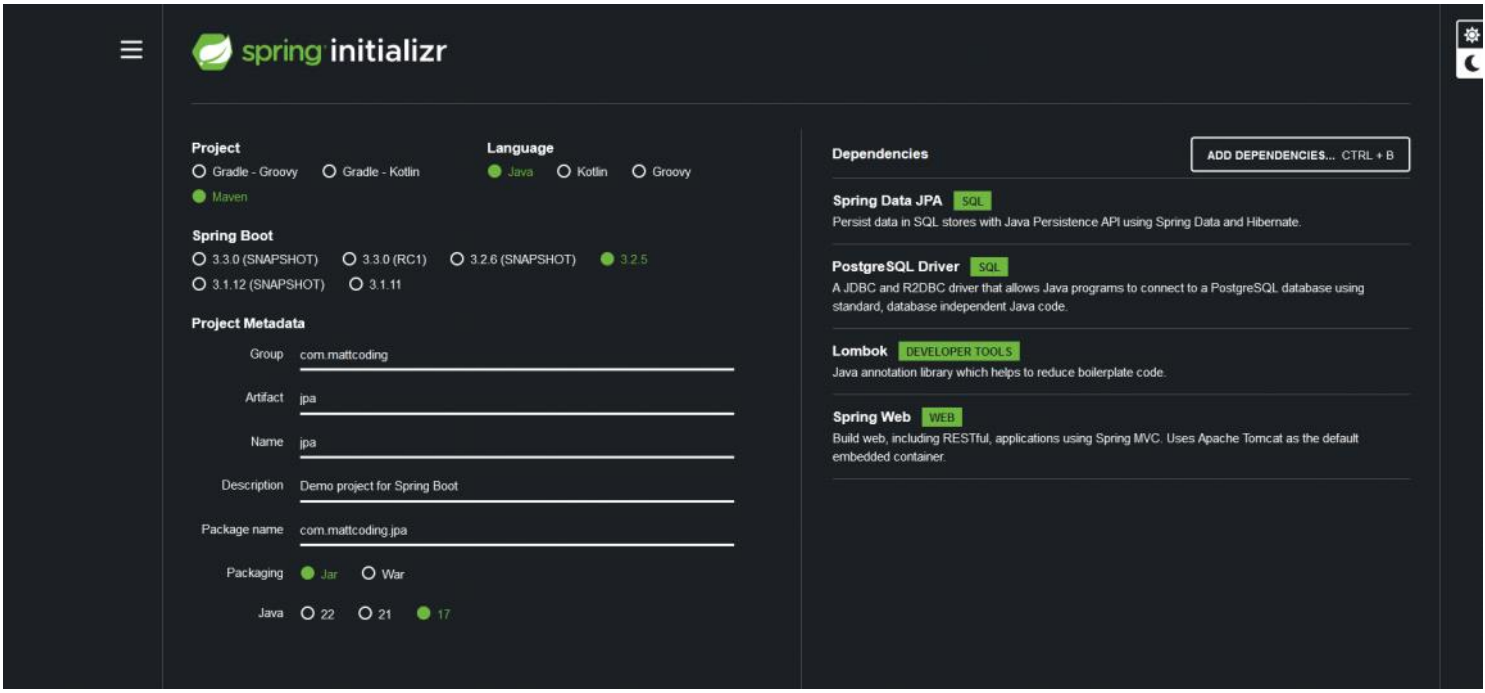
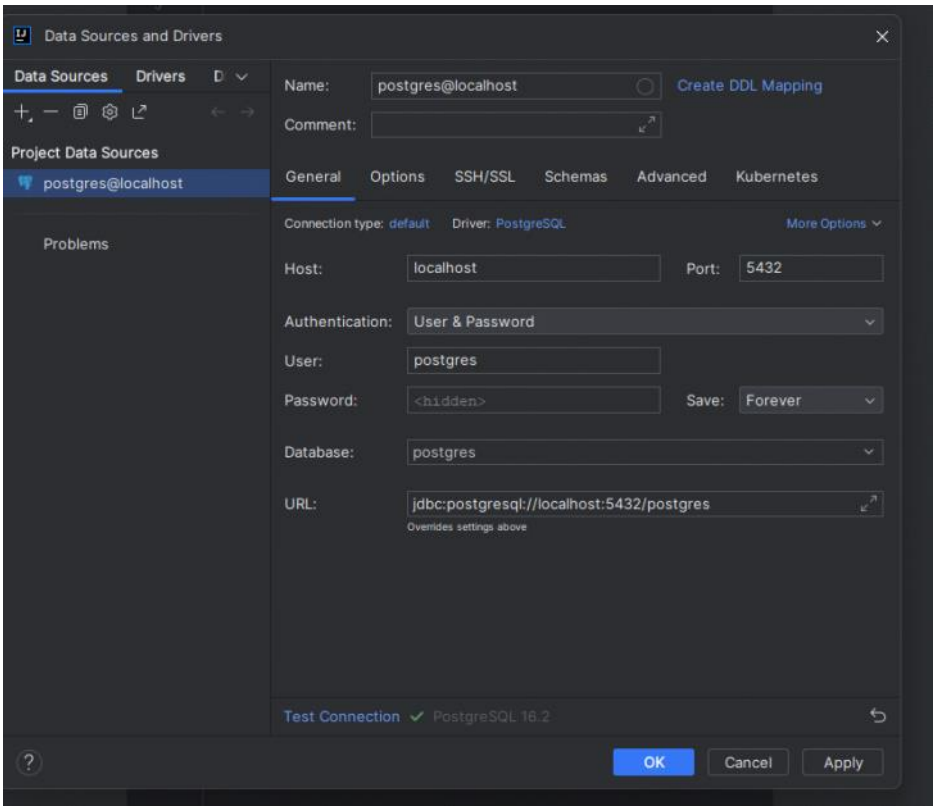
Intializr & Postgres

11 May 2024 08:22





Enable public Schema (0 of 3 click on)

Configuring the database

11 May 2024 11:02

Everything quite self-explanatory

Driver-class name - what driver should spring user to interact with the db

Ddl-auto - what should spring do to the database on start

Note on dll-auto option update - it will create new attributes but won't remove existing attributes

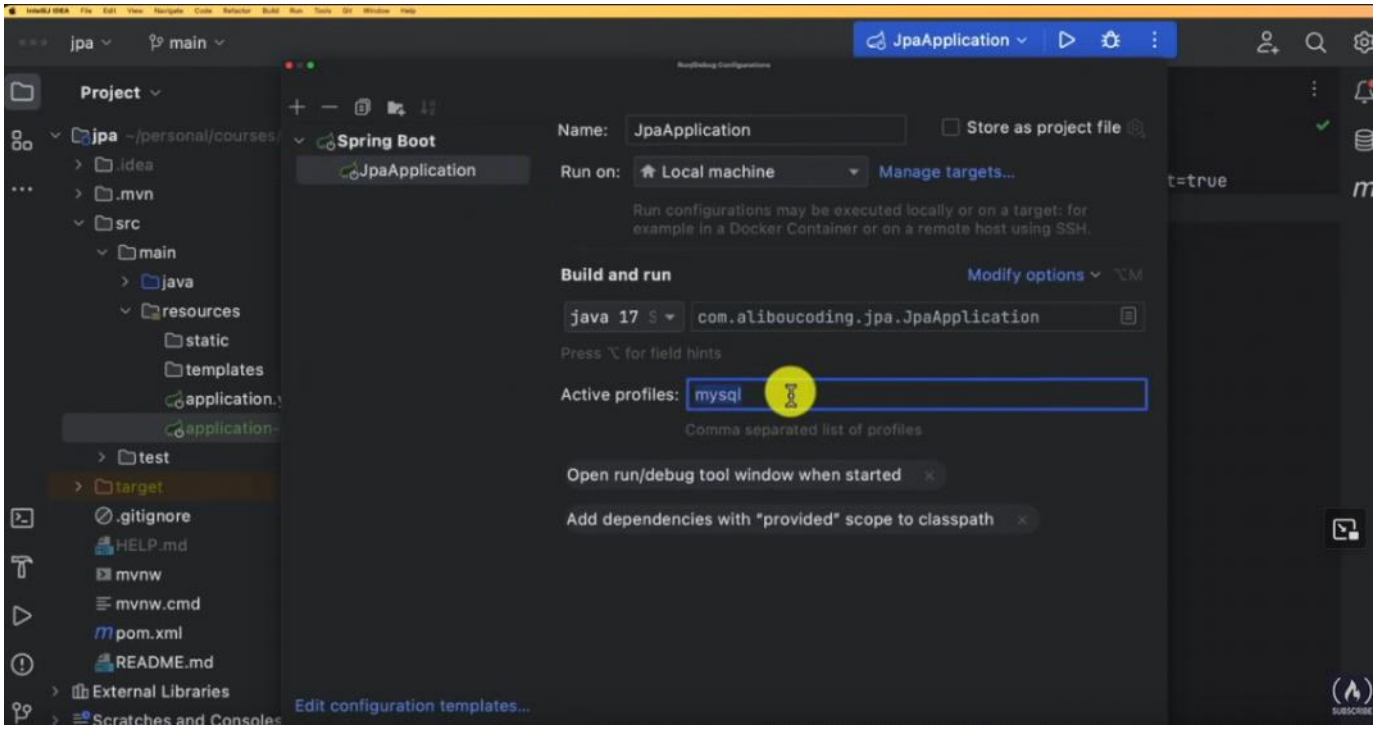
```yaml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/data_jpa
    username: postgres
    password: Buzz0001
    driver-class-name: org.postgresql.Driver
  jpa:
    hibernate:
      ddl-auto: create-drop
    database: postgresql
    show-sql: true
```

If some database options we can use "create if does not exist"

```yaml
  url: jdbc:mysql://localhost:3306/data_jpa?createDatabaseIfNotExist=true
```

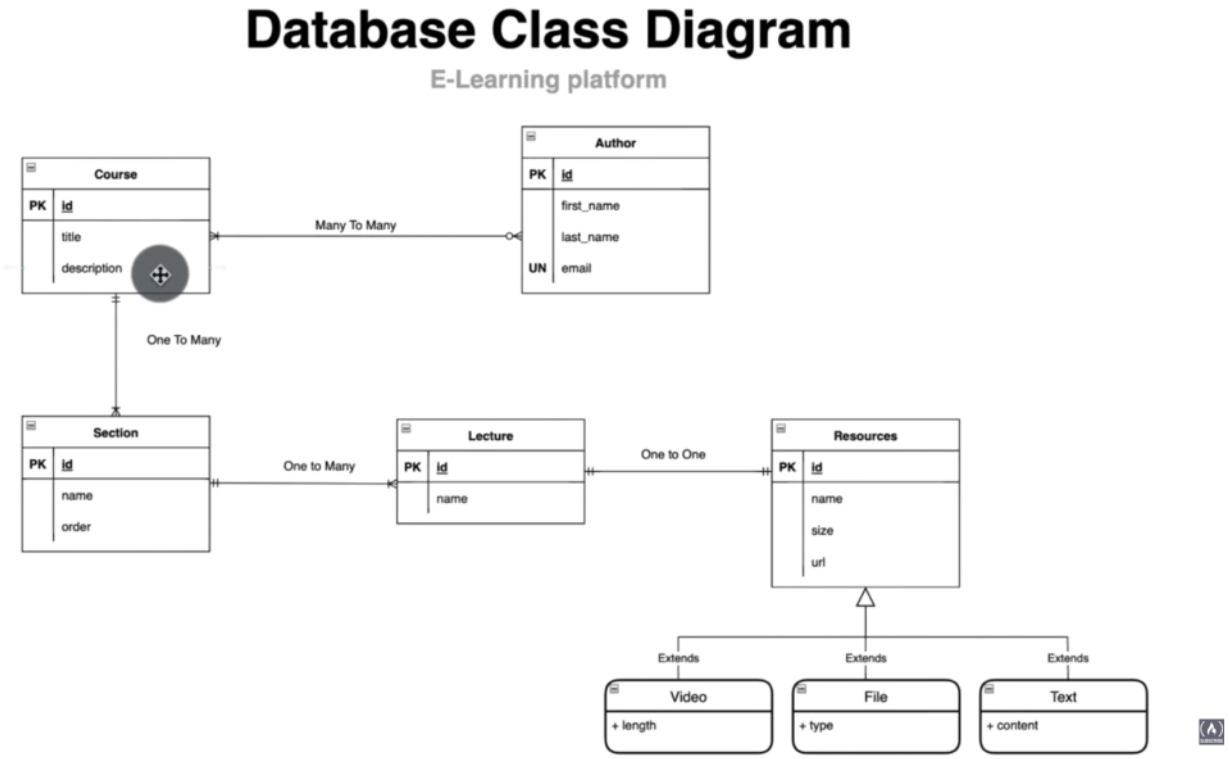
If you want to switch between the different database types, make it profilable:

Application-mysql-yaml (example)


jpa
main
JpaApplication
Project
jpa ~/personal/courses/
.idea
.mvn
src
main
java
resources
static
templates
application.
application-
test
target
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
README.md
External Libraries
Scratches and Consoles
Spring Boot
JpaApplication
Name: JpaApplication
Store as project file
Run on: Local machine
Manage targets...
Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.
Build and run
Modify options
java 17
com.aliboucoding.jpa.JpaApplication
Press ⌥ for field hints
Active profiles: mysql
Comma separated list of profiles
Open run/debug tool window when started
Add dependencies with "provided" scope to classpath
Edit configuration templates...
t=true

Database Class Diagram (Relationships)

11 May 2024 11:37



Explanation:

A course can have many different authors - An author can create many different courses (Many to Many)

A course can have many sections - but a section can only be assigned to one course (One to Many)

A Section can only have one lecture - but a lecturer can cover multiple sections (One to Many)

A lecture has one resource - it only belongs to one lecture

A resource has many sub-classes that extend it of different types (video, file,text)

Hibernate Vs Spring Data JPA

11 May 2024 12:06

Spring data JPA - just sits on top of JPA to reduce the amount of code and effort needed to implement **data access objects (DAO). Spring data starter will configure basically everything for us**

JPA - is just a spec that allows for object relational mapping in Java applications - its like a java interface where you define your data access methods. This allows you to easily switch JPA implementation

Hibernate - is a JPA implementation and generates SQL. Its our ORM (Object relational mapper)

JDBC - handles database side, saving,reading and deleting.

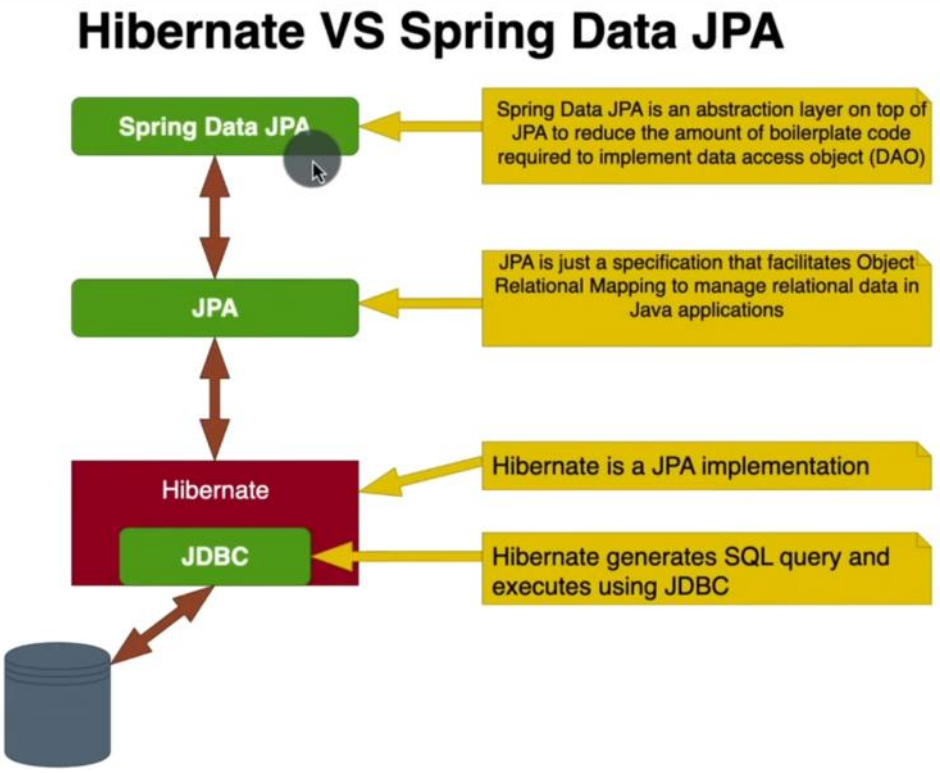
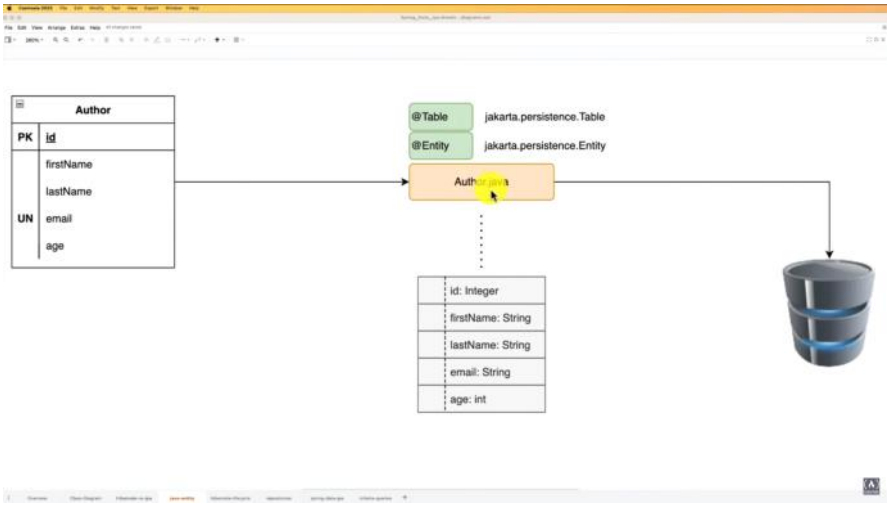


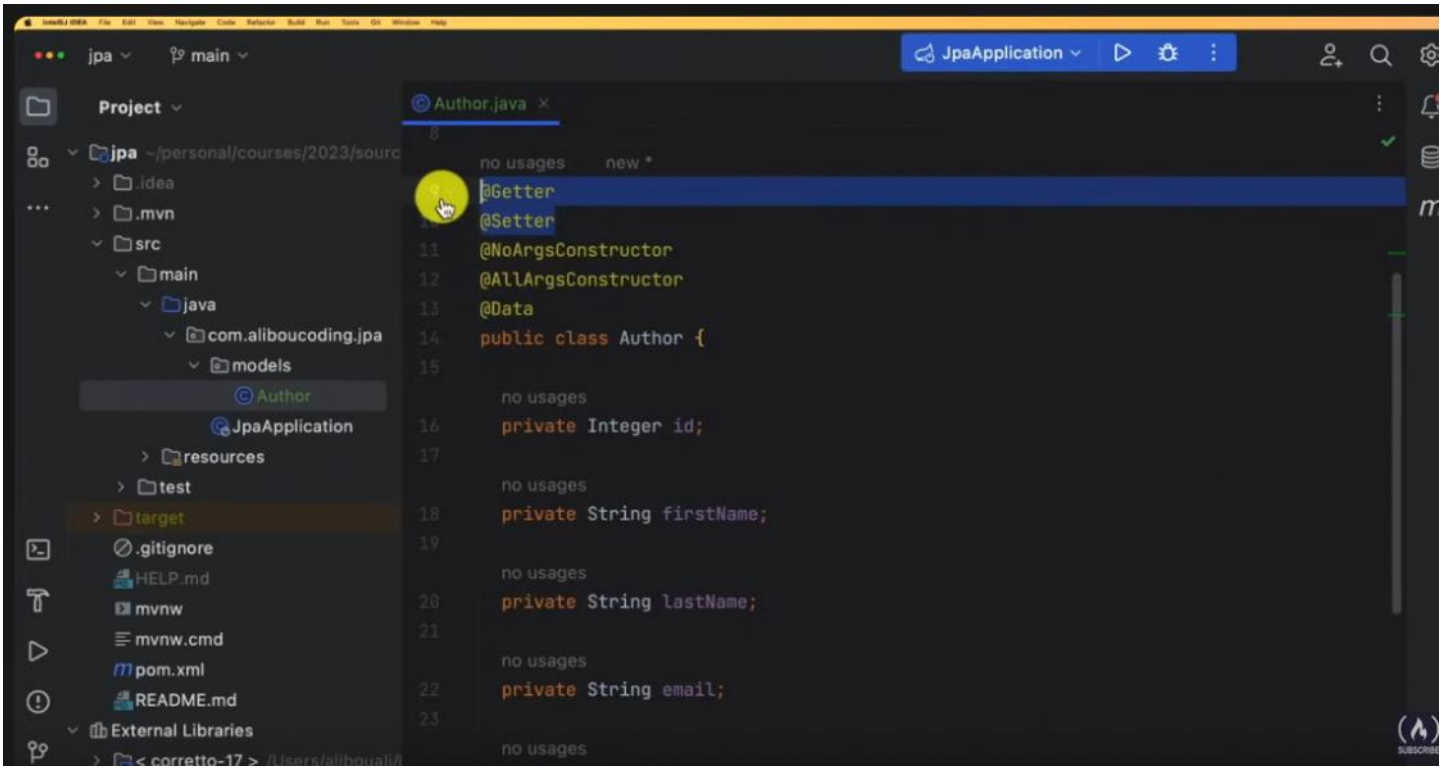
Table to Java Class + Lombock

11 May 2024 12:13

Diagram:



Lombock has lots of useful annotations to save you time:



Only required one in our case is @Data which gives us: constructor, custom toString, Setters and Getters.

