```
    return "redirect:/";
  }

}
```

Aside from injecting `OrderRepository` into the controller, the only significant changes in `OrderController` are in the `processOrder()` method. Here, the `Order` object submitted in the form (which also happens to be the same `Order` object maintained in session) is saved via the `save()` method on the injected `OrderRepository`.

Once the order is saved, you don't need it hanging around in a session anymore. In fact, if you don't clean it out, the order remains in session, including its associated tacos, and the next order will start with whatever tacos the old order contained. Therefore, the `processOrder()` method asks for a `SessionStatus` parameter and calls its `setComplete()` method to reset the session.

All of the JDBC persistence code is in place. Now you can fire up the Taco Cloud application and try it out. Feel free to create as many tacos and as many orders as you'd like.

You might also find it helpful to dig around in the database. Because you're using H2 as your embedded database, and because you have Spring Boot DevTools in place, you should be able to point your browser to http://localhost:8080/h2-console to see the H2 Console. The default credentials should get you in, although you'll need to be sure that the JDBC URL field is set to `jdbc:h2:mem:testdb`. Once logged in, you should be able to issue any query you like against the tables in the Taco Cloud schema.

Spring's `JdbcTemplate`, along with `SimpleJdbcInsert`, makes working with relational databases significantly simpler than plain vanilla JDBC. But you may find that JPA makes it even easier. Let's rewind your work and see how to use Spring Data to make data persistence even easier.

3.2 Persisting data with Spring Data JPA

The Spring Data project is a rather large umbrella project comprised of several subprojects, most of which are focused on data persistence with a variety of different database types. A few of the most popular Spring Data projects include these:

- *Spring Data JPA*—JPA persistence against a relational database
- *Spring Data MongoDB*—Persistence to a Mongo document database
- *Spring Data Neo4j*—Persistence to a Neo4j graph database
- *Spring Data Redis*—Persistence to a Redis key-value store
- *Spring Data Cassandra*—Persistence to a Cassandra database

One of the most interesting and useful features provided by Spring Data for all of these projects is the ability to automatically create repositories, based on a repository specification interface.

To see how Spring Data works, you're going to start over, replacing the JDBC-based repositories from earlier in this chapter with repositories created by Spring Data JPA. But first, you need to add Spring Data JPA to the project build.

3.2.1 Adding Spring Data JPA to the project

Spring Data JPA is available to Spring Boot applications with the JPA starter. This starter dependency not only brings in Spring Data JPA, but also transitively includes Hibernate as the 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, then you'll need to, at least, exclude the Hibernate dependency and include the JPA library of your choice. For example, to use EclipseLink instead of Hibernate, you'll need to alter the build as follows:

```
<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>
```

Note that there may be other changes required, depending on your choice of JPA implementation. Consult the documentation for your chosen JPA implementation for details. Now let's revisit your domain objects and annotate them for JPA persistence.

3.2.2 Annotating the domain as entities

As you'll soon see, Spring Data does some amazing things when it comes to creating repositories. But unfortunately, it doesn't help much when it comes to annotating your domain objects with JPA mapping annotations. You'll need to open up the `Ingredient`, `Taco`, and `Order` classes and throw in a few annotations. First up is the `Ingredient` class.

Listing 3.16 Annotating `Ingredient` for JPA persistence

```
package tacos;

import javax.persistence.Entity;
import javax.persistence.Id;

import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Entity
public class Ingredient {

  @Id
  private final String id;
  private final String name;
  private final Type type;

  public static enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
  }

}
```

In order to declare this as a JPA entity, `Ingredient` must be annotated with `@Entity`. And its `id` property must be annotated with `@Id` to designate it as the property that will uniquely identify the entity in the database.

In addition to the JPA-specific annotations, you'll also note that you've added a `@NoArgsConstructor` annotation at the class level. JPA requires that entities have a no-arguments constructor, so Lombok's `@NoArgsConstructor` does that for you. You don't want to be able to use it, though, so you make it `private` by setting the access attribute to `AccessLevel.PRIVATE`. And because there are `final` properties that must be set, you also set the `force` attribute to `true`, which results in the Lombok-generated constructor setting them to `null`.

You also add a `@RequiredArgsConstructor`. The `@Data` implicitly adds a required arguments constructor, but when a `@NoArgsConstructor` is used, that constructor gets removed. An explicit `@RequiredArgsConstructor` ensures that you'll still have a required arguments constructor in addition to the `private` no-arguments constructor.

Now let's move on to the `Taco` class and see how to annotate it as a JPA entity.

