

WebFlux

Advanced Java I. Functional, Asynchronous, Reactive Java
Module 7

think.
create.
accelerate.

luxoft | training
A DXC Technology Company

Approaches for client - DB server communication

- **Retrieve the entire result set**
- **Retrieve result set in chunks**
- **Retrieve results as a stream** as soon as such results are obtained during the query execution.
 - On top of that, the client may also inform the database about the demand and propagate logical backpressure that may, in turn, impact the query execution process. Such an approach requires almost no additional buffers, and the client receives the first row of the result as soon as it is possible.

Benefits of reactive access to DB

- **Effective thread management**, since no thread is required to ever block on IO operations. This usually means that fewer threads are created, there's less overhead on thread scheduling, there's less memory footprint allocated for the Thread object's stack, and consequently, it's able to handle a massive amount of concurrent connections.
- **Smaller latency to the first results** of a query. These may become available even before the query finishes. It may be convenient for search engines and interactive UI components that target low latency operation.
- **Lower memory footprint**. This is useful as less data is required to be buffered when processing a query for outgoing or incoming traffic. Also, the client may unsubscribe from a reactive stream and reduce the amount of data sent over the network as soon as it has enough data to fulfill its needs.
- **Backpressure propagation** informs the client about a database's ability to consume new data. Also, it permits informing the database server about the client's ability to process query results. In this case, more urgent work may be done instead.
- As no threads hold exclusive rights over query objects and no client code is ever blocked, it is possible to **share a single wire connection to the database** and forget about connection pooling.
- **smooth integration of the persistence layer with a fluent reactive code** of the reactive application

Reactive R2DBC drivers

<https://r2dbc.io/>

- **Based on the Reactive Streams specification.** R2DBC is founded on the Reactive Streams specification, which provides a fully-reactive non-blocking API.
- **Works with relational databases.** In contrast to the blocking nature of JDBC, R2DBC allows you to work with SQL databases using a reactive API.
- **Supports scalable solutions.** With Reactive Streams, R2DBC enables you to move from the classic “one thread per connection” model to a more powerful and scalable approach.
- **Provides an open specification.** R2DBC is an open specification and establishes a Service Provider Interface (SPI) for driver vendors to implement and clients to consume.

Driver Implementations:

[cloud-spanner-r2dbc](#): driver for Google Cloud Spanner

[iasync-sql](#): R2DBC wrapper for Java & Kotlin Async Database Driver for MySQL and PostgreSQL written in Kotlin.

[r2dbc-h2](#): native driver implemented for H2 as a test database.

[r2dbc-mariadb](#): native driver implemented for MariaDB.

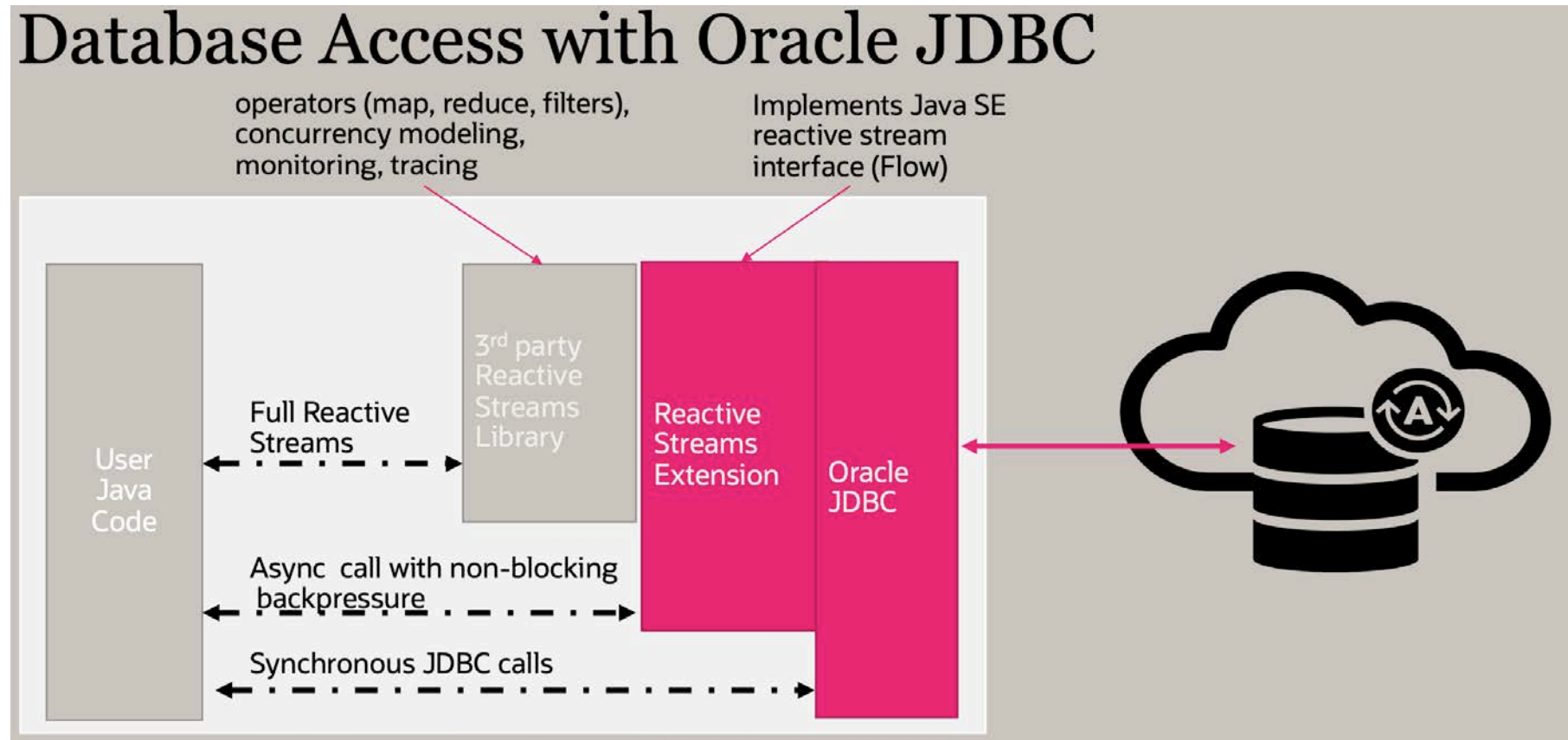
[r2dbc-mssql](#): native driver implemented for Microsoft SQL Server.

