
R2DBC
Reactive Relational
Database Connectivity
Towards 1.0
REACTIVE
FOUNDATION

Introduction

Mark Paluch

- Spring Data Project Lead
- R2DBC Spec Lead
- Lettuce Redis Driver Maintainer
- Music Producer

@mp911de github.com/mp911de

What is R2DBC?

- Community-driven specification to integrate SQL databases using reactive programming on the JVM
- De-facto standard
- Based on Reactive Streams and Java 8

- GA version: 0.8
- Development version: 0.9

Design Principles

- Embrace Reactive Types and Patterns
- Low barrier to entry (depends on Reactive Streams and Java 8 only)
- Non-blocking, all the way to the database
- De-duplicate driver efforts
- Documented specification
- Shrink the driver SPI
- Enable multiple "humane" APIs

R2DBC 0.8

Initial Release

R2DBC 0.8 (Initial Release)

- End-to-end reactive non-blocking database communication
- Batching
- BLOB/CLOB
- Extensive Type Conversion
- Savepoints
- Transactions
- Leveraging Database-specific features
- ServiceLoader-based Driver discovery
- Connection URLs
- Categorized exceptions (Bad grammar, Data integrity violation, …)

R2DBC Eco-System

- Specification
- R2DBC SPI
- R2DBC Proxy
- Connection Pooling

- Client Implementations
  - JOOQ
  - Kotysa
  - Micronaut
  - MyBatis
  - Querydsl
  - R2DBC Migrate
  - Spring
  - Testcontainers

- Driver Community
  - Google Cloud Spanner
  - H2
  - Microsoft SQL Server
  - MySQL Drivers (r2dbc-mysql, jasync-sql)
  - MariaDB
  - PostgreSQL
  - Oracle
  - SAP HANA (in development)
  - Yugabyte (in development)

R2DBC In Action

```
Publisher<String> values = connectionFactory.create()
  .flatMapMany(conn ->
    conn.createStatement("SELECT value FROM test")
        .execute()
        .flatMap(result ->
            result.map((row, metadata) -> row.get("value",
                                                  String.class))))
```

R2DBC Configuration (1/3)

```
ConnectionFactory connectionFactory = ConnectionFactories.get(ConnectionFactoryOptions.builder()
    .option(DRIVER, MSSQL_DRIVER)
    .option(HOST, SERVER.getHost())
    .option(PORT, SERVER.getPort())
    .option(PASSWORD, SERVER.getPassword())
    .option(USER, SERVER.getUsername())
    .build());
```

R2DBC Configuration (2/3)

```
ConnectionFactory connectionFactory =
      ConnectionFactories.get("r2dbc:pool:mssql://localhost:1433/my_database")
```

R2DBC Configuration (3/3)

```
ConnectionFactory connectionFactory =
      ConnectionFactories.get("r2dbc:pool:mssql://localhost:1433/my_database")

ConnectionFactoryOptions.parse("r2dbc:pool:mssql://localhost:1433/my_database")
   .mutate()
   .option(USER, …)
```

R2DBC 0.9

In Development

R2DBC 0.9 (In development)

- Extensible Transaction Definitions
- Bind Parameter Directions (IN/OUT) and Types
- Consumption of OUT Parameters (e.g. Stored Procedures)
- Result Segment API
- Connection Lifecycle API
- Standard Lock Timeout and Statement Timeout API
- Access to SQL through Exceptions
- Metadata, Configuration, and Exception Refinements

Starting a Transaction

- Transaction Setup
  - Set Isolation Level
  - Set Transaction Mutability
  - Set Transaction Attributes
  - `BEGIN`
- Multiple roundtrips, latency

Extensible Transaction Definitions

- Define transaction
- Let the driver start a transaction with all attributes in place
- Fewer roundtrips, less latency
- Standard API for vendor-specific transaction options

- Postgres: Mutability, Deferrable/Non-deferrable
- SQL Server: Transaction name, lock timeout, WITH MARK

Extensible Transaction Definitions Example

```
Mono<Void> begin = connection.beginTransaction(MssqlTransactionDefinition
                        .from(IsolationLevel.READ_COMMITTED)
                        .lockTimeout(Duration.ofSeconds(20))
                        .name("my_transaction"));
```

Bind Parameters

- Serialization-safe binding of values to parameter placeholders
- Used also with prepared statements
- zero-based index/named parameters
- Database type derived from value
- In 0.8: IN-parameters only

