FUTURES & PROMISES

in Scala 2.10

PHILIPP HALLER

FREDRIK EKHOLDT

with HEATHER MILLER

Typesafe

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Agenda

FUTURES/PROMISES

EXECUTION CTXS

FUTURES IN PLAY

scala.concurrent.

FUTURE & PROMISE

FIRST, SOME *Motivation*

1

Several important libraries have their own future/promise implementation

1

Several important libraries have their own Future/Promise implementation

java.util.concurrent.Future
scala.actors.Future
com.twitter.util.Future

akka.dispatch.Future
scalaz.concurrent.Promise
net.liftweb.actor.LAFuture

This makes it clear that…

THIS MAKES IT CLEAR THAT...

FUTURES ARE AN IMPORTANT, POWERFUL ABSTRACTION

THIS MAKES IT CLEAR THAT...

→ FUTURES ARE AN IMPORTANT, POWERFUL ABSTRACTION

→ THERE'S FRAGMENTATION IN THE SCALA ECOSYSTEM

no hope of interop!

Furthermore…

Furthermore...

2 Java futures neither efficient nor composable

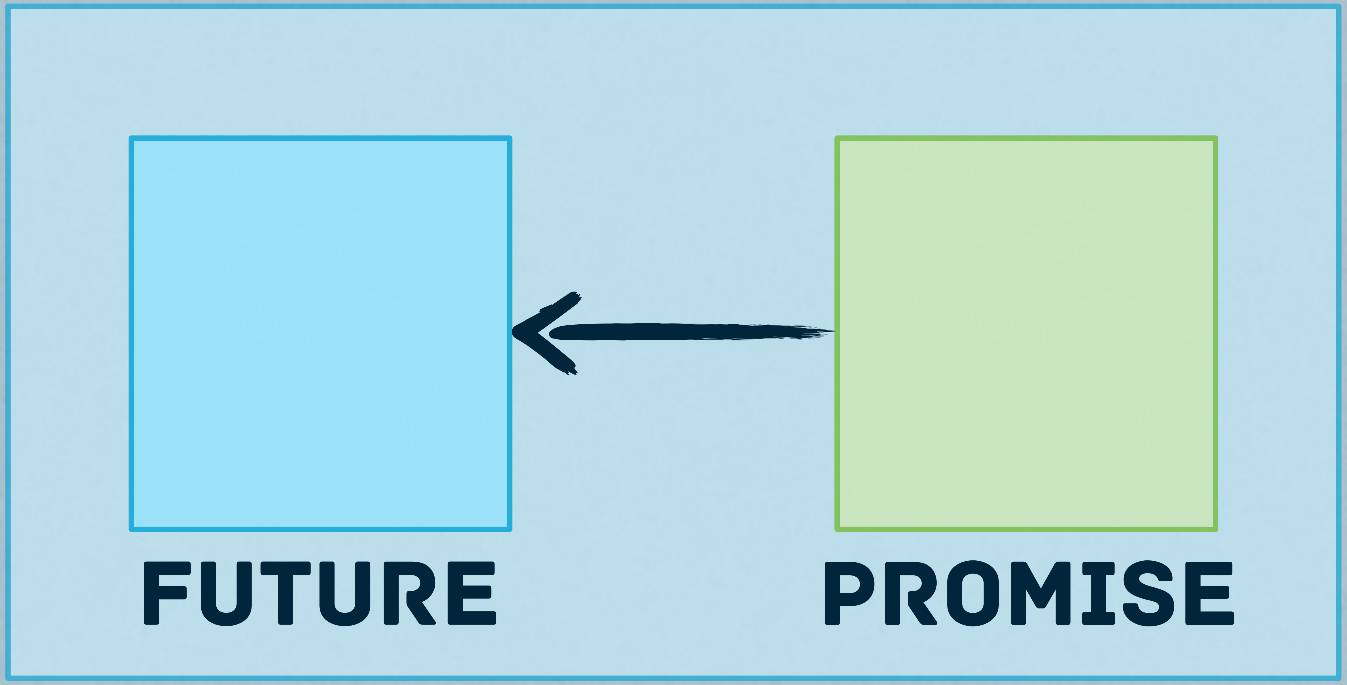
Furthermore...

2. Java futures neither efficient nor composable

3. We could make futures more powerful, by taking advantage of Scala's features

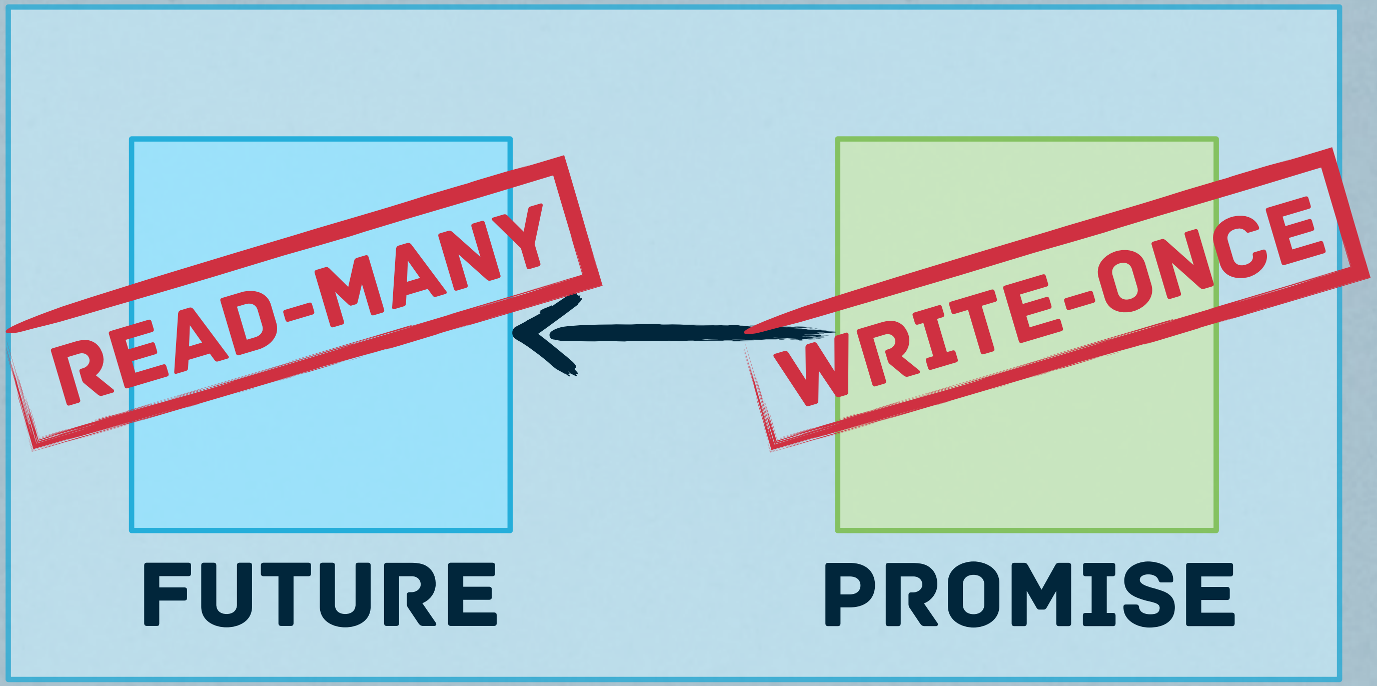
Futures&Promises

Can be thought of as a combined concurrency abstraction



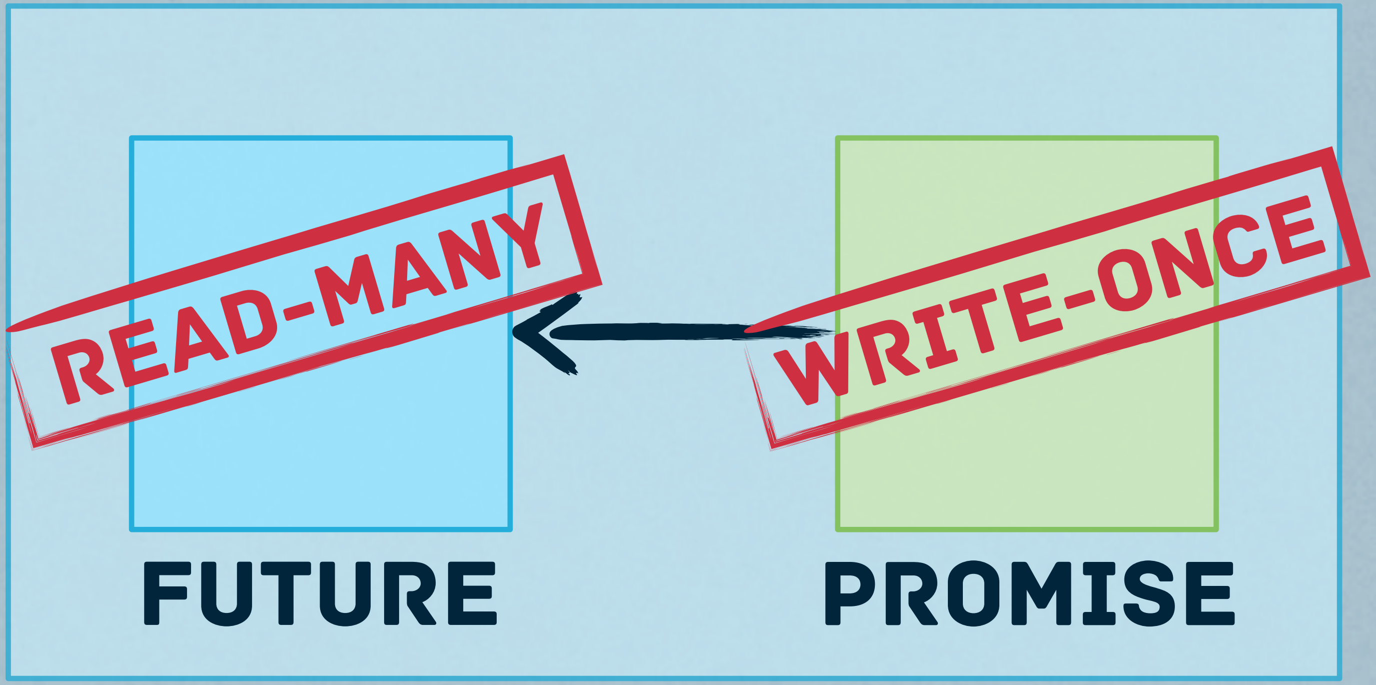
Futures&Promises

can be thought of as a combined concurrency abstraction



Futures&Promises

Can be thought of as a combined concurrency abstraction



Important Ops

- ✓ Start async computation
- ✓ Wait for result
- ✓ Assign result value
- ✓ Obtain associated future object

Success& Failure

A PROMISE `p` **OF TYPE** `Promise[T]` **CAN BE COMPLETED IN TWO WAYS...**

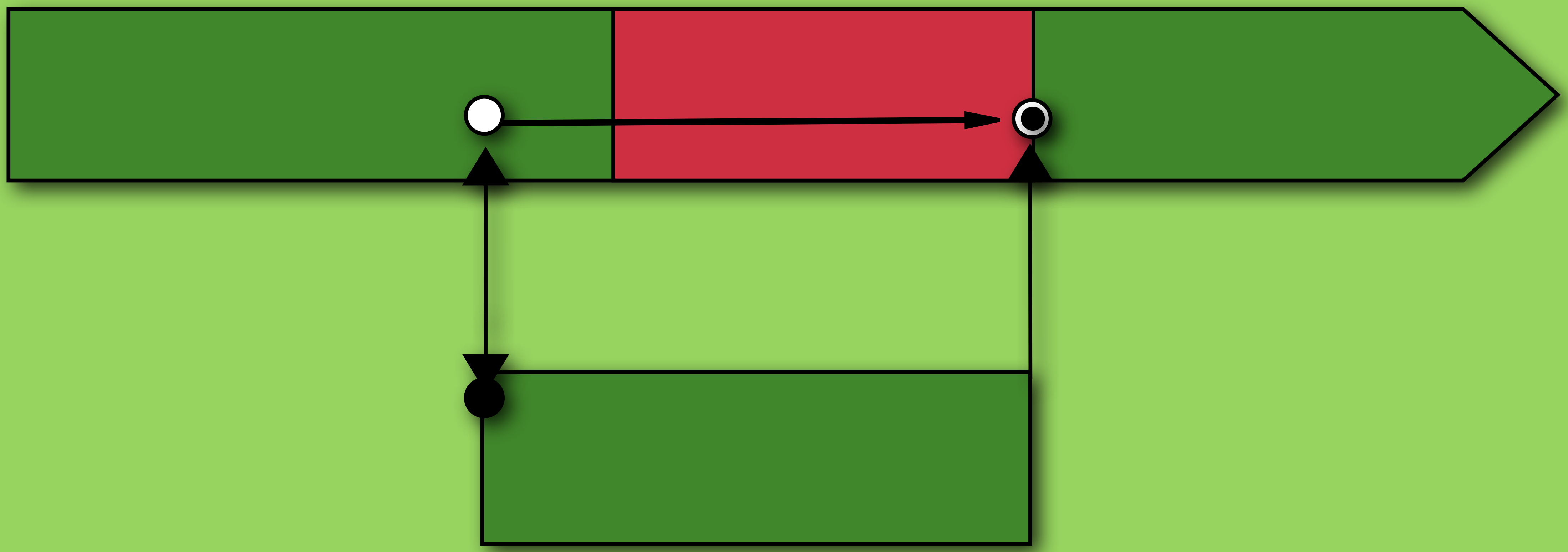
Success

```
val result: T = ...
p.success(result)
```

