3. Working with data

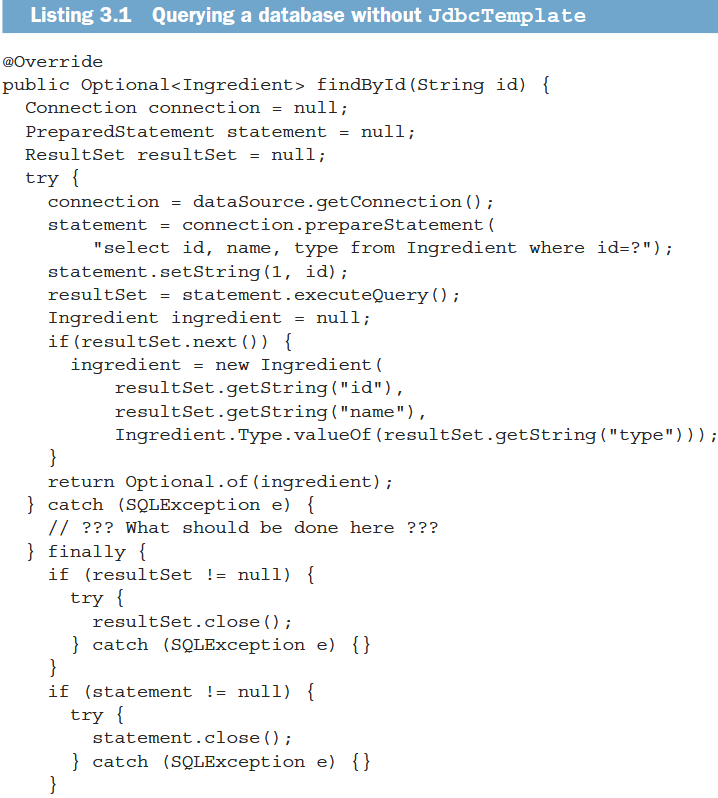
# 3.1 Reading and writing data with JDBC

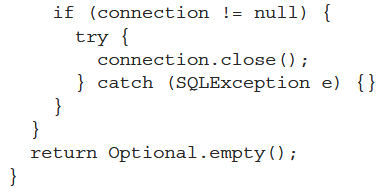
-Relational database and SQL are leading choice for data persistence.

-Working with relational data: JDBC and JPA

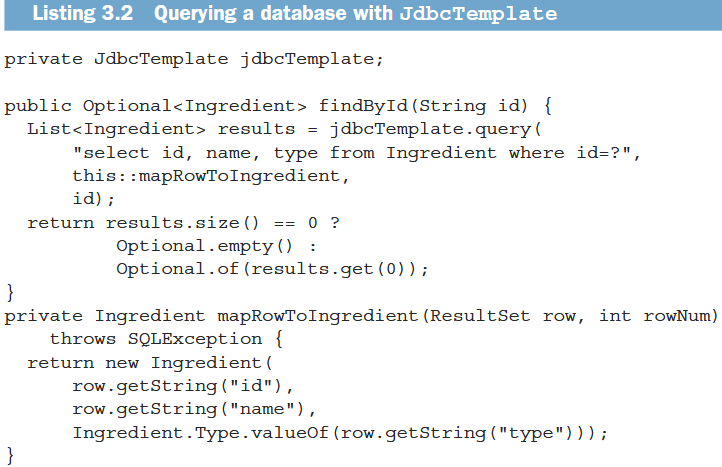
-Spring JDBC support in **JdbcTemplate** class: provide means that developer can perform SQL operations against relational database without preparing configuration JDBC

-Query a database without JdbcTemplate:





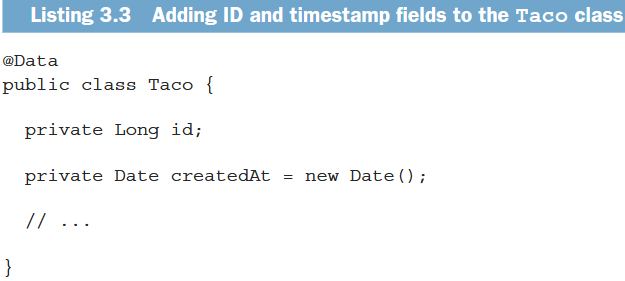
-Use JdbcTemplate

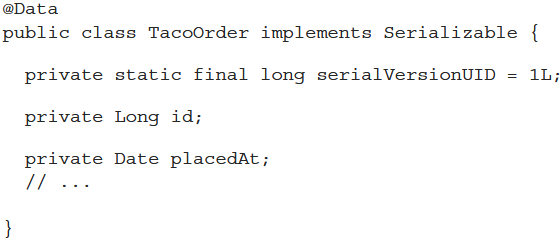


## 3.1.1 Adapting the domain for persistence

-Persisting objects to a database, it’s a good idea to have one field that uniquely identifies the object.

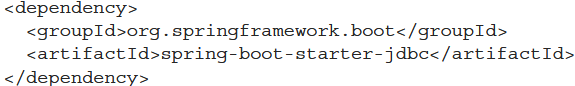
-It’s useful to know when Taco is created and TacoOrder is placed. Also add a field to each object to capture the data and time that the objects are saved.



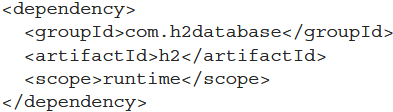


## 3.1.2 Working with JdbcTemplate

-Add JdbcTemplate to the project classpath.



