

R2DBC - Reactive Relational Database Connectivity

Ben Hale<bhale@pivotal.io>, Mark Paluch <mpaluch@vmware.com>, Greg
Turnquist <gturnquist@pivotal.io>, Jay Bryant <jbryant@pivotal.io>, Elena
Felder

Version 0.8.6.RELEASE, 2021-09-20: Final

Table of Contents

Preface	2
License	2
Foreword	2
Organization of this document	3
1. Introduction	4
1.1. The R2DBC SPI	4
1.2. Requirements and Conventions	4
1.3. Target Audience	4
1.4. Acknowledgements	5
1.5. Following Development	5
1.6. Project Metadata	5
2. Goals	6
2.1. Enabling Reactive Relational Database Connectivity	6
2.2. Fitting into Reactive JVM platforms	6
2.3. Offering Vendor-neutral Access to Standard Features	7
2.4. Embracing Vendor-specific Features	7
2.5. Keeping the Focus on SQL	7
2.6. Keeping It Minimal and Simple	7
2.7. Providing a Foundation for Tools and Higher-level APIs	7
2.8. Specifying Requirements Unambiguously	7
3. Compliance	8
3.1. Definitions	8
3.2. Guidelines and Requirements	9
3.3. R2DBC SPI Compliance	10
4. Overview	11
4.1. Establishing a Connection	11
4.1.1. Using <code>ConnectionFactory</code> Discovery	11
4.1.2. R2DBC Connection URL	12
4.2. Running SQL and Retrieving Results	16
5. Connections	17
5.1. The <code>ConnectionFactory</code> Interface	17
5.1.1. ConnectionFactory Metadata	18
5.2. <code>ConnectionFactory</code> Discovery Mechanism	18
5.3. The <code>ConnectionFactoryOptions</code> Class	19
5.4. Obtaining <code>Connection</code> Objects	21
5.5. Connection Metadata	21
5.6. Validating <code>Connection</code> Objects	21
5.7. Closing <code>Connection</code> Objects	22

6. Transactions	23
6.1. Transaction Boundaries	23
6.2. Auto-commit Mode	23
6.3. Transaction Isolation	24
6.3.1. Performance Considerations	24
6.4. Savepoints	24
6.4.1. Working with Savepoints	25
6.4.2. Releasing a Savepoint	26
7. Statements	27
7.1. The Statement Interface	27
7.1.1. Creating Statements	27
7.1.2. Running Statement Objects	27
7.2. Parameterized Statements	27
7.2.1. Binding Parameters	28
7.2.2. Batching	29
7.2.3. Setting NULL Parameters	29
7.3. Retrieving Auto Generated Values	29
7.4. Performance Hints	30
8. Batches	31
8.1. The Batch Interface	31
8.1.1. Creating Batches	31
8.1.2. Executing Batch Objects	31
9. Results	32
9.1. Result Characteristics	32
9.2. Creating Result Objects	33
9.2.1. Cursor Movement	33
9.3. Rows	34
9.3.1. Retrieving Values	34
9.4. Interface Methods	34
10. Column and Row Metadata	36
10.1. Obtaining a RowMetadata Object	36
10.2. Retrieving ColumnMetadata	36
10.3. Retrieving General Information for a Column	37
11. Exceptions	38
11.1. General Exceptions	38
11.1.1. IllegalArgumentException	38
11.1.2. IllegalStateException	38
11.1.3. UnsupportedOperationException	38
11.1.4. R2dbcException	38
11.2. Categorized Exceptions	39
11.2.1. Non-Transient Exceptions	39

11.2.2. Transient Exceptions	39
12. Data Types	41
12.1. Mapping of Data Types	41
12.2. Mapping of Advanced Data Types	45
12.2.1. Blob and Clob Objects	45
12.2.2. Creating Blob and Clob Objects	45
12.2.3. Retrieving Blob and Clob Objects from a Row	45
12.2.4. Accessing Blob and Clob Data	46
12.2.5. Releasing Blob and Clob	46
13. Extensions	47
13.1. Wrapped Interface	47
13.1.1. Usage	47
13.1.2. Interface Methods	47
13.1.3. The unwrap Method	47
13.2. Closeable Interface	48
13.2.1. Usage	48
13.2.2. Interface Methods	48
13.2.3. The close Method	48



Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Preface

License

Specification: R2DBC - Reactive Relational Database Connectivity

Version: 0.8.6.RELEASE

Status: Final

Specification Lead: Reactive Foundation

Release: 2021-09-20

Copyright 2017-2021 the original author or authors.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<https://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

Foreword

R2DBC brings a reactive programming API to relational data stores. The [Introduction](#) contains more details about its origins and explains its goals.

This document describes the first and initial generation of R2DBC.

Organization of this document

This document is organized into the following parts:

- [Introduction](#)
- [Goals](#)
- [Compliance](#)
- [Overview](#)
- [Connections](#)
- [Transactions](#)
- [Statements](#)
- [Batches](#)
- [Results](#)
- [Column and Row Metadata](#)
- [Exceptions](#)
- [Data Types](#)
- [Extensions](#)

Chapter 1. Introduction

R2DBC stands for Reactive Relational Database Connectivity. R2DBC started as an experiment and proof of concept to enable integration of SQL databases into systems that use reactive programming models — Reactive in the sense of an event-driven, non-blocking, and functional programming model that does not make assumptions over concurrency or asynchronicity. Instead, it assumes that scheduling and parallelization happen as part of runtime scheduling.

1.1. The R2DBC SPI

The R2DBC SPI provides reactive programmatic access to relational databases from Java and other JVM-based programming languages.

R2DBC specifies a service-provider interface (SPI) that is intended to be implemented by driver vendors and used by client libraries. By using the R2DBC SPI, applications written in a JVM programming language can run SQL statements and retrieve results by using an underlying data source. You can also use the R2DBC SPI to interact with multiple data sources in a distributed, heterogeneous environment. R2DBC targets primarily, but is not limited to, relational databases. It aims for a range of data sources whose query and statement interface is based on SQL (or an SQL-like dialect) and that represent their data in a tabular form.

A key difference between R2DBC and imperative data access SPIs is the deferred nature of execution. R2DBC is, therefore, based on [Reactive Streams](#) and uses the concepts of [Publisher](#) and [Subscriber](#) to allow non-blocking back-pressure-aware data access.

1.2. Requirements and Conventions

R2DBC requires Java 8 as baseline and [Reactive Streams](#) in terms of a reactive inter-op API.

A note about modules: R2DBC allow for deployment to JDK 9's module path ("Jigsaw"). For use in Jigsaw-enabled applications, the R2DBC SPI comes with "Automatic-Module-Name" manifest entries which define stable language-level module names ("r2dbc.spi") instead using the artifact name. Of course, R2DBC jars keep working fine on the classpath on both JDK 8 and 9+.

R2DBC's SPI uses non-null semantics by default. Nullable API members are annotated with [@io.r2dbc.spi.Nullable](#). This annotation and [@NonNullApi](#) are meta-annotated with [JSR 305](#) annotations (a dormant but wide-spread JSR). JSR-305 meta-annotations let tooling vendors like IDEA or Kotlin provide null-safety support in a generic way, without having to hard-code support for R2DBC annotations. It is not necessary nor recommended to add a JSR-305 dependency to the project classpath to take advantage of a null-safe API.

R2DBC generally uses zero-based indexes for parameter binding and columns.

1.3. Target Audience

This specification is targeted primarily towards:

- Vendors of drivers that implement the R2DBC SPI.

- Vendors of client implementations who wish to implement a client on top of the R2DBC SPI.
- Vendors of runtime libraries who wish to embed R2DBC into their eco-system to provide R2DBC runtime services.

This specification is also intended to offer:

- An introduction for end-users whose applications use the R2DBC SPI.
- A starting point for developers of other SPIs layered on top of the R2DBC SPI.

Code examples in this specification assume external subscription to **Publisher** objects for brevity.

1.4. Acknowledgements

The R2DBC specification work is being conducted as an effort of individuals that recognized the demand for a reactive, standardized API for relational database access. We want to thank all [contributing members](#) for their countless hours of work and discussion.

Thanks also go to Ollie Drotbohm, without whom this initiative would not even exist.

1.5. Following Development

For information on R2DBC source code repositories, nightly builds, and snapshot artifacts, see the [R2DBC](#) homepage. You can help make R2DBC best serve the needs of the community by interacting with developers through the community. To follow developer activity, look for the mailing list information on the R2DBC homepage. If you encounter a bug or want to suggest an improvement, please create a ticket on the R2DBC issue tracker. R2DBC has an open-source organization on GitHub that bundles the various projects (SPI and drivers) under R2DBC.

To stay up to date with the latest news and announcements in the R2DBC eco system, you can subscribe to the mailing list. You can also follow the project team on Twitter ([@R2DBC](#)).

