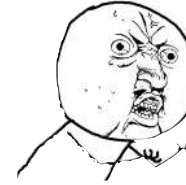


Making most of Scala

Akka, Scala, Spray, Specs2; all in 50 minutes!

Jan Machacek



- Chief whip at Cake Solutions
- Author of Pro Spring, Pro Spring 2.5 and other books & articles
- Contributor to Akka Patterns, Specs2 Spring, Scalad, Spring Extensions, Spock Spring Integration
- Editor of the Open Source Journal
- @honzam399, github.com/janm399, janm@cakesolutions.net

Brings all together

- Object oriented language

Everything is an object: even `1`, `true`, ...; mixin composition (multiple implementation inheritance with terms & conditions)

- Functional programming constructs

Even a function is object that can be bound to a variable, passed as argument

- Static typing and pattern matching

Compiler infers & enforces type safety;

- Compiles to Java bytecode

Adopt Scala slowly, don't throw away your existing Java code

Heavy lifting to DSLs

- Use Scala to implement the most complex code
Java of the future: everything in Java, functions, traits, pattern matching;
rich type inference
- Use Scala to implement the ordinary code
Typical programming tasks can be done with much less syntactical
noise
- Use Scala for DSLs
It is easy to design & implement *statically typed* DSLs

Let's build something

- A *proper* e-commerce app, where you must create an account when you want to buy a widget for £10, giving your date of birth, a letter from the hospital in which your mother was born, ...; then receive text message with an activation code.

