FREE eBook

LEARNING akka

Free unaffiliated eBook created from **Stack Overflow contributors.**

#akka

Table of Contents

About ... 1

Chapter 1: Getting started with akka ... 2

Remarks ... 2

Examples ... 2

Installation or Setup ... 2

Chapter 2: Actor DSL ... 4

Examples ... 4

Simple Actor DSL ... 4

Context switching ... 4

Life-cycle Management ... 4

Nested Actors ... 5

Supervision ... 5

Stash support ... 5

Chapter 3: Akka HTTP ... 6

Introduction ... 6

Examples ... 6

Akka HTTP server: Hello World (Scala DSL) ... 6

Chapter 4: Akka Streams ... 7

Examples ... 7

Akka Streams: Hello World ... 7

Akka-Streams: subflows ... 9

Chapter 5: akka-streams custom shapes ... 10

Remarks ... 10

Examples ... 10

TwoThreeShape ... 10

Chapter 6: Dispatchers ... 12

Examples ... 12

Default Dispatcher ... 12

Setting the dispatcher for an Actor ... 12

Chapter 7: Hello world ... 14

Examples 14
Akka hello world (Scala) 14
Simple Actor Implementation 15
Akka Hello World (Java 8) 16
Chapter 8: Injecting dependencies into an actor 18
Examples 18
Spring-wired actor 18
Chapter 9: Supervision and Monitoring in Akka 21
Remarks 21
Examples 21
What is supervision? 21
Supervision Strategies 23
What is Monitoring? 24
Code Repository 24
Credits 26

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: akka

It is an unofficial and free akka ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official akka.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with akka

Remarks

Akka is an open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM. It implements the actor model known from Erlang.

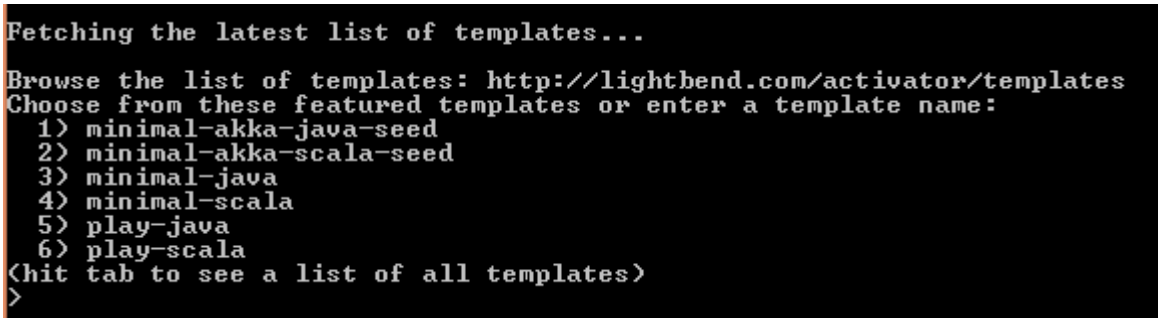
It should also mention any large subjects within akka, and link out to the related topics. Since the Documentation for akka is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

1. Install JDK 8 (Windows, Linux) and set the path (Windows).
2. Install Scala (Linux), For Windows visit http://www.scala-lang.org/download/ download and install binary distribution, set the environment variable for scala in PATH which is in `\scala\bin`.
3. Installing Typesafe activator (It contains Scala, Akka, Play, SBT) + project scaffolding and templates. For quick start download the `mini-package`.
4. Extract the Typesafe activator and set the PATH to `activator-x.x.xx-minimal\bin` (It includes the bash and bat scripts to run the activator).
5. Time to create a sample project and import into your favorite IDE.

- Type `activator new` in cmd/terminal.



- You can choose `4` because Hello World example is based on Scala.
- Import the project to your favorite IDE and start with the Hello World example.
- Done !.

1. Download akka-2.0.zip distribution of Akka from http://akka.io/downloads/

2. Unzip akka-2.0.zip in any directory. (Example - /home/USERNAME/tools/akka-2.0) You would like to have Akka installed.

3. Set the `AKKA_HOME`

4. For Linux.

```
# First got to the installed location
cd /home/USERNAME/tools/akka-2.0

# Export the location as AKKA_HOME
export AKKA_HOME=`pwd`

# Check if PATH is Exported.
echo $AKKA_HOME
/home/USERNAME/tools/akka-2.0
```

5. For Windows

```
# First got to the installed location
C:\USERNAME\akka> cd akka-2.0

# Set the location as AKKA_HOME
C:\USERNAME\akka\akka-2.0> set AKKA_HOME=%cd%

# Check if PATH is Exported.
C:\USERNAME\akka\akka\akka-2.0> echo %AKKA_HOME%
C:\USERNAME\akka\akka-2.0
```

Read Getting started with akka online: https://riptutorial.com/akka/topic/2041/getting-started-with-akka

Chapter 2: Actor DSL

Examples

Simple Actor DSL

To create simple actors without creating a new class, you can use:

```
import akka.actor.ActorDSL._
import akka.actor.ActorSystem

implicit val system = ActorSystem("demo")

val a = actor(new Act {
  become {
    case "hello" ⇒ sender() ! "hi"
  }
})
```

Context switching

The two possible ways of issuing a context.become (replacing or adding the new behavior) are offered separately to enable a clutter-free notation of nested receives:

```
val a = actor(new Act {
  become { // this will replace the initial (empty) behavior
    case "info" ⇒ sender() ! "A"
    case "switch" ⇒
      becomeStacked { // this will stack upon the "A" behavior
        case "info"   ⇒ sender() ! "B"
        case "switch" ⇒ unbecome() // return to the "A" behavior
      }
    case "lobotomize" ⇒ unbecome() // OH NOES: Actor.emptyBehavior
  }
})
```

