THE FUNCTIONAL WEB STACK

HTTP4S, DOOBIE & CIRCE

Gary Coady
gcoady@gilt.com

LET'S BUILD A WEBSITE

- It's going to be CRUD (create/read/update/delete)
- How do we convert data to/from database structure?
- How do we convert data to/from JSON?
- Converting from one format to another is most of a developer's job!

CONVERTING TO/FROM JSON


```
case class MyClass(
  s: String,
  t: Instant,
  i: Int)
```

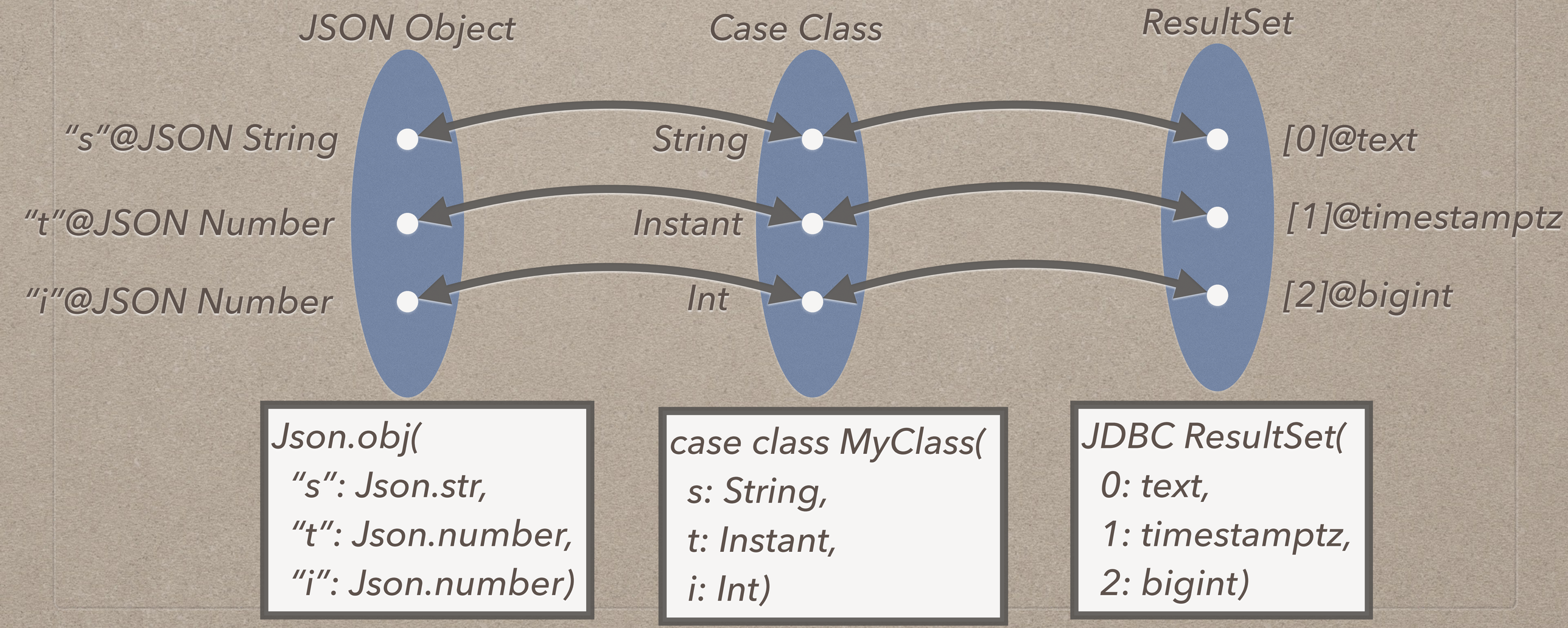
```
Json.obj(
  "s": Json.str,
  "t": Json.number,
  "i": Json.number)
```

CONVERTING TO/FROM JDBC


```
case class MyClass(
  s: String,
  t: Instant,
  i: Int)
```

```
JDBC ResultSet(
  0: text,
  1: timestamptz,
  2: bigint)
```

JDBC AND JSON COMBINED



JSON CONVERSION WITH CIRCE

- Very fast, usable JSON library
- Automatic derivation of JSON codecs for case classes
- Integration available with http4s
- http://circe.io

JSON CONVERSION WITH CIRCE

A => Json

```
trait Encoder[A] {
 def apply(a: A): Json
}
```

java.time.Instant => Json

```
implicit val jsonEncoderInstant: Encoder[Instant] =
  Encoder[Long].contramap(_.toEpochMilli)
```

