

Programación funcional en Scala

Tema 5 - Efectos, EDSLs e Interpretación



Juan Manuel Serrano
Habla Computing
juanmanuel.serrano@hablapps.com
[@jmshac](https://twitter.com/jmshac)

Índice

- ¿Qué son los efectos?
 - Los *efectos de lado*
 - Efectos de lado y modularidad
- ¿Cómo se programan funcionalmente?
 - Instrucciones monádicas
 - Librerías de operadores
 - Azúcar sintáctico
- Conclusión

Un **efecto** es cualquier *interacción* del programa con el entorno en el que se ejecuta

Ejemplos de *efectos*

- E/S
 - Lectura/escritura en pantalla
 - Lectura/escritura de ficheros
- Server interactions
 - Web service request/response
 - Email request/receptions
 - Database queries
 - Logging
- Otros
 - Excepciones, *stateful computations*, futuros, etc.

ScalaMAD: Scala Programming @ Madrid

Home Members Sponsors Photos Discussions More

Join us!

Madrid,
Founded Jun

Scaleros

Group review

Upcoming
Meetups

Past Meetups

Our calendar



Organizers

Juan Manuel
Serrano,
Jesús López-
González,
Nando Sola



Bienvenido:

Upcoming 5

Suggested 0

Past

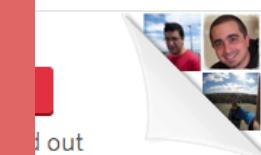
Calendar

Scala for dummies

Scala es un lenguaje de programación orientado a objetos y, a la vez, funcional. La combinación de estos dos paradigmas ha llevado a la creación de iScala. En este sitio web podrás encontrar información sobre el lenguaje, sus características y aplicaciones. Además, iScala tiene como objetivo promover la programación funcional entre los desarrolladores de Scala. Si estás interesado en aprender más sobre Scala, te recomendamos que visites la sección "Aprende Scala" y comiences con los tutoriales y ejemplos de código.

- Si el usuario o el grupo no existen se ignora la solicitud
- Si el usuario ya pertenece al grupo, se ignora la solicitud
- Si la entrada de nuevos miembros no está moderada, se crea el nuevo miembro
- Si la entrada está moderada y el usuario ya tenía una petición de entrada registrada, la nueva petición se ignora
- Si la entrada está moderada, se deja constancia de la petición

Scala es un lenguaje de programación orientado a objetos y, a la vez, funcional. La combinación de estos dos paradigmas ha llevado a la creación de iScala. En este sitio web podrás encontrar información sobre el lenguaje, sus características y aplicaciones. Además, iScala tiene como objetivo promover la programación funcional entre los desarrolladores de Scala. Si estás interesado en aprender más sobre Scala, te recomendamos que visites la sección "Aprende Scala" y comiences con los tutoriales y ejemplos de código.



What's new





<https://github.com/hablapps/training/tree/master/scala/examples/meetup>

```
def join(request: JoinRequest): JoinResponse
```

```
case class JoinRequest(  
    jid: Option[Int],  
    uid: Int,  
    gid: Int)
```

```
case class User(  
    uid: Option[Int],  
    name: String)
```

```
case class Group(  
    id: Option[Int],  
    name: String,  
    city: String,  
    must_approve: Boolean)
```

```
type JoinResponse =  
    Either[JoinRequest, Member]
```

```
case class Member(  
    mid: Option[Int],  
    uid: Int,  
    gid: Int  
)
```

```
def join(request: JoinRequest): JoinResponse = {
    val JoinRequest(_, uid, gid) = request
    DB.withSession { implicit session =>
        val group: Option[Group] = (for {
            group <- group_table if group.gid === gid
        } yield group).firstOption
        val user: Option[User] = (for {
            user <- user_table if user.uid === uid
        } yield user).firstOption
        if (!group.isDefined)
            throw new RuntimeException(s"Group $gid not found")
        else if (!user.isDefined)
            throw new RuntimeException(s"User $uid not found")
        else if (group.get.must_approve) {
            val maybeId = join_table returning join_table.map(_.jid) += request
            Left(request.copy(jid = maybeId))
        } else {
            val mid = member_table returning member_table.map(_.mid) +=
                Member(None, uid, gid)
            Right(Member(mid, uid, gid))
        }
    }
}
```



