Quarkus - Using the MongoDB Client

MongoDB is a well known NoSQL Database that is widely used.

In this guide, we see how you can get your REST services to use the MongoDB database.

> 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.

Prerequisites

To complete this guide, you need:

- less than 15 minutes

- an IDE

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately

- Apache Maven 3.5.3+

- MongoDB installed or Docker installed

Architecture

The application built in this guide is quite simple: the user can add elements in a list using a form and the list is updated.

All the information between the browser and the server is formatted as JSON.

The elements are stored in MongoDB.

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-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:1.3.0.Alpha1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=mongodb-quickstart \
    -DclassName="org.acme.rest.json.FruitResource" \
    -Dpath="/fruits" \
    -Dextensions="resteasy-jsonb,mongodb-client"
cd mongodb-quickstart
```

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

Creating your first JSON REST service

In this example, we will create an application to manage a list of fruits.

First, let's create the `Fruit` bean as follows:

```java
package org.acme.rest.json;

import java.util.Objects;

public class Fruit {

    private String name;
    private String description;

    public Fruit() {
    }

    public Fruit(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public String getName() {
        return name;
    }

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

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Fruit)) {
            return false;
        }

        Fruit other = (Fruit) obj;

        return Objects.equals(other.name, this.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.name);
    }
}
```

Nothing fancy. One important thing to note is that having a default constructor is required by the JSON serialization layer.

Now create a `org.acme.rest.json.FruitService` that will be the business layer of our application and store/load the fruits from the mongoDB database.

```java
package org.acme.rest.json;

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import org.bson.Document;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import java.util.ArrayList;
import java.util.List;

@ApplicationScoped
public class FruitService {

    @Inject MongoClient mongoClient;

    public List<Fruit> list(){
        List<Fruit> list = new ArrayList<>();
        MongoCursor<Document> cursor = getCollection().find()
.iterator();

        try {
            while (cursor.hasNext()) {
                Document document = cursor.next();
                Fruit fruit = new Fruit();
                fruit.setName(document.getString("name"));
                fruit.setDescription(document.getString(
"description"));
                list.add(fruit);
            }
        } finally {
            cursor.close();
        }
        return list;
    }

    public void add(Fruit fruit){
        Document document = new Document()
                .append("name", fruit.getName())
                .append("description", fruit.getDescription());
        getCollection().insertOne(document);
    }

    private MongoCollection getCollection(){
        return mongoClient.getDatabase("fruit").getCollection(
"fruit");
    }
}
```

Now, edit the `org.acme.rest.json.FruitResource` class as follows:

```java
@Path("/fruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class FruitResource {

    @Inject FruitService fruitService;

    @GET
    public List<Fruit> list() {
        return fruitService.list();
    }

    @POST
    public List<Fruit> add(Fruit fruit) {
        fruitService.add(fruit);
        return list();
    }
}
```

The implementation is pretty straightforward and you just need to define your endpoints using the JAX-RS annotations and use the `FruitService` to list/add new fruits.

Configuring the MongoDB database

The main property to configure is the URL to access to MongoDB, almost all configuration can be included in the connection URI so we advise you to do so, you can find more information in the MongoDB documentation: https://docs.mongodb.com/manual/reference/connection-string/

A sample configuration should look like this:

```
# configure the mongoDB client for a replica set of two nodes
quarkus.mongodb.connection-string =
mongodb://mongo1:27017,mongo2:27017
```

In this example, we are using a single instance running on localhost:

```
# configure the mongoDB client for a single instance on localhost
quarkus.mongodb.connection-string = mongodb://localhost:27017
```

If you need more configuration properties, there is a full list at the end of this guide.

Running a MongoDB Database

As by default, `MongoClient` is configured to access a local MongoDB database on port 27017 (the default MongoDB port), if you have a local running database on this port, there is nothing more to do before being able to test it!

If you want to use Docker to run a MongoDB database, you can use the following command to launch one:

```
docker run -ti --rm -p 27017:27017 mongo:4.0
```

Creating a frontend

Now let's add a simple web page to interact with our `FruitResource`. Quarkus automatically serves static resources located under the `META-INF/resources` directory. In the `src/main/resources/META-INF/resources` directory, add a `fruits.html` file with the content from this fruits.html file in it.

You can now interact with your REST service:

- start Quarkus with `./mvnw compile quarkus:dev`
- open a browser to `http://localhost:8080/fruits.html`
- add new fruits to the list via the form

Reactive MongoDB Client

A reactive MongoDB Client is included in Quarkus. Using it is as easy as using the classic MongoDB Client. You can rewrite the previous example to use it like the following.

```java
package org.acme.rest.json;

import io.quarkus.mongodb.ReactiveMongoClient;
import io.quarkus.mongodb.ReactiveMongoCollection;
import org.bson.Document;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import java.util.List;
import java.util.concurrent.CompletionStage;

@ApplicationScoped
public class ReactiveFruitService {

    @Inject
    ReactiveMongoClient mongoClient;

    public CompletionStage<List<Fruit>> list(){
        return getCollection().find().map(doc -> {
            Fruit fruit = new Fruit();
            fruit.setName(doc.getString("name"));
            fruit.setDescription(doc.getString("description"));
            return fruit;
        }).toList().run();
    }

    public CompletionStage<Void> add(Fruit fruit){
        Document document = new Document()
                .append("name", fruit.getName())
                .append("description", fruit.getDescription());
        return getCollection().insertOne(document);
    }

    private ReactiveMongoCollection<Document> getCollection(){
        return mongoClient.getDatabase("fruit").getCollection(
"fruit");
    }
}
```

```java
package org.acme.rest.json;

import javax.inject.Inject;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import java.util.List;
import java.util.concurrent.CompletionStage;

@Path("/reactive_fruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class ReactiveFruitResource {

    @Inject ReactiveFruitService fruitService;


    @GET
    public CompletionStage<List<Fruit>> list() {
        return fruitService.list();
    }

    @POST
    public CompletionStage<List<Fruit>>  add(Fruit fruit) {
        fruitService.add(fruit);
        return list();
    }
}
```

Simplifying MongoDB Client usage using BSON codec

By using a Bson Codec, the MongoDB Client will take care of the transformation of your domain object to/from a MongoDB Document automatically.

First you need to create a Bson Codec that will tell Bson how to transform your entity to/from a MongoDB Document. Here we use a CollectibleCodec as our object is retrievable from the database (it has a MongoDB identifier), if not we would have used a Codec instead. More information in the codec documentation: https://mongodb.github.io/mongo-java-driver/3.10/bson/codecs.

```java
package org.acme.rest.json.codec;

import com.mongodb.MongoClientSettings;
import org.acme.rest.json.Fruit;
import org.bson.*;
import org.bson.codecs.Codec;
```

```java
import org.bson.codecs.CollectibleCodec;
import org.bson.codecs.DecoderContext;
import org.bson.codecs.EncoderContext;

import java.util.UUID;

public class FruitCodec implements CollectibleCodec<Fruit> {

    private final Codec<Document> documentCodec;

    public FruitCodec() {
        this.documentCodec = MongoClientSettings
.getDefaultCodecRegistry().get(Document.class);
    }

    @Override
    public void encode(BsonWriter writer, Fruit fruit,
EncoderContext encoderContext) {
        Document doc = new Document();
        doc.put("name", fruit.getName());
        doc.put("description", fruit.getDescription());
        documentCodec.encode(writer, doc, encoderContext);
    }

    @Override
    public Class<Fruit> getEncoderClass() {
        return Fruit.class;
    }

    @Override
    public Fruit generateIdIfAbsentFromDocument(Fruit document) {
        if (!documentHasId(document)) {
             document.setId(UUID.randomUUID().toString());
        }
        return document;
    }

