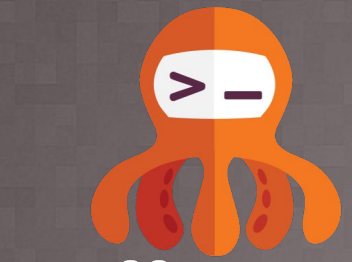


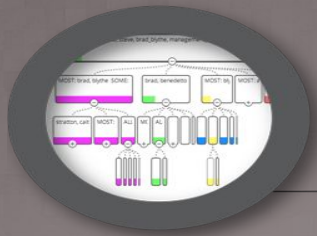
TIKAL

Fullstack as a service

The journey to reactiveness

Chaim Turkel

Chaim Turkel



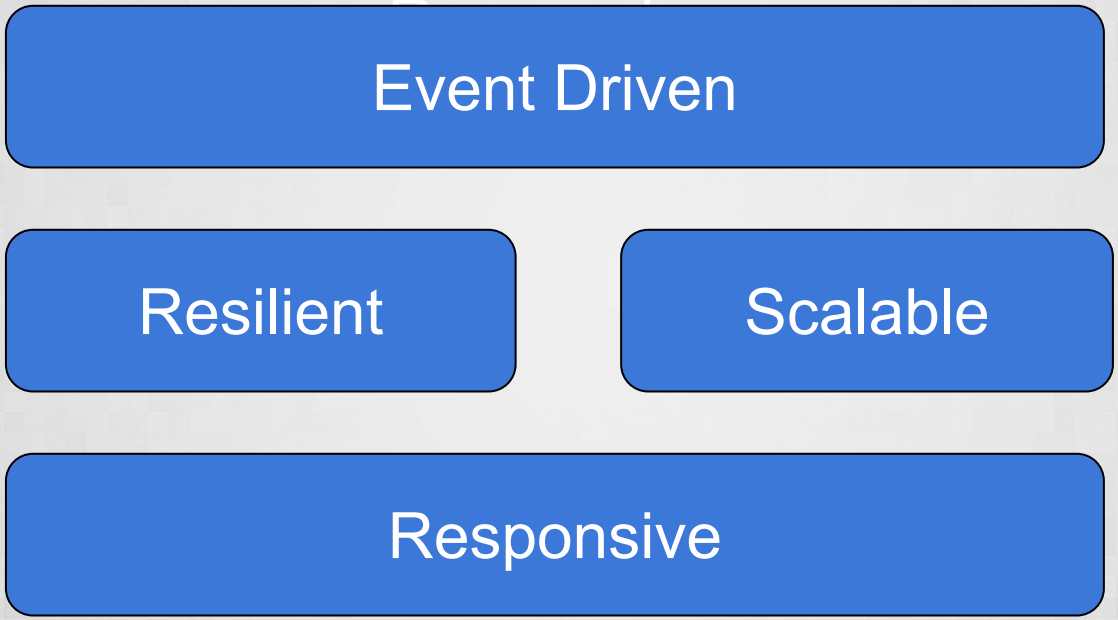
verview

Fullstack Developers

The journey to reactive ...

- Four Reactive Cornerstones - Recap
- Vehicle Fleet Tracker - overview of application
- Reactive refactoring
  - Breakdown of monolithic app to microservices
  - Salesforce CRM update Use Case
    (Emphasising Responsive (readable asynchronous programing))
    Salesforce multithreading - Scala Futures
  - Consolidate UI request (performance degradation due to microservices)
    - Observable patterns with option to abort - RXJava
  - Analytics performance boost, solution for thread exceptions
    - Resilience / Message driven - Akka

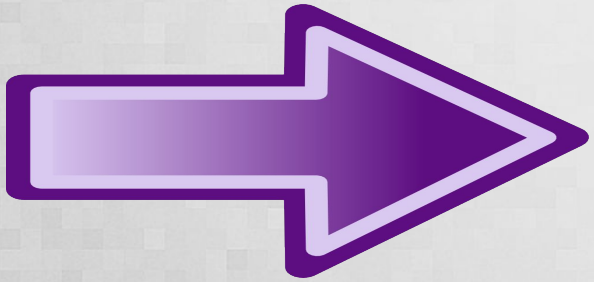
Four Reactive Cornerstones



New Tools for a New Era

As a result we should build system that

react to messages - **`Message Driven`**
react to high load - **`Scalable`**
react to failures - **`Resilient`**
react to users - **`Responsive`**

