

Quarkus - Datasources

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

The usual way of obtaining connections to a database is to use a datasource and configure a JDBC driver. But you might also prefer using a reactive driver to connect to your database in a reactive way.

Quarkus has you covered either way:

- For JDBC, the preferred datasource and connection pooling implementation is [Agroal](#).
- For reactive, we use the [Vert.x](#) reactive drivers.

Both are configured via unified and flexible configuration.



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

This guide is mainly about datasource configuration. If you want more details about how to consume and make use of a reactive datasource, please refer to the [Reactive SQL clients guide](#).

TL;DR

This is a quick introduction to datasource configuration. If you want a better understanding of how all this works, this guide has a lot more information in the subsequent paragraphs.

JDBC datasource

Add the `agroal` extension plus one of `jdbc-derby`, `jdbc-h2`, `jdbc-mariadb`, `jdbc-mssql`, `jdbc-mysql` or `jdbc-postgresql`.

Then configure your datasource:

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=<your username>
quarkus.datasource.password=<your password>

quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/hibernate_orm_test
quarkus.datasource.jdbc.min-size=4
quarkus.datasource.jdbc.max-size=16
```

Reactive datasource

Add either the `reactive-pg-client` or the `reactive-mysql-client` extension.

Then configure your reactive datasource:

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=<your username>
quarkus.datasource.password=<your password>

quarkus.datasource.reactive.url=postgresql:///your_database
quarkus.datasource.reactive.max-size=20
```

Default datasource

A datasource can be either a JDBC datasource, a reactive one or both. It all depends on how you configure it and which extensions you added to your project.

To define a datasource, start with the following:

```
quarkus.datasource.db-kind=h2
```

The database kind defines which type of database you will connect to.

We currently include these built-in database kinds:

- Derby: `derby`
- H2: `h2`
- MariaDB: `mariadb`
- Microsoft SQL Server: `mssql`
- MySQL: `mysql`
- PostgreSQL: `postgresql`, `pgsql` or `pg`

Giving Quarkus the database kind you are targeting will facilitate configuration. By using a JDBC driver extension and setting the kind in the configuration, Quarkus resolves the JDBC driver automatically, so you don't need to configure it yourself. If you want to use a database kind that is not part of the built-in ones, use **other** and define the JDBC driver explicitly.



You can use any JDBC driver in a Quarkus app in JVM mode (see [Using other databases](#)). It is unlikely that it will work when compiling your application to a native executable though.

If you plan to make a native executable, we recommend you use the existing JDBC Quarkus extensions (or contribute one for your driver).

There is a good chance you will need to define some credentials to access your database.

This is done by configuring the following properties:

```
quarkus.datasource.username=<your username>
quarkus.datasource.password=<your password>
```

You can also retrieve the password from Vault by [using a credential provider](#) for your datasource.

Once you have defined the database kind and the credentials, you are ready to configure either a JDBC datasource, a reactive one, or both.

JDBC datasource

JDBC is the most common database connection pattern. You typically need a JDBC datasource when using Hibernate ORM.

Install the Maven dependencies

First, 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"
```



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:

- Derby - **jdbc-derby**
- H2 - **jdbc-h2**

- MariaDB - `jdbc-mariadb`
- Microsoft SQL Server - `jdbc-mssql`
- MySQL - `jdbc-mysql`
- PostgreSQL - `jdbc-postgresql`

See [Use a database with no built-in extension or with a different driver](#) if you want to use a JDBC driver for another database.



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, run the following command:

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

Configure the JDBC connection

Configuring your JDBC connection is easy, the only mandatory property is the JDBC URL.