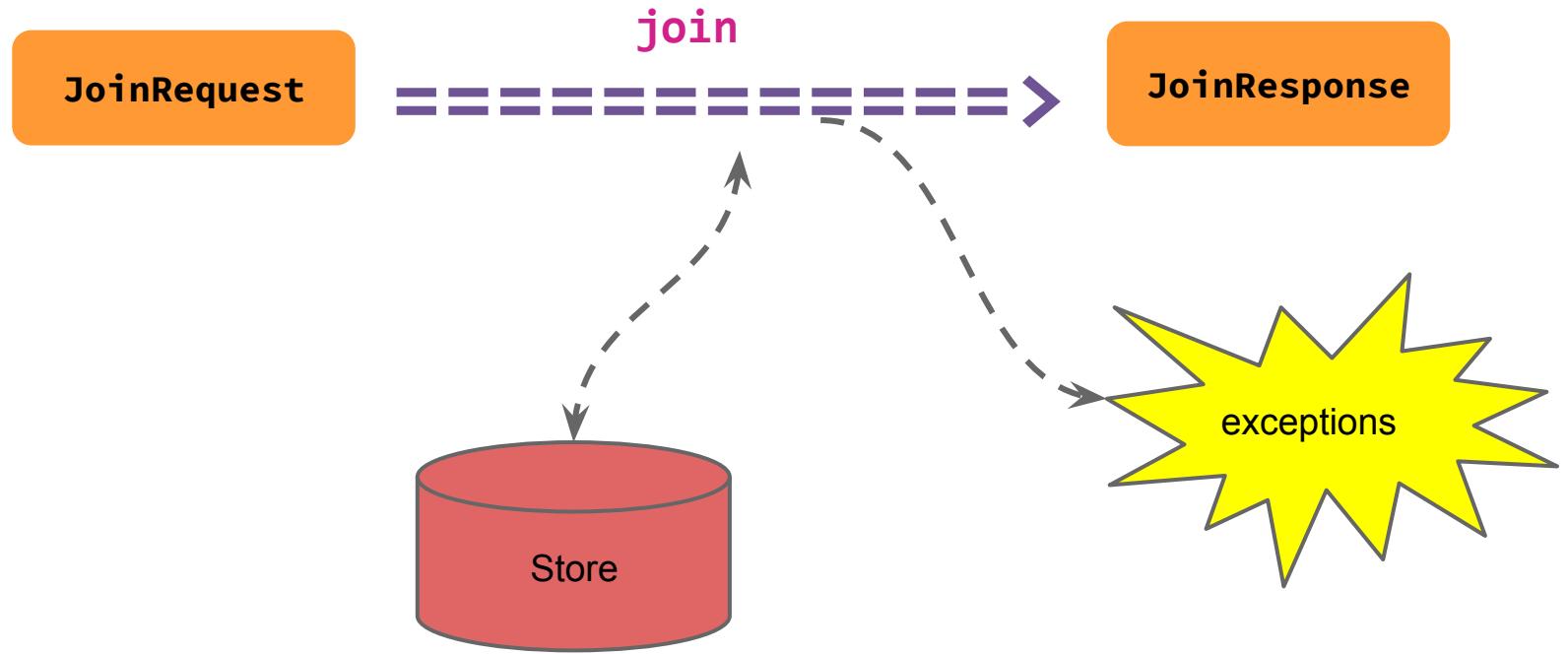
```

def join(request: JoinRequest): JoinResponse = {
  val JoinRequest(_, uid, gid) = request
  DB.withSession { implicit session =>
    val group: Option[Group] = (for {
      group <- group_table if group.gid === gid
    } yield group).firstOption
    val user: Option[User] = (for {
      user <- user_table if user.uid === uid
    } yield user).firstOption
    if (!group.isDefined)
      throw new RuntimeException(s"Group $gid not found")
    else if (!user.isDefined)
      throw new RuntimeException(s"User $uid not found")
    else if (group.get.must_approve) {
      val maybeId = join_table returning join_table.map(_.jid) += request
      Left(request.copy(jid = maybeId))
    } else {
      val mid = member_table returning member_table.map(_.mid) +=
        Member(None, uid, gid)
      Right(Member(mid, uid, gid))
    }
  }
}

```



Un **efecto de lado** es cualquier interacción de la función con el entorno *no declarada* en la signatura de la función



¿Por qué son malos los *side effects*?

Comprendibilidad

- ¿Podemos entender fácilmente cómo está estructurada nuestra aplicación?
- ¿Podemos entender fácilmente cómo funciona?

Testability

- ¿Podemos hacer pruebas unitarias de nuestros módulos?

¿Por qué son malos los *side effects*?

Modularidad

- ¿Podemos estructurar nuestra aplicación en módulos que puedan ser *compuestos, modificados y reutilizados* fácilmente?

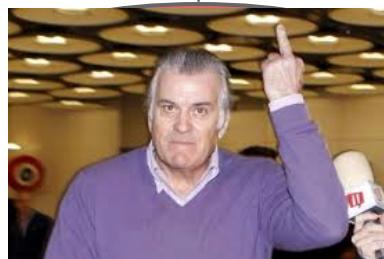
Eficiencia/escalabilidad

- ¿hacemos uso realmente de los multi-cores?
- ¿Podemos distribuir nuestra aplicación en la nube?

JoinRequest

join

JoinResponse



La solución *orientada a objetos*

```
trait Store{  
    def getGroup(gid: Int): Group  
    def getUser(uid: Int): User  
    def putJoin(join: JoinRequest): JoinRequest  
    def putMember(member: Member): Member  
    def isMember(uid: Int, gid: Int): Boolean  
    def isPending(uid: Int, gid: Int): Boolean  
}
```



```
trait MySQLStore extends Store{  
    def getGroup(gid: Int): Group =  
        DB.withSession { implicit session => ... }  
    ...  
}
```

La solución *orientada a objetos*

```
trait Services{ self: Store =>

    def join(request: JoinRequest): JoinResponse = {
        val JoinRequest(_, uid, gid) = request

        val _ = getUser(uid)
        val group = getGroup(gid)
        if (group.must_approve)
            Left(putJoin(request))
        else
            Right(putMember(Member(None, uid, gid)))
    }

}
```

