Databases: a talk about the brand new library for working with DynamoDB in a purely functional way in Scala

Vlad Podilnyk, SWE at PlayQ

PlayQ

~~ANOTHER DYNAMODB WRAPPER~~

Vlad Podilnyk, SWE at PlayQ

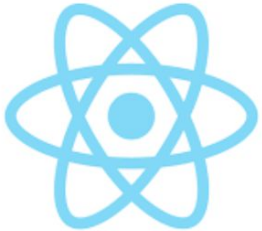
PlayQ

Charm King

PlayQ


TensorFlow


unity




amazon
web services™

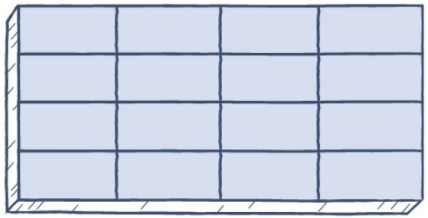
Outlines

1. NoSQL refresher & problem overview (really brief :~))
2. Current solutions overview
3. Top-Secret hyper-pragmatic library

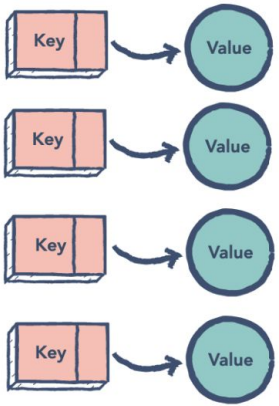
NoSQL DB refresher

VS

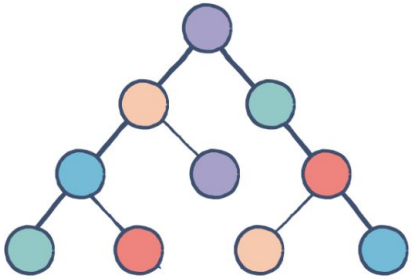
Relational



Key-Value



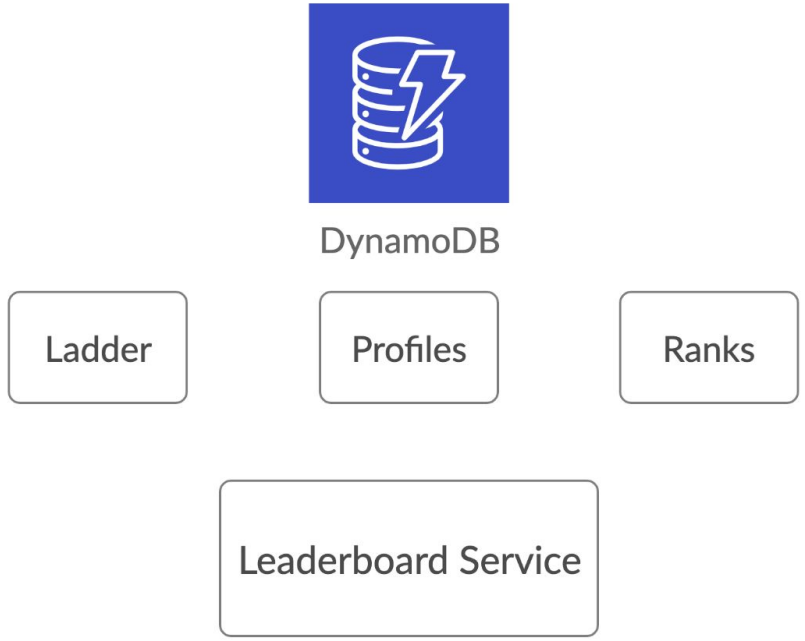
Document



Problem overview

LeaderboardService API:

- GET /LeaderboardService/ladder
- POST /LeaderboardService/ladder/{id}/{score}
- GET /LeaderboadService/profiles/{id}
- POST /LeaderboardService/profiles/{id}
  - -d '{"name": “Vlad", "description": “dev"}'

