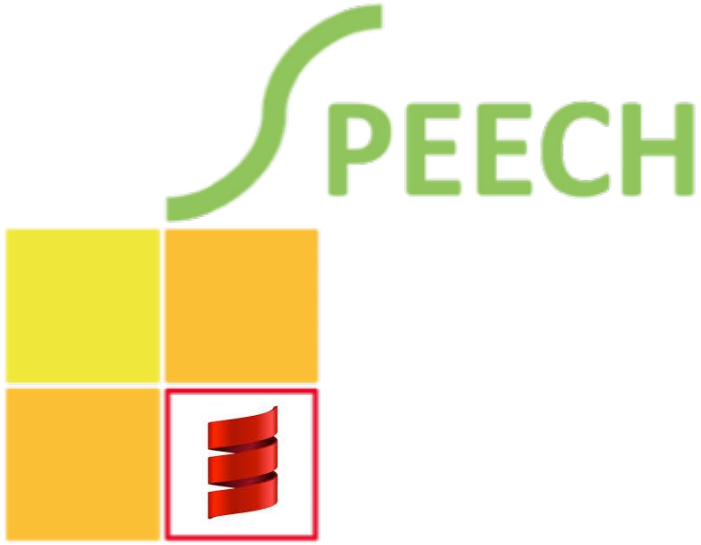
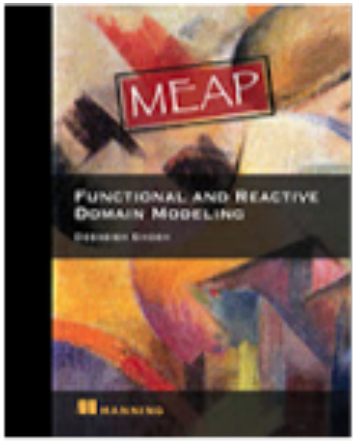
Functional Programming in Scala

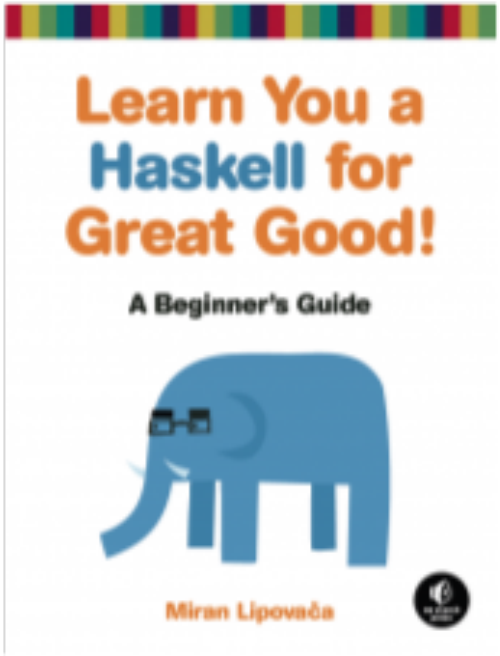
Juan Manuel Serrano Hidalgo
Habla Computing
@jmshac

Jesús López González
Habla Computing
@jeslg


SPEECH


Functional Programming in
Scala
Paul Chiusano
Rúnar Bjarnason


MEAP
Functional and Reactive Domain Modeling


Learn You a Haskell for Great Good!
A Beginner's Guide
Miran Lipovača

Functional programming

~~*Haskell*~~ *for all*

Gabriel Gonzalez

The category design pattern

The functor design pattern

....

Program0 - Ad-hoc composition

Program1 - Function composition

Program2 - Side effects

Program3 - Logging effects

Program4 - Option effects

Program5 - Logging + Option effects

Program6 - Operators for combined effects

Program7 - Kleisli arrow for Logging

Program8 - Kleisli arrow for combined effects

Program9 - Kleisli conversions and combinat.

