Reaktive Datenbank Zugriffe

Überblick

Reaktive / Asynchrone Datenbank-Zugriffe

- Blocking I/Os - Non-Blocking I/Os
- Asynchroner Treiber ADBA (alternative mit Oracle fibers)
- Project Loom für Virtual Threads ab JDK 16/17

Reaktive-Webstack

- Server Seite
- Blocking Requests vs non-Blocking i/o
- Server-Unterstützung
- Unter der Haube

R2DBC

- Fakten
- Spring Data Konfiguration
- Zugriff mit reaktivem Client
- Batch, Verifier
- Transaktionen
- Operatoren und Event Handling

Reaktive-WebClients

- Consuming mit RxJS in Angular
- Consuming mit Spring WebClient

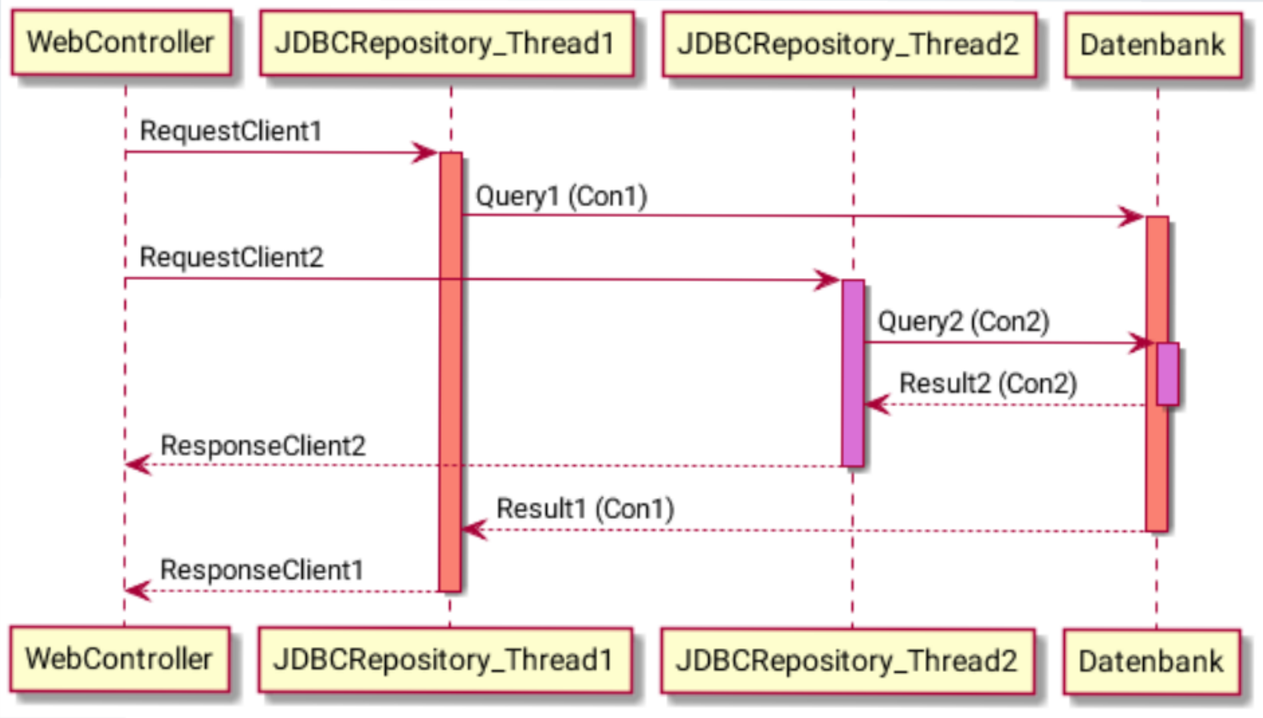
Benchmark

- Hibernate / JDBC / sql2o + R2DBC mit Spring MVC und SpringWebflux
- Testumfang mit Apache Benchmark

