

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.

First: an example

Panache allows you to write your MongoDB entities like this:

```
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!



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.



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

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:1.8.1.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=mongodb-panache-quickstart \
  -DclassName="org.acme.mongodb.panache.PersonResource" \
  -Dpath="/persons" \
  -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, add the dependency in your `pom.xml`:

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

Setting up and configuring MongoDB with Panache

To get started:

- add your settings in `application.properties`
- Make your entities extend `PanacheMongoEntity` (optional if you are using the repository pattern)
- Optionally, use the `@MongoEntity` annotation to specify the name of the collection, the name of the database or the name of the client.

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 (if not overridden by `@MongoEntity`).

The `@MongoEntity` annotation allows configuring:

- the name of the client for multi-tenant application, see [Multiple MongoDB Clients](#). Otherwise, the default client will be used.
- the name of the database, otherwise, the `quarkus.mongodb.database` property will be used.
- the name of the collection, otherwise the simple name of the class will be used.

For advanced configuration of the MongoDB client, you can follow the [Configuring the MongoDB database guide](#).

Solution 1: using the active record pattern

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, the database, or the client.

```
@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;
}
```



Annotating with `@MongoEntity` is optional. 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:

```
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 perform:

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

// persist it: if you keep the default ObjectId ID field, it will
// be populated by the MongoDB driver
person.persist();

person.status = Status.Dead;

// You 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
// here we build a new ObjectId but you can also retrieve it from
the existing entity after being persisted
ObjectId personId = new ObjectId(idAsString);
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'
long updated = Person.update("name", "Mortal").where("status",
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());

```



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.

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.

```
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");
    }
}
```

Solution 2: using the repository pattern

Defining your entity

You can define your entity as regular POJO. You can add the `@MongoEntity` annotation to your entity if you need to customize the name of the collection, the database, or the client.

```
@MongoEntity(collection="ThePerson")
public class Person {
    public ObjectId id; // used by MongoDB for the _id field
    public String name;
    public LocalDate birth;
    public Status status;
}
```



Annotating with `@MongoEntity` is optional. 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.



You can use public fields or private fields with getters/setters. If you don't want to manage the ID by yourself, you can make your entity extends [PanacheMongoEntity](#).

Defining your repository

When using Repositories, you can get the exact same convenient methods as with the active record pattern, injected in your Repository, by making them implements [PanacheMongoRepository](#):

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

All the operations that are defined on [PanacheMongoEntityBase](#) 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 = "Loïc";
person.birth = LocalDate.of(1910, Month.FEBRUARY, 1);
person.status = Status.Alive;

// persist it: if you keep the default ObjectId ID field, it will
// be populated by the MongoDB driver
personRepository.persist(person);

person.status = Status.Dead;

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

// delete it
personRepository.delete(person);

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

// finding a specific person by ID
// here we build a new ObjectId but you can also retrieve it from
// the existing entity after being persisted
ObjectId personId = new ObjectId(idAsString);
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'
long updated = personRepository.update("name",
"Mortal").where("status", Status.Alive);

```

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

```

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

```



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.



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 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.

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 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, 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': '?1'}`
- `<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 (both String and JavaScript patterns are supported).

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}}`
- `status in ?1` will be mapped to `{'status':{'$in': [?1]}}`

We also handle some basic date type transformations: all fields of type `Date`, `LocalDate`, `LocalDateTime` or `Instant` will be mapped to the [BSON Date](#) using the `ISODate` type (UTC datetime). The MongoDB POJO codec doesn't support `ZonedDateTime` and `OffsetDateTime` so you should convert them prior usage.

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

MongoDB with Panache offers operations to update multiple documents based on an update document and a query : `Person.update("foo = ?1, bar = ?2", fooName, barName).where("name = ?1", name)`.

For these operations, you can express the update document the same way you express your queries, here are some examples:

- `<singlePropertyName>` (and single parameter) which will expand to the update document `{'$set' : {'singleColumnName': '?1'}}`
- `firstname = ?1, status = ?2` will be mapped to the update document `{'$set' : {'firstname': ?1, 'status': ?2}}`
- `firstname = :firstname, status = :status` will be mapped to the update document `{'$set' : {'firstname': :firstname, 'status': :status}}`
- `{'firstname' : ?1, 'status' : ?2}` will be mapped to the update document `{'$set' : {'firstname': ?1, 'status': ?2}}`
- `{'firstname' : firstname, 'status' : :status}` will be mapped to the update document `{'$set' : {'firstname': :firstname, 'status': :status}}`

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.

```
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.

Query debugging

As MongoDB with Panache allows writing simplified queries, it is sometimes handy to log the generated native queries for debugging purpose.

This can be achieved by setting to **DEBUG** the following log category inside your **application.properties**:

```
quarkus.log.category."io.quarkus.mongodb.panache.runtime".level=DEBUG
```

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 to 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`:

```
@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:

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



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 automatically registered if your project depends on either the RESTEasy JSON-B extension or the RESTEasy

Jackson extension.

If you use the standard `ObjectId` ID type, don't forget to retrieve your entity by creating a new `ObjectId` when the identifier comes from a path parameter. For example:



```
@GET
@Path("/{id}")
public Person findById(@PathParam("id") String id) {
    return Person.findById(new ObjectId(id));
}
```

Working with Kotlin Data classes

Kotlin data classes are a very convenient way of defining data carrier classes, making them a great match to define an entity class.

But this type of class comes with some limitations: all fields need to be initialized at construction time or be marked as nullable, and the generated constructor needs to have as parameters all the fields of the data class.

MongoDB with Panache uses the `PojoCodec`, a MongoDB codec which mandates the presence of a parameterless constructor.