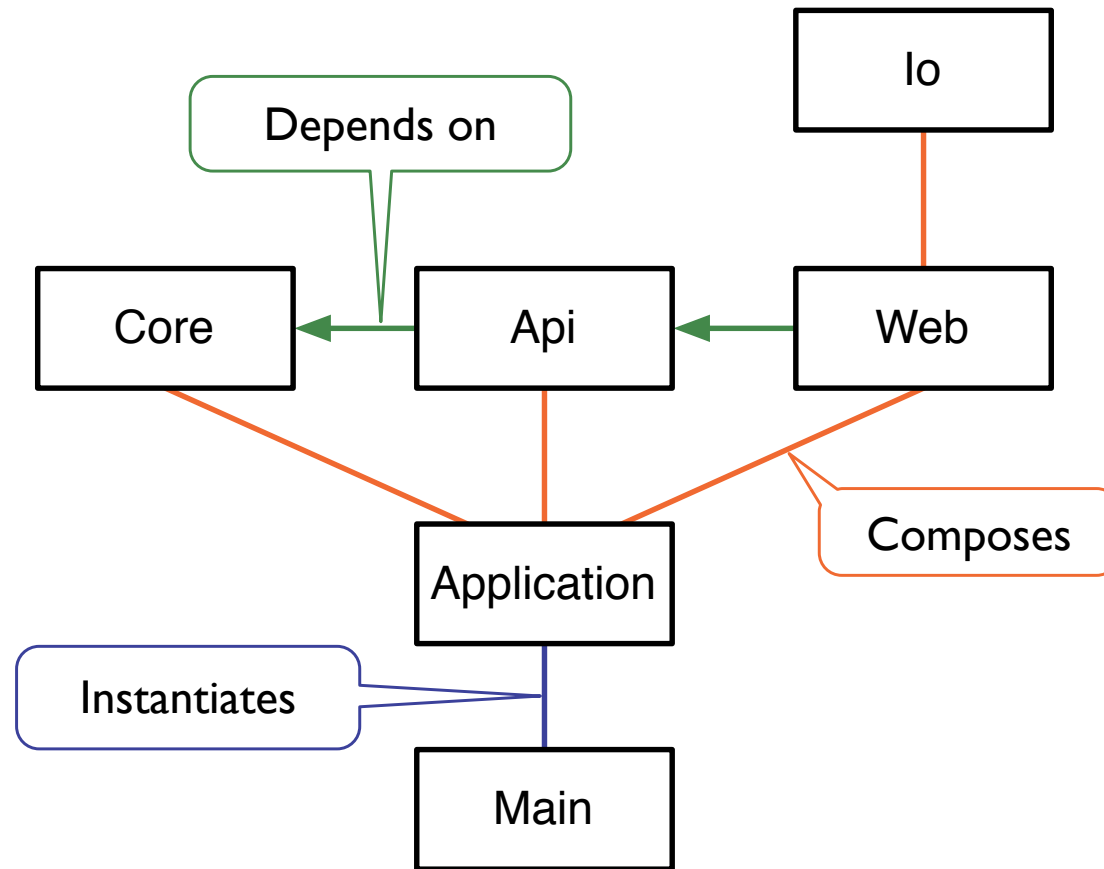


Only to find out that they don't deliver to your work address

Let's build something

- We'll do a HTTP POST with a JSON document that can be mapped onto the User instance
 - Unmarshal the JSON document
 - Register the user and send the activation code
 - Marshal the instance of the reply (succeeded, failed) into a JSON document

Let's build something



The components

```
trait ServerCore {  
  implicit def actorSystem: ActorSystem  
  
  implicit val timeout = Timeout(30000)  
  
  val core = actorSystem.actorOf(  
    Props[ApplicationActor], "application")  
  
  Await.ready(core ? Start(), timeout.duration)  
}
```


The components

```
trait HttpIO {  
  implicit def actorSystem: ActorSystem  
  
  lazy val ioBridge = IOExtension(actorSystem).ioBridge  
  
  private lazy val httpClient: ActorRef =  
    actorSystem.actorOf(Props(new HttpClient(ioBridge)))  
  
  def makeConduit(host: String): ActorRef =  
    actorSystem.actorOf(Props(new HttpConduit(  
      httpClient, host, port = 443, sslEnabled = true)))  
}
```

The components

```
trait HttpIO {  
  implicit def actorSystem: ActorSystem  
  
  lazy val ioBridge = IOExtension(actorSystem).ioBridge  
  
  private lazy val httpClient: ActorRef =  
    actorSystem.actorOf(Props(new HttpClient(ioBridge)))  
  
  def makeConduit(host: String): ActorRef =  
    actorSystem.actorOf(Props(new HttpConduit(  
      httpClient, host, port = 443, sslEnabled = true)))  
}
```

```
trait ActorHttpIO extends HttpIO {  
  this: Actor =>  
  
  final implicit def actorSystem = context.system  
}
```

The components

```
trait Api extends RouteConcatenation {  
  this: ServerCore =>  
  
  val routes =  
    new HomeService().route ~  
    new UserService().route  
  
  val rootService = actorSystem.actorOf(Props(  
    new RoutedHttpService(routes)))  
  
}
```

The components

```
trait Web extends HttpIO {  
  this: Api with ServerCore =>  
  
  val httpServer = actorSystem.actorOf(Props(  
    new HttpServer(ioBridge,  
                  SingletonHandler(rootService))),  
    name = "http-server"  
  )  
  
  httpServer ! HttpServer.Bind("localhost", 8080)  
}
```

The components

```
trait Web {  
  this: Api with ServerCore =>  
  ...  
}
```

```
trait Api {  
  this: ServerCore =>  
  ...  
}
```

```
trait ServerCore {  
  implicit def actorSystem: ActorSystem  
  ...  
}
```

class Application(val actorSystem: ActorSystem) extends Api

class Application(val actorSystem: ActorSystem) extends Web

class Application(val actorSystem: ActorSystem) extends Api with Web

**class Application(val actorSystem: ActorSystem)
 extends ServerCore with Api with Web**

The components

```
object Main extends App {  
  val system = ActorSystem("AkkaPatterns")
```

