Introduction to Reactive Programming

Hands on with Akka and Java/Scala

Introduction

- **Duncan K. DeVore**
- --> SWE @ Typesafe
- --> Co-Author, Reactive Application Development, Manning
- **Henrik Engstrom**
- --> SWE @ Typesafe

Agenda Day One

- **What is Reactive?**
- **Message Driven**
- --> Actor Basics
- --> Actor Lifecycle
- **Testing Actors**
- --> Synchronous unit testing
- --> Asynchronous unit testing

Agenda Day Two

- **Resilience**
- --> Supervision
- --> Self Healing
- **Elasticity**
- --> Routers
- --> Dispatchers
- **Behavior**
- --> Become & Stash

What is Reactive?

Moth's and Programming

One of the fascinating things found in nature is the ability of a **species to adapt** to its changing environment. The canonical example of this is Britain's **Peppered Moth**.

When newly industrialized Great Britain became polluted in the nineteenth century, slow-growing, light-colored lichens that covered trees died and resulted in a blackening of the trees bark.

The impact of this was quite **profound**!

Moth's and Programming

- light-colored peppered moths, **camouflaged** and the **majority**
- found themselves the **obvious** target of many a hungry bird
- the rare, dark ones, now **blended** into the polluted ecosystem
- as the birds, **changed** from eating dark to light moths
- the dynamics of Britain’s moth population **changed**

Typesafe

Moth's and Programming

So what do moths have to do with programming?

The peppered moth was able to survive due to a mutation that allowed it to **react** to its changing environment. This ability to react *on-the-fly* is what a **Reactive Application**. In reactive terms:

- React to load
- React to failure
- React to users

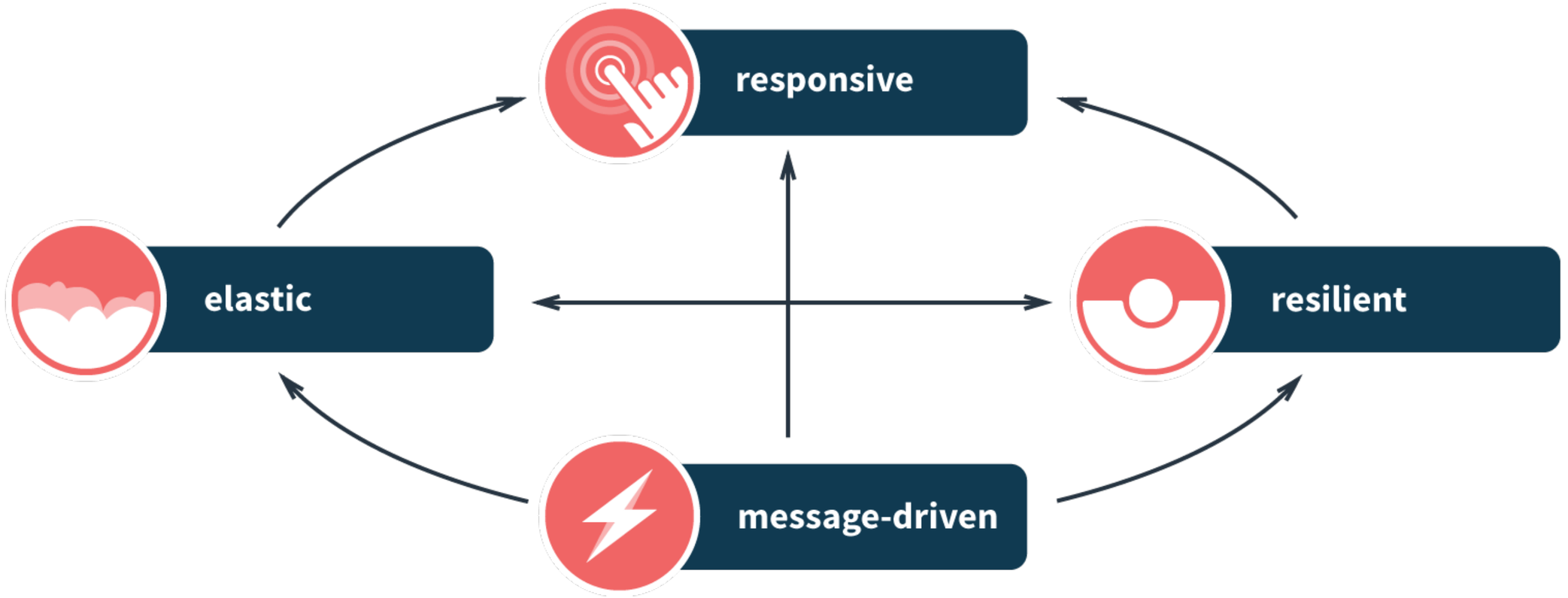
Typesafe

Reactive Manifesto

- http://www.reactivemanifesto.org/
- Published on September 16 2014. (v2.0)
- Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson
- 11K + Signatures

Typesafe

Reactive Principles



Typesafe

Message Driven

Reacting to messages: Based on asynchronous communication where the design of sender and recipient are *not affected* by the means of *message propagation*. As a result, you can design your system in isolation without worrying about how the transmission of messages occurs. Message-driven communication leads to a loosely coupled design that provides scalability, resilience and responsiveness.

Elasticity

Reacting to load: The system stays responsive under varying workload. Reactive applications can actively scale up and down or scale in and out based upon usage or other metrics utilized by system designers, saving money on unused computing power but most importantly ensuring the servicing of growing or spiking user base.

Resilience

Reacting to failure: The system stays responsive in the face of failure. Failure is expected and embraced and since many systems exist in isolation, a single point of failure remains just that. The system responds appropriately with strategies for restarting or re-provisioning, seamless to the overall systems.

Responsiveness

React to users: The system responds promptly if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively.

Why Reactive?

A Little History

- One of the greatest impacts in the last 50 years has been The **internet**
- The US commissioned research to build a **robust**, **fault-tolerant** computer network
- Began with a series of memos by J.C.R. Licklider of MIT in August 1962
- Became known as the **Galactic Network** concept.
- He envisioned a **globally** interconnected network of computers
- It would allow users to access data and programs from **anywhere** in the world.
- J.C.R. Licklider was the Director of the Information Processing Techniques Office
- (IPTO) was part of the Pentagon's ARPA, the Advanced Research Projects Agency
- Today, know as as DARPA, the Defense Advanced Research Projects Agency

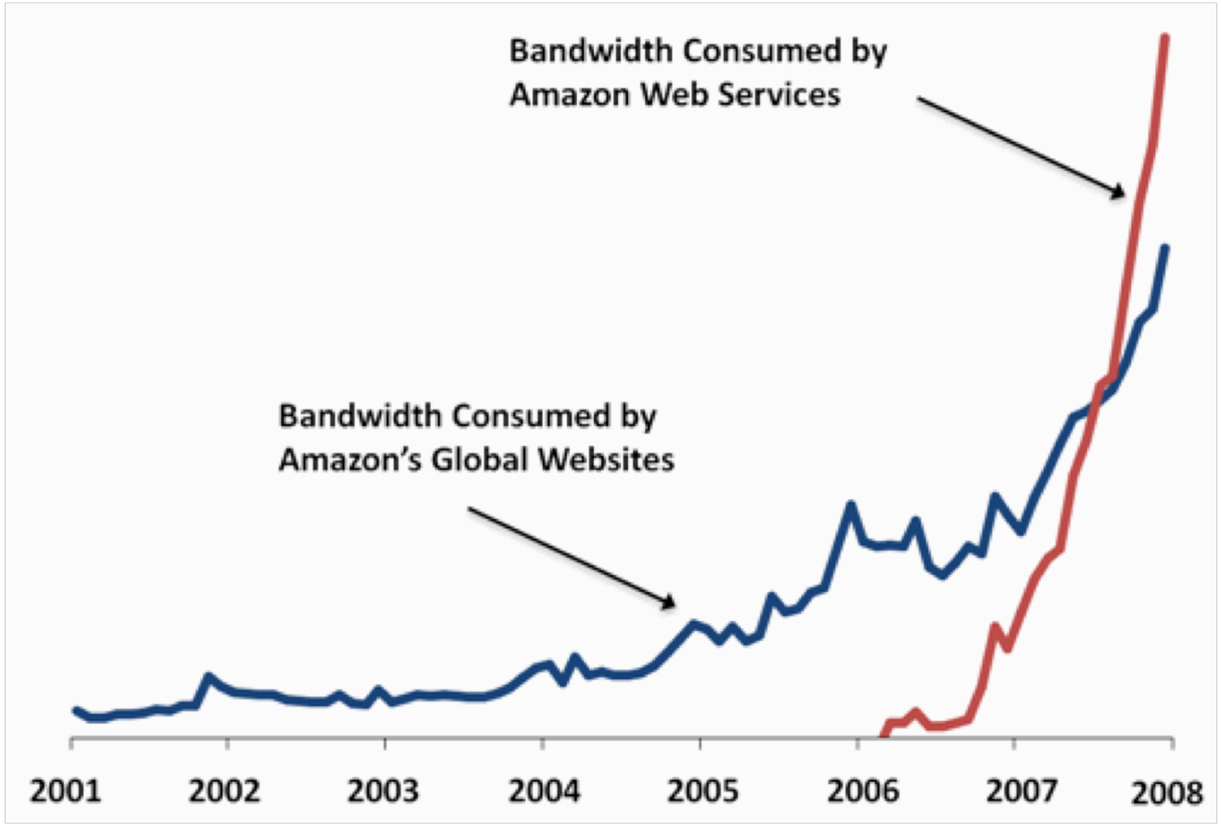
Distributed Computing

- A new computer model, **Distributed Systems** came into being
- It represented a shift in the computing paradigm.
- Before, the model was **large**, **expensive** mainframe systems
- Affectionately referred to as **Big Iron**.
- Mainframes used a **centralized** computing model
- Focusing on efficiency, local scalability, and reliability.

