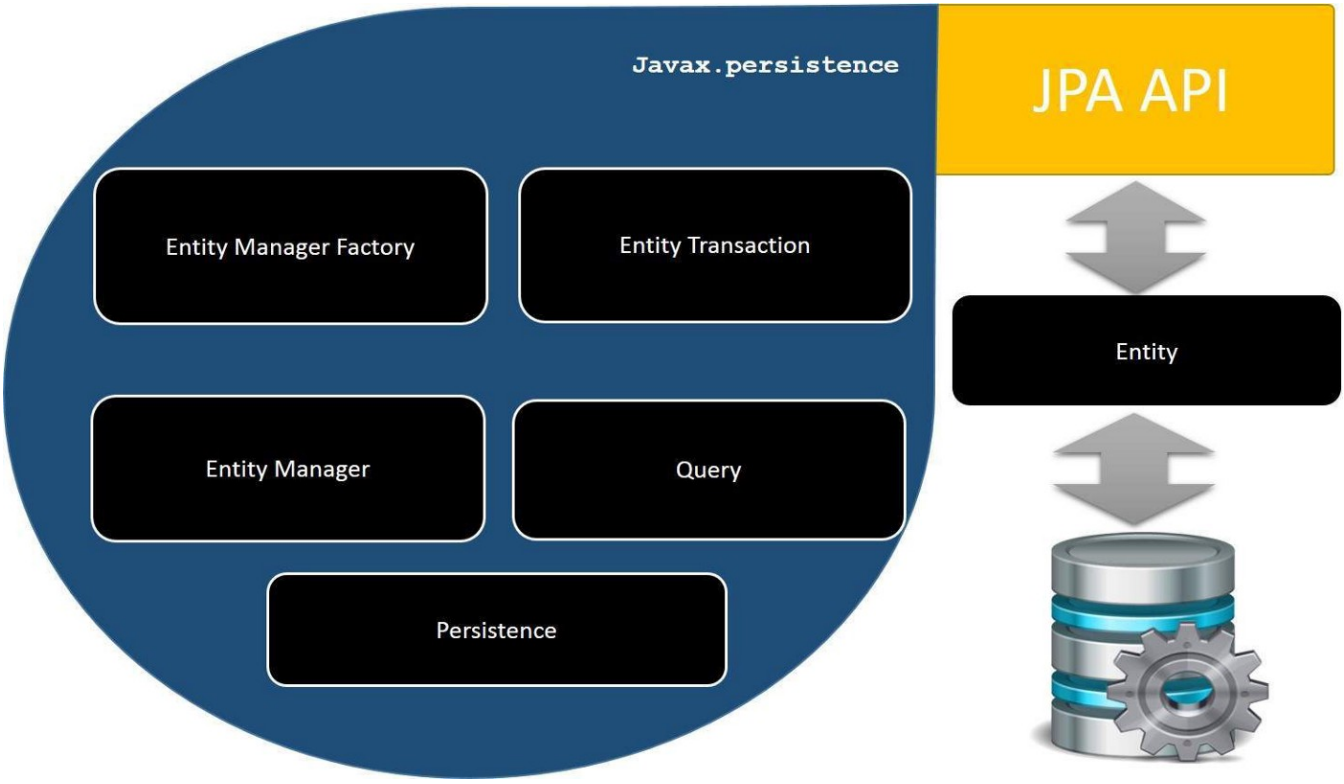
Published in Nerd For Tech

Daniel S. Blanco Follow
Sep 12, 2020 · 3 min read · 32:45

Apache Camel: CRUD with JPA



We have already seen other posts about Apache Camel, you can see all in my profile. Today we will see how to do the database operations with JPA.

As always we will start with the configuration. Talking about dependencies, we need to add the following ones:

- The Apache Camel *camel-jpa-starter* component that will allow us to use the JPA component.

We must remember some basic things about JPA. One is the quality of handle relational data as Java objects. Allowing us to manage within Java classes the information inside a database. These classes will be called *Entities*. Another is the existence of *JPQL*, a language that will allow us to perform queries on these entities.