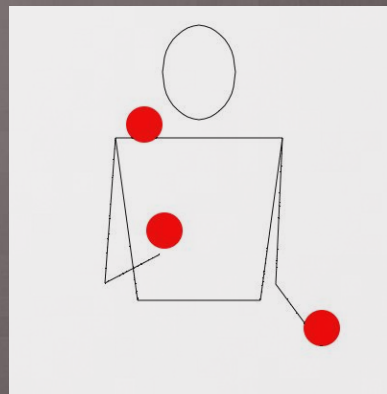


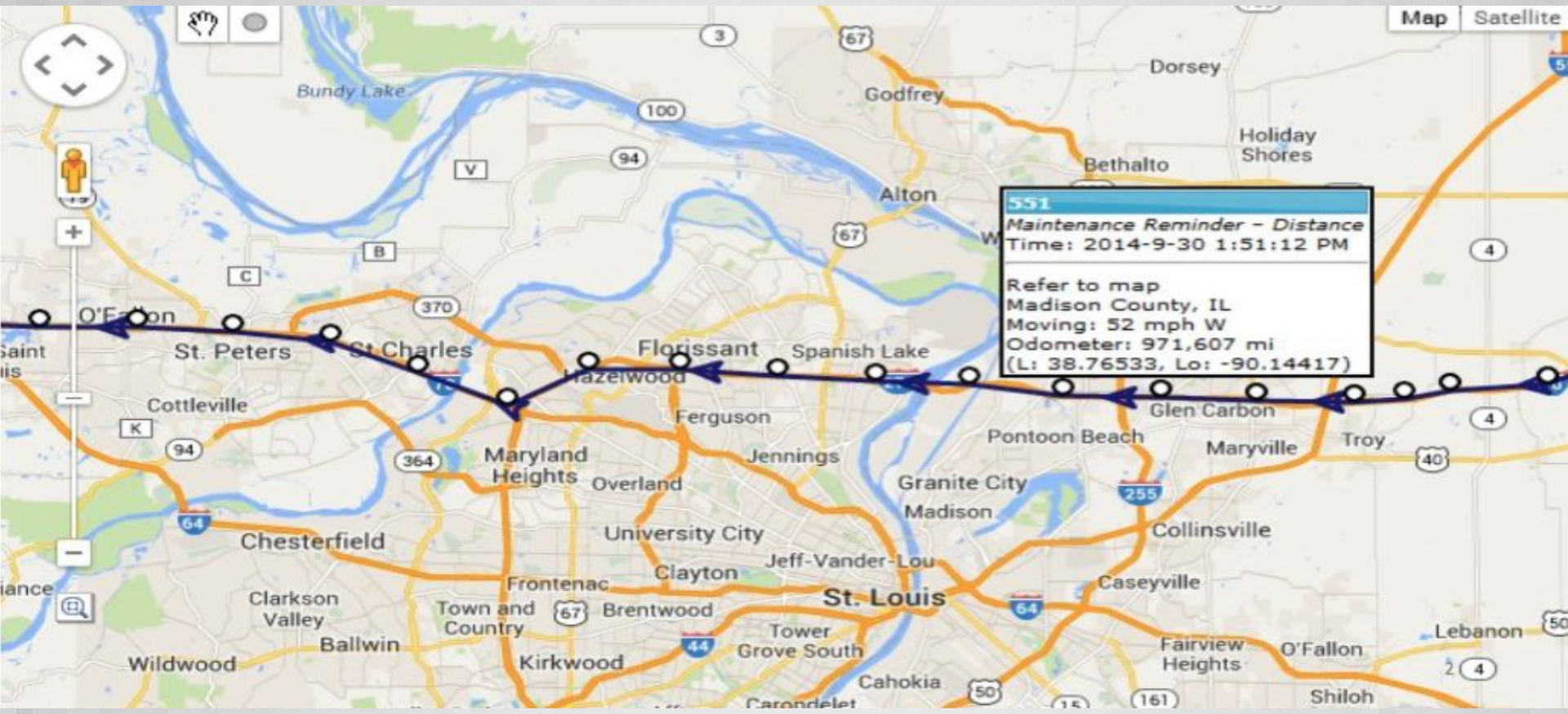
Reactive Applications

Fullstack Developers

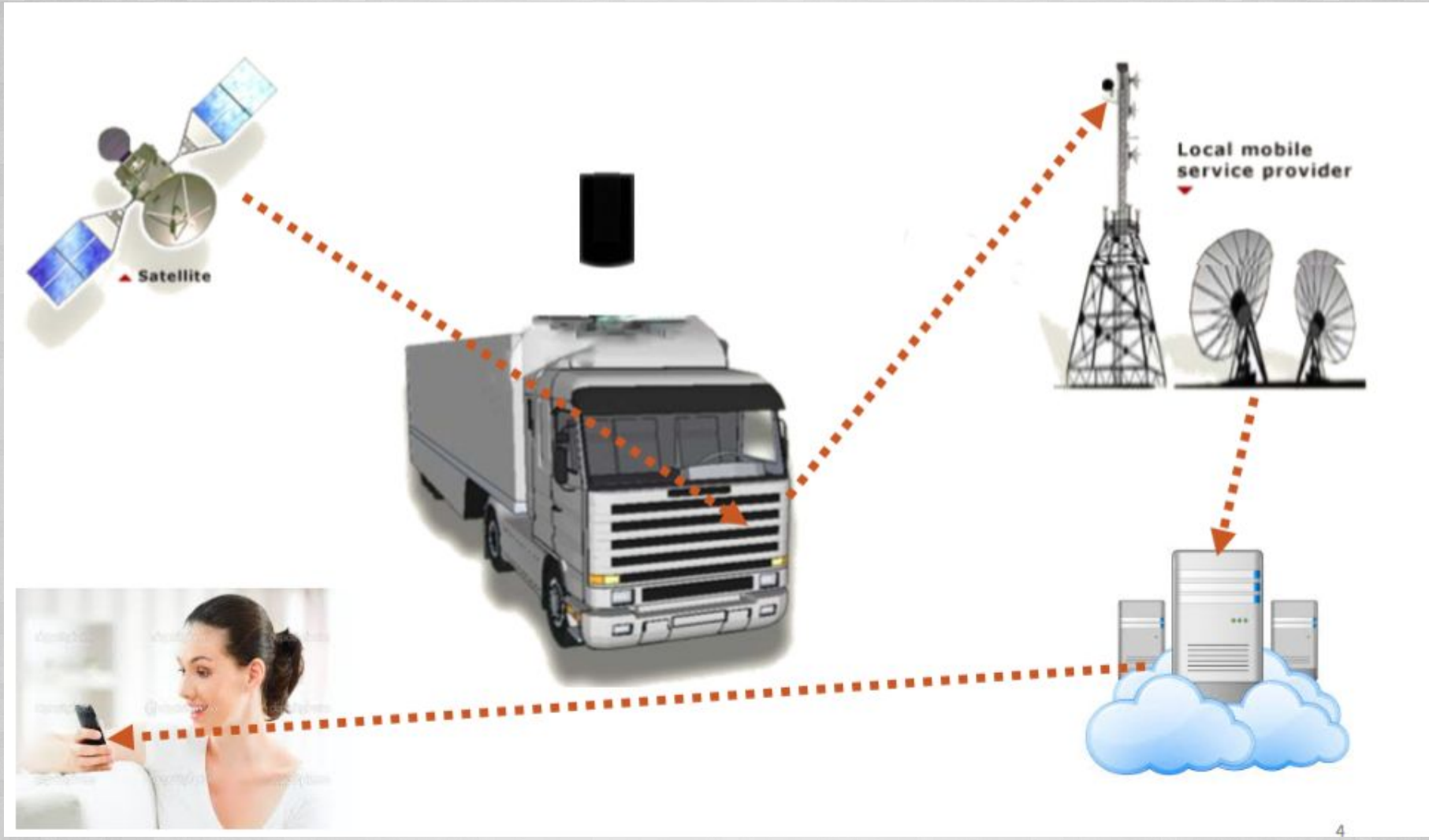
Vehicle Fleet Tracker

Reactive concepts

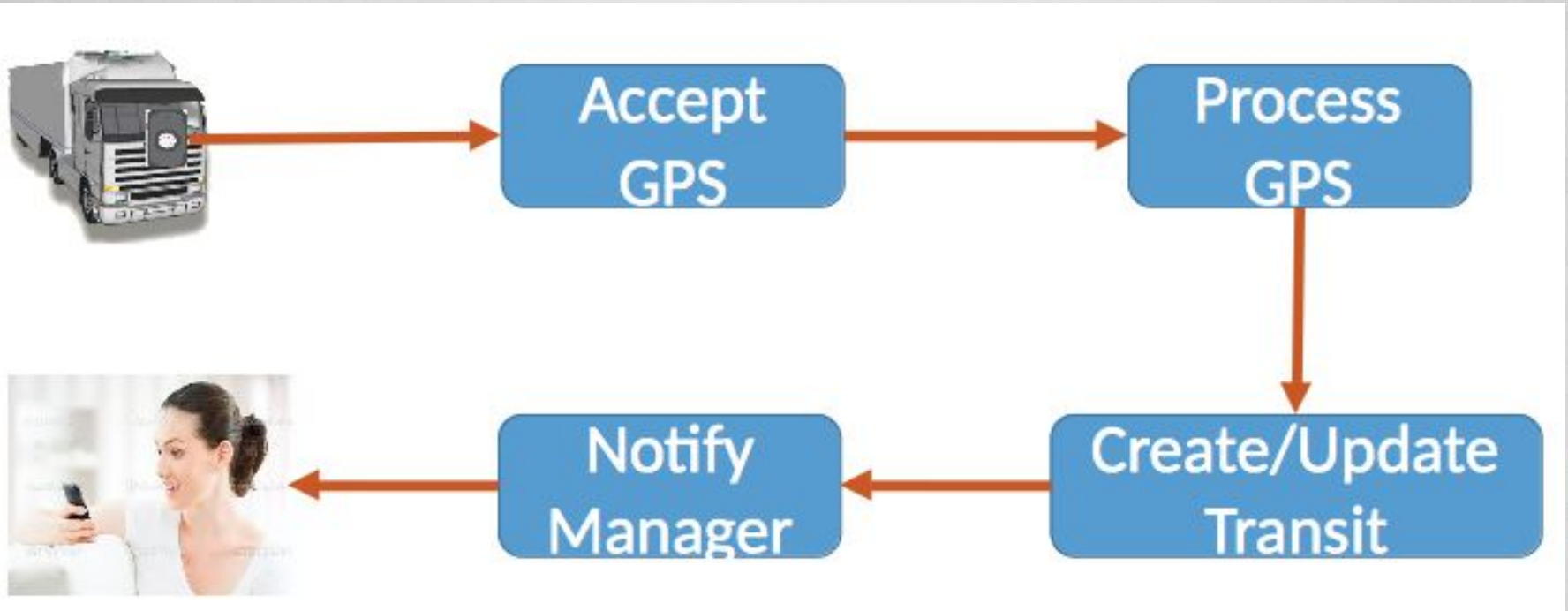