Life-cycle Management

Life-cycle hooks are also exposed as DSL elements, where later invocations of the methods shown below will replace the contents of the respective hooks:

```
val a = actor(new Act {
  whenStarting { testActor ! "started" }
  whenStopping { testActor ! "stopped" }
})
```

The above is enough if the logical life-cycle of the actor matches the restart cycles (i.e. whenStopping is executed before a restart and whenStarting afterwards). If that is not desired, use the following two hooks:

```
val a = actor(new Act {
  become {
    case "die" ⇒ throw new Exception
  }
  whenFailing { case m @ (cause, msg) ⇒ testActor ! m }
  whenRestarted { cause ⇒ testActor ! cause }
})
```

Nested Actors

It is also possible to create nested actors, i.e. grand-children, like this:

```
// here we pass in the ActorRefFactory explicitly as an example
val a = actor(system, "fred")(new Act {
  val b = actor("barney")(new Act {
    whenStarting { context.parent ! ("hello from " + self.path) }
  })
  become {
    case x ⇒ testActor ! x
  }
})
```

Supervision

It is also possible to assign a supervision strategy to these actors with the following:

```
superviseWith(OneForOneStrategy() {
  case e: Exception if e.getMessage == "hello" ⇒ Stop
  case _: Exception                            ⇒ Resume
})
```

Stash support

Last but not least there is a little bit of convenience magic built-in, which detects if the runtime class of the statically given actor subtype extends the RequiresMessageQueue trait via the Stash trait (this is a complicated way of saying that new Act with Stash would not work because its runtime erased type is just an anonymous subtype of Act). The purpose is to automatically use the appropriate deque-based mailbox type required by Stash. If you want to use this magic, simply extend ActWithStash:

```
val a = actor(new ActWithStash {
  become {
    case 1 ⇒ stash()
    case 2 ⇒
      testActor ! 2; unstashAll(); becomeStacked {
        case 1 ⇒ testActor ! 1; unbecome()
      }
  }
})
```

Read Actor DSL online: https://riptutorial.com/akka/topic/2392/actor-dsl

Chapter 3: Akka HTTP

Introduction

Akka HTTP is a light-weight HTTP server and client library, using akka-streams under the hood

Examples

Akka HTTP server: Hello World (Scala DSL)

The following app will start an HTTP server listening on port 8080 that returns `Hello world` on `GET /hello/world`

```
import akka.actor.ActorSystem
import akka.http.scaladsl.Http
import akka.http.scaladsl.server.Directives._
import akka.http.scaladsl.server._
import akka.stream.ActorMaterializer

import scala.concurrent.Await
import scala.concurrent.duration.Duration

object HelloWorld extends App {

  implicit val system = ActorSystem("ProxySystem")
  implicit val mat = ActorMaterializer()

  val route: Route = get {
    path("hello" / "world") {
      complete("Hello world")
    }
  }

  val bindingFuture = Http().bindAndHandle(Route.handlerFlow(route), "127.0.0.1", port = 8080)

  Await.result(system.whenTerminated, Duration.Inf)

}
```

Read Akka HTTP online: https://riptutorial.com/akka/topic/10108/akka-http

Chapter 4: Akka Streams

Examples

Akka Streams: Hello World

Akka Streams allows you to easily create a stream leveraging the power of the Akka framework without explicitly defining actor behaviors and messages. Every stream will have at least one `Source` (origin of the data) and at least one `Sink` (destination of the data).

```
import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.stream.scaladsl.{Sink, Source}
import java.io.File

val stream = Source(Seq("test1.txt", "test2.txt", "test3.txt"))
  .map(new File(_))
  .filter(_.exists())
  .filter(_.length() != 0)
  .to(Sink.foreach(f => println(s"Absolute path: ${f.getAbsolutePath}")))
```

In this quick example we have a `Seq` of filenames that we input into the stream. First we map them to a `File`, then we filter out files which don't exist, then files which length is 0. If a file went through the filters, it gets printed into the `stdout`.

Akka streams also allows you to do streams in a modular way. You can create `Flow`s with the partial modules of your stream. If we take the same example we could also do:

```
import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.stream.scaladsl.{Sink, Source}
import java.io.File

implicit val actorSystem = ActorSystem("system")
implicit val actorMaterializer = ActorMaterializer()

val source = Source(List("test1.txt", "test2.txt", "test3.txt"))
val mapper = Flow[String].map(new File(_))
val existsFilter = Flow[File].filter(_.exists())
val lengthZeroFilter = Flow[File].filter(_.length() != 0)
val sink = Sink.foreach[File](f => println(s"Absolute path: ${f.getAbsolutePath}"))

val stream = source
  .via(mapper)
  .via(existsFilter)
  .via(lengthZeroFilter)
  .to(sink)

stream.run()
```

In this second version we can see that `mapper`, `existsFilter`, `lengthZeroFilter` are `Flow`s. You can compose them in stream by using the method `via`. This capability would allow you to reuse your

pieces of code. One important thing to mention is that `Flow`s can be stateless or stateful. In the case of stateful, you need to be careful when reusing them.