1.6. Project Metadata

- Version control: <https://github.com/r2dbc/r2dbc-spi>
- Mailing list: <https://groups.google.com/forum/#!forum/r2dbc>
- Issue tracker: <https://github.com/r2dbc/r2dbc-spi/issues>
- Release repository: <https://repo1.maven.org/maven2>
- Snapshot repository: <https://oss.sonatype.org/content/repositories/snapshots>

Chapter 2. Goals

This section outlines the goals for R2DBC and the design philosophy for its SPI. It covers the following topics:

- [Enabling Reactive Relational Database Connectivity](#)
- [Fitting into Reactive JVM platforms](#)
- [Offering Vendor-neutral Access to Standard Features](#)
- [Embracing Vendor-specific Features](#)
- [Keeping the Focus on SQL](#)
- [Keeping It Minimal and Simple](#)
- [Providing a Foundation for Tools and Higher-level APIs](#)
- [Specifying Requirements Unambiguously](#)

2.1. Enabling Reactive Relational Database Connectivity

The R2DBC specification aims for establishing an interface that has a minimal API surface to integrate with relational databases by using a reactive programming model. The most significant goals are honoring and embracing the properties of reactive programming, including the following:

- Non-blocking I/O
- Deferred execution
- Treating application control as a series of events (data, errors, completion, and so on)
- No longer assuming control of resources but leaving resource scheduling to the runtime or platform (“React to resource availability”)
- Efficiently using resources
- Leaving flow control to be handled by the runtime
- Stream-oriented data consumption
- Functional programming within operators
- Removing assumptions over concurrency from the programming model and leaving this aspect up the runtime
- Using back-pressure to allow flow control, deferring the actual execution and not overwhelming consumers

2.2. Fitting into Reactive JVM platforms

R2DBC aims for seamless integration of reactive JVM platforms, targeting Java as its primary platform. R2DBC should also be usable from other platforms (such as Kotlin or Scala) without scarifying its SPI for the sake of idiomatic use on a different platform.

2.3. Offering Vendor-neutral Access to Standard Features

R2DBC SPI strives to provide access to features that are commonly found across different vendor implementations. The goal here is to provide a balance between features that are implemented in a driver and these that are better implemented in a client library.

2.4. Embracing Vendor-specific Features

Each database comes with its own feature set and how these are implemented. R2DBC's goal is to define a minimal standard over commonly used functionality and allow for vendor-specific deviation. Drivers can implement additional functionality or make these transparent through the R2DBC SPI.

2.5. Keeping the Focus on SQL

The focus of R2DBC is on accessing relational data from the Java programming language by using databases that provide an SQL interface with which to interact.

The goal here is not to limit implementations to relational-only databases. Instead, the goal is to provide guidance for uniform reactive data access by using tabular data consumption patterns.

2.6. Keeping It Minimal and Simple

R2DBC does not aim to be a general-purpose data-access API.

R2DBC specializes in reactive data access and common usage patterns that result from relational data interaction. R2DBC does not aim to abstract common functionality that needs to be re-implemented by driver vendors in a similar manner. It aims to leave this functionality to client libraries, of which there are typically fewer implementations than drivers.

2.7. Providing a Foundation for Tools and Higher-level APIs

The R2DBC SPI aims for being primarily consumed through client library implementations.

It does not aim to be an end-user or application developer programming interface.

Having a uniform reactive relational data access SPI makes R2DBC a valuable target platform for tool vendors and application developers who want to create portable tools and applications.

2.8. Specifying Requirements Unambiguously

The requirements for R2DBC compliance should be unambiguous and easy to identify. The R2DBC specification and the API documentation (Javadoc) clarify which features are required and which are optional.

Chapter 3. Compliance

This chapter identifies the required features of a D2DBC driver implementation to claim compliance. Any features not identified here are considered optional.

3.1. Definitions

To avoid ambiguity, we will use the following terms in the compliance section and across this specification:

R2DBC driver implementation

(short form: **driver**) A driver that implements the R2DBC SPI. A driver may provide support for features that are not implemented by the underlying database or expose functionality that is not declared by the R2DBC SPI (See **Extension**).

Supported feature

A feature for which the R2DBC SPI implementation supports standard syntax and semantics.

Partially supported feature

A feature for which some methods are implemented with standard syntax and semantics and some required methods are not implemented (typically covered by **default** interface methods).

Extension

A feature that is not covered by R2DBC or a non-standard implementation of a feature that is covered.

Fully implemented

Term to express that an interface has all its methods implemented to support the semantics defined in this specification.

Must implement

Term to express that an interface must be implemented, although some methods on the interface are considered optional. Methods that are not implemented rely on the **default** implementation.

3.2. Guidelines and Requirements

The following guidelines apply to R2DBC compliance:

- An R2DBC SPI should implement SQL support as its primary interface. R2DBC does not rely upon (nor does it presume) a specific SQL version. SQL and aspects of statements can be entirely handled in the data source or as part of the driver.
- The specification consists of this specification document and the specifications documented in each interface's Javadoc.
- Drivers supporting parametrized statements must support bind parameter markers.
- Drivers supporting parametrized statements must support at least one parameter binding method (indexed or named).
- Drivers must support transactions.
- Index references to columns and parameters are zero-based. That is, the first index begins with **0**.

3.3. R2DBC SPI Compliance

A driver that is compliant with the R2DBC specification must do the following:

- Adhere to the guidelines and requirements listed under [Guidelines and Requirements](#).
- Support `ConnectionFactory` discovery through Java Service Loader of `ConnectionFactoryProvider`.
- Implement a non-blocking I/O layer.
- Fully implement the following interfaces:
 - `io.r2dbc.spi.ConnectionFactory`
 - `io.r2dbc.spi.ConnectionFactoryMetadata`
 - `io.r2dbc.spi.ConnectionFactoryProvider`
 - `io.r2dbc.spi.Result`
 - `io.r2dbc.spi.Row`
 - `io.r2dbc.spi.RowMetadata`
 - `io.r2dbc.spi.Batch`
- Implement the `io.r2dbc.spi.Connection` interface, except for the following optional methods:
 - `createSavepoint(...)`: Calling this method should throw an `UnsupportedOperationException` exception for drivers that do not support savepoints.
 - `releaseSavepoint(...)`: Calling this method should be a no-op for drivers that do not support savepoint release.
 - `rollbackTransactionToSavepoint(...)`: Calling this method should throw an `UnsupportedOperationException` exception for drivers that do not support savepoints.
- Implement the `io.r2dbc.spi.Statement` interface, except for the following optional methods:
 - `returnGeneratedValues(...)`: Calling this method should be a no-op for drivers that do not support key generation.
 - `fetchSize(...)`: Calling this method should be a no-op for drivers that do not support fetch size hints.
- Implement the `io.r2dbc.spi.ColumnMetadata` interface, except for the following optional methods:
 - `getPrecision()`
 - `getScale()`
 - `getNullability()`
 - `getJavaType()`
 - `getNativeTypeMetadata()`

A driver can implement optional [Extensions](#) if it is able to provide extension functionality specified by R2DBC.

Chapter 4. Overview

R2DBC provides an API for Java programs to access one or more sources of data. In the majority of cases, the data source is a relational DBMS and its data is accessed using SQL. R2DBC drivers are not limited to RDBMS but can be implemented on top of other data sources, including stream-oriented systems and object-oriented systems. A primary motivation for R2DBC SPI is to provide a standard API for reactive applications to integrate with a wide variety of data sources.

This chapter gives an overview of the API and the key concepts of the R2DBC SPI. It includes the following topics:

- [Establishing a Connection](#)
- [Using `ConnectionFactory` Discovery](#)
- [R2DBC Connection URL](#)
- [Running SQL and Retrieving Results](#)

4.1. Establishing a Connection

R2DBC uses the `Connection` interface to define a logical connection API to the underlying data source. The structure of a connection depends on the actual requirements of a data source and how the driver implements these.

In a typical scenario, an application that uses R2DBC connects to a target data source by using one of two mechanisms:

- **ConnectionFactories:** R2DBC SPI provides this fully implemented class. It provides `ConnectionFactory` discovery functionality for applications that want to obtain a connection without using a vendor-specific API. When an application first attempts to connect to a data source, `ConnectionFactories` automatically loads any R2DBC driver found on the classpath by using Java's `ServiceLoader` mechanism. See [ConnectionFactory Discovery](#) for the details of how to implement the discovery mechanism for a particular driver.
- **ConnectionFactory:** A `ConnectionFactory` is implemented by a driver and provides access to `Connection` creation. An application that wants to configure vendor-specific aspects of a driver can use the vendor-specific `ConnectionFactory` creation mechanism to configure a `ConnectionFactory`.