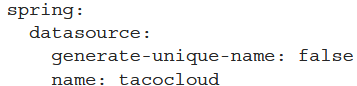
-For development purposes, use H2 embedded database



+By default, the database name is automatically generated. It’s ideal to pin down the database name:



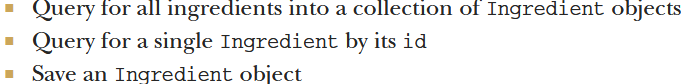
Or change to application.yml by YAML format:



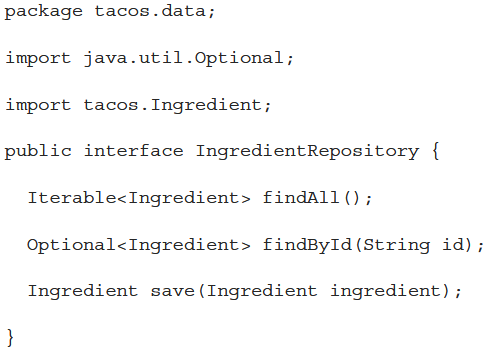
+The database URL is “**jdbc:h2:mem:tacocloud**”

-Defining JDBC Repositories

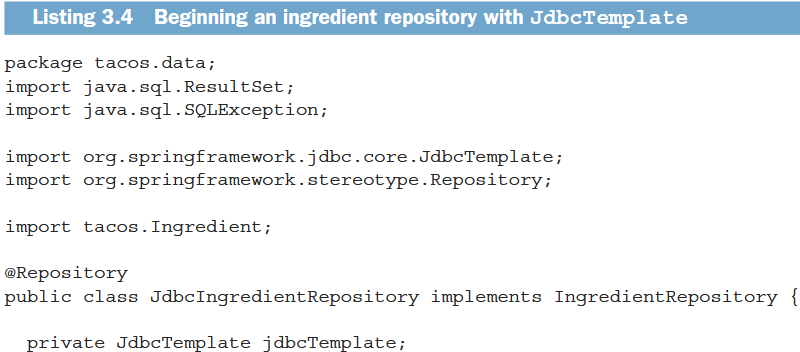
+Ingredient repository performs operations:

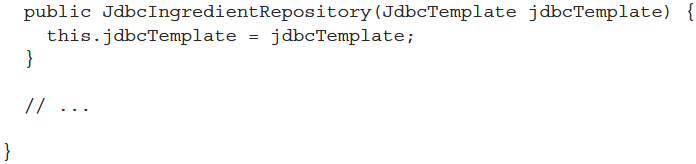


+**IngredientRepository** interface defines 3 operations:



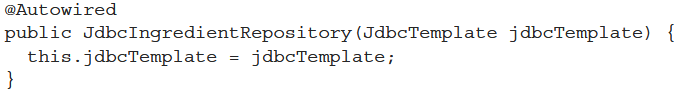
+You still need to write an **implementation** of IngredientRepository using **JdbcTemplate** to query database:



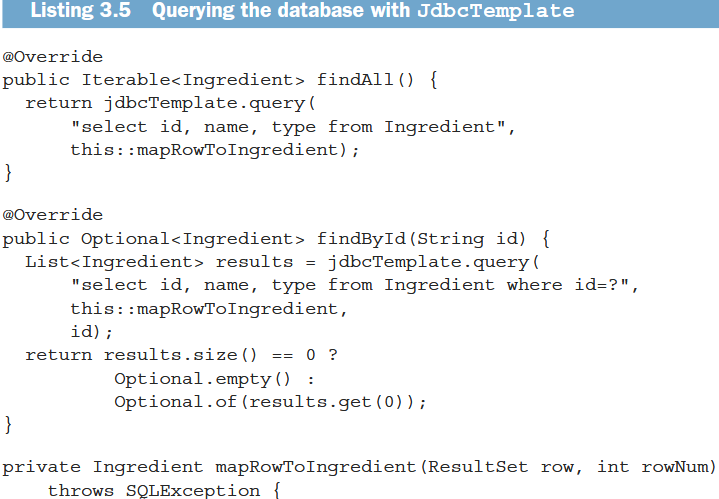


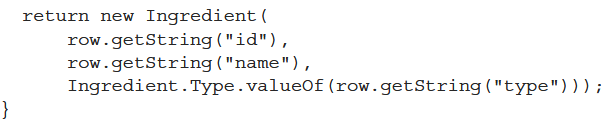
+JdbcIngredientRepository is annotated with **@Repository**: It should be automatically discovered by Spring component scanning and instantiated as a bean in Spring application context.

+When Spring creates JdbcIngredientRepository bean, it **injects** it with **JdbcTemplate**. If there is more one constructor, annotate constructor with **@Autowired**



+3 others methods:



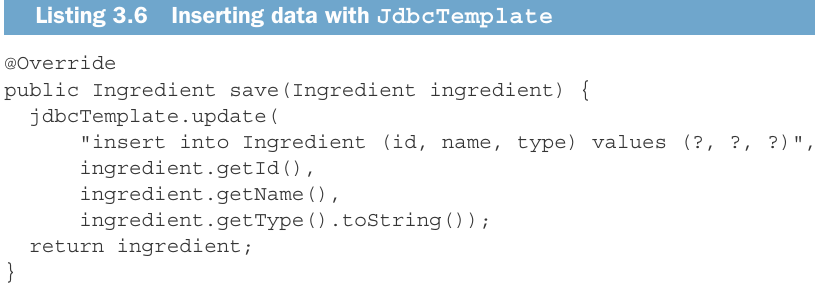


+**query()** accepts SQL for the query and an **implementation** of Spring’s **RowMapper** to **map** each row in the result set to object.

