

순수 함수형 스칼라로 웹 애플리케이션 만들기

안녕하세요, 박지수입니다.

- 라스칼라코딩단
- (주)두물머리 CTO
- 스칼라 프로그래밍 2012~2019
- akka, shapeless, doobie 등 기여
- TW, GH, FB @guersam

목표

목표

- 이 질문에 답하기

목표

- 이 질문에 답하기

*“ 함수형 프로그래밍 좋다는 얘기는 많이 듣는데
대체 실무에는 어떻게 쓰나요?*

목표

- 이 질문에 답하기

*“ 함수형 프로그래밍 좋다는 얘기는 많이 듣는데
대체 실무에는 어떻게 쓰나요?”*

- 더 알아보실 분 만들기

목표

- 이 질문에 답하기

*“ 함수형 프로그래밍 좋다는 얘기는 많이 듣는데
대체 실무에는 어떻게 쓰나요?”*

- 더 알아보실 분 만들기
- 시간 내에 마치기

목표가 아닌 것

목표가 아닌 것

- 완벽한 이론 - "Monad란..."

목표가 아닌 것

- 완벽한 이론 - "Monad란..."
- 완전한 이해 - 다 못 알아들으셔도 괜찮아요

목표가 아닌 것

- 완벽한 이론 - "Monad란..."
- 완전한 이해 - 다 못 알아들으셔도 괜찮아요
- 일대일 비교 - "이건 자바의 뽀빠에 해당하고..."

예제: 칭찬 슬랙봇

- 현대인은 외롭다
- 뭔가 달성하면 칭찬해주는 로봇을 만들자
 - `/done` [업적]
- 모두의 업적 보기
 - `/done_list`

시연

<https://github.com/guersam/kcd2019>

순수 함수형 프로그래밍

함수형 프로그래밍?

함수형 프로그래밍?

- 람다?

함수형 프로그래밍?

- 람다?
- 자바8 스트림?

함수형 프로그래밍?

- 람다?
- 자바8 스트림?
- Rx?

세 가지 패러다임

세 가지 패러다임

- 1968 - 구조적
- 1966 - 객체지향
- 1957 - 함수형

세 가지 패러다임

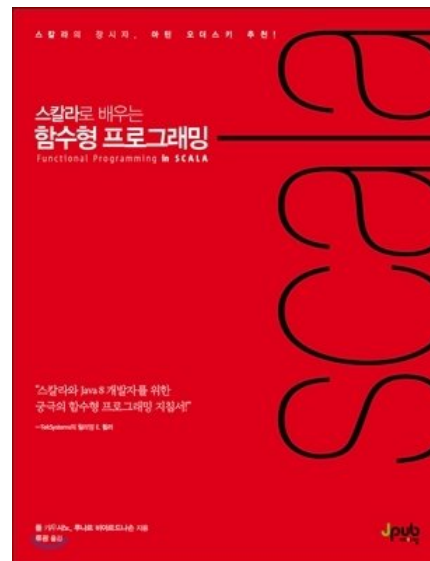
- 1968 - 구조적
- 1966 - 객체지향
- 1957 - 함수형

그리고 50년이 지났습니다

각 패러다임은 뭔가를 제약합니다.

*“ 한 수준에서의 제약은
다른 수준에서의 자유와 힘으로 이어진다.*

- Runar Bjarnason



각 패러다임이 제약하는 것

- 구조적 - GOTO문
- 객체지향 - 함수 포인터
- 함수형 - 대입

불변성

- 함수 조립의 필요조건
- 추론하기 쉬움
- 스레드 안전
- React?
- Event Sourcing

범위를 좁혀 봅시다

- 정적 타입에 한해
 - 대수적 자료구조, 기타 등등
 - 유용하고 재미있지만 오늘은 생략
- 순수 함수형에 한해

"순수" 함수형 프로그래밍

- 대입을 제약

"순수" 함수형 프로그래밍

- 대입을 제약
- + 부작용을 제약

부작용

(Side Effect, 부수효과)

부작용

(Side Effect, 부수효과)

=

참조 투명하지 않은 것

참조 투명성

Referential Transparency

표현식과 참조를 서로 바꿔 써도
프로그램이 동일하게 동작하면

참조상 투명

표현식과 참조를 서로 바꿔 써도
프로그램이 동일하게 동작하면

참조상 투명

```
val a = 3  
  
(a, a) <-> (3, 3)
```


표현식과 참조를 서로 바꿔 써도
프로그램이 동일하게 동작하면

참조상 투명

```
val a = 3  
  
(a, a) <-> (3, 3)
```

그렇지 않으면

참조상 불투명

표현식과 참조를 서로 바꿔 써도
프로그램이 동일하게 동작하면

참조상 투명

```
val a = 3  
  
(a, a) <-> (3, 3)
```

그렇지 않으면

참조상 불투명

```
val a = print("hi")  
  
(a, a) <!--> (print("hi"), print("hi"))
```

