

# Relational Database Access

with *Slick*

# *Slick*

Reactive Functional Relational Mapping for Scala

— *<http://slick.lightbend.com>*

- Library to access **relational databases**
- Can express **queries** in a **functional way** (`map`, `filter`...)
- Also supports **plain SQL**
- Supports asynchronicity (`Future`) and streaming (*Reactive Streams*)

# Database Profile

# Database Profile

- Many differences between databases and SQL dialects
- A **profile** allows
  - to unify peculiarities,
  - and also to benefit from proprietary features.

# Tailoring a *PostgreSQL* Profile with *Slick-pg*

```
// Extend PostgreSQL profile with java.time and Spray JSON support
trait ExtendedPostgresProfile extends ExPostgresProfile with PgDate2Support with PgSprayJsonSupport {
  val pgjson: String = "jsonb" // jsonb type for JSON column

  // Add 'insert or update' capability
  override protected def computeCapabilities: Set[Capability] =
    super.computeCapabilities + JdbcCapabilities.insertOrUpdate

  override val api: ExtendedAPI = ExtendedAPI
  // Add API support for Date and Time, and JSON
  trait ExtendedAPI extends API with DateTimeImplicits with JsonImplicits
  object ExtendedAPI extends ExtendedAPI
}

object ExtendedPostgresProfile extends ExtendedPostgresProfile
```

# Importing API of the *PostgreSQL* Profile

```
import ExtendedPostgresProfile.api._
```

Import when in need to

- to connect to the **database**
- to describe **tables**,
- to describe **queries**,
- and more.

# Describing **Tables** and Mapping **Records**

# Customers Table and Customer Record

```
class Customers(tag: Tag) extends Table[Customer](tag, "customers") {  
  def id = column[Long]("id", 0.PrimaryKey, 0.AutoInc)  
  def firstName = column[String]("first_name")  
  def lastName = column[String]("last_name")  
  def * = (id.?, firstName, lastName).mapTo[Customer]  
  
  def fullName = firstName ++ " " ++ lastName // Calculated column  
}  
  
object Customers {  
  val table = TableQuery[Customers]  
}  
  
case class Customer(id: Option[Long], firstName: String, lastName: String)
```



# Table Classes

**Table Class** describes a **table in a database**

- Name of **table** in database (Table)
- Name and type of **columns** in table (column)
- **Mapping** of table row to Record Class (mapTo, <>)
- Foreign keys (foreignKey)
- Indexes (index)
- Calculated columns

# Record Classes

**Record Class** describes a **row in a table**

- Practically a **case class**
- **Strictly reflects table columns**, even foreign key columns
  - Might be a selection of columns
  - Might be **substructured**
  - Might use **custom column types**
- **Not part of the domain model**
  - Additional mapping to and from the model

# When mapTo Does Not Compile

```
class Customers(tag: Tag) extends Table[Customer](tag, "customers") {  
  def id = column[Long]("id", 0.PrimaryKey, 0.AutoInc)  
  def firstName = column[String]("first_name")  
  def lastName = column[String]("last_name")  
  def * = (id.?, firstName, lastName) <> ((Customer.apply _).tupled, Customer.unapply)  
}
```

# Making Sense of <> Method

```
(id.?, firstName, lastName) <> ((Customer.apply _).tupled, Customer.unapply)
```

## Part

## Type

---

```
Customer.apply _
```

```
(Option[Long], String, String) =>  
Customer
```

---

```
(Customer.apply _).tupled
```

```
((Option[Long], String, String)) =>  
Customer
```

---

```
Customer.unapply _
```

```
Customer => Option[(Option[Long],  
String, String)]
```

# Orders Table and Order Record

```
class Orders(tag: Tag) extends Table[Order](tag, "orders") {  
  def id = column[Long]("id", 0.PrimaryKey, 0.AutoInc)  
  def customerId = column[Long]("customer_id")  
  def date = column[LocalDate]("date")  
  def * = (id.?, customerId, date).mapTo[Order]  
  
  def customer = foreignKey("fk_orders_customer_id", customerId, Customers.table)(_id)  
}  
  
object Orders {  
  val table = TableQuery[Orders]  
}  
  
case class Order(id: Option[Long], customerId: Long, date: LocalDate)
```