Reaktive / Asynchron Datenbank Zugriffe

- Blocking I/Os - Non-Blocking I/Os

Blocking I/Os

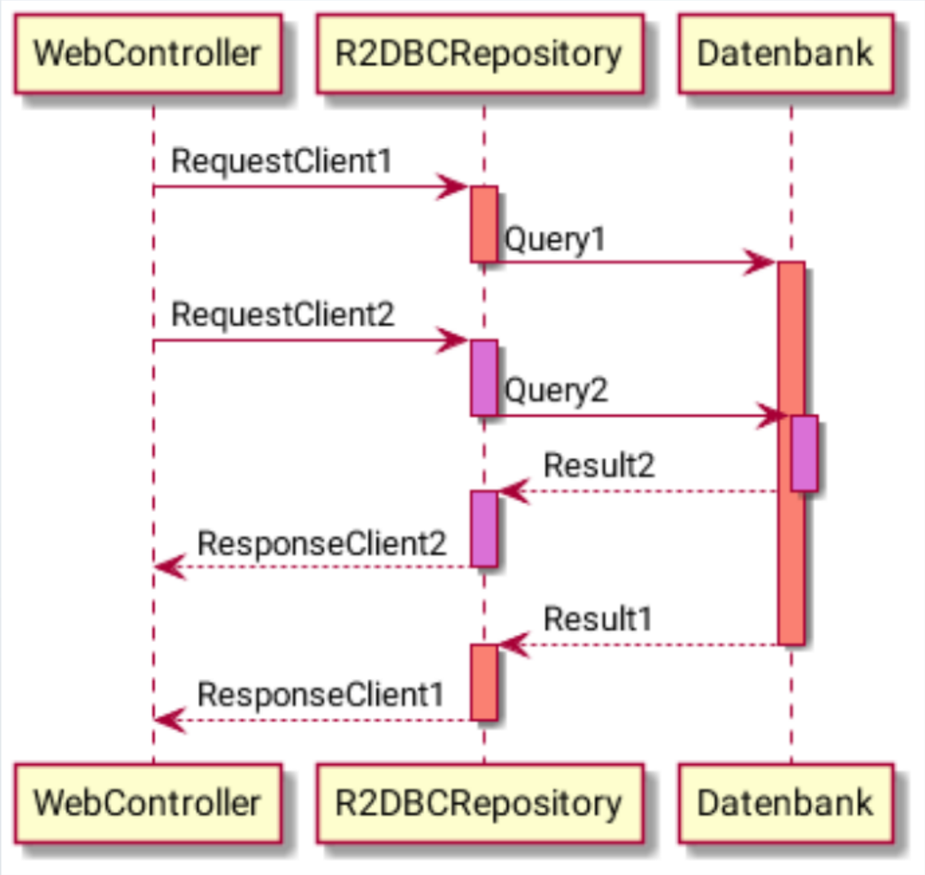


Blocking I/Os

Blocking Threads für Datenbank-Verbindung

- Zugriffe über JDBC:
  I/O blockierende System Threads für Schreib-/Lese- Zugriffe und führt zu neuem Thread für jede neue Connection.
- System Thread Switch kostet auch Zeit und CPU.
- Ein Connection Pool wird dadurch nicht optimal benutzt.

Non Blocking I/Os



Non Blocking I/Os

Non Blocking Threads für Datenbank-Verbindung

- Reaktive Treiber und Clients ermöglichen, dass der Client-Prozess nicht blockiert wird und wie ein EventListener auf einen Rückruf des lesenden Prozesses reagiert.
- Besonders interessant vor allem wenn vielen Clients parallel zugreifen wollen, ohne blockiert zu werden.
- Besonders relevant für Zugriffe aus immer häufiger Reaktiv Web Stacks (seit WebServlet3.1).
- Datenbankzugriff ist dann Teil der „reaktiven“ Kette.

