HIBERNATE

In Relation To

Back to home

Hibernate ORM with Panache in Quarkus

Posted by Stephane Epardaud | Nov 19, 2019 Hibernate ORM Quarkus

We have talked about this Quarkus "Panache" thing recently in this blog, but we should probably get into a little more detail about what it is, and why it matters.

First thing, to clarify, we're talking about "Hibernate ORM with Panache", which roughly means we're talking about a flamboyant and reckless flavour of Hibernate ORM.

This means two essential things:

- It's the full Hibernate ORM, not anything less
- But it looks different to what you're used to
- You can fallback at any time and even mix vanilla JPA and Panache

Let's start with a typical entity example

The JPA entity style we've all been taught to write for a long time will look like this:

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String firstName;
    private String lastName;
    private Date birth;
```

```
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Date getBirth() {
        return birth;
    }

    public void setBirth(Date birth) {
        this.birth = birth;
    }
}
```

And we will traditionally define the model logic in a Data Access Object bean such as this:

```
@Singleton
public class PersonDao {

    @Inject
    private EntityManager entityManager;
```

```
    public void persist(Person person) {
        entityManager.persist(person);
    }

    public void delete(Person person) {
        entityManager.remove(person);
    }

    public Person findById(Long id) {
        return entityManager.find(Person.class, id);
    }

    public List<Person> findAll() {
        return entityManager.createQuery("FROM Person", Person.class).getResult
    }

    public List<Person> findByName(String lastName) {
        return entityManager.createQuery("FROM Person WHERE lastName = :lastNam
    }

    public List<Person> findBornAfter(Date date) {
        return entityManager.createQuery("FROM Person WHERE birth > :date", Per
    }
}
```

Which in turn will be used like this in a JAX-RS REST endpoint:

```
@Path("/")
public class PersonEndpoint {
    @Inject
    private PersonDao personDao;

    @GET
    @Path("people")
    public List<Person> all() {
        return personDao.findAll();
    }

    @GET
    @Path("people/by-name")
    public List<Person> findByName(@PathParam String name) {
        return personDao.findByName(name);
```

```
    }

    @GET
    @Path("people/born-after")
    public List<Person> findBornAfter(@PathParam Date date) {
        return personDao.findBornAfter(date);
    }

    @GET
    @Path("person/{id}")
    public Person findById(@PathParam Long id) {
        Person p = personDao.findById(id);
        if(p == null)
            throw new WebApplicationException(Status.NOT_FOUND);
        return p;
    }

    @PUT
    @Path("person/{id}")
    public void updatePerson(@PathParam Long id, Person newPerson) {
        Person p = personDao.findById(id);
        if(p == null)
            throw new WebApplicationException(Status.NOT_FOUND);
        p.setBirth(newPerson.getBirth());
        p.setFirstName(newPerson.getFirstName());
        p.setLastName(newPerson.getLastName());
    }

    @DELETE
    @Path("person/{id}")
    public void deletePerson(@PathParam Long id) {
        Person p = personDao.findById(id);
        if(p == null)
            throw new WebApplicationException(Status.NOT_FOUND);
        personDao.delete(p);
    }

    @POST
    @Path("people")
    public Response newPerson(@Context UriInfo uriInfo, Person newPerson) {
        Person p = new Person();
        p.setBirth(newPerson.getBirth());
        p.setFirstName(newPerson.getFirstName());
```

```
        p.setLastName(newPerson.getLastName());
        personDao.persist(p);

        URI uri = uriInfo.getAbsolutePathBuilder()
                .path(PersonEndpoint.class)
                .path(PersonEndpoint.class, "findById")
                .build(p.getId());
        return Response.created(uri).build();
    }
}
```

We did not use any Data Transfer Object in our REST service, simply to keep the example short and to the point. We dot not advise that you use your entities as DTOs for real code.

A few observations on the traditional JPA way

We've all seen hundreds of entities and DAOs written this way. They don't have anything surprising.

What they do have plenty of, on the other hand is boilerplate:

- The generated ID field. Very often, all your entities will use the same auto-generated ID type.
- All those property accessors that do nothing in the entity. They are required for encapsulation, and because Java does not support first-class properties in the language. Most people either generate them from their IDE, or using Lombok.
- All those `persist` / `delete` / `findById` / `findAll` methods on every DAO. All DAOs have them.
- Those DAO queries all start with `FROM Person` and have to repeat the `Person.class` all over the place.

A first look at what Hibernate ORM with Panache can do for us

Let's jump forward to Quarkus and in particular, what Hibernate ORM with Panache can do for us. It turns out to be quite a lot.

Quarkus allows us to do a lot of bytecode modification at build-time, which (among many benefits) lets us side-step Java's lack of support for first-class properties by:

- Writing public fields instead of private+getter+setter
- Hibernate ORM with Panache will actually generate any missing getter+setter for public fields, and
- It will replace all field accesses with accesses to the getters and setters.

This system allows us to write code as if we were using public fields, but behind the scenes, we still get encapsulation and forward-compatibility if we ever add getters or setters that do more than just access the field.

On top of that, Hibernate ORM with Panache comes with support for DAOs that already have a lot of the methods that you commonly write.

We can thus rewrite our previous entity class by extending `PanacheEntity` which comes with a predefined auto-generated ID field:

```
@Entity
public class Person extends PanacheEntity {

    public String firstName;
    public String lastName;
    public Date birth;
}
```

And we can rewrite our DAO by extending `PanacheRepository` to get all the common methods:

```
@Singleton
public class PersonDao implements PanacheRepository<Person> {

    public List<Person> findByName(String lastName) {
        return find("lastName", lastName).list();
    }
```

```
    public List<Person> findBornAfter(Date date) {
        return find("birth > :date", Parameters.with("date", date)).list();
    }
}
```

This is already a long way towards reducing boilerplate, don't you think?

Note that the `find` convenience method allows HQL, but also *simplified* HQL (think of it as contextualized HQL):

- If your query is empty, it expands to `FROM <entityType>`
- If your query starts with `FROM` or `SELECT`, it is left alone as HQL
- If your query starts with `ORDER BY…` it expands to `FROM <entityType> ORDER BY…`
- If your query only has a single property and argument, it expands to `FROM <entityType> WHERE <property> = <argument>`
- Otherwise, your query is taken as a `WHERE…` clause and expands to `FROM <entityType> WHERE…`

This allows many simple queries to be simplified to a minimum, while allowing complex queries to be left as-is.

Now, our REST endpoint is not changed much, but let's include it for good measure:

```
@Path("/")
public class PersonEndpoint {
    @Inject
    private PersonDao personDao;

    @GET
    @Path("people")
    public List<Person> all() {
        return personDao.findAll().list();
    }

    @GET
    @Path("people/by-name")
    public List<Person> findByName(@PathParam String name) {
```

```
        return personDao.findByName(name);
    }

    @GET
    @Path("people/born-after")
    public List<Person> findBornAfter(@PathParam Date date) {
        return personDao.findBornAfter(date);
    }

    @GET
    @Path("person/{id}")
    public Person findById(@PathParam Long id) {
        Person p = personDao.findById(id);
        if(p == null)
            throw new WebApplicationException(Status.NOT_FOUND);
        return p;
    }

    @PUT
    @Path("person/{id}")
    public void updatePerson(@PathParam Long id, Person newPerson) {
        Person p = personDao.findById(id);
        if(p == null)
            throw new WebApplicationException(Status.NOT_FOUND);
        p.birth = newPerson.birth;
        p.firstName = newPerson.firstName;
        p.lastName = newPerson.lastName;
    }

    @DELETE
    @Path("person/{id}")
    public void deletePerson(@PathParam Long id) {
        Person p = personDao.findById(id);
        if(p == null)
            throw new WebApplicationException(Status.NOT_FOUND);
        personDao.delete(p);
    }

    @POST
    @Path("people")
    public Response newPerson(@Context UriInfo uriInfo, Person newPerson) {
        Person p = new Person();
        p.birth = newPerson.birth;
```

```
        p.firstName = newPerson.firstName;
        p.lastName = newPerson.lastName;
        personDao.persist(p);

