

# 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 deleteStefs(){
        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.



what was described above is essentially the [active record pattern](#), sometimes just called the entity pattern. Hibernate with Panache also allows for the use of the more classical [repository pattern](#) via `PanacheRepository`.

# Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

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

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

## Setting up and configuring Hibernate ORM with Panache

To get started:

- add your settings in `application.properties`
- annotate your entities with `@Entity`
- make your entities extend `PanacheEntity` (optional if you are using the repository pattern)

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.db-kind = postgresql
quarkus.datasource.username = sarah
quarkus.datasource.password = connor
quarkus.datasource.jdbc.url =
jdbc:postgresql://localhost:5432/mydatabase

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

## Solution 1: using the active record pattern

### 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 perform:

```

// 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();

// delete by id
boolean deleted = Person.deleteById(personId);

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

```

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

```
try (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.

As they perform I/O operations, they should be closed via the **close()** method or via a try-with-resource to close the underlying **ResultSet**. If not, you will see warnings from Agroal that will close the underlying **ResultSet** for you.

## Adding entity methods

Add custom queries on your entities inside the entities themselves. That way, you and your co-workers can find them easily, and queries are co-located with the object they operate on. Adding them as static methods in your entity class is the Panache Active Record way.

```
@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");
    }
}
```

## Solution 2: using the repository pattern

### Defining your entity

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

```

@Entity
public class Person {
    @Id @GeneratedValue private Long id;
    private String name;
    private LocalDate birth;
    private Status status;

    public Long getId(){
        return id;
    }
    public void setId(Long id){
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public LocalDate getBirth() {
        return birth;
    }
    public void setBirth(LocalDate birth) {
        this.birth = birth;
    }
    public Status getStatus() {
        return status;
    }
    public void setStatus(Status status) {
        this.status = status;
    }
}

```



If you don't want to bother defining getters/setters for your entities, you can make them extend **PanacheEntityBase** and Quarkus will generate them for you. You can even extend **PanacheEntity** and take advantage of the default ID it provides.

## 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 implements **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");
    }
}

```

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
PersonRepository personRepository;

@GET
public long count(){
    return 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
Person person = new 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
List<Person> allPersons = personRepository.listAll();

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

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

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

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

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

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

// delete all persons
personRepository.deleteAll();

// delete by id
boolean 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.

```
Stream<Person> persons = personRepository.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.



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.

## Advanced Query

### 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
long 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.

## Using a range instead of pages

**PanacheQuery** also allows range-based queries.

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

// make it use a range: start at index 0 until index 24
(inclusive).
livingPersons.range(0, 24);

// get the range
List<Person> firstRange = livingPersons.list();

// to get the next range, you need to call range again
List<Person> secondRange = livingPersons.range(25, 49).list();
```



You cannot mix ranges and pages: if you use a range, all methods that depend on having a current page will throw an `UnsupportedOperationException`; you can switch back to paging using `page(Page)` or `page(int, int)`.

## 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.

## 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 = 'Mortal' where status =  
?", Status.Alive);
```

## Named queries

You can reference a named query instead of a (simplified) HQL query by prefixing its name with the '#' character.

```
@Entity  
@NamedQuery(name = "Person.getByName", query = "from Person where  
name = :name")  
public class Person extends PanacheEntity {  
    public String name;  
    public LocalDate birth;  
    public Status status;  
  
    public static Person findByName(String name){  
        return find("#Person.getByName", name).firstResult();  
    }  
}
```



Named queries can only be defined inside your JPA entity classes (being the Panache entity class, or the repository parameterized type), or on one of its super classes.

## 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**).

## Query projection

Query projection can be done with the **project(Class)** method on the **PanacheQuery** object that is returned by the **find()** methods.

You can use it to restrict which fields will be returned by the database.

Hibernate will use **DTO projection** and generate a SELECT clause with the attributes from the projection class. This is also called **dynamic instantiation** or **constructor expression**, more info can be found on the Hibernate guide: [hql select clause](#)

The projection class needs to be a valid Java Bean and have a constructor that contains all its attributes, this constructor will be used to instantiate the projection DTO instead of using the entity class. This must be the only constructor of the class.

