DI IS THE PRODUCTIVITY CORNERSTONE

WHAT IS DEPENDENCY INJECTION?

A magic tool which performs *wiring* automatically

MANUAL VS AUTOMATIC WIRING

```
val client = new HttpClient()
val accounts = new AccountsService()
val feed = new StocksFeed(client)
val app = new App(accounts, feed)
app.run()
```

```
make[App].run()
```

- Wiring may happen in "global static" context (before the application starts)
  - ...and in "local computation" context
- Parameter passing is evil

WORST-CASE SCENARIO: CONFIGURATIONS

```
def makeUserRepo(config: Config): IUserRepo[F] = {
  config.database match {
    case OracleDb =>
      if (!config.isProd) {
        throw new RuntimeException("Oracle unsupported in test mode!")
      } else {
        new ProdOracleUserRepo[F]( /*...*/ )
      }
    case PgDb =>
      new ProdPGUserRepo[F]( /*...*/ )
    case Unset =>
      if (config.isProd) {
        throw new RuntimeException("Database is not set for prod mode!")
      } else {
        new DummyUserRepo[F]()
      }
  }
}
```

DI AS A PATTERN

- You describe a problem in terms of *products*, *ingredients* and *recipes*
- You declare *products*, *ingredients* and *recipes*
- You don't declare individual actions
- You don't order the actions

DI APPLICATIONS

- Application startup process
- Orchestration
  - Application lifecycle
  - Workflows (e.g. builds)
  - OS lifecycle
  - Distributed system lifecycle

DI TOOLS ARE EVERYWHERE

- PL & Runtimes
  - DI Frameworks: Spring, Guice, etc
  - Adhoc DI: implicit resolution
- OS
  - Init systems: systemd
- Build tools
- Distributed systems
  - Declarative deployment tools: Terraform
  - Orchestrators: Kubernetes

WHY DO WE NEED DI?

- Ordering complexity is always worse than linear
- Refactoring => huge rewiring => waste of time

WHAT'S WRONG WITH DI?

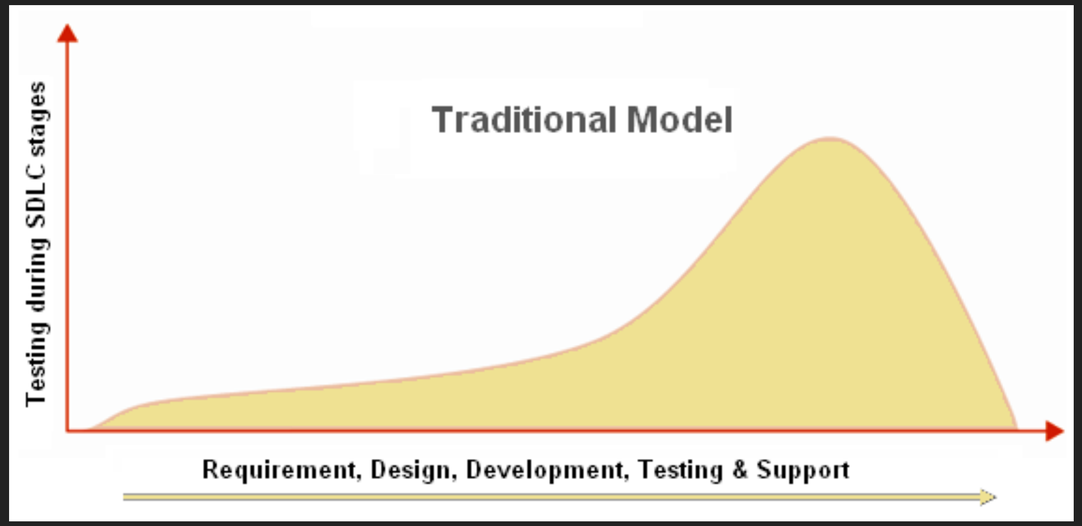
- Spring/Guice: slow, unreliable, too many things happen in runtime
- No DI tools for many applications
  - Or they aren't good enough
- Sometimes we don't even understand that *good* DI is the answer