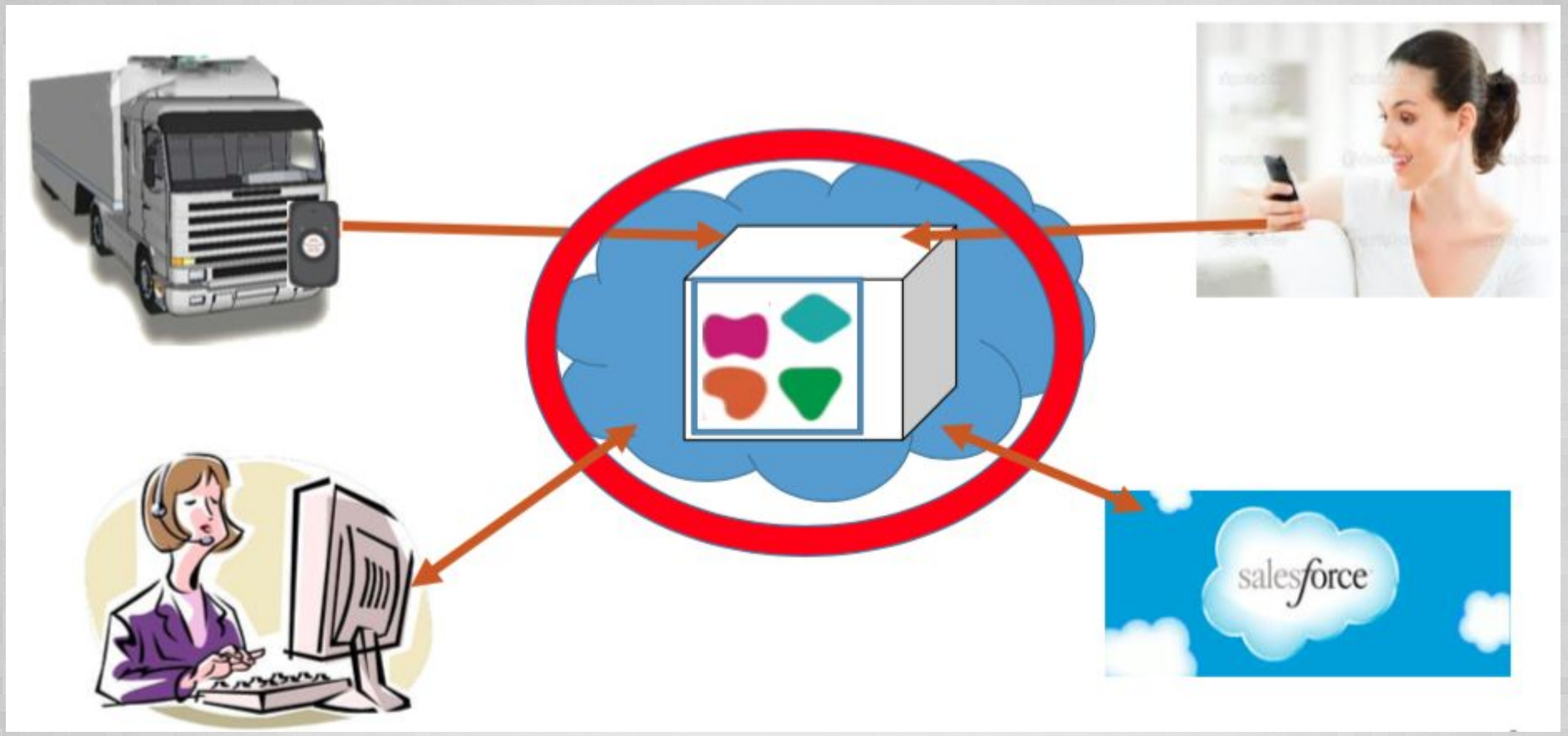
Vehicle Fleet Tracker



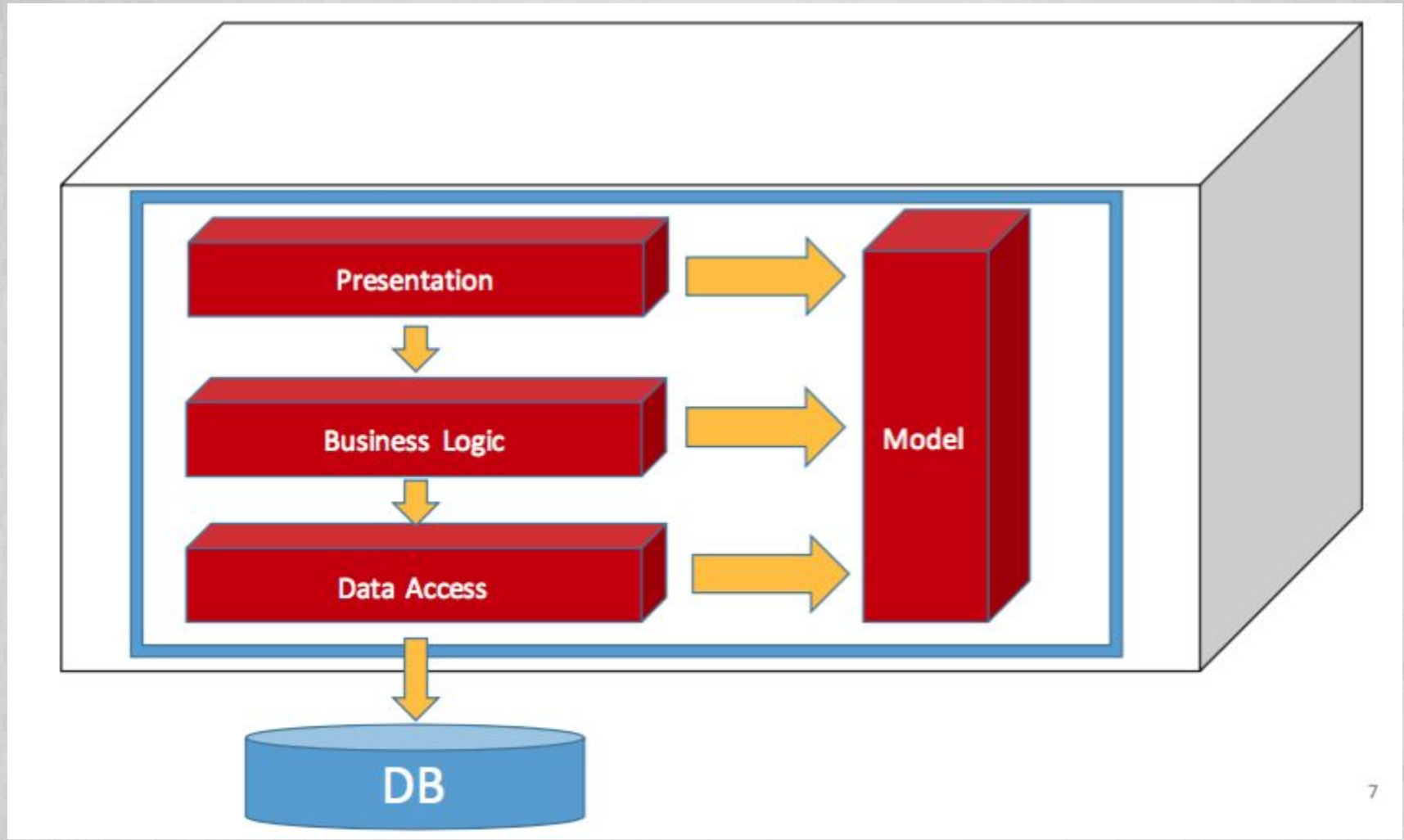
Vehicle Fleet Tracker



Vehicle Fleet Tracker



Single Monolithic Spring Application



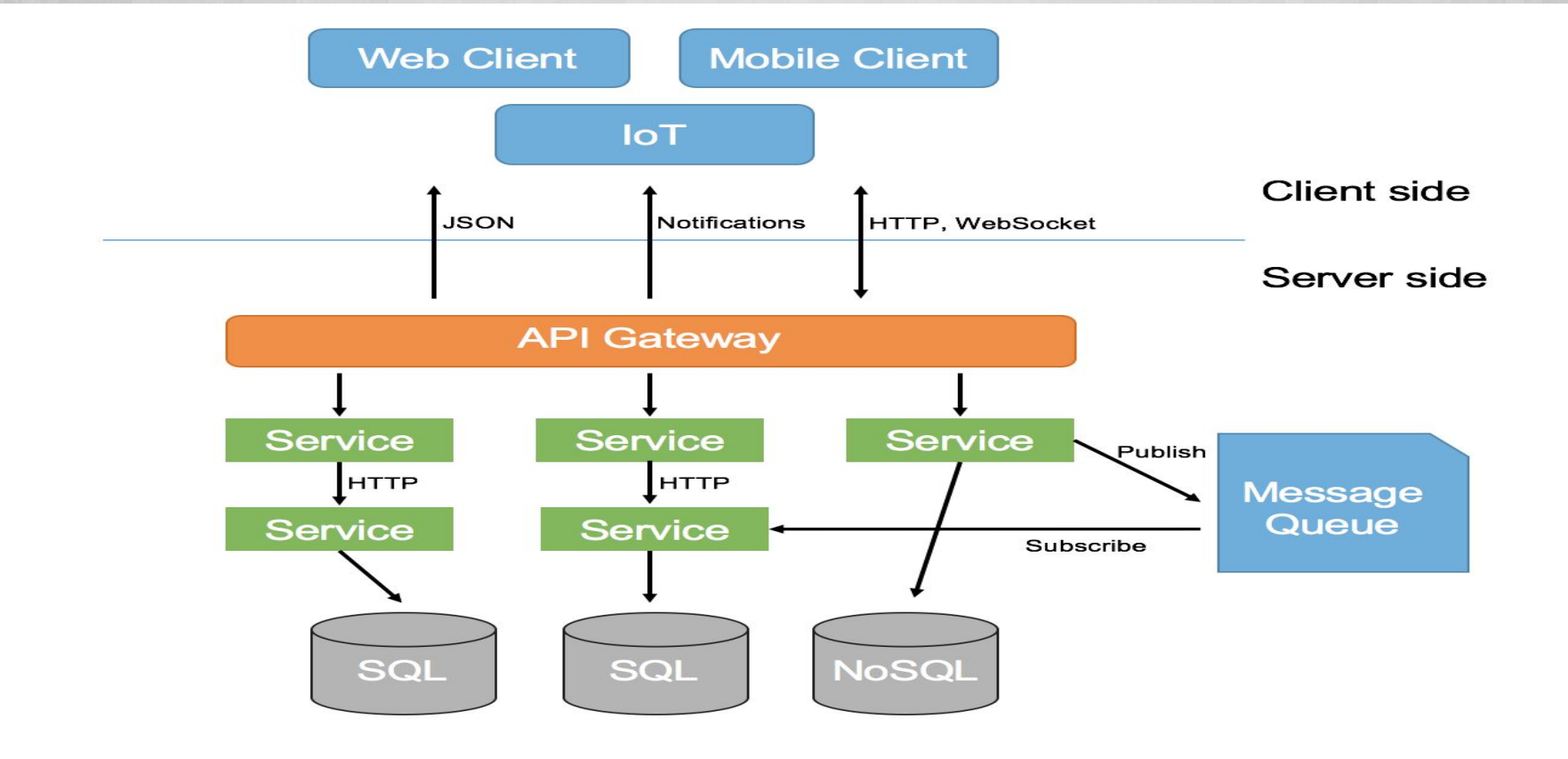
Single Monolithic Spring Application

