Hibernate Reactive und Kotlin: elegant und performant

Juergen.Zimmermann@h-ka.de

7. Juli 2022

Prof. Dr. Jürgen Zimmermann
Hochschule Karlsruhe
Fakultät Informatik und Wirtschaftsinformatik
Moltkestraße 30
76133 Karlsruhe
http://www.h-ka.de

Zum Referenten

- Prof. für Softwarearchitektur und Software Engineering Hochschule Karlsruhe
- Practice Manager des Sun Java Center
- Seniorberater bei Capgemini
- Consulting Manager bei Object Design
- Dissertation an der TU Darmstadt

Gliederung

1. Grundlagen
   - Kotlin
   - Impedance Mismatch und OR-Mapping
   - Reactive Programming
2. Hibernate Reactive mit Kotlin
   - Annotationen aus Jakarta Persistence (JPA)
   - find
   - Criteria Queries
   - persist, merge, remove
3. Fazit

1 Grundlagen

1. Kotlin
2. Impedance Mismatch und OR-Mapping
3. Reactive Programming

1.1 Kotlin

- Einfach, verständlich und robust
- Einfache Nutzung von (Java-) Bibliotheken
- Properties
- Funktionen auch außerhalb von Klassen

- Higher Order Functions und Lambda-Ausdrücke
- Null Safety und der Elvis-Operator
- Extension Functions
- Sealed Interfaces und Exhaustive When statt Exceptions
- Suspend Functions für Reactive Programming
- …

Higher Order Functions und Lambda-Ausdrücke

```
kunden.filter  { kunde -> kunde.adresse.plz[0] == '7' }
      .flatMap { kunde -> kunde.bestellungen }
      .filter  { bestellung -> bestellung.summe > 100 }
      .map     { bestellung -> bestellung.lieferant }
      .forEach { lieferant -> … }
```

Null-Safety und der Elvis-Operator

```
val id = …
val kunde = session.find<Kunde>(id)   // Kunde?
            ?: return NotFound(id)    // Elvis-Op.

println("Name: ${kunde.name}")        // null safety
```

Sealed Interface und Exhausive When

```
sealed interface CreateResult
data class Created(val kunde: Kunde) : CreateResult
data class ConstraintViolations(
                val violations: Set<…>) : CreateResult
```

```
val result = service.create(kunde)      // CreateResult
when (result) {
    is Created -> println(result.kunde) // try
    is ConstraintViolations -> …        // catch
    else -> …
}
```

Extension Functions

```
fun MyList.swap(index1: Int, index2: Int) {
    if (…)
        throw …
    val tmp = this[index1]    // this: MyList
    this[index1] = this[index2]
    this[index2] = tmp
}
```

Analog zu: Swift, C#, …

1.2 Impedance Mismatch und Lösungsansätze

- ***Data Mapper*** mit ***Repository Pattern***
  z.B. durch Jakarta Persistence (JPA)
- ***Active Record***
  - Martin Fowler:
    "Patterns of Enterprise Application Architecture", **2002**

➢ "***ORM is the Vietnam of Computer Science***"

– Ted Neward, **2006**

– https://blogs.tedneward.com/post/the-vietnam-of-computer-science

Nutzung von DB-Systemen mit Spring Data

Quelle: Keynote von Andy Wilkinson, Spring IO **2017**

- 75 % relationale DB-Systeme
- 10 % MongoDB
- 5 % Cassandra
- 5 % ElasticSearch
- 5 % sonstige

OR-Mapping

- Handhabung und Generierung von *Primärschlüssel*n
- *Referenzen* vs. Fremdschlüssel
- *Attributtypen* vs. Spaltentypen
  BigDecimal, URL, Currency, DateTime, …
  Enums und Collections von Enums
- 1:1, 1:N, N:M *Beziehungen*
- *Listen* mit stabiler Reihenfolge
- Eager und Lazy *Fetching*
- *Kaskadierungen* für INSERT, UPDATE, DELETE
- Strategien für *Vererbung*
- …

