

Quarkus - Simplified Hibernate ORM with Panache

Hibernate ORM is the de facto JPA implementation and offers you the full breadth of an Object Relational Mapper. It makes complex mappings possible, but it does not make simple and common mappings trivial. Hibernate ORM with Panache focuses on making your entities trivial and fun to write in Quarkus.

First: an example

What we're doing in Panache is allow you to write your Hibernate ORM entities like this:

```
@Entity
public class Person extends PanacheEntity {
    public String name;
    public LocalDate birth;
    public Status status;

    public static Person findByName(String name){
        return find("name", name).firstResult();
    }

    public static List<Person> findAlive(){
        return list("status", Status.Alive);
    }

    public static void deleteStef(){
        delete("name", "Stef");
    }
}
```

You have noticed how much more compact and readable the code is? Does this look interesting? Read on!



the `list()` method might be surprising at first. It takes fragments of HQL (JP-QL) queries and contextualizes the rest. That makes for very concise but yet readable code.

Setting up and configuring Hibernate ORM with Panache

To get started:

- add your settings in `application.properties`
- annotate your entities with `@Entity` and make them extend `PanacheEntity`
- place your entity logic in static methods in your entities

Follow the [Hibernate set-up guide for all configuration](#).

In your `pom.xml`, add the following dependencies:

- the Panache JPA extension
- your JDBC driver extension (`quarkus-jdbc-postgresql`, `quarkus-jdbc-h2`, `quarkus-jdbc-mariadb`, ...)

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

  <!-- JDBC driver dependencies -->
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-jdbc-postgresql</artifactId>
  </dependency>
</dependencies>
```

Then add the relevant configuration properties in `application.properties`.

```
# configure your datasource
quarkus.datasource.url =
jdbc:postgresql://localhost:5432/mydatabase
quarkus.datasource.driver = org.postgresql.Driver
quarkus.datasource.username = sarah
quarkus.datasource.password = connor

# drop and create the database at startup (use `update` to only
update the schema)
quarkus.hibernate-orm.database.generation = drop-and-create
```

Defining your entity

To define a Panache entity, simply extend `PanacheEntity`, annotate it with `@Entity` and add your columns as public fields:

```

@Entity
public class Person extends PanacheEntity {
    public String name;
    public LocalDate birth;
    public Status status;
}

```

You can put all your JPA column annotations on the public fields. If you need a field to not be persisted, use the `@Transient` annotation on it. If you need to write accessors, you can:

```

@Entity
public class Person extends PanacheEntity {
    public String name;
    public LocalDate birth;
    public Status status;

    // return name as uppercase in the model
    public String getName(){
        return name.toUpperCase();
    }

    // store all names in lowercase in the DB
    public void setName(String name){
        this.name = name.toLowerCase();
    }
}

```

And thanks to our field access rewrite, when your users read `person.name` they will actually call your `getName()` accessor, and similarly for field writes and the setter. This allows for proper encapsulation at runtime as all fields calls will be replaced by the corresponding getter/setter calls.

Most useful operations

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

```

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

// persist it
person.persist();

// 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(person.isPersistent()){
    // delete it
    person.delete();
}

// getting a list of all Person entities
List<Person> allPersons = Person.listAll();

// finding a specific person by ID
person = Person.findById(personId);

// finding a specific person by ID via an Optional
Optional<Person> optional = Person.findByIdOptional(personId);
person = optional.orElseThrow(() -> new NotFoundException());

// finding all living persons
List<Person> livingPersons = Person.list("status", Status.Alive);

// counting all persons
long countAll = Person.count();

// counting all living persons
long countAlive = Person.count("status", Status.Alive);

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

// delete all persons
Person.deleteAll();

// update all living persons
Person.update("name = 'Moral' where status = ?1", Status.Alive);

```

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

```
Stream<Person> persons = Person.streamAll();
List<String> namesButEmmanuels = persons
    .map(p -> p.name.toLowerCase() )
    .filter( n -> ! "emmanuel".equals(n) )
    .collect(Collectors.toList());
```



The `stream` methods require a transaction to work.

Paging

You should only use `list` and `stream` methods if your table contains small enough data sets. For larger data sets you can use the `find` method equivalents, which return a `PanacheQuery` on which you can do paging:

```
// create a query for all living persons
PanacheQuery<Person> livingPersons = Person.find("status", Status
    .Alive);

// make it use pages of 25 entries at a time
livingPersons.page(Page.ofSize(25));

// get the first page
List<Person> firstPage = livingPersons.list();

// get the second page
List<Person> secondPage = livingPersons.nextPage().list();

// get page 7
List<Person> page7 = livingPersons.page(Page.of(7, 25)).list();

// get the number of pages
int numberOfPages = livingPersons.pageCount();

// get the total number of entities returned by this query without
// paging
int count = livingPersons.count();

