CONTENTS

Reactive Programming with Akka

Message-Driven, Elastic, Resilient, and Responsive Java

UPDATED BY JOHAN ANDRÉN
ORIGINAL BY RYAN KNIGHT

INTRODUCTION

Akka is a toolkit and runtime for building concurrent, distributed, and reactive applications and systems on the JVM. The Reactive Manifesto (reactivemanifesto.org) defines reactive in terms of its four guiding principles:

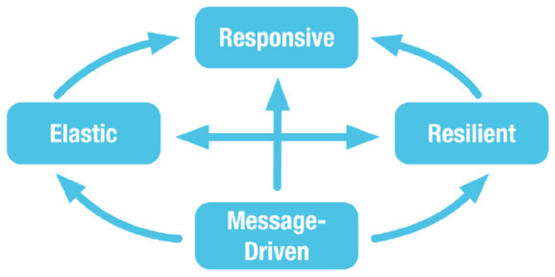
1. Message-driven
2. Elastic
3. Resilient
4. Responsive

Message-driven means the application reacts to events by using a message-driven programming model. This allows the application to more effectively share resources by doing work only in response to outside messages.

Elastic means the application is able to react to increasing load by making the architecture highly concurrent and distributed.

When an application is **Resilient**, it can easily deal with failure and recovery. Instead of the application simply dying, it manages failure through fault isolation so other parts of the application can keep running.

The final property, being **Responsive**, means the application responds quickly not only in ideal conditions, but also when there are application failures.



Akka was designed to enable developers to easily build reactive systems using a high level of abstraction, without having to deal with low-level concepts like threads, mutexes, and deadlocks. It does so by leveraging the Actor Model of concurrency and fault-tolerance. This is a powerful model that allows the behavior and state of the application to be encapsulated and modeled as an actor in a very natural and simple way. The key principle behind an actor is that the application only interacts with it through messages and never talks with it directly. This abstraction allows actor-based applications to seamlessly scale from a single machine to large clusters.

CREATING A NEW AKKA APPLICATION

To create a new Akka application, use your favorite IDE and build tool to create a new empty project. Akka is provided as regular jar files available through Maven central, so you can use it by just adding the following dependency to your project (after taking a look at akka.io to see which is the latest stable version). NOTE: 2.5.0 will be available very soon.

Maven:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-actor_2.11</artifactId>
  <version>2.5.0-RC1</version>
</dependency>
```

Gradle:

```
compile ‘com.typesafe.akka:akka-actor_2.11:2.5.0’
```

SBT:

```
“com.typesafe.akka” %% “akka-actor” % “2.5.0-RC1”
```

With the project in place, create a simple Java class called Application with a main method, and inside of that, create an ActorSystem:

Because microservices and streaming applications need more than clouds and endpoints.

Based on the Actor Model, everything in Akka is designed to work in a distributed setting. It is stateless, asynchronous, and location transparent and as such ideally suited for hybrid cloud architectures and the elastic scaling of cloud platforms.

- Distributed by default
- Resilient self-healing
- Integration with back pressure
- Advanced healing
- Expert instrumentation for monitoring

Learn to build Reactive Systems with Akka with a **free eBook**

DOWNLOAD EBOOK

An annual subscription to Lightbend Reactive Platform is the perfect foundation for your modernization strategy and enables your truly distributed microservices architectures to perform reliable and efficiently at any scale.

“We’re delivering the future of money to over 148 million people worldwide with Akka.” - PayPal

akka Lightbend

```
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;

public class Application {

  public static void main(String[] args) throws
Exception {
    final ActorSystem system = ActorSystem.
create();

