I have seen the top of Akka Mountain, and it is good!

In One Sentence....

“To help the audience understand Actors, Supervision, Futures, Routers in both Scala and Java”

A note about my presentation style

- Trying to achieve the right mix of demonstration and slides.
- Code for this presentation is available on GitHub:

  `http://github.com/dhinojosa/akka-study.git`

- Goal is to provide you with code that you can use and reference long after the presentation.
- Presentation slides are also updated as a pdf on the repository.

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?

- Imagine life asynchronously
- Actors manage their own state, no leaks
- Easier thread management
- Web sites potentially can become faster because they delegated tasks away
- e.g. What if you need something persisted later, don't necessarily need it right now?
- e.g. What if emails can be sent later on?

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

Actors (The basics)

- Create a message
- Send a message to an actor
- Each message has to be immutable
- Each message should not be a closure.

What does immutable look like? (Java)

```
public class Person {
    private String firstName;
    private 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 boolean equals(Object o) {...}

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

What does immutable look like? (Scala)

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

What does closure look like? (Scala)

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

Getting Started with Actors

Inside the Actor's Studio

Actor Visual

Actor Visual

Actor Visual

Actor Visual

Actor Visual

Actor Visual

Inside the Actor's Studio Again

Actor Visual

Actor Visual

Actor Visual

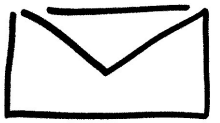
Actor Visual

Actor Visual

Actor Visual

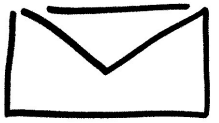
Actor Visual

Actor Visual



1

Actor Visual



2

Actor Visual

3

Actor Visual

How hard is it to set up these actors?

Untyped Actor in Java

```
public class SimpleActorJava extends UntypedActor {
    LoggingAdapter log = Logging.getLogger
                         (getContext().system(), this);

    public void onReceive(Object message) {
        if (message instanceof String)
            log.info(
                "Received String message in SimpleActorJava {}",
                 message);
        else
            unhandled(message);
    }
}
```

Untyped Actor in Scala

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

   def receive = {
     case "test" ⇒ log.info
                   ("received message test
                     in Simple Actor Scala")
     case _ ⇒ log.info("received unknown message
                        test in Simple Actor Scala")
   }
}
```

How do we run these actors?

Using Actors in Java

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef myActor = system.actorOf(
     new Props(SimpleActorJava.class), "simpleActorJava");
ActorRef deadLettersActor = system.actorFor("/deadLetters");
myActor.tell("Bueno!", deadLettersActor);
```

akka-study/src/test/java/akkastudy/simpleactor/java/SimpleActorTest.java

Using Actors in Scala

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[SimpleActorScala],
              name = "simpleActorScala")

myActor ! "Simple Test"
myActor ! "test"
```

akka-study/src/test/scala/akkastudy/simpleactor/scala/SimpleActorTest.scala

Actor System

- Houses Untyped and Typed Actors
- Actors are tasks
  - Typically split up and delegated
  - Minimized to be handled in one piece
- Root of, and supervisor of, all Actors
- Represents one logical application

Actor System in Java

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef myActor = system.actorOf(
     new Props(SimpleActorJava.class), "simpleActorJava");
ActorRef deadLettersActor = system.actorFor("/deadLetters");
myActor.tell("Bueno!", deadLettersActor);
```

akka-study/src/test/java/akkastudy/simpleactor/java/SimpleActorTest.java

Actor System in Scala

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[SimpleActorScala],
              name = "simpleActorScala")

myActor ! "Simple Test"
myActor ! "test"
```

akka-study/src/test/scala/akkastudy/simpleactor/scala/SimpleActorTest.scala

Props

- Immutable Configuration Class for each Actor
- Analogous to a recipe for Actor creation and look up
- No argument constructor and factory style creation available