Listing 3.17 Annotating `Taco` as an entity

```
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.OneToMany;
import javax.persistence.PrePersist;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import lombok.Data;

@Data
@Entity
public class Taco {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @NotNull
  @Size(min=5, message="Name must be at least 5 characters long")
  private String name;

  private Date createdAt;

  @ManyToMany(targetEntity=Ingredient.class)
  @Size(min=1, message="You must choose at least 1 ingredient")
  private List<Ingredient> ingredients;

  @PrePersist
  void createdAt() {
    this.createdAt = new Date();
  }
}
```

As with `Ingredient`, the `Taco` class is now annotated with `@Entity` and has its `id` property annotated with `@Id`. Because you're relying on the database to automatically generate the ID value, you also annotate the `id` property with `@GeneratedValue`, specifying a `strategy` of `AUTO`.

To declare the relationship between a `Taco` and its associated `Ingredient` list, you annotate `ingredients` with `@ManyToMany`. A `Taco` can have many `Ingredient` objects, and an `Ingredient` can be a part of many `Tacos`.

You'll also notice that there's a new method, `createdAt()`, which is annotated with `@PrePersist`. You'll use this to set the `createdAt` property to the current date and time before `Taco` is persisted. Finally, let's annotate the `Order` object as an entity. The next listing shows the new `Order` class.

Listing 3.18 Annotating `Order` as a JPA entity

```
package tacos;
import java.io.Serializable;
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.OneToMany;
import javax.persistence.PrePersist;
import javax.persistence.Table;
import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.NotBlank;
import lombok.Data;

@Data
@Entity
@Table(name="Taco_Order")
public class Order implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  private Date placedAt;

  ...

  @ManyToMany(targetEntity=Taco.class)
  private List<Taco> tacos = new ArrayList<>();

  public void addDesign(Taco design) {
    this.tacos.add(design);
  }

  @PrePersist
  void placedAt() {
    this.placedAt = new Date();
  }

}
```

As you can see, the changes to `Order` closely mirror the changes to `Taco`. But there's one new annotation at the class level: `@Table`. This specifies that `Order` entities should be persisted to a table named `Taco_Order` in the database.

Although you could have used this annotation on any of the entities, it's necessary with Order. Without it, JPA would default to persisting the entities to a table named Order, but *order* is a reserved word in SQL and would cause problems. Now that the entities are properly annotated, it's time to write your repositories.

3.2.3 Declaring JPA repositories

In the JDBC versions of the repositories, you explicitly declared the methods you wanted the repository to provide. But with Spring Data, you can extend the CrudRepository interface instead. For example, here's the new IngredientRepository interface:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
         extends CrudRepository<Ingredient, String> {

}
```

CrudRepository declares about a dozen methods for CRUD (create, read, update, delete) operations. Notice that it's parameterized, with the first parameter being the entity type the repository is to persist, and the second parameter being the type of the entity ID property. For IngredientRepository, the parameters should be Ingredient and String.

You can similarly define the TacoRepository like this:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Taco;

public interface TacoRepository
         extends CrudRepository<Taco, Long> {

}
```

The only significant differences between IngredientRepository and TacoRepository are the parameters to CrudRepository. Here, they're set to Taco and Long to specify the Taco entity (and its ID type) as the unit of persistence for this repository interface. Finally, the 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> {

}
```

And now you have your three repositories. You might be thinking that you need to write the implementations for all three, including the dozen methods for each implementation. But that's the good news about Spring Data JPA—there's no need to write an implementation! When the application starts, Spring Data JPA automatically generates an implementation on the fly. This means the repositories are ready to use from the get-go. Just inject them into the controllers like you did for the JDBC-based implementations, and you're done.

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 see how to customize the repositories to perform queries unique to your domain.

3.2.4 Customizing JPA repositories

Imagine that in addition to the basic CRUD operations provided by `CrudRepository`, you also need to fetch all the orders delivered to a given ZIP code. As it turns out, this can easily be addressed by adding the following method declaration to `OrderRepository`:

```
List<Order> findByDeliveryZip(String deliveryZip);
```

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). In essence, Spring Data defines a sort of miniature domain-specific language (DSL) where persistence details are expressed in repository method signatures.

Spring Data knows that this method is intended to find `Orders`, because you've parameterized `CrudRepository` with `Order`. The method name, `findByDeliveryZip()`, makes it clear that this method should find all `Order` entities by matching their `deliveryZip` property with the value passed in as a parameter to the method.

The `findByDeliveryZip()` method is simple enough, but Spring Data can handle even more-interesting method names as well. Repository methods are composed of a verb, an optional subject, the word *By*, and a predicate. In the case of `findByDeliveryZip()`, the verb is *find* and the predicate is *DeliveryZip*; the subject isn't specified and is implied to be an `Order`.

Let's consider another, more complex example. Suppose that you need to query for all orders delivered to a given ZIP code within a given date range. In that case, the following method, when added to `OrderRepository`, might prove useful:

```
List<Order> readOrdersByDeliveryZipAndPlacedAtBetween(
      String deliveryZip, Date startDate, Date endDate);
```

Figure 3.2 illustrates how Spring Data parses and understands the readOrdersByDeliveryZipAndPlacedAtBetween() method when generating the repository implementation. As you can see, the verb in readOrdersByDeliveryZipAndPlacedAtBetween() is read. Spring Data also understands find, read, and get as synonymous for fetching one or more entities. Alternatively, you can also use count as the verb if you only want the method to return an int with the count of matching entities.

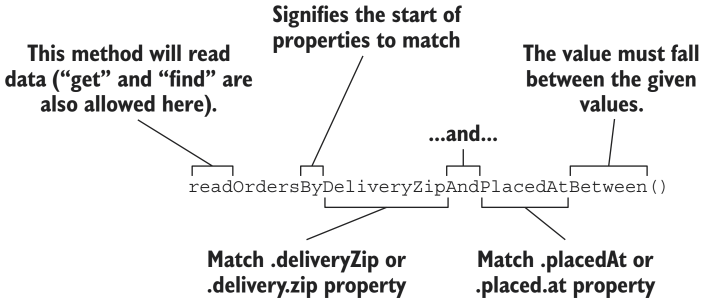
Signifies the start of properties to match
This method will read data ("get" and "find" are also allowed here).
The value must fall between the given values.
...and...
readOrdersByDeliveryZipAndPlacedAtBetween()
Match .deliveryZip or .delivery.zip property
Match .placedAt or .placed.at property

Figure 3.2 Spring Data parses repository method signatures to determine the query that should be performed.

Although the subject of the method is optional, here it says Orders. Spring Data ignores most words in a subject, so you could name the method readPuppiesBy... and it would still find Order entities, as that is the type that CrudRepository is parameterized with.

The predicate follows the word By in the method name and is the most interesting part of the method signature. In this case, the predicate refers to two Order properties: deliveryZip and placedAt. The deliveryZip property must be equal to the value passed into the first parameter of the method. The keyword Between indicates that the value of deliveryZip must fall between the values passed into the last two parameters of the method.

In addition to an implicit Equals operation and the Between operation, Spring Data method signatures can also include any of these operators:

- IsAfter, After, IsGreaterThan, GreaterThan
- IsGreaterThanEqual, GreaterThanEqual
- IsBefore, Before, IsLessThan, LessThan
- IsLessThanEqual, LessThanEqual
- IsBetween, Between
- IsNull, Null
- IsNotNull, NotNull
- IsIn, In
- IsNotIn, NotIn
- IsStartingWith, StartingWith, StartsWith

- IsEndingWith, EndingWith, EndsWith
- IsContaining, Containing, Contains
- IsLike, Like
- IsNotLike, NotLike
- IsTrue, True
- IsFalse, False
- Is, Equals
- IsNot, Not
- IgnoringCase, IgnoresCase

As alternatives for IgnoringCase and IgnoresCase, you can place either AllIgnoringCase or AllIgnoresCase on the method to ignore case for all String comparisons. For example, consider the following method:

```
List<Order> findByDeliveryToAndDeliveryCityAllIgnoresCase(
        String deliveryTo, String deliveryCity);
```

Finally, you can also place OrderBy at the end of the method name to sort the results by a specified column. For example, to order by the deliveryTo property:

```
List<Order> findByDeliveryCityOrderByDeliveryTo(String city);
```

Although the naming convention can be useful for relatively simple queries, it doesn't take much imagination to see that method names could get out of hand for more-complex queries. In that case, feel free to name the method anything you want and annotate it with @Query to explicitly specify the query to be performed when the method is called, as this example shows:

```
@Query("Order o where o.deliveryCity='Seattle'")
List<Order> readOrdersDeliveredInSeattle();
```

In this simple usage of @Query, you ask for all orders delivered in Seattle. But you can use @Query to perform virtually any query you can dream up, even when it's difficult or impossible to achieve the query by following the naming convention.

Summary

- Spring's JdbcTemplate greatly simplifies working with JDBC.
- PreparedStatementCreator and KeyHolder can be used together when you need to know the value of a database-generated ID.
- For easy execution of data inserts, use SimpleJdbcInsert.
- Spring Data JPA makes JPA persistence as easy as writing a repository interface.