    System.out.println("Press any key to
terminate");
    System.in.read();
    System.out.println("Shutting down actor
system...");
    system.terminate();
  }
}
```

`ActorSystem` starts and maintains thread pools, and will keep running until you explicitly tell it to terminate, so in this sample we block the main thread until a key is pressed, which triggers termination. If we did not do that, the main method would return and the **`ActorSystem`** would keep running.

This Refcard introduces Akka by modeling a simple robot named *AkkaBot* with Akka's Java APIs. There is a full set of corresponding Scala APIs, but those will not be covered. The AkkaBot will be modeled as an actor, which is either moving or still, and has a direction it is moving in.

WHAT IS THE ACTOR MODEL?

The Actor Model was originally invented by Carl Hewitt in 1973 and has seen a recent resurgence in popularity. In the Actor Model, an application is broken up into small units of execution called actors. These actors encapsulate the behavior and state of a part of the application. The first part of the definition of actors makes them sound very much like objects in a standard object-oriented design, which is true. However, what distinguishes an actor from a standard object is a third property: communication.

Actors never interact directly with each other and instead communicate via messages. The behavior of an actor is primarily defined by how it handles messages as they are received. Since the state of an actor is isolated from the rest of the system, an actor never has to worry about synchronization issues, such as thread locking, when modifying its state.

Actors in Akka are extremely lightweight since they are standard objects that only consume a few hundred bytes each. The only constraint in number of actors is the amount of memory they consume. This means that a single application can easily create thousands or even millions of concurrent actors.

An important part of the Actor Model is that you never create actors directly, Instead, they are created by the ActorSystem. The returned value is not the actual actor; instead it is a reference to that actor called an ActorRef. There are three advantages to using this level of indirection and never accessing the actor directly:

1. The actor can be transparently restarted after failure.
2. The location of the actor is transparent, allowing Akka to manage when and where the actor runs.
3. The actor is able to maintain mutable state without worrying about concurrent access to this state.

Shared mutable state is a cause of many concurrency problems. Akka actors solve this problem by processing a single message at a time and transparently handling memory visibility when the actor executes on different threads over time. Many threads can hold an **`ActorRef`** and send messages to it concurrently, yet the actor will process them in a linear fashion. Accessing or mutating the internal state of an actor is fully thread safe since the thread is protected by the Actor Model.

The strong isolation principles of actors, together with the message-driven model and location transparency, make it easy to solve hard concurrency and scalability problems in an intuitive way. It abstracts away the underlying threading model and actor scheduling so you can focus on building your application and not infrastructure.

An actor is created by defining a class that extends **`AbstractActor`** and implements the **`createReceive`** method to define how the actor should react to the different messages it receives.

To start building our robot, the AkkaBot, we first need to create an actor that will represent it. It will contain information about whether the robot is moving the direction of movement. It has an **`onReceive`** method that defines its behavior for how it should react upon receiving a move message.

DEFINING AN ACTOR

Create the file AkkaBot.java in the source directory and enter the Java code below:

```
import akka.actor.AbstractActor;

public class AkkaBot extends AbstractActor {

  public Receive createReceive() {
    return receiveBuilder().build();
  }
}
```

CREATING ACTORS

Now that we have defined the actor, let's create an instance of it. Since we need Akka to manage the actor's lifecycle, we don't create it directly just using the "new" operator, but instead ask the **`ActorSystem`** to create and manage the actor for us. What we get back is not an instance of the actor itself, but an **`ActorRef`** pointing to our actor instance.

This level of indirection adds a lot of power and flexibility. It enables location transparency, which means that the **`ActorRef`** can, while retaining the same semantics, represent an instance of the running actor in-process or on a remote machine (i.e. location doesn't matter). This also means that the runtime can optimize the system by changing an actor's location or the application's topology while it is running. This level of indirection also enables the "let it crash" model of failure management, in which the system can heal itself by crashing and restarting faulty actors.

To create (top level) actors in Akka you use the **`ActorSystem.actorOf`** factory method. This method takes a configuration object called **`Props`** and an optional name.

The **`ActorSystem`** in Akka is also much more than just a factory for creating actors. It manages the entire lifecycle of the actor, maintains the execution contexts (thread pools) in which actors run, a scheduling service, an event stream of what is happening, and more.

Previously, we added the basic code for creating an **`ActorSystem`** and the AkkaBot in the **`Application`** class. Now we'll update it to actually create an instance of your actor in the main method:

```
public static void main(String[] args) throws
Exception {
  final ActorSystem system = ActorSystem.create();