```
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/hibernate_orm_test
```



Note the `jdbc` prefix in the property name. All the configuration properties specific to JDBC have this prefix.



For more information about the JDBC URL format, please refer to the [JDBC url reference section](#).

When using one of the built-in datasource kinds, the JDBC driver is resolved automatically to the following values:

Table 1. Database kind to JDBC driver mapping

Database kind	JDBC driver	XA driver
<code>derby</code>	<code>org.apache.derby.jdbc.ClientDriver</code>	<code>org.apache.derby.jdbc.ClientXADataSource</code>
<code>h2</code>	<code>org.h2.Driver</code>	<code>org.h2.jdbcx.JdbcDataSource</code>

Database kind	JDBC driver	XA driver
<code>mariadb</code>	<code>org.mariadb.jdbc.Driver</code>	<code>org.mariadb.jdbc.MySQLDataSource</code>
<code>mssql</code>	<code>com.microsoft.sqlserver.jdbc.SQLServerDriver</code>	<code>com.microsoft.sqlserver.jdbc.SQLServerXADataSource</code>
<code>mysql</code>	<code>com.mysql.cj.jdbc.Driver</code>	<code>com.mysql.cj.jdbc.MysqlXADataSource</code>
<code>postgresql</code>	<code>org.postgresql.Driver</code>	<code>org.postgresql.xa.PGXADDataSource</code>



As previously stated, most of the time, this automatic resolution will suit you and you won't need to configure the driver.

Use a database with no built-in extension or with a different driver

You can use a specific driver if you need to (for instance for using the OpenTracing driver) or if you want to use a database for which Quarkus does not have a built-in JDBC driver extension.

Without an extension, the driver will work fine in any Quarkus app running in JVM mode. It is unlikely that it will work when compiling your application to a native executable though. If you plan to make a native executable, we recommend you use the existing JDBC Quarkus extensions (or contribute one for your driver).

Here is how you would use the OpenTracing driver:

```
quarkus.datasource.jdbc.driver=io.opentracing.contrib.jdbc.TracingDriver
```

Here is how you would define access to a database with no built-in support (in JVM mode):

```
quarkus.datasource.db-kind=other
quarkus.datasource.jdbc.driver=oracle.jdbc.driver.OracleDriver
quarkus.datasource.jdbc.url=jdbc:oracle:thin:@192.168.1.12:1521/ORCL_SVC
quarkus.datasource.username=scott
quarkus.datasource.password=tiger
```

More configuration

You can configure a lot more things, for instance the size of the connection pool.

Please refer to the [JDBC configuration reference](#) for all the details about the JDBC configuration knobs.

Consuming the datasource

If you are using Hibernate ORM, the datasource will be consumed automatically.

If for whatever reason, access to the datasource is needed in code, it can be obtained as any other bean in the following manner:

```
@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` as the injected type.

Reactive datasource

If you prefer using a reactive datasource, Quarkus offers MySQL/MariaDB and PostgreSQL reactive clients.

Install the Maven dependencies

Depending on whether you wish to use MySQL/MariaDB or PostgreSQL, you need to add either the `quarkus-reactive-pg-client` or the `quarkus-reactive-mysql-client` extension.

The installed extension must be consistent with the `quarkus.datasource.db-kind` you define in your datasource configuration.

Configure the JDBC datasource

Once the driver is there, you just need to configure the connection URL.

Optionally, but highly recommended, you should define a proper size for your connection pool.

```
quarkus.datasource.reactive.url=postgresql:///your_database
quarkus.datasource.reactive.max-size=20
```

Both

By default, if you include both a JDBC extension (+ Agroal) and a reactive datasource extension handling the given database kind, both will be created.

If you want to disable the JDBC datasource explicitly, use:

```
quarkus.datasource.jdbc=false
```

If you want to disable the reactive datasource explicitly, use:

```
quarkus.datasource.reactive=false
```



Most of the time, the configuration above won't be necessary as either a JDBC driver or a reactive datasource extension will be present and not both.

Multiple Datasources

Configuring Multiple Datasources

For now, multiple datasources are only supported for JDBC and the Agroal extension. So it is not currently possible to create multiple reactive datasources.



Currently, Hibernate ORM supports only one persistence unit which uses the default datasource. If you want to use multiple datasources, you will need to manipulate them directly.

Multiple persistence units support is planned for a future version of Quarkus.

Defining multiple datasources works exactly the same way as defining a single datasource, with one important change: you define a name.

In the following example, you have 3 different datasources:

- The default one,
- A datasource named **users**,
- A datasource named **inventory**,

each with its own configuration.

```

quarkus.datasource.db-kind=h2
quarkus.datasource.username=username-default
quarkus.datasource.jdbc.url=jdbc:h2:tcp://localhost/mem:default
quarkus.datasource.jdbc.min-size=3
quarkus.datasource.jdbc.max-size=13

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

quarkus.datasource.inventory.db-kind=h2
quarkus.datasource.inventory.username=username2
quarkus.datasource.inventory.jdbc.url=jdbc:h2:tcp://localhost/mem:inventory
quarkus.datasource.inventory.jdbc.min-size=2
quarkus.datasource.inventory.jdbc.max-size=12