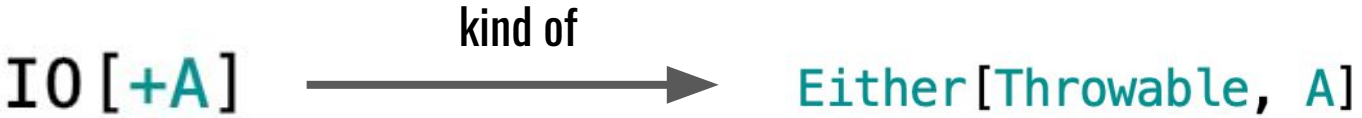


Problem overview

```
trait Ladder[F[_, _]] {
  def submitScore(userId: UserId, score: Score): F[QueryFailure, Unit]
  def getScores: F[QueryFailure, List[UserWithScore]]
}

trait Profiles[F[_, _]] {
  def setProfile(userId: UserId, profile: UserProfile): F[QueryFailure, Unit]
  def getProfile(userId: UserId): F[QueryFailure, Option[UserProfile]]
}

trait Ranks[F[_, _]] {
  def getRank(userId: UserId): F[QueryFailure, Option[RankedProfile]]
}
```



Variance

```
class Foo[+A] // a covariant class
class Bar[-A] // a contravariant class
class D4C[A]  // a invariant class
```

```
trait Vehicle
case class Car(brand: String) extends Vehicle
case class Bus(brand: String) extends Vehicle
```

Foo[Car] <: Foo[Vehicle]

BIO vs IO

```
BIO[F[+_, +_]]
```

- perform side effects before return result type
- could fail with statically typed error
- allows to map over left type parameter
- compose easily with programs that returns other error types
- no runtime error mapping

```
IO[+A]
```

- perform side effects before return result type A
- could fail with Throwable
- has dynamically-typed errors

Use Java directly in Scala code

```scala
def makeClient(cfg: DynamoCfg): DynamoDbClient = {
  DynamoDbClient
    .builder()
    .httpClientBuilder(new ApacheSdkHttpService().createHttpClientBuilder())
    .pipe(_.endpointOverride(URI.create(cfg.uri)))
    .credentialsProvider(
      StaticCredentialsProvider
        .create(
          AwsBasicCredentials.create( accessKeyId = "x",  secretAccessKey = "x")
        )
    )
    .region(Region.of(cfg.region))
    .build()
}
```

Use Java directly in Scala code

```scala
private[java] def createTable[F[+_, +_]: BIO: BlockingIO](client: DynamoDbClient, cfg: DynamoCfg, tableName: String) = {
  val rq = CreateTableRequest
    .builder()
    .tableName(tableName)
    .billingMode(cfg.provisioning.mode)
    .keySchema(
      KeySchemaElement.builder().attributeName( attributeName = "userId").keyType(KeyType.HASH).build()
    )
    .attributeDefinitions(
      AttributeDefinition
        .builder()
        .attributeName( attributeName = "userId")
        .attributeType(ScalarAttributeType.S)
        .build()
    )
    .provisionedThroughput(
      ProvisionedThroughput
        .builder()
        .readCapacityUnits(cfg.provisioning.read)
        .writeCapacityUnits(cfg.provisioning.write)
        .build()
    )
    .build()

  F.syncBlocking {
    client.createTable(rq)
  }
}
```

Lots of boilerplate

Use Java directly in Scala code

```scala
final class AwsLadder[F[+_, +_]: BIO: BlockingIO](client: DynamoDbClient) extends Ladder[F] {

  override def getScores: F[QueryFailure, List[UserWithScore]] = {
    F.syncBlocking(processPages)
      .leftMap(err => QueryFailure(err.getMessage, err))
      .map(_.iterator.map {
        item =>
          (item.get("userId"), item.get("score")) match {
            case (Some(id), Some(score)) => UserWithScore(UserId(UUID.fromString(id.s())), Score(score.n().toLong))
          }
      }.toList)
  }

  override def submitScore(userId: UserId, score: Score): F[QueryFailure, Unit] = {
    val rq = UpdateItemRequest
      .builder()
      .tableName(DynamoHelper.ladderTable)
      .key(Map("userId" -> AttributeValue.builder().s(userId.value.toString).build()).asJava)
      .updateExpression( updateExpression = s"SET score = :score")
      .expressionAttributeValues(Map(":score" -> AttributeValue.builder().n(score.value.toString).build()).asJava)
      .build()

    F.syncBlocking(client.updateItem(rq)).leftMap(err => QueryFailure(err.getMessage, err)).void
  }
```

