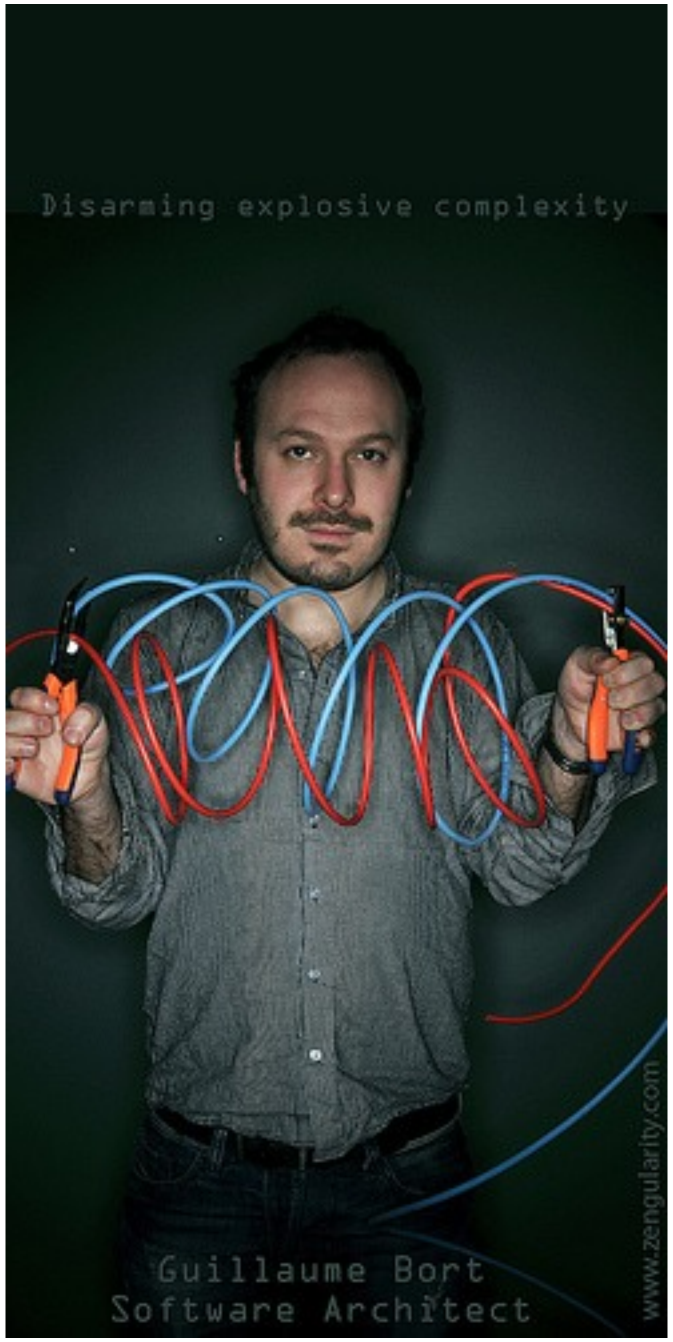
play!