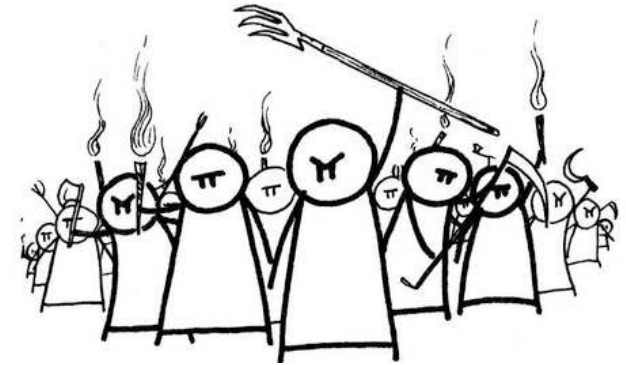
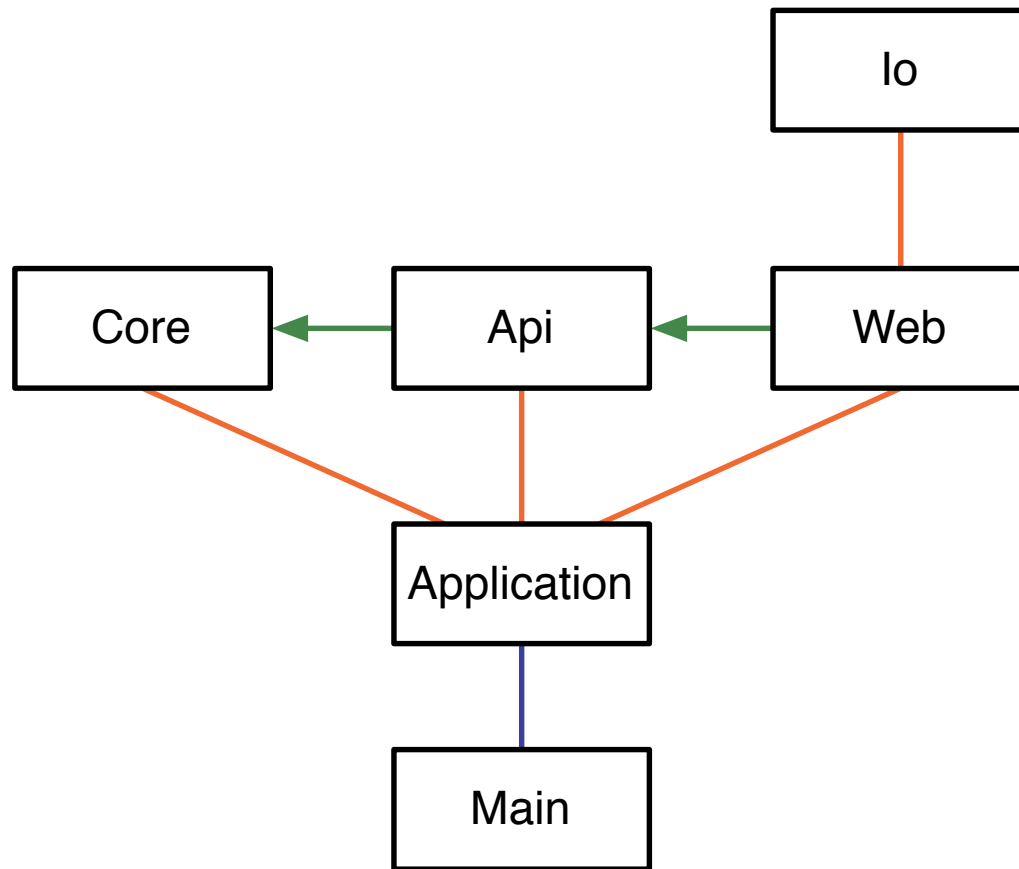
```
  class Application(val actorSystem: ActorSystem) extends
```

```
    ServerCore with  
    Api with  
    Web
```

```
    trait ServerCore {  
      implicit def actorSystem: ActorSystem  
      ...  
    }
```

```
    new Application(system)  
  }
```

Let's build something



Let's build something

Application {

POST /users (ru: RegisterUser) →
422: Failure
200: RegisteredUser

GET /users/{UserReference} →
404: None
200: Some(User)

}

The components

```
trait Api extends RouteConcatenation {  
  this: ServerCore =>
```

```
  val routes =  
    new HomeService().route ~  
    new UserService().route
```



```
val route =  
  path("users") {  
    post {  
      handleWith { ru: RegisterUser =>  
        (userRegistrationActor ? ru).mapTo[  
          Either[ApplicationFailure, RegisteredUser]]  
      }  
    }  
  }
```