Thanks to Spring boot and the camel starter library we will not have to do much about configuration, but at the level of the Java class we will have to add a few annotations:

```java
@Entity(name = "BOOK")
@Table(name = "BOOK")
@NamedQuery(name = "findAll", query = "SELECT b FROM BOOK b")
public class Book {

    @JsonProperty
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @JsonProperty
    private String name;
    @JsonProperty
    private String author;
}
```

Book_JPA.Java hosted with ❤ by GitHub view raw

@Entity will allow us to indicate that it is a class to be managed through JPA. With *@Id* we will indicate which is its primary key and with *@GeneratedValue* its generation will be managed. In the case of *MySQL,* it will be Auto.

In the first example, we'll see that the *JPA* component is very similar to the *SQL* component. Only that we will use *JPQL* and we must indicate which is the entity on which we are going to perform the query.

```java
rest().get("book/{id}").description("Details of an book by id")
   .outType(Book.class).produces(MediaType.APPLICATION_JSON_VALUE).route()
   .toD("jpa://" + Book.class.getName() + "?query=select b from " + Book.class.getName()+ " b wh
   .log("--- select a book ${body} ---");
```

ApacheCamelJPA_GET.java hosted with ❤ by GitHub view raw

Furthermore, in this specific case, we want to obtain a book based on an identifier that comes as a parameter of the request, through the expression language. In order to achieve this, we will use the *ToD* method instead of *To*. It will allow us to route dynamically the flow and have the expression language interpreted.

Another quality of *JPA* is the *namedQuery*, which allows us to define *JPQL* queries at the class level. In the following example, we will see how to use *namedQueries* through the *JPA* component. The query to use is *findAll* (we have seen the declaration before).

```
rest().get("book").produces(MediaType.APPLICATION_JSON_VALUE).route()
      .to("jpa://"+ Book.class.getName() + "?resultClass="+ Book.class.getName() + "&namedQuery=
      .log("---select all books---");
```

ApacheCamelJPA_findAll.java hosted with ❤ by GitHub view raw

In this second example, we can see the difference in using the *outType* method to indicate the type of an object at the output. Or if we prefer, we can indicate the same through the query parameter *resultClass* of the *JPA* component itself.

We can also do operations that allow us to persist such objects in the database. As we can see in previous example, in the case of saving the value, the operation is very simple. We only have to indicate the object to persist and use the query parameter *usePersist* to indicate that it should persist the object. Of course, the flow must include the data of the object to be stored.

```
rest().post("book").produces(MediaType.APPLICATION_JSON_VALUE).type(Book.class)
      .route().routeId("postBookRoute").log("--- binded ${body} ---")
      .to("jpa:" + Book.class.getName() + "?usePersist=true")
      .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(201)).setBody(constant(null));
```

ApacheCamelJPA_POST.java hosted with ❤ by GitHub view raw

For updates and deletion of data, we need to use the query parameter *useExecuteUpdate*. In this example, we will use the process method to create the object to be updated from the header and message body data.

```
                Book book = exchange.getIn().getBody(Book.class);
                Integer id = Integer.valueOf(exchange.getIn().getHeader("id").toString());
                book.setId(id);
                exchange.getIn().setBody(book, Book.class);
            }
        }).to("jpa:" + Book.class.getName() + "?useExecuteUpdate=true")
        .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(200)).setBody(constant(null));
```

ApacheCamel_JPA_PUT.java hosted with ❤ by GitHub view raw

Finally, we will see the example associated with data deletion. And by the way we will see how to make native queries. That is the use of common *SQL* language instead of *JPQL*.

```
rest().delete("book/{id}").produces(MediaType.APPLICATION_JSON_VALUE).route()
        .toD("jpa:com.example.home.ApacheCamelRestExample.pojo.Book"
                + "?nativeQuery=DELETE FROM library.BOOK where id = ${header.id}&useExecuteUpdate=true
        .setHeader(Exchange.CONTENT_TYPE, constant(MediaType.APPLICATION_JSON_VALUE))
        .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(200)).setBody(constant(null));
```

ApacheCamel_JPA_DELETE.java hosted with ❤ by GitHub view raw

As you can see, some queries can be a little more difficult than using simple *SQL, b*ut also others are very simple. And we are always working with objects. Which, when programming in an object-oriented language is very grateful.