Default Constructor Props in Java

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef myActor = system.actorOf(
     new Props(SimpleActorJava.class), "simpleActorJava");
ActorRef deadLettersActor = system.actorFor("/deadLetters");
myActor.tell("Bueno!", deadLettersActor);
```

Default Constructor Props in Scala

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[SimpleActorScala],
              name = "simpleActorScala")

myActor ! "Simple Test"
myActor ! "test"
```

Factory Props in Java

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef myActor = system.actorOf(new Props(
    new UntypedActorFactory() {
            @Override
            public Actor create() throws Exception {
                return new SimpleActorJava();
            }
    }), "simpleActorJava");
ActorRef deadLettersActor = system.actorFor("/deadLetters");
myActor.tell("Bueno!", deadLettersActor);
```

Factory Props in Scala

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props(new SimpleActorScala),
              name = "simpleActorScala")

myActor ! "Simple Test"
myActor ! "test"
```

actorOf()

- Creates an actor onto a `ActorSystem`
- Creates an actor given the a set of properties to describe the `Actor`
- Creates an `Actor` with an identifiable name, if provided.
- Returns an `ActorRef`

actorOf in Java

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef myActor = system.actorOf(
     new Props(SimpleActorJava.class), "simpleActorJava");
ActorRef deadLettersActor = system.actorFor("/deadLetters");
myActor.tell("Bueno!", deadLettersActor);
```

akka-study/src/test/java/akkastudy/simpleactor/java/SimpleActorTest.java

actorOf in Scala

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[SimpleActorScala],
              name = "simpleActorScala")

myActor ! "Simple Test"
myActor ! "test"
```

akka-study/src/test/scala/akkastudy/simpleactor/scala/SimpleActorTest.scala

Actor Refs

- Any subtype of `ActorRef`
- `ActorRef` is the only way to interact with an `Actor`
- Intent is to send messages to `Actor` that it represents, proxy.
- Each actor has reference to `self()` refers to it's own reference.
- Each actor also has reference to the `sender()`, the actor that sent the message.
- Any reference can be sent to another actor so that actor can send messages to it.

ActorRef in Java

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef myActor = system.actorOf(
     new Props(SimpleActorJava.class), "simpleActorJava");
ActorRef deadLettersActor = system.actorFor("/deadLetters");
myActor.tell("Bueno!", deadLettersActor);
```

akka-study/src/test/java/akkastudy/simpleactor/java/SimpleActorTest.java

ActorRef in Scala

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[SimpleActorScala],
              name = "simpleActorScala")

myActor ! "Simple Test"
myActor ! "test"
```

akka-study/src/test/scala/akkastudy/simpleactor/scala/SimpleActorTest.scala

TYPE IS INFERRED

Various Other Types Of References

- Pure Local Actor References
- Local Actor References
- Local Actor References with Routing
- Remote Actor References
- Promise Actor References
- Dead Letter Actor References
- Cluster Actor References

Sending to an Actor in Java

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef myActor = system.actorOf(
     new Props(SimpleActorJava.class), "simpleActorJava");
ActorRef deadLettersActor = system.actorFor("/deadLetters");
myActor.tell("Bueno!", deadLettersActor);
```

akka-study/src/test/java/akkastudy/simpleactor/java/SimpleActorTest.java

Sending to an Actor in Scala

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[SimpleActorScala],
              name = "simpleActorScala")

myActor ! "Simple Test"
myActor ! "test"
```

akka-study/src/test/scala/akkastudy/simpleactor/scala/SimpleActorTest.scala

Demo of Running an Actor

Actor Location

Paths

- Can be local or remote
- Can be absolute or relative

```
"akka://my-sys/user/service-a/worker1" // purely
local

"akka://my-sys@host.example.com:5678/user/service-b
" // local or remote

"cluster://my-cluster/service-c" // clustered
(Future Extension)
```