Improved Bind Parameters

- Directionality
  - IN, IN/OUT, OUT
- Type specification
  - Derived type (R2DBC 0.8 compatibility)
  - R2DBC Type (`io.r2dbc.spi.R2dbcType`)
  - Vendor-specific type (e.g. Postgres JSON/Enum)
- Easier interface for nullable values

Improved Bind Parameters Example

```
connection.createStatement("EXEC my_procedure @P0, @Greeting OUTPUT")
    .bind("@P0", "Walter")
    .bind("@Greeting", Parameters.out(R2dbcType.VARCHAR))
```

```
PostgresTypes types = PostgresTypes.from(connection);
PostgresType type = types.lookupType("my_enum").block();

connection.createStatement("INSERT INTO table_with_enum VALUES($1)")
            .bind("$1", Parameters.in(type, "HELLO"))
```

Consumption of OUT Parameters (1/3)

- R2DBC started with focus on tabular data
- Over time, demand for consumption of stored procedures results

Consumption of OUT Parameters (2/3)

- API
  - `OutParameters`
  - `OutParameterMetadata`/`OutParametersMetadata`
- Retrofitted Result/Row/ColumnMetadata
  - `Readable` Interface (base for `OutParameters` and `Row`)
  - `Result.map(Function<Readable, T>)`

Consumption of OUT Parameters Example

```
connection.createStatement("EXEC my_procedure @Name, @Greeting OUTPUT")
    .bind("@Name", "Walter")
    .bind("@Greeting", Parameters.out(R2dbcType.VARCHAR))
            .execute()
            .flatMap(it -> it.map((readable) -> {
                return readable.get(0) // readable.get("Name");
            }))
```

Consumption of OUT Parameters (3/3)

- Only registered OUT Parameters can be consumed
  - by name
  - by index
    - zero-based, starts counting from first registered out parameter
- Lose definition of `Readable` consumption via

  `Result.map(Function<Readable, T>)`
  - `Readable` can be `Row` or `OutParameters`

Result Segment API (1/3)

- In R2DBC 0.8: Consumption of either updated rows count or tabular results. Not both.
- No access to messages (informations, error) or vendor-specific result segments

Result Segment API (2/3)

- R2DBC 0.9: `Result` corresponds with `Publisher<Segment>`
- Enables functional consumption of various results from a statement
- New operators
  - `filter(Predicate<Segment>)`
  - `flatMap(Function<Segment, Publisher>)`
- Segments
  - `RowSegment`
  - `OutSegment`
  - `UpdateCount`
  - `Message`

Result Segment API Example

```
Flux<Long> updateCounts = statement.execute()
   .filter(Result.UpdateCount.class::isInstance)
   .flatMap(result -> result.getRowsUpdated().cast(Result.UpdateCount.class)
       .flatMap(segment -> Mono.just(segment.value())));
```

