



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DI is outdated and doesn't compose with FP?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 start longer than they run"*
2. DI is unsafe
 - ▶ *"my program compiles but crashes at runtime 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% of the 5000-page Spring manual but still don't understand why I need to put these 12 annotations here. I've tried Guice but it failed with 10-megabyte stacktrace after five minutes and 300 retries of database connection initialization"*

DI is outdated and doesn't compose with FP?

"Given its native support for type classes and higher-kinded types — both features indispensable to functional programming — DI Stage is one of the leading dependency injection libraries out there.

*Bonus points for being built by a wicked-smart team that
contributes to ZIO!"*

– John A. De Goes

TLDR

```
1  import distage._, scalaz.zio.IO
2
3  trait Repository[F[_], _]
4  class ProductionRepository[F[_], _] extends Repository[F]
5  class DummyRepository[F[_], _] extends Repository[F]
6  class App[F[_], _](repository: Repository[F]) { def run = ??? }
7
8  class MyAppProd[F[_], _]: TagKK] extends ModuleDef {
9    make[Repository[F]].from[ProductionRepository[F]]
10   make[App[F]]
11 }
12 class Main[F[_], _] { def main(args: Array[String]) =
13   Injector()
14     .produceF[F](new MyAppProd[F], roots=Set(DIKey.get[App[F]]))
15     .use(_.get[App[F]].run)
16 }
17 object Main extends Main[IO]
```

distage: overview

1. Staging: *plan work ahead of time*,
2. Garbage Collection: *instantiate reachable instances only*,
3. Higher-Kinded Types: *use typeclasses & parametricity*,
4. Lifecycle: *inject any `cats.effect.Resource[F, A]`*,
5. Plan introspection: *graphviz, text dump, dependency trees*,
6. Plan rewriting,
7. Roles: *multiple services in one process*,
8. Dynamic Plugins¹ and Testkit,
9. Circular Dependencies support,
10. Auto-Traits and Auto-Factories²,
11. Automatic Sets: *prepopulate sets with all instances of a class*

¹Runtime with compile-time verification

²Runtime or compile-time generation

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. Creation takes 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 setup boilerplate.

Example: Garbage Collection and tests

```
1  class ProductionRepository[F[_], _] extends Repository[F]
2  class DummyRepository[F[_], _] extends Repository[F]
3
4  class MyAppPlugin extends PluginDef {
5    make[Repository[IO]].from[ProductionRepository[IO]]
6    make[Repository[IO]].tagged("test").from[DummyRepository[IO]]
7  }
8  class RepoTest extends DistagePluginSpec {
9    "repository" must {
10      "work correctly" in diIO {
11        (repository: Repository[IO]) => // repository is GC root
12        // Repository is DummyRepository - "test" tag prioritized
13        // ProductionRepository will not be instantiated!
14        for { kv <- randomIO[KeyValue]
15              _ <- repository.put(kv)
16              kv2 <- repository.get(kv.key)
17        } yield assert(kv == kv2)
18      }
19    }
20  }
```

Garbage Collection for deployment: flexible monoliths

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

1. Develop services (*Roles*¹) separately, even in multirepo,
2. Each Role is a Garbage Collection Root,
3. Build a single Docker image with multiple Roles in it,
4. Pass Roles you want to start as commandline parameters,
5. ⇒ higher computation density, savings on infrastructure,
6. ⇒ *substantial* development simplification: full environment can be started on a low-end machine with one command,

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

¹Previous slides on the subject: <https://goo.gl/iaMt43>

Lifecycle

- ▶ An application must manage a lot of global resources:
 1. Connection pools, thread pools
 2. Servers, external endpoints, databases
 3. Configurations, metrics, heartbeats
 4. External log sinks
- ▶ They have to be started and closed in integration tests,
- ▶ We shouldn't set them up manually for every test,
- ▶ We want to create reusable components that correctly share a single resource.