Cloud Computing

- Distributed gave way to what we know as **Cloud Computing**
- A more powerful **less expensive** computing solution
- Cloud computing represents another **paradigm** shift
- Changing the way we reason about computer applications
- Distributed systems focus on the **technical** details
- Cloud computing focuses on the **economics** side of the equation

Rethinking One's Value Proposition



As a result, many companies have begun to rethink their value proposition. Case and point:

- In January of 2008 Amazon announced that Amazon Web Services now **consume more bandwidth** than their entire global network of retail services, as shown in this figure from Amazon Blogs.[1]
- What is Amazon? An **online bookstore** or **provider of Cloud Services?**

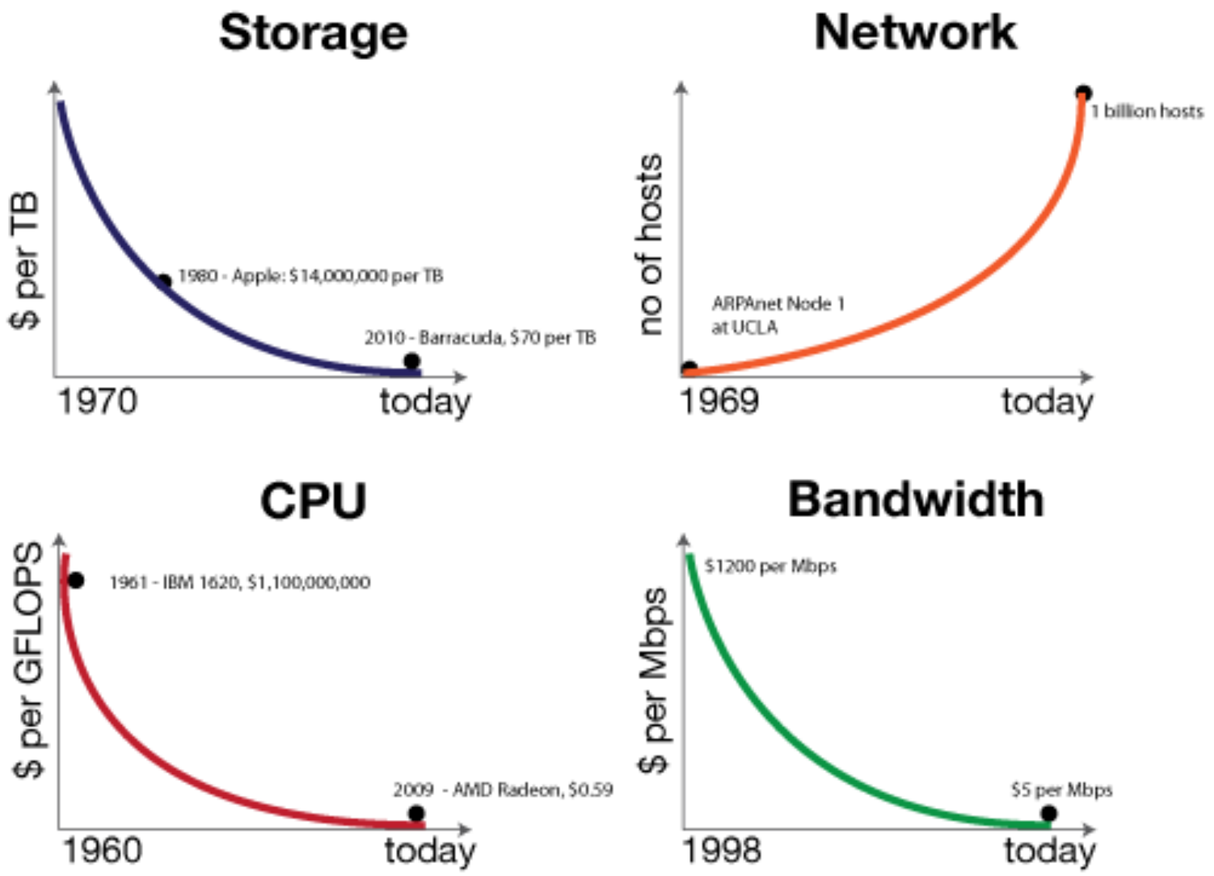
[1] image from Amazon Blogs - http://aws.amazon.com/blogs/aws/lots-of-bits/

Typesafe

Storage, Network, CPU and Bandwidth

This new landscape of distributed cloud computing represents a **dramatic change** for the modern programmer, much like the Industrial Revolution of the nineteenth century did for the Peppered moth.

Recent hardware enhancement such as multi-core CPU's and multi-socket servers provide computing capabilities that were non-existent as little as 8 years ago. The following shows the state of storage, CPU, and bandwidth compared to the number of network nodes. Notice the increase from the 70's![2]



[2] image from Oreilly Radar - http://radar.oreilly.com/2011/08/building-data-startups.html

Typesafe

The Fast Fish

"In the new world, it is not the big fish which eats the small fish, it’s the fast fish which eats the slow fish." --Klaus Schwab

Typesafe

What is Akka?

"Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM." --Akka.io

Akka is Reactive

- **Message Driven**: System foundation for
- --> elastic, resilient responsiveness
- **Elastic**: System stays responsive under varying workload
- **Resilient**: System stays responsive in the face of failure
- **Responsive**: System responds in a timely manner

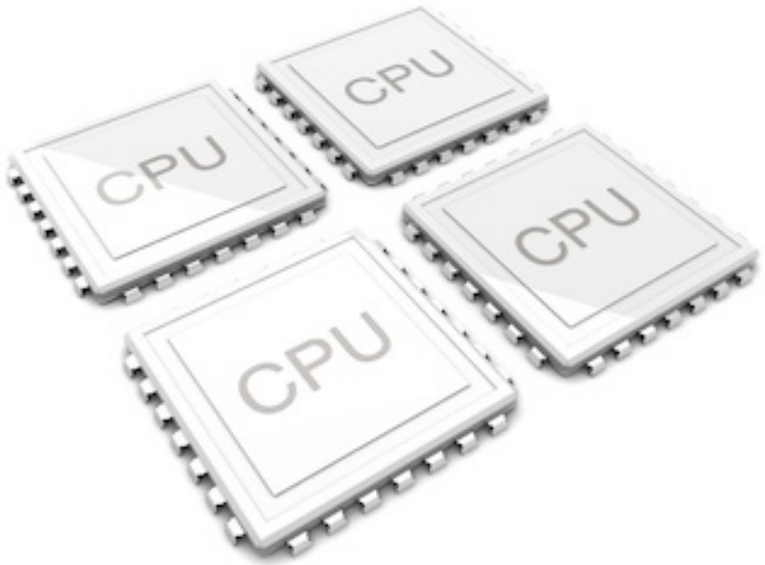
Akka's Value Proposition

- **Single Unified Programming Model**
- --> simpler concurrency
- --> simpler distribution
- --> simpler fault tolerance

Typesafe

Simpler Concurrency

- **single-threaded** illusion
- No **locks** needed
- No **synchronized** methods
- No **primitives** needed



Typesafe

Simpler Distribution

- **Distributed** by default
- Local to **Remote** by optimization
- Up **==** Out

Typesafe

Simpler Fault-Tolerance

- Akka decouples communication from failure handling:
- --> Supervisors handle failure
- --> Callers need not care

Typesafe

Case Study

Typesafe

The Reactive Restaurant

We are going to build the **Reactive Restaurant**. A fine dinning underwater sea food restaurant specializing in tantalizing dishes:

- **Our Menu Contains**
- --> Akkacore
- --> MahiPlay
- --> KingScala
- **There will be Challenges**
- --> Waiters will get frustrated
- --> Cooks will become bottlenecks
- --> Customer will run out of money

Group Exercise Part One

- Look at project structure and prepared code
- Run sbt from the `reactive-restaurant` directory
- From the base project do the following:
- --> Run the `man` command
- --> Read and follow the instructions
- --> Run the `next` command to advance to `common`

Typesafe

Group Exercise Part Two

- From the `common` project do the following:
- `-->` Run the `man` command
- `-->` Read and follow the instructions
- `-->` Run the `next` command to advance to `entry_state`

Typesafe

Group Exercise Part Three

- From the `entry_state` project do the following:
- --> Run the `man` command
- --> Read and follow the instructions
- --> Run the `next` command to advance to `001_implement_actor`
- --> At this point we are ready to start coding!
- --> Before we get our hands dirty, let's first explore some actor basics.

Message Driven

The Actor Model

"The actor is the fundamental unit of computation embodying processing, storage and communication." --Carl Hewitt

- Invented 1973 by Carl Hewitt
- **Akka**
- --> Processing = behavior
- --> Storage = state

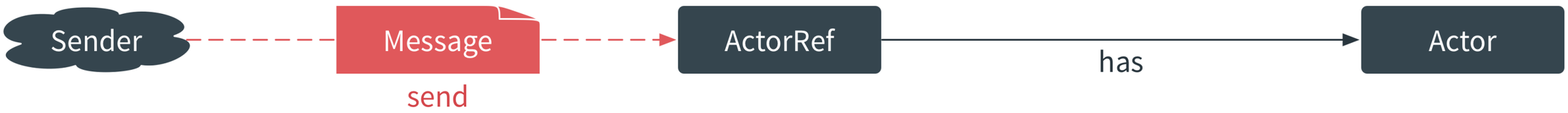
Typesafe

Fundamental Concepts of the Actor Model

- Everything is an actor
- Each actor has an address
- **Actors can**
- --> create new actors
- --> send messages to other actors
- --> change the behavior for handling the next message
- --> ...