# OrderLines Table and OrderLine Record

```
class OrderLines(tag: Tag) extends Table[OrderLine](tag, "order_lines") {
  def id = column[Long]("id", 0.PrimaryKey, 0.AutoInc)
  def orderId = column[Long]("order_id")
  def itemId = column[Long]("item_id")
  def quantity = column[Int]("quantity")
  def * = (id.?, orderId, itemId, quantity).mapTo[OrderLine]

  def order = foreignKey("fk_order_lines_order_id", orderId, Orders.table)(_id)
  def item = foreignKey("fk_order_lines_item_id", itemId, Items.table)(_id)
  def orderIdItemIdIndex = index("idx_order_lines_order_id_item_id", (orderId, itemId), unique = true)
}

object OrderLines {
  val table = TableQuery[OrderLines]
}

case class OrderLine(id: Option[Long], orderId: Long, itemId: Long, quantity: Int)
```

# Items Table and Item Record

```
class Items(tag: Tag) extends Table[Item](tag, "items") {  
  def id = column[Long]("id", 0.PrimaryKey, 0.AutoInc)  
  def name = column[String]("name")  
  def * = (id.?, name).mapTo[Item]  
}
```

```
object Items {  
  val table = TableQuery[Items]  
}
```

```
case class Item(id: Option[Long], name: String)
```

# Substructuring a Record Class

## Substructuring the record class

- Customer record class can have an address attribute of type Address that maps to some of the fields in the customers table.
- Order record can also hold a billingAddress and a shippingAddress (also of type Address), each mapping to different fields of the orders table.
- Much better than super flat record
- Also allows to overcome the 22 fields limit



# Custom Column Types

## Custom column types

- Order record class can have an attribute of enumeration type `OrderStatus`.
- `OrderStatus` can be declared as custom column type to be stored as `VARCHAR` in the `orders` table.

# Describing **Queries** (Query)

# Query (Query)

```
Query[+E, U, C[_]] /* extends Rep[C[U]] */  
// E = SELECT Row Representation  
// U = In-memory Row Class  
// C = Collection Type (Seq, Set...)
```

- Describes a **SELECT query** retrieving **rows**
- Does **no side-effect**, just a query waiting to be run
- Must be **interpreted** against a **database** to do side-effects
- Will return a C[U] when run (for example Seq[Customer])

# Filtering (WHERE)

```
val selectCustomersQuery: Query[Customers, Customer, Seq] =  
  Customers.table  
    .filter(_.firstName.like("A%"))  
  
val selectItemQuery: Query[Items, Item, Seq] =  
  Items.table  
    .filter(_.id === 11)
```

# Mapping (SELECT)

```
val selectFirstNameAndLastNameQuery:  
  Query[(Rep[String], Rep[String]), (String, String), Seq] =  
    Customers.table  
      .filter(_.id !== 11)  
      .map(c => (c.firstName, c.lastName))
```

```
val selectFullNameQuery:  
  Query[Rep[String], String, Seq] =  
    Customers.table  
      .filter(_.firstName.startsWith("A"))  
      .map(c => c.firstName ++ " " ++ c.lastName)
```

# Joining (JOIN)

```
val selectOrdersAndOrderLinesQuery:  
  Query[  
    (Orders, Rep[Option[OrderLines]]),  
    (Order, Option[OrderLine]),  
    Seq  
  ] =  
  Orders.table joinLeft OrderLines.table on (_.id === _.orderId)
```

# Sorting (ORDER BY)

```
val selectOrdersAndOrderLinesOrderedQuery =  
  selectOrdersAndOrderLinesQuery  
    .sortBy { case (order, maybeOrderLine) =>  
      (order.id, maybeOrderLine.map(_.id))  
    }
```

# Advanced Queries

- [Unioning](#) (UNION, UNION ALL)
- [Aggregating](#) (GROUP BY)
- [Plain SQL queries](#)
  - INTERSECT, EXCEPT
  - Other advanced SQL features
- [Compiled queries](#)
  - Precompile a **parameterized query** for better performance



# Describing **Database I/Os** (DBIO)

# Database I/O (DBIO)

```
DBIO[A] // A = Result
```

- Describes a **program accessing a database**
- Does **no side-effect**, just a program waiting to be run
- Must be **interpreted** against a **database** to do side-effects
- When interpreted, it will either
  - **succeed** with a **result** of type A,
  - or **fail** holding an **exception**.