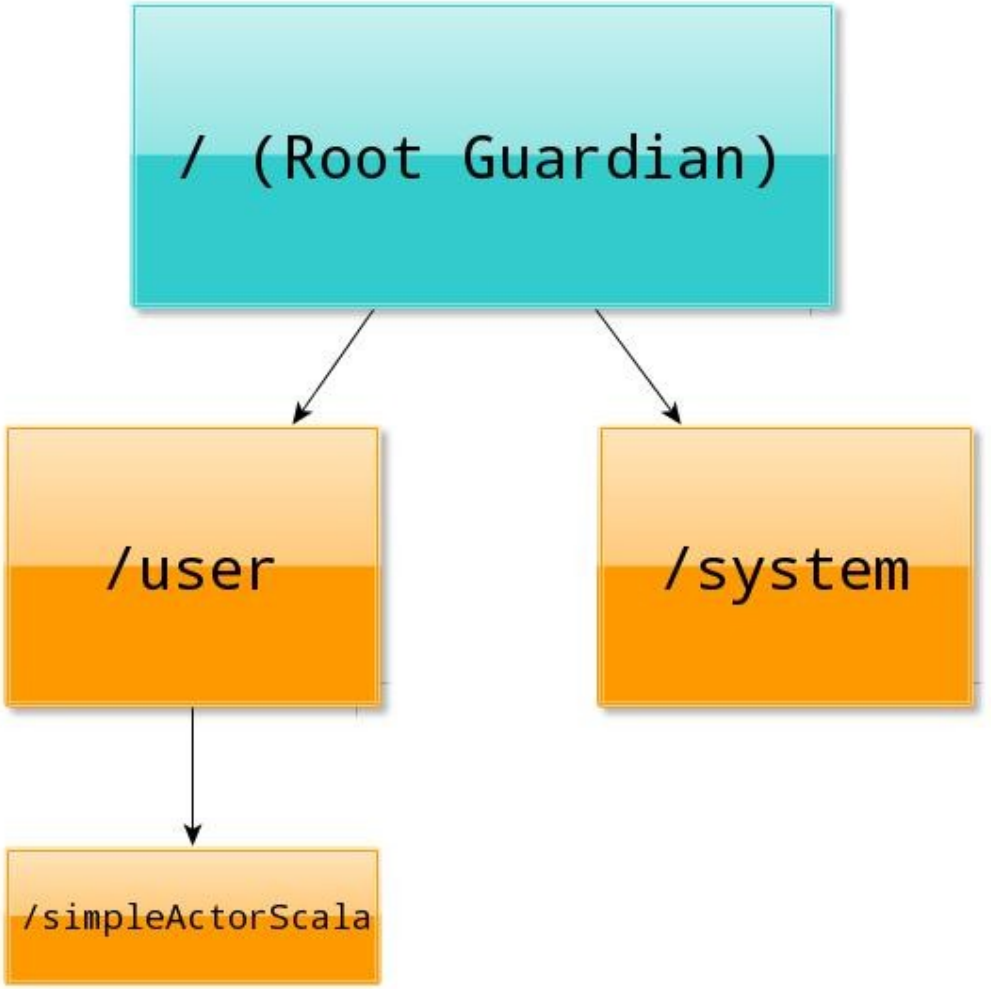
Location Transparency

- Vertical and Horizontal Growth
- Horizontal Growth driven by remoting systems
- Vertical Growth driven by routers
- All configuration based

Logical/Physical Paths

- A logical path is seen as if one element is a without recognized if one actor is remote
- A physical path is the direct path to the server and the direct location.

Anatomy of the Actor System



akka://{system-name}/user/{actor-name}

actorFor()

- Actor references can be looked up using `ActorSystem.actorFor` method.
- `ActorSystem.actorFor` can return a local or a remote reference.
- Reference can be used as long as the actor is alive.
- Only ever looks up an existing actor, i.e. does not create one.
- For Local Actor References
  - The actor must exist
  - If not, you will receive an `EmptyLocalActorRef`
- For Remote Actor References
  - A search by path on the remote system will occur.

actorFor()

```
val system = ActorSystem("MySystem")

system.actorOf(Props[SimpleActorScala],
name = "simpleActorJava")

val myActor = system.actorFor
("akka://MySystem/user/simpleActorJava")

myActor ! "Simple Test"
```