Failure

```
val exc = new Exception(“something went wrong”)
p.failure(exc)
```

java.util.concurrent.FUTURE

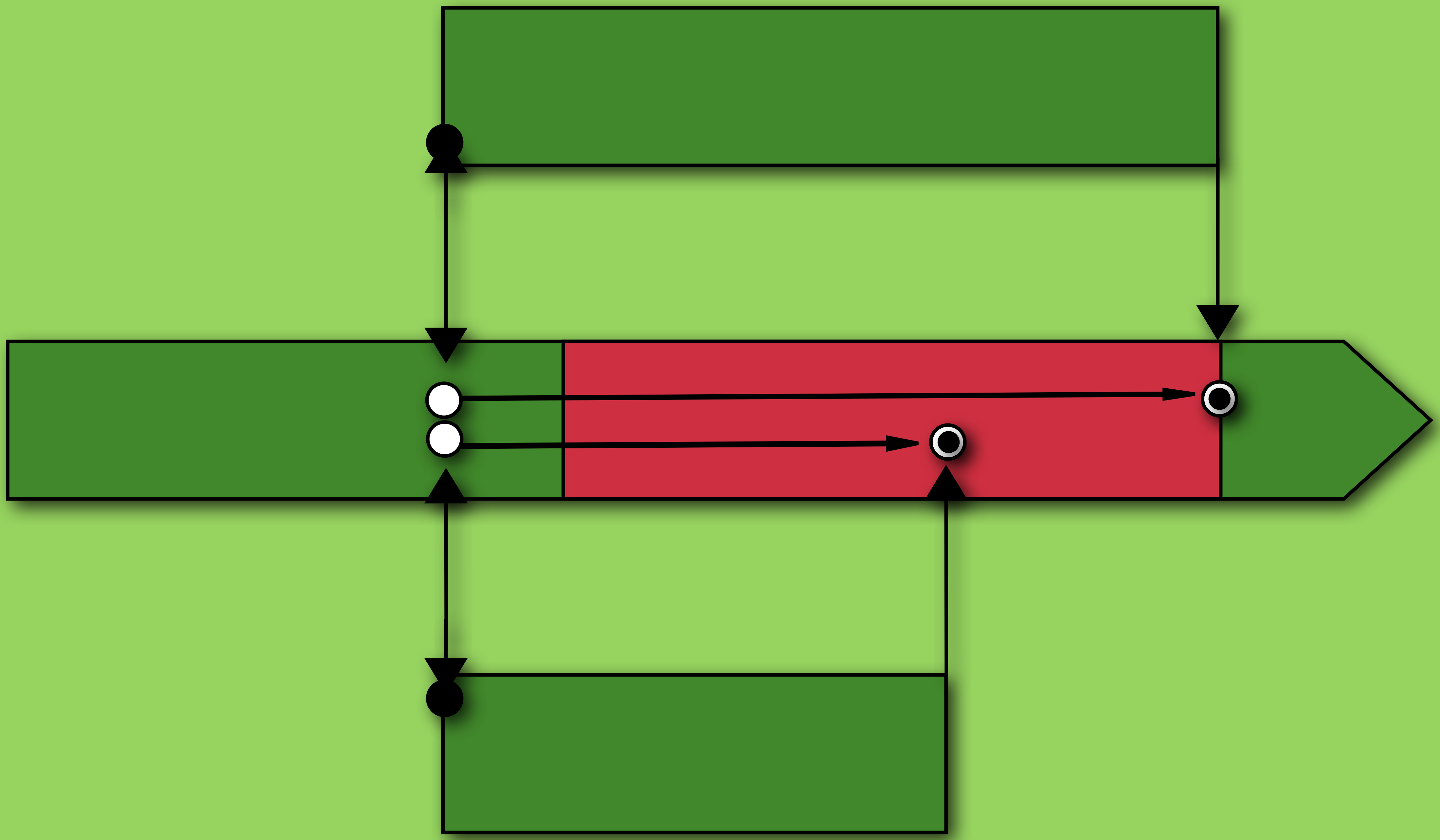


- ○ FUTURE
- ● PROMISE
- ◉ FUTURE WITH VALUE

Green meaningful work

Red thread waiting on the result of another thread

java.util.concurrent.FUTURE

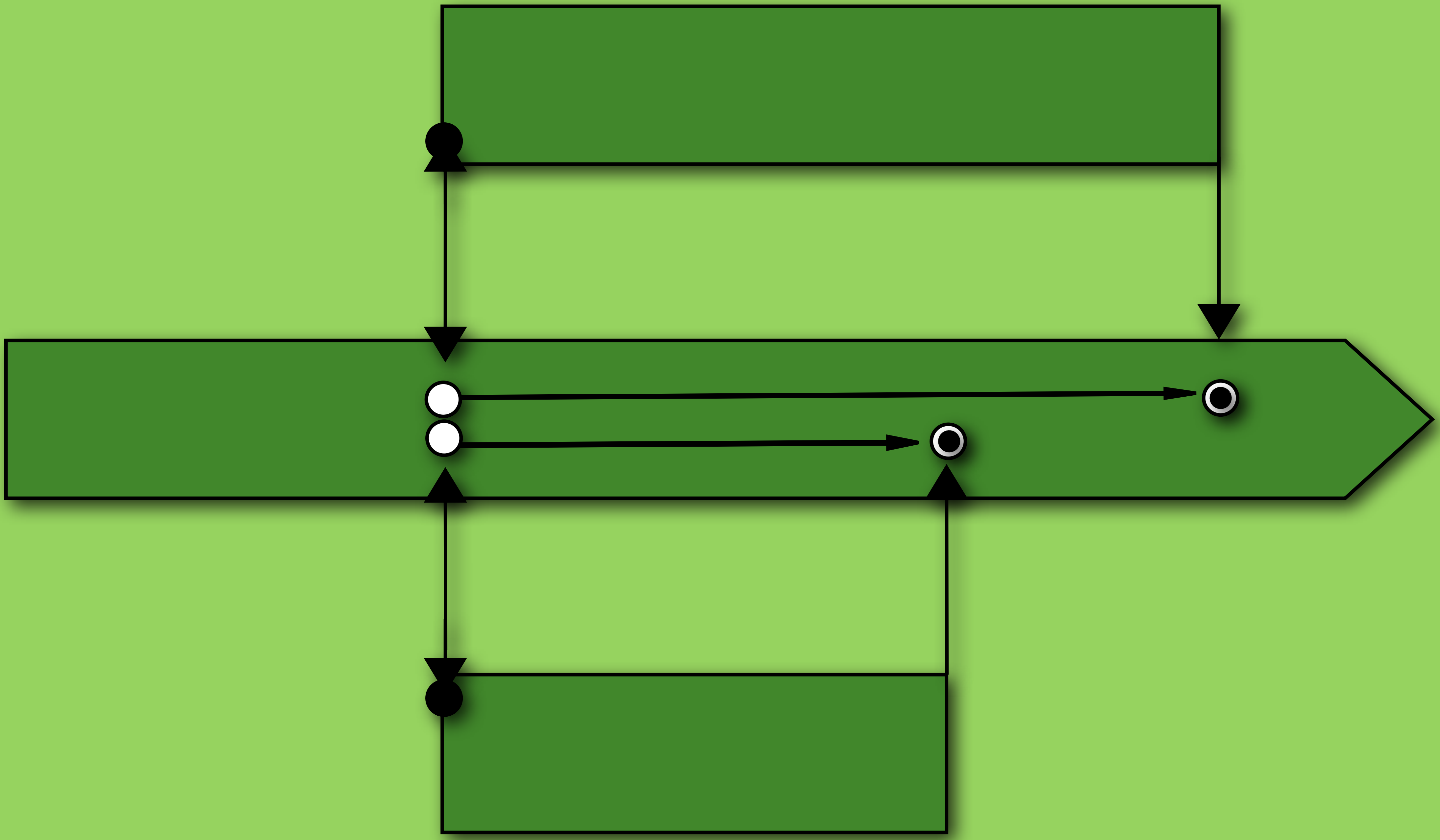


- ○ FUTURE
- ● PROMISE
- ◉ FUTURE WITH VALUE

Green meaningful work

Red thread waiting on the result of another thread

what we'd like to do instead



- ○ FUTURE
- ● PROMISE
- ◉ FUTURE WITH VALUE

Green meaningful work

Red thread waiting on the result of another thread

Async&NonBlocking

Async&NonBlocking

GOAL: Do not block current thread while waiting for result of future

Async&NonBlocking

GOAL: *Do not block current thread while waiting for result of future*

Callbacks

- **REGISTER CALLBACK** *which is invoked (asynchronously) when future is completed*
- **ASYNC COMPUTATIONS NEVER BLOCK** *(except for managed blocking)*

Async&NonBlocking

GOAL: *Do not block current thread while waiting for result of future*

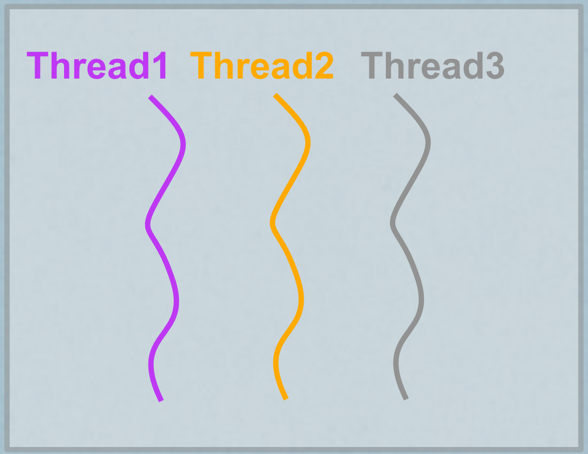
Callbacks

- **REGISTER CALLBACK** *which is invoked (asynchronously) when future is completed*
- **ASYNC COMPUTATIONS NEVER BLOCK** *(except for managed blocking)*

USER DOESN'T HAVE TO EXPLICITLY MANAGE CALLBACKS. HIGHER-ORDER FUNCTIONS INSTEAD!

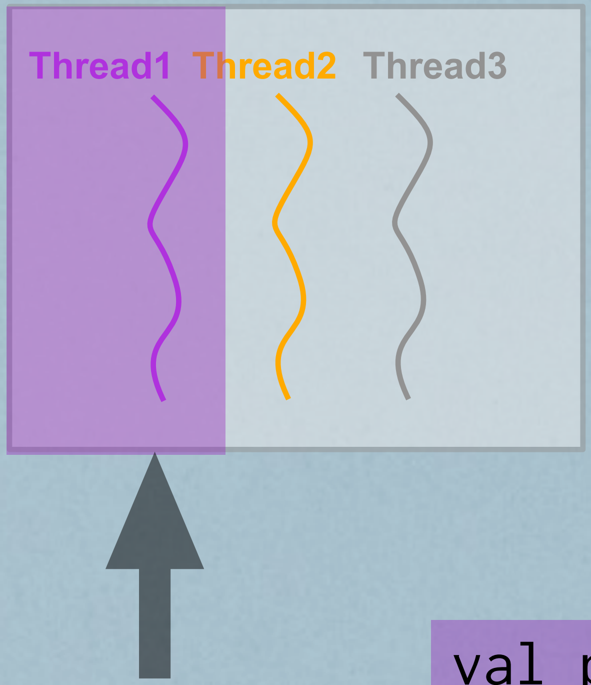
Futures&Promises

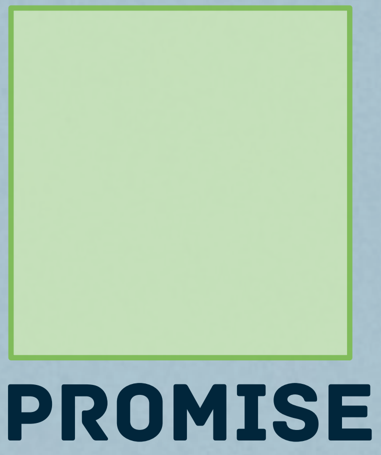
EXAMPLE



Futures&Promises

EXAMPLE



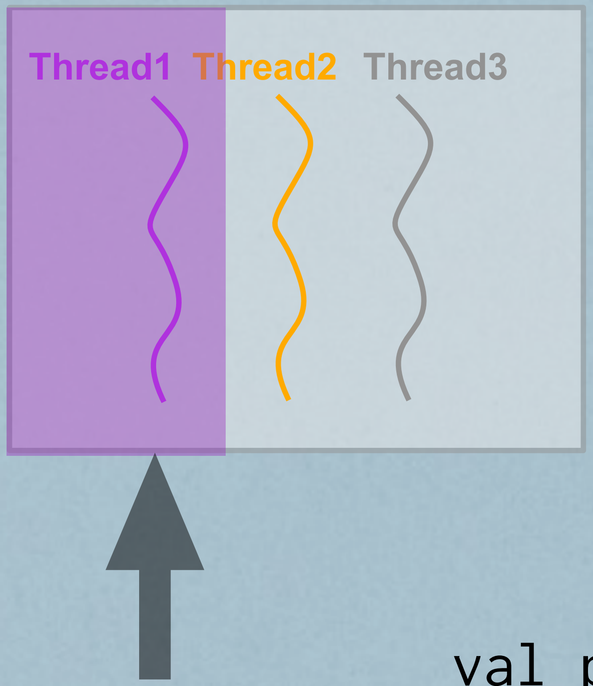


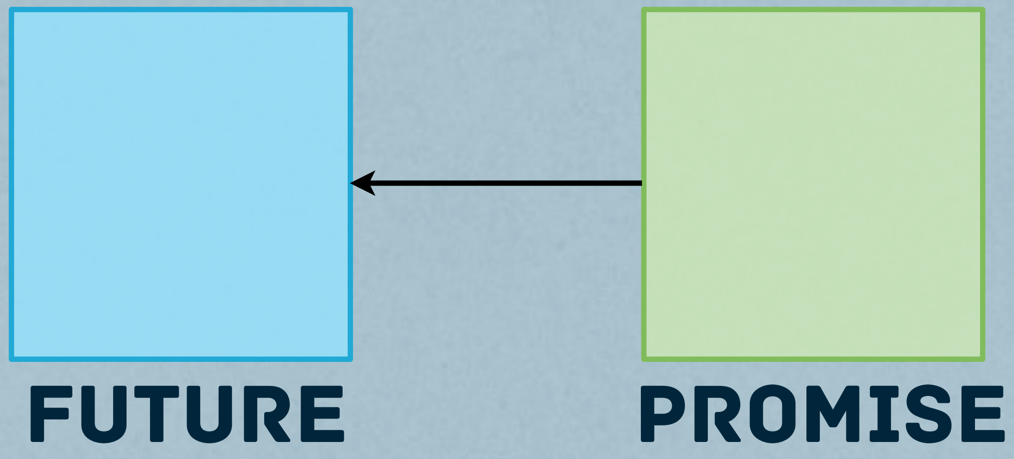
```
val p = Promise[Int]() // Thread 1
```

(CREATE PROMISE)

