R2DBC - Reactive Relational Database Connectivity

Ben Hale<bhale@pivotal.io>, Mark Paluch <mpaluch@pivotal.io>, Greg Turnquist <gturnquist@pivotal.io>

Version 0.8.0.M8, 2019-05-14

Preface

License

Foreword

R2DBC is an endeavor to bring a reactive programming API to relational data stores. The Introduction contains more details about its origins and will explain its goals.

This document describes the first and initial generation of R2DBC.

Organisation of this document

This document is organized in x parts:

- tbd

Chapter 1. Introduction

1.1. What is R2DBC?

R2DBC stands for Reactive Relational Database Connectivity. R2DBC started as experiment and proof of concept to enable integration of relational databases into systems using 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 scheduling and parallelization to happen as part of the runtime scheduling.

1.2. The R2DBC SPI

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

R2DBC specifies a service-provider interface (SPI) intended to be implemented by driver vendors and used by client libraries. Using the R2DBC SPI, applications written in a JVM programming language can execute SQL statements, and retrieve results using an underlying data source. The R2DBC SPI can also be used 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 a SQL-like dialect) and 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 to use the concept of `Publisher` and `Subscriber` to allow non-blocking backpressure-aware data access.

1.3. Target Audience

This specification is targeted primarily towards:

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

This specification is also intended to serve the following purposes:

- Introduction for end-users whose applications use the R2DBC SPI.
- Starting point for developers of other SPIs layered on top of the R2DBC SPI.

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 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 forms an open-source organization on GitHub bundling various projects (SPI, drivers) under R2DBC.

To stay up to date with the latest news and announcements in the R2DBC eco system, 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://repo.spring.io/libs-release
- Milestone repository: https://repo.spring.io/libs-milestone
- Snapshot repository: https://repo.spring.io/libs-snapshot

Chapter 2. Goals

This section outlines the goals for R2DBC and the design philosophy for its SPI.

2.1. Enable Reactive Relational Database Connectivity

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

- Non-blocking I/O
- Deferred execution
- Treat application control as series of events (data, errors, completion)
- No longer assume control of resources but leave resource scheduling to the runtime/platform („React to resource availability“)
- Efficient usage of resources
- Leave flow control to be handled by the runtime
- Stream-oriented data consumption
- Functional programming within operators
- Remove assumptions over concurrency from the programming model and leave this aspect up the runtime
- Use back-pressure to allow flow control, to defer the actual execution and to not overwhelm consumers

2.2. Fit 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 in a different platform.

2.3. Offer 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 providing a balance between features that are implemented in a driver and these that are better implemented in a client library.

2.4. Embrace vendor-specific features

Each database comes with its very own feature set and how these are implemented. R2DBC’s goal here 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 R2DBC SPI.

2.5. Keep the focus on SQL

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

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

2.6. Keep it minimal and simple

R2DBC does not aim for being 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 for abstracting common functionality that needs to be re-implemented by driver vendors in a similar manner. It aims for leaving this functionality to client libraries of which there are typically fewer implementations than drivers.

2.7. Provide a foundation for tools and higher-level APIs

R2DBC SPI aims for being primarily consumed though client library implementations.

It does not aim for being 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. Specify 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 not identified features 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: driver): A driver implementing the R2DBC SPI. A driver may provide support for features which are not implemented by the underlying database or expose functionality that is not declared by the R2DBC SPI (*"Extension"*).
- *Supported feature*: A feature for which the R2DBC API implementation supports standard syntax and semantics.
- *Partially supported feature*: A feature for which some methods are implemented via 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 API 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 specifications documented in each interface's Javadoc.
- Drivers must support bind parameter markers.
- Drivers must support transactions.
- Drivers must support native and indexed access to column and parameter references.
- Index references to columns and parameters are zero-based. The first index begins with `0`.

3.3. R2DBC API Compliance

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

- Adhere to the guidelines and requirements above.
- 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`
- Must implement `io.r2dbc.spi.Statement` interface with the exception of the following optional methods:
  - `returnGeneratedValues(…)`: Calling this method should be a no-op for drivers not supporting key generation.
- Must implement `io.r2dbc.spi.ColumnMetadata` interface with the exception of 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 API is providing 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 API.