Reaktive / Asynchron Datenbank-Zugriffe

- Asynchroner Treiber ADBA
- Project Loom

Asynchroner Treiber ADBA

- ADBA (Asynchronous Database Access) bis Java 12
- Von Oracle seit September 2019 nicht mehr unterstützt.

```
public DataSource getDataSource(String url, String user, String pass)
{
   return DataSourceFactory.forName("Oracle Database")
   .builder()
   .url("oracle:database:@//javaone.oracle.com:5521/javaone")
   .username("scott")
   .password("tiger")
   .connectionProperty(JdbcConnectionProperty.TRANSACTION_ISOLATION,TransactionIsolation.SERIALIZABLE)
   .build();
}
```

Asynchroner Treiber ADBA

- ADBA (Asynchronous Database Access) ab Java 12
- Von Oracle seit September 2019 nicht mehr unterstützt.

```
public Future<Integer>selectIdForAnswer(DataSource ds,int answer)
{
   String sql = "selectid, name, answer from tab" +"where answer = @target";
   try(Connection conn = ds.getConnection()) {
    return conn
    .<List<Integer>>rowOperation(sql)
    .set("target", 42, JdbcType.NUMERIC)
    .initialValue( () -> newArrayList<>() )
    .rowAggregator( (list, row) -> {
       list.add(row.get("id", Integer.class));
       return list;
    } )
    .submit()
    .toCompletableFuture()
    .thenApply( l ->l.get(0) );
   }
}
```

Project Loom (Fibers/ Virtual Threads ab Java 17)

- Leichtgewichtige Virtual Threads, ähnlich wie Coroutines von Kotlin.
- Standard net/nio Libraries verwenden es dann intern.
- JDBC Clients sollen dann über Virtual Threads implementiert werden.

```
ThreadFactory factory = Thread.builder().virtual().factory()
```

java.net.Socket
java.net.ServerSocket
java.net.DatagramSocket/MulticastSocket
java.nio.channels.SocketChannel
java.nio.channels.ServerSocketChannel
java.nio.channels.DatagramChannel
java.nio.channels.Pipe.SourceChannel
java.nio.channels.Pipe.SinkChannel
java.net.InetAddress


R2DBC

DEMO

R2DBC

- Fakten
- Spring Data Konfiguration
- Zugriff mit reaktivem Client
- Batch, Verifier
- Transaktionen
- Operatoren und Event Handling

Fakten

Reactive Treiber und Clients:

- SPI APIs + Client APIs
- MongoDB
- Vert.X Client
- Wird von Hibernate Reactive verwendet (auch im Quarkus Reactive database client…gute Kombination mit GraalVM und Vert.XWeb route)

R2DBC

- Seit 2017, aus Pivotal Initiative
- Drivers
  - jasync (Kotlinwrapper für Postgres & MariaDB)
  - Native Treiber (PostgreSQL, Oracle, h2, MariaDB, MySQL, MS-SQL)
- Clients:
  - PureReactor
  - **Spring-data**, Kotysa(Kotlin), JOOQ
- Version 0.9 Release im September 2021 (StoredProcedures)

Spring Data R2DBC - Konfiguration

```
public class R2DBCConfiguration extends AbstractR2dbcConfiguration {
@Bean
 public ConnectionFactory connectionFactory() {
ConnectionFactory factory = ConnectionFactories
.get(ConnectionFactoryOptions.builder()
        .option(ConnectionFactoryOptions.DRIVER, "pool")
        .option(ConnectionFactoryOptions.PROTOCOL, "postgresql")
        .option(ConnectionFactoryOptions.HOST, "localhost")
        .option(ConnectionFactoryOptions.USER, "postgres")
        .option(ConnectionFactoryOptions.PASSWORD, "postgres4jmk!")
        .option(ConnectionFactoryOptions.DATABASE, "testDB")
        .option(MAX_SIZE, 400)
        .build());
```

```
ConnectionPoolConfiguration configuration =
ConnectionPoolConfiguration.builder(factory)
                .maxSize(400)
                .maxCreateConnectionTime(Duration.ofSeconds(1))
                .clock(Clock.systemDefaultZone())
                .build();
   return new ConnectionPool(configuration);
}
```