Typesafe

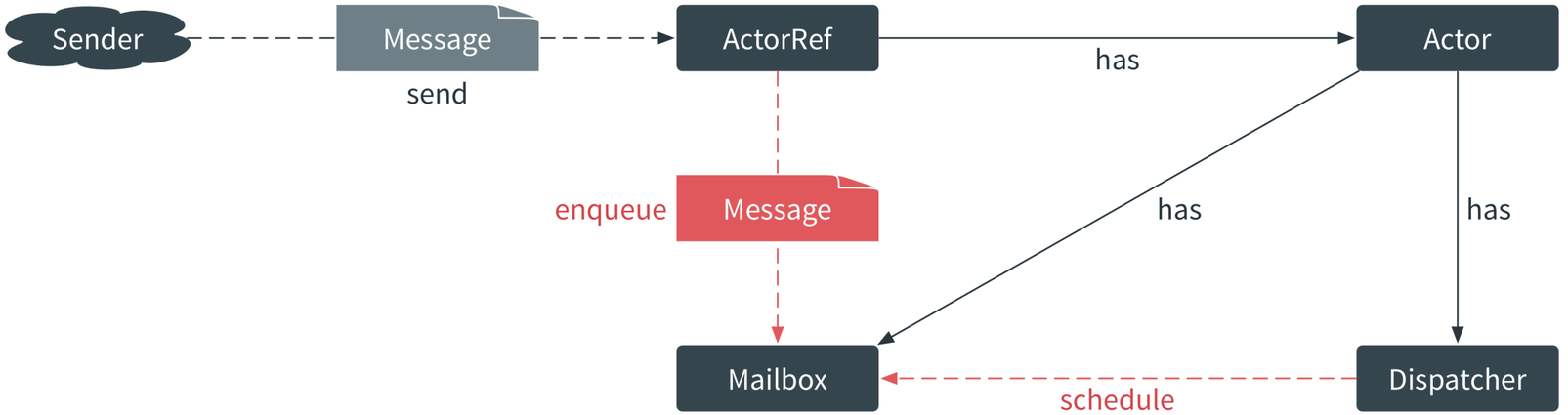
Anotomy of An Actor I



- Each actor is represented by an `ActorRef`
- You never get access to an `Actor` instance
- An actor reference lets you send messages to the actor

Typesafe

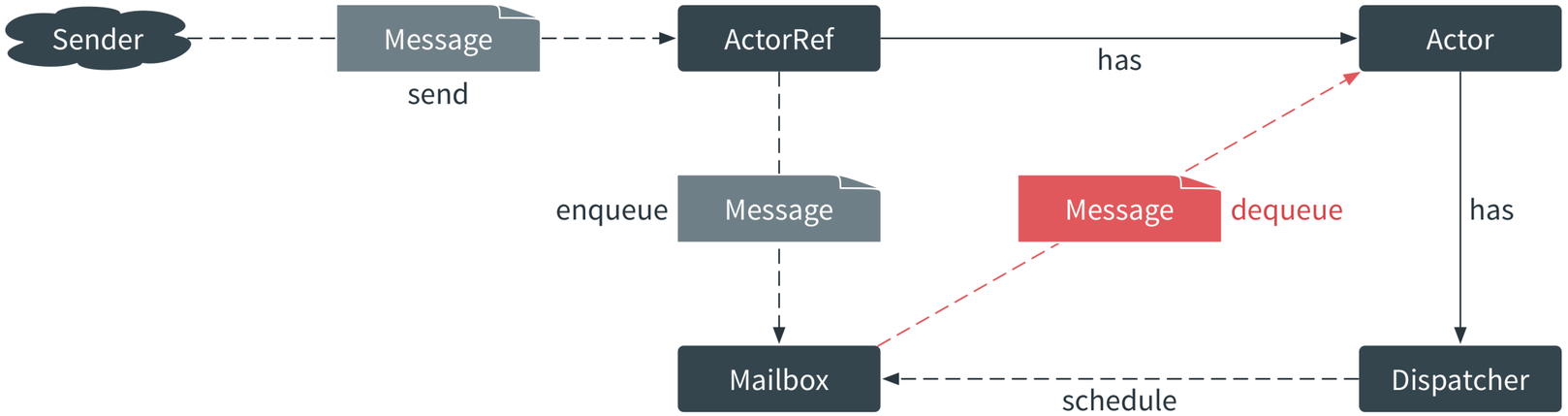
Anatomy of an Actor II



- Each actor has a `mailbox` and a `dispatcher`
- The `dispatcher` enqueues and schedules message delivery

Typesafe

Anatomy of an Actor III



- Only one message at a time is passed to the actor
- Delivery/processing - separate activities may be different threads

Typesafe

Actors and Mutability

- Actors may have mutable state:
- --> Akka takes care of memory consistency
- --> Attention: **Don't** share mutable state!
- --> Actors exclusively communicate with message passing
- --> Attention: Messages **must be immutable!**

Typesafe

Actor Systems I

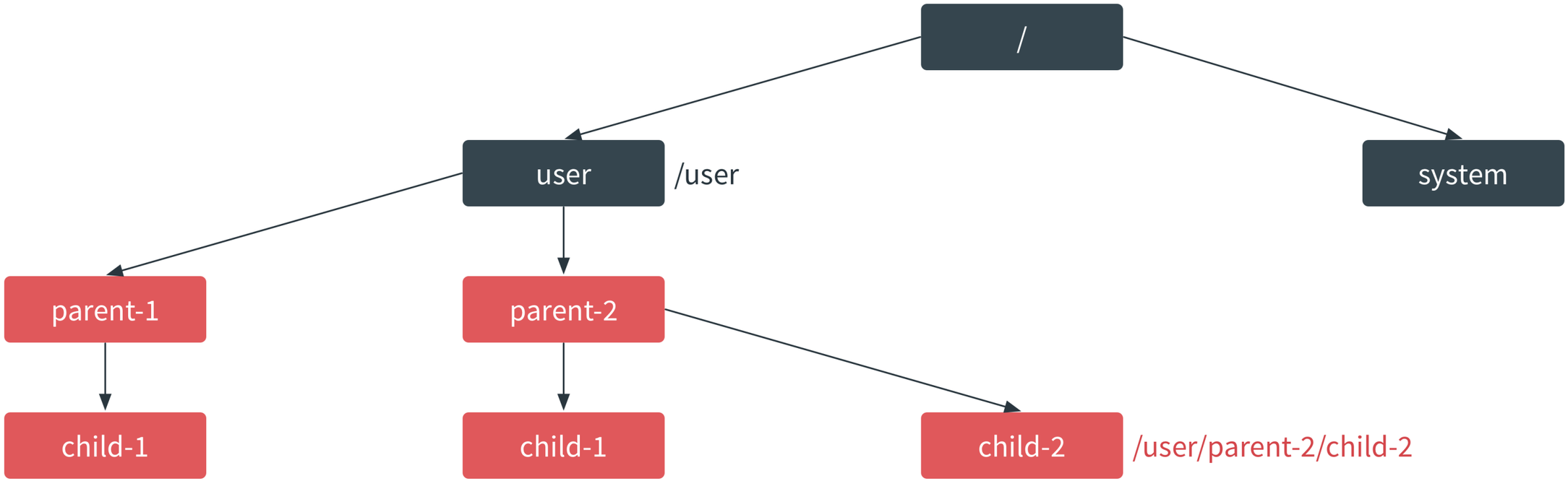
"One actor is no actor, they come in systems." --Carl Hewitt

- An actor system is a collaborating ensemble of actors
- Actors are arranged in a hierarchy:
- --> Actors can split up and delegate tasks to child actors
- --> Child actors are supervised and delegate their failure back to their parent

Actor Systems II

- Actor Systems provide shared facilities:
- --> Factory for top-level actors: actorOf
- --> Dispatchers and thread pools: heavyweight
- --> Scheduling service: scheduler
- --> Access to configuration: settings.config
- --> Publish-subscribe eventStream:
- ----> used internally for logging,
- ----> unhandled messages and dead letters, but open for user code
- There can be multiple actor systems per JVM or even per classloader, because Akka doesn't use any global state

Anatomy of an Actor System



Typesafe

Anatomy of an Actor System

- Within an actor system actors are arranged in a hierarchy
- Therefore each actor has a parent:
- --> Top-level actors are children of the guardian
- --> Each actor can create child actors
- Each actor has a name which is unique amongst its siblings

Implementing an Actor

```
class Restaurant extends Actor with ActorLogging {

  val name = "Reactive Restaurant"
  log.debug(s"$name has opened!")

  // TODO Define behavior
}
```

- To implement an actor mix-in the `Actor` trait
- To use Akka's logging facility mix-in the `ActorLogging` trait
- Constructor is body of class

Initial Behavior

```
class Restaurant extends Actor {

  override def receive: Receive = {
    case _ => println("Welcome to the Reactive Restaurant!")
  }
}
```

- Behavior is simply a PartialFunction[Any, Unit]:
- --> Receive is a type alias defined in the Actor trait
- The receive method returns the actor's initial behavior:
- --> Messages are not handled by receive, but by its return value
- --> The behavior can be changed at runtime (more to come later)

Unhandled Messages

```
def unhandled(message: Any): Unit = ...
```

- What happens if a message isn't handled by the behavior?
- --> an `UnhandledMessage` event is published to the event stream
- --> or a `DeathPactException` is thrown for an unhandled `Terminated` message (more to come later)
- This could be customized by overriding the `unhandled` method
- In order to log unhandled messages, set the configuration setting `akka.actor.debug.unhandled` to on

Exercise 001: Implement an Actor

- In this exercise, we will implement the `Restaurant` actor with logging
- Make sure you are in the `[001] implement_actor >` project
- Run `man` to see the instructions

Typesafe

Creating an Actor System

```
val system = ActorSystem("reactive-restaurant")
...
system.shutdown()
```

- To create an actor system call the ActorSystem factory method
- Attention: As an actor system is heavyweight, don't forget to shutdown the actor system "at the end"!