4.1. Establishing a Connection

R2DBC uses the `Connection` interface to define a logical connection API to the underlying data source. A 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 using R2DBC connects to a target data source 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 vendor-specific API. When an application first attempts to connect to a data source, `ConnectionFactories` automatically loads any R2DBC driver found within the CLASSPATH using Java's `ServiceLoader` mechanism. See ConnectionFactory Discovery for details on 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 `ConnectionFactories` for `ConnectionFactory` lookup.
- Programmatic: In addition to a URL-based configuration, R2DBC provides a programmatic approach so applications can supply structured configuration options to obtain a `ConnectionFactory`.

R2DBC embraces in addition to the two methods mentioned above 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 using an 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.

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

      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
\___/ \_____/ \___/   \___________/\_________/\___________/
  |       |      |           |           |           |
scheme  driver  protocol  authority    path        query
```

- scheme: Identify that the URL is a valid R2DBC URL. Valid schemes are `r2dbc` and `r2dbcs` (configure SSL usage).
- driver: Identifier for a driver. R2DBC 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 endpoint and authorization. The authority may contain a single host or a collection of hostnames/hostname and port tuples by separating these with a comma (",").
- path (Optional): Used as an initial schema/database name.
- query (Optional): Used to pass additional configuration options in the form of `String` key-value pairs using the key name as 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 (using `ConnectionFactoryOptions` constants):

Example URL .R2DBC Connection URL

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

Table 1. Parsed Standard Options

Option	URL Part	Value as per Example
`ConnectionFactoryOptions.DRIVER`	`driver`	`a-driver`
`ConnectionFactoryOptions.PROTOCOL`	`protocol`	`pipes`
`ConnectionFactoryOptions.USER`	User-part of `authority`	`hello`
`ConnectionFactoryOptions.PASSWORD`	Password-part of `authority`	`world`
`ConnectionFactoryOptions.HOST`	Host-part of `authority`	`localhost`
`ConnectionFactoryOptions.PORT`	Port-part of `authority`	`3306`
`ConnectionFactoryOptions.DATABASE`	`path` without leading `/`	`my_database`

Table 2. Parsed Extended Options

Option	URL Part	Value as per Example
`locale`	key-value tuple from `query`	`en_US`

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

4.1.3. Executing SQL and Retrieving Results

TBD.

Chapter 5. Connections

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

The data source can be an RDBMS, a stream-oriented data system some other source of data with a corresponding R2DBC driver. A single application using R2DBC API may 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 thread-safe in the sense that it can be shared across multiple Threads that concurrently execute statements or change its state. A connection object can be shared across multiple Threads that execute operations serially using appropriate synchronization mechanisms.

To obtain a connection, the application may interact with either:

- the `ConnectionFactories` class working with one or more `ConnectionFactoryProvider` implementations

OR

- directly 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 used protocol or aspects that require the use of a different `ConnectionFactory` implementation.

Example 2. `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.

ConnectionFactory Metadata

ConnectionFactories are required to expose metadata to identify the driver (`ConnectionFactory`) and its capabilities. Metadata must not require a connection to a data source.

Example 3. `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 attempts to load any R2DBC driver classes referenced by the `ConnectionFactoryProvider` interface listed in the Java Service Provider manifests available on the CLASSPATH.

Drivers must include the file `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 using this mechanism, `io.r2dbc.spi.ConnectionFactoryProvider` implementations are required to provide a no-argument constructor.

Example 4. META-INF/services/io.r2dbc.spi.ConnectionFactoryProvider file contents

```
com.example.ConnectionFactoryProvider
```

Example 5. `ConnectionFactoryProvider` Interface

```
public interface ConnectionFactoryProvider {

    ConnectionFactory create(ConnectionFactoryOptions connectionFactoryOptions);

    boolean supports(ConnectionFactoryOptions connectionFactoryOptions);