표현식과 참조를 서로 바꿔 써도
프로그램이 동일하게 동작하면

참조상 투명

```
val a = 3  
  
(a, a) <-> (3, 3)
```

그렇지 않으면

참조상 불투명

```
val a = print("hi")  
  
(a, a) <!--> (print("hi"), print("hi"))
```

```
val a = iter.next()  
  
(a, a) <!--> (iter.next(), iter.next())
```

참조 투명하면 뭐가 좋나요

- 추론하기 쉽고
- 리팩토링하기 쉽고
- 지연 평가(Lazy Evaluation)도 가능하고
- 좋아요

안 순수한 함수 언어도 있나요?

안 순수한 함수 언어도 있나요?

- 많아요

안 순수한 함수 언어도 있나요?

- 많아요
- LISP (Scheme, Clojure, ...)
- ML (Racket, OCaml, Reason, ...)
- Erlang (Elixir)
- 기타 등등...

Scala

- Martin Odersky
- 2004년 1.0 발표
- OOP + FP
- Impure
- JVM, JavaScript, Native 백엔드
- 트위터, 링크드인, 버라이즌, 모건 스탠리, ...

스칼라와 순수 함수형 프로그래밍

스칼라와 순수 함수형 프로그래밍

- 더 나은 자바 vs 모자란 하스켈(?)

스칼라와 순수 함수형 프로그래밍

- 더 나은 자바 vs 모자란 하스켈(?)
- **Cats**

스칼라와 순수 함수형 프로그래밍

- 더 나은 자바 vs 모자란 하스켈(?)
- **Cats**
- 또는
 - **Scalaz**
 - (스칼라는 아니지만) **Eta**

도메인

자료구조

```
// 업적
case class Achievement(teamId: TeamId, userId: UserId, text: String)

// 축하용 밈
case class Meme(uri: Uri)

// 축하축하
case class Congratulation(achievement: Achievement, text: String, meme: Option[Meme])
```

어디서 많이 본 모양

```
trait AchievementRegistry {  
  def registerAchievement(achievement: Achievement): Unit  
  def findAllAchievementsByTeam(teamId: TeamId): List[Achievement]  
}  
  
trait MemeFinder {  
  def findRandomMemeByKeyword(keyword: String): Meme  
}
```

어디서 많이 본 모양

```
trait AchievementRegistry {  
  def registerAchievement(achievement: Achievement): Unit  
  def findAllAchievementsByTeam(teamId: TeamId): List[Achievement]  
}  
  
trait MemeFinder {  
  def findRandomMemeByKeyword(keyword: String): Meme  
}
```

데이터베이스?

어디서 많이 본 모양

```
trait AchievementRegistry {  
  def registerAchievement(achievement: Achievement): Unit  
  def findAllAchievementsByTeam(teamId: TeamId): List[Achievement]  
}  
  
trait MemeFinder {  
  def findRandomMemeByKeyword(keyword: String): Meme  
}
```

데이터베이스?

비동기?

어디서 많이 본 모양

```
trait AchievementRegistry {  
  def registerAchievement(achievement: Achievement): Unit  
  def findAllAchievementsByTeam(teamId: TeamId): List[Achievement]  
}  
  
trait MemeFinder {  
  def findRandomMemeByKeyword(keyword: String): Meme  
}
```

데이터베이스?

비동기?

동시성?

IO[_]

```
import cats.effect.IO

trait AchievementRegistry {

  def registerAchievement(achievement: Achievement): IO[Unit]

  def findAllAchievementsByTeam(teamId: TeamId): IO[List[Achievement]]
}

trait MemeFinder {
  def findRandomMemeByKeyword(keyword: String): IO[Meme]
}
```

IO[_]

```
import cats.effect.IO

trait AchievementRegistry {

  def registerAchievement(achievement: Achievement): IO[Unit]

  def findAllAchievementsByTeam(teamId: TeamId): IO[List[Achievement]]
}

trait MemeFinder {
  def findRandomMemeByKeyword(keyword: String): IO[Meme]
}
```