Typesafe

Creating an Actor

```
val fooProps = Props(new Foo("bar")) // by-name argument
factory.actorOf(fooProps)
```

- To create an actor's you need `Props`:
- --> Props configure an actor, most notably its class
- --> To create `Props` use one of its factory methods
- To create an actor call the `actorOf` method of an `ActorRefFactory`:
- --> This prevents you from accessing an actor directly
- --> The optional name must not be empty or start with `$`

Props Factories

```
object SomeActor {

  def props(someParam: String, anotherParam: Int): Props =
    Props(new SomeActor(someParam, anotherParam))
}
```

- You could create `Props` in place when needed
- Yet for remoting `Props` need to be serializable:
- --> The by-name argument factory for `Props` closes over the outer object
- --> If this outer object is an actor the `Props` are not serializable
- **Best practice**: Define a props factory method in the actor's companion object

Creating a Top-Level Actor

```
val restaurant = createRestaurant()

def createRestaurant(): ActorRef =
  system.actorOf(Restaurant.props, "restaurant")
```

- To create a top-level actor call `ActorSystem.actorOf`
- If you give a name, it has to be unique amongst its siblings
- If you create an anonymous actor, Akka synthesizes a name
- Creating an actor is an asynchronous operation
- **Best practice**: Use dedicated factory methods for creating top-level actors to facilitate testing

Digression: Configuration

```
akka {
  actor {
    debug {
      lifecycle = on
      unhandled = on
    }
  }
}
```

- Akka uses the Typesafe Config Library
- By default Akka looks for `application.conf` on the classpath
- For more details see the Akka documentation

Digression: Logging Configuration

```
akka {
  loggers = [akka.event.slf4j.Slf4jLogger]
  loglevel = debug
}
```

- For real-world scenarios use the `Slf4jLogger`
- The Akka logging level is defined with `loglevel`
- The reactive-restaurant project is already ready for logging:
- --> Library dependencies on `akka-slf4j` and `Logback`
- --> `logback.xml` configuration file
- --> **Attention**: You have to use `tail -f reactive-restaurant.log` to watch the log output!

Typesafe

Exercise 002: Create a Top Level Actor

- In this exercise, we will make `Restaurant` a top-level actor and implement some configuration properties
- Make sure you are in the `[002] top_level_actor >` project
- Run `man` to see the instructions

Typesafe

Communication

```
val restaurant: ActorRef = ...
restaurant ! "Nice Food!"
```

- To send a message to an actor, you need an actor reference
- Call its ! operator with any message
- Execution continues without waiting for a response in *fire-and-forget* manner

Typesafe

Messages

```
object Waiter {
  case class OrderFood(name: String)
  case class FoodOrdered(name: String, customer: ActorRef)
}

class Waiter extends Actor {
  import Waiter._

  override def receive: Receive = {
    case OrderFood(name) =>
      println(s"Order food $name")
    case FoodOrdered(name, customer) =>
    println(s"Food $name ordered for $customer")
  }
}
```

Messages

- Attention: Messages must be immutable!
- **Best practice**:
- --> Use case objects and/or case classes
- --> Define message protocol in the actor's companion object

Exercise 003: Send a Message to an Actor

- In this exercise, we will send a message to our `Restaurant`
- Make sure you are in the `[003] message_actor >` project
- Run `man` to see the instructions

Typesafe

Including the Sender

```
trait ScalaActorRef {
  def !(msg: Any)(implicit sender: ActorRef = Actor.noSender): Unit
  ...
}
```

- Using ! tries to implicitly include the sender
- Quiz:
- --> What happens if you call ! from within an actor?
- --> What happens if there is no implicit actor reference in scope?
- --> **Hint**: Take a look at the API documentation

Using the Sender

```
class Restaurant extends Actor {
  override def receive: Receive = {
    case _ => sender() ! "Welcome to the Restaurant!"
  }
}
```

- sender gives you access to the sender of the current message
- **Quiz**: Why might the sender be ActorSystem.deadLetters?
- **Attention**: sender is a method, don't let it leak!

Exercise 004: Use the Sender

- In this exercise, we will use the implicit sender to respond from `Restaurant`
- Make sure you are in the `[004] use_sender >` project
- Run `man` to see the instructions

Typesafe

Actor Context

```
trait Actor {
  implicit val context: ActorContext = ...
  ...
}
```

- Each actor has an implicit `ActorContext`
- This provides contextual information and operations:
- --> Access to self and the current `sender`
- --> Access to parent and children
- --> Create child actors and stop actors
- --> Death watch, change behavior, etc. (more to come later)

Forwarding Messages

```
abstract class ActorRef {
  def forward(message: Any)(implicit context: ActorContext) = ...
  ...
}
```

- forward sends a message passing along the sender from the actor context
- This way you can make an actor you are sending a message to respond to the actor you received the current message from
- **Quiz**: Why can you call forward from within an actor?

Creating a Child Actor

```
class Restaurant extends Actor {

  val guest = createGuest()

  def createGuest(): ActorRef =
    context.actorOf(Guest.props)
}
```

- To create a child actor call `ActorContext.actorOf`
- Like for top-level actors you get back an `ActorRef`
- **Best practice**: Use dedicated factory methods for creating child actors to facilitate testing

Exercise 005: Create Child Actor

- In this exercise, we will use `Restaurant` to create `Guest` as a child actor
- Make sure you are in the `[005] create_child_actor >` project
- Run `man` to see the instructions

Typesafe

Looking up Actors

```
context.actorSelection("/user/reactive-restaurant")
context.actorSelection("../sibling")
context.actorSelection("/user/restaurant/*")
```

- To look up actors use the `actorSelection` method of an `ActorRefFactory`
- You can use absolute or relative paths or even wildcards

Typesafe

Looking up Actors

- The ActorSelection you get back
- --> is a logical view of a section of the actor tree
- --> is unverified, i.e. messages might end up in deadLetters
- --> is less performant when it comes to messaging, because the actor hierarchy has to be traversed can't be used for death watch
- **Attention**: Don't overuse actor lookup!

Typesafe

Identifying Actors

```
val sibling = context.actorSelection("../sibling")
sibling ! Identify(1)

override def receive: Receive = {
  case ActorIdentity(1, Some(actorRef)) => // Lookup successful
  case ActorIdentity(1, None)           => // Lookup failed
}
```

- To obtain an actor reference send `Identify` to an `ActorSelection`
- For a successful lookup the `ActorIdentity` response contains an `ActorRef`
- `Identify` is treated specially: If the lookup fails, you still get a response

Actor State

```
class Counter(offset: Int) extends Actor ...

system.actorOf(Props(new Counter(0)))
```

- Actors can have both mutable and immutable state
- Parameters are a typical example for immutable state
- To create an actor with parameters you have to use the `Props` factory method taking a by-name argument

Mutable Actor State

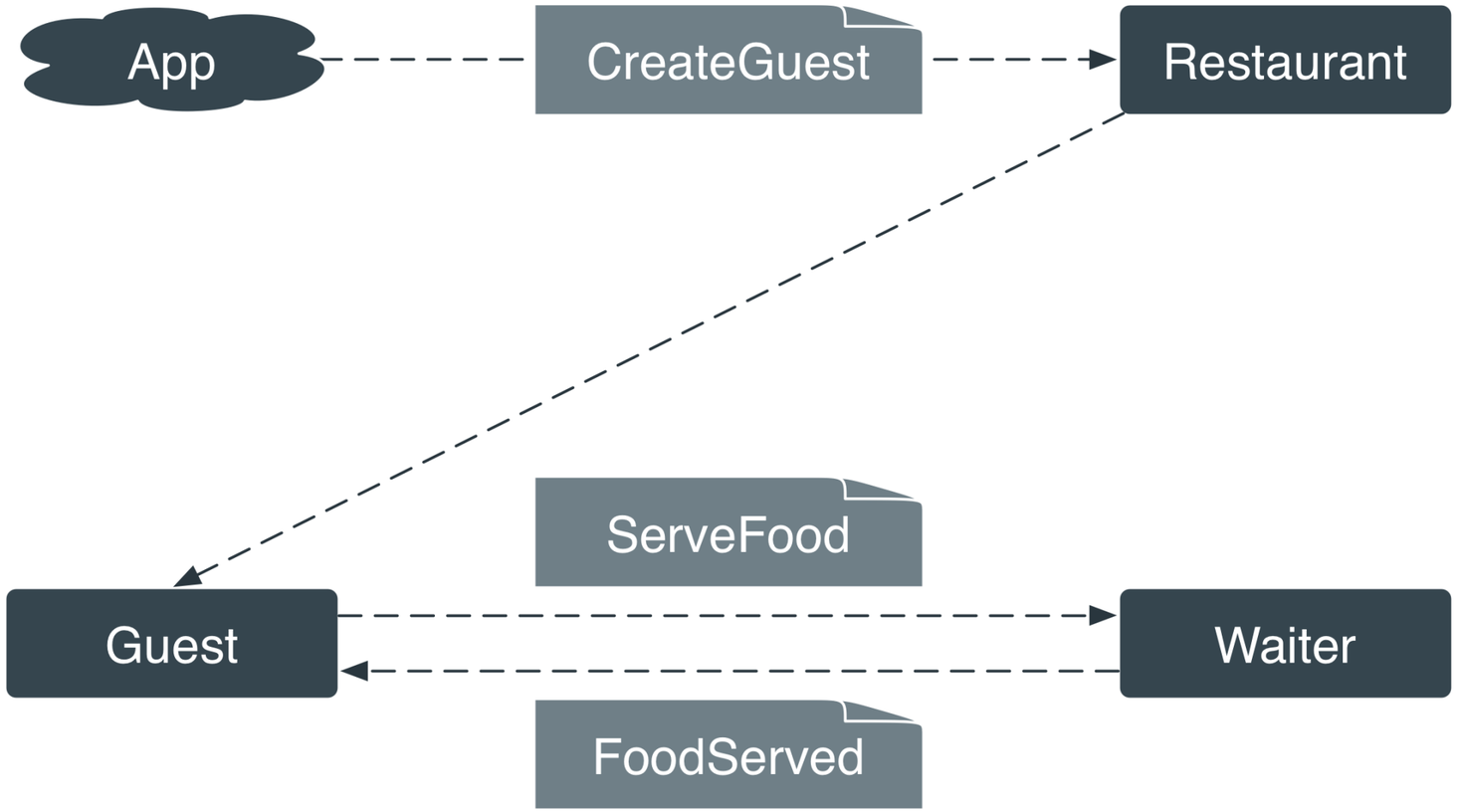
```
class Counter(offset: Int) extends Actor {

  private var count = offset

  override def receive: Receive = {
    case _ => count += 1
  }
}
```

