mediarithmics

Start Enjoying Your Data

Hibernate : du printemps à l'été avec la DB monad

contact@mediarithmics.com

mediarithmics

Nos Speakers

Matthieu Grenonville
Lead Developer
mgrenonville@mediarithmics.com
mediarithmics

Loïc Girault
Software Engineer
lgirault@mediarithmics.com
mediarithmics

mediarithmics

Sommaire

mediarithmics

Le Scava historique

Historique

- Remonte à mi-2013
- La mode de Grails/Spring Roo
- Une erreur de jeunesse ?
  - Manque d'expérience en FP
  - Spring @Transactional est un standard de facto

mediarithmics

ScavaEE™

```java
@Entity
public class Organisation implements EntityWithId {
    @Id
    @Column(name = "id")
    private Long id;

    @Version
    @Column(name = "version")
    private Integer version;

    private String name;
    // getters/setters
}
```

```
@Repository
class OrganisationRepository
 extends RepositoryCRUD[Organisation]
 with SearchRepository[OrganisationSearchCriteria,
Organisation]{

 @PersistenceContext
 var entityManager: EntityManager = _

 val clazz = classOf[Organisation]
 // ….
```

```
@Service("organisationService")
class OrganisationServiceImpl @Autowired()(organisationRepository: OrganisationRepository) {

 @Transactional
 override def findOrganisations(criteria : OrganisationSearchCriteria): PagedResult[Organisation] =
   organisationRepository.findEntities(criteria)
```

mediarithmics

Spring Transaction - Sous le capot

```
public abstract class TransactionAspectSupport implements BeanFactoryAware, InitializingBean {
  private static final ThreadLocal<TransactionInfo> transactionInfoHolder =
          new NamedThreadLocal<>("Current aspect-driven transaction");

  protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass,
                                           final InvocationCallback invocation) throws Throwable {
      TransactionInfo txInfo = createTransactionIfNecessary(....);
      transactionInfoHolder.set(txInfo);
      Object retVal = null;
      try {
          retVal = invocation.proceedWithInvocation();
      } catch (Throwable ex) {
          completeTransactionAfterThrowing(txInfo, ex); throw ex;
      } finally {
          cleanupTransactionInfo(txInfo);
      }
      commitTransactionAfterReturning(txInfo);
      return retVal;
  }
}
```

mediarithmics

Spring Transaction - ThreadLocal ?

```
class ThreadLocal[T] {
 val threadToValue: mutable.Map[Thread, T] = ???

 def get(): T =  threadToValue(Thread.currentThread())
 def set(t: T) = threadToValue += Thread.currentThread() -> t

}
```

mediarithmics

Spring Transaction - ThreadLocal + Future = 😱

```
val threadLocal = new ThreadLocal[String]()

threadLocal.set("Hello from main thread !")
println(Thread.currentThread() + ": " + threadLocal.get())
// Thread[run-main-d,5,run-main-group-d]: Hello from main thread !

Future {
 println(Thread.currentThread() + ": " + threadLocal.get())
// Thread[scala-execution-context-global-1343,5,run-main-group-d]: null
}
```

Sur le chemin...

Tentative de Slick 2.0, avec le cake pattern
et des Transactions en paramètres implicits

Guice + JPA (pour retirer Spring)

mediarithmics

KPI sur le code Legacy

```
$ find . | grep '\.java$' | wc -l
193

$ git grep 'Transactional' | wc -l
670
```

mediarithmics

Vers un scala **plus fonctionnel**

mediarithmics

Point de départ

```
class Transactioner(getEntityManager: () => EntityManager) {
 def transactionally[A](a: => A): A = {
   val em = getEntityManager()
   val ts = em.getTransaction
   ts.begin()
   Try(a) match {
      case Success(res) => ts.commit(); em.close(); res
      case Failure(exc) => ts.rollback(); em.close(); throw exc
   }
 }
}
```

mediarithmics

Problèmes

```
val tcter : Transactioner = ???

def foo(): A = tcter.transactionally( doFoo() )

def bar() : B =  tcter.transactionally{
  doBar( foo() ) // ne partagent pas la même transaction !!
```

Comment gérer le **contexte** ?
Des **implicites** ?
Explicitement et **manuellement** ?