Effect = Value

IO[_]

```
import cats.effect.IO

trait AchievementRegistry {

  def registerAchievement(achievement: Achievement): IO[Unit]

  def findAllAchievementsByTeam(teamId: TeamId): IO[List[Achievement]]
}

trait MemeFinder {
  def findRandomMemeByKeyword(keyword: String): IO[Meme]
}
```

Effect = Value

값(표현식) 중심 프로그래밍

Future[_]

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import cats.implicits._ // for >> operator

val ha = Future(print("h")) >> Future(print("a"))
// ha
ha >> ha

Future(print("h")) >> Future(print("a")) >> Future(print("h")) >> Future(print("a"))
// haha
```

Future[_]

참조상 불투명

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import cats.implicits._ // for >> operator

val ha = Future(print("h")) >> Future(print("a"))
// ha
ha >> ha

Future(print("h")) >> Future(print("a")) >> Future(print("h")) >> Future(print("a"))
// haha
```

Future[_]

참조상 불투명

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import cats.implicits._ // for >> operator

val ha = Future(print("h")) >> Future(print("a"))
// ha
ha >> ha

Future(print("h")) >> Future(print("a")) >> Future(print("h")) >> Future(print("a"))
// haha
```

IO[_]

```
import cats.effect.IO
import cats.implicits._

val ha = IO(print("h")) >> IO(print("a"))
(ha >> ha).unsafeRunSync()
// haha

(IO(print("h")) >> IO(print("a")) >> IO(print("h")) >> IO(print("a"))).unsafeRunSync()
// haha
```


Future[_]

참조상 불투명

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import cats.implicits._ // for >> operator

val ha = Future(print("h")) >> Future(print("a"))
// ha
ha >> ha

Future(print("h")) >> Future(print("a")) >> Future(print("h")) >> Future(print("a"))
// haha
```

IO[_]

참조상 투명

```
import cats.effect.IO
import cats.implicits._

val ha = IO(print("h")) >> IO(print("a"))
(ha >> ha).unsafeRunSync()
// haha

(IO(print("h")) >> IO(print("a")) >> IO(print("h")) >> IO(print("a"))).unsafeRunSync()
// haha
```

결과값을 바꾼다

다른 IO와 합친다

순차적으로 실행한다

결과값을 바꾼다

```
IO(1).map(_ + 10) <-> IO(11)
```

다른 IO와 합친다

순차적으로 실행한다

결과값을 바꾼다

```
IO(1).map(_ + 10) <-> IO(11)
```

다른 IO와 합친다

```
case class Foo(bar: Int, baz: String)  
  
(IO(42), IO("answer")).mapN(Foo) <-> IO(Foo(42, "answer"))
```

순차적으로 실행한다

결과값을 바꾼다

```
IO(1).map(_ + 10) <-> IO(11)
```

다른 IO와 합친다

```
case class Foo(bar: Int, baz: String)

(IO(42), IO("answer")).mapN(Foo) <-> IO(Foo(42, "answer"))
```

순차적으로 실행한다

```
def fetchEmployee(id: EmployeeId): IO[Employee]

def fetchBranch(id: BranchId): IO[Branch]

// ...

fetchEmployee("guersam").flatMap(e => fetchBranch(e.branchId)): IO[Branch]
```

언제까지?

가능한 한 멀리, 런타임 직전까지


```
import cats.effect.IOApp
import cats.implicits._















































object App extends IOApp {

  val app = new AppF[IO]

  def run(args: List[String]): IO[ExitCode] =
    app.run.use(_ => IO.never).as(ExitCode.Success)
}
```

정말 이게 다 필요할까?

▼   IO[A]

-   map[B](A => B): IO[B]
-   flatMap[B](A => IO[B]): IO[B]
-   attempt: IO[Either[Throwable, A]]
-   runAsync(Either[Throwable, A] => IO[Unit]): SyncIO[Unit]
-   runCancelable(Either[Throwable, A] => IO[Unit]): SyncIO[CancelToken[IO]]
-   unsafeRunSync(): A
-   unsafeRunAsync(Either[Throwable, A] => Unit): Unit
-   unsafeRunAsyncAndForget(): Unit
-   unsafeRunCancelable(Either[Throwable, A] => Unit): CancelToken[IO]
-   unsafeRunTimed(Duration): Option[A]
-   unsafeToFuture(): Future[A]
-   start(ContextShift[IO]): IO[Fiber[IO, A @uncheckedVariance]]
-   uncancellable: IO[A]
-   to[F[_]](LiftIO[F]): F[A @uncheckedVariance]
-   timeoutTo[A2 >: A](FiniteDuration, IO[A2])(Timer[IO], ContextShift[IO]): IO[A2]
-   timeout(FiniteDuration)(Timer[IO], ContextShift[IO]): IO[A]
-   bracket[B](A => IO[B])(A => IO[Unit]): IO[B]
-   bracketCase[B](A => IO[B])((A, ExitCase[Throwable]) => IO[Unit]): IO[B]
-   guarantee(IO[Unit]): IO[A]
-   guaranteeCase(ExitCase[Throwable] => IO[Unit]): IO[A]
-   handleErrorWith[AA >: A](Throwable => IO[AA]): IO[AA]
-   redeem[B](Throwable => B, A => B): IO[B]
-   redeemWith[B](Throwable => IO[B], A => IO[B]): IO[B]
-   toString: String