# Querying (SELECT)

```
val selectCustomersDBIO: DBIO[Seq[Customer]] =  
  selectCustomersQuery.result
```

```
val selectItemDBIO: DBIO[Option[Item]] =  
  selectItemQuery.result.headOption
```

```
val selectOrdersAndOrderLinesOrderedDBIO: DBIO[Seq[(Order, Option[OrderLine])]] =  
  selectOrdersAndOrderLinesOrderedQuery.result
```

# Inserting (INSERT)

```
val insertCustomerDBIO: DBIO[Int] =  
  Customers.table +=  
    Customer(None, "April", "Jones")  
  
val insertCustomersDBIO: DBIO[Seq[Customer]] =  
  (Customers.table returning Customers.table) ++= Seq(  
    Customer(None, "Anders", "Petersen"),  
    Customer(None, "Pedro", "Sanchez"),  
    Customer(None, "Natacha", "Borodine")  
  )
```

# Updating (UPDATE)

```
val updateCustomerDBIO: DBIO[Int] =  
  Customers.table  
    .filter(c => c.firstName === "Anders" && c.lastName === "Petersen")  
    .map(c => (c.firstName, c.lastName))  
    .update("Anton", "Peterson")
```

# Deleting (DELETE)

```
val deleteCustomerDBIO: DBIO[Int] =  
  Customers.table  
    .filter(_.firstName === "April")  
    .delete
```

# Combining DBIOs into a DBIO Program

```
case class Result(insertedCustomers: Seq[Customer],
                  customers: Seq[Customer],
                  maybeItem: Option[Item],
                  ordersAndOrderLines: Seq[(Order, Option[OrderLine])])

val program: DBIO[Result] = for {
  _ <- insertCustomerDBIO
  insertedCustomers <- insertCustomersDBIO
  _ <- updateCustomerDBIO
  _ <- deleteCustomerDBIO
  customers <- selectCustomersDBIO
  maybeItem <- selectItemDBIO
  ordersAndOrderLines <- selectOrdersAndOrderLinesOrderedDBIO
} yield Result(insertedCustomers, customers, maybeItem, ordersAndOrderLines)
```

# Making a DBIO Transactional

```
val transactionalProgram: DBIO[Result] = program.transactionally
```

- transactionally method results in a DBIO that will be run as a **single transaction**.
- Otherwise, each composing DBIO would be run in a separate transaction.



# Running **Database I/Os** (DBIO)

# Database Configuration (application.conf)

```
ecommerce {  
  database {  
    # http://slick.lightbend.com/doc/3.2.3/api/index.html#  
    # slick.jdbc.JdbcBackend$DatabaseFactoryDef@forConfig(String,Config,Driver,ClassLoader):Database  
  
    # numThreads = 200          # (Int, optional, default: 20)  
    # queueSize = 100          # (Int, optional, default: 1000)  
  
    # HikariCP Configuration (add "slick-hikaricp" dependency)  
    url = "jdbc:postgresql://localhost:5432/ecommerce?currentSchema=ecommerce"  
    driver = "org.postgresql.Driver"  
    user = "ecommerceapi"  
    password = "password"  
    # isolation = SERIALIZABLE # (String, optional)  
    # maxConnections = 20      # (Int, optional, default: numThreads)  
    # minConnections = 20      # (Int, optional, default: numThreads)  
  }  
}
```

# Loading Database Configuration

```
// Load configuration from application.conf (using Lightbend Config library)
val config = ConfigFactory.load()

// Create database object from configuration
val database = Database.forConfig(
    "ecommerce.database",
    config
)
```

# Running DBIO on Database

```
val eventualResult: Future[Result] = database.run(transactionalProgram)
```

- Interprets the program described by the DBIO[A] against a database
- Returns a Future[A], a **promise** for a result that will eventually
  - **succeed** with a **result** of type A,
  - or **fail** holding an **exception**.
- Exception is just used as a value and is **never thrown**.

