Developing Reactive Apps with Play Framework and Akka

Daniel Hinojosa

Developing Reactive Apps with Play Framework and Akka (Part 1)

- About Reactive Applications
- Requirements of Akka
- Requirements of Play
- Installing Akka
- Creating an Actor System
- Creating an Actor
- Understanding Futures
- Asking an Actor

Developing Reactive Apps with Play Framework and Akka (Part 2)

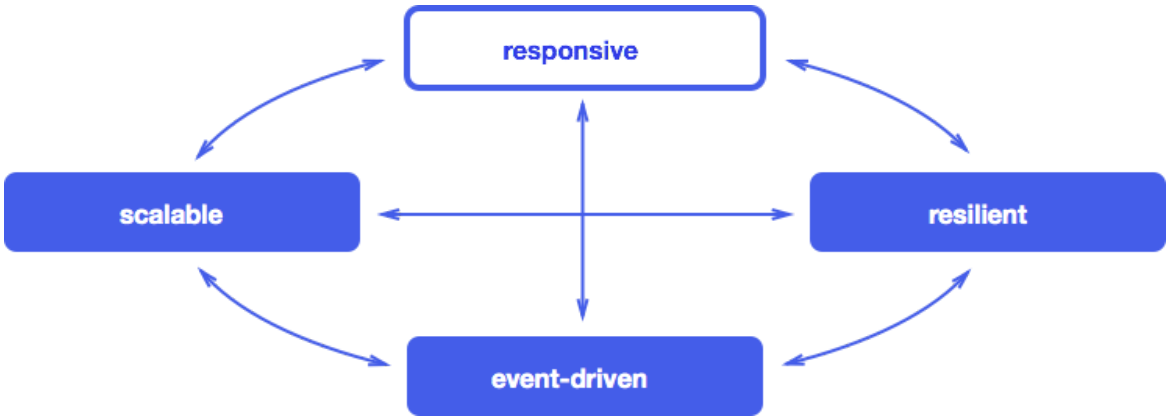
- Installing Play
- Create an Application
- Routing in our Application
- Forms
- Sending out information to a remote Actor
- Templating
- Twitter Bootstrap

About this workshop

- Hands on
- Most of the scaffolding done for you
- Play can be used in both Java and Scala, we will *only* focus on scala

Reactive Applications

- http://www.reactivemanifesto.org



Reactive Definition

> *Readily responsive to a stimulus*

Event-Driven

- Dormant System that are not required until needed
- Event Based and Asynchronous
- Message *pushing* instead of pull or poll to systems to respond
- Non-blocking
- Decoupled

Scalable

- Expanded on demand
- Ability to add and remote nodes
- Location transparent, where the inevitable resource can change even if you have a single address

Resilient

- Ability to take on errors, faults, disasters and recover easily
- Components, actors, are isolated
- Actor Systems can self-heal with various strategies

Responsive

- Applications need to respond in less than 1s
- Done with Observable Models that react to other state changes
- Avoid blocking at all levels
- Use asynchronous processing

Requirements of Akka & Play and this workshop

- JDK 1.6.0 or later (latest is the greatest)
- Ensure that `javac` and `java` function before proceeding
- WARNING: Java 7 pre update 9 on MacOS contains bugs

Set up `scala-play-akka-workshop` project

- Retrieve `scala-play-akka-workshop` from either stick or server
- Run `start` for Mac/Linux
- Run `start.bat` for Windows

Opening `scala-play-akka-workshop` in your favorite editor

For Eclipse Users

- Launch Eclipse
- Select File >> Import
- Select "Existing Projects into Workspace"
- Locate `scala-play-akka-workshop` and select Choose

For IntelliJ Users

- Launch IntelliJ IDEA and select Open… from the File menu.
- Locate `scala-play-akka-workshop` and click OK.

About Akka

- Set of libraries used to create concurrent, fault-tolerant and scalable applications.
- It contains many API packages:
- Actors, Logging, Futures, STM, Dispatchers, Finite State Machines, and more…
- We are going to only focus on some of the core items.
- Akka's managing processes can run on
  - in the Same VM
  - in a Remote VM
- Akka's Actor's will replace Scala's Actors

Game Changer?

Making the case

- No `synchronized`, no `wait`, no `notifyAll`
- No Boilerplate
- Failure Tolerance
- Better Scaling (Up or Out)
- Divide and Conquer
- Scala Actors to be deprecated out of the language in favor of Akka

Starting with Actors

- Based on the Actor Model from Erlang.
- Encapsulates State and Behavior
- Concurrent processors that exchange messages.
- Each message is immutable (cannot be changed, this is required!)
- Each message should not be a closure
- Breath of fresh air if you have suffered concurrency

Recognizing Immutable

What does immutable look like in Java?

JAVA

```java
public class Person {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    @Override
    public String toString() {...}

    @Override
    public boolean equals(Object o) {...}

    @Override
    public int hashCode() {...}
}
```

What does immutable look like in Scala?

JAVA
```
case class Person(firstName:String, lastName:String)
```

or

JAVA
```
class Person(val firstName:String, val lastName:String) {
  override def toString() = {...}
  override def equals(x:AnyRef):Boolean = {...}
  override def hashCode:Int = {...}
}
```

Recognizing closures

What is a closure in Java?

JAVA

```java
import java.util.function.Function;

public class Closures {

  public static Integer foo(Function<Integer, Integer> w) {
     return w.apply(5);
  }

  public static void main(String[] args) {
      Integer x = 3;
      Function<Integer, Integer> y = (Integer z) ->  x + z;
      System.out.println(foo(y)); //8
  }
}
```

What is a closure in Scala?

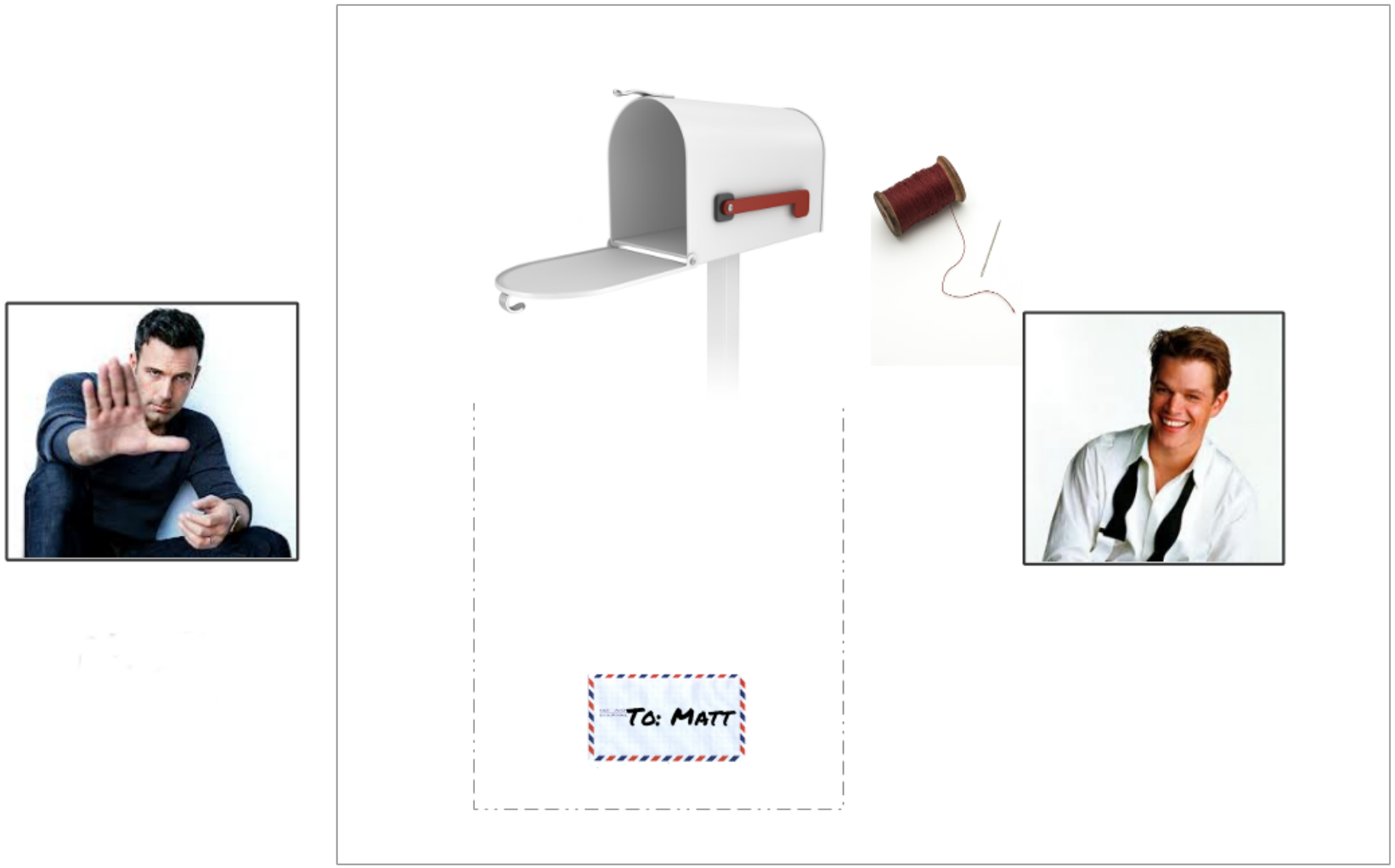
```JAVA
var x = 3
val y = {z:Int => x + z}
def foo(w: Int => Int) = w(5)
println(foo(y)); //8
```