Effect, Effect, Effect

- `monix.eval.Task`
- `monix.eval.Coeval`
- `scalaz.concurrent.Task`
- `scalaz.effect.IO`
- `scalaz.zio.IO`

더 복잡한 타입

- `OptionT[IO, ?]`

더 복잡한 타입

- `OptionT[IO, ?]`
- `EitherT[IO, E, ?]`

더 복잡한 타입

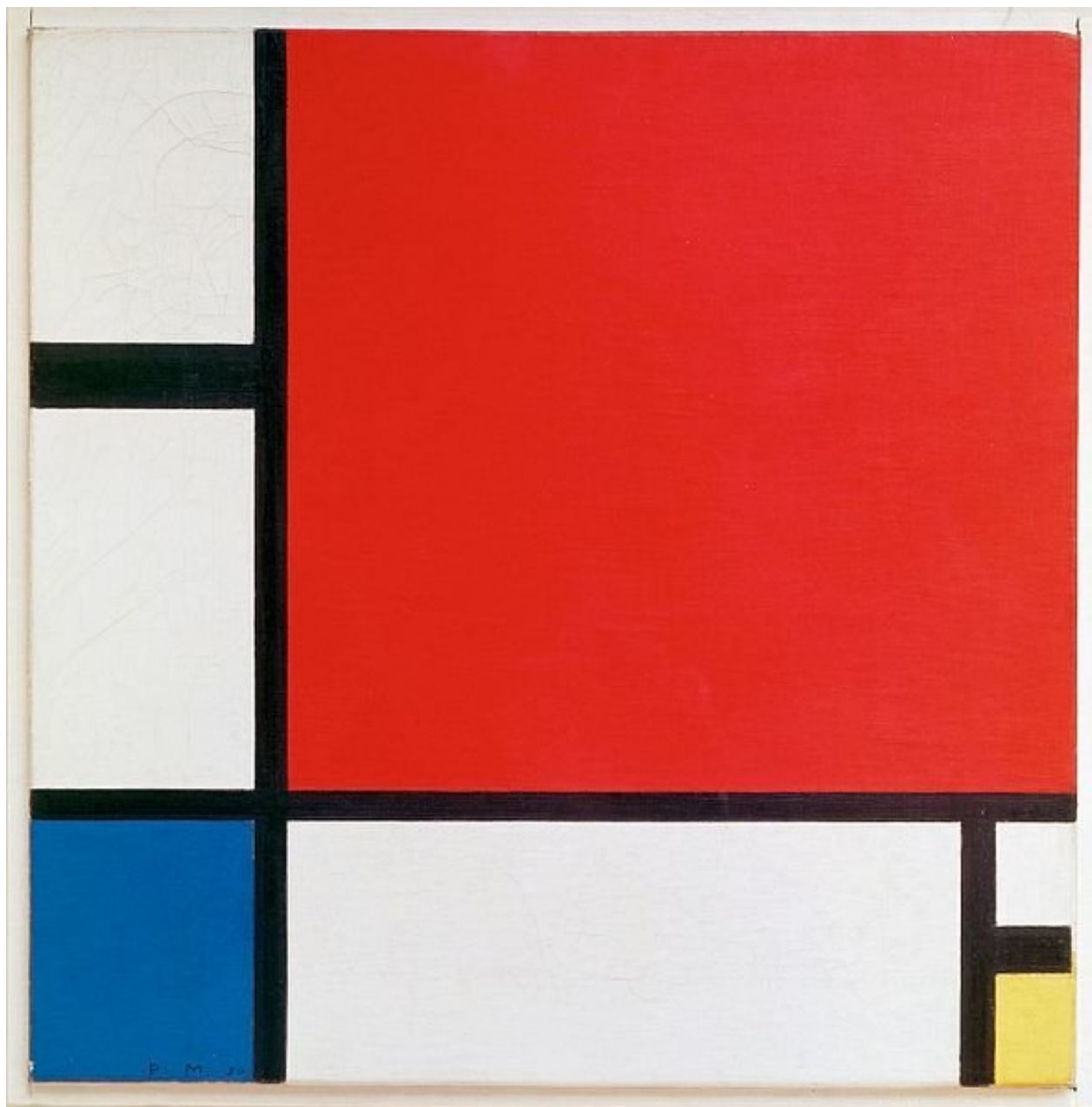
- `OptionT[IO, ?]`
- `EitherT[IO, E, ?]`
- `ActionT[IO, S, E, ?]`