4.1.1. Using `ConnectionFactory` Discovery

As mentioned [earlier](#), R2DBC supports the concept of discovery to find an appropriate driver for a connection request. Providing a `ConnectionFactory` to an application is typically a configuration infrastructure task. Applications that wish to bootstrap an R2DBC client typically handle this aspect directly in application code and, so, discovery can become a task for application developers.

`ConnectionFactories` provides two standard mechanisms to bootstrap a `ConnectionFactory`:

- **URL-based:** R2DBC supports a uniform URL-based configuration scheme with a well-defined structure and well-known configuration properties. URLs are represented as Java `String` and

can be passed to `ConnectionFactory`s for `ConnectionFactory` lookup.

- Programmatic: In addition to a URL-based configuration, R2DBC provides a programmatic approach so that applications can supply structured configuration options to obtain a `ConnectionFactory`.

In addition to the two preceding methods, R2DBC embraces a mixed mechanism as typical configuration infrastructure mixes URL- and programmatic-based configuration of data sources for enhanced flexibility. A typical use case is the separation of concerns in which data-source coordinates are supplied by using a URL while login credentials originate from a different configuration source.

4.1.2. R2DBC Connection URL

R2DBC defines a standard URL format that is an enhanced form of [RFC 3986 Uniform Resource Identifier \(URI\): Generic Syntax](#) and its amendments supported by Java's `java.net.URI` type.

The following listing shows the syntax Components from [RFC3986](#):


```

URI      = scheme ":" driver [ ":" protocol ] ":" hier-part [ "?" query ] [
"#" fragment ]

scheme    = "r2dbc" / "r2dbcS"

driver    = ALPHA *( ALPHA )

protocol  = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." / ":" )

hier-part = "//" authority path-abempty
           / path-absolute
           / path-rootless
           / path-empty

authority = [ userinfo "@" ] host [ ":" port ] [ "," host [ ":" port ] ]

userinfo  = *( unreserved / pct-encoded / sub-delims / ":" )

host      = IP-literal / IPv4address / reg-name

port      = *DIGIT

path-abempty = *( "/" segment )
path-absolute = "/" [ segment-nz *( "/" segment ) ]
path-rootless = segment-nz *( "/" segment )
path-empty   = 0<pchar>

segment    = *pchar
segment-nz = 1*pchar
segment-nz-nc = 1*( unreserved / pct-encoded / sub-delims / "@" )
               ; non-zero-length segment without any colon ":"

query      = *( pchar / "/" / "?" )

fragment   = *( pchar / "/" / "?" )

pct-encoded = "%" HEXDIG HEXDIG

pchar      = unreserved / pct-encoded / sub-delims / ":" / "@"

sub-delims = "!" / "$" / "&" / "'" / "(" / ")"
             / "*" / "+" / "," / ";" / "="

unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"

```

Example 1. R2DBC Connection URL

```
r2dbc:a-driver:pipes://localhost:3306/my_database?locale=en_US
```

The diagram illustrates the structure of the R2DBC URL. It shows the URL with dashed lines and backslashes indicating the boundaries between its components: `scheme` (r2dbc), `driver` (a-driver), `protocol` (pipes), `authority` (localhost:3306), `path` (/my_database), and `query` (?locale=en_US).

- **scheme**: Identify that the URL is a valid R2DBC URL. Valid schemes are `r2dbc` and `r2dbcssl` (configure SSL usage).
- **driver**: Identifier for a driver. The specification has no authority over driver identifiers.
- **protocol**: Used as optional protocol information to configure a driver-specific protocol. Protocols can be organized hierarchically and are separated by a colon (:).
- **authority**: Contains an endpoint and authorization. The authority may contain a single host or a collection of hostnames and port tuples by separating these with a comma (,).
- **path**: (optional) Used as an initial schema or database name.
- **query**: (optional) Used to pass additional configuration options in the form of `String` key-value pairs by using the key name as the option name.
- **fragment**: Unused (reserved for future use).

`ConnectionFactoryOptions.parse(String)` parses a R2DBC URL into `ConnectionFactoryOptions` using standard and optional extended options. A R2DBC Connection URL is parsed into the following options (by using `ConnectionFactoryOptions` constants):

The following listing shows an example URL:

Example 2. R2DBC Connection URL

```
r2dbc:a-driver:pipes://hello:world@localhost:3306/my_database?locale=en_US
```

The following table describes the standard options:

Table 1. Parsed Standard Options

Option	URL Part	Value as per Example
<code>ConnectionFactoryOptions.SSL</code>	<code>r2dbc</code>	Unconfigured.
<code>ConnectionFactoryOptions.DRIVER</code>	<code>driver</code>	<code>a-driver</code>
<code>ConnectionFactoryOptions.PROTOCOL</code>	<code>protocol</code>	<code>pipes</code>
<code>ConnectionFactoryOptions.USER</code>	User-part of <code>authority</code>	<code>hello</code>

Option	URL Part	Value as per Example
<code>ConnectionFactoryOptions.PASSWORD</code>	Password-part of <code>authority</code>	<code>world</code>
<code>ConnectionFactoryOptions.HOST</code>	Host-part of <code>authority</code>	<code>localhost</code>
<code>ConnectionFactoryOptions.PORT</code>	Port-part of <code>authority</code>	<code>3306</code>
<code>ConnectionFactoryOptions.DATABASE</code>	<code>path</code> without the leading <code>/</code>	<code>my_database</code>

The following table describes the extended options:

Table 2. Parsed Extended Options

Option	URL Part	Value as per Example
<code>locale</code>	key-value tuple from <code>query</code>	<code>en_US</code>



R2DBC defines well-known standard options that are available as runtime constants through `ConnectionFactoryOptions`. Additional options identifiers are created through `Option.valueOf(...)`.



Note that Connection URL Parsing cannot access `Option` type information `T` due to Java's type erasure. Options configured by URL parsing are represented as `String` values.

Example 3. Obtaining a `ConnectionFactory` using R2DBC URL

```
ConnectionFactory factory = ConnectionFactories.get("r2dbc:a-
driver:pipes://localhost:3306/my_database?locale=en_US");
```

Example 4. Obtaining a `ConnectionFactory` using `ConnectionFactoryOptions`

```
ConnectionFactoryOptions options = ConnectionFactoryOptions.builder()
    .option(ConnectionFactoryOptions.DRIVER, "a-driver")
    .option(ConnectionFactoryOptions.PROTOCOL, "pipes")
    .option(ConnectionFactoryOptions.HOST, "localhost")
    .option(ConnectionFactoryOptions.PORT, 3306)
    .option(ConnectionFactoryOptions.DATABASE, "my_database")
    .option(Option.valueOf("locale"), "en_US")
    .build();

ConnectionFactory factory = ConnectionFactories.get(options);
```

4.2. Running SQL and Retrieving Results

Once a connection has been established, an application using the R2DBC SPI can execute queries and updates against the connected database. The R2DBC SPI provides a text-based command interface to the most commonly used features of SQL databases. R2DBC driver implementations may expose additional functionality in a non-standard way.

Applications use methods in the `Connection` interface to specify transaction attributes and create `Statement` or `Batch` objects. These statements are used to execute SQL and retrieve results and allow for binding values to parameter bind markers. The `Result` interface encapsulates the results of an SQL query. Statements may also be batched, allowing an application to submit multiple commands to a database as a single unit of execution.

Chapter 5. Connections

R2DBC uses the `Connection` interface to define a logical connection API to the underlying data source. The structure of a connection depends on the actual requirements of the data source and how the driver implements these.

The data source can be an RDBMS, a stream-oriented data system, or some other source of data with a corresponding R2DBC driver. A single application that uses R2DBC SPI can maintain multiple connections to either a single data source or across multiple data sources. From a R2DBC driver perspective, a `Connection` object represents a single client session. It has associated state information, such as user ID and what transaction semantics are in effect. A `Connection` object is not safe for concurrent state-changing by multiple subscribers. A connection object can be shared across multiple threads that serially run operations by using appropriate synchronization mechanisms.

To obtain a connection, the application can:

- Interact with the `ConnectionFactoryies` class by working with one or more `ConnectionFactoryProvider` implementations.
- Directly interact with a `ConnectionFactory` implementation.

See [Establishing a Connection](#) for more details.

5.1. The `ConnectionFactory` Interface

R2DBC drivers must implement the `ConnectionFactory` interface as a mandatory part of the SPI. Drivers can provide multiple `ConnectionFactory` implementations, depending on the protocol in use or aspects that require the use of a different `ConnectionFactory` implementation. The following listing shows the `ConnectionFactory` interface:

Example 5. `ConnectionFactory` Interface

```
public interface ConnectionFactory {  
  
    Publisher<? extends Connection> create();  
  
    ConnectionFactoryMetadata getMetadata();  
  
}
```

The following rules apply:

- A `ConnectionFactory` represents a resource factory for deferred connection creation. It may create connections by itself, wrap a `ConnectionFactory`, or apply connection pooling on top of a `ConnectionFactory`.
- A `ConnectionFactory` provides metadata about the driver itself through

`ConnectionFactoryMetadata`.