Inside the Actor's Studio

Message Written

Message Sent

Message Processed By Thread



Message Processed By Thread

Thread goes away

Flooding Matt Damon with Messages

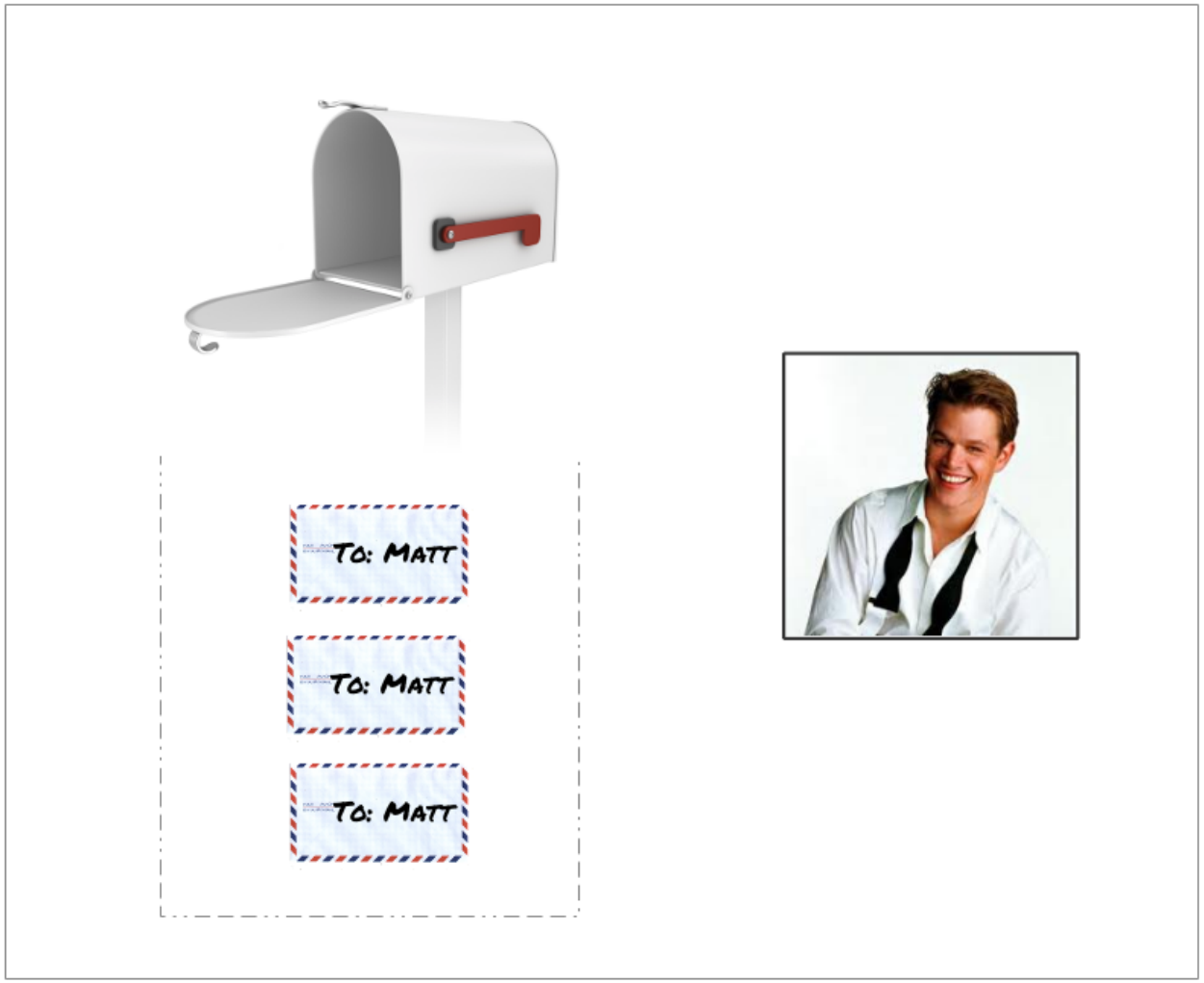
Flooding Matt Damon with Messages

Flooding Matt Damon with Messages

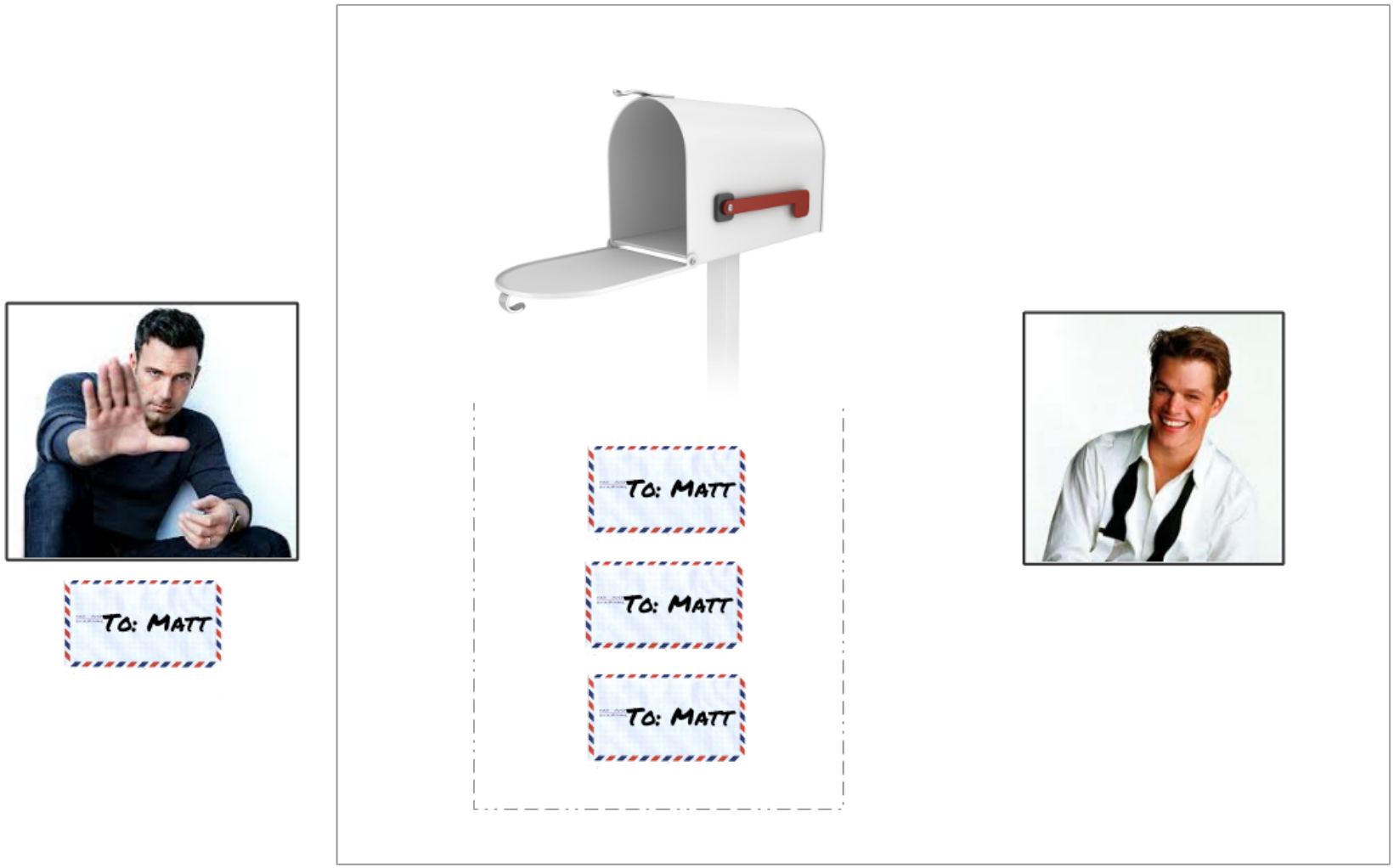
Flooding Matt Damon with Messages

Flooding Matt Damon with Messages

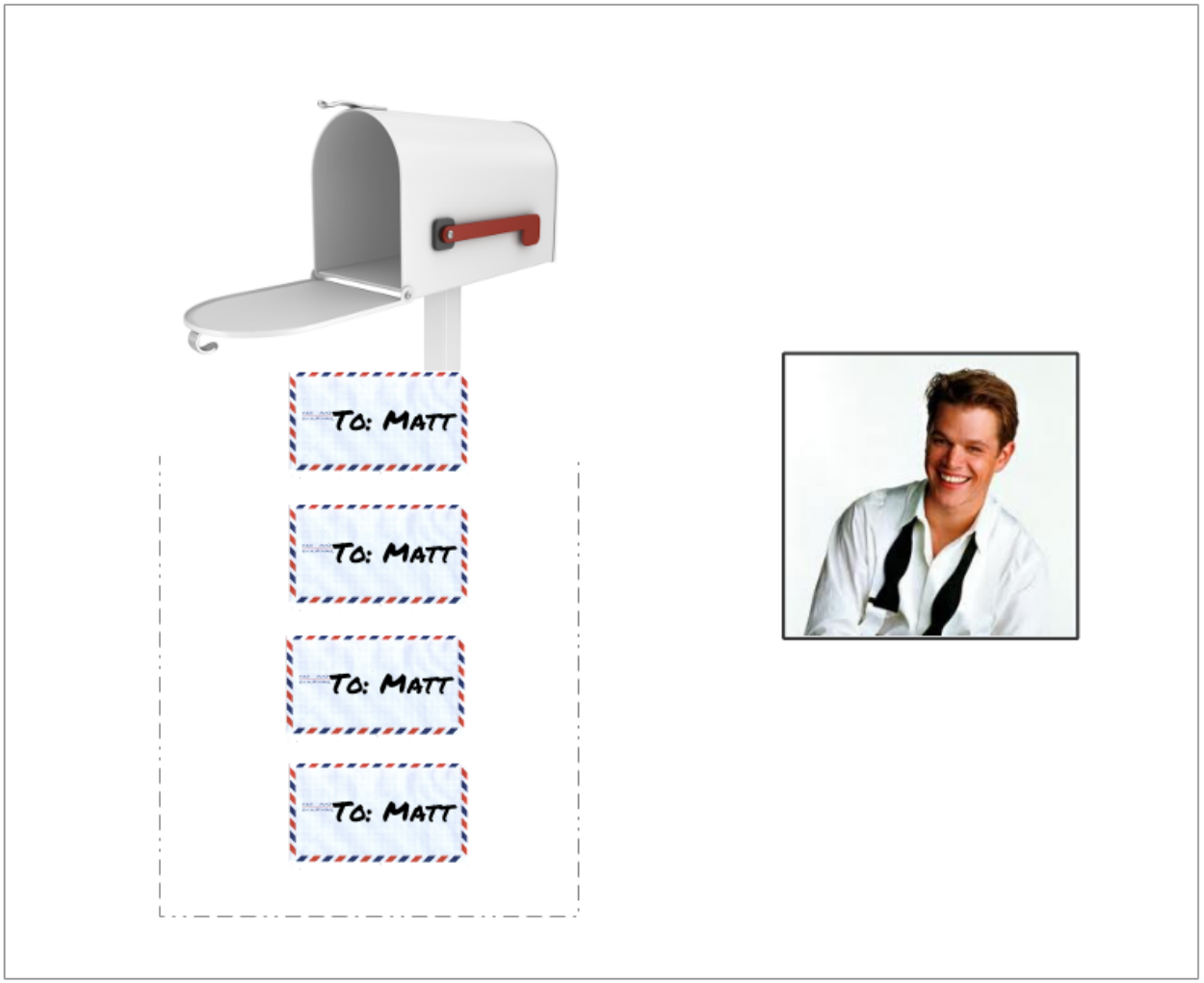
Flooding Matt Damon with Messages



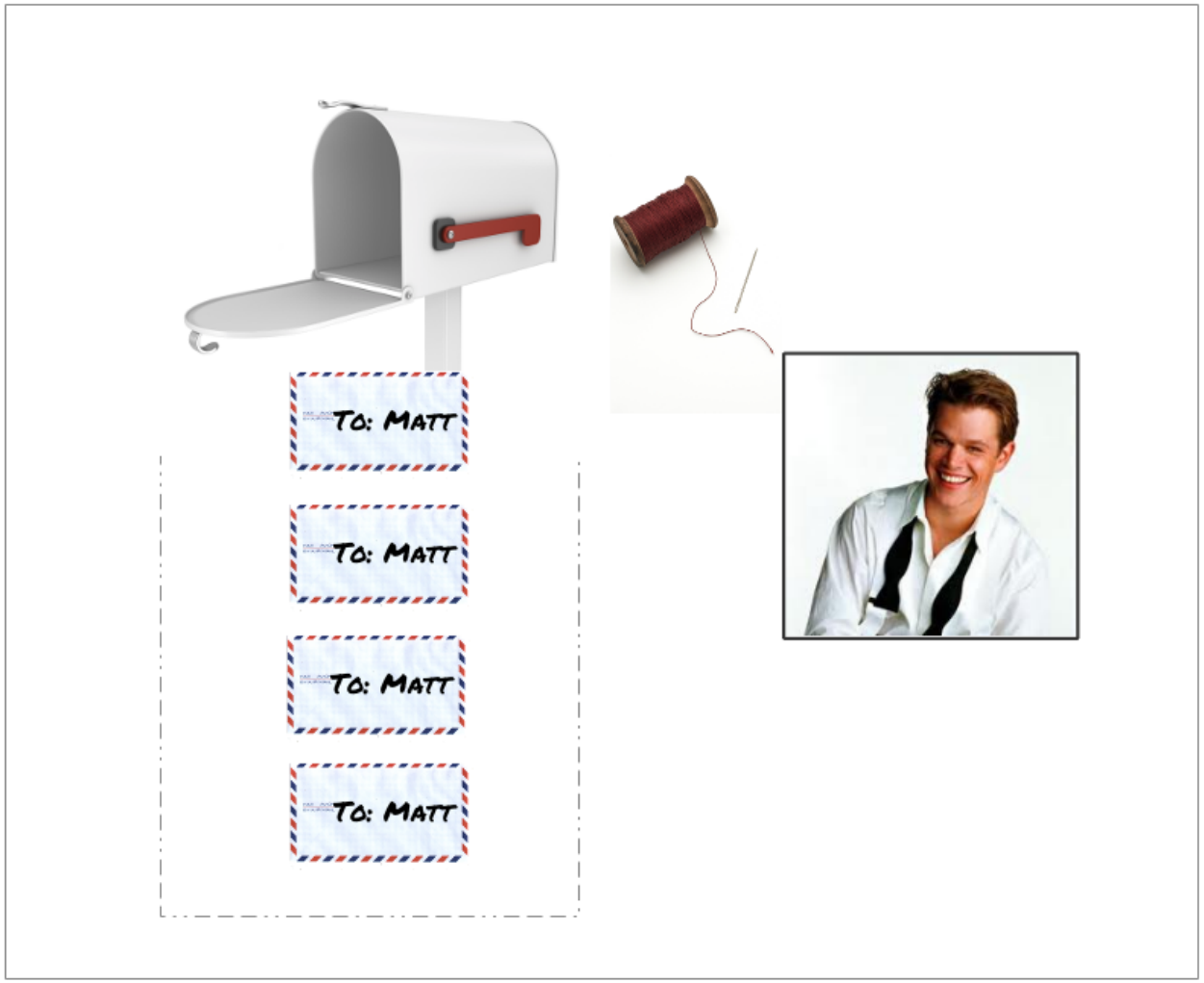
Flooding Matt Damon with Messages



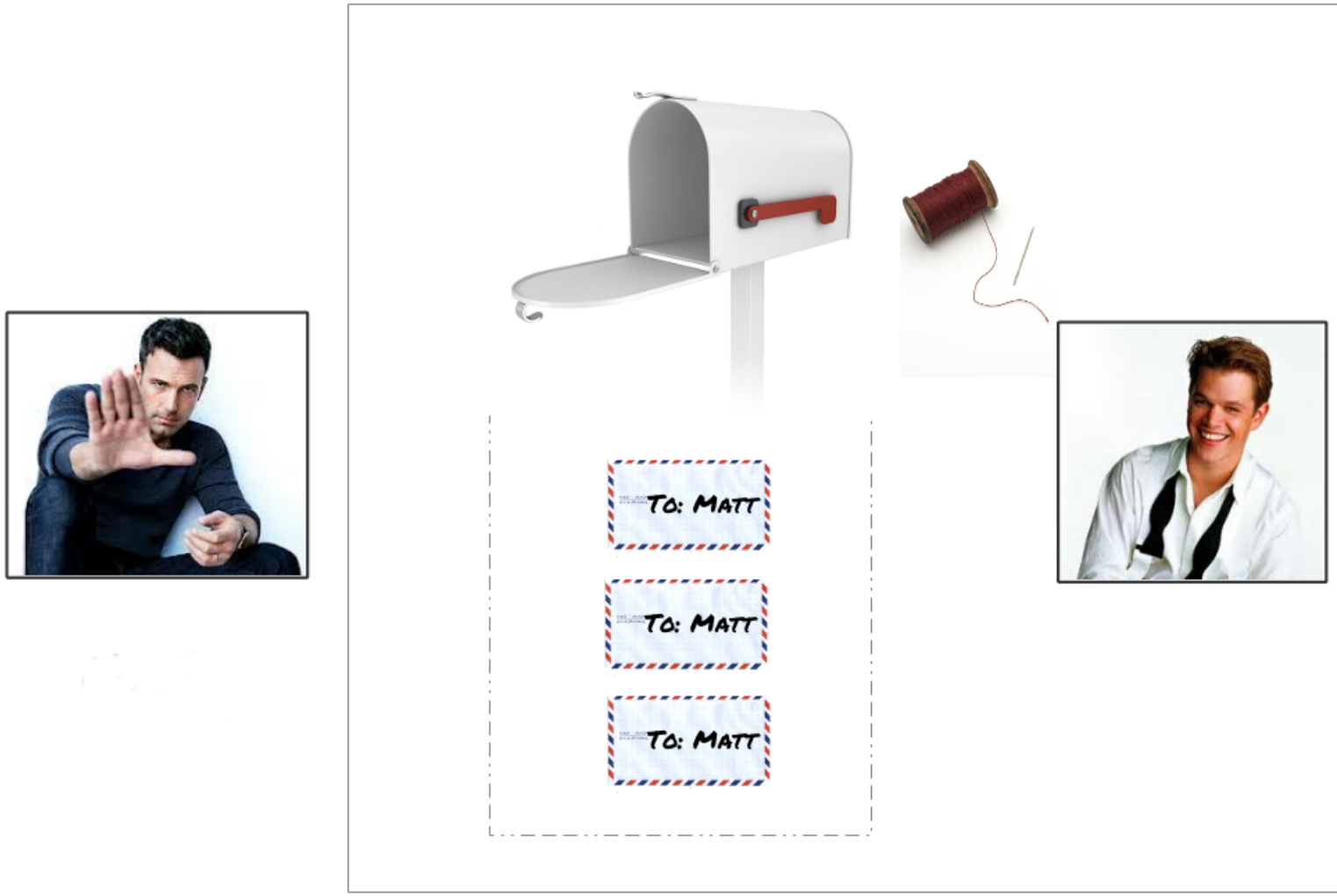
Flooding Matt Damon with Messages



Flooding Matt Damon with Messages



Flooding Matt Damon with Messages



Flooding Matt Damon with Messages

Flooding Matt Damon with Messages

Flooding Matt Damon with Messages

About sbt

- Simple Build Tool
- One of many build tools used in scala projects

About `sbt` commands

- It runs on `sbt` on the backend
- Contains amazing set of tools
  - `console` - provides a console, classloaded with production code
  - `test` - run all unit tests
  - `compile` - compile all code
  - `run` - run the application in development mode (non-daemonic)
  - `start` - start the application in production mode (daemonic)
  - `clean` - clean the `target` directory
  - `clean-all` - clean everything including the cache

