Inyección de dependencias en

a tu manera

ScalaMAD, enero 2018

Quién soy y qué hago

Roberto Serrano

https://github.com/bilki

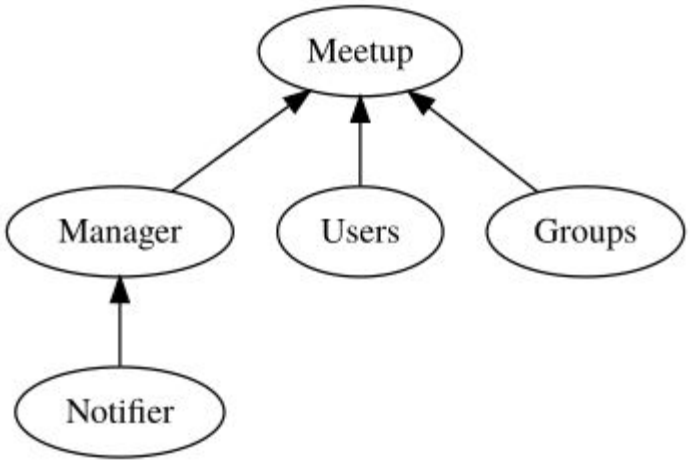
Desarrollador backend
en Blue Indico (Beeva)

Scala + Play

¿Por qué inyección de dependencias?

- Código modular y reutilizable
- Facilidad para testing
- Íntimamente relacionado con el concepto de inversión de control (IoC)

Grafo de dependencias del ejemplo



Aproximación naive

```
class MeetupImpl extends Meetup {

  private val users: Users = new UsersImpl

  private val groups: Groups = new GroupsImpl

  private val manager: Manager = new ManagerImpl

}

class ManagerImpl extends Manager {

  private val notifier: Notifier = new NotifierImpl

}
```

Aproximación naive (testing)

```
val meetup = new MeetupImpl

val joinAttempt = for {
  uid   <- meetup.registerUser(User.Name("Pepe"), User.Age(25))
  gid   <- meetup.registerGroup(Group.Name("ScalaMAD"))
  group <- meetup.joinUserToGroup(uid, gid)    // Notifying users!
} yield {

  group.gid shouldBe Some(gid)

  group.users should have size 1

}
```

Resumen de la aproximación naive

- Código opaco, no modular
- Peligroso e inconveniente para el testing
- Nula flexibilidad a la hora de variar comportamiento

DI manual

```
class MeetupImpl(
  users: Users,
  groups: Groups,
  manager: Manager
) extends Meetup { }

class ManagerImpl(
  notifier: Notifier
) extends Manager { }
```

DI manual (testing)

```
val stubbedNotifier = new Notifier {} // Stub

val manager = new ManagerImpl(stubbedNotifier)

val users = new UsersImpl
val groups = new GroupsImpl

val meetup = new MeetupImpl(users, groups, manager)

// Some initialization code

meetup.joinUserToGroup(uid, gid)  // No user notified in real life
```

Resumen de la DI manual

Pros

- Total flexibilidad a la hora de construir el grafo de dependencias
- Seguridad en el cableado, todo se verifica en tiempo de compilación
- Sencillo de utilizar, ningún requisito de librerías externas

Cons

- El cableado se vuelve tedioso al aumentar el número de dependencias y la complejidad del grafo
- Orden de inicialización estricto (resoluble mediante *lazy val*)

Evitando los mocks

```
class IdentityClient(jsonClient: JsonClient)  {

 def fetchIdentity(accessToken: String) : Future[Option[Identity]]  = {
   jsonClient.getWithoutSession(
     Path("identities"),
     Params("access_token" -> accessToken)
   ).map { case JsonResponse(OkStatus, json, _, _)  =>
      Some(Identity.from(json))  case _ => None
   }
 }
}
```