Classes must be marked as @Entity to make it usable as model

All models need primary keys use @Id to be unique

Hibernate will also handle id creation with @GeneratedValue (must be used with Id annotations) - the strategy for this generation can be changed

Why use Integer over int in this context:

By default int = 0, Integer = null

Hibernate knows that if the id is null, it will create object . If it isn't null, it will try and find it

So its important you use wrapper (integer over int) in our models

Final Class:

```
no usages
@Entity
@Data
public class Author {

    @Id
    @GeneratedValue
    private  Integer id;
    private String firstName;
    private String lastName;
    private String email;
    private Integer age;


}
```

How hibernate generates Id's:

@GeneratedValue Explained

11 May 2024 12:48

Hibernate will also handle id creation with @GeneratedValue (must be used with @Id annotation) - the strategy for this generation can be changed

We can set id generation strategy manually with:

Strategy parm and generator string name parm

```
no usages
@Entity
@Data
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
    generator = "author_sequence")
    private  Integer id;
    private String firstName;
    private String lastName;
    private String email;
    private Integer age;


}
```

Strategies:

Auto - hibernate will choose the best way to generate ids depending on db

Sequence - a number stored on the server which changes by one each milisecond. Spring boot will generate these or you can point it to it

We can customize this with @SequenceGenerator and create custom sequences for each entity

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
generator = "author_sequence")
@SequenceGenerator(name = "author_sequence", sequenceName = "author_sequence",
        allocationSize = 1)
private  Integer id;
```

Table - Creates a table to manage and generate our ids

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE,
generator = "author_id_generator")
@TableGenerator(name = "author_id_generator", table = "id_generator", pkColumnName = "id_name",
        valueColumnName = "id_value", allocationSize = 1)
```

WHERE ORDER BY

	id_value	id_name
1	0	author

Class to database

11 May 2024 12:27

@Column Explained - rename

14 May 2024 09:12

Column is used to change rules surrounding a column

Column has many different uses:

Name - allows you to rename the column for a value - useful if we have an existing database

Unique - forces the entered value to be unique to other values in the column

Insertable - true or false, tells spring if it should be allowed to insert data into this column

Updatable - true or false, allows spring to update existing entities in the column

Length - set a minimum column length

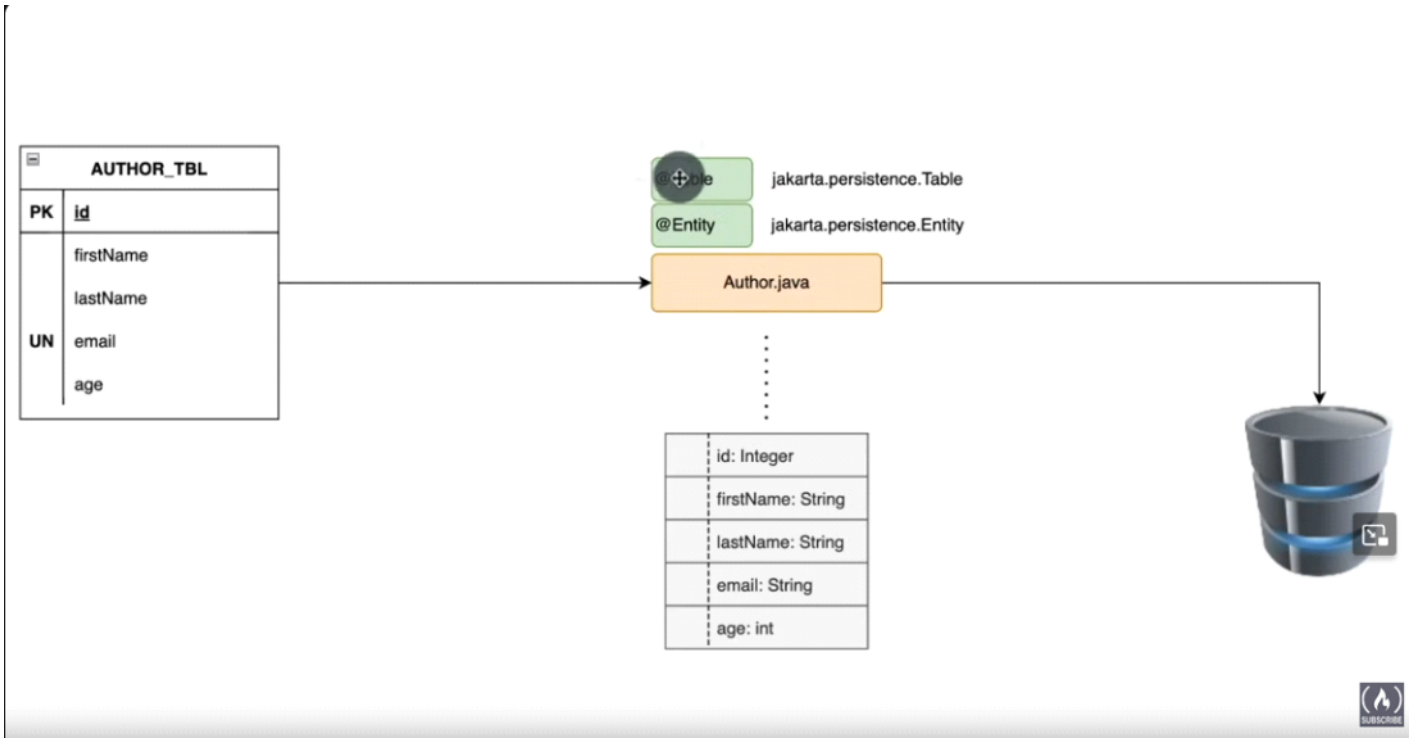
Nullable - true or false, can be empty

```java
@Entity
@Data
public class Author {

    @Id
    @GeneratedValue
//    @GeneratedValue(strategy = GenerationType.TABLE,
//    generator = "author_id_generator")
//    @TableGenerator(name = "author_id_generator", table = "id_generator", pkColumnName = "id_name",
//            valueColumnName = "id_value", allocationSize = 1)
    private  Integer id;

    @Column(
            name = "f-name",
            length = 40
    )
    private String firstName;
    private String lastName;
    @Column(
            unique = true,
            nullable = false
    )
    private String email;
    private Integer age;
    @Column(
            insertable = true,
            updatable = false,
            nullable = false
    )
    private LocalDateTime createdAt;

@Column(
        insertable = false,
        updatable = true
)
    private LocalDateTime lastModified;


}
```

```
}
```

Table modification

14 May 2024 09:49



We want to modify our table

@Table

Name - we can modify the name of table (database in not case sensitive)

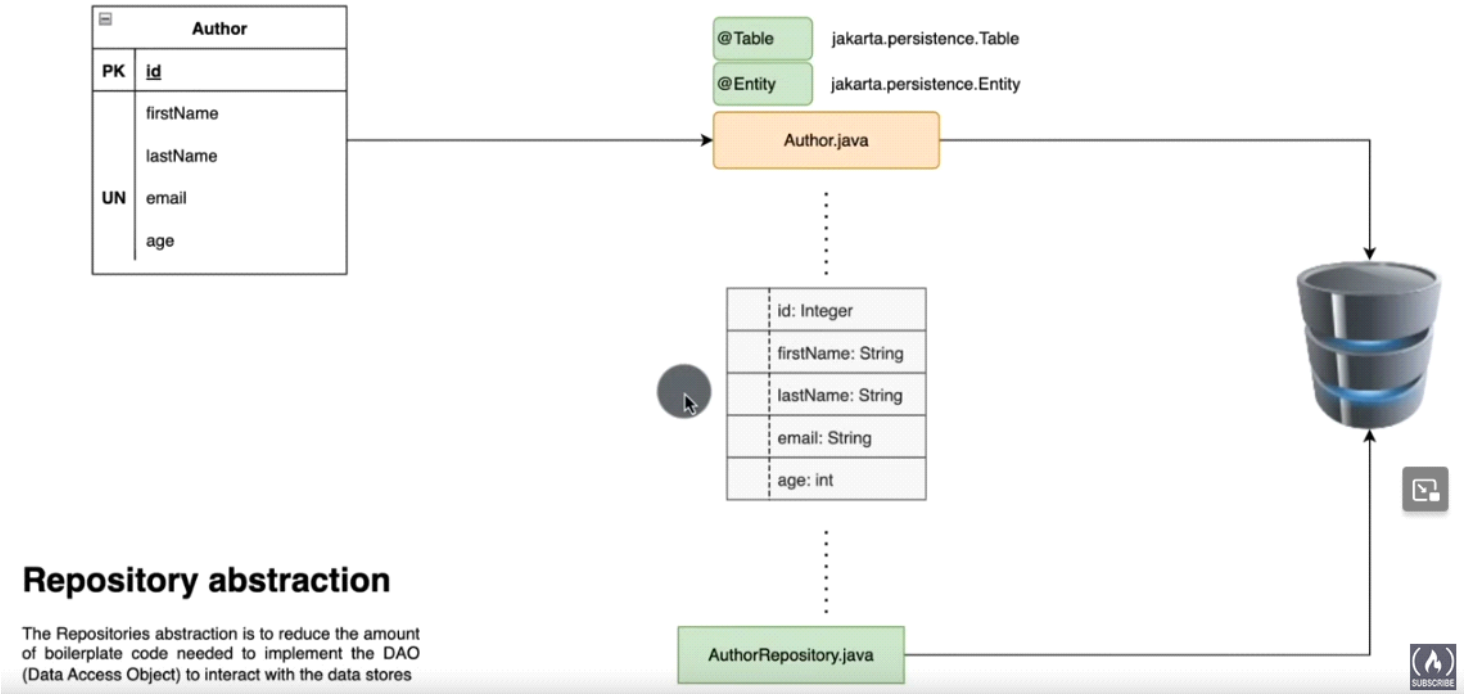
Catalog - we can change the catalog

Schema - we can change our table schema from here

Repository Abstraction

14 May 2024 09:58

The repository abstraction is a Spring JPA feature that allows us to use a DAO (Data access object) to interact with our database easily. Reducing the amount of code needed to interact with a datastore.



Repositories are used to separate data access logic from the business logic

Repositories are used to perform CRUD operations (pagination and sorting also)

Repositories are interfaces that extend one of the 3 types:

JPA Repositories

CRUD repositories

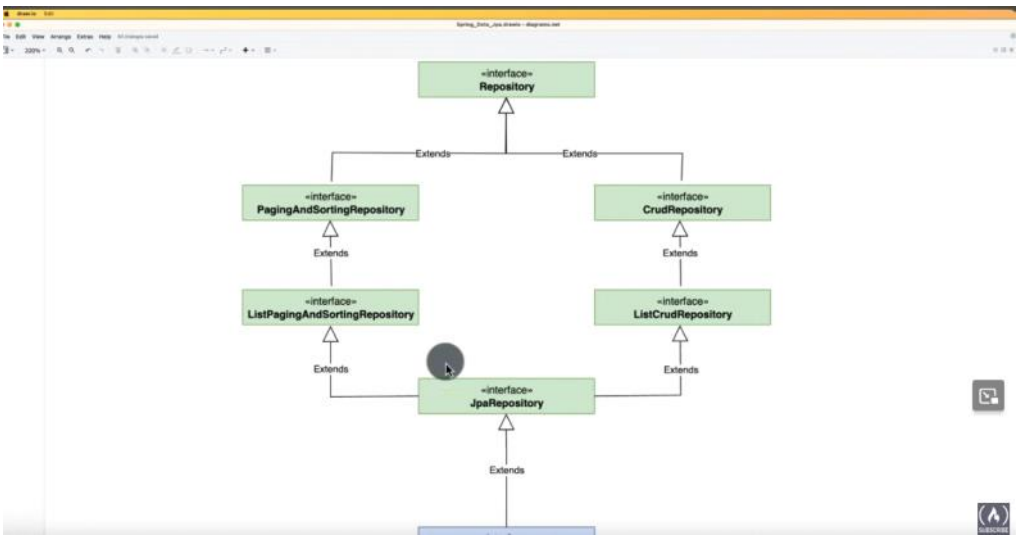
Paging And Repositories

All of these contain the common database methods

Inheritance chart:

All of these classes have their own methods we use

If we only want select methods, just extend one of these instead of JPA Repository



ListCrudRepository - Contains all the listing methods (FindAll, SaveAll etc)

PagingAndSortingRepository - Contains all methods that take in Sort object

Author Repository + Added Annotations

14 May 2024 10:27

Repositories take in generic types (<Type class, ID Type>

```java
@Entity
@Data
@AllArgsConstructor
@Builder
//@Table(name = "AUTHOR_TBL")
public class Author {

    @Id
    @GeneratedValue
//    @GeneratedValue(strategy = GenerationType.TABLE,
//    generator = "author_id_generator")
//    @TableGenerator(name = "author_id_generator", table = "id_generator", pkColumnName = "id_name",
//            valueColumnName = "id_value", allocationSize = 1)
    private  Integer id;

//    @Column(
//            name = "f-name",
//            length = 40
//    )
    private String firstName;
    private String lastName;
    @Column(
            unique = true,
            nullable = false
    )
    private String email;
    private Integer age;
    @Column(
            insertable = true,
            updatable = false,
            nullable = false
    )
    private LocalDateTime createdAt;

@Column(
        insertable = false,
        updatable = true
)
    private LocalDateTime lastModified;


}
```

