Chapter 3 – Working with Data

* 3.1 READING AND WRITING DATA WITH JDBC
  + Spring JDBC support is rooted in JdbcTemplate class.
  + JdbcTemplate allow developers to perform SQL operations in relational database without all the boilerplate typically required when working with JDBC.
  + 3.1.1 ADAPTING THE DOMAIN FOR PERSISTENCE
    - When persisting objects to a database, it is generally a good idea to have one field that uniquely identifies the object.
    - You ingredient class already has an id field but you need to add id fields to both Taco and Order.
    - It might also be useful to know when a taco is created and when the order is placed
      * So, add a field that captures the date and time that objects are saved
    - Now, in Taco class:

package tacos;

**import java.util.Date;**

import java.util.List;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

import lombok.Data;

//automatically generates essential JavaBean methods at runtime

@Data

public class Taco {

**private final Long id;**

**private final Date createAt;**

//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;

@NotNull

//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<String> ingredients;

}

* + - Similarly in Order.java

package tacos;

**import java.util.Date;**

import javax.validation.constraints.Digits;

import javax.validation.constraints.Pattern;

import javax.validation.constraints.NotBlank;

import org.hibernate.validator.constraints.CreditCardNumber;

import lombok.Data;

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

@Data

public class Order {

**private final Long id;**

**private final 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;

}

* + 3.1.2 WORKING WITH JDBC TEMPLATE
    - Add jdbc Template to your project classPath.

<dependency>

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

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

* + - You will also need a database to store data. You can use the embedded H2 database.

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

* + - Lets write a repository that fetches and saves Ingredient data.
      * The ingredient repository must perform these operations
        + Query for all ingredients into a collection of Ingredient objects
        + Query for a single ingredient by its id.
        + Save an Ingredient object.
      * The following IngredientRepository interface defines those three operations as method declarations:

package tacos.data;

import tacos.Ingredient;

public interface IngredientRepository {

//method to find all the ingredients

Iterable<Ingredient> findAll();

//method to search only specific ingredient by providing it's id as an argument

Ingredient findOne(String id);

//method to save the ingredient to the database

Ingredient save(Ingredient ingredient);

}

* + - You need to write an implementation of IngredientRepository that uses JdbcTemplate to query the database.

package tacos.data;

import java.sql.ResultSet;

import java.sql.SQLException;

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

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.stereotype.Repository;

import tacos.Ingredient;

/\*@Repository declares that this class should be automatically discovered by Spring component

\* scanning and instantiated as a bean in Spring application context

\*/

*@Repository*

public class JdbcIngredientRepository implements IngredientRepository {

private JdbcTemplate jdbc;

/\*

\* When JdbcIngredientRepository bean is created, spring injects it with

\* JdbcTemplate via @Autowired annotated constructor. The constructor assigns

\* JdbcTemplate to an instance variable that will be used in other methods

\*/

*@Autowired*

public JdbcIngredientRepository(JdbcTemplate jdbc) {

this.jdbc = jdbc;

}

*@Override*

/\*

\* This method returns collection of objects. The sql query returns the result

\* set. The result set is then converted to Ingredient object using mapRowToIngredient()

\* method.

\*/

public Iterable<Ingredient> findAll() {

return jdbc.query("select id, name, type from Ingredient",

this::mapRowToIngredient);

}

*@Override*

/\*

\* This method returns a single Ingredient object. First sql query is run where

\* ? = id which is provided in the 3rd argument. The result set produced from

\* the query is then converted to Ingredient object using mapRowToIngredient().