You can also think about streams as `Graphs`. Akka Streams also provides a powerful `GraphDSL` to define complicated streams in a simple way. Following with the same example we could do:

```
import java.io.File
import akka.actor.ActorSystem
import akka.stream.{ActorMaterializer, ClosedShape}
import akka.stream.scaladsl.{Flow, GraphDSL, RunnableGraph, Sink, Source}

implicit val actorSystem = ActorSystem("system")
implicit val actorMaterializer = ActorMaterializer()

val graph = RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val source = Source(List("test1.txt", "test2.txt", "test3.txt"))
  val mapper = Flow[String].map(new File(_))
  val existsFilter = Flow[File].filter(_.exists())
  val lengthZeroFilter = Flow[File].filter(_.length() != 0)
  val sink = Sink.foreach[File](f => println(s"Absolute path: ${f.getAbsolutePath}"))

  source ~> mapper ~> existsFilter ~> lengthZeroFilter ~> sink

  ClosedShape
})

graph.run()
```

It is also possible to create aggregated flow using the `GraphDSL`. For example, if we would like to combine the mapper and two filters in one we could do:

```
val combinedFlow = Flow.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._

  val mapper = builder.add(Flow[String].map(new File(_)))
  val existsFilter = builder.add(Flow[File].filter(_.exists()))
  val lengthZeroFilter = builder.add(Flow[File].filter(_.length() != 0))

  mapper ~> existsFilter ~> lengthZeroFilter

  FlowShape(mapper.in, lengthZeroFilter.out)
})
```

And then use it as a individual block. `combinedFlow` would be a `FlowShape` or a `PartialGraph`. We can us for example with `via`:

```
val stream = source
  .via(combinedFlow)
  .to(sink)

stream.run()
```

Or using the `GraphDSL`:

```
val graph = RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val source = Source(List("test1.txt", "test2.txt", "test3.txt"))
  val sink = Sink.foreach[File](f => println(s"Absolute path: ${f.getAbsolutePath}"))

  source ~> combinedFlow ~> sink

  ClosedShape
})

graph.run()
```

Akka-Streams: subflows

You can dynamically fork a flow in multiple subflows using `groupBy`. The continuing stages are applied to each subflow until you merge them back using `mergeSubstreams`.

```
val sumByKey: Flow[(String, Int), Int, NotUsed] =
  Flow[(String, Int)].
    groupBy(Int.maxValue, _._1).  //forks the flow
    map(_._2).                    //this is applied to each subflow
    fold(0)(_ + _).
    mergeSubstreams               //the subflow outputs are merged back together
```

Read Akka Streams online: https://riptutorial.com/akka/topic/7394/akka-streams

Chapter 5: akka-streams custom shapes

Remarks

akka provides some pre-defined shapes, that should probably fit 99.9% of your usage. creating a new shape should only be done in some very rare cases. the pre-defined shapes are:

- `Source` - 1 outlet, no inlets
- `Sink` - 1 inlet, no outlets
- `Flow` - 1 inlet, 1 outlet
- `BidiFlow` - 2 inlets, 2 outlets
- `Closed` - no inlets, no outlets
- `FanInN` - `N` inlets (`N` <= 22), 1 outlet
- `FanOutN` - `N` outlets (`N` <= 22), 1 inlet
- `UniformFanIn` - any number of inlets of the same type, 1 outlet
- `UniformFanOut` - any number of outlets of the same type, 1 inlet
- `Amorphous` - any number of inlets or outlets, but untyped.

Examples

TwoThreeShape

a simple example of how to define a custom shape with 2 inlets and 3 outlets.

```
case class TwoThreeShape[-In1, -In2, +Out1, +Out2, +Out3](
          in1: Inlet[In1@uncheckedVariance],
          in2: Inlet[In2@uncheckedVariance],
          out1: Outlet[Out1@uncheckedVariance],
          out2: Outlet[Out2@uncheckedVariance],
          out3: Outlet[Out3@uncheckedVariance]) extends Shape {

  override val inlets: immutable.Seq[Inlet[_]] = List(in1, in2)
  override val outlets: immutable.Seq[Outlet[_]] = List(out1, out2, out3)

  override def deepCopy(): TwoThreeShape[In1, In2, Out1, Out2, Out3] =
    TwoThreeShape(in1.carbonCopy(),
                  in2.carbonCopy(),
                  out1.carbonCopy(),
                  out2.carbonCopy(),
                  out3.carbonCopy())

  override def copyFromPorts(inlets: immutable.Seq[Inlet[_]], outlets:
immutable.Seq[Outlet[_]]): Shape = {
    require(inlets.size == 2, s"proposed inlets [${inlets.mkString(", ")}] do not fit
TwoThreeShape")
    require(outlets.size == 3, s"proposed outlets [${outlets.mkString(", ")}] do not fit
TwoThreeShape")
    TwoThreeShape(inlets(0), inlets(1), outlets(0), outlets(1), outlets(2))
  }
}
```

an example usage for this weird shape: a stage that will pass through elements of 2 flows, while keeping a ratio of how many elements passed in the flows:

```
def ratioCount[X,Y]: Graph[TwoThreeShape[X,Y,X,Y,(Int,Int)],NotUsed] = {
  GraphDSL.create() { implicit b =>
    import GraphDSL.Implicits._

    val x = b.add(Broadcast[X](2))
    val y = b.add(Broadcast[Y](2))
    val z = b.add(Zip[Int,Int])

    x.out(1).conflateWithSeed(_ => 1)((count,_) => count + 1) ~> z.in0
    y.out(1).conflateWithSeed(_ => 1)((count,_) => count + 1) ~> z.in1

    TwoThreeShape(x.in,y.in,x.out(0),y.out(0),z.out)
  }
}
```

Read akka-streams custom shapes online: https://riptutorial.com/akka/topic/3282/akka-streams-custom-shapes

Chapter 6: Dispatchers

Examples

Default Dispatcher

An Akka MessageDispatcher is what makes Akka Actors "tick", it is the engine of the machine so to speak. All MessageDispatcher implementations are also an ExecutionContext, which means that they can be used to execute arbitrary code, for instance Futures.