@Builder allows us to test using a command line runner and use the field names as methods

The commandLine runner bean allows us to test different elements of our application on startup

Entity Lifecycle

14 May 2024 10:47

An entity - an object that is or is going to be wrote to a DB table

Entity states:

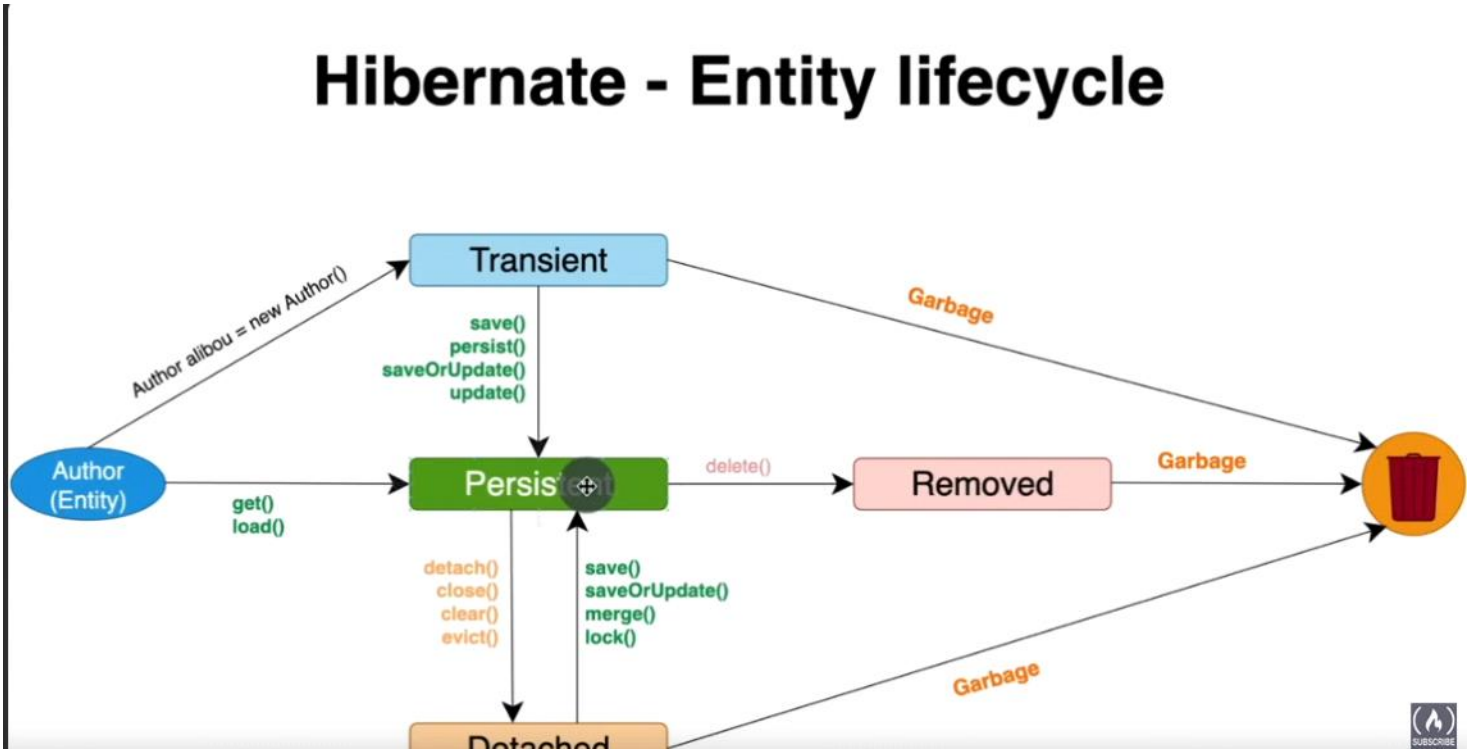
Transient - this is the state of an entity that is not managed by hibernate / wrote to DB

Persistent - Once a transient object is has save(),persist(), SaveOrUpate(),Update() run on it - The object is wrote to the DB in this state, including any changes.

Detached - an entity that was saved to DB has detch(), close(), clear() or evict() run on it. Anything we do it won't be saved to DB. We can reattach using save(), merge(), lock()

Removed - an entity that has had delete() run on it, it is in this state

Spring data JPA handles this in the background, we don't need to be concerned with state



All other state apart from Persistent go to the garbage by default

By default using get() or load() you go to persistent directly from DB

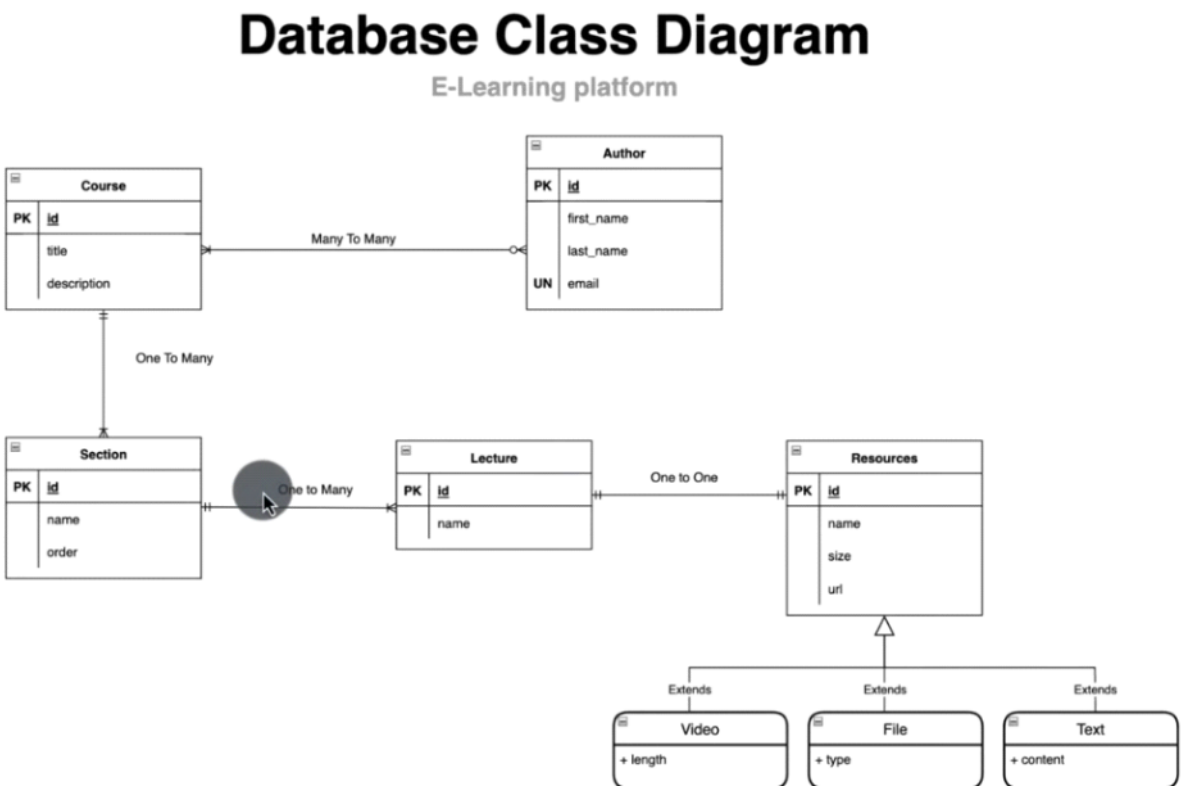
Relationships

14 May 2024 11:14

Creating relationships allows you to model the real world relationships between different pieces of data more accurately

Relationships also help with data integrity because it ensures changes are repeated across entities and are consistent

Creating relationships helps improve performance because we can avoid storing redundant data across tables by using Foreign Keys



Many to Many relationships can be done with:

Unidirectional - the relationship is only defined on one side (Course will have a Foreign Key List) only from one side - Can only navigate from Courses

Bi-directional - the relationship is defined on both sides (Courses, and Author have a Foreign Key Lists) from both sides - Can navigate from both sides however, are more complex as you have to update both object's foreign keys

Courses to Author (Many to Many)

14 May 2024 11:36

One of the entities must be Owner (JPA Many to Many). The Owner must maintain the join table and its foreign keys. Course in our case

The inverse entity (Author in our case), should be given the mapped_by attribute - @ManyToMany(mappedby="EXACT FIELD NAME OF LIST))

The owner must have @JoinTable annoation - **LOOK AT SCREENSHOT**

The joinColumn holds the primary keys of the secondary object

Author (Inverse Object):

```
@Entity
@Data
@AllArgsConstructor
@Builder
//@Table(name = "AUTHOR_TBL")
public class Author {

    @Id
    @GeneratedValue
//    @GeneratedValue(strategy = GenerationType.TABLE,
//    generator = "author_id_generator")
//    @TableGenerator(name = "author_id_generator", table = "id_generator", pkColumnName = "id_name",
//            valueColumnName = "id_value", allocationSize = 1)
    private  Integer id;

//    @Column(
//            name = "f-name",
//            length = 40
//    )
    private String firstName;
    private String lastName;
    @Column(
            unique = true,
            nullable = false
    )
    private String email;
    private Integer age;

    @ManyToMany(mappedBy = "authors") ////Exact field name of list
    private List<Course> courses;

}
```

Course (Owner):

```java
@Entity
@Data
@AllArgsConstructor
@Builder
public class Course {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    private String description;


    @ManyToMany
    @JoinTable(
            name = "authors_courses",
            joinColumns = {
                    @JoinColumn(name = "course_id")
            },
            inverseJoinColumns = {
                    @JoinColumn(name="author_id")
            }
    )
    private List<Author> authors;

}
```

Course to Section (One to Many)

14 May 2024 14:54

One course can have many sections

Many sections can be part of one course

It always the object with 'many' things attached that has the list

Course(Can have many sections):

```java
@Entity
@Data
@AllArgsConstructor
@Builder
public class Course {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    private String description;


    @ManyToMany
    @JoinTable(
            name = "authors_courses",
            joinColumns = {
                    @JoinColumn(name = "course_id")
            },
            inverseJoinColumns = {
                    @JoinColumn(name="author_id")
            }
    )
    private List<Author> authors;


    @OneToMany(mappedBy = "course")
    private List<Section> sections;
}
```

Section(Can only be assigned to one course, Owner):

```java
@Entity
@Data
@AllArgsConstructor
@Builder
public class Section {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    private int sectionOrder;


    @ManyToOne
    @JoinColumn(name = "course_id")
    private Course course;
}
```

Section to Lecture (One to many)

14 May 2024 15:16

One section can have many lectures

Section(can have many different lecturers):

```java
@Entity
@Data
@AllArgsConstructor
@Builder
public class Section {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    private int sectionOrder;


    @ManyToOne
    @JoinColumn(name = "course_id")
    private Course course;

    @OneToMany(mappedBy = "section")
    private List<Lecture> lectures;
}
```

Lecturer(can only be assigned to one section, Owner)

```java
@Entity
@Data
@AllArgsConstructor
@Builder
public class Lecture {


    @Id
    @GeneratedValue
    private Integer id;
    private String name;


    @ManyToOne
    @JoinColumn(name = "section_id")
    private Section section;

}
```

Lecture to Resource (One to One)

14 May 2024 15:24

Uni-directional:

```
@Entity
@Data
@AllArgsConstructor
@Builder
public class Lecture {


    @Id
    @GeneratedValue
    private Integer id;
    private String name;


    @ManyToOne
    @JoinColumn(name = "section_id")
    private Section section;

    @OneToOne
    @JoinColumn(name="resource_id")
    private Resource resource;

}
```