Functions

```
val factorial: Int => Int

val scale: (Int, Image) => Image

val animation: Double => Image

val server: Request => State => (Result, State)

val ... << choose your favourite domain >>
```

Functions

```
val factorial: Function1[Int, Int]

val scale: Function2[Int, Image, Image]

val animation: Function1[Double, Image]

val server: Function1[Request, Function1[State, (Result, State)]

val ... << choose your favourite domain >>
```

Functions

```
def factorial(i: Int): Int

def scale(i: Int, img1: Image): Image

def animation: Function1[Double, Image]

def server(req: Request)(implicit ctx: State): (Result, State)

def ... << choose your favourite domain >>
```

Think of **functions** as *composable* computational devices that transform input *values* into output values, and *do nothing more.*

Values

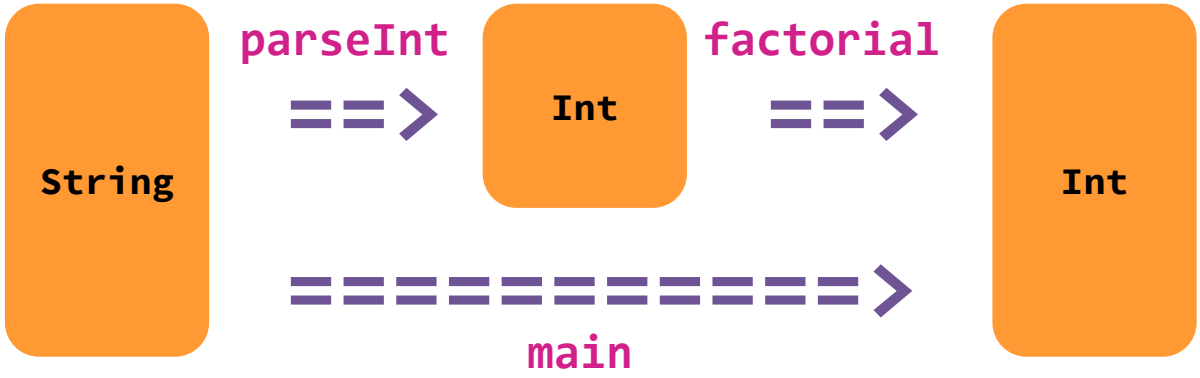
```
val i: Int = 3

val image: Image = bitmap(“file.png”)

val state: State = (users, entries)

val users: List[User] = List(user1,user2)

val user1: User = User(name=“juan”, age=21)

val factorial: Int => Int = (x: Int) => ...

val scale: Int => (Image => Image) = ...
```

Large programs are made up of many functions which are **composed** together to create more aggregate functions, that eventually give rise to the *single* function that represents the whole program.

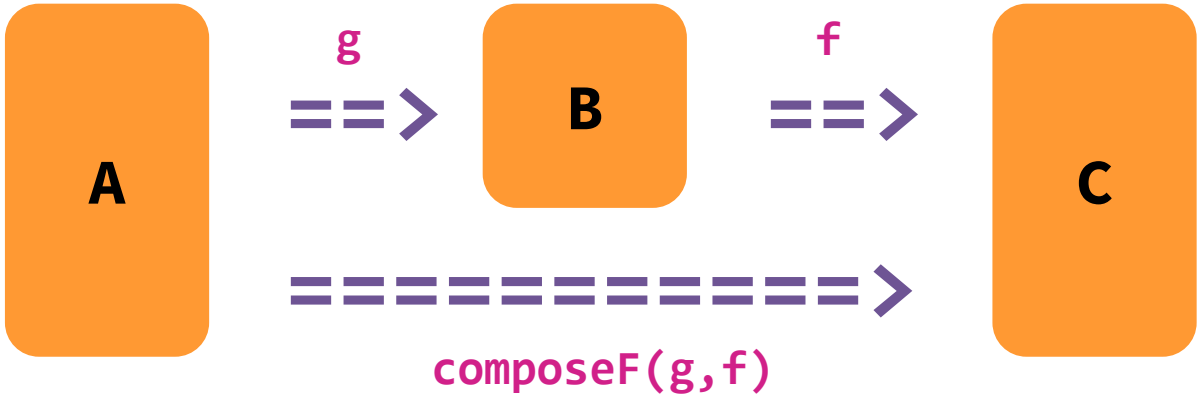
Composition

```
def parseInt(i: String): Int =
  “transforms a string into a number”


def factorial(n: Int): Int =
  “compute the factorial of a number”


def main(s: String): Int =
  “compute the factorial of the number represented by the string”
```

