* 3.2 PERSISTING DATA WITH SPRING DATA JPA
  + 3.2.1ADDING SPRING DATA JPA TO THE PROJECT
    - JPA starter not only brings Spring Data JPA but also includes Hibernate as a JPA implementation

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-data-jpa</**artifactId**>

</**dependency**>

* + - If you want to use a different JPA implementation, you will need to remove Hibernate dependency and include JPA library of your choice.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

<exclusions>

<exclusion>

<artifactId>hibernate-entitymanager</artifactId>

<groupId>org.hibernate</groupId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.eclipse.persistence</groupId>

<artifactId>eclipselink</artifactId>

<version>2.5.2</version>

</dependency>

* + 3.2.2 ANNOTATING THE DOMAIN AS ENTITIES
    - Here is the code for Ingredient class:

package tacos;

import javax.persistence.Entity;

import javax.persistence.Id;

import lombok.AccessLevel;

import lombok.Data;

import lombok.NoArgsConstructor;

import lombok.RequiredArgsConstructor;

//generates getters, hashcode(), toString(),constructors, setters, equals()

@Data

/\*

\* //generates constructor with required arguments that has all final properties

\* as arguments. @Data implicitly adds this constructor but when

\* noArgsConscturtor is used, this constructor gets removed. An

\* explicit @RequiredArgsConstructor ensures that you will have both required

\* and no required argument constructor

\*/*@RequiredArgsConstructor*

/\*

\* JPA requires entities has a no argument constructor, so

\* lomboks @NoArgsConsctructor does that. It must not be usable so you make it

\* private. There are final properties that needs to be set so you set force =

\* true which makes their values to null.

\*/

@*NoArgsConstructor*(access = *AccessLevel*.***PRIVATE***, force = true)

/\* Declaring this class as a JPA entity \*/

@*Entity*

public class Ingredient {

/\*

\* indicates that id property will uniquely identify the entity in the database

\*/

*@Id*

private final String id;

private final String name;

private final *Type* type;

public static enum *Type* {

***WRAP***, ***PROTEIN***, ***VEGGIES***, ***CHEESE***, ***SAUCE***

}

}

* + - Now lets write code for Taco class:

package tacos;

import java.util.Date;

import java.util.List;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.ManyToMany;

import javax.persistence.PrePersist;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

import lombok.Data;

//automatically generates essential JavaBean methods at runtime

@Data

/\* Indicates this class as a JPA entity \*/

@Entity

public class Taco {

/\*

\* indicates id as a property that uniquely identifies this entity in the

\* database

\*/

@Id

/\* indicates that the id property is generated automatically by the database \*/

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private Date createdAt;

//javax validation annotation to make sure that the name field is not null

@NotNull

//javax validation annotation to make sure that the name field must have at least 5 characters

@Size(min = 5, message = "Name must be at least 5 characters long")

private String name;

/\*

\* declaring relationship between the Taco and Ingredient as ManyToMany because

\* a taco can have many ingredients and an ingredient can be a part of many

\* tacos

\*/

@ManyToMany(targetEntity = Ingredient.class)

//javax validation annotation to make sure that the user choose at least two ingredients for the taco

@Size(min = 2, message = "You must choose at least 2 ingredients")

private List<Ingredient> ingredients;

/\* indicates that this method is executed before the Taco is persisted \*/

@PrePersist

/\* method to set createdAt property to current date and time \*/

void createdAt() {

this.createdAt = new Date();

}

}

* + - This is the code for Order class:

package tacos;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.ManyToMany;

import javax.persistence.PrePersist;

import javax.persistence.Table;

import javax.validation.constraints.Digits;

import javax.validation.constraints.NotBlank;

import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;

import lombok.Data;

//Automatically defines the basic getter, setter, equal, hash and tostring() method

@Data

/\* Indicates this class as a JPA entity \*/

@Entity

/\*

\* Specifies that Order entity should be persisted to a table named 'Taco\_Order'

\* in the database. Without it, JPA will try to persist the entity to the table

\* named Order, but Order is a reserved word in sql and could cause problem.