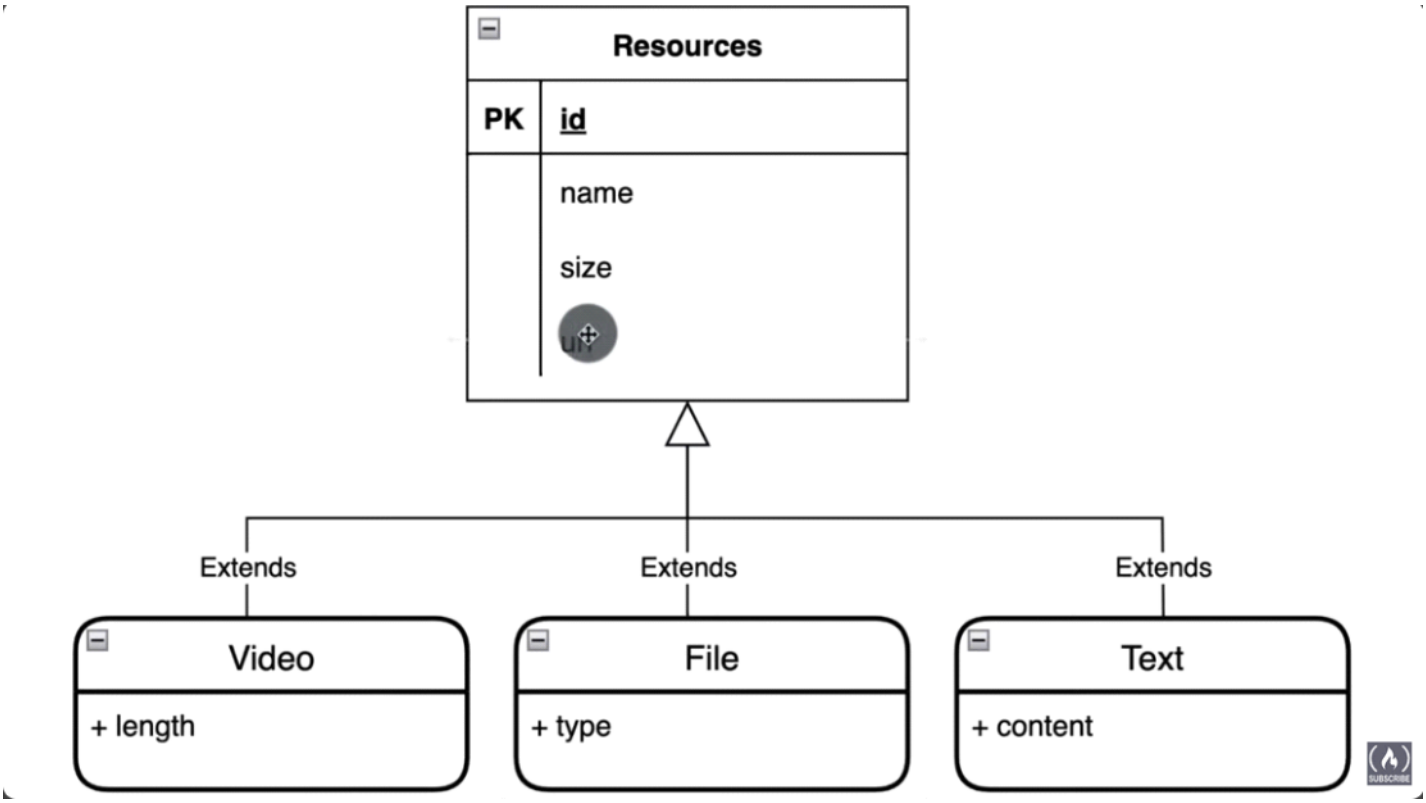
Bi-Directional:

```
@Entity
@Data
@AllArgsConstructor
@Builder
public class Lecture {


    @Id
    @GeneratedValue
    private Integer id;
    private String name;


    @ManyToOne
    @JoinColumn(name = "section_id")
    private Section section;

    @OneToOne
    @JoinColumn(name="resource_id")
    private Resource resource;

}
```

```java
@Entity
@Data
@AllArgsConstructor
@Builder
public class Resource {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;
    private int size;

    private String url;

    @OneToOne
    @JoinColumn(name="lecture_id")
    private Lecture lecture;

}
```

Object Types (Composition vs Inheritance)

14 May 2024 15:52



Basically, we can create lots of different types of resource by creating different classes that inherit from an overarching resource class.

In spring data JPA you can create a base class of your entities then create a sub-class of your entities.

Useful to avoid duplation

Pros of inheritance:

- Çode reuse
- Better queries (you can write queries that check various resource types)

Cons of inheritance:

- Makes code more complex
- Inheritance makes code less dynamic because quick changes are harder

You should avoid using inheritance and use composition instead because its less complex and more flexible because you can change things without changing the class itself.

:
Composition:

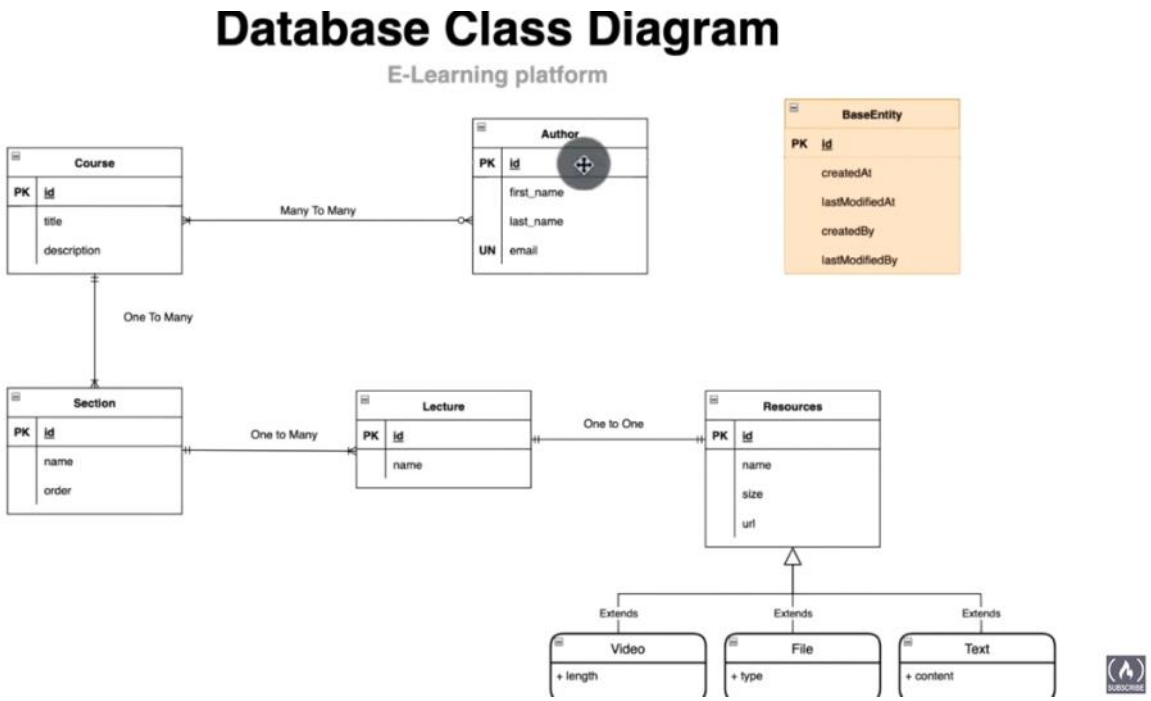
https://www.digitalocean.com/community/tutorials/composition-in-java-example

Adding object inheritance to our entities

14 May 2024 17:41

Remember, we should use Composition instead.

This will involve creating a parent class with our common fields within



We must use the @mappedSuperClass annotation for this

We must create a superclass base entity within our code, this won't be saved to the db but will be an abstraction within our code

This base entity cannot be queried because it doesn't exist in the db

Anything that inherits from a mappedSuperClass will get its fields

All Spring annotations are available here - apart from when using builder should use @SuperBuilder

Note all the annoations used

All entitys that use a parent class must use @SuperBuillder for builder

BaseEntity (Parent Class):

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@MappedSuperclass
public class BaseEntity {

    @Id
    @GeneratedValue
//    @GeneratedValue(strategy = GenerationType.TABLE,
//    generator = "author_id_generator")
//    @TableGenerator(name = "author_id_generator", table = "id_generator", pkColumnName = "id_name",
//            valueColumnName = "id_value", allocationSize = 1)
    private  Integer id;

    private LocalDateTime createdAt;

    private LocalDateTime lastModifiedAt;

    private String createdBy;

    private String lastModifiedBy;


}
```

Author (For example, inheriting from base Entity):

```java
@Data
@EqualsAndHashCode(callSuper = true)
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@Entity
//@Table(name = "AUTHOR_TBL")
public class Author extends BaseEntity {



//    @Column(
//            name = "f-name",
//            length = 40
//    )
    private String firstName;
    private String lastName;
    @Column(
            unique = true,
            nullable = false
    )
    private String email;
    private Integer age;

    @ManyToMany(mappedBy = "authors") ////Exact field name of list
    private List<Course> courses;

}
```

Single table Inheritance

14 May 2024 18:42

Single table inheritance is used in JPA to ensure that all sub classes of the inherited class as mapped to the same table

A discriminator column contains information about what row in the table belongs to what subclass. This is required for single table inheritance.

This is the simplest way to have persistent inheritance

However it can lead to large tables and complex queries, only use on small datasets and non-deep inheritance.

@Inheritance(STRAT) - used to set our inheritance strategy

@DiscriminationColum - allows us to customize the column including name

@DiscrimnationValue - allows us to control the content given to column by our entities that inherit

Base class:

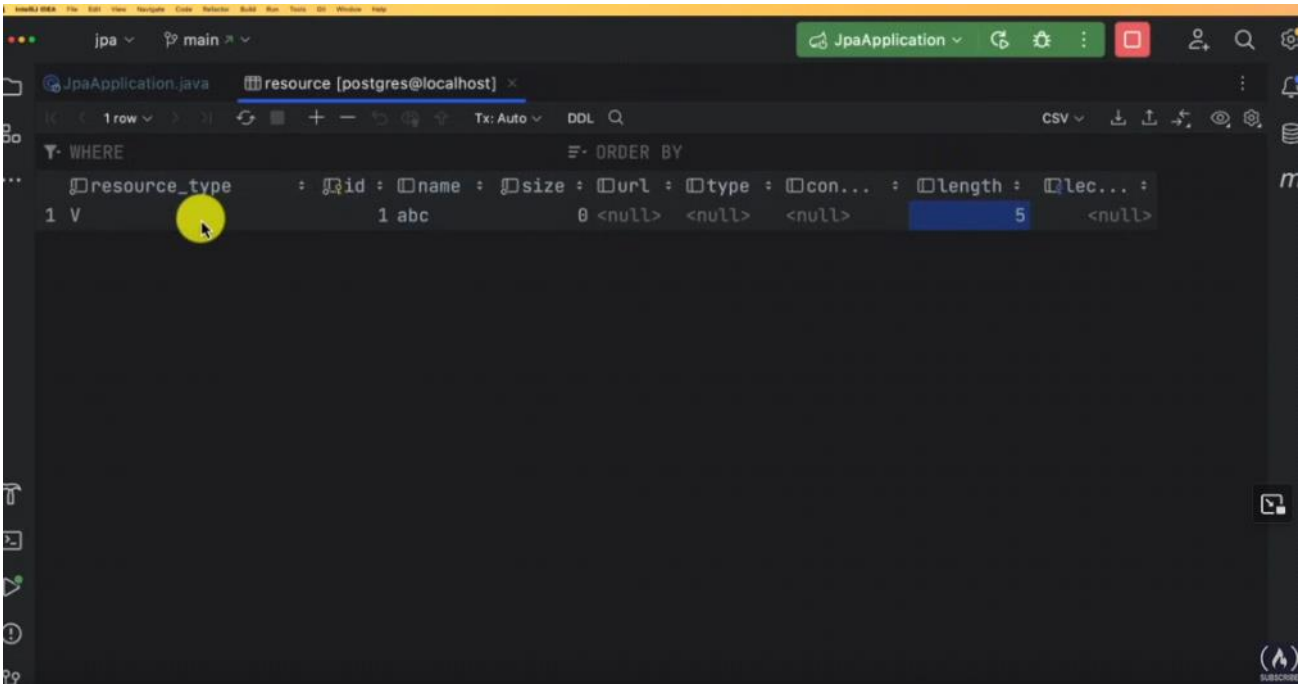
```
@Data
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "resource_type")
public class Resource {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;
    private int size;

    private String url;