- Mutable actor state can be used in a single-threaded illusion
- Akka takes care of all concurrency related aspects

Exercise 006: Actor State



Typesafe

Exercise 006: Actor State

- In this exercise, we will implement state by tracking a `Guest` actors favorite Food
- make sure you are in the `[006] actor_state >` project
- run `man` to see the instructions

Typesafe

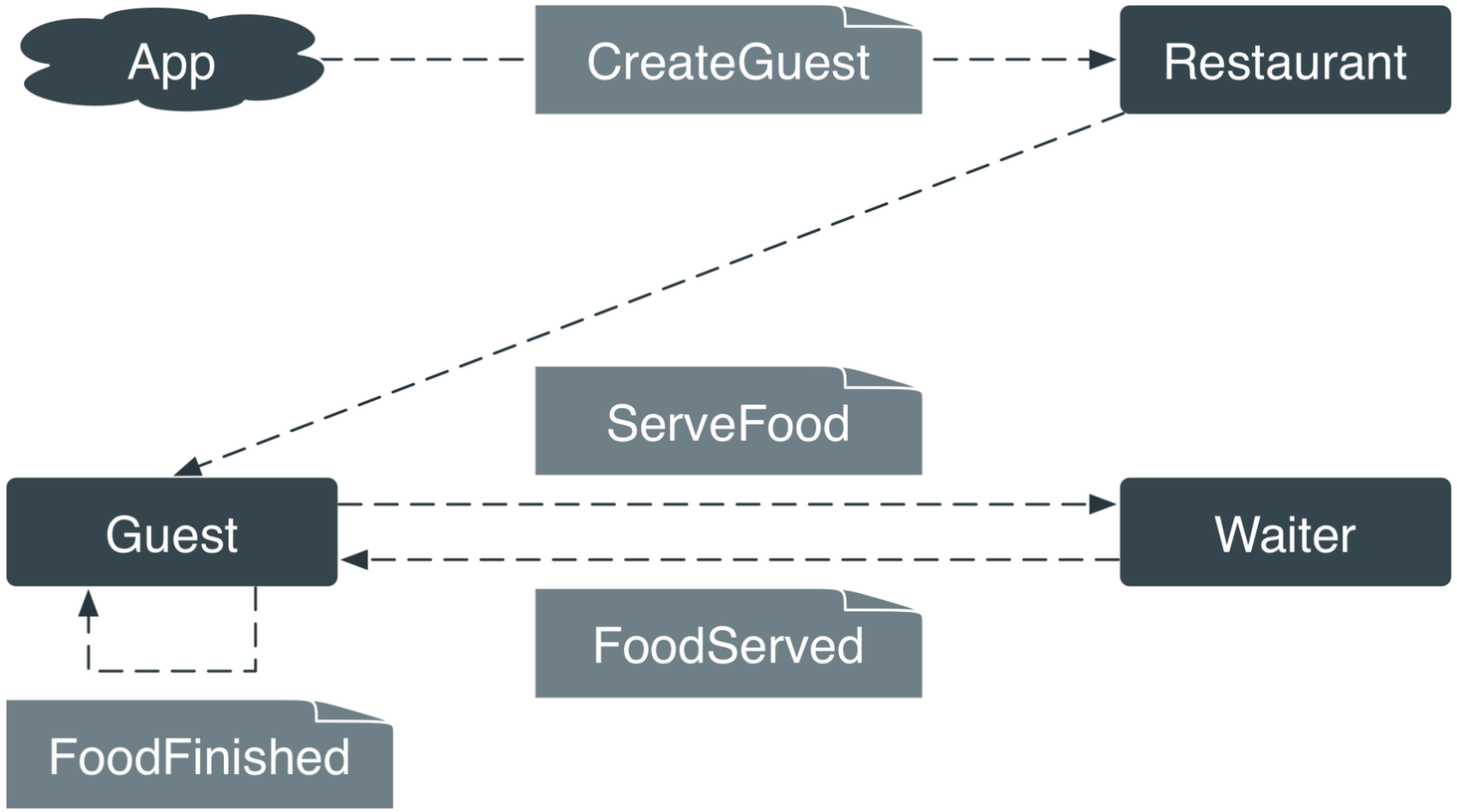
Scheduler Service

```
import context.dispatcher
import scala.concurrent.duration._

context.system.scheduler.scheduleOnce(
  2 seconds,
  self,
  FoodFinished
)
```

- The actor system offers a scheduler:
- --> You can run a task or send a message to an actor
- --> This can be scheduled once or periodically
- You need an implicit ExecutionContext, e.g. an actor's dispatcher
- The duration DSL can also be used in the configuration file

Exercise 007: Use Scheduler



Typesafe

Exercise 007: Use Scheduler

- In this exercise, we will implement the Akka Scheduler to simulate the `Guest` ordering more Food
  Make sure you are in the `[007] use_schedular >` project
  Run `man` to see the instructions

Typesafe

Testing Actors

Testing Actors is Different

- Two properties of the actor model complicate testing:
- --> Actors can only be accessed through their actor reference
- --> Multi-threaded messaging is non-deterministic
- Akka provides the `akka-testkit` module addressing both issues

Akka Testing Styles

- Unit testing:
- --> Isolated pieces of code
- --> Bypassing the actor model
- --> Single-threading scheduling
- --> Deterministic messaging
- Integration testing:
- --> Actor interactions
- --> Multi-threaded scheduling
- --> Non-deterministic messaging

Digression: ScalaTest Quick Reference

```
class CalculatorSpec extends WordSpec with Matchers {

  "Calling divide" should {
    "calculate the correct result" in {
      divide(4, 2) shouldEqual 2
    }
    "throw an ArithmeticException for division by 0" in {
      an[ArithmeticException] should be thrownBy divide(4, 0)
    }
  }
}
```

- Extend from `WordSpec` and mix-in `Matchers`
- Describe the subject (system under test) followed by `should`
- Describe each test followed by `in`
- Use matchers to implement the tests

Synchronous Unit Testing

```
val counter = TestActorRef(new Counter)
val counterActor = counter.underlyingActor
counter ! Tick
counterActor.count shouldEqual 1
```

- `TestActorRef` enables white-box testing of individual actors:
- --> Get a reference to the underlying actor
- --> Directly invoke the actor's initial behavior
- In addition you get synchronous messaging
- To create a `TestActorRef` you need an implicit actor system
- **Attention**: Don't overuse `TestActorRef`!

Asynchronous Integration Testing

```
"Sending Ping to PongActor" should {
  "result in sending a Pong response" in {
    val ping = TestProbe()
    val pong = system.actorOf(Props(new PongActor(ping.ref)))
    pong ! Ping
    ping.expectMsg(Pong)
  }
}
```

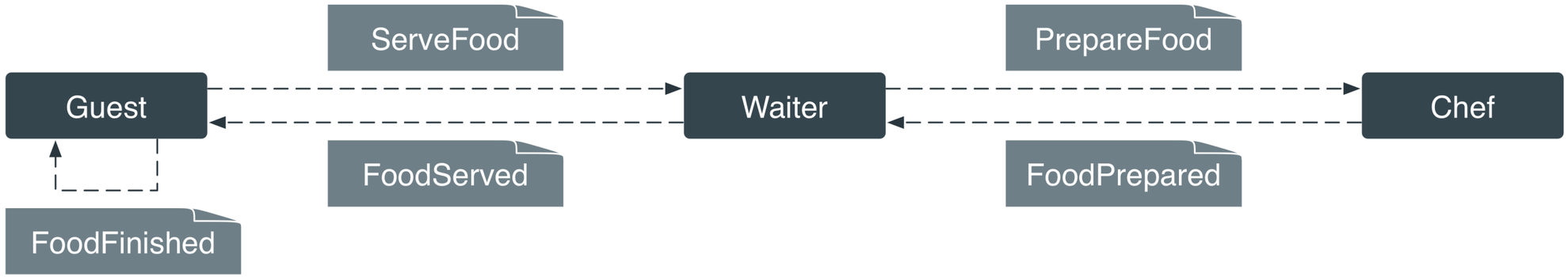
- `TestProbe` can be used to mock actors
- Inspect message flows using one of many built-in assertions, e.g.:
- `--> expectMsg`
- `--> expectNoMsg`

Typesafe

Group Exercise

- Use the `TestKit`
- Let's take a look at the GuestSpec class
- See how `BeforeAndAfterAll` is used to shutdown the actor system at the end
- See how `expectMsg` and within are used to inspect the message flow between `Guest` and `Waiter`

Exercise 8: Busy Actor



Typesafe

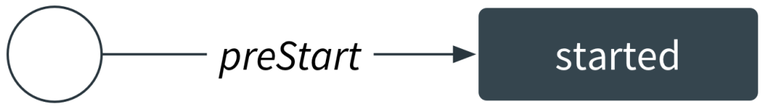
Exercise 8: Busy Actor

- In this exercise, we introduce a `Chef` actor who specializes in making our fine food and will keep our other actors busy
- make sure you are in the `[008] keep_actor_busy >` project
- run `man` to see the instructions

