# RestFUL Web Services with Akka HTTP

Daniel Hinojosa

# About Me...

**Daniel Hinojosa**
**Programmer, Consultant, Trainer**

Testing in Scala (Book)
Beginning Scala Programming (Video)
Scala Beyond the Basics (Video)

Contact:
dhinojosa@evolutionnext.com
@dhinojosa

# Slides and Code Available @:

https://github.com/dhinojosa/akkahttp

# Akka Basics

# About Akka

"Swedified" from Áhkká

# Set of Libraries to Create.. Concurrent, Fault Tolerant, and Scalable Applications…now with Restful HTTP

http://akka.io

# Actors, Location Transparency, Finite State Machines, Actor Persistence, Clustering, Agents

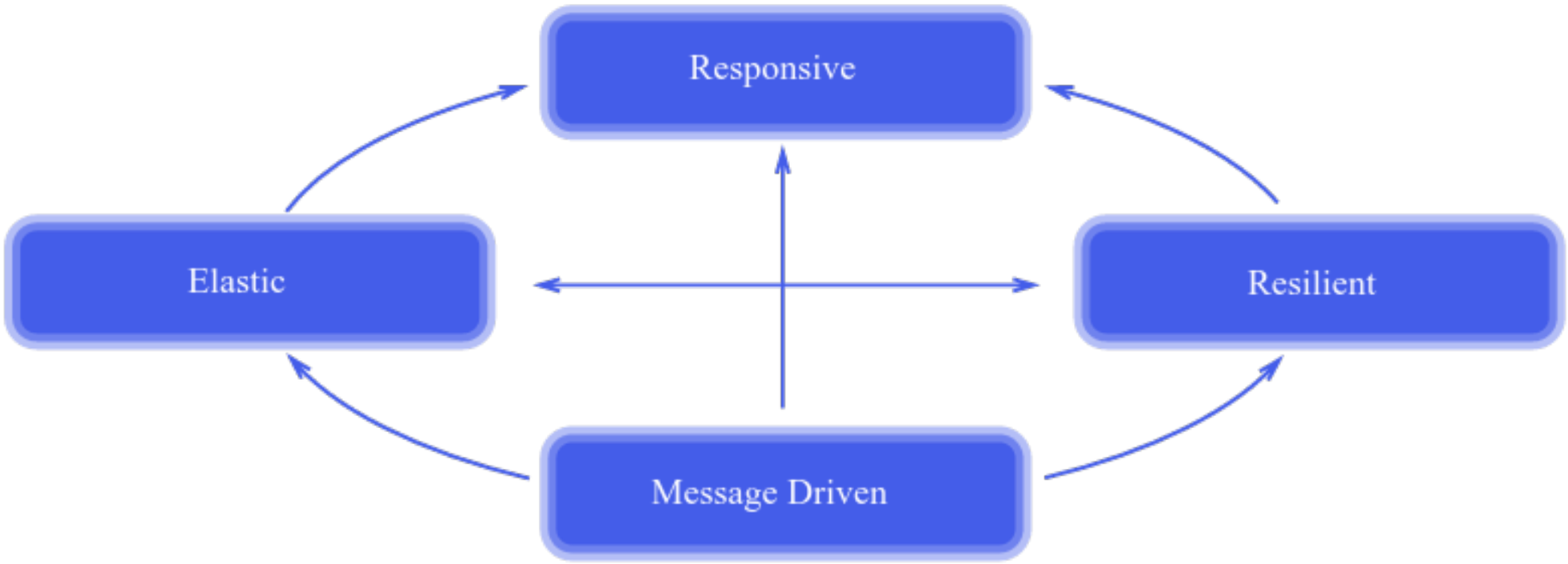
# Same VM or Different VM

# Failure Tolerance

# Easy Scaling - Up or Out

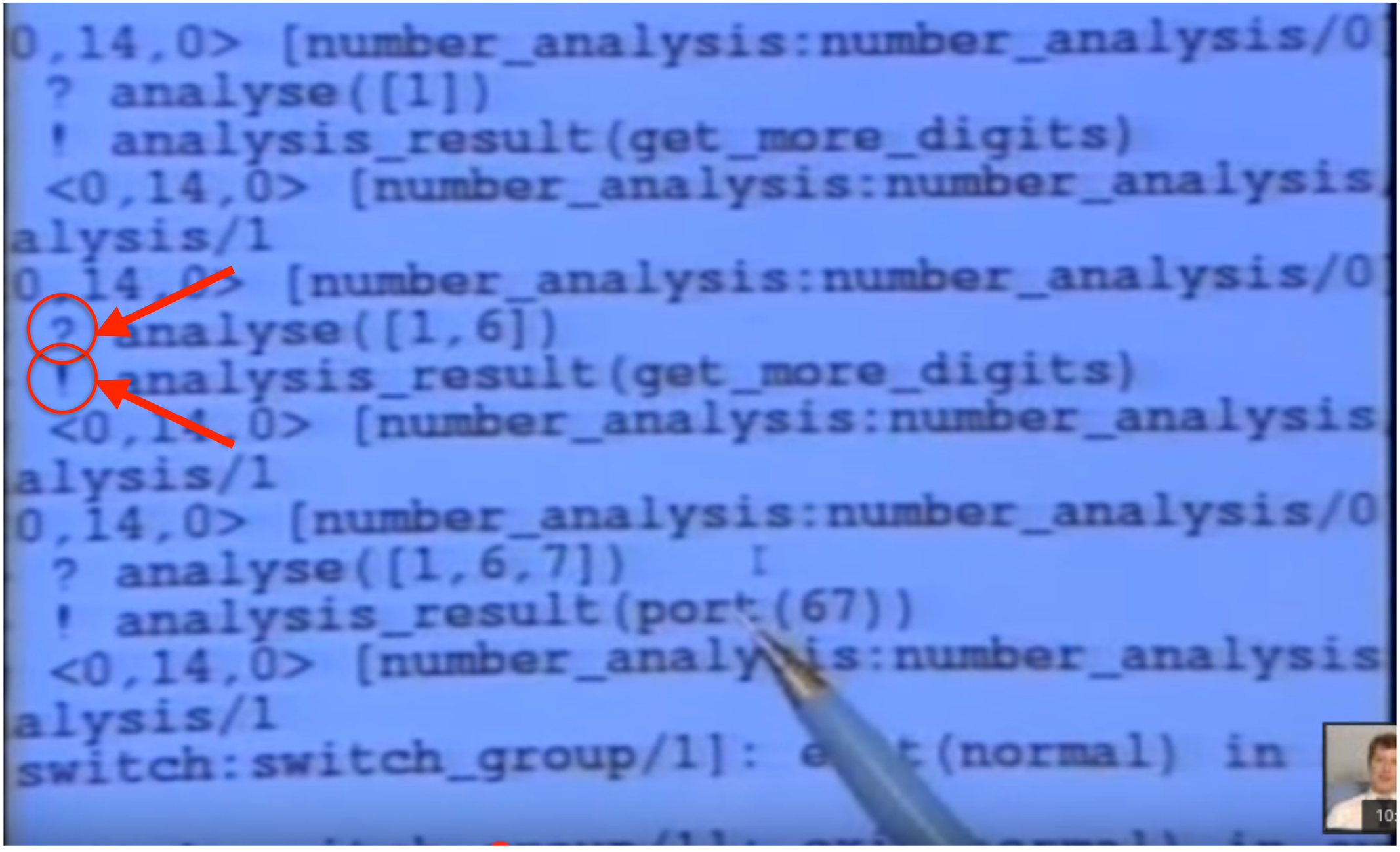
# Microservices

http://www.reactivemanifesto.org/


Responsive
Elastic
Resilient
Message Driven

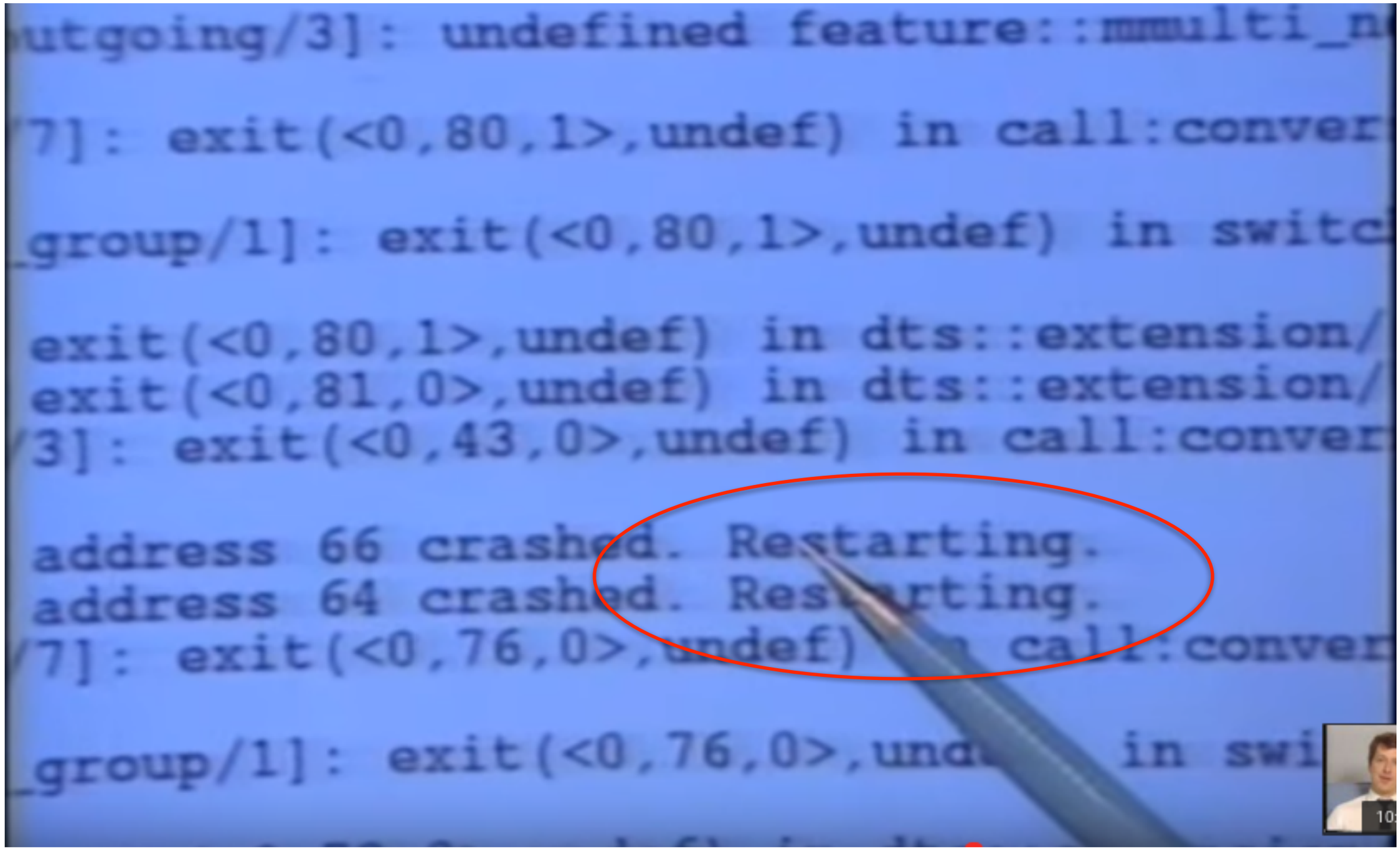
# Akka Actors


ERLANG




# Erlang the Movie!

https://www.youtube.com/watch?v=xrIjfIjssLE




# Erlang the Movie!

https://www.youtube.com/watch?v=xrIjfIjssLE

# Encapsulate State and Behavior

# Concurrent processors that exchange messages

# Inside the Actor's Studio...

To: Matt

To: Matt

To: Matt

# Flooding Matt Damon with Messages

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt

To: Matt


To: Matt


To: Matt


To: Matt

To: Matt

To: Matt

To: Matt

# Demo: Akka Actors

# Akka-Streams

# Demo: Akka Streams

# Akka-Http

# Akka-Http

“Not a framework” based content server that is meant to be on the sidelines.

# Akka-Http

Meant to be a your REST/HTTP Microservice face to the rest of the world

# Akka-Http

Some work derived from Spray.

# Akka-Http

Meant to interact with Akka Actors

# Akka-Http

Higher-level functionality potential for (un)marshalling, (de)compression

# Akka-Http-Core

A complete, mostly low-level, server- and client-side implementation of HTTP

# Akka-Http-testkit

A test harness and set of utilities for verifying server-side service implementations

# Akka-Http-Spray-JSON

Predefined glue-code for (de)serializing custom types from/to JSON with spray-json

# Akka-Http-Spray-XML

Predefined glue-code for (de)serializing custom types from/to XML with scala-xml

# Akka-Http DSLs

DSL for defining HTTP-based APIs on the server-side

# Bringing in Akka-Http

Group Id: `com.typesafe.akka`

Artifact Id: `akka-http`

Version: `3.0.0-RC1`

# Demo:Review Organization of Project

# HTTPModel

# HTTPModel

```
import akka.http.scaladsl.model._
```

# HTTPModel

- `HttpRequest` and `HttpResponse`, the central message model
- `headers`, the package containing all the predefined HTTP header models and supporting types
- Supporting types like `Uri`, `HttpMethods`, `MediaTypes`, `StatusCodes`, etc.
- All objects immutable

# HTTPModel Locations

- `HttpModels` are located in entity with a trailing `s`
- For example:
    - `HttpMethod` instances live in the `HttpMethods` object.
    - `HttpCharset` instances live in the `HttpCharsets` object.
    - `HttpEncoding` instances live in the `HttpEncodings` object.
    - `HttpProtocol` instances live in the `HttpProtocols` object.
    - `MediaType` instances live in the `MediaTypes` object.
    - `StatusCode` instances live in the `StatusCodes` object.

# HTTPRequest

# HTTPRequest

```
import HttpMethods._

val homeUri = Uri("/abc")

HttpRequest(GET, uri = homeUri)
```

# HTTPRequest

```
import HttpMethods._

HttpRequest(GET, uri = "/index")
```

# HTTPRequest

```
import HttpMethods._

val data = ByteString("abc")

HttpRequest(POST, uri = "/receive",
entity = data)
```

# HTTPRequest

```
import HttpProtocols._

import MediaTypes._

import HttpCharsets._
```

```
val userData = ByteString(“abc")
val authorization =
    headers.Authorization(
        BasicHttpCredentials("user", "pass"))

HttpRequest( PUT, uri = "/user",
   entity = HttpEntity
            (`text/plain` withCharset `UTF-8`, userData),
   headers = List(authorization),
   protocol = `HTTP/1.0`)
```

# HTTPResponse

# HTTPResponse

```
import StatusCodes._

HttpResponse(200)
```

# HTTPResponse

```
import StatusCodes._

HttpResponse(NotFound)
```

# HTTPResponse

```
HttpResponse(404, entity =
"Unfortunately, the resource couldn't
be found.")
```

# HTTPResponse

```
val locationHeader =
headers.Location("http://example.com/
other")

HttpResponse(Found, headers =
List(locationHeader))
```

# Marshallers

# Marshalling

Converting a higher level object into a lower level over the wire format

# Marshalling

Each Marshaller is a

```
A => Future[List[Marshalling[B]]]
```

# Marshalling

`Future` - Asynchronous Computation

`List` - To provide representation for possibly multiple items

`Marshalling[B]` - Allows for querying of `MediaType` and `HttpCharset`

# Marshalling

Comes with a predefined list of Marshallers for the most common types, e.g.

`Array[Byte]`, `ByteString`, `Array[Char]`, `HttpRequest`, `HttpResponse`, `FormData`, `MessageEntity`, `Multipart`, etc

# Marshalling

Are brought in using implicit resolution

# Unmarshallers

# Unmarshalling

Converting a lower level on the wire format into a higher level over object

# Unmarshalling Types

```
type FromEntityUnmarshaller[T] = Unmarshaller[HttpEntity, T]

type FromMessageUnmarshaller[T] = Unmarshaller[HttpMessage, T]

type FromResponseUnmarshaller[T] = Unmarshaller[HttpResponse, T]

type FromRequestUnmarshaller[T] = Unmarshaller[HttpRequest, T]

type FromByteStringUnmarshaller[T] = Unmarshaller[ByteString, T]

type FromStringUnmarshaller[T] = Unmarshaller[String, T]

type FromStrictFormFieldUnmarshaller[T] = Unmarshaller[StrictForm.Field,
T]
```

# Unmarshalling

Comes with a predefined list of Unmarshallers for the most common types, e.g.

`Array[Byte]`, `ByteString`, `Array[Char]`, `Byte`, `Short`, `Int`, `Long`, `FormData`, etc

# Unmarshalling

Unmarshallers are typically derived from persisting unmarshallers

# Demo: Implicit Resolution

# Routes

# Routes

## Routes at the lower level are pattern matched:

```
val requestHandler: HttpRequest => HttpResponse = {
     case HttpRequest(GET, Uri.Path("/"), _, _, _) =>
       HttpResponse(entity = HttpEntity(
         ContentTypes.`text/html(UTF-8)`,
         "<html><body>Hello world!</body></html>"))

   case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>
     HttpResponse(entity = "PONG!")

   case HttpRequest(GET, Uri.Path("/crash"), _, _, _) =>
     sys.error("BOOM!")

   case r: HttpRequest =>
     r.discardEntityBytes() // important to drain incoming HTTP Entity stream
     HttpResponse(404, entity = "Unknown resource!")
}
```

That can get kind of rough

# Routes

## The Alternative is a Routing DSL

```
val route =
  get {
    pathSingleSlash {
      complete(HttpEntity(ContentTypes.`text/html(UTF-8)`,
          "<html><body>Hello world!</body></html>"))
    } ~
    path("ping") {
      complete("PONG!")
    } ~
    path("crash") {
      sys.error("BOOM!")
    }
}
```

# Routes

Routes are done with import

```
akka.http.scaladsl.server.Directives._
```

# Routes

```
type Route =
RequestContext => Future[RouteResult]
```

`RequestContext` is an wrapper over `HTTPRequest`

# RouteResult

```
sealed trait RouteResult

object RouteResult {

  final case class Complete(response: HttpResponse) extends
RouteResult


  final case class Rejected(rejections:
immutable.Seq[Rejection]) extends RouteResult


}
```

# RouteTree

```
val route =
  a {
    b {
      c {
          ... // route 1
        } ~
      d {
        ... // route 2
      } ~
      ... // route 3
    } ~
    e {
      ... // route 4
    }
}
```

Route 1 will only be reached if directives a, b and c all let the request pass through.

# RouteTree

```
val route =
  a {
    b {
      c {
          ... // route 1
        } ~
      d {
        ... // route 2
      } ~
      ... // route 3
    } ~
    e {
      ... // route 4
    }
}
```

Route 2 will run if a and b pass, c rejects and d passes.

# RouteTree

```
val route =
  a {
    b {
      c {
          ... // route 1
        } ~
      d {
        ... // route 2
      } ~
      ... // route 3
    } ~
    e {
      ... // route 4
    }
}
```

Route 3 will run if a and b pass, but c and d reject.

# Directives

# Routes

`Directives` create `Routes`:

```
val route = complete("fun")
```

# Routes Structure

```
name(arguments) { extractions =>
  ... // inner route
}
```

# Routes Example

```
val route: Route =
  path("order" / IntNumber) { id =>
    get {
      complete {
        "Received GET request for order " + id
      }
    } ~
    put {
      complete {
        "Received PUT request for order " + id
      }
    }
}
```

# Routes Complex Example

```
val route =
  path("order" / IntNumber) { id =>
    (get | put) { ctx =>
      ctx.complete(s"Received ${
{ctx.request.method.name} request for
order $id")
    }
  }
```

# Routes Complex Example

```
val route =
  path("order" / IntNumber) { id =>
    (get | put) {
      extractMethod { m =>
        complete(s"Received ${m.name}
request for order $id")
      }
    }
  }
```

This is an extractor

# Routes Compose Example

```
val getOrPut = get | put

val route =
  path("order" / IntNumber) { id =>
    getOrPut {
      extractMethod { m =>
        complete(s"Received ${m.name}
request for order $id")
      }
    }
 }
```

# Routes Simplified Example

```
val getOrPut = get | put
val route =
  (path("order" / IntNumber) & getOrPut
& extractMethod) { (id, m) =>
    complete(s"Received ${m.name}
request for order $id")
  }
```

# Directives/Extractors

There are a number of predefined directives / extractors that manages:

- Security
- Handling Requests
- Logging
- Cookies
- More...

http://doc.akka.io/docs/akka-http/current/scala/http/routing-dsl/directives/alphabetically.html

# Demo: Creating a Basic Server

# Demo: Restful Servers

# JSON

# JSON Support

Using Marshalling and Unmarshalling mechanisms is how JSON is manipulated

# JSON Support

To include the basic JSON Support you require:

```
"com.typesafe.akka" %% "akka-http-spray-json" % "3.0.0-RC1"
```

# JSON Support

A multitude of supported libraries are available.

# JSON Support

```
import
akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._
```

# JSON Support

Pretty Format Available

# Demo: JSON Server/Client

XML

# XML Support

Using Marshalling and Unmarshalling mechanisms is how XML is manipulated

# XML Support

To include the basic JSON Support you require:

```
"com.typesafe.akka" %% “akka-http-xml” % "3.0.0-RC1"
```

# XML Support

Marshalling and Unmarshalling uses ScalaXML NodeSeq

# XML Support

Opt in by calling:

```
import
akka.http.scaladsl.marshallers.xml.ScalaXmlSupport._
```

# XML Support

or Mixin as a trait

`akka.http.scaladsl.marshallers.xml.ScalaXmlSupport`

# Java API

# Routing Java Style

```
public Route createRoute() {
   // This handler generates responses to `/hello?name=XXX` requests
   Route helloRoute =
     parameterOptional("name", optName -> {
       String name = optName.orElse("Mister X");
       return complete("Hello " + name + "!");
     });

   return
     // here the complete behavior for this server is defined

     // only handle GET requests
     get(() -> route(
       // matches the empty path
       pathSingleSlash(() ->
         // return a constant string with a certain content type
         complete(HttpEntities.create(ContentTypes.TEXT_HTML_UTF8, "<html><body>Hello world!</body></html>"))
       ),
       path("ping", () ->
         // return a simple `text/plain` response
         complete("PONG!")
       ),
       path("hello", () ->
         // uses the route defined above
         helloRoute
       )
     ));
  }
}
```

# Performance

# Final Thoughts

# Thank You