dist✪ge

Modern Staged Dependency Injection for Scala

Modular Functional Programming
with
Context Minimization
through
Garbage Collection

Septimal Mind Ltd

team@7mind.io

DI is outdated and doesn't compose with FP?

Many Scala folks think that:

1. DI is heavy and slow
   - *"tests initialize longer than they execute"*
2. DI is unsafe
   - *"my program may compile but fail on startup after a huge delay"*
3. DI doesn't work for modern FP code
   - *"we cannot inject* `IO[_, _]` *into* `Repository[_[_, _]]`*"*
4. DI is full of magic and error-prone
   - *"I've read 80% out of 5000-page Spring manual but still don't understand why do I need to put these twelwe annotations here. Also I've tried Guice but it failed with 10-megabytes stack after five minutes and 300 retries of database connection instantiation"*

TLDR

```scala
import distage._, scalaz.zio.IO

trait Repository[F[_, _]] {}
class ProductionRepository[F[_, _]]() extends Repository[F]
class DummyRepository[F[_, _]]() extends Repository[F]
class App[F[_, _]](repository: Repository[F]) { def run = ??? }

class MyAppProd[F[_, _]] extends PluginDef {
  make[Repository[F]].from[ProductionRepository[F]]
  make[App[F]]
}
class Main[F[_, _]] { def main(args: Array[String]): Unit = {
  Injector()
    .produce(MyAppProd[F], roots = Set(DIKey.get[App[F]]))
    .run { app: App[F] => app.run() }
}}
object Main[IO]
```

distage: overview

1. Staged Model: *plans the job first then do*,
2. Garbage Collection: *instantiates reachable instances only*,
3. Higher-Kinded Types support: *injects typeclass instances*,
4. Path-Dependent Types support,
5. Plan introspection: *dumps, graphviz, dependency trees*,
6. Plan rewriting,
7. Roles: *multiple services in one process*,
8. Dynamic Plugins[2] and Testkit,
9. Circular Dependencies[1],
10. Trait Augmentation and Assisted Injection[1],
11. Automatic Sets: *prepopulated sets all the instances of a class*

[1] Both run-time and compile-time support

[2] Run-time with compile-time verification

Garbage Collection for better and faster tests

1. Define all your test and production dependencies as a flat list,
2. Put discrimination tags on test-specific definitions,
3. Only the instances required for your tests will be instantiated,
4. It takes just milliseconds, not like in Spring,
5. ⇒ Significant savings on test startup time.
6. You don't need to setup your context, it's done automatically by Plugin Loader and Garbage Collector,
7. ⇒ Substantial savings on test context configuration.

Example: Garbage Collection and tests

```
import distage._, scalaz.zio.IO

trait Repository[F[_, _]] {}
class ProductionRepository[F[_, _]]() extends Repository[F]
class DummyRepository[F[_, _]]() extends Repository[F]

class MyAppDef[F[_, _]] extends PluginDef {
  make[Repository[F]].from[ProductionRepository[F]]
  make[Repository[F]].tagged("test").from[DummyRepository[F]]
}

object MyApp extends MyApp[IO]

class RepoTest extends DistagePluginSpec { "repository" must {
"do something" in di {
  (repository: Repository[IO]) => // repository is a GC root
    // ProductionRepository will not be instantiated!
} } }
```

Garbage Collection for deployment: flexible monoliths

We may fuse Microservices with Monoliths keeping *all* their benefits:

1. Develop software components (*Roles*[1]) as usual[2],
2. Each Role is a Garbage Collection Root,
3. Build one Docker image with multiple Roles in it,
4. Define Roles you want to start as commandline parameters,
5. ⇒ higher computational density, savings on infrastructure,
6. ⇒ *substantial* development simplification: full environment may be started on a low-end machine with one command,

```
server1# docker run company/product +analytics
server2# docker run company/product +accounting +users
developer1# docker run company/product +*
developer2# docker run company/product --dummy-repositories +*
```

[1] Slides with more details: https://goo.gl/iaMt43
[2] You are not prohibited from using multirepo layout

Config support

distage has HOCON configuration extension.