Composition



Composition




```
def composeF(g: B => C, f: A => B): A => C
```

Composition (infix notation)

```
trait Function1[-T1, +R] extends AnyRef {

  def apply(v1: T1): R

  def compose[A](g: A => T1): A => R = {
    x => apply(g(x))
  }

  def andThen[A](g: R => A): T1 => A = {
    x => g(apply(x))
  }

}
```

Side effects

```
def factorial(n: Int): Int =
  if (n < 0)
    throw new IllegalArgumentException
  else {
    val result = if (n==0) 1 else n * factorial(n-1)
    println(s"factorial($n)=$result")
    result
  }
```

Effects refer to the actual changes effected to the environment, beyond the computed values.

Side effects are those effects which are not declared in the function signature, and that are nonetheless enacted when the function is computed.

Effects

```
/* PURE */

def parseInt(s: String): Logging[Int]

def factorial1(n: Int): Logging[Int]

def main: String => Logging[Int]

/* IMPURE */

def logInterpreter[T](logging: => Logging[T]): Unit
```

Effects

```
/* PURE */

def parseInt(s: String):Option[Int]

def factorial1(n: Int): Option[Int]

def main: String => Option[Int]

/* IMPURE */

def optInterpreter[T](result: => Option[T]): Unit
```

Effects

```
/* PURE */

def parseInt(s: String): Logging[Option[Int]]

def factorial1(n: Int): Logging[Option[Int]]

def main: String => Logging[Option[Int]]

/* IMPURE */

def interpreter[T](result: => Logging[Option[Int]]): Unit

def logInterpreter[T](logging: => Logging[T]): Unit

def optInterpreter[T](result: => Option[T]): Unit
```

Why Functional Programming?

- Modularity (aka composability)
  - Is our program structured into modules that can be easily separated, composed and reused?
- Testability
  - Can we unit test *all* of our modules?
- Understandability
  - Is it easy to reason about the flow of computation?
- Efficiency
  - e.g., does our program really make use of multi-cores?

Algebraic data types (ADTs)

```
// ADT declarations

sealed trait DataType[T]
case object Case1 extends DataType[...]
case class Case2[T](arg1: T1, arg2: T2, ...) extends DataType[T]


// pattern matching

val t: DataType[T]
val v: Int = t match {
  case c1@Case1 => ...:Int
  case c2@Case2(a1,a2,...) => ...:Int
}
```

Algebraic data types (ADTs)

```
sealed trait Option[+T]
case object None extends Option[Nothing]
case class Some[+T](value: T) extends Option[T]


sealed trait Logging[A]
case class Debug[A](msg: String, next: Logging[A]) extends Logging[A]
case class Error[A](msg: String, next: Logging[A]) extends Logging[A]
case class Return[A](value: A) extends Logging[A]
```

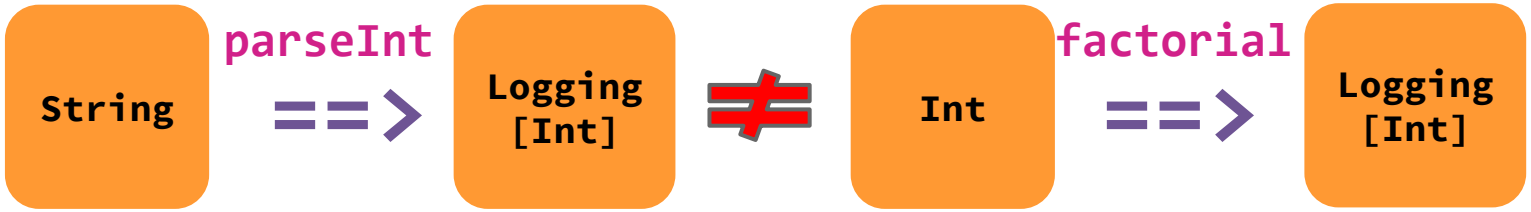
Monads

"a monad is a monoid in the category of endofunctors, what's the problem?" *particular phrasing by James Iry*

