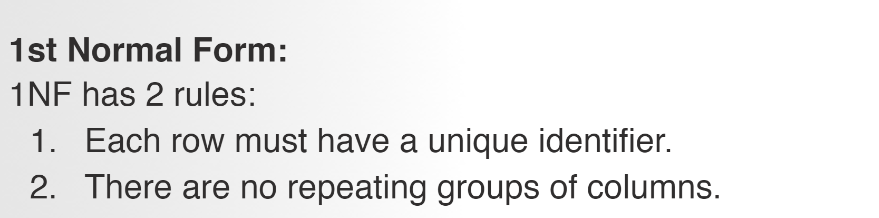
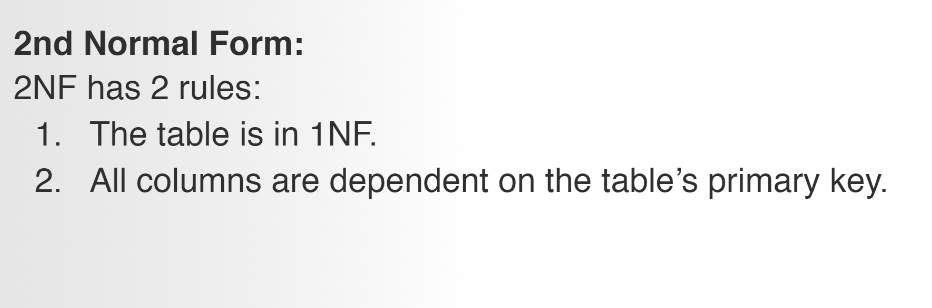
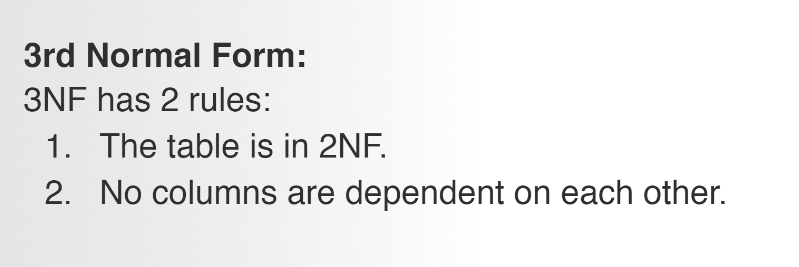
07/17/2019

* *@NotNull:* a constrained *CharSequence*, *Collection*, *Map,* or *Array* is valid as long as it’s not null, but it can be empty
* *@NotEmpty:* a constrained *CharSequence*, *Collection*, *Map,* or *Array* is valid as long as it’s not null and its size/length is greater than zero
* *@NotBlank:* a constrained *String* is valid as long as it’s not null and the trimmed length is greater than zero







Sql words in uppercase

Create database <databasename>;

CREATE DATABASE students\_db;

2 dash – comment

Ends with semicolon

USE students\_db;

CREATE TABLE students\_db(

Student\_name VARCHAR(255) NOT NULL,

Student\_age INTEGER,

Is\_enrolled BOOLEAN default true,

Date\_enrolled DATE

};

insert INTO students(student\_name, student\_age, is\_enrolled, Date\_enrolled)

VALUES ("Emma", 20, true, "2018-9-18");

Date is in double quotes

update students SET student\_name = "Anna" WHERE student\_age = 17;

DELETE FROM students

where student\_name = "Emma"

adding a primary key to a column name

CREATE TABLE restaurants (

id INTEGER primary key,

Restaurant\_name VARCHAR(255),

Restaurant\_type VARCHAR(255),

Permanently\_closed BOOLEAN DEFAULT FALSE

);

CREATE TABLE Restaurant (

id INTEGER auto\_increment not null,

Restaurant\_name VARCHAR(255),

Restaurant\_type VARCHAR(255),

Permanently\_closed BOOLEAN DEFAULT FALSE,

primary key(id)

);

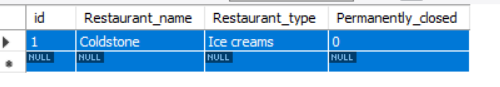
INSERT INTO Restaurant (Restaurant\_name,Restaurant\_type, Permanently\_closed)

VALUES

("Coldstone", "Ice creams", FALSE);