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

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

The extension:

1. Enumerates all the missing references in a Plan,
2. Searches them for a specific @ConfPath annotation,
3. Tries to find corresponding sections in config source,
4. Extends plan with config values,
5. ⇒ Config values are being resolved before instantiation begins,
6. ⇒ problems are being shown quickly and all at once.

Dynamic Plugins

Just drop your modules into your classpath:

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

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

```scala
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.

Circular dependencies

1. Supported, `Proxy` concept used,
2. Optional: you may turn Circular Dependency support off.
3. By-name parameters (`class C(param: => P)`) supported, without run-time code-generation,
4. Compile-time and run-time code-generation for other cases,

Limitations:

1. You cannot use an injected parameter immediately in a constructor,
2. you cannot have circular non-by-name dependencies with final classes,

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[1] by `distage`,
2. Null-arg abstract methods will be wired with context values,

[1] both runtime and compile-time cogen supported

Assisted Injection (Factory Methods)

```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.

Extension: Automatic Sets

1. All instances of type `T` (like `AutoCloseable`) as a `Set[T]`,
2. Strong and Weak References:
   - GC collects weak referenced members with no more references

Example: free-of-charge basic resource support:

```
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()) }
}
```

How it works: Plans

`distage` takes your bindings and then:

1. translates bindings into simple Turing-incomplete DSL (like `make`, `reference`, etc.),
2. represents the DSL statements as Directed Acyclic Graph using dependecy information and breaking circular dependencies if any,
3. resolves conflicts (one DAG node with several associated operations),
4. performs garbage collection,
5. applies other transformations (like config reference resolution),
6. turns the DAG back into sequential form — a Plan — with topological sorting.
7. $\Rightarrow$ the Plan may be introspected, printed, executed in compile-time by a code generator or executed in run-time.

Plan Introspection: example context

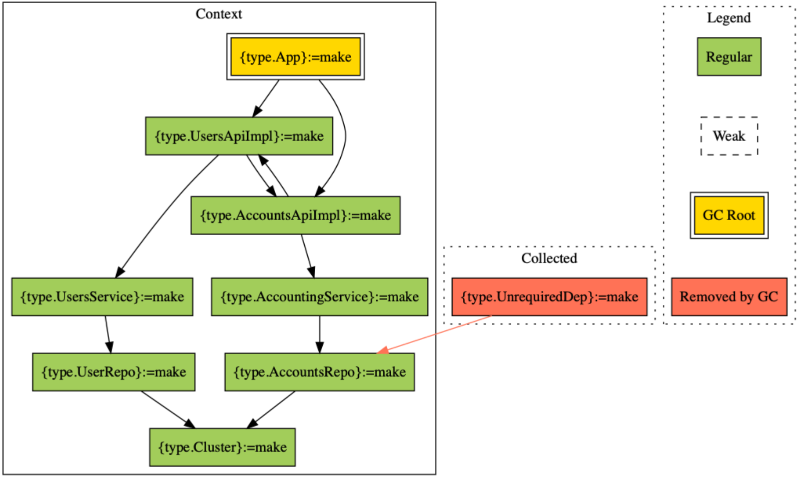
```scala
class Cluster
trait UsersService
trait AccountingService
trait UserRepo
trait AccountsRepo

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: graphviz dumps[1]



[1] This picture has been generated automatically by distage extension

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.

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)
    )
}
```

distage

7mind Stack

distage: status and things to do

distage is:

1. ready to use,
2. in real production,
3. all Run-time features are available,
4. all Compile-time features except of full Producer are available.

Our plans:

1. `ProducerF[_]` — Producer within a monad,
2. New Roles API,
3. Scala.js support,
4. Compile-time Producer,
5. Isolated Classloaders for Roles (in future),
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 . . .

```
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:

```
{
  "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

distage website: 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
7. Download slides: https://github.com/7mind/slides/

Thank you for your attention

distage website: 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
7. Download slides: https://github.com/7mind/slides/