Fullstack
Developers

What solutions are out there

Fullstack
Developers

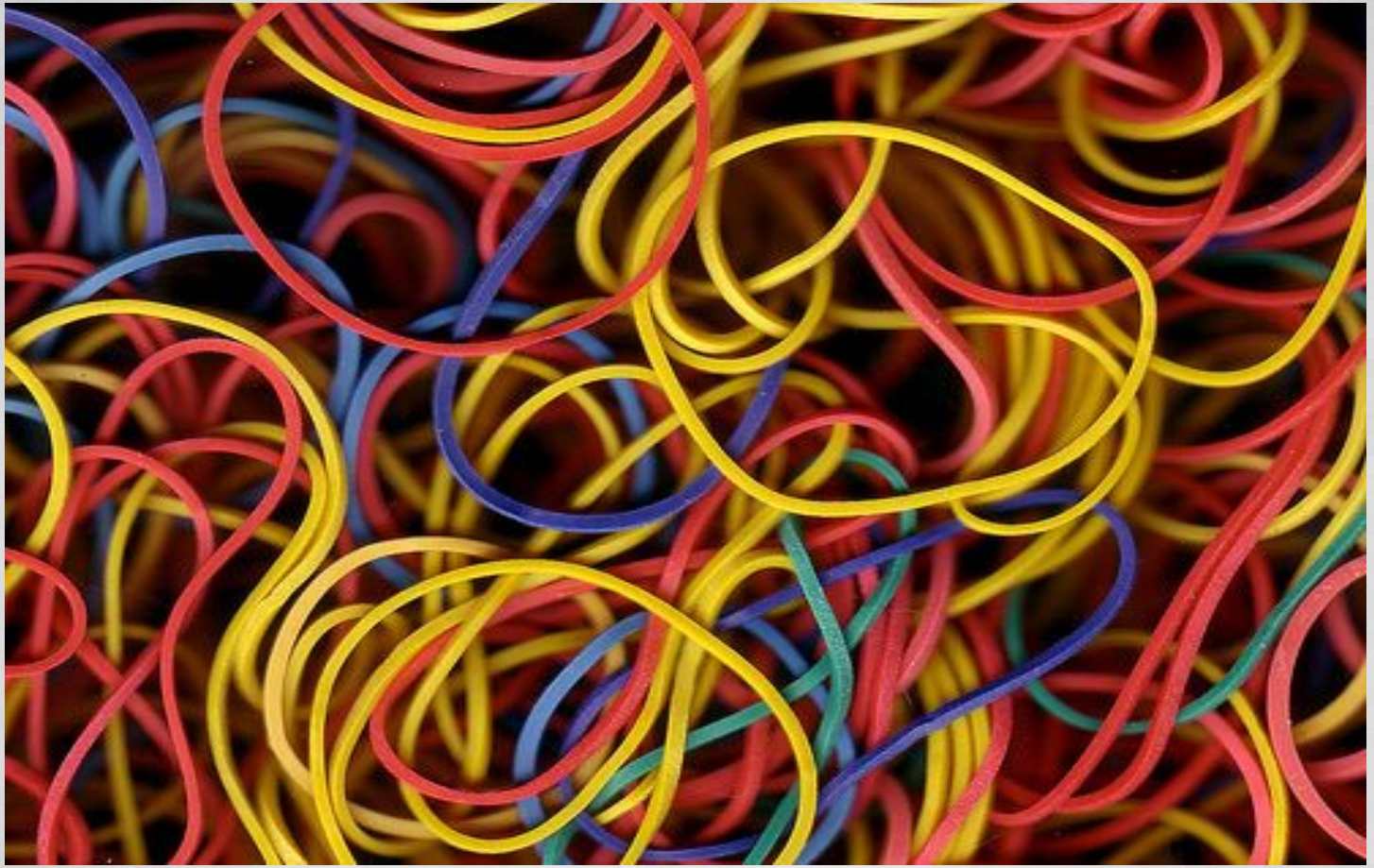
Message Driven
Scaleable



Microservices

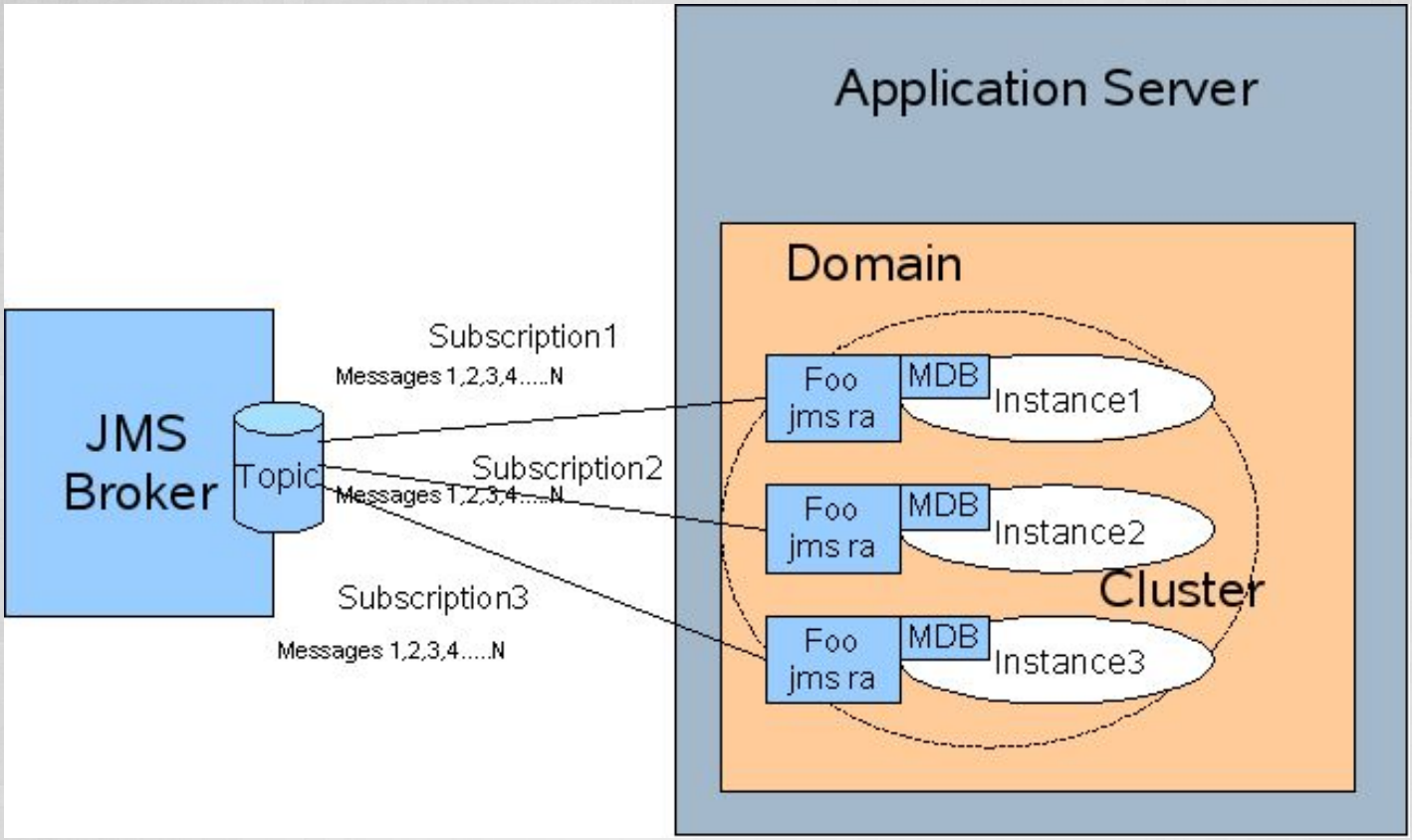
Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

