

Building a Truly Modern Web App/API with Scala, Akka and Spray

Alex Cruise
May 23rd, 2014

Who am I?

- Just getting to be a greybeard, but not a manager!
- Self-taught
 - Been programming since I was a little kid, in the early 80s
 - I came to Computer Science as such (not the same as writing code for a living!) because I saw how it would help me solve problems I was already having, not because it seemed like it would be a good job when I was 18. :)

Why Are We Here?

- THE FUTURE

Why Are We Here?

- THE FUTURE IS...

Why Are We Here?

- THE FUTURE IS REACTIVE!
 - Well, that's just like, your OPINION, man.

Why Are We Here?

- THE FUTURE IS REACTIVE!
 - Well, that's just like, your OPINION, man.
 - Well, not JUST mine.

<http://reactivemanifesto.org/>

Why Are We Here?

- Scaling is hard

Why Are We Here?

- Scaling is hard
- Fault tolerance is hard

Why Are We Here?

- Scaling is hard
- Fault tolerance is hard
- Concurrency is hard

Why Are We Here?

- Scaling is hard
- Fault tolerance is hard
- Concurrency is hard
- Correctness is hard

Why Are We Here?

- Scaling is hard
- Fault tolerance is hard
- Concurrency is hard
- Correctness is hard
- Let's go shopping!

Why Are We Here?

- Scaling is hard
- Fault tolerance is hard
- Concurrency is hard
- Correctness is hard
- ~~Let's go shopping!~~ ಠ_ಠ

Why Are We Here?

- Scaling is hard
- Fault tolerance is hard
- Concurrency is hard
- Correctness is hard
- ~~Let's go shopping!~~
- Well... It's not like we have any choice, really. :/

Why Are We Here?

- Scaling is hard
- Fault tolerance is hard
- Concurrency is hard
- Correctness is hard
- ~~Let's go shopping!~~
- Well... It's not like we have any choice, really. :/
- We need this stuff to stay relevant.

Why Are We Here?

- Scaling is hard
- Fault tolerance is hard
- Concurrency is hard
- Correctness is hard
- ~~Let's go shopping!~~
- Well... It's not like we have any choice, really. :/
- We need this stuff to stay ~~relevant~~ employed.

If Only There Were Some Magic Framework...

- SURELY all this stuff is a Simple Matter of Programming...

If Only There Were Some Magic Framework...

- SURELY all this stuff is a Simple Matter of ~~Programming~~ *sufficiently advanced* frameworks...

If Only There Were Some Magic Framework...

- SURELY all this stuff is a Simple Matter of ~~Programming~~ sufficiently advanced libraries...
- *shrug* well... / haven't seen the Magic Framework yet. And I'm OLD.

If Only There Were Some Magic Framework...

- SURELY all this stuff is a Simple Matter of ~~Programming~~ sufficiently advanced libraries...
- *shrug* well... / haven't seen the Magic Framework yet. And I'm OLD.
- **Why not?**

No Unobtainium Bullet

“Happy families are all alike; every unhappy family is unhappy in its own way”

—Leo Tolstoy

No Unobtainium Bullet

“Happy families are all alike; every unhappy family is unhappy in its own way”

—Leo Tolstoy

- The small systems all start to look the same after awhile

No Unobtainium Bullet

“Happy families are all alike; every unhappy family is unhappy in its own way”

—Leo Tolstoy

- The small systems all start to look the same after awhile
 - if you squint
 - when you're bitter enough

No Unobtainium Bullet

“Happy families are all alike; every unhappy family is unhappy in its own way”

—Leo Tolstoy

- The small systems all start to look the same after awhile
 - if you squint
 - when you're bitter enough
- But large systems are **UNIQUE** and **TERRIFYING**.

But... All is Not Lost!

- There's no magic technology that will make everything easy.

But... All is Not Lost!

- There's no magic technology that will make everything easy.
- But there are some “new” tools that can help:

But... All is Not Lost!

- There's no magic technology that will make everything easy.
- But there are some “new” tools that can help:
 - Have confidence in your programs' correctness

But... All is Not Lost!

- There's no magic technology that will make everything easy.
- But there are some “new” tools that can help:
 - Have confidence in your programs' correctness
 - Reason about concurrency without blood coming out of your ears

But... All is Not Lost!

- There's no magic technology that will make everything easy.
- But there are some “new” tools that can help:
 - Have confidence in your programs' correctness
 - Reason about concurrency without blood coming out of your ears
 - Write code and still be able to understand it next year

But... All is Not Lost!

- There's no magic technology that will make everything easy.
- But there are some “new” tools that can help:
 - Have confidence in your programs' correctness
 - Reason about concurrency without blood coming out of your ears
 - Write code and still be able to understand it next year
 - Enjoy your work (at least / still do... After 5+ years of Scala!)

But... All is Not Lost!

- There's no magic technology that will make everything easy.
- But there are some “new” tools that can help:
 - Have confidence in your programs' correctness
 - Reason about concurrency without blood coming out of your ears
 - Write code and still be able to understand it next year
 - Enjoy your work (at least / still do... After 5+ years of Scala!)
 - Make scaling less awful

Tools

- Scala
- Akka
- Spray
- Spray-JSON
- mumble database mumble

Scala

- wow

much Hybrid OO-FP

such JVM

very Java compatibility

nice codings

Akka

- Industrial strength, battle-tested implementation of the Actor Model
- Concurrency with sanity
- Fault tolerance
- Location transparency without lies

Spray

- A tasteful HTTP server layer built on Akka
- High performance
- Immutable requests and responses
- Pleasant and powerful routing DSL
- NOT Yet Another Web Framework (use your own template system if you want to generate HTML)

Spray-JSON

- Typeclass-based JSON serialization/deserialization library
- Type safe (compile-time error if you try to ser/des a type with no known format)
- Nicely maps to and from Scala types
- No annotation mess
- Very little reflection
- Domain model and JSON codec completely decoupled

mumble Database *mumble*

- I haven't found a fantastic solution yet
- Hibernate works ***okay*** if you can stomach it
 - Immutable domain model works, but it insists on mutating collections in-place
- I have an ORM that I like, but if you're picky about mapping like me, it might be painful
- Lots of innovation ongoing in this space
 - Async drivers (non-blocking, streaming results...)
 - macros to avoid boilerplate

Part 1: Scala

Scala!