Futures&Promises

EXAMPLE




FUTURE

PROMISE

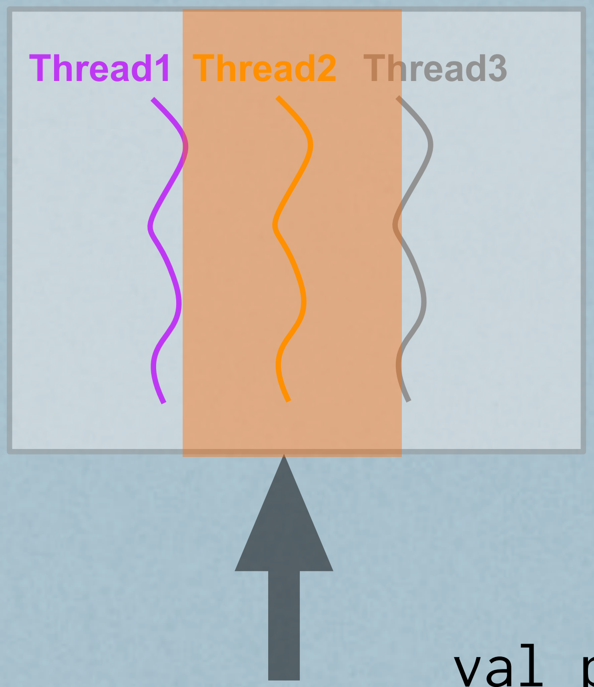
```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
```

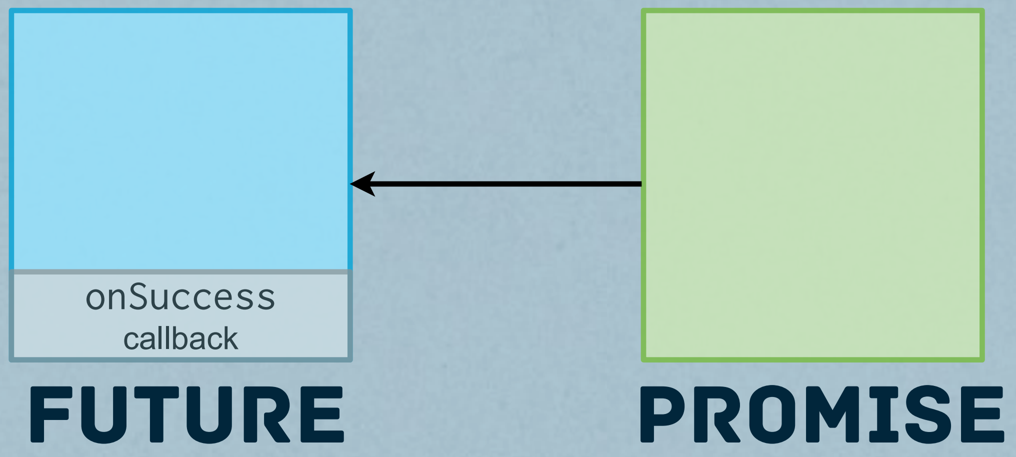
(CREATE PROMISE)

(GET REFERENCE TO FUTURE)

Futures&Promises

EXAMPLE




onSuccess
callback

FUTURE

PROMISE

```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
f onSuccess {           // Thread 2
  case x: Int => println(“Successful!”)
}
```

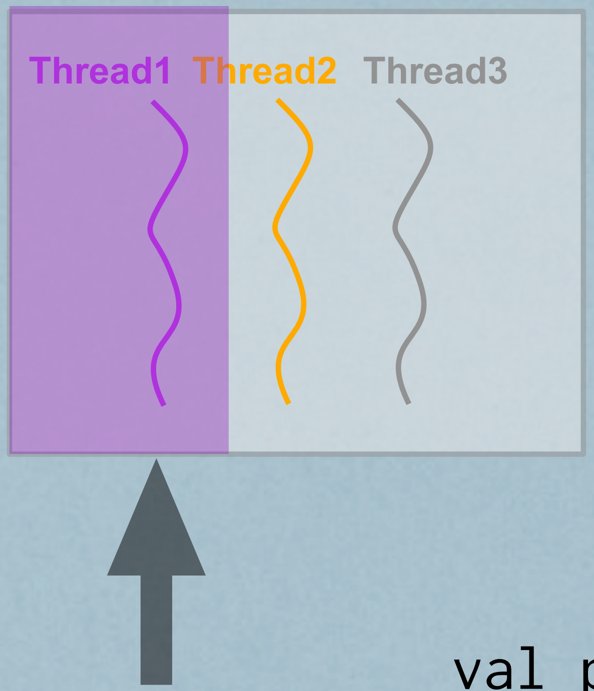
(CREATE PROMISE)

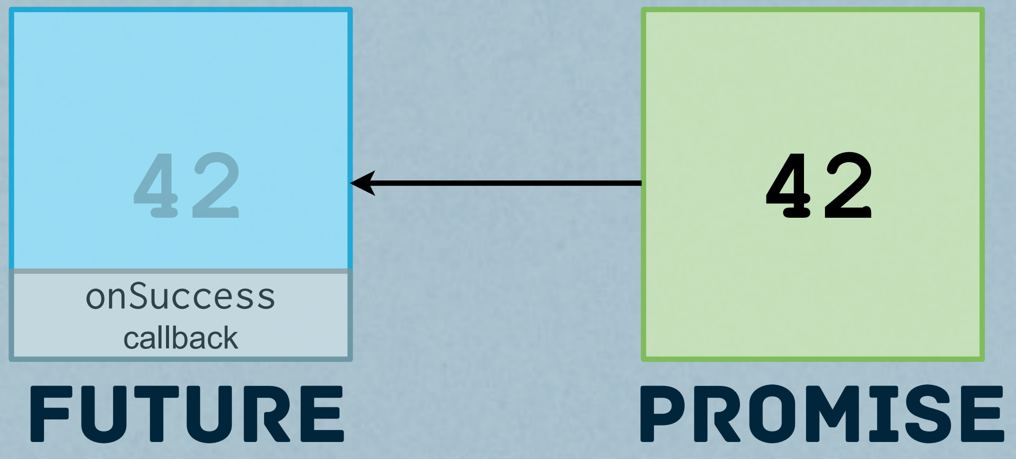
(GET REFERENCE TO FUTURE)

(REGISTER CALLBACK)

Futures&Promises

EXAMPLE





```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
f onSuccess {           // Thread 2
  case x: Int => println(“Successful!”)
}
p.success(42)          // Thread 1
```

(CREATE PROMISE)

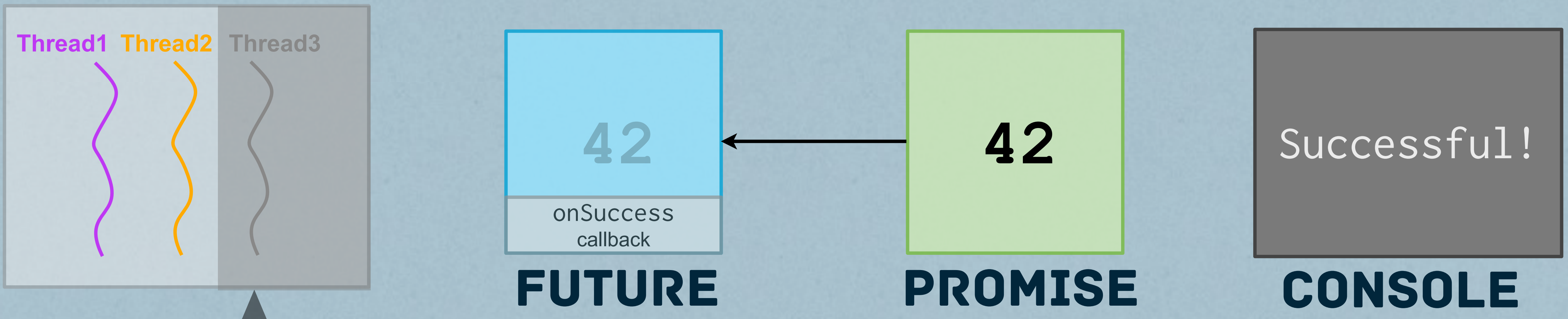
(GET REFERENCE TO FUTURE)

(REGISTER CALLBACK)

(WRITE TO PROMISE)

Futures&Promises

EXAMPLE



```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
f onSuccess {           // Thread 2
  case x: Int => println("Successful!")
}                      // Thread
p.success(42)          // Thread 1
```

(CREATE PROMISE)

(GET REFERENCE TO FUTURE)

(REGISTER CALLBACK)

(EXECUTE CALLBACK)

(WRITE TO PROMISE)

NOTE: onSuccess **CALLBACK EXECUTED EVEN IF** f **HAS ALREADY BEEN COMPLETED AT TIME OF REGISTRATION**

Combinators

- Composability thru higher-order funcs
- Standard monadic combinators

```
def map[S](f: T => S): Future[S]
```

```
val purchase: Future[Int] = rateQuote map {
  quote => connection.buy(amount, quote)
}
```

```
def filter(pred: T => Boolean): Future[T]
```

```
val postBySmith: Future[Post] =
   post.filter(_.author == “Smith”)
```

Combinators

- COMPOSABILITY THRU HIGHER-ORDER FUNCS
- STANDARD MONADIC COMBINATORS

```
def map[S](f: T => S): Future[S]
```

```
val purchase: Future[Int] = rateQuote map {
  quote => connection.buy(amount, quote)
}
```

IF MAP FAILS: purchase is completed with unhandled exception

```
def filter(pred: T => Boolean): Future[T]
```

```
val postBySmith: Future[Post] =
   post.filter(_.author == “Smith”)
```

IF FILTER FAILS: postBySmith completed with NoSuchElementException

Combinators

ADDITIONAL FUTURE-SPECIFIC HIGHER-ORDER FUNCTIONS HAVE BEEN INTRODUCED

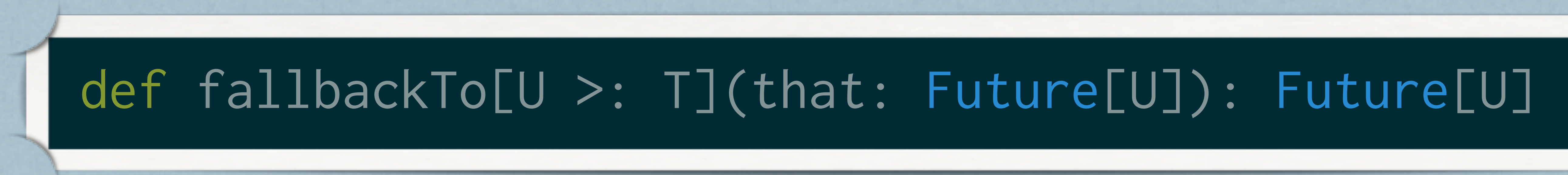
```scala
def fallbackTo[U >: T](that: Future[U]): Future[U]
```

```scala
val fut: Future[T] = Future.firstCompletedOf[T](futures)
```

```scala
def andThen(pf: PartialFunction[...]): Future[T]
```

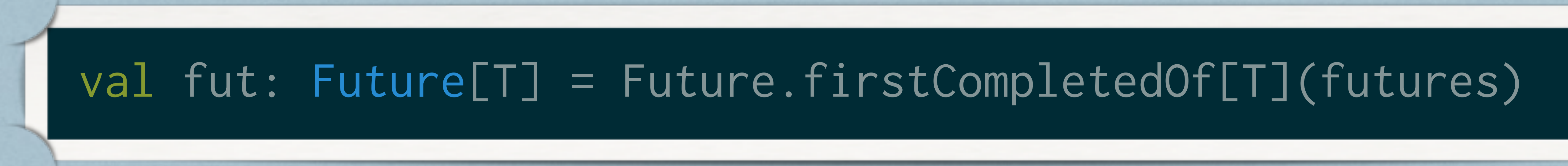
Combinators

ADDITIONAL FUTURE-SPECIFIC HIGHER-ORDER FUNCTIONS HAVE BEEN INTRODUCED



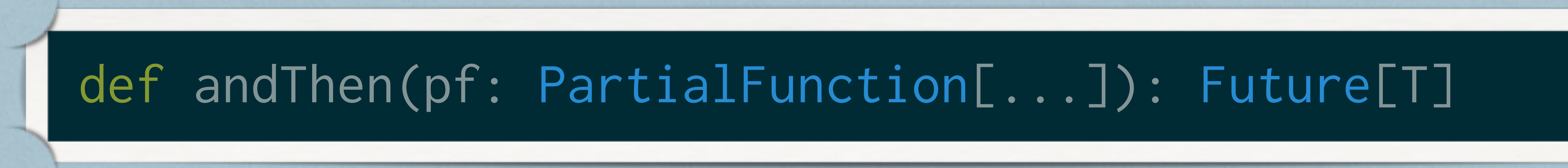
```
def fallbackTo[U >: T](that: Future[U]): Future[U]
```

"falls back" to `that` future in case of failure



```
val fut: Future[T] = Future.firstCompletedOf[T](futures)
```

returns a future completed with result of first completed future



```
def andThen(pf: PartialFunction[...]): Future[T]
```

allows one to define a sequential execution over a chain of futures

scala.concurrent. EXECUTION CONTEXT

Threadpools...