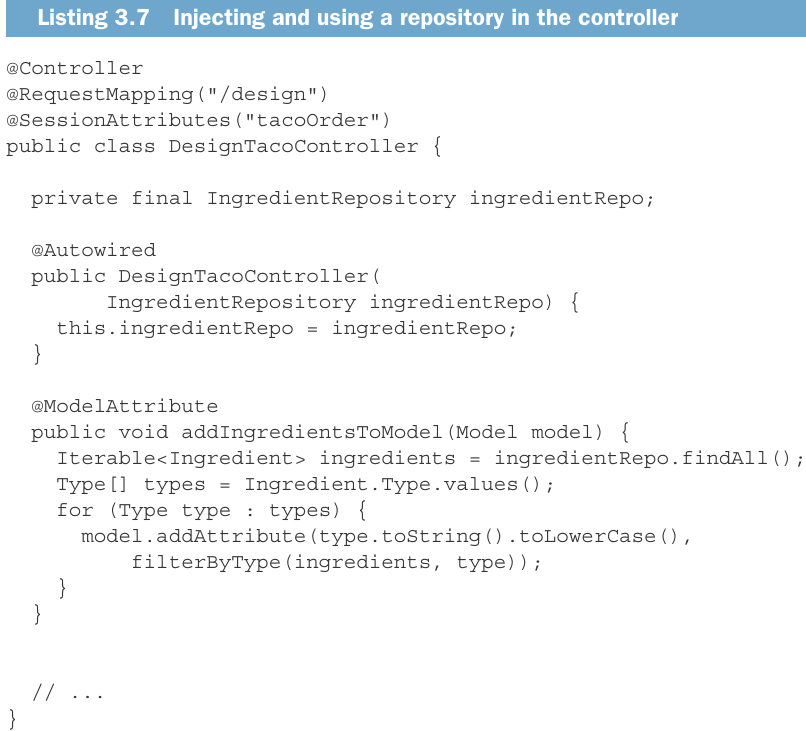
+Reading data from database is only part of the story. Let see how to write data to database.

-Inserting a row:

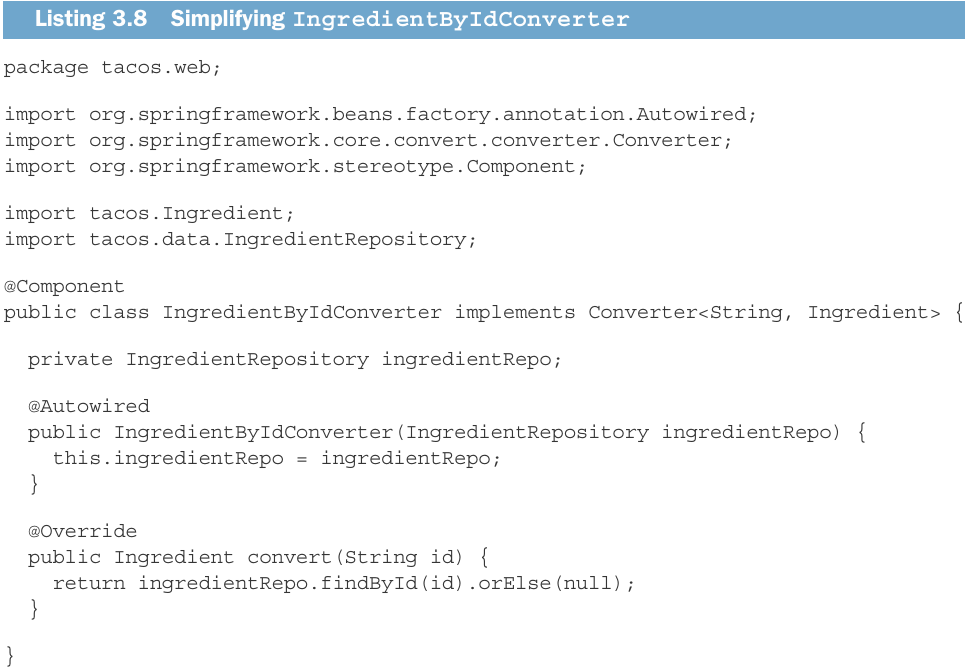
+JdbcTemplate **update()**: **write** or **updates** data



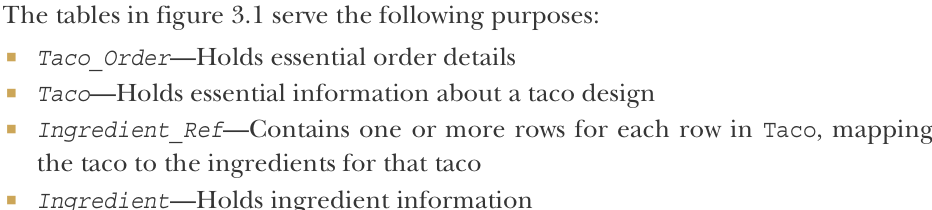
-Now inject JdbcIngredientRepository into DesignTacoController to provide a list of Ingredient objects:

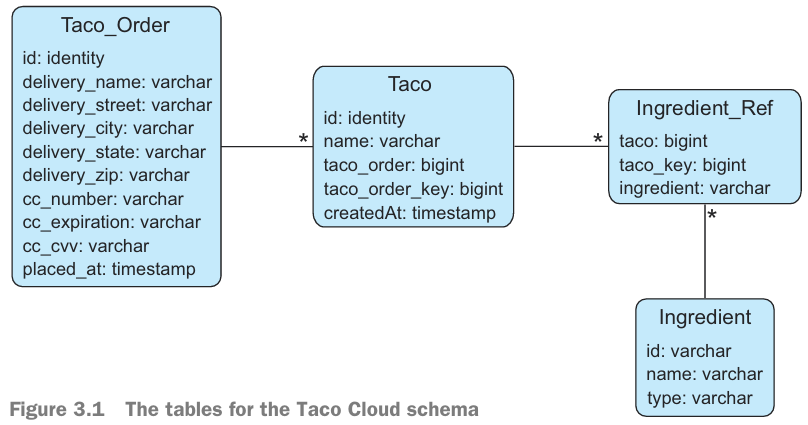


-Simplify the **IngredientByIdConverter**



## 3.1.3 Defining a schema and preloading data

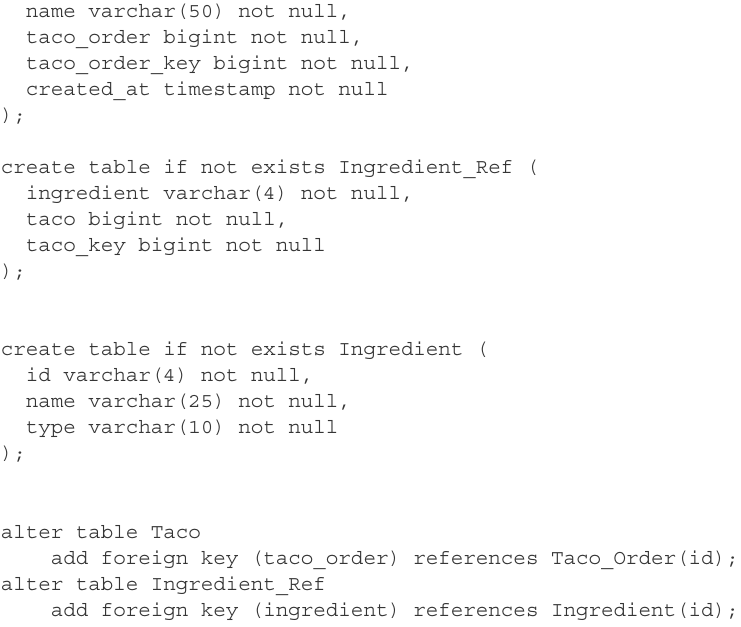
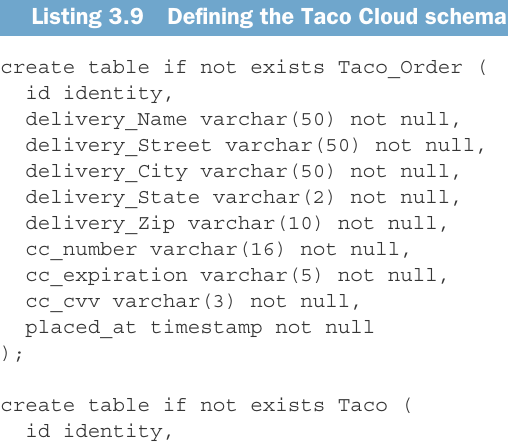




+Taco\_Order and Taco are members of an aggregate where Taco\_Order is the aggregate root.

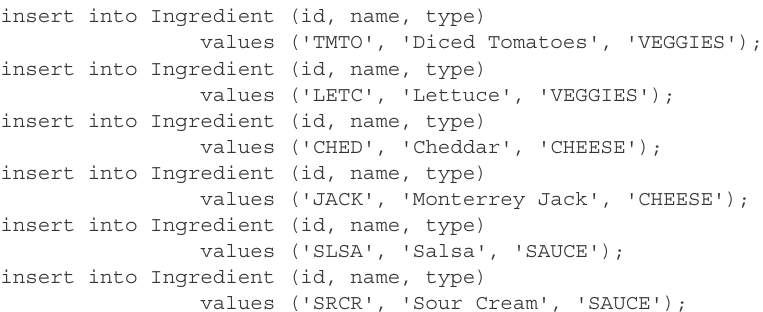
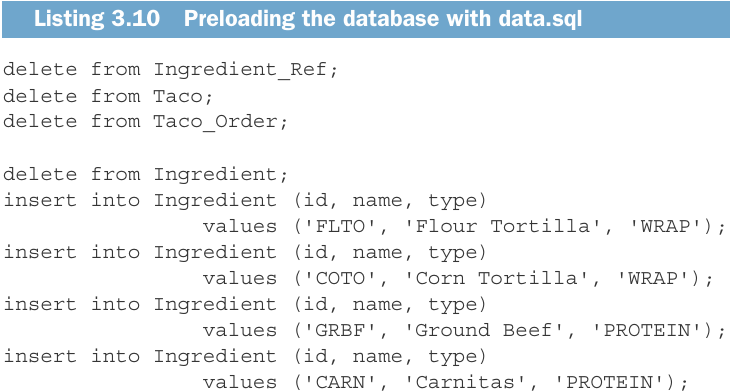
-Note: Aggregates and aggregate roots are core concept of **domain-driven design (DDD)**: The structure and language of software code should match the business domain. See more: *Domain-Driven Design: Tackling Complexity in the Heart of Software (https://www.dddcommunity .org/book/evans\_2003/)*

-SQL create tables:



-Spring Boot puts this schema definition. If there’s a file named schema.sql in the root of app’s classpath, SQL in that file will be executed when the app starts.->Place schema.sql in src/main/resource

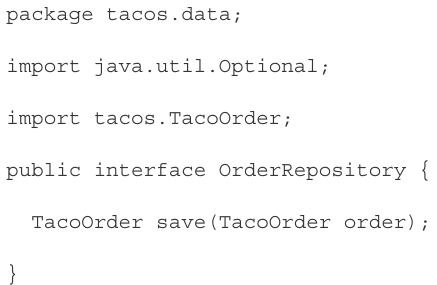
-Preload the database: SB will also execute a file data.sql from the root of classpath when app starts.



