

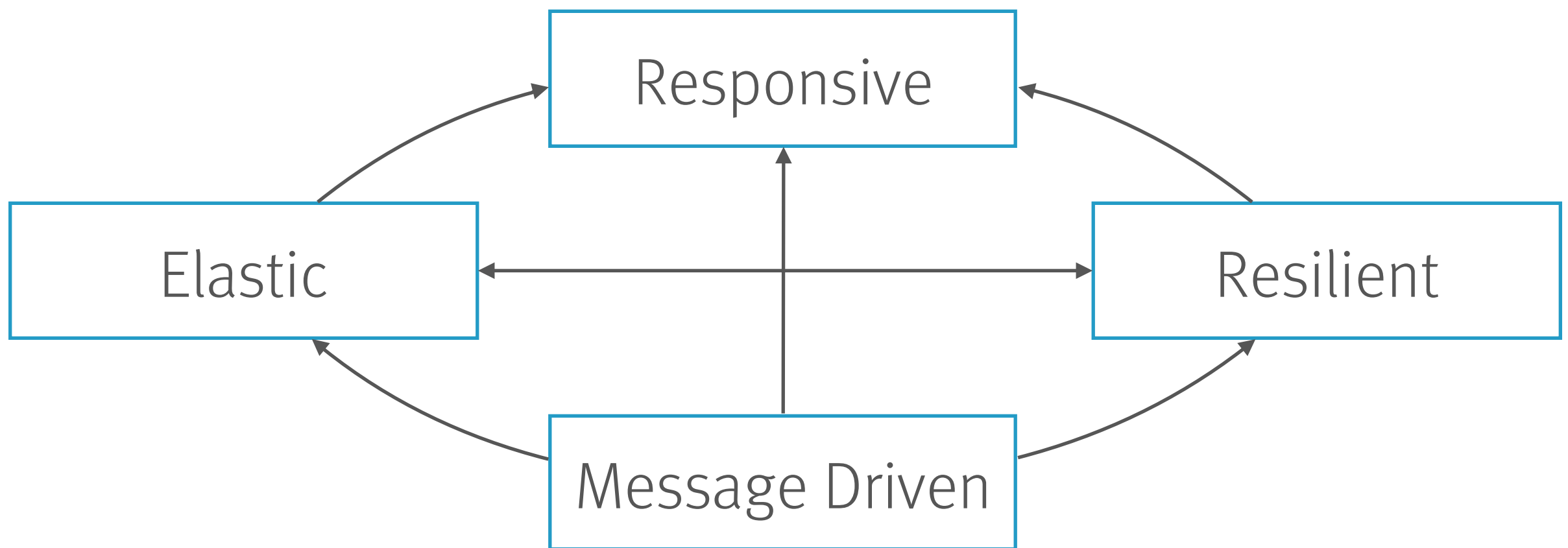
Reactive Play and Scala

From one System to a System of Systems

Tobias Neef | tobias.neef@innoq.com

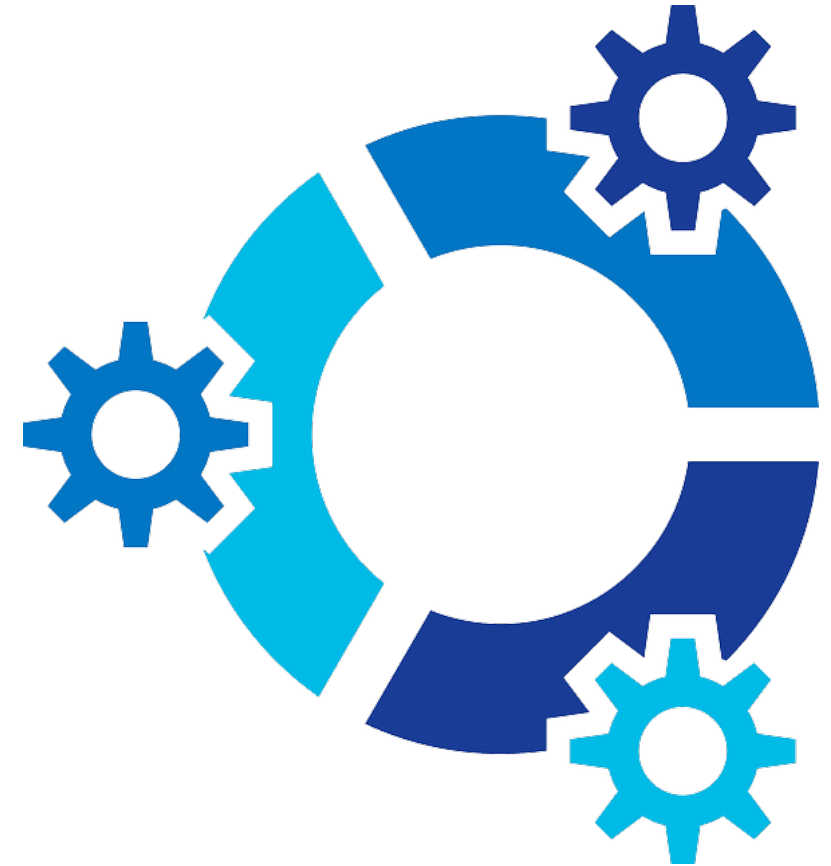
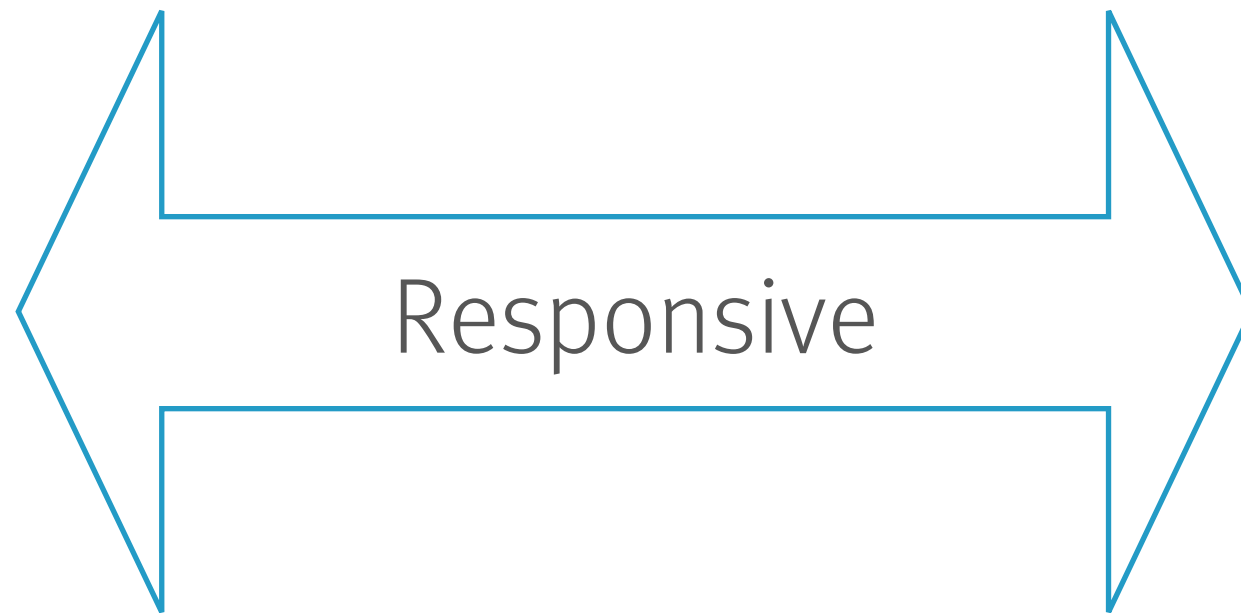