ARE NEEDED BY:

- **FUTURES** for executing callbacks and function arguments
- **ACTORS** for executing message handlers, scheduled tasks, etc.
- **PARALLEL COLLECTIONS** for executing data-parallel operations

Scala 2.10 introduces

EXECUTION CONTEXTS

Scala 2.10 introduces

EXECUTION CONTEXTS

Goal

PROVIDE GLOBAL THREADPOOL AS PLATFORM SERVICE TO BE SHARED BY ALL PARALLEL FRAMEWORKS

Scala 2.10 introduces

EXECUTION CONTEXTS

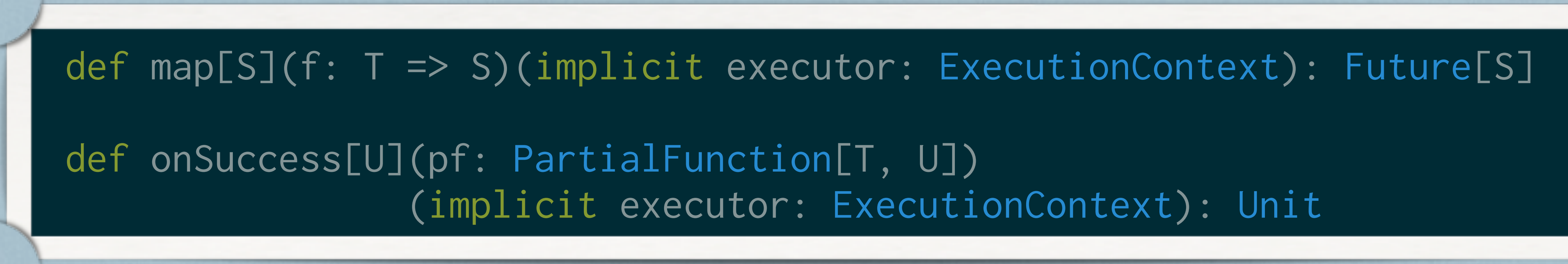
Goal

PROVIDE GLOBAL THREADPOOL AS PLATFORM SERVICE TO BE SHARED BY ALL PARALLEL FRAMEWORKS

- `scala.concurrent` *package provides global* `ExecutionContext`
- Default `ExecutionContext` backed by the most recent fork join pool (collaboration with Doug Lea, SUNY Oswego)

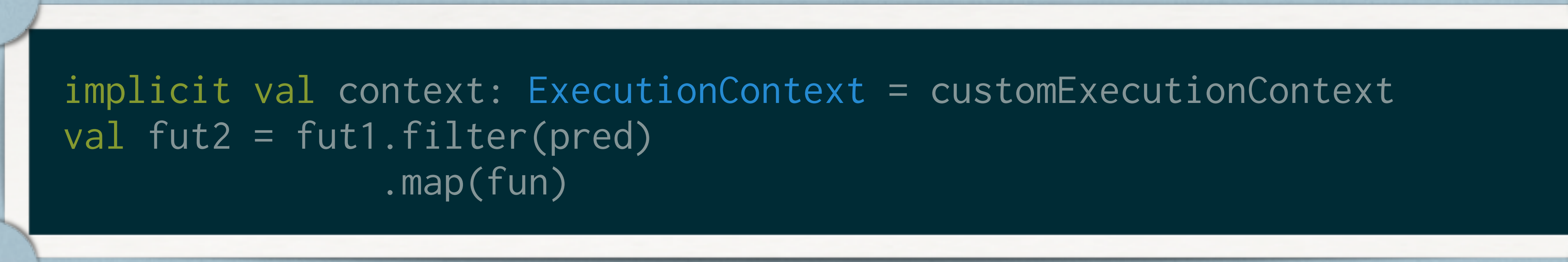
Implicit Execution Ctxs

Asynchronous computations are executed on an `ExecutionContext` which is provided implicitly.



```scala
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]

def onSuccess[U](pf: PartialFunction[T, U])
                (implicit executor: ExecutionContext): Unit
```

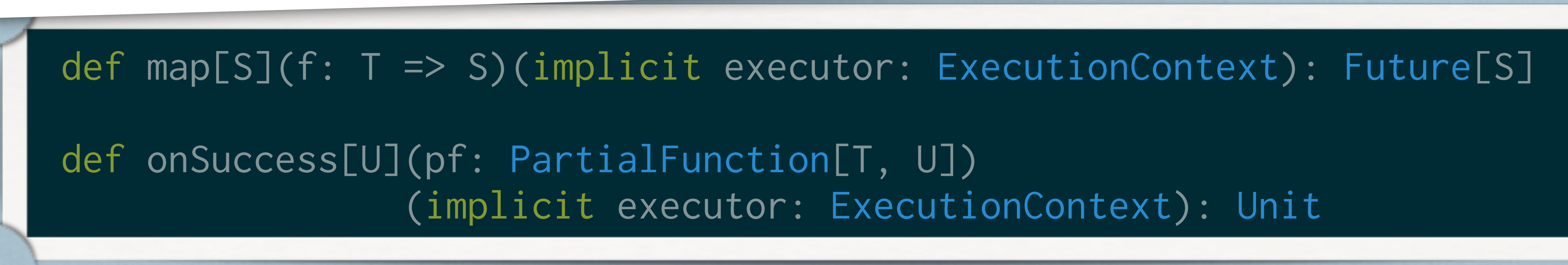
Implicit parameters enable fine-grained selection of the `ExecutionContext`:



```scala
implicit val context: ExecutionContext = customExecutionContext
val fut2 = fut1.filter(pred)
               .map(fun)
```

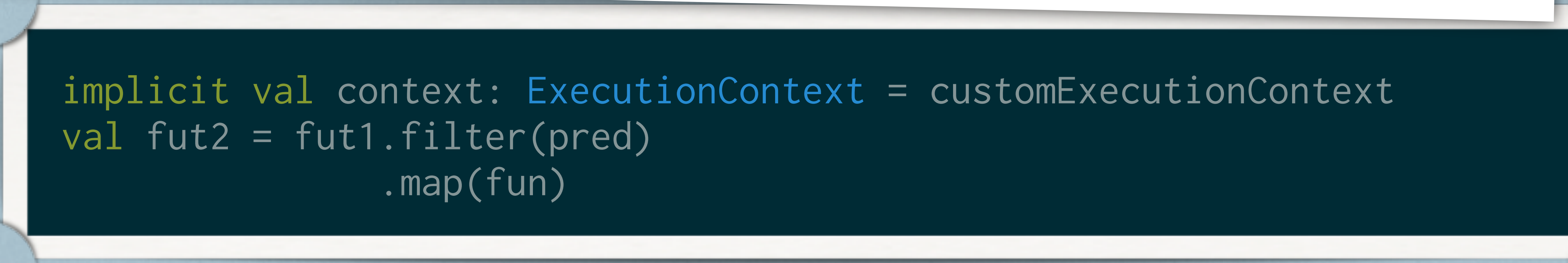
Implicit Execution Ctxs

IMPLICIT ExecutionContexts **ALLOW SHARING ECS BETWEEN FRAMEWORKS**



```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]

def onSuccess[U](pf: PartialFunction[T, U])
                (implicit executor: ExecutionContext): Unit
```

ENABLES FLEXIBLE SELECTION OF EXECUTION POLICY



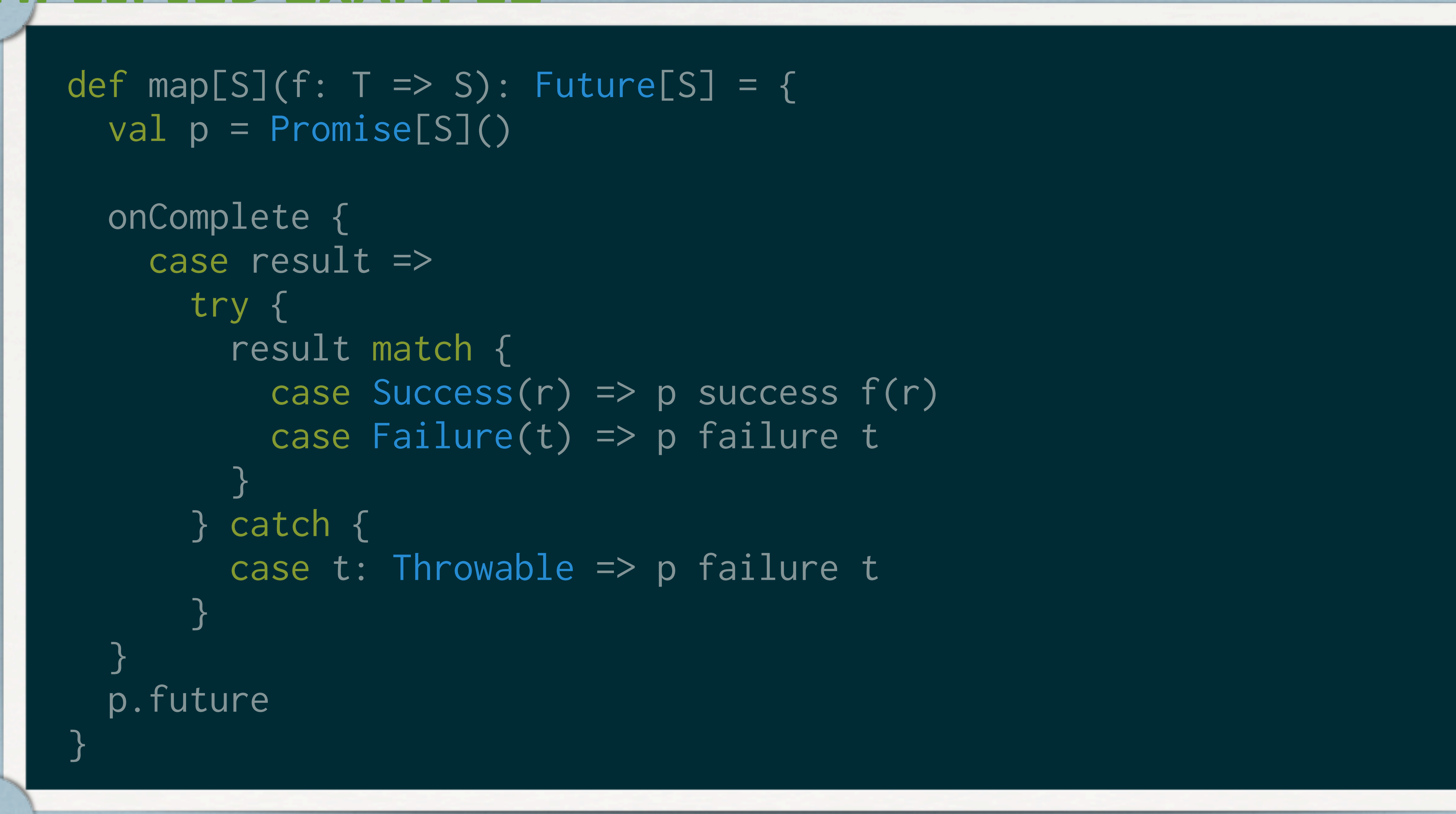
```
implicit val context: ExecutionContext = customExecutionContext
val fut2 = fut1.filter(pred)
               .map(fun)
```

Future

THE IMPLEMENTATION

Many operations implemented in terms of promises

SIMPLIFIED EXAMPLE

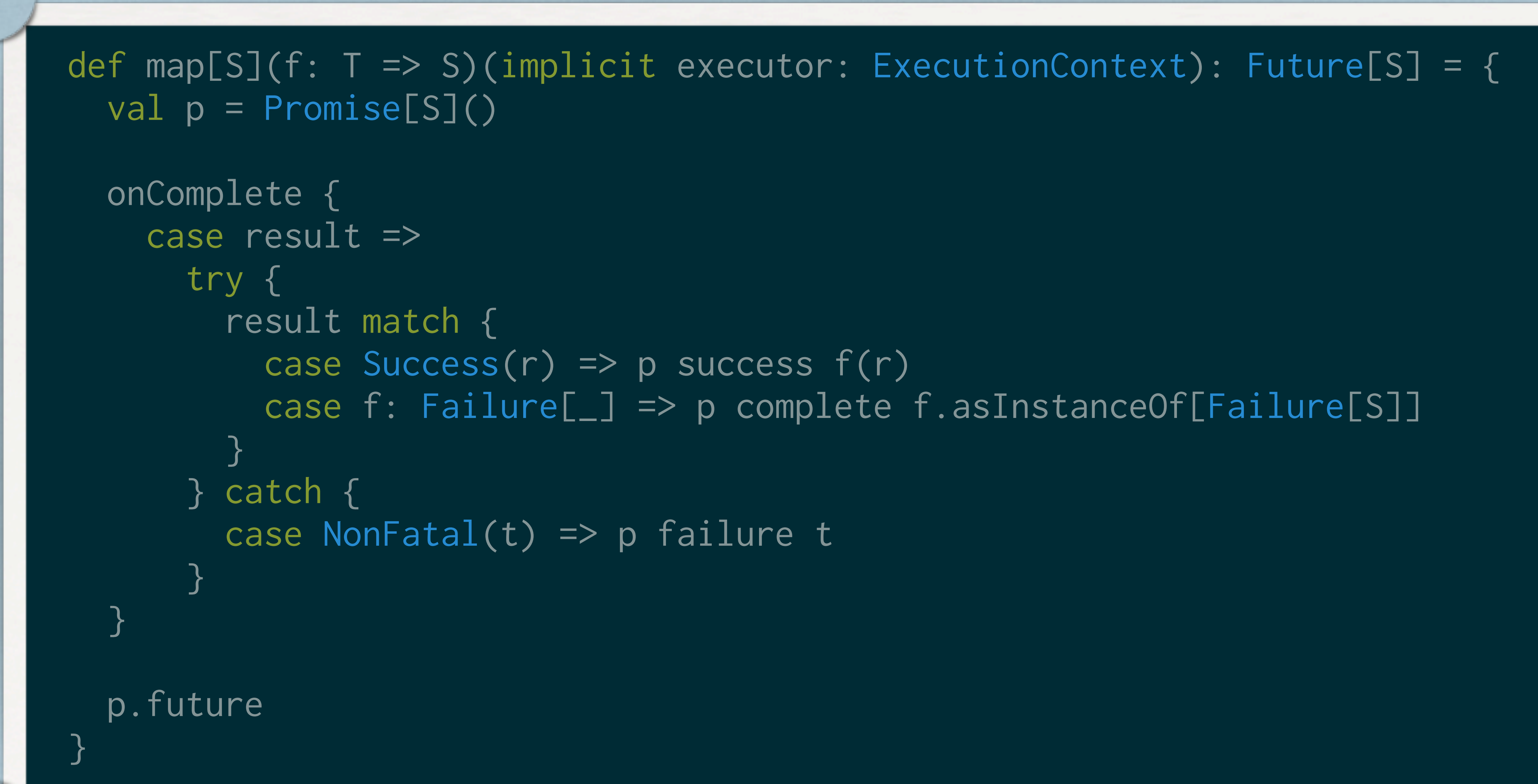


```scala
def map[S](f: T => S): Future[S] = {
  val p = Promise[S]()

  onComplete {
    case result =>
      try {
        result match {
          case Success(r) => p success f(r)
          case Failure(t) => p failure t
        }
      } catch {
        case t: Throwable => p failure t
      }
  }
  p.future
}
```

Future

THE REAL IMPLEMENTATION

The real implementation (a) adds an implicit ExecutionContext, (b) avoids extra object creations, and (c) catches only non-fatal exceptions:



```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S] = {
  val p = Promise[S]()

  onComplete {
    case result =>
      try {
        result match {
          case Success(r) => p success f(r)
          case f: Failure[_] => p complete f.asInstanceOf[Failure[S]]
        }
      } catch {
        case NonFatal(t) => p failure t
      }
  }

  p.future
}
```

Promise
THE IMPLEMENTATION

Promise *is the work horse of the futures implementation.*

A Promise[T] *can be in one of two states:*

PENDING

No result has been written to the promise.
State represented using a list of callbacks (initially empty).

