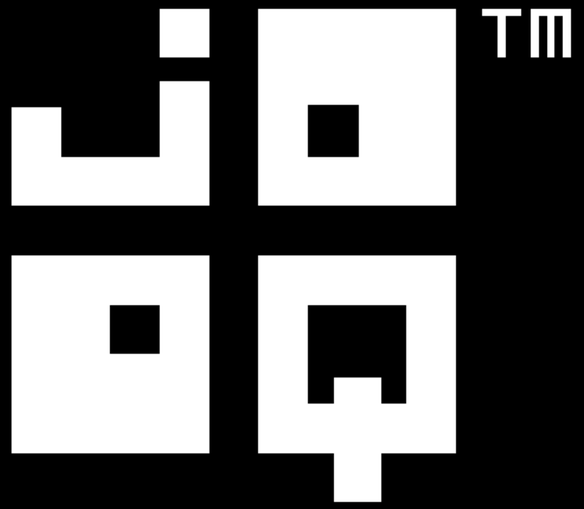
JOOQ

JESPER HOLMBERG

CADEC 2023.01.19 & 2023.01.25 | CALLISTAENTERPRISE.SE

CALLISTA

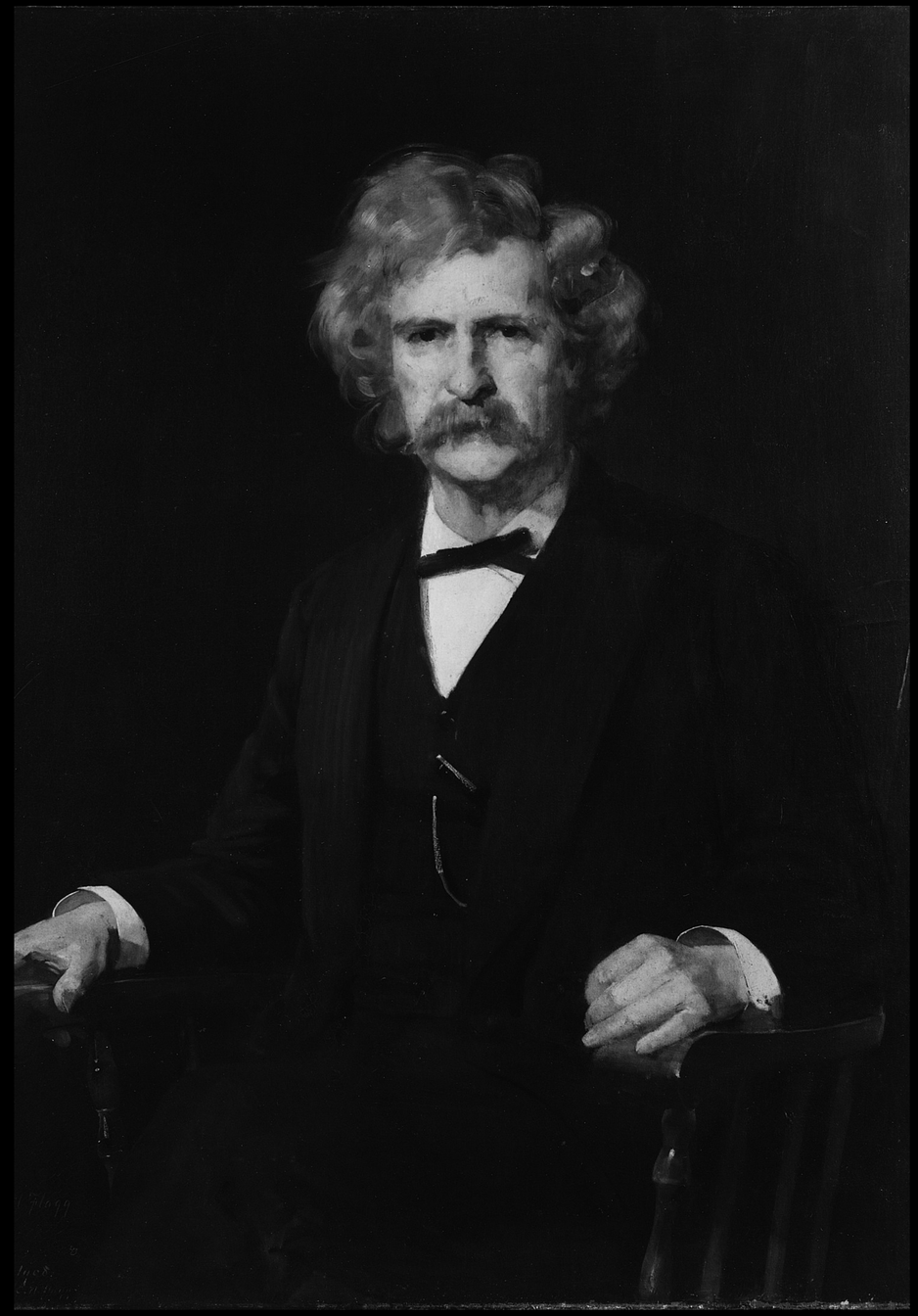
AGENDA



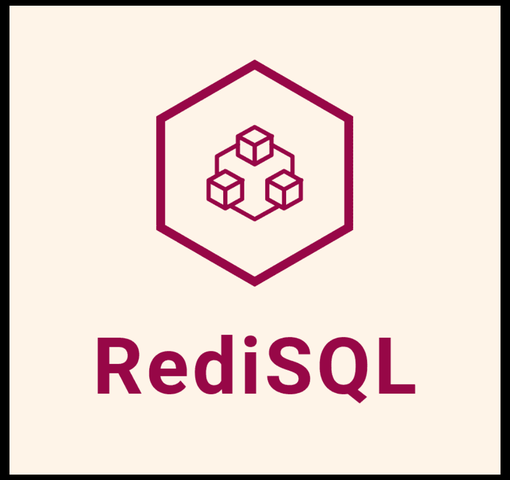
SQL IS STILL ALIVE

"The reports of my death are greatly exaggerated."

- Ten years ago, SQL would soon be dead
- But no general alternative to SQL has appeared
- "No SQL" -> "Not Only SQL" -> "No, SQL!"



SQL INTERFACES AND CLOUD OPTIONS ABOUND



NOT YOUR GRANDMOTHER'S SQL

- SQL has changed
- SQL-99 and later standards have expanded what SQL can be used for
- All relational databases today support
  - Common table expressions (CTE)
  - Recursive CTEs
  - Window functions
  - Lateral joins
  - JSON support
  - etc, etc

A FEW SQL EXAMPLES

FINDING THE MOST RECENT MESSAGE BY TYPE

```
create table message (
    type varchar,
    text varchar,
    time timestamp
);
```

FINDING THE MOST RECENT MESSAGE BY TYPE

```
create table message (
    type varchar,
    text varchar,
    time timestamp
);

select type,
       text,
       time
from (select type,
             text,
             time,
             row_number() over
                 (partition by type
                      order by time desc)
                  as rank
      from message) ranked
where rank = 1;
```

Window function

CONSTRUCTING A TREE

```
create table organization (
    person_id bigint,
    boss_id   bigint
);
```

CONSTRUCTING A TREE

```
create table organization (
    person_id bigint,
    boss_id   bigint
);
```

Recursion

```
with recursive relations (
                    person_id,
                    boss_id)