To start the sbt console in our project

```
./start
```

or

```
.\start.bat
```

Typically though, you would use the command `sbt`

Multimodule projects

There are three projects

- akka_project
- play_project
- scala_project

Change focus on projects

To change focus to any of these projects merely type:

```
> project <project-name>
```

Try This! Change to the akka_project

```
> project akka_project
```

Creating an Actor

Every Actor within scala must extend `Actor`

Try This! In the `src/main/scala/com/example` folder create a class `RecordHolderActor` with the following:

SCALA

```scala
package com.example

import akka.actor.Actor

class RecordHolderActor extends Actor{

}
```

Create a `onReceive` method

All `Actor` will need to implement a `receive` method. This method will be called by a message.

Try This! implement the `onReceive` method for your `RecordHolderActor`

SCALA

```scala
package com.example

import akka.actor.Actor
import akka.actor.Actor.Receive

class RecordHolderActor extends Actor{
  override def receive: Receive = ???
}
```

Creating state within the actor

Much like you are used to with Java you can hold state within an actor. Threading an concurrency is all controlled by the `ActorSystem`

Try This! Add a `Map<String, Integer>` member variable

JAVA

```
package com.example

import akka.actor.Actor
import scala.collection.mutable

class RecordHolderActor extends Actor{
  var mutableMap = mutable.Map[String,Int]
  override def receive: Receive = ???
}
```

Using the Lifecycle methods

Once you subclass the `Actor` you have certain lifecyle methods that you can opt to override:

- `preStart()` - Lifecycle call when the actor is started
- `preRestart()` - Lifecycle call before the actor is restarted
- `postRestart()` - Lifecycle call after the actor is restarted
- `postStop()` - Lifecycle call after the actor stopped

Using the `preStart` lifecycle method

Try This! Override the `preStart` lifecycle method, and initialize your `Map`

JAVA

```
package com.example

import akka.actor.Actor

import scala.collection.mutable

class RecordHolderActor extends Actor {
  var mutableMap: mutable.Map[String, Int] = _

  override def preStart() = {
    super.preStart()
    this.mutableMap = mutable.Map.empty[String, Int]
  }

  override def receive: Receive = ???
}
```

Ensure that the message we are getting is a `String`

`String` is a perfect message, it is immutable!

Try this! Ensure that within the `onReceive` message that we are dealing with a `String`

```
...
  override def receive = {
     case message:String => ???
     case badMessage => unhandled(badMessage)
  }
...
```

Let's determine if an exercise that is being transmitted is a record!