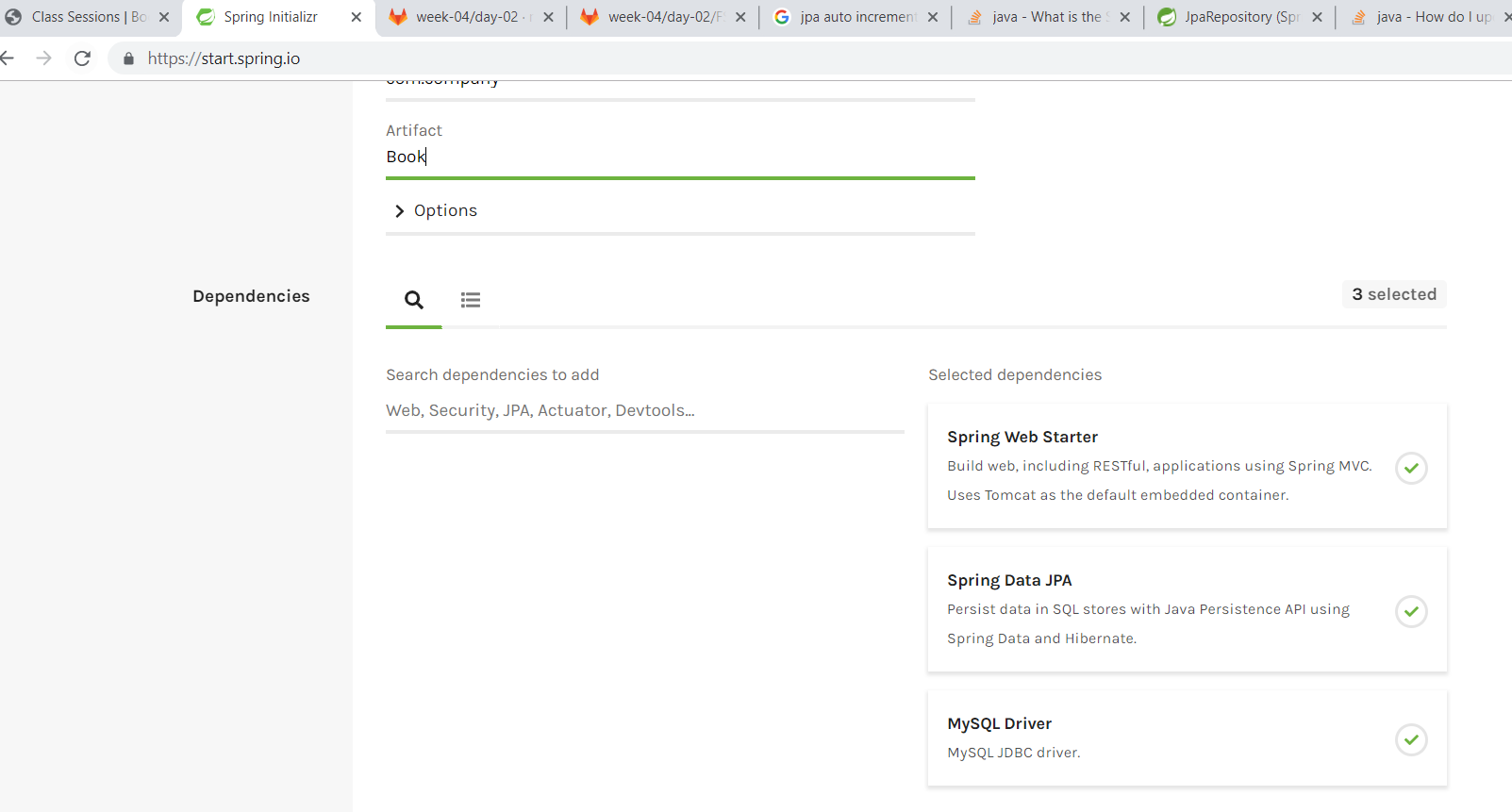
select \*

from Restaurant;



Auto\_increment gives value to the corresponding column…user don’t need to enter it

**JPA:**



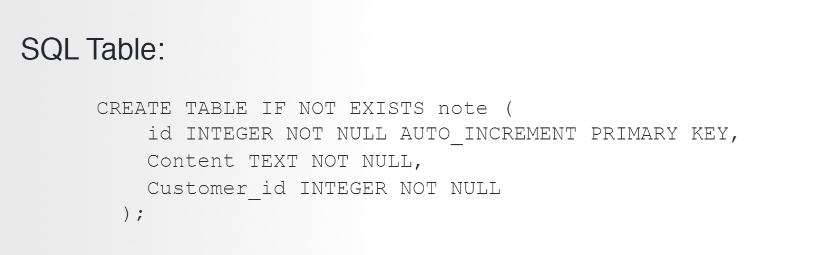
Web starter, JPA & mysql

Java class -> table

@Entity – class – table

For DTO – primitive data types cannot be used—have to use the class data types – Integer,

**private** Integer **id**; *// row attribute can have null so int cant have null whereas class Integer can be nulled*



The above table will be created in mysql with the below java

@Entity  
@JsonIgnoreProperties({**"hibernateLazyInitializer"**, **"handler"**})  
@Table(name = **"note"**)  
**public class** Note {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***AUTO***)  
 **private** Integer **id**; *// row attribute can have null so int cant have null whereas class Integer can be nulled* @NotBlank  
 **private** String **Content**;  
  
 @NotBlank  
 **private** Integer **Customer\_id**;