Typed Actor

- POJO and Actor Hybrids
- Consists of Two Parts, a public interface, and an implementation
- Cannot use `become`/`unbecome` to the type.
- Proxied
- Use Sparingly

Typed Actor in Java

```
public class Person {
    private String firstName;
    private String lastName;
    //bunch of junk
}


public interface RegistrationActor {
     public void registerPerson(Person person);
     public int getCount();
}
```

Typed Actor in Java

```
public class RegistrationActorImpl
      implements RegistrationActor, TypedActor.PreStart {

      private List<Person> people;

      @Override
      public void preStart() {
          this.people = new ArrayList<>();
      }

      @Override
      public void registerPerson(Person person) {
          this.people.add(person);
      }

      @Override
      public int getCount() {
          return people.size();
      }
}
```

Running the Typed Actor in Java

```
ActorSystem system = ActorSystem.create("MySystem");
RegistrationActor registrationActor =
      TypedActor.get(system).typedActorOf(
            new
      TypedProps<RegistrationActorImpl>(RegistrationActor.class,
RegistrationActorImpl.class),"registrationActor");
      registrationActor.registerPerson(new Person("Abraham",
"Lincoln"));
```

Typed Actor in Scala

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

trait RegistrationActor {
  def registerPerson(person: Person)

  def getCount: Int
}

class RegistrationActorImpl extends RegistrationActor {
  var list = List[Person]()

  def registerPerson(person: Person) {
    list = list :+ person
  }

  def getCount: Int = list.size
}
```

Running the Typed Actor in Scala

```
val system = ActorSystem("MySystem")
val registrationActor = TypedActor(system)
      .typedActorOf(
           TypedProps[RegistrationActorImpl],
               name = "registrationActor")
registrationActor.registerPerson(Person("Cesar", "Chavez"))
```

Demo of Typed Actors!

References and Lifecycles

self() , sender(), context

- `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.

Lifecycle Methods (as seen in Java)

```
public void preStart() {
}

public void preRestart(Throwable reason, scala.Option<Object>
message) {
  for (ActorRef each : getContext().getChildren()) {
    getContext().unwatch(each);
    getContext().stop(each);
  }
  postStop();
}

public void postRestart(Throwable reason) {
  preStart();
}

public void postStop() {
}
```

Lifecycle Methods (as seen in Scala)

```
def preStart(): Unit = ()

def postStop(): Unit = ()

def preRestart(reason: Throwable, message: Option[Any]): Unit = {
  context.children foreach { child ⇒
    context.unwatch(child)
    context.stop(child)
  }
  postStop()
}

def postRestart(reason: Throwable): Unit = {
  preStart()
}
```

Futures and Promises

About 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`

Executor Service

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

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;
}
```

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.
- The Fixed Thread Pool 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.”
- In a single thread executor, 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.

Futures in java.util.concurrent

```
ExecutorService executorService =
        Executors.newCachedThreadPool();
Callable<String> asynchronousTask = new Callable<String>()
{
    @Override
    public String call() throws Exception {
        //something expensive
        Thread.sleep(5000);
        return "Asynchronous String Result";
    }
};

Future<String> future =
     executorService.submit(asynchronousTask);

String value = future.get(); //waits if necessary
```

Callback Futures with Guava

```
ListenableFuture<String> listenableFuture =
service.submit(asynchronousTask);
Futures.addCallback(listenableFuture,
  new FutureCallback<String>() {
    @Override
    public void onSuccess(String result) {
        System.out.println("Got the result and the answer
is? " + result);
    }

    @Override
    public void onFailure(Throwable t) {
        System.out.println("Things happened man. Bad
things");
    }
});
```