[r2dbc-mysql](#): native driver implemented for MySQL.

[r2dbc-postgres](#): native driver implemented for PostgreSQL.

What about Oracle?

- Oracle was developing its own standard ADBA (asynchronous database access) since 2017
- Oracle has stopped ADBA development since September 2019 and declared that further development will be related with Fibers (project Loom) which is currently in development
- <https://www.oracle.com/a/tech/docs/dev6323-reactivestreams-fiber.pdf>



Reactive driver schema loader

Automatic schema generation is not supported.

To use R2DBC you should create schema and load it manually with SchemaLoader:

```
private Mono<String> getSchema() throws URISyntaxException {
    Path path = Paths.get(ClassLoader
        .getSystemResource("schema.sql").toURI());
    return Flux.using(() -> Files.lines(path),
        Flux::fromStream, BaseStream::close)
        .reduce((line1, line2) -> line1 + "\n" + line2);
}

private Mono<Integer> executeSql(DatabaseClient client, String sql) {
    return client.execute(sql).fetch().rowsUpdated();
}

@Bean
public ApplicationRunner seeder(DatabaseClient client) {
    return args -> getSchema()
        .flatMap(sql -> executeSql(client, sql))
        .subscribe(count -> System.out.println("Schema created"));
}
```

Spring Data Reactive repository

Reactive repository is looking and working very similar to not reactive.

```
public interface PersonRepository extends R2dbcRepository<Person, Long> {  
    Flux<Person> findByName(String name);  
}
```

```
public interface R2dbcRepository<T, ID> extends ReactiveCrudRepository<T, ID> {}
```

```
public interface ReactiveCrudRepository<T, ID> extends Repository<T, ID> {  
    <S extends T> Mono<S> save(S entity);  
    <S extends T> Flux<S> saveAll(Iterable<S> entities);  
    <S extends T> Flux<S> saveAll(Publisher<S> entityStream);  
    Mono<T> findById(ID id);  
    Mono<T> findById(Publisher<ID> id);  
    Mono<Boolean> existsById(ID id);  
    Mono<Boolean> existsById(Publisher<ID> id);  
    Flux<T> findAll();  
    Flux<T> findAllById(Iterable<ID> ids);  
    Flux<T> findAllById(Publisher<ID> idStream);  
    Mono<Long> count();  
    Mono<Void> deleteById(ID id);  
    Mono<Void> deleteById(Publisher<ID> id);  
    Mono<Void> deleteAll(Iterable<? extends T> entities);  
    Mono<Void> deleteAll(Publisher<? extends T> entityStream);  
    Mono<Void> deleteAll();  
}
```

ReactiveCrudRepository

Interface for generic CRUD operations on a repository for a specific type. This repository follows reactive paradigms and uses Project Reactor types which are built on top of Reactive Streams.

Pagination support

Reactive Spring Data MongoDB repositories **do not provide paging** in the sense of paging how it's designed for imperative repositories. Imperative paging requires additional details while fetching a page. In particular:

- The number of returned records for a paging query
- Optionally, total count of records the query yields if the number of returned records is zero or matches the page size to calculate the overall number of pages

Both aspects do not fit to the notion of efficient, non-blocking resource usage. Waiting until all records are received (to determine the first chunk of paging details) would remove a huge part of the benefits you get by reactive data access. Additionally, executing a count query is rather expensive, and increases the lag until you're able to process data.

You can still fetch chunks of data yourself by passing a Pageable (PageRequest) to repository query methods:

```
interface ReactivePersonRepository extends Repository<Person, Long> {  
    Flux<Person> findByFirstnameOrderByLastname(String firstname, Pageable pageable);  
}
```

Spring Data will apply pagination to the query by translating Pageable to LIMIT and OFFSET.

WebFlux: functional-based controllers

In the functional realm, a web service is referred to as a route and the traditional concept of `@Controller` and `@RequestMapping` is replaced by a `RouterFunction`. This makes routing more lightweight. Moreover, the combination of pure functional routing fits sufficiently with new reactive programming approaches.

@Component

```
public class RootHandler {  
    public Mono<ServerResponse> root(  
        ServerRequest serverRequest) {  
        return ServerResponse.ok().bodyValue("hi");  
    }  
  
    public Mono<ServerResponse> hello(  
        ServerRequest request) {  
        Flux<String> data = Flux  
            .just("Hello", "From reactive",  
                "Spring", "WebFlux", "!")  
            .delayElements(Duration.ofSeconds(1))  
            .map(s->s+" ");  
  
        return ServerResponse.ok().body(data, String.class);  
    }  
}
```

@Configuration

```
public class HelloRouter {  
    @Bean  
    public RouterFunction<ServerResponse> route  
        (RootHandler rootHandler) {  
        return RouterFunctions  
            .route(RequestPredicates.GET(""),  
                rootHandler::root)  
            .andRoute(RequestPredicates.GET("/hello"),  
                rootHandler::hello);  
    }  
}
```

We should note that in the functional approach, the `route()` method returns a `RouterFunction` instead of the response body.
It's the definition of the route, not the execution of a request.