  final ActorRef akkaBot = system.actorOf(
    Props.create(AkkaBot.class),
    "akkaBot");
```

Creating an actor using the **`ActorSystem`** directly creates the actor at the top of the hierarchy. Actors can also be created as children of other actors using an actor's local **`ActorContext`**. The **`ActorContext`** contains information about the ActorSystem relevant to each actor, such as who its parent and children are. When an actor uses this context to create another actor, the new actor becomes a child actor. In this way, the actor hierarchy gets built out.

DEFINING MESSAGES

An actor does not have a public API in terms of methods that you can invoke. Instead, its public API is defined through a protocol, the set of messages, that the actor handles. Messages can be of arbitrary type (any subtype of **`Object`**). This means that we can send boxed primitive values (such as **`String`**, **`Integer`**, **`Boolean`**, etc.) as messages or plain data structures like arrays, collection types, and value objects. However, since the messages are the actor's public API, you should define messages with good names and rich semantic and domain-specific meaning, even if it's just wrapping your data type. This makes it easier to use, understand, and debug actor-based systems.

This is typically done using public static classes defined together in the beginning or end of the actor class. Then, when someone is looking at the actor code, they can easily see what messages are handled for the actor. This also makes the messages part of the auto-generated Javadoc.

Now we want to define two different messages:

- **`Move`** starts the robot moving
- **`Stop`** stops the robot

Let's define the messages by putting them inside the **`AkkaBot`** class. It is very important that the messages we create are immutable, meaning that they cannot be changed after construction, this guarantees that it is safe to share the messages between different threads of the JVM without using any concurrency primitives such as locks.

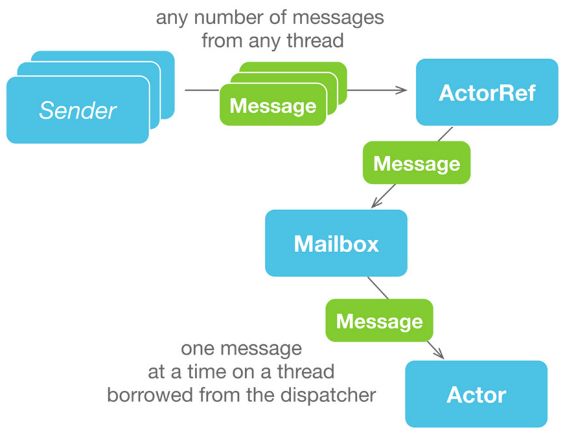
```
// Inside AkkaBot.java code add the following inner
classes
public enum Direction { FORWARD, BACKWARDS, RIGHT, LEFT }
public static class Move {
  public final Direction direction;
  public Move(Direction direction) {
    this.direction = direction;
  }
}
public static class Stop {}
public static class GetRobotState {}
public static class RobotState {
  public final Direction direction;
  public final boolean moving;
  public RobotState(Direction direction, boolean moving) {
    this.direction = direction;
    this.moving = moving;
  }
}
```

TELL THE ACTOR (TO DO SOMETHING)

All communication with actors is done through asynchronous message passing. This is what makes actors reactive. An actor doesn't do anything unless it's been told to do something, and you tell it to do something by sending the message.

Sending a message asynchronously means that the sender does not stick around waiting for the message to be processed by the recipient actor. Instead, the sender hands the message off by putting it on the recipient's **`Mailbox`** and is then free to do something more important than waiting for the recipient to react on the message.

The mailbox is not accessed directly in code, the sender of a message just sends a message to the **`ActorRef`**, and the actor simply has the message delivered to it, however it is useful to remember that the messages actually sit around in the mailbox until processed.



The actor's mailbox is essentially a message queue and has ordering semantics. This guarantees that the ordering of multiple messages sent from the same sender is preserved, while the same messages can be interleaved with the messages sent by another sender. When the actor is not processing messages it does not consume any resources, apart from memory.

You tell an actor to do something by sending it a message with the tell method on the **`ActorRef`**. This method puts the message on the actor's mailbox and then returns immediately.

Internally, the actor chooses its behavior depending on the type and properties of incoming messages. This behavior can be any standard logic such as modifying the internal state of an actor, creating or sending messages to other actors, business logic, or even changing the behavior of the actor itself.

To define the behaviors of an actor, you implement **`createReceive`** and return an instance of Receive, which is created using the method **`receiveBuilder()`**. After defining the types of messages that are handled, the builder is made to construct a Receive by calling **`build()`**.

First let's define the behavior for the **`Move`** and **`Stop`** messages. Make the following changes to AkkaBot.java:

```
// Add the following inside AkkaBot
private Optional<Direction> direction = Optional.empty();
private boolean moving = false;

public Receive createReceive() {
  return receiveBuilder()
      .match(Move.class, this::onMove)
      .match(Stop.class, this::onStop)
      .build();
}

private void onMove(Move move) {
  moving = true;
  direction = Optional.of(move.direction);
  System.out.println("I am now moving " + direction.
get());
}

private void onStop(Stop stop) {
  moving = false;
  System.out.println("I stopped moving");
}
```

We can now test out the actors by sending them some simple commands from the Bot Main app.

Add the following to the main method in the Application class:

```
akkaBot.tell(
  new AkkaBot.Move(AkkaBot.Direction.FORWARD),
  ActorRef.noSender());
akkaBot.tell(
  new AkkaBot.Move(AkkaBot.Direction.BACKWARDS),
  ActorRef.noSender());
akkaBot.tell(
  new AkkaBot.Stop(),
  ActorRef.noSender());
```

When you run the application, you should see some basic logging of the application such as:

```
I am now moving FORWARD
I am now moving BACKWARDS
I stopped moving
```

THE 'SELF' REFERENCE

Sometimes the communication pattern is not just one-way, but instead lends itself towards request-reply. One explicit way of doing that is by adding a reference of yourself as part of the message so the receiver can use that reference to send a reply back to you. This is such a common scenario that Akka directly supports it. For every message you send, you have the option of passing along the sender reference (the actor's **`ActorRef`**). If you are sending a message from within an actor, you have access to your own **`ActorRef`** through the self-reference. You can access the self-reference through the **`getSelf()`** method.

```
// From within an actor
actorRef.tell(
  new AkkaBot.Move(AkkaBot.Direction.FORWARD),
  getSelf());
```

If you are not inside an actor or do not want to pass the sender, use **`ActorRef.noSender()`** instead. An example of this case is in the main class where messages were sent to the robot without a sender.

THE 'SENDER' REFERENCE

Since each message is paired with its unique sender reference, the "current" sender reference will change with each new message you process. You can access it using the getSender() method. For example, to send a message back to the sender of the message that is being handled, use:

```
// From within an actor
getSender().tell(new Greeting(greeting), getSelf());
```

If you need to use a specific sender reference after some asynchronous processing, like after messaging with other actors, you will need to store a reference to the sender in a variable. The reason for this is that the sender might change if other messaging or processing happens in the meantime.

In some cases there might not be a sender, like when a message is sent from outside an actor or the sender was **`Actor.noSender`**. In these cases, sending a message to sender would cause it to be sent to dead letters. The dead-letter actor is where all unhandled messages end up. For debugging and auditing purposes, you can watch for messages to the dead-letter actor.

ACTOR HIERARCHIES

The real power of the Actor Model is in actor hierarchies. An actor hierarchy is when a parent actor creates and supervises child actors.

This structure helps avoid cascading failures that can take down an entire system by isolating and supervising child nodes.

Creating child actors is very similar to creating top level actors - the only difference is the context the child actors are created in. The actor context can be thought of as where in the actor hierarchy an actor lives. As an example, let's create a Bot Master that creates several children. Add the class **`BotMaster`** that creates several children in the constructor:

```
import akka.actor.AbstractActor;
import akka.actor.Props;

public class BotMaster extends AbstractActor {

  public BotMaster() {
    for (int index = 0; index < 10; index++) {
      getContext().actorOf(Props.create(AkkaBot.class));
    }
  }

  @Override
  public Receive createReceive() {
    return emptyBehavior();
  }
}
```

We access the local actor context and create new actors in that context. That makes all of the AkkaBot actors we created children of the Bot Master actor. Now that the Bot Master has children, it can interact with the children directly. To do this, let's add a simple method to make the child bots move.

Add the **`StartChildBots`** message:

```
// inside the Bot Master class
public static class StartChildBots {}
```

Then modify what messages the Bot Master handles:

```
// replace the previous emptyBehavior
public Receive createReceive() {
  return receiveBuilder()
      .match(StartChildBots.class, this::onStartChildBots)
      .build();
}

private void onStartChildBots(StartChildBots
startChildBots) {
  final AkkaBot.Move move =
    new AkkaBot.Move(AkkaBot.Direction.FORWARD);
  for (ActorRef child : getContext().getChildren()) {
    System.out.println("Master started moving " + child);
    child.tell(move, getSelf());
  }
}
```

What this does is look in the Master Bot's context for all of its children. It then iterates through the children and sends each of them a message. As the message is immutable, it is perfectly safe to send the same message instance to all the children.

To test this out, let's modify the Akka Bots so they print out their own path to make it easier to see where trace statements are coming from. Add a prefix of **`getSelf().path()`** to the **`println`**'s in AkkaBot:

```
System.out.println(getSelf().path() + ": I am now moving "
+ direction.get());
```

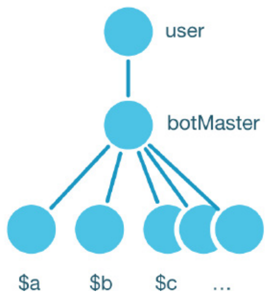
Then modify the Application main method to start the Bot Master instead of the AkkaBot:

```
final ActorRef botMaster = system.actorOf(
  Props.create(BotMaster.class),
  "botMaster");

botMaster.tell(new BotMaster.StartChildBots(), ActorRef.
noSender());
```

The method call to **`getSelf().path()`** returns the path of the current actor. This path will be something like:

```
/user/botMaster/$a
```

User is a top level system actor that is always the root parent of all user-created actors, *botMaster* we created by calling **`ActorSystem.actorOf`**, and $a is the actual bot actor. Since we didn't give it an explicit name, Akka gave it a name that is unique among its siblings. From the path we can see this actor is a child of *botMaster*.



A child actor can get the **`ActorRef`** of its parent through **`getContext().parent()`**.

This exercise should give you a good sense for the context in where an actor lives in the hierarchy of other actors, since you can see how an actor can get its parents and children.

FAULT-TOLERANT AND SELF-HEALING

The reason actor hierarchies are so powerful is they provide a way to build systems that are fault-tolerant and self-healing. This can be done by isolating the errors to child actors and monitoring those actors for failures.

Failure (exceptions) is a normal part of any application, and can be a result of a wide variety of causes. With actors, the failures can happen during startup, message processing, or other lifecycle events. What is unique about the actor model is that failure doesn't cause the entire application to crash or pass the responsibility to solve the problem to the calling code; rather the fault is isolated to an individual actor and the failure is passed to its parent, which gets to decide what should happen.

There are two ways actors can deal with failures. The first is through customizing the supervision strategy. Each actor has a default supervisor strategy that defines how it handles failures in its children. The default supervisor strategy for an exception is to restart the actor. This supervisor strategy can be overridden to define a different behavior. Besides restart, other possible actions include escalating the exception up the hierarchy, simply resuming the actor, or stopping the actor.

Resuming and restarting might seem to be the same, however there is one key difference. When an actor is restarted, a new instance of the actor is created, which resets its state. If an actor is resumed, the actor state is maintained. With both strategies the message that caused the fault is lost, but the actor does not lose any of its other messages waiting in the mailbox.

The second way a parent can manage the failures of its children is to watch the children. Then the parent receives an `akka.actor.Terminated` message when the child is terminated. As an example, let's assume that if a child robot stops, we want to do some special processing. To do this we simply add a watch after creating the robot.

This is done by changing the constructor of the Bot Master by watching each created child:

```
public BotMaster() {
  for (int index = 0; index < 10; index++) {
    final ActorRef child =
      getContext().actorOf(Props.create(AkkaBot.class));
    getContext().watch(child);
  }
}
```

And then handle the case of receiving a `Terminated` message:

```
public Receive createReceive() {
  return receiveBuilder()
      .match(StartChildBots.class, this::onStartChildBots)
      .match(Terminated.class, this::onChildTerminated)
      .build();
}

private void onChildTerminated(Terminated terminated) {
  System.out.println("Child has stopped, starting a new
one");
  final ActorRef child =
    getContext().actorOf(Props.create(AkkaBot.class));
  getContext().watch(child);
}
```

Now we need to make the child randomly fail. This can be done by adding the following to the Move message handler in the AkkaBot:

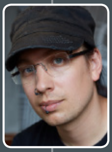
```
System.out.println(getSelf().path() + ": I am now moving "
+ direction.get());

final Random random = new java.util.Random();
final int nextInt = random.nextInt(10);
if ((nextInt % 2) == 0) {
  getContext().stop(getSelf());
}
```

You can now see how the actors deal with failure.

FURTHER LEARNING

Now you have learned the basics of Akka actors, but there's a lot more to it than just that. Actors can trivially be distributed across multiple nodes within a cluster, handle load balancing and sharding as well as persistence, or streaming using Akka Streams. If you want to learn more about what is in the toolkit, check out the Akka reference documentation at akka.io/docs and the Akka samples at github.com/akka/akka-samples.

ABOUT THE AUTHOR



JOHAN ANDRÉN is a senior developer and member of the core Akka Team at Lightbend. He is based out of Stockholm, Sweden where he has been working as a consultant and mentor around Scala, Akka and Play Framework the last 5 years and Java the last 10. In his spare time he co-organizes Scala Usergroup Stockholm and when not in front of a keyboard he enjoys mountain biking and bouldering.