- OO in that everything is an object
- FP in that functions are values
- Opinionated but not dogmatic
 - imperative style is not punished (much)
 - ergonomic defaults to FP style
 - e.g. `Map(1 → “one”)` is an *immutable* map
- People like to joke about MOAR LANGUAGE FEATURES!!!one~~ but:
 - the features that are there are pretty orthogonal
 - Martin is gunshy about letting anyone else into the boat.
 - the Cascade of Attention Deficit Grad Students is SO OVER

Quick Tour of Scala: FP Basics

- Collections are everywhere, and a pleasure to work with
- Functions (e.g. lambdas, closures, methods) make life easier
- Higher-order functions are cute and cuddly

Scala: FP Basics Example

```
case class Person(name: String, age: Int)

val people = List(
  Person("Alex", 40),
  Person("Dexter", 7),
  Person("Holly", 29),
  Person("Spike", 4))

val (minors, adults) = people.partition(_.age < 19)
// minors => List(Person(Dexter,7),Person(Spike,4))
// adults => List(Person(Alex,40),Person(Holly,29))

minors.foreach(_.goToSchool)
adults.foreach(_.goToWork)
```


Scala: FP Basics Example

```
case class Person(name: String, age: Int)
```

```
val people = List(  A collection "literal"
```

```
    Person("Alex", 40),  
    Person("Dexter", 7),  
    Person("Holly", 29),  
    Person("Spike", 4))
```

```
val (minors, adults) = people.partition(_.age < 19)
```

```
// minors => List(Person(Dexter,7),Person(Spike,4))
```

```
// adults => List(Person(Alex,40),Person(Holly,29))
```

```
minors.foreach(_.goToSchool)
```

```
adults.foreach(_.goToWork)
```

Scala: FP Basics Example

```
case class Person(name: String, age: Int)
```

```
val people = List( A collection “literal”
```

```
  Person(“Alex”, 40),
```

```
  Person(“Dexter”, 7),
```

```
  Person(“Holly”, 29),
```

```
  Person(“Spike”, 4))
```

A Higher-Order Function



```
val (minors, adults) = people.partition(_.age < 19)
```

```
// minors => List(Person(Dexter,7),Person(Spike,4))
```

```
// adults => List(Person(Alex,40),Person(Holly,29))
```

```
minors.foreach(_.goToSchool)
```

```
adults.foreach(_.goToWork)
```

Scala: FP Basics Example

```
case class Person(name: String, age: Int)
```

```
val people = List(  A collection "literal"
```

```
    Person("Alex", 40),
```

```
    Person("Dexter", 7),
```

```
    Person("Holly", 29),    A Higher-Order Function
```

```
    Person("Spike", 4))
```

```
 Pattern Matching  
val (minors, adults) = people.partition(_.age < 19)
```

```
// minors => List(Person(Dexter,7),Person(Spike,4))
```

```
// adults => List(Person(Alex,40),Person(Holly,29))
```

```
minors.foreach(_.goToSchool)
```

```
adults.foreach(_.goToWork)
```

Scala: FP Basics Example

```
case class Person(name: String, age: Int)
```

```
val people = List( A collection “literal”
```

```
  Person(“Alex”, 40),
```

```
  Person(“Dexter”, 7),
```

```
  Person(“Holly”, 29),
```

```
  Person(“Spike”, 4))
```

A Higher-Order Function

A lambda
(anonymous function literal)

```
 Pattern Matching
```

```
val (minors, adults) = people.partition(_.age < 19)
```

```
// minors => List(Person(Dexter,7),Person(Spike,4))
```

```
// adults => List(Person(Alex,40),Person(Holly,29))
```

```
minors.foreach(_.goToSchool)
```

```
adults.foreach(_.goToWork)
```

Scala: FP Basics Example

```
case class Person(name: String, age: Int)
```

```
val people = List( A collection “literal”
```

```
  Person(“Alex”, 40),
```

```
  Person(“Dexter”, 7),
```

```
  Person(“Holly”, 29),
```

```
  Person(“Spike”, 4))
```

A Higher-Order Function

A lambda
(anonymous function literal)

```
 Pattern Matching
```

```
val (minors, adults) = people.partition(_.age < 19)
```

```
// minors => List(Person(Dexter,7),Person(Spike,4))
```

```
// adults => List(Person(Alex,40),Person(Holly,29))
```

```
minors.foreach(_.goToSchool)
```

```
adults.foreach(_.goToWork)
```

More lambdas
(no return values though)

Scala FP Basics: Handy HOFs

- Handy higher-order functions found on nearly all collections (including Maps!)
 - map, filter, collect, find, exists, forall, flatMap, foreach, etc.
- For comprehensions
 - work on everything that has map/filter/flatMap
 - Not just nominally typed collections/iterables
- flatMap is... special.
 - The “M” word! DUN DUN DUN

Scala FP Basics: Handy HOFs

- map
 - given a collection $C[A]$ and a function $A \Rightarrow B$, produce a $C[B]$ by applying the function to each element
- filter
 - given a collection $C[A]$ and a predicate $A \Rightarrow \text{Boolean}$, produce another $C[A]$ consisting of only those elements for which the predicate was satisfied

Scala FP Basics: Handy HOFs

- collect
 - Basically filter+map in one traversal
 - Given a collection C[A] and a PartialFunction[A,B], build a new collection C[B] consisting of the A's that satisfied the PartialFunction and were then transformed by it into B's
- find
 - Given a collection C[A] and a predicate A => Boolean, find the **first** element of C that satisfied the predicate and wrap it in a Some[A], otherwise None
 - Also see collectFirst

Scala FP Basics: Handy HOFs

- exists
 - Given a collection C[A] and a predicate A => Boolean, return true as soon as the first element satisfying the predicate is found, or false if no satisfactory element is found
- forall
 - Given a collection C[A] and a predicate A => Boolean, return true iif **every** element in the collection satisfied the predicate, or false if **no** element satisfied the predicate
 - Note! **empty.forall(anything)** is always true!
 - Even **empty.forall(_ => false)**

Scala FP Basics: Handy HOFs

- `foreach`
 - Given a collection `C[A]`, and a ***procedure*** `A => Unit`, call the procedure on each element of `C`.
 - It's all about side effects
- `flatMap`
 - Given a collection `C[A]`, and a function `A => C[B]`, returns the `C[B]` resulting from concatenating every “small” `C[B]` resulting from the function into one “big” `C[B]`