Monad !

mediarithmics

The M word

```
trait Monad[F[_]] {

  def pure[A](a : A) : F[A] // a.k.a point, finish or return

  def flatMap[A, B](fa : F[A])(f: A => F[B]) : F[B] // a.k.a. chain or bind

  def map[A, B](fa : F[A])(f : A => B): F[B] // a.k.a. as … map

}
```

mediarithmics

case class Kleisli[F[_], A, B](run: A => F[B])

```
type Reader[A, B] = Kleisli[Id, A, B]

def monadInstance[F[_] : Monad, A] =   new Monad[Kleisli[F, A, ?]] {
    override def pure[B](x: B): Kleisli[F, A, B] =  Kleisli(_ => Monad[F].pure(x))

    override def flatMap[B, C](fa: Kleisli[F, A, B])
                                          (f: B => Kleisli[F, A, C]): Kleisli[F, A, C] =
      Kleisli { a =>
        for {
          b <- fa.run(a) // run : a => F[B]
          c <- f(b).run(a) // run : a => F[C]
        } yield c
      }
  }
```

mediarithmics

Type class & concrete types

```
trait Transactioner[F[_], EM] {
  def transactionally[A](fa: EM => F[A]): F[A]
}

type TxContext = (EntityManager, EntityTransaction)

type TransactionState = Option[TxContext]

type Transactionable[M[_], A] = Kleisli[M, TransactionState, A]
```

mediarithmics

Implémentation de Transactioner

```
def instance[F[_]](getEntityManager: () => EntityManager)
                  (implicit S: Sync[F]): Transactioner[Transactionable[F, ?], EntityManager] =
new Transactioner[Transactionable[F, ?], EntityManager] {
  val KS = Sync.catsKleisliSync[F, TransactionState]

  override def transactionally[A](fa: EntityManager => Transactionable[F, A]): Transactionable[F, A] =
    for {
      state <- Kleisli.ask[F, TransactionState] //  Kleisli(a => F.pure(a))
      result <-
        state match {
          case Some((em, _)) => fa(em)
          case None => transactionBracket(getEntityManager, fa)(KS)
        }
    } yield result
}
```

mediarithmics

Bracket

```
sealed abstract class ExitCase[+E]
final case object Completed extends ExitCase[Nothing]
final case class Error[+E](e: E) extends ExitCase[E]
final case object Canceled extends ExitCase[Nothing]

trait Bracket[F[_], E] extends MonadError[F, E] {

def bracketCase[A, B](acquire: F[A])(use: A => F[B])
            (release: (A, ExitCase[E]) => F[Unit]): F[B]
}
```

mediarithmics

TransactionBracket 1/4 : les signatures

```
private def transactionBracket[F[_], A](getEntityManager: () => EntityManager,
                                        fa: EntityManager => Transactionable[F, A])
                                       (implicit S: Sync[Transactionable[F, ?]]): Transactionable[F, A] = {

val acquire: Transactionable[F, TxContext] = ???

val use: TxContext => Transactionable[F, A] = ???

val release: (TxContext, ExitCase[Throwable]) => Transactionable[F, Unit] = ???

S.bracketCase[TxContext, A](acquire)(use)(release)

}
```

mediarithmics

TransactionBracket 2/4 : acquisition

```
private def transactionBracket[F[_], A](getEntityManager: () => EntityManager,
                                        fa: EntityManager => Transactionable[F, A])
                                       (implicit S: Sync[Transactionable[F, ?]]): Transactionable[F, A] = {

val acquire: Transactionable[F, TxContext] =
S.delay {
  val em = getEntityManager()
  val t = em.getTransaction
  t.begin()
  (em, t)
}
//…
```

mediarithmics

TransactionBracket 3/4 : utilisation

```
private def transactionBracket[F[_], A](getEntityManager: () => EntityManager,
                                        fa: EntityManager => Transactionable[F, A])
                                       (implicit S: Sync[Transactionable[F, ?]]): Transactionable[F, A] = {
//…
val use: TxContext => Transactionable[F, A] = {
case (em, t) =>
  Kleisli.local[F, A, TransactionState] {
    _ => //set transaction, current state = None
      Some((em, t))
  }(fa(em))}

//…
```

```
def local[M[_], A, R](f: R => R)
                                                (fa: Kleisli[M, R, A]): Kleisli[M, R, A] =
  Kleisli(a => fa.run(f(a)))
```