La solución *orientada a objetos*

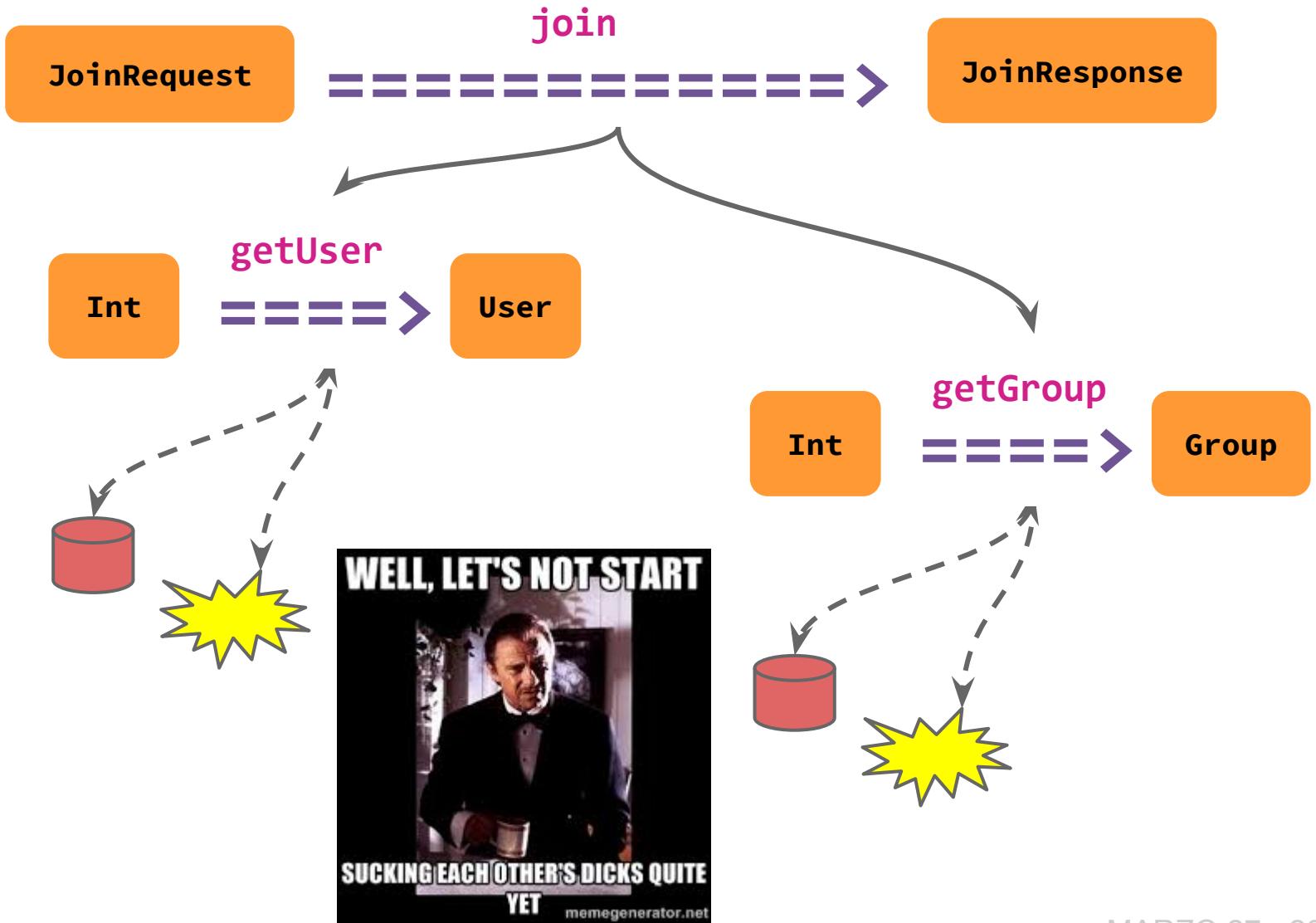
```
trait Services{ self: Store =>

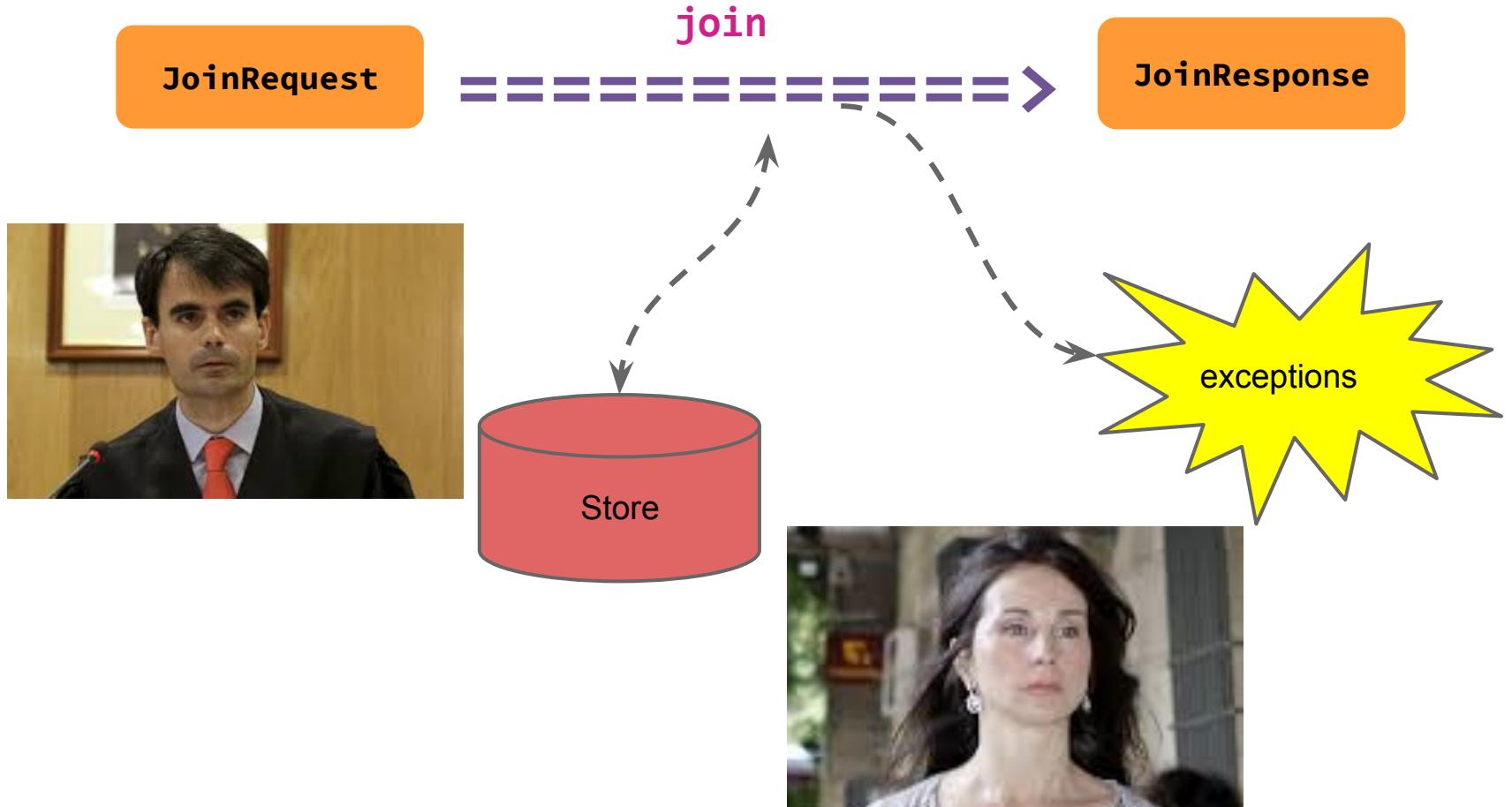
    def join(request: JoinRequest): JoinResponse = {
        val JoinRequest(_, uid, gid) = request

        val _ = getUser(uid)
        val group = getGroup(gid)
        if (group.must_approve)
            Left(putJoin(request))
        else
            Right(putMember(Member(None, uid, gid)))
    }

}
```