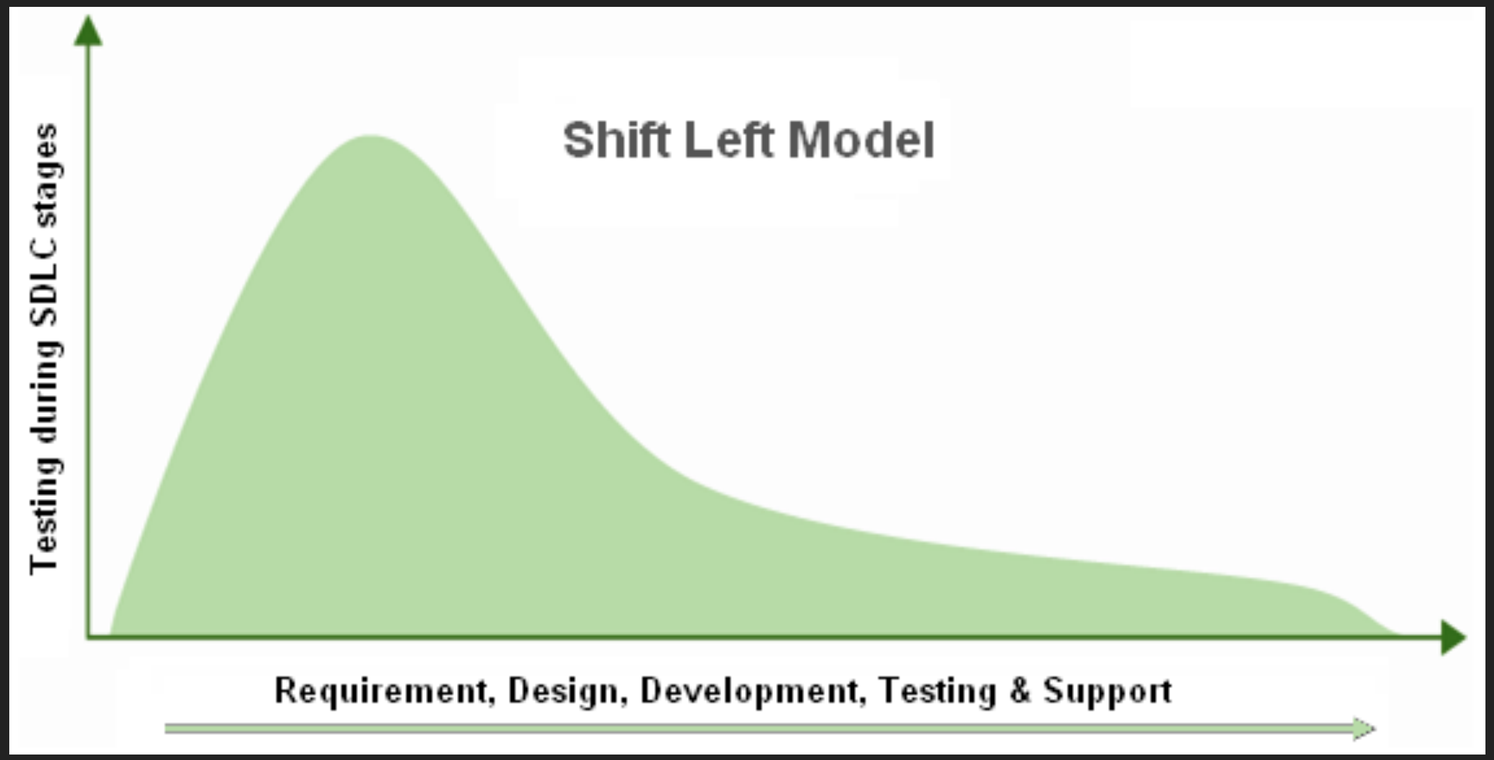
IS READER A DI?

- No
- It removes wirings from your business code
- But you still need to wire your components somehow
- Reader is an implementation of Service Locator

DI automates *Wiring*

Good DI enforces *Shift Left* practice


Traditional Model
Testing during SDLC stages
Requirement, Design, Development, Testing & Support


Shift Left Model
Testing during SDLC stages
Requirement, Design, Development, Testing & Support

DUAL TEST TACTIC: LIGHTNING-FAST INTEGRATION

- Run same tests twice: with real implementations and mocks
  - Skip real tests in case dependencies are unmet
  - Hard to implement without *good* tools
- Helps draft business logic quickly
- Enforces SOLID design
- Does not prevent integration tests
- Speeds up onboarding

ARE MICROSERVICES IN MULTIREPOS GOOD?

- No
- Global refactorings are hard
- Integration is hard
- Distributed systems which may not need to be distributed
- Integration *shifts right*
- Deployment density

Good things: isolation, less interlocking (not really true, integration is hard)

ARE MONOLITHS IN MONOREPOS GOOD?