mediarithmics

TransactionBracket 4/4 : libération

```
private def transactionBracket[F[_], A](getEntityManager: () => EntityManager,
                                        fa: EntityManager => Transactionable[F, A])
                                       (implicit S: Sync[Transactionable[F, ?]]): Transactionable[F, A] = {
//…
val release: (TxContext, ExitCase[Throwable]) => Transactionable[F, Unit] = {
  case ((em, t), ExitCase.Completed) =>   S.delay { t.commit(); em.close() }

  case ((em, t), ExitCase.Canceled) =>  S.delay { t.rollback(); em.close() }

  case ((em, t), ExitCase.Error(err)) =>  S.suspend { t.rollback(); em.close(); S.raiseError(err) }
 }

S.bracketCase[TxContext, A](acquire)(use)(release)
}
```

mediarithmics

Le retour d'Hibernate

```
trait EntityManager[F[_], EM] {

def findById[A: ClassTag](em: EM, id: Long): F[A]

def persist(em: EM, a: Any): F[Unit]

def merge[A](em: EM, a: A): F[A]

def remove[A <: EntityWithId : ClassTag](em: EM,
 e: A): F[Unit]

}
```

```
import javax.persistence.{EntityManager => JEM}

object EntityManager {

def instance[F[_]](implicit S: Sync[F]) =
         new EntityManager[F, JEM] {
             def persist(em: JEM, a: Any): F[Unit] =
                    S.delay {
                       em.persist(a)
                       em.flush()
                    }
         // ...
}
```

mediarithmics

Le reste du puzzle

```
class DB[F[_], EM: EntityManager[F, ?] : Transactioner[F, ?]] {

def transactionally[A](fa: => F[A]): F[A] =  Transactioner[F, EM].transactionally(_ => fa)

def findById[A: ClassTag](id: Long): F[A] =
    Transactioner[F, EM].transactionally(EntityManager[F, EM].findById(_, id))

def persist(a: Any): F[Unit] = ...

def merge[A](a: A): F[A] = ...

def remove[A <: EntityWithId : ClassTag](e: A): F[Unit] = ...

}
```

mediarithmics

DB en pratique !

```
class UserService[F[_], EM](implicit DB: DB[F, EM], M : Monad[F]) {

 def addUserToGroup(userId: TLong[UserId], groupId: TLong[GroupId]): F[Unit] =
 DB.transactionally {
    for {
      user <- DB.findById[User](userId)
      group <- DB.findById[Group](groupId)
      _ = user.addGroup(group)
      _ <- DB.persist(user)
    } yield ()
 }

}
```

mediarithmics

Tests

```
case class TestContext(users: mutable.Map[Long, UserEntity], groups: mutable.Map[Long, GroupEntity])

object TestContext {

implicit def EntityManager = new EntityManager[Id, TestContext] {

  override def findById[A: ClassTag](em: TestContext, id: Long): Id[A] = {
      val clazz = implicitly[ClassTag[A]].runtimeClass
      clazz.getSimpleName match {
           case "User" => em.users(id).asInstanceOf[A]
           case "Group" => em.groups(id).asInstanceOf[A]
           case n => throw new Exception(s"unknown entity $n")
    }
  }
```

mediarithmics

Le bout du monde

```
application.conf :
play.application.loader = "com.mediarithmics.CustomApplicationLoader"

class CustomApplicationLoader extends GuiceApplicationLoader() {
  override def builder(context: ApplicationLoader.Context): GuiceApplicationBuilder =
     super.builder(context).bindings(new CustomModule)
}

class CustomModule extends AbstractModule {
override def configure(): Unit = ()

@Singleton  @Provides def provideSessionFactory(DBApi: DBApi): SessionFactory = ???

@Singleton @Provides  def provideDB(sessionFactory: SessionFactory): DB[TransactionableIO, EntityManager] =
  ioInstance(() => sessionFactory.createEntityManager())(IOContextShift.global)

//...
```

mediarithmics

Try it out !

https://github.com/MEDIARITHMICS/psug-db-monad

mediarithmics

Start Enjoying Your Data

Merci !

contact@mediarithmics.com