1.3 Reactive Programming

- Betriebssystem-Threads sind teuer

- Gewünschte Eigenschaften von Anwendungen:
  - ***Keine Blockierungen*** beim Zugriff auf externe Systeme z.B. DB-Systeme, Web Services, …
  - ***Gute Skalierbarkeit*** mit wenigen Threads in einem Pool
  - ***Backpressure*** zur Vermeidung von Überlastung

➢ ***suspend*** Funktionen in Kotlin

Asynchrone Treiber für relationale DB-Systeme

- PostgreSQL
- SQL Server
- MySQL
- MariaDB
- SAP Hana
- Google Cloud Spanner
- Oracle
- Db2

- oft auf Basis von Eclipse Vert.x entwickelt

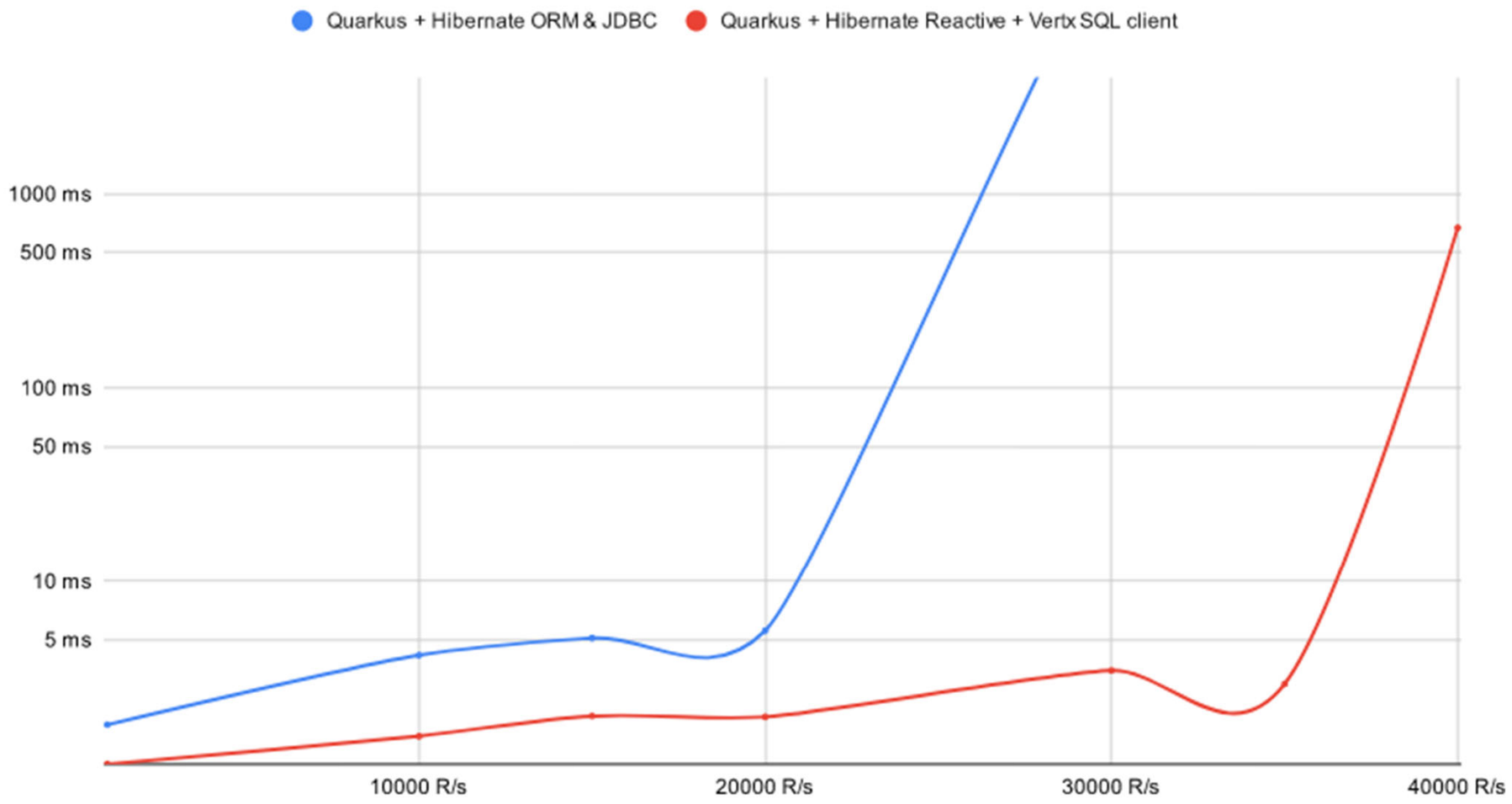

Figure 1. Techempower benchmark, Multiple Queries. This graph plots latency: higher numbers are worse.

Quelle: https://in.relation.to/**2021**/10/27/hibernate-reactive-performance

2 Hibernate Reactive mit Kotlin

1. Hibernate Reactive mit Kotlin
2. Annotationen aus Jakarta Persistence (JPA)
3. find
4. Criteria Queries
5. persist, merge, remove

2.1 Annotationen aus Jakarta Persistence (JPA)

```
@Entity
class Kunde (
    var nachname: String,

    @OneToOne(cascade = [PERSIST, REMOVE])
    @JoinColumn(updateable = false)
    var adresse: Adresse,

    @OneToMany(…, fetch = EAGER)
    @OrderColumn(name = "idx")
    var bestellungen: MutableList<Bestellung>,

    @Id
    @GeneratedValue
    var id: UUID? = null,
)
```

Named Queries

```
@Entity
@NamedQuery(
    name = "byNachname",
    query = """
        SELECT k
        FROM   Kunde k
        WHERE  k.nachname LIKE :nachname
    """,
)
…
class Kunde(…)
```

2.2 find

```
@Service
class KundeService(private val factory: SessionFactory) {
    suspend fun findById(id: KundeId): … {

        val kunde = factory.withSession { session ->
            session.find<Kunde>(id)          // Extension Func.
