dist✪ge

Modern Staged Dependency Injection for Scala

Modular Functional Programming
with
Context Minimization
through
Garbage Collection

Septimal Mind Ltd

team@7mind.io

The motivation behind DI pattern and DI frameworks

1. Systems we work with may be represented as graphs. Nodes are components (usually instances), edges are references,
2. Graph transformation complexity grows non-linearly with node count (need to add one constructor parameter, have to modify k classes),
3. Graph composition has combinatoric complexity (need to run tests choosing between mock/production repositories and external APIs, have to write four configurations).

We have several possible options to address these problems:

1. Singletons and Factories: partially solve (2), code still tight coupled $\Rightarrow$ expensive tests and refactorings,
2. Service Locator: bit less coupled code, still very expensive,
3. Dependency Injection: less invasive, simplifies isolation but requires more complex machinery.

"DI doesn't compose with FP": Problems

1. Typical DI framework is OOP oriented and does not support advanced concepts required for modern FP (typeclasses, higher-kinded types),
2. Almost all the DI frameworks are working in runtime while many modern FP concepts are compile-time by their nature,
3. Less guarantees: program which compiles correctly can break on wiring in runtime. After a huge delay,
4. Wiring is non-deterministic: Guice can spend several minutes trying to re-instantiate heavy instance multiple times (once per dependency) then fail,
5. Wiring is opaque: it's hard or impossible to introspect the context. E.g. in Guice it's a real pain to close all the instantiated `Closeables`. Adding missing values into the context (config injections) is not trivial as well.

"DI doesn't compose with FP": Notes

1. We have some compile-time DI frameworks or mechanisms (see `MacWire`) allowing us to implement DI as pattern though purely compile-time tools are not convenient when we have to deal with purely runtime entities (like plugins and config values),
2. Graph composition problem is not addressed by any existing tool.

DI implementations are broken...

...so we may build better one, which must:

1. be well-integrated with type system of our target language (higher-kinded types, implicits, typeclasses),
2. allow us to introspect and modify our context on the fly,
3. be able to detect as many as possible problems quickly, better during compilation,
4. give us a way to stop making atomic or conditional contexts.

Staged approach

1. Let's apply *Late Binding*,
2. let's collect our graph information first,
3. then build a DAG representing our context (so-called *Project Network*, let's call it *Plan*),
4. then analyse this graph for errors (missing references, conflicts),
5. then apply additional transformations,
6. then interpret the graph.

This is a cornercase of more generic pattern – PPER (Percept, Plan, Execute, Repeat).

Staged approach: outcome

What we get:

1. Planner is *pure*: it has no side-effects,
2. A plan is a Turing-incomplete program for a simple machine. It will always terminate in known finite time,
3. An interpreter may perform instantiations at runtime or. . . just generate Scala code that will do that when compiled,
4. All the job except for instantiations can be done in compile-time,
5. Interpreter is free to run independent instantiations in parallel,
6. Extremely important: we can transform (rewrite) the plan before we run iterpreter.

Compile-Time and Runtime DI

A Plan:

```
myRepository := create[MyRepository]()
myservice    := create[MyService](myRepository)
```

May be interpreted as:

Code tree (compile-time):

```scala
val myRepository =
    new MyRepository()
val myservice =
    new MyService(myRepository)
```

Set of instances (runtime):

```scala
plan.foldLeft(Context.empty) {
case (ctx, op) =>
    ctx.withInstance(
        op.key
        , interpret(action)
    )
}
```

Incomplete plans

This code:

```scala
class UsersRepoImpl(cassandraCluster: Cluster)
    extends UsersRepo
class UsersService(repository: UsersRepo)

class UsersModule extends ModuleDef {
  make[UsersRepo].from[UsersRepoImpl]
  make[UsersService]
}
```