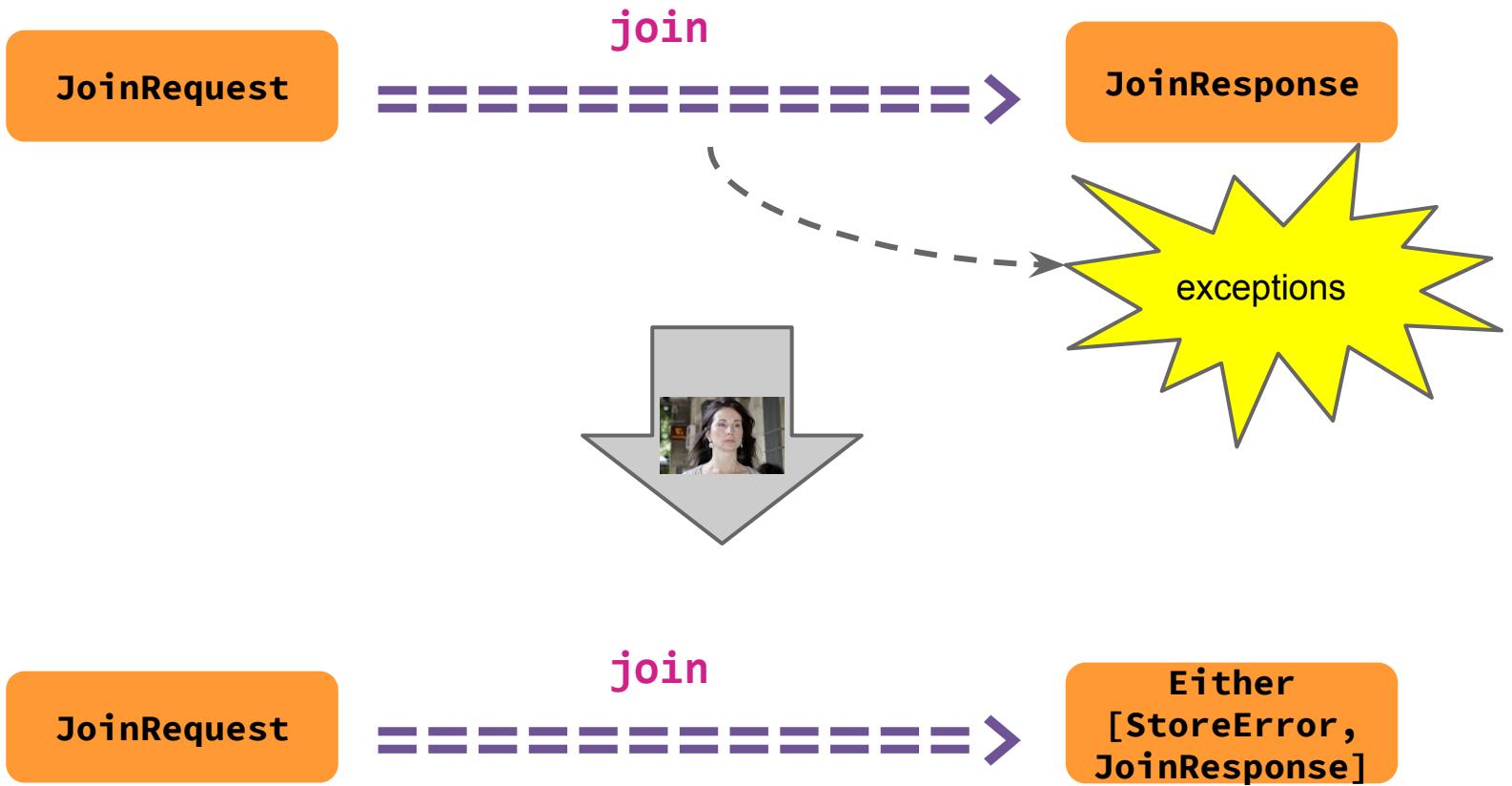




Índice

- ¿Qué son los efectos?
 - Los *efectos de lado*
 - Efectos de lado y modularidad
- ¿Cómo se programan funcionalmente?
 - Instrucciones monádicas
 - Librerías de operadores
 - Azúcar sintáctico
- Conclusión

Eliminando excepciones



Eliminando excepciones

```
sealed abstract class StoreError(val msg: String)

case class NonExistentEntity(id: Int)
  extends StoreError("...")

case class ConstraintFailed(constraint: Store[Boolean])
  extends StoreError("...")

case class GenericError(override val msg: String)
  extends StoreError(msg)
```



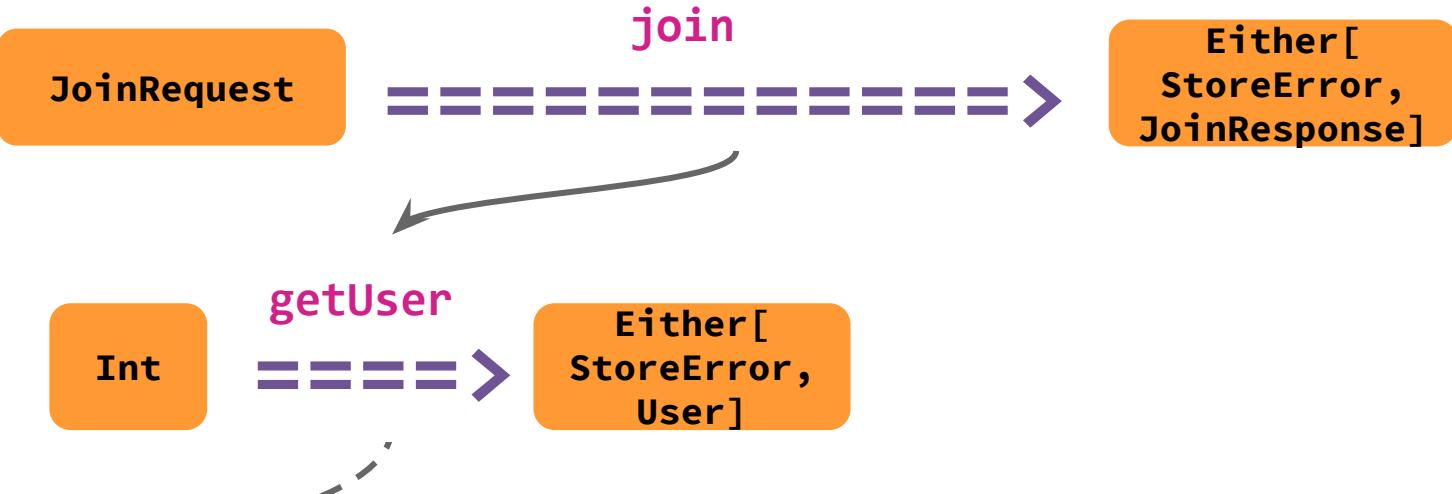
Eliminando excepciones

```
trait Store{
    def getGroup(gid: Int): Either[StoreError, Group]
    def getUser(uid: Int): Either[StoreError, User]
    def putJoin(join: JoinRequest): Either[StoreError, JoinRequest]
    def putMember(member: Member): Either[StoreError, Member]
    ...
}
```

