SEPTIMAL MIND

SCALA, FUNCTIONAL PROGRAMMING AND TEAM PRODUCTIVITY

WHO WE ARE

- Pavel and Kai
- Septimal Mind, an Irish consultancy
- Engineer’s productivity is our primary concern
- Love Scala and pure FP
- Scala contributors

WHAT WE DO

- We identify productivity issues in SDLC pipeline
- We build tools to address these issues

- How do we decide to write a new tool?
  - Prototype should take one engineer at most two weeks

CASE STUDY: STARTING POINT (JAN/2018)

- Millions of users, +100K each month
- About 30 engineers
- Back-end: About 10 microservices, multirepo
- Front-end: mobile apps, web portal
- Scala as Java+ (akka-http, guice), C#, Go, Typescript

CASE STUDY: OUTCOME (JUN/2019)

Development costs cut by 50%

(management estimation)

WHAT WAS WRONG?

Late problem detection

Actual integration postponed to production

WHAT WAS WRONG

- A lot of issues discovered in production
  - Bugs
  - Component incompatibilities
  - Frontend incompatibilities
- Not enough insights to debug after failure
- Hard to just observe the product: 13 repositories
- Hard to get people on board

WHAT WAS WRONG?

No *Real* Continuous Integration

CASE STUDY: WHAT WE DID

The problems mentioned are fundamental...

... but we made things better...

...for this particular company

CASE STUDY: WHAT WE DID

- Formal interfaces: own IDL/DML/RPC compiler
- Better insights & metrics: own effort-free structured logging
- Convenient monorepo, better GIT flow
- Code quality: pure FP and Bifunctor Tagless Final
  - Asynchronous stacktraces for ZIO
- **Fast and reliable tests (including integration ones)**
- **"Flexible Monolith", multi-tenant applications**
  - **Proper Continuous Integration**
- **Simplified onboarding**

CASE STUDY: WHAT WE DID

Issue	Solution
Late problem detection	Early Integration (IDL, Cheap Simulations, Dual tests, Bifunctor IO, Pure FP, TF)
Complicated onboarding	Better Observability (Monorepo, Integration Checks, Dual Tests)
Lack of insights	Effortless insights (Structural Logging, Async Stacktraces)
Code sharing	Observability, No Local Publishing (Monorepo, Unified Stack, Unified Framework)

CASE STUDY: WHAT WE DID

Let's focus on

Tests and Microservices

WHAT WAS WRONG WITH TESTS

- Speed
  - 600 tests
  - 20 minutes total
    - 17 for component startup and shutdown
- Interference
  - Can’t run tests in parallel
- Slow External Dependencies
  - Hard to wire Dummies (Guice)

WHAT WAS WRONG WITH MICROSERVICES

- Hard to share & reuse the code
- One shared Dev Env, hard to replicate locally
  - Up to 70% of developer's time spent for retries and startups
- Automated Deployments & Orchestration
  - Hard
  - Fast, correct startup & shutdown are important...
    - ...but hard to implement manually (order, variability)

GOALS

- Improve development workflows
- Improve deployment workflows
- Continuously Integrate microservice fleet
- Improve team collaboration
- Simplify onboarding

GOALS

How?

– Automatically Reconfigurable Applications

GOALS

Product := Build(Components, Goals)

AUTOMATIC APPLICATION COMPOSITION

GOALS

- Test Business Logic without external dependencies
- Test Business Logic with real external dependencies too
- Correctly share (memoize) heavy resources between tests
- Let a new developer checkout & test immediately
- Let front-end engineers run isolated dev envs quickly & easily

DIStage Framework

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
  }
```

AUTOMATED APPLICATION COMPOSITION: THE IDEA

distage app

=

set of formulas

Product := Recipe(Materials, ...)

+

set of "root" products

Set(Roots, ...)

+

configuration rules

Activation

AUTOMATED APPLICATION COMPOSITION: THE DIFFERENCE

- Like Guice and other DIs, distage
  - turns application wiring code into a set of declarations
  - automatically orders wiring actions

AUTOMATED APPLICATION COMPOSITION: THE DIFFERENCE

But not just that

AUTOMATIC CONFIGURATIONS

```
make[Repository[F]].tagged(Repo.Dummy).from[DummyRepository[F]]

make[Repository[F]].tagged(Repo.Prod).from[ProdRepository[F]]
```

AUTOMATIC CONFIGURATIONS

```
Injector(Activation(Repo -> Repo.Prod))
```

```
make[Repository[F]].tagged(Repo.Dummy).from[DummyRepository[F]]

make[Repository[F]].tagged(Repo.Prod).from[ProdRepository[F]]
```

AUTOMATIC CONFIGURATIONS

```
Injector(Activation(Repo -> Repo.Prod))
```

```
~~make[Repository[F]].tagged(Repo.Dummy).from[DummyRepository[F]]~~

make[Repository[F]].tagged(Repo.Prod).from[ProdRepository[F]]
```

RESOURCES

```
trait DIResource[F[_], Resource] {
  def acquire: F[Resource]
  def release(resource: Resource): F[Unit]
}
```

- Functional description of lifecycle
- Initialisation without mutable state

RESOURCES

```
// library ...
class DbDriver[F[_]] {
  def shutdown: F[Unit]
}
def connectToDb[F[_]: Sync]: DbDriver[F]
// ...
```

```
class ProdRepository[F[_]](db: DbDriver[F]) extends Repository

// definition ...
make[Repository[F]].tagged(Repo.Prod).from[ProdRepository[F]]
make[DbDriver[F]].fromResource(DIResource.make(connectToDb)(_.shutdown))
// ...
```

- Graceful shutdown is always guaranteed, even in case of an exception
- Docker Containers may be expressed as resources
  - Embedded replacement for docker-compose

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 = ???
}
```

DUAL TEST TACTIC

- Helps draft business logic quickly
- Enforces SOLID design
- Does not prevent integration tests
- Speeds up onboarding

DUAL TEST TACTIC: CODE

```
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

```
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

AUTOMATED APPLICATION COMPOSITION: THE DIFFERENCE

DISTAGE:

- verifies correctness ahead of time
- supports resources and lifecycle
- provides a way to define configurable apps
- can perform integration checks ahead of time
- can perform graceful shutdown automatically
- can share components between multiple tests & entrypoints

ROLES AS MEANS OF CONTINUOUS INTEGRATION

Yes, *Integration*

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
```

ROLES

- Multiple "microservices" per process
  - More flexibility for deployments
  - Simulations

ROLES

- Dev environment simulation in one process
  - Even with in-memory mocks!
- Not a replacement for the dev env, but nice addition.
- Imperfect simulations are great anyway!
- Model imperfectness can be granularly adjusted

CONTINUOUS INTEGRATION

- A Microservice Fleet is a Distributed Application
  - And that application is The Product
- Microservices are just components of our Product
- We should integrate them continuously too!
- But it's hard to start and test a distributed application

DISTRIBUTED APPLICATION SIMULATIONS

- Several "microservices" (roles) in one process
- Dummies for every Integration Point
- Easy configuration

- Cheap product models for Continuous Integration
  - There is no need to fail in production

ROLES AND TEAMS

- Multi-tenant applications do not require a Monorepo
  - It's possible to have a repository per role
    - Each role repository may define individual role launcher
    - And there may be one or more launcher repositories
- Drawback: unified framework *is* required

Thank you!

distage example:

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

Pavel:

https://twitter.com/shirshovp

Kai:

https://twitter.com/kai_nyasha