JSON CONVERSION WITH CIRCE

Json => A

```
trait Decoder[A] {
 def apply(c: HCursor): Decoder.Result[A]
}
```

Json => java.time.Instant

```
implicit val jsonDecoderInstant: Decoder[Instant] =
  Decoder[Long].map(Instant.ofEpochMilli)
```

CONVERTING CASE CLASSES WITH CIRCE

Manual

```
// for case class Person(id: Int, name: String)

object Person {
  implicit val decodePerson: Decoder[Person] =
    Decoder.forProduct2("id", "name")(Person.apply)

  implicit val encodePerson: Encoder[Person] =
    Encoder.forProduct2("id", "name")(p =>
      (p.id, p.name)
    )
}
```

Automatic

```
import io.circe.generic.auto._
```

CONVERTING TO/FROM JSON


```
case class MyClass(
  s: String,
  t: Instant,
  i: Int)
```

```
Json.obj(
  "s": Json.str,
  "t": Json.number,
  "i": Json.number)
```

JDBC USING DOOBIE

- Doobie is a “pure functional JDBC layer for Scala”
- Complete representation of Java JDBC API
- https://github.com/tpolecat/doobie
- tpolecat.github.io/doobie-0.2.3/00-index.html (Documentation)

JDBC MAPPING WITH DOOBIE

```scala
sealed trait Meta[A] {
  /** Destination JDBC types to which values of type `A` can be written. */
  def jdbcTarget: NonEmptyList[JdbcType]

  /** Source JDBC types from which values of type `A` can be read. */
  def jdbcSource: NonEmptyList[JdbcType]

  /** Constructor for a `getXXX` operation for type `A` at a given index. */
  val get: Int => RS.ResultSetIO[A]

  /** Constructor for a `setXXX` operation for a given `A` at a given index. */
  val set: (Int, A) => PS.PreparedStatementIO[Unit]

  /** Constructor for an `updateXXX` operation for a given `A` at a given index. */
  val update: (Int, A) => RS.ResultSetIO[Unit]

  /** Constructor for a `setNull` operation for the primary JDBC type, at a given index. */
  val setNull: Int => PS.PreparedStatementIO[Unit]
}
```

JDBC MAPPING WITH DOOBIE

Reusing an existing Meta definition

```
implicit val metaInstant =
  Meta[java.sql.Timestamp].nxmap(
    _.toInstant,
    (i: Instant) => new java.sql.Timestamp(i.toEpochMilli)
  )
```

Meta definitions exist for:

Byte
Short
Int
Long
Float
Double
java.math.BigDecimal
Boolean
String
Array[Byte]
BigDecimal
java.sql.Time
java.sql.TimeStamp
java.sql.Date
java.util.Date

CONVERTING CASE CLASSES WITH DOOBIE

Nothing extra needed!

QUERIES WITH DOOBIE

Given the table

```
                                  Table "public.people"
    Column     |            Type             |                       Modifiers
---------------+-----------------------------+-------------------------------------------------------
 id            | integer                     | not null default nextval('people_id_seq'::regclass)
 name          | text                        |
Indexes:
    "people_pkey" PRIMARY KEY, btree (id)
```

Define a query

```
sql"select id, name from people".query[Person]
```

Using prepared statements & variable interpolation

```
sql"select id, name from people where id = $id".query[Person]
```

DEFINE THE EXPECTED RESULT SIZE

- `myQuery.unique` – Expect a single row from the query
- `myQuery.option` – Expect 0-1 rows from the query, return an Option
- `myQuery.list` – Return the results as a List
- `myQuery.vector` – Return the results as a Vector
- `myQuery.process` – Return the results as a stream of data

RUNNING QUERIES WITH DOOBIE

Define a Transactor for your Database

```
val xa = DriverManagerTransactor[Task](
  "org.postgresql.Driver", "jdbc:postgresql:demo", "demo", ""
)
```

Run your query using the Transactor

```
myQuery.list.transact(xa)
```

Your query runs in a transaction (rollback on uncaught exception)

UPDATES WITH DOOBIE

Call .update instead of .query (returns number of rows modified)

```
def updatePerson(id: Int, name: String): ConnectionIO[Int] =
  sql"update people set name=$name where id=$id"
    .update
```

.withUniqueGeneratedKeys provides data from updated row

```
def updatePerson(id: Int, name: String): ConnectionIO[Person] =
  sql"update people set name=$name where id=$id"
    .update
    .withUniqueGeneratedKeys("id", "name")
```