Spring Data R2DBC Direktzugriff mit Client

```
ConnectionFactory factory = new R2DBCConfiguration().connectionFactory();
DatabaseClient client = DatabaseClient.create(factory);
ReactiveTransactionManager tm = new R2dbcTransactionManager(factory);
TransactionalOperator operator = TransactionalOperator.create(tm);
```

```
client.execute("CREATE TABLE player" +
        "(id BIGINT GENERATED BY DEFAULT AS IDENTITY," +
        "name VARCHAR(255)," +
        "age INT,"+
        "photo BYTEA)")
      .fetch()
      .rowsUpdated()
      .as(StepVerifier::create)
      .expectNextCount(1)
      .verifyComplete();
```

Spring Data R2DBC - Batch nur mit SPI

```
Flux.from(factory.create())
      .flatMap(con ->
        Flux.from(con.createBatch()
          .add("insert into player(name, age)" +  "values('Kaka', 37)")
          .add("insert into player(name, age)" +  "values('Messi', 32)")
          .add("insert into player(name, age)" +  "values('Mbappé', 32)")
          .add("insert into player(name, age)" +  "values('CR7', 32)")
          .execute())
        .doFinally((st) -> con.close())
       )
      .log()
      .blockLast();
```

```
client.insert()
     .into(Player.class)
     .using(new Player(null,"Lewandowski", 32, lewandowskiImage))
     .fetch()
     .rowsUpdated()
     .then()
     .as(StepVerifier::create)
     .expectNextCount(1)
     .verifyComplete();
```

Transaktionen über SPI oder Client

```
Mono.from(factory.create())
 .flatMap(c -> Mono.from(c.beginTransaction())
  .then(Mono.from(c.createStatement("insert into player(name, age)
       values('Griezmann',29)")
    .returnGeneratedValues("id")
    .execute()))
  .map(result -> result.map((row, meta) ->
     new Player(row.get("id", Long.class),
     testPlayer06.getName(),
     testPlayer06.getAge())))
  .flatMap(pub -> Mono.from(pub))
  .delayUntil(r -> c.commitTransaction())
  .doFinally((st) -> c.close()));
```

```
client.execute("insert into player(name, age, photo) values(:name, :age,
:photo)")
        .bind("name", testPlayer06.getName())
        .bind("age", testPlayer06.getAge())
        .bind("photo", testPlayer06.getPhoto())
    .fetch()
    .rowsUpdated()
    .then()
    .as(operator::transactional)
    .subscribe(
          data -> log.info("data saved"),
          err -> log.error("err: {}", err)
       );
```

Operatoren und Event Handling

```
Flux<Player> players =
client.select()
  .from(Player.class)
  .matching(Criteria.where("name").like("M%")
    .and("age").in(32, 34))
  .orderBy(Sort.Order.desc("id"))
  .fetch()
  .all()
  .doOnNext(p -> System.out.println("inline output: "+p.getName()))
  .doOnError(System.out::println)
  .doOnComplete(System.out::println)
  .onErrorReturn(errorPlayer)
  .log()
  .onBackpressureBuffer(5);
```

```
players.subscribe(System.out::println);
players.subscribe(p -> log.info("Second subscriber: "+p.getName()));
```

DEMO

Reaktiver Webstack

- Server-Seite
- Blocking Requests vs non-Blocking i/o
- Server-Unterstützung
- Unter der Haube

Server Seite

- RxJava (Ab spring 5)
- Smallrye Mutiny (Vert.X)
- Reactor: Spring Webflux

```
@Repository
public interface R2DBCPlayerRepositoryItf extends ReactiveCrudRepository<Player, Integer> {
@Query("select player.id, player.name, player.age, player.photo from player where player.name = :name")
Flux<Player> findAllByName(String name);
```

```
@Autowired private R2DBCPlayerRepositoryItf playerRepository;
public Flux<Player> findAllPlayers(String name){
  return playerRepository.findAllByName(name);
}
```

```
@RestController
@RequestMapping("WebFluxPlayers")
public class R2DBCWebFluxPlayerController {
    @Autowired private R2DBCPlayerService playerDao;
    @GetMapping("/R2DBC/{name}")
     public Flux<Player> getAllPlayers() {
        return playerDao.findAllPlayers(name);
}
```

