

Magnum

A New look for Scala Database Clients

August H Nagro

- Scala Engineer at Axoni, developing financial infrastructure for equity markets.
- First experienced Scala at Iowa State University, a sister school to EPFL.
- Open source history includes
 - [The first example of generic coroutines on top of Project Loom](#)
 - [UTF-8 validation using Java's incubating SIMD functionality](#)
 - [NativeConverter, a Scala.js Typeclass](#)
- Live in WA state
- Have written many thousands of lines of SQL

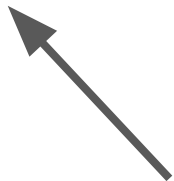


Key Questions

- What types of SQL queries do developers write?
- What types of Database clients are popular on the JVM?
- What makes Magnum unique?
- How is the performance story, and how do the macros work? (time permitting)

Why (yet another) DB Client?

Why (yet another) DB Client?



It worked for Circe right?

What types of SQL statements do we write?

What types of SQL statements do we write?

1. CRUD operations

What types of SQL statements do we write?

1. CRUD operations
2. Simple statements with a few joins

What types of SQL statements do we write?

1. CRUD operations
2. Simple statements with a few joins
3. Complex SQL (many joins, DB-specific operators)
4. Dynamic queries (think search page with filters & pagination)

What types of SQL statements do we write?

1. CRUD operations
2. Simple statements with a few joins
3. Complex SQL (many joins, DB-specific operators)
4. Dynamic queries

Common



Rare

Easy



Challenging

What Database Clients are popular on the JVM?

- Object Oriented Repositories (Spring Data)
- Functional DSLs (JOOQ, Slick, Quill)
- SQL String Interpolators (Anorm, Doobie/Skunk, plain JDBC)

Object Oriented Repositories

1. CRUD operations - Derived at runtime
2. Simple statements with a few joins - JPQL
3. Complex SQL (many joins, DB-specific operators) - Native Queries
4. Dynamic queries - Java EE Specifications (unpleasant & slow)

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {  
  
    List<Customer> findByLastName(String lastName);  
  
    Customer findById(long id);  
}
```

Functional DSLs

1. CRUD operations - N/A
2. Simple statements with a few joins - Good
3. Complex SQL (many joins, DB-specific operators) - Difficult (although most DSL libraries offer a string interpolater)
4. Dynamic queries - Ok (PLs are good at composition)

```
import io.getquill._


case class Person(name: String, age: Int)

val q = quote {
  query[Person].filter(p => p.name == "John").map(p => p.age)
}

testDB.run(q)
```

Functional DSLs makes maintenance hard

- If a query in production behaves poorly, we need to understand both
 - What is wrong with the generated SQL
 - How can we change the DSL to generate a better query
- SQL editors like DataGrip, DbBeaver, and PgAdmin are amazing tools to iteratively build and test SQL queries. But you can't copy and paste between these applications and your Scala editor when using DSLs.



DBBeaver Community

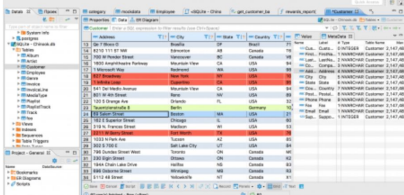
Free Universal Database Tool

Star 32,280
Follow @dbeaver_news

[Home](#)
[About](#)
[Download](#)
[Documentation](#)
[News](#)
[Support](#)
[DBBeaver PRO](#)
[CloudBeaver](#)
[DBBeaver Merch](#)

Universal Database Tool

Free multi-platform database tool for developers, database administrators, analysts and all people who need to work with databases. Supports all popular databases: MySQL, PostgreSQL, SQLite, Oracle, DB2, SQL Server, Sybase, MS Access, Teradata, Firebird, Apache Hive, Phoenix, Presto, etc.



[Download](#)



DataGrip

Many databases, one tool

pgAdmin

PostgreSQL Tools

pgAdmin is the most popular and feature rich Open Source administration and development platform for PostgreSQL, the most advanced Open Source database in the world.

pgAdmin may be used on Linux, Unix, macOS and Windows to manage PostgreSQL and EDB Advanced Server 10 and above.



SQL String Interpolaters

1. CRUD operations - N/A
2. Simple statements with a few joins - Great
3. Complex SQL (many joins, DB-specific operators) - Great
4. Dynamic queries - Difficult (SQL is notoriously hard to compose)

What makes Magnum Unique?

- Combines aspects of all 3 client types
- Supports any database with a JDBC driver, including Postgres, MySql, Oracle, ClickHouse, H2, and Sqlite
- Purely-functional API
- Common queries (like insert, update, delete) generated at compile time
- Simple API with minimal abstraction (writing good sql is hard enough)
- And Much More

`connect` Creates a Database Connection

```
import com.augustnagro.magnum.*  
  
val ds: DataSource = ???  
  
val users: Vector[User] = connect(ds):  
    sql"SELECT * FROM user".query[User].run()
```

`transact` Creates a Database Transaction

```
// update is rolled back  
transact(ds):  
    sql"UPDATE user SET first_name = $firstName WHERE id = $id".update.run()  
    thisMethodThrows()
```

Type-Safe Transaction & Connection Management

```
def runUpdateAndGetUsers()(using DbTx): Vector[User] =  
  userRepo.deleteById(1L)  
  getUsers
```

```
def getUsers(using DbCon): Vector[User] =  
  sql"SELECT * FROM user".query.run()
```

Customizing the Transactions JDBC Connection

```
transact(ds(), withRepeatableRead):
```

```
    ???
```

```
def withRepeatableRead(con: Connection): Unit =
```

```
    con.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ)
```

Sql Interpolater, Frag, Query, and Update