Find yourself writing another DynamoDB wrapper...

```scala
private[this] def processPages: List[Map[String, AttributeValue]] = {
  import scala.jdk.CollectionConverters._

  @scala.annotation.tailrec
  def loop(acc: List[Map[String, AttributeValue]], rq: ScanRequest): List[Map[String, AttributeValue]] = {
    val rsp   = client.scan(rq)
    val next  = rsp.lastEvaluatedKey()
    val items = rsp.items().asScala.toList.map(_.asScala.toMap)
    if (!next.isEmpty) {
      loop(items ++ acc, rq.toBuilder.exclusiveStartKey(next).build())
    } else
      items ++ acc
  }

  val rq = ScanRequest
    .builder()
    .tableName(DynamoHelper.ladderTable)
    .build()

  loop(List.empty, rq)
}
```

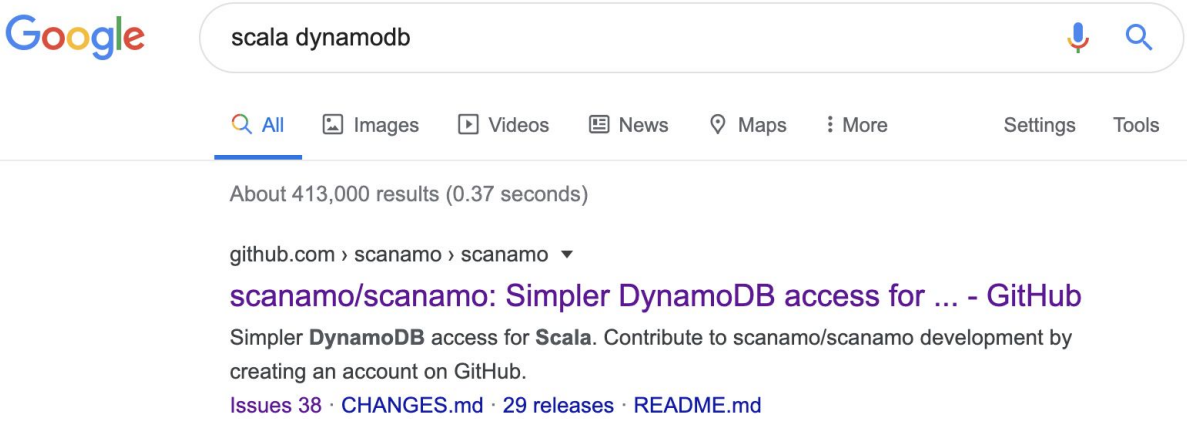
Pros:

- pretty straightforward
- easy to use

Cons:

- a lot of boilerplate
- breaks referential transparency (without BIO or any other effect)

Scanamo


Google
scala dynamodb
All
Images
Videos
News
Maps
More
Settings
Tools
About 413,000 results (0.37 seconds)
github.com › scanamo › scanamo
scanamo/scanamo: Simpler DynamoDB access for ... - GitHub
Simpler DynamoDB access for Scala. Contribute to scanamo/scanamo development by creating an account on GitHub.
Issues 38 · CHANGES.md · 29 releases · README.md

Scanamo

```
import org.scanamo._
import org.scanamo.syntax._
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType._

LocalDynamoDB.createTable(client)("teams")("name" -> S)

case class Team(name: String, goals: Int, scorers: List[String], mascot: Option[String])
```

```
val teamTable = Table[Team]("teams")
scanamo.exec {
  for {
    _ <- teamTable.put(Team("Watford", 1, List("Blissett"), Some("Harry the Hornet")))
    updated <- teamTable.update("name" -> "Watford",
      set("goals" -> 2) and append("scorers" -> "Barnes") and remove("mascot"))
  } yield updated
}
```

Scanamo - batched requests support

```
import org.scanamo._
import org.scanamo.syntax._
import org.scanamo.generic.auto._
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType._

val client = LocalDynamoDB.client()
val scanamo = Scanamo(client)

LocalDynamoDB.createTable(client)("lemmings")("role" -> S)

case class Lemming(role: String, number: Long)
```

```
val lemmingsTable = Table[Lemming]("lemmings")
val ops = for {
  _ <- lemmingsTable.putAll(Set(
    Lemming("Walker", 99), Lemming("Blocker", 42), Lemming("Builder", 180)
  ))
  bLemmings <- lemmingsTable.getAll("role" -> Set("Blocker", "Builder"))
  _ <- lemmingsTable.deleteAll("role" -> Set("Walker", "Blocker"))
  survivors <- lemmingsTable.scan()
} yield (bLemmings, survivors)
val (bLemmings, survivors) = scanamo.exec(ops)
bLemmings.flatMap(_.toOption)
survivors.flatMap(_.toOption)
```