Scala FP Basics: For Comprehensions (1/3)

- With one generator and “yield”, translates to calls to map/filter

```
val xs = List(1,2,3)
```

```
for (x <- xs) yield x * 2
```

```
// returns List(2,4,6)
```

```
// exactly equivalent to:
```

```
xs.map(x => x*2)
```

```
for (x <- xs if x % 2 == 0) yield x * 10
```

```
// returns List(20)
```

```
// exactly equivalent to:
```

```
xs.filter(x => x % 2 == 0).map(x => x * 10)
```

Scala FP Basics: For Comprehensions (2/3)

- With multiple generators and “yield”, translates to map/flatMap
- Looks like a nested loop but isn't

```
val ys = List(10,20,30)
for (x <- xs; y <- ys) yield x*y
// returns List(10, 20, 30, 20, 40, 60, 30,
60, 90)
// exactly equivalent to:
xs.flatMap(x => ys.map(y => x * y))
```

Scala FP Basics: For Comprehensions (3/3)

- Without “yield”, translates to calls to foreach

```
val xs = List(1,2,3)
val ys = List(10,20,30)
```

```
for (x <- xs) printf("%d ", x * 2)
// prints "2 4 6 ", doesn't return anything
// exactly equivalent to:
xs.foreach(x => printf("%d ", x*2))
```

```
for (x <- xs; y <- ys) printf("%d ", x*y)
// prints "10 20 30 20 40 60 30 60 90 ", doesn't return
// exactly equivalent to:
xs.foreach(x => ys.foreach(y => x * y)))
```

Scala: FP Basics Exercise

- Implement List
 - Data constructors
 - Empty
 - Non-Empty
 - foreach method
 - map method

Scala: Algebraic Data Types

- Define families of related types
- Concise declaration syntax
- Pattern matcher can test for missing cases
- case classes
 - Boilerplate generated for you
 - equals, hashCode, toString, constructor, getters/setters
- Uniform Access Principle
 - All access is through methods (generated or overridden), not fields
 - Refactor without flipping back and forth between `foo.bar`, `foo.getBar()`, `foo.bar()`

Scala: ADTs in the Library

- `Option[X]`
 - Value is present or not present
 - `Some(x)` or `None`
 - Can still call methods on `None`, without risking NPE
 - has `map`, `flatMap`, `foreach`
 - surprisingly useful, especially when the value might be `None`

Scala: ADTs in the Library

- `Either[A,B]`
 - Value is `Left(a)` or `Right(b)`
 - By convention, `Right` means “correct” and `Left` means “error”, but there's no inherent bias in the library
 - A lot of people would have preferred a Right-biased `Either...`

Scala: ADTs in the Library

- List[A]
 - Classical immutable singly linked list
 - Value is non-empty or empty
 - Data constructors are:
 - `::(hd: A, tl: List[A])` (pronounced “cons”)
 - `Nil`
 - Easy to understand, but be careful with performance characteristics
 - $O(1)$ prepend
 - $O(n)$ append, scan, size, reverse
 - Use Vector if unsure about performance

Scala: ADTs Example

```
/** the state of a request */  
sealed trait RequestState[+T]  
  
/** we made a request at time `when` */  
case class Requested(when: Long) extends  
RequestState[Nothing]  
  
/** our request was denied for some `reason` :( */  
case class Rejected(reason: String) extends  
RequestState[Nothing]  
  
/** we waited `howLong` ms but to no avail */  
case class TimedOut(howLong: Long) extends  
RequestState[Nothing]  
  
/** we got `it`! */  
case class Got[T](it: T) extends RequestState[T]
```

Scala: ADTs Example

```
/** the state of a request */  
sealed trait RequestState[+T]  
  
/** we made a request at time `when` */  
case class Requested(when: Long) extends  
RequestState[Nothing]  
  
/** our request was denied for some `reason` :( */  
case class Rejected(reason: String) extends  
RequestState[Nothing]  
  
/** we waited `howLong` ms but to no avail */  
case class TimedOut(howLong: Long) extends  
RequestState[Nothing]  
  
/** we got `it`! */  
case class Got[T](it: T) extends RequestState[T]
```

Scala: ADTs Example

```
/** the state of a request */  
sealed trait RequestState[+T]  
  
/** we made a request at time `when` */  
case class Requested(when: Long) extends  
RequestState[Nothing]  
  
/** our request was denied for some `reason` :( */  
case class Rejected(reason: String) extends  
RequestState[Nothing]  
  
/** we waited `howLong` ms but to no avail */  
case class TimedOut(howLong: Long) extends  
RequestState[Nothing]  
  
/** we got `it`! */  
case class Got[T](it: T) extends RequestState[T]
```

Scala: ADTs Example

```
/** the state of a request */  
sealed trait RequestState[+T]  
  
/** we made a request at time `when` */  
case class Requested(when: Long) extends  
  RequestState[Nothing]  
  
/** our request was denied for some `reason` :( */  
case class Rejected(reason: String) extends  
  RequestState[Nothing]  
  
/** we waited `howLong` ms but to no avail */  
case class TimedOut(howLong: Long) extends  
  RequestState[Nothing]  
  
/** we got `it`! */  
case class Got[T](it: T) extends RequestState[T]
```

Scala: ADTs Example

```
/** the state of a request */  
sealed trait RequestState[+T]  
  
/** we made a request at time `when` */  
case class Requested(when: Long) extends  
RequestState[Nothing]  
  
/** our request was denied for some `reason` :( */  
case class Rejected(reason: String) extends  
RequestState[Nothing]  
  
/** we waited `howLong` ms but to no avail */  
case class TimedOut(howLong: Long) extends  
RequestState[Nothing]  
  
/** we got `it`! */  
case class Got[T](it: T) extends RequestState[T]
```

Scala: ADTs Example

```
/** the state of a request */  
sealed trait RequestState[+T]  
  
/** we made a request at time `when` */  
case class Requested(when: Long) extends  
RequestState[Nothing]  
  
/** our request was denied for some `reason` :( */  
case class Rejected(reason: String) extends  
RequestState[Nothing]  
  
/** we waited `howLong` ms but to no avail */  
case class TimedOut(howLong: Long) extends  
RequestState[Nothing]  
  
/** we got `it`! */  
case class Got[T](it: T) extends RequestState[T]
```