Spring Rest controllers returning reactive results Mono/Flux

```
@RestController
@RequestMapping("/person")
public class PersonController {
    private final PersonRepository personRepository;

    @GetMapping
    public Flux<Person> list(@RequestParam(defaultValue = "0") Long start,
                           @RequestParam(defaultValue = "3") Long count) {
        return personRepository.findAll()
            .skip(start).take(count);
    }

    @PostMapping
    public Mono<Person> add(@RequestBody Person person) {
        return personRepository.save(person);
    }
}
```

Spring Rest controllers returning SSE

Server-Sent-Events, or SSE for short, is an HTTP standard that allows a web application to handle a unidirectional event stream and receive updates whenever server emits data.

Flux is a reactive representation of a stream of events – it's handled differently based on the specified request or response media type.

To create an SSE streaming endpoint, we'll have to follow the [W3C specifications](#) and designate its MIME type as *text/event-stream*:

```
@GetMapping(path = "/stream", produces = "text/event-stream")
public Flux<String> getStream() {
    return Flux
        .just("Sasha", "Misha", "Dima", "Vasya")
        .delayElements(Duration.ofSeconds(2));
}
```

```
@GetMapping(path = "/persons", produces = "text/event-stream")
public Flux<Person> persons() {
    return nameGenerator.persons()
        .delayElements(Duration.ofSeconds(1))
        .take(Duration.ofSeconds(10));
}
```

Retrieving reactive data from server with WebClient

WebClient is the reactive replacement for the old RestTemplate

```
WebClient client = WebClient
    .builder()
    .baseUrl("http://localhost:8080")
    .defaultHeader(HttpHeaders.USER_AGENT, "Spring 5 WebClient")
    .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
    .build();

client.get()
    .uri("/person/names")
    .retrieve()
    .bodyToFlux(String.class)
    .timeout(Duration.ofMillis(1000)) // limited time for execution
    .subscribe(System.out::println, err -> System.out.println("ERROR
"+err.getMessage()));
```

Retrieving reactive data from server with WebClient

If we need to consume SSE, we should define ParameterizedTypeReference:

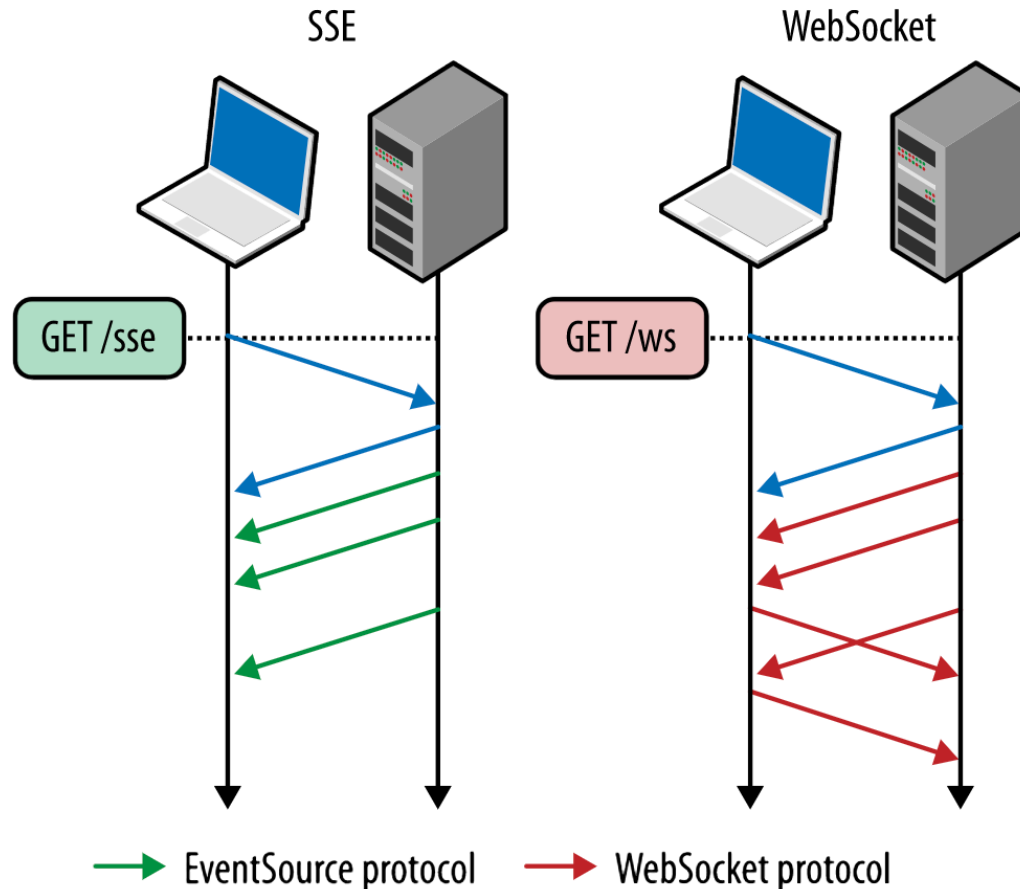
```
ParameterizedTypeReference<ServerSentEvent<String>> type =  
    new ParameterizedTypeReference<ServerSentEvent<String>>() {};
```

```
client.get()  
    .uri("/person/stream")  
    .accept(TEXT_EVENT_STREAM)  
    .retrieve()  
    .bodyToFlux(type)  
    .timeout(Duration.ofMillis(5000))  
    .subscribe(content -> {  
        System.out.println(content.data());  
    }, err -> System.out.println("ERROR "+err.getMessage()));
```


SSE and WebSocket protocols

WebSocket connections can both **send** data to the browser and **receive** data from the browser. A good example of an application that could use websockets is a chat application, and various data streams exchange.

SSE connections can only push data to the browser. Online stock quotes, or twitters updating timeline or feed are good examples of an application that could benefit from SSE.



