JPA @Embedded And @Embeddable

In this tutorial, we'll see how we can map one entity that contains embedded properties to a single database table.

So, for this purpose, we'll use the @Embeddable and @Embedded annotations provided by the Java Persistence API (JPA).

Data Model Context

First of all, let's define a table called company.

The company table will store basic information such as company name, address, and phone, as well as the information of a contact person:

```
public class Company {
    private Integer id;
    private String name;
    private String address;
    private String phone;
    private String contactFirstName;
    private String contactLastName;
    private String contactPhone;
    // standard getters, setters
}
```

The contact person, though, seems like it should be abstracted out to a separate class. The problem is that we don't want to create a separate table for those details. So, let's see what we can do.

@Embeddable

JPA provides the @Embedded annotation to declare that a class will be embedded by other entities.

Let's define a class to abstract out the contact person details:

```
@Embeddable
public class ContactPerson {
   private String firstName;
   private String lastName;
   private String phone;
   // standard getters, setters
}
```

@Embedded

The JPA annotation @Embedded is used to embed a type into another entity.

Let's next modify our Company class. We'll add the JPA annotations and we'll also change to use ContactPerson instead of separate fields:

```
@Entity
public class Company {
```

```
    @Id
    @GeneratedValue
    private Integer id;

    private String name;
     private String address;
     private String phone;

    @Embedded
    private ContactPerson contactPerson;

    // standard getters, setters
}
```

As a result, we have our entity Company, embedding contact person details, and mapping to a single database table.

We still have one more problem, though, and that is how JPA will map these fields to database columns.

Attributes Override

The thing is that our fields were called things like contactFirstName in our original Company class and now firstName in our ContactPerson class. So, JPA will want to map these to contact_first_name and first_name, respectively.

Aside from being less than ideal, it will actually break us with our now-duplicated phone column.

So, we can use @AttributeOverrides and @AttibuteOverride to override the column properties of our embedded type.

Let's add this to the ContactPerson field in our Company entity:

```
@Embedded
@AttributeOverrides({
  @AttributeOverride( name = "firstName", column = @Column(name = "contact_first_name")),
  @AttributeOverride( name = "lastName", column = @Column(name = "contact_last_name")),
  @AttributeOverride( name = "phone", column = @Column(name = "contact_phone"))
})
private ContactPerson contactPerson;
```

Note that, since these the annotations go on the field, we can have different overrides for each enclosing entity.

In this tutorial, we've configured an entity with some embedded attributes and mapped them to the same database table as the enclosing entity. For that, we used the @Embedded, @Embeddable, @AttributeOverrides and @AttributeOverride annotations provided by the Java Persistence API.