Typesafe

Actor Lifecycle

Starting Actors



- Creating an actor automatically starts it
- A started actor is fully operable, i.e. it can handle messages
- You can override the preStart hook

Typesafe

Stopping Actors

- An actor can stop itself or be stopped by another one
- A stopped actor is no longer operable (it won't process any messages)
- Override the postStop hook; the actor is considered terminated

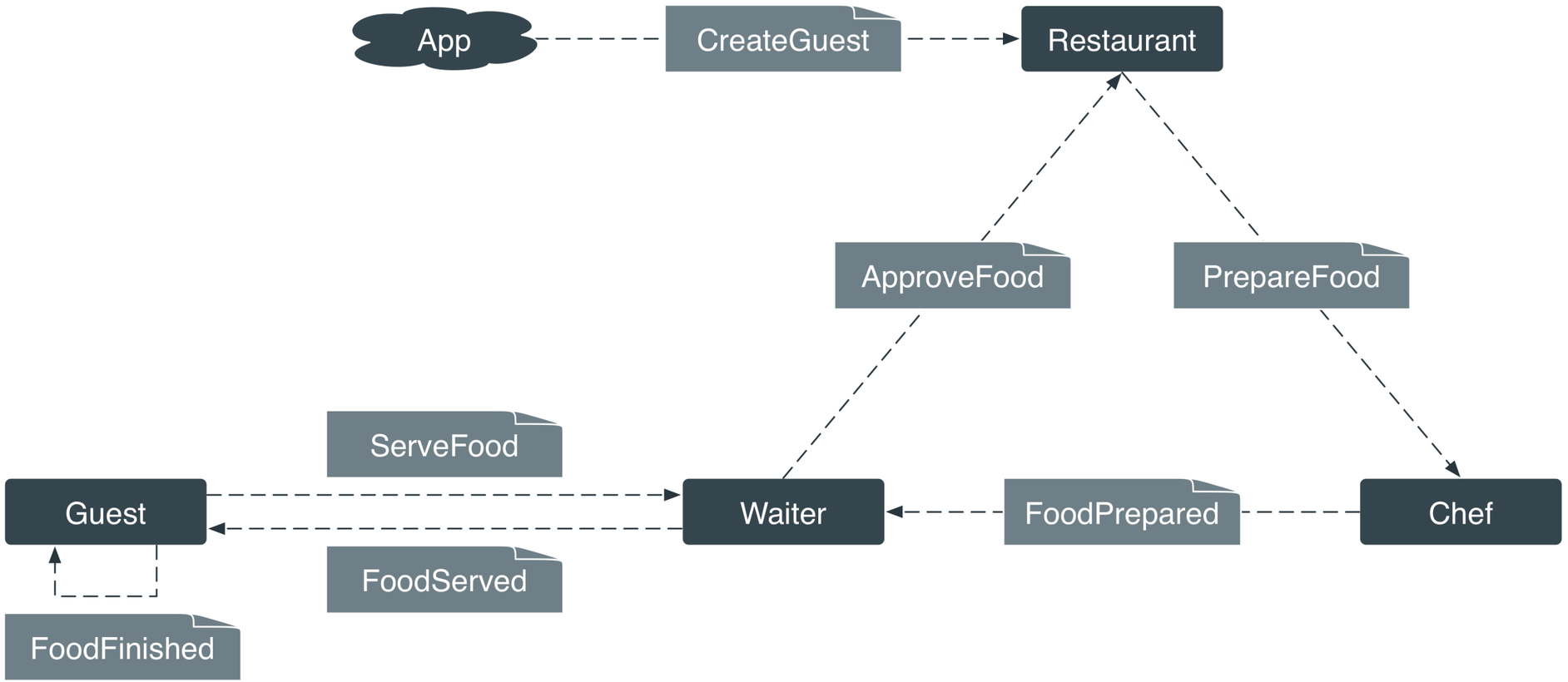
Typesafe

Stopping Actors with the API

```
context.stop(self)
context.stop(other)
```

- Both `ActorSystem` and `ActorContext` provide a stop method
- Stopping an actor is an asynchronous and recursive operation:
- --> The actor finishes processing the current message
- --> if any, suspends message processing, stops its children,
- --> waits for their termination confirmations, then terminates itself.
- Stopping an actor inevitably stops all its descendants

Exercise 009: Stop an Actor



Typesafe

Exercise 009: Stop an Actor

- In this exercise we will limit the number of `foods` a `Guest` consumes by setting a food limit per Guest. When the Guest reaches their limit, we will stop the actor
- Make sure you are in the `[009] stop_actor >` project
- Run `man` to see the instructions

Typesafe

Stopping Actors gracefully

```
other ! StopMeGracefully
```

- If an actor needs to perform certain tasks before stopping, it
- --> should advertise a dedicated message for that purpose,
- --> handle it as appropriate and
- --> stop itself thereafter
  The `PoisonPill` offered by Akka is deprecated and will be removed:
- --> It is too simplistic
- --> The sender needs intimate knowledge of the recipient's inner workings in order to determine whether `PoisonPill` will have the intended effect

Lifecycle Monitoring

```
context.watch(other)
case Terminated(other) => // Do something ...
```

- An actor can monitor another actor's termination, aka death watch
- **Quiz**: Why is there no birth watch?
- On termination the monitoring actor is sent a `Terminated` message
- If `Terminated` is not handled, a `DeathPactException` is thrown by default

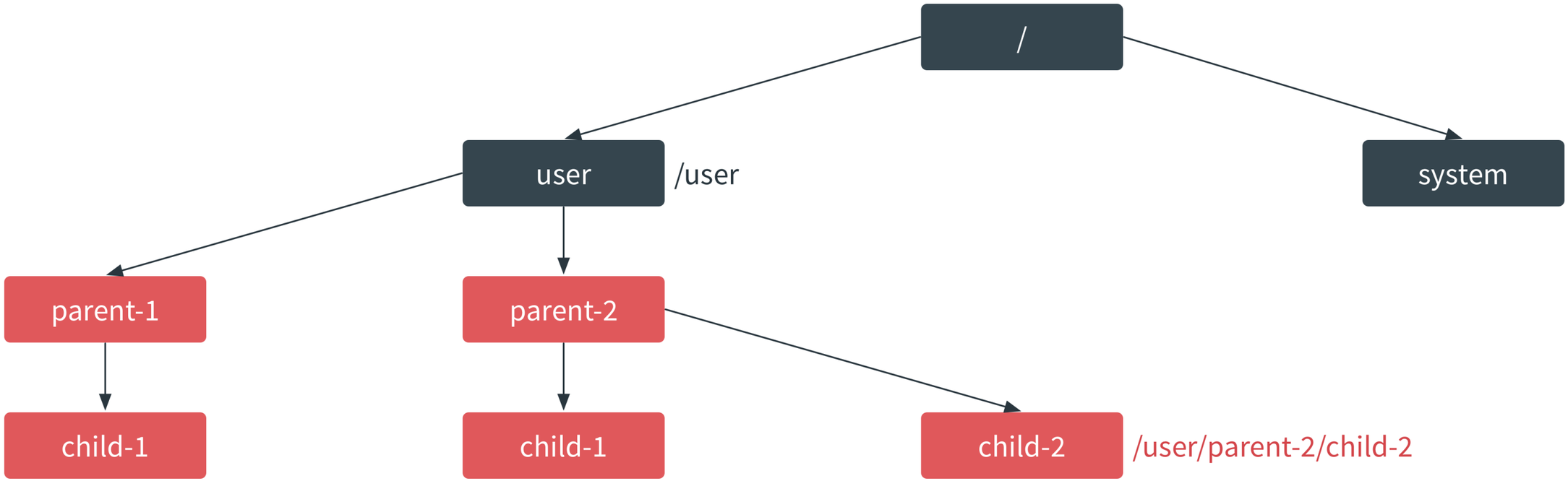
Exercise 010: Lifecycle Monitoring

- Sometimes we need to perform certain tasks before stopping an actor. In this exercise, we will explore this idea by watching for the `Termination` message
- Make sure you are in the `[010] lifecycle_monitoring >` project
- Run `man` to see the instructions

Typesafe

Fault Tolerance

Actor Systems Revisited



Typesafe

Actor Systems Revisited

- Within an actor system actors are arranged in a hierarchy
- Therefore each actor has a parent
- Each actor has a name which is unique amongst its siblings
- Therefore each actor can be identified by a unique sequence of names

Actor Path

```
akka://my-system/user/ParentA/ChildA
akka.tcp://my-system@host.domain.com:5678/user/ParentA/ChildA

scala> val path = coffeeHouse.path
path: ...ActorPath = akka://coffee-house-system/user/coffee-house

scala> path.name
res0: String = coffee-house
```

- An ActorPath encapsulates
- --> the sequence of actor names together with
- --> the transport protocol and
- --> the address of the actor system

Actor Path

```
akka://my-system/user/ParentA/ChildA
akka.tcp://my-system@host.domain.com:5678/user/ParentA/ChildA

scala> val path = coffeeHouse.path
path: ...ActorPath = akka://coffee-house-system/user/coffee-house

scala> path.name
res0: String = coffee-house
```

- To obtain an actor path call path on ActorRef
- To get an actor's name call name on ActorPath

Embrace Failure

```
1 + 1 = 3
```

- Errors are a fact of life
- Don't worry, just let it crash
- Instead of trying to prevent failure simply handle it properly

Typesafe

Failure in Akka

- Akka deals with failure at the level of individual actors
- An actor fails when it throws an exception (NonFatal throwable)
- Failure can occur
- --> during message processing
- --> during initialization
- --> within a lifecycle hook, e.g. preStart
- How should such failure be handled?
- Let's see what happens by default ...

