Quarkus - Simplified Hibernate ORM with Panache and Kotlin

Hibernate ORM is the de facto standard JPA implementation and is well-known in the Java ecosystem. Panache offers a new layer atop this familiar framework. This guide will not dive in to the specifics of either as those are already covered in the Panache guide. In this guide, we'll cover the Kotlin specific changes needed to use Panache in your Kotlin-based Quarkus applications.

First: an example

As we saw in the Panache guide, Panache allows us to extend the functionality in our entities and repositories (also known as DAOs) with some automatically provided functionality. When using Kotlin, the approach is very similar to what we see in the Java version with a slight change or two. To Panache-enable your entity, you would define it something like:

```
@Entity
class Person: PanacheEntity {
    lateinit var name: String
    lateinit var birth: LocalDate
    lateinit var status: Status
}
```

As you can see our entities remain simple. There is, however, a slight difference from the Java version. The Kotlin language doesn't support the notion of static methods in quite the same way as Java does. Instead, we must use a [companion object](https://kotlinlang.org/docs/tutorials/kotlin-for-py/objects-and-companion-objects.html#companion-objects):

```
@Entity
class Person : PanacheEntity {
    companion object: PanacheCompanion<Person> {  ①
        fun findByName(name: String) = find("name",
name).firstResult()
        fun findAlive() = list("status", Status.Alive)
        fun deleteStefs() = delete("name", "Stef")
    }

    lateinit var name: String  ②
    lateinit var birth: LocalDate
    lateinit var status: Status
}
```

① The companion object holds all the methods not related to a specific instance allowing for general management and querying bound to a specific type.

② Here there are options, but we've chosen the `lateinit` approach. This allows us to declare these fields as non-null knowing they will be properly assigned either by the constructor (not shown) or by hibernate loading data from the database.

These types differ from the Java types mentioned in those tutorials. For Kotlin support, all the Panache types will be found in the `io.quarkus.hibernate.orm.panache.kotlin` package. This subpackage allows for the distinction between the Java and Kotlin variants and allows for both to be used unambiguously in a single project.

In the Kotlin version, we've simply moved the bulk of the `active record pattern` functionality to the `companion object`. Apart from this slight change, we can then work with our types in ways that map easily from the Java side of world.

Using the repository pattern

Defining your entity

When using the repository pattern, you can define your entities as regular JPA entities.

```
@Entity
class Person {
    @Id
    @GeneratedValue
    var id: Long? = null;
    lateinit var name: String
    lateinit var birth: LocalDate
    lateinit var status: Status
}
```

Defining your repository

When using Repositories, you get the exact same convenient methods as with the active record pattern, injected in your Repository, by making them implement `PanacheRepository`:

```
class PersonRepository: PanacheRepository<Person> {
     fun findByName(name: String) = find("name",
name).firstResult()
     fun findAlive() = list("status", Status.Alive)
     fun deleteStefs() = delete("name", "Stef")
}
```

All the operations that are defined on `PanacheEntityBase` are available on your repository, so using it is exactly the same as using the active record pattern, except you need to inject it:

```
@Inject
lateinit var personRepository: PersonRepository

@GET
fun count() = personRepository.count()
```

Most useful operations

Once you have written your repository, here are the most common operations you will be able to perform:

```
// creating a person
var person = Person()
person.name = "Stef"
person.birth = LocalDate.of(1910, Month.FEBRUARY, 1)
person.status = Status.Alive

// persist it
personRepository.persist(person)

// note that once persisted, you don't need to explicitly save your
entity: all
// modifications are automatically persisted on transaction commit.

// check if it's persistent
if(personRepository.isPersistent(person)){
    // delete it
    personRepository.delete(person)
}

// getting a list of all Person entities
val allPersons = personRepository.listAll()

// finding a specific person by ID
person = personRepository.findById(personId) ?: throw Exception("No
person with that ID")

// finding all living persons
val livingPersons = personRepository.list("status", Status.Alive)

// counting all persons
val countAll = personRepository.count()

// counting all living persons
val countAlive = personRepository.count("status", Status.Alive)

// delete all living persons
personRepository.delete("status", Status.Alive)

// delete all persons
personRepository.deleteAll()

// delete by id
val deleted = personRepository.deleteById(personId)

// set the name of all living persons to 'Mortal'
personRepository.update("name = 'Mortal' where status = ?1",
Status.Alive)
```

All `list` methods have equivalent `stream` versions.

```
val persons = personRepository.streamAll();
val namesButEmmanuels = persons
    .map { it.name.toLowerCase() }
    .filter { it != "emmanuel" }
```

The `stream` methods require a transaction to work.

The rest of the documentation show usages based on the active record pattern only, but keep in mind that they can be performed with the repository pattern as well. The repository pattern examples have been omitted for brevity.

For more examples, please consult the Java version for complete details. Both APIs are the same and work identically except for some Kotlin-specific tweaks to make things feel more natural to Kotlin developers. These tweaks include things like better use of nullability and the lack of `Optional` on API methods.

Setting up and configuring Hibernate ORM with Panache

To get started using Panache with Kotlin, you can, generally, follow the steps laid out in the Java tutorial. The biggest change to configuring your project is the Quarkus artifact to include. You can, of course, keep the Java version if you need but if all you need are the Kotlin APIs then include the following dependency instead:

```
<dependencies>
    <!-- Hibernate ORM specific dependencies -->
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-hibernate-orm-panache-
kotlin</artifactId>  ①
    </dependency>
</dependencies>
```

① Note the addition of `-kotlin` on the end. Generally you'll only need this version but if your project will be using both Java and Kotlin code, you can safely include both artifacts.