Quarkus - Simplified MongoDB with Panache

MongoDB is a well known NoSQL Database that is widely used, but using its raw API can be cumbersome as you need to express your entities and your queries as a MongoDB `Document`.

MongoDB with Panache provides active record style entities (and repositories) like you have in Hibernate ORM with Panache and focuses on making your entities trivial and fun to write in Quarkus.

It is built on top of the MongoDB Client extension.

> This technology is considered preview.
>
> In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.
>
> For a full list of possible extension statuses, check our FAQ entry.

First: an example

Panache allows you to write your MongoDB entities like this:

```java
public class Person extends PanacheMongoEntity {
    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 deleteLoics(){
        delete("name", "Loïc");
    }
}
```

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

> **ℹ** the `list()` method might be surprising at first. It takes fragments of PanacheQL queries (subset of JPQL) and contextualizes the rest. That makes for very concise but yet readable code. MongoDB native queries are also supported.

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 `mongodb-panache-quickstart` directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.2.0.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=mongodb-panache-quickstart \
    -DclassName="org.acme.mongodb.panache.FruitResource" \
    -Dpath="/fruits" \
    -Dextensions="resteasy-jsonb,mongodb-panache"
cd mongodb-panache-quickstart
```

This command generates a Maven structure importing the RESTEasy/JAX-RS, JSON-B and MongoDB with Panache extensions. After this, the `quarkus-mongodb-panache` extension has been added to your `pom.xml`.

If you don't want to generate a new project you can add the dependency in your `pom.xml`:

```xml
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-mongodb-panache</artifactId>
    </dependency>
</dependencies>
```

Setting up and configuring MongoDB with Panache

To get started:

- add your settings in `application.properties`
- Make your entities extend `PanacheMongoEntity`, you can use the `@MongoEntity` annotation to specify the name of the database and the name of the collection (it will default to the name of your entity).
- place your entity logic in static methods in your entities

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

```
# configure the MongoDB client for a replica set of two nodes
quarkus.mongodb.connection-string =
mongodb://mongo1:27017,mongo2:27017
# mandatory if you don't specify the name of the database using
@MongoEntity
quarkus.mongodb.database = person
```

The `quarkus.mongodb.database` property will be used by MongoDB with Panache to determine the name of the database where your entities will be persisted.

For advanced configuration of the MongoDB client, you can follow the Configuring the MongoDB database guide.

Defining your entity

To define a Panache entity, simply extend `PanacheMongoEntity` and add your columns as public fields. You can add the `@MongoEntity` annotation to your entity if you need to customize the name of the collection and/or the database.

```
@MongoEntity(collection="ThePerson")
public class Person extends PanacheMongoEntity {
    public String name;

    // will be persisted as a 'birth' field in MongoDB
    @BsonProperty("birth")
    public LocalDate birthDate;

    public Status status;
}
```

> **ℹ** annotating with `@MongoEntity` is optional, it allows you to configure the name of the collection and the name of the database. Here the entity will be stored in the `ThePerson` collection instead of the default `Person` collection.

MongoDB with Panache uses the PojoCodecProvider to map your entities to a MongoDB `Document`.

You will be allowed to use the following annotations to customize this mapping:

- @BsonId: allows you to customize the ID field, see Custom IDs.

- @BsonProperty: customize the serialized name of the field.

- @BsonIgnore: ignore a field during the serialization.

If you need to write accessors, you can:

```java
public class Person extends PanacheMongoEntity {
    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:

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

// persist it
person.persist();

person.status = Status.Dead;

// Your must call update() in order to send your entity
modifications to MongoDB
person.update();

// 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();
```

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());
```

> ℹ️ A `persistOrUpdate()` method exist that persist or update an entity in the database, it uses the *upsert* capability of MongoDB to do it in a single query.

Paging

You should only use `list` and `stream` methods if your collection 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 an optional Sort parameter, which allows you to abstract your sorting:

```java
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:

```java
public class Person extends PanacheMongoEntity {
    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 deleteLoics(){
        delete("name", "Loïc");
    }
}
```

Simplified queries

Normally, MongoDB queries are of this form: {'firstname': 'John', 'lastname':'Doe'}, this is what we call MongoDB native queries.

You can use them if you want, but we also support what we call **PanacheQL** that can be seen as a subset of JPQL (or HQL) and allows you to easily express a query. MongoDB with Panache will then map it to a MongoDB native query.

If your query does not start with {, we will consider it a PanacheQL query:

- `<singlePropertyName>` (and single parameter) which will expand to `{'singleColumnName': '?'}`

- `<query>` will expand to `{<query>}` where we will map the PanacheQL query to MongoDB native query form. We support the following operators that will be mapped to the corresponding MongoDB operators: 'and', 'or' (mixing 'and' and 'or' is not currently supported), '=', '>', '>=', '<', '⇐', '!=', 'is null', 'is not null', and 'like' that is mapped to the MongoDB `$regex` operator.

Here are some query examples:

- `firstname = ?1 and status = ?2` will be mapped to `{'firstname': ?1, 'status': ?2}`

- `amount > ?1 and firstname != ?2` will be mapped to `{'amount': {'$gt': ?1}, 'firstname': {'$ne': ?2}}`

- `lastname like ?1` will be mapped to `{'lastname': {'$regex': ?1}}`. Be careful that this will be MongoDB regex support and not SQL like pattern.

- `lastname is not null` will be mapped to `{'lastname':{'$exists': true}}`

We also handle some basic date type transformations: all fields of type `Date`, `LocalDate` or `LocalDateTime` will be mapped to the BSON Date using the `ISODate` type (UTC datetime).

MongoDB with Panache also supports extended MongoDB queries by providing a `Document` query, this is supported by the find/list/stream/count/delete methods.

Query parameters

You can pass query parameters, for both native and PanacheQL queries, by index (1-based) as shown below:

```
Person.find("name = ?1 and status = ?2", "Loïc", Status.Alive);
Person.find("{'name': ?1, 'status': ?2}", "Loïc", Status.Alive);
```

Or by name using a `Map`:

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

// use it as-is
Person.find("{'name': :name, 'status': :status}",
        Parameters.with("name", "Loïc").and("status", Status.
Alive));
```

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

When you use query parameters, be careful that PanacheQL queries will refer to the Object parameters name but native queries will refer to MongoDB field names.

Imagine the following entity:

```
public class Person extends PanacheMongoEntity {
    @BsonProperty("lastname")
    public String name;
    public LocalDate birth;
    public Status status;

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

    public static Person findByNameWithNativeQuery(String name){
        return find("{'lastname': ?1}", name).firstResult();
    }
}
```

Both `findByNameWithPanacheQLQuery()` and `findByNameWithNativeQuery()` methods will return the same result but query written in PanacheQL will use the entity field name: name, and native query will use the MongoDB field name: lastname.

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, the ID field will always be returned but it's not mandatory to include it inside the projection class.

For this, you need to create a class (a Pojo) that will only contain the projected fields. This pojo needs to be annotated with `@ProjectionFor(Entity.class)` where `Entity` is the name of your entity

class. The field names, or getters, of the projection class will be used to restrict which properties will be loaded from the database.

Projection can be done for both PanacheQL and native queries.

```java
import io.quarkus.mongodb.panache.ProjectionFor;
import org.bson.codecs.pojo.annotations.BsonProperty;

// using public fields
@ProjectionFor(Person.class)
public class PersonName {
    public String name;
}

// using getters
@ProjectionFor(Person.class)
public class PersonNameWithGetter {
    private String name;

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }
}

// only 'name' will be loaded from the database
PanacheQuery<PersonName> shortQuery = Person.find("status ",
Status.Alive).project(PersonName.class);
PanacheQuery<PersonName> query = Person.find("'status': ?1",
Status.Alive).project(PersonNameWithGetter.class);
PanacheQuery<PersonName> nativeQuery = Person.find("{'status':
'ALIVE'}", Status.Alive).project(PersonName.class);
```

Using @BsonProperty is not needed to define custom column mappings, as the mappings from the entity class will be used.

You can have your projection class extends from another class. In this case, the parent class also needs to have use @ProjectionFor annotation.

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 PanacheMongoRepository:

```java
@ApplicationScoped
public class PersonRepository implements PanacheMongoRepository
<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 deleteLoics(){
        delete("name", "Loïc");
    }
}
```

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

```java
@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 PanacheMongoEntity 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

⚠️ MongoDB offers ACID transactions since version 4.0. MongoDB with Panache doesn't provide support for them.

Custom IDs

IDs are often a touchy subject. In MongoDB, they are usually auto-generated by the database with an `ObjectId` type. In MongoDB with Panache the ID are defined by a field named `id` of the `org.bson.types.ObjectId` type, but if you want ot customize them, once again we have you covered.

You can specify your own ID strategy by extending `PanacheMongoEntityBase` instead of `PanacheMongoEntity`. Then you just declare whatever ID you want as a public field by annotating it by `@BsonId`:

```java
@MongoEntity
public class Person extends PanacheMongoEntityBase {

    @BsonId
    public Integer myId;

    //...
}
```

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

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

> ℹ️ When using `ObjectId`, MongoDB will automatically provide a value for you, but if you use a custom field type, you need to provide the value by yourself.

`ObjectId` can be difficult to use if you want to expose its value in your REST service. So we created JSON-B and Jackson providers to serialize/deserialize them as a `String` which are are automatically registered if your project depends on one of the RESTEasy with JSON-B or RESTEasy with Jackson extensions.

How and why we simplify MongoDB API

When it comes to writing MongoDB 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.

- MongoDB queries are super powerful, but overly verbose for common operations, requiring you to write queries even when you don't need all the parts.

- MongoDB queries are JSON based, so you will need some String manipulation or using the `Document` type and it will need a lot of boilerplate code.

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

- Make your entities extend `PanacheMongoEntity`: it has an ID field that is auto-generated. If you require a custom ID strategy, you can extend `PanacheMongoEntityBase` 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", "Loïc", Status.Alive)` or even better `Person.find("name", "Loïc")`.

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

Defining entities in external projects or jars

MongoDB 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.