Properties of Reactive Systems





User



System

Elastic

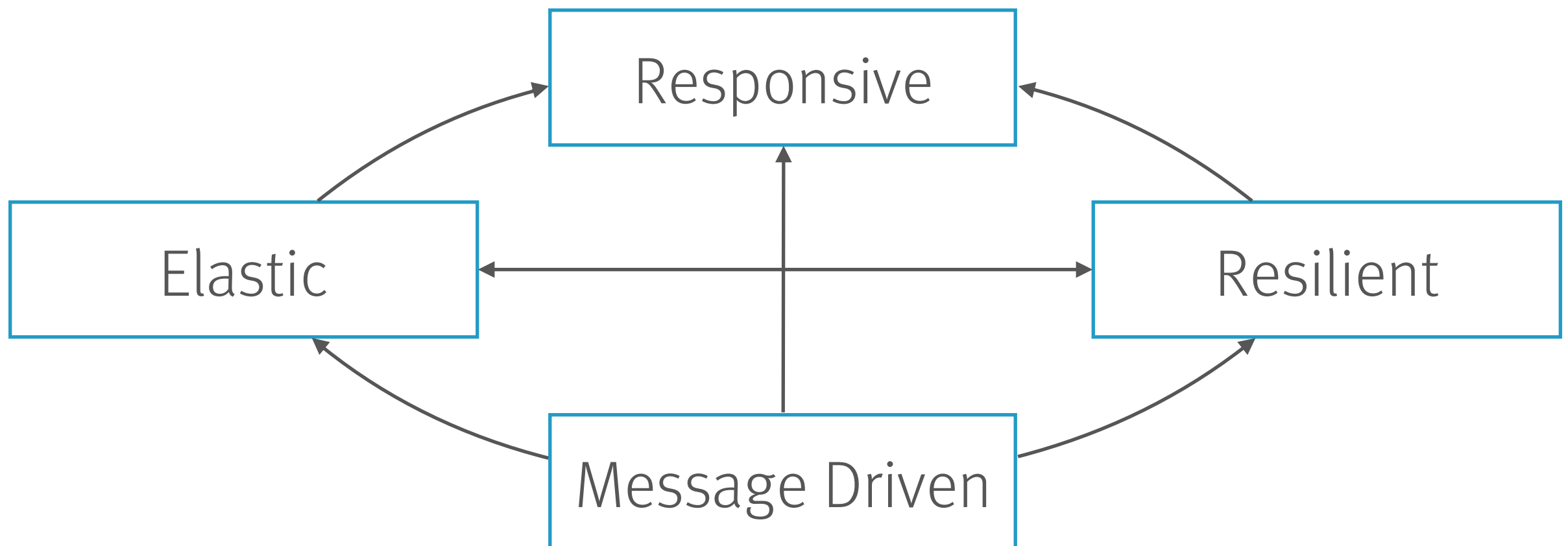
- › **The system** stays responsive under varying workload
- › ... reacts to changes in the input rate by increasing or decreasing the resources
- › ... no contention points or central bottlenecks

Resilient

- › **The system** stays responsive in the face of failure
- › Resilience is achieved by replication, containment, isolation and delegation

Message Driven

- › **Reactive Systems** rely on asynchronous message-passing
- › Establish a boundary between components
- › Ensure loose coupling, isolation, location transparency, and provides the means to delegate errors as messages



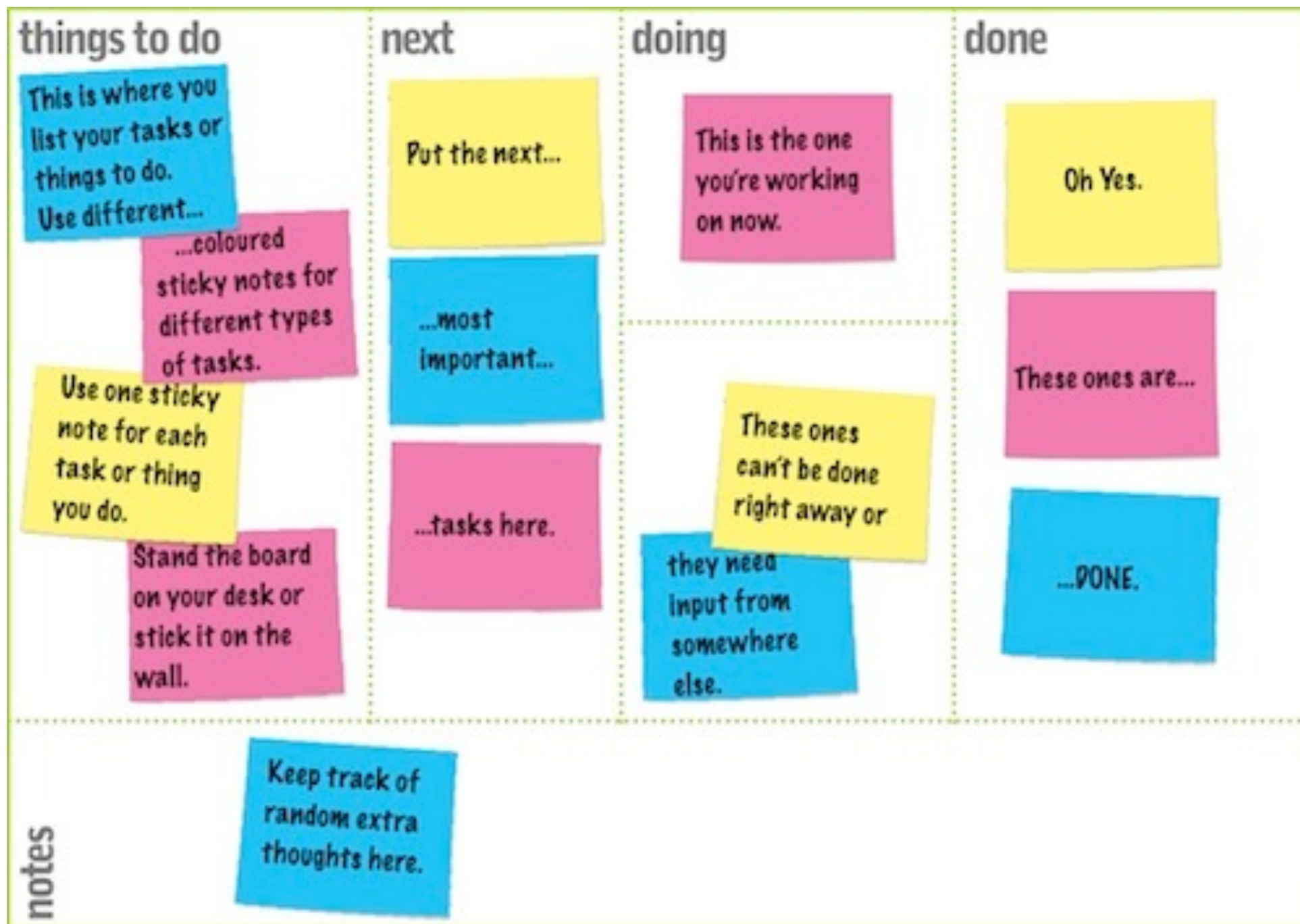
The story of a Play App

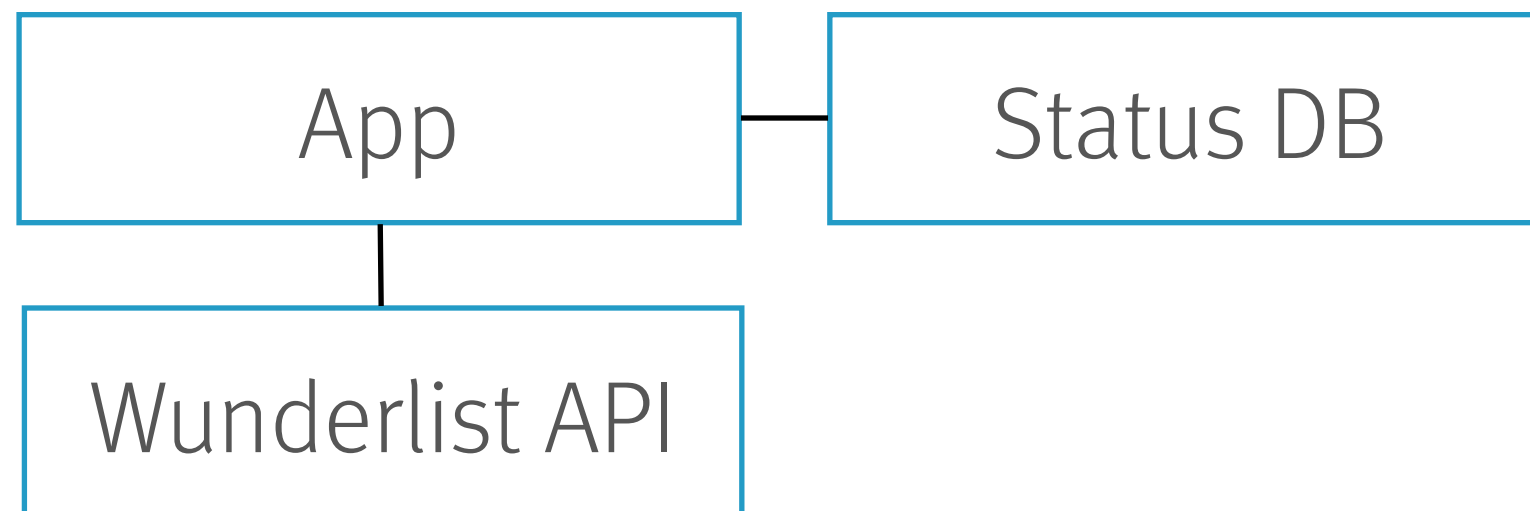
Wunderban

Wunderlist



kanbanfor1





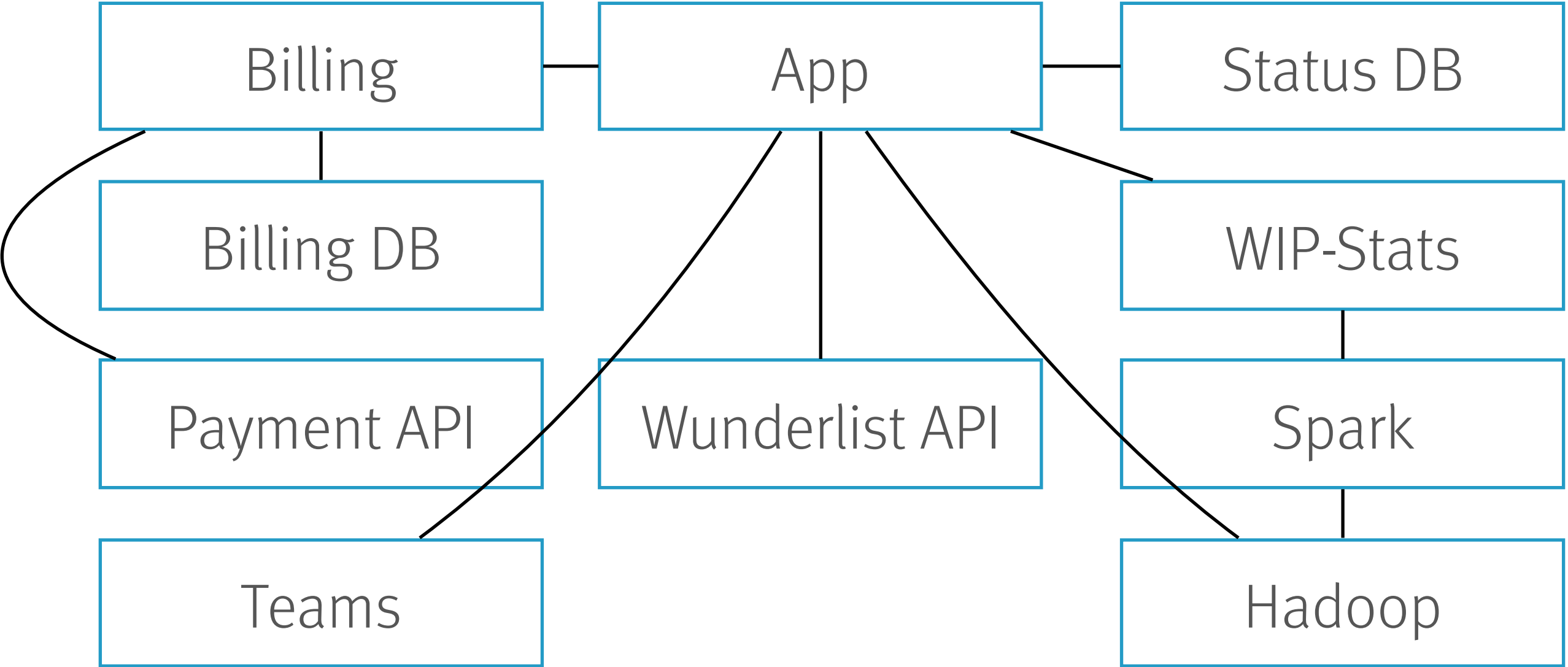
<https://github.com/tobnee/wunderban>

c-o-d-e

12 months later

Breaking the Monolith

<https://www.innoq.com/blog/st/2013/10/on-monoliths/>



„Large systems are composed of smaller ones and therefore depend on the Reactive properties of their constituents. “

— reactivemanifesto.org

Revolution vs. Evolution

The Play Stack



HTTP-Stack | Asset-Mgmt | Templates | DB-Integration | ...



Play's-App-Engine | Fault-Tolerance | Location-Transparent

Akka Actors

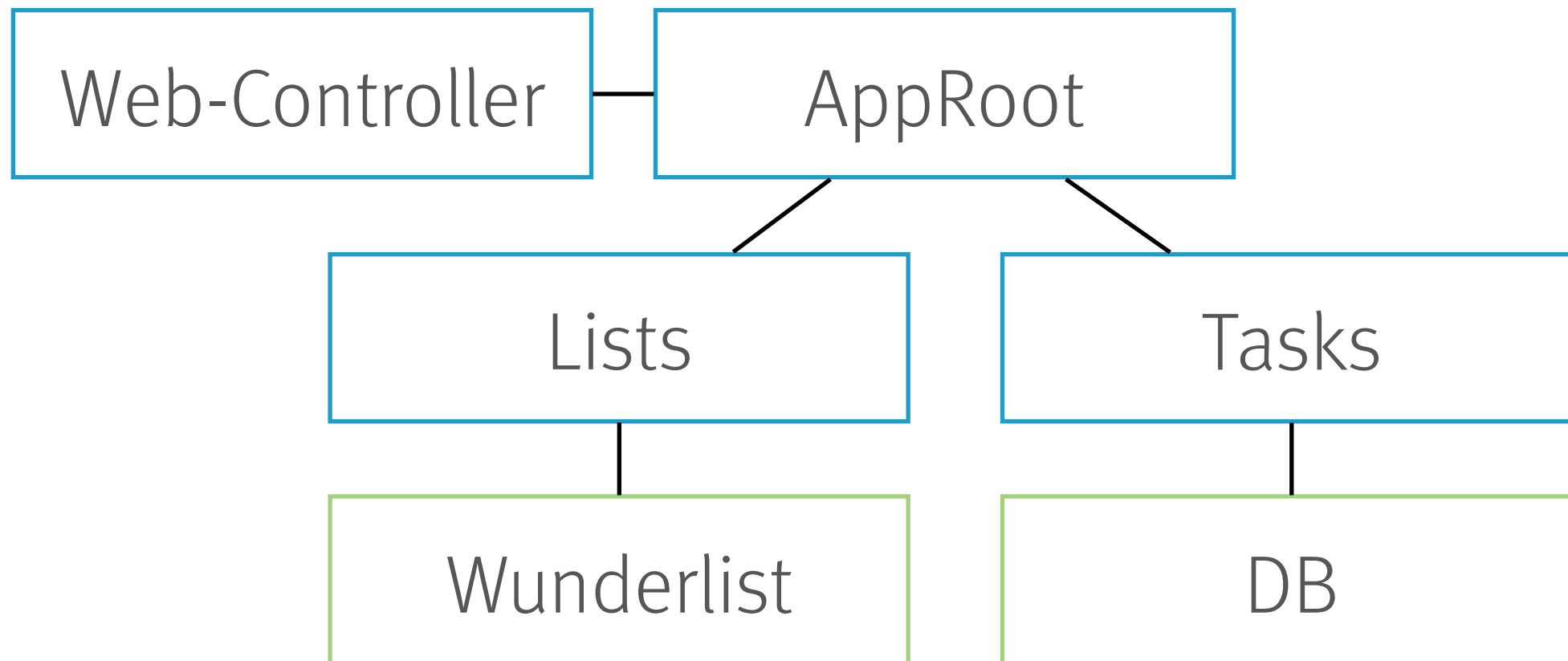
- › Actors are message driven and async
- › Support resilience
- › Support scale up and scale out mechanisms in one model

Step 1



Migrate Logic to Akka

A basic actor model



```
// actor protocol  
case class GetTasks(id: Long)  
case class TaskNotFound(id: Long)  
case class TaskResult(id: Long,  
                      tasks: List[Task])
```

```
// controller
def tasks(id: Long) = Action.async {
  val wunderlist : ActorRef = ...
  (wunderlist ? GetTasks(id)).map {
    case TaskNotFound(id) => NotFound
    case TaskResult(id, tasks) => Ok
  }
}
```

```
// simplified actor
class ListsActor extends Actor {

  def receive = {
    case GetTasks(id) =>
      val tasks = lookupTasks(id)
      val result =
        if(tasks.isEmpty) TaskNotFound(id)
        else TaskResult(id, tasks)
      sender() ! result
  }
}
```

**„Location Transparency is often mistaken for
'transparent distributed computing', while it is
actually the opposite“**

— reactivemanifesto.org

Step 2



Distribute App

Step 3

See the App
crashing ;)

Step 4



Make resilient

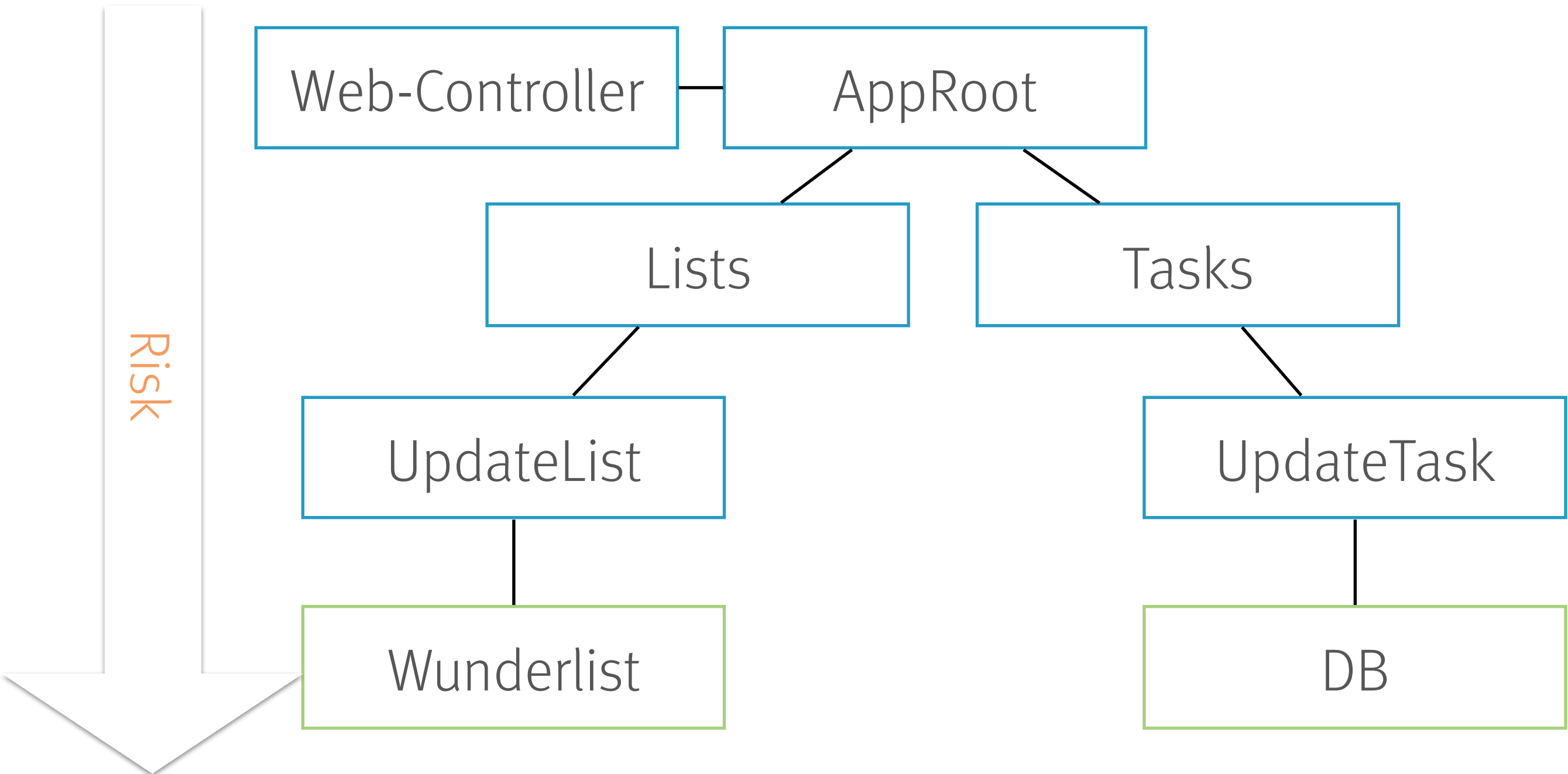
Apply stability patterns

- › Bulkheads
- › Circuit Breakers
- › Handshaking

Building Bulkheads with Actor Supervision



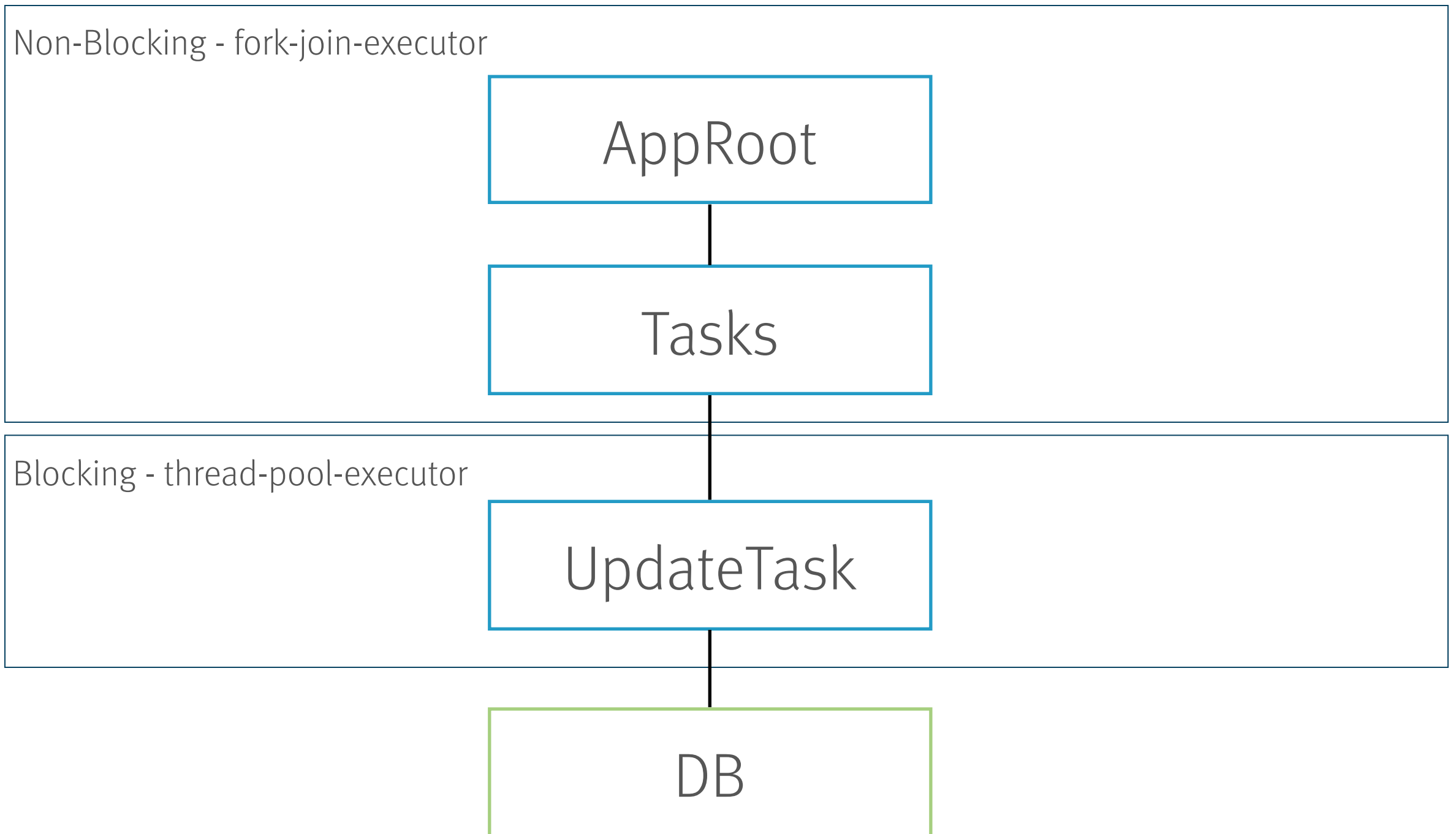
A wunderban actor model



Building Bulkheads with Actor Dispatchers



Resource Management

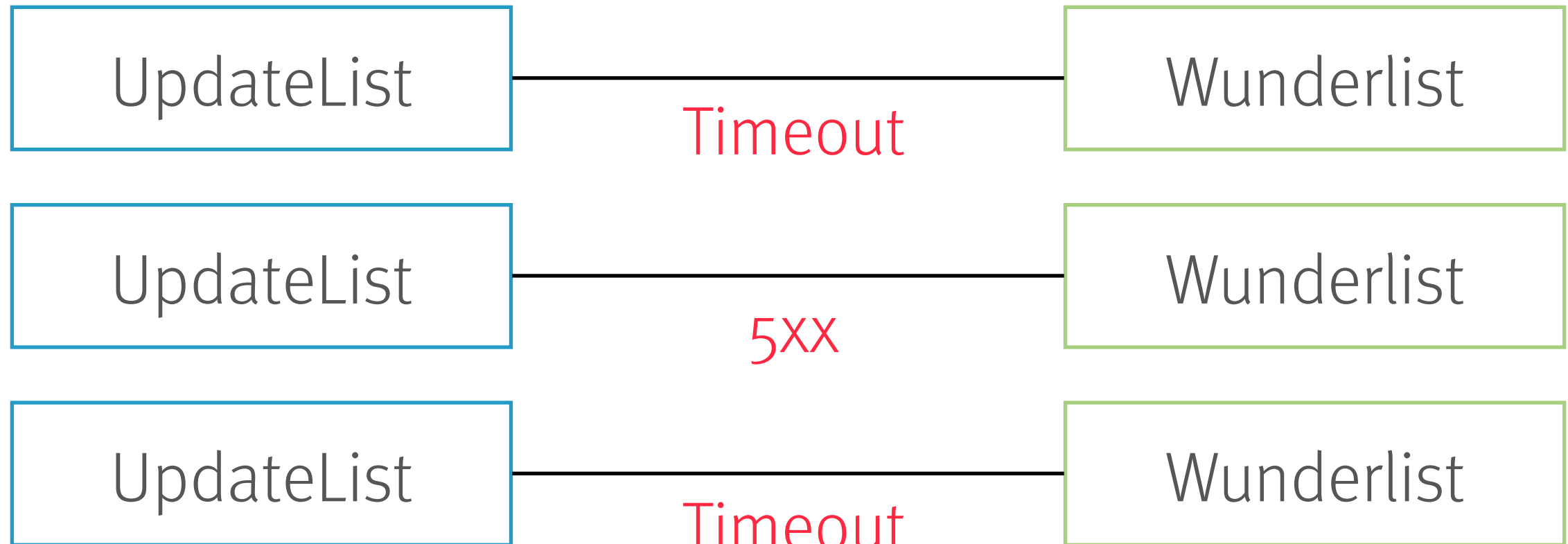


Building Circuit Breakers with Akka



System Isolation

Closed



Open



stability patterns —>
resilience

Step 4

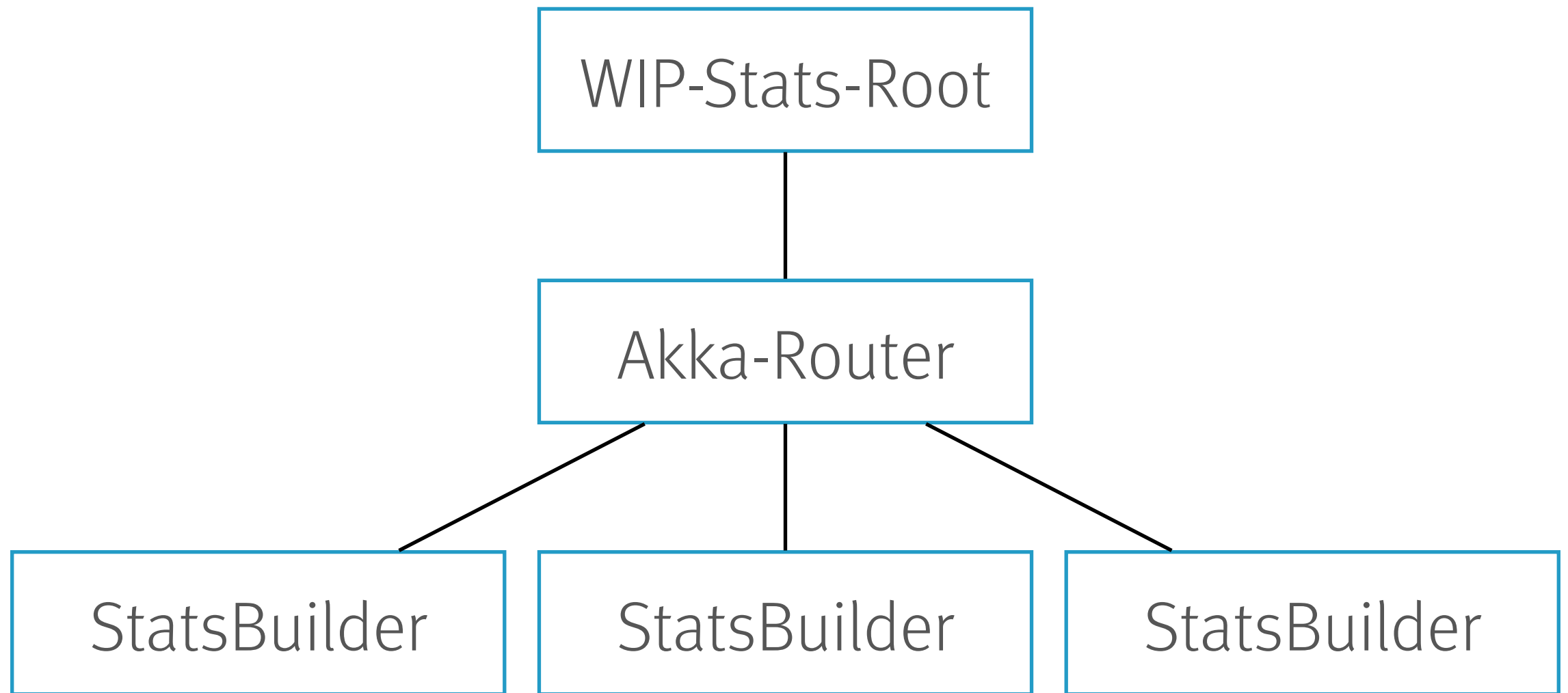


Scale up with Routers

WIP-Stats-Root

StatsBuilder





Further Scaling Options

- › Scale out on a system level HTTP/LB
- › Scale out using Akka (Remoting, Cluster)

Lessons Learned

- › The path from a simple, monolithic Play App to a distributed, reactive app often involves Akka features
- › Akka gives you powers to apply stability patterns with its standard tools
- › Akka gives you ways to scale but it's not a silver bullet

Further Topics

- › Reactive Streams / Akka Streams
- › Akka Persistence
- › Akka Cluster

Thank you!
Questions?
Comments?



Tobias Neef |  tobnee
tobias.neef@innoq.com



Code Samples



```
val defaultDecider: Decider = {  
  case _: ActorInitializationException ⇒ Stop  
  case _: ActorKilledException ⇒ Stop  
  case _: DeathPactException ⇒ Stop  
  case _: Exception ⇒ Restart  
}
```

```
val defaultStrategy: SupervisorStrategy = {  
  OneForOneStrategy()(defaultDecider)  
}
```

Resource Allocation by Configuration



```
fix-thread-pool-dispatcher {  
  type = Dispatcher  
  executor = "thread-pool-executor"  
  thread-pool-executor {  
    core-pool-size-min = 2  
    core-pool-size-max = 10  
  }  
}  
  
akka.actor.deployment {  
  /approot/tasks/udpdatetask {  
    dispatcher = my-dispatcher  
  }  
}  
  
val myActor = context.actorOf(Props[DbActor],  
  "udpdatetask")
```

```
val breaker =  
    new CircuitBreaker(actorSystem.scheduler,  
        maxFailures = config.maxFailures,  
        callTimeout = config.callTimeout,  
        resetTimeout = config.resetTimeout)  
        .onOpen(notifyMeOnOpen())  
        .onClose(notifyMeOnClose())  
        .onHalfOpen(notifyMeOnHalfOpen())  
  
def execute[T](call: Future[T]): Future[T] = {  
    breaker.withCircuitBreaker(call)  
}
```

```
akka.actor.deployment {  
  /wipstats/statsbuilder {  
    router = round-robin-pool  
    nr-of-instances = 3  
  }  
}
```

```
val statsRouter = context.actorOf(  
  FromConfig.props(Props[StatsBuilder]),  
  "statsbuilder")
```