Futures and Promises in Akka

- Data structure 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

Akka Futures in Java

```
ExecutionContext executionContext = ExecutionContext$.MODULE$.
                                    fromExecutorService
                                    (Executors.newFixedThreadPool(12));

Callable<String> callable = new Callable<String>() {
    @Override
    public String call() throws Exception {
        return "Test Basic Futures: Hello World";
    }
};
Future<String> future = Futures.future(callable, executionContext);
Timeout timeout = new Timeout(Duration.create(5, "seconds"));
System.out.println("Test Basic Futures: Step 1");
System.out.println(Await.result(future, timeout.duration())); //blocking
System.out.println("Test Basic Futures: Step 2");
```

Akka Non-Blocking Futures in Java

```
ExecutorService executorService = Executors.newFixedThreadPool(12);
ExecutionContext executionContext =
ExecutionContext$.MODULE$.fromExecutorService(executorService);

Callable<String> callable = new Callable<String>() {
    @Override
    public String call() throws Exception {
        return "Test Asynchronous Call: Hello World";
    }
};

Future<String> future = Futures.future(callable, executionContext);

System.out.println("Test Asynchronous Call: Step 1");
future.onComplete(new PrintResult<String>(), executionContext);
System.out.println("Test Asynchronous Call: Step 2");
System.out.println("Test Asynchronous Call: Step 3");
System.out.println("Test Asynchronous Call: Step 4");
```

OnComplete in Java

```
public final static class PrintResult<T> extends OnComplete<T> {
    @Override
    public void onComplete(Throwable failure, T success) throws Throwable {
        if (failure == null) System.out.println(success.toString());
        else failure.printStackTrace();
    }
}
```

Futures by themselves, blocking

```
implicit val executionContext =
ExecutionContext.fromExecutorService(
       Executors.newFixedThreadPool(12))

val future = Future {
   "Hello" + " " + "World"
}

implicit val timeout = Timeout(5 seconds)

println("Step 1")
val result = Await.result(future, timeout.duration) //blocking
println("Step 2: " + result)
```

Futures by themselves, nonblocking

```
implicit val executionContext =
ExecutionContext.fromExecutorService(
    Executors.newFixedThreadPool(12))

val future = Future { "Asynchronous String Result" }

future foreach (x => println("I got an answer:" + x))
//asynchonous
println("Doing something in the mean time")
```

Composing Futures Akka & Scala

```
implicit val executionContext =
ExecutionContext.fromExecutorService(
    Executors.newFixedThreadPool(12))

val future1 = Future {180/2}
val future2 = Future {90/3}
val result = future1.flatMap {x=>
   future2.map {y=>
     (x + y)
   }
}
println("Getting Ready to Run")
result foreach (x=> println("result: " + x))
println("Doing Something in the Meantime")
```

Demo: Futures & Asking in Scala

Configuration

HOCON

"Human-Optimized Config Object Notation"

HOCON

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

HOCON

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

HOCON

```
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`

Typical application.conf

```
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
    }
  }

  remote {
    log-sent-messages = on
    log-received-messages = on
  }
}
```

Creating a remote system

Adding custom configuration

```
remote-system {
  akka {
    daemonic = on
    actor {
       provider = "akka.remote.RemoteActorRefProvider"
    }
    remote {
       log-sent-messages = on
       log-received-messages = on

       enabled-transports = ["akka.remote.netty.tcp"]

       netty {
         hostname = "127.0.0.1"
         port = 10190
       }
    }
  }
}
```

Remote Actor Call in Java

```
Config config = ConfigFactory.load();
ActorSystem system = ActorSystem.create("MySystem",
   config.getConfig("remote-system").withFallback(config));

ActorRef myActor = system.actorOf(
   new Props(SimpleActorJava.class), "simpleActorJava");

ActorRef deadLettersActor = system.actorFor("/deadLetters");

myActor.tell("Bueno!", deadLettersActor);
```

Remote Actor Call in Java

```
Config config = ConfigFactory.load();
ActorSystem system = ActorSystem.create("MySystem",
   config.getConfig("remote-system").withFallback(config));

ActorRef myActor = system.actorOf(
   new Props(SimpleActorJava.class), "simpleActorJava");

ActorRef deadLettersActor = system.actorFor("/deadLetters");

myActor.tell("Bueno!", deadLettersActor);
```

Where is the config name?

