dist✪ge

Modern Staged Dependency Injection for Scala

Modular Functional Programming
with
Context Minimization
through
Garbage Collection

Septimal Mind Ltd
team@7mind.io

7mind

DI is outdated and doesn't compose with FP?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"*

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[_, _]: TagKK] extends ModuleDef {
  make[Repository[F]].from[ProductionRepository[F]]
  make[App[F]]
}
class Main[F[_, _]: TagKK: BIO] extends App {
  Injector()
    .produceF[F[Throwable, ?]](
      new MyAppProd[F], roots=Set(DIKey.get[App[F]])
    ).use(_.get[App[F]].run)
}
object Main extends Main[IO]
```

distage: overview

1. Staging: *plan all the work before you do anything*,
2. Garbage Collection: *instantiate reachable instances only*,
3. Higher-Kinded Types: *inject typeclass instances, use parametricity*,
4. Path-Dependent Types support,
5. Lifecycle and Resources: *inject any cats.effect.Resource[F, A]*,
6. Plan introspection: *graphviz, text dump, dependency trees*,
7. Plan rewriting,
8. Roles: *multiple services in one process*,
9. Dynamic Plugins[1] and Testkit,
10. Circular Dependencies support,
11. Assisted Injection and Trait Augmentation[2],
12. Automatic Sets: *prepopulate sets with all instances of a class*

[1] Runtime with compile-time verification

[2] Runtime 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. $\Rightarrow$ 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. $\Rightarrow$ Substantial savings on test setup boilerplate.

Example: Garbage Collection and tests

```scala
class ProductionRepository[F[_, _]] extends Repository[F]
class DummyRepository[F[_, _]] extends Repository[F]

class MyAppPlugin extends PluginDef {
  make[Repository[IO]].from[ProductionRepository[IO]]
  make[Repository[IO]].tagged("test").from[DummyRepository[IO]]
}
class RepoTest extends DistagePluginSpec {
  "repository" must {
    "work correctly" in diIO {
      (repository: Repository[IO]) => // repository is GC root
      // Repository is DummyRepository - "test" tag prioritized
      // ProductionRepository will not be instantiated!
      for { kv  <- randomIO[KeyValue]
            _   <- repository.put(kv)
            kv2 <- repository.get(kv.key)
      } yield assert(kv == kv2)
}}}
```

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.

```
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] Previous slides on the subject: https://goo.gl/iaMt43

Lifecycle

- Applications 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

```scala
object App extends IOApp {
  val blazeClientModule = new ModuleDef {
    make[ExecutionContext].from(ExecutionContext.global)
    addImplicit[Bracket[IO, Throwable]]

    make[Client[IO]].fromResource { ec: ExecutionContext =>
      BlazeClientBuilder[IO](ec).resource
  }}

  def run(args: List[String]): IO[ExitCode] =
    Injector().produceF[IO](blazeClientModule)
    .use { // Client allocated
      _.get[Client[IO]].expect[String]("https://google.com")
    }.as(ExitCode.Success) // Client closed
}
```

Effectful creation: `.fromEffect`

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

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

case class UploadConfig(maxParallelUploads: Long)

class UploaderModule extends ModuleDef {
  make[Semaphore].named("upload-limit").fromEffect {
    conf: UploadConfig @ConfPath("myapp.uploads") =>
      Semaphore.make(conf.maxParallelUploads) }
  make[Uploader]
}
class Uploader(limit: Semaphore @Id("upload-limit")) {
  def upload(content: Content): IO[Throwable, Unit] =
    limit.withPermit(...)
}
```

Config support

distage has HOCON configuration extension.

```scala
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 parsed before instantiation begins,
6. ⇒ Problems are shown quickly and all at once,
7. ⇒ Compile-time checker plugin validates config.

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 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 run-time code-generation,
3. Circular dependency support can be disabled.

Limitations:

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

Trait Augmentation

```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. Abstract defs will be wired with values from the object graph,

[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. 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!)

```
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 runtime.

Plan Introspection: example context

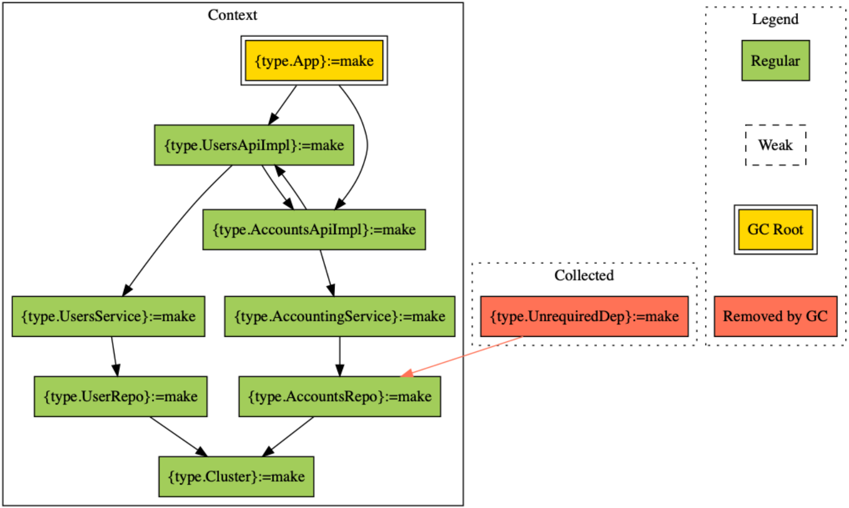
```
class Cluster
trait UsersService
trait AccountingService
trait UserRepo
trait AccountsRepo

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] Generated automatically by `GraphDumpObserver` distage extension

Plan Introspection: plan dumps

```scala
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:

```
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 marked with a circle symbol.

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` 0.7:

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

What's next:

1. New Resource-based 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: zero-cost 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`!

"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

Teaser: LogStage

A simple logging call . . .

```
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:

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

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 19 years, 12 years of hands-on commercial engineering experience,
2. has been leading a cluster orchestration team in Yandex, "the Russian Google",
3. Created "*Yandex 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/