Responsiveness - each service is dedicated to a single task and does this in the best possible manner. Since it is dedicated to one task, the server will be able to respond to more client request.



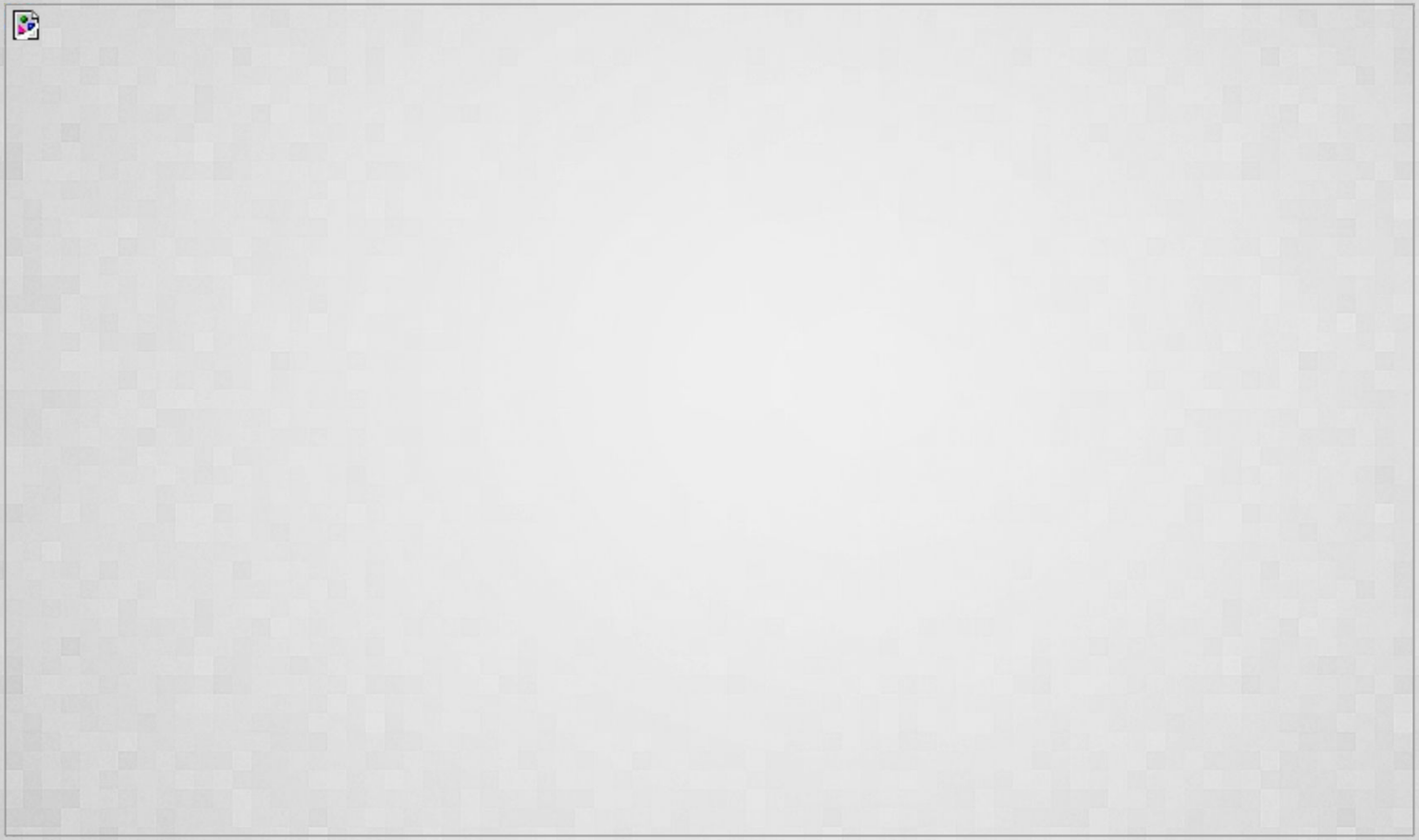
Elasticity - We can scale both up/down out/in according to demand.
We can add more nodes per service that are overloaded or remove nodes for services that are under utilized.

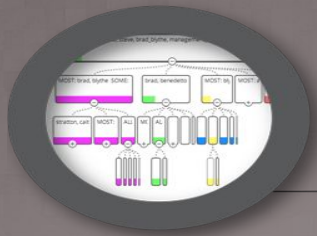
Resilience - Since each service is isolated, we can have a cluster of services behind a loadbalancer. So in the case of failure other instances can pick up. In case of failure we can implement the “Circuit Breaker” Pattern. Service Discovery is needed so that we can handle which machines are up and available.



Message Driven - protocol between microservices should be done via a message broker like kafka. Messaging allows for non-blocking handling of responses. Established boundaries between services (in-proc and out of proc) which allows for loose coupling.

Microservices





verview

Fullstack Developers

The journey to reactive ...

- Four Reactive Cornerstones - Recap
- Vehicle Fleet Tracker - overview of application
- Reactive refactoring
  - Breakdown of monolithic app to microservices
  - Salesforce CRM update Use Case
    (Emphasising Responsive (readable asynchronous programing))
    Salesforce multithreading - Scala Futures
  - Consolidate UI request (performance degradation due to microservices)
    - Observable patterns with option to abort - RXJava
  - Analytics performance boost, solution for thread exceptions
    - Resilience / Message driven - Akka

