Quarkus - Datasources

Many projects that use data require connections to a relational database.

The main way of obtaining connections to a database is to use a datasource and configure a JDBC driver.

In Quarkus, the preferred datasource and connection pooling implementation is Agroal.

Agroal is a modern, light weight connection pool implementation designed for very high performance and scalability, and features first class integration with the other components in Quarkus, such as security, transaction management components, health metrics.

This guide will explain how to:

- configure a datasource, or multiple datasources
- how to obtain a reference to those datasources in code
- which pool tuning configuration properties are available

Prerequisites

To complete this guide, you will need:

- less than 10 minutes
- an IDE
- JDK 1.8+ installed with JAVA_HOME configured appropriately
- Apache Maven 3.5.3+

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=agroal-quickstart \
    -DclassName="org.acme.datasource.GreetingResource" \
    -Dpath="/hello"
cd agroal-quickstart
```

This will generate:

- the Maven structure
- a landing page accessible on http://localhost:8080

- an example `Dockerfile` files for both `native` and `jvm` modes

- the application configuration file

- an `org.acme.datasource.GreetingResource` resource

- an example integration test

Adding maven dependencies

Next, you will need to add the `quarkus-agroal` dependency to your project.

You can add it using a simple Maven command:

```
./mvnw quarkus:add-extension -Dextensions="agroal"
```

> **ℹ** Agroal comes as a transitive dependency of the Hibernate ORM extension so if you are using Hibernate ORM, you don't need to add the Agroal extension dependency explicitly.

You will also need to choose, and add, the Quarkus extension for your relational database driver.

Quarkus provides driver extensions for:

- H2 - jdbc-h2

- PostgreSQL - jdbc-postgresql

- MariaDB - jdbc-mariadb

- MySQL - jdbc-mysql

- Microsoft SQL Server - jdbc-mssql

- Derby - jdbc-derby

> **ℹ** The H2 and Derby databases can normally be configured to run in "embedded mode"; the extension does not support compiling the embedded database engine into native images.
>
> Read Testing with in-memory databases (below) for suggestions regarding integration testing.

As usual, you can install the extension using `add-extension`.

To install the PostgreSQL driver dependency for instance, just run the following command:

```
./mvnw quarkus:add-extension -Dextensions="jdbc-postgresql"
```

Configuring the datasource

Once the dependencies are added to your pom.xml file, you'll need to configure Agroal.

This is done in the `src/main/resources/application.properties` file.

A viable configuration file would be:

```
quarkus.datasource.url=jdbc:h2:tcp://localhost/mem:default
quarkus.datasource.driver=org.h2.Driver
quarkus.datasource.username=username-default
quarkus.datasource.min-size=3
quarkus.datasource.max-size=13
```

There are other configuration options, detailed below.

> 💡 For more information about the Agroal extension configuration please refer to the
> Configuration Reference.

JDBC URL configurations

Each of the supported databases contains different JDBC URL configuration options. Going into each of those options is beyond the scope of this document, but it gives an overview of each database URL and link to the official documentation.

H2

```
jdbc:h2:{          {.|mem:}[name]          |          [file:]fileName          |
{tcp|ssl}:[//]server[:port][,server2[:port]]/name }[;key=value…]
```

Example

```
jdbc:h2:tcp://localhost/~/test, jdbc:h2:mem:myDB
```

H2 is an embedded database. It can run as a server, based on a file, or live completely in memory. All of these options are available as listed above. You can find more information at the official documentation.

PostgreSQL

PostgreSQL only runs as a server, as do the rest of the databases below. As such, you must specify connection details, or use the defaults.

```
jdbc:postgresql:[//][host][:port][/database][?key=value…]
```

Example

```
jdbc:postgresql://localhost/test
```

Defaults for the different parts are as follows:

host

localhost

port

5432

database

same name as the username

The official documentation go into more detail and list optional parameters as well.

MariaDB

```
jdbc:mariadb:[replication:|failover:|sequential:|aurora:]//<hostDescriptio
n>[,<hostDescription>…]/[database][?<key1>=<value1>[&<key2>=<value2>]]
hostDescription::                      <host>[:<portnumber>]                      or
address=(host=<host>)[(port=<portnumber>)][(type=(master|slave))]
```

Example

```
jdbc:mariadb://localhost:3306/test
```

You can find more information about this feature and others detailed in the official documentation.

MySQL