- No
- Isolation is hard
- Wiring is complicated

Good things: coherence , observability, cheaper/free workflows

WHAT'S THE ALTERNATIVE?

- Role-based applications
  - Configurable monoliths
- All the microservice-oriented flows are supported
- All the monolith-oriented flows are supported
- Same for monorepo/multirepo
- Great simulations (DTT mocks can be reused)

Role-Based Applications
Shift a lot of SDLC problems Left

`distage` provides you these features

with

full compile-time safety

AUTOMATED APPLICATION COMPOSITION: THE MODEL

```
object ServiceDemo {
  trait Repository[F[_, _]]
  class DummyRepository[F[_, _]] extends Repository[F]
  class ProdRepository[F[_, _]] extends Repository[F]

  class Service[F[_, _]](c: Repository[F]) extends
Repository[F]

  class App[F[_, _]](s: Service[F]) {
    def run: F[Throwable, Unit] = ???
  }
}
```

AUTOMATED APPLICATION COMPOSITION: DEFINITIONS

```
def definition[F[_, _]: TagKK] = new ModuleDef {
  make[App[F]]
  make[Service[F]]
  make[Repository[F]].tagged(Repo.Dummy).from[DummyRepository[F]]
  make[Repository[F]].tagged(Repo.Prod).from[ProdRepository[F]]
}
```

AUTOMATED APPLICATION COMPOSITION: STARTING THE APP

```
Injector(Activation(Repo -> Prod))
  .produceRunF(definition[zio.IO]) {
    app: App[zio.IO] =>
      app.run
  }
```

DUAL TEST TACTIC: CODE

```scala
abstract class ServiceTest extends DistageBIOEnvSpecScalatest[ZIO] {
  "our service" should {
    "do something" in {
      service: Service[zio.IO] => for {
        // ...
      } yield ()
    }
  }
}

trait DummyEnv extends DistageBIOEnvSpecScalatest[ZIO] {
  override def config: TestConfig = super.config.copy(
    activation = Activation(Repo -> Dummy))
}

trait ProdEnv extends DistageBIOEnvSpecScalatest[ZIO] {
  override def config: TestConfig = super.config.copy(
    activation = Activation(Repo -> Prod))
}

final class ServiceProdTest extends ServiceTest with ProdEnv
final class ServiceDummyTest extends ServiceTest with DummyEnv
```

MEMOIZATION

```scala
abstract class DistageTestExample extends DistageBIOEnvSpecScalatest[ZIO] {
  override def config: TestConfig = {
    super.config.copy(memoizationRoots = Set(DIKey.get[DbDriver[zio.IO]]))
  }

  "our service" should {
    "do something" in {
      repository: Repository[zio.IO] =>
      //...
    }

    "and do something else" in {
      repository: Repository[zio.IO] =>
      //...
    }
  }
}
```

MEMOIZATION

- DbDriver will be shared across all the tests
  - Without any singletons
  - With graceful shutdown
  - With separate contexts for different configurations

ROLES

```
class UsersRole extends RoleService[zio.Task] {
  override def start(params: RawEntrypointParams, args: Vector[String]): DIResource[zio.Task, Unit] =
    DIResource.make(acquire = {
      // initialisation
    })(release = {
      _ =>
      // shutdown
    })
}

object UsersRole extends RoleDescriptor {
  override final val id = "users"
}
```

```
# java -jar myapp.jar -u repo:prod -u transport:prod :users :accounts

# java -jar myapp.jar -u repo:dummy -u transport:vm :users :accounts
```

INTEGRATION CHECKS

- Check if an external service is available
- Tests will be skipped if checks fail
- The app will not start if checks fail
- Integration checks run before all initialisation

INTEGRATION CHECKS

- Check if an external service is available
- Tests will be skipped if checks fail
- The app will not start if checks fail
- Integration checks run before all initialisation

```
class DbCheck extends IntegrationCheck {
  def resourcesAvailable(): ResourceCheck = {
    if (checkPort()) ResourceCheck.Success()
    else ResourceCheck.ResourceUnavailable("Can't connect to DB", None)
  }
  private def checkPort(): Boolean = ???
}
```

Thank you!

https://github.com/7mind
https://github.com/7mind/distage-example

https://twitter.com/shirshovp

https://twitter.com/kai_nyasha