R2DBC - Reactive Relational Database Connectivity

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

Version 1.0.0.M7, 2019-02-14

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()`

Chapter 4. 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.

4.1. 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 1. 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();
        // …
    }
})
```

4.2. 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.

4.3. 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 2. 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.