```

Notice there is 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 as the value. Using the above properties to configure three different datasources, you can also inject each one as follows:

```

@Inject
AgroalDataSource defaultDataSource;

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

@Inject
@DataSource("inventory")
AgroalDataSource inventoryDataSource;

```

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.jdbc.enable-metrics` to `false` (or `quarkus.datasource.<datasource name>.jdbc.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.jdbc.enable-metrics` to `true` (or `quarkus.datasource.<datasource name>.jdbc.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 test suite 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` and `io.quarkus:quarkus-test-derby`, 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
```

Common Datasource Configuration Reference


 Configuration property fixed at build time - All other configuration properties are overridable at runtime





Configuration property	Type	Default
 <code>quarkus.datasource.db-kind</code> The kind of database we will connect to (e.g. h2, postgresql...).	string	

<p> <code>quarkus.datasource.health.enabled</code></p> <p>Whether or not an health check is published in case the smallrye-health extension is present. This is a global setting and is not specific to a datasource.</p>	boolean	<code>true</code>
<p> <code>quarkus.datasource.metrics.enabled</code></p> <p>Whether or not datasource metrics are published in case the smallrye-metrics extension is present. This is a global setting and is not specific to a datasource. NOTE: This is different from the "jdbc.enable-metrics" property that needs to be set on the JDBC datasource level to enable collection of metrics for that datasource.</p>	boolean	<code>false</code>
<p><code>quarkus.datasource.username</code></p> <p>The datasource username</p>	string	
<p><code>quarkus.datasource.password</code></p> <p>The datasource password</p>	string	
<p><code>quarkus.datasource.credentials-provider</code></p> <p>The credentials provider name</p>	string	
<p><code>quarkus.datasource.credentials-provider-type</code></p> <p>The credentials provider type. It is the <code>@Named</code> value of the credentials provider bean. It is used to discriminate if multiple <code>CredentialsProvider</code> beans are available. For Vault it is: <code>vault-credentials-provider</code>. Not necessary if there is only one credentials provider available.</p>	string	
<p><code>quarkus.datasource.max-size</code></p>	int	<code>20</code>
Additional named datasources	Type	Default
<p> <code>quarkus.datasource."datasource-name".db-kind</code></p> <p>The kind of database we will connect to (e.g. h2, postgresql...).</p>	string	
<p><code>quarkus.datasource."datasource-name".username</code></p> <p>The datasource username</p>	string	
<p><code>quarkus.datasource."datasource-name".password</code></p> <p>The datasource password</p>	string	

<code>quarkus.datasource."datasource-name".credentials-provider</code>	string	
The credentials provider name		
<code>quarkus.datasource."datasource-name".credentials-provider-type</code>	string	
The credentials provider type. It is the <code>@Named</code> value of the credentials provider bean. It is used to discriminate if multiple <code>CredentialsProvider</code> beans are available. For Vault it is: <code>vault-credentials-provider</code> . Not necessary if there is only one credentials provider available.		
<code>quarkus.datasource."datasource-name".max-size</code>	int	20


JDBC Configuration Reference

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
 <code>quarkus.datasource.jdbc</code>	boolean	true
If we create a JDBC datasource for this datasource.		
 <code>quarkus.datasource.jdbc.driver</code>	string	
The datasource driver class name		
 <code>quarkus.datasource.jdbc.transactions</code>	enable d, xa, disabl ed	enable d
Whether we want to use regular JDBC transactions, XA, or disable all transactional capabilities. When enabling XA you will need a driver implementing <code>javax.sql.XADataSource</code> .		
 <code>quarkus.datasource.jdbc.enable-metrics</code>	boolean	
Enable datasource metrics collection. If unspecified, collecting metrics will be enabled by default if the smallrye-metrics extension is active.		
<code>quarkus.datasource.jdbc.url</code>	string	
The datasource URL		

<code>quarkus.datasource.jdbc.initial-size</code>		
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	
<code>quarkus.datasource.jdbc.min-size</code>		
The datasource pool minimum size	int	0
<code>quarkus.datasource.jdbc.max-size</code>		
The datasource pool maximum size	int	20
<code>quarkus.datasource.jdbc.background-validation-interval</code>		
The interval at which we validate idle connections in the background. Set to 0 to disable background validation.	Duration ?	2M
<code>quarkus.datasource.jdbc.acquisition-timeout</code>		
The timeout before cancelling the acquisition of a new connection	Duration ?	5