COMPLETED

The promise has been assigned a successful result or exception.
State represented using an instance of Try[T]

Invoking Promise.complete triggers a transition from state Pending to Completed

A PROMISE CAN BE COMPLETED AT MOST ONCE:

```scala
def complete(result: Try[T]): this.type =
  if (tryComplete(result)) this
  else throw new IllegalStateException("Promise already completed.")
```

Completing a Promise

```
def tryComplete(value: Try[T]): Boolean = {
  val resolved = resolveTry(value)
  (try {
    @tailrec
    def tryComplete(v: Try[T]): List[CallbackRunnable[T]] = {
      getState match {
        case raw: List[_] =>
          val cur = raw.asInstanceOf[List[CallbackRunnable[T]]]
          if (updateState(cur, v)) cur else tryComplete(v)
        case _ => null
      }
    }
    tryComplete(resolved)
  } finally {
    synchronized { notifyAll() } // Notify any blockers
  }) match {
    case null             => false
    case rs if rs.isEmpty => true
    case rs               =>
      rs.foreach(_.executeWithValue(resolved)); true
  }
}
```

THE AWKWARD SQUAD

```
abstract class AbstractPromise {
    private volatile Object _ref;
    final static long _refoffset;

    static {
        try {
            _refoffset =
                Unsafe.instance.objectFieldOffset(
                    AbstractPromise.class.getDeclaredField("_ref"));
        } catch (Throwable t) {
            throw new ExceptionInInitializerError(t);
        }
    }

    protected boolean updateState(Object oldState, Object newState) {
        return
            Unsafe.instance.compareAndSwapObject(this, _refoffset,
                                                 oldState, newState);
    }

    protected final Object getState() {
        return _ref;
    }
}
```

Integrating Futures & Actors

Futures are results of asynchronous message sends

When a response is expected

```
val response: Future[Any] = socialGraph ? getFriends(user)
```

Implementing synchronous send (untyped):

```
def syncSend(to: ActorRef, msg: Any, timeout: Duration): Any = {
  val fut = to ? msg
  Await.result(fut, timeout)
}
```

Fu
wr

Recovering types

```
val friendsFut: Future[Seq[Friend]] = response.mapTo[Seq[Friend]]
```

INTEGRATING Futures & Actors

Futures are results of asynchronous message sends

WHEN A RESPONSE IS EXPECTED

```scala
val response: Future[Any] = socialGraph ? getFriends(user)
```

`friendsFut` **IS EITHER COMPLETED WITH A SUCCESSFUL RESULT OR WITH A WRAPPED EXCEPTION IF RESPONSE TIMES OUT OR IS NOT OF TYPE** `Seq[Friend]`

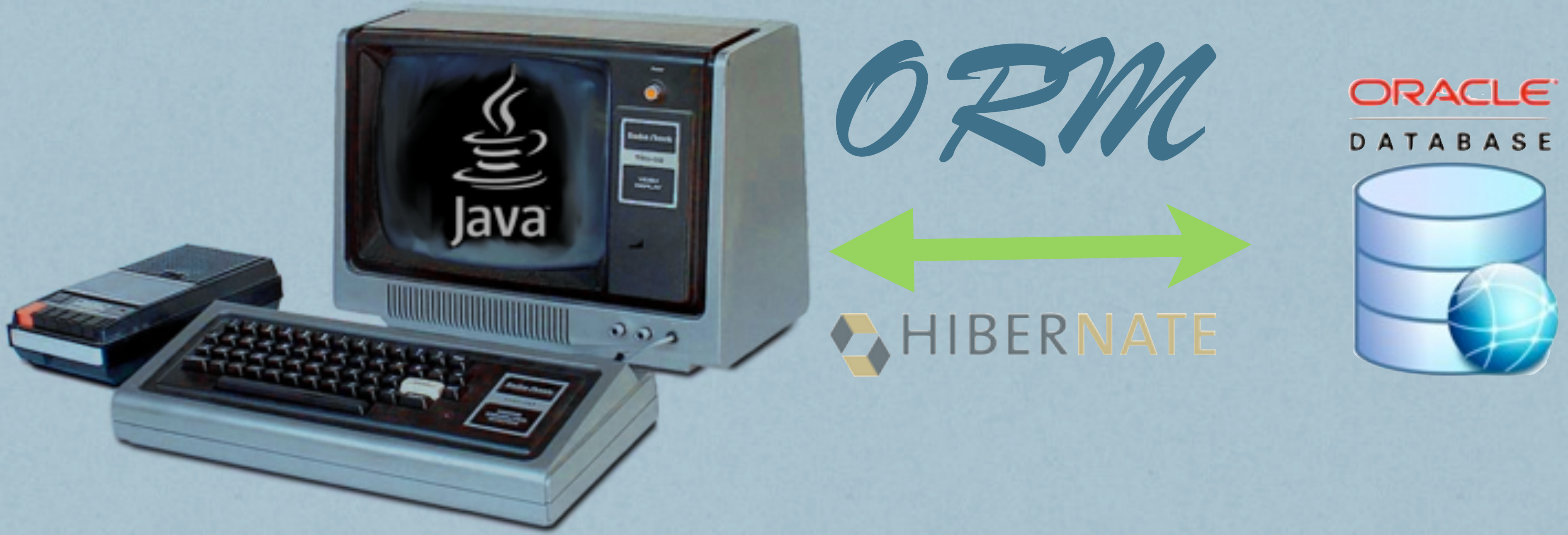
RECOVERING TYPES

```scala
val friendsFut: Future[Seq[Friend]] = response.mapTo[Seq[Friend]]
```

THE PLAY *Example*

Ye Olde Webapp

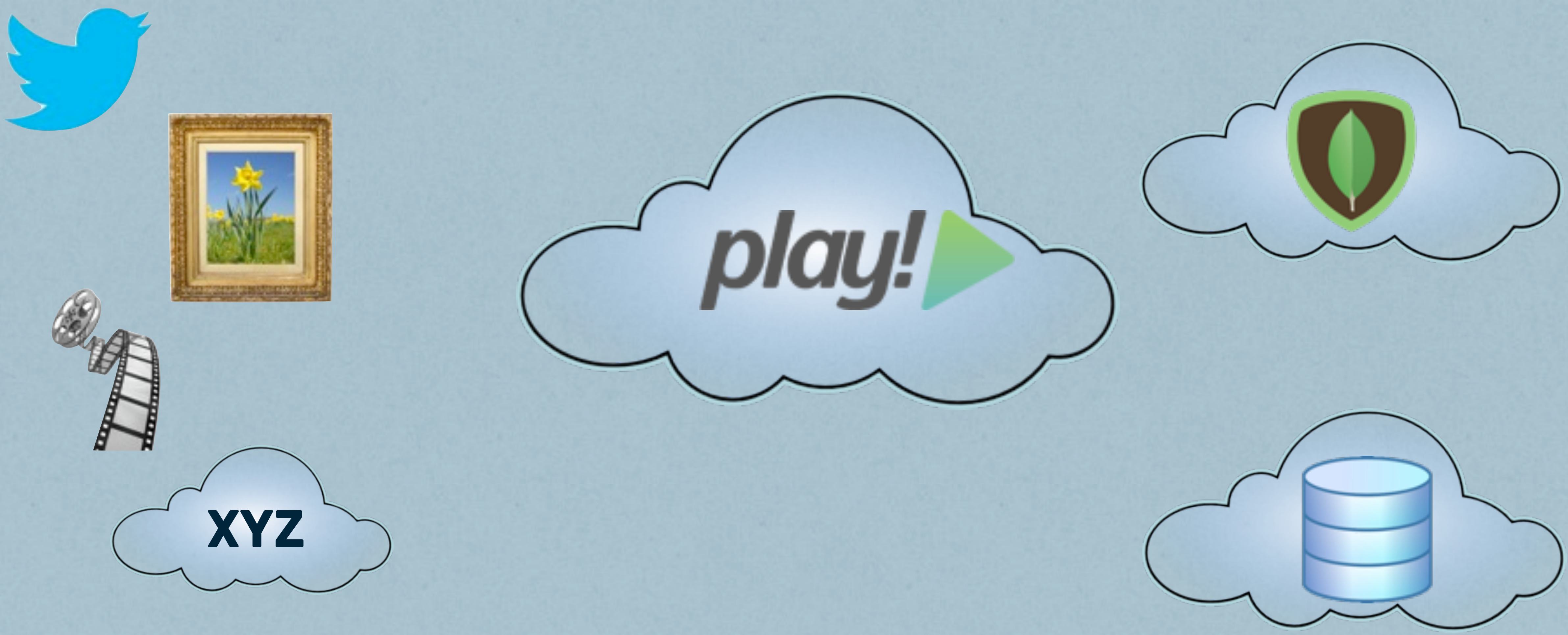
databases



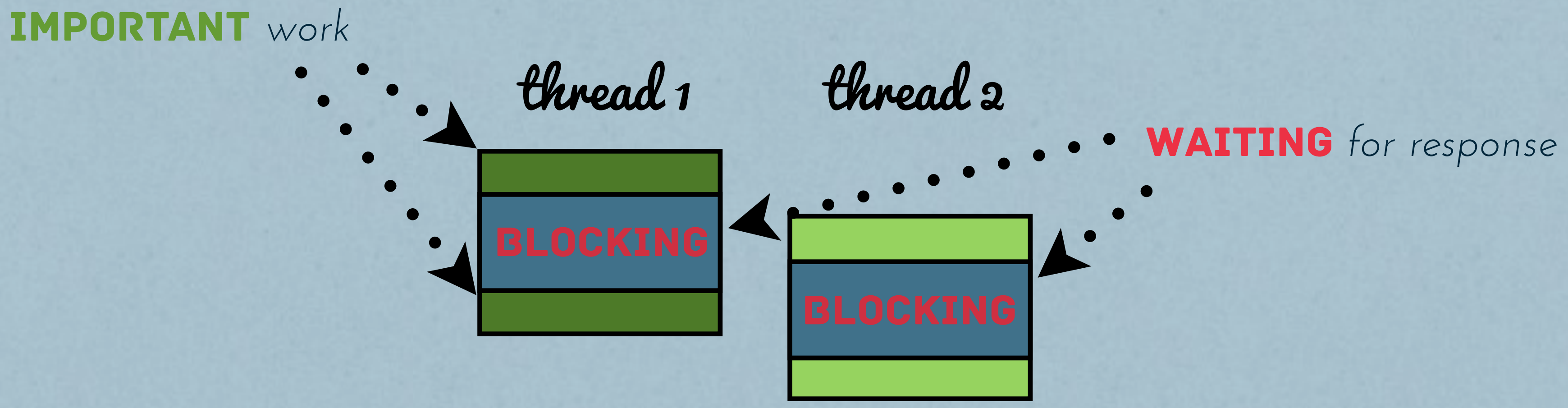
ORM

The Future of webapps is Now

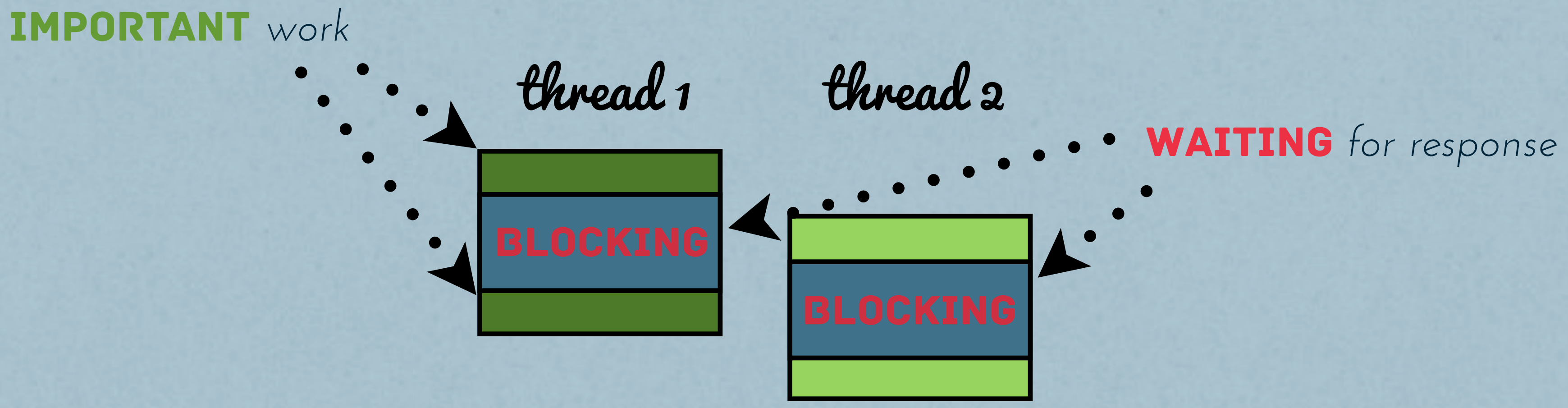
Services



Synchronous IO



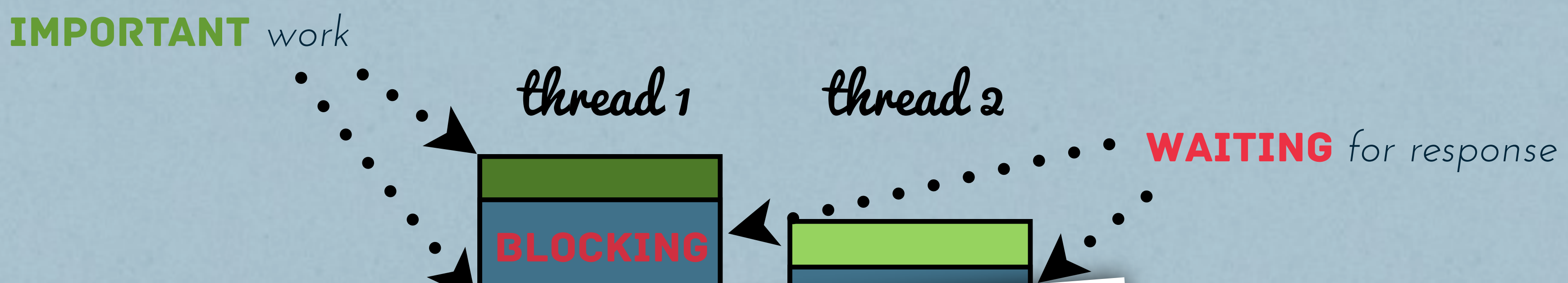
Synchronous IO



MEANS:

N requests == N threads

Synchronous IO



IMPORTANT work

thread 1

thread 2

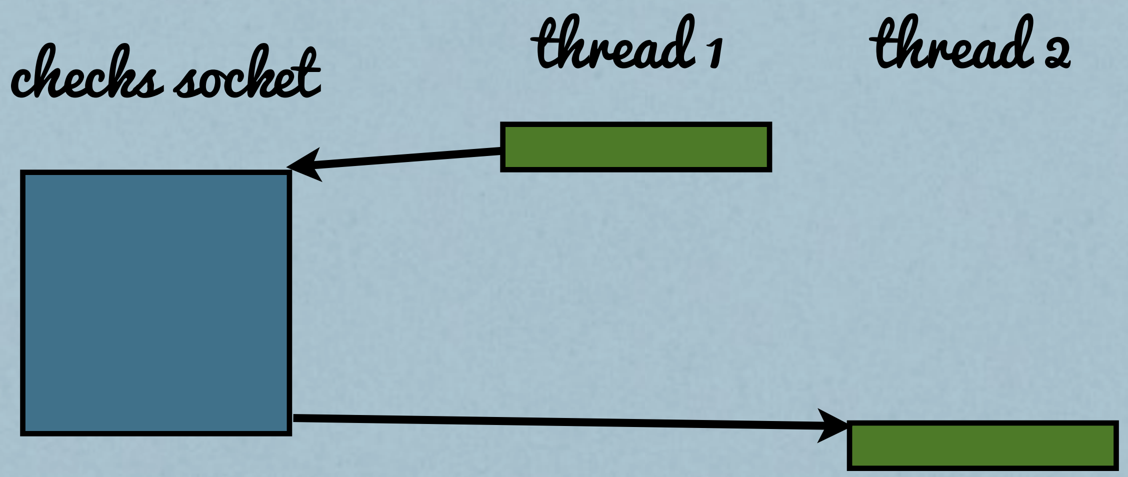
BLOCKING

WAITING for response

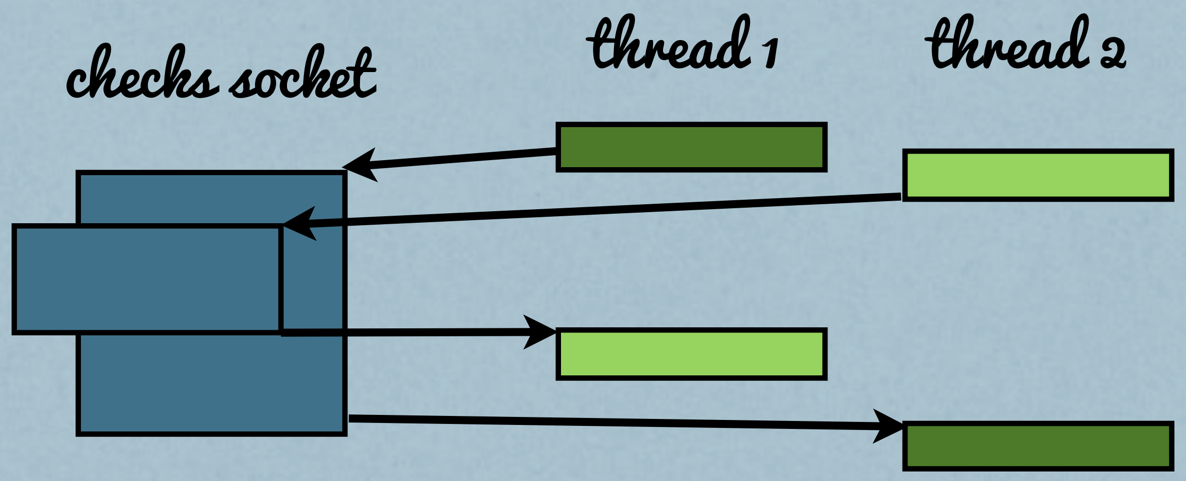
DOES
NOT
SCALE

MEANS: N requests == N threads

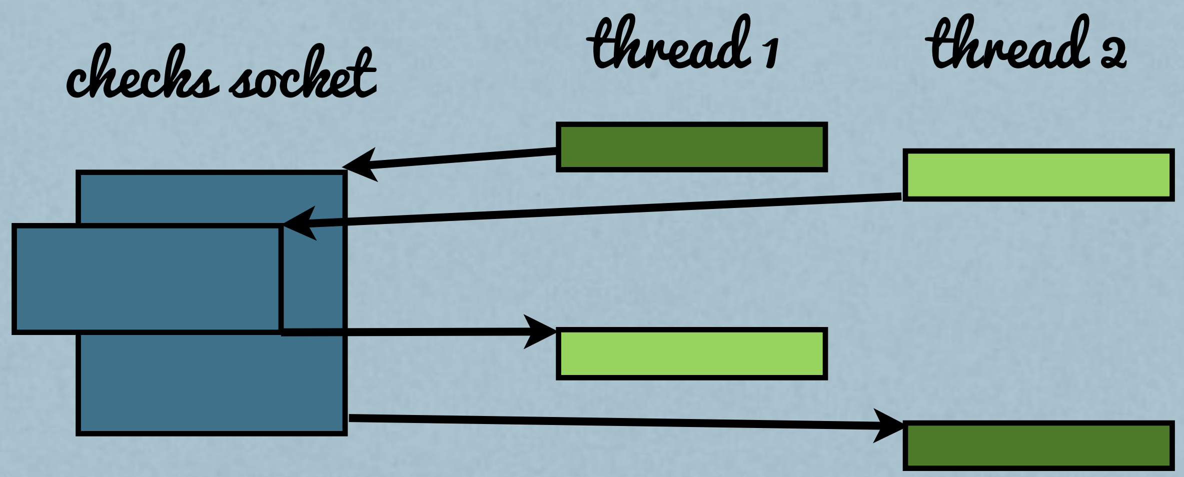
Asynchronous IO



Asynchronous IO



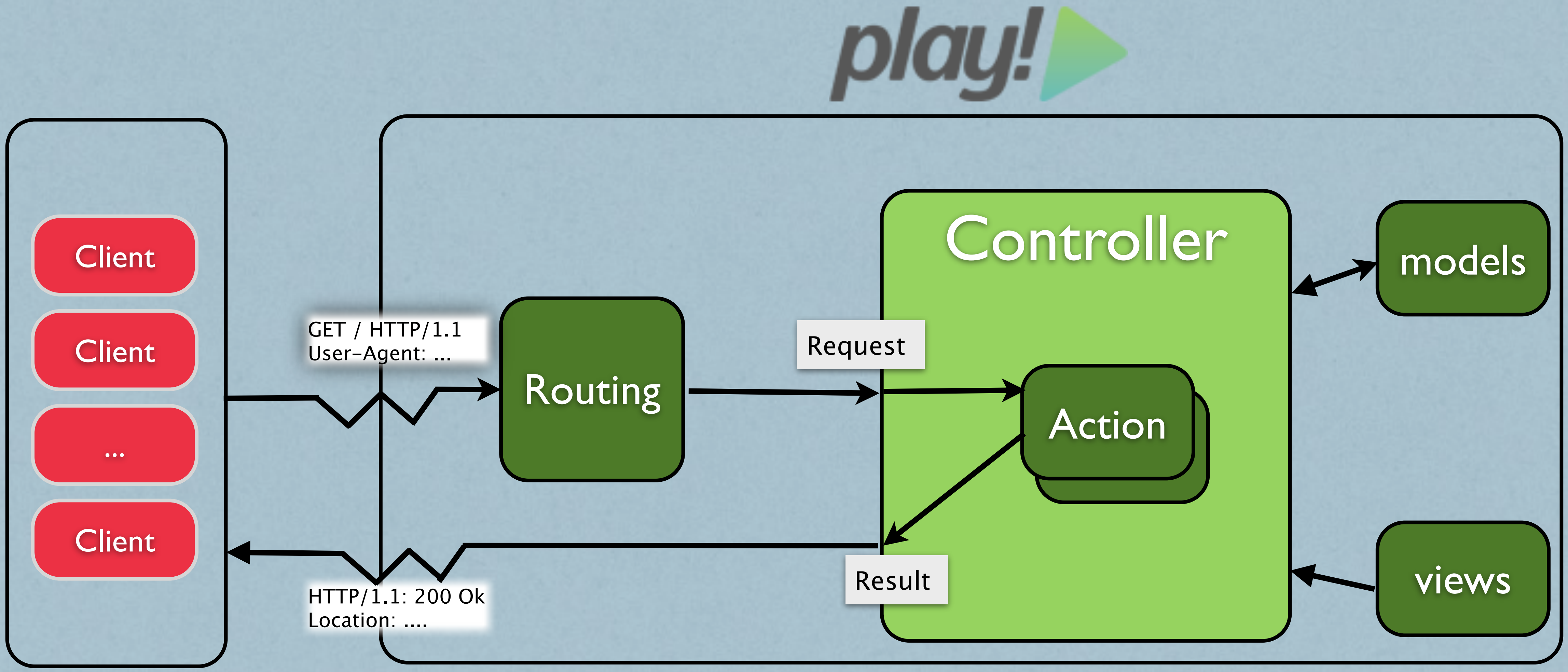
Asynchronous IO



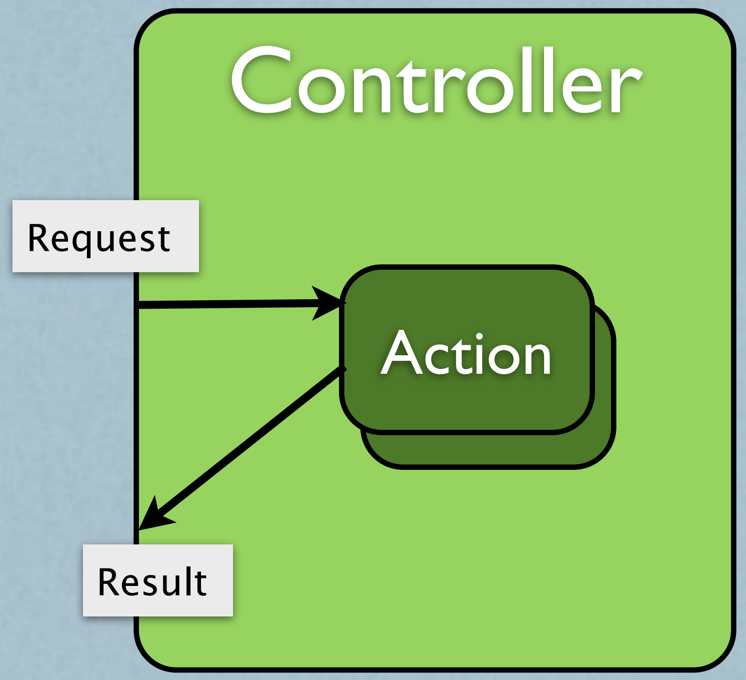
MEANS:

We now scale!

Play 101


play!
Client
Client
...
Client
GET / HTTP/1.1
User-Agent: ...
Routing
Request
Controller
Action
Result
HTTP/1.1: 200 Ok
Location: ....
models
views