Data types that represent computations of values of some type which can be *“concatenated”*. This is the **flatMap** combinator.

They also must allow us to create a pure computation from a given value, through the **point** operator.

Composition fails !?




```
scala> (factorial _) compose parseInt
<console>:15: error: type mismatch;
 found   : Logging[Int]
 required: Int
              (factorial _) compose parseInt
                                    ^
```

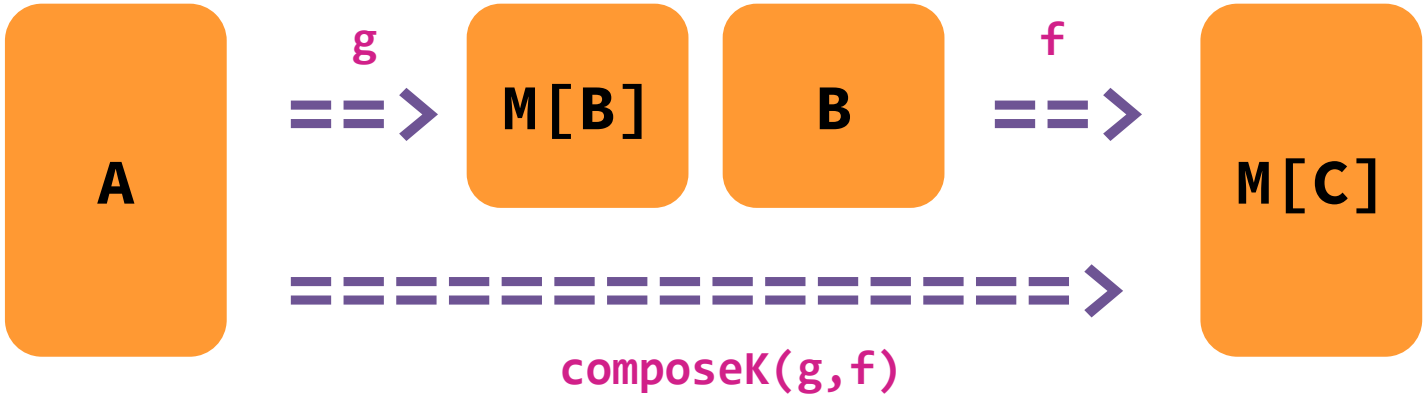
Combinators (reloaded)

Functions are not the only kind of things that compmose. Composable things are called **arrows** in category theory. All this leads to **categorical programming ...**