- A `ConnectionFactory` uses deferred initialization and should initiate connection resource allocation after requesting the item (`Subscription.request(1)`).
- Connection creation must emit exactly one `Connection` or an error signal.
- Connection creation must be cancellable (`Subscription.cancel()`). Canceling connection creation must release (“close”) the connection and all associated resources.
- A `ConnectionFactory` should expect that it can be wrapped. Wrappers must implement the `Wrapped<ConnectionFactory>` interface and return the underlying `ConnectionFactory` when `Wrapped.unwrap()` gets called.

5.1.1. ConnectionFactory Metadata

`ConnectionFactory` instances are required to expose metadata to identify the driver (`ConnectionFactory`) and its capabilities. Metadata must not require a connection to a data source. The following listing shows the `ConnectionFactoryMetadata` interface:

Example 6. ConnectionFactoryMetadata Interface

```
public interface ConnectionFactoryMetadata {  
  
    String getName();  
  
}
```

See the R2DBC SPI Specification for more details.

5.2. ConnectionFactory Discovery Mechanism

As part of its usage, the `ConnectionFactories` class tries to load any R2DBC driver classes referenced by the `ConnectionFactoryProvider` interface listed in the Java Service Provider manifests that are available on the classpath.

Drivers must include a file called `META-INF/services/io.r2dbc.spi.ConnectionFactoryProvider`. This file contains the name of the R2DBC driver’s implementation (or implementations) of `io.r2dbc.spi.ConnectionFactoryProvider`. To ensure that drivers can be loaded by using this mechanism, `io.r2dbc.spi.ConnectionFactoryProvider` implementations are required to provide a no-argument constructor. The following listing shows a typical `META-INF/services/io.r2dbc.spi.ConnectionFactoryProvider` file:

Example 7. META-INF/services/io.r2dbc.spi.ConnectionFactoryProvider file contents

```
com.example.ConnectionFactoryProvider
```

The following listing shows the `ConnectionFactoryProvider` interface:

Example 8. `ConnectionFactoryProvider` Interface

```
public interface ConnectionFactoryProvider {  
  
    ConnectionFactory create(ConnectionFactoryOptions connectionFactoryOptions);  
  
    boolean supports(ConnectionFactoryOptions connectionFactoryOptions);  
  
    String getDriver();  
  
}
```

`ConnectionFactories` uses a `ConnectionFactoryOptions` object to look up a matching driver by using a two-step model:

1. Look up an adequate `ConnectionFactoryProvider`.
2. Obtain the `ConnectionFactory` from the `ConnectionFactoryProvider`.

`ConnectionFactoryProvider` implementations are required to return a `boolean` indicator whether or not they support a specific configuration represented by `ConnectionFactoryOptions`. Drivers must expect any plurality of `Option` instances to be configured. Drivers must report that they support a configuration only if the `ConnectionFactoryProvider` can provide a `ConnectionFactory` based on the given `ConnectionFactoryOptions`. A typical task handled by `supports` is checking driver and protocol options. Drivers should gracefully fail if a `ConnectionFactory` creation through `ConnectionFactoryProvider.create(...)` is not possible (i.e. when required options were left unconfigured). The `getDriver()` method reports the driver identifier that is associated with the `ConnectionFactoryProvider` implementation to provide diagnostic information to users in misconfiguration cases.

See the R2DBC SPI Specification and [ConnectionFactory Discovery](#) for more details.

5.3. The `ConnectionFactoryOptions` Class

The `ConnectionFactoryOptions` class represents a configuration for a request a `ConnectionFactory` from a `ConnectionFactoryProvider`. It enables the [programmatic connection creation](#) approach without using driver-specific classes. `ConnectionFactoryOptions` instances are created by using the builder pattern, and properties are configured through `Option<T>` identifiers. A `ConnectionFactoryOptions` is immutable once created. `Option` objects are reused as part of the built-in constant pool. Options are identified by a literal.

`ConnectionFactoryOptions` defines a set of well-known options:

Table 3. Well-known Options

Constant	URL Literal	Type	Description
SSL	ssl	java.lang.Boolean	Whether the connection is configured to require SSL.
DRIVER	driver	java.lang.String	Driver identifier.
PROTOCOL	protocol	java.lang.String	Protocol details, such as the network protocol used to communicate with a server.
USER	user	java.lang.String	User account name.
PASSWORD	password	java.lang.CharSequence	User or database password.
HOST	host	java.lang.String	Database server name.
PORT	port	java.lang.Integer	Database server port number.
DATABASE	database	java.lang.String	Name of the particular database on a server.
CONNECT_TIMEOUT	connectTimeout	java.time.Duration	Connection timeout to obtain a connection.

The following rules apply:

- The set of options is extensible.
- Drivers can declare which well-known options they require and which they support.
- Drivers can declare which extended options they require and which they support.
- Drivers should not fail in creating a connection if more options are declared than the driver consumes, as a `ConnectionFactory` should expect to be wrapped.
- Connection URL Parsing cannot access `Option` type information `T` due to Java's type erasure. Options obtained by URL parsing beyond well-known keys are represented as `String` values.

The following example shows how to set options for a `ConnectionFactoryOptions`:

Example 9. Configuration of `ConnectionFactoryOptions`

```
ConnectionFactoryOptions options = ConnectionFactoryOptions.builder()
    .option(ConnectionFactoryOptions.HOST, "...")
    .option(Option.valueOf("tenant"), "...")
    .option(Option.sensitiveValueOf("encryptionKey"), "...")
    .build();
```

See the R2DBC SPI Specification for more details.

5.4. Obtaining **Connection** Objects

Once a **ConnectionFactory** is bootstrapped, connections are obtained from the `create()` method. The following example shows how to obtain a connection:

*Example 10. Obtaining a **Connection***

```
// factory is a ConnectionFactory object
Publisher<? extends Connection> publisher = factory.create();
```

The connection is active once it has been emitted by the **Publisher** and must be released (“closed”) once it is no longer in use.

5.5. Connection Metadata

Connections are required to expose metadata about the database they are connected to. Connection Metadata is typically discovered dynamically based from information obtained during **Connection** initialization.

*Example 11. **ConnectionMetadata** Interface*

```
public interface ConnectionMetadata {

    String getDatabaseProductName();

    String getDatabaseVersion();

}
```

See the R2DBC SPI Specification for more details.

5.6. Validating **Connection** Objects

The `Connection.validate(...)` method indicates whether the **Connection** is still valid. The **ValidationDepth** argument passed to this method indicates the depth to which a connection should be validated: **LOCAL** or **REMOTE**.

- **ValidationDepth.LOCAL**: Requests client-side-only validation without engaging a remote conversation to validate a connection.
- **ValidationDepth.REMOTE**: Initiates a remote validation by issuing a query or other means to validate a connection and the remote session.

If `Connection.validate(...)` emits `true`, the **Connection** is still valid. If `Connection.validate(...)` emits `false`, the **Connection** is not valid, and any attempt to perform database interaction fails. Callers of

this method do not expect error signals or empty completion.

5.7. Closing **Connection** Objects

Calling **Connection.close()** prepares a close handle to release the connection and its associated resources. Connections must be closed to ensure proper resource disposal. You can use **Connection.validate(...)** to determine whether a **Connection** has been closed or is still valid. The following example shows how to close a connection:

*Example 12. Closing a **Connection***

```
// connection is a ConnectionFactory object
Publisher<Void> close = connection.close();
```

See the R2DBC SPI Specification for more details.

Chapter 6. Transactions

Transactions are used to provide data integrity, isolation, correct application semantics, and a consistent view of data during concurrent database access. All R2DBC-compliant drivers are required to provide transaction support. Transaction management in the R2DBC SPI reflects SQL concepts:

- Auto-commit mode
- Transaction isolation levels
- Savepoints

This section explains transaction semantics associated with a single `Connection` object.

6.1. Transaction Boundaries

You can implicitly or explicitly start transactions. You can implicitly start a transaction by starting SQL execution when a `Connection` is in auto-commit mode (which is the default for newly created connections). When auto-commit mode is disabled, you can explicitly start a transaction by invoking the `beginTransaction()` method. Transactions are started by either an R2DBC driver or by the underlying database.

The `Connection` attribute `auto-commit` mode specifies when to end transactions. Enabling auto-commit mode causes a transaction commit after each SQL statement as soon as that statement is completely executed.