Alpakka & Scanamo = match made in heaven!

```
implicit val system = ActorSystem("scanamo-alpakka")
implicit val materializer = ActorMaterializer.create(system)
implicit val executor = system.dispatcher

val alpakkaClient = DynamoClient(
    DynamoSettings(
      region = "",
      host = "localhost",
      port = 8042,
      parallelism = 2,
      credentialsProvider = DefaultAWSCredentialsProviderChain.getInstance
    )
)

// ...
ScanamoAlpakka(alpakkaCient).execFuture(???)
```

* Requires to create alpakka client and actor system additionally

...but there are drawbacks

Support AWS SDKv2

Extensible/Flexible dsl

Rich DynamoDB operations support

d4s - “DynamoDB Database Done Scala-way”

PlayQ

Let's describe tables first

```
final class LadderTable(implicit meta: DynamoMeta) extends TableDef {
  private[this] val mainKey = DynamoKey(hashKey = DynamoField[UUID]( name = "userId"))

  override val table: TableReference = TableReference("d4s-ladder-table", mainKey)

  override val ddl: TableDDL = TableDDL(table)

  def mainFullKey(userId: UserId): Map[String, AttributeValue] = {
    mainKey.bind(userId.value)
  }
}
```

Let's describe tables first

*Create data type that describes attribute values

```
object LadderTable {
  final case class UserIdWithScoreStored(userId: UUID, score: Long){
    def toAPI: UserWithScore = UserWithScore(UserId(userId), Score(score))
  }
  object UserIdWithScoreStored {
    implicit val codec: D4SCodec[UserIdWithScoreStored] = D4SCodec.derive[UserIdWithScoreStored]
  }
}
```

Ladder Persistence Implementation

```
final class D4SLadder[F[+_, +_]: BIO](connector: DynamoConnector[F], ladderTable: LadderTable) extends Ladder[F] {
  import ladderTable._

  override def getScores: F[QueryFailure, List[UserWithScore]] = {
    connector
      .run( label = "get scores query") {
        table.scan.decodeItems[UserIdWithScoreStored].execPagedFlatten()
      }
      .leftMap(err => QueryFailure(err.queryName, err.cause))
      .map(_.map(_.toAPI))
  }

  override def submitScore(userId: UserId, score: Score): F[QueryFailure, Unit] = {
    connector
      .run( label = "submit user's score") {
        table.updateItem(UserIdWithScoreStored(userId.value, score.value))
      }.leftMap(err => QueryFailure(err.queryName, err.cause)).void
  }
}
```

Take care of initialization and dependency

```
def repoD4S[F[+_, +_]: TagKK]: ModuleDef = new ModuleDef {
  tag(CustomAxis.D4S)

  make[LadderTable]
  make[ProfilesTable]

  many[TableDef]
    .weak[LadderTable]
    .weak[ProfilesTable]

  make[Ladder[F]].from[D4SLadder[F]].named( name = "d4s-ladder")
  make[Profiles[F]].from[D4SProfiles[F]].named( name = "d4s-profiles")
}
```

***TableDef** is already created by *D4SModule*

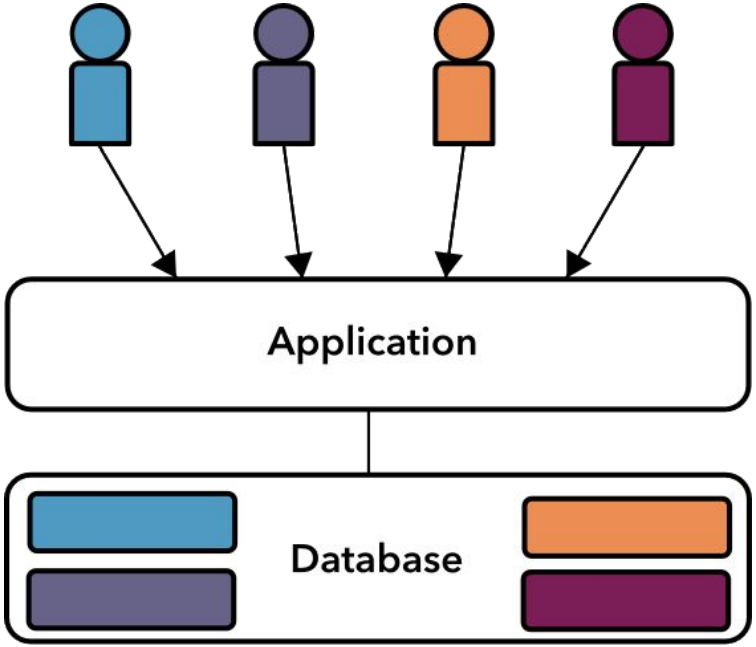
```
object LeaderboardPlugin extends PluginDef {
  //...
  include(d4s.modules.D4SModule[IO])
```