Every ActorSystem will have a default dispatcher that will be used in case nothing else is configured for an Actor. The default dispatcher can be configured, and is by default a `Dispatcher` with the specified `default-executor`. If an ActorSystem is created with an `ExecutionContext` passed in, this ExecutionContext will be used as the default executor for all dispatchers in this ActorSystem. If no ExecutionContext is given, it will fallback to the executor specified in `akka.actor.default-dispatcher.default-executor.fallback`. By default this is a `fork-join-executor`, which gives excellent performance in most cases.

Setting the dispatcher for an Actor

So in case you want to give your Actor a different dispatcher than the default, you need to do two things, of which the first is to configure the dispatcher in your `application.conf`:

```
my-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "fork-join-executor"
  # Configuration for the fork join pool
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 2
    # Parallelism (threads) ... ceil(available processors * factor)
    parallelism-factor = 2.0
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

And here's another example that uses the "thread-pool-executor":

```
my-thread-pool-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "thread-pool-executor"
```

```
  # Configuration for the thread pool
  thread-pool-executor {
    # minimum number of threads to cap factor-based core number to
    core-pool-size-min = 2
    # No of core threads ... ceil(available processors * factor)
    core-pool-size-factor = 2.0
    # maximum number of threads to cap factor-based number to
    core-pool-size-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

You can then define the dispatcher to use for your actor inside you config, e.g.

```
akka.actor.deployment {
  /myactor {
    dispatcher = my-dispatcher
  }
}
```

and create this actor with the name specified in the config:

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor], "myactor")
```

Or you can lookup your dispatcher with:

```
import akka.actor.Props
val myActor =
  context.actorOf(Props[MyActor].withDispatcher("my-dispatcher"), "myactor1")
```

Read Dispatchers online: https://riptutorial.com/akka/topic/3228/dispatchers

Chapter 7: Hello world

Examples

Akka hello world (Scala)

1. Add akka-actor dependency (SBT example)

```
libraryDependencies += "com.typesafe.akka" % "akka-actor_2.11" % "2.4.8"
```

2. Create actor classes:

Actor for string output:

```
class OutputActor extends Actor {
  override def receive: Receive = {
    case message => println(message)
  }
}
```

Actor for string modifying:

```
class AppendActor(outputActor: ActorRef) extends Actor {
  override def receive: Receive = {
    case message: String =>
      val changed = s"Hello, $message!"
      outputActor ! changed

    case unknown =>
      println(s"unknown message: $unknown")
  }
}
```

3. Create actor systems and send message

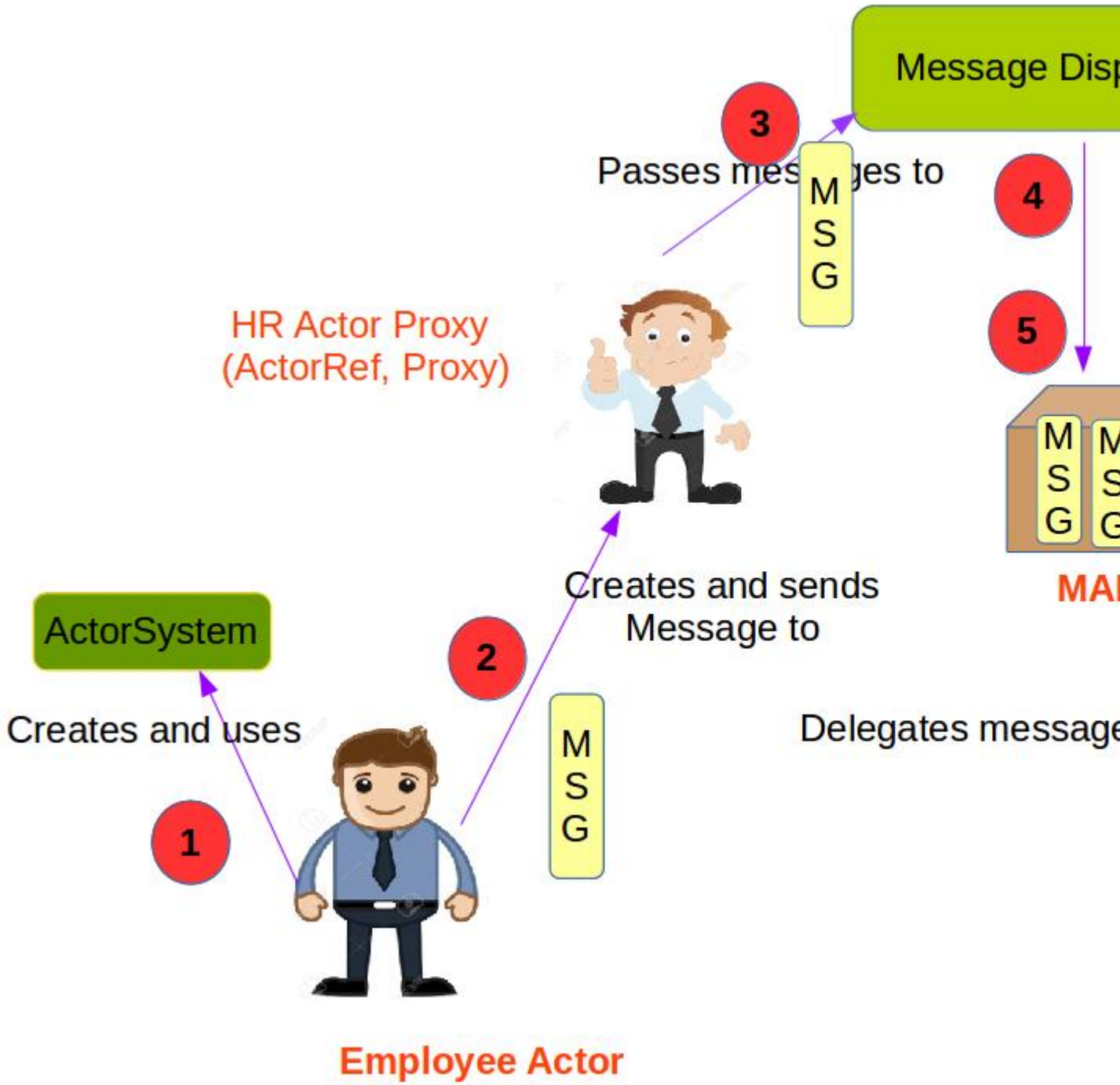
```
object HelloWorld extends App {
  val system = ActorSystem("HelloWorld")
  val outputActor = system.actorOf(Props[OutputActor], name = "output")
  val appendActor = system.actorOf(Props(classOf[AppendActor], outputActor), name =
"appender")