# Handling Future Result

# Handling Success

```
val eventualCompletion: Future[Unit] = for {  
  Result(insertedCustomers, customers, maybeItem, ordersAndOrderLines) <- eventualResult  
} yield {  
  logger.info(s"insertedCustomers=$insertedCustomers")  
  logger.info(s"customers=$customers")  
  logger.info(s"maybeItem=$maybeItem")  
  logger.info(s"ordersAndOrderLines=$ordersAndOrderLines")  
}
```

# Handling Failure

```
val eventualSafeCompletion: Future[Unit] = eventualCompletion
  .transform {
    case failure @ Failure(exception) =>
      // Log exception and keep failure as is
      logger.error("Exception occurred", exception)
      failure

    case success @ Success(_) =>
      // Keep success as is
      success
  }
// Always close database after completion (either success or failure)
.transformWith(_ => database.shutdown)
```

# Waiting for **Completion**

```
Await.result(eventualSafeCompletion, 5.seconds)
```

- Will **block** until Future completes
  - Return the **result** in case of **success**
  - Raise the **exception** in case of a **failure**
  - **Timeout** after 5 seconds and fail with a `TimeoutException`
- Use this very sparingly!
- *Akka HTTP* handles futures directly without the hassle.



# Combining **Database IOs** (DBIO)

# Basic DBIOs

```
val success: DBIO[Int] = DBIO.successful(42)
```

```
// Will produce result 42 when run
```

```
val failure: DBIO[Nothing] = DBIO.failed(new IllegalStateException("Failure"))
```

```
// Will never produce a result and fail with IllegalStateException when run
```

# Finding Customer, Order and OrderLines

```
def findCustomer(id: Long): DBIO[Customer] =  
    Customers.table.filter(_.id === id).result.head  
  
def findOrder(id: Long): DBIO[Order] =  
    Orders.table.filter(_.id === id).result.head  
  
def findOrderLines(orderId: Long): DBIO[Seq[OrderLine]] =  
    OrderLines.table.filter(_.orderId === orderId).result
```

# Transforming DBIO (map)

```
def findOrderDescription(orderId: Long): DBIO[String] = {  
  findOrder(orderId).map { order =>  
    s"Order #$orderId for customer ${order.customerId}"  
  }  
}
```

# Transforming DBIO (for / yield)

```
def findOrderDescription(orderId: Long): DBIO[String] = {  
  for {  
    order <- findOrder(orderId)  
  } yield s"Order #$orderId for customer ${order.customerId}"  
}
```

# Sequencing DBIOs (broken map)

```
def findOrderCustomer(orderId: Long): DBIO[DBIO[Customer]] = {  
  findOrder(orderId).map { order =>  
    findCustomer(order.customerId)  
  }  
}
```

- Wrong **nested** type DBIO[DBIO[Customer]]
- Needs to be made **flat** somehow as DBIO[Customer]

# Sequencing DBIOs (flatMap)

```
def findOrderCustomer(orderId: Long): DBIO[Customer] = {  
  findOrder(orderId).flatMap { order =>  
    findCustomer(order.customerId)  
  }  
}
```

# Sequencing DBIOs (for / yield)

```
def findOrderCustomer(orderId: Long): DBIO[Customer] = {  
  for {  
    order <- findOrder(orderId)  
    customer <- findCustomer(order.customerId)  
  } yield customer  
}
```



# Sequencing with Non-DBIO

```
def findOrderAndCustomerAndOrderLines(orderId: Long): DBIO[Result] = {  
  for {  
    order <- findOrder(orderId)  
    customerId = order.customerId // Not a DBIO, '=' instead of '<-'  
    orderLines <- findLines(orderId)  
    customer <- findCustomer(customerId)  
  } yield Result(customer, order, orderLines)  
}
```

# Pyramid of maps and flatMaps 🍆

```
def findOrderAndCustomerAndOrderLines(orderId: Long): DBIO[Result] =  
  findOrder(orderId).flatMap { order =>  
    findCustomer(order.customerId).flatMap { customer =>  
      findLines(orderId).map { orderLines =>  
        Result(customer, order, orderLines)  
      }  
    }  
  }  
}
```

# Flatten Them All 🙇

```
def findOrderAndCustomerAndOrderLines(orderId: Long): DBIO[Result] =  
  for {  
    order <- findOrder(orderId)  
    customer <- findCustomer(order.customerId)  
    orderLines <- findLines(orderId)  
  } yield Result(customer, order, orderLines)
```

# Anatomy of for Comprehension

**for comprehension is not a for loop.**

It can be a for loop...