A message!

```
class UserService(implicit val actorSystem: ActorSystem)
  extends Directives with UserServiceMarshallers
  with DefaultTimeout {

  def userRegistrationActor =
    actorSystem.actorFor("/user/application/registration")

  val route =
    path("users") {
      post {
        handleWith { ru: RegisterUser =>
          (userActor ? ru).mapTo[
            Either[ApplicationFailure, RegisteredUser]]
        }
      }
    }
}
```

A message!

```
case class RegisterUser(  
    username: String,  
    password: String,  
    email:    String,  
    mobile:   String)
```

A message!

```
case class RegisterUser(  
    username: String,  
    password: String,  
    email:    EmailAddress,  
    mobile:   MobileNumber)  
  
sealed trait Address  
  
case class EmailAddress(address: String) extends Address  
  
case class MobileNumber(number: String) extends Address
```

A message!

```
case class RegisterUser(  
    username: String,  
    password: String,  
    email:     EmailAddress,  
    mobile:    MobileNumber)
```

```
sealed trait Address
```

```
case class EmailAddress(address: String) extends Address
```

```
case class MobileNumber(number: String) extends Address
```

```
case class RegisteredUser(user: User)
```

```
trait ApplicationFailure
```

```
case class ValidationFailed(...) extends ApplicationFailure
```

```
case object UsernameTaken extends ApplicationFailure
```

A message!

```
case class User(id: UserReference,  
                username: String, hashedPassword: String,  
                activationCode: Option[String],  
                email: EmailAddress,  
                mobile: MobileNumber)
```

```
package object domain {  
  type UserReference = UUID  
}
```

A message!

```
class UserService(implicit val actorSystem: ActorSystem)
  extends Directives with UserServiceMarshallers
  with DefaultTimeout {

  def userRegistrationActor =
    actorSystem.actorFor("/user/application/registration")

  val route =
    path("users") {
      post {
        handleWith { ru: RegisterUser =>
          (userRegistrationActor ? ru).mapTo[
            Either[ApplicationFailure, RegisteredUser]]
        }
      }
    }
}
```

RegisterUser



ApplicationFailure

RegisteredUser

RegisterUser



ApplicationFailure

RegisteredUser

Hierarchies



http://en.wikipedia.org/wiki/Class_sketch

The user actor structure

- UserActor creates and supervises the
 - MessageDeliveryActor, which sends e-mails and text messages
 - UserRegistrationActor, which takes care of the registration (and sends the activation code using the MessageDeliveryActor)
 - UserManagementActor, which provides the boring “management” operations

The user actor structure

```
class UserActor extends Actor {  
  val messageDelivery =  
    context.actorOf(Props[MessageDeliveryActor])  
  val registration =  
    context.actorOf(  
      Props(new UserRegistrationActor(messageDelivery)),  
      "registration")  
  val management =  
    context.actorOf(  
      Props(new UserManagementActor(messageDelivery)),  
      "management")  
  ...  
}
```

```
class MessageDeliveryActor extends Actor  
class UserRegistrationActor(md: ActorRef) extends Actor  
class UserManagementActor(md: ActorRef) extends Actor
```

The user actor structure

```
class UserRegistrationActor(messageDelivery: ActorRef)
  extends Actor {

  def receive = {
    case RegisterUser(username, password, email, mobile) =>
      val user = ... // prepare the User instance
      messageDelivery ! DeliverActivationCode(
        mobile, user.activationCode.get)
      sender ! Right(RegisteredUser(user))
  }
}
```

The user actor structure

```
class MessageDeliveryActor extends Actor {  
  
  def receive = {  
    case DeliverActivationCode(MobileNumber(number), code) =>  
      deliverTextMessage(number, "Your code is " + code)  
  }  
}
```

The user actor structure

```
class MessageDeliveryActor extends Actor {  
  this: TextMessageDelivery =>  
  
  def receive = {  
    case DeliverActivationCode(MobileNumber(number), code) =>  
      deliverTextMessage(number, "Your code is " + code)  
  }  
}
```