Try This! Create logic in the `onReceive()` method that would determine if an exercise record was broken. All messages sent will be in the form "message:time"

SCALA

```
...
override def receive = {
  case message:String =>
    val tokens = message.split(":")
    val exercise = tokens(0)
    val time = tokens(1).toInt
    mutableMap.get(exercise) match {
      case Some(lastRecord) => if (time > lastRecord) {
        mutableMap.update(exercise, time)
      }
      case _ => mutableMap += ((exercise, time))
    }
  case badMessage => unhandled(badMessage)
}
...
```

Actor References within an `Actor`

- `self()` reference to the `ActorRef` of the actor
- `sender()` reference sender Actor of the last received message, typically used as described in Reply to messages
- `context()` exposes contextual information for the actor and the current message.

Enabling Logging within an `Actor`

- Logging is done by Akka asynchronously as well
- Ability to plugin a SLF4j logger or your own
- Uses a `system()` reference to establish the logger

Try This! Add logging to the actor

SCALA

```scala
package com.example

import akka.actor.Actor
import akka.event.Logging

import scala.collection.mutable

class RecordHolderActor extends Actor {
  val log = Logging(context.system, this)
```

Debug the success scenario!

Try This! Use Logging to display when the message was received!

```java
override def receive = {
   case message:String =>
     val tokens = message.split(":")
     val exercise = tokens(0)
     val time = tokens(1).toInt
     mutableMap.get(exercise) match {
       case Some(lastRecord) => if (time > lastRecord) {
           log.debug("Record was broken with exercise {} and time of {} seconds", exercise, time)
           mutableMap.update(exercise, time)
       }
       case _ => mutableMap += ((exercise, time))
     }
   case badMessage => unhandled(badMessage)
}
```

A quick intro to HOCON

- "Human-Optimized Config Object Notation"
- Uses Typesafe Configuration Library https://github.com/typesafehub/config
- JSON Like Features
- Typical configuration file: 'application.conf'

Sample HOCON Configuration

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""
```

Sample HOCON Configuration (Alternate 1)

```
db {
  default.driver=org.h2.Driver
  default.url="jdbc:h2:mem:play"
  default.user=sa
  default.password=""
}
```

Sample HOCON Configuration (Alternate 2)

```
db {
  default {
    driver=org.h2.Driver
    url="jdbc:h2:mem:play"
    user=sa
    password=""
  }
}
```

`application.conf`

- Contains Settings for Actor Systems
- All HOCON (Human Optimized Config Object Notation)
- One 'application.conf' per application
- Typically stored `src/main/resources`

Create an `application.conf` file

Try This! Create a folder called 'resources' inside of 'src/main', then create a file called 'application.conf' with the following:

```
akka {
  event-handlers = ["akka.event.Logging$DefaultLogger"]
  loglevel = DEBUG
  stdout-loglevel = DEBUG

  actor {
    provider = "akka.actor.LocalActorRefProvider"

    default-dispatcher {
      throughput = 10
    }

    debug {
      autoreceive = on
      lifecycle = on
      fsm = on
      event-stream = on
    }
  }
}
```

`ActorSystem` and `Props`

- An `ActorSystem`
  - Houses multiple `Actor`
  - Is heavy, and requires only on per logical application
- `Props`
  - An immutable object
  - Recipe with details how to create an `Actor`

Test the Actor!

Try This! Create a runnable object in `src/main/scala/com/example` called `Runner`

SCALA

```scala
package com.example

import akka.actor.{Props, ActorSystem}

object Runner {
   def main(args:Array[String]):Unit = {
     val actorSystem = ActorSystem("MySystem")
     val recordHolderActorRef = actorSystem.actorOf(Props[RecordHolderActor], "recordHolderActor")
     recordHolderActorRef ! "Swimming:14"
     recordHolderActorRef ! "Swimming:19"
     recordHolderActorRef ! "Swimming:12"
     recordHolderActorRef ! "Swimming:14"
     recordHolderActorRef ! "Swimming:60"
     Thread.sleep(5000)
     actorSystem.shutdown()
     actorSystem.awaitTermination()
   }
}
```

Futures and Promises

`java.util.concurrent`

- Contains an `Executor`
- `Executor` manages independent threading tasks and decouples task submission from task execution
- Contains an `ExecutorService` that takes `Runnable` or `Callable` tasks and comes complete with a lifecycle and monitoring systems

The `Executor`

```
public interface Executor {
   void execute(Runnable command);
}
```

`ExecutorService`

- An `ExecutorService` is a subinterface of `Executor`
- Provides the ability to create and track Futures from `Runnable` and Callables
- Services can be started up and shut down
- Can create your own
- Likely will use `Executors` class to create `ExecutorServices`

Snippet of `ExecutorService`

```
public interface ExecutorService extends Executor {
  ...
   <T> Future<T> submit(Callable<T> task);
   <T> Future<T> submit(Runnable task, T result);
   Future<?> submit(Runnable task);
  ...
}
```

Executors

Utility Factory that can create `ExecutionServices`

`FixedThreadPool`

- `Executors.newFixedThreadPool()`
- *“Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.”* according to the API
- Keeps threads constant and uses the queue to manage tasks waiting to be run
- If a thread fails, a new one is created in its stead
- If all threads are taken up, it will wait on an unbounded queue for the next available thread

`SingleThreadExecutor`

- `Executors.newSingleThreadExecutor()`
- *“Creates an Executor that uses a single worker thread operating off an unbounded queue.”*
- If a thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.

`CachedThreadPool`

- `Executors.newCachedThreadPool()`
- Factory method is a flexible thread pool implementation that will reuse previously constructed threads if they are available
- If no existing thread is available, a new thread is created and added to the pool
- Threads that have not been used for sixty seconds are terminated and removed from the cache

`ScheduledThreadPool`

- `Executors.newScheduledThreadPool()`
- Can run your tasks after a delay or periodically
- This method does not return an `ExecutorService`
- Returns a ScheduledExecutorService which contains methods to help you set not only the task but the delay or periodic schedule

`Future`

- An asynchronous computation
- Contains methods determine completion of said computation
- Can be in running state, completed state, or have thrown an Exception

`Future` interface

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws
         InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
         throws InterruptedException,
           ExecutionException, TimeoutException;
}
```

Futures in `scala.concurrent`

- Data structures used to retrieve the result of some concurrent operation.
- It can be retrieved “synchronously” (blocked) or “asynchronously” (unblocked)
- Futures need an ExecutionContext in scope in order to work

Promises


TICKET
KEEP THIS COUPON
TICKET
KEEP THIS COUPON
TICKET
KEEP THIS COUPON

Ask

- Actors can be asked for return information from an actor using the ask pattern
- Will return a `Future` so that you can wait for an answer
- Sends a message asynchronously and returns a Future representing a possible reply.
- Performance Hit especially with remote actors

Retrofitting our actor to respond to messages

Try This! Let's change our logic around to send messages back, `true` if a record was broken, `false` if it wasn't.

SCALA

```
override def receive = {
   case message:String =>
     val tokens = message.split(":")
     val exercise = tokens(0)
     val time = tokens(1).toInt
     mutableMap.get(exercise) match {
       case Some(lastRecord) => if (time > lastRecord) {
         log.debug("Record was broken with exercise {} and time of {} seconds", exercise, time)
         mutableMap.update(exercise, time)
         sender().tell(true, self)
       } else {
         sender().tell(false, self)
       }
       case _ =>
         mutableMap += ((exercise, time))
         sender().tell(false, self)
     }
   case badMessage => unhandled(badMessage)
}
```

Refactoring our actor using guards

Try This! Let's use pattern matching guards instead for cleaner code

SCALA

```scala
override def receive = {
  case message:String =>
    val tokens = message.split(":")
    val exercise = tokens(0)
    val time = tokens(1).toInt
    mutableMap.get(exercise) match {
      case Some(lastRecord) if time > lastRecord =>
        log.debug("Record was broken with exercise {} and time of {} seconds", exercise, time)
        mutableMap.update(exercise, time)
        sender().tell(true, self)
      case Some(lastRecord) if time < lastRecord => sender().tell(false, self)
      case _ =>
        mutableMap += ((exercise, time))
        sender().tell(false, self)
    }
  case badMessage => unhandled(badMessage)
}
```

Rework our `Runner` that makes use of the `Future`

Now that we have callback we need a `Timeout` in order to determine if the `Future` responds in an adequate amount of time. We also need to import `akka.util.Timeout`, `java.util.TimeUnit`, and `akka.pattern.ask`.