Eliminando excepciones

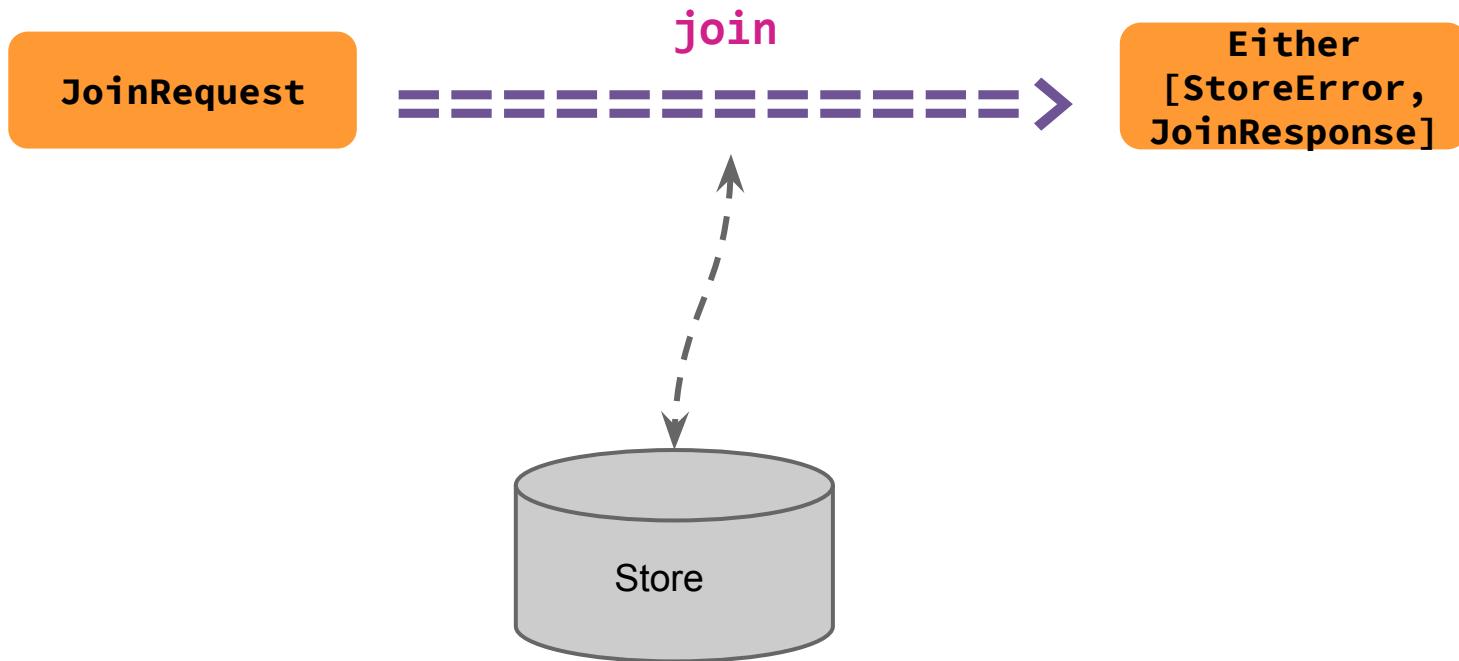
```
def join(request: JoinRequest): Either[StoreError, JoinResponse] =  
{  
    val JoinRequest(_, uid, gid) = request  
    getUser(uid).right.flatMap( user =>  
        getGroup(gid).right.flatMap( group =>  
            if (group.must_approve)  
                putJoin(request).right.map(Left.apply)  
            else  
                putMember(Member(None, uid, gid)).right.map(Right.apply)  
        )  
    )  
}
```



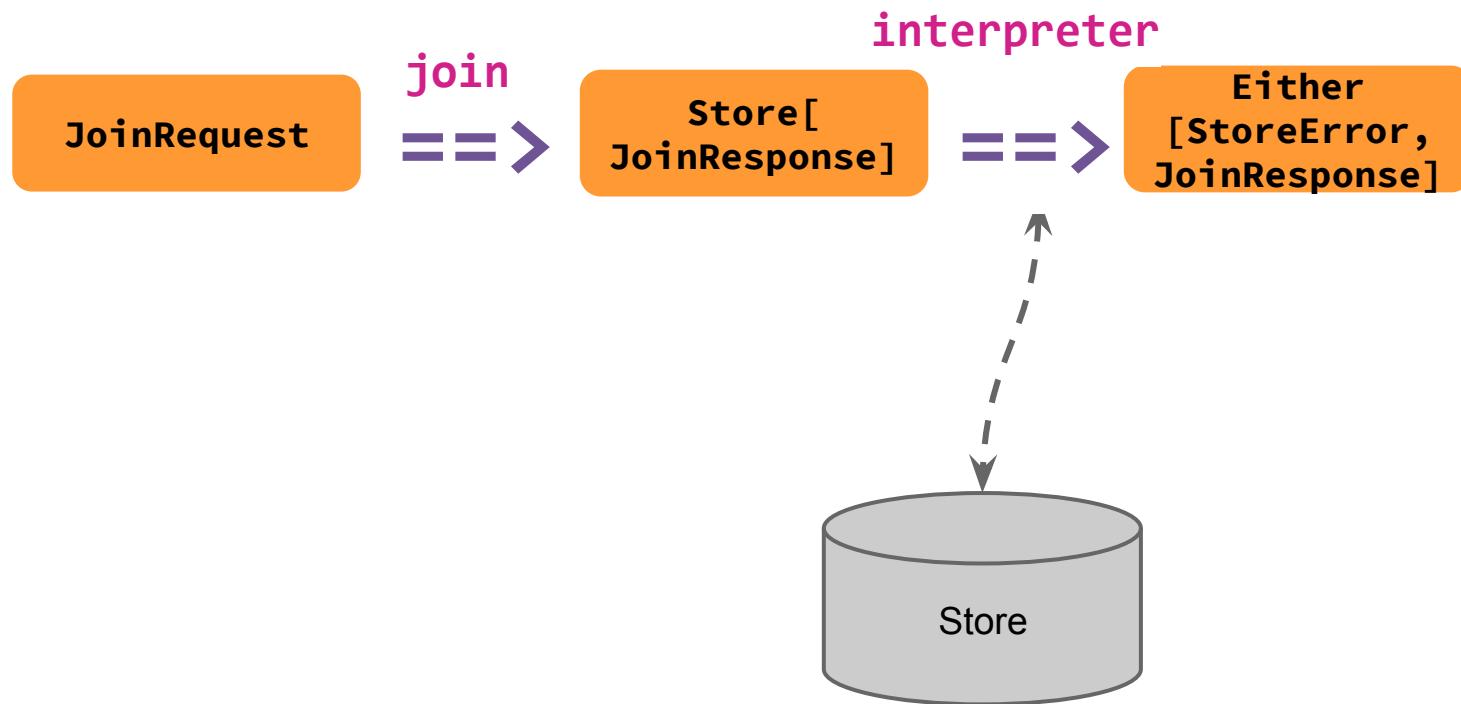


```
def getGroup(gid: Int): Either[StoreError, Group] =  
  DB.withSession { implicit session =>  
    val maybeGroup = (for {  
      group <- group_table if group.gid === gid  
    } yield group).firstOption  
    maybeGroup.toRight(NonExistentEntity(gid))  
  }
```