The user actor structure

```
class MessageDeliveryActor extends Actor {  
  this: TextMessageDelivery =>  
  
  def receive = {  
    case DeliverActivationCode(MobileNumber(number), code) =>  
      deliverTextMessage(number, "Your code is " + code)  
  }  
}  
  
trait TextMessageDelivery {  
  def deliverTextMessage(number: String, message: String)  
}
```


The user actor structure

```
class MessageDeliveryActor extends Actor {  
  this: TextMessageDelivery =>  
  
  def receive = {  
    case DeliverActivationCode(MobileNumber(number), code) =>  
      deliverTextMessage(number, "Your code is " + code)  
  }  
}  
  
trait TextMessageDelivery {  
  def deliverTextMessage(number: String, message: String)  
}  
  
trait NexmoTextMessageDelivery extends TextMessageDelivery {  
  this: HttpIO =>  
  private lazy val pipeline =  
    HttpConduit.sendReceive(makeConduit("rest.nexmo.com"))  
  
  def deliverTextMessage(number: String, message: String) {  
    val request = HttpRequest(POST, "https://...")  
    pipeline(request) onSuccess { ... }  
  }  
}
```

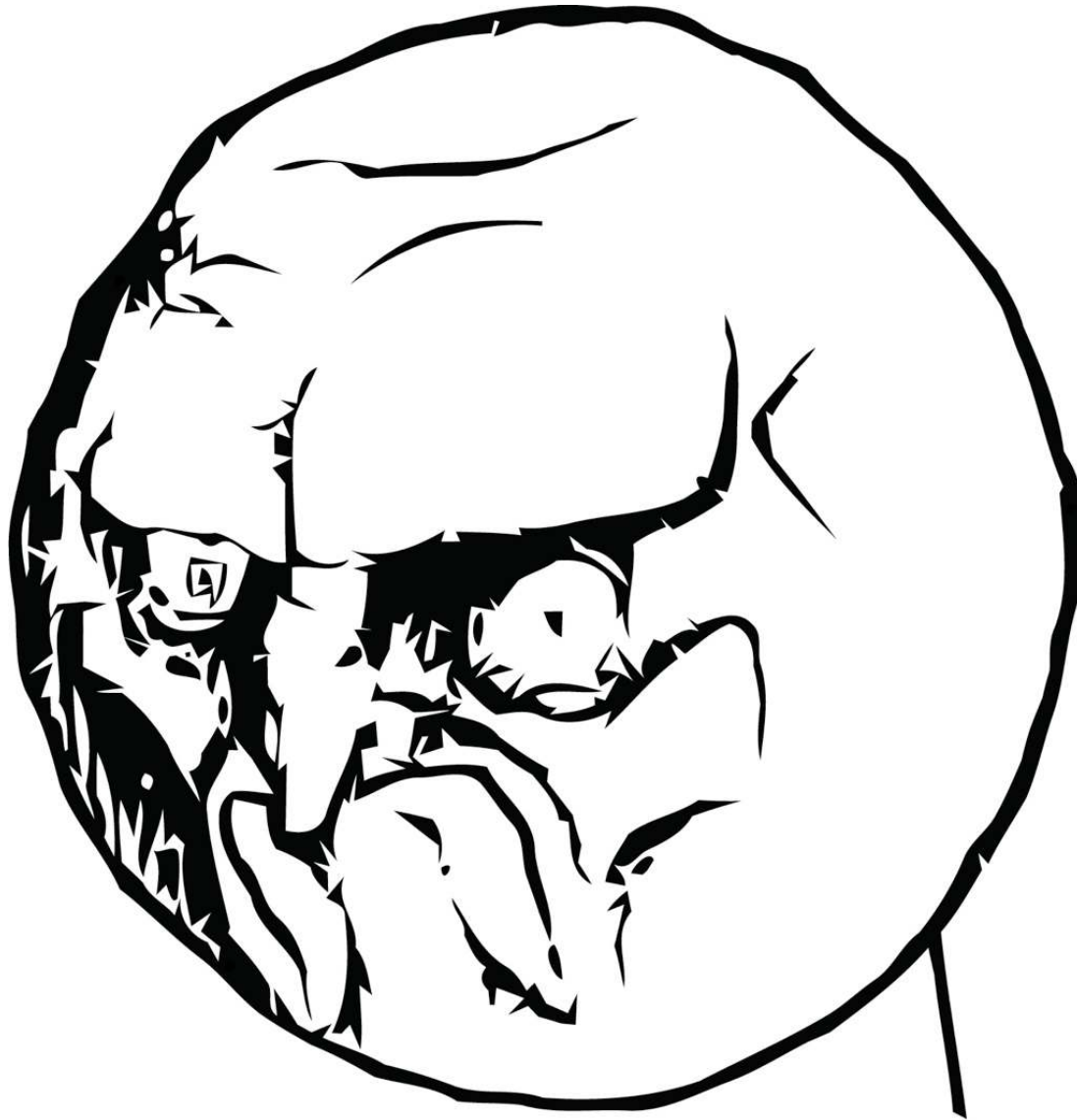
The user actor structure

```
class UserActor extends Actor {  
  val messageDelivery =  
    context.actorOf(Props[MessageDeliveryActor])  
  
  val registration =  
    context.actorOf(  
      Props(new UserRegistrationActor(messageDelivery)),  
      "registration")  
  val management =  
    context.actorOf(  
      Props(new UserManagementActor(messageDelivery)),  
      "management")  
  ...  
}
```