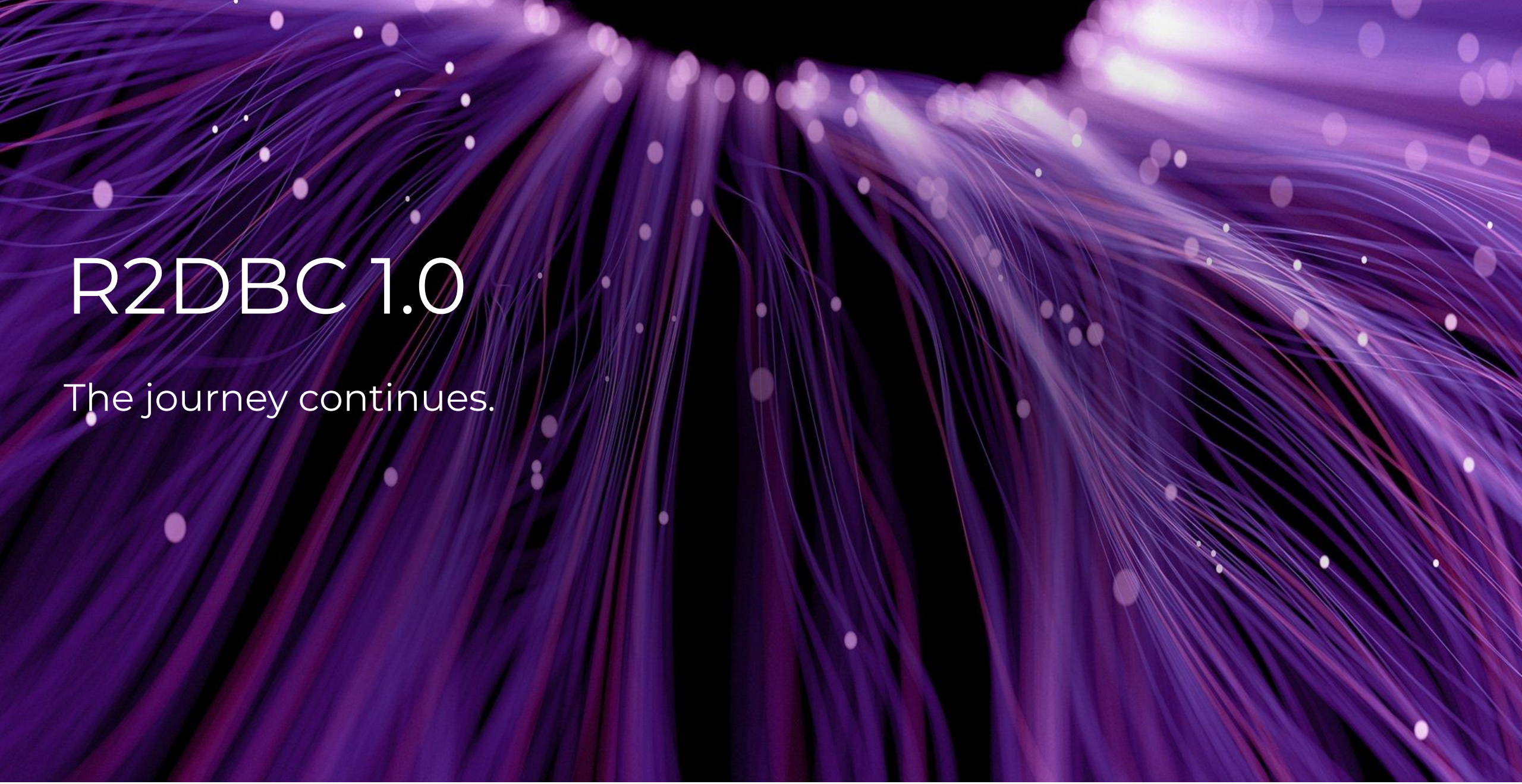
```
Flux<Long> updateCounts = statement.execute()
   .flatMap(result -> result.flatMap(segment -> {

       if (segment instanceof Result.Message) {
           R2dbcException exception = ((Result.Message) segment).exception();
       }

       return null;
   }));
```

Result Segment API (3/3)

- Allows for filtered results
  - e.g. remove messages so that `Result.map(...)` never terminates with an error signal
  - e.g. filter all segments to suppress results
- `flatMap(...)` allows conditional emission of `0...1...N` elements


R2DBC 1.0
The journey continues.

R2DBC is here to stay

- R2DBC is feature-complete for version 1.0
- De-facto standard
- Wide adoption across the community
  - Driver vendors
  - Data access library authors
  - Application frameworks
  - Utilities, observability libraries, …

What problem solves R2DBC?

- Non-blocking and streaming access to SQL databases
- Interchangeable drivers
  - Stable specification for driver vendors
  - Satisfies well-defined expectations from data access library maintainers
- Modern type system

Project Loom

- Aims to deliver high-throughput lightweight concurrency
- Building blocks for virtual thread scheduling
- Allows existing code using blocking APIs to run as if it was non-blocking

R2DBC vs. **Project Loom**

- Loom is a virtual Thread endeavour
  - Enables existing code with little changes to run as-if it was non-blocking
  - Imperative Programming Model
  - Improves allocated resources efficiency by switching to work that can progress
  - Database result streaming requires proper libraries, polling, and retention of JDBC `ResultSet`
  - No built-in cancellation protocol
  - No built-in backpressure/demand communication
  - Availability unknown

R2DBC vs. Project Loom

- R2DBC is a data access specification
  - Built on top of Reactive Streams
  - Reactive Programming Model
  - Allows end-to-end streaming baked without change to code
  - Built-in cancellation to stop result/cursor consumption
  - Backpressure can improve performance profile by communicating subscriber demand
  - Available since 2019

R2DBC vs. Project Loom

- There is no **vs**.
- Why do you do Reactive?
  - Scheduling efficiency: Check out Loom
  - More than just efficient CPU usage: Reactive is your home

R2DBC 1.0

- Java 1.8 baseline
- Based on Reactive Streams
- Planned for mid 2022
- Minor refinements and cleanups

REACTIVE
FOUNDATION

Thank you

r2dbc.io · twitter.com/r2dbc · github.com/r2dbc