¿Cómo eliminar el acceso a BBDD?



Mediante un nuevo tipo **Store[T]**



```
sealed trait Store[+U]
```

```
case class GetGroup[U](  
    id: Int, next: Group => Store[U]) extends Store[U]
```

```
case class IsMember[U](  
    uid: Int, gid: Int, next: Boolean => Store[U]) extends Store[U]
```

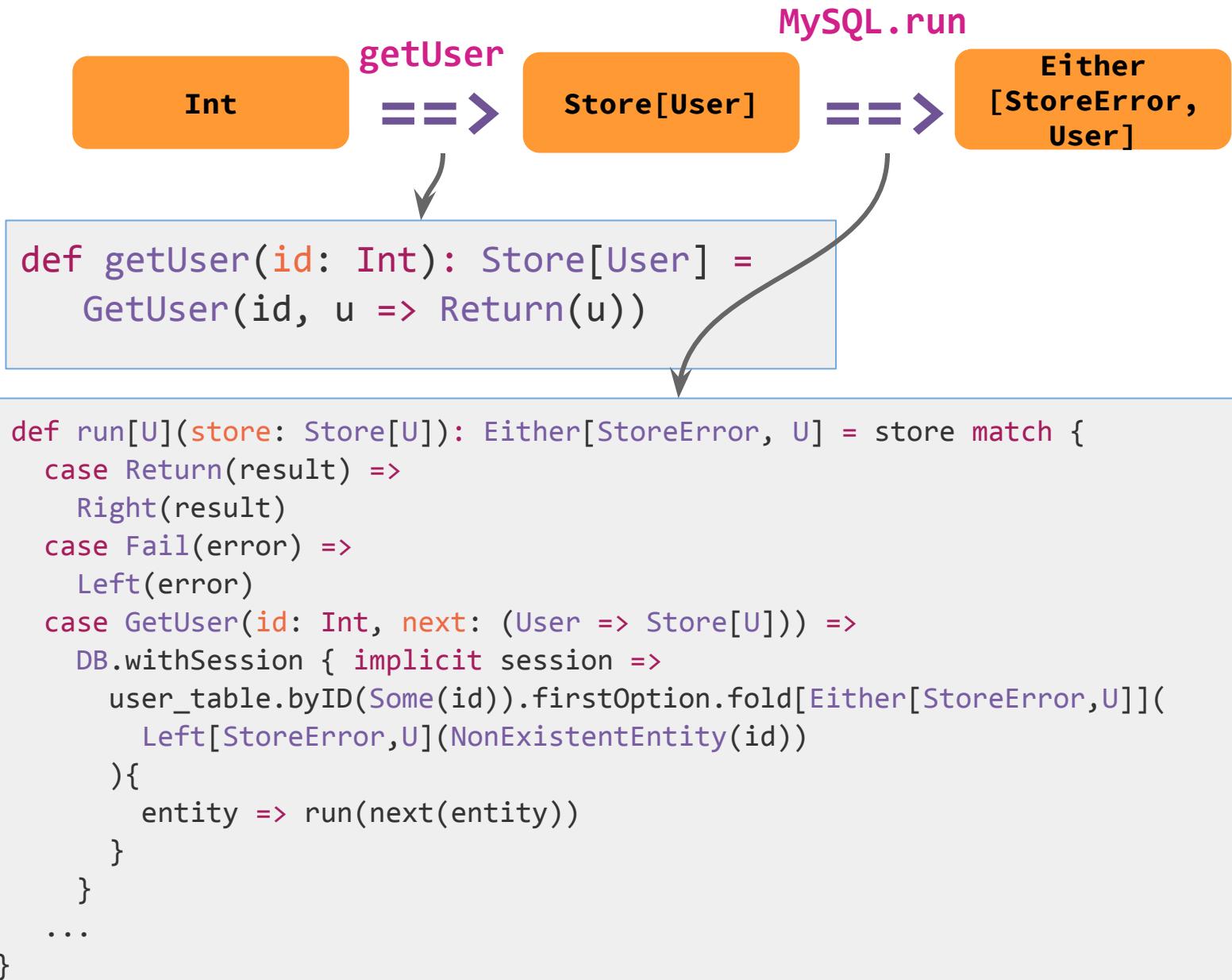
```
case class PutJoin[U](  
    join: JoinRequest, next: JoinRequest => Store[U]) extends Store[U]
```