Protocols differences:

- SSE does not support binary encoding and limits events to UTF-8 encoding
- For SSE, Spring WebFlux offers the same message converter configurations as for a typical REST controller.
- WebSocket works as a full-duplex protocol and allows binary messages exchange

Using WebSocket: client

```
WebSocketClient client = new ReactorNettyWebSocketClient();
client.execute(
    URI.create("ws://localhost:8080/event-emitter"),
    session -> session.send(
        Mono.just(session.textMessage("hi from websocket client")))
        .thenMany(session.receive()
            .map(WebSocketMessage::getPayloadAsText)
            .map(el -> el+"!")
            .map(session::textMessage)
            .map(Mono::just)
            .flatMap(session::send)
        )
        .then())
    .take(Duration.ofSeconds(10))
    .block();
```

WebSockeClient implementations are:

- StandardWebSocketClient
- JettyWebSocketClient
- ReactorNettyWebSocketClient
- TomcatWebSocketClient
- UndertowWebSocketClient

Using WebSocket: server side

```
@Configuration
public class WSConfiguration {
    @Autowired
    @Qualifier("WSHandler")
    private WebSocketHandler webSocketHandler;

    @Bean
    public HandlerMapping webSocketHandlerMapping() {
        Map<String, WebSocketHandler> map = new HashMap<>();
        map.put("/event-emitter", webSocketHandler);

        SimpleUrlHandlerMapping handlerMapping =
            new SimpleUrlHandlerMapping();
        handlerMapping.setOrder(1);
        handlerMapping.setUrlMap(map);
        return handlerMapping;
    }

    @Bean
    public WebSocketHandlerAdapter handlerAdapter() {
        return new WebSocketHandlerAdapter();
    }
}
```

```
@Component("WSHandler")
public class WSHandler implements WebSocketHandler {
    @Override
    public Mono<Void> handle(WebSocketSession session) {
        return session.send(
            Flux.interval(Duration.ofSeconds(1))
                .map(v -> "next val: "+v)
                .map(session::textMessage)
        ).and(
            session.receive()
                .map(WebSocketMessage::getPayloadAsText)
                .doOnNext(e1 -> System.out.println(e1))
                .log()
        );
    }
}
```

Using WebSocket to retrieve JSON: client

We need to retrieve person data as JSON by its id: 1, 2, 3.

```
WebSocketClient client = new ReactorNettyWebSocketClient();
client.execute(
    URI.create("ws://localhost:8080/wsperson"),
    session -> {
        Mono<Void> output = session.send(Flux.just(1,2,3)
            .delayElements(Duration.ofSeconds(3))
            .map(Object::toString)
            .map(session::textMessage)
            .doOnComplete(()-> System.out.println("output completed")))
        );
        Mono<Void> input = session.receive()
            .map(WebSocketMessage::getPayloadAsText)
            .doOnNext(System.out::println)
            .then();
        return Mono.first(input,
            output.then(Mono.delay(Duration.ofSeconds(1))))
            .then();
    })
    .block();
```

Finish as soon as output completes.
However, we need some more time so that
the last message could finish processing.

We need either to subscribe to input and return output, or return output combined with input.

For this aim, we can use:

1) **then()** - waiting output to complete,
then waiting input to complete:

output.then(input);

In this case we will start receiving results
only after output completes.

2) **and()** - waiting both to complete:

output.and(input)

Since input wouldn't complete until server
completes, we may use `.take(3)` or `.takeUntil()` to make
it complete.

3) **Mono.first()** - waiting either to complete:

`Mono.first(output, input)`

4) **Mono.zip()** can be used for the same (as
recommended in docs):

`Mono.first(output, input);`

Using WebSocket to retrieve JSON: server

@Configuration

```
public class WSPersonConfiguration {  
    ObjectMapper converter = new ObjectMapper();
```

WebFlux1/websocket/person/WSPersonConfiguration

@Bean

```
public HandlerMapping wsPerson() {  
    return new SimpleUrlHandlerMapping(Map.<String, WebSocketHandler>of(  
        "/wsperson", session -> {  
            Flux<Long> idFlux =  
                session.receive().map(WebSocketMessage::getPayloadAsText).map(Long::valueOf);
```

```
            Flux<Person> personFlux = idFlux.concatMap(personRepository::findById);
```

```
            Flux<String> personStringFlux = personFlux.map(p -> {  
                try {  
                    return converter.writeValueAsString(p);  
                } catch (JsonProcessingException e) {  
                    return "{\"error\":\"cannot convert to JSON\"}";  
                }  
            });
```

```
            Flux<WebSocketMessage> personMessages = personStringFlux.map(session::textMessage);  
            return session.send(personMessages);
```

```
        }), 1);
```

order for SimpleUrlHandlerMapping

Using WebSocket to retrieve binary data

```
WebSocketClient client = new ReactorNettyWebSocketClient();
client.execute(URI.create("ws://localhost:8080/wsbinary"),
    session -> {
        Mono<Void> output = session.send(Flux.just(1,2,3).delayElements(Duration.ofSeconds(3))
            .map(Object::toString).map(session::textMessage)
            .doOnComplete(()-> System.out.println("output completed")))
        );

        Mono<Void> input = session.receive().map(WebSocketMessage::getPayload)
            .map(dataBuffer -> {
                ByteBuffer buffer = dataBuffer.asByteBuffer();
                byte[] bytes = new byte[buffer.remaining()]; ← Convert binary data to String
                buffer.get(bytes);
                return new String(bytes);
            })
            .doOnNext(System.out::println).then();

        return Mono.first(
            output.then(Mono.delay(Duration.ofSeconds(1))), input).then();
    }).block();
```