But it can handle **many other things**

like Option, Future and... DBIO.

# for Comprehension **Types**

```
def findOrderAndCustomerAndLines(orderId: Long): DBIO[Result] = {  
  for {  
    order      /* Order          */ <- findOrder(orderId)      /* DBIO[Order]          */  
    customerId /* Long           */ = order.customerId          /* Long                 */  
    lines      /* Seq[OrderLine] */ <- findLines(order.id.get)   /* DBIO[Seq[OrderLine]] */  
    customer   /* Customer       */ <- findCustomer(customerId) /* DBIO[Customer]      */  
  } yield Result(customer, order, lines) /* Result */  
} /* DBIO[Result] */
```

# for Comprehension **Type Rules**

	val type	operator	expression type
generator	A	<-	DBIO[A]
assignment	B	=	B
	for comprehension type		yield expression type
result	DBIO[R]		R

- **Only** DBIO[T], **no mix** with Option[T], Future[T], Seq[T]... Otherwise 💣 *it won't compile* 💣
- But it could be **only** Option[T], or **only** Future[T], or **only** Seq[T]...

# for Comprehension **Scopes**

```
def findOrderAndCustomerAndLines(orderId: Long): DBIO[Result] = {  
  for {  
    order <- findOrder(orderId)           /* order                */  
    customerId = order.customerId         /* 0      customerId       */  
    lines <- findLines(order.id.get)      /* 0      |      orderLines */  
    customer <- findCustomer(customerId) /* |      0      |      customer */  
  } yield Result(customer, order, lines) /* 0      |      0      0      */  
}
```



# for Comprehension **Implicit Nesting**

```
def findOrderAndCustomerAndLines(orderId: Long): DBIO[Result] = {  
  for {  
    order <- findOrder(orderId)  
    /* | */ customerId = order.customerId  
    /* | | */ lines <- findLines(order.id.get)  
    /* | | | */ customer <- findCustomer(customerId)  
  } /* | | | | */ yield Result(customer, order, lines)  
}
```

Visually flattens, but still implicitly nested

# Conditions and Loops with **Database IOs (DBIO)**

# A Tale of Free Welcome Order

- A free gift for every customer having never ordered anything
- Materialized by a fictitious order
- Let's call it *Free Welcome Order* or *FWO*
- Yes, this is a bit contrived 😊

# Count Orders of a Customer

```
def findOrderCountByCustomerId(customerId: Long): DBIO[Int] =  
    Orders.table  
        .filter(_.customerId === customerId)  
        .size // Rep[Int]  
        .result
```

# Insert FWO for a Customer

```
def insertFwoByCustomerId(customerId: Long): DBIO[Unit] =  
  for {  
    orderId <-  
      (Orders.table returning Orders.table.map(_.id)) +=  
        Order(None, customerId, LocalDate.now())  
  
    _ <- OrderLines.table += OrderLine(None, orderId, 1, 1)  
  } yield ()
```

# Conditionally Insert FWO for a Customer

```
def conditionallyInsertFwo(customerId: Long): DBIO[Boolean] =  
  for {  
    orderCount <- findOrderCountByCustomerId(customerId)  
  
    done <-  
      if (orderCount == 0)  
        insertFwoByCustomerId(customerId).flatMap(_ => DBIO.successful(true))  
      else  
        DBIO.successful(false)  
  } yield done
```

# Repeatedly Insert FWO for Customers

```
def repeatedlyInsertFwo(customerIds: Seq[Long]): DBIO[Seq[Boolean]] = {  
  val seqOfDbio: Seq[DBIO[Boolean]] = customerIds.map(conditionallyInsertFwo)  
  val dbioOfSeq: DBIO[Seq[Boolean]] = DBIO.sequence(seqOfDbio)  
  dbioOfSeq  
}
```

- Make a Seq[DBIO[Boolean]] using map over a Seq[Long]
- Turn it into a DBIO[Seq[Boolean]] using DBIO.sequence