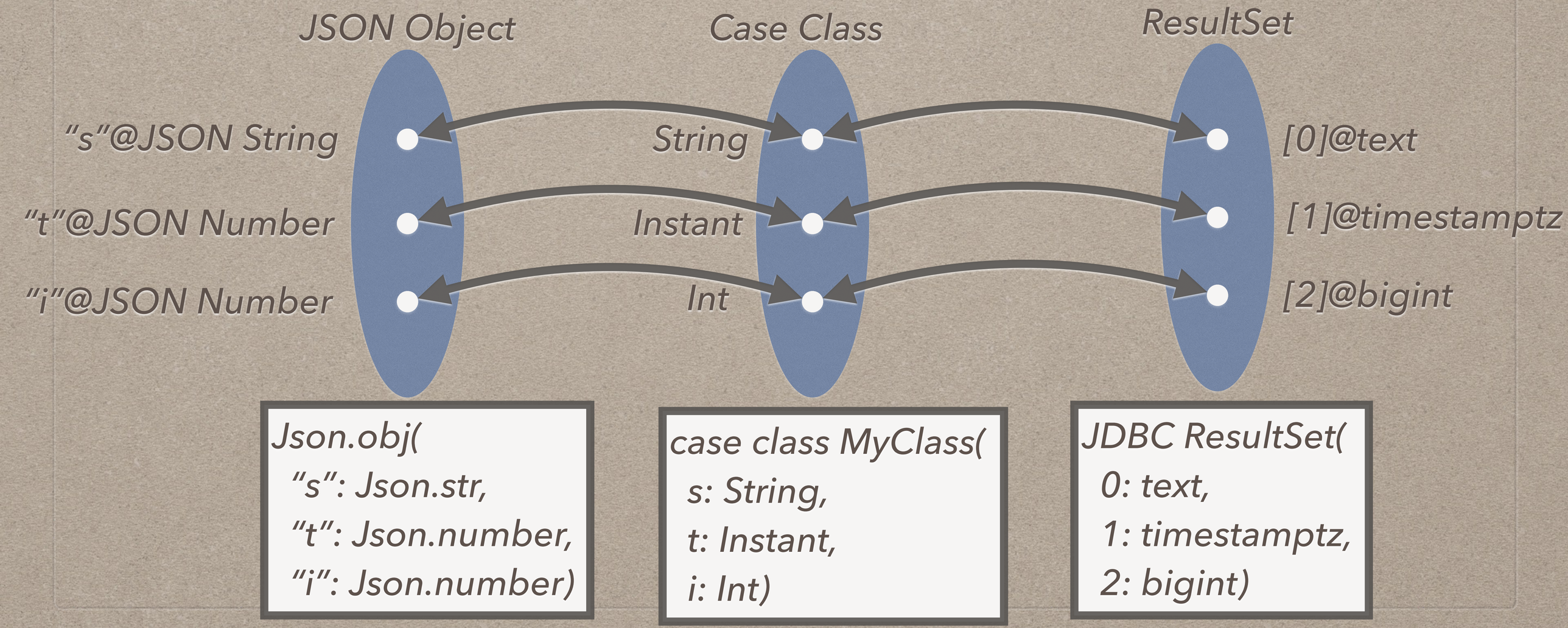
.withGeneratedKeys provides a stream of data, when multiple rows are updated

CONVERTING TO/FROM JDBC


```
case class MyClass(
  s: String,
  t: Instant,
  i: Int)
```

```
JDBC ResultSet(
  0: text,
  1: timestamptz,
  2: bigint)
```

JDBC AND JSON COMBINED



HTTP4S

- http4s is a typeful, purely functional HTTP library for client and server applications written in Scala
- http4s.org/
- https://github.com/http4s/http4s
- https://github.com/TechEmpower/FrameworkBenchmarks/tree/master/frameworks/Scala/http4s (from TechEmpower benchmarks)
- www.lyranthe.org/http4s/ (some programming guides)

WRITING A WEB SERVICE WITH HTTP4S

```
type HttpService = Service[Request, Response]
```

represents

```
Request => Task[Response]
```

Three requirements:

- *Setup service*
- *Parse request*
- *Generate response*

PATTERN MATCHING ON THE REQUEST

Extract Method and Path from Request

```
<METHOD> -> <PATH>
```

For example

```
case GET -> path
```

Extract path components from Path

```
Root / "people" / id
```

PATTERN MATCHING ON THE REQUEST

Extract typed components with extractors

```
import java.util.UUID

object UUIDVar {
  def unapply(s: String): Option[UUID] = {
    try {
      UUID.fromString(s)
    } catch {
      case e: IllegalArgumentException =>
        None
    }
  }
}
```

```
Root / "people" / UUIDVar(uuid)
```