I’ll see your async and raise you reactive

@sadache



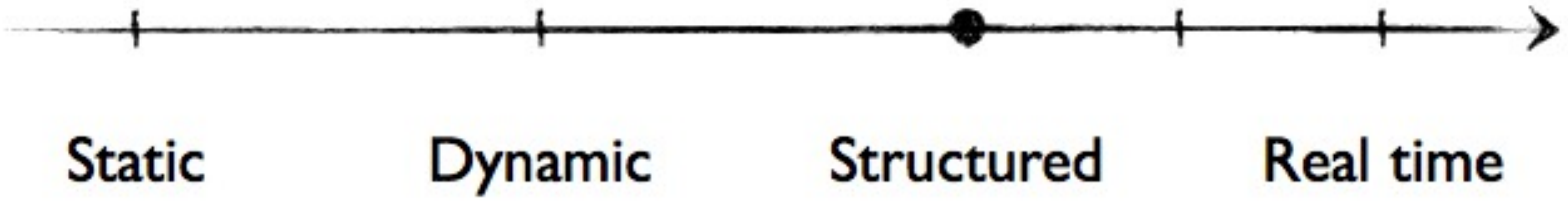
@guillaumebort

@zenexity

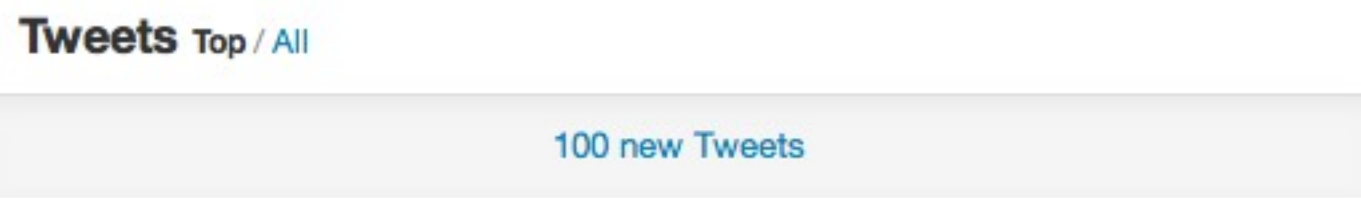
What is Play?

- The Play framework makes it easier to build web applications with Java & Scala.
- Play is based on a lightweight, stateless, web-friendly architecture for highly-scalable applications - thanks to its reactive model, based on Iteratee IO.

The Web Evolved

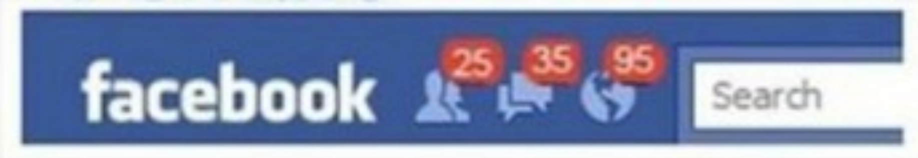


Flows of data everywhere



Tweets Top / All

100 new Tweets



Flows of data everywhere

- Polling, Server Sent Events, Comet, Websockets
- Data upload

And it changes everything

- Handle continuous flows of data
- Improve expressiveness for concurrent code
- Scale vertically and horizontally

With Play1.x

- We decided not to go WAR!
- one request one thread doesn't scale
- Hint: a thread doesn't need to be blocked when doing IO

Java.io.InputStream

public abstract int **read**() throws IOException

Reads the next byte of data from the input stream. [...] This method blocks until input data is available,

Limitations of Java’s current streams model

- Resources consumption (memory, threads, disk)
- Pro-active waiting

A reactive model

- Inversion of control
- The source controls the execution
- Without loosing control

Play2

- Composable streams handling with Iteratees
- Reactive
- Control

An Iteratee

- Is the Consumer
- Represents a state of processing input, can be either:
- Done with a computed value and input left from last chunk
- Cont with a function which represents the way you push more Input
- Error with a message and the input which caused that error

An Iteratee

```
def fold[B](
  //Done with a computed value and input left from last chunk
  done: (A, Input[E]) => Promise[B],
  //Cont with a function which represents the way you push
more Input
  cont: (Input[E] => Iteratee[E,A]) => Promise[B],
 //Error with a message and the input which caused that error
  error: (String, Input[E]) => Promise[B] ): Promise[B]
```

Input

- Represents chunks of input that will be passed into an iteratee, can be either of:
- El(), EOF, Empty

A Done Iteratee

```
   val doneIteratee = new Iteratee[String,Int] {
  def fold[B](
    done: (A, Input[E]) => Promise[B],
    cont: (Input[E] => Iteratee[E, A]) => Promise[B],
    error: (String, Input[E]) => Promise[B]): Promise[B]
= done(1,Input.Empty)
}
```

An Iteratee

```
def fold[B](
  done: (A, Input[E]) => Promise[B],
  cont: (Input[E] => Iteratee[E,A]) => Promise[B],
  error: (String, Input[E]) => Promise[B] ): Promise[B]
```

Should I write all of that for creating a simple Iteratee?