* Creates all necessary stuff for you, such as DDLService, TableService, etc

Deal with data using streams

```
connector.runStreamed( label = "streamed-query") {
  usersTable.query
    .decodeItems[UserInfo]
    .withKey(mainKey.bind(user.user))
    .execStreamed
    .retryWithPrefix(ddl)
}
```

Satisfy multi-tenancy restriction using .retryWithPrefix combinator



Perform batched operations...

```
connector.run( label = "delete-query-batched") {
  table
    .deleteItemBatch(items.map(c => mainKey.bind(c.userId, c.couponId)))
    .withPrefix(appID)
}
```

```
connector.run( label = "batched-request") {
  ticketsTable.table
    .getItemBatch(tickets.map(ticketsTable.mainKey.bind))
    .decodeItems[TicketStored]
    .withPrefix(appId)
    .retryWithPrefix(ticketsTable.ddl)
}
```

... and even more complex queries

```
delete = testTable.table
  .queryDeleteBatch(testTable.mainKey.bind( hashValue = "batch_test"))
  .withPrefix(prefix)
  .retryWithPrefix(testTable.ddl)
connector.runUnrecorded(delete)
```

* perform Query request and then DeleteItemBatch in parallel

Handle pagination with elegance

```
table
  .query(key)
  .decodeItems[TableRow]
  .execPagedFlatten()
  .prefixed(appLadderId.appId)
```

Use condition expressions with ease

```
connector.run( label = "add-item-to-cart") {
  table.putItem(item).ifNotExists().optConditionFailure.void
}
```

```
val queryRequest =
  table
    .query(ladderScoreIndex)
    .withCondition(ladderScoreKeys.hashKey === FullLadderId(ladder) && orderingOp(scoreDateKey, scoreDate))
    .filterTtl(now)
    .decodeItems[Item]
```

```
connector.run( label = "begins-with-query") {
  table
    .query(userIndex, userKey(userId))
    .withCondition("nickName".of[String] beginsWith prefix)
    .decodeItems[User]
}
```

Get only what you want

```
connector
  .run( label = "get scores query") {
    table.scan
      .withProjectionExpression( expr = "score")
      .decodeItems[Score]
      .execPaged()
  }.map(_.flatten)
```

GOD Mode = modify already completed queries

```
connector.run( label = "modify-combinator") {
  table
    .updateItem(key)
    .prefixed(appId)
    .modify(_.withItem(item))
    .void
}
```

Describe secondary indexes

```
val globalIndex = GlobalIndex(
    "Global-Index-Name",
    userSecondaryKey,
    Projection
      .builder()
      .projectionType(ProjectionType.ALL)
      .build()
  )
```

```
val localIndex = LocalIndex(
    "Local-Index_Name",
    scoreTableKeys,
    Projection
      .builder()
      .projectionType(ProjectionType.ALL)
      .build()
  )
```

```
override val ddl: TableDDL =
  TableDDL(
    tableReference = table,
    localIndexes   = Set(localIndex),
    globalIndexes  = Set(globalIndex)
  )
```

Update expressions

```
connector.run( label = "add value") {
  countersTable.table
    .updateItem(Table1Key(v1, v2))
    .withUpdateExpression( expr = "ADD counterField :value")
    .withAttributeValues( value = ":value" -> DynamoAttributeEncoder.encodeAttribute(0L))
}
```

Create custom queries using RawRequest

```
val rawRequest = new RawRequest[DescribeLimitsRequest, DescribeLimitsResponse] {
  override def interpret[F[+_, +_]](rq: DescribeLimitsRequest)
                                   (implicit F: bio.BIOTemporal[F], client: DynamoClient[F]): F[Throwable, DescribeLimitsResponse] = {
    F.syncThrowable(client.raw(_.describeLimits(this.toAmz))).flatten
  }

  override def toAmz: DescribeLimitsRequest = {
    DescribeLimitsRequest.builder().build()
  }
```