PARSING REQUEST BODY

Register Circe as JSON decoder

```
implicit def circeJsonDecoder[A](implicit decoder: Decoder[A]) =
  org.http4s.circe.jsonOf[A]
```

Decode to a Person object

```
req.decode[Person] { person =>
  ...
}
```

CREATING RESPONSE

Output response code and String as body

```
Ok("Hello world")
```

Output response code and any type with an EntityEncoder

```
implicit def circeJsonEncoder[A](implicit encoder: Encoder[A]) =
  org.http4s.circe.jsonEncoderOf[A]
```

```
Ok(Person(1, "Name"))
```

PUTTING IT ALL TOGETHER

```
case class Person(id: Int, firstName: String, familyName: String, registeredAt: Instant)
case class PersonForm(firstName: String, familyName: String)

object PersonDAO {
  implicit val metaInstant =
    Meta[java.sql.Timestamp].nxmap(
      _.toInstant,
      (i: Instant) => new java.sql.Timestamp(i.toEpochMilli)
    )

  val listPeople: ConnectionIO[List[Person]] =
    sql"select id, first_name, family_name, registered_at from people"
      .query[Person]
      .list

  def getPerson(id: Long): ConnectionIO[Option[Person]] =
    sql"select id, first_name, family_name, registered_at from people where id = $id"
      .query[Person]
      .option

  def updatePerson(id: Int, firstName: String, familyName: String): ConnectionIO[Person] =
    sql"update people set first_name=$firstName, family_name=$familyName where id=$id"
      .update
      .withUniqueGeneratedKeys("id", "first_name", "family_name", "registered_at")

  def insertPerson(firstName: String, familyName: String, registeredAt: Instant = Instant.now()): ConnectionIO[Person] =
    sql"insert into people (first_name, family_name, registered_at) values ($firstName, $familyName, $registeredAt)"
      .update
      .withUniqueGeneratedKeys("id", "first_name", "family_name", "registered_at")
}
```

PUTTING IT ALL TOGETHER

```scala
case class Person(id: Int, firstName: String, familyName: String, registeredAt: Instant)
case class PersonForm(firstName: String, familyName: String)

object DemoService {
  implicit def circeJsonDecoder[A](implicit decoder: Decoder[A]) =
    org.http4s.circe.jsonOf[A]

  implicit def circeJsonEncoder[A](implicit encoder: Encoder[A]) =
    org.http4s.circe.jsonEncoderOf[A]

  def service(xa: Transactor[Task]) = HttpService {
    case GET -> Root / "people" =>
      Ok(PersonDAO.listPeople.transact(xa))

    case GET -> Root / "people" / IntVar(id) =>
      for {
        person <- PersonDAO.getPerson(id).transact(xa)
        result <- person.fold(NotFound())(Ok.apply)
      } yield result

    case req @ PUT -> Root / "people" / IntVar(id) =>
      req.decode[PersonForm] { form =>
        Ok(PersonDAO.updatePerson(id, form.firstName, form.familyName).transact(xa))
      }

    case req @ POST -> Root / "people" =>
      req.decode[PersonForm] { form =>
        Ok(PersonDAO.insertPerson(form.firstName, form.familyName).transact(xa))
      }
  }
}
```

PUTTING IT ALL TOGETHER

```
object Main {
  def main(args: Array[String]): Unit = {
    val xa = DriverManagerTransactor[Task](
      "org.postgresql.Driver", "jdbc:postgresql:demo", "demo", ""
    )

    val server =
      BlazeBuilder
        .bindHttp(8080)
        .mountService(DemoService.service(xa))
        .run

    server.awaitShutdown()
  }
}
```

ADDING HTML TEMPLATING WITH TWIRL

Add to project/plugins.sbt

```
addSbtPlugin("com.typesafe.sbt" % "sbt-twirl" % "1.1.1")
```

Place templates in src/main/twirl

PACKAGING WITH SBT-NATIVE-PACKAGER

Add to project/plugins.sbt

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.1.0-RC1")
```

Enable AutoPlugin

```
enablePlugins(JavaServerAppPackaging)
```

STREAMING DATA WITH SCALAZ-STREAM

- *Response can take a scalaz-stream Process*
- *Doobie can create a scalaz-stream Process from a Resultset*
- *Data sent incrementally using chunked Transfer-Encoding*

```
val streamPeople: Process[ConnectionIO, Person] =
  sql"select id, first_name, family_name, registered_at from people"
    .query[Person]
    .process
```

```
case GET -> Root / "stream" =>
  Ok(PersonDAO.streamPeople.transact(xa).map(p => p.id + "\n"))
```

QUESTIONS?

https://github.com/fiadliel/http4s-talk