```
remote-system {
  akka {
    daemonic = on
    actor {
       provider = "akka.remote.RemoteActorRefProvider"
    }
    remote {
       log-sent-messages = on
       log-received-messages = on

       enabled-transports = ["akka.remote.netty.tcp"]

       netty {
         hostname = "127.0.0.1"
         port = 10190
       }
    }
  }
}
```

Remote Actor Call in Scala

```
val config = ConfigFactory.load()
val system = ActorSystem("RemoteActorSystem",
      config.getConfig("remote-system").withFallback(config))
val myActor = system.actorOf(Props[SimpleActorScala],
      name = "simpleActorJava")
myActor ! "Simple Test"
```

Remote Actor Call in Scala

```
val config = ConfigFactory.load()
val system = ActorSystem("RemoteActorSystem",
      config.getConfig("remote-system").withFallback(config))
val myActor = system.actorOf(Props[SimpleActorScala],
      name = "simpleActorJava")
myActor ! "Simple Test"
```

Demo: Remote Actors

Fault Tolerance

Fault Tolerance

- Each actor
  - can be a parent, and create children
  - is responsible for their children (a.k.a. subordinate)
- When failure occurs
  - a child or all children are suspended
  - parent determines the next course of action
  - mailbox contents are maintained

One for One Strategy

- Each child is treated is separately
- Typically the normal one that should be used
- Default if no strategy is defined

All for One Strategy

- Each child will be given the same treatment
- If one fails, they all essentially "fail"
- Used if tight coupling between children is required

Default Fault Tolerance

- `ActorInitializationException` will stop the failing child actor
- `ActorKilledException` will stop the failing child actor
- `Exception` will restart the failing child actor
- Other types of `Throwable` will be escalated to parent actor

Demo: Fault Tolerance

Routers

Routers

- Actor that reroutes message to other actors
- Different Strategies available
- Create your own

Out of the box routers

- akka.routing.RoundRobinRouter
- akka.routing.RandomRouter
- akka.routing.SmallestMailboxRouter
- akka.routing.BroadcastRouter
- akka.routing.ScatterGatherFirstCompletedRouter
- akka.routing.ConsistentHashingRouter

Setup the application.conf

```
routing-system {
  akka.actor.deployment {
     /simplerouter {
        router = round-robin
        nr-of-instances = 5
     }
  }
```

Setting it up from code in Java

```
ActorRef actor1 =
system.actorOf(Props.create(ExampleActor.class));

ActorRef actor2 =
system.actorOf(Props.create(ExampleActor.class));

ActorRef actor3 =
system.actorOf(Props.create(ExampleActor.class));

Iterable<ActorRef> routees = Arrays.asList(new
ActorRef[] { actor1, actor2, actor3 });

ActorRef router2 =
system.actorOf(Props.empty().withRouter

        (RoundRobinRouter.create(routees)));
```

Setting it up from code in Scala

```
val actor1 = system.actorOf(Props[ExampleActor1])
val actor2 = system.actorOf(Props[ExampleActor1])
val actor3 = system.actorOf(Props[ExampleActor1])
val routees = Vector(actor1, actor2, actor3)
val router2 = system.actorOf(Props.empty.withRouter(
  RoundRobinRouter(routees = routees)))
```

Setting it up from code in Java

```
ActorRef actor1 =
system.actorOf(Props.create(ExampleActor.class));

ActorRef actor2 =
system.actorOf(Props.create(ExampleActor.class));

ActorRef actor3 =
system.actorOf(Props.create(ExampleActor.class));

Iterable<ActorRef> routees = Arrays.asList(new
ActorRef[] { actor1, actor2, actor3 });

ActorRef router2 =
system.actorOf(Props.empty().withRouter

        (RoundRobinRouter.create(routees)));
```

Demo: Routers

Demo: Becoming & Unbecoming

Thank You!