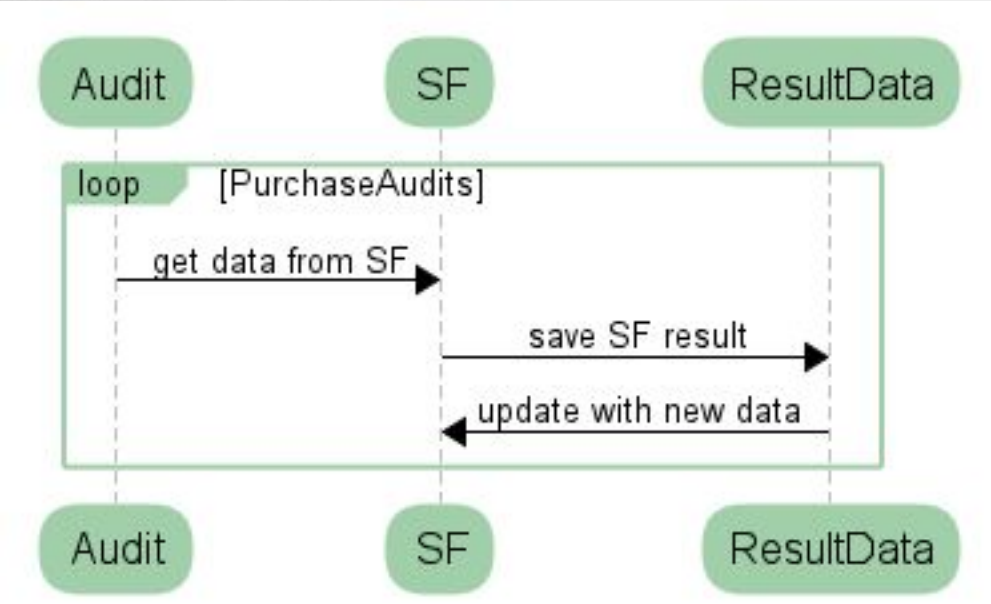
Fullstack
Developers

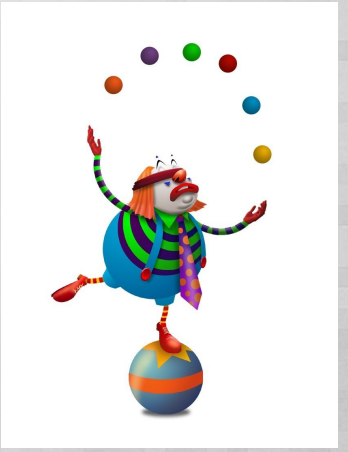
Responsive

SalesForce Update

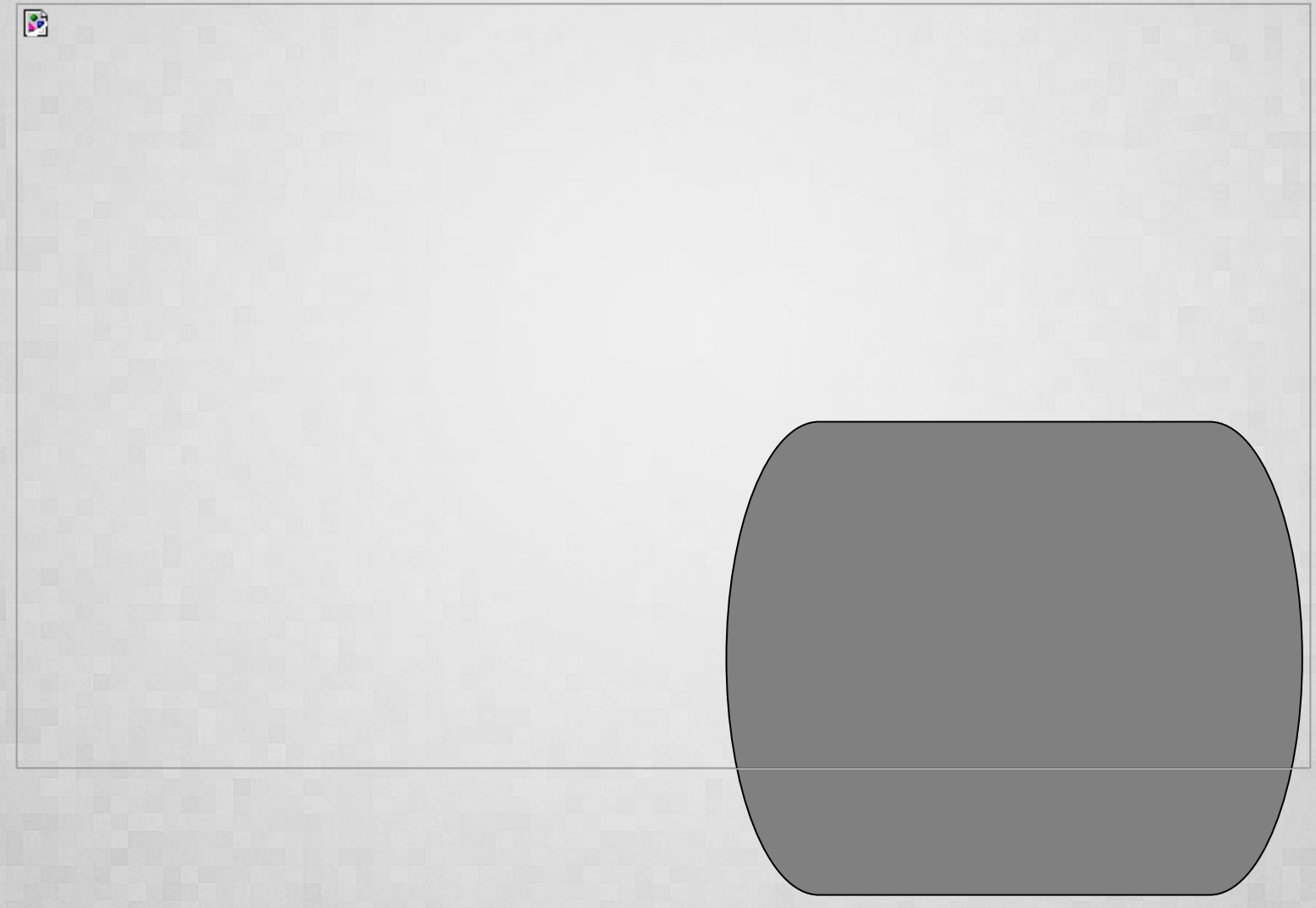
Batch process of auditing sent to SalesForce

On a schedule we need to update the SalesForce with purchase account information.





Network Requests



Scala Future - Simple

```
import scala.util.{Success, Failure}

val f: Future[List[String]] = Future {
 session.getRecentPosts
}

f onComplete {
 case Success(posts) => for (post <- posts) println(post)
 case Failure(t) => println("An error has occured: " + t.getMessage)
}
```

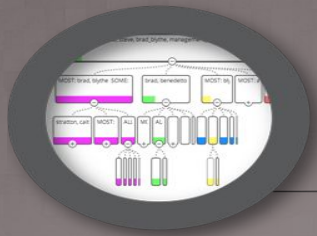
Two streams one for success and one for error (stdout, stderr)

Scala Future - Composable Multithread

```
import scala.util.{Success, Failure}

val f: Future[List[String]] = Future {
 //session.getRecentPosts
 List("one", "two", "three")
}

val calc : Future[List[Int]] = f.map(list => list.map(_.length))

calc onComplete {
 case Success(posts) => for (postLength <- posts) println(postLength)
 case Failure(t) => println("An error has occured: " + t.getMessage)
}
```

Fleet Tracker - Future

```
case class RequestWrapper(audit: PurchaseAudit, sfData: SFData)
case class ResultWrapper(audit: PurchaseAudit, sfResult: SFResult)

def getAuditDataFromSF(audit: PurchaseAudit): SFData = new SFData
def sendUpdateToSF(audit: PurchaseAudit, sfData: SFData): SFResult
def saveToDB(audit: PurchaseAudit, sfResult: SFResult): Unit

def sendToSF(purchaseAudit: List[PurchaseAudit]): Unit = {
 val sfUpdateList: List[Future[ResultWrapper]] =
  for (audit: PurchaseAudit <- purchaseAudit)
    yield {
      val map: Future[ResultWrapper] = Future {
        RequestWrapper(audit, getAuditDataFromSF(audit))
      }
        .map(wrapper => {
          ResultWrapper(wrapper.audit, sendUpdateToSF(wrapper.audit, wrapper.sfData))
        })
      map
    }

 val futureList: Future[List[ResultWrapper]] = Future.sequence(sfUpdateList)
 futureList onComplete {
  case Success(wrapperList) => {
    for (wrapper <- wrapperList) {
      saveToDB(wrapper.audit, wrapper.sfResult)
    }
  }
  case Failure(error) => {
    log.error(error.getMessage)
  }
 }
```



verview

Fullstack Developers

The journey to reactive ...

