ScalikeJDBC

A *We Know Scala Series* presentation

Miklos Csuka – 16 June 2023

What is ScalikeJDBC?

- SQL-based RDBMS access library for Scala
- A layer of abstraction over the JDBC Java API
- Provides blocking DB access (ScalikeJDBC-Async in the alpha stage)
- Should work with any JDBC driver, but the following are officially supported:
  - PostgreSQL
  - MySQL
  - H2 Database Engine
  - HSQLDB

Why ScalikeJDBC?

- Build queries in functional style:
  - Flexible transaction control
  - Prepared statement execution: Query, Update, Execute, Batch
  - SQL syntax support in string interpolation
  - Query DSL
- There are some alternatives:
  - Doobie (Typelevel/Cats)
  - Quill (ZIO)
  - Slick (Lightbend) – up to Scala 2.13

Transaction Support

- ScalikeJDBC provides several scopes for session/transaction control
  - readOnly {} – creates a read-only session, throws SQLException on updates
  - autoCommit {} – each SQL update commits within this scope
  - localTx {} – creates a new transaction, rollback on exception, commit on success
  - futureLocalTx {} - use Future's state as transaction boundary, rollback if Future fails
- If none of the above scopes fit your purpose, you can use
  - withinTx {} – reuses an existing transaction, begin/commit/rollback should be handled by the code. E.g.:
    - piggybacking Akka persistence transactions, when your projection DB reuses the persistence DB
    - rollback depending on result value
  - localTx {}(implicit boundary: TxBoundary[A]) – override the transaction handling behaviour

Query

- SQL interpolation

```
val name: Option[String] = DB.readOnly { implicit session =>
  sql"select name from emp where id = ${id}".map(rs => rs.string("name")).single.apply()
}
```

- Prepared statement string

```
case class EmployeeId(id: Int)

val ids: Seq[EmployeeId] = DB.readOnly { implicit session =>
  SQL("select id from emp where name = ?")
    .bind(name).map(rs => EmployeeId(rs.int(1))).list.apply()
}
```

- Map with type binders

```
implicit val employeeIdBinder: TypeBinder[EmployeeId] = new TypeBinder[EmployeeId] {
  def apply(rs: ResultSet, label: String): EmployeeId = EmployeeId(rs.getInt(label))
  def apply(rs: ResultSet, index: Int): EmployeeId = EmployeeId(rs.getInt(index))
}

val ids: Seq[EmployeeId] = DB.readOnly { implicit session =>
  sql"select id from emp".map(_.get[EmployeeId]("id")).list.apply()
}
```

Update, Execute

- **Update**

```
DB.localTx { implicit session =>

  sql"insert into emp (id, name, created_at) values (${id}, ${name}, current_timestamp)"
    .update.apply()

  val newId = sql"insert into emp (name, created_at) values (${name}, current_timestamp)"
    .updateAndReturnGeneratedKey.apply()

  sql"update emp set name = ${newName} where id = ${newId}".update.apply()

  sql"delete from emp where id = ${newId}".update.apply()
}
```

- **Execute**

```
DB.autoCommit { implicit session =>
  sql"create table emp (id integer primary key, name varchar(30))".execute.apply()
}
```

Batch

- Bind by position

```
DB.localTx { implicit session =>
  val batchParams: Seq[Seq[Any]] = (2001 to 3000).map(i => Seq(i, "name_" + i))
  SQL("insert into emp (id, name) values (?, ?)")
    .batch(batchParams: _*)
    .apply()
}
```

- Bind by name

```
DB.localTx { implicit session =>
  sql"insert into emp (id, name) values ({id}, {name})"
    .batchByName(
      Seq(Seq("id" -> 1, "name" -> "Alice"), Seq("id" -> 2, "name" -> "Bob")):_*
    )
    .apply()
}
```

Dynamic SQL queries

- `SQLSyntax (sqls)`

```
case class Employee(id: Int, name: String)

object Employee {
  def apply(rs: WrappedResultSet): Employee =
    Employee(rs.int("id"), rs.string("name"))
}

def listEmployees(isDesc: Boolean)(implicit session: DBSession): List[Employee] = {
  val ordering = if (isDesc) sqls"desc" else sqls"asc"
  sql"select id, name from emp order by id ${ordering} limit 10".map(rs => Employee(rs)).list.apply()
}
```

- `SQLSyntaxSupport`

```
object Employee extends SQLSyntaxSupport[Employee] {
  def apply(e: ResultName[Employee])(rs: WrappedResultSet) =
    new Employee(rs.int(e.id), rs.string(e.name))
}

def employee(id: Int)(implicit session: DBSession): Option[Employee] = {
  val e = Employee.syntax("e")
  sql"select ${e.result.*} from ${Employee.as(e)} where ${e.id} = ${id}"
    .map(Employee(e.resultName)).single.apply()
}
```

Query DSL

```scala
implicit session: DBSession = ???

case class Employee(id: Int, name: String)

object Employee extends SQLSyntaxSupport[Employee] {
  override val schemaName = Some("myschema")
  override val tableName = "emp"

  def apply(e: ResultName[Employee])(rs: WrappedResultSet) =
    new Employee(rs.int(e.id), rs.string(e.name))
}

val e = Employee.syntax("e")
val employees: List[Employee] = withSQL {
  select
    .from(Employee as e)
    .where
    .eq(e.name, "Joe")
    .and
    .gt(e.id, 100)
}.map(Employee(e.resultName)).list.apply()
```

Concurrent execution

- Imagine you need to build an application that executes queries and updates on a database. The database is very performant, storage is sharded to multiple disks, it can handle dozens of concurrent queries
  - expose REST, gRPC, … services with sequential DB queries. Concurrency is the problem of a higher layer, or
  - use Scala Future to query DB before receiving the response of the previous queries, or
  - use cats-effect, or ZIO, or other asynchronous runtime to achieve concurrency

Demo

- Build a simple order entry system, that
  - receives orders with multiple order items
  - each item allocates a number of product items (pieces of fruits)
  - there must be enough products on stock to accept the order
  - an order has to be accepted or rejected as a whole
- Let’s implement it with
  - sequential execution
  - Future
  - cats-effect IO