```
import io.quarkus.runtime.annotations.RegisterForReflection;

@RegisterForReflection ①
public class PersonName {
    public final String name; ②

    public PersonName(String name){ ③
        this.name = name;
    }
}

// only 'name' will be loaded from the database
PanacheQuery<PersonName> query = Person.find("status",
Status.Alive).project(PersonName.class);
```

1. If you plan to deploy your application as a native executable, you must register manually the projection class for reflection.
2. We use public fields here, but you can use private fields and getters/setters if you prefer.
3. This constructor will be used by Hibernate, it must be the only constructor in your class and have all the class attributes as parameters.



The implementation of the `project(Class)` method uses the constructor's parameter names to build the select clause of the query, so the compiler must be configured to store parameter names inside the compiled class. This is enabled by default if you are using the Quarkus Maven archetype. If you are not using it, add the property `<maven.compiler.parameters>true</maven.compiler.parameters>` to your pom.xml.

## 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 active record 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> {
    //...
}
```

## Mocking

### Using the active record pattern

If you are using the active record pattern you cannot use Mockito directly as it does not support mocking static methods, but you can use the `quarkus-panache-mock` module which allows you to use Mockito to mock all provided static methods, including your own.

Add this dependency to your `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-panache-mock</artifactId>
  <scope>test</scope>
</dependency>
```

Given this simple entity:

```
@Entity
public class Person extends PanacheEntity {

    public String name;

    public static List<Person> findOrdered() {
        return find("ORDER BY name").list();
    }
}
```

You can write your mocking test like this:

```
@QuarkusTest
public class PanacheFunctionalityTest {

    @Test
    public void testPanacheMocking() {
        PanacheMock.mock(Person.class);

        // Mocked classes always return a default value
    }
}
```

```

    Assertions.assertEquals(0, Person.count());

    // Now let's specify the return value
    Mockito.when(Person.count()).thenReturn(231);
    Assertions.assertEquals(23, Person.count());

    // Now let's change the return value
    Mockito.when(Person.count()).thenReturn(421);
    Assertions.assertEquals(42, Person.count());

    // Now let's call the original method
    Mockito.when(Person.count()).thenCallRealMethod();
    Assertions.assertEquals(0, Person.count());

    // Check that we called it 4 times
    PanacheMock.verify(Person.class,
Mockito.times(4)).count();①

    // Mock only with specific parameters
    Person p = new Person();
    Mockito.when(Person.findById(121)).thenReturn(p);
    Assertions.assertSame(p, Person.findById(121));
    Assertions.assertNull(Person.findById(421));

    // Mock throwing
    Mockito.when(Person.findById(121)).thenThrow(new
WebApplicationException());
    Assertions.assertThrows(WebApplicationException.class, ()
-> Person.findById(121));

    // We can even mock your custom methods

Mockito.when(Person.findOrdered()).thenReturn(Collections.emptyList
());
    Assertions.assertTrue(Person.findOrdered().isEmpty());

    // Mocking a void method
    Person.voidMethod();

    // Make it throw
    PanacheMock.doThrow(new
RuntimeException("Stef2")).when(Person.class).voidMethod();
    try {
        Person.voidMethod();
        Assertions.fail();
    } catch (RuntimeException x) {
        Assertions.assertEquals("Stef2", x.getMessage());
    }

    // Back to doNothing

```

```

        PanacheMock.doNothing().when(Person.class).voidMethod();
        Person.voidMethod();

        // Make it call the real method

        PanacheMock.doCallRealMethod().when(Person.class).voidMethod();
        try {
            Person.voidMethod();
            Assertions.fail();
        } catch (RuntimeException x) {
            Assertions.assertEquals("void", x.getMessage());
        }