Arrows

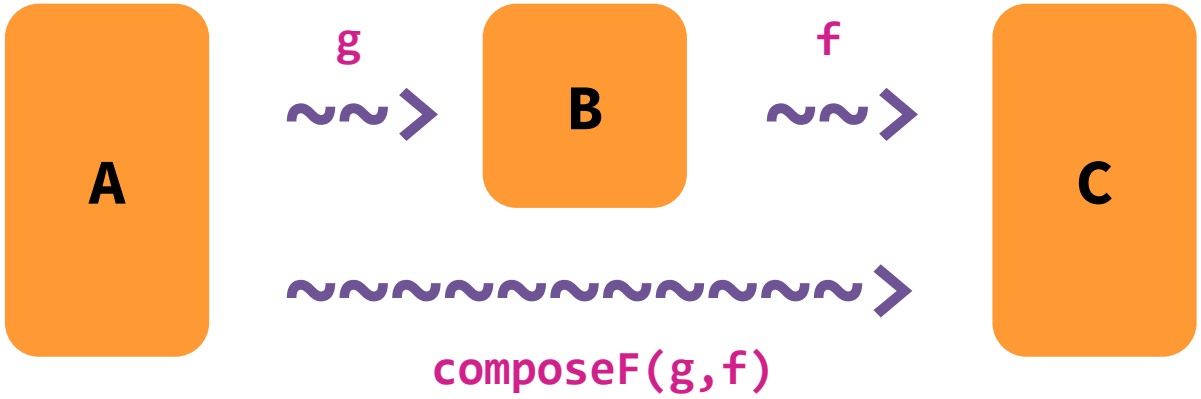
```
type Function[-T,+R]

type KleisliLogging[-T,R] = T => Logging[R]

type KleisliOption[-T,R] = T => Option[R]

type KleisliMyEffect[-T,R] = T => Logging[Option[R]]

type IterateeOption[T,E] = ...

type List[T] = ... !

type Integer = ... !
```

Kleisli composition




```
def composeK[A,B,C](
    g: B => M[C]
    f: A => M[B]
): A => M[C]
```

Kleisli composition




```
type ~>[A,B] = A => M[B]

def composeK[A,B,C](g: B ~> C, f: A ~> B): A ~> C
```

is FP so costly?

- Effect manipulation
  - concat, changeValue, value, ...
- Effect combinators
  - concatMap
- New arrows
  - Kleisli, composeK
- Arrow combinators
  - if_K

Scalaz to the rescue!

avoid pattern matching?

- (endo)functors, monads!

compose effects?

- Monad transformers!

compose different arrows?

- compose(f: A => Option[B], f: B => C): A => Option[C]
- Functors!

make our data types composable?

- Free monads!

Hands On!

Deploying Services in Play


http://www.etsy.com/shop/sharpwriter


NOT BAD
ENTRANCE
PLEASE WAIT
FOR GATEKEEPER TO
OPEN THE GATE FOR
YOU. THANK YOU! :)

Deploying Services in Play

- Play!
  - Why Play?
  - Introduction
- Web Dictionary
  - models
  - services
  - controller
  - testing

Why Play?