The user actor structure

```
class UserActor extends Actor {  
  val messageDelivery =  
    context.actorOf(Props(  
      new MessageDeliveryActor  
        with MexmoTextMessageDelivery  
        with ActorHttpIO))  
  val registration =  
    context.actorOf(  
      Props(new UserRegistrationActor(messageDelivery)),  
      "registration")  
  val management =  
    context.actorOf(  
      Props(new UserManagementActor(messageDelivery)),  
      "management")  
  ...  
}
```

Testing



Observe the messages

```
class UserRegistrationActorSpec extends
  TestKit(ActorSystem()) with Specification with
  ImplicitSender {

  val actor = TestActorRef(
    new UserRegistrationActor(testActor))

  "register the user and send the code" in {
    actor ! RegisterUser("janm", "*****", ...)
    val code = expectMsgType[DeliverActivationCode]
    val user = expectMsgType[Either[
      ApplicationFailure, RegisteredUser]]
    user.activationCode.get == code

    case RegisterUser(username, password, email, mobile) =>
      val user = ... // prepare the User instance
      messageDelivery ! DeliverActivationCode(
        mobile, user.activationCode.get)
      sender ! Right(RegisteredUser(user))
  }
}
```

Observe the messages

```
class UserRegistrationActorIntegrationSpec extends
  TestKit(ActorSystem()) with Specification with
  ImplicitSender with CoreSystem {

  val actor = system.actorFor(
    "/user/application/user/registration")

  "register the user and send the code" in {
    actor ! RegisterUser("janm", "*****", ...)

    val user = expectMsgType[Either[
      ApplicationFailure, RegisteredUser]]

    ...
  }
}
```

Observe the messages

```
class UserServiceSpec extends TestKit(ActorSystem())  
  with TestKitRouteTest with Specification  
  with ImplicitSender with CoreSystem with Api {
```

```
  "register the user and send the code" in {  
    val username = "janm"  
    Post(login, RegisterUser(username, "****", ...)) ~>  
      routes ~> check {  
        val resp = entityAs[RegisteredUser]  
        resp.user.username mustEqual username  
      }  
  }  
}
```

```
case class RegisteredUser(user: User)
```

Scala, Akka, Spray, ...

- *Akka* brings asynchronous & non-blocking behaviour (!, ?) and ability to dynamically change the structure of your application at runtime
- *Spray* wraps it in REST API
- *Specs2* simplifies very expressive tests
- *Scala* gives type safety and expressiveness, functions as first-class citizens (directives in the route), very flexible way to “inject” dependencies