```
jdbc:mysql:[replication:|failover:|sequential:|aurora:]//<hostDescription>
[,<hostDescription>…]/[database][?<key1>=<value1>[&<key2>=<value2>]]
hostDescription::                      <host>[:<portnumber>]                      or
address=(host=<host>)[(port=<portnumber>)][(type=(master|slave))]
```

Example

```
jdbc:mysql://localhost:3306/test
```

You can find more information about this feature and others detailed in the official documentation.

Microsoft SQL Server

Microsoft SQL Server takes a connection URL in the following form:

```
jdbc:sqlserver://[serverName[\instanceName][:portNumber]][;property=value[
;property=value]]
```

Example

```
jdbc:sqlserver://localhost:1433;databaseName=AdventureWorks
```

The Microsoft SQL Server JDBC driver works essentially the same as the others. More details can be found in the official documentation.

Derby

```
jdbc:derby:[//serverName[:portNumber]/][memory:]databaseName[;property=value[;property=value]]
```

Example

```
jdbc:derby://localhost:1527/myDB, jdbc:derby:memory:myDB;create=true
```

Derby is an embedded database. It can run as a server, based on a file, or live completely in memory. All of these options are available as listed above. You can find more information at the official documentation.

Injecting a Datasource

Because Quarkus uses CDI, injecting a datasource is very simple:

```
@Inject
AgroalDataSource defaultDataSource;
```

In the above example, the type is `AgroalDataSource` which is a subtype of `javax.sql.DataSource`. Because of this, you can also use `javax.sql.DataSource`.

Multiple Datasources

Agroal allows you to configure multiple datasources. It works exactly the same way as a single datasource, with one important change: a name.

```
quarkus.datasource.driver=org.h2.Driver
quarkus.datasource.url=jdbc:h2:tcp://localhost/mem:default
quarkus.datasource.username=username-default
quarkus.datasource.min-size=3
quarkus.datasource.max-size=13

quarkus.datasource.users.driver=org.h2.Driver
quarkus.datasource.users.url=jdbc:h2:tcp://localhost/mem:users
quarkus.datasource.users.username=username1
quarkus.datasource.users.min-size=1
quarkus.datasource.users.max-size=11

quarkus.datasource.inventory.driver=org.h2.Driver
quarkus.datasource.inventory.url=jdbc:h2:tcp://localhost/mem:inventory
quarkus.datasource.inventory.username=username2
quarkus.datasource.inventory.min-size=2
quarkus.datasource.inventory.max-size=12
```

Notice there's an extra bit in the key. The syntax is as follows: `quarkus.datasource.[optional name.][datasource property]`.

Named Datasource Injection

When using multiple datasources, each `DataSource` also has the `io.quarkus.agroal.DataSource` qualifier with the name of the datasource in the property as the value. Using the above properties to configure three different datasources, you can also inject each one as follows:

```java
@Inject
AgroalDataSource defaultDataSource;

@Inject
@DataSource("users")
AgroalDataSource dataSource1;