    @OneToOne
    @JoinColumn(name="lecture_id")
    private Lecture lecture;

}
```

Sub class (example):

```java
@EqualsAndHashCode(callSuper = true)
@Data
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@Entity
@DiscriminatorValue("F")
public class file extends Resource {

    private String type;
}
```

Result in Database & Creation:

```java

    //@Bean
    no usages
    public CommandLineRunner commandLineRunner(
            AuthorRepository repository,
            videoRepository videoRepository
    ) {
        return args -> {
//          Author author = Author.builder().firstName("matt")
//                  .lastName("butler")
//                  .age(19)
//                  .email("mattbutler0001@gmail.com")
//                  .build();
//          repository.save(author);

            video video = video.builder() VideoBuilder<capture of ?, capture of ?>
                    .name("abc") capture of ?
                    .length(6).build();
            videoRepository.save(video);
        };

    }
```


jpa
main
JpaApplication
JpaApplication.java
resource [postgres@localhost]
1 row
Tx: Auto
DDL
CSV
WHERE
ORDER BY
resource_type : id : name : size : url : type : con... : length : lec... :
1 V 1 abc 0 <null> <null> <null> 5 <null>

Joined Table Inheritance

15 May 2024 05:37

Each sub-class is mapped to a separate table with a foreign key pointing to the base table

These tables only contains the values that is required by the sub-class + the foreign key

This is more efficient for queries, but it means you need more tables and foreign keys, making DB more complex

The joined table is a good choice when you have a lot of sub-classes, with differences in their properties and you want quick queries

However, you can't query every inherited class

@DiscriminatorColumn, @DiscriminatorValue is only needed for single table

@PrimaryKeyJoinColumn can be used to rename the join colmun

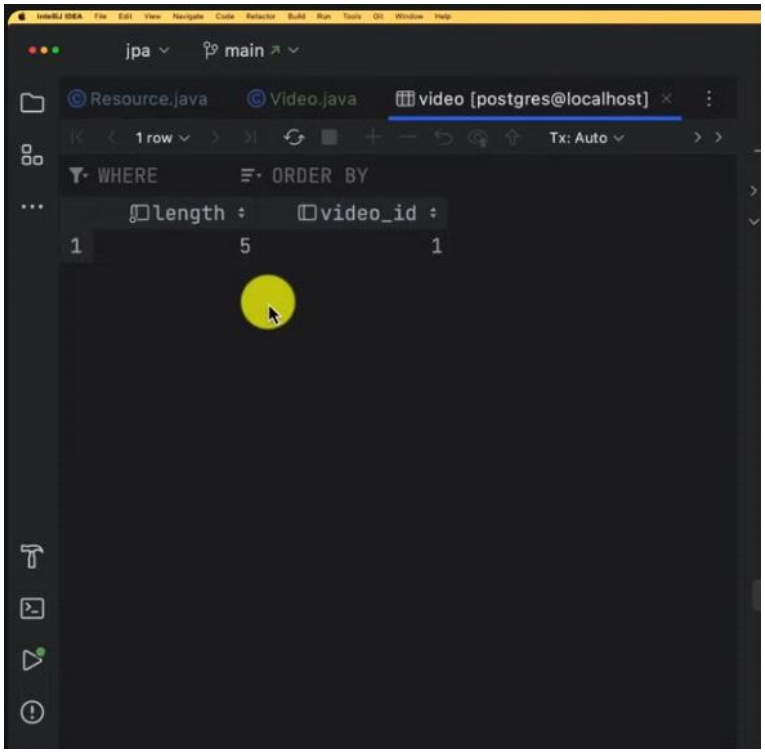
@Inheritance Strat change to JOINED

Base Class (Resource):

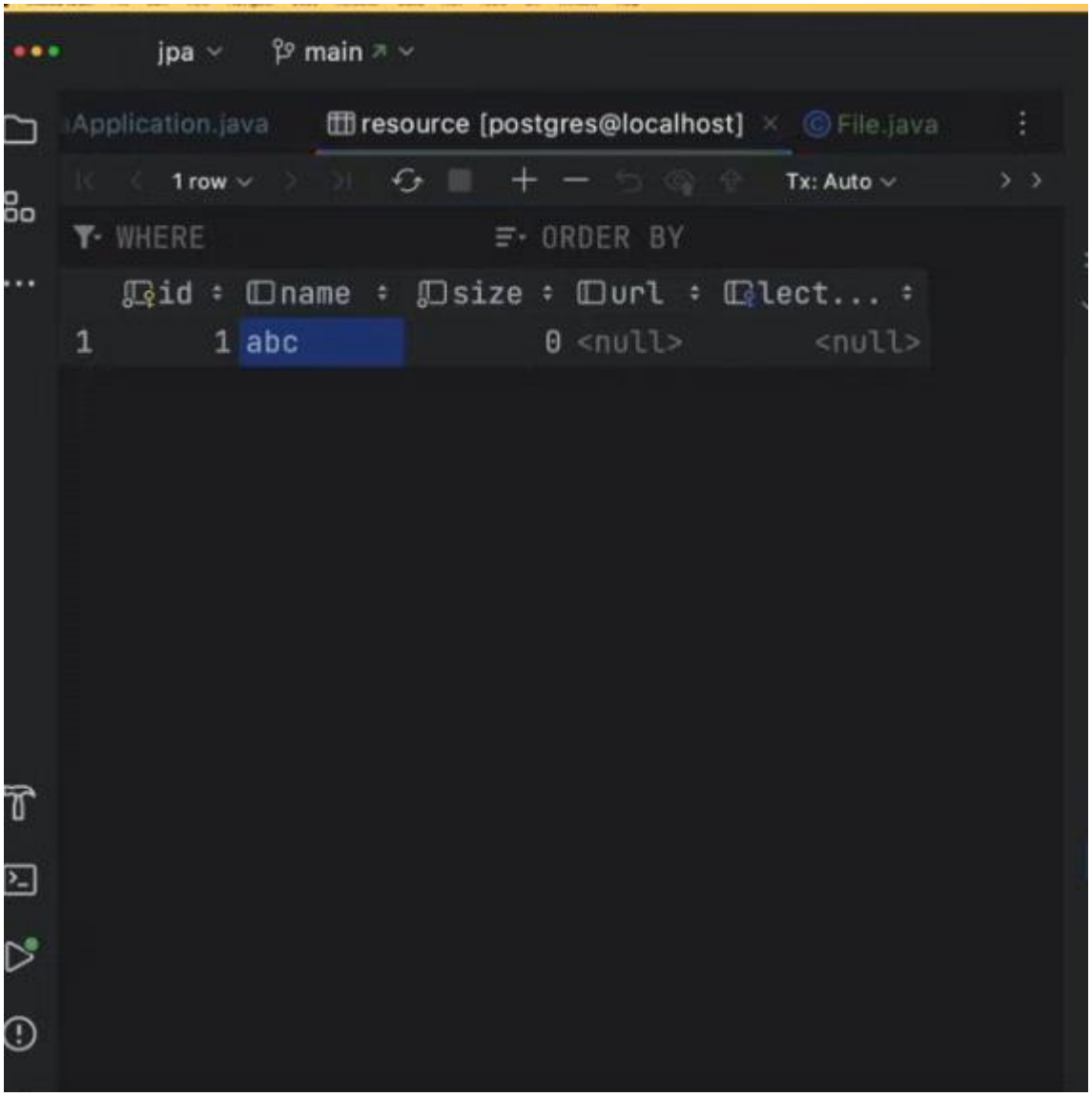
```
@Data
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
//@DiscriminatorColumn(name = "resource_type")
public class Resource {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;
    private int size;

    private String url;

    @OneToOne
    @JoinColumn(name="lecture_id")
    private Lecture lecture;

}
```

Video (Sub class example):

```java
@EqualsAndHashCode(callSuper = true)
@Data
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@Entity
//@DiscriminatorValue("V")
public class Video extends Resource {

    private int length;