WebFlux1/websocket/binary/WSClientPerson

Using WebSocket to retrieve binary data

@Configuration

```
public class WSPersonConfiguration {
```

@Bean

```
public HandlerMapping wsPerson() {
```

```
    return new SimpleUrlHandlerMapping(
```

```
        Map.<String, WebSocketHandler>of(
```

```
            "/wsperson", session -> {
```

```
                Flux<Long> idFlux = session.receive()
```

```
                    .map(WebSocketMessage::getPayloadAsText)
```

```
                    .map(Long::valueOf);
```

```
                Flux<Person> personFlux = idFlux.concatMap(personRepository::findById);
```

```
                Flux<WebSocketMessage> personMessages =
```

```
                personFlux.map(v -> session.binaryMessage(
```

```
                    dataBufferFactory ->
```

```
                    dataBufferFactory.wrap(v.toString().getBytes())
```

```
                ));
```

```
            return session.send(personMessages);
```

```
        }, 1);
```

```
    }
```

```
}
```

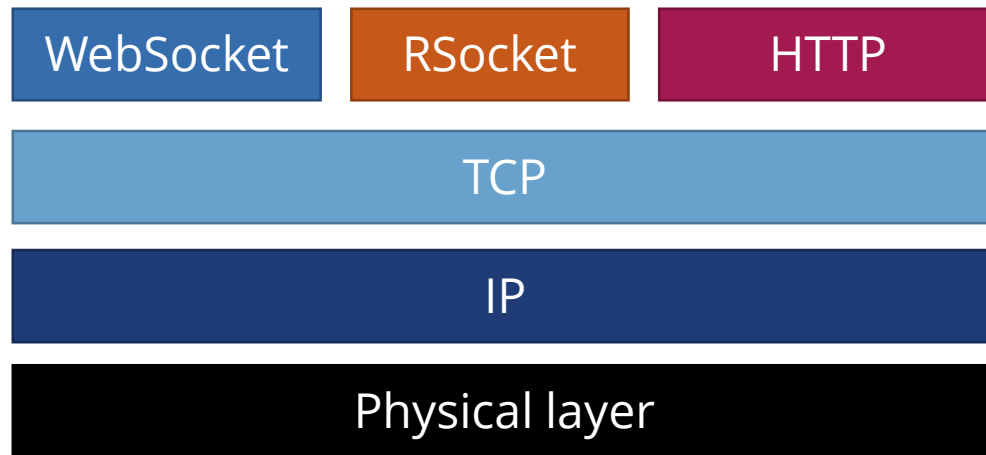
Convert String to binary data

WebFlux1/websocket/binary/WSPersonConfiguration

RSocket protocol

RSocket is a binary protocol for use on byte stream transports such as TCP, WebSockets, and Aeron.

RSocket implements the Reactive Streams specification over the network boundary. It is an application-level communication protocol with **framing**, **session resuming**, and **backpressure built-in** that works over the network. Also RSocket supports **load balancing**.



Application layer: guarantee of delivery (or no delivery)

Transport layer: no guarantee that data is delivered and in which order

Network layer

RSocket vs. WebSocket protocol

Websockets do not provide **application-level backpressure**, only TCP-based byte-level backpressure.

Websockets also only provide framing, they do not provide application semantics. It is up to the developer to build out an application protocol for interacting with the websocket.

RSocket provides framing, application semantics, application-level backpressure, and it is not tied to a specific transport.

With **WebSocket**, we need to create our **own communication protocol**.

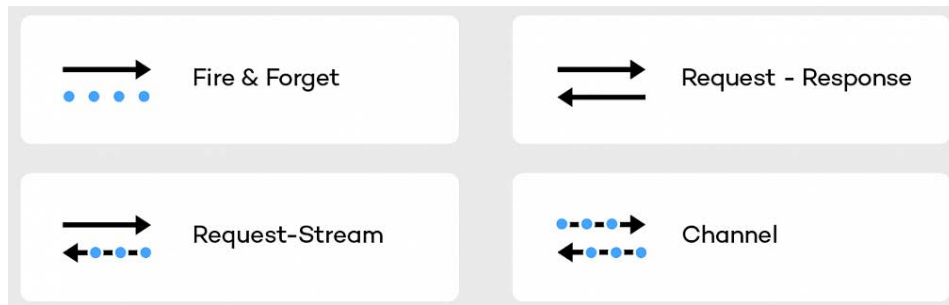
With **RSocket**, we are able to use a **standard protocol** (close to REST).

Overall, RSocket can be perceived as a “reactive REST”.

RSocket interface

RSocket enables the following symmetric interaction models via async message passing over a single connection:

- request/response (stream of 1)
- request/stream (finite stream of many)
- fire-and-forget (no response)
- channel (bi-directional streams)



Dependency (Gradle):

implementation 'org.springframework.boot:spring-boot-starter-rsocket'

```
public interface RSocket extends Availability, Closeable {  
  
    Mono<Void> fireAndForget(Payload payload);  
  
    Mono<Payload> requestResponse(Payload payload);  
  
    Flux<Payload> requestStream(Payload payload);  
  
    Flux<Payload> requestChannel(Publisher<Payload> payloads);  
  
}
```

Low-level examples can be found at
Examples/RSocket/Standalone

RSocket controller: server side

```
@RestController
@RequestMapping("/")
public class PersonRSocketController {
    private final PersonRepository personRepository;

    @GetMapping("findById") // request response
    Mono<Person> getPersonById(Long id) {
        return personRepository.findById(id);
    }

    @GetMapping("all") // request stream
    Flux<Person> getAllPersons() {
        return personRepository.findAll();
    }

    @GetMapping("add") // fire and forget
    Mono<Void> addPerson(Person person) {
        personRepository.save(person);
        return Mono.empty();
    }

    @GetMapping("byIds") // channel
    Flux<Person> getPersonByIds(Flux<Long> ids) {
        return ids.flatMap(personRepository::findById);
    }
}
```

RSocket enables the following symmetric interaction models via async message passing over a single connection:

- request/response (stream of 1)
- request/stream (finite stream of many)
- fire-and-forget (no response)
- channel (bi-directional streams)

RSocket client

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .encoders(encoders -> encoders.add(new
Jackson2JsonEncoder()))
    .decoders(decoders -> decoders.add(new
Jackson2JsonDecoder()))
    .build();
```

First decoder
defined is used as
a default data
MIME type – JSON
in this case