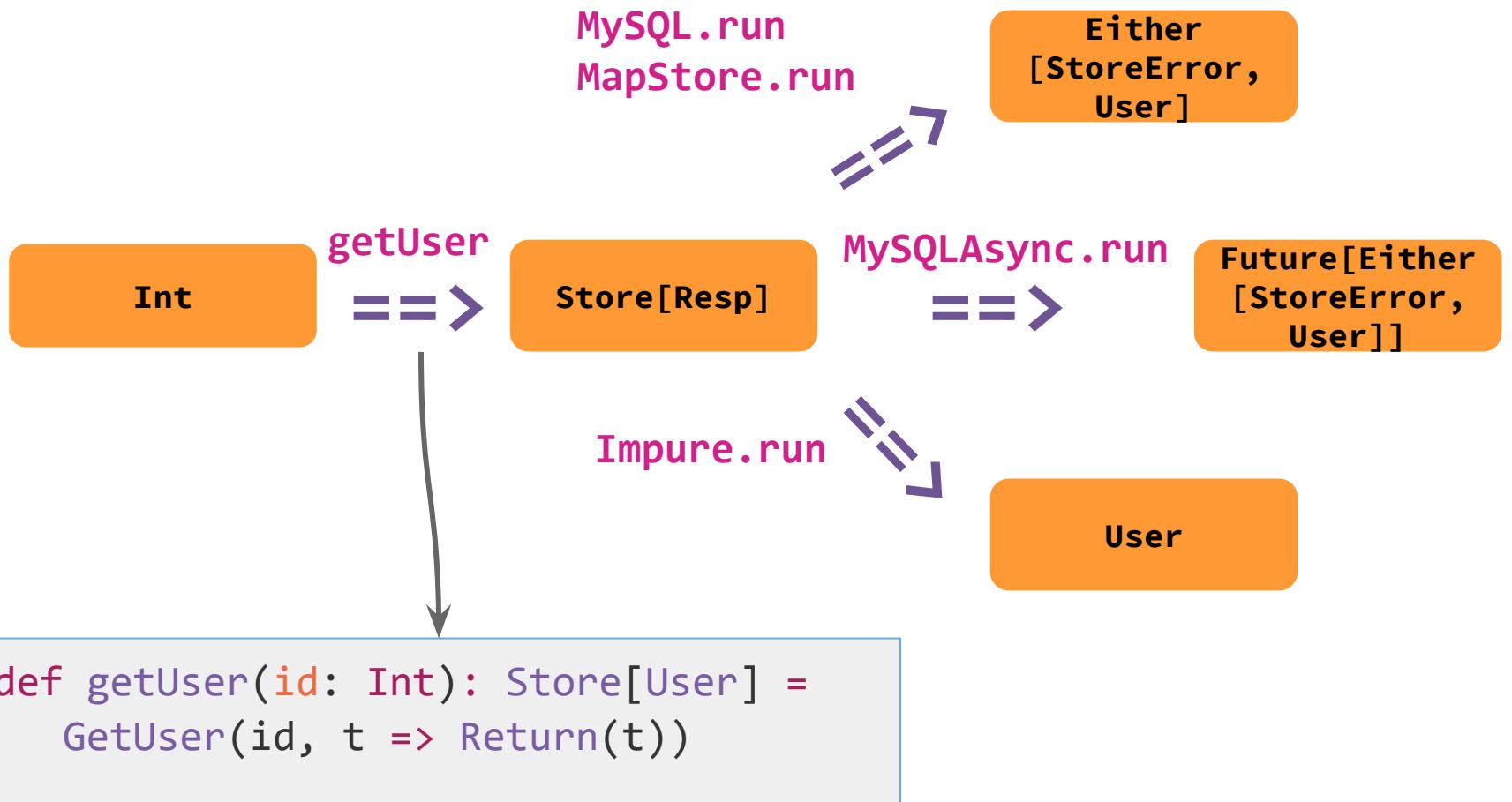
```
...
```

```
case class Return[U](t: U) extends Store[U]
```

```
case class Fail(error: StoreError) extends Store[Nothing]
```







```
def join(request: JoinRequest): Store[JoinResponse] = {
  val JoinRequest(_, uid, gid) = request
  GetUser(uid, user =>
    GetGroup(gid, group =>
      IsMember(uid, gid, isMember =>
        if (isMember)
          Fail(GenericError("..."))
        else
          if (!group.must_approve)
            PutMember(Member(None, uid, gid), member =>
              Return(Right(member)))
          )
        else
          ...
      )
    )
  )
}
```

Secuenciación de efectos

```
def isMember(uid: Int, gid: Int): Store[Boolean] =  
  IsMember(uid, gid, isMember => Return(isMember))  
  
def putIfNotMember(isMember: Boolean): Store[Member] =  
  if (!isMember)  
    putMember(Member(None, uid, gid))  
  else  
    Fail(GenericError(..))  
  
def checkAndPutIfNotMember(uid: Int, gid: Int): Store[Member] =  
  IsMember(uid, gid, isMember =>  
    if (!isMember)  
      putMember(Member(None, uid, gid))  
    else  
      Fail(GenericError(..))  
)
```

Secuenciación de efectos

```
abstract class Store[U]{  
    def flatMap[V](f: U => Store[V]): Store[V]  
}  
  
def checkAndPutIfNotMember(uid: Int, gid: Int): Store[Member] =  
    isMember(uid, gid) flatMap putIfNotMember
```

Índice

- ¿Qué son los efectos?
 - Los *efectos de lado*
 - Efectos de lado y modularidad
- ¿Cómo se programan funcionalmente?
 - Instrucciones monádicas
 - Librerías de operadores
 - Azúcar sintáctico
- Conclusión

¿Es posible hacerlo mejor?

```
def joinWithMonad(request: JoinRequest): Store[JoinResponse]] = {  
    val JoinRequest(_, uid, gid) = request  
    Store.getUser(uid).flatMap( user =>  
        Store.getGroup(gid).flatMap( group =>  
            Store.isMember(uid, gid).flatMap( isMember =>  
                if (isMember)  
                    Fail(GenericError("..."))  
                else  
                    if (!group.must_approve)  
                        Store.putMember(Member(None, uid, gid)).map( member =>  
                            Right(member)  
                        )  
                    else  
                        ...  
            )  
        )  
    )  
}
```



Los **lenguajes específicos de dominio (DSLs)** permiten representar computaciones con un propósito específico. Hay dos aspectos básicos en el diseño de DSLs: la semántica (las instrucciones fundamentales y los operadores de composición), y la sintaxis.