\*/

public Ingredient findOne(String id) {

return jdbc.queryForObject("select id, name, type from Ingredient where id = ?",

this::mapRowToIngredient,id);

}

*@Override*

/\*

\* This method saves the ingredient to the database with table named Ingredient. The

\* three ?,?,? are replaced by the 3 parameters which comes after it in update

\* method

\*/

public Ingredient save(Ingredient ingredient) {

jdbc.update(

"insert into Ingredient(id,name,type) values (?,?,?)",

ingredient.getId(),

ingredient.getName(),

ingredient.getType().toString()

);

return ingredient;

}

/\*

\* This method takes result set and the number of rows in the resultset as an

\* argument and converts that result set into Ingredient object. It creates new

\* Ingredient object with parameter id, name, and type which it gets from the

\* resultset and type is converted to Type datatype as defined in the Ingredient class.

\*/

private Ingredient mapRowToIngredient(ResultSet rs, int RowNum) throws SQLException{

return new Ingredient(

rs.getString("id"),

rs.getString("name"),

Ingredient.*Type*.*valueOf*(rs.getString("type"))

);

}

}

* + - Now, we will inject JdbcIngredientRepository into DesignTacoController and use it to provide list of Ingredient objects instead of using hard coded values.

package tacos.web;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

import javax.validation.Valid;

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.validation.Errors;

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

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

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

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

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

import lombok.extern.slf4j.Slf4j;

import tacos.Ingredient;

import tacos.Taco;

import tacos.data.IngredientRepository;

import tacos.Ingredient.Type;

/\*Lombok provided annotation that creates SLF4J(Simple logging facade for java) logger in the class at the runtime\*/

@Slf4j

/\* Identify this class as a controller and mark it for component scanning \*/

*@Controller*

/\*

\* Specifies that this class will handle requests whose path begin with /design

\* i.e in browser localhost:8080/design

\*/

*@RequestMapping*("/design")

*@SessionAttributes*("order")

public class DesignTacoController {

private final IngredientRepository ingredientRepo;

*@Autowired*

public DesignTacoController(IngredientRepository ingredientRepo) {

this.ingredientRepo = ingredientRepo;

}

//indicates that this method adds one or more model attributes. Will always call this method first.

*@ModelAttribute*

public void addIngredientsToModel(Model model) {

// creating new objects of Ingredient.java class and storing the objects in a

// list

List<Ingredient> ingredients = new ArrayList<>();

ingredientRepo.findAll().forEach(i->ingredients.add(i));

// Getting all the values from the Type enum in the ingredient class and storing

// them in an array.

*Type*[] types = Ingredient.*Type*.*values*();

// for each elements in the types array

for (*Type* type : types) {

// add model attribute(wrap, new ingredient object with type wrap) etc which can

// directly be accessed by thymeleaf

model.addAttribute(type.toString().toLowerCase(), filterByType(ingredients, type));

}

}

// returns all the objects from ingredients list that has type equal to the type

// in the argument

private List<Ingredient> filterByType(List<Ingredient> ingredients, *Type* type) {

return ingredients

.stream()

.filter(x -> x.getType().equals(type)) // ingredients.getType = type where getTYpe is automatically managed by Lombok annotation

.collect(Collectors.*toList*());// converts stream into a list

}

/\*

\* specifies that when an HTTP GET request is received for design, this method

\* will handle the request.

\*/

*@GetMapping*

public String showDesignForm(Model model) {

// adding "design' attribute to the model which is a new Taco() object

model.addAttribute("design", new Taco());

// This is a thymeleaf view which is displayed in the browser

return "design";

}

// Handles post request for /design i.e when the user enters the submit button

// after designing the taco

*@PostMapping*

/\*

\* first check if the submitted form has any validation errors. Errors object

\* will store any error details during the validation check

\*/

public String processDesign(*@Valid* *@ModelAttribute*("design") Taco design, Errors errors) {

// if there is an error then just return the 'design' view

if (errors.hasErrors()) {

return "design";