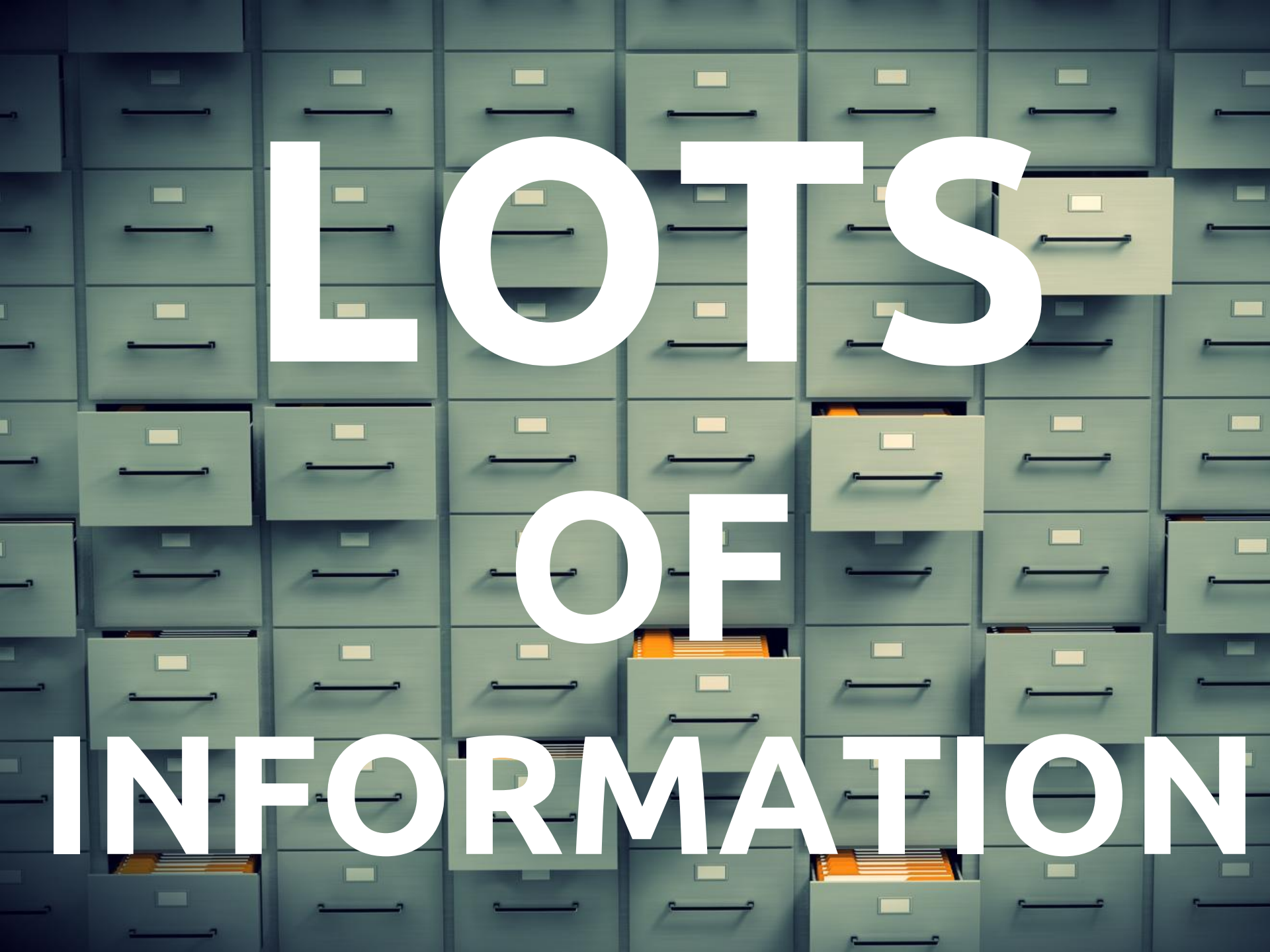
★ Fits CodeMotion (Web)

★ Popular Project

★ Real Life

Play - Introduction

- Web Framework (Typesafe)
  - Guillaume Bort (2007)
- Java & **Scala**
- Stateless
- Asynchronous
- **MVC**
- **HTTP awareness**