    as (select person_id,
               boss_id
        from organization
            where person_id = 1
        union all
        select o.person_id,
               o.boss_id
        from organization o
        inner join relations r
            on o.boss_id = r.person_id)
select *
from relations;
```

JSON SUPPORT IN SQL

```
create table audit (
    audit_id bigint,
    created  timestamp,
    content  jsonb
);
```

JSON SUPPORT IN SQL

```
create table audit (
    audit_id bigint,
    created  timestamp,
    content  jsonb
);

select audit_id,
       created,
       content
from audit
where content ->> 'name' = 'John Doe';
```

Condition on element in json strucure - indexable!

ORM

```java
@Entity
@Table(name = "comments")
public class Comment {

    @Id
    @SequenceGenerator(name = "COMMENT_ID_SEQ", sequenceName = "COMMENT_ID_SEQ", initialValue = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "COMMENT_ID_SEQ")
    private Long id;

    @Column(name = "city_id")
    private String city;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "location_id", referencedColumnName = "c_id")
    private Location location;

}
```

ORM

```java
@Entity
@Table(name = "comments")
public class Comment {

    @Id
    @SequenceGenerator(name = "COMMENT_ID_SEQ", sequenceName = "COMMENT_ID_SEQ", initialValue = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "COMMENT_ID_SEQ")
    private Long id;

    @Column(name = "city_id")
    private String city;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "location_id", referencedColumnName = "c_id")
    private Location location;

}
```