May produce a plan like:

```
cassandraCluster          := import[Cluster]
usersRepo: UsersRepo      := create[UsersRepoImpl](cassandraCluster)
usersService              := create[UsersService](usersRepo)
```

distage

Patterns and Extensions

Pattern: Plan completion

Once we have such a plan:

```
cassandraCluster      := import[Cluster]
usersRepo: UsersRepo  := create[UsersRepoImpl](cassandraCluster)
usersService          := create[UsersService](usersRepo)
```

We may add missing values[1]:

```
val plan = Injector.plan(definitions)
val resolved = plan.map {
  case i: Import if i.is[Cluster] =>
    val cluster: Cluster = ???
    Reference(cluster)
  case op => op
}
```

[1] Pseudocode, real API is bit different

Extension: Configuration Support

distage has HOCON configuration support implemented as an extension.

```
case class HostPort(host: String, port: Int)

class HttpServer(@ConfPath("http.listen") listenOn: HostPort) {
  // ...
}
```

The extension:

1. Takes all the Imports of a Plan,
2. Searches them for a specific `@ConfPath` annotation,
3. Tries to find corresponding sections in config,
4. Extends plan with config values,

All the config values are resolved even before instantiation of services ⇒ problems are being shown quickly and all at once.

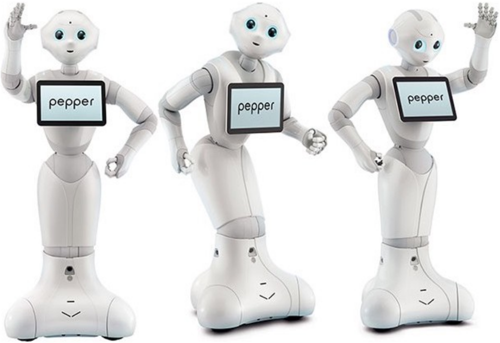
Extension: Automatic Sets

1. distage can find all instances type `T` (like `AutoCloseable`) in the context, put them all into a `Set[T]` then inject that set.
2. $\Rightarrow$ basic lifecycle support, free of charge.

```
trait Resource {
  def start(): Unit
  def stop(): Unit
}
trait App { def main(): Unit }
locator.run { (resources: Set[Resource], app: App) =>
  try {
    resources.foreach(_.start())
    app.main()
  } finally {
    resources.foreach(_.close())
  }
}
```

The Principle Behind: Feedback Loops

They use so-called *Feedback Loops* in robotics...



The Principle Behind: PPER Loop[1]

We found a very generic and important pattern of *Feedback Loop* class:

1. Acquire data from the outer world (*Percept*)
2. Produce a Project Network, *Plan*. It may be incomplete, but should allow us to progress (*Plan*)
   - Plan is a DAG, actions are nodes, edges are dependencies
3. Execute the Plan (*Execute*).
   - Perform the steps of the Plan
   - Mark your Plan nodes according to the results of their execution
   - Let's call marked plan as *Trace*
4. Go to step 1 unless termination criteria reached (*Repeat*)

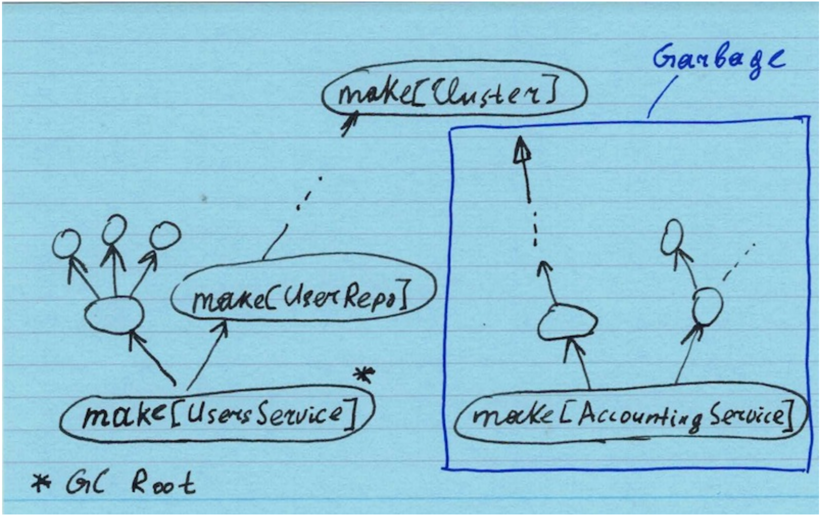
Let's call it *PPER* (pronounced: *pepper*).