        PanacheMock.verify(Person.class).findOrdered();
        PanacheMock.verify(Person.class,
Mockito.atLeast(4)).voidMethod();
        PanacheMock.verify(Person.class,
Mockito.atLeastOnce()).findById(Mockito.any());
        PanacheMock.verifyNoMoreInteractions(Person.class);
    }
}

```

- ① Be sure to call your **verify** and **do\*** methods on **PanacheMock** rather than **Mockito**, otherwise you won't know what mock object to pass.

## Using the repository pattern

If you are using the repository pattern you can use Mockito directly, using the **quarkus-junit5-mockito** module, which makes mocking beans much easier:

```

<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-junit5-mockito</artifactId>
  <scope>test</scope>
</dependency>

```

Given this simple entity:

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    public Long id;

    public String name;
}
```

And this repository:

```
@ApplicationScoped
public class PersonRepository implements PanacheRepository<Person>
{
    public List<Person> findOrdered() {
        return find("ORDER BY name").list();
    }
}
```

You can write your mocking test like this:

```
@QuarkusTest
public class PanacheFunctionalityTest {
    @InjectMock
    PersonRepository personRepository;

    @Test
    public void testPanacheRepositoryMocking() throws Throwable {
        // Mocked classes always return a default value
        Assertions.assertEquals(0, personRepository.count());

        // Now let's specify the return value
        Mockito.when(personRepository.count()).thenReturn(231);
        Assertions.assertEquals(23, personRepository.count());

        // Now let's change the return value
        Mockito.when(personRepository.count()).thenReturn(421);
        Assertions.assertEquals(42, personRepository.count());

        // Now let's call the original method

        Mockito.when(personRepository.count()).thenCallRealMethod();
        Assertions.assertEquals(0, personRepository.count());

        // Check that we called it 4 times
    }
}
```

```

Mockito.verify(personRepository, Mockito.times(4)).count();

// Mock only with specific parameters
Person p = new Person();
Mockito.when(personRepository.findById(121)).thenReturn(p);
Assertions.assertSame(p, personRepository.findById(121));
Assertions.assertNull(personRepository.findById(421));

// Mock throwing
Mockito.when(personRepository.findById(121)).thenThrow(new
WebApplicationException());
Assertions.assertThrows(WebApplicationException.class, ()
-> personRepository.findById(121));

Mockito.when(personRepository.findOrdered()).thenReturn(Collections
.emptyList());

Assertions.assertTrue(personRepository.findOrdered().isEmpty());

// We can even mock your custom methods
Mockito.verify(personRepository).findOrdered();
Mockito.verify(personRepository,
Mockito.atLeastOnce()).findById(Mockito.any());
Mockito.verifyNoMoreInteractions(personRepository);
    }
}

```

## 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.
- With the active record pattern: put all your entity logic in static methods in your entity class and don't create DAOs. 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.

It attempts to identify archives with Panache entities (and consumers of Panache entities) by the presence of the marker file `META-INF/panache-archive.marker`. Panache includes an annotation processor that will automatically create this file in archives that depend on Panache (even indirectly). If you have disabled annotation processors you may need to create this file manually in some cases.



If you include the `jpa-modelgen` annotation processor this will exclude the Panache annotation processor by default. If you do this you should either create the marker file yourself, or add the `quarkus-panache-common` as well, as shown below:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>${compiler-plugin.version}</version>
  <configuration>
    <annotationProcessorPaths>
      <annotationProcessorPath>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-jpamodelgen</artifactId>
        <version>${hibernate.version}</version>
      </annotationProcessorPath>
      <annotationProcessorPath>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-panache-common</artifactId>
        <version>${quarkus.platform.version}</version>
      </annotationProcessorPath>
    </annotationProcessorPaths>
  </configuration>
</plugin>
```