Scala: ADTs Exercise

- **TODO TODO TODO**

Scala: Pattern Matching

- When you have a family of types, you need to be able to distinguish between them at runtime
- Replacement for typecheck-and-downcast ladders
- Matches not just type, but inner structure of values
- Extracts values from inner structure
- OO dogma says “favour polymorphism over conditionals”, but...
 - Functionality gets spread all over the place, hard to read actual functionality at a glance

Scala: Simple Pattern Matching

```
val username: Option[String] = ???
```


```
username match {  
  case Some(name) =>  
    println(s"Hi there $name!")  
  case Some(name) if name.toLowerCase == "alex" =>  
    println("Hey, 'sup Alex buddy!")  
  case None =>  
    println("Howdy stranger!")  
}
```

Scala: Simple Pattern Matching

```
val username: Option[String] = ???
```

```
username match {  
  case Some(name) =>  
    println(s"Hi there $name!")  
  case Some(name) if name.toLowerCase == "alex" =>  
    println("Hey, 'sup Alex buddy!")  
  case None =>  
    println("Howdy stranger!")  
}
```

Type check




Scala: Simple Pattern Matching

```
val username: Option[String] = ???
```

```
username match {  
  case Some(name) =>  
    println(s"Hi there $name!")  
  case Some(name) if name.toLowerCase == "alex" =>  
    println("Hey, 'sup Alex buddy!")  
  case None =>  
    println("Howdy stranger!")  
}
```


Value extraction



Scala: Simple Pattern Matching

```
val username: Option[String] = ???
```

```
username match {  
  case Some(name) =>  
    println(s"Hi there $name!")  
  case Some(name) if name.toLowerCase == "alex" =>  
    println("Hey, 'sup Alex buddy!")  
  case None =>  
    println("Howdy stranger!")  
}
```



Pattern guard

Scala: Simple Pattern Matching

```
val username: Option[String] = ???

username match {
  case Some(name) =>
    println(s"Hi there $name!")
  case Some(name) if name.toLowerCase == "alex" =>
    println("Hey, 'sup Alex buddy?")
  case None =>
    println("Howdy stranger!")
}
```

- Spot the bug? :)

Scala: Simple Pattern Matching

```
val username: Option[String] = ???

username match {
  case Some(name) =>
    println(s"Hi there $name!")
  case Some(name) if name.toLowerCase == "alex" =>
    println("Hey, 'sup Alex buddy?")
  case None =>
    println("Howdy stranger!")
}
```

- Spot the bug?
- The compiler did! :)

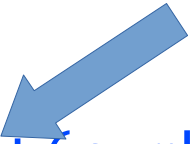
```
<console>:14: warning: unreachable code
      println("Hey, 'sup Alex buddy?")
```


Scala: Less Simple Pattern Matching

```
val response: Option[Either[String, List[Int]]] = ???
```

```
response match {  
  case Some(Right(numbers)) =>  
    println(s"Yay, I got ${numbers.sum}")  
  case None =>  
    println("No numbers yet :(")  
}
```

Test & extract from nested values



Scala: Less Simple Pattern Matching

```
val response: Option[Either[String,List[Int]]] = ???
```

```
response match {  
  case Some(Right(numbers)) =>  
    println(s"Yay, I got ${numbers.sum}")  
  case None =>  
    println("No numbers yet :(")  
}
```

- Spot the bug?

Scala: Less Simple Pattern Matching

```
val response: Option[Either[String, List[Int]]] = ???
```

```
response match {  
  case Some(Right(numbers)) =>  
    println(s"Yay, I got ${numbers.sum}")  
  case None =>  
    println("No numbers yet :(")  
}
```

- Spot the bug?
- The compiler did!

```
<console>:9: warning: match may not be exhaustive.  
It would fail on the following input: Some(Left(_))
```

Scala: Less Simple Pattern Matching

```
val response: Option[Either[String,List[Int]]] = ???
```

```
response match {  
  case Some(Right(numbers)) =>  
    println(s"Yay, I got ${numbers.sum}")  
  case Some(Left(err)) =>  
    println(s"Aww, we got an error: $err")  
  case None =>  
    println("No numbers yet :(")  
}
```

- Note: Those are warnings by default, not errors. Use `-Xfatalwarnings` if you want to be a cool kid.
 - I'm not a cool kid

Scala: Pattern Matching Exercise

- **TODO TODO TODO**

Scala: Partial Functions

- A function $A \Rightarrow B$ that returns some B for every possible A is a ***total function***
- But many things that we call “functions” in the mathematical sense are not in fact total, e.g.:
 - $f(x) = 1/x$ where $x = 0$
 - $f(x) = \tan(x)$ where $x = \pi/2$
- Instead of throwing an exception... Why not ask the function?

Scala: Partial Functions

```
trait PartialFunction[-A,+B] extends (A=>B) {  
  // inherited from Function1[A,B]  
  // def apply(a: A): B  
  def isDefinedAt(x: A): Boolean  
}
```

Scala: Partial Functions

```
val divide: PartialFunction[(Int, Int),Int] = ???
```

```
val x = ??? // values from elsewhere
```

```
val y = ???
```

```
if (divide.isDefinedAt((x, y))) {  
    divide((x,y)) // foo(bar) is sugar for foo.apply(bar)  
} else {  
    println("Nice try!")  
}
```


Scala: Partial Functions

```
val divide: PartialFunction[(Int, Int), Int] = ???
```

```
val x = ??? // values from elsewhere
```

```
val y = ???
```

Sugar for Tuple2[Int,Int]



Sugar for Tuple2(x,y)



```
if (divide.isDefinedAt((x, y))) {  
    divide((x,y)) // foo(bar) is sugar for foo.apply(bar)  
} else {  
    println("Nice try!")  
}
```

Scala: Partial Functions

```
val divide: PartialFunction[(Int, Int), Int] = ???
```

```
val x = ??? // values from elsewhere
```

```
val y = ???
```

Ask nicely first!

```
if (divide.isDefinedAt((x, y))) {  
    divide((x,y)) // foo(bar) is sugar for foo.apply(bar)  
} else {  
    println("Nice try!")  
}
```

Scala: Partial Functions

```
val divide: PartialFunction[(Int, Int), Int] = ???
```

```
val x = ??? // values from elsewhere
```

```
val y = ???
```

```
if (divide.isDefinedAt((x, y))) {
```

```
    divide((x,y)) // foo(bar) is sugar for foo.apply(bar)
```

```
} else {
```

```
    println("Nice try!")
```

```
}
```