        }.awaitSuspending()                  // Mutiny

        if (kunde == null) …
        return …
    }
}
```

org.hibernate.reactive.mutiny.Mutiny mit inneren Klassen

- SessionFactory
- Session

Extension Function für find

Statt `session.find(Kunde::class.java, id)`

- `session.find<Kunde>(id)`

```
inline fun <reified E: Any> Session.find(id: Any): Uni<E?> =
    find(E::class.java, id)
```

Typinformation bleibt zur Laufzeit erhalten

analog zu

```
factory.withSession { session ->

    session.createNamedQuery<Kunde>("byNachname")
           .setParameter("nachname", "…")
           .resultList

}.awaitSuspending()
```

2.3 Criteria Queries

```
SELECT k
FROM   Kunde k
WHERE  k.nachname LIKE :nachname
```

```
val criteriaBuilder = factory.criteriaBuilder

val criteriaQuery = criteriaBuilder.createQuery<Kunde>()
val root = criteriaQuery.from()

val predicate =
    criteriaBuilder.like(root.get(Kunde::nachname),
                         "…")
criteriaQuery.where(predicate)

factory.withSession { session ->
    session.createQuery(criteriaQuery)
           .resultList
}.awaitSuspending()
```

Property: Type Safety

***ohne** Metamodel Klassen*

Kotlin JDSL Reactive von LINE Corp.

```
val queryFactory = HibernateMutinyReactiveQueryFactory(…)
queryFactory.withFactory { factory ->
    val kunden = factory.listQuery<Kunde> {  // singleQuery
        select(entity(Kunde::class))
        from(entity(Kunde::class))
        where(column(Kunde::nachname).like("…"))
    }
}
```

- seit Dez. 2021
- https://linecorp.com mit LINE Messenger

2.4 persist, merge, remove

```
factory.withTransaction { session ->
    session.persist(kunde)
}.awaitSuspending()
```

3 Fazit

Besserer Durchsatz durch Reactive Programming

Kotlin als elegantes Java