}

**Application.properties file under main:**

**This contains connection information**

spring.datasource.url=jdbc:mysql://localhost:3306/simple\_crm ---database name, it has to be created

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.jpa.hibernate.ddl-auto=create ---table is created if it is not avbl

spring.jpa.show-sql=true

1. Create the database in mysql
2. Create a table that you wanna create in a java class in intelli j with the structure of how want it
3. Specify your database information in **Application.properties file under main: In Resources folder**
4. Create an interface with the extension of

File: CusomerRepository --- Interface-🡪 interacts with d/b - dao

Here you declare all the custom methods that are going to be used in the Controller - rest

**public interface CusomerRepository** **extends** JpaRepository<Customer, Integer> { // JpaRepository<Customer, Integer> -- here Customer is the table name and the next Integer says the type of the primary key of the table  
  
 List<Customer> findBylastNameAnd\_company(String lastName, String company ); *// declare in the interface if it is  
 // not there in the JPA. We do not need to declare this method if we just want to find by last name and company. When we use this method in the api route file, Springboot automatically builds that definition for us based on the format.* Customer findByid(Integer id);  
}

Interface extends JPARepository which has all the methods

[List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html?is-external=true)<[T](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html)> findAll()

**Specified by:**

[findAll](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html?is-external=true#findAll--) in interface [CrudRepository](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html?is-external=true" \o "class or interface in org.springframework.data.repository)<[T](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html),[ID](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html)>

This lists all the table’s content

<https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html>

1. Last is the controller file that has all the rest api routes where we use the Interface’s methods to interact with the database

@RestController  
**public class** CustomerController {  
  
 @Autowired  
 **private CusomerRepository** **customerRepo**;

**public** Motorcycle insertRecordIntoMotorcycle(@RequestBody @**Valid** Motorcycle motorcycle){  
 **motorcycleRepository**.save(motorcycle);  
 **return** motorcycle;  
}

@Valid – in @RequestBody verifies if it matches all the constraint when we defined the table. This is done for mostly Insert and update

All private members wont be accessible in subclass..have to put protected

@AutoWired

@Valid

Patch

**Api – web layer – controller**

Gets request and sends response

Data layer – d/b- gets data from d/b – DTO, DAO

**Service layer:**

Business logic

@Component – class

**Parent table:**

@OneToMany(mappedBy = **"roasterId"**, cascade = CascadeType.***ALL***, fetch = FetchType.***EAGER***)  
**private** Set<Coffee> **coffees**;

@NotEmpty -> For strings. It includes Not null and should be > 0

@NotNull -> For all integers, double

Package names – Lower case

**public class** Author {

@OneToMany(mappedBy=**"author"**, cascade=CascadeType.***ALL***, fetch=FetchType.***EAGER***)  
**private** Set<Book> **books**;

}

**public class** Book {

@ManyToOne(fetch=FetchType.***EAGER***)  
@JoinColumn(name=**"authorId"**)  
**private** Author **author**;

}

<https://hellokoding.com/jpa-one-to-many-relationship-mapping-example-with-spring-boot-hsql/>

@Table maps the entity with the table. If no @Table is defined, the default value is used: the class name of the entity.

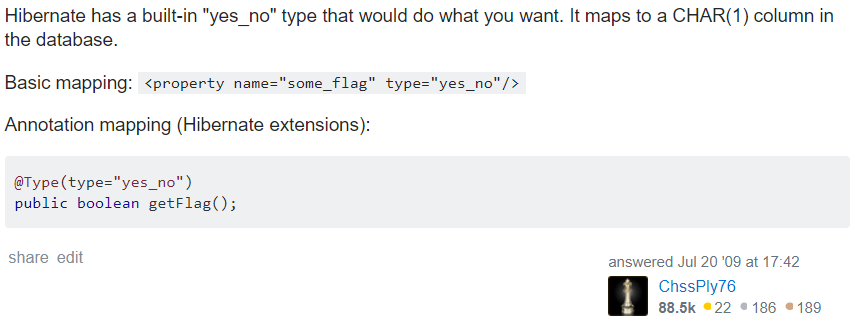
@Id declares the identifier property of the entity.

@Column maps the entity's field with the table's column. If @Column is omitted, the default value is used: the field name of the entity.

@OneToMany and @ManyToOne defines a one-to-many and many-to-one relationship between 2 entities. @JoinColumn indicates the entity is the owner of the relationship: the corresponding table has a column with a foreign key to the referenced table. mappedBy indicates the entity is the inverse of the relationship.

So what does @Transactional mean if you annotate your test suite with it? Well it means that every test method in your suite is surrounded by an overarching Spring transaction. This transaction will be rolled back at the end of the test method regardless of it's outcome.

<dependency>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-devtools</artifactId>  
            <optional>true</optional>  
        </dependency>



<https://stackoverflow.com/questions/1154833/configure-hibernate-using-jpa-to-store-y-n-for-type-boolean-instead-of-0-1>

To make the server run in different port, add the below in application.properties

server.port=8081