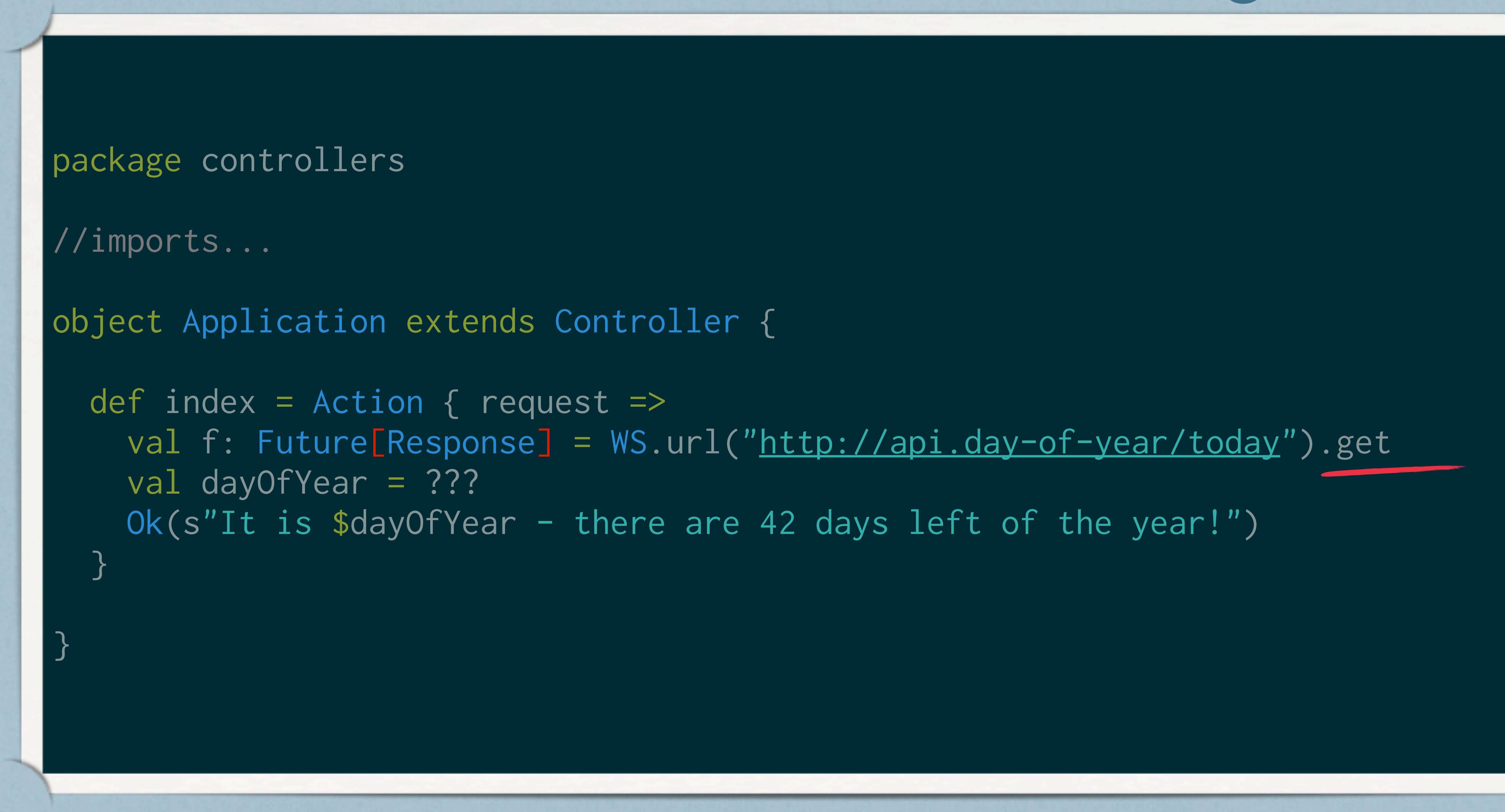
Play 101



ACTIONS IN Play

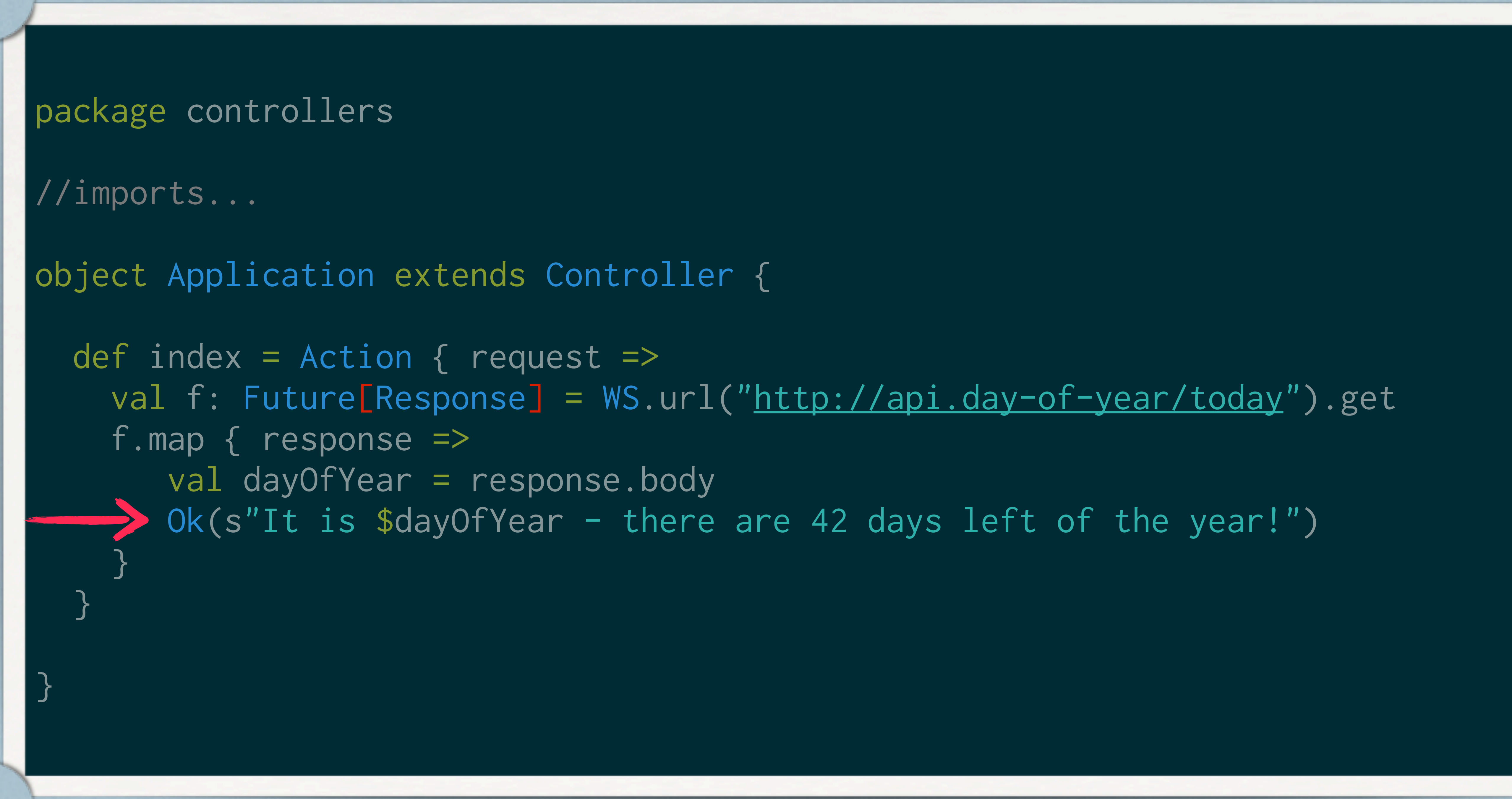
```scala
package controllers

//imports...

object Application extends Controller {

  def index = Action { request =>
    Ok("It is November 19th - there are 42 days left of the year!")
  }

}
```

Simple webservices in Play



```
package controllers

//imports...

object Application extends Controller {

  def index = Action { request =>
    val f: Future[Response] = WS.url("http://api.day-of-year/today").get
    val dayOfYear = ???
    Ok(s"It is $dayOfYear - there are 42 days left of the year!")
  }

}
```

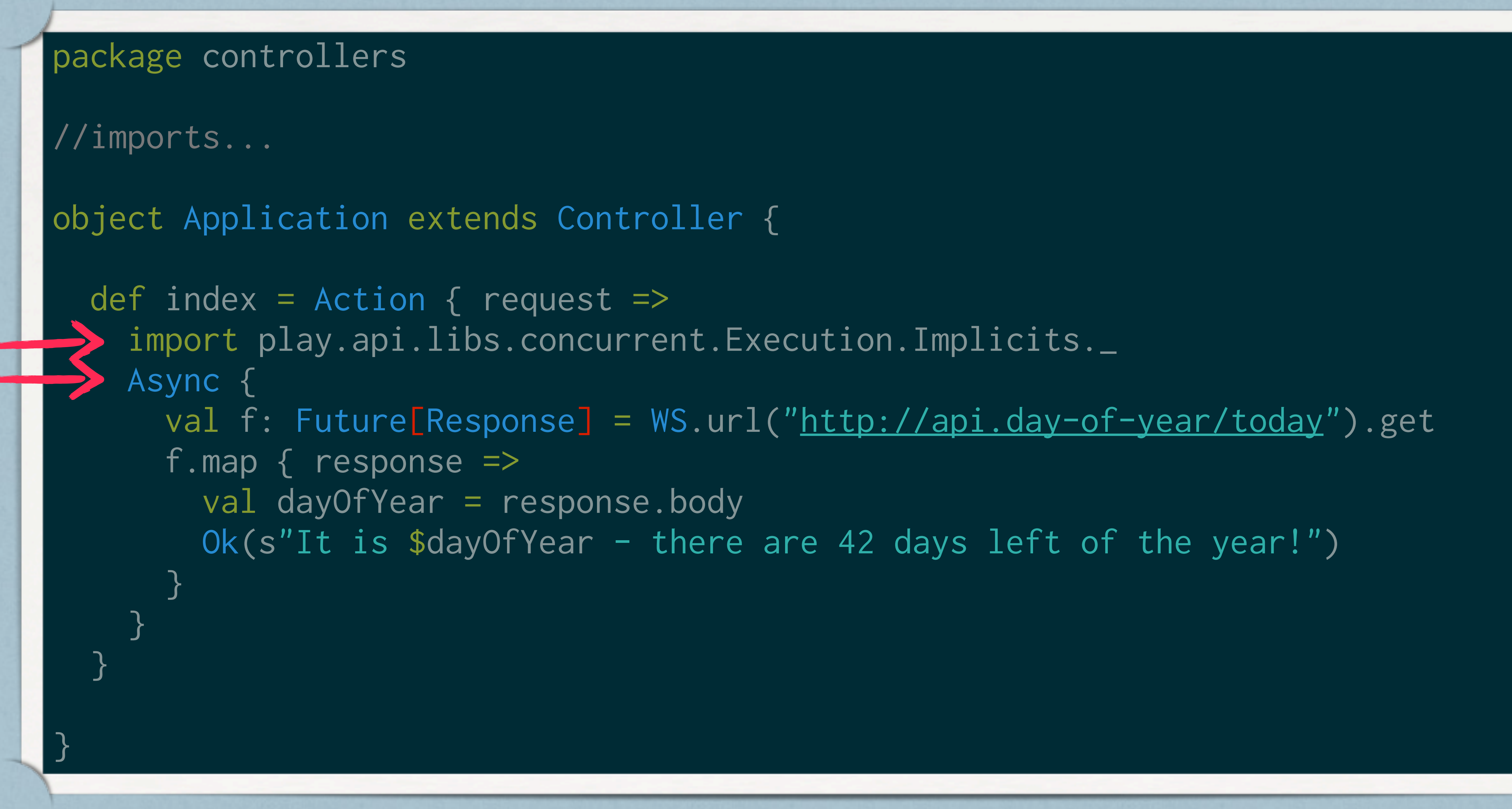
FUTURE IN Play



```
package controllers

//imports...

object Application extends Controller {

  def index = Action { request =>
    val f: Future[Response] = WS.url("http://api.day-of-year/today").get
    f.map { response =>
       val dayOfYear = response.body
       Ok(s"It is $dayOfYear - there are 42 days left of the year!")
    }
  }

}
```

FUTURE IN Play

EXECUTION CONTEXT & ASYNC

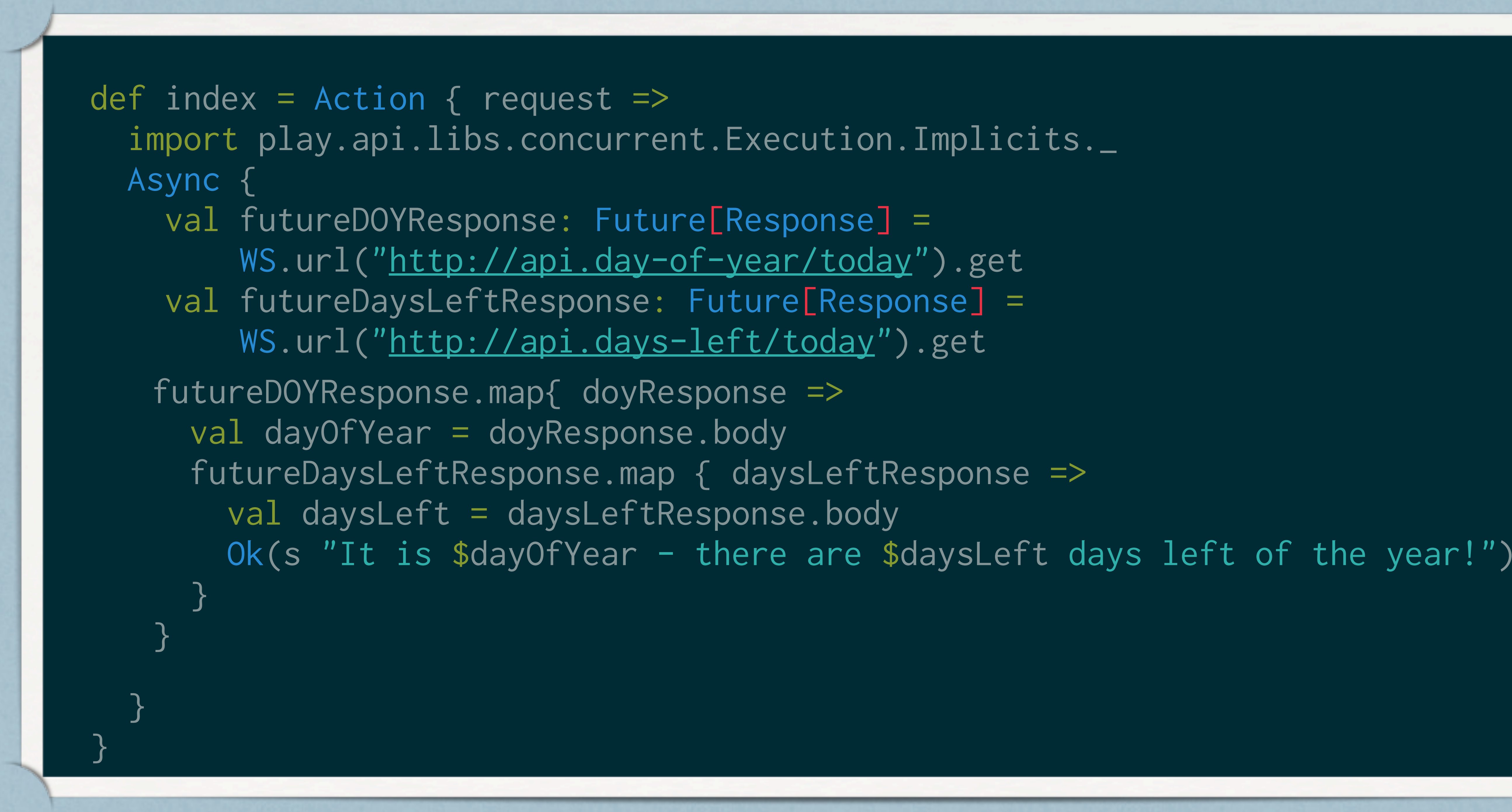


```
package controllers

//imports...

object Application extends Controller {

  def index = Action { request =>
    import play.api.libs.concurrent.Execution.Implicits._
    Async {
      val f: Future[Response] = WS.url("http://api.day-of-year/today").get
      f.map { response =>
        val dayOfYear = response.body
        Ok(s"It is $dayOfYear - there are 42 days left of the year!")
      }
    }
  }

}
```