// and you can chain methods of course
return Person.find("status", Status.Alive)
    .page(Page.ofSize(25))
    .nextPage()
    .stream();
```

The `PanacheQuery` type has many other methods to deal with paging and returning streams.

Sorting

All methods accepting a query string also accept the following simplified query form:

```
List<Person> persons = Person.list("order by name,birth");
```

But these methods also accept an optional `Sort` parameter, which allows you to abstract your sorting:

```
List<Person> persons = Person.list(Sort.by("name").and("birth"));

// and with more restrictions
List<Person> persons = Person.list("status", Sort.by("name").and(
    "birth"), Status.Alive);
```

The `Sort` class has plenty of methods for adding columns and specifying sort direction.

Adding entity methods

In general, we recommend not adding custom queries for your entities outside of the entities themselves, to keep all model queries close to the models they operate on. So we recommend adding them as static methods in your entity class:

```
@Entity
public class Person extends PanacheEntity {
    public String name;
    public LocalDate birth;
    public Status status;

    public static Person findByName(String name){
        return find("name", name).firstResult();
    }

    public static List<Person> findAlive(){
        return list("status", Status.Alive);
    }

    public static void deleteStefs(){
        delete("name", "Stef");
    }
}
```

Simplified queries

Normally, HQL queries are of this form: `from EntityName [where ...] [order by ...]`, with optional elements at the end.

If your select query does not start with `from`, we support the following additional forms:

- `order by ...` which will expand to `from EntityName order by ...`
- `<singleColumnName>` (and single parameter) which will expand to `from EntityName where <singleColumnName> = ?`
- `<query>` will expand to `from EntityName where <query>`

If your update query does not start with `update`, we support the following additional forms:

- `from EntityName ...` which will expand to `update from EntityName ...`
- `set? <singleColumnName>` (and single parameter) which will expand to `update from EntityName set <singleColumnName> = ?`
- `set? <update-query>` will expand to `update from EntityName set <update-query>`



You can also write your queries in plain [HQL](#):

```
Order.find("select distinct o from Order o left join fetch  
o.lineItems");  
Order.update("update from Person set name = 'Moral' where status =  
?", Status.Alive);
```

Query parameters

You can pass query parameters by index (1-based) as shown below:

```
Person.find("name = ?1 and status = ?2", "stef", Status.Alive);
```

Or by name using a `Map`:

```
Map<String, Object> params = new HashMap<>();  
params.put("name", "stef");  
params.put("status", Status.Alive);  
Person.find("name = :name and status = :status", params);
```

Or using the convenience class `Parameters` either as is or to build a `Map`:

```

// generate a Map
Person.find("name = :name and status = :status",
           Parameters.with("name", "stef").and("status", Status.
Alive).map());

// use it as-is
Person.find("name = :name and status = :status",
           Parameters.with("name", "stef").and("status", Status.
Alive));

```

Every query operation accepts passing parameters by index (`Object...`), or by name (`Map<String, Object>` or `Parameters`).

The DAO/Repository option

Repository is a very popular pattern and can be very accurate for some use case, depending on the complexity of your needs.

Whether you want to use the Entity based approach presented above or a more traditional Repository approach, it is up to you, Panache and Quarkus have you covered either way.

If you lean towards using Repositories, you can get the exact same convenient methods injected in your Repository by making it implement `PanacheRepository`:

```

@ApplicationScoped
public class PersonRepository implements PanacheRepository<Person>
{
    // put your custom logic here as instance methods

    public Person findByName(String name){
        return find("name", name).firstResult();
    }

    public List<Person> findAlive(){
        return list("status", Status.Alive);
    }

    public void deleteStefs(){
        delete("name", "Stef");
    }
}

```

Absolutely all the operations that are defined on `PanacheEntityBase` are available on your DAO, so using it is exactly the same except you need to inject it:

```

@Inject
PersonRepository personRepository;

@GET
public long count(){
    return personRepository.count();
}

```

So if Repositories are your thing, you can keep doing them. Even with repositories, you can keep your entities as subclasses of `PanacheEntity` in order to get the ID and public fields working, but you can even skip that and go back to specifying your ID and using getters and setters if that's your thing. Use what works for you.

Transactions

Make sure to wrap methods modifying your database (e.g. `entity.persist()`) within a transaction. Marking a CDI bean method `@Transactional` will do that for you and make that method a transaction boundary. We recommend doing so at your application entry point boundaries like your REST endpoint controllers.

JPA batches changes you make to your entities and sends changes (it's called flush) at the end of the transaction or before a query. This is usually a good thing as it's more efficient. But if you want to check optimistic locking failures, do object validation right away or generally want to get immediate feedback, you can force the flush operation by calling `entity.flush()` or even use `entity.persistAndFlush()` to make it a single method call. This will allow you to catch any `PersistenceException` that could occur when JPA send those changes to the database. Remember, this is less efficient so don't abuse it. And your transaction still has to be committed.

Here is an example of the usage of the flush method to allow making a specific action in case of `PersistenceException`:

```

@Transactional
public void create(Parameter parameter){
    try {
        //Here I use the persistAndFlush() shorthand method on a
        Panache repository to persist to database then flush the changes.
        return parameterRepository.persistAndFlush(parameter);
    }
    catch(PersistenceException pe){
        LOG.error("Unable to create the parameter", pe);
        //in case of error, I save it to disk
        diskPersister.save(parameter);
    }
}

```

Lock management

Panache provides direct support for database locking with your entity/repository, using `findById(Object, LockModeType)` or `find().withLock(LockModeType)`.

The following examples are for the entity pattern but the same can be used with repositories.

First: Locking using findById().