> **NOTE** `actorSystem.dispatcher()` will return the same thread pool that the `ActorSystem` is using.

Try This! Rework the simple in Runner, and see how asking an actor works!

SCALA

```scala
package com.example

import akka.actor.{Props, ActorSystem}
import akka.util.Timeout
import scala.concurrent.duration._
import akka.pattern.ask

object Runner {
  def main(args:Array[String]):Unit = {
    val actorSystem = ActorSystem("MySystem")
    val recordHolderActorRef = actorSystem.actorOf(Props[RecordHolderActor], "recordHolderActor")

    import actorSystem.dispatcher // Execution Context to support the Future
    implicit val timeout = Timeout(5 seconds)

    (recordHolderActorRef ? "Swimming:14").foreach(answer => println("Seconds: 14" + answer))
    (recordHolderActorRef ? "Swimming:19").foreach(answer => println("Seconds: 19" + answer))
    (recordHolderActorRef ? "Swimming:12").foreach(answer => println("Seconds: 12" + answer))
    (recordHolderActorRef ? "Swimming:14").foreach(answer => println("Seconds: 14" + answer))
    (recordHolderActorRef ? "Swimming:60").foreach(answer => println("Seconds: 60" + answer))

    Thread.sleep(10000)
    actorSystem.shutdown()
    actorSystem.awaitTermination()
  }
}
```

Converting our actor into a remote actor

Adding Akka Remote as a Dependency

The project that we are working uses sbt, we will need to include `akka-remote` in order to use any remote configuration.

Try This! Ensure `akka-remote` is in as a dependency, the end result should look like the following:

```
name := """akka-project"""

version := "1.0-SNAPSHOT"

scalaVersion := "2.11.5"

libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.3.3",
  "com.typesafe.akka" %% "akka-testkit" % "2.3.3" % "test",
  "com.typesafe.akka" %% "akka-remote" % "2.3.3",
  "junit" % "junit" % "4.11" % "test",
  "com.novocode" % "junit-interface" % "0.10" % "test")
```

Note: Be sure to refresh your IDE or perform a `reload` and `update` on SBT

Configuring a remote system

Configuring our actor is all a matter of adding configuration to `application.conf`. We will need to describe the name of the configuration, what kind of `ActorRef` will be served, and netty and protocol configuration.

Try This! Add the following to `application.conf`

```
remote-system {
  akka {
    loglevel = DEBUG
    stdout-loglevel = DEBUG

    actor {
      provider = "akka.remote.RemoteActorRefProvider"
      default-dispatcher {
        throughput = 10
      }
      debug {
        autoreceive = on
        lifecycle = on
        fsm = on
        event-stream = on
      }
    }

    remote {
      log-sent-messages = on
      log-received-messages = on
      log-remote-lifecycle-events = on
      enabled-transports = ["akka.remote.netty.tcp"]
      netty.tcp {
        hostname = "127.0.0.1"
        port = 10191
      }
    }
  }
```

Akka Kernel

The purpose of the Akka Microkernel is to offer a bundling mechanism so that you can distribute an Akka application as a single payload. Without the need to run in a Java Application Server or manually having to create a launcher script.

Including a Kernel

In order to use any of the kernel classes you need to declare a dependency for `akka-kernel`

Try This! Add a dependency for `akka-kernel` in `build.sbt` (This may already be done for you)

```
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.3.9",
  "com.typesafe.akka" %% "akka-testkit" % "2.3.9" % "test",
  "com.typesafe.akka" %% "akka-remote" % "2.3.9",
  "com.typesafe.akka" %% "akka-kernel" % "2.3.9",
  "junit" % "junit" % "4.11" % "test",
  "com.novocode" % "junit-interface" % "0.10" % "test")
```

Generating a Kernel with lifecycle methods

Try This! Once the dependency has been loaded, create a class in `src/main/java/com/example` called `RecordHolderKernel` with the following content:

SCALA

```scala
package com.example

import akka.actor.{ActorSystem, Props}
import akka.kernel.Bootable
import com.typesafe.config.ConfigFactory

class RecordHolderKernel extends Bootable {
  val config = ConfigFactory.load()
  val actorSystem = ActorSystem("remoteActorSystem",
    config.getConfig("remote-system").withFallback(config))

  def startup() = {
    val recordHolderActorRef = actorSystem.actorOf(Props[RecordHolderActor], "recordHolderActor")
  }

  def shutdown() = {
    actorSystem.shutdown()
  }
}
```

Download the Akka Distribution

In order to use the kernel:

- Download the kernel distribution for http://akka.io/downloads, or download from other sources
- Select the distribution based on the scala version that is contained in your `build.sbt`
- Expand the .zip file in a your favorite folder

Package Copy and Run

- In the sbt console, switch to the akka-project
- Type `package` in the sbt console
- Copy the `scala-play-akka-workshop/akka-project/target/scala-2.11/my-akka-system_2.11-1.0.jar` into the `deploy` directory of your distribution
- Go to your akka distribution directory
- Run `bin/akka com.example.RecordHolderKernel` in Linux and Mac
- Run `bin\akka.bat com.example.RecordHolderKernel` in Windows

Play

Playing with Play

Try This! In the sbt console switch to the play_project

```SH
> project play_project
```

Starting Your New Web Application

Try This! Start up the Web Application

```SH
> run
```

or

```SH
% run <port>
```

Viewing your website!

Try it! Go to http://localhost:9000

Stopping the Play Framework

CTRL+D

Try This! Stop the server using **CTRL+D** and restart it using the `run` command you just learned

Folder Structure

```
app                      → Application sources
 └ assets                → Compiled asset sources
    └ stylesheets        → Typically LESS CSS sources
    └ javascripts        → Typically CoffeeScript sources
 └ controllers           → Application controllers
 └ models                → Application business layer
 └ views                 → Templates
conf                     → Configurations files
 └ application.conf      → Main configuration file
 └ routes                → Routes definition
public                   → Public assets
 └ stylesheets           → CSS files
 └ javascripts           → Javascript files
 └ images                → Image files
project                  → sbt configuration files
 └ build.properties      → Marker for sbt project
 └ Build.scala           → Application build script
 └ plugins.sbt           → sbt plugins
lib                      → Unmanaged libraries dependencies
logs                     → Standard logs folder
 └ application.log       → Default log file
target                   → Generated stuff
 └ scala-2.9.1
    └ cache
    └ classes            → Compiled class files
    └ classes_managed    → Managed class files (templates, ...)
    └ resource_managed   → Managed resources (less, ...)
    └ src_managed        → Generated sources (templates, ...)
test                     → source folder for unit or functional tests
```

About the packages

- `play.api.*` packages are for Scala use only
- `play.mvc.*` packages are for Java use only

Controllers

A Controller is a class that extends `java.mvc.Controller` which houses one more Action methods.

Try This! *Create a controller inside of* `app/controllers` *called* `FitnessController.scala`

JAVA

```
package controllers

import play.api.mvc._


object FitnessController extends Controller {

}
```

> **NOTE** You do not need to restart play, check the console, it already compiled!