[1] Slides: https://goo.gl/okZ8Bw

Garbage Collector and Context Minimization

1. Let's assume that we have a `UsersService` and `AccountingService` in your context,
2. ...and we want to write a test for `UsersService` only,
3. We may exploit staged design and *collect the garbage* out of Plan before executing it.
4. We define a *garbage collection root*, `UsersService`, and keep only the operations it transitively depends on. The rest is being thrown out **even before it's being instantiated**,
5. Garbage Collector allows us to compose contexts easier.

Garbage Collector and Context Minimization



Context Minimization for Tests

Context minimization allows us to:

1. Instantiate only the instances which are required for your tests,
2. Save on test startup time (savings may be significant),
3. Save on configuring per-test contexts manually (savings may be substantial).

Context Minimization for Deployment

Context minimization allows us to:

1. Have one image with all our software components (*Roles*[1]),
2. . . . keeping development flows of these components isolated,
3. Decide which components we want to run when we start the image,
4. Have higher computational density
5. *substantially* simplify development flows: we may run full environment with a single command on a low-end machine,
6. Fuse Microservices with Monoliths keeping *all* their benefits.

```
server1# docker run -ti company/product +analytics
server2# docker run -ti company/product +accounting +users
laptop# docker run -ti company/product --run-everything
```

[1] Slides: https://goo.gl/iaMt43

distage

Scala Typesystem Integration: Fusional Programming

Kind-Polymorphic Type Tags

- distage uses types as lookup keys
- But scalac has TypeTags only for kind *!
- distage added any-kinded 'Tag's
- Modules can be polymorphic on types of any kind

```scala
def m1[F[_]: TagK: Monad, A: Encoder]
  = new ModuleDef {
      addImplicit[Encoder[A]]
      make[SendJson[F, A]]
    }
// `Tag` implicit accepts
// type parameters of any kind
def meta[FT[_[_], _]: Tag.auto.T] =
  new ModuleDef {
    make[FT[IO, Int]]
  }
```

```scala
type TagK[F[_]] = HKTag[{ type Arg[A] = F[A] }]
type TagKK[F[_, _]] = HKTag[{ type Arg[A, B] = F[A, B] }]
```

[1] Tag.auto.T macro will be added in next version

Typeclass instance injection (Implicit Injection)

1. distage wires *all* parameters, inlcuding implicits
2. Leaving implicits solely to implicit resolution would've been *unmodular*, because we want to bind instances *late*, not at definition time
3. Therefore implicits should be declared explicitly
4. Tagless-final style is encouraged – add implicit parameters, instead of importing concrete implicits!

```scala
def polyModule[F[_]: Sync: TagK] = new ModuleDef {
  make[Sync[F]].from(implicitly[Sync[F]])
  make[F[Unit]].named("hello").from(
    S: Sync[F] => S.delay(println("Hello World!")))
}
Injector()
  .produce(polyModule[cats.effect.IO])
  .get[IO[Unit]]("hello").unsafeRunSync()
```

Lambda Injection, ProviderMagnet

1. We can bind functions as constructors
2. Arbitrary-arity functions can run with arguments from Locator
3. Put annotations on types to summon named instances or config values (from `distage-config`)

```scala
case class A()
case class B(a: A)
val ctx: Locator = Injector(config).produce(new ModuleDef {
  make[A].from(() => A())
  make[B].from(B(_))
})
case class C(a: A, b: B, host: HostPort)
val c: C = ctx.run {
  (a: A, b: B, host: HostPort @ConfPath("http.listen")) =>
    C(a, b, host)
}
```