* *RawRequest.raw* takes DynamoClient and perform a raw operation on it

```
listTablesRq    = RawRequest.raw(_.raw(_.listTables()))
listTablesQuery = listTablesRq.toQuery.decode(_.tableNames().asScala.toSet)
tablesFromAws <- connector.runUnrecorded(listTablesQuery)
```

TTL fields supported !!!

```
val tableRef = TableReference("name", mainKey, Some(ttlField))
```

```
connector.run( label = "query-name") {
  table
    .query(key)
    .filterTtl(now)
    .execPaged()
}
```

Use utility operations via DynamoTableService or deal with it directly

The following methods are provided by DynamoTableService

```
trait DynamoTablesService[F[_, _]] {
  def create(tables: Set[TableDef]): F[Throwable, Unit]
  def createPrefixed[P: TablePrefix](prefix: P)(tables: Set[TableDef]): F[Throwable, Unit]

  def delete(tables: Set[TableDef]): F[Throwable, Unit]
  def deletePrefixed[P: TablePrefix](prefix: P)(tables: Set[TableDef]): F[Throwable, Unit]

  def listTables: F[Throwable, List[String]]
  def listTablesByRegex(regex: Regex): F[Throwable, List[String]]
}
```

Also, the one could use connector & DynamoExecution to achieve the same

```
connector.runUnrecorded(DynamoExecution.listTables.map(_.toSet))
```

Rich support for almost all DynamoDB operations with ability to write custom queries

all operations related to individual items in the table, such as GetItem, UpdateItem, Scan, Query, etc.

batched operations + raw requests + complex queries on top of that (QueryDeleteBatch)

supports utility functions such as UpdateTable, Describe, ListTables, etc.

Things that makes you even happier (more productive)

PlayQ

Exposed DI modules

Boost your productivity with DIstage

```
def base[F[+_, +_]: TagKK]: ModuleDef = new ModuleDef {
  make[DynamoClient[F]].from[DynamoClient.Impl[F]]
  make[DynamoConnector[F]].from[DynamoConnector.Impl[F]]
  make[DynamoInterpreter[F]].from[DynamoInterpreter.Impl[F]]
  make[DynamoTablesService[F]].from[DynamoTablesService.Impl[F]]

  make[DynamoDBHealthChecker[F]]
  make[DynamoDDLService[F]].fromResource[DynamoDDLService[F]]
  make[DynamoComponent].fromResource[DynamoComponent.Impl[F]]

  many[TableDef]

  make[PortCheck].from(new PortCheck( timeout = 3))
}
```

No more manual tables setup !

Metrics

*d4s collects metrics for you!

```
➜  scalaua2020 git:(master) ✗ curl -X GET http://localhost:8080/LeaderboardService/metrics | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   791  100   791    0     0   7203      0 --:--:-- --:--:-- --:--:--  7256
[
  {
    "timer": {
      "role": "leaderboard",
      "label": "dynamo:get-profile:timer",
      "initial": 0
    }
  },
  {
    "meter": {
      "role": "leaderboard",
      "label": "dynamo:request-error:get-profile",
      "initial": 0
    }
  },
  {
    "meter": {
      "role": "leaderboard",
      "label": "dynamo:request-error:get scores query",
      "initial": 0
    }
  },
  {
    "timer": {
      "role": "leaderboard",
      "label": "dynamo:set-profile:timer",
      "initial": 0
    }
  },
```

Structured logs included

```
: [73:pool-3-thread-4 ] Dynamo: Got error during executing operation=Scan for tableName=dev-d4s-ladder-table. Failure=software.amazon.aw
:ssion: Attribute name is a reserved keyword; reserved keyword: user (Service: DynamoDb, Status Code: 400, Request ID: 21aed718-c291-4a3
lleErrorResponse(HandleResponseStage.java:115)
lleResponse(HandleResponseStage.java:73)
:ute(HandleResponseStage.java:58)
```

```
ilure: Query "get scores query" failed with Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved keyword: user (Service:
d4s.D4SLadder.$anonfun$getScores$1(D4SLadder.scala:18)
onfun$andThen$1(Function1.scala:85)
ause.scala:95)
Cause.scala:93)
Map(Cause.scala:467)
```

Additionally...

- implements codecs using Circe and Magnolia
- provides dynamo health check for you
- provides DynamoConnector via ZIOenv
- supports Scala 2.13 & 2.12

Thank you for your attention

PlayQ

Appendix

https://github.com/PlayQ/d4s

https://github.com/VladPodilnyk/scalaua2020

https://github.com/7mind/izumi