더 복잡한 타입

- `OptionT[IO, ?]`
- `EitherT[IO, E, ?]`
- `ActionT[IO, S, E, ?]`
- `Kleisli[OptionT[IO, ?], A, ?]`

더 복잡한 타입

- `OptionT[IO, ?]`
- `EitherT[IO, E, ?]`
- `ActionT[IO, S, E, ?]`
- `Kleisli[OptionT[IO, ?], A, ?]`
- ...



F[_]

```
trait AchievementRegistry[F[_]] {  
  def registerAchievement(achievement: Achievement): F[Unit]  
  def findAllAchievementsByTeam(teamId: TeamId): F[List[Achievement]]  
}  
  
trait MemeFinder[F[_]] {  
  def findRandomMemeByKeyword(keyword: String): F[Meme]  
}
```

몰라 나중에 정할래

추상화

- 디테일은 버린다
 - 적극적으로 무시
- 일반화와는 다르다

결과값을 바꾼다

다른 IO와 합친다

순차적으로 실행한다

결과값을 바꾼다

```
Functor[F].map(fa: F[Int])(_.toString): F[String]
```

다른 IO와 합친다

순차적으로 실행한다

결과값을 바꾼다

```
Functor[F].map(fa: F[Int])(_.toString): F[String]
```

다른 IO와 합친다

```
Applicative[F].product(fa: F[A], fb: F[B]): F[(A, B)]
```

순차적으로 실행한다

결과값을 바꾼다

```
Functor[F].map(fa: F[Int])(_.toString): F[String]
```

다른 IO와 합친다

```
Applicative[F].product(fa: F[A], fb: F[B]): F[(A, B)]
```

순차적으로 실행한다

```
def fetchEmployee(id: EmployeeId): F[Employee]

def fetchBranch(id: BranchId): F[Branch]

// ...

Monad[F].flatMap(fetchEmployee("guersam")) {
  e => fetchBranch(e.branchId)
}: F[Branch]
```