Code example: IO Injection

```
trait MonadFilesystem[F[_, _]] {
  def readFile(fileName: String): F[Exception, String]
}

class ZioFilesystemModule extends ModuleDef {
  make[MonadFilesystem[IO]].from(new MonadFilesystem[IO] {
    def readFile(fileName: String) =
      IO.syncException(Source.fromFile(fileName)("UTF-8")
        .mkString)
  })
}

class GetCpusModule extends ModuleDef {
  make[IO[Exception, Int]].named("getCpus").from {
    fs: MonadFilesystem[IO] =>
      fs.readFile("/proc/cpuinfo").flatMap(...)
  }
}
```

Code example: Tagless Final Style

```scala
trait Interaction[F[_]] {
  def say(msg: String): F[Unit]
  def ask(prompt: String): F[String]
}

class TaglessHelloWorld[F[_]: Monad: Interaction] {
  def program = for {
      userName <- Interaction[F].ask("What's your name?")
      _        <- Interaction[F].say(s"Hello $userName!")
  } yield ()
}

val wiring = new ModuleDef {
  make[TaglessHelloWorld[Try]]
  make[Interaction[Try]].from(new Interaction[Try] {...})
}
```

Path-dependent types

1. Path-dependent types and projections are supported
2. Path-prefix should be wired for the child type to be wired

```scala
trait Cake { trait Child }

val cakeModule = new ModuleDef {
  make[Cake]
  make[Cake#Child]
}
val cake = Injector().produce(cakeModule).get[Cake]

val instanceCakeModule = new ModuleDef {
  make[cake.type].from[cake.type](cake: cake.type)
  make[cake.Child]
}
val cakeChild = Injector().produce(instanceCakeModule)
  .get[cake.Child]
```

distage

Features to boost productivity

Dynamic Plugins

Just drop your modules into your classpath:

```
class AccountingModule extends PluginDef {
  make[AccountingService].from[AccountingServiceImpl]
  // ...
}
```

Then you may pick up all the modules and build your context:

```
val plugins = new PluginLoaderDefaultImpl(
  PluginConfig(Seq("com.company.plugins"))
).load()
// ... pass to an Injector
```

1. Useful while you are prototyping your app,
2. In maintenance phase you may switch to static configuration.

Dynamic Testkit (ScalaTest)

```scala
class AccountingServiceTest extends DistagePluginSpec {
  "accounting service" must {
    "be resolved dynamically" in di {
      (acc: AccountingService) =>
        assert(acc.getBalance("bad-user") == 0)
    }
  }
}
```

1. You don't need to setup your context, it's done automatically by Plugin Loader and Garbage Collector,
2. And it takes just milliseconds, not like in Spring,
3. Garbage collection roots are inferred from test's signature,
4. Only the things required for a particular test are being instantiated.

Tags

1. Each binding may be marked with a *tag*,
2. Some bindings may be excluded from the context before planning by a predicate,
3. This is unsafe but convenient way to reconfigure contexts conditionally.

```scala
class ProductionPlugin extends PluginDef {
  tag("production", "cassandra")
  make[UserRepo].from[CassandraUserRepo]
}
class MockPlugin extends PluginDef {
  tag("test", "mock")
  make[UserRepo].from[MockUserRepo]
}
// ...
val disabledTags = t"mock" && t"dummy"
val plan = injector.plan(definition.filter(disabledTags))
```

Circular dependencies

1. Supported, `Proxy` concept used,
2. By-name parameters (`class C(param: => P)`) supported, no runtime code-generation requried,
3. Other cases are supported as well with runtime code-generation,
4. Limitations: typical. You cannot use an injected parameter immediately in a constructor, you cannot have circular non-by-name dependencies with final classes,
5. Circular dependency resolution is optional, you may turn it off.