\*/

@Table(name = "Taco\_Order")

public class Order {

/\*

\* indicates id as a property that uniquely identifies this entity in the

\* database

\*/

@Id

/\* indicates that the id property is generated automatically by the database \*/

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private Date placedAt;

// javax validation to make sure that name field is not blank

@NotBlank(message = "Name is required")

private String name;

@NotBlank(message = "Street is required")

private String street;

@NotBlank(message = "City is required")

private String city;

@NotBlank(message = "State is required")

private String state;

@NotBlank(message = "zip code is required")

private String zip;

// passes Luhn's algorithm check to validate the entered credit card number

@CreditCardNumber(message = "Not a valid credit card number")

private String ccNumber;

// passes regex to make sure that user inputs the value in mm/yy format

@Pattern(regexp = "^(0[1-9]|1[0-2])([\\/])([1-9][0-9])$", message = "Must be formatted MM/YY")

private String ccExpiration;

// ensures that the value contains exactly three numeric digits

@Digits(integer = 3, fraction = 0, message = "Invalid CVV")

private String ccCVV;

/\*

\* declaring relationship between the Order and Taco as ManyToMany because

\* an order can have many Tacos and a Taco can be a part of many

\* orders. This is used to create taco\_Order\_tacos table by JPA

\*/

@ManyToMany(targetEntity = Taco.class)

/\*

\* This list of tacos will be used to link order id and tacos related to that

\* order in jdbcOrderRepository

\*/

private List<Taco> tacos = new ArrayList<>();

public void addDesign(Taco design) {

this.tacos.add(design);

}

@PrePersist

void PlacedAt() {

this.placedAt = new Date();

}

}

* + 3.2.3 DECLARING JPA REPOSITORIES:
    - In JDBC, you explicitly declared methods you wanted repository to provide.
    - But in Spring Date you can extend CrudeRepository interface instead.
    - Here is the IngredientRepository:

package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository extends CrudRepository<Ingredient,String>{

}

* + - * CrudRepository declared about a dozen methods for CRUD(create, read, update and delete)
      * Notice that it’s first parameter is the entity type that the repository must persist and second parameter is the type of the enityt ID property.
    - Similarly TacoRepository looks like this:

package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Taco;

public interface TacoRepository extends CrudRepository<Taco,Long>{

}

* + - * Here Taco type must be persisted and the id of that Taco entity is of long type.
    - Same changes can be applied to OrderRepository

package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Order;

public interface OrderRepository extends CrudRepository<Order,Long>{

}

* + - There’s no need to write an implementation for these repositories. Spring data JPA automatically generates an implementation on the fly.
    - The methods provided by CrudRepository are great for general purpose persistence of entities. But what if you have some requirements beyond basic persistence? Let’s customize the repositories to perform queries unique to your domain.
  + 3.2.4 CUSTOMIZING JPA REPOSITORIES;
    - Lets say you want to fetch all the orders delivered to given Zip code. This can be done by adding the following method declaration to the OrderRepository interface:
      * List<Order> findByZip(String zip);
    - When generating the repository implementation , Spring Data examines any methods in the repository interface, parses the method name, and attempts to understand the method’s purpose in the context of the persisted Object(an Order, in this case)
    - Spring Data know that this method is intended to find Orders, because you’ve parameterized CrudRepository with Order.
    - The method findByZip() makes it clear that this method should find all Order entities by matching their zip property with the value passed in as a parameter to the method
    - Repository methods are composed of a verb, an optional subject, the word By, and a predicate. In findByZip(), the verb is ‘find’, subject isn’t specified and is implied ‘Order’, and predicate is ‘Zip’.
    - Let’s consider another more complex example:
      * List<Order> readOrdersByZipAndPlacedAtBetween(String zip, Date startDate, Date endDate);
      * Find, read and get is treated as same by Spring Data.
      * You can also use count as a verb if you want only the count of matching entities.
      * Although the subject is optional,here is says Orders.Spring Data ignores most words in subject so even if you write readPuppiesBy…., it would still find Order entities as that is type CrudRepository is parameterized with.
      * In this case, the predicate referes to two Order properties: zip and placedAt.
      * Zip property must equal to the value passed into the first parameter.
      * The keyword ‘between’ indicates that the value of zip must fall between the values passed into the last two parameters of the method.
      * In addition to implicit Equals operation and the Between operation, Spring Data method signatures can also include any of these operators:
        + ϒ IsAfter, After, IsGreaterThan, GreaterThan
        + ϒ IsGreaterThanEqual, GreaterThanEqual
        + ϒ IsBefore, Before, IsLessThan, LessThan
        + ϒ IsLessThanEqual, LessThanEqual
        + ϒ IsBetween, Between
        + ϒ IsNull, Null
        + ϒ IsNotNull, NotNull
        + ϒ IsIn, In
        + ϒ IsNotIn, NotIn
        + ϒ IsStartingWith, StartingWith, StartsWith
        + ϒ IsEndingWith, EndingWith, EndsWith
        + ϒ IsContaining, Containing, Contains
        + ϒ IsLike, Like
        + ϒ IsNotLike, NotLike
        + ϒ IsTrue, True
        + ϒ IsFalse, False
        + ϒ Is, Equals
        + ϒ IsNot, Not
        + ϒ IgnoringCase, IgnoresCase
    - You can also place either AllIgnoringCase or AllIgnoresCase on the method to ignore case for all String comparisons.
    - For example:
      * List<Order> findByStreetAndCityAllIgnoresCase(String street, String city);
    - Finally, you can also place OrderBy at the end of the method name to sort the results by specified column:
      * List<Order> findByCityOrderByStreet(String city);
    - Sometimes method can be complex to understand by JPA, so we can explicitly use @Query to specify the query to be performed:
      * @Query(“Order o where o.city = ‘Seattle’”)
      * List<Order>readOrdersByCitySeattle
    - We also delete data.sql and schema.sql because JPA automatically generates the schema for us.
    - To see how JPA is generating the tables, we can put this line in application.properties files:
      * spring.jpa.show-sql=true
    - JPA doesn’t add the data to the Ingredient table yet. So, we need to make some changes in TacoCloudApplication.java