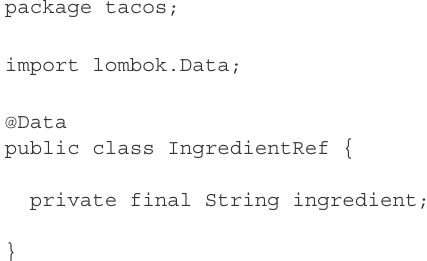
## 3.1.4 Inserting data

-Taco objects don’t exist outside of the content of a TacoOrder. We only need to define a repository to persist TacoOrder object.

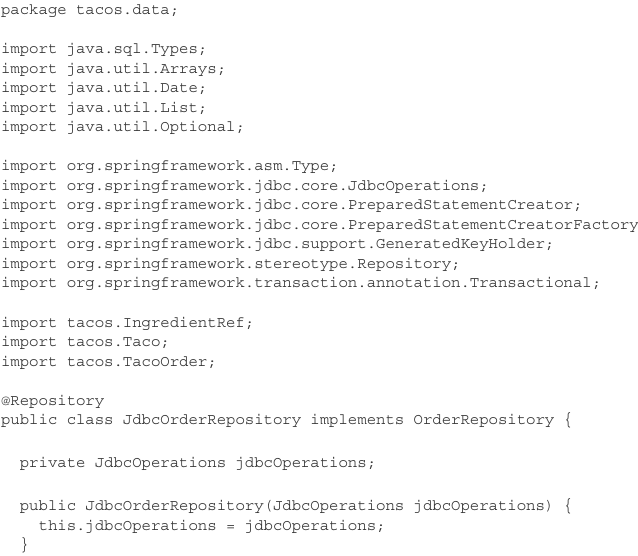
-**OrderRepository** interface:

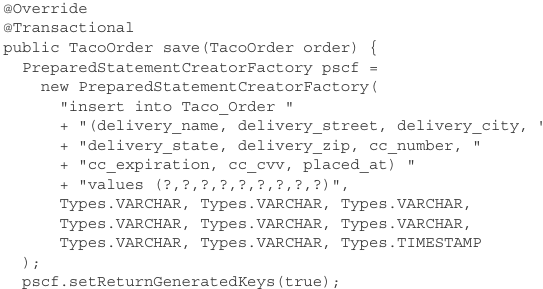


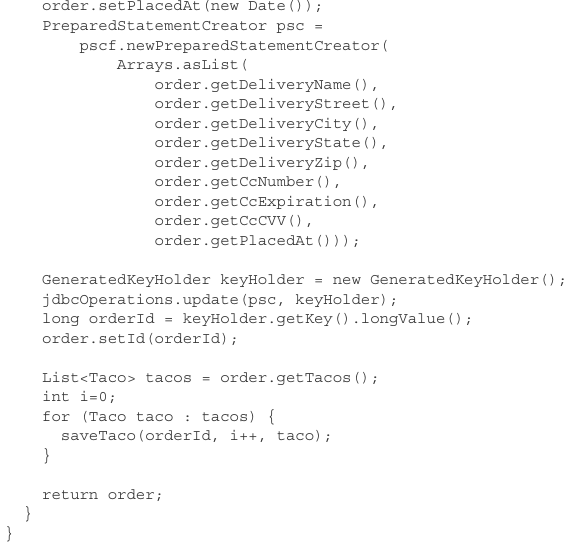
+When you save TacoOrder, also save Taco objects -> save an object that represents the link between Taco and each Ingredient-> **IngredientRef** class defines that link.



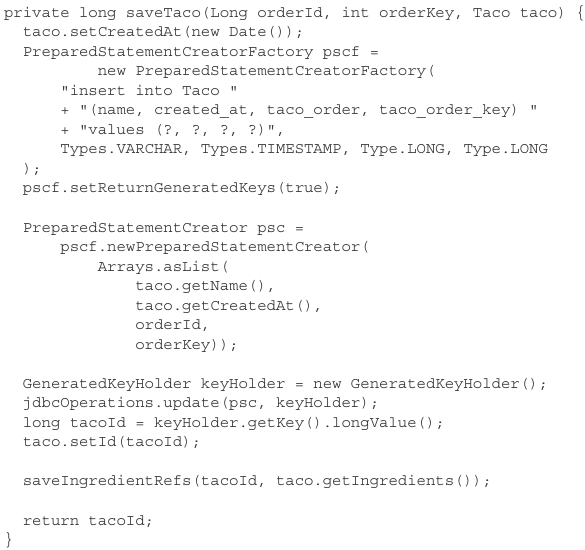
-id property on Taco\_Order table is **identity**: the database determines the value automatically. Spring offers **GeneratedKeyHolder** type to return that value. It involves working with **prepared statement** in **save()**:





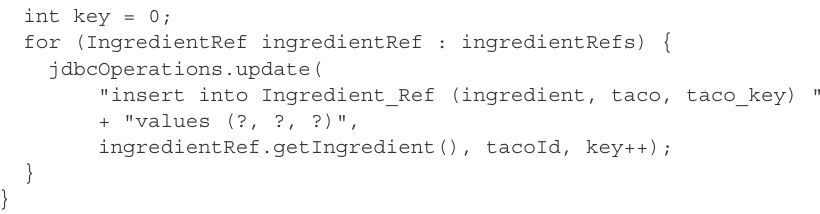


-The order has been saved, but need to also save Taco objects associated with the order: **saveTaco()**