Call when we know it's safe

Scala: Partial Functions

```
val divide: PartialFunction[(Int, Int), Int] = ???
```

```
val x = ??? // values from elsewhere
```

```
val y = ???
```

```
if (divide.isDefinedAt((x, y))) {  
    divide((x,y)) // foo(bar) is sugar for foo.apply(bar)  
} else {  
    println("Nice try!")  Handle the "exception" cleanly  
}
```

Scala: Partial Functions

- In practice, partial functions aren't generally this noisy, because...
- Partial functions and pattern matching go together like PB&J
- Any place a `Function1[A,B]` or `PartialFunction[A,B]` is expected, you can pass a “partial function literal”

Scala: Partial Functions

- A “partial function literal” should look VERY familiar:

```
// declaration site
```

```
def doStuff(pf: PartialFunction[(Int,Int),Int]) = ???
```

```
// call site
```

```
doStuff {  
  case (x,y) if x == y => ...  
  case (x,y) => ...  
}
```

Scala: Partial Functions

- Martin used to tout PartialFunction as an example of OO-FP synergy
- Since functions are objects just like everything else, they can have other methods, not just apply
- In practice, the inheritance relationship between PartialFunction and Function1 is not without controversy...

Scala: Partial Functions Exercise

- implement integer division as a PartialFunction
 - class Foo extends PartialFunction[Tuple2[Int,Int],Int]
 - def apply(t: (Int, Int)): Int
 - def isDefinedAt(t: (Int, Int)): Boolean
 - Tuple reminders:
 - pattern matching: case (x,y) => ...
 - construction: (x,y)
 - selection (ewww) tuple._1, tuple._2

Scala: Implicit

- When there's only one obviously reasonable thing to do, you'd rather not have to spell it out
- Implicit parameters
- Implicit classes (enhancing existing types with new functionality)
- typeclasses

Scala: Implicit Parameters

- Methods (and constructors) can have **multiple parameters lists**; the *last* parameter list can be marked “implicit”
- Implicit parameters can always be passed explicitly. If omitted, the compiler will **search** for an implicit value that's in scope
- Values must be ***declared as implicit*** in order to be found
 - The compiler will not pick up the random junk you left lying around that happens to be the correct type
- Useful for stuff that's always there but not always needed, like loggers, context objects, etc.

Scala: Implicit Resolution

- If ***more than one*** implicit value is found to be in scope, it's a compile-time error
- If ***no*** implicit value is found, it's a compile-time error
- Implicit resolution is powerful and a bit confusing when used to its fullest... ask me about the details later :)

Scala: Implicit Classes

- You have an `X`, you want to be able to say `X.foo`, but `X` has no `foo` method
- `X#foo` is trivially implemented in terms of `X`'s existing data/behaviour/state
- No need to wrap/box explicitly

Scala: Implicit Classes

```
val x: X = ???
```

```
x.foo // error: foo is not a member of X
```

```
implicit class XWithFoo(val x: X)
    extends AnyVal
{
    def foo = x.???
}
```

```
x.foo // OK, compiler wrapped x in a XWithFoo
```

Scala: Implicit Classes Exercise

- Make it possible to call “halve”, “double” and “cube” methods on an Int
- Reminders:

```
implicit class Foo(val something: T)
    extends AnyVal
{
    def bar(...) = something.???
}
```

Scala: Typeclasses

- Shamelessly *but lovingly* borrowed from a Haskell language feature

```
class Eq a where  
    (==) :: a -> a -> Bool  
  
instance Eq Integer where  
    x == y = x `integerEq` y
```

- Not a language feature in Scala :)

Scala: Typeclasses

- We often want to treat values of various unpredictable types as if they also had some other functionality, e.g.:
 - serialize yourself as JSON
 - compare yourself to others
 - parse yourself from a String
- We could use an implicit class, but that's not as cool!

Scala: Typeclasses

- Sometimes you can't modify the original types
 - e.g. closed source library, dependencies on particular versions
- You're Wrong To Want That™
 - Keep the domain model free of integration-specific concerns like wire formats

Scala: Typeclass Example

```
// instances can render an A as a String
trait Printable[A] {
  def print(a: A): String
}
```

Scala: Typeclass Example

```
// a Printable instance for Int
implicit object IntIsPrintable
  extends Printable[Int]
{
  def print(a: Int) = a.toString
}
```

Scala: Typeclass Example

```
// an instance for Double that's not
// just .toString
implicit object DoubleIsPrintable
    extends Printable[Double]
{
    def print(d: Double) = printf("%.3f",d)
}
```

Scala: Typeclass Example

```
// call site
def print[A](a: A)(implicit printable: Printable[A]) {
  println(printable.print(a))
}
```

- Gimme some sugar, baby!

```
// call site
def print[A : Printable](a: A) {
  val pr = implicitly[Printable[A]]
  println(pr.print(a))
}
```

- “Context bounds” FTW

Scala: Typeclass Exercise

- Implement the Eq typeclass for String several different ways
 - case-insensitive version
 - perverse version (equal when reversed?)
 - demonstrate how to switch between the instances
- BONUS: Implement an Eq instance for Double that tolerates minuscule differences

Digression A: Project Risk Factors

- Problem domain risk factors
 - Some problem domains are inherently full of hairy yaks
 - Some are not necessarily complex, but unfamiliar
- Tooling/infrastructure risk factors
 - Some tools are immature or buggy
 - Some are hard to use and/or unfamiliar
- Scale/distributed architecture is itself a huge risk factor
- Complex + unfamiliar problem domain + tooling problems + distributed architecture = PROJECT DEATH
- Minimize as many of these risk categories as you can at any given time

Part 2: Akka

Actor Model of Computation

- As old as me! (Hewitt 1973)
- An actor ***can***:
 - Receive messages
 - Send messages
 - Note that sending and receiving are ***not tightly coupled!***
The receipt of one message can result in ***zero, one or many*** sends, none of which is necessarily addressed to the sender of the original message
 - Change its own behaviour
 - Spawn children

Actor Model of Computation

- An actor **cannot**:
 - Process more than one message at a time
 - Implication: Actors can safely make use of (local) mutable state
- An actor ***should*** not:
 - Directly observe or modify the state of other actors, except by sending/receiving messages
 - Send mutable messages