- Four Reactive Cornerstones - Recap
- Vehicle Fleet Tracker - overview of application
- Reactive refactoring
  - Breakdown of monolithic app to microservices
  - Salesforce CRM update Use Case
    (Emphasising Responsive (readable asynchronous programing))
    Salesforce multithreading - Scala Futures
  - Consolidate UI request (performance degradation due to microservices)
    - Observable patterns with option to abort - RXJava
  - Analytics performance boost, solution for thread exceptions
    - Resilience / Message driven - Akka

UI Performance

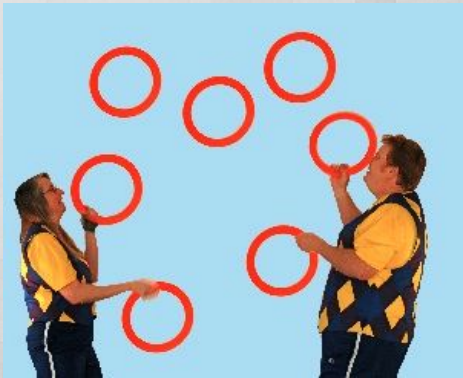
Moving to microservices - causes the client to send multiple requests to multiple servers

Mobile devices bring new challenges:

- Devices performance are not the same as desktops
- Mobile networks are slow

Display information:

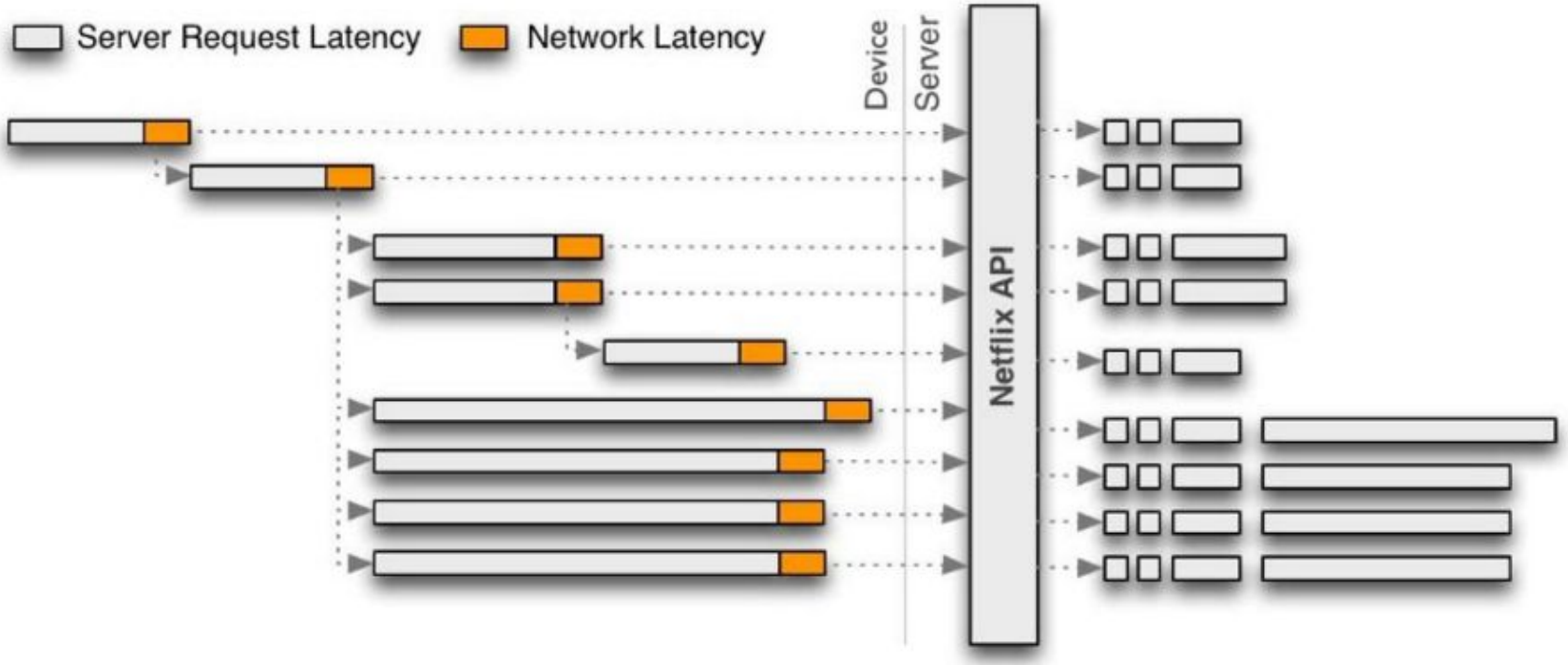
- Display Truck GPS points on a map (GPS Service)
- Display Trucks Availability (Vehicle Management)
- Display Tucks Health Status & Metrics (Analytics)



Network Requests

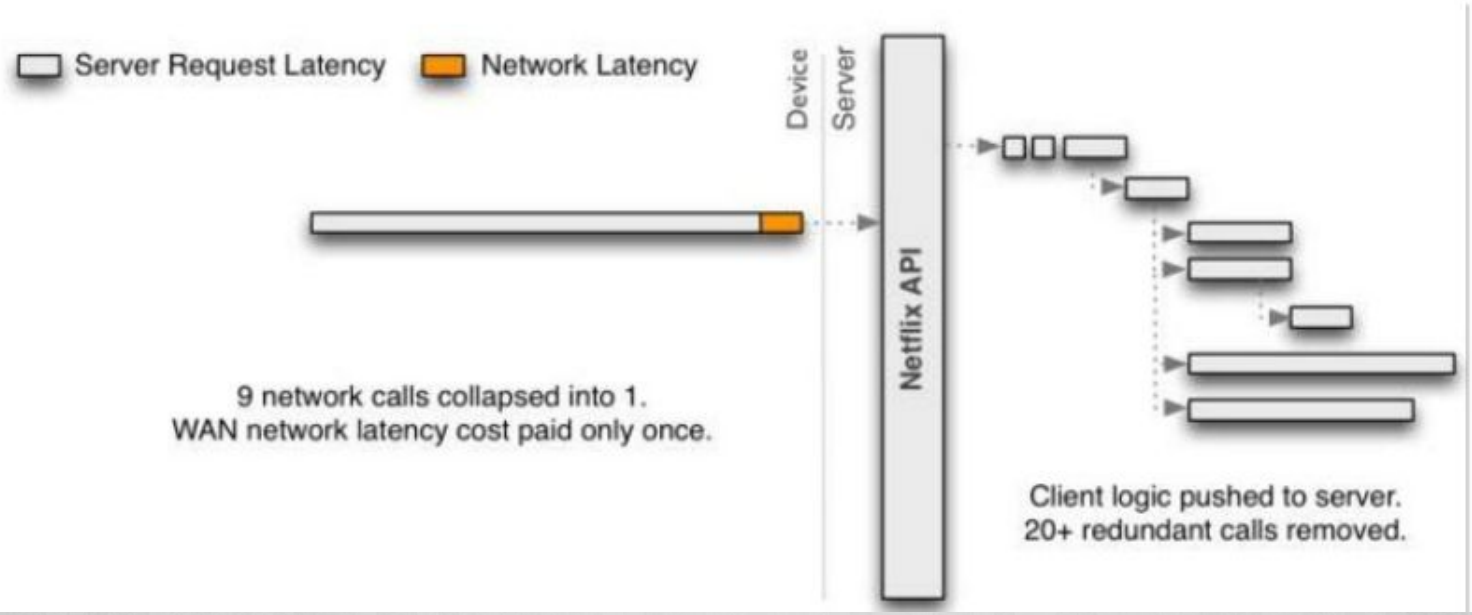
Network Requests

- How to reducing network chattiness?
  - Granular API.
  - client applications to make multiple calls that need to be assembled in order to render a single user experience