결과값을 바꾼다

```
Functor[F].map(fa: F[Int])(_.toString): F[String]
```

다른 IO와 합친다

```
Applicative[F].product(fa: F[A], fb: F[B]): F[(A, B)]
```

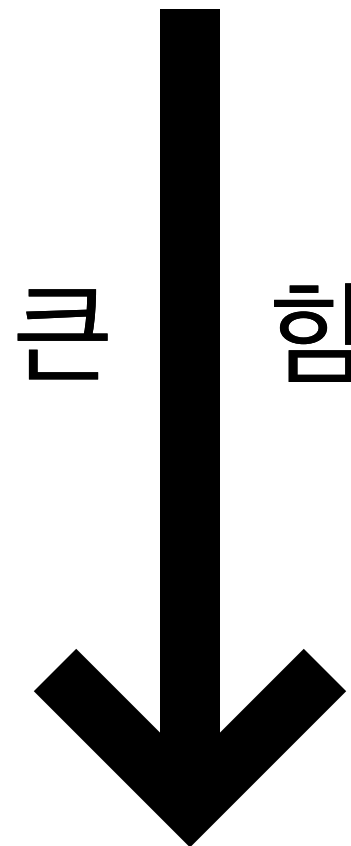
순차적으로 실행한다

```
def fetchEmployee(id: EmployeeId): F[Employee]

def fetchBranch(id: BranchId): F[Branch]

// ...

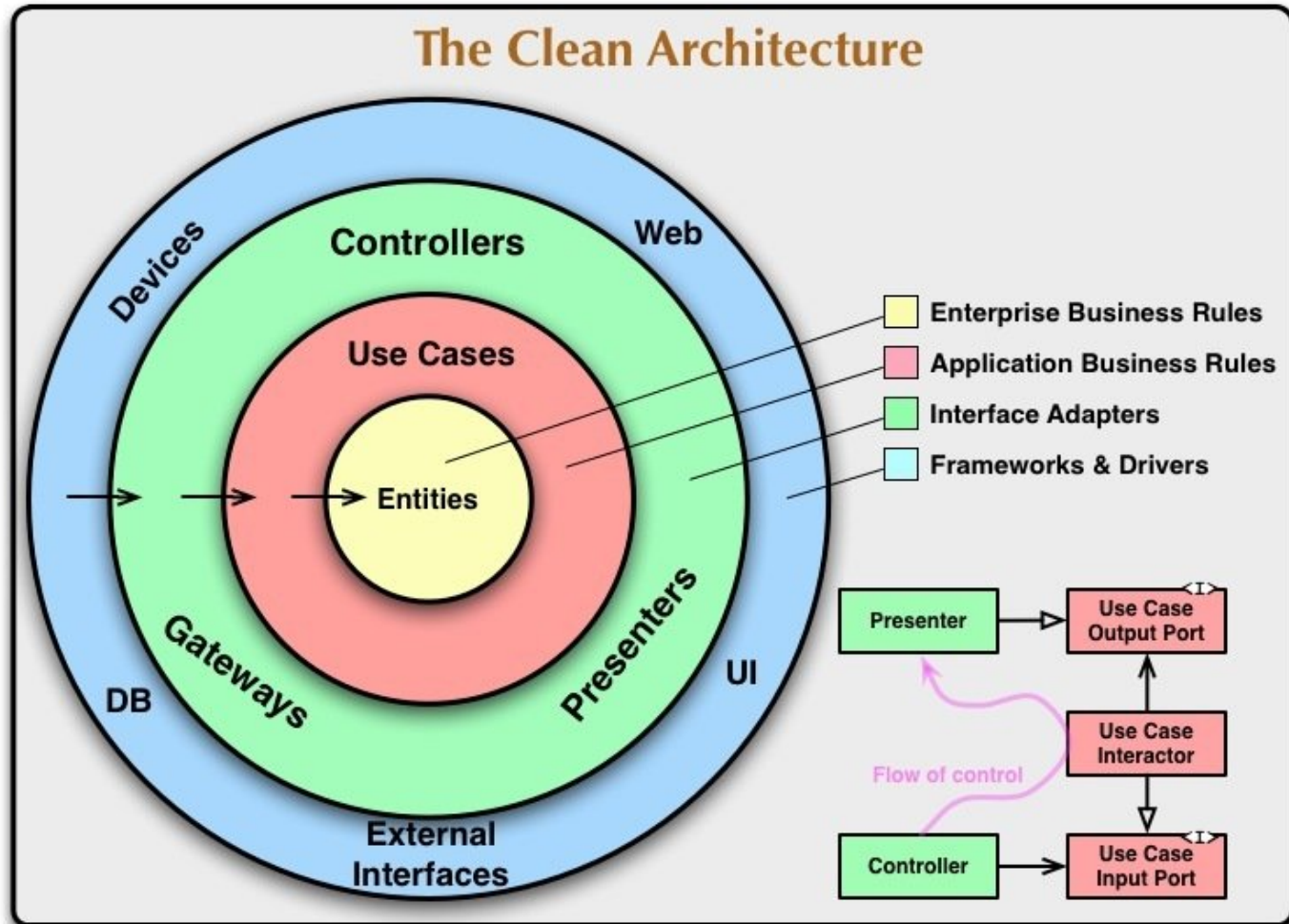
Monad[F].flatMap(fetchEmployee("guersam")) {
  e => fetchBranch(e.branchId)
}: F[Branch]
```