}
```

`ConnectionFactories` uses a `ConnectionFactoryOptions` object to lookup a matching driver using a two-step model:

1. Lookup of an adequate `ConnectionFactoryProvider`.
2. Obtain the `ConnectionFactory` from the `ConnectionFactoryProvider`.

`ConnectionFactoryProvider` implementations are required to return a `boolean` indicator whether they support a specific configuration represented by `ConnectionFactoryOptions`. Drivers must expect any plurality of `Option`s to be configured. Drivers must report that they support a configuration only if the `ConnectionFactoryProvider` can provide a `ConnectionFactory` based on the given `ConnectionFactoryOptions`. Drivers should gracefully fail if a `ConnectionFactory` creation through `ConnectionFactoryProvider.create(…)` is not possible.

See the R2DBC SPI Specification for more details.

5.3. The `ConnectionFactoryOptions` Class

The `ConnectionFactoryOptions` class represents a configuration to request a `ConnectionFactory` from a `ConnectionFactoryProvider`. It represents the programmatic connection creation approach without using driver-specific classes. `ConnectionFactoryOptions` instances are created 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	Literal	Type	Description
`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.

Constant	Literal	Type	Description
`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 that 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.

Example 6. 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.

Example 7. Obtaining a `Connection`

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

The connection is active once it was emitted by the `Publisher` and must be released ("closed") once it is no longer in use.

5.5. 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 management.

Example 8. Closing a `Connection`

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

See the R2DBC SPI Specification for more details.

Chapter 6. Statements

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

6.1. The Statement Interface

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

6.1.1. Creating Statements

`Statement` objects are created by `Connection` objects, as is done in the following example:

Example 9. 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 may be used concurrently by the program and executed at any time. Resources that are associated with a statement are released as soon as the connection is closed.

6.1.2. Executing Statement Objects

`Statement` objects are executed by calling the `execute()` method. Depending on the SQL, the resulting `Publisher` may return one to many `Result` objects. A `Statement` is always associated with its `Connection`, therefore, the connection state affects `Statement` execution at execution time.

1. Executing a `Statement`

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

6.2. Parameterized Statements

SQL used to create a statement can be parameterized using vendor-specific bind markers. Portability of SQL statements across R2DBC implementation is a non-goal.

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

Example 10. Creating three parameterized `Statement` objects 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 resuse (e.g. for prepared statement execution).

6.2.1. 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 ordinal parameter position starting at `0` (zero) or the parameter placeholder representation. The second and any remaining parameters specify the value to be assigned to the parameter.

Example 11. Binding parameters to a `Statement` object by placeholder

```
// 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:

Example 12. 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 it can be executed. The method used to execute a parameterized `Statement` an `IllegalStateException` if a value is not supplied for a bind marker.

6.2.2. Batching

Parameterized `Statement` objects accept multiple parameter binding sets submit a batch of commands to the database for execution. Batch execution 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 bing methods accordingly.

Example 13. Executing 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();
```

Batch execution emits one to many `Result` objects depending on how the implementation executes the batch.

6.2.3. Setting `NULL` Parameters

The method `bindNull` can be used to set any parameter to `NULL`. It takes two parameters, either the ordinal position of the bind marker or the name, and the value type of the parameter.

Example 14. Setting a `NULL` value.

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

6.3. Retrieving Auto Generated Values

Many database systems provide a mechanism that automatically generates a value when inserting a row. The value that is generated may or may not be unique or represent a key value depending on the executed SQL, and table definition. The method `returnGeneratedValues`, which can be called to retrieve the generated value, indicates to the `Statement` object to retrieve generated values. The method accepts a var-arg parameter to specify the column name(s) for which to return generated keys. The emitted `Result` exposes a column for each automatically generated value taking the column name hint into account.

Example 15. 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 not specifying column name(s), the R2DBC driver implementation will determine the columns or value to return.

See the R2DBC SPI Specification for more details.

Chapter 7. Batches

This section describes the `Batch` interface.

7.1. The Batch Interface

The `Batch` interface defines methods for executing SQL statements. SQL statements may not contain parameter bind markers for input parameters. A batch is created to execute multiple SQL statements for performance reasons.

7.1.1. Creating Batches

`Batch` objects are created by `Connection` objects, as is done in the following example:

Example 16. Creating a `Batch`

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

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

7.1.2. Executing Batch Objects

`Batch` objects are executed 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 execution time.

Example 17. Executing 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 8. Results

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

8.1. 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 cannot be longer 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 executed 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 update for SQL Data Manipulation Language (DML) statements. The update count may 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 cannot be longer consumed.

Example 18. Consuming update count

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

The streaming nature of a result allows either consumption of tabular results or 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.

8.2. Creating `Result` Objects

A `Result` object is most often created as the result of executing a `Statement` object. The `Statement` method `execute()` returns a `Publisher` that emits `Result` objects as result of statement execution.

Example 19. 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 will emit a `Row` for each row in the table `books`, containing the two columns `title` and `author`. The following sections detail how these rows and columns can be consumed.

8.2.1. Cursor Movement

`Result` objects can be backed by direct results (i. e. 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 resources associated with the `Result` object.

8.3. Rows

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

8.3.1. Retrieving Values

The `Result` interface provides a `map(…)` method for retrieving values from `Row` objects. The `map` method accepts `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 are invalid outside of the mapping function callback. Thus `Row` objects must be entirely consumed in the mapping function.

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

8.4. Interface Methods

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

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

Both `get` methods accept a column identifier that can be either the column name or the column index. Column names used as input to the `get` methods are case insensitive. Column names do not necessarily reflect the column names how they are in the underlying tables but rather how columns are represented (e.g. aliased) in the result.

Example 20. Creating and Consuming a `Row` using its index

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

Example 21. Creating and Consuming a `Row` through 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. Specifying a target type, the R2DBC driver attempts to convert the value to the target type.

Example 22. 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));
```