package tacos;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import tacos.Ingredient.Type;

import tacos.data.IngredientRepository;

*@SpringBootApplication*

public class TacoCloudApplication {

//this is the method that will run when the JAR file is executed

public static void main(String[] args) {

SpringApplication.*run*(TacoCloudApplication.class, args);

}

*@Bean*

/\* IngredientRepository is injected to the method parameter \*/

*@Autowired*

/\*

\* Command line runner is executed as soon as the application context is loaded

\* and before the run is executed. So, we use it to load the data to the

\* database before the application starts

\*/

public CommandLineRunner dataLoader(IngredientRepository repo) {

return new CommandLineRunner() {

*@Override*

public void run(String... args) throws Exception {

repo.save(new Ingredient("FLTO", "Flour Tortilla", *Type*.***WRAP***));

repo.save(new Ingredient("COTO", "Corn Tortilla", *Type*.***WRAP***));

repo.save(new Ingredient("GRBF", "Ground Beef", *Type*.***PROTEIN***));

repo.save(new Ingredient("CARN", "Carnitas", *Type*.***PROTEIN***));

repo.save(new Ingredient("TMTO", "Diced Tomatoes", *Type*.***VEGGIES***));

repo.save(new Ingredient("LETC", "Lettuce", *Type*.***VEGGIES***));

repo.save(new Ingredient("CHED", "Cheddar", *Type*.***CHEESE***));

repo.save(new Ingredient("JACK", "Monterrey Jack", *Type*.***CHEESE***));

repo.save(new Ingredient("SLSA", "Salsa", *Type*.***SAUCE***));

repo.save(new Ingredient("SRCR", "Sour Cream", *Type*.***SAUCE***));

}

};

}

}

* + - IngredientByIdConverter class must also be changed a little:

package tacos.web;

import java.util.Optional;

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

import org.springframework.core.convert.converter.Converter;

import org.springframework.stereotype.Component;

import tacos.Ingredient;

import tacos.data.IngredientRepository;

/\*This class converts the string of ingredients that the user selects and converts them into the Ingredient Type\*/

*@Component*

public class IngredientByIdConverter implements Converter<String,Ingredient>{

private IngredientRepository ingredientRepo;

/\* Injects JdbcIngredientRepository into the ingredientRepo variable and uses it to set instance variable\*/

*@Autowired*

public IngredientByIdConverter(IngredientRepository ingredientRepo) {

this.ingredientRepo = ingredientRepo;

}

/\*

\* Converts the id which is the value of the ingredient that is selected by user

\* into the Ingredient type

\*/

*@Override*

public Ingredient convert(String id) {

Optional<Ingredient> optionalIngredient = ingredientRepo.findById(id);

return optionalIngredient.isPresent() ?

optionalIngredient.get() : null;

}

}

* + - Or, you can insert initialization data to ingredient but make sure to put these on the application properties:

spring.jpa.show-sql=true

spring.h2.console.enabled = true

spring.datasource.platform = h2

spring.datasource.url = jdbc:h2:mem:rohan

* + - To not get in trouble of Type enum, just use list
    - There are some changes in practice JPA which is easier to understand and execute.