Blocking Requests (über Spring MVC)
oder
non-Blocking i/o (über Spring WebFlux)

```
@RequestMapping(„MVCPlayers")
@GetMapping("/R2DBC/all")
public List<Player> getAllPlayers() {
      return playerDao.findAllPlayers()
.subscribeOn(Schedulers.boundedElastic())
.blockLast();
}
@GetMapping("/R2DBC/{id}")
 public Optional<Player> getPlayer(@PathVariable("id") Long id) {
      return playerDao.findById(id)
.subscribeOn(Schedulers.boundedElastic())
.blockOptional();
}
```

```
@RequestMapping("WebFluxPlayers")
@GetMapping("/R2DBC/all")
public Flux<Player> getAllPlayers() {
      return playerDao.findAllPlayers();
}
@GetMapping("/R2DBC/{id}")
 public Mono<Player> getPlayer(@PathVariable("id") Long id) {
      return playerDao.findById(id);
}
```

Server Unterstützung

- Netty (Reactor Netty)
- Tomcat ab 8.5 (Servlet 3.1+)
- Jetty (Servlet 3.1+)

...

Unter der Haube

- Ab Servlet 3.1. Async IO und Spring 5
- Kleineren Thread Pool serverseitig
- **ReadListeners** auf ServletInputStream (pro Socket-Kanal):
  Sobald der Request-Buffer vom Client befüllt worden ist, kann der nächste Request vom nicht-blockierte Thread gelesen und behandelt werden.
  Der Servlet Container triggert dabei die Methode **onReadPossible()** vom Listener.
- **WriteListeners** auf ServletOutputStream (pro Socket-Kanal):
  Sobald der Response-Buffer vom Client geleert worden ist, kann der nächste Response vom nicht-blockierten Thread geschrieben werden.
  Der Servlet Container triggert dabei die Methode **onWritePossible()** vom Listener.

Reaktive WebClients

- Consuming mit RxJS in Angular
- Consuming mit Spring WebClient

DEMO

Beispiel Consuming mit RxJS in Angular
(Single / Observables)

```
constructor(private readonly http: HttpClient){};
getAllWebFluxR2DBCPlayers() : Observable<Player[]>{
   return this.http.get<Player[]>
('http://localhost:4200/WebFluxPlayers/R2DBC/all');
}
```

```
players$: Observable<Player[]>;
this.playerData.getAllWebFluxR2DBCPlayers()
   .subscribe({
     next: (data) => this.successFn(data),
     error: (message) => console.log(message),
     complete: () => console.log("finished")
   });
successFn(data) {
   this.players$ = data;
   console.log(this.players$);
  }
```

Beispiel Consuming mit Spring Reactive Web Client

```
Flux<Player> players = client.get()
        .uri("http://localhost:8080/WebFluxPlayers/R2DBC/all")
        .accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .bodyToFlux(Player.class)
        .doOnNext(p -> System.out.println("inline output findAll: "+p.getName()));
players.subscribe(p -> System.out.println(p.getName()));
Mono<Player> player = client.get()
        .uri("http://localhost:8080/WebFluxPlayers/R2DBC/{id}", 1)
        .accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .bodyToMono(Player.class);
System.out.println("Player from retrieve: " + player.block().name);
```

Benchmark

- Hibernate / JDBC / sql2o + R2DBC mit Spring MVC und SpringWebflux
- PostgreSQL
- Testumfang mit Apache Benchmark