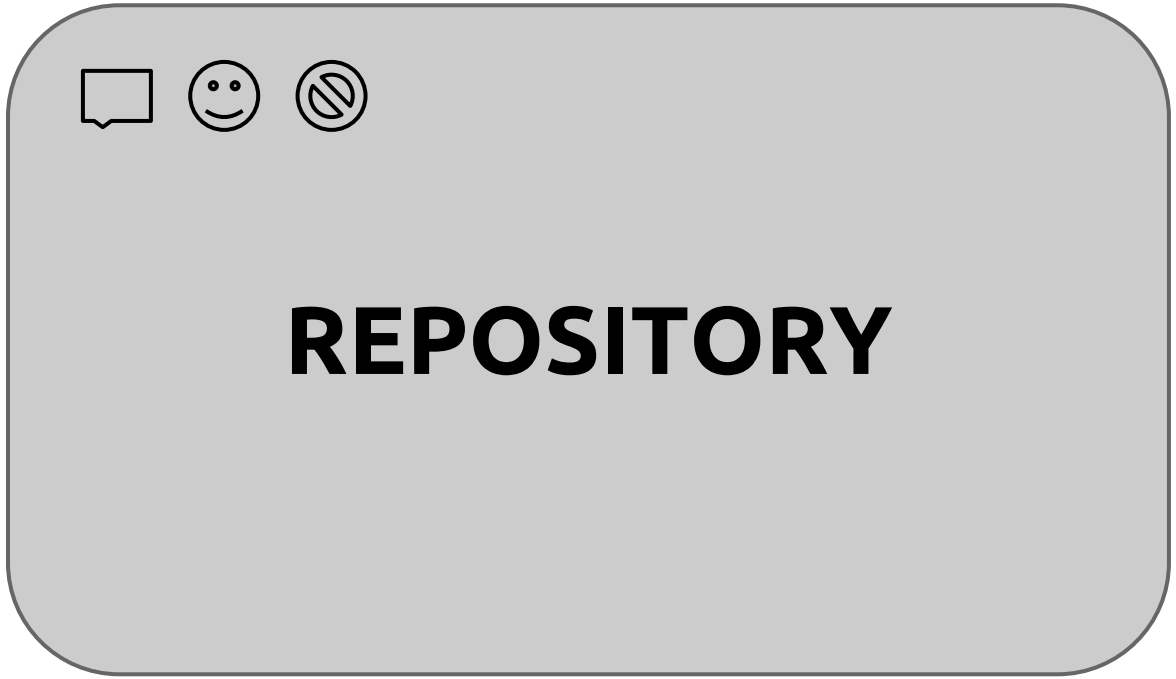

LOTS
OF
INFORMATION

Dictionary - models

`es.scalamad.dictionary.models`

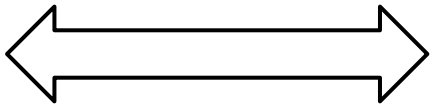
- **User**: name, last and associated permission
- **Permission**: read/write permission
- **Word**: just a *String*

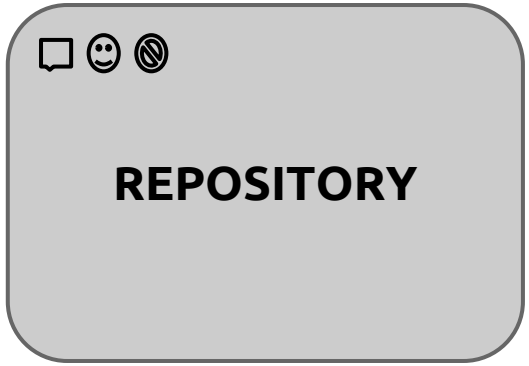
Dictionary - models



Dictionary- models

- ➔ Get Entry
- ➔ Set Entry
- ➔ Remove Entry
- ➔ Get User
- ➔ Set User
- ➔ Remove User
- ➔ Can a user Read?
- ➔ Can a user Write?





Dictionary - services

`es.scalamad.dictionary.services`

- **Word** services
  - Get/Add/Remove word entry
- **User** services
  - Get/Add/Remove user
- **Permission** services
  - Can read/write?

