boolean published);

  List<Tutorial> findByTitleContaining(String title);
}
```

repository/*CommentRepository.java*

```
package com.bezkoder.spring.hibernate.onetomany.repository;

import java.util.List;

import javax.transaction.Transactional;

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

import com.bezkoder.spring.hibernate.onetomany.model.Comment;

public interface CommentRepository extends JpaRepository<Comment, Long> {
  List<Comment> findByTutorialId(Long postId);

  @Transactional
  void deleteByTutorialId(long tutorialId);
}
```

Now we can use JpaRepository's methods: `save()`, `findOne()`, `findById()`, `findAll()`, `count()`, `delete()`, `deleteById()`... without implementing these methods.

We also define custom finder methods:

- `findByPublished()`: returns all Tutorials with `published` having value as input `published`.
- `findByTitleContaining()`: returns all Tutorials which title contains input `title`.
- `findByTutorialId()`: returns all Comments of a Tutorial specified by `tutorialId`.
- `deleteByTutorialId()`: deletes all Comments of a Tutorial specified by `tutorialId`.

The implementation is plugged in by Spring Data JPA automatically.

Now we can see the pros of `@ManyToOne` annotation.
– With `@OneToMany`, we need to declare a collection inside parent class, we cannot limit the size of that collection, for example, in case of pagination.
– With `@ManyToOne`, you can modify Repository:

- to work with Pagination, the instruction can be found at:
  Spring Boot Pagination & Filter example | Spring JPA, Pageable

- or to sort/order by multiple fields:
  Spring Data JPA Sort/Order by multiple Columns | Spring Boot

Please notice that above tutorials are for `TutorialRepository`, you need to apply the same way for `CommentRepository` .

More Derived queries at:
JPA Repository query example in Spring Boot

Custom query with `@Query` annotation:
Spring JPA @Query example: Custom query in Spring Boot

You also find way to write Unit Test for this JPA Repository at:
Spring Boot Unit Test for JPA Repositiory with @DataJpaTest

Create Spring Rest APIs Controller

Finally, we create controller that provides APIs for CRUD operations: creating, retrieving, updating, deleting and finding Tutorials and Comments.

controller/TutorialController.java

```
package com.bezkoder.spring.hibernate.onetomany.controller;

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import
com.bezkoder.spring.hibernate.onetomany.exception.ResourceNotFoundException;
import com.bezkoder.spring.hibernate.onetomany.model.Tutorial;
import com.bezkoder.spring.hibernate.onetomany.repository.TutorialRepository;
```

```java
@CrossOrigin(origins = "http://localhost:8081")
@RestController
@RequestMapping("/api")
public class TutorialController {

  @Autowired
  TutorialRepository tutorialRepository;

  @GetMapping("/tutorials")
  public ResponseEntity<List<Tutorial>> getAllTutorials(@RequestParam(required
= false) String title) {
    List<Tutorial> tutorials = new ArrayList<Tutorial>();

    if (title == null)
      tutorialRepository.findAll().forEach(tutorials::add);
    else
      tutorialRepository.findByTitleContaining(title).forEach(tutorials::add);

    if (tutorials.isEmpty()) {
      return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }

    return new ResponseEntity<>(tutorials, HttpStatus.OK);
  }

  @GetMapping("/tutorials/{id}")
  public ResponseEntity<Tutorial> getTutorialById(@PathVariable("id") long id)
{
    Tutorial tutorial = tutorialRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Not found Tutorial
with id = " + id));

    return new ResponseEntity<>(tutorial, HttpStatus.OK);
  }

  @PostMapping("/tutorials")
  public ResponseEntity<Tutorial> createTutorial(@RequestBody Tutorial
tutorial) {
    Tutorial _tutorial = tutorialRepository.save(new
Tutorial(tutorial.getTitle(), tutorial.getDescription(), true));
```

```
    return new ResponseEntity<>(_tutorial, HttpStatus.CREATED);
  }

  @PutMapping("/tutorials/{id}")
  public ResponseEntity<Tutorial> updateTutorial(@PathVariable("id") long id,
@RequestBody Tutorial tutorial) {
    Tutorial _tutorial = tutorialRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Not found Tutorial
with id = " + id));