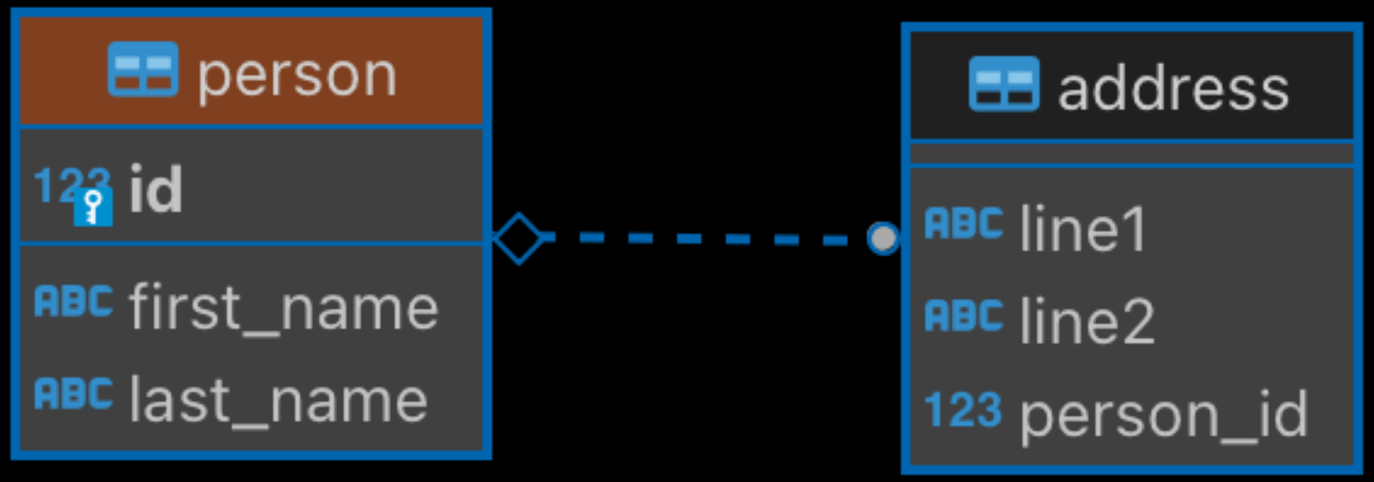
EXPANDING ORM FUNCTIONALITY WITH QUERIES

```
@Entity
@Table(name = "comments")
public class Comment {

    @Query(value = "SELECT * FROM COMMENT c WHERE c.status = 1", nativeQuery = true)
    Collection<Comment> findAllActiveComments();


    @Query("select c from Comment c where c.id.countryCode = ?1 and c.id.municipalityCode = ?2 and
        c.id.type = ?3 and v.location.id not in ?4 order by v.location.name")
    List<LocalComment> getLocalComment(String countryCode, String cityCode,
        String type, List<BigInteger> notLocationIds);
   }
```

OBJECT-RELATIONAL IMPEDANCE MISMATCH



We still need to bridge the fundamental difference between relational databases and object models.

ESSENTIAL COMPLEXITY OR ACCIDENTAL COMPLEXITY?

Do we need all these extra layers atop SQL?

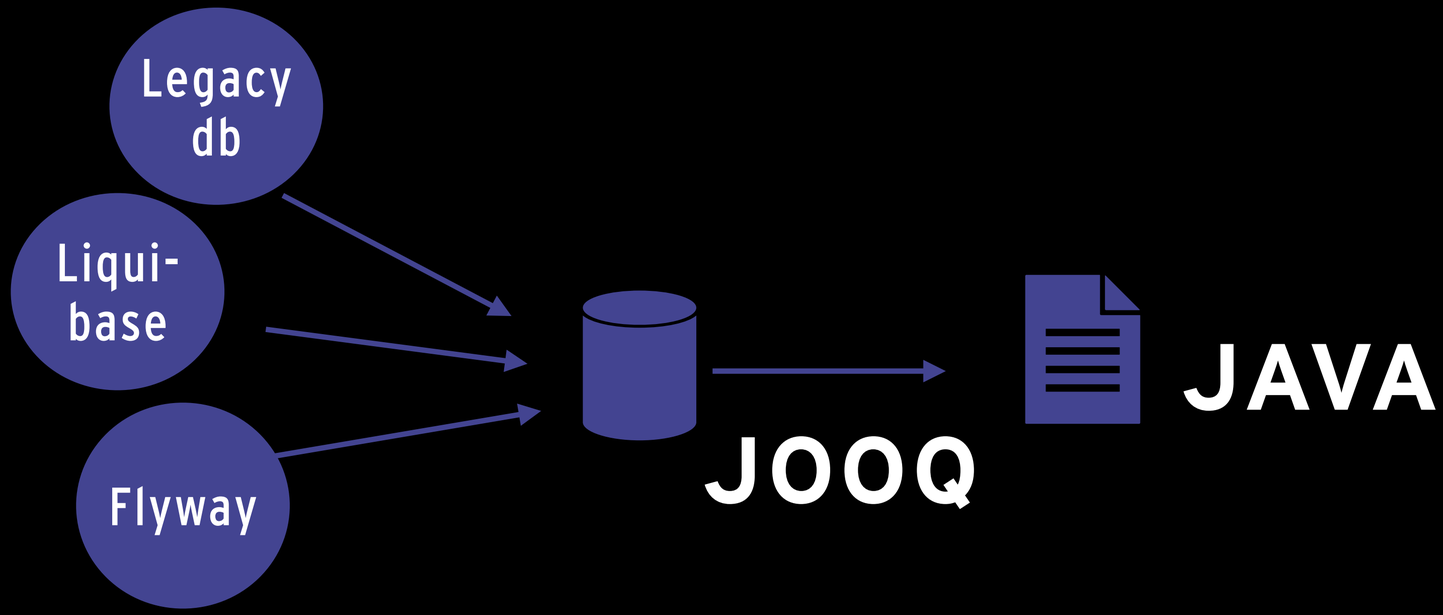
Why not use some real SQL instead?