  appendActor ! "Akka" // send test message
  Thread.sleep(500) // wait for async evaluation
  system.terminate() // terminate actors system
}
```

Program output:

    Hello, Akka!

Simple Actor Implementation

Consider a communication happening between a Employee and its HR Department.



Broadly these are explained in the following six steps when a message is passed to the actor:

1. Employee creates something called an `ActorSystem`.
2. It uses the ActorSystem to create something called as `ActorRef`. The message(MSG) is sent

to the `ActorRef` (a proxy to HR Actor).

3. Actor ref passes the message along to a `Message Dispatcher`.
4. The Dispatcher enqueues the message in the target Actor's `MailBox`.
5. The Dispatcher then puts the `Mailbox` on a Thread (more on that in the next section).
6. The `MailBox` dequeues a message and eventually delegates that to the actual HR Actor's `receive` method.

```
    /** The Main Program consider it as a Employee Actor that is sending the requests  **/

    object EmployeeActorApp extends App{
     //Initialize the ActorSystem
      val actorSystem=ActorSystem("HrMessageingSystem")

     //construct the HR Actor Ref
      val hrActorRef=actorSystem.actorOf(Props[HrActor])

     //send a message to the HR Actor
      hrActorRef!Message

     //Let's wait for a couple of seconds before we shut down the    system
      Thread.sleep (2000)

     //Shut down the ActorSystem.
      actorSystem.shutdown()

    }


    /** The HRActor reads the message sent to it and performs action based on the message Type
**/
    class HRActor extends Actor {
       def receive  = {
            case s: String if(s.equalsIgnoreCase("SICK")) => println("Sick Leave applied")
            case s: String if(s.equalsIgnoreCase("PTO")) => println("PTO applied ")
}
```

}

Akka Hello World (Java 8)

Add this dependency to your project POM:

```
<dependency>
        <groupId>com.typesafe.akka</groupId>
        <artifactId>akka-actor_2.11</artifactId>
        <version>2.4.4</version>
</dependency>
```

Create an Actor

```
public class HelloWorldActor extends AbstractActor {
```

```
    public HelloActor() {
        receive(ReceiveBuilder
            .match(SayHello.class, this::sayHello)
            .match(SayBye.class, this::sayBye)
            .build());
    }

    private void sayHello(final SayHello message) {
        System.out.println("Hello World");
    }

    private void sayHello(final SayBye message) {
        System.out.println("Bye World");
    }

    public static Props props() {
        return Props.create(HelloWorldActor.class);
    }
}
```

Create a Junit test for the actor

```
public class HelloActorTest {

    private ActorSystem actorSystem;

    @org.junit.Before
    public void setUp() throws Exception {
        actorSystem = ActorSystem.create();
    }

    @After
    public void tearDown() throws Exception {
        JavaTestKit.shutdownActorSystem(actorSystem);
    }

    @Test
    public void testSayHello() throws Exception {
        new JavaTestKit(actorSystem) {
            {
                ActorRef helloActorRef = actorSystem.actorOf(HelloWorldActor.props());
                helloActorRef.tell(new SayHello(), ActorRef.noSender());
                helloActorRef.tell(new SayBye(), ActorRef.noSender());
            }
        };
     }
 }
```

Read Hello world online: https://riptutorial.com/akka/topic/3283/hello-world

Chapter 8: Injecting dependencies into an actor

Examples

Spring-wired actor

Due to very specific way of actor instantiation, injecting dependencies into an actor instance is not trivial. In order to intervene in actor instantiation and allow Spring to inject dependencies one should implement a couple of akka extensions. First of those is an `IndirectActorProducer`:

```
import akka.actor.Actor;
import akka.actor.IndirectActorProducer;
import java.lang.reflect.Constructor;
import java.util.Arrays;
import org.springframework.beans.factory.config.AutowireCapableBeanFactory;
import org.springframework.context.ApplicationContext;