6.2. Auto-commit Mode

A `ConnectionFactory` creates new `Connection` objects with auto-commit mode enabled, unless specified otherwise through connection configuration options. The `Connection` interface provides two methods to interact with auto-commit mode:

- `setAutoCommit`
- `isAutoCommit`

R2DBC applications should change auto-commit mode by invoking the `setAutoCommit` method instead of executing SQL commands to change the connection configuration. If the value of auto-commit is changed during an active transaction, the current transaction is committed. If `setAutoCommit` is called and the value for auto-commit is not changed from its current value, this is treated as a no-op.

Changing auto-commit mode typically engages database activity. Therefore, the method returns a `Publisher`. Querying auto-commit mode is typically a local operation that involves driver state without database communication.

When auto-commit is disabled, you must explicitly start and clean up each transaction by calling the `Connection` methods `beginTransaction` and `commitTransaction` (or `rollbackTransaction`), respectively.

This is appropriate for cases where transaction management is being done in a layer above the driver, such as:

- The application needs to group multiple SQL statements into a single transaction.
- An application container manages the transaction state.

6.3. Transaction Isolation

Transaction isolation levels define the level of visibility (“isolation”) for statements that are run within a transaction. They impact concurrent access while multiple transactions are active.

The default transaction level for a `Connection` object is vendor-specific and determined by the driver that supplied the connection. Typically, it defaults to the transaction level supported by the underlying data source.

The `Connection` interface provides two methods to interact with transaction isolation levels:

- `setTransactionIsolationLevel`
- `getTransactionIsolationLevel`

R2DBC applications should change transaction isolation levels by invoking the `setTransactionIsolationLevel` method instead of running SQL commands to change the connection configuration.

Changing transaction isolation levels typically involves database activity. Therefore, the method returns a `Publisher`. Changing an isolation level during an active transaction results in implementation-specific behavior. Querying transaction isolation levels is typically a local operation that involves driver state without database communication. The return value of the `getTransactionIsolationLevel` method should reflect the current isolation level when it actually occurs. `IsolationLevel` is an extensible runtime constant, so drivers can define their own isolation levels. A driver may not support transaction levels. Calling `getTransactionIsolationLevel` results in returning the vendor-specific `IsolationLevel` object.

6.3.1. Performance Considerations

When you increase the transaction isolation level, databases typically require more locking and resource overhead to ensure isolation level semantics. This, in turn, lowers the degree of concurrent access that can be supported. As a result, applications may see degraded performance when they use higher transaction isolation levels. For this reason, a transaction manager, whether it is the application itself or part of the application container, should weigh the need for data consistency against the requirements for performance when determining which transaction isolation level is appropriate.

6.4. Savepoints

Savepoints provide a fine-grained control mechanism by marking intermediate points within a transaction. Once a savepoint has been created, a transaction can be rolled back to that savepoint without affecting preceding work.

6.4.1. Working with Savepoints

The `Connection` interface defines methods to interact with savepoints:

- `createSavepoint`
- `releaseSavepoint`
- `rollbackTransactionToSavepoint`

Savepoints are created during an active transaction and are valid only as long as the transaction is active. You can use the `createSavepoint` method to set a savepoint within the current transaction. A transaction is started if `createSavepoint` is invoked and there is no active transaction (switching the connection to disabled auto-commit mode). The `rollbackTransactionToSavepoint` method is used to roll back work to a previous savepoint without rolling back the entire transaction. the following example shows how to roll back a transaction to a savepoint:

Example 13. Rolling back a transaction to a savepoint

```
// connection is a Connection object
Publisher<Void> begin = connection.beginTransaction();

Publisher<? extends Result> insert1 = connection.createStatement("INSERT INTO
books VALUES ('John Doe')").execute();

Publisher<Void> savepoint = connection.createSavepoint("savepoint");

Publisher<? extends Result> insert2 = connection.createStatement("INSERT INTO
books VALUES ('Jane Doe')").execute();

...

Publisher<Void> partialRollback =
connection.rollbackTransactionToSavepoint("savepoint");

...

Publisher<Void> commit = connection.commit();

// publishers are materialized in the order: begin, insert1, savepoint, insert2,
partialRollback, commit
```

Drivers that do not support savepoint creation and rolling back to a savepoint should throw an `UnsupportedOperationException` to indicate these features are not supported.

6.4.2. Releasing a Savepoint

Savepoints allocate resources on the databases, and some vendors may require releasing a savepoint to dispose resources. The `Connection` interface defines the `releaseSavepoint` method to release savepoints that are no longer needed.

Savepoints that were created during a transaction are released and are invalidated when the transaction is committed or when the entire transaction is rolled back. Rolling a transaction back to a savepoint automatically releases it. A rollback also invalidates any other savepoints that were created after the savepoint in question.

Calling `releaseSavepoint` for drivers that do not support savepoint release results in a no-op.

Chapter 7. Statements

This section describes the `Statement` interface. It also describes related topics, including parameterized statement and auto-generated keys.

7.1. The Statement Interface

The `Statement` interface defines methods for running SQL statements. SQL statements may contain parameter bind markers for input parameters.

7.1.1. Creating Statements

`Statement` objects are created by `Connection` objects, as the following example shows:

Example 14. Creating a non-parameterized `Statement`

```
// connection is a Connection object
Statement statement = connection.createStatement("SELECT title FROM books");
```

Each `Connection` object can create multiple `Statement` objects that the program can concurrently run at any time. Resources that are associated with a statement are released as soon as the connection is closed.

7.1.2. Running Statement Objects

`Statement` objects are run by calling the `execute()` method. Depending on the SQL, the resulting `Publisher` may return one or many `Result` objects. A `Statement` is always associated with its `Connection`. Therefore, the connection state affects `Statement` execution at execution time. The following example shows how to run a statement:

Example 15. Running a `Statement`

```
// statement is a Statement object
Publisher<? extends Result> publisher = statement.execute();
```

7.2. Parameterized Statements

The SQL that is used to create a statement can be parameterized by using vendor-specific bind markers. The portability of SQL statements across R2DBC implementations is not a goal.

Parameterized `Statement` objects are created by `Connection` objects in the same manner as non-parameterized `Statements`. See the the following example:

*Example 16. Creating three parameterized **Statement** objects by using vendor-specific parameter bind markers*

```
// connection is a Connection object
Statement statement1 = connection.createStatement("SELECT title FROM books WHERE
author = :author");

Statement statement2 = connection.createStatement("SELECT title FROM books WHERE
author = @P0");

Statement statement3 = connection.createStatement("SELECT title FROM books WHERE
author = $1");
```

Parameter bind markers are identified by the **Statement** object. Parameterized statements may be cached by R2DBC implementations for reuse (for example, for prepared statement execution).

7.2.1. Binding Parameters

The **Statement** interface defines **bind(...)** and **bindNull(...)** methods to provide parameter values for bind marker substitution. The parameter type is defined by the actual value that is bound to a parameter. Each bind method accepts two arguments. The first is either an ordinal position parameter starting at 0 (zero) or the parameter placeholder representation. The method of parameter binding (positional or by identifier) is vendor-specific, and a driver should document its preferred binding mechanism. The second and any remaining parameters specify the value to be assigned to the parameter. The following example shows how to bind parameters to a statement object by using placeholders:

*Example 17. Binding parameters to a **Statement** object by using placeholders*

```
// connection is a Connection object
Statement statement = connection.createStatement("SELECT title FROM books WHERE
author = $1 and publisher = $2");
statement.bind("$1", "John Doe");
statement.bind("$2", "Happy Books LLC");
```

Alternatively, parameters can be bound by index, as the following example shows:

*Example 18. Binding parameters to a **Statement** object by index*

```
// connection is a Connection object
Statement statement = connection.createStatement("SELECT title FROM books WHERE
author = $1 and publisher = $2");
statement.bind(0, "John Doe");
statement.bind(1, "Happy Books LLC");
```


A value must be provided for each bind marker in the `Statement` object before the statement can be run. The `execute` method validates a parameterized `Statement` and throws an `IllegalStateException` if a bind marker is left without a binding.

7.2.2. Batching

Parameterized `Statement` objects accept multiple parameter binding sets to submit a batch of commands to the database for running. A batch run is initiated by invoking the `add()` method on the `Statement` object after providing all parameters. After calling `add()`, the next set of parameter bindings is provided by calling bind methods accordingly. The following example shows how to run a batch `Statement`:

Example 19. Running a `Statement` batch

```
// connection is a Connection object
Statement statement = connection.createStatement("INSERT INTO books (author,
publisher) VALUES ($1, $2)");
statement.bind(0, "John Doe").bind(1, "Happy Books LLC").add();
statement.bind(0, "Jane Doe").bind(1, "Scary Books Inc");
Publisher<? extends Result> publisher = statement.execute();
```

A batch run emits one or many `Result` objects, depending on how the implementation executes the batch.