Therefore, if you want to use a data class as an entity class, you need a way to make Kotlin generate an empty constructor. To do so, you need to provide default values for all the fields of your classes. The following sentence from the Kotlin documentation explains it:

On the JVM, if the generated class needs to have a parameterless constructor, default values for all properties have to be specified (see Constructors).

If for whatever reason, the aforementioned solution is deemed unacceptable, there are alternatives.

First, you can create a BSON Codec which will be automatically registered by Quarkus and will be used instead of the `PojoCodec`. See this part of the documentation: [Using BSON codec](#).

Another option is to use the `@BsonCreator` annotation to tell the `PojoCodec` to use the Kotlin data class default constructor, in this case all constructor parameters have to be annotated with `@BsonProperty`: see [Supporting pojos without no args constructor](#).

This will only work when the entity extends `PanacheMongoEntityBase` and not `PanacheMongoEntity`, as the ID field also needs to be included in the constructor.

An example of a `Person` class defined as a Kotlin data class would look like:

```
data class Person @BsonCreator constructor (
    @BsonId var id: ObjectId,
    @BsonProperty("name") var name: String,
    @BsonProperty("birth") var birth: LocalDate,
    @BsonProperty("status") var status: Status
): PanacheMongoEntityBase()
```



Here we use `var` but note that `val` can also be used.

The `@BsonId` annotation is used instead of `@BsonProperty("_id")` for brevity's sake, but use of either is valid.

The last option is to use the `no-arg` compiler plugin. This plugin is configured with a list of annotations, and the end result is the generation of no-args constructor for each class annotated with them.

For MongoDB with Panache, you could use the `@MongoEntity` annotation on your data class for this:

```
@MongoEntity
data class Person (
    var name: String,
    var birth: LocalDate,
    var status: Status
): PanacheMongoEntity()
```

Reactive Entities and Repositories

MongoDB with Panache allows using reactive style implementation for both entities and repositories. For this, you need to use the Reactive variants when defining your entities : `ReactivePanacheMongoEntity` or `ReactivePanacheMongoEntityBase`, and when defining your repositories: `ReactivePanacheMongoRepository` or `ReactivePanacheMongoRepositoryBase`.



Mutiny

The reactive API of the MongoDB with Panache uses Mutiny reactive types, if you're not familiar with them, read the [Getting Started with Reactive guide](#) first.

The reactive variant of the `Person` class will be:

```

public class ReactivePerson extends ReactivePanacheMongoEntity {
    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();
    }
}

```

You will have access to the same functionalities of the *imperative* variant inside the reactive one: bson annotations, custom ID, PanacheQL, ... But the methods on your entities or repositories will all return reactive types.

See the equivalent methods from the imperative example with the reactive variant:

```

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

// persist it: if you keep the default ObjectId ID field, it will
// be populated by the MongoDB driver,
// and accessible when uni1 will be resolved
Uni<Void> uni1 = person.persist();

person.status = Status.Dead;

// You must call update() in order to send your entity
// modifications to MongoDB
Uni<Void> uni2 = person.update();

// delete it
Uni<Void> uni3 = person.delete();

// getting a list of all persons
Uni<List<ReactivePerson>> allPersons = ReactivePerson.listAll();

// finding a specific person by ID
// here we build a new ObjectId but you can also retrieve it from
// the existing entity after being persisted

```

```

ObjectId personId = new ObjectId(idAsString);
Uni<ReactivePerson> personById = ReactivePerson.findById(personId);

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

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

// counting all persons
Uni<Long> countAll = ReactivePerson.count();

// counting all living persons
Uni<Long> countAlive = ReactivePerson.count("status",
Status.Alive);

// delete all living persons
Uni<Long> deleteCount = ReactivePerson.delete("status",
Status.Alive);

// delete all persons
deleteCount = ReactivePerson.deleteAll();

// delete by id
Uni<Boolean> deleted = ReactivePerson.deleteById(personId);

// set the name of all living persons to 'Mortal'
Uni<Long> updated = ReactivePerson.update("name",
"Mortal").where("status", Status.Alive);

```



If you use MongoDB with Panache in conjunction with RESTEasy, you can directly return a reactive type inside your JAX-RS resource endpoint as long as you include the `quarkus-resteasy-mutiny` extension.

The same query facility exists for the reactive types, but the `stream()` methods act differently: they return a `Multi` (which implement a reactive stream `Publisher`) instead of a `Stream`.

It allows more advanced reactive use cases, for example, you can use it to send server-sent events (SSE) via RESTEasy:

```
import org.jboss.resteasy.annotations.SseElementType;
import org.reactivestreams.Publisher;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@GET
@Path("/stream")
@Produces(MediaType.SERVER_SENT_EVENTS)
@SseElementType(MediaType.APPLICATION_JSON)
public Multi<ReactivePerson> streamPersons() {
    return ReactivePerson.streamAll();
}
```



`@SseElementType(MediaType.APPLICATION_JSON)` tells RESTEasy to serialize the object in JSON.

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:

```
public class Person extends PanacheMongoEntity {

    public String name;

    public static List<Person> findOrdered() {
        return findAll(Sort.by("lastname", "firstname")).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();❶

        // 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());

        PanacheMock.verify(Person.class).findOrdered();
        PanacheMock.verify(Person.class,
Mockito.atLeastOnce()).findById(Mockito.any());
```

```
        PanacheMock.verifyNoMoreInteractions(Person.class);
    }
}
```

- ① Be sure to call your `verify` 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:

```
public class Person {

    @BsonId
    public Long id;

    public String name;
}
```

And this repository:

```
@ApplicationScoped
public class PersonRepository implements
PanacheMongoRepository<Person> {
    public List<Person> findOrdered() {
        return findAll(Sort.by("lastname", "firstname")).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 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.
- 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", "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.

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.