        URI uri = uriInfo.getAbsolutePathBuilder()
                .path(PersonEndpoint.class)
                .path(PersonEndpoint.class, "findById")
                .build(p.id);
        return Response.created(uri).build();
    }
}
```

Going the extra mile and getting rid of the DAO

Data Access Objects are mostly useful when you have one or more of the following situations:

- The entity type is shared between projects written for different stacks. One project will use DAOs written for WildFly, another for Spring.
- The entity type is shared between projects written for different use-cases. One project will handle the entity in one way, while another will differ entirely.
- You need to mock your DAOs in tests.
- Your entity type is crammed so full of getters and setters that adding any model method will exceed the maximum method count.

While the last reason started as a joke, we humans have the tendency to split things up when they are getting too big. A Class with tons of getters/setters makes us reluctant to add more methods.

If you don't absolutely need DAOs, they come with drawbacks:

- You need to have one extra class per entity.
- You need to inject DAOs everywhere you use them.
- You cannot inject DAOs in methods without going out and adding a field, making this quite costly in terms of editing flow.
- You cannot discover a DAO methods without injecting it and trying completion. If this is

not the DAO you're looking for, you need to go back to the injected field and change its type and name to try again.

- Your IDEs are not helping you with any of these drawbacks.
- Any model refactoring requires you to examine queries in the DAO that corresponds to the entity you modified, making this poorly encapsulated.

In Hibernate ORM with Panache, we support the DAO use-cases, as we've seen, but we advise users to skip DAOs entirely and put the model methods in the entity class as static methods. They can be copied directly from the `PanacheRepository` to the entity class by adding the `static` modifier.

This allows you to:

- Create one less class per entity
- Keep entity model refactorying within a single file
- Do not require injection to manipulate them (does not break the editing flow)
- Have great discoverability: just type the entity type and complete to get all methods

Let's review our new entity class:

```
@Entity
public class Person extends PanacheEntity {

    public String firstName;
    public String lastName;
    public Date birth;

    public static List<Person> findByName(String lastName) {
        return find("lastName", lastName).list();
    }

    public static List<Person> findBornAfter(Date date) {
        return find("birth > :date", Parameters.with("date", date)).list();
    }
}
```

And this is now our REST endpoint:

```
@Path("/")
public class PersonEndpoint {

    @GET
    @Path("people")
    public List<Person> all() {
        return Person.findAll().list();
    }

    @GET
    @Path("people/by-name")
    public List<Person> findByName(@PathParam String name) {
        return Person.findByName(name);
    }

    @GET
    @Path("people/born-after")
    public List<Person> findBornAfter(@PathParam Date date) {
        return Person.findBornAfter(date);
    }

    @GET
    @Path("person/{id}")
    public Person findById(@PathParam Long id) {
        Person p = Person.findById(id);
        if(p == null)
            throw new WebApplicationException(Status.NOT_FOUND);
        return p;
    }

    @PUT
    @Path("person/{id}")
    public void updatePerson(@PathParam Long id, Person newPerson) {
        Person p = Person.findById(id);
        if(p == null)
            throw new WebApplicationException(Status.NOT_FOUND);
        p.birth = newPerson.birth;
        p.firstName = newPerson.firstName;
        p.lastName = newPerson.lastName;
    }

    @DELETE
    @Path("person/{id}")
```

```
    public void deletePerson(@PathParam Long id) {
        Person p = Person.findById(id);
        if(p == null)
            throw new WebApplicationException(Status.NOT_FOUND);
        p.delete();
    }

    @POST
    @Path("people")
    public Response newPerson(@Context UriInfo uriInfo, Person newPerson) {
        Person p = new Person();
        p.birth = newPerson.birth;
        p.firstName = newPerson.firstName;
        p.lastName = newPerson.lastName;
        Person.persist(p);

        URI uri = uriInfo.getAbsolutePathBuilder()
                .path(PersonEndpoint.class)
                .path(PersonEndpoint.class, "findById")
                .build(p.id);
        return Response.created(uri).build();
    }
}
```

How does this look?

This is the Hibernate ORM we've come to love with its solid core and tons of features, with just enough varnish to make it simpler to write your data layer. That's what we call "with Panache".

Create your first Quarkus application using Hibernate ORM with Panache

Using the online project generator

Go and start building your first Quarkus application today by picking the following extensions:

- Hibernate ORM with Panache
- JDBC Driver - PostgreSQL

- JSON-B

And create your project to get coding with Hibernate ORM with Panache.

By copying our quickstart

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The quickstart is located in the hibernate-orm-panache-quickstart directory.

Using our command-line tool

Alternately, you can generate a skeleton project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.0.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=hibernate-orm-panache-quickstart \
    -DclassName="org.acme.rest.json.PersonEndpoint" \
    -Dpath="/" \
    -Dextensions="resteasy-jsonb, hibernate-orm-panache, jdbc-postgresql"
cd hibernate-orm-panache-quickstart
```