7.2.3. Setting `NULL` Parameters

You can use the `bindNull` method to set any parameter to `NULL`. It takes two parameters:

- Either the ordinal position of the bind marker or the name.
- The value type of the parameter.

The following example shows how to set `NULL` value:

Example 20. Setting a `NULL` value.

```
// statement is a Statement object
statement.bindNull(0, String.class);
```

7.3. Retrieving Auto Generated Values

Many database systems provide a mechanism that automatically generates a value when a row is inserted. The value that is generated may or may not be unique or represent a key value, depending on the SQL and the table definition. You can call the `returnGeneratedValues` method to retrieve the generated value. It tells the `Statement` object to retrieve generated values. The method accepts a variable-argument parameter to specify the column names for which to return generated keys. The

emitted **Result** exposes a column for each automatically generated value (taking the column name hint into account). The following example shows how to retrieve auto-generated values:

Example 21. Retrieving auto-generated values

```
// connection is a Connection object
Statement statement = connection.createStatement("INSERT INTO books (author,
publisher) VALUES ('John Doe', 'Happy Books LLC')").returnGeneratedValues("id");
Publisher<? extends Result> publisher = statement.execute();

// later
result.map((row, metadata) -> row.get("id"));
```

When column names are not specified, the R2DBC driver implementation determines the columns or value to return.

See the R2DBC SPI Specification for more details.

7.4. Performance Hints

The **Statement** interface provides a method that you can use to provide hints to a R2DBC driver. Calling **fetchSize** applies a fetch-size hint to each query produced by the statement. Hints provided to the driver through this interface may be ignored by the driver if they are not appropriate or supported.

Back-pressure hints can be used by drivers to derive an appropriate fetch size. To optimize for performance, it can be useful to provide hints to the driver on a per-statement basis to avoid unwanted interference of back-pressure hint propagation.

Note that back-pressure should be considered a utility for flow control and not to limit the result size. Result size limitations should be part of the query statement.

Chapter 8. Batches

This section describes the **Batch** interface.

8.1. The Batch Interface

The **Batch** interface defines methods for running groups of SQL statements. SQL statements may not contain parameter bind markers for input parameters. A batch is created to run multiple SQL statements for performance reasons.

8.1.1. Creating Batches

Batch objects are created by **Connection** objects, as the following example shows:

*Example 22. Creating a **Batch***

```
// connection is a Connection object
Batch batch = connection.createBatch();
```

Each **Connection** object can create multiple **Batch** objects that can be used concurrently by the program and can be run at any time. Resources that are associated with a batch are released as soon as the connection is closed.

8.1.2. Executing Batch Objects

Batch objects are run by calling the **execute()** method after adding one or more SQL statements to a **Batch**. The resulting **Publisher** returns a **Result** object for each statement in the batch. A **Batch** is always associated with its **Connection**. Therefore, the connection state affects **Batch** execution at run time.

The following example shows how to run a batch:

*Example 23. Running a **Batch***

```
// connection is a Connection object
Batch batch = connection.createBatch();
Publisher<? extends Result> publisher = batch.add("SELECT title, author FROM
books")
    .add("INSERT INTO books VALUES('John Doe', 'HappyBooks LLC')")
    .execute();
```

See the R2DBC SPI Specification for more details.

Chapter 9. Results

This section explains the `Result` interface and the related `Row` interface. It also describes related topics, including result consumption.

9.1. Result Characteristics

`Result` objects are forward-only and read-only objects that allow consumption of two result types:

- Tabular results
- Update count

Results move forward from the first `Row` to the last one. After emitting the last row, a `Result` object gets invalidated and rows from the same `Result` object can no longer be consumed. Rows contained in the result depend on how the underlying database materializes the results. That is, it contains the rows that satisfy the query at either the time the query is run or as the rows are retrieved. An R2DBC driver can obtain a `Result` either directly or by using cursors.

`Result` reports the number of rows affected for SQL statements, such as updates for SQL Data Manipulation Language (DML) statements. The update count can be empty for statements that do not modify rows. After emitting the update count, a `Result` object gets invalidated and rows from the same `Result` object can no longer be consumed. The following example shows how to get a count of updated rows:

Example 24. Consuming an update count

```
// result is a Result object
Publisher<Integer> rowsUpdated = result.getRowsUpdated();
```

The streaming nature of a result allows consumption of either tabular results or an update count. Depending on how the underlying database materializes results, an R2DBC driver can lift this limitation.

A `Result` object is emitted for each statement result in a forward-only direction.

9.2. Creating **Result** Objects

A **Result** object is most often created as the result of running a **Statement** object. The **Statement.execute()** method returns a **Publisher** that emits a **Result** objects as the result of running the statement. The following example shows how to create a **Result** object:

*Example 25. Creating a **Result** object*

```
// connection is a Connection object
Statement statement = connection.createStatement("SELECT title, author FROM
books");
Publisher<? extends Result> results = statement.execute();
```

The **Result** object emits a **Row** object for each row in the **books** table (which contains two columns: **title** and **author**). The following sections detail how these rows and columns can be consumed.

9.2.1. Cursor Movement

Result objects can be backed by direct results (that is, a query that returns results directly) or by cursors. By consuming **Row** objects, an R2DBC driver advances the cursor position. Thus, external cursor navigation is not possible.

Canceling subscription of tabular results stops cursor reads and releases any resources associated with the **Result** object.

9.3. Rows

A **Row** object represents a single row of tabular results.

9.3.1. Retrieving Values

The **Result** interface provides a `map(...)` method for retrieving values from **Row** objects. The `map` method accepts a **BiFunction** (also referred to as mapping function) object that accepts **Row** and **RowMetadata**. The mapping function is called upon row emission with **Row** and **RowMetadata** objects. A **Row** is only valid during the mapping function callback and is invalid outside of the mapping function callback. Thus, **Row** objects must be entirely consumed by the mapping function.

The **Column and Row Metadata** section contains additional details on metadata.

9.4. Interface Methods

The following methods are available on the **Row** interface:

- `Object get(int)`
- `Object get(String)`
- `<T> T get(int, Class<T>)`
- `<T> T get(String, Class<T>)`

`get(int[, Class])` methods accept column indexes starting at 0, `get(String[, Class])` methods accept column name aliases as they are represented in the result. Column names used as input to the `get` methods are case insensitive. Column names do not necessarily reflect the column names as they are in the underlying tables but, rather, how columns are represented (for example, aliased) in the result. The following example shows how to create and consume a **Row** by using its index:

*Example 26. Creating and Consuming a **Row** using its index*

```
// result is a Result object
Publisher<Object> values = result.map((row, rowMetadata) -> row.get(0));
```

The following example shows how to create and consume a **Row** by using its column name:

*Example 27. Creating and Consuming a **Row** by using its column name*

```
// result is a Result object
Publisher<Object> titles = result.map((row, rowMetadata) -> row.get("title"));
```

Calling `get` without specifying a target type returns a suitable value representation according to **Mapping of Data Types**. When you specify a target type, the R2DBC driver tries to convert the value to the target type. The following example shows how to create and consume a **Row** with type

conversion:

*Example 28. Creating and Consuming a **Row** with type conversion*

```
// result is a Result object
Publisher<String> values = result.map((row, rowMetadata) -> row.get(0,
String.class));
```

You can also consume multiple columns from a **Row**, as the following example shows:

*Example 29. Consuming multiple columns from a **Row***

```
// result is a Result object
Publisher<Book> values = result.map((row, rowMetadata) -> {
    String title = row.get("title", String.class);
    String author = row.get("author", String.class);

    return new Book(title, author);
});
```

When the column value in the database is SQL **NULL**, it can be returned to the Java application as **null**.



null values cannot be returned as Reactive Streams values and must be wrapped for subsequent usage.



Invalidating a **Row** does **not** release **Blob** and **Clob** objects that were obtained from the **Row**. These objects remain valid for at least the duration of the transaction in which they were created, unless their **discard()** method is called.

Chapter 10. Column and Row Metadata

The `RowMetadata` interface is implemented by R2DBC drivers to provide information about tabular results. It is used primarily by libraries and applications to determine the properties of a row and its columns.

In cases where the result properties of an SQL statement are unknown until it is run, the `RowMetadata` can be used to determine the actual properties of a row.

`RowMetadata` exposes `ColumnMetadata` for each column in the result. Drivers should provide `ColumnMetadata` on a best-effort basis. Column metadata is typically a by-product of statement execution. The amount of available information is vendor-dependent. Metadata retrieval can require additional lookups (internal queries) to provide a complete metadata descriptor. Issuing queries during result processing conflicts with the streaming nature of R2DBC. Consequently, `ColumnMetadata` declares two sets of methods: methods that must be implemented and methods that can optionally be implemented by drivers.