Akka

- Initially heavily inspired by Erlang
- Supervision hierarchy for fault tolerance
- Message passing
- Behaviour modification
- Location transparency (remoting, clustering)

Akka Basics

- An actor's initial behaviour is defined by returning a `PartialFunction[Any,Unit]` from the `receive` method:

```
final class MyActor extends akka.actor.Actor {  
  def receive: PartialFunction[Any,Unit] = {  
    case MyMessage(Some(thing), Left(over)) =>  
      // do stuff...  
    case SomeOtherMessage =>  
      // do different stuff...  
  }  
}
```

Akka: Sending Messages

- An actor can send a message to another:

```
final class MyActor extends akka.actor.Actor {  
  def receive = {  
    case MyMessage(Some(thing), Left(over)) =>  
      val myBuddy: ActorRef = ???  
      myBuddy ! HiThere(over)  
    case SomeOtherMessage =>  
      // do different stuff...  
  }  
}
```

Akka: Spawning

- An actor can create a child...

```
final class MyActor extends akka.actor.Actor {  
  def receive: PartialFunction[Any,Unit] = {  
    case MyMessage(Some(thing), Left(over)) =>  
      val kid = context.actorOf(  
        Props(new ChildActor(thing)  
      )  
  
      // kid is now started  
      // let's boss him around  
  
      kid ! GoMowTheLawn  
  }  
}
```

Akka: Replying to Messages

- An actor can reply to messages...

```
final class MyActor extends akka.actor.Actor {  
  def receive: PartialFunction[Any,Unit] = {  
    case MyMessage(Some(thing), Left(over)) =>  
      sender() ! HeyHowsItGoing(over, "there")  
  }  
}
```

Akka: Blocking on Replies

- An actor can wait for a reply to its message...
 - But try to avoid it! You're blocking a thread, and cannot process any other messages while waiting.

```
import akka.pattern.ask

final class MyActor extends akka.actor.Actor {
  def receive: PartialFunction[Any,Unit] = {
    case MyMessage(Some(thing), Left(over)) =>
      val myBuddy: ActorRef = ???
      val replyF: Future[Howdy] =
        (myBuddy ? Hey(over, "there")).mapTo[Howdy]
      val reply: Howdy = Await.result(replyF, 30.seconds)
  }
}
```


Akka: Replies

- Much better to treat replies as just another message
 - If there are multiple ongoing “conversations”, include fields in the messages to track which conversation each message belongs to

```
final class MyActor extends akka.actor.Actor {  
  def receive: PartialFunction[Any,Unit] = {  
    case MyMessage(Some(thing), Left(over)) =>  
      val myBuddy: ActorRef = ???  
      myBuddy ! Hey(over, “there”)  
    case Howdy(...) =>  
      // Continue the conversation in a civilized way  
  }  
}
```

Akka: Forwarding Messages

- An actor can forward a message to another actor
 - original sender is preserved, replies pass us by

```
final class MyActor extends akka.actor.Actor {  
  def receive: PartialFunction[Any,Unit] = {  
    case msg: MyMessage =>  
      val myBuddy: ActorRef = ???  
      myBuddy.forward(msg)  
  }  
}
```

Akka: Watching Other Actors

- An actor can request to be notified when another actor (not necessarily its child) has terminated:

```
final class MyActor extends akka.actor.Actor {  
  def receive = {  
    case MyMessage(Some(thing), Left(over)) =>  
      val myBuddy: ActorRef = ???  
      context.watch(myBuddy)  
  
    case SomeOtherMessage(...) => // ...  
  
    case Terminated(victim) =>  
      lament()  
      wail()  
  
      // I can't go on without you, buddy!  
      context.stop(self)  
  }  
}
```

Akka: ReceiveTimeout

- An actor can request to be notified when no message has been received in some timeout:

```
final class MyActor extends akka.actor.Actor {  
  def receive: PartialFunction[Any,Unit] = {  
    case MyMessage(Some(thing), Left(over)) =>  
      val kid = context.actorOf(  
        Props(new ChildActor(thing)  
      )  
  
      context.setReceiveTimeout(10.seconds)  
  
    case SomeOtherMessage(...) =>  
      // ...  
  
    case ReceiveTimeout =>  
      // I know when I'm not wanted... :(  
      context.stop(self)  
  }  
}
```

Akka: Scheduler

- Akka includes a scheduler for periodic tasks

```
final class MyActor extends akka.actor.Actor {  
  def receive: PartialFunction[Any,Unit] = {  
    case MyMessage(Some(thing), Left(over)) =>  
      context.system.scheduler.schedule(  
        10.seconds, // initial delay  
        1.second,   // period  
        self,       // whom to send the message to  
        TheTick     // what message to send  
      )  
  
    case SomeOtherMessage(...) =>  
      // ...  
  
    case TheTick =>  
      doPeriodicVillainCleanup()  
  }  
}
```

Akka: Behaviour Modification

- An actor can change its behaviour
 - Pass new state as parameters to the behaviour methods
 - Cleaner than keeping a bunch of dirty vars around
 - Analogous to the recursive “loop” function in Erlang
- Behaviours are permanent by default

```
final class MyActor extends akka.actor.Actor {  
  def receive = {  
    case MyMessage(Some(thing), Left(over)) =>  
      context.become(thinkDifferent(thing))  
  }  
  
  def thinkDifferent(thing: Thing): Receive = {  
    case SomeOtherMessage(...) =>  
  }  
}
```

Akka: Behaviour Modification

- If a temporary change in behaviour is desired, behaviours can be pushed onto and popped off of a stack

```
final class MyActor extends akka.actor.Actor {  
  def receive = {  
    case MyMessage(Some(thing), Left(over)) =>  
      context.become(justASec, discardOld=false)  
  }  
  
  def justASec: Receive = {  
    case SomeOtherMessage(...) =>  
      // do something, quick!  
      dostuff()  
      context.unbecome()  
  }  
}
```

Akka: Lifecycle Hooks (1/2)

- Can be overridden if desired
- `def preStart(): Unit`
 - Called after the actor is initialized, but before it's started
 - No message can be received from any other actor until `preStart` is completed
 - A good place to send yourself an init message!
 - Don't do initialization in your constructor/body!
 - Exceptions sometimes get eaten by grues

Akka: Lifecycle Hooks (2/2)