Dictionary - services

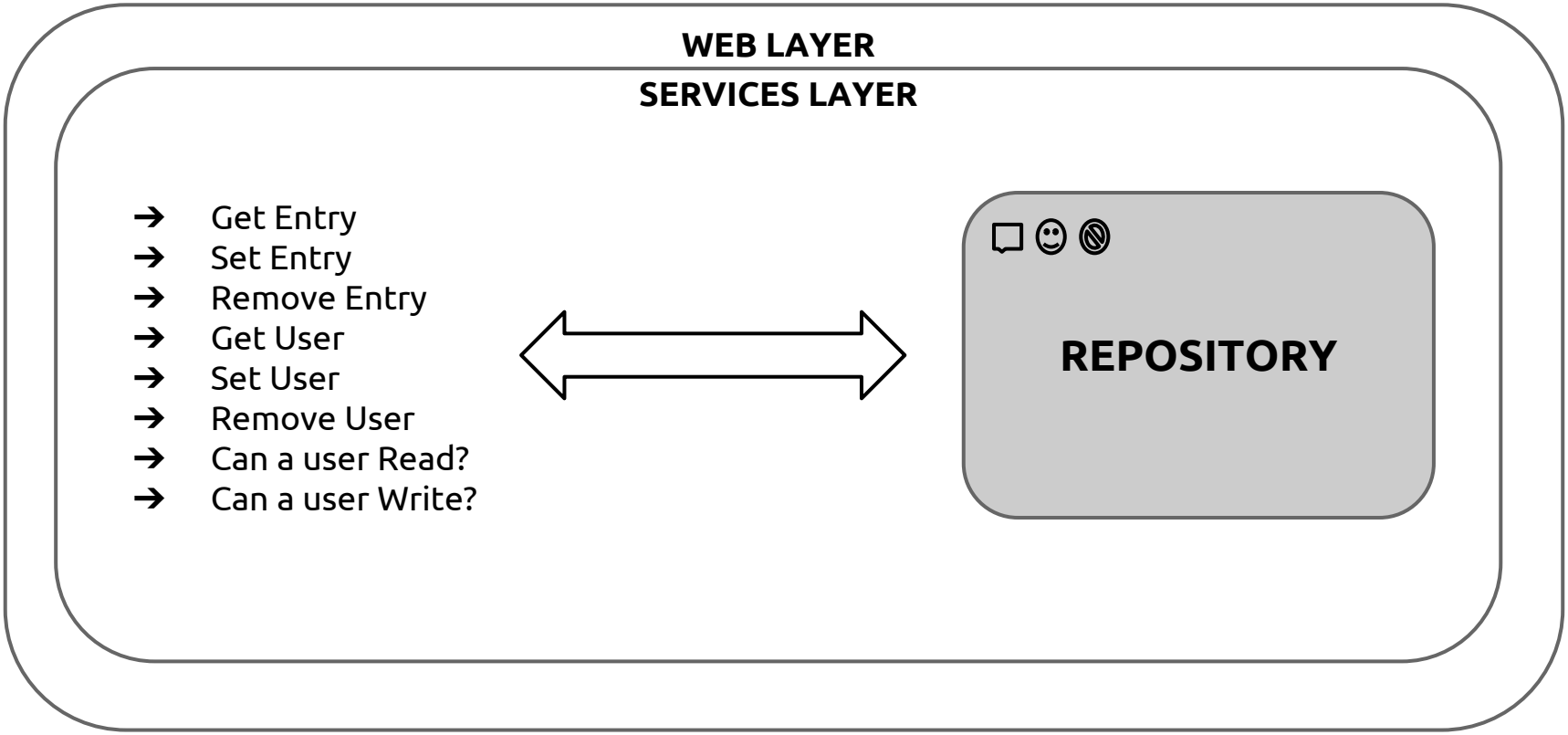
```
In => Repo[Out]
```

Dictionary - controller

`es.scalamad.dictionary.controllers`

- Actions
  - Service Wrappers

Dictionary - controller



Dictionary - controller

```
Request[Body] => Result
```

Dictionary - controller

POST /
(“emotion”, “a feeling of any kind”)

201 Created
-

```
Request[(String, String)] => Result
```

Dictionary - controller

GET /emotion
-

200 OK
a feeling of any kind

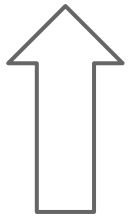
```
Request[Unit] => Result
```

Dictionary - controller

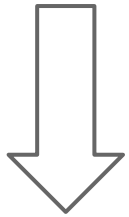
```
                In => Repo[Out]
                   ¿?
Request[Body] => Result
```

Dictionary - controller

```
In => Repo[Out]
```



translator: Request[Body] => In



interpreter: Repo[Out] => Out

andThen

result: Out => Result

```
Request[Body] => Result
```

Dictionary - testing

`es.scalamad.dictionary.test`

- Suite
  - Some scenarios


YOU
SAID
BEGINNERS

Takeaways

- ➔ Remove side effects & delay effects
- ➔ “À la carte” interpreters
- ➔ Take functional programming seriously
- ➔ Beware of impure stuff in Scala, Play and any other framework, library, …
- ➔ Get ready to learn everyday something new


¡ATENCIÓN!
PREGUNTA...