}
```



Database result:



Resource table has our video (id is foreign + primary key, Joined)

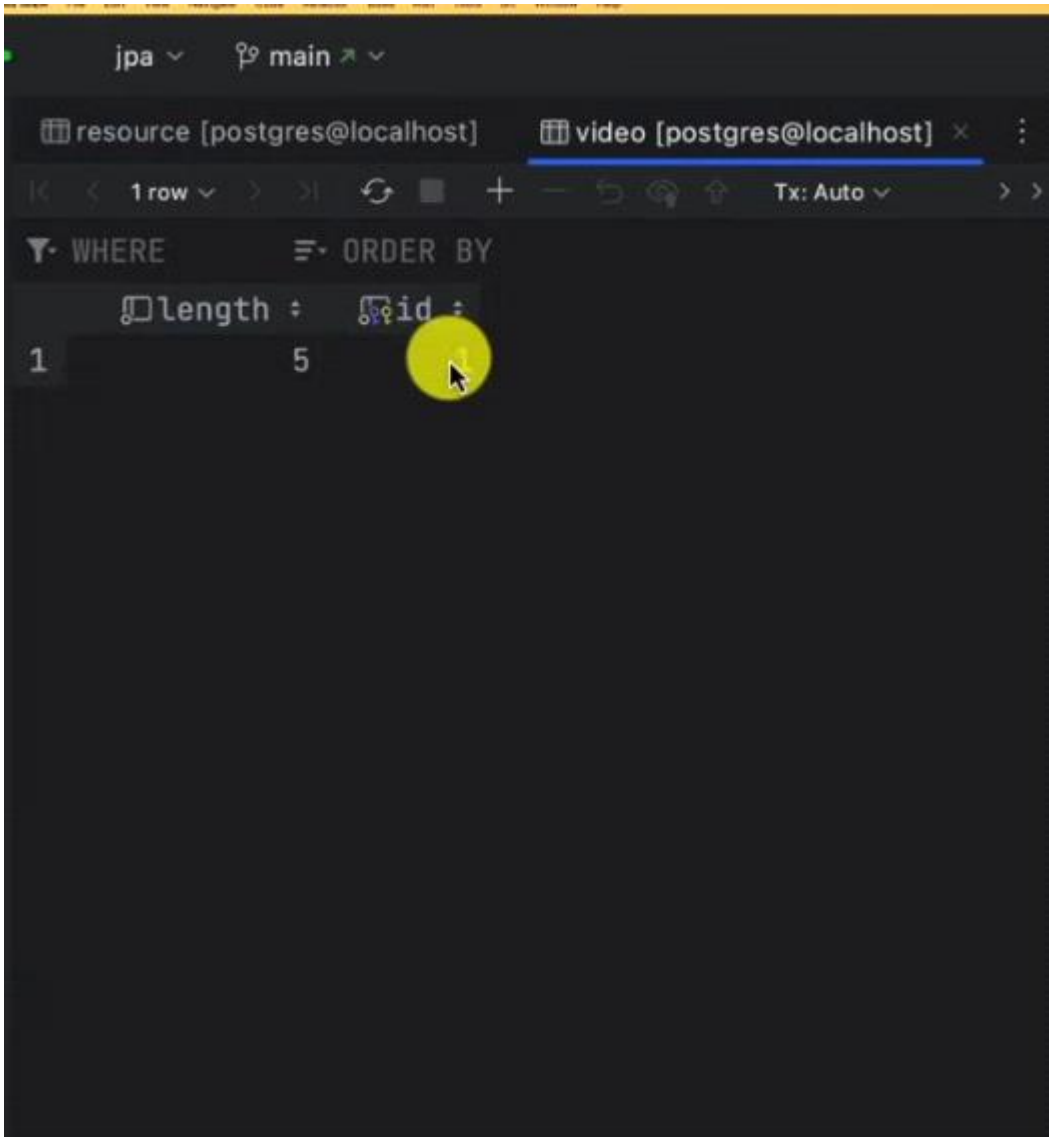

jpa
main
resource [postgres@localhost]
video [postgres@localhost]
1 row
Tx: Auto
WHERE
ORDER BY
length
id
1
5

Table Per Class Inheritance

15 May 2024 05:55

Table per class puts all the attributes of the base class in the sub-class table. It copies them over

This is the most efficent queries

But it makes the DB schema more complex and larger

Good for a small number of sub-classes with big differences

And when queries are important

Its not suitable for needing to query all sub-classes because you must query a lot of different tables

@Inheriteance Strat changed to Table_per_class that's it

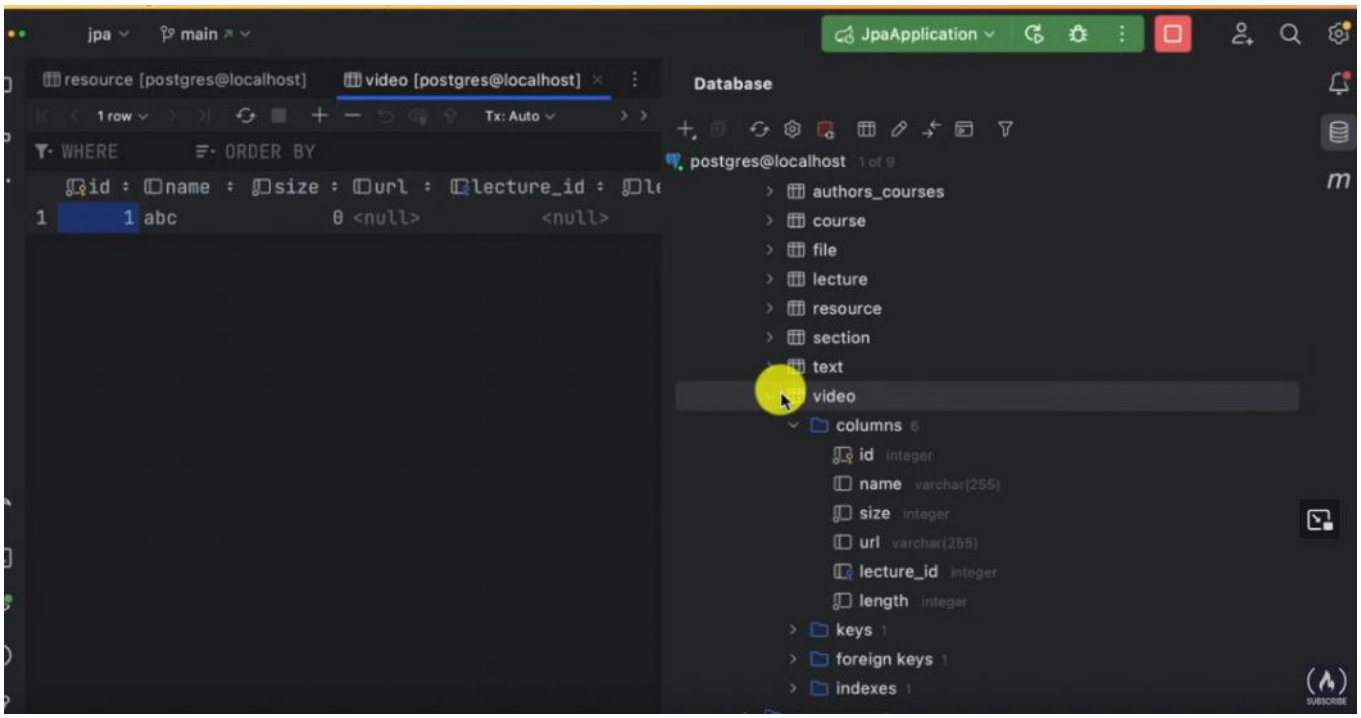
Base Class (Resource):

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
//@DiscriminatorColumn(name = "resource_type")
public class Resource {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;
    private int size;

    private String url;

    @OneToOne
    @JoinColumn(name="lecture_id")
    private Lecture lecture;

}
```

Video (Sub Class):

```java
@EqualsAndHashCode(callSuper = true)
@Data
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@Entity
//@PrimaryKeyJoinColumn(name = "video_id")
//@DiscriminatorValue("V")
public class Video extends Resource {

    private int length;


}
```

Effect on DB:



Video contains a copy of all fields

Inheritance base query + issues

15 May 2024 06:02

The base class (resource) can be queried using a Repository like normal

If you want to query the base class and just get its fields.

This is done using a union statement. Which is slow.

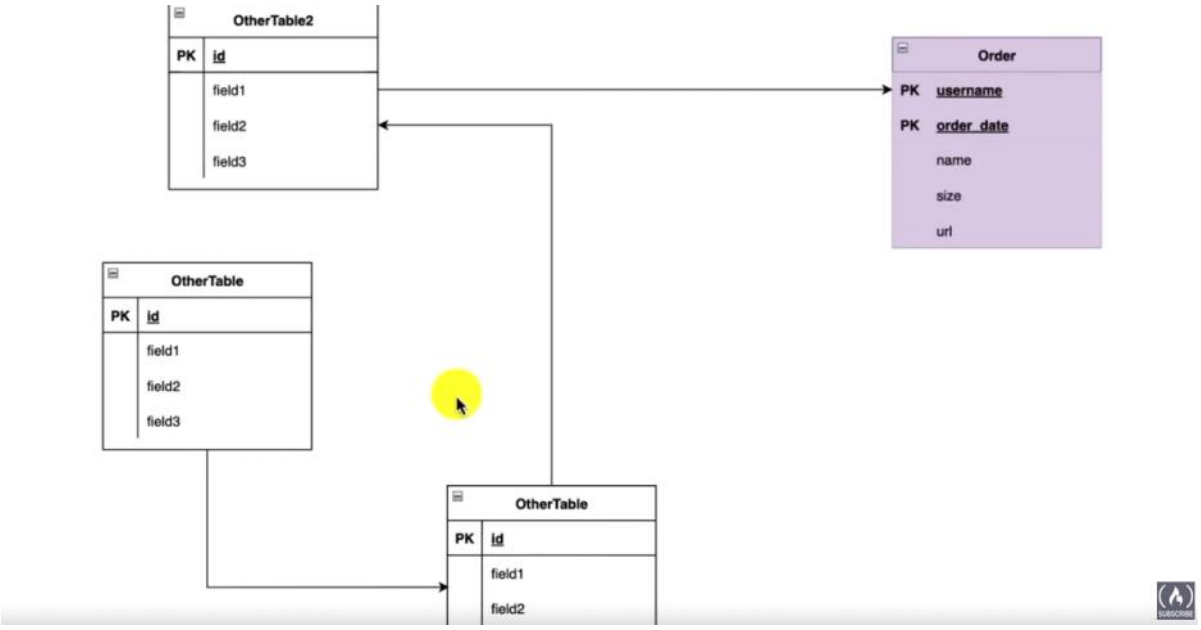
We also can't use identify id generation

If we don't want it to return the query in type of base class (in our case resource), instead in our type (video, file, text) do this:

```
@EqualsAndHashCode(callSuper = true)
@Data
@NoArgsConstructor
@AllArgsConstructor
@SuperBuilder
@Entity
//@PrimaryKeyJoinColumn(name = "video_id")
//@DiscriminatorValue("V")
@Polymorphism(type = PolymorphismType.EXPLICIT)
public class Video extends Resource {

    private int length;


}
```

Embedded ID (Composite IDs)

15 May 2024 06:07



Composite ID's are made by of a combination of values to create an ID

We can then decode this to get infomation

Use case:

If we have a microservice that produces Order Entities

The composite ID could be made up of Username and OrderDate

Because these are unique values when combinded

We must use @Embeddedable to tell spring data that this is an embbded entity

Composite ID:

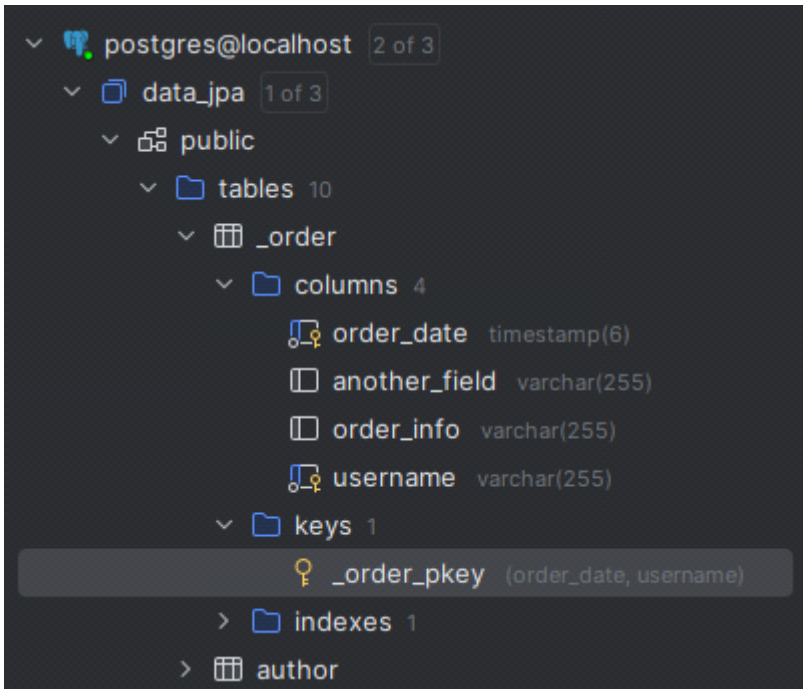
```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Embeddable
public class orderId implements Serializable {

    private String username;

    private LocalDateTime orderDate;

}
```

This Id must implement java serializable

Use of EmbbededID (order):

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class order {

    @EmbeddedId
    private orderId id;

    private String orderInfo;
    private String anotherField;
}
```

Result in database:



Embedded Fields (Address)

15 May 2024 06:55

We can use Embedded fields to store objects instead of doing inheritance using composition

Embbededable entities are good for code reusability and maintainability

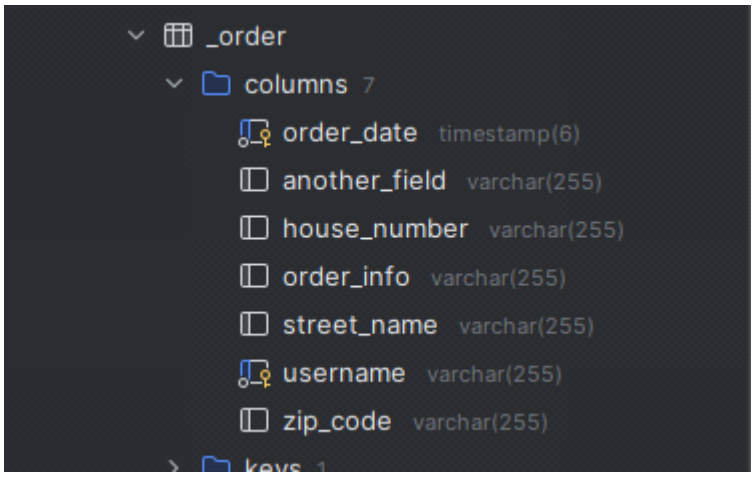
Address (Embedded object):

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Embeddable
public class Address {

    private String streetName;

    private String houseNumber;

    private String zipCode;
}
```