Plan Introspection: example context

```scala
class Cluster
trait UsersService
trait AccountingService
trait UserRepo
trait AccountsRepo

class UserRepoImpl(cluster: Cluster) extends UserRepo
class AccountsRepoImpl(cluster: Cluster) extends AccountsRepo
class UserServiceImpl(userRepo: UserRepo) extends UsersService
class AccountingServiceImpl(accountsRepo: AccountsRepo)
    extends AccountingService

class UsersApiImpl(service: UsersService
    , accountsApi: AccountsApiImpl)
class AccountsApiImpl(service: AccountingService
    , usersApi: UsersApiImpl) // circular dependency
class App(uapi: UsersApiImpl, aapi: AccountsApiImpl)
```

Plan Introspection: example bindings[1]

```scala
val definition = new ModuleDef {
    make[Cluster]
    make[UserRepo].from[UserRepoImpl]
    make[AccountsRepo].from[AccountsRepoImpl]
    make[UsersService].from[UserServiceImpl]
    make[AccountingService].from[AccountingServiceImpl]
    make[UsersApiImpl]
    make[AccountsApiImpl]
    make[App]
}
val injector = Injector()
val plan = injector.plan(definition)
```

[1] Full code example: https://goo.gl/7ZwHfX

Plan Introspection: plan dumps

```
println(plan.render) // look for the circular dependency!
```

```
Cluster (BasicTest.scala:353) := make[Cluster] ()
UserRepo (BasicTest.scala:354) := make[UserRepoImpl] (
  par cluster: Cluster = lookup(Cluster)
)
AccountsRepo (BasicTest.scala:355) := make[AccountsRepoImpl] (
  par cluster: Cluster = lookup(Cluster)
)
AccountingService (BasicTest.scala:357) := make[AccountingServiceImpl] (
  par accountsRepo: AccountsRepo = lookup(AccountsRepo)
)
UsersService (BasicTest.scala:356) := make[UserServiceImpl] (
  par userRepo: UserRepo = lookup(UserRepo)
)
AccountsApiImpl (BasicTest.scala:359) := proxy(UsersApiImpl) {
  AccountsApiImpl (BasicTest.scala:359) := make[AccountsApiImpl] (
    par service: AccountingService = lookup(AccountingService)
    par usersApi: UsersApiImpl = lookup(UsersApiImpl)
  )
}
UsersApiImpl (BasicTest.scala:358) := make[UsersApiImpl] (
  par service: UsersService = lookup(UsersService)
  par accountsApi: AccountsApiImpl = lookup(AccountsApiImpl)
)
App (BasicTest.scala:360) := make[App] (
  par uapi: UsersApiImpl = lookup(UsersApiImpl)
  par aapi: AccountsApiImpl = lookup(AccountsApiImpl)
)
AccountsApiImpl (BasicTest.scala:359) -> init(UsersApiImpl)
```

Plan Introspection: dependency trees

You may explore dependencies of a component:

```scala
val dependencies = plan.topology.dependencies
println(dependencies.tree(DIKey.get[AccountsApiImpl]))
```

```
➤ com.github.pshirshov.izumi.distage.fixtures.BasicCases.AnimalModel.AccountsApiImpl
  ↳ 1: com.github.pshirshov.izumi.distage.fixtures.BasicCases.AnimalModel.AccountingService
    ↳ 2: com.github.pshirshov.izumi.distage.fixtures.BasicCases.AnimalModel.AccountsRepo
      ↳ 3: com.github.pshirshov.izumi.distage.fixtures.BasicCases.AnimalModel.Cluster
  ↳ 1: com.github.pshirshov.izumi.distage.fixtures.BasicCases.AnimalModel.UsersApiImpl
    ↳ 2: com.github.pshirshov.izumi.distage.fixtures.BasicCases.AnimalModel.UsersService
      ↳ 3: com.github.pshirshov.izumi.distage.fixtures.BasicCases.AnimalModel.UserRepo
        ↳ 4: com.github.pshirshov.izumi.distage.fixtures.BasicCases.AnimalModel.Cluster
    ⥀ 2: com.github.pshirshov.izumi.distage.fixtures.BasicCases.AnimalModel.AccountsApiImpl
```

Circular dependencies are specifically marked.