Actions

An action is a method within a `play.api.mvc.Controller` that represents a single unit of work that

- Performs a task
- Returns a `play.api.mvc.Result` that represents an HTTP response call (e.g. `200 OK`)

Try This! *Create an action method inside of* `app/controllers/FitnessController.java`

JAVA

```
package controllers

import play.api.mvc._

object FitnessController extends Controller {
  def welcome = Action {
    Ok("Welcome to Borg Fitness! Time to assimilate into fitness!")
  }
}
```

Result

- A `play.api.mvc.Result` is a class that represents an HTTP response.
- A predefined list of `play.api.mvc.Result` can be found in the `play.api.mvc.Results`

Some Predefined `java.api.mvc.Result`

SCALA

```
ok = OK("Everything OK");
notFound = NotFound("Resource not found");
forbidden = Forbidden("Can't touch this!");
created = Created("Whatever you were trying to create, you did it");
unauthorized = Unauthorized("You shall not pass");
badRequest = BadRequest("What are you talking about man?");
internalServerError = InternalServerError("I don't feel too good");
```

Routing

A router

- Managed page and component that defines what action method to run based on what RestFUL URI was called
- Configuration page is located in `conf/routes` inside the application
- Each route is space or tab delimited and contains in order
  - The HTTP method called (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`)
  - The URI Pattern that any one of your users will call
  - The FQN (Fully Qualified Name) of the controller class and action that will handle the request
- Comments are any string preceeded by a `#`
- The router that's highest on the page has precedence

Routing Example

Try This! *Create a file called* `conf/routes` *with the following information*

```txt
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# Map static resources from the /public folder to the /assets URL path
GET     /assets/*file               controllers.Assets.at(path="/public", file)
```

Make Your Own Route

Try This! *Add Your Own Route and then visit http://localhost:9000/welcome*

```
GET     /welcome                    controllers.FitnessController.welcome()
```

Add a Parameter to your Action

We can create a custom route by creating another action that accepts a parameter into a method

Try This! Add the following method to `app/controllers/FitnessController.scala`

SCALA

```scala
def welcomeWithName(name:String) = Action {
    Ok(s"Welcome to Borg Fitness $name! Time to assimilate into fitness!")
}
```

Mapping Parameter from the URI to the Action

Accessing the parameter is now easy by merely create a route with the parameter and sending it to the action inside of the `play.api.mvc.Controller`

> **WARNING** You cannot use method overloading for an action

Try This! *Add a new route in* `conf/routes` *and go to http://localhost:9000/welcome/dan*

```java
GET        /welcome/:name       controllers.FitnessController.welcomeWithName(name:String)
```

Creating an Exercise!

Now, let's say we want to create an "Exercise of the Day" on our website!+

Try This! *Add a* `models` *directory under the* `app` *directory, and add a new class* `Exercise.scala`

SCALA

```scala
package models

case class Exercise(name:String, minutes: Int)
```

Making it look pretty on an HTML5 webpage!

About Views:

- Each page must end in `.scala.html`
- A page can be placed inside of folders for better organization
- A page with the name `index.scala.html` will be referred to in source code as `views.html.index`
- A page with the name `analysis\index.scala.html` will be referred to in source code as `views.html.analysis.index`

Try This! Create a folder called `views` under `app` and make a page called `exerciseoftheday.scala.html` in the `views` folder

```HTML5
@(exercise:Exercise)
<!DOCTYPE html>
<html>
    <head>
        <title>Borg Fitness: Exercise of the Day @exercise.name</title>
    </head>
    <body>
        Our exercise of the day is @exercise.name and going for @exercise.minutes minutes
    </body>
</html>
```

Creating an action that binds an "Exercise of the Day"

Try This! We want to fill in the value for our "Exercise of the Day" so we can view it on a page!

SCALA

```scala
def exerciseOfTheDay = Action {
    Ok(views.html.exerciseoftheday(Exercise("Swimming", 60)))
}
```

Let's bind the route!

Try This! Let bind the route in `conf/routes` so that we can see all our hard work pay off by going to http://localhost:9000/exerciseoftheday

```
GET        /exerciseoftheday     controllers.FitnessController.exerciseOfTheDay()
```

Ok, so how do collections work in these templates?

First off, we need a collection to send into a page!

Try This! *Create a new action method that will send out a collection of* `Exercise`

SCALA

```
def workoutOfTheDay = Action {
   val list = List(Exercise("Running Sprints", 10),
                   Exercise("Running Light Jog", 20),
                   Exercise("Running Sprints", 10),
                   Exercise("Cool Down", 10))
   Ok(views.html.workoutOfTheDay(list))
}
```

> **NOTE** We need to create our view called workoutOfTheDay!

Create our workout!

Now that we have our action, let's apply to our exercise to the new view!

Spaces are important, after the `for` be sure to not have a space.

Also note that we are using `scala.collection.List`

Another thing to know about Play, is this is a Scala templating engine.

Try This! *Make the page* `workoutoftheday.scala.html` *in the* `views` *folder*

JAVA

```
@(exercises: List[Exercise])
<!DOCTYPE html>
<html>
    <head>
        <title>Borg Fitness Workout of the Day</title>
    </head>
    <body>
        <ol>
            @for(exercise <- exercises) {
                <li>@exercise.name and go @exercise.minutes minutes</li>
            }
        </ol>
    </body>
</html>
```

It's not complete until we route the URI to the action!

Try This! *Add the following route to* `conf/routes`*. Then visit http://localhost:9000/workoutoftheday*

```TXT
GET        /workoutoftheday          controllers.FitnessController.workoutOfTheDay()
```

But I want to enter my own exercises!

The `play.api.data.Form` is a class that represent the form that stores what people enter. The `Form` requires the class that it will represent.

Try This! Create an Exercise Form that will be used in FitnessController.scala

Add the following imports

```
import play.api.data._
import play.api.data.Forms._
```

…and inside the controller add the form definition

```
val exerciseForm = Form(
       mapping(
          "name" -> text,
          "minutes" -> number
       )(Exercise.apply)(Exercise.unapply)
)
```

Using the form in an initial action

Try This! *Create an action to setup the form*

```java
def initExercise = Action {
  Ok(views.html.createexercise.render(exerciseForm))
}
```

Plugging in the form

Now that we have the form object we can just accept the form and plug it into the page.

***Try This!** Create* `views/createexercise.scala.html`

JAVA

```
@(form: Form[Exercise])
@import play.data.Form

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
        <form name="exercise_form" action="/exercise/create" method="POST">
            <div>
                <label id="name_label" for="name">Name:</label>
                <input name="name" value="@form("name").value()"/>
            </div>
            <div>
                <label id="minutes_label" for="minutes">Minutes:</label>
                <input name="minutes" value="@form("minutes").value()"/>
            </div>
            <input id="submit" name="submit" type="submit" value="Create"/>
        </form>
    </body>
</html>
```

Let’s add another route

Try this! Add a route to `conf/routes` that would map the `/exercise/create` to the `initExercise()` method. Go to http://localhost:9000/exercise/create to see if it looks right.

```txt
GET        /exercise/create         controllers.FitnessController.initExercise()
```

And, what does the action that handles the form look like?

The action uses a form declaration and calls `bindFromRequest()`. This gets the values of the form and sets the fields into an `Exercise` object.

Try This! *Create an action called* `createExercise` *which gets the values from the form*

SCALA

```
def createExercise = Action {implicit request =>
    exerciseForm.bindFromRequest.fold(
      formWithErrors => {
        BadRequest(views.html.createexercise(formWithErrors))
      },
      exercise => {
        Ok("Received exercise for %s".format(exercise))
      }
   )
}
```

Time to add a route!

This time though we aren't performing a `GET` we are performing a `POST`

Try This! Add a post route to the `createExercise()` method you just created and go to http://localhost:9000/exercise/create and see if it all works well.

```java
POST        /exercise/create          controllers.FitnessController.createExercise()
```

Très bon! But there is nothing in the web page to display any errors.

Errors are already located on the `play.api.data.Form`. It is just a matter of accessing those errors and putting them on the page however you like.

Try This! Add some errors to be displayed on `views/createexercise.scala.html` then verify it works at http://localhost:9000/exercise/create

HTML

```
@(form: Form[Exercise])
@import play.api.data.Form
@import play.api.i18n.Messages

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
        <form name="exercise_form" action="/exercise/create" method="POST">
            <div>
                <label id="name_label" for="name">Name:</label>
                <input name="name" value="@form("name").value"/>
                @for(error <- form.errors("name")) {
                    <span style="color : red">@Messages(error.message)</span>
                }
            </div>
            <div>
                <label id="minutes_label" for="minutes">Minutes:</label>
                <input name="minutes" value="@form("minutes").value"/>
                @for(error <- form.errors("minutes")) {
                    <span style="color : red">@Messages(error.message)</span>
                }
            </div>
            <input id="submit" name="submit" type="submit" value="Create"/>
        </form>
    </body>
</html>
```

Right now the messages are kind of raw, lets change that!