```
RSocketRequester rsocketRequester = RSocketRequester.builder()
    //dataMimeType(MediaType.APPLICATION_JSON)
    .rsocketStrategies(strategies)
    .connectTcp("localhost", 7000)
    .block();
```

Waiting to connect

```
System.out.println("=== Request-Response: Find by id");
//request response
rsocketRequester.route("findById")
    .data(1)
    .retrieveMono(Person.class)
    .subscribe(System.out::println);
```

```
Thread.sleep(1000);
```

```
System.out.println("=== Request Stream: retrieve all person");
// request stream
```

```
rsocketRequester
    .route("all")
    .retrieveFlux(Person.class)
    .subscribe(System.out::println);
```

```
Thread.sleep(1000);
```

```
System.out.println("=== Channel: get persons by id");
// channel
```

```
rsocketRequester.route("byId")
    .data(Flux.just(3,2,1))
    .retrieveFlux(Person.class)
    .subscribe(System.out::println);
```

```
Thread.sleep(1000);
```

WebFlux1/rsocket/RSocketClient

RSocket client with binary data exchange

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .encoders(encoders -> encoders.add(new Jackson2CborEncoder()))
    .decoders(decoders -> decoders.add(new Jackson2CborDecoder()))
    .encoders(encoders -> encoders.add(new Jackson2JsonEncoder()))
    .decoders(decoders -> decoders.add(new Jackson2JsonDecoder()))
    .build();
```

```
RSocketRequester rsocketRequester = RSocketRequester.builder()
    //dataMimeType(MediaType.APPLICATION_JSON)
    .rsocketStrategies(strategies)
    .connectTcp("localhost", 7000)
    .block();
```

First decoder defined is used as a default data MIME type – **CBOR in this case**

It's still possible to change a data type (but appropriate encoder/decoder should be defined)


Concise Binary Object Representation (CBOR) is a binary data serialization format loosely based on JSON. Like JSON it allows the transmission of data objects that contain name-value pairs, but in a more concise manner. This increases processing and transfer speeds at the cost of human-readability.

RSocket controller: client side

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .encoders(encoders -> encoders.add(new Jackson2JsonEncoder()))
    .decoders(decoders -> decoders.add(new Jackson2JsonDecoder()))
    .build();
```

```
RSocketRequester rsocketRequester = RSocketRequester.builder()
    //dataMimeType(MediaType.APPLICATION_JSON)
    .rsocketStrategies(strategies)
    .connectTcp("localhost", 7000)
    .block();
```

First decoder defined is used
as a default data MIME type –
JSON in this case



```
System.out.println("=== Request-Response: Find by id");
//request response
rsocketRequester.route("findById")
    .data(1)
    .retrieveMono(Person.class)
    .subscribe(System.out::println);
```

```
Thread.sleep(1000);
```

```
System.out.println("=== Request Stream: retrieve all persons");
// request stream
rsocketRequester
    .route("all")
    .retrieveFlux(Person.class)
    .subscribe(System.out::println);
```

```
Thread.sleep(1000);
```

```
System.out.println("=== Channel: get persons by id");
// channel
rsocketRequester.route("byId")
    .data(Flux.just(3,2,1))
    .retrieveFlux(Person.class)
    .subscribe(System.out::println);
```

```
Thread.sleep(1000);
```

WebFlux1/rsocket/RSocketClient

RSocket with load balancing

```
Function<Integer, RSocket> getConnector = port -> RSocketConnector
    .create()
    .dataMimeType(MimeTypeUtils.APPLICATION_JSON_VALUE)
    .metadataMimeType(MIME_ROUTER)
    .connect(TcpClientTransport.create(port))
    .doOnSubscribe(s -> System.out.println(
        "RSocket connection established on port " + port))
    .block();
```

```
List<RSocketSupplier> socketSuppliers = Flux.just(7000, 7001)
    .map(port -> new RSocketSupplier(
        ()->Mono.just(getConnector.apply(port))))
    .collectList().block();
```

```
LoadBalancedRSocketMono balancer =
    LoadBalancedRSocketMono
        .create(Flux.just(socketSuppliers));
```

```
@MessageMapping("findById2") // request response
Mono<Person> getPersonByIdForLoadBalancing(Long id) {
    return personRepository.findById(1L).map(person ->
        new Person(port, id, person.getName(), person.getSurname()));
}
```

```
AtomicInteger id = new AtomicInteger(1);
Flux.range(1,4)
    .flatMap(i -> balancer)
    .map(rSocket ->
        RSocketRequester.wrap(rSocket,
            MimeTypeUtils.APPLICATION_JSON,
            MimeType.valueOf(MIME_ROUTER),
            strategies))
    .doOnNext(rSocket -> rSocket
        .route("findById2")
        .data(id.getAndIncrement())
        .retrieveMono(Person.class)
        .doOnNext(System.out::println)
        .block()
    )
    .blockLast();
```

```
RSocket connection established on port 7001
RSocket connection established on port 7000
Person(id=1, name=Vasya, surname=Poupkin, port=7000)
Person(id=2, name=Vasya, surname=Poupkin, port=7001)
Person(id=3, name=Vasya, surname=Poupkin, port=7001)
Person(id=4, name=Vasya, surname=Poupkin, port=7000)
```

RSocketLB

Weighted load balancing

```
Function<Integer, Mono<RSocket>> getConnector = port ->
    Mono.from(RSocketConnector
        .create()
        // .reconnect(Retry.fixedDelay(100, Duration.ofSeconds(1)))
        .dataMimeType(MimeTypeUtils.APPLICATION_JSON_VALUE)
        .metadataMimeType(MIME_ROUTER)
        .connect(TcpClientTransport.create(port))
        .doOnSubscribe(s -> System.out.println("RSocket connection established on port " + port))
    );