Example 23. 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 may 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 9. 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 a SQL statement are unknown until execution, 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 query execution) to provide a complete metadata descriptor. Issuing queries during result processing conflicts with the streaming nature of R2DBC and so `ColumnMetadata` declares two sets of methods: Methods that must be implemented and methods that can optionally be implemented by drivers.

9.1. 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 using an anonymous inner class:

Example 24. Using `RowMetadata` and retieving `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();
        // …
    }
})
```

9.2. Retrieving `ColumnMetadata`

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

- `getColumnMetadata(…)` returns the `ColumnMetadata` by using a column identifier. The identifier is either a zero-based index or the column name, see Guidelines and Requirements.
- `getColumnMetadatas()` returns an unmodifiable collection of `ColumnMetadata` objects.

9.3. 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 statement execution must be made available through `ColumnMetadata`. Metadata exposure requiring interaction with the database (e.g. issuing queries to information schema entities to resolve type properties) should not be exposed as 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 illustrates how to consume `ColumnMetadata` using lambdas:

Example 25. 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 10. 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.

10.1. General Exceptions

10.1.1. `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 a SQLState.

10.1.2. `IllegalStateException`

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

10.1.3. `UnsupportedOperationException`

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

10.1.4. `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. The `String` containing the description can be retrieved by invoking `R2dbcException.getMessage()`. Drivers may provide a localized message variant.
- A SQLState. The `String` containing the `String` can be retrieved by calling the method `R2dbcException.getSqlState()`. The value of the SQLState string will depend on the underlying data source.
- An error code. The code is an integer value identifying 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. The error code can be retrieved using the `R2dbcException.getErrorCode()` method.
- A cause. This is another `Throwable` which caused this `R2dbcException` to occur.

10.2. 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 an SQLState-based exception translation resulting in more portable error-handling code.

R2DBC categorizes exceptions into two top-level categories:

- `R2dbcNonTransientException`
- `R2dbcTransientException`

10.2.1. 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 is corrected. After a non-transient exception other than `R2dbcNonTransientResourceException`, the application may 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.

10.2.2. 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 an 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) is exceeded. This could have different causes depending on the database API in use but 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 11. Data Types

This chapter discusses the use of data types from Java and database perspectives. The R2DBC SPI gives applications access to data types defined as of 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.

11.1. Mapping of Data Types

This section explains how SQL-specific types are mapped to Java types. The list is nonexhaustive 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

Table 4. SQL Type Mapping for Character Types

SQL Type	Description	Java Type
`CHARACTER` (`CHAR`)	Character string, fixed length.	`java.lang.String`
`CHARACTER VARYING` (`VARCHAR`)	Variable-length character string, maximum length fixed.	`java.lang.String`
`NATIONAL CHARACTER` (`NCHAR`)	`NATIONAL CHARACTER` type is the same as `CHARACTER` except that it holds standardized multibyte characters or Unicode characters.	`java.lang.String`
`NATIONAL CHARACTER VARYING` (`NVARCHAR`)	`NATIONAL CHARACTER VARYING` type is the same as `CHARACTER VARYING` except that it holds standardized multibyte characters or Unicode characters.	`java.lang.String`
`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.	`io.r2dbc.spi.Clob`

SQL Type	Description	Java Type
`NATIONAL CHARACTER LARGE OBJECT` (`NCLOB`)	`NATIONAL CHARACTER LARGE OBJECT` type is the same as `CHARACTER LARGE OBJECT` except that it holds standardized multibyte characters or Unicode characters.	`io.r2dbc.spi.Clob`