/**
 * An actor producer that lets Spring autowire dependencies into created actors.
 */
public class SpringWiredActorProducer implements IndirectActorProducer {

    private final ApplicationContext applicationContext;
    private final Class<? extends Actor> actorBeanClass;
    private final Object[] args;

    public SpringWiredActorProducer(ApplicationContext applicationContext, Class<? extends
Actor> actorBeanClass, Object... args) {
        this.applicationContext = applicationContext;
        this.actorBeanClass = actorBeanClass;
        this.args = args;
    }

    @Override
    public Actor produce() {
        Class[] argsTypes = new Class[args.length];
        for (int i = 0; i < args.length; i++) {
            if (args[i] == null) {
                argsTypes[i] = null;
            } else {
                argsTypes[i] = args[i].getClass();
            }
        }
        Actor result = null;
        try {
            if (args.length == 0) {
                result = (Actor) actorBeanClass.newInstance();
            } else {
                try {
                    result = (Actor)
actorBeanClass.getConstructor(argsTypes).newInstance(args);
                } catch (NoSuchMethodException ex) {
                    // if types of constructor don't match exactly, try to find appropriate
```

```
constructor
                    for (Constructor<?> c : actorBeanClass.getConstructors()) {
                        if (c.getParameterCount() == args.length) {
                            boolean match = true;
                            for (int i = 0; match && i < argsTypes.length; i++) {
                                if (argsTypes[i] != null) {
                                    match =
c.getParameters()[i].getType().isAssignableFrom(argsTypes[i]);
                                }
                            }
                            if (match) {
                                result = (Actor) c.newInstance(args);
                                break;
                            }
                        }
                    }
                }
            }
            if (result == null) {
                throw new RuntimeException(String.format("Cannot find appropriate constructor
for %s and types (%s)", actorBeanClass.getName(), Arrays.toString(argsTypes)));
            } else {

applicationContext.getAutowireCapableBeanFactory().autowireBeanProperties(result,
AutowireCapableBeanFactory.AUTOWIRE_BY_TYPE, true);
            }
        } catch (ReflectiveOperationException e) {
            throw new RuntimeException("Cannot instantiate an action of class " +
actorBeanClass.getName(), e);
        }
        return result;
    }

    @Override
    public Class<? extends Actor> actorClass() {
        return (Class<? extends Actor>) actorBeanClass;
    }

}
```

This producer instantiates an actor and injects dependencies before returning the actor instance.

We can prepare `Props` for creation an actor using the `SpringWiredActorProducer` the following way:

```
Props.create(SpringWiredActorProducer.class, applicationContext, actorBeanClass, args);
```

However it would be better to wrap that call into following spring bean:

```
import akka.actor.Extension;
import akka.actor.Props;
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;

/**
 * An Akka Extension to inject dependencies to {@link akka.actor.Actor}s with
 * Spring.
```

```
 */
@Component
public class SpringProps implements Extension, ApplicationContextAware {

    private volatile ApplicationContext applicationContext;

    /**
     * Creates a Props for the specified actorBeanName using the
     * {@link SpringWiredActorProducer}.
     *
     * @param actorBeanClass The class of the actor bean to create Props for
     * @param args arguments of the actor's constructor
     * @return a Props that will create the named actor bean using Spring
     */
    public Props create(Class actorBeanClass, Object... args) {
        return Props.create(SpringWiredActorProducer.class, applicationContext,
actorBeanClass, args);
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws
BeansException {
        this.applicationContext = applicationContext;
    }

}
```

You can autowire `SpringProps` anywhere an actor gets created (even in the actor itself) and create spring-wired actors the following way:

```
@Autowired
private SpringProps springProps;
//...
actorSystem.actorOf(springProps.create(ActorClass.class), actorName);
//or inside an actor
context().actorOf(springProps.create(ActorClass.class), actorName);
```

Assuming that `ActorClass` extends `UntypedActor` and has properties annotated with `@Autowired`, those dependencies will be injected right after instantiation.

Read Injecting dependencies into an actor online: https://riptutorial.com/akka/topic/4717/injecting-dependencies-into-an-actor

Chapter 9: Supervision and Monitoring in Akka

Remarks

References: akka.io/docs

Check out my blog: https://blog.knoldus.com/2016/08/07/supervision-and-monitoring-in-akka/

Examples

What is supervision?

Describes a dependency relationship between actors, the parent and child releationship. Parent is unique because it has created the child actor, so the parent is responsible for reacting when failures happens in his child.

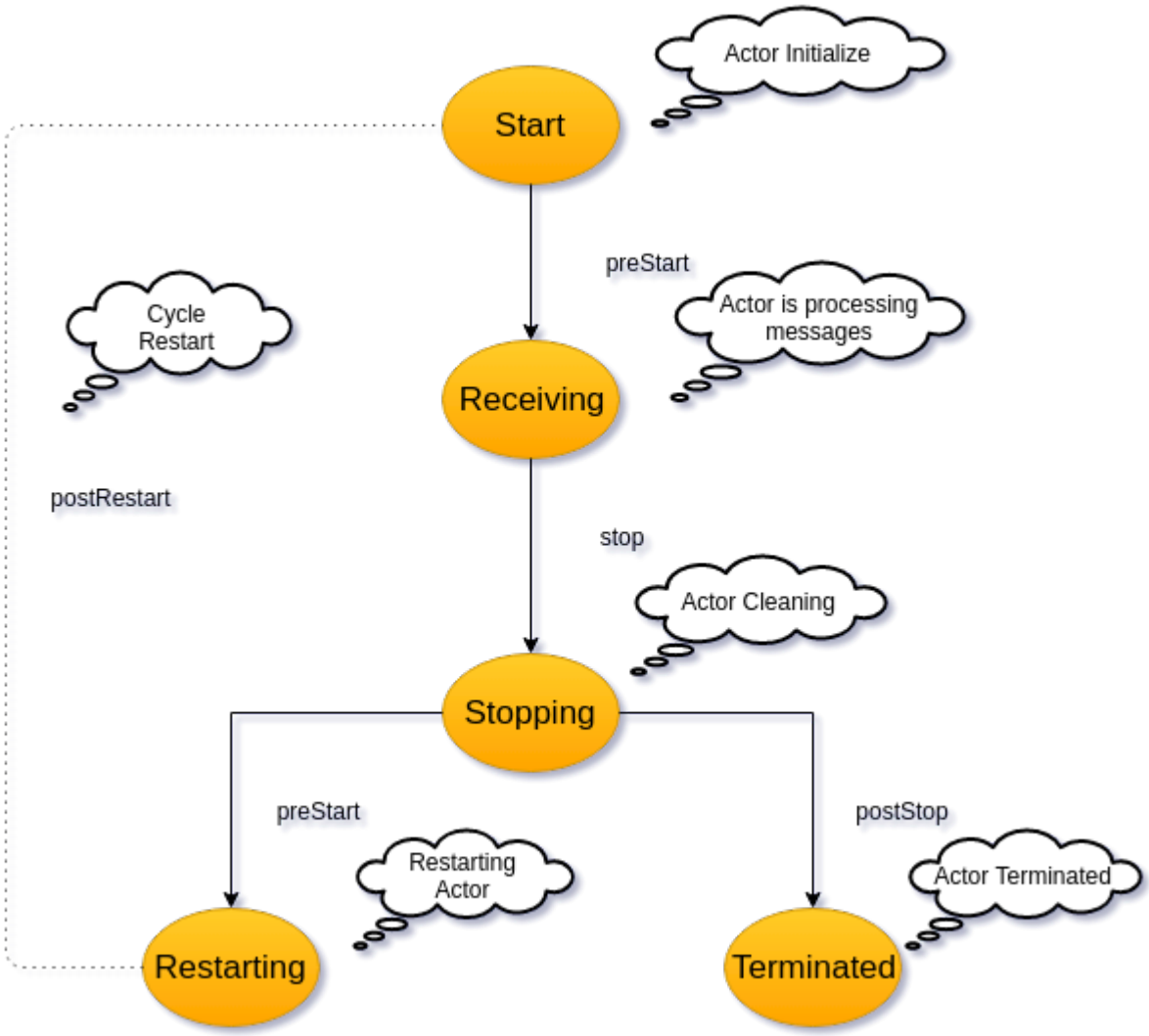
And parent decides which choice needs to be selected. When a parent receives the failure signal from it's child then depending on the nature of failure, the parent decides from following options:

Resume: Parent starts the child actor keeping its internal state.

Restart: Parent starts the child actor by clearing it's internal state.

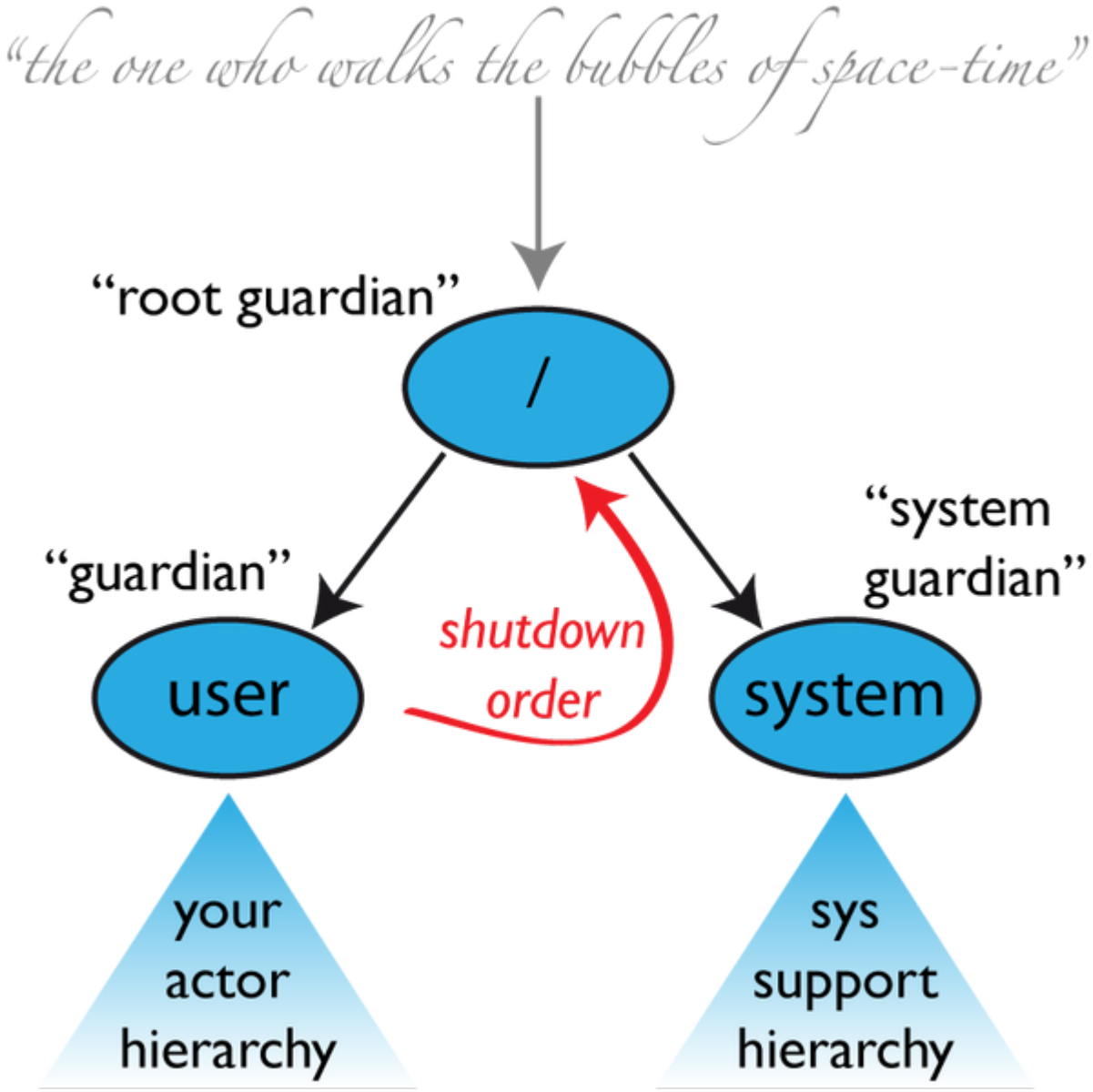
Stop: Stop the child permanently.

Escalate: Escalate the failure by failing itself and propagate failure to its parent.



Akka Life Cycle

It is always important to view a part of supervision hierarchy, which explains the escalate option. Each supervisor should cover with all possible failure cases.



Actor System: Source: doc.akka.io

/user: **The User Guardian Actor**

Actor created using system.actorOf() are children of user guardian actor. Whenever user guardian terminates, all user created actors will be terminated too. Top level user created actors are determined by user guardian actor that how they will be supervised. Root Guardian is the supervisor of user guardian.

/root: **The Root Guardian**

The root guardian actor is the father of all actor system. It supervises user guardian actor and system guardian actor.

Supervision Strategies

There are two type of supervision strategies that we follow to supervise any actor:

1. One-For-One Strategy

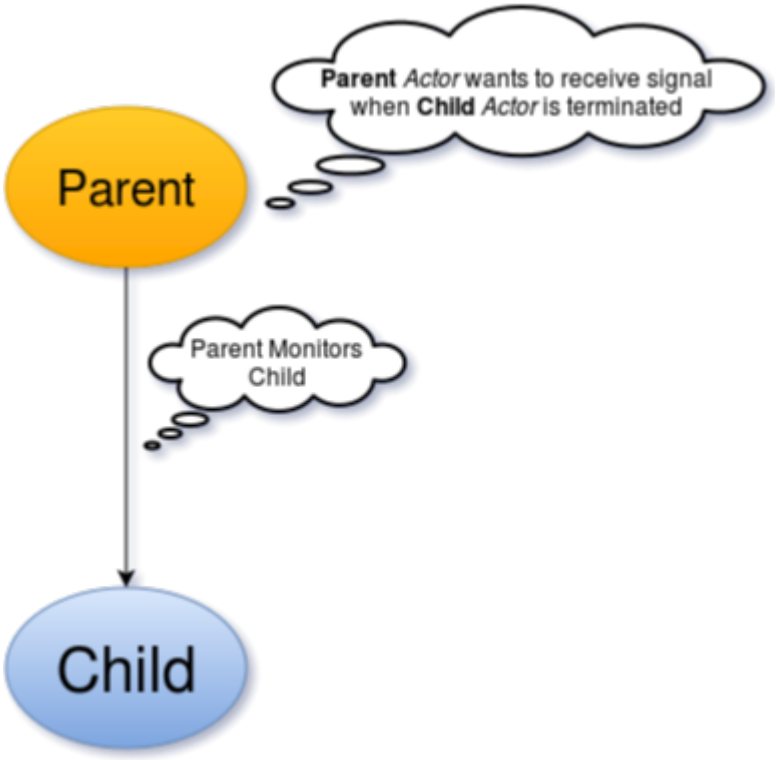
2. One-For-All Strategy