}

// Save the taco design

// We'll do this is chapter 3

***log***.info("Processing design: " + design);

// redirects the user to the relative path /orders/current i.e in the browser

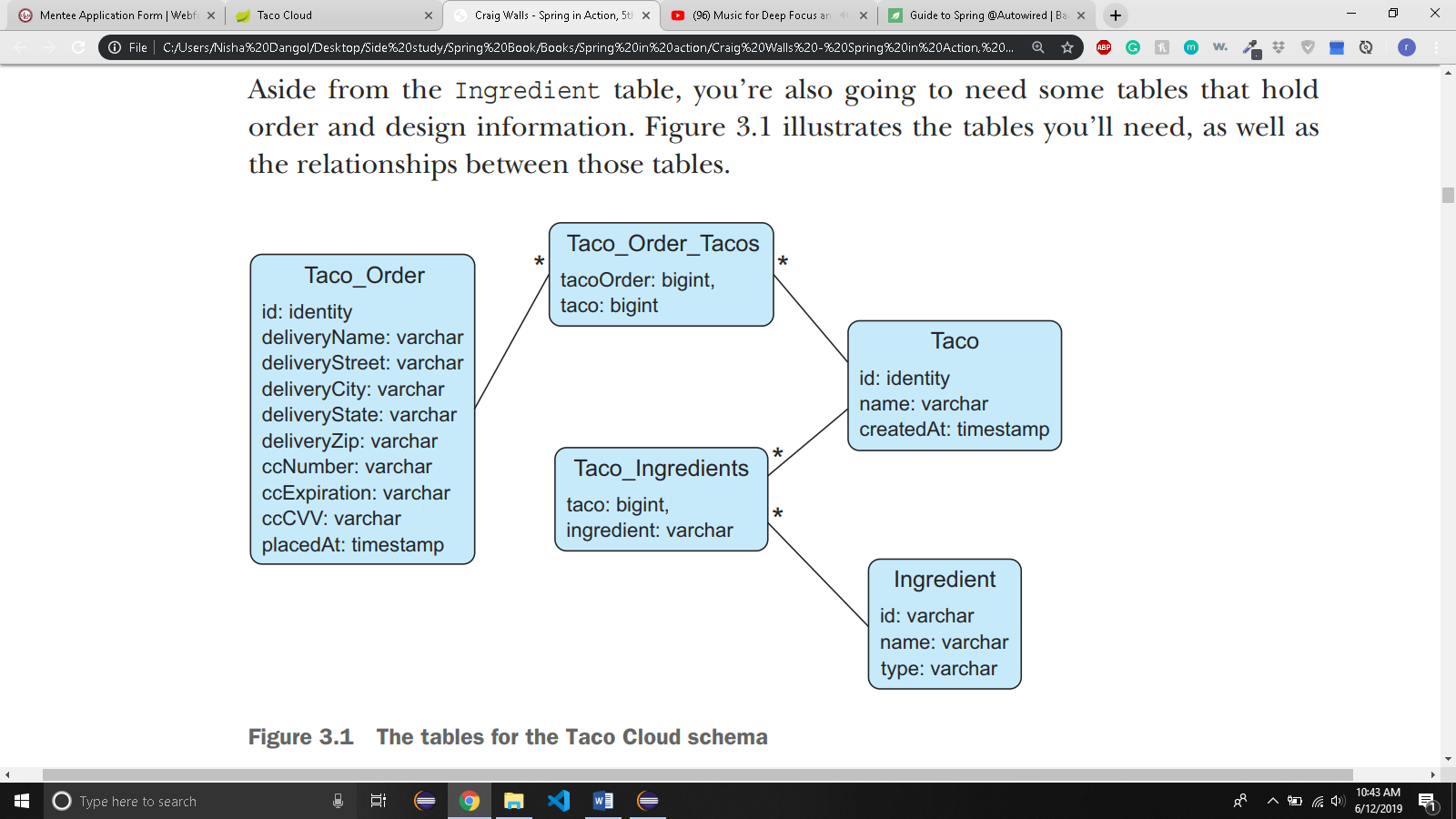
// http:8080/orders/current

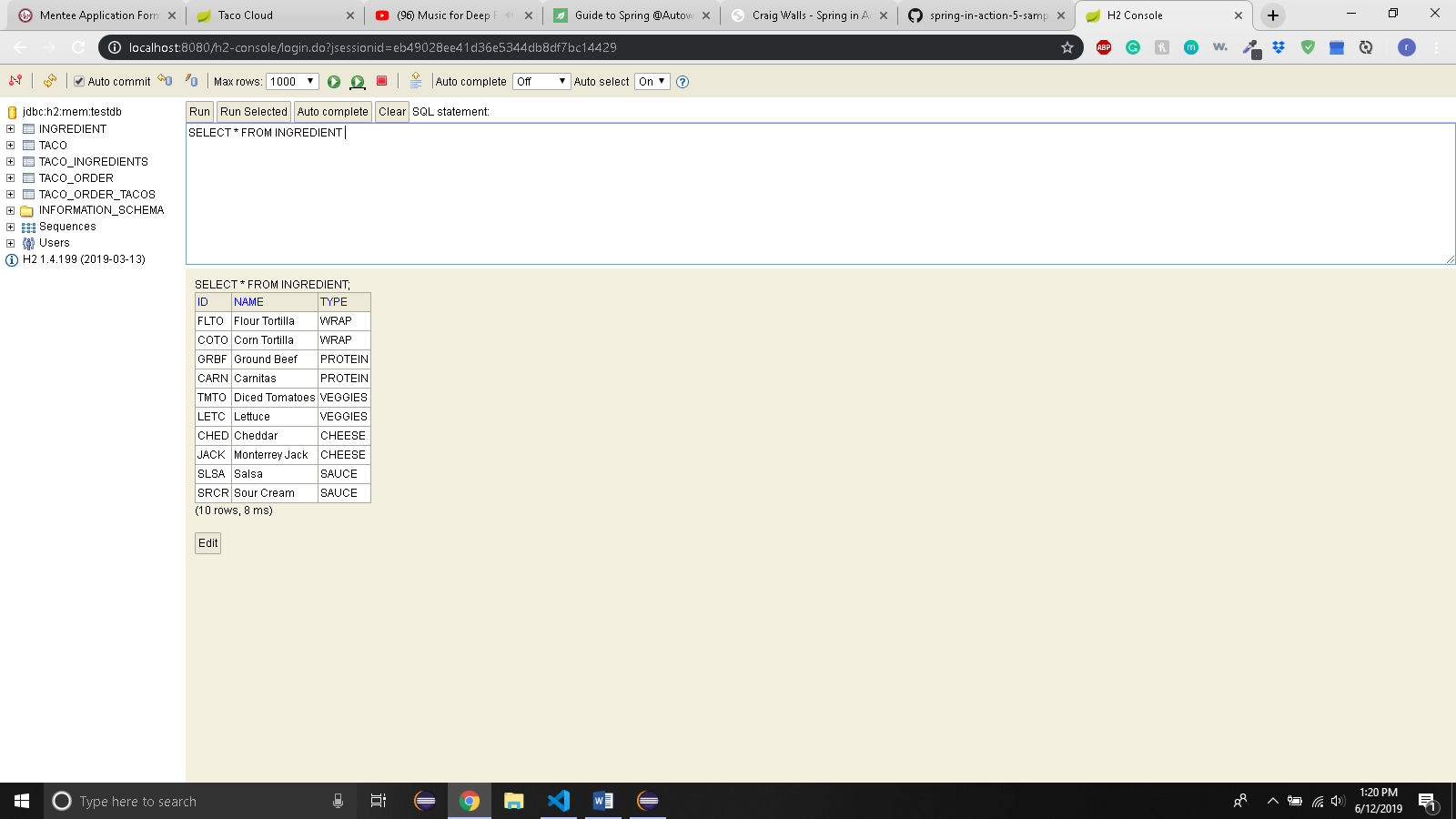
return "redirect:/orders/current";