```
public class PersonEndpoint {  
  
    @GET  
    @Transactional  
    public Person findByIdForUpdate(Long id){  
        Person p = Person.findById(id, LockModeType  
.PESSIMISTIC_WRITE);  
        //do something useful, the lock will be released when the  
transaction ends.  
        return person;  
    }  
  
}
```

Second: Locking in a find().

```
public class PersonEndpoint {  
  
    @GET  
    @Transactional  
    public Person findByNameForUpdate(String name){  
        Person p = Person.find("name", name).withLock(LockModeType  
.PESSIMISTIC_WRITE).findOne();  
        //do something useful, the lock will be released when the  
transaction ends.  
        return person;  
    }  
  
}
```

Be careful that locks are released when the transaction ends, so the method that invokes the lock query must be annotated with the `@Transactional` annotation.

Custom IDs

IDs are often a touchy subject, and not everyone's up for letting them handled by the framework, once again we have you covered.

You can specify your own ID strategy by extending `PanacheEntityBase` instead of `PanacheEntity`. Then you just declare whatever ID you want as a public field:

```
@Entity
public class Person extends PanacheEntityBase {

    @Id
    @SequenceGenerator(
        name = "personSequence",
        sequenceName = "person_id_seq",
        allocationSize = 1,
        initialValue = 4)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"personSequence")
    public Integer id;

    //...
}
```

If you're using repositories, then you will want to extend `PanacheRepositoryBase` instead of `PanacheRepository` and specify your ID type as an extra type parameter:

```
@ApplicationScoped
public class PersonRepository implements PanacheRepositoryBase
<Person,Integer> {
    //...
}
```

How and why we simplify Hibernate ORM mappings

When it comes to writing Hibernate ORM entities, there are a number of annoying things that users have grown used to reluctantly deal with, such as:

- Duplicating ID logic: most entities need an ID, most people don't care how it's set, because it's not really relevant to your model.
- Dumb getters and setters: since Java lacks support for properties in the language, we have to create fields, then generate getters and setters for those fields, even if they don't actually do anything more than read/write the fields.

- Traditional EE patterns advise to split entity definition (the model) from the operations you can do on them (DAOs, Repositories), but really that requires an unnatural split between the state and its operations even though we would never do something like that for regular objects in the Object Oriented architecture, where state and methods are in the same class. Moreover, this requires two classes per entity, and requires injection of the DAO or Repository where you need to do entity operations, which breaks your edit flow and requires you to get out of the code you're writing to set up an injection point before coming back to use it.
- Hibernate queries are super powerful, but overly verbose for common operations, requiring you to write queries even when you don't need all the parts.
- Hibernate is very general-purpose, but does not make it trivial to do trivial operations that make up 90% of our model usage.

With Panache, we took an opinionated approach to tackle all these problems:

- Make your entities extend `PanacheEntity`: it has an ID field that is auto-generated. If you require a custom ID strategy, you can extend `PanacheEntityBase` instead and handle the ID yourself.
- Use public fields. Get rid of dumb getter and setters. Under the hood, we will generate all getters and setters that are missing, and rewrite every access to these fields to use the accessor methods. This way you can still write *useful* accessors when you need them, which will be used even though your entity users still use field accesses.
- Don't use DAOs or Repositories: put all your entity logic in static methods in your entity class. Your entity superclass comes with lots of super useful static methods and you can add your own in your entity class. Users can just start using your entity `Person` by typing `Person.` and getting completion for all the operations in a single place.
- Don't write parts of the query that you don't need: write `Person.find("order by name")` or `Person.find("name = ?1 and status = ?2", "stef", Status.Alive)` or even better `Person.find("name", "stef")`.

That's all there is to it: with Panache, Hibernate ORM has never looked so trim and neat.

Defining entities in external projects or jars

Hibernate ORM with Panache relies on compile-time bytecode enhancements to your entities. If you define your entities in the same project where you build your Quarkus application, everything will work fine. If the entities come from external projects or jars, you can make sure that your jar is treated like a Quarkus application library by indexing it via Jandex, see [How to Generate a Jandex Index](#) in the CDI guide. This will allow Quarkus to index and enhance your entities as if they were inside the current project.