Evitando los mocks

```
object IdentityClient  {
  def fetchIdentity(
    accessTokenInfo: Future[JsonResponse]
  ) : Future[Option[Identity]]  =
    accessTokenInfo.map {
      case JsonResponse(OkStatus, json)  => Some(Identity.from(json))
      case _ => None
    }
}

object RealAccessTokenService {
  def reallyCheckAccessToken(jsonClient: JsonClient)(
      accessToken:  String): Future[JsonResponse]  =
    jsonClient.getWithoutSession(Path() / "identity",
                                 Params( "access_token"  -> accessToken))
}
```

Guice

```
object MeetupModule extends AbstractModule {

 override def configure(): Unit = {

   bind(classOf[Users]).to(classOf[UsersImpl]).asEagerSingleton()
   bind(classOf[Groups]).to(classOf[GroupsImpl]).asEagerSingleton()
   bind(classOf[Notifier]).to(classOf[NotifierImpl]).asEagerSingleton()

   bind(classOf[Manager]).to(classOf[ManagerImpl])

   bind(classOf[Meetup]).to(classOf[MeetupImpl])

 }

}
```

Guice

```
import javax.inject.{Inject, Singleton}

@Singleton
class ManagerImpl @Inject()(
  notifier: Notifier
) extends Manager { }

@Singleton
class MeetupImpl @Inject()(
  users: Users,
  groups: Groups,
  manager: Manager
) extends Meetup { }
```

Guice (testing)

```
object TestModule extends AbstractModule {
  override def configure(): Unit = {
    bind(classOf[Notifier]).toInstance(stubbedNotifier)
  }
}

val guice = Guice.createInjector(
  Modules.`override`(MeetupModule).`with`(TestModule)
)

val meetup = guice.getInstance(classOf[Meetup])
```

Resumen de Guice

Pros

- Las anotaciones nos permiten un control fino, pudiendo nombrar instancias concretas
- Nivel de adopción y experiencia de los programadores
- Automágico

Cons

- Si se nos olvida cablear alguna cosa, estallará en tiempo de ejecución
- Automágico

Cake Pattern

```
trait ManagerComponent {

  def manager: Manager

  trait Manager {

    def addUserToGroup(
      user: User, group: Group
    ): Either[MeetupError, Group]

    def removeUserFromGroup(
      uid: User.Id, group: Group
    ): Either[MeetupError, Group]

  }

}
```

Cake Pattern

```
trait ManagerComponentImpl  extends ManagerComponent  {
  self: NotifierComponent =>

  override val manager: Manager = new ManagerImpl

  class ManagerImpl extends Manager { }

}
```

Cake Pattern

```
trait MeetupImpl extends Meetup {
  self: UsersComponent
    with GroupsComponent
    with ManagerComponent =>

  override def joinUserToGroup(
    uid: User.Id, gid: Group.Id
  ): Either[MeetupError, Group]  = {

   for {
     user           <- users.getUser(uid)
     group          <- groups.getGroup(gid)
     groupWithUser  <- manager.addUserToGroup(user, group)
     _              <- groups.saveGroup(groupWithUser)
   } yield {
     groupWithUser
   }

 }
}
```

Cake Pattern (testing)

```
trait NotifierComponentMockImpl extends NotifierComponent {
  override val notifier: Notifier = (event: GroupEvent, group: Group) => {
    println(s"Not notifying users of group ${group.gid}")
  }
}

val meetup = new MeetupImpl
  with UsersComponentImpl
  with GroupsComponentImpl
  with NotifierComponentMockImpl
  with ManagerComponentImpl
```

Resumen del Cake Pattern

Pros

- Favorece la composición de componentes
- Algunas variedades “thin” lo mejoran

Cons

- Cantidad enorme de boilerplate, tiempos de compilación muy largos
- Orden de inicialización no definido
- Actualmente en desuso en favor de otras técnicas más ligeras

DI mediante implícitos

```
trait Manager {

 def addUserToGroup(user: User, group: Group)
   (implicit notifier: Notifier): Either[MeetupError, Group]

 def removeUserFromGroup(uid: User.Id, group: Group)
   (implicit notifier: Notifier): Either[MeetupError, Group]

}

object ManagerImpl extends Manager {

   def addUserToGroup(user: User, group: Group)
     (implicit notifier: Notifier): Either[MeetupError, Group] = {}

   def removeUserFromGroup(uid: User.Id, group: Group)
     (implicit notifier: Notifier): Either[MeetupError, Group] = { }

}
```