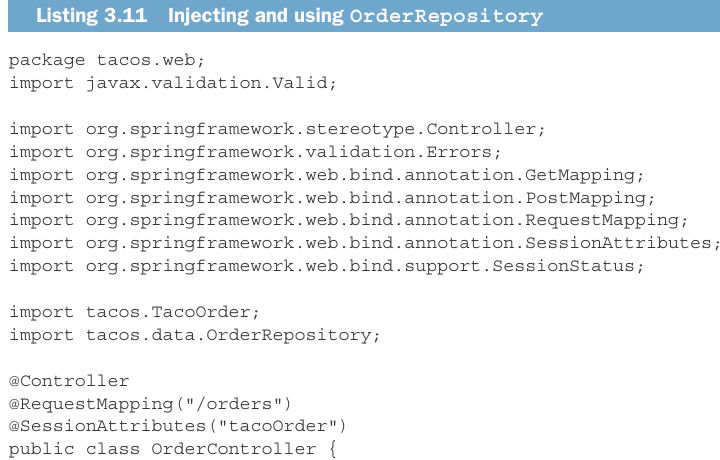


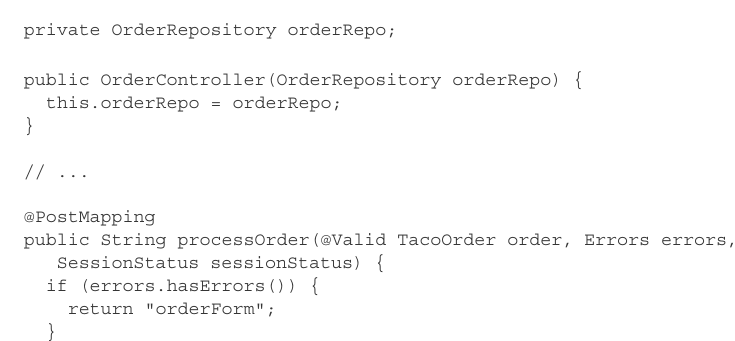
-**saveIngredientRefs()** create a row in Ingredient\_Ref table to link Taco row to Ingredient row.

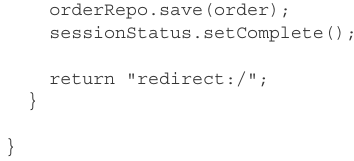




-Inject OrderRepository into OrderController saving order





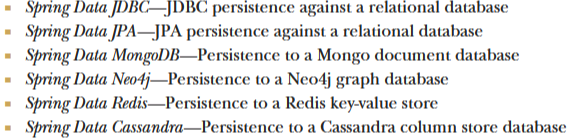


-JdbcTemplate makes working with relational database simpler, but some persistence tasks are still challenging: persist nested domain objects in aggregate…

# 3.2 Working with Spring Data JDBC

-**Spring Data** project is a large project comprising several subprojects, most of which are focused on data persistence with different database types.

-**Spring Data projects** would include:

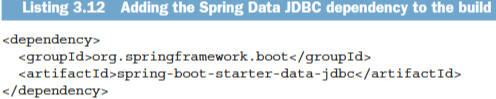


-The most interesting **feature**: automatically **create** **repositories** based on a repository **specification** **interface**.->persistence with SD projects has little or no persistence logic and involves writing one or more repository interfaces.

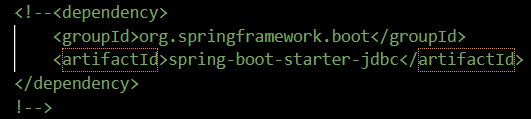
## 3.2.1 Adding Spring Data JDBC to the build

-**Spring Data JDBC** is available as a starter dependency for Spring Boot apps.

-Add in pom.xml:

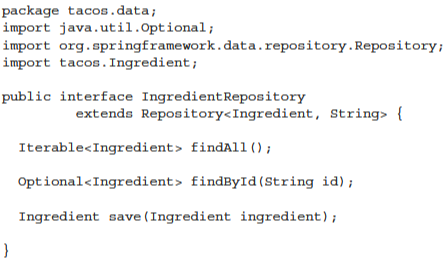


+You will no longer need JDBC starter that gave us JdbcTemplate, remove it:



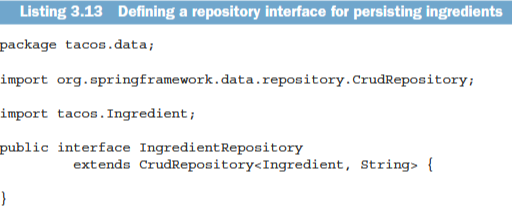
## 3.2.2 Defining repository interfaces

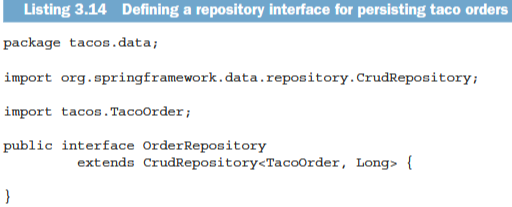
-Spring Data will **automatically** generate **implementation** for our repository interfaces at run time if interfaces **extend** one of **SD repository interfaces**:



+**Repository interface**: **1st para** is the type of **object** persisted by this repository, **2nd** **para** is the type of persisted **object’s ID** field.