    _tutorial.setTitle(tutorial.getTitle());
    _tutorial.setDescription(tutorial.getDescription());
    _tutorial.setPublished(tutorial.isPublished());

    return new ResponseEntity<>(tutorialRepository.save(_tutorial),
HttpStatus.OK);
  }

  @DeleteMapping("/tutorials/{id}")
  public ResponseEntity<HttpStatus> deleteTutorial(@PathVariable("id") long
id) {
    tutorialRepository.deleteById(id);

    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
  }

  @DeleteMapping("/tutorials")
  public ResponseEntity<HttpStatus> deleteAllTutorials() {
    tutorialRepository.deleteAll();

    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
  }

  @GetMapping("/tutorials/published")
  public ResponseEntity<List<Tutorial>> findByPublished() {
    List<Tutorial> tutorials = tutorialRepository.findByPublished(true);

    if (tutorials.isEmpty()) {
      return new ResponseEntity<>(HttpStatus.NO_CONTENT);
```

```
    }

    return new ResponseEntity<>(tutorials, HttpStatus.OK);
  }
}
```

controller/CommentController.java

```
package com.bezkoder.spring.hibernate.onetomany.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import
com.bezkoder.spring.hibernate.onetomany.exception.ResourceNotFoundException;
import com.bezkoder.spring.hibernate.onetomany.model.Comment;
import com.bezkoder.spring.hibernate.onetomany.repository.CommentRepository;
import com.bezkoder.spring.hibernate.onetomany.repository.TutorialRepository;

@CrossOrigin(origins = "http://localhost:8081")
@RestController
@RequestMapping("/api")
public class CommentController {

  @Autowired
  private TutorialRepository tutorialRepository;

  @Autowired
```

```
  private CommentRepository commentRepository;


  @GetMapping("/tutorials/{tutorialId}/comments")
  public ResponseEntity<List<Comment>>
getAllCommentsByTutorialId(@PathVariable(value = "tutorialId") Long
tutorialId) {
    if (!tutorialRepository.existsById(tutorialId)) {
      throw new ResourceNotFoundException("Not found Tutorial with id = " +
tutorialId);
    }


    List<Comment> comments = commentRepository.findByTutorialId(tutorialId);
    return new ResponseEntity<>(comments, HttpStatus.OK);
  }


  @GetMapping("/comments/{id}")
  public ResponseEntity<Comment> getCommentsByTutorialId(@PathVariable(value =
"id") Long id) {
    Comment comment = commentRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Not found Comment
with id = " + id));


    return new ResponseEntity<>(comment, HttpStatus.OK);
  }


  @PostMapping("/tutorials/{tutorialId}/comments")
  public ResponseEntity<Comment> createComment(@PathVariable(value =
"tutorialId") Long tutorialId,
      @RequestBody Comment commentRequest) {
    Comment comment = tutorialRepository.findById(tutorialId).map(tutorial ->
{
      commentRequest.setTutorial(tutorial);
      return commentRepository.save(commentRequest);
    }).orElseThrow(() -> new ResourceNotFoundException("Not found Tutorial
with id = " + tutorialId));


    return new ResponseEntity<>(comment, HttpStatus.CREATED);
  }


  @PutMapping("/comments/{id}")
```

```
  public ResponseEntity<Comment> updateComment(@PathVariable("id") long id,
@RequestBody Comment commentRequest) {
    Comment comment = commentRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("CommentId " + id +
"not found"));

    comment.setContent(commentRequest.getContent());

    return new ResponseEntity<>(commentRepository.save(comment),
HttpStatus.OK);
  }

  @DeleteMapping("/comments/{id}")
  public ResponseEntity<HttpStatus> deleteComment(@PathVariable("id") long id)
{
    commentRepository.deleteById(id);

    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
  }

  @DeleteMapping("/tutorials/{tutorialId}/comments")
  public ResponseEntity<List<Comment>>
deleteAllCommentsOfTutorial(@PathVariable(value = "tutorialId") Long
tutorialId) {
    if (!tutorialRepository.existsById(tutorialId)) {
      throw new ResourceNotFoundException("Not found Tutorial with id = " +
tutorialId);
    }

    commentRepository.deleteByTutorialId(tutorialId);
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
  }
}
```