    @Override
    public boolean documentHasId(Fruit document) {
        return document.getId() != null;
    }

    @Override
    public BsonValue getDocumentId(Fruit document) {
        return new BsonString(document.getId());
    }

    @Override
    public Fruit decode(BsonReader reader, DecoderContext
decoderContext) {
```

```
        Document document = documentCodec.decode(reader,
decoderContext);
        Fruit fruit = new Fruit();
        if (document.getString("id") != null) {
            fruit.setId(document.getString("id"));
        }
        fruit.setName(document.getString("name"));
        fruit.setDescription(document.getString("description"));
        return fruit;
    }
}
```

Then you need to create a `CodecProvider` to link this `Codec` to the `Fruit` class.

```
package org.acme.rest.json.codec;

import org.acme.rest.json.Fruit;
import org.bson.codecs.Codec;
import org.bson.codecs.configuration.CodecProvider;
import org.bson.codecs.configuration.CodecRegistry;

public class FruitCodecProvider implements CodecProvider {
    @Override
    public <T> Codec<T> get(Class<T> clazz, CodecRegistry registry)
{
        if (clazz == Fruit.class) {
            return (Codec<T>) new FruitCodec();
        }
        return null;
    }

}
```

Quarkus takes care of registering the `CodecProvider` for you.

Finally, when getting the `MongoCollection` from the database you can use directly the `Fruit` class instead of the `Document` one, the codec will automatically map the `Document` to/from your `Fruit` class.

Here is an example of using a `MongoCollection` with the `FruitCodec`.

```java
package org.acme.rest.json;

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import java.util.ArrayList;
import java.util.List;

@ApplicationScoped
public class CodecFruitService {

    @Inject MongoClient mongoClient;

    public List<Fruit> list(){
        List<Fruit> list = new ArrayList<>();
        MongoCursor<Fruit> cursor = getCollection().find().
iterator();

        try {
            while (cursor.hasNext()) {
                list.add(cursor.next());
            }
        } finally {
            cursor.close();
        }
        return list;
    }

    public void add(Fruit fruit){
        getCollection().insertOne(fruit);
    }

    private MongoCollection<Fruit> getCollection(){
        return mongoClient.getDatabase("fruit").getCollection(
"fruit", Fruit.class);
    }
}
```

Simplifying MongoDB with Panache

The MongoDB with Panache extension facilitates the usage of MongoDB by providing 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.

Connection Health Check

If you are using the `quarkus-smallrye-health` extension, `quarkus-mongodb` will automatically add a readiness health check to validate the connection to the cluster.

So when you access the `/health/ready` endpoint of your application you will have information about the connection validation status.

This behavior can be disabled by setting the `quarkus.mongodb.health.enabled` property to `false` in your `application.properties`.

The legacy client

We don't include the legacy MongoDB client by default. It contains the now retired MongoDB Java API (DB, DBCollection,...) and the `com.mongodb.MongoClient` that is now superseded by `com.mongodb.client.MongoClient`.

If you want to use the legacy API, you need to add the following dependency to your `pom.xml`:

```xml
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-legacy</artifactId>
</dependency>
```

Building a native executable

You can use the MongoDB client in a native executable.

If you want to use SSL/TLS encryption, you need to add these properties in your `application.properties`:

```
quarkus.mongodb.tls=true
quarkus.mongodb.tls-insecure=true # only if TLS certificate cannot
be validated
```

You can then build a native executable with the usual command `./mvnw package -Pnative`.

Running it is as simple as executing `./target/mongodb-quickstart-1.0-SNAPSHOT-runner`.

You can then point your browser to `http://localhost:8080/fruits.html` and use your application.

> ⚠️ Currently, Quarkus doesn't support the `mongodb+srv` protocol in native mode.
>
> Client-Side Field Level Encryption is also not supported in native mode.

Conclusion

Accessing a MongoDB database from a MongoDB Client is easy with Quarkus as it provides configuration and native support for it.

Configuration Reference

⊟ Configuration property fixed at build time - ⚙ Configuration property overridable at runtime

Configuration property	Type	Default
⊟ `quarkus.mongodb.health.enabled` Whether or not an health check is published in case the smallrye-health extension is present.	boolean	`true`

⚙ `quarkus.mongodb.connection-string` Configures the connection string. The format is: `mongodb://[username:password@]host1[:port1][,host2[:port2],…[,hostN[:portN]]][/[database.collection][?options]]` `mongodb://` is a required prefix to identify that this is a string in the standard connection format. `username:password@` are optional. If given, the driver will attempt to login to a database after connecting to a database server. For some authentication mechanisms, only the username is specified and the password is not, in which case the ":" after the username is left off as well. `host1` is the only required part of the connection string. It identifies a server address to connect to. `:portX` is optional and defaults to :27017 if not provided. `/database` is the name of the database to login to and thus is only relevant if the `username:password@` syntax is used. If not specified the `admin` database will be used by default. `?options` are connection options. Note that if `database` is absent there is still a `/` required between the last host and the `?` introducing the options. Options are name=value pairs and the pairs are separated by "&". An alternative format, using the `mongodb+srv` protocol, is: `mongodb+srv://[username:password@]host[/[database][?options]]` - `mongodb+srv://` is a required prefix for this format. - `username:password@` are optional. If given, the driver will attempt to login to a database after connecting to a database server. For some authentication mechanisms, only the username is specified and the password is not, in which case the ":" after the username is left off as well - `host` is the only required part of the URI. It identifies a single host name for which SRV records are looked up from a Domain Name Server after prefixing the host name with `"_mongodb._tcp"`. The host/port for each SRV record becomes the seed list used to connect, as if each one were provided as host/port pair in a URI using the normal mongodb protocol. - `/database` is the name of the database to login to and thus is only relevant if the `username:password@` syntax is used. If not specified the "admin" database will be used by default. - `?options` are connection options. Note that if `database` is absent there is still a `/` required between the last host and the `?` introducing the options. Options are name=value pairs and the pairs are separated by "&". Additionally with the mongodb+srv protocol, TXT records are looked up from a Domain Name Server for the given host, and the text value of each one is prepended to any options on the URI itself. Because the last specified value for any option wins, that means that options provided on the URI will override any that are provided via TXT records.	string	
⚙ `quarkus.mongodb.hosts` Configures the MongoDB server addressed (one if single mode). The addresses are passed as `host:port`.	list of string	`127.0.0.1:27017`
⚙ `quarkus.mongodb.database` Configure the database name.	string	

15

⚙️ `quarkus.mongodb.application-name` Configures the application name.	string	
⚙️ `quarkus.mongodb.max-pool-size` Configures the maximum number of connections in the connection pool.	int	
⚙️ `quarkus.mongodb.min-pool-size` Configures the minimum number of connections in the connection pool.	int	
⚙️ `quarkus.mongodb.max-connection-idle-time` Maximum idle time of a pooled connection. A connection that exceeds this limit will be closed.	Duration ❓	
⚙️ `quarkus.mongodb.max-connection-life-time` Maximum life time of a pooled connection. A connection that exceeds this limit will be closed.	Duration ❓	
⚙️ `quarkus.mongodb.wait-queue-timeout` The maximum wait time that a thread may wait for a connection to become available.	Duration ❓	
⚙️ `quarkus.mongodb.maintenance-frequency` Configures the time period between runs of the maintenance job.	Duration ❓	
⚙️ `quarkus.mongodb.maintenance-initial-delay` Configures period of time to wait before running the first maintenance job on the connection pool.	Duration ❓	
⚙️ `quarkus.mongodb.wait-queue-multiple` This multiplier, multiplied with the `maxPoolSize` setting, gives the maximum number of threads that may be waiting for a connection to become available from the pool. All further threads will get an exception right away.	int	
⚙️ `quarkus.mongodb.connect-timeout` How long a connection can take to be opened before timing out.	Duration ❓	

	Type	Default
⚙ `quarkus.mongodb.socket-timeout` How long a send or receive on a socket can take before timing out.	Duration ❓	
⚙ `quarkus.mongodb.tls-insecure` If connecting with TLS, this option enables insecure TLS connections.	boolean	`false`
⚙ `quarkus.mongodb.tls` Whether to connect using TLS.	boolean	`false`
⚙ `quarkus.mongodb.replica-set-name` Implies that the hosts given are a seed list, and the driver will attempt to find all members of the set.	string	
⚙ `quarkus.mongodb.server-selection-timeout` How long the driver will wait for server selection to succeed before throwing an exception.	Duration ❓	
⚙ `quarkus.mongodb.local-threshold` When choosing among multiple MongoDB servers to send a request, the driver will only send that request to a server whose ping time is less than or equal to the server with the fastest ping time plus the local threshold.	Duration ❓	
⚙ `quarkus.mongodb.heartbeat-frequency` The frequency that the driver will attempt to determine the current state of each server in the cluster.	Duration ❓	
⚙ `quarkus.mongodb.read-preference` Configures the read preferences. Supported values are: `primary｜primaryPreferred｜secondary｜secondaryPreferred｜nearest`	string	
⚙ `quarkus.mongodb.max-wait-queue-size` Configures the maximum number of concurrent operations allowed to wait for a server to become available. All further operations will get an exception immediately.	int	
Write concern	**Type**	**Default**

⚙ `quarkus.mongodb.write-concern.safe` Configures the safety. If set to `true`: the driver ensures that all writes are acknowledged by the MongoDB server, or else throws an exception. (see also `w` and `wtimeoutMS`). If set fo - `false`: the driver does not ensure that all writes are acknowledged by the MongoDB server.	boolean	`true`
⚙ `quarkus.mongodb.write-concern.journal` Configures the journal writing aspect. If set to `true`: the driver waits for the server to group commit to the journal file on disk. If set to `false`: the driver does not wait for the server to group commit to the journal file on disk.	boolean	`true`
⚙ `quarkus.mongodb.write-concern.w` When set, the driver adds `w: wValue` to all write commands. It requires `safe` to be `true`. The value is typically a number, but can also be the `majority` string.	string	
⚙ `quarkus.mongodb.write-concern.retry-writes` If set to `true`, the driver will retry supported write operations if they fail due to a network error.	boolean	`false`
⚙ `quarkus.mongodb.write-concern.w-timeout` When set, the driver adds `wtimeout : ms` to all write commands. It requires `safe` to be `true`.	Duration ❓	
Credentials and authentication mechanism	**Type**	**Default**
⚙ `quarkus.mongodb.credentials.username` Configures the username.	string	
⚙ `quarkus.mongodb.credentials.password` Configures the password.	string	
⚙ `quarkus.mongodb.credentials.auth-mechanism` Configures the authentication mechanism to use if a credential was supplied. The default is unspecified, in which case the client will pick the most secure mechanism available based on the sever version. For the GSSAPI and MONGODB-X509 mechanisms, no password is accepted, only the username. Supported values: `MONGO-CR`|`GSSAPI`|`PLAIN`|`MONGODB-X509`	string	

⚙ `quarkus.mongodb.credentials.auth-source` Configures the source of the authentication credentials. This is typically the database that the credentials have been created. The value defaults to the database specified in the path portion of the connection string or in the 'database' configuration property.. If the database is specified in neither place, the default value is `admin`. This option is only respected when using the MONGO-CR mechanism (the default).	string	
⚙ `quarkus.mongodb.credentials.auth-mechanism-properties` Allows passing authentication mechanism properties.	`Map<String,String>`	required ❗

> ℹ️ *About the Duration format*
>
> The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.
>
> You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, `PT` is implicitly prepended to the value to obtain a standard `java.time.Duration` format.