Embedded into Order:

```
no usages
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name="_order")
public class order {

    @EmbeddedId
    private orderId id;

    @Embedded
    private Address address;

    private String orderInfo;
    private String anotherField;
}
```

Effect on Database (fields get stored embedded into table within):



Derivative query methods

15 May 2024 07:05

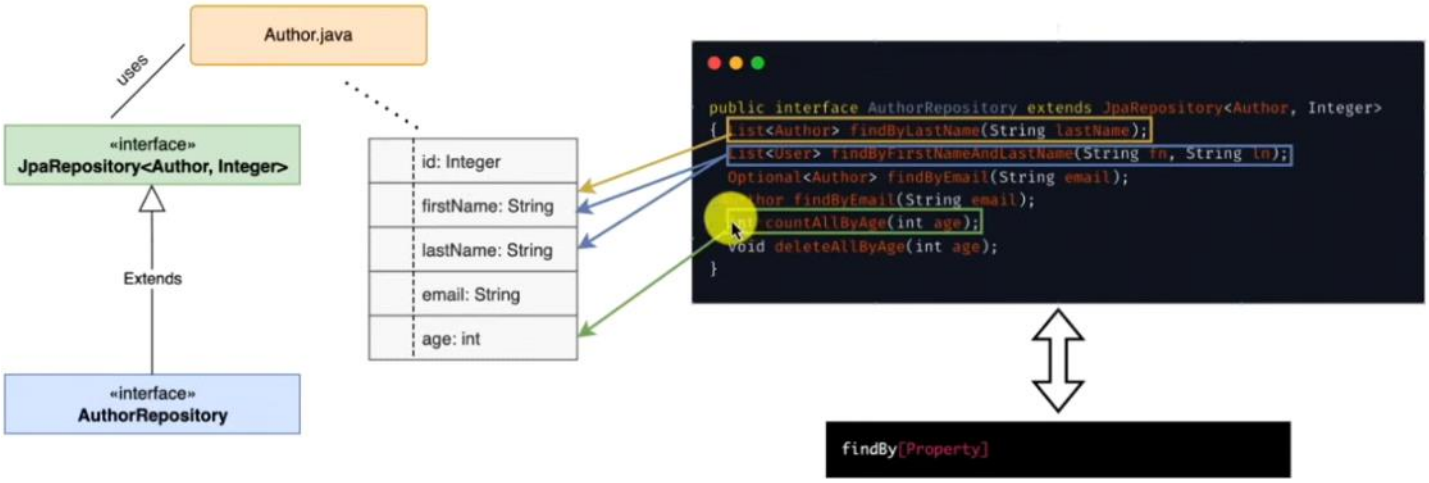
Derivative query methods allow you to define queries based on the query name

Spring will then generate the correct SQL queries

This done using interfaces:

RETURN-TYPE QUERY-WITH-OPPERATION (REQUIRED QUERY INPUT)

QUERY WITH OPERATION: DeleteBy, CountAll,FindAll, FindBy



https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

When using FindByX, and not all etc, we must use Optinal<> Generics

https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

Queries:

Containing means 'has input this within'

You can query for multiple values at once CHECK SCREENSHOT

```
@Repository
public interface AuthorRepository extends JpaRepository<Author, Integer> {

    //Select * from author where first_name = 'al'
     no usages
     List<Author> findAllByFirstName(String fn);


     // select * from author where first_name like al%
     no usages
     List<Author> findAllByFirstNameIgnoreCase(String fx);

     // select * from author where first-name like 'al%'
     no usages
     List<Author> findAllByFirstNameContainingIgnoreCase(String fx);



    no usages
    List<Author> findAllByFirstNameStartsWithIgnoreCase(String fn);

    // select * from author where first_name like 'al%'
    no usages
    List<Author> findAllByFirstNameEndingWithIgnoreCase(String fn);

    // select * from author where firstname in ("ali', 'bou', 'coding'


    no usages
    List<Author> findAllByFirstNameInIgnoreCase(List<String> firstnames);


}
```

Faker (faking data)

15 May 2024 08:12

Faker is used to generate fake data in java including names, addresses

```
<dependency>
    <groupId>com.github.javafaker</groupId>
    <artifactId>javafaker</artifactId>
</dependency>
```

```
@SpringBootApplication
public class JpaApplication {

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

    @Bean
    public CommandLineRunner commandLineRunner(
            AuthorRepository repository,
            videoRepository videoRepository
    ) {
        return args -> {
            for(int i =0; i < 50; i++) {
                Faker faker = new Faker();
                Author author = Author.builder().firstName(faker.name().firstName())
                        .lastName(faker.name().lastName())
                        .age(faker.number().numberBetween(19,50))
                        .email(faker.name().username()+ "@gmail.com")
                        .build();
                repository.save(author);
            }
```

Repository Save method

15 May 2024 08:33

When calling .save() using a repository, if the object you are attempting to save exists (given its ID) the existing one will be modified instead of making a new one

Below id=1 object already exists so its fields were changed to match our input:

```
Author authorOne = Author.builder() AuthorBuilder<capture of ?, capture of ?>
        .id(1) capture of ?
        .firstName("matt")
        .lastName("butler")
        .age(19)
        .email("mattbutler0001@mail.com")
        .build();
repository.save(authorOne);
```

```
Hibernate: insert into author (age,created_at,created_by,email,first_name,last_modified_at,last_modified_by,last_name,id) values (?,?,?,?,?,?,?,?,?)
Hibernate: insert into author (age,created_at,created_by,email,first_name,last_modified_at,last_modified_by,last_name,id) values (?,?,?,?,?,?,?,?,?)
Hibernate: select a1_0.id,a1_0.age,a1_0.created_at,a1_0.created_by,a1_0.email,a1_0.first_name,a1_0.last_modified_at,a1_0.last_modified_by,a1_0.last_name from author a1_0 where a1_0.id=?
Hibernate: update author set age=?,created_at=?,created_by=?,email=?,first_name=?,last_modified_at=?,last_modified_by=?,last_name=? where id=?
```

Custom Queries

15 May 2024 11:27

Using @Query annotation we can write custom queries in SQL

Very useful for bulk updates

Make sure if its an updating query to include @Modifying

We must make it transactional to make a custom SQL query work @Transactional

AuthorRepository Custom Query (update age):

```
@Repository
public interface AuthorRepository extends JpaRepository<Author, Integer> {

    //Select * from author where first_name = 'al'
     no usages
     List<Author> findAllByFirstName(String fn);


     // select * from author where first_name like al%
     no usages
     List<Author> findAllByFirstNameIgnoreCase(String fx);

     // select * from author where first-name like 'al%'
     no usages
     List<Author> findAllByFirstNameContainingIgnoreCase(String fx);



    no usages
    List<Author> findAllByFirstNameStartsWithIgnoreCase(String fn);

    // select * from author where first_name like 'al%'
    no usages
    List<Author> findAllByFirstNameEndingWithIgnoreCase(String fn);

    // select * from author where firstname in ("ali', 'bou', 'coding'

    no usages
    List<Author> findAllByFirstNameInIgnoreCase(List<String> firstnames);

    //Custom SQL Query
    1 usage
    @Modifying
    @Transactional
    @Query("update Author a set a.age = :age where a.id = :id")
    int updateAuthor(int age, int id);


}
```

Usage:

```
//Changing the age using custom query SQL
repository.updateAuthor( age: 24, id: 1);
```

Mass Update Query

15 May 2024 11:38

```
//Update all authors using custom SQL query
repository.updateAllAUthors( age: 99);

};
```

```
@Modifying
@Transactional
@Query("update Author a set a.age = :age")
void updateAllAUthors(int age);
}
```

@NamedQueries usecases

15 May 2024 11:42

@NamedQueries are used for organising and maintaining query definitions

Common use:

- Encapsulation
- Reusability
- Help to optimize performance by validation and optimization during startup
- Centralized, stored in one place

Use case:

- When you have complex queries that are reused
- When you want to optimize performance for frequently used queries
- When you want to improve readability
- When you want to improve organisation

NamedQueries are not the best choice if you need flexibility or lots of user input

@NamedQuery

15 May 2024 11:49

Read above for more Info

@NamedQueries is just a way to organise @namedQuery

@NamedQuery should be installed within entity

@NamedQuery require a name and a query

@Param is used for repository input, should match SQL input

Author (where named query applied):

When updating we always need Transactional, and Modifying annotations

@Param is used for repository input, should match SQL input

```
@NamedQuery(
        name = "Author.findByNamedQuery",
        query = "select a from Author a where a.age >= :age"
)

public class Author extends BaseEntity {
```

in Author repository:

```
@Repository
public interface AuthorRepository extends JpaRepository<Author, Integer> {

    1 usage
    @Transactional
    List<Author> findByNamedQuery(@Param("age") int age);

    1 usage
    @Modifying
    @Transactional
    void updateByNamedQuery(@Param("age") int age);
```

```
@NamedQuery(
        name = "Author.updateByNamedQuery",
        query = "update Author a set a.age = :age"
)



public class Author extends BaseEntity {
```

```
@Repository
public interface AuthorRepository extends JpaRepository<Author, Integer> {

    1 usage
    @Transactional
    List<Author> findByNamedQuery(@Param("age") int age);

    1 usage
    @Modifying
    @Transactional
    void updateByNamedQuery(@Param("age") int age);
```

Specification Queries

15 May 2024 12:22

This is very reusable and dynamic

Spring Specification Queries allow you to build complex queries that are flexible using the Specification interface

You must extend your repository with Specification to be able to access common methods like findAll and FindOne

We can use logical operators like 'and, or'

You can reuse specification and its good for complex queries that require user input

Root holds the entity information throughout our query

```
return builder.equal(root.get("age"), age);
```

Root.get() - Means get all attributes with age then builder.equal will check them for correct age within our specification

Builder. - You can use any SQL method in your specification

When using params you must use % %

You must extend Repostitory with jpaSpec first:

```
2 usages
@Repository
public interface AuthorRepository extends JpaRepository<Author, Integer>, JpaSpecificationExecutor<Author> {
```

Example Specification:

```
public class AuthSpecification {

    1 usage
    public static Specification<Author> hasAge(int age) {
        return (Root<Author> root, CriteriaQuery<?> query, CriteriaBuilder builder) -> {
            if(age < 0) {
                return null;
            }
            return builder.equal(root.get("age"), age);
        };
    }
    1 usage
    public static Specification<Author> firstnameContains(String firstname) {
        return (Root<Author> root, CriteriaQuery<?> query, CriteriaBuilder builder) -> {
            if(firstname == null) {
                return null;
            }
            return builder.like(root.get("firstName"), pattern: "%" + firstname +"%");
        };
    }
}
```

Usage:

```
Specification<Author> spec = Specification
        .where(AuthSpecification.hasAge(34).and(AuthSpecification.firstnameContains("Matt")));

repository.findAll(spec).forEach(System.out::println);

};
```

Will look for Matt at age 34

You don't have to use the entire spec in your code

e.g call without using 'firstnameContains'

```
Specification<Author> spec = Specification
        .where(AuthSpecification.hasAge(34)
        //.and(AuthSpecification.firstnameContains("Matt")
        );

repository.findAll(spec).forEach(System.out::println);
```

Or you can switch the logical operator (or example):

```
Specification<Author> spec = Specification
        .where(AuthSpecification.hasAge(34)
        .or(AuthSpecification.firstnameContains("Matt")
        ));

repository.findAll(spec).forEach(System.out::println);

};
```

This is what makes it dynamic and reusable