DI mediante implícitos (testing)

```
implicit val mockNotifier = new Notifier {}

implicit val manager = ManagerImpl

implicit val users = new UsersImpl

implicit val groups = new GroupsImpl

val meetup = MeetupImpl

val joinAttempt = for {
     uid   <- meetup.registerUser(User.Name("Pepe"), User.Age(25))
     gid   <- meetup.registerGroup(Group.Name("ScalaMAD"))
     group <- meetup.joinUserToGroup(uid, gid)
   } yield { }

override def joinUserToGroup(uid: User.Id, gid: Group.Id)
  (implicit users: Users, groups: Groups, manager: Manager,
            notifier: Notifier): Either[MeetupError, Group]  = { }
```

DI mediante implícitos (bonus: typeclasses!)

```scala
trait Show[A] {
  def show(s: A): String
}

case class Person(name: String, age: Int)

object Show {
  def apply[S](implicit ev: Show[S]) = ev

  implicit val personShowInstance = new Show[Person] {
   override def show(s: Person) = s"${s.name} is ${s.age} years old"
 }
}

object ShowSyntax {
 implicit class FromShow[S](s: S)(implicit val ev: Show[S]) {
   def show: String = ev.show(s)
 }
}

import Show._
import ShowSyntax._

val person = Person("Paco", 25)

person.show
```

Resumen de la DI mediante implícitos

Pros

- Cableado seguro mediante descubrimiento automático de instancias para las dependencias
- Mucha granularidad
- De nuevo, no es necesario ningún framework o librería externa

Cons

- Cuando la usamos vía métodos, los implícitos contaminan todas las listas de argumentos
- Trazar el origen de una instancia implícita puede ser complicado...
- Si existen ambigüedades perdemos la ventaja de la resolución automática

Inversion of Control is really just a pretentious way to say 'taking an argument'

@runarorama Dead-simple DI

DI funcional

```
val joinUserToGroup :
    (GetUser, GetGroup, SaveGroup, JoinUserToGroup) =>
    (User.Id, Group.Id) =>
    Either[MeetupError, Group] =
  (getUser, getGroup, saveGroup, joinUserToGroup) => (uid, gid) => {

    for {
      user           <- getUser(uid)
      group          <- getGroup(gid)
      groupWithUser  <- joinUserToGroup(user, group)
      _              <- saveGroup(groupWithUser)
    } yield {
      groupWithUser
    }

 }

def joinUserToGroup(getUser: GetUser, getGroup: GetGroup, saveGroup: SaveGroup,
                    joinUserToGroup: JoinUserToGroup)
                   (uid: User.Id, gid: Group.Id): Either[MeetupError, Group]
```

DI funcional

```
// Some types are identical
type GetUser = User.Id => Either[UsersError, User]
type CreateUser = User => Either[MeetupError, User.Id]
type DeleteUser = User.Id => Either[MeetupError, User]
type GetGroup = Group.Id => Either[GroupsError, Group]
type SaveGroup = Group => Either[MeetupError, Group.Id]
type CloseGroup = Group.Id => Either[MeetupError, Group]
type JoinUserToGroup = (User, Group) => Either[MeetupError, Group]
type RemoveUserFromGroup = (User.Id, Group) =>
Either[MeetupError,Group]
```

DI funcional (testing)

```
val users = new UsersImpl
val groups = new GroupsImpl

val joinManager =
FuncManagerImpl.addUserToGroup(stubbedNotifier.notifyManagerEvent)

val userRegister = FuncMeetupImpl.registerUser(users.createUser)
val groupRegister = FuncMeetupImpl.registerGroup(groups.saveGroup)

val joinMeetup = FuncMeetupImpl.joinUserToGroup(
  users.getUser, groups.getGroup, groups.saveGroup, joinManager
)

val joinAttempt = for {
  uid   <- userRegister(User.Name("Pepe"), User.Age(25))
  gid   <- groupRegister(Group.Name("ScalaMAD"))
  group <- joinMeetup(uid, gid)   // No user notified in real life
} yield {}
```