Future composition in Play

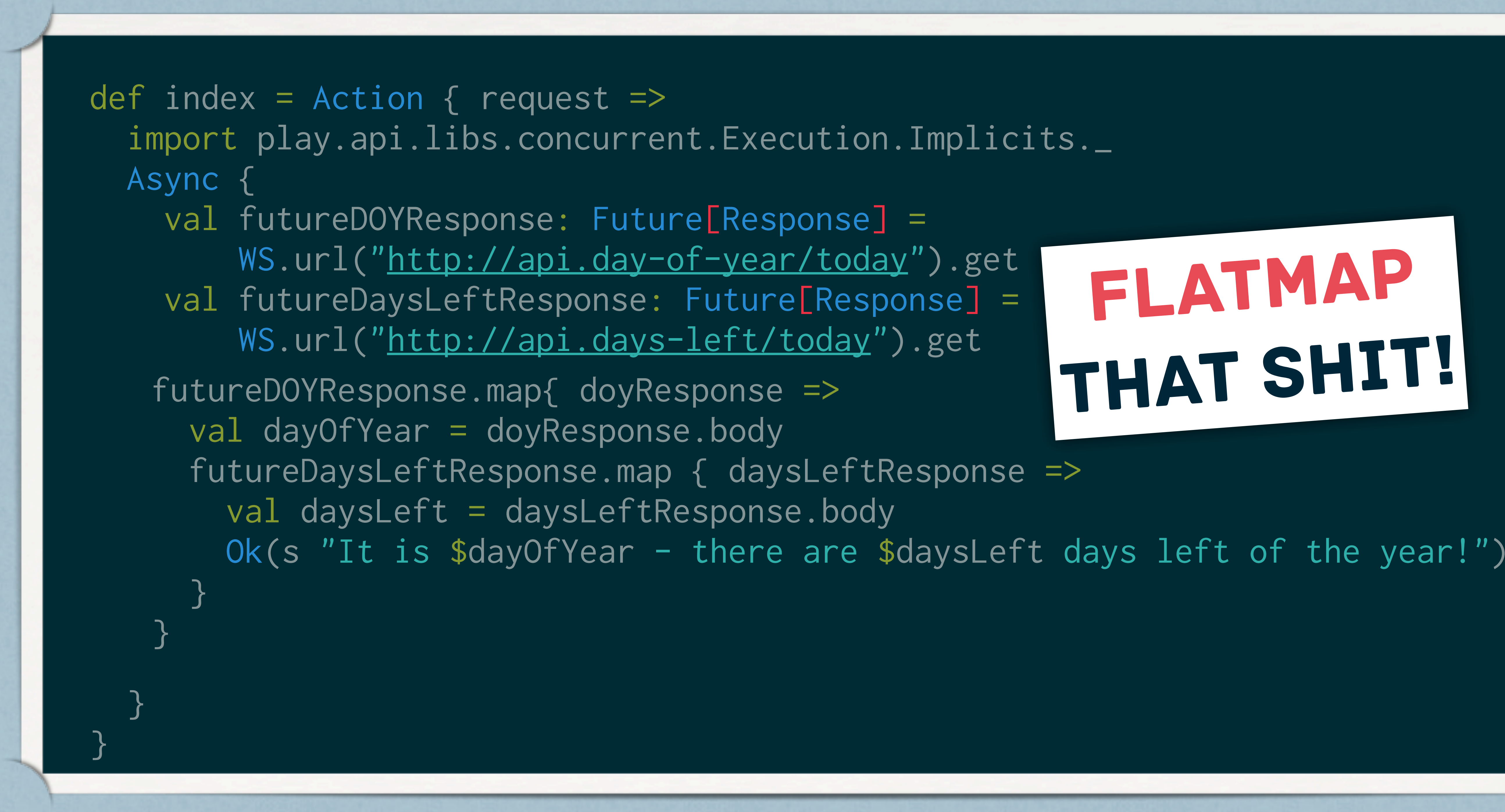
```
def index = Action { request =>
  import play.api.libs.concurrent.Execution.Implicits._
  Async {
    val futureDOYResponse: Future[Response] =
        WS.url("http://api.day-of-year/today").get
    val futureDaysLeftResponse: Future[Response] =
        WS.url("http://api.days-left/today").get








  }
}
```

FUTURE COMPOSITION IN Play



```scala
def index = Action { request =>
  import play.api.libs.concurrent.Execution.Implicits._
  Async {
    val futureDOYResponse: Future[Response] =
        WS.url("http://api.day-of-year/today").get
    val futureDaysLeftResponse: Future[Response] =
        WS.url("http://api.days-left/today").get
   futureDOYResponse.map{ doyResponse =>
     val dayOfYear = doyResponse.body
     futureDaysLeftResponse.map { daysLeftResponse =>
       val daysLeft = daysLeftResponse.body
       Ok(s "It is $dayOfYear - there are $daysLeft days left of the year!")
     }
   }

  }
}
```

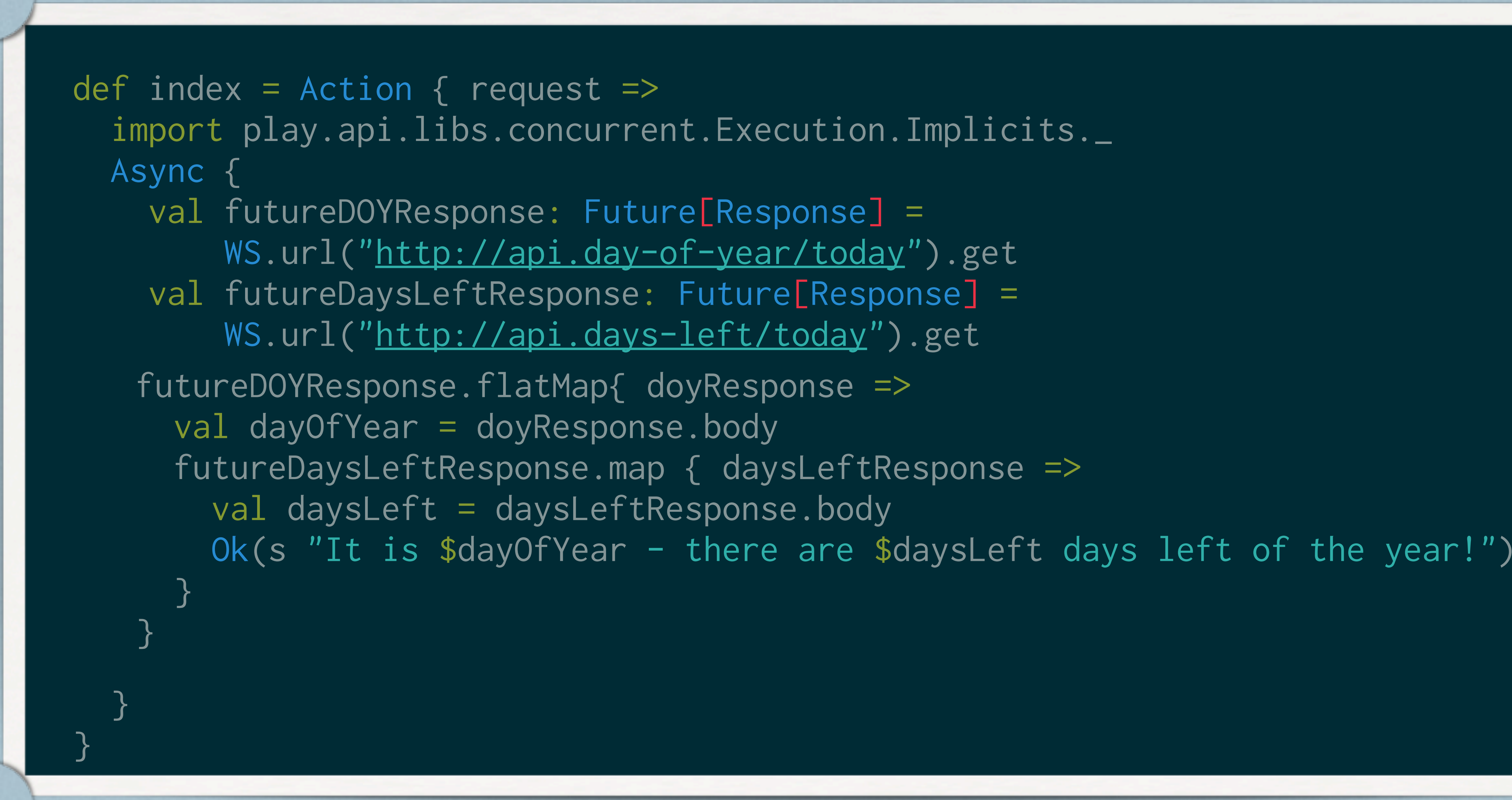
FUTURE COMPOSITION IN Play



```
def index = Action { request =>
  import play.api.libs.concurrent.Execution.Implicits._
  Async {
    val futureDOYResponse: Future[Response] =
        WS.url("http://api.day-of-year/today").get
    val futureDaysLeftResponse: Future[Response] =
        WS.url("http://api.days-left/today").get

   futureDOYResponse.map{ doyResponse =>
     val dayOfYear = doyResponse.body
     futureDaysLeftResponse.map { daysLeftResponse =>
       val daysLeft = daysLeftResponse.body
       Ok(s "It is $dayOfYear - there are $daysLeft days left of the year!")
     }
   }

  }
}
```

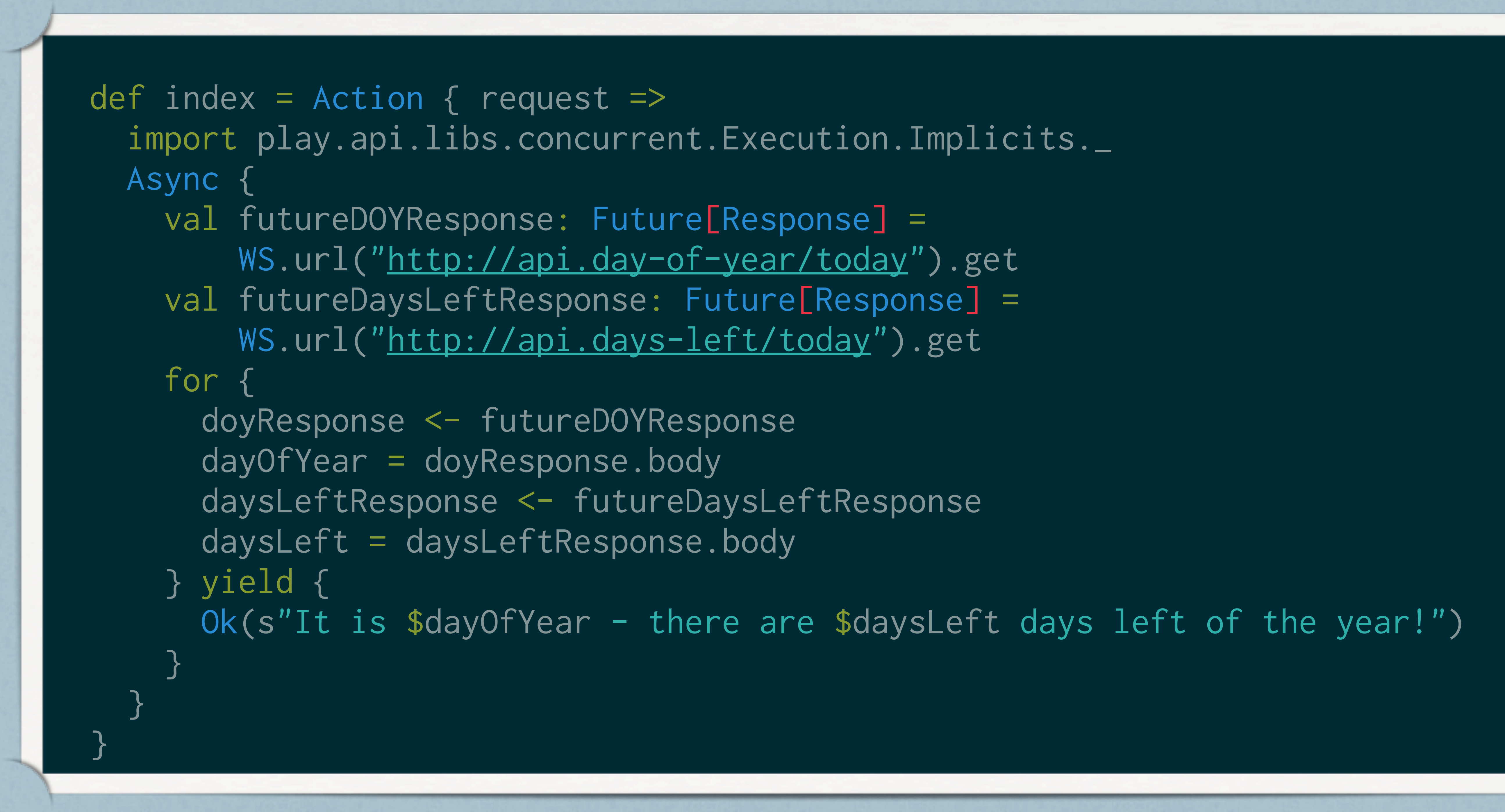
FLATMAP THAT SHIT!

FUTURE COMPOSITION IN Play



```
def index = Action { request =>
  import play.api.libs.concurrent.Execution.Implicits._
  Async {
    val futureDOYResponse: Future[Response] =
        WS.url("http://api.day-of-year/today").get
    val futureDaysLeftResponse: Future[Response] =
        WS.url("http://api.days-left/today").get
   futureDOYResponse.flatMap{ doyResponse =>
     val dayOfYear = doyResponse.body
     futureDaysLeftResponse.map { daysLeftResponse =>
       val daysLeft = daysLeftResponse.body
       Ok(s "It is $dayOfYear - there are $daysLeft days left of the year!")
     }
   }

  }
}
```

Future Composition^2 in Play



```
def index = Action { request =>
  import play.api.libs.concurrent.Execution.Implicits._
  Async {
    val futureDOYResponse: Future[Response] =
        WS.url("http://api.day-of-year/today").get
    val futureDaysLeftResponse: Future[Response] =
        WS.url("http://api.days-left/today").get
    for {
      doyResponse <- futureDOYResponse
      dayOfYear = doyResponse.body
      daysLeftResponse <- futureDaysLeftResponse
      daysLeft = daysLeftResponse.body
    } yield {
      Ok(s"It is $dayOfYear - there are $daysLeft days left of the year!")
    }
  }
}
```

FUTURE IN Play RECOVER

```
Async {
  val futureDOYResponse: Future[Response] = //...
  val futureDaysLeftResponse: Future[Response] = //...

  val futureResult = for {
    doyResponse <- futureDOYResponse
    dayOfYear = doyResponse.body
    daysLeftResponse <- futureDaysLeftResponse
    daysLeft = daysLeftResponse.body
  } yield {
    Ok(s”It is $dayOfYear - there are $daysLeft days left of the year!”)
  }

  futureResult.recover {
    case t: Throwable =>
      BadRequest(s”It is 21st December 2012 - end of the world?”)
  }
}
```

CREDITS

VIKTOR KLANG
TYPESAFE

PHILIPP HALLER
TYPESAFE

ALEX PROKOPEC
EPFL

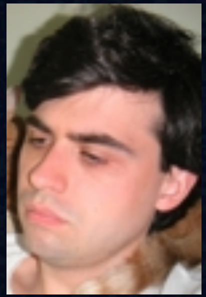
VOJIN JOVANOVIC
EPFL

MARIUS ERIKSEN
TWITTER

HEATHER MILLER
EPFL

ROLAND KUHN
TYPESAFE

DOUG LEA
SUNY

HAVOC PENNINGTON
TYPESAFE


QUESTIONS?

http://docs.scala-lang.org/sips/pending/futures-promises.html

http://www.playframework.org/documentation/2.0.4/ScalaAsync