# Repeating with **Recursion** 🤔

```
def insertCustomer(n: Int): DBIO[Int] =  
  Customers.table += Customer(None, s"First Name $n", s"Last Name $n")  
  
def insertCustomers(n: Int): DBIO[Int] =  
  if (n > 0)  
    for {  
      count <- insertCustomer(n)  
      restCount <- insertCustomers(n - 1) // Recursion  
    } yield count + restCount  
  else  
    DBIO.successful(0)
```



# Replacing Recursion with **Fold**

```
def insertCustomers(n: Int): DBIO[Int] = {  
  val counts: Seq[DBIO[Int]] = (1 to n).map(insertCustomer)  
  val totalCount: DBIO[Int] = DBIO.fold(counts, 0)(_ + _)  
  totalCount  
}
```

- Recursion can be hard to read
- Prefer using simpler alternatives whenever possible
  - `DBIO.sequence`
  - `DBIO.fold`

# Further with *Slick*

# Generating Table Creation Script

```
val schema =  
  Customers.table.schema ++  
  Orders.table.schema ++  
  OrderLines.table.schema ++  
  Items.table.schema
```

```
schema.createStatements.foreach(sql => println(s"$sql;"))
```

```
create table "customers" ("id" BIGSERIAL NOT NULL PRIMARY KEY,"first_name" VARCHAR NOT NULL,"last_name" VARCHAR NOT NULL);  
create table "orders" ("id" BIGSERIAL NOT NULL PRIMARY KEY,"customer_id" BIGINT NOT NULL,"date" date NOT NULL);  
create table "order_lines" ("id" BIGSERIAL NOT NULL PRIMARY KEY,"order_id" BIGINT NOT NULL,"item_id" BIGINT NOT NULL,"quantity" INTEGER NOT NULL);  
create unique index "idx_order_lines_order_id_item_id" on "order_lines" ("order_id","item_id");  
create table "items" ("id" BIGSERIAL NOT NULL PRIMARY KEY,"name" VARCHAR NOT NULL);  
alter table "orders" add constraint "fk_orders_customer_id" foreign key("customer_id") references "customers"("id") on update NO ACTION on delete NO ACTION;  
alter table "order_lines" add constraint "fk_order_lines_item_id" foreign key("item_id") references "items"("id") on update NO ACTION on delete NO ACTION;  
alter table "order_lines" add constraint "fk_order_lines_order_id" foreign key("order_id") references "orders"("id") on update NO ACTION on delete NO ACTION;
```

# Testing for Existence

```
def findOrderExistenceByCustomerId(customerId: Long): DBIO[Boolean] =  
  Orders.table  
    .filter(_.customerId === customerId)  
    .exists // Rep[Boolean]  
    .result
```

- Avoid `.size` that counts all matching records
- Prefer `.exists` that stops on first matching record

## `.headOption` vs `.head`

- `.result.headOption` **always succeeds** with an `Option[T]`
- `.result.head` **might succeed** with a `T` **or fail** with an exception
- Favor `.result.headOption`

# Configuring *Slick* Logs in `logback.xml`

Be sure to add *Logback* dependency and a `logback.xml` file

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <!-- ... -->
  <logger name="slick.basic" level="INFO"/>
  <logger name="slick.compiler" level="INFO"/>
  <logger name="slick.jdbc" level="DEBUG"/> <!-- Log SQL -->
  <logger name="slick.memory" level="INFO"/>
  <logger name="slick.relational" level="INFO"/>
  <logger name="slick.util" level="INFO"/>

  <logger name="com.zaxxer.hikari" level="INFO"/>
</configuration>
```

# More about *Slick*

- [Slick](#) documentation
  - [Queries](#)
  - [Database I/O Actions](#)
  - [Database Configuration](#)
  - [Logging](#) with *SLF4J*
- [Essential Slick](#) book

# Related Libraries Supported by *Slick*

- [Slick-pg](#) Slick extensions for *PostgreSQL*
- [HikariCP](#), database connection pool
- [Lightbend Config](#), application configuration (`application.conf`)
- [SL4J](#), logging facade
  - Already a dependency of *Slick...* and *Akka HTTP*
- [Scala Logging](#), recommended Scala wrapper for *SLF4J*
- [Logback](#), standard implementation for *SLF4J*