-Spring Data also offers **CrudRepository**: base interface for **common operations** -> use this





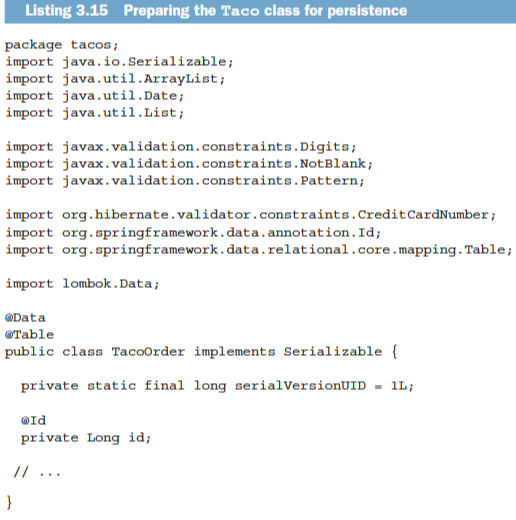
+CrudRepository already defines methods you need -> there’s no need to define them.

-There is **no need** to **write implementation**. When app starts, SD automatically generates them -> you can delete 2 implementation classes.

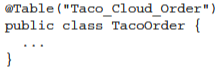
## 3.2.3 Annotating the domain for persistence

-The only other thing is annotate our domain classes so that SD JDBC will know how to persist them. -> **Annotating** the **identity property** with **@Id** + **optionally** annotate the **class** with **@Table**

**-**TacoOrder class annotated:



**+@Table** is optional: By **default**, object is mapped to a table based on domain class name (TacoOrder->Taco\_Order). You can use @Table to map object to a **different table name**:

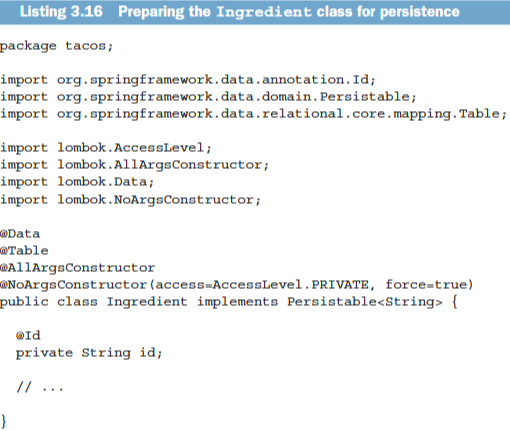


**+@Id**: designate the id property as the identity for a TacoOrder.

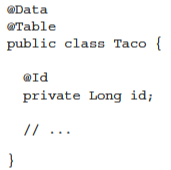
+All **other properties** will be mapped automatically to columns based on their property names. To define the column name mapping, use **@Column**:



-Ingredient:



-Taco:



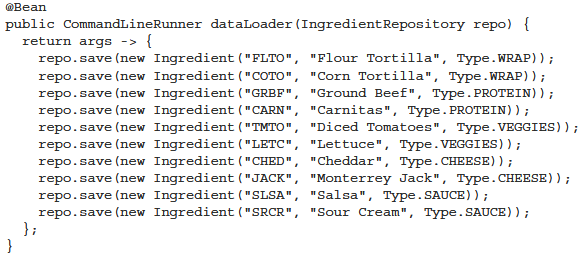
-IngredientRef: don’t need to change.

## 3.2.4 Preloading data with CommandLineRunner

-JdbcTemplate: **preload data** using **data.sql**, can use it with **SD JDBC**

-SB offers 2 interfaces for executing logic when app starts up: **CommandLineRunner** and **ApplicationRunner**. You can declare them as bean in configuration class using **@Bean**

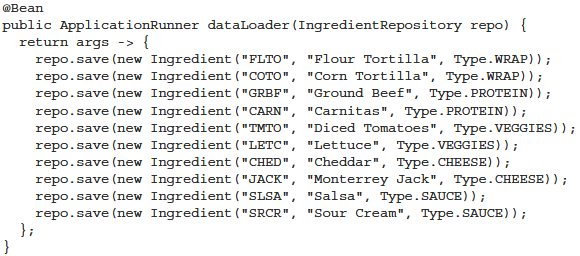
-CommandLineRunner:



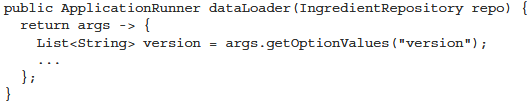
+IngredientRepository is injected into the bean method

+Functional interface: run() accept a single para containing all **command-line argument**. We don’t need them so args is ignored.

-ApplicationRunner:



+args para offers methods for accessing arugments as parsed components of command line.



-Both of interfaces can use the repositories to create persisted objects instead of SQL script -> work well for relational database and norelational database.

# 3.3 Persisting data with Spring Data JPA

-SD JDBC makes easy work for persisting data.

-Java Persistence API (JPA) is another popular option for working with data in relational database.

-SD JPA offers an approach to persistence with JPA similar to what SD JDBC gave us for JDBC.

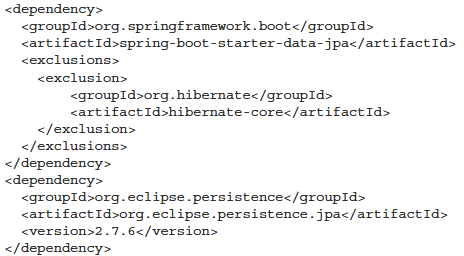
## 3.3.1 Adding Spring Data JPA to the project

-SD JPA is available to SB apps. This starter dependency brings in SD **JPA** + **Hibernate** as **JPA implementation**.