JOOQ

- Open source product developed by Data Geekery since 2010
- Dual-licensed:
  - free for use with open source databases
  - paid license for non-open source databases
- Very active development and substantial user community

FIRST STEP - CODE GENERATION

- Generated code knows your database and your SQL dialect
- Existing database or created at build time through Liquibase, Flyway
- Integrate with Gradle, Maven



CODE!

JOOQ CHARACTERISTICS

- Database first
- Clean Java code
  - No annotations
  - Testable code - few side effects in runtime
  - Java code -> all the usual constructs, refactorings
- No magic
  - No hidden joins or unexpected merges
  - No caching
  - No lazy loading
- Support for transactions is included
- Virtually every SQL construct is supported

FINDING THE MOST RECENT MESSAGE BY TYPE

```
select type,
       text,
       time
from (select type,
             text,
             time,
             row_number() over
                 (partition by type
                      order by time desc)
                  as rank
      from message) ranked
where rank = 1;
```

FINDING THE MOST RECENT MESSAGE BY TYPE

```
select type,
       text,
       time
from (select type,
             text,
             time,
             row_number() over
                 (partition by type
                      order by time desc)
                  as rank
      from message) ranked
where rank = 1;
```

```
jooq.
   select(
       field(name("type"), VARCHAR),
       field(name("text"), VARCHAR),
       field(name("time"), LOCALDATETIME))
   .from(select(
               TYPE,
               TEXT,
               TIME,
               rowNumber()
                   .over(partitionBy(TYPE)
                               .orderBy(TIME
                                           .desc())
                   .as("rank"))
               .from(MESSAGE))
   .where(field(name("rank"), INTEGER).eq(1))
   .fetchInto(Message.class);
```

CONSTRUCTING A TREE

```
with recursive relations(
                    person_id,
                    boss_id)
    as (select person_id,
               boss_id
        from organization
            where person_id = 1
        union all
        select o.person_id,
               o.boss_id
        from organization o
        inner join relations r
            on o.boss_id = r.person_id)
select *
from relations;
```

CONSTRUCTING A TREE

```
with recursive relations(
                   person_id,
                   boss_id)
    as (select person_id,
               boss_id
        from organization
            where person_id = 1
        union all
        select o.person_id,
               o.boss_id
        from organization o
        inner join relations r
            on o.boss_id = r.person_id)
select *
from relations;
```

```
jooq.withRecursive("relations",
                   "person_id",
                   "boss_id")
        .as(select(PERSON_ID,
                   BOSS_ID)
            .from(ORGANIZATION)
                .where(PERSON_ID.eq(personId))
            .unionAll(
             select(PERSON_ID,
                    BOSS_ID)
             .from(ORGANIZATION)
             .innerJoin(relations)
             .on(ORGANIZATION.BOSS_ID
                .eq(field(name("relations",
                               "person_id"),
                               BIGINT)))))
    .selectFrom(relations)
    .fetchInto(Relation.class);
```

JSON SUPPORT IN SQL

```
select audit_id,
       created,
       content
from audit
where content->>'name' = 'John Doe';
```

JSON SUPPORT IN SQL

```
select audit_id,
       created,
       content
from audit
where content ->> 'name' = 'John Doe';
```

```
private Condition hasName(Field<JSONB> field,
                          String value)  {
    return
        DSL.condition("{0} ->> 'name' = {1}",
                      field,
                      value);
}
```

JSON SUPPORT IN SQL

```
select audit_id,
       created,
       content
from audit
where content ->> 'name' = 'John Doe';
```

```
jooq.select(AUDIT_ID,
            CREATED,
            CONTENT)
    .from(AUDIT)
    .where(hasName(CONTENT, "John Doe"))
    .fetch(mapping(Audit::new));

private Condition hasName(Field<JSONB> field,
                          String value)  {
    return
        DSL.condition("{0}->>'name' = {1}",
                      field,
                      value);
}
```

WHY USE JOOQ?

ORMs such as Hibernate are mature and work well - use them if you like them!

But jOOQ works well when:

- the database model influences your application design
- you have more complex queries than simple CRUD
- your data is not only used as objects
- you expect your data to outlive your application
- you value architectural and code simplicity
- you want to learn SQL rather the intricacies of an ORM

SQL IS A GREAT, MYSTERIOUS LANDSCAPE WORTH EXPLORING