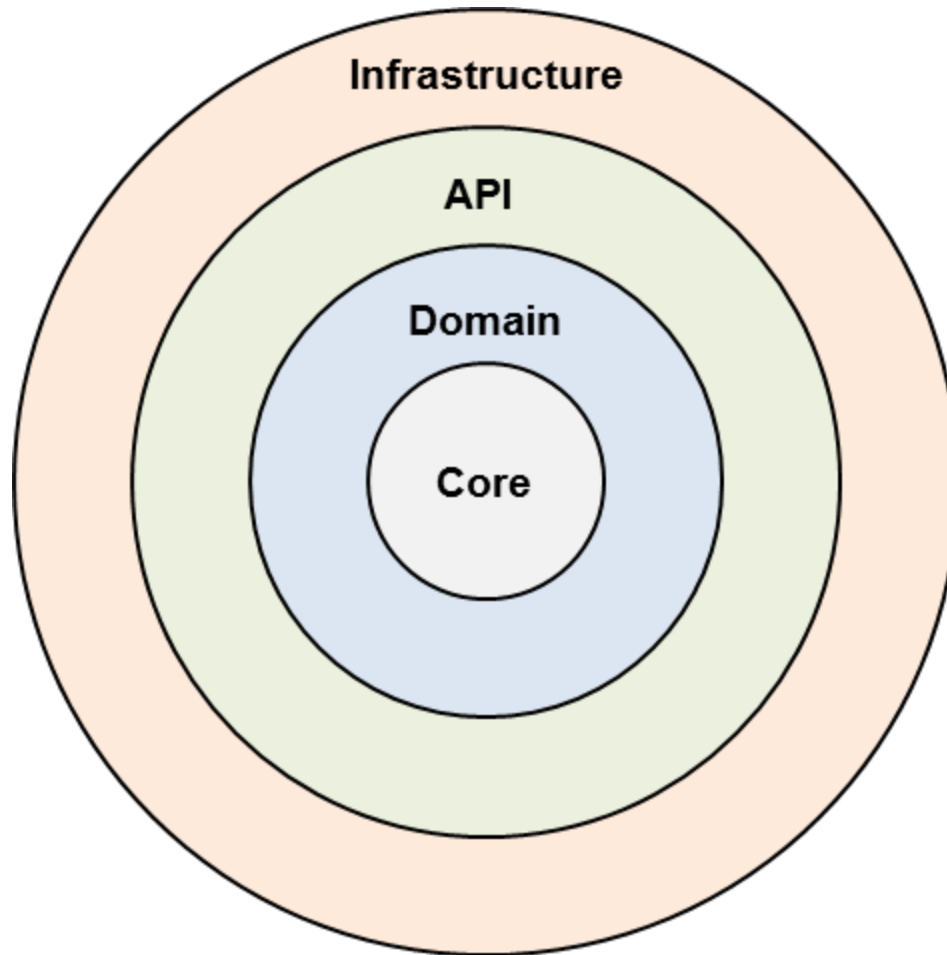
정확히 무슨 타입인지 몰라도 조립할 수 있어요

```
trait CongratulationService[F[_]] {  
  def congratulate(achievement: Achievement): F[Congratulation]  
}  
  
object CongratulationService {  
  
  def apply[F[_]: ApplicativeError[?[_], Throwable]](  
    achievementRegistry: AchievementRegistry[F],  
    memeFinder: MemeFinder[F]  
  ) =  
    new CongratulationService[F] {  
  
      def congratulate(achievement: Achievement): F[Congratulation] = {  
        val registrationF: F[Unit] =  
          achievementRegistry.registerAchievement(achievement)  
  
        val memeF: F[Option[Meme]] =  
          memeFinder  
            .findRandomMemeByKeyword("congratulations")  
            .attempt.map(_.toOption)  
  
        (registrationF *> memeF).map { maybeMeme =>  
          Congratulation(achievement, "Congratulations!", maybeMeme)  
        }  
      }  
    }  
}
```

아키텍처



아키텍처



비즈니스 로직을 중심에
나머지는 늦게 결정할 수록 좋아요

웹은 디테일이다

```
class ApiRoutes[F[_]: Effect](
    congratulationService: CongratulationService[F],
    achievementRegistry: AchievementRegistry[F],
    ) extends Http4sDsl[F] {

  // ...

  private val registerAchievement = HttpRoutes.of[F] {
    case req @ POST -> Root / "done" =>
      req.decode[Achievement] { achievement =>
        for {
          congrats <- congratulationService.congratulate(achievement)
          slackResp = SlackPresenter.renderCongratulation(congrats)
          resp <- Ok(slackResp)
        } yield resp
      }
  }

  private val listTeamAchievements = HttpRoutes.of[F] {
    case req @ POST -> Root / "done-list" =>
      // ...
  }

  val routes: HttpRoutes[F] =
    registerAchievement <+> listTeamAchievements
}
```