- Iteratee constructors for various Input consuming scenarios

foreach Iteratee

```
val printlnIteratee: Iteratee[String,Unit] =
Iteratee.foreach[String](s => println(s))
```

fold Iteratee

```
  val inputLength: Iteratee[Array[Byte],A] =
  Iteratee.fold[Array[Byte],Int](0)
{ (length, bytes) => length + bytes.size }
```

consume Iteratee

```
val consume = Iteratee.consume
[String]()
```

But how to push data into an Iteratee?

- Enumerators are the input source (producer)
- Socket In, File, Events

An Enumerator

```
trait Enumerator[E] {

     /**
      * Apply this Enumerator to an Iteratee
      */
      def apply[A](i: Iteratee[E, A]): Promise[Iteratee[E, A]]

  }
```

Fold again and again?

```
def fold[B](
  done: (A, Input[E]) => Promise[B],
  cont: (Input[E] => Iteratee[E,A]) => Promise[B],
  error: (String, Input[E]) => Promise[B] ): Promise[B]
```

Fold again and again?

- Use Enumerator Constructors

List Enumerator

```
 val enumerateUsers: Enumerator[String] = {

  Enumerator("Guillaume", "Sadek", "Peter", "Erwan")
}

val consume = Iteratee.foreach[String](s => println(s))

enumerateUsers(printLn)

//or

enumerateUsers |>> printLn
```

Callback Enumerator

```
   Enumerator.fromCallback { () =>
     Promise.timeout(Some(new Date), 100
milliseconds)
    }
```

Callback Enumerator

```
  def fromCallback[E](
  retriever: () => Promise[Option[E]],
  onComplete: () => Unit = () => (),
  onError: (String, Input[E]) => Unit = (_:
String, _: Input[E]) => ()
): Enumerator[E] = {
  ...
}
```

Callback Enumerator

```
  val timeStream = Enumerator.fromCallback { () =>
  Promise.timeout(Some(new Date), 100 milliseconds)
}

val printlnSink = Iteratee.foreach[Date](date =>
println(date))

timeStream |>> printlnSink
```

Push Enumerator

```
val channel = Enumerator.pushee[String]
{ onStart = pushee =>

  pushee.push("Hello")
  pushee.push("World")
}

channel |>> Iteratee.foreach(println)
```

Enumerators à la carte

```
   object AvailableStreams {

  val cpu: Enumerator[JsValue] = Enumerator.fromCallback( /* code here */ )

  val memory: Enumerator[JsValue] = Enumerator.fromCallback( /* code here
*/ )

  val threads: Enumerator[JsValue] = Enumerator.fromCallback( /* code here
*/ )

  val heap: Enumerator[JsValue] = Enumerator.fromCallback( /* code here */ )

}

val physicalMachine = AvailableStreams.cpu >- AvailableStreams.memory
val jvm = AvailableStreams.threads >- AvailableStreams.heap

def usersWidgetsComposition(prefs: Preferences) = {
  // do the composition dynamically
}
```

And adapters: Enumeratees!

```
   val sum: Iteratee[Int,Int] = Iteratee.fold[Int,Int](0)
   { (s,e) => s + e }

val strings: Enumerator[String] = Enumerator("1","2","3","4")

//create am Enumeratee using the map method on Enumeratee
val toInt: Enumeratee[String,Int] = Enumeratee.map[String]{ s
=> s.toInt }

val adaptedIteratee: Iteratee[String,Int] = toInt.transform
(sum)

//this works!
strings |>> adaptedIteratee

//or
strings &> toInt >>> sum
```

Where to use Iteratees, Enumerators and Enumeratees?

File Upload

```
  trait Action[A] extends (Request[A] =>
  Result) {
  def parser: BodyParser[A]
}

trait BodyParser[+T] extends (RequestHeader
=> Iteratee[Array[Byte],T] )
```

Streaming

- Streaming big files to the client
- Http 1.1 Chunking and Comet
- Websockets, Server Sent Events