```
val firstNameOpt = Some("John")
val twoDaysAgo = OffsetDateTime.now.minusDays(2)

val frag: Frag =
  sql"""
    SELECT id, last_name FROM user
    WHERE first_name = $firstNameOpt
    AND created <= $twoDaysAgo
    """
```

Sql Interpolater, Frag, Query, and Update

```
val query = frag.query[(Long, String)] // Query[(Long, String)]
```

Sql Interpolater, Frag, Query, and Update

```
val update: Update =  
    sql"UPDATE user SET first_name = 'Buddha' WHERE id = 3".update
```


Sql Interpolator, Frag, Query, and Update

```
transact(ds):  
  val tuples: Vector[(Long, String)] = query.run()  
  val updatedRows: Int = update.run()
```

Batch Updates

```
connect(ds):  
  val users: Iterable[User] = ???  
  val updateResult: BatchUpdateResult =  
    batchUpdate(users): user =>  
      sql"...".update
```

Immutable Repositories

- `count`
- `existsById`
- `findAll`
- `findAll(Spec)`
- `findById`
- `findAllById`

```
@Table(PostgresDbType, SqlNameMapper.CamelToSnakeCase)
case class User(
  @Id id: Long,
  firstName: Option[String],
  lastName: String,
  created: OffsetDateTime
) derives DbCodec

val userRepo = ImmutableRepo[User, Long]

transact(ds):
  val cnt = userRepo.count
  val userOpt = userRepo.findById(2L)
```

Immutable Repositories

```
class UserRepo extends ImmutableRepo[User, Long]:  
  def firstNamesForLast(lastName: String)(using DbCon): Vector[String] =  
    sql"""  
      SELECT DISTINCT first_name  
      FROM user  
      WHERE last_name = $lastName  
      """.query[String].run()  
  
  // other User-related queries here
```

Repo Extends ImmutableRepo, Additionally Deriving:

- delete
- deleteById
- truncate
- deleteAll
- deleteAllById
- insert
- insertAll
- insertReturning
- insertAllReturning
- update
- updateAll

```
@Table(PostgresDbType, SqlNameMapper.CamelToSnakeCase)
case class User(
  @Id id: Long,
  firstName: Option[String],
  lastName: String,
  created: OffsetDateTime
) derives DbCodec
```

```
val userRepo = Repo[User, User, Long]
```

```
val countAfterUpdate = transact(ds):
  userRepo.deleteById(2L)
  userRepo.count
```

Repositories

```
class UserRepo extends Repo[User, User, Long]
```


Database Generated Columns

It is often the case that database columns are auto-generated, for example, primary key IDs. This is why the Repo class has 3 type parameters.

The first defines the Entity-Creator, which should omit any fields that are auto-generated. The entity-creator class must be an 'effective' subclass of the entity class, but it does not have to subclass the entity. This is verified at compile time.

The second type parameter is the Entity class, and the final is for the ID. If the Entity does not have a logical ID, use Null.

```
case class UserCreator(  
  firstName: Option[String],  
  lastName: String,  
) derives DbCodec  
  
@Table(PostgresDbType, SqlNameMapper.CamelToSnakeCase)  
case class User(  
  @Id id: Long,  
  firstName: Option[String],  
  lastName: String,  
  created: OffsetDateTime  
) derives DbCodec  
  
val userRepo = Repo[UserCreator, User, Long]  
  
val newUser: User = transact(ds):  
  userRepo.insertReturning(  
    UserCreator(Some("Adam"), "Smith")  
  )
```

Spec - A Re-Imagining of the JEE Specification

1. If you need to perform joins to get the data needed, first create a database view.
2. Next, create an entity class that derives DbReader.
3. Finally, use the Spec class to create a specification.

Spec - A Re-Imagining of the JEE Specification

```
val partialName = "Ja"
val searchDate = OffsetDateTime.now.minusDays(2)
val idPosition = 42L

val spec = Spec[User]
  .where(sql"first_name ILIKE '$partialName%'"")
  .where(sql"created >= $searchDate")
  .seek("id", SeekDir.Gt, idPosition, SortOrder.Asc)
  .limit(10)

val users: Vector[User] = userRepo.findAll(spec)
```

Scala 3 Enum Support

```
@Table(PostgresDbType, SqlNameMapper.CamelToUpperSnakeCase)
enum Color derives DbCodec:
  case Red, Green, Blue
```

```
@Table(PostgresDbType, SqlNameMapper.CamelToSnakeCase)
case class User(
  @Id id: Long,
  firstName: Option[String],
  lastName: String,
  created: OffsetDateTime,
  favoriteColor: Color
) derives DbCodec
```

DbCodec: A Typeclass for JDBC Reading and Writing

```
val rs: ResultSet = ???  
val ints: Vector[Int] = DbCodec[Int].read(rs)  
  
val ps: PreparedStatement = ???  
DbCodec[Int].writeSingle(22, ps)
```

Magnum

1. CRUD operations - Derived at compile time
2. Simple statements with a few joins - Great (sql interpolater)
3. Complex SQL (many joins, DB-specific operators) - Great (sql interpolater)
4. Dynamic queries - Built-in support for specifications

Give it a Try

Version 1.0.0 was released on maven central today.

```
“com.augustnagro” %% “magnum” % “1.0.0”
```


Thanks!

‘Given Defaults Pattern’

- Typeclass derivation is not possible for trait with more than 1 type parameter.
- Macro annotations are still a wip in Scala 3. From my (potentially stale) understanding, you cannot use them to generate new types in the companion object. Besides, the API is currently very low level compared to quotation.

(see Repo.scala)