```
case object ResumeException extends Exception

case object StopException extends Exception
case object RestartException extends Exception

override val supervisorStrategy =
 OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 second){
 case ResumeException => Resume
 case RestartException => Restart
 case StopException => Stop
 case _: Exception => Escalate
 }
```

What is Monitoring?

Lifecycle Monitoring in Akka is usually referred to as DeathWatch.

Monitoring is thus used to tie one actor to another so that it may react to the other actor's termination, in contrast to supervision which reacts to failure.



Monitoring

Monitoring is particularly useful if a supervisor cannot simply restart its children and has to terminate them, e.g. in case of errors during actor initialization. In that case it should monitor those children and re-create them or schedule itself to retry this at a later time.

Code Repository

Supervision and Monitoring

Read Supervision and Monitoring in Akka online:
https://riptutorial.com/akka/topic/7831/supervision-and-monitoring-in-akka

Credits

S. No	Chapters	Contributors
1	Getting started with akka	12Sandy, Community, noelyahan
2	Actor DSL	Martin Seeler
3	Akka HTTP	Cyrille Corpet, Konrad 'ktoso' Malawski
4	Akka Streams	Cyrille Corpet, hveiga
5	akka-streams custom shapes	gilad hoch
6	Dispatchers	Martin Seeler
7	Hello world	12Sandy, Cortwave, Fabien Benoit-Koch, Konrad 'ktoso' Malawski, tmbo
8	Injecting dependencies into an actor	Oleg Kurbatov
9	Supervision and Monitoring in Akka	Prabhat Kashyap