Http4s

`Request[F] => F[Response[F]]`

데이터베이스도 디테일이다

```
object PostgresAchievementRegistry extends AchievementRegistry[ConnectionIO] {

  def registerAchievement(achievement: Achievement): ConnectionIO[Unit] =
    Statements
      .registerAchievement(achievement.teamId, achievement.userId, achievement.text)
      .run.void

  def findAllAchievementsByTeam(teamId: TeamId): ConnectionIO[List[Achievement]] =
    Statements.findAchievementsByTeam(teamId).to[List]

  object Statements {

    def registerAchievement(teamId: TeamId, userId: UserId, achievement: String): Update0 =
      sql"""INSERT INTO achievements (team_id, user_id, text)
        VALUES (${teamId}, ${userId}, ${achievement})
      """.update

    def findAchievementsByTeam(teamId: TeamId): Query0[Achievement] =
      sql"""SELECT team_id, user_id, text
        FROM achievements
        WHERE team_id = ${teamId}
        ORDER BY id DESC
      """.query

  }

  // ...
}
```

Doobie

ConnectionIO ~> F

법칙과 속성에 따른 테스트

```
trait AchievementRegistryLaws[F[_]] {  
  def algebra: AchievementRegistry[F]  
  implicit def M: Monad[F]  
  
  def registerFindAllComposition(a: Achievement) =  
    (  
      algebra.registerAchievement(a) >>  
      algebra.findAllAchievementsByTeam(a.teamId).map(_.headOption)  
    ) <->  
      M.pure(Some(a))  
}
```

```
trait AchievementRegistryTests[F[_]] extends Laws {  
  def laws: AchievementRegistryLaws[F]  
  
  def algebra(implicit  
    arbAchievement: Arbitrary[Achievement],  
    eqFOptAchievement: Eq[F[Option[Achievement]]]) =  
    new SimpleRuleSet(  
      name = "Achievements",  
      "register and findAll compose" -> forAll(laws.registerFindAllComposition _)  
    )  
}
```

법칙과 속성에 따른 테스트 (1)

```
class InMemoryAchievementRegistryTest
  extends FunSuite
  with Discipline
  with ArbitraryInstances
  with TestInstances {

  implicit val context = TestContext()

  val registry = InMemoryAchievementRegistry.make[IO].unsafeRunSync

  checkAll(
    "InMemoryAchievementRegistry",
    AchievementRegistryTests(registry).algebra
  )
}
```

법칙과 속성에 따른 테스트 (2)

```
class PostgresAchievementRegistryTest
  extends FunSuite
  with Discipline
  with ArbitraryInstances
  with ConnectionIOInstances[IO]
  with TestInstances {

  implicit val context = TestContext()
  implicit val M: Monad[ConnectionIO] = Async[ConnectionIO]

  val transactor = TestTransactor.autoRollback

  checkAll(
    "PostgresAchievementRegistry",
    AchievementRegistryTests(PostgresAchievementRegistry).algebra
  )
}
```


감사합니다.

@guersam

참고자료

함수형 프로그래밍

- John Huges - Why Functional Programming Matters
<https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf>
 - 한주영님의 한글 번역본 <https://bit.ly/2BVUacR>
- Runar Bjarnason - Constraints Liberate, Liberties Constrain
<https://youtu.be/GqmsQeSzMdw>
- Valentin Kasas - Carpenters and Cartographers
https://youtu.be/_JOHRAgfl5Y

참고자료

스칼라 기초

- Marin Ordersky - Programming in Scala (한글판)
<https://www.aladin.co.kr/shop/wproduct.aspx?ItemId=108696276>
- EPFL - Functional Programming Principles in Scala
<https://www.coursera.org/learn/progfun1>

웹 브라우저에서 스칼라 배우기

- <https://www.scala-exercises.org/>

참고자료

스칼라와 함수형 프로그래밍

- Paul Chiusano and Runar Bjarnason - 스칼라로 배우는 함수형 프로그래밍
<https://www.aladin.co.kr/shop/wproduct.aspx?ItemId=54199516>
- Noel Welsh and Dave Gurnell - Scala with Cats
<https://underscore.io/books/scala-with-cats/>
- Sam Halliday - Functional Programming for Mortals
<https://leanpub.com/fpmortals>

참고자료

함수형 프로그래밍과 이펙트

- Rob Norris - Functional Programming with Effects

<https://na.scaladays.org/schedule/functional-programming-with-effects>

함수형 스칼라와 도메인 모델링

- Debasish Ghosh - Functional and Algebraic Domain Modeling

https://youtu.be/BskNvfNjU_8

참고자료

함수형 HTTP, JDBC

- Jakub Kozlowski - Lightweight, Functional Microservices with Http4s and Doobie
<https://youtu.be/fQfMiUDsLv4>

테스팅

- Marcin Rzeźnicki - Tagless with Discipline — Testing Scala Code The Right Way
<https://medium.com/iterators/tagless-with-discipline-testing-scala-code-the-right-way-e74993a0d9b1>