Resumen de la DI funcional

Pros

- No existe ninguna técnica con mayor potencial de composición y reutilización
- Posibilidad de utilizar sintaxis de métodos o de funciones

Cons

- Los tipos deben ser lo suficientemente expresivos para que las funciones no se inyecten equivocadamente
- La sintaxis de currificación es menos legible que la de métodos

Reader/Kleisli

```
trait Meetup[F[_]] {
 def registerUser(name: User.Name, age: User.Age): F[Either[MeetupError, User.Id]]

  def deleteUser(uid: User.Id): F[Either[MeetupError, User]]
}

case class MeetupContext(
   getUser: GetUser, createUser: CreateUser, deleteUser: DeleteUser,
   getGroup: GetGroup, saveGroup: SaveGroup, closeGroup: CloseGroup,
   joinUserToGroup: JoinUserToGroup, removeUserFromGroup: RemoveUserFromGroup
)

type MeetupOperation[S] = Reader[MeetupContext, S]

object MeetupImpl extends Meetup[MeetupOperation] {
  override def registerUser(name: User.Name, age: User.Age) = Reader { ctx =>
    FuncMeetupImpl.registerUser(ctx.createUser)(name, age)
  }
  override def deleteUser(uid: User.Id) = Reader { ctx =>
    FuncMeetupImpl.deleteUser(ctx.deleteUser)(uid)
  }
}
```

Reader/Kleisli (testing)

```
val manager = ManagerImpl
val users = new UsersImpl
val groups = new GroupsImpl
val meetup = MeetupImpl

val context = MeetupContext(users.getUser, users.createUser, users.deleteUser,
                            groups.getGroup, groups.saveGroup, groups.closeGroup,
                            manager.addUserToGroup(_, _)
                              .run(stubbedNotifier.notifyManagerEvent),
                            manager.removeUserFromGroup(_,_)
                              .run(stubbedNotifier.notifyManagerEvent))

val joinAttempt = for {
    uid   <- meetup.registerUser(User.Name("Pepe"), User.Age(25)).withEitherT
    gid   <- meetup.registerGroup(Group.Name("ScalaMAD")).withEitherT
    group <- meetup.joinUserToGroup(uid, gid).withEitherT
  } yield {}

joinAttempt.run(context) shouldBe a[Right[_, _]]
```

Una pequeña revelación...

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

type ReaderT[F[_], A, B] = Kleisli[F, A, B]
```

Resumen de Reader/Kleisli

Pros

- Composición más agradable que usando únicamente funciones, sobre todo cuando se trata de apilar efectos

Cons

- Como cualquier otra técnica funcional, la curva de aprendizaje para aprender la teoría subyacente

¿Entonces, cuál elegir?

- Como siempre, no hay bala de plata, todo depende de nuestras necesidades
- Podemos alcanzar un equilibrio mediante la combinación de técnicas
- **Manual (o MacWire)** + **Reader** es una opción interesante, con algún implícito

Otras opciones

- **MacWire**: macros por constructores anotados
- **Scaldi**: implementada con módulos, e inyectores implícitos
- **Grafter**: reader/kleisli mediante anotaciones
- **Spring**, **Dagger**, etc.

Referencias

- ***Dead-simple DI***, Rúnar Óli Bjarnason
  - http://functionaltalks.org/2013/06/17/runar-oli-bjarnason-dead-simple-dependency-injection/
- ***DI in Scala: guide***, Adam Warski *(MacWire)*
  - http://di-in-scala.github.io/
- ***Testing without mocking in Scala***, Jessica Kerr
  - http://engineering.monsanto.com/2015/07/28/avoiding-mocks/
- ***Scrap your cake pattern***, Jason Arhart
  - http://blog.originate.com/blog/2013/10/21/reader-monad-for-dependency-injection/
- ***Dependency injection without the gymnastics***, Tony Morris
  - http://2017.phillyemergingtech.com/2012/system/presentations/di-without-the-gymnastics.pdf

Código de ejemplo

https://github.com/bilki/scalamad-di/tree/meetup

Preguntas

?