



---

## 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 nodes 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: solves tight coupling but expensive tests and refactorings,
2. Service Locator: bit less coupling but still expensive,
3. Dependency Injection: less invasive and supports 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 of 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 interpreter.

# Compile-Time and Runtime DI

A Plan:

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

May be interpreted as:

Code tree (compile-time):

```
1 | val myRepository =  
2 |   new MyRepository()  
3 | val myservice =  
4 |   new MyService(myRepository)
```

Set of instances (runtime):

```
1 | plan.foldLeft(Context.empty) {  
2 |   case (ctx, op) =>  
3 |     ctx.withInstance(  
4 |       op.key  
5 |       , interpret(action)  
6 |     )  
7 | }
```



## Incomplete plans

This code:

```
1  class UsersRepoImpl(cassandraCluster: Cluster)
2      extends UsersRepo
3  class UsersService(repository: UsersRepo)
4
5  class UsersModule extends ModuleDef {
6      make[UsersRepo].from[UsersRepoImpl]
7      make[UsersService]
8  }
```

May produce a plan like:

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



---

## Patterns and Extensions

---

## Pattern: Plan completion

Once we have such a plan:

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

We may add missing values<sup>1</sup>:

```
1 | val plan = Injector.plan(definitions)
2 | val resolved = plan.map {
3 |   case i: Import if i.is[Cluster] =>
4 |     val cluster: Cluster = ???
5 |     Reference(cluster)
6 |   case op => op
7 | }
```

---

<sup>1</sup>Pseudocode, real API is bit different

## Extension: Configuration Support

distage has HOCON configuration support implemented as an extension.

```
1  case class HostPort(host: String, port: Int)
2
3  class HttpServer(@ConfPath("http.listen") listenOn: HostPort) {
4    // ...
5  }
```

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 the services being instantiated  $\Rightarrow$  problems are being shown quickly and all at once.

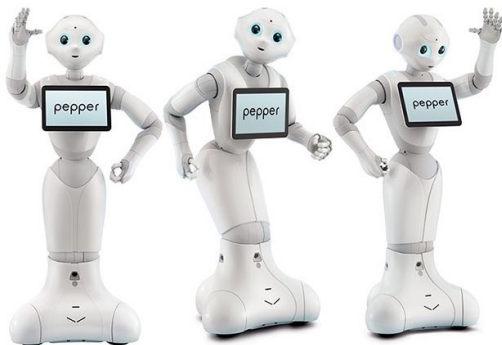
## Extension: Automatic Sets

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.

```
1  trait Resource {  
2      def start(): Unit  
3      def stop(): Unit  
4  }  
5  trait App { def main(): Unit }  
6  locator.run { (resources: Set[Resource], app: App) =>  
7      try {  
8          resources.foreach(_.start())  
9          app.main()  
10         } finally {  
11             resources.foreach(_.close())  
12         }  
13     }
```

# The Principle Behind: Feedback Loops

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



# The Principle Behind: PPER Loop<sup>1</sup>

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* (and pronounce *pepper*).

---

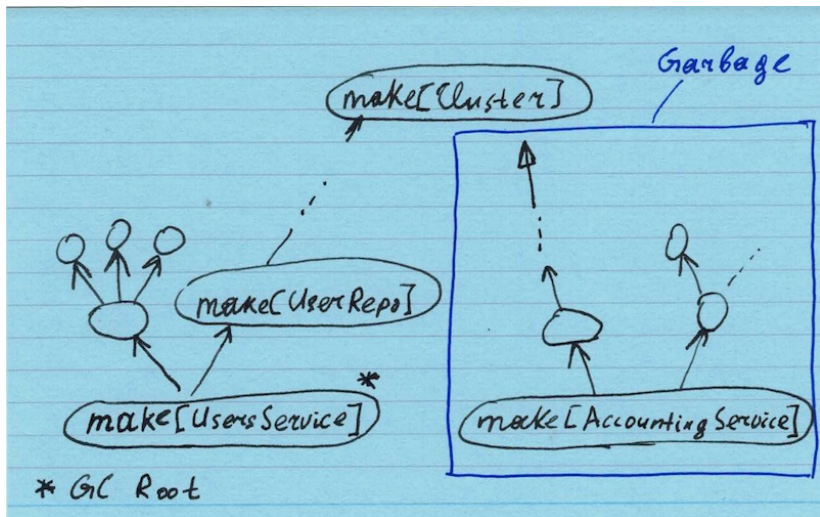
<sup>1</sup>Slides: <https://goo.gl/okZ8Bw>