Lifecycle: .fromResource

1. Inject any cats-effect Resource
2. Global resources deallocate when the app or test ends

```
1  object App extends IOApp {  
2    val blazeClientModule = new ModuleDef {  
3      make[ExecutionContext].from(ExecutionContext.global)  
4      addImplicit[Bracket[IO, Throwable]]  
5  
6      make[Client[IO]].fromResource { ec: ExecutionContext =>  
7        BlazeClientBuilder[IO](ec).resource  
8    }}  
9  
10   def run(args: List[String]): IO[ExitCode] =  
11     Injector().produceF[IO](blazeClientModule)  
12       .use { // Client allocated  
13         _.get[Client[IO]].expect[String]("https://google.com")  
14       }.as(ExitCode.Success) // Client closed  
15 }
```

Effectful creation: `.fromEffect`

Global mutable state must be created effectfully, but doesn't have to be deallocated. e.g. a global parallelism limiter:

```
1  import distage._, import scalaz.zio._
2
3  case class UploadConfig(maxParallelUploads: Long)
4
5  class UploaderModule extends ModuleDef {
6    make[Semaphore].named("upload-limit").fromEffect {
7      conf: UploadConfig @ConfPath("myapp.uploads") =>
8        Semaphore.make(conf.maxParallelUploads) }
9    make[Uploader]
10 }
11
12 class Uploader(limit: Semaphore @Id("upload-limit")) {
13   def upload(content: Content): IO[Throwable, Unit] =
14     limit.withPermit(...)
```

Config support

distage has HOCON configuration extension.

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

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. \Rightarrow Config values are parsed before instantiation begins,
6. \Rightarrow Problems are shown quickly and all at once,
7. \Rightarrow Compile-time plugin checker validates config.

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 loaded modules to Injector
```

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

Circular dependencies

1. Supported via Proxies,
2. Cyclic by-name parameters (`class C(param: => P)`) will work without proxies,
3. Circular dependency support can be disabled.

Limitations:

1. You cannot use an injected parameter immediately during `initii`,
2. You cannot use non-by-name circular dependencies with final classes,

Trait Completion

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

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

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

1. Corresponding class will be generated¹ by distage,
2. Abstract defs will be wired with values from the object graph,

¹both runtime and compile-time cogen supported

Assisted Injection (Factory Methods)

```
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. Abstract methods with parameters 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: basic lifecycle support: (please use Resource bindings in real apps!)

```
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 { resources.foreach(_.close()) }  
11 }
```

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 dependency 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 runtime.

Plan Introspection: example context

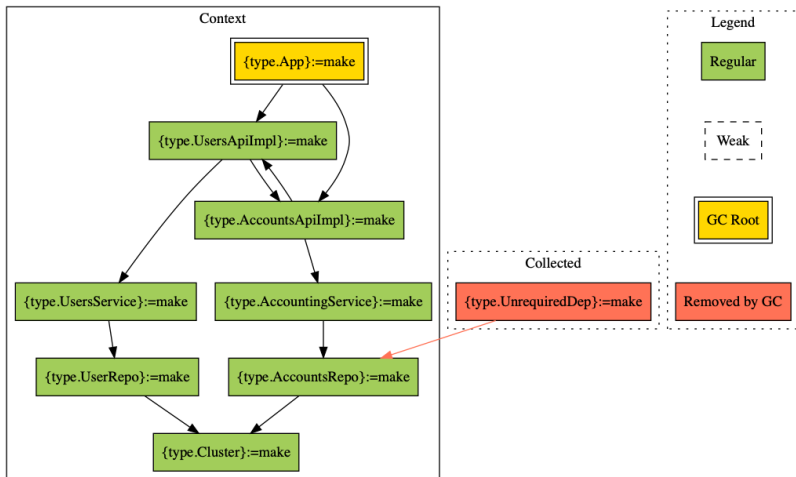
```
1  class Cluster
2  trait UserService
3  trait AccountingService
4  trait UserRepo
5  trait AccountsRepo
6
7  class UserRepoImpl(cluster: Cluster) extends UserRepo
8  class AccountsRepoImpl(cluster: Cluster) extends AccountsRepo
9  class UserServiceImpl(userRepo: UserRepo) extends UserService
10 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 bindings¹

```
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)
```

¹Full code example: <https://goo.gl/7ZwHfX>

Plan Introspection: graphviz dump¹



¹Generated automatically by GraphDumpObserver distage extension

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 marked with a circle symbol.

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 | }
```


distage

7mind Stack

distage: status and things to do

distage 0.7 is:

1. ready to use,
2. in production for over 1 year,
3. all runtime features are available,
4. all compile-time features except for full compile-time mode are available.

What's next:

1. New Roles API,
2. Scala.js support,
3. Compile-time Producer,
4. Isolated Classloaders for Roles (in future),
5. 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: structured 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 simple logging call ...

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

May be rendered as text:

17:05:18 UserService.login user=John Doe logged in
with sessionId=DEADBEEF!

Or as 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, 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. At Yandex, worked on “*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. Slides: <https://github.com/7mind/slides/>