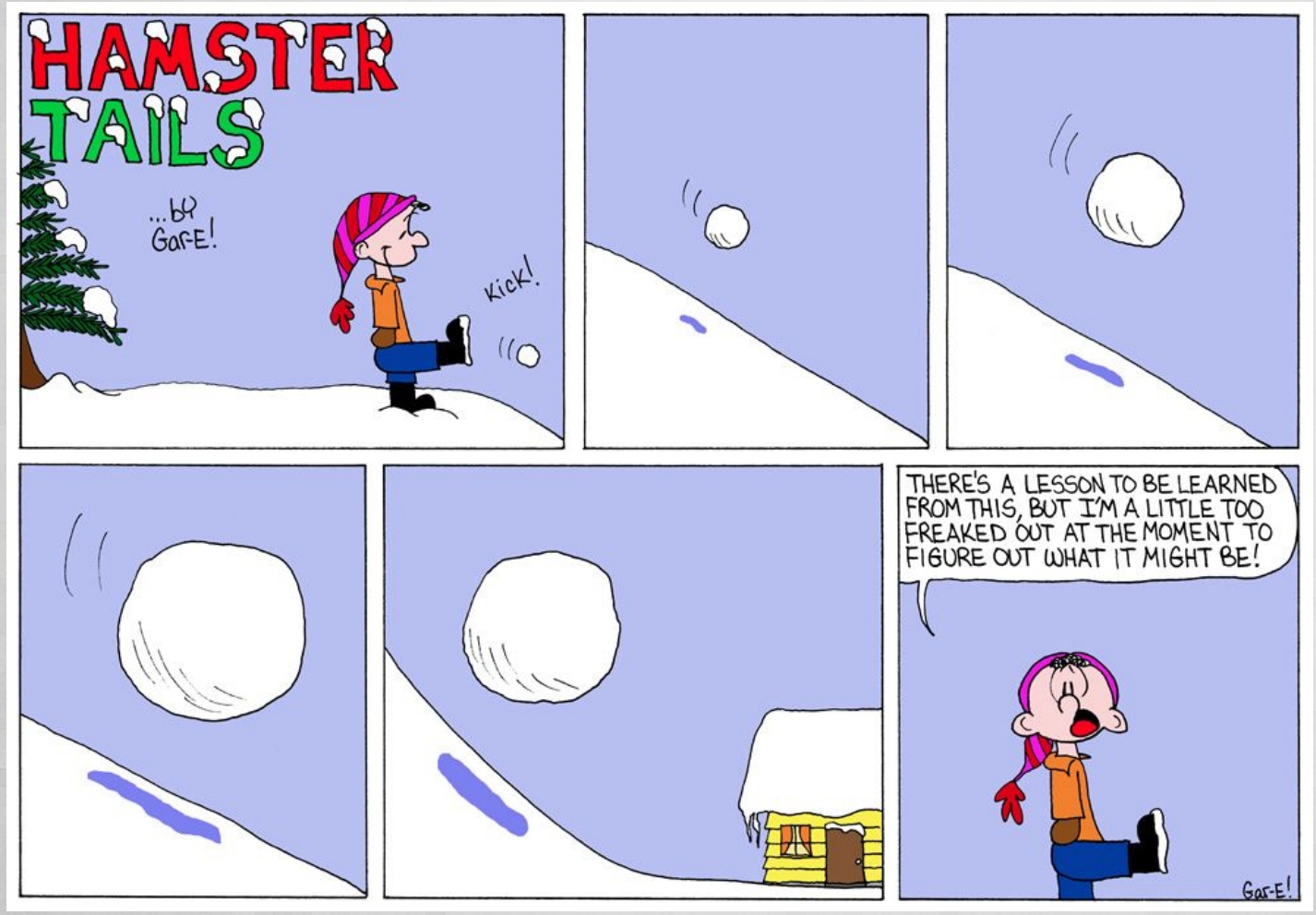
Network Requests

- Discrete requests from devices, should be collapsed into a single request
- Server-side concurrency is needed to effectively reduce network chattiness.
- Nested, conditional, parallel execution of backend network calls.



Snowball Effect

Every refresh generates 3 new request on server side



ReactiveX - RXJava

- Was first introduced by microsoft in 2009 and version 2 in 2012
- Ported to the JVM by netflix in 2014
- Rx is based on the Observer pattern, but adds the option to subscribe and unsubscribe (observable does right).
- Current Options in Java:
  - Future -> you need a get in the end
  - Callback -> callback hell
- Rx is a way of chaining callbacks together for a given notification
- RX is event based and not message based

- ReactiveX is a library for composing asynchronous and-event based programs by using observable sequences.
- ReactiveX is a design pattern the effects all classes of code (you must return objects of observables)

http://reactivex.io/

RXJava

Iterable	Observable
pull	*push*
T next()	onNext(T)
throws Exception	onError(Exception)
returns;	onCompleted()

```
// Iterable<String>
// that contains 75 Strings
getDataFromLocalMemory()
 .skip(10)
 .take(5)
 .map({ s ->
  return s + "_transformed"})
 .forEach{
    { println "next => " + it})
```

```
// Observable<String>
// that emits 75 Strings
getDataFromNetwork()
 .skip(10)
 .take(5)
 .map({ s ->
  return s + "_transformed"})
 .subscribe{
    { println "onNext => " + it})
```

RXJava - Basic

```
Observable.just(1, 2 ,3 ,4)
        .filter( val -> val % 2 == 0 )
        .map( val -> val*2 )
        .subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        System.out.println("Complete!");
    }

    @Override
    public void onError(Throwable e) {}

    @Override
    public void onNext(Integer value) {
        System.out.println("onNext: " + value);
    }
});
```

RXJava - Create

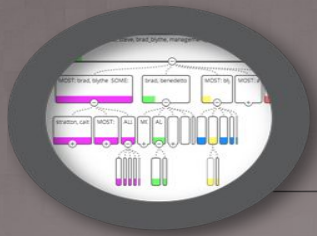
```
Observable.OnSubscribe<String> subscribeFunction = (s) -> {
    Subscriber subscriber = (Subscriber)s;

    for (int ii = 0; ii < 10; ii++) {
        if (!subscriber.isUnsubscribed()) {
            subscriber.onNext("Pushed value " + ii);
        }
    }

    if (!subscriber.isUnsubscribed()) {
        subscriber.onCompleted();
    }
};

Observable createdObservable = Observable.create(subscribeFunction);

createdObservable.subscribe(
        (incomingValue) -> System.out.println("incomingValue " + incomingValue),
        (error) -> System.out.println("Something went wrong" + ((Throwable)error).getMessage()),
        () -> System.out.println("This observable is finished")
);
```

Send request to salesforce

```
final JsonParser jsonParser = new JsonParser();

urlRequest1 = Observable.just(new RequestWrapper<String>("gps","http://gpsservice/seg1/134"));
urlRequest2 = Observable.just(new RequestWrapper<String>("trucksavailability","http://vehiclemanagement/availability"));
urlRequest3 = Observable.just(new RequestWrapper<String>("truckshealth","http://Analyticsservice/trucks/health"));

Observable<RequestWrapper<String>> allRequests = Observable.concat(urlRequest1, urlRequest2,urlRequest3);
allRequests.flatMap(request -> Observable.just(request)
    .observeOn(Schedulers.io())
    .map(this::mapRequestToResponse)
    .retry(2)
    .filter(response -> response.payload.getStatusLine().getStatusCode() == HttpStatus.SC_OK)
    .map(value -> mapResponseToJson(jsonParser,value))
).toList()
.subscribe(
      jsonElementList -> {
         JsonObject clientResult = new JsonObject();
         for (RequestWrapper<JsonElement> jsonElement : jsonElementList) {
            clientResult.add(jsonElement.header,jsonElement.payload);
         }
         response.setResponse(clientResult);
      },
      error -> System.out.println("error"),
      () -> System.out.println("completed"));
```

RXJava - Error Handling

- onErrorResumeNext( ) — instructs an Observable to emit a sequence of items if it encounters an error (converts error to new Observer)
- onErrorReturn( ) — instructs an Observable to emit a particular item when it encounters an error (converts error to different error)
- onExceptionResumeNext( ) — instructs an Observable to continue emitting items after it encounters an exception (but not another variety of throwable)
- retry( ) — if a source Observable emits an error, re-subscribe to it in the hopes that it will complete without error
- retryWhen( ) — if a source Observable emits an error, pass that error to another Observable to determine whether to re-subscribe to the source



verview

Fullstack Developers

The journey to reactive ...

- Four Reactive Cornerstones - Recap
- Vehicle Fleet Tracker - overview of application
- Reactive refactoring
  - Breakdown of monolithic app to microservices
  - Salesforce CRM update Use Case
    (Emphasising Responsive (readable asynchronous programing))
    Salesforce multithreading - Scala Futures
  - Consolidate UI request (performance degradation due to microservices)
    - Observable patterns with option to abort - RXJava
  - Analytics performance boost, solution for thread exceptions
    - Resilience / Message driven - Akka