}

}

* + - * Second line of addIngredientsToModel() makes a call to IngredientRepository’s findAll() method.
      * The findAll() method fetches all the ingredients from the database before filtering them into distinct types in the model.
    - You also need to create a Ingredient table and populate with some ingredient data.
  + 3.1.3 DEFINING A SCHEMA AND PRELOADING DATA



* + - The tables in fig 3.1 has following purposes:
      * Ingredient – Holds ingredient information
      * Taco – Holds essential information about a taco design
      * Taco\_ingredients – Contains one or more rows for each row in Taco, mapping the taco to the ingredient of that taco.
      * Taco\_order – Holds essential order details
      * Taco\_Order\_Tacos – Contains one or more rows for each row in Taco\_order, mapping the order to the tacos in the order.
    - If there is a file names schema.sql in the root of application’s classpath, then the SQL in that file will be executed against the database when the application starts
      * So, we put schema.sql in src/main/resources/schema.sql
      * create table if not exists Ingredient(
      * id varchar(4) not null,
      * name varchar(25) not null,
      * type varchar(10) not null
      * );
      * create table if not exists Taco(
      * id identity,
      * name varchar(50) not null,
      * createdAt timestamp not null
      * );
      * create table if not exists Taco\_Ingredients(
      * taco bigint not null,
      * ingredient varchar(4) not null
      * );
      * alter table Taco\_Ingredients
      * add foreign key (taco) references Taco(id);
      * alter table Taco\_Ingredients
      * add foreign key (ingredient) references Ingredient(id);
      * create table if not exists Taco\_Order(
      * id identity,
      * deliveryName varchar(50) not null,
      * deliveryStreet varchar(50) not null,
      * deliveryCity varchar(50) not null,
      * deliveryState varchar(2) not null,
      * deliveryZip varchar(10) not null,
      * ccNumber varchar(16) not null,
      * ccExpiration varchar(5) not null,
      * ccCVV varchar(3) not null,
      * placedAt timestamp not null
      * );
      * create table if not exists Taco\_Order\_Tacos(
      * tacoOrder bigint not null,
      * taco bigint not null
      * );
      * alter table Taco\_Order\_Tacos
      * add foreign key(tacoOrder) references Taco\_Order(id);
      * alter table Taco\_Order\_Tacos
      * add foreign key(taco) references Taco(id);
    - You also need to preload the database with some ingredient data.
      * Spring boot also execute a file named data.sql from the root of the classpath when the application starts i.e src/main/resources/data.sql
    - Now you can start the application and see JdbcIngredientRepository in action.
* 
  + - Now, you will write repositories for persisting Taco, Order and data.
  + 3.1.4 INSERTING DATA
    - There are two ways to save data with JdbcTemplate:
      * Directly using the Update method
      * Using the SimpleJdbcInsert wrapper class
    - Let’s see how to use update() method when the persistence needs are more complex than what was required to save Ingredient.
    - SAVING DATA WITH JDBC TEMPLATE
      * To save Taco objects, the TacoRepository declares a save method:

package tacos.data;

import tacos.Taco;

public interface TacoRepository {

Taco save(Taco design);

}

* + - * To save Order objects, the OrderRepository also declares a save method:

package tacos.data;

import tacos.Order;

public interface OrderRepository {

Order save(Order order);

}

* + - * Saving a taco design requires us to also save the ingredients in that taco which will be associated to Taco\_Ingredients table.
      * Similarly, saving an order requires us to also save the tacos associated with the order to the Taco\_Order\_Tacos table.
      * To implement TacoRepository, you need a JdbcTacoRepository class with a save method that starts by saving essential taco design details (Ex. Name and time of creation), and then inserts one row into Taco\_Ingredients for each ingredient in the Taco object.

package tacos.data;

import java.sql.Timestamp;

import java.sql.Types;

import java.util.Arrays;

import java.util.Date;

import javax.management.Query;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.PreparedStatementCreator;

import org.springframework.jdbc.core.PreparedStatementCreatorFactory;

import org.springframework.jdbc.support.GeneratedKeyHolder;

import org.springframework.jdbc.support.KeyHolder;

import org.springframework.stereotype.Repository;

import tacos.Ingredient;

import tacos.Taco;