Trait Completion

```scala
trait UsersService {
  protected def repository: UsersRepo
  def add(user: User): Unit = {
    repository.put(user.id, user)
    ???
  }
}
```

We may bind this trait directly, without an implementation class:

```scala
make[UsersService]
```

1. Corresponding class will be generated by `distage`,
2. Null-arg abstract methods will be wired with context values,
3. Works in both runtime and compile-time.

Factory Methods (Assisted Injection)

```scala
class UserActor(sessionId: UUID, sessionRepo: SessionRepo)

trait ActorFactory {
  def createActor(sessionId: UUID): UserActor
}
```

1. `createActor` is a factory method,
2. `createActor` will be generated by `distage`,
3. non-null-arg abstract methods are treated as factory methods,
4. Non-invasive *assisted injection*: `sessionId: UUID` will be taken from method parameter, `sessionRepo: SessionRepo` will be wired from context,
5. Useful for `Akka`, lot more convenient than Guice,
6. Works in both runtime and compile-time.

distage

7mind Stack

distage: status and things to do

`distage` is:

1. ready to use,
2. in real production,
3. all Runtime APIs are available,
4. Compile-time verification, trait completion, assisted injections and lambda injections are available.

Our plans:

1. Refactor Roles API,
2. Support running Producer within a monad (to use with `Scalaz ZIO`, `Monix`, `cats-effect`, etc),
3. Support Scala.js,
4. Support optional isolated classloaders (in foreseeable future),
5. Publish compile-time Producer,
6. Check our GitHub: https://github.com/pshirshov/izumi-r2.

distage is just a part of our stack

We have a vision backed by our tools:

1. Idealingua: transport and codec agnostic gRPC alternative with rich modeling language,
2. LogStage: zero-cost logging framework,
3. *Fusional Programming and Design* guidelines. We love both FP and OOP,
4. *Continous Delivery* guidelines for Role-based process,
5. *Percept-Plan-Execute* Generative Programming approach, abstract machine and computational model. Addresses Project Planning (see Operations Research). Examples: orchestration, build systems.

Altogether these things already allowed us to significantly reduce development costs and delivery time for our client.

More slides to follow.

You use Guice?
Switch to distage!

Teaser: LogStage

A log call . . .

```scala
log.info(s"$user logged in with $sessionId!")
```

. . . may be rendered as a text like `17:05:18 UserService.login user=John Doe logged in with sessionId=DEADBEEF!`
. . . or a structured JSON:

```json
{
  "user": "John Doe",
  "sessionId": "DEADBEEF",
  "_template": "$user logged in with $sessionId!",
  "_location": "UserService.scala:265",
  "_context": "UserService.login",
}
```

Teaser: Idealingua

```
id UserId { uid: str }
data User {  name: str /* ... more fields */ }
data PublicUser {
 + InternalUser
 - SecurityAttributes
}
adt Failure = NotFound | UnknownFailure
service Service {
  def getUser(id: UserId): User !! Failure
}
```

1. Convenient Data and Interface Definition Language,
2. Extensible, transport-agnostic, abstracted from wire format,
3. JSON + HTTP / WebSocket at the moment,
4. C#, go, Scala, TypeScript at the moment,
5. Better than gRPC/ REST / Swagger/ etc.

Thank you for your attention

https://izumi.7mind.io/

We're looking for clients, contributors, adopters and colleagues ;)

About the author:

1. coding for 18 years, 10 years of hands-on commercial engineering experience,
2. has been leading a cluster orchestration team in Yandex, "the Russian Google",
3. implemented "*Interstellar Spaceship*" – an orchestration solution to manage 50K+ physical machines across 6 datacenters,
4. Owns an Irish R&D company, https://7mind.io,
5. Contact: team@7mind.io,
6. Github: https://github.com/pshirshov