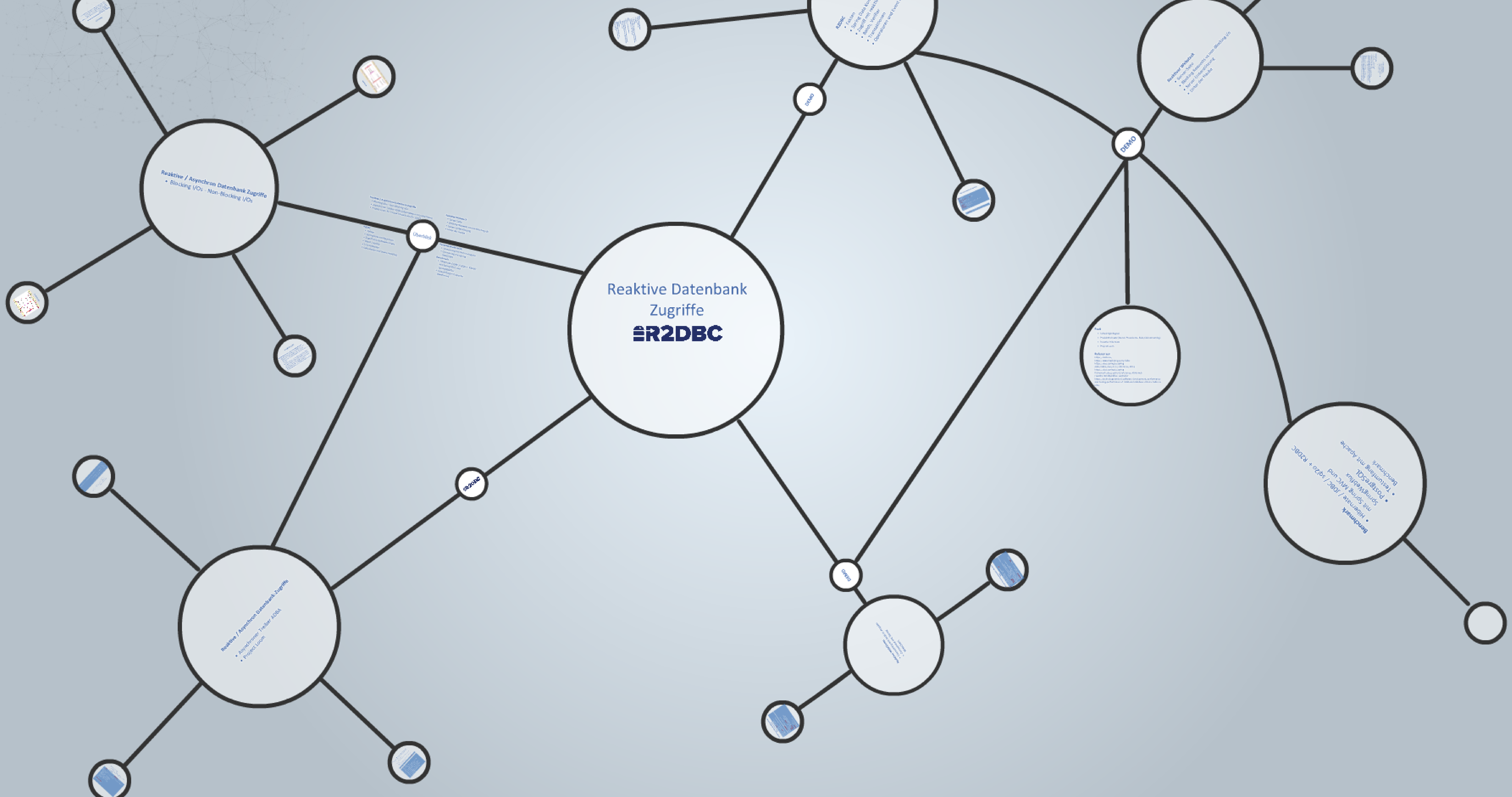
Fazit

- Schwierigkeitsgrad
- Produktiveinsatz (Stored Procedures, Blob/Clob streaming)
- Reactive Hibernate
- Project Loom

Referenzen

https://r2dbc.io/
https://www.baeldung.com/r2dbc
https://docs.spring.io/spring-data/r2dbc/docs/1.0.x/reference/html/
https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-controller
https://technology.amis.nl/software-development/performance-and-tuning/performance-of-relational-database-drivers-r2dbc-vs-jdbc/

Fragerunde


Reaktive Datenbank Zugriffe
R2DBC
Reaktive / Asynchron Datenbank Zugriffe
Blocking I/Os - Non-Blocking I/Os
DEMO
DEMO