*@Repository*

public class JdbcTacoRepository implements TacoRepository {

private JdbcTemplate jdbc;

public JdbcTacoRepository(JdbcTemplate jdbc) {

this.jdbc = jdbc;

}

*@Override*

public Taco save(Taco taco) {

/\* The key that is returned from saveTacoInfo() is stored as a tacoId variable\*/

long tacoId = saveTacoInfo(taco);

/\* Sets the value in tacoId as an id property of Taco class \*/

taco.setId(tacoId);

/\*

\* for each ingredient selected by the user and stored in Taco object which is

\* provided by taco.getIngredients(), call saveIngredientToTaco() method to save the tacoId and

\* the selected ingredients to the Taco\_Ingredients table.

\*/

for (Ingredient ingredient : taco.getIngredients()) {

saveIngredientToTaco(ingredient, tacoId);

}

return taco;

}

/\*

\* This method accepts Taco object and returns a unique key that is set as it's

\* id

\*/

private long saveTacoInfo(Taco taco) {

/\* sets the createdAt property of taco class \*/

taco.setCreatedAt(new Date());

/\*

\* Prepares a statement that will be passed in update method and inserts the

\* taco name and created date to 'name' and 'createdAt' field of 'Taco' table

\*/

PreparedStatementCreator psc = new PreparedStatementCreatorFactory(

"insert into Taco(name,createdAt) values (?,?)", Types.***VARCHAR***, Types.***TIMESTAMP***

).newPreparedStatementCreator(

Arrays.*asList*(taco.getName(), new Timestamp(taco.getCreatedAt().getTime()))

);

/\* Provides the auto generated keys \*/

KeyHolder keyHolder = new GeneratedKeyHolder();

/\*

\* Passes the psc and keyHolder in the jdbc update method to execute the Query.

\* I don't know why keyHolder is passed

\*/ jdbc.update(psc, keyHolder);

/\*returns the unique key provided by keyHolder which will be set as the tacoId\*/

return keyHolder.getKey().longValue();

}

/\*

\* Method to insert ingredients and unique tacodId that are provided as parameter

\* to the 'ingredient' and 'taco' field of 'Taco\_Ingredients' table

\*/

private void saveIngredientToTaco(Ingredient ingredient, Long tacoId) {

jdbc.update("insert into Taco\_Ingredients(taco,ingredient)" + "values(?,?)", tacoId, ingredient.getId());

}

}

* + - Not you need to inject TacoRepository into DesignController

package tacos.web;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

import javax.validation.Valid;

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.validation.Errors;

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

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

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

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

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

import tacos.Ingredient;

import tacos.Taco;

import tacos.data.IngredientRepository;

import tacos.data.TacoRepository;

import tacos.Ingredient.Type;

import tacos.Order;

/\* Identify this class as a controller and mark it for component scanning \*/

@Controller

/\*

\* Specifies that this class will handle requests whose path begin with /design

\* i.e in browser localhost:8080/design

\*/

@RequestMapping("/design")

/\*

\* You need order to be present across multiple requests so that you can create

\* multiple tacos and add them to order. This annotation specifies any model

\* objects like the order attribute to be kept in session and available for

\* multiple requests

\*/

@SessionAttributes("order")

public class DesignTacoController {

/\*

\* This is final because ingredients don't change

\*/ private final IngredientRepository ingredientRepo;

private TacoRepository designRepo;

@Autowired

public DesignTacoController(IngredientRepository ingredientRepo, TacoRepository designRepo) {

this.ingredientRepo = ingredientRepo;

this.designRepo = designRepo;

}

/\* Ensures that an order object is created in model \*/

@ModelAttribute(name = "order")

public Order order() {

return new Order();

}

/\* Ensured that the taco objectis created in model \*/

@ModelAttribute(name = "taco")

public Taco taco() {

return new Taco();

}

/\*

\* specifies that when an HTTP GET request is received for design, this method

\* will handle the request.