Typesafe

Exercise 011: Create a Faulty Actor

- In this exercise, we will explore resilience by managing a faulty actor
- Make sure you are in the `[011] faulty_actor >` project
- Run `man` to see the instructions

Typesafe

Resilience

```
class SomeActor extends Actor {
  override val supervisorStrategy: SupervisorStrategy = ...
}
```

- As you can see, a faulty actor doesn't bring down the whole system
- This fault tolerance is implemented through **parental supervision**:
- --> If an actor fails, its message processing is suspended,
- --> its children are suspended recursively – i.e. all descendants – and
- --> its parent has to handle the failure

Typesafe

Resilience

```
class SomeActor extends Actor {
  override val supervisorStrategy: SupervisorStrategy = ...
}
```

- Each actor has a supervisor strategy for handling failure of child actors
- As you can see there is a default supervisor strategy in place
- In most cases a `val` should be used to `override supervisorStrategy`

Typesafe

Supervisor Strategies

- Akka ships with two highly configurable supervisor strategies:
- --> `OneForOneStrategy`: Only the faulty child is affected when it fails
- --> `AllForOneStrategy`: All children are affected when one child fails
- Both are configured with a `Decider`:
- --> type `Decider` = `PartialFunction[Throwable, Directive]`
- --> A decider maps specific failure to one of the possible directives
- --> If not defined for some failure, the supervisor itself is considered faulty

Supervisor Strategy Directives

- `Resume`: Simply resume message processing
- `Restart`:
- --> Transparently replace affected actor(s) with new instance(s)
- --> Then resume message processing
- `Stop`: Stop affected actor(s)
- `Escalate`: Delegate the decision to the supervisor's parent

Restarting vs. Resuming

- Like stopping, restarting and resuming are recursive operations
- In both cases, no messages get lost, except for the "faulty" message, if any
- Resuming simply resumes message processing for the faulty actor and its descendants:
- --> The actor state remains unchanged
- --> Use `Resume` if the state is still considered valid
- Restarting transparently replaces the affected actor(s) with new instance(s):
- --> Actor state and behavior get reinitialized
- --> Use `Restart` if the state is considered corrupted because of the failure
- --> By default all children get stopped (see the preRestart lifecycle hook)
- --> Any children that don't get stopped get restarted

Fine-tuning Restarting I

```
override val supervisorStrategy: SupervisorStrategy =
  OneForOneStrategy(maxNrOfRetries = 1) { ... }
```

- `maxNrOfRetries` limits the number of possible restarts
- A negative number, which is the default, means no limit
- If an actor can't be restarted within the limit, it gets stopped

Typesafe

Fine-tuning Restarting II

```
override val supervisorStrategy: SupervisorStrategy =
  OneForOneStrategy(5, 1 minute) { ... }
```

- Restarting is tried up to `maxNrOfRetries` times within consecutive time windows defined by `withinTimeRange`
- If a window has passed without reaching the retry limit, retry counting begins again for the next window
- By default `withinTimeRange` is `Duration.Inf`, i.e. there is only one window

Typesafe

Default Supervisor Strategy

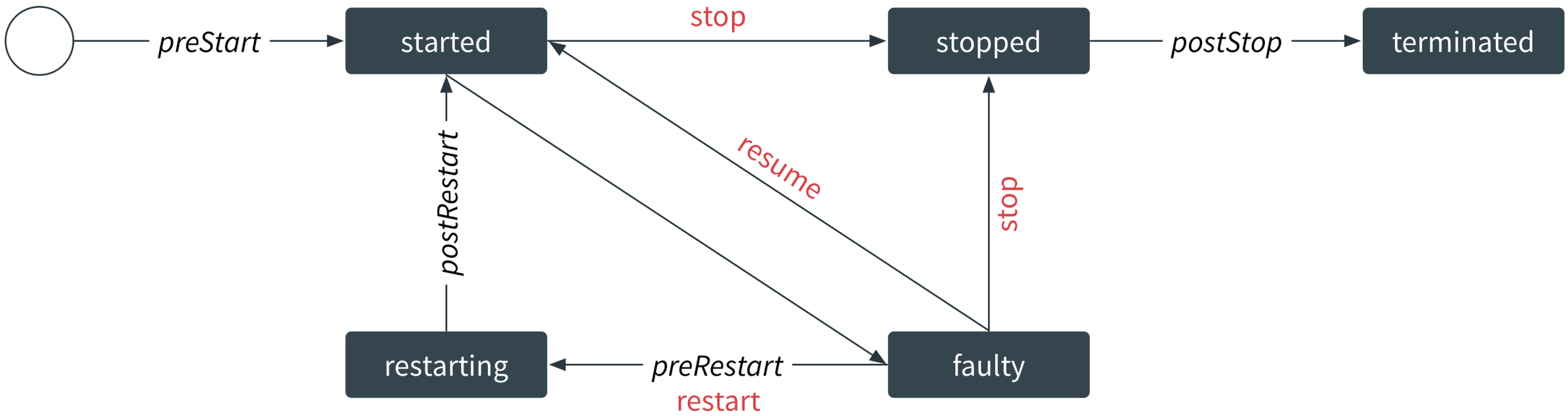
- If you don't override supervisorStrategy, a OneForOneStrategy with the following decider is used by default:
- --> `ActorInitializationException` → `Stop`
- --> `ActorKilledException` → `Stop`
- --> `DeathPactExceptions` → `Stop`
- --> Other Exceptions → `Restart`
- --> Other Throwables → `Escalates` to it's parent
- Therefore, in many cases, your actor will be restarted by default

Exercise 012: Custom Supervision

- In this exercise, we will further explore resilience by implementing custom supervision
- Make sure you are in the `[012] custom_supervision >` project
- Run `man` to see the instructions

Typesafe

Actor Lifecycle revisited



Typesafe

Actor Lifecycle revisited

- The full lifecycle contains the additional states faulty and restarting
- You can override the preRestart and postRestart hooks, which are called whenever an actor is restarted by its supervisor

Restart Hooks

```
def preRestart(reason: Throwable, message: Option[Any]): Unit = ...
def postRestart(reason: Throwable): Unit = ...
```

- **Quiz**: Why is the message parameter of `preRestart` optional?
- By default `preRestart` stops any children, then calls `postStop`
- By default `postRestart` calls `preStart`
- **Quiz**: On which actor instances are these hooks called?

Self Healing

- Failure could easily stop a system from working properly, e.g. because messages or actor state get lost
- Therefore it is essential to build a self healing system:
- --> If the supervisor has enough information, it can reconstruct all state and resend all messages
- --> If not, we need other ways to heal (not covered here)
- Let's take a look at an example ...

Exercise 013: Another Faulty Actor

- In this exercise, we will introduce another faulty actor in the form of our Chef where sometimes they make the wrong food. When this happens, the Guest will complain and reorder. If the Waiter receives too many complaints, he will become frustrated
- Make sure you are in the `[013] another_faulty_actor >` project
- Run `man` to see the instructions

Typesafe

Exercise 014: Implement Self Healing

- In this exercise, we will correct a problem introduced in the last exercise by implement self-healing
- Make sure you are in the `[014] self_healing >` project
- Run `man` to see the instructions

Typesafe

Elasticity

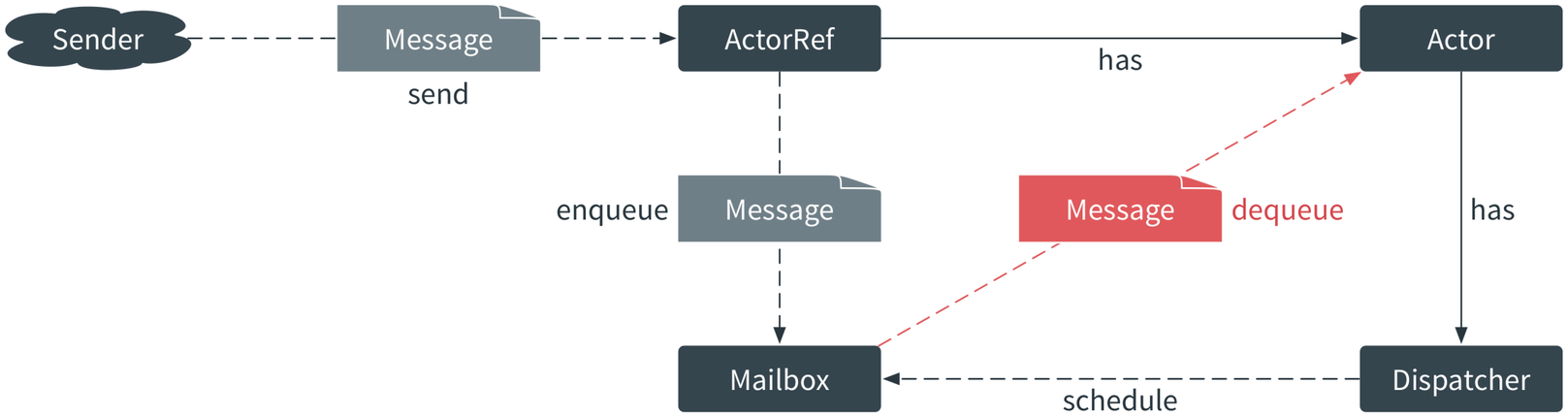
Routers

Exercise 015: Detect the Bottleneck

- In this exercise, we will change our configuration settings to see if we can detect a bottleneck
- Make sure you are in the `[015] detect_bottleneck >` project
- Run `man` to see the instructions