## Garbage Collector and Context Minimization

1. Let's assume that we have a `UserService` and `AccountingService` in your context,
2. ...and we want to write a test for `UserService` only,
3. We may exploit staged design and *collect the garbage* out of Plan before executing it.
4. We define a *garbage collection root*, `UserService`, 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*<sup>1</sup>),
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.

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

---

<sup>1</sup>Slides: <https://goo.gl/iaMt43>



---

# 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

```

1  def m1[F[_]: TagK: Monad, A: Encoder]
2      = new ModuleDef {
3          addImplicit[Encoder[A]]
4          make[SendJson[F, A]]
5      }
6  // `Tag` implicit accepts
7  // type parameters of any kind
8  def meta[FT[_[_], _]: Tag.auto.T] =
9      new ModuleDef {
10         make[FT[IO, Int]]
11     }

```

```

1  type TagK[F[_]] = HKTag[{ type Arg[A] = F[A] }]
2  type TagKK[F[_], _] = HKTag[{ type Arg[A, B] = F[A, B] }]

```

<sup>1</sup>Tag.auto.T macro will be added in next version

## Typeclass instance injection (Implicit Injection)

1. distage wires *all* parameters, including 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!

```
1  def polyModule[F[_]: Sync: TagK] = new ModuleDef {
2    make[Sync[F]].from(implicitly[Sync[F]])
3    make[F[Unit]].named("hello").from(
4      S: Sync[F] => S.delay(println("Hello World!")))
5  }
6  Injector()
7    .produce(polyModule[cats.effect.IO])
8    .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 distageconfig)

```
1  case class A()
2  case class B(a: A)
3  val ctx: Locator = Injector(config).produce(new ModuleDef {
4      make[A].from(() => A())
5      make[B].from(B(_))
6  })
7  case class C(a: A, b: B, host: HostPort)
8  val c: C = ctx.run {
9      (a: A, b: B, host: HostPort @ConfPath("http.listen")) =>
10         C(a, b, host)
11  }
```

## Code example: IO Injection

```
1  trait MonadFileSystem[F[_], _]] {  
2    def readFile(fileName: String): F[Exception, String]  
3  }  
4  
5  class ZioFileSystemModule extends ModuleDef {  
6    make[MonadFileSystem[IO]].from(new MonadFileSystem[IO] {  
7      def readFile(fileName: String) =  
8        IO.syncException(Source.fromFile(fileName)("UTF-8")  
9          .mkString)  
10    })  
11  }  
12  
13  class GetCpusModule extends ModuleDef {  
14    make[IO[Exception, Int]].named("getCpus").from {  
15      fs: MonadFileSystem[IO] =>  
16        fs.readFile("/proc/cpuinfo").flatMap(...)  
17    }  
18  }
```



## Code example: Tagless Final Style

```
1  trait Interaction[F[_]] {  
2    def say(msg: String): F[Unit]  
3    def ask(prompt: String): F[String]  
4  }  
5  
6  class TaglessHelloWorld[F[_]: Monad: Interaction] {  
7    def program = for {  
8      userName <- Interaction[F].ask("What's your name?")  
9      _ <- Interaction[F].say(s"Hello $userName!")  
10   } yield ()  
11  }  
12  
13  val wiring = new ModuleDef {  
14    make[TaglessHelloWorld[Try]]  
15    make[Interaction[Try]].from(new Interaction[Try] {...})  
16  }
```

## Path-dependent types

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

```
1  trait Cake { trait Child }
2
3  val cakeModule = new ModuleDef {
4      make[Cake]
5      make[Cake#Child]
6  }
7  val cake = Injector().produce(cakeModule).get[Cake]
8
9  val instanceCakeModule = new ModuleDef {
10     make[cake.type].from[cake.type](cake: cake.type)
11     make[cake.Child]
12 }
13 val cakeChild = Injector().produce(instanceCakeModule)
14     .get[cake.Child]
```



---

Features to boost productivity

---

## Dynamic Plugins

Just drop your modules into your classpath:

```
1 class AccountingModule extends PluginDef {  
2   make[AccountingService].from[AccountingServiceImpl]  
3   // ...  
4 }
```

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

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

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

## Dynamic Testkit (ScalaTest)

```
1  class AccountingServiceTest extends DistagePluginSpec {  
2      "accounting service" must {  
3          "be resolved dynamically" in di {  
4              (acc: AccountingService) =>  
5                  assert(acc.getBalance("bad-user") == 0)  
6              }  
7          }  
8      }
```

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.

```
1  class ProductionPlugin extends PluginDef {  
2    tag("production", "cassandra")  
3    make[UserRepo].from[CassandraUserRepo]  
4  }  
5  class MockPlugin extends PluginDef {  
6    tag("test", "mock")  
7    make[UserRepo].from[MockUserRepo]  
8  }  
9  // ...  
10 val disabledTags = t"mock" && t"dummy"  
11 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 required,
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

```
1  class Cluster
2  trait UserService
3  trait AccountingService
4  trait UserRepo
5  trait AccountsRepo
6
7  class UserRepoImpl(cluster: Cluster) extends UserRepo
8  class AccountsRepoImpl(cluster: Cluster) extends AccountsRepo
9  class UserServiceImpl(userRepo: UserRepo) extends UserService
10 class AccountingServiceImpl(accountsRepo: AccountsRepo)
11     extends AccountingService
12
13 class UsersApiImpl(service: UserService
14     , accountsApi: AccountsApiImpl)
15 class AccountsApiImpl(service: AccountingService
16     , usersApi: UsersApiImpl) // circular dependency
17 class App(uapi: UsersApiImpl, aapi: AccountsApiImpl)
```



## Plan Introspection: example bindings<sup>1</sup>

```
1  val definition = new ModuleDef {  
2      make[Cluster]  
3      make[UserRepo].from[UserRepoImpl]  
4      make[AccountsRepo].from[AccountsRepoImpl]  
5      make[UserService].from[UserServiceImpl]  
6      make[AccountingService].from[AccountingServiceImpl]  
7      make[UsersApiImpl]  
8      make[AccountsApiImpl]  
9      make[App]  
10 }  
11 val injector = Injector()  
12 val plan = injector.plan(definition)
```

---

<sup>1</sup>TODO

# Plan Introspection: plan dumps

```
1 | 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)
)
UserService (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: UserService = lookup(UserService)
  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:

```
1 | val dependencies = plan.topology.dependencies
2 | 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

```
1  trait UserService {  
2    protected def repository: UsersRepo  
3    def add(user: User): Unit = {  
4      repository.put(user.id, user)  
5      ???  
6    }  
7  }
```

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

```
1  | make[UserService]
```

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)

```
1  class UserActor(sessionId: UUID, sessionRepo: SessionRepo)
2
3  trait ActorFactory {
4      def createActor(sessionId: UUID): UserActor
5  }
```

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. *Continuous 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 ...

```
1 | 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:

```
1 | {  
2 |   "user": "John Doe",  
3 |   "sessionId": "DEADBEEF",  
4 |   "_template": "$user logged in with $sessionId!",  
5 |   "_location": "UserService.scala:265",  
6 |   "_context": "UserService.login",  
7 | }
```

## Teaser: Idealingua

```
1  id UserId { uid: str }
2  data User {  name: str /* ... more fields */ }
3  data PublicUser {
4    + InternalUser
5    - SecurityAttributes
6  }
7  adt Failure = NotFound | UnknownFailure
8  service Service {
9    def getUser(id: UserId): User !! Failure
10 }
```

1. Convenient Data and Interface Definition Language,
2. Extensible, transportagnostic, abstracted from wire format,
3. ServertoServer, ServertoClient & ClienttoServer communication,
4. Bifunctor Error Model, Strongly typed API calls no longer remove type safety with microservices!

# 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. Contacts: [team@7mind.io](mailto:team@7mind.io),
6. Github: <https://github.com/pshirshov>