\*/

@ModelAttribute

public void addIngredientsToModel(Model model) {

// creating new objects of Ingredient.java class and storing the objects in a

// list

List<Ingredient> ingredients = new ArrayList<>();

// Getting all the values from the Type enum in the ingredient class and storing

// them in an array.

ingredientRepo.findAll().forEach(i -> ingredients.add(i));

Type[] types = Ingredient.Type.values();

// for each elements in the types array

for (Type type : types) {

// add model attribute(wrap, new ingredient object with type wrap) etc which can

// directly be accessed by thymeleaf

model.addAttribute(type.toString().toLowerCase(), filterByType(ingredients, type));

}

}

@GetMapping

public String showDesignForm(Model model) {

return "design";

}

// returns all the objects from ingredients list that has type equal to the type

// in the argument

private List<Ingredient> filterByType(List<Ingredient> ingredients, Type type) {

return ingredients.stream().filter(x -> x.getType().equals(type)) // ingredients.getType = type where getTYpe is

// automatically managed by Lombok

// annotation

.collect(Collectors.toList());// converts stream into a list

}

// Handles post request for /design i.e when the user enters the submit button

// after designing the taco

@PostMapping

/\*

\* first check if the submitted form has any validation errors. Errors object

\* will store any error details during the validation check

\*

\* @ModelAttribute will get the Order object from the model and indicate that it's value should come

\* directly from model and Spring MVC shouldn't attempt to bind request parameters to it.

\*

\* The design object comes from request parameters which will have name and ingredients set by the users

\*

\*/

public String processDesign(@Valid Taco design, Errors errors, @ModelAttribute Order order) {

// if there is an error then just return the 'design' view

if (errors.hasErrors()) {

return "design";

}

Taco saved = designRepo.save(design);

order.addDesign(saved);

System.out.println(design);

// redirects the user to the relative path /orders/current i.e in the browser

// http:8080/orders/current

return "redirect:/orders/current";

}

}

* + - * @modelAttribute annotated method “order” ensures that an Order object will be created in the model.
      * Unlike Taco object in the session, you need the order to be present across multiple requests so that you can create multiple tacos and add them to the order
      * The class level @sessionAttributes annotation specifies any model objects like the order attribute that should be kept in session and available across multiple requests.
      * In process design, the Order parameter is annotated with @ModelAttribute to indicate that its value should come from the model and that Spring mvc shouldn’t attempt to bind request parameters to it.
      * After validation check, the processDesign uses the injected TacoRepository to save the taco. It then adds the Taco object to the Order that’s kept in the session.
      * The Order object is kept in session and isn’t saved to the database until the user completes and submits the order form.
      * At that point, OrderController needs to implement OrderRepository to save the order.
      * Note: change version of h2 to 1.4.196 because 1.4.197 gives error in key holder values.
    - While saving an order, you must also save tacos that are associated with the order in the Taco\_Order\_Tacos table.
    - Instead of using preparedStatement which is cumbersome, we can use SimpleJdbcInsert, which is an object that wraps JdbcTemplate to make it easier to insert data into a table.
    - Let’s start by writing JdbcOrderRepository

package tacos.data;

import java.util.Date;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

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

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.simple.SimpleJdbcInsert;

import org.springframework.stereotype.Repository;

import com.fasterxml.jackson.databind.ObjectMapper;

import tacos.Order;

import tacos.Taco;

*@Repository*

public class JdbcOrderRepository implements OrderRepository {

private SimpleJdbcInsert orderInserter;

private SimpleJdbcInsert orderTacoInserter;

private ObjectMapper objectMapper;

*@Autowired*

public JdbcOrderRepository(JdbcTemplate jdbc) {

/\*

\* linked with Taco\_Order table and each row in the table will have

\* autogenerated id

\*/

this.orderInserter = new SimpleJdbcInsert(jdbc).withTableName("Taco\_Order").usingGeneratedKeyColumns("id");