Typesafe

Message Processing revisited



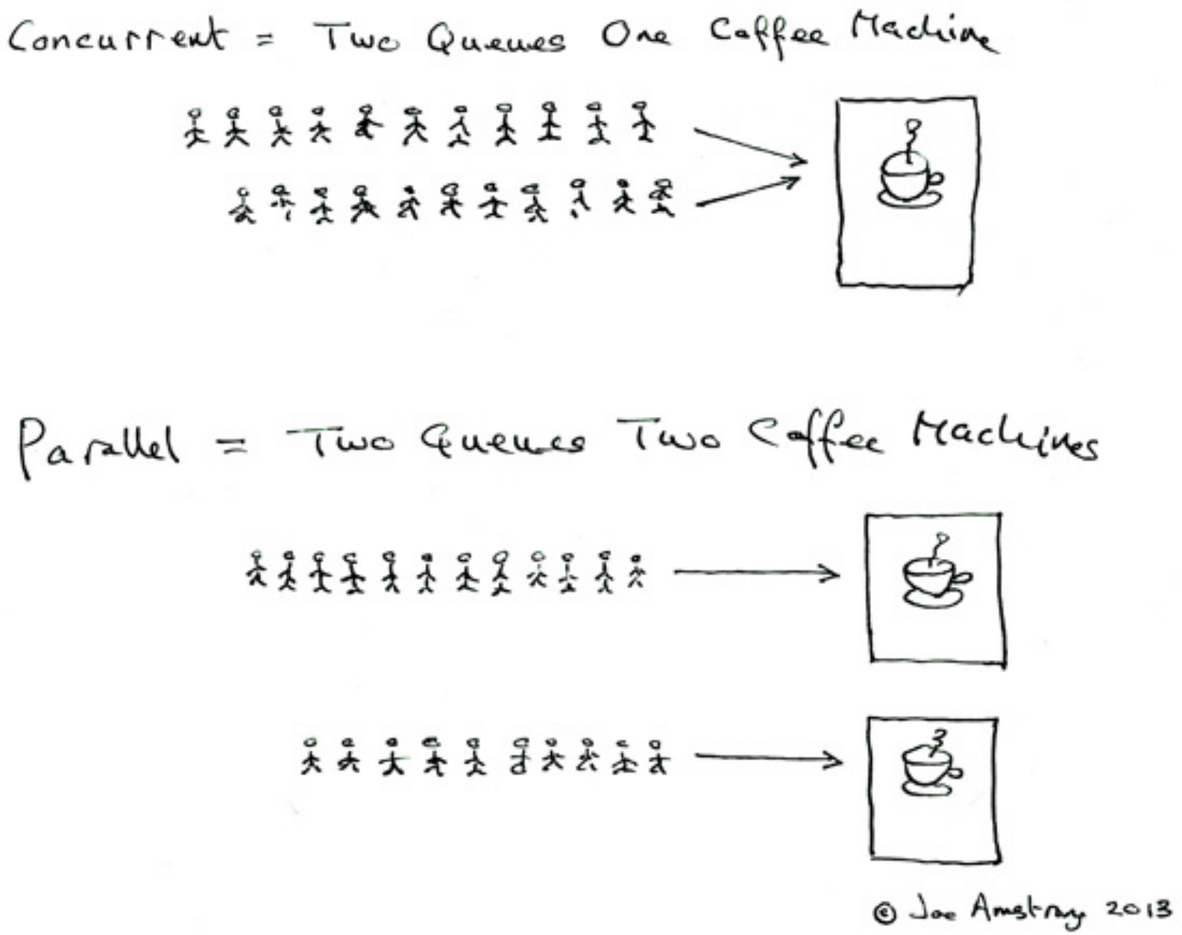
- An actor processes (at most) one message at a time
- If you want to scale up, you have to use multiple actors in parallel

Typesafe

Digression: Concurrency and Parallelism I

- Definition: Two or more tasks are concurrent, if the order in which they get executed in time is not predetermined
- --> In other words, concurrency introduces non-determinism
- --> Concurrent tasks may or may not get executed in parallel
- --> Hence concurrency is a more general concept than parallelism
- Concurrent programming is primarily concerned with the complexity that arises due to non-deterministic control flow
- Parallel programming aims at improving throughput and making control flow deterministic

Digression: Concurrency and Parallelism II



- Concurrency is a property of the program
- Parallel execution is a property of the machine

Typesafe

Akka Routers

- A router routes messages to destination actors called routees
- Depending on your needs, different routing strategies can be applied
- Routers can be used standalone or as self contained router actors

Routing Strategies provided by Akka

- `RandomRoutingLogic`
- `RoundRobinRoutingLogic`
- `SmallestMailboxRoutingLogic`
- `ConsistentHashingRoutingLogic`
- `BroadcastRoutingLogic`
- `ScatterGatherFirstCompletedRoutingLogic`
- To write your own routing strategy, extend RoutingLogic:
- --> **Attention**: The implementation must be thread-safe!

Typesafe

Router Actors

- Akka provides two flavors of self contained router actors:
- --> `Pool` router: creates routees as child actors
- --> `Group` router: routees are provided via actor path
- Message delivery is optimized:
- --> Messages don't get enqueued in the mailbox of the router actor
- --> Instead, messages are delivered to a routee directly

Specially handled Messages

- Every message sent to a router is delivered to one of its routees
- Yet the following messages are handled in a special way:
- --> `PoisonPill` is not delivered to any routee
- --> `Kill` is not delivered to any routee
- --> The payload of `Broadcast` is delivered to all routees
- **Quiz**:
- --> What happens to the routees if you send the PoisonPill to a router?
- --> How can you stop all routees gracefully, i.e. handling already received messages before stopping?

Creating a Router Actor

```
context.actorOf(
  Props(new SomeActor).withRouter(FromConfig()),
  "some-actor"
)

context.actorOf(
  RoundRobinPool(4).props(Props(new SomeActor)),
  "some-actor"
)
```

Creating a Router Actor

- Router actors must be created programmatically
- Either use `withRouter` with a `RouterConfig`:
- --> `FromConfig` completely relies on external configuration
- --> Other RouterConfigs use a mix of programmatic and external configuration
- Or use the props method of a `Pool` or `Group` configuration

Important Router Configuration

- Settings can be defined in configuration or programmatically:
- --> If both are given, configuration wins
- A pool router creates `nrOfInstances` child actors as routees:
- --> An optional resizer can dynamically adjust the number of routees
- --> The default supervisorStrategy escalates all failure
- A group router uses existing routees
- **Quiz**: What happens when a routee of a pool or a group router fails?

Router Configuration Example

```
akka {
  actor {
    deployment {
      /top-level/child-a {
        router = smallest-mailbox-pool
        nr-of-instances = 4
      }
      /top-level/child-b {
        router = round-robin-pool
        resizer {
          lower-bound = 1
          upper-bound = 4
        }
      ...
```

Exercise 016: Uses a Router

- In this exercise, we will introduce parallelism through the use of routers
- Make sure you are in the `[016] use_router >` project
- Run `man` to see the instructions

Typesafe

Dispatchers

Akka Dispatchers

```
akka.actor.default-dispatcher = ...
context.actorOf(
  Props[SomeActor].withDispatcher("my-dispatcher")
)
```

Typesafe

Akka Dispatchers

- Dispatchers are Akka's engine, they make actors "tick" by
- --> implementing `scala.concurrent.ExecutionContext` and
- --> registering an actor's mailbox for execution
- Dispatchers determine execution time and context and therefore provide the physical capabilities for scaling up
- Each actor system has a dispatcher used as default for all actors
- You can set an externally configured dispatcher for an actor

Dispatchers provided by Akka

- `Dispatcher` (default): Event-driven dispatcher, sharing threads from a thread pool for its actors
- `PinnedDispatcher`: Dedicates a unique thread for each actor
- `BalancingDispatcher`: Event-driven dispatcher redistributing work from busy actors to idle ones
- `CallingThreadDispatcher`: For testing, runs invocations on the current thread only
- To use your own dispatcher, provide a `MessageDispatcherConfigurator`

Dispatcher Configuration Example

```
akka {
  actor {
    default-dispatcher {
      fork-join-executor {
        parallelism-min = 4
        parallelism-factor = 2.0
        parallelism-max = 64
      }
      throughput = 5 // default
    }
  ...
```

Typesafe

Exercise 017: Configure Dispatcher

- In this exercise, we will optimize parallelism through the configuring a dispatcher
- Make sure you are in the `[017] config_dispatcher >` project
- Run `man` to see the instructions

Typesafe

Behavior

Hot swapping Actor Behavior

```
override def receive: Receive =
  ready

def ready: Receive = {
  case msg =>
    // Send *Done* to *self* when done
    context.become(busy)
}

def busy: Receive = {
  case Done => context.become(ready)
}
```

Hot swapping Actor Behavior

- `receive` defines the initial behavior
- `become` lets you swap in a new one
- Idiomatic use case: finite state machine
- **Quiz**: Which behavior is used after a restart?

Behavior Stack

```
def ready: Receive = {
  case msg =>
    // Send *Done* to *self* when done
    context.become(busy, discardOld = false)
}

def busy: Receive = {
  case Done => context.unbecome()
}
```

Behavior Stack

- The current behavior can be popped onto a behavior stack
- `unbecome` swaps in the topmost behavior again
- **Attention**: Make sure you don't create memory leaks!

Stashing away Messages

```
def ready: Receive = {
  case msg =>
    // Send *Done* to *self* when done
    context.become(busy, discardOld = false)
}

def busy: Receive = {
  case Done =>
    unstashAll()
    context.unbecome()
  case _ =>
    stash()
}
```

Stashing away Messages

- Messages not handled by the current behavior get lost
- What if a later behaviour could potentially handle these?
- Mix in Stash and use stash and unstashAll to retain unhandled messages
- **Attention**: Akka will automatically use a dequeue based mailbox

Bounded and unbounded Stash

- `akka.actor.default-mailbox.stash-capacity` defines the capacity of the `stash`
- Its default is -1, i.e. the stash is unbounded
- Mix in `UnboundedStash` to enforce an unbounded stash

Exercise 018: Changing Actor Behavior

- In this exercise, we will demonstrate through the use of become and stash to modify actor
- Make sure you are in the `[018] become_stash` > project
- Run `man` to see the instructions

Typesafe