<code>quarkus.datasource.jdbc.leak-detection-interval</code>		
The interval at which we check for connection leaks.	Duration ?	
<code>quarkus.datasource.jdbc.idle-removal-interval</code>		
The interval at which we try to remove idle connections.	Duration ?	5M
<code>quarkus.datasource.jdbc.max-lifetime</code>		
The max lifetime of a connection.	Duration ?	

<code>quarkus.datasource.jdbc.transaction-isolation-level</code> The transaction isolation level.	undefined, none, read-uncommitted, read-committed, repeatable-read, serializable	
<code>quarkus.datasource.jdbc.detect-statement-leaks</code> 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
<code>quarkus.datasource.jdbc.new-connection-sql</code> Query executed when first using a connection.	string	
<code>quarkus.datasource.jdbc.validation-query-sql</code> Query executed to validate a connection.	string	
Additional named datasources	Type	Default
🔒 <code>quarkus.datasource."datasource-name".jdbc</code> If we create a JDBC datasource for this datasource.	boolean	true
🔒 <code>quarkus.datasource."datasource-name".jdbc.driver</code> The datasource driver class name	string	
🔒 <code>quarkus.datasource."datasource-name".jdbc.transactions</code> Whether we want to use regular JDBC transactions, XA, or disable all transactional capabilities. When enabling XA you will need a driver implementing <code>javax.sql.XADataSource</code> .	enabled, xa, disabled	enabled

 <code>quarkus.datasource."datasource-name".jdbc.enable-metrics</code> Enable datasource metrics collection. If unspecified, collecting metrics will be enabled by default if the smallrye-metrics extension is active.	boolean	
<code>quarkus.datasource."datasource-name".jdbc.url</code> The datasource URL	string	
<code>quarkus.datasource."datasource-name".jdbc.initial-size</code> 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	
<code>quarkus.datasource."datasource-name".jdbc.min-size</code> The datasource pool minimum size	int	0
<code>quarkus.datasource."datasource-name".jdbc.max-size</code> The datasource pool maximum size	int	20
<code>quarkus.datasource."datasource-name".jdbc.background-validation-interval</code> The interval at which we validate idle connections in the background. Set to 0 to disable background validation.	Duration ?	2M
<code>quarkus.datasource."datasource-name".jdbc.acquisition-timeout</code> The timeout before cancelling the acquisition of a new connection	Duration ?	5
<code>quarkus.datasource."datasource-name".jdbc.leak-detection-interval</code> The interval at which we check for connection leaks.	Duration ?	
<code>quarkus.datasource."datasource-name".jdbc.idle-removal-interval</code> The interval at which we try to remove idle connections.	Duration ?	5M
<code>quarkus.datasource."datasource-name".jdbc.max-lifetime</code> The max lifetime of a connection.	Duration ?	

<code>quarkus.datasource."datasource-name".jdbc.transaction-isolation-level</code> The transaction isolation level.	undefined, none, read-uncommitted, read-committed, repeatable-read, serializable	
<code>quarkus.datasource."datasource-name".jdbc.detect-statement-leaks</code> 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
<code>quarkus.datasource."datasource-name".jdbc.new-connection-sql</code> Query executed when first using a connection.	string	
<code>quarkus.datasource."datasource-name".jdbc.validation-query-sql</code> Query executed to validate a connection.	string	



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.

JDBC URL Reference

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 the following section gives an overview of each database URL and a 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: ] // < hostDescription  
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=val
ue[;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](#).


Reactive Datasource Configuration Reference

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
 <code>quarkus.datasource.reactive</code> If we create a Reactive datasource for this datasource.	boolean	<code>true</code>


<code>quarkus.datasource.reactive.url</code> The datasource URL.	string	
<code>quarkus.datasource.reactive.max-size</code> The datasource pool maximum size.	int	

Reactive MariaDB/MySQL Specific Configuration

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
<code>quarkus.datasource.reactive.mysql.cache-prepared-statements</code> Whether prepared statements should be cached on the client side.	boolean	
<code>quarkus.datasource.reactive.mysql.charset</code> Charset for connections.	string	
<code>quarkus.datasource.reactive.mysql.collation</code> Collation for connections.	string	

Reactive PostgreSQL Specific Configuration

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
<code>quarkus.datasource.reactive.postgresql.cache-prepared-statements</code> Whether prepared statements should be cached on the client side.	boolean	
<code>quarkus.datasource.reactive.postgresql.pipelining-limit</code> The maximum number of inflight database commands that can be pipelined.	int	