LUNATECH

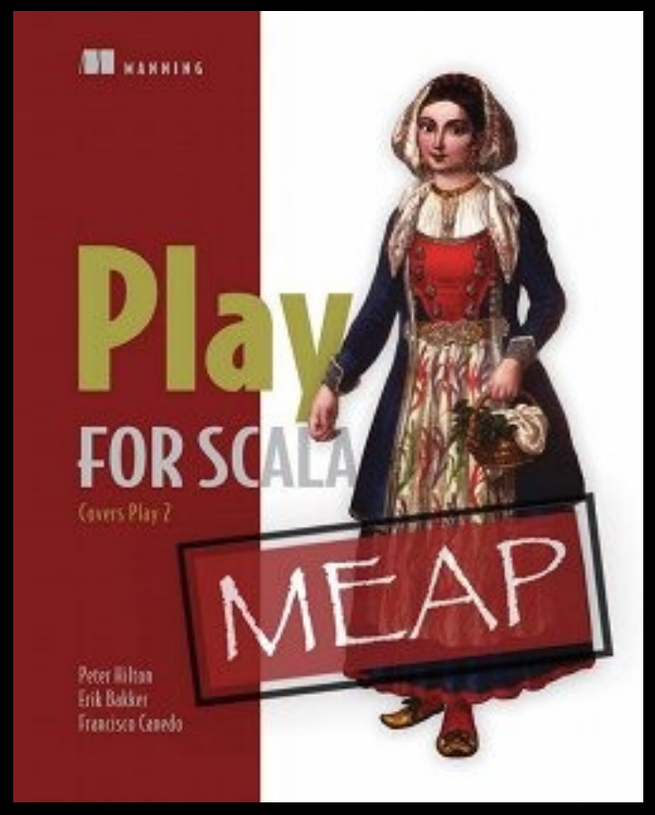
RESEARCH

Core Play 2

Erik Bakker - DuSE - 14-6-2012

About me

- Working at Lunatech in Rotterdam
- Doing a project with several Play 2.0 apps
- Co-authoring ‘Play for Scala’



LUNATECH RESEARCH

Don’t forget to Google for a coupon!

Today's Topics

- What is Play 2
- Requests and responses
- Iteratees
- Websockets
- Hands-on

LUNATECH
RESEARCH

Before we start

- Clone the repo on http://github.com/eamelink/core-play
- Run ‘sbt’ in that directory

LUNATECH
RESEARCH

You need SBT 0.11.2. It may work with 0.11.3 if you update project/build.properties accordingly. If you get errors resolving commons-codec or slf4j, remove ~/.ivy2/cache/commons-codec and ~/.ivy2/cache/org.slf4j These errors are caused by Play 1, corrupting your Ivy cache...

Part One

What is Play 2

LUNATECH
RESEARCH

What is Play 2

- A “Full Stack” web framework
  - Netty server
  - Request router
  - Template engine
  - Compiler
- Stateless (sort of)
- You just need a machine with Java

LUNATECH
RESEARCH

Sort of stateless, because we will see how we can keep thousands of connections open.

Play 2 project itself

- Six Scala SBT sub projects:
  - Anorm
  - Console
  - Play
  - Play-Test
  - SBT-Plugin
  - Templates

LUNATECH
RESEARCH

Play is now just an SBT project. You don't need to download the distribution, except for easy debugging, and creating a new application.

Anorm

- Wrapper for JDBC
  - Works with Strings containing SQL queries
  - Parses results with parser combinators

LUNATECH
RESEARCH

First part very non-Scalaish, second part very Scalaish.

Anorm

```
def spokenLanguages(countryCode: String): Option[SpokenLanguages] = DB.withConnection
{ implicit conn =>
    val languages: List[(String, String)] = SQL(
      """
      select c.name, l.language from Country c
      join CountryLanguage l on l.CountryCode = c.Code
      where c.code = {code};
    """)
      .on("code" -> countryCode)
      .as(str("name") ~ str("language") map (flatten) *)

    languages.headOption.map { f =>
      SpokenLanguages(f._1, languages.map(_._2))
    }
  }
```


LUNATECH
RESEARCH

Don’t use Anorm. The reasons for non type safe DSL are not convincing. Squeryl is great, although it has some limitations. It’s fine to use Squeryl for the majority of your app and do the few ninja-SQL queries you need with Anorm.

Console

- Provides the Play ‘new’ command, which copies an application skeleton to your new app directory
- Also holds the ASCII art logo:

```
val logo = Colors.yellow(

    """|       _                 _
       | _ __ | | __ _ _  _| |
       || '_ \| |/ _' | || |_|
       ||  __/|_|\____|\__ (_)
       ||_|             |__/
       |
       |""".stripMargin) + ("play! " + play.core.PlayVersion.current + ",
                           http://www.playframework.org")
```


LUNATECH
RESEARCH

SBT Plugin

- Compiles and runs your app
- Compiles CoffeeScript and LESS to Javascript and CSS, and also minifies and gzips them
- Generates ‘dist’ package: a zip file with your app, all dependencies and a start script.
- Provides an ‘SBTLink’ to the Play application
  - App can reload themself if the sources changed

SBT Plugin

- Compiles templates and routes
  - From Template and Route syntax to Scala sources
  - From Scala sources to byte code
  - Error positions are mapped back to original source positions

Templates

- Type-safe scala-like template syntax

```
@(users: Seq[User])

@main("User list") {
  <h1>List of users</h1>
  <ul>
  @for(user <- users) {
    <li>
      This user has name @user.name,
     and he is @user.age years old
    </li>
  }
  </ul>
}
```


LUNATECH
RESEARCH

Type signature same as in Scala
Creates an object, with name inferred from template filename

Templates

- Parsing of the templates is done with parser combinators

```
def parentheses: Parser[String] = {
  "(" ~ (several((parentheses | not(")") ~> any))) ~ commit(")") ^^ {
    case p1 ~ charList ~ p2 => p1 + charList.mkString + p2
  }
}
```

```
def expression: Parser[Display] = {
  at ~> commit(positioned(methodCall ^^ { case code => Simple(code) })) ~ several(expressionPart) ^^ {
    case first ~ parts => Display(ScalaExp(first :: parts))
  }
}
```


LUNATECH
RESEARCH

Just two examples.

Templates

- Get compiled into Scala code:

```
object userlist extends BaseScalaTemplate[Html,Format[Html]](HtmlFormat) with Template1[Seq[User],Html]

    /**/
    def apply/*1.2*/(users: Seq[User]):Html = {
        _display_ {

Seq[Any](format.raw/*1.20*/("""

"""),_display_(Seq[Any](/*3.2*/main("User list")/*3.19*/ {_display_(Seq[Any](format.raw/*3.21*/("""
  <h1>List of users</h1>
  <ul>
  """),_display_(Seq[Any](/*6.4*/for(user <- users) yield /*6.22*/ {_display_(Seq[Any](format.raw/*6.24*
("""
    <li>
      This user has name """),_display_(Seq[Any](/*8.25*/user/*8.29*/.name)),format.raw/*8.34*/("""",
      and he is """),_display_(Seq[Any](/*9.18*/user/*9.22*/.age)),format.raw/*9.26*/(""" years old
    </li>
  """)))})),format.raw/*11.4*/("""
  </ul>
"""))})))}
    }
```


LUNATECH
RESEARCH

Object name from template file name. Apply signature from template signature.
Note the source position annotations. Sequence of ‘Any’, and a _display_ method that knows how to display those things.

Templates

- The _display_ method:

```
case class BaseScalaTemplate[T <: Appendable[T], F <: Format[T]](format: F) {

    def _display_(o: Any)(implicit m: Manifest[T]): T = {
      o match {
        case escaped if escaped != null && escaped.getClass == m.erasure =>
          escaped.asInstanceOf[T]
        case () => format.raw("")
        case None => format.raw("")
        case Some(v) => _display_(v)
        case xml: scala.xml.NodeSeq => format.raw(xml.toString)
        case escapeds: TraversableOnce[_] =>
          escapeds.foldLeft(format.raw(""))(_ + _display_(_))
        case escapeds: Array[_] => escapeds.foldLeft(format.raw(""))(_ + _display_(_))
        case string: String => format.escape(string)
        case v if v != null => _display_(v.toString)
        case _ => format.raw("")
      }
    }

}
```


LUNATECH
RESEARCH

Pattern matching on the element. The template gets a “Format” determines by the extension of the file. In this case, “HtmlFormat”. Takes care of escaping suitable for Html. “XmlFormat” does escaping for Xml, etcetera.
But are we going to use the template engine? I don’t often do. I build a REST api with Play, and a Backbone.js app on the frontend.

Play-Test

- Fake objects
  - FakeRequest
  - FakeApplication
  - FakeHeaders
- Helpers to check responses
- Wrapper around Selenium

LUNATECH
RESEARCH

Main dish:

Play

LUNATECH
RESEARCH

Play 2 stats

- 97 Scala files
- 10402 lines of code
- 4823 lines of comment
- 3182 blank lines

- + 5222 lines of code for the Java API
- But we don’t care

LUNATECH
RESEARCH

Compared to 31076 lines of Java for Play 1

What is Play 2

- An application:

```
object NettyServer

  ...

  def main(args: Array[String]) {
    args.headOption.orElse(
      Option(System.getProperty("user.dir"))).map(
        new File(_)).filter(p => p.exists && p.isDirectory).map { applicationPath =>
        createServer(applicationPath).getOrElse(System.exit(-1))
      }.getOrElse {
        println("Not a valid Play application")
      }
  }
}
```


LUNATECH
RESEARCH

Just to show you that Play is an application...

Play and Netty

- Play has a built-in HTTP server, Netty:
  - Based on NIO
  - Many connections per thread
  - Highly efficient

LUNATECH
RESEARCH

NIO, introduced in Java 1.4 reached the age of 10 in February this year...

Play's main philosophies

Handle many connections

Process datastreams reactively, one chunk at a time

LUNATECH
RESEARCH

We want thousands of clients to be connected to our systems. We can send notifications, streaming data, live updates to our users. Or have them chat.
Often, requests never end. We are continuously streaming data. We don't want to buffer stuff.

How to handle many connections

- Easy, Netty does it for you
- Caveat: we can't use a Netty thread for too long

LUNATECH
RESEARCH

Netty creates a single 'boss' thread per port it opens, and handles incoming requests to one of a pool of 'worker' threads.

How to process data stream reactively

- We generally can't buffer full requests
- Instead, we work with chunks of data
  - Something produces chunks
  - Something else consumes these chunks
- In case of HTTP requests, these chunks are Array[Byte], but the abstraction is useful in general

LUNATECH
RESEARCH

Chunks may be strings, in a one-on-one chat. But it's likely JSON. Or maybe HTML. Chunks can be anything, from very low level byte arrays to rich objects.

How to process data stream reactively

- Play ships a library “iteratee”, with immutable implementations of a consumer and producer
- Iteratee = Consumer
- Enumerator = Producer
- We’ll get back to this later, but remember this

LUNATECH
RESEARCH

Similar to Iteratee in Haskell and Scalaz

Part Two

Requests and Responses

LUNATECH
RESEARCH

A web application maps an HTTP request
to an HTTP response

LUNATECH
RESEARCH

In Play, we map a Request[A], with body type A,
to a Result

LUNATECH
RESEARCH

So, what are the building blocks we use for this in Play?

Requests

- A Request has a type parameter A, the type of the body
- An HTML request body is always Array[Byte]
- So the HTML request body must be parsed into A, where A
- That is done using a BodyParser[A]
- A BodyParser[A] constructs an Iteratee[Array[Byte], A]

Results

- Results are constructed from
  - An HTTP status code
  - A body of type T or an Enumerator[T]
  - Implicit parameters Writable[T] and ContentTypeOf[T]

Actions

- An Action defines how to map an HTTP request to an HTTP response
- An Action has two properties:
  - A BodyParser[A]
  - Function from Request[A] to Result

How it looks: constructing results

```
// Creating a result with status code 200 and no body
Status(200)


// Shortcut for the above
Ok


// Adding a non-chunked body
Ok("Hello DUSE!")


// The above is similar to
Ok("Hello DUSE!")(Writeable.wString(Codec.utf_8), ContentTypeOf.contentTypeOf_String(Codec.utf_8))
```


How it looks: Actions and using BodyParsers

```
object Requests extends Controller {

  // Default body parser
  def actionOne() = Action { request =>
    Ok("Hello world!")
  }

  // Explicit body parser
  def actionTwo() = Action(parse.json) { request =>
    // We now have request.body of type JsValue
    val json = request.body
    Ok
  }

  // Custom body parser of type BodyParser[Unit]
  def actionThree() = Action(BodyParsers.loggingDiscardingBodyParser) { request =>
    Ok
  }

}
```


LUNATECH
RESEARCH

You can find the loggingDiscardingBodyParser in the sample code...

But what do we do when the result is really large?

Or when the request is really large?

LUNATECH
RESEARCH

HTTP

LUNATECH
RESEARCH

HTTP 1.0

- Connection was closed at the end of the request
- Content-Length header optional
- Streaming data without knowing how much it's going to be is possible
- Performance overhead of opening new connections for every resource

LUNATECH
RESEARCH

If there is no Content-Length header, the request is done when the server closes the connection.

HTTP 1.1

- Connection is kept alive after an HTTP request, and reused for other HTTP requests
- Without Content-Length header, client doesn’t know when all data is received
- Problem when streaming data without known size

Solution: HTTP Chunked transfer encoding

- Send a stream of unknown length in chunks
- Add a size indicator before each chunk
- When done, send an empty chunk

HTTP Demo

LUNATECH
RESEARCH

Do-it-yourself-demo

- Run the project, and with ‘telnet localhost 9000’, do HTTP 1.0 and 1.1 requests to:
  - /responses/simple-without-header
  - /responses/simple-feeding
  - /responses/chunked-result

LUNATECH
RESEARCH

To make a request, just type: “GET /responses/simple-without-header HTTP/1.0”. Replace 1.0 with 1.1 for an HTTP 1.1 request. May not be according to the specs, but Netty understands what you mean ;)

So we will have to output our large data in chunks

LUNATECH
RESEARCH

Part Three

Iteratees

LUNATECH
RESEARCH

Iteratee

- Iteratee[E, A] = Consumer that consumes chunks of type E, and will eventually produce an A
- Iteratee is in one of three states
  - Cont (continue, it's still accepting input)
  - Done
  - Error

LUNATECH
RESEARCH

Actually producing something is optional. Maybe we just want to log something. Or send something elsewhere. But it may be useful, for example if you stream to a bucket at S3, you can return the ID of what you just put there.

Iteratee

- The Done state contains the remaining input, and the result
- The Cont state contains a continuation, a method that you can feed a new chunk and it will produce the new Iteratee state
- The Error state contains an error string, and the remaining input

LUNATECH
RESEARCH

Has some similarities to parsers, they also contain not consumed input.

Iteratee

- An Iteratee[E, A] consumes Input[E], and they can be one of three input chunks:
  - El[E], a chunk with data
  - Empty, an empty chunk
  - EOF, indicating the end of the stream

Example Iteratee

```
def counter[E]: Iteratee[E, Int] = {
  def step(s: Int)(i: Input[E]): Iteratee[E, Int] = i match {
    case Input.EOF => Done(s, Input.EOF)
    case Input.Empty => Cont[E, Int](i => step(s)(i))
    case Input.El(e) => Cont[E, Int](i => step(s + 1)(i))
  }

  Cont[E, Int](i => step(0)(i))
}
```


LUNATECH
RESEARCH

Iteratee that counts all chunks it receives, and produces that number. It gets in the ‘Done’ state when it receives an Input.EOF. It starts in the Cont state, with ‘0’ as the internal state.

Provided Iteratees

```
// Consume and concatenate input
val i1 = Iteratee.consume[String]

// Folding iteratee, this one sums the chunks
val i2 = Iteratee.fold(0)((state, chunk: Int) => state + chunk)

// Apply method to each chunk
val i3 = Iteratee.foreach((chunk: Any) => println(chunk))

// Ignore input...
val i4 = Iteratee.ignore
```


Feeding data into an Iteratee

- Similar to an Either, an Iteratee can be in one of multiple states.
- Use the ‘fold’ method on Iteratee to unify into a single type:

```
trait 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]
}
```


LUNATECH
RESEARCH

So the ‘fold’ method is used by the thing that puts data into the Iteratee.

Feeding data into an Iteratee

```
def feed[E, A](iteratee: Iteratee[E, Int], values: Seq[E]): Promise[Int] = {
  iteratee.fold(
    done = (result, remainingInput) => Promise.pure(result),
    cont = (consumeMethod) => {
      val newIteratee = consumeMethod(values.headOption.map(Input.El(_)).getOrElse(Input.EOF))
      feed(newIteratee, if(values.isEmpty) Nil else values.tail)
    },
    error = (error, remainingInput) => Promise.pure(-1))
}
```


LUNATECH
RESEARCH

This is just an example, you generally won't use fold, but an Enumerator to feed data into an Iteratee. Also, this thing is recursive, but not tail-recursive so it will break pretty quickly...

Enumerator

- An Enumerator[E] produces chunks of type E
- An Enumerator[E] can be applied to an Iteratee[E, A]
- Enumerators can be composed
- Some useful Enumerators are provided

LUNATECH
RESEARCH

Creating Enumerators

```
// From a set of values
val e1 = Enumerator("Hello! ", "these", "are", "the", "elements")

// From a file
val e2 = Enumerator.fromFile(file)

// An imperative 'PushEnumerator'
val e3 = Enumerator.imperative[String]()
e3.push("Foo")
e3.push("Bar")

// From a callback
val e4 = Enumerator.fromCallback { () =>
  Promise.timeout(Some("Hello!"), 1 seconds)
}
```


Enumerator composition and modification

```
// Interleave
val e5 = e1 interleave e3

// Concatenate
val e6 = e1 andThen e3

// Transform chunks
val e7 = e1.map(_.toInt)
```


LUNATECH
RESEARCH

There are aliases for all these composition methods, like >- for interleave (looks like a zipper).

Enumeratees

- Enumeratees can transform and adapt streams
- Can be composed with Enumerators using ‘through’
- Can be composed with Iteratees using ‘transform’
- Can be composed with other Enumeratees
- You can get the original enumerator or iteratee out again

Enumeratees

- Many ways to construct Enumeratees:
  - Enumeratee.filter
  - Enumeratee.map
  - Enumeratee.collect
  - Enumeratee.drop, take, dropwhile, takeWhile
  - Enumeratee.mapInput (map the entire Input[E] object)

LUNATECH
RESEARCH

Iteratees and Enumerators in Play

- Enumerators
  - To create the body of a Result
  - To create the body of a request to a webservice
- Iteratees
  - To consume the body of a Request
  - To consume the result of a webservice call
- Both: for WebSockets (we'll get to that)

LUNATECH
RESEARCH

They're not in the default Database API, which is a bit odd

Mini demo

- Run project, and with ‘telnet localhost 9000’, try while observing the console:

```
POST /bodyparsers/logging-body-parser HTTP/1.1
Transfer-Encoding: chunked

5
Hello
F
RingRingRingRin
0
```


The chunk length is hexadecimal. So F = 15 bytes.

So, how does the request lifecycle look in Play?

Simplified Request lifecycle

- A request comes in
- Single Netty ‘boss’ thread accepts it and hands over to one of a bunch of Netty ‘worker’ threads
- Worker thread calls messageReceived(ctx: Context, e: MessageEvent), we enter Play code
- A RequestHeader object is created for the request
- The router is consulted for a matching route and it returns an Action

LUNATECH
RESEARCH

Action is combination of a function from Request[A] to Response, and a BodyParser[A]

Simplified Request lifecycle

- In an Akka actor, the BodyParser is applied to the request headers, which creates an Iteratee that takes Array[Byte] input and creates an A
- If the request is not chunked, a single element Enumerator is created from the Http request body, and applied to the Iteratee
- If the request is chunked, all incoming chunks are manually fed to the Iteratee when they arrive

LUNATECH
RESEARCH

Manual feeding of chunks to the iteratee probably because of performance reasons.

Simplified Request lifecycle

- When the body parsing Iteratee delivered its A, a Request[A] object is constructed, and sent to an Akka actor together with the Action and a Response object.
- This Response object has a ‘handle’ method, that handles various Result types that an action can return:
  - SimpleResult
  - ChunkedResult
  - AsyncResult (which wraps a promise of a result)

LUNATECH
RESEARCH

Body can be complete for two reasons: Full Http body is retrieved, or Iteratee reaches ‘Done’ state earlier

Request lifecycle take-away points

- Routing done with Http request headers
- Bodyparser created in Akka actor
- Body fed to bodyparser until done
- Action method executed in Akka actor
- Result fed back to Netty

Error handling

- Previous slides were simplified
- BodyParser[A] does not create Iteratee[Array[Byte], A], but Iteratee[Array[Byte], Either[Result, A]]
- This means that the BodyParser can return error responses directly, without the Action ever being invoked
- And because it also gets the request headers, it can do everything it wants with the request!

LUNATECH
RESEARCH

The Either[Result, ...] pattern is used in a lot of places in the Play core.

Part 4

WebSockets

LUNATECH
RESEARCH

Websockets

- Bi-directional communication
- Long-living connections
- “Upgrade” of HTTP protocol
- Allows servers to push data to clients
- Multiple messages / frames

LUNATECH
RESEARCH

Upgrade of HTTP means that it will go through most firewalls without problem. Works over port 80 and 443, same as HTTP and HTTPS

WebSockets demo

LUNATECH
RESEARCH

How it works in Play

- In a Websocket action, we create
  - An Enumerator, that will produce what we want to send to the client
  - An Iteratee, that will consume what we receive from the client
- We return these, and Play will hook them up to Netty

Websockets in Play

```
def logging() = WebSocket.using[String] { request =>
  println("Connected")
  val in = Iteratee.foreach[String] { msg =>
    println(msg)
  }.mapDone { _ =>
    println("Disconnected")
  }

  val out = Enumerator[String]()

  (in, out)
}


def counter() = WebSocket.using[String] { request =>
  var i = 0;
  val out = Enumerator.fromCallback { () =>
    Promise.timeout({ i += 1; Some(i.toString) }, 1 seconds)
  }

  val in = Iteratee.ignore[String]

  (in, out)
}
```


LUNATECH
RESEARCH

Websockets in Play

```
// A websocket action that combines echoing and counting using enumerator composition
def echoAndCounter() = WebSocket.using[String] { request =>
  var i = 0;
  val counter = Enumerator.fromCallback { () =>
    Promise.timeout({ i += 1; Some(i.toString) }, 1 seconds)
  }

  val echoer = Enumerator.imperative[String]()
  val in = Iteratee.foreach[String] { msg => echoer.push(msg) }

  (in, echoer >- counter)
}
```


Here, we see the >- alias for interleave in action.

EOF

LUNATECH
RESEARCH

En of presentation part. Now, the hands-on part follows.

Part Four

Hands On

LUNATECH
RESEARCH

Hands on!

- Download the sample code from
  http://github.com/eamelink/core-play

- Idea #1:
  Build a WebSocket action, and register each connecting client with an Actor, that sends the load average of the machine to each client every 3 seconds

- Idea #2:
  Allow users to sign up to multiple data streams (cpu usage, heap usage) over a single Websocket connection

Questions, comments?

Twitter: @eamelink

Email: erik@eamelink.net

LUNATECH
RESEARCH