Los **lenguajes específicos de dominio embebidos (EDSLs)** son DSLs implementados aprovechando la sintaxis de un lenguaje HOST, en lugar de con una sintaxis propia.

Combinadores (I)

```
Store.isMember(uid, gid).flatMap( isMember =>
  if (isMember)
    Fail(GenericError("..."))
  else
    if (!group.must_approve)
      Store.putMember(Member(None, uid, gid)).map( member =>
        Right(member)
      )
    else
      Store.isPending(uid, gid).flatMap( isPending =>
        if (isPending)
          Fail(GenericError("..."))
        else
          Store.putJoin(request).map( join =>
            Left(join)
          )
      )
    )
  )
```

Combinadores (I)

```
Store.isMember(uid, gid).flatMap( isMember =>
  if (isMember)
    Fail(GenericError("..."))
  else
    if (!group.must_approve)
      Store.putMember(Member(None, uid, gid)).map( member =>
        Right(member)
      )
    else
      Store.isPending(uid, gid).flatMap( isPending =>
        if (isPending)
          Fail(GenericError("..."))
        else
          Store.putJoin(request).map( join =>
            Left(join)
          )
      )
    )
  )
```

Combinadores (I)

```
abstract class Store[U]{  
  def unless(violation: Store[Boolean]): Store[U] =  
    violation.flatMap( violated =>  
      if (violated)  
        Fail(ConstraintFailed(violation))  
      else  
        this  
    )  
}
```

Combinadores (I)

```
def join(request: JoinRequest): Store[JoinResponse] = {
    val JoinRequest(_, uid, gid) = request
    Store.getUser(uid).flatMap( user =>
        (Store.getGroup(gid).flatMap(
            group => (
                if (!group.must_approve)
                    Store.putMember(Member(None, uid, gid)).map( member =>
                        Right(member)
                    )
                else
                    (Store.putJoin(request).map( join =>
                        Left(join)
                    )).unless(Store.isPending(uid, gid))
                ).unless(Store.isMember(uid, gid))
            )
        )
    )
}
```

Combinadores (II)

```
def join(request: JoinRequest): Store[JoinResponse] = {
    val JoinRequest(_, uid, gid) = request
    Store.getUser(uid).flatMap( user =>
        Store.getGroup(gid).flatMap(
            group => (
                if (!group.must_approve)
                    Store.putMember(Member(None, uid, gid)).map( member =>
                        Right(member)
                    )
                else
                    Store.putJoin(request).map( join =>
                        Left(join)
                    ).unless(Store.isPending(uid, gid))
                ).unless(Store.isMember(uid, gid))
            )
        )
    }
}
```

Combinadores (II)

```
object Store{  
  
    def If[U,V](cond: => Boolean)(  
        _then: Store[V],  
        _else: Store[U]): Store[Either[U,V]] =  
        if (cond)  
            _then.map(u => Right(u))  
        else  
            _else.map(v => Left(v))  
  
    }  
}
```

Combinadores (II)

```
def join(request: JoinRequest): Store[JoinResponse] = {
    val JoinRequest(_, uid, gid) = request
    Store.getUser(uid).flatMap( user =>
        Store.getGroup(gid).flatMap(
            group => (
                Store.If(!group.must_approve)(
                    _then = Store.putMember(Member(None, uid, gid)),
                    _else = Store.putJoin(request)
                        unless Store.isPending(uid, gid)
                )
            ).unless(Store.isMember(uid, gid))
        )
    )
}
```

Índice

- ¿Qué son los efectos?
 - Los *efectos de lado*
 - Efectos de lado y modularidad
- ¿Cómo se programan funcionalmente?
 - Instrucciones monádicas
 - Librerías de operadores
 - Azúcar sintáctico
- Conclusión

For-comprehensions

```
def join(request: JoinRequest): Either[StoreError, JoinResponse] = {  
    val JoinRequest(_, uid, gid) = request  
    for {  
        user <- getUser(uid).right  
        group <- getGroup(gid).right  
        result <-  
            if (group.must_approve)  
                (putJoin(request).right map Left.apply).right  
            else  
                (putMember(Member(None, uid, gid)).right map Right.apply).right  
    } yield result  
}
```

For-comprehensions (con V)

```
def join(request: JoinRequest): StoreError \/ JoinResponse = {  
    val JoinRequest(_, uid, gid) = request  
    for {  
        user <- getUser(uid)  
        group <- getGroup(gid)  
        result <-  
            if (group.must_approve)  
                (putJoin(request) map Left.apply)  
            else  
                (putMember(Member(None, uid, gid)) map Right.apply)  
    } yield result  
}
```

```
def join(request: JoinRequest): Store[JoinResponse] = {
    val JoinRequest(_, uid, gid) = request
    for{
        user <- getUser(uid)
        group <- getGroup(gid)
        joinOrMember <-
            If (!group.must_approve) (
                _then = putMember(Member(None, uid, gid)),
                _else = putJoin(request) unless isPending(uid, gid)
            ) unless isMember(uid, gid)
    } yield joinOrMember
}
```

Índice

- ¿Qué son los efectos?
 - Los *efectos de lado*
 - Efectos de lado y modularidad
- ¿Cómo se programan funcionalmente?
 - Instrucciones monádicas
 - Librerías de operadores
 - Azúcar sintáctico
- Conclusión

Conclusiones



PUREZA



_ OR NOT



PRODUCTIVIDAD



COMPLEJIDAD

Conclusión (I): Pureza

- vs. Object-oriented design
 - Patrón de diseño *Interpreter*
- Eliminación de efectos de lado
 - Separación de aspectos óptima
 - Composicionalidad garantizada
 - Testability
 - Máxima reusabilidad

Conclusión (II): Productividad

- Las instrucciones de la mónada son el comienzo
- Después: generalización e identificación de operadores de composición reutilizables
- Syntactic sugar: concisión, legibilidad
- Organización del proceso de desarrollo:
DSL team

Conclusión (III): complejidad

- functors, monads
- aplicativos, typeclasses
- free-monads
- monad transformers, co-products
- kleisli arrows
- natural transformations
- ...
- categorical programming!

Conclusión (III): complejidad

- ¡No reinventes la rueda!
 - SCALAZ
 - github.com/scalaz/scalaz
 - SHAPELESS
 - github.com/milessabin/shapeless