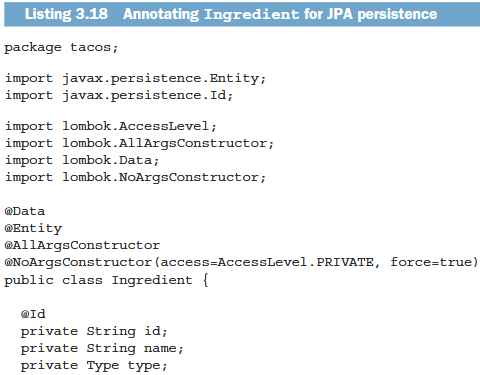
-If you want to use a different **JPA implementation**: exclude Hibernate dependency + JPA library:

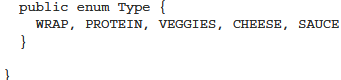
+Example: EcpliseLink instead of Hibernate



## 3.3.2 Annotating the domain as entities

-You must open up Ingredient, Taco, TacoOrder classes and throw annotations:





**+@Entity**: declare as JPA entity+ **@Id** (jakarta.persistence.Id) to designate id property.

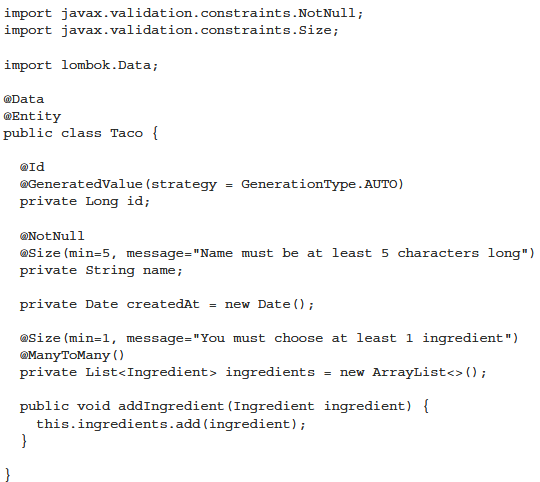
+implement **Persistable**: determine whether or not an entity was to be created new or update an existing entity, JPA sorts that out automatically.

**+@NoArgsConstructor**: JPA requires that entity have a no-argument constructor. Don’t use it: **AccessLeve.PRIVATE**, set final properties: **force:true**

**+@AllArgsContructor**: create Ingredient object will all properties initialized easier.

**+@RequireArgsConstructor**: @Data annotation implicitly adds a required arguments constructor. Don’t use when @NoArgsConstructor is used.

-Taco:

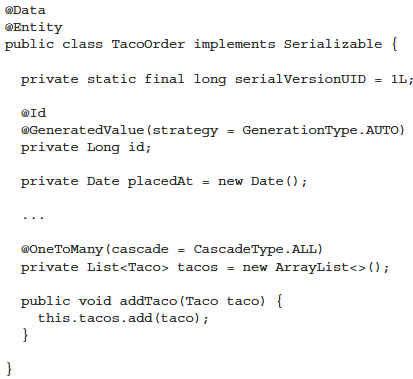


**+@GeneratedValue(strategy=GenerationType.AUTO)**

: rely on the database to automatically generate Id.

**+@ManyToMany()**: declare the relationship between a Taco and its associated Ingredient list. A Taco can have many Ingredient objects, an Ingredient can be part of many Tacos.

-TacoOrder:



**+@OneToMany**: tacos are all specific to this one order. **CascadeType.ALL**: if the order is deleted, the related tacos will also be deleted.

## 3.3.3 Declaring JPA repositories

-**CrudRepository** works equally well for SD JPA. This interface is commonly used across many of SD projects.

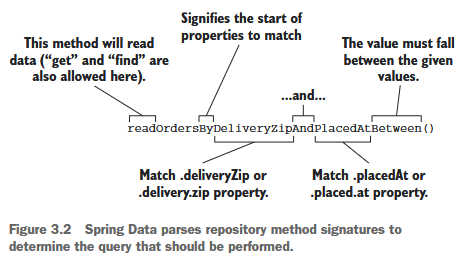
## 3.3.4 Customizing repositories

-In addition to CRUD operations, you also need to fetch all order base on ZIP code:

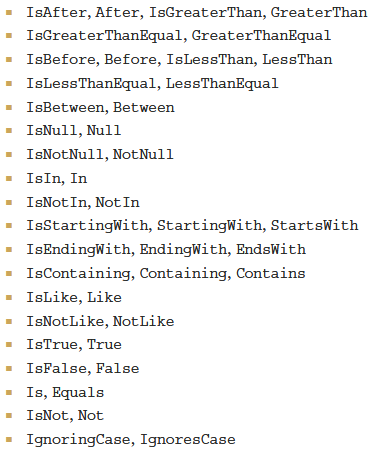


+When generating repository implementation, SD examines each method in interface, parses the method name, attempts to understand method’s purpose. SD defines a sort of domain-specific language (DSL) where persistence details are expressed in repository method signatures.

-**Repository method** are composed: **a verb + optional subject + By + a predicate** 



-Method signatures can also include operators:



+Alternatives for IgnoringCase and IgnoresCase: **AllIgnoringCase** or **AllIgnoreCase** to ignore case for all String comparisons



-You can also place **OrderBy** at the end of method name:

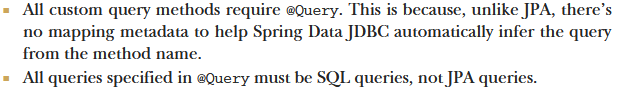


**-@Query**: specify the query to be perform when the method is called



+You can use @Query to perform any **JPA query** you can dream up.

+Custome query methods also work with **SD JDBC** with key differences:



# -Summary