- `def postStop(): Unit`
 - Called after the actor has stopped
- `def preRestart(reason: Throwable,
 message: Option[Any])
 : Unit`
 - Called after the actor has crashed, but before it has been restarted
 - The exception that caused/signaled the crash is provided
 - If the crash was caused by a particular message, it's provided

Akka: Supervision Hierarchy

- Every actor has a parent
- When an actor dies:
 - its parent (and any watchers) are notified
 - the parent actor can choose how to react:
 - restart the failed child (default)
 - Can set a maximum restarts per time policy, to prevent flapping
 - restart all its children
 - escalate the failure (i.e. crash itself, relying on its own parent to handle the situation)

Akka Bootstrapping

- Create an ActorSystem
 - They're heavyweight (among other things, they have a thread pool), so one per JVM is good, unless you have some special needs

```
object MyProgram {  
  def main(args: Array[String]) {  
    val system = ActorSystem("myProgram")  
  
    val rootActor = system.actorOf(  
      Props(new RootActor())  
    )  
  
    // rootActor is started now  
  }  
}
```

- Root actors (direct children of an ActorSystem) get ***Special High Intensity Supervision*** by default

Akka Bootstrapping (2)

- The ActorSystem won't shut down automatically
 - any actor can do it; Root is a good candidate
`context.system.shutdown()`
 - Some other signaling mechanism can be used
 - Or just run forever until killed
 - No, really!

Akka: Exercises

- Two-actor ping/pong
 - Count the number of round trips, log it
 - Implement rate limiting
- Guess the Number game
 - Single-player game
 - One actor instance per player/game
 - Receives Guess messages, replies with one of:
 - High
 - Low
 - Won
 - Lost

Digression B: Avoid Premature Abstraction (1/2)

- Founder of Architecture Astronauts Anon group on FB
- I keep stepping on the YAGNI rake in the grass, one day I'll learn...
- Delay abstraction until repetition is too painful
- Trust your tools
 - Branch early, branch often
 - Be willing to abandon failed experiments before you fall in love with them
 - Refactor mercilessly, but make sure it compiles before you SHIPIT :)
 - Types!!!
 - Sadly, actors...

Digression B: Avoid Premature Abstraction (1/2)

- Richer languages/libraries => less temptation for premature abstraction; lots of nice ones already there for you
 - Either/Option/Pattern matching instead of polymorphism
 - Actors instead of hodgepodge of buggy, confusing, low-level concurrency/message-passing/fault tolerance constructs
- More concise languages/libraries => reduced code volume => less need for abstraction (abstraction primarily benefits readers)
- Small code is best code

Part 3: Spray

Spray: What is it?

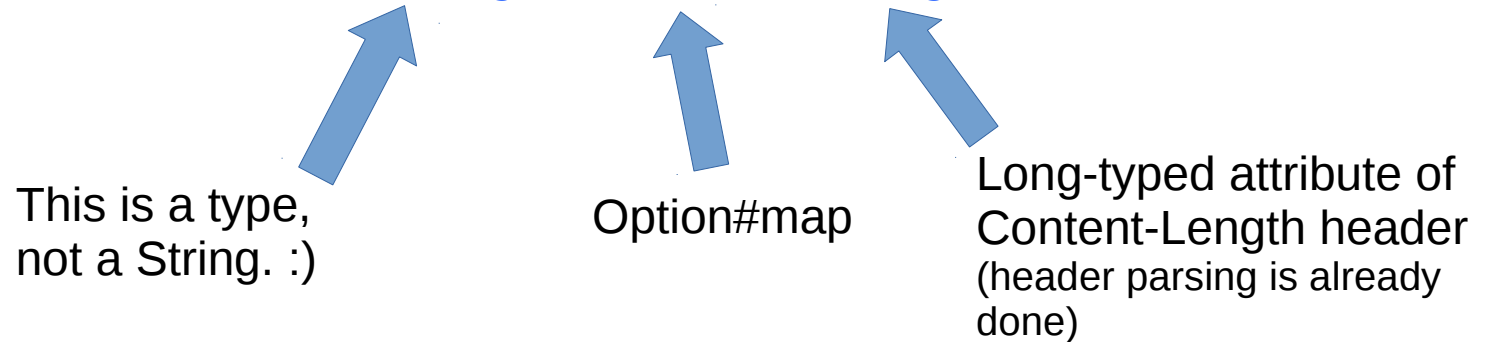
- Asynchronous HTTP client and server library
 - We're ignoring the client today
- Very good performance
- Built with Akka, gets along well with other actors
- Designed to be embedded
 - Not an app server!
- In the process of being adopted, becoming Akka-HTTP
- Not really a web framework *per se*, more of a service framework
 - Generating HTML is passé :)
 - No template system built-in, use whatever you want

Spray: Architecture

- Request and Response are immutable
 - Huge requests and responses use explicit chunking
- Type safety wherever feasible

```
import HttpHeaders.`Content-Length`
```

```
val contentLength: Option[Long] =  
  request.header[`Content-Length`].map(_.length)
```



This is a type,
not a String. :)

Option#map

Long-typed attribute of
Content-Length header
(header parsing is already
done)

Spray: Features

- Built-in SSL support
- Two different ways to build services:
 - Raw actors, for bare metal services
 - REALLY raw. :)
 - Routing DSL, for web apps

Spray: Raw Actors Mode

```
final class MyHandler extends Actor {  
  def receive = {  
    case HttpRequest(HttpMethods.GET,  
                      Uri.Path("/hello"),  
                      headers, _, _) =>  
      // the sender of the request *is* the connection  
      sender() ! HttpResponse(  
        StatusCodes.OK,  
        HttpEntity(  
          MediaTypes.`text/html`,  
          "<h1>Hi there!</h1>"  
        )  
      )  
  }  
}
```

Spray: Routing DSL Mode

- Extremely simplistic example

```
object Main extends App with SimpleRoutingApp {  
  implicit val system = ActorSystem("my-system")  
  
  startServer(interface = "localhost", port = 8080) {  
    path("hello") {  
      get {  
        complete {  
          <h1>Say hello to spray</h1>  
        }  
      }  
    }  
  }  
}
```

Spray: Routing DSL Directives

- Many, many, many, many, many, many, many built-in features
 - static file serving, partial GETs, directory listings
 - caching, compression
 - logging
 - request/response encoding and decoding
 - cookies
 - form fields, URL parameters
- Way too many to cover in a day, let alone an hour

Spray: Selected Directives

- path, pathPrefix, pathEnd
- get/put/post/delete
- complete
- <http://spray.io/documentation/1.2.1/spray-routing/predefined-directives-alphabetically/>

Spray: Exercise

- Build a web app that drives the Guess the Number actor

Digression C: State is Just a Library in Scala (1/2)

- Shared mutable state is hard, make someone else do it
- Use STM for unavoidable sharing
- Actors, become not mutate
 - bucket brigade—each member only knows how to handle his own bucket of state
 - Akka is Just A Library (so were the original scala-actors)

Digression C: State is Just a Library in Scala (2/2)

- Make the database handle the long-lived mutations for you, it's good at that
- Immutable domain model is a huge win
 - In-memory objects can't be inconsistent, only out of date
 - Share freely, never defensively copy

Part 4: Spray-JSON

Spray-JSON: What is it?

- A JSON serialization/deserialization library
- Not coupled to Spray HTTP at all
- Ser/des formats are totally customizable
 - But the default formats are fine and easy to use
 - Keep the formats out of your domain model classes
- Easy round-tripping between domain objects, AST, and bytes/strings

Spray-JSON: AST

- JsValue
 - JsObject
 - JsArray
 - JsNull
 - JsString
 - etc.

Spray-JSON: Example Domain

```
case class Person(name: String, age: Int)
case class Family(surname: String,
                  people: Seq[Person])
case class Dwelling(address: Address,
                    families: Seq[Family])
case class Address(street1: String,
                  street2: Option[String],
                  city: String,
                  province: String,
                  postalCode: String,
                  country: String)
```

Spray-JSON: Parsing & Printing

- Parsing

- JsonReader typeclass

```
trait JsonReader[T] {  
    def read(json: JsValue): T  
}
```

- Take an AST node, produce a T

Spray-JSON: Parsing & Printing

- Printing
 - JsonWriter typeclass

```
trait JsonWriter[T] {  
  def write(obj: T): JsValue  
}
```
 - Take a T, produce an AST node

Spray-JSON: Manual Parsing Horrors

- Here's one horrible way to do it. This is only the parser, not the printer. :(

```
object PersonReader extends JsonReader[Person] {  
  def read(json: JsValue): Person = json match {  
    case JsObject(fields) =>  
      fields.get("name") match {  
        case Some(JsString(name)) =>  
          fields.get("age") match {  
            case Some(JsNumber(age)) =>  
              Person(name, age.toInt)  
            case None =>  
              sys.error("age is required")  
          }  
        case None =>  
          sys.error("name is required")  
      }  
    case other =>  
      sys.error(s"Can't deserialize a Person from a $other")  
  }  
}
```

Spray-JSON: Manual Parsing Horrors

- Here's a slightly less horrible way to do it. This is only the parser, not the printer. :(

```
object PersonReader extends JsonReader[Person] {  
  def read(json: JsValue): Person = json match {  
    case JsObject(fields) =>  
      val mPerson: Option[Person] = for {  
        JsString(name) <- fields.get("name")  
        JsNumber(age) <- fields.get("age")  
      } yield Person(name, age.toInt)  
  
      // One problem with using Option here is we  
      // can't tell which field was missing  
      mPerson.getOrElse(sys.error("A required field was  
missing"))  
  
    case other =>  
      sys.error(s"Can't deserialize a Person from a $other")  
  }  
}
```

Spray-JSON Parsing


- And here's the nice way!

```
import spray.json.DefaultJsonProtocol.  
implicit val personFormat = jsonFormat2(Person)  
implicit val familyFormat = jsonFormat2(Family)  
// etc...
```

Number of fields in this class



Companion
Object
(w/apply
method)



- The JsonWriter is also included in the default format
- Only a teensy bit of reflection used, to get the constructor parameter names

Spray-JSON Usage: Parsing

- Assuming the implicit JsonFormats are in scope...

```
import spray.json._  
import spray.json.DefaultJsonProtocol._  
  
// get the string from somewhere  
val personJson: String = ???  
  
// parse to AST  
val personAst: JsValue = personJson.asJson  
  
// convert to Person  
val person: Person = personAst.convertTo[Person]
```

Spray-JSON Usage: Printing

- Again, assuming the implicit JsonFormats are in scope...

```
import spray.json._
import spray.json.DefaultJsonProtocol._

// get the Person from somewhere
val person: Person = ???

// convert to AST
val personAst: JsValue = person.toJson

// convert to String
val personJson: String = personAst.compactPrint
                          // or .prettyPrint
```

Spray-JSON: Problems

- Exclusively uses exceptions for error handling :(ul> - I've filed a feature request to add Either/Validation support
- Parsing performance is not stellar on large documents; should be fixed soon
- No streaming API (entire document must fit in memory, unsuitable for huge documents)

Spray-JSON: Exercise

- Build a new REST front-end for the Guess a Number game
 - JSON in and out
 - Test with cURL or what have you

Digression D: Location Transparency

- Want to be able to shift from local to distributed, without distorting our architecture beyond recognition
- e.g. move from high-efficiency, all-in-one-VM server to high-throughput, big cluster
- Akka
 - ActorRef abstraction is powerful
 - local is a performance optimization, everything is async message passing anyway
 - remote is slower and more likely to fail, but the failures exhibit the same way (timeouts happen anyway... Might be due to bugs, load, crashes, squirrels, EBS being pissy, high load cloud-neighbours)
 - clustering decouples topological decisions from our core code
 - Configure it only at the entry points, core service code shouldn't have to care (except the distribution of failure modes might shift around)

Database Access (sorry... :()

- Async/non-blocking DB access layer?
 - I'd like to be able to recommend one, but I ran out of time to do any real research
 - Also, it's pretty bleeding edge
- Hibernate/JPA works, although it's ugly
- I like MapperDAO, but if you're picky about both your object model and your relational model, you'll have trouble
- Slick isn't an ORM, but has a nice query abstraction
- Most NoSQL databases have reasonably good Scala clients

ASK ALL THE QUESTIONS!

- Oh, to make up for being a little underprepared and sleep-deprived...
 - You get a car! And YOU get a car!
 - I mean...
 - Two hours free email consulting
 - Or, buy me lunch/dinner/beer and I'll hang out with your team and let you pepper me with questions for an hour or two