/\* linked to Tao\_Order\_Tacos table \*/

this.orderTacoInserter = new SimpleJdbcInsert(jdbc).withTableName("Taco\_Order\_Tacos");

/\* objectmapper for converting between different datatypes \*/

this.objectMapper = new ObjectMapper();

}

*@Override*

public Order save(Order order) {

/\*

\* set the placedAt's value to new Date()

\*/ order.setPlacedAt(new Date());

/\* save the orderId which is returned by saveOrderDetails() method \*/

long orderId = saveOrderDetails(order);

/\* Then set the Order objects id field to orderId \*/

order.setId(orderId);

/\* gets all the tacos associated to this order \*/

List<Taco> tacos = order.getTacos();

/\* for each taco save the taco and orderId into the Taco\_Order\_Tacos table \*/

for(Taco taco: tacos) {

saveTacoToOrder(taco,orderId);

}

/\* returns the Order object \*/

return order;

}

/\* Method to return the orderId and fill Taco\_Order table \*/

private long saveOrderDetails(Order order) {

*@SuppressWarnings*("unchecked")

/\* coverts the Order type into Map type and store in variable named 'values' \*/

Map<String, Object> values = objectMapper.convertValue(order, Map.class);

/\*

\* Then execute put method of Map where key = placedAt and value =

\* order.getPlacedAt() which is set in save method above

\*/

values.put("placedAt", order.getPlacedAt());

/\*

\* inserts the map into Taco\_Order table where key = column in table and value =

\* the actual values in that column. Returns the id which is autogenerated and

\* convert into long value which is the returned by the method

\*/

long orderId = orderInserter.executeAndReturnKey(values).longValue();

return orderId;

}

/\* method to fill Taco\_Order\_Tacos table \*/

private void saveTacoToOrder(Taco taco, long orderId) {

Map<String, Object> values = new HashMap<>();

/\*

\* tacooOrder and taco are the actual columns of Taco\_ORder\_Tacos table whose

\* value is set to taco.getId() and orderId respectively

\*/

values.put("tacoOrder", orderId);

values.put("taco", taco.getId());

/\* inserts the (key,value) pair from values map into the table \*/

orderTacoInserter.execute(values);

}

}

* + - JdbcTemplate is injected through the constructor
    - Then the constructor uses the template to construct a couple of simpleJdbcInsert instances.
    - The first instance assigned to orderInserter is configured to work with Taco\_Order table and the id property is auto generated by the database.
    - The second instance assigned to orderTacoInserter is configured to work with Taco\_Order\_Tacos table but doesn’t say how many ids will be generated in the table.
    - The constructor also creates an instance of jackson’s ObjectMapper and assigns it to instance variable.
    - SimpleJdbcInsert has a coupe of useful methods for executing the insert:
      * execute() and executeAndReturnKey().
      * Both accept a Map<String, Object>, where the map keys correspond to the column names in the table and data is inserted into.
      * The map values are inserted into those columns.
    - ObjectMapper’s convertValue() converts an Order into Map.
    - Once the map is created, you will set the placedAt entry to the value of the Order object’s placedAt property. This is necessary because ObjectMapper would otherwise convert the Date property into a long, which is incompatible with the placedAt field in the Taco\_Order table.
    - executeAndReturnKey() on orderInserter saves the order information to the Taco\_Order table and returns the database generated Id as a Number object, which a call to longValue() converts to a long returned from the method.
    - saveTacoToOrder() method creates a map and sets the appropriate values. The map keys correspond to column names in the table.
    - orderTacoInserter’s execute() method performs the insert
    - Now, inject OrderRepository into OrderController.
    - The Order object submitted in the form(which is the same Order object maintained in the session) is saved via the save() method on the injected orderRepository.
    - Once the order is saved, you need to clean it out from the session so we set the session to complete