LoadBalancedRSocketMono balancer = create(
    Flux.just(7000, 7001).map(port -> new RSocketSupplier(() -> getConnector.apply(port))).collectList()
);

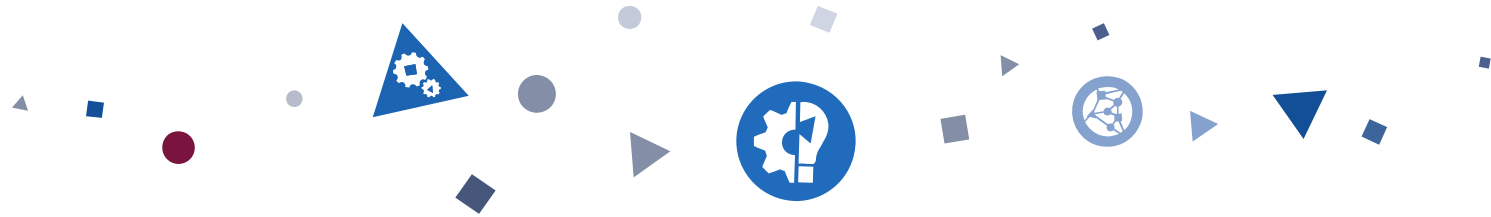
// we need to wait at least 1 RSocket in the balancer
while (balancer.availability() == 0.0) {
    Thread.sleep(1);
}
```

```
@Value("${spring.rsocket.server.port}") Long port;

@MessageMapping("findById3") // request response
Mono<Person> getPersonByIdForLoadBalancing(Long id) {
    if (port == 7000) sleep(50);
    return personRepository.findById(1L).map(person ->
        new Person(port, id, person.getName(), person.getSurname()));
}
```

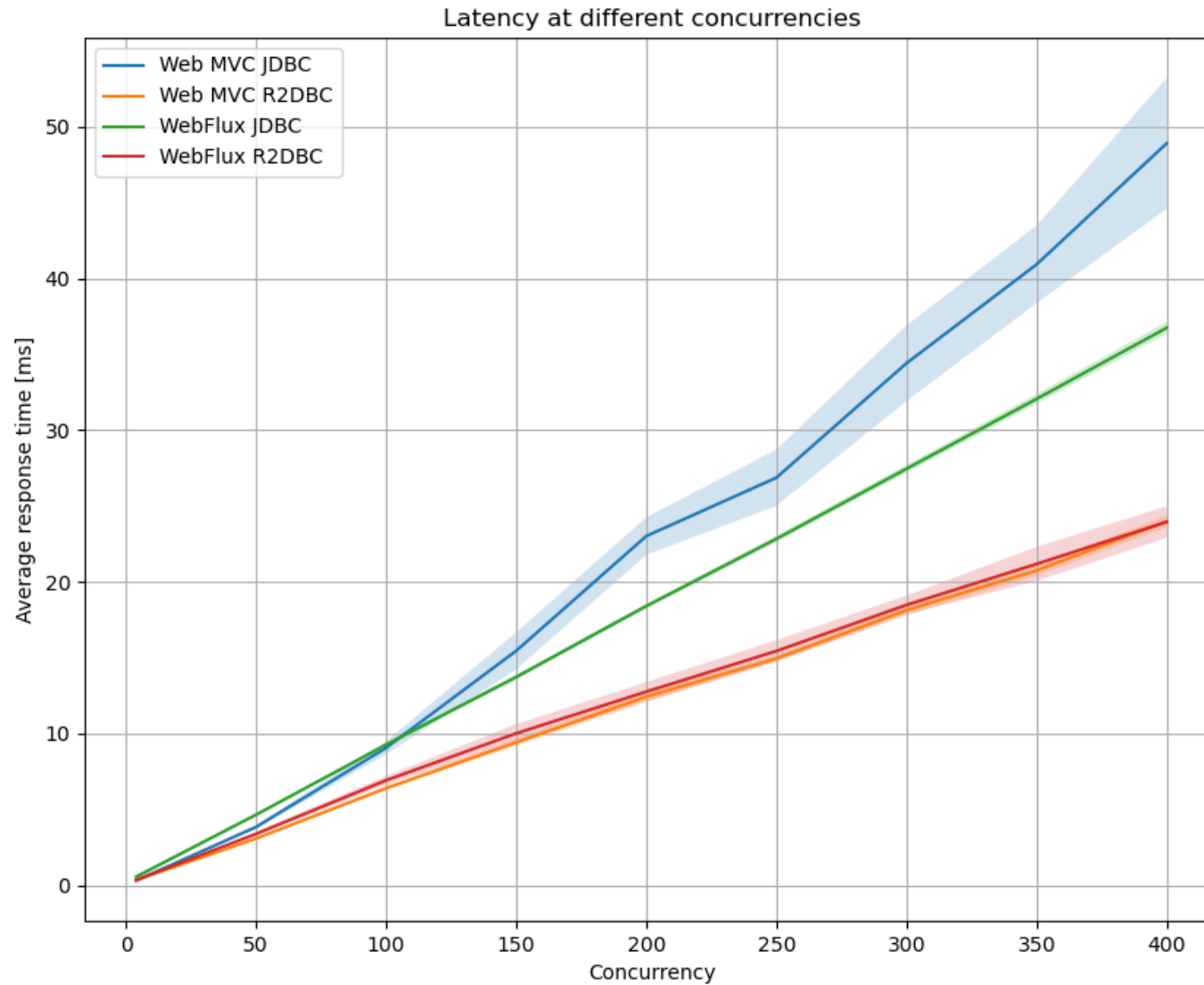
```
AtomicInteger id = new AtomicInteger(1);
Flux.range(1, 5000)
    .flatMap(i -> balancer)
    .retry()
    .map(rSocket ->
        RSocketRequester.wrap(rSocket,
            MimeTypeUtils.APPLICATION_JSON,
            MimeType.valueOf(MIME_ROUTER),
            strategies))
    .flatMap(rSocket ->
        rSocket
        .route("findById2")
        .data(id.getAndIncrement())
        .retrieveMono(Person.class)
        .retryWhen(Retry.fixedDelay(
            10, Duration.ofSeconds(20)))
    )
    .groupBy(Person::getPort)
    .flatMap(port ->
        Mono.zip(Mono.just(port.key()), port.count()))
    .map(p -> p.getT1() + ": " + p.getT2())
    .doOnNext(System.out::println)
    .blockLast();
```