10.1. Obtaining a `RowMetadata` Object

A `RowMetadata` object is created during tabular results consumption through `Result.map(...)`. It is created for each row. The following example illustrates retrieval and usage by using an anonymous inner class:

Example 30. Using `RowMetadata` and retrieving `ColumnMetadata`

```
// result is a Result object
result.map(new BiFunction<Row, RowMetadata, Object>() {

    @Override
    public Object apply(Row row, RowMetadata rowMetadata) {
        ColumnMetadata my_column = rowMetadata.getColumnMetadata("my_column");
        ColumnMetadata.Nullability nullability = my_column.getNullability();
        // ...
    }
})
```

10.2. Retrieving `ColumnMetadata`

`RowMetadata` methods are used to retrieve metadata for a single column or all columns.

- `getColumnMetadata(int)` returns the `ColumnMetadata` by using a zero-based index. See [Guidelines and Requirements](#).
- `getColumnMetadata(String)` returns the `ColumnMetadata` by using the column name.
- `getColumnMetadatas()` returns an unmodifiable collection of `ColumnMetadata` objects.

10.3. Retrieving General Information for a Column

`ColumnMetadata` declares methods to access column metadata on a best-effort basis. Column metadata that is available as a by-product of running a statement must be made available through `ColumnMetadata`. Metadata exposure requiring interaction with the database (for example, issuing queries to information schema entities to resolve type properties) should not be exposed, because methods on `ColumnMetadata` are expected to be non-blocking.



Implementation note: Drivers can use metadata from a static mapping or obtain metadata indexes on connection creation.

The following example shows how to consume `ColumnMetadata` by using lambdas:

Example 31. Retrieving `ColumnMetadata` information

```
// row is a RowMetadata object
row.getColumnMetadatas().forEach(columnMetadata -> {

    String name = columnMetadata.getName();
    Integer precision = columnMetadata.getPrecision();
    Integer scale = columnMetadata.getScale();
});
```

See the API specification for more details.

Chapter 11. Exceptions

This section explains how R2DBC uses and declares exceptions to provide information about various types of failures.

An exception is thrown by a driver when an error occurs during interaction with the driver or a data source. R2DBC differentiates between generic and data-source-specific error cases.

11.1. General Exceptions

R2DBC defines the following general exceptions:

- `IllegalArgumentException`
- `IllegalStateException`
- `UnsupportedOperationException`
- `R2dbcException`

11.1.1. `IllegalArgumentException`

Drivers throw `IllegalArgumentException` if a method has been received an illegal or inappropriate argument (such as values that are out of bounds or an expected parameter is `null`). This exception is a generic exception that is not associated with an error code or an `SQLState`.

11.1.2. `IllegalStateException`

Drivers throw `IllegalStateException` if a method has received an argument that is invalid in the current state or when an argument-less method is invoked in a state that does not allow execution in the current state (such as interacting with a closed connection object). This exception is a generic exception that is not associated with an error code or an `SQLState`.

11.1.3. `UnsupportedOperationException`

Drivers throw `UnsupportedOperationException` if the driver does not support certain functionality (such as when a method implementation cannot be provided). This exception is a generic exception that is not associated with an error code or an `SQLState`.

11.1.4. `R2dbcException`

Drivers throw an instance of `R2dbcException` when an error occurs during an interaction with a data source.

The exception contains the following information:

- A textual description of the error. You can retrieve the `String` that contains the description by invoking `R2dbcException.getMessage()`. Drivers may provide a localized message variant.
- An `SQLState`. The `String` that contains the `SQLState` can be retrieved by calling the `R2dbcException.getSqlState()` method. The value of the `SQLState` string depends on the

underlying data source.

- An error code. The code is an integer value that identifies the error that caused the `R2dbcException` to be thrown. Its value and meaning are implementation-specific and may be the actual error code returned by the underlying data source. You can retrieve the error code by using the `R2dbcException.getErrorCode()` method.
- A cause. This is another `Throwable` that caused this `R2dbcException` to occur.

11.2. Categorized Exceptions

Categorized exceptions provide a standard mapping to common error states. An R2DBC driver should provide specific subclasses to indicate affinity with the driver. Categorized exceptions provide a standardized approach for R2DBC clients and R2DBC users to translate common exceptions into an application-specific state without the need to implement `SQLState`-based exception translation, resulting in more portable error-handling code.

R2DBC categorizes exceptions into two top-level categories:

- [Non-Transient Exceptions](#)
- [Transient Exceptions](#)

11.2.1. Non-Transient Exceptions

A non-transient exception must extend the abstract class, `R2dbcNonTransientException`. A non-transient exception is thrown when a retry of the same operation would fail unless the cause of the error is corrected. After a non-transient exception other than `R2dbcNonTransientResourceException`, the application can assume that a connection is still valid.

R2DBC defines the following subclasses of non-transient exceptions:

- `R2dbcBadGrammarException`: Thrown when the SQL statement has a problem in its syntax.
- `R2dbcDataIntegrityViolationException`: Thrown when an attempt to insert or update data results in a violation of an integrity constraint.
- `R2dbcPermissionDeniedException`: Thrown when the underlying resource denied a permission to access a specific element, such as a specific database table.
- `R2dbcNonTransientException`: Thrown when a resource fails completely and the failure is permanent. A connection may not be considered valid if this exception is thrown.

11.2.2. Transient Exceptions

A transient exception must extend the abstract class, `R2dbcTransientException`. A transient exception is thrown when a previously failed operation might be able to succeed if the operation is retried without any intervention in application-level functionality. After a non-transient exception other than `R2dbcTransientResourceException`, the application may assume that a connection is still valid.

- `R2dbcRollbackException`: Thrown when an attempt to commit a transaction resulted in an unexpected rollback due to deadlock or transaction serialization failures.

- **R2dbcTimeoutException**: Thrown when the timeout specified by a database operation (query, login, and so on) is exceeded. This could have different causes (depending on the database API in use) but is most likely thrown after the database interrupts or stops the processing of a query before it has completed.
- **R2dbcNonTransientException**: Thrown when a resource fails temporarily and the operation can be retried. A connection may not be considered valid if this exception is thrown.

Chapter 12. Data Types

This chapter discusses the use of data types from Java and database perspectives. The R2DBC SPI gives applications access to data types that are defined as SQL. R2DBC is not limited to SQL types, and, in fact, the SPI is type-agnostic.

If a data source does not support a data type described in this chapter, a driver for that data source is not required to implement the methods and interfaces associated with that data type.

12.1. Mapping of Data Types

This section explains how SQL-specific types are mapped to Java types. The list is not exhaustive and should be received as a guideline for drivers. R2DBC drivers should use modern types and type descriptors to exchange data for consumption by applications and consumption by the driver. Driver implementations should implement the following type mapping and can support additional type mappings:

- [Character Types](#)
- [Boolean Types](#)
- [Binary Types](#)
- [Numeric Types](#)
- [Datetime Types](#)
- [Collection Types](#)

The following table describes the SQL type mapping for character types:

Table 4. SQL Type Mapping for Character Types

SQL Type	Description	Java Type
CHARACTER (CHAR)	Character string, fixed length.	<code>java.lang.String</code>
CHARACTER VARYING (VARCHAR)	Variable-length character string, maximum length fixed.	<code>java.lang.String</code>
NATIONAL CHARACTER (NCHAR)	The NATIONAL CHARACTER type is the same as CHARACTER except that it holds standardized multibyte characters or Unicode characters.	<code>java.lang.String</code>
NATIONAL CHARACTER VARYING (NVARCHAR)	The NATIONAL CHARACTER VARYING type is the same as CHARACTER VARYING except that it holds standardized multibyte characters or Unicode characters.	<code>java.lang.String</code>

SQL Type	Description	Java Type
CHARACTER LARGE OBJECT (CLOB)	A Character Large Object (or CLOB) is a collection of character data in a DBMS, usually stored in a separate location that is referenced in the table itself. Note that drivers may default to Clob when materializing a CLOB value requires additional database communication.	java.lang.String, io.r2dbc.spi.Clob
NATIONAL CHARACTER LARGE OBJECT (NCLOB)	The NATIONAL CHARACTER LARGE OBJECT type is the same as CHARACTER LARGE OBJECT except that it holds standardized multibyte characters or Unicode characters. Note that drivers may default to Clob when materializing a NCLOB value requires additional database communication.	java.lang.String, io.r2dbc.spi.Clob

The following table describes the SQL type mapping for boolean types:

Table 5. SQL Type Mapping for Boolean Types

SQL Type	Description	Java Type
BOOLEAN	A value that represents a boolean state.	java.lang.Boolean

The following table describes the SQL type mapping for binary types:

Table 6. SQL Type Mapping for Binary Types

SQL Type	Description	Java Type
BINARY	Binary data, fixed length.	java.nio.ByteBuffer
BINARY VARYING (VARBINARY)	A variable-length character string, the maximum length of which is fixed.	java.nio.ByteBuffer

SQL Type	Description	Java Type
BINARY LARGE OBJECT (BLOB)	A Binary Large Object (or BLOB) is a collection of binary data in a database management system, usually stored in a separate location that is referenced in the table itself. Note that drivers may default to Blob when materializing a BLOB value requires additional database communication.	<code>java.nio.ByteBuffer</code> , <code>io.r2dbc.spi.Blob</code>

The following table describes the SQL type mapping for numeric types:

Table 7. SQL Type Mapping for Numeric Types

SQL Type	Description	Java Type
INTEGER	Represents an integer. The minimum and maximum values depend on the DBMS (typically 4-byte precision).	<code>java.lang.Integer</code>
TINYINT	Same as the INTEGER type except that it might hold a smaller range of values, depending on the DBMS (typically 1-byte precision).	<code>java.lang.Byte</code>
SMALLINT	Same as the INTEGER type except that it might hold a smaller range of values, depending on the DBMS (typically 1- or 2-byte precision).	<code>java.lang.Short</code>
BIGINT	Same as the INTEGER type except that it might hold a larger range of values, depending on the DBMS (typically 8-byte precision).	<code>java.lang.Long</code>
DECIMAL(p, s), NUMERIC(p, s)	Fixed precision and scale numbers with precision (p) and scale (s). In other words, a number that can have a decimal point in it.	<code>java.math.BigDecimal</code>

SQL Type	Description	Java Type
FLOAT(p)	Represents an approximate numerical with mantissa precision (p). Databases that use IEEE representation can map values to either 32-bit or 64-bit floating point types depending on the precision parameter (p).	<code>java.lang.Double</code> or <code>java.lang.Float</code>
REAL	Same as the FLOAT type except that the DBMS defines the precision.	<code>java.lang.Float</code>
DOUBLE PRECISION	Same as the FLOAT type except that the DBMS defines the precision. It has greater precision than REAL .	<code>java.lang.Double</code>

The following table describes the SQL type mapping for datetime types:

Table 8. SQL Type Mapping for Datetime Types

SQL Type	Description	Java Type
DATE	Represents a date without specifying a time part and without a timezone.	<code>java.time.LocalDate</code>
TIME	Represents a time without a date part and without a timezone.	<code>java.time.LocalTime</code>
TIME WITH TIME ZONE	Represents a time with a timezone offset.	<code>java.time.OffsetTime</code>
TIMESTAMP	Represents a date and time without a timezone.	<code>java.time.LocalDateTime</code>
TIMESTAMP WITH TIME ZONE	Represents a date and time with a timezone offset.	<code>java.time.OffsetDateTime</code>

The following table describes the SQL type mapping for collection types:

Table 9. SQL Type Mapping for Collection Types

SQL Type	Description	Java Type
COLLECTION (ARRAY, MULTISSET)	Represents a collection of items with a base type.	Array-Variant of the corresponding Java type (for example, <code>Integer[]</code> for INTEGER ARRAY)

Vendor-specific types (such as spatial data types, structured JSON or XML data, and user-defined types) are subject to vendor-specific mapping.

12.2. Mapping of Advanced Data Types

The R2DBC SPI declares default mappings for advanced data types. The following list describes data types and the interfaces to which they map:

- **BLOB**: The **Blob** interface
- **CLOB**: The **Clob** interface

12.2.1. Blob and Clob Objects

An implementation of a **Blob** or **Clob** object may either be locator-based or fully materialize the object in the driver. Drivers should prefer locator-based **Blob** and **Clob** interface implementations to reduce pressure on the client when materializing results.

For implementations that fully materialize Large Objects (LOBs), the **Blob** and **Clob** objects remain valid until the LOB is consumed or the **discard()** method is called.

Portable applications should not depend upon the LOB validity past the end of a transaction.

12.2.2. Creating Blob and Clob Objects

Large objects are backed by a **Publisher** that emits the component type of the large object, such as **ByteBuffer** for **BLOB** and **CharSequence** (or a subtype of it) for **CLOB**.

Both interfaces provide factory methods to create implementations to be used with **Statement**. The following example shows how to create a **Clob** object:

Example 32. Creating and using a Clob object

```
// charstream is a Publisher<String> object
// statement is a Statement object
Clob clob = Clob.from(charstream);
statement.bind("text", clob);
```

12.2.3. Retrieving Blob and Clob Objects from a Row

The Binary Large Object (**BLOB**) and Character Large Object (**CLOB**) data types are treated similarly to primitive built-in types. You can retrieve values of these types by calling the **get(...)** methods on the **Row** interface. The following example shows how to do so:

Example 33. Retrieving a Clob object

```
// result is a Row object
Publisher<Clob> clob = result.map((row, rowMetadata) -> row.get("clob",
Clob.class));
```

The **Blob** and **Clob** interfaces contain methods for returning the content and for releasing resources associated with the object instance. The API documentation provides more details.



LOB value consumption requires special attention due to large object capacity needs. [Mapping of Data Types](#) defines a default LOB mapping using scalar types. The actual default data type for LOB data types can be vendor-specific to avoid blocking if the database requires LOB materialization from a locator or requires database communication during retrieval.

12.2.4. Accessing **Blob** and **Clob** Data

The **Blob** and **Clob** interfaces declare methods to consume the content of each type. Content streams follow Reactive Streams specifications and reflect the stream nature of large objects. As a result, **Blob** and **Clob** objects can be consumed only once. Large object data consumption can be canceled by calling the `discard()` method if the content stream was not consumed at all. Alternatively, if the content stream was consumed, a **Subscription** cancellation releases resources that are associated with the large object.

The following example shows how to consume **Clob** contents:

*Example 34. Creating and using a **Clob** object*

```
// clob is a Clob object
Publisher<CharSequence> charstream = clob.stream();
```

12.2.5. Releasing **Blob** and **Clob**

Blob and **Clob** objects remain valid for at least the duration of the transaction in which they are created. This could potentially result in an application running out of resources during a long-running transaction. Applications may release **Blob** and **Clob** by either consuming the content stream or disposing of resources by calling the `discard()` method.

The following example shows how to free **Clob** resources without consuming it:

*Example 35. Freeing **Clob** object resources*

```
// clob is a Clob object
Publisher<Void> charstream = clob.discard();
charstream.subscribe(...);
```

Chapter 13. Extensions

This section covers optional extensions to the [R2DBC Core](#). Extensions provide features that are not mandatory for R2DBC implementations.

13.1. Wrapped Interface

The **Wrapped** interface provides a way to access an instance of a resource which has been wrapped and for implementors to expose wrapped resources. This mechanism helps to eliminate the need to use non-standard means to access vendor-specific resources.

13.1.1. Usage

A wrapper for a R2DBC SPI type is expected to implement the **Wrapped** interface so that callers can extract the original instance. Any R2DBC SPI interface type can be wrapped. The following example shows how to expose a wrapped resource:

*Example 36. Wrapping a **Connection** and exposing the underlying resource.*

```
class ConnectionWrapper implements Connection, Wrapped<Connection> {  
  
    private final Connection wrapped;  
  
    @Override  
    public Connection unwrap() {  
        return this.wrapped;  
    }  
  
    // constructors and implementation methods omitted for brevity.  
}
```

13.1.2. Interface Methods

The following methods are available on the **Wrapped** interface:

- **unwrap**

13.1.3. The **unwrap** Method

The **unwrap** method is used to return an object that implements the specified interface, allowing access to vendor-specific methods. The returned object may either be the object found to implement the specified interface or a wrapper for that object. Wrappers can be unwrapped recursively. The following example shows how to unwrap a wrapped object:

Example 37. Unwrapping a wrapped object.

```
// connection is a Connection object implementing Wrapped

if (connection instanceof Wrapped) {
    connection = ((Wrapped<Connection>) connection).unwrap();
}
```

13.2. Closeable Interface

The `io.r2dbc.spi.Closeable` interface provides a mechanism for objects associated with resources to release these resources once the object is no longer in use. The associated resources are released without blocking the caller.

13.2.1. Usage

A closeable object is expected to implement the `Closeable` interface so that callers can obtain a `Publisher` to initiate the close operation and get notified upon completion. The following example shows how to close a connection:

Example 38. Closing a `Connection`.

```
// connection is a Connection object
Publisher<Void> close = connection.close();
```

`Connection` implements `Closeable` as a mandatory part of R2DBC. Any stateful object (such as `ConnectionFactory`) can implement `Closeable` to provide a way to release its resources.

13.2.2. Interface Methods

The following methods are available on the `Closeable` interface:

- `close`

13.2.3. The `close` Method

The `close` method is used to return a `Publisher` to start the close operation and get notified upon completion. If the object is already closed, then subscriptions complete successfully and the close operation has no effect.