Table 5. SQL Type Mapping for Boolean Types

SQL Type	Description	Java Type
`BOOLEAN`	Single-bit representing a boolean state.	`java.lang.Boolean`

Table 6. SQL Type Mapping for Binary Types

SQL Type	Description	Java Type
`BINARY`	Binary data, fixed length.	`java.nio.ByteBuffer`
`BINARY VARYING` (`VARBINARY`)	Variable-length character string, maximum length fixed.	`java.nio.ByteBuffer`
`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.	`io.r2dbc.spi.Blob`

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.	`java.lang.Integer`
`SMALLINT`	Same as `INTEGER` type except that it might hold a smaller range of values, depending on the DBMS, typically 1- or 2-byte precision.	`java.lang.Short`
`BIGINT`	Same as `INTEGER` type except that it might hold a larger range of values, depending on the DBMS, typically 8-byte precision.	`java.lang.Long`

SQL Type	Description	Java Type
`DECIMAL(p, s)`, `NUMERIC(p, s)`	Fixed precision and scale numbers with precision `p`, scale `s`. A decimal number, that is a number that can have a decimal point in it. The size argument has two parts: precision and scale.	`java.math.BigDecimal`
`FLOAT(p)`	Represents an approximate numerical with mantissa precision `p`.	`java.lang.Double`
`REAL`	Same as `FLOAT` type except that the DBMS defines the precision.	`java.lang.Double`

Table 8. SQL Type Mapping for Datetime Types

SQL Type	Description	Java Type
`DATE`	Represents a date without specifying a time part and without timezone.	`java.time.LocalDate`
`TIME`	Represents a time without a date part and without timezone.	`java.time.LocalTime`
`TIMESTAMP`	Represents a date/time without a timezone.	`java.time.LocalDateTime`
`TIMESTAMP` with Timezone Offset	Represents a date/time with a timezone offset.	`java.time.OffsetDateTime`
`TIMESTAMP` with Timezone	Represents a date/time with a timezone.	`java.time.ZonedDateTime`
`INTERVAL`	Interval date types such as `YEAR`, `MONTH`, `DAY`, `HOUR` and similar representing a time quantity.	`java.time.Duration`

Table 9. SQL Type Mapping for Collection Types

SQL Type	Description	Java Type
`COLLECTION` (`ARRAY`, `MULTISET`)	Represents a collection of items with a base type.	Array-Variant of the corresponding Java type (e.g. `Integer[]` for `INTEGER ARRAY`)

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

11.2. Mapping of Advanced Data Types

The R2DBC API 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

11.2.1. `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 the Large Objects (LOB), 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.

11.2.2. Creating `Blob` and `Clob` Objects

Large Objects are backed by a `Publisher` emitting 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 explains how to create a `Clob` object:

Example 26. 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);
```

11.2.3. 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. Values of these types can be retrieved by calling the `get(···)` methods on the `Row` interface.

Example 27. Retrieving a `Clob` object

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

The `Clob` interface contains methods for returning the content and for releasing resources associated with the `Clob` object instance. The API documentation provides more details.

11.2.4. 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 hence `Blob` and `Clob` objects can be consumed only once. Large object data consumption can be canceled by either 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 associated with the large object.

The following example explains how to consume `Clob` contents:

Example 28. Creating and using a `Clob` object

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

11.2.5. 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 29. Freeing `Clob` object resources

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

Chapter 12. Extensions

This section covers optional extensions to R2DBC Core. Extensions provide features that are not mandatory to implement by R2DBC implementations.

12.1. Wrapped Interface

The `Wrapped` interface provides a mechanism for users 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.

12.1.1. 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.

Example 30. 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.
}
```

12.1.2. Interface Methods

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

- `unwrap`

12.1.3. `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.

Example 31. Unwrapping a wrapped object.

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

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