```
RSocket connection established on port 7001
RSocket connection established on port 7000
7000: 268
7001: 4732
```



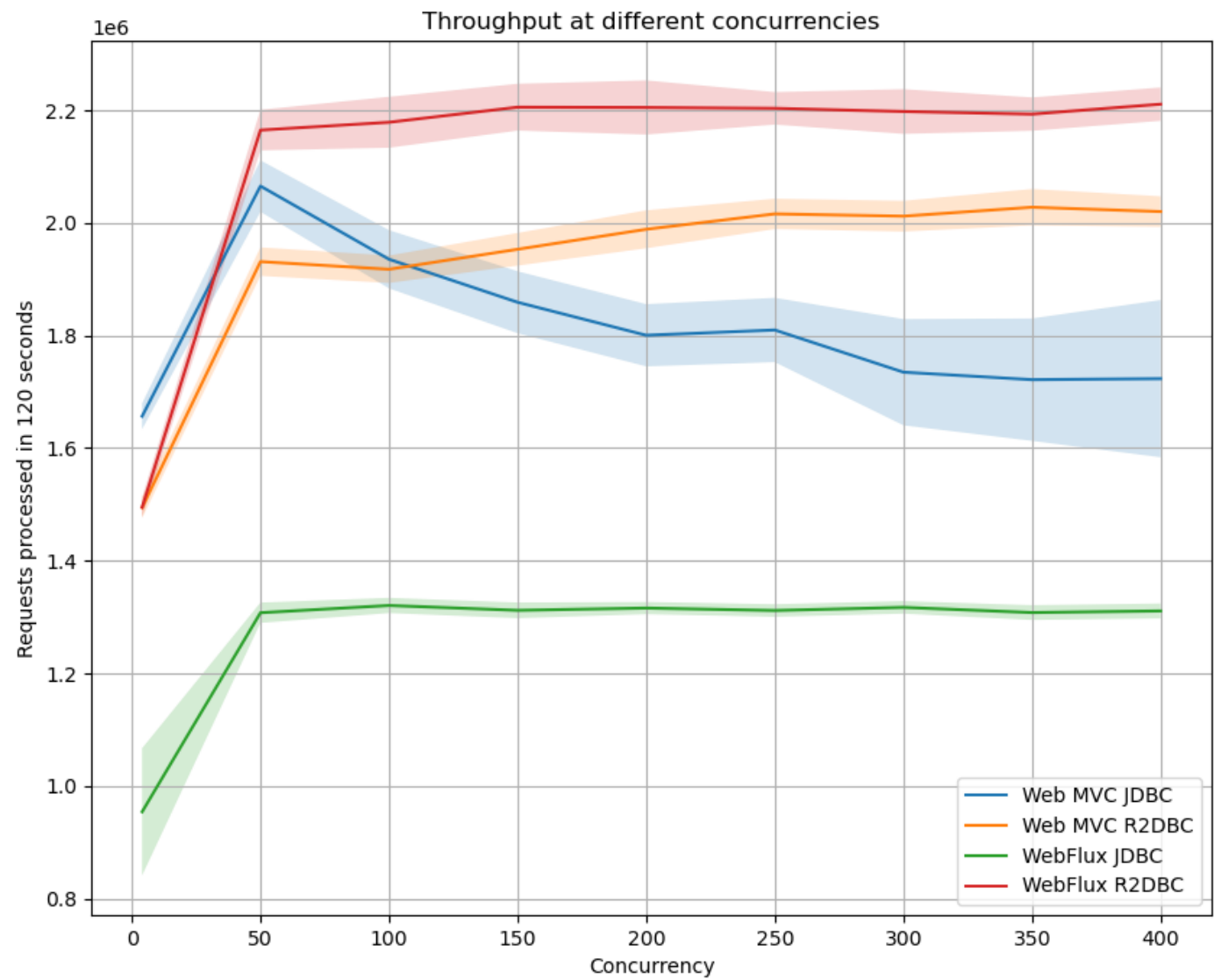
WEBFLUX BENCHMARKS

LATENCY WITH DIFFERENT APPROACHES



Source: <https://technology.amis.nl/2020/04/10/spring-blocking-vs-non-blocking-r2dbc-vs-jdbc-and-webflux-vs-web-mvc/>

THROUGHPUT WITH DIFFERENT APPROACHES



SUMMARY

R2DBC and WebFlux, a good idea at high concurrency!

- At high concurrency, the benefits of using R2DBC instead of JDBC and WebFlux instead of Web MVC are obvious.
 - Less CPU is required to process a single request.
 - Less memory required to process a single request.
 - Response times at high concurrency are better.
 - Throughput at high concurrency is better
 - The fat JAR size is smaller (no JPA with R2DBC)
- When using only blocking components, memory and CPU usage will become less efficient at high concurrency
- WebFlux with JDBC does not appear to be a good idea. Web MVC with R2DBC works better at high concurrency than Web MVC with JDBC.
- You're not required to have a completely non-blocking stack to reap the benefits of using R2DBC. It is however best to combine it with WebFlux in case of Spring.
- At low concurrency (somewhere below 200 concurrent requests), using Web MVC and JDBC, might give better results. Test this to determine your own break-even point!



Thank You!

think.
create.
accelerate.



Luxoft | training
A DXC Technology Company