@Inject
@DataSource("inventory")
AgroalDataSource dataSource2;
```

Datasource Health Check

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

So when you access the `/health/ready` endpoint of your application you will have information about the datasource validation status. If you have multiple datasources, all datasources will be checked and the status will be `DOWN` as soon as there is one datasource validation failure.

This behavior can be disabled via the property `quarkus.datasource.health.enabled`.

Datasource Metrics

If you are using the `quarkus-smallrye-metrics` extension, `quarkus-agroal` can expose some data source metrics on the `/metrics` endpoint. This can be turned on by setting the property `quarkus.datasource.metrics.enabled` to true.

For the exposed metrics to contain any actual values, it is necessary that metric collection is enabled internally by Agroal mechanisms. By default, this metric collection mechanism gets turned on for all data sources if the `quarkus-smallrye-metrics` is present and metrics for the Agroal extension are enabled. If you want to disable metrics for a particular data source, this can be done by setting `quarkus.datasource.enable-metrics` to `false` (or `quarkus.datasource.<datasource name>.enable-metrics` for a named datasource). This disables collecting the metrics as well as exposing them in the `/metrics` endpoint, because it does not make sense to expose metrics if the mechanism to collect them is disabled.

Conversely, setting `quarkus.datasource.enable-metrics` to `true` (or `quarkus.datasource.<datasource name>.enable-metrics` for a named datasource) explicitly can be used to enable collection of metrics even if the `quarkus-smallrye-metrics` extension is not in use. This can be useful if you need to access the collected metrics programmatically. They are available after calling `dataSource.getMetrics()` on an injected `AgroalDataSource` instance. If collection of metrics is disabled for this data source, all values will be zero.

Narayana Transaction Manager integration

If the Narayana JTA extension is also available, integration is automatic.

You can override this by setting the `transactions` configuration property - see the Configuration Reference below.

Testing with in-memory databases

Some databases like H2 and Derby are commonly used in "embedded mode" as a facility to run quick integration tests.

Our suggestion is to use the real database you intend to use in production; container technologies made this simple enough so you no longer have an excuse. Still, there are sometimes good reasons to also want the ability to run quick integration tests using the JVM powered databases, so this is possible as well.

It is important to remember that when configuring H2 (or Derby) to use the embedded engine, this will work as usual in JVM mode but such an application will not compile into a native image, as the Quarkus extensions only cover for making the JDBC client code compatible with the native compilation step: embedding the whole database engine into a native image is currently not implemented.

If you plan to run such integration tests in the JVM exclusively, it will of course work as usual.

If you want the ability to run such integration test in both JVM and/or native images, we have some cool helpers for you: just add either `@QuarkusTestResource(H2DatabaseTestResource.class)` or `@QuarkusTestResource(DerbyDatabaseTestResource.class)` on any class in your integration tests, this will make sure the testsuite starts (and stops) the embedded database into a separate process as necessary to run your tests.

These additional helpers are provided by the artifacts having Maven coordinates `io.quarkus:quarkus-test-h2:1.3.0.Alpha1` and `io.quarkus:quarkus-test-derby:1.3.0.Alpha1`, respectively for H2 and Derby.

Follows an example for H2:

```
package my.app.integrationtests.db;

import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.h2.H2DatabaseTestResource;

@QuarkusTestResource(H2DatabaseTestResource.class)
public class TestResources {
}
```

This will allow you to test your application even when it's compiled into a native image, while the database will run in the JVM as usual.

Connect to it using:

```
quarkus.datasource.url=jdbc:h2:tcp://localhost/mem:test
quarkus.datasource.driver=org.h2.Driver
```

Agroal Configuration Reference

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

Configuration property	Type	Default
🗄 `quarkus.datasource.driver` The datasource driver class name	string	
🗄 `quarkus.datasource.transactions` Whether we want to use regular JDBC transactions, XA, or disable all transactional capabilities. When enabling XA you will need a driver implementing `javax.sql.XADataSource`.	`enabled`, `xa`, `disabled`	`enabled`
🗄 `quarkus.datasource.enable-metrics` Enable datasource metrics collection. If unspecified, collecting metrics will be enabled by default if the smallrye-metrics extension is active.	boolean	
🗄 `quarkus.datasource.health.enabled` Whether or not an health check is published in case the smallrye-health extension is present	boolean	`true`

🗄 `quarkus.datasource.metrics.enabled` Whether or not datasource metrics are published in case the smallrye-metrics extension is present (default to false). NOTE: This is different from the "enable-metrics" property that needs to be set on data source level to enable collection of metrics for that data source.	boolean	`false`
⚙ `quarkus.datasource.url` The datasource URL	string	
⚙ `quarkus.datasource.username` The datasource username	string	
⚙ `quarkus.datasource.password` The datasource password	string	
⚙ `quarkus.datasource.credentials-provider` The credentials provider name	string	
⚙ `quarkus.datasource.credentials-provider-type` The credentials provider type. It is the `@Named` value of the credentials provider bean. It is used to discriminate if multiple CredentialsProvider beans are available. For Vault it is: vault-credentials-provider. Not necessary if there is only one credentials provider available.	string	
⚙ `quarkus.datasource.initial-size` The initial size of the pool. Usually you will want to set the initial size to match at least the minimal size, but this is not enforced so to allow for architectures which prefer a lazy initialization of the connections on boot, while being able to sustain a minimal pool size after boot.	int	
⚙ `quarkus.datasource.min-size` The datasource pool minimum size	int	`0`
⚙ `quarkus.datasource.max-size` The datasource pool maximum size	int	`20`

⚙ `quarkus.datasource.background-validation-interval` The interval at which we validate idle connections in the background. Set to `0` to disable background validation.	Duration ❓	`2M`
⚙ `quarkus.datasource.acquisition-timeout` The timeout before cancelling the acquisition of a new connection	Duration ❓	`5`
⚙ `quarkus.datasource.leak-detection-interval` The interval at which we check for connection leaks.	Duration ❓	
⚙ `quarkus.datasource.idle-removal-interval` The interval at which we try to remove idle connections.	Duration ❓	`5M`
⚙ `quarkus.datasource.max-lifetime` The max lifetime of a connection.	Duration ❓	
⚙ `quarkus.datasource.transaction-isolation-level` The transaction isolation level.	`undefined`, `none`, `read-uncommitted`, `read-committed`, `repeatable-read`, `serializable`	
⚙ `quarkus.datasource.detect-statement-leaks` When enabled Agroal will be able to produce a warning when a connection is returned to the pool without the application having closed all open statements. This is unrelated with tracking of open connections. Disable for peak performance, but only when there's high confidence that no leaks are happening.	boolean	`true`
⚙ `quarkus.datasource.new-connection-sql` Query executed when first using a connection.	string	

	Type	Default
⚙ `quarkus.datasource.validation-query-sql` Query executed to validate a connection.	string	
Additional named datasources	**Type**	**Default**
▤ `quarkus.datasource."datasource-name".driver` The datasource driver class name	string	
▤ `quarkus.datasource."datasource-name".transactions` Whether we want to use regular JDBC transactions, XA, or disable all transactional capabilities. When enabling XA you will need a driver implementing `javax.sql.XADataSource`.	`enabled`, `xa`, `disabled`	`enabled`
▤ `quarkus.datasource."datasource-name".enable-metrics` Enable datasource metrics collection. If unspecified, collecting metrics will be enabled by default if the smallrye-metrics extension is active.	boolean	
⚙ `quarkus.datasource."datasource-name".url` The datasource URL	string	
⚙ `quarkus.datasource."datasource-name".username` The datasource username	string	
⚙ `quarkus.datasource."datasource-name".password` The datasource password	string	
⚙ `quarkus.datasource."datasource-name".credentials-provider` The credentials provider name	string	
⚙ `quarkus.datasource."datasource-name".credentials-provider-type` The credentials provider type. It is the @Named value of the credentials provider bean. It is used to discriminate if multiple CredentialsProvider beans are available. For Vault it is: vault-credentials-provider. Not necessary if there is only one credentials provider available.	string	

⚙ `quarkus.datasource."datasource-name".initial-size` The initial size of the pool. Usually you will want to set the initial size to match at least the minimal size, but this is not enforced so to allow for architectures which prefer a lazy initialization of the connections on boot, while being able to sustain a minimal pool size after boot.	int	
⚙ `quarkus.datasource."datasource-name".min-size` The datasource pool minimum size	int	0
⚙ `quarkus.datasource."datasource-name".max-size` The datasource pool maximum size	int	20
⚙ `quarkus.datasource."datasource-name".background-validation-interval` The interval at which we validate idle connections in the background. Set to 0 to disable background validation.	Duration ❓	2M
⚙ `quarkus.datasource."datasource-name".acquisition-timeout` The timeout before cancelling the acquisition of a new connection	Duration ❓	5
⚙ `quarkus.datasource."datasource-name".leak-detection-interval` The interval at which we check for connection leaks.	Duration ❓	
⚙ `quarkus.datasource."datasource-name".idle-removal-interval` The interval at which we try to remove idle connections.	Duration ❓	5M
⚙ `quarkus.datasource."datasource-name".max-lifetime` The max lifetime of a connection.	Duration ❓	

⚙ `quarkus.datasource."datasource-name".transaction-isolation-level` The transaction isolation level.	`undefined`, `none`, `read-uncommitted`, `read-committed`, `repeatable-read`, `serializable`	
⚙ `quarkus.datasource."datasource-name".detect-statement-leaks` When enabled Agroal will be able to produce a warning when a connection is returned to the pool without the application having closed all open statements. This is unrelated with tracking of open connections. Disable for peak performance, but only when there's high confidence that no leaks are happening.	`boolean`	`true`
⚙ `quarkus.datasource."datasource-name".new-connection-sql` Query executed when first using a connection.	`string`	
⚙ `quarkus.datasource."datasource-name".validation-query-sql` Query executed to validate a connection.	`string`	

ℹ *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.