Try This! Create a file conf/messages.en_US with the following information

```
error.max=Maximum value reached
error.required=Required
```

Adding a reverse route

One of the nice things about Play is that you can always change the route to a different address, and as long as the method of the action stay consistent, reverse routing will always attach to the right URI.

Try This! *Change the action of the* `<form>` *to a route instead of an hard coded URI address in* `createexercise.scala.html`

HTML

```
@(form: Form[Exercise])
@import play.api.data.Form
@import play.api.i18n.Messages

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
        <form name="exercise_form" action="@routes.FitnessController.createExercise()" method="POST">
            <div>
                <label id="name_label" for="name">Name:</label>
                <input name="name" value="@form("name").value"/>
                @for(error <- form.errors("name")) {
                    <span style="color : red">@Messages(error.message)</span>
                }
            </div>
            <div>
                <label id="minutes_label" for="minutes">Minutes:</label>
                <input name="minutes" value="@form("minutes").value"/>
                @for(error <- form.errors("minutes")) {
                    <span style="color : red">@Messages(error.message, 0)</span>
                }
            </div>
            <input id="submit" name="submit" type="submit" value="Create"/>
        </form>
    </body>
</html>
```

Form Helpers

The Play Framework has helpers that does most of the work for you.

Try This! *Replace the* `<form>` *tag with a helper. Verify it works at http://localhost:9000/exercise/create*

HTML

```
@(form: Form[Exercise])
@import play.api.data.Form
@import play.api.i18n.Messages

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
        @helper.form(action = routes.FitnessController.createExercise(), 'id -> "exercise_form") {
            <div>
                <label id="name_label" for="name">Name:</label>
                <input name="name" value="@form("name").value"/>
                @for(error <- form.errors("name")) {
                    <span style="color : red">@Messages(error.message)</span>
                }
            </div>
            <div>
                <label id="minutes_label" for="minutes">Minutes:</label>
                <input name="minutes" value="@form("minutes").value"/>
                @for(error <- form.errors("minutes")) {
                    <span style="color : red">@Messages(error.message, 0)</span>
                }
            </div>
            <input id="submit" name="submit" type="submit" value="Create"/>
        }
    </body>
</html>
```

Input Form Helpers

A field constructor is an implicit object that gives a whole lot of html to provide labels, error fields, help and more. All the elements of the field require an underscore.

Try This! *replace* `<input>` *text fields with a* `@helper.inputText` *fields. Verify it works at http://localhost:9000/exercise/create*

HTML

```
@(form: Form[Exercise])
@import play.api.data.Form

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
        @helper.form(action = routes.FitnessController.createExercise(), 'id -> "exercise_form") {
            <div>
                @helper.inputText(form("name"), '_id -> "name",
                                  '_label -> "Exercise:", '_showConstraints -> false)
            </div>
            <div>
                @helper.inputText(form("minutes"), '_id -> "minutes",
                                  '_label -> "Minutes:", '_showConstraints -> false)
            </div>
            <input id="submit" name="submit" type="submit" value="Create"/>
        }
    </body>
</html>
```

Calling our actor reactively!

Try This! First off, we would like to connect to our actor system that is loaded in our kernel, so lets make our webapp a client

Inside our application.conf lets add some basic configuration!

```
akka {
  stdout-loglevel = "DEBUG"
  loglevel = "DEBUG"
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    log-sent-messages = on
    log-received-messages = on
    log-remote-lifecycle-events = on
    netty.tcp {
      hostname = "127.0.0.1"
      port = 0
    }
  }
}
```

Next let's change our code to send the actor a message!

Try this! Play Framework comes with an ActorSystem ready to go, in fact it's built on actors, let's call it up and send a message to the remote machine! Check the logs to make sure it works! Bravo! Now try sending it to others. Ohh la la!

SCALA

```
def createExercise = Action {implicit request =>
    exerciseForm.bindFromRequest.fold(
      formWithErrors => {
        BadRequest(views.html.createexercise(formWithErrors))
      },
      exercise => {
        import akka.util.Timeout
        import scala.concurrent.duration._
        import akka.pattern.ask

        val selection =
Akka.system.actorSelection("akka.tcp://remoteActorSystem@127.0.0.1:10191/user/recordHolderActor")
        implicit val timeout = Timeout(5 seconds)
        implicit val executionContext = Akka.system().dispatcher
        (selection ? s"${exercise.name}:${exercise.minutes}").foreach(x => println(x))
        Ok("Received exercise for %s".format(exercise))
      }
    )
 }
```

Templating

How does templating work?

The nice thing about the play framework is that pages are done using scala, and you can treat each page like a function that has paramaters and you can call pages from other pages to create effect.

Try this! *Create a page in* `views` *called* `template.scala.html` *that creates a surrounding template for all your pages. Look for a 'borg' image from the Internet and make it apart of your template.*

HTML

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
<head>
    <title>@title</title>
</head>
<body>
     <img src="http://www.startrek.com/legacy_media/images/200508/tng-142-j25-borg-cube/320x240.jpg">
     <h1>Welcome to Borg Fitness</h1>
     <div>@content</div>
</body>
</html>
```

Changing our Create Exercise page to use our template

Try This! *Let's change* `createexercise.scala.html` *to use the template and view the changes at http://localhost:9000/exercise/create. Remember it is just a method call now!*

HTML

```
@(form: Form[Exercise])
@import play.api.data.Form

@template(title="Create a new Exercise") {
    @helper.form(action = routes.FitnessController.createExercise(), 'id -> "exercise_form") {
        <div>
        @helper.inputText(form("name"), '_id -> "name", '_label -> "Exercise:", '_showConstraints -> false)
        </div>
        <div>
        @helper.inputText(form("minutes"), '_id -> "minutes", '_label -> "Minutes:", '_showConstraints -> false)
        </div>
        <input id="submit" name="submit" type="submit" value="Create"/>
    }
}
```

How to set up "The Purdy"

Twitter Bootstrap is a collection of css, javascript, image files that creates an aesthetic web experience.

Remember how routing works

Now that we have a collection of javascript and css files, we need to reference those files. If you take a look at the routes you will notice that there was a route that was prepacked when you created your app for the first time.

```TXT
GET        /assets/*file            controllers.Assets.at(path="/public", file)
```

This route means that if you want to access any of the javascript, css, or other assets you can view them using the `/assets/` prefix.

Using WebJars for Nicer Looking Designs

Visit http://www.webjars.org and look for whatever CSS, Javascript resources that you are looking for.

Try This Let's try a better way to include bootstrap!

- Include `"org.webjars" %% "webjars-play" % "2.3.0-2"` in `libraryDependencies` section of 'build.sbt'
- Add `"org.webjars" % "bootstrap" % "3.3.1"` in `libraryDependencies` section of 'build.sbt'
- Stop the Application
- Run `reload` then `update`
- Start the server again with `run` or `run <port>`

Setting up the route to serve up webjars resources

We need to make sure that there is a route to point to webjar resources

Try This Add the following route!

```
GET     /webjars/*file              controllers.WebJarAssets.at(file)
```

Changing the template page to use the latest webjars

Now that we downloaded the dependencies, we have to change the template to accept the webjars that were downloaded.

Try this Change the `link`, and the two `script` elements to use the reverse routing to pick up the resources. Try out your new reactive application.

HTML

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
    <head>
        <title>@title</title>
        <link rel='stylesheet' href='@routes.WebJarAssets.at(WebJarAssets.locate("css/bootstrap.min.css"))'>
        <script type='text/javascript' src='@routes.WebJarAssets.at(WebJarAssets.locate("jquery.min.js"))'>
</script>
        <script type='text/javascript' src='@routes.WebJarAssets.at(WebJarAssets.locate("bootstrap.min.js"))'>
</script>
    </head>
    <body>
        <img src="http://www.startrek.com/legacy_media/images/200508/tng-142-j25-borg-cube/320x240.jpg"
class="img-polaroid">
        <h1>Welcome to Borg Fitness</h1>
        <div>@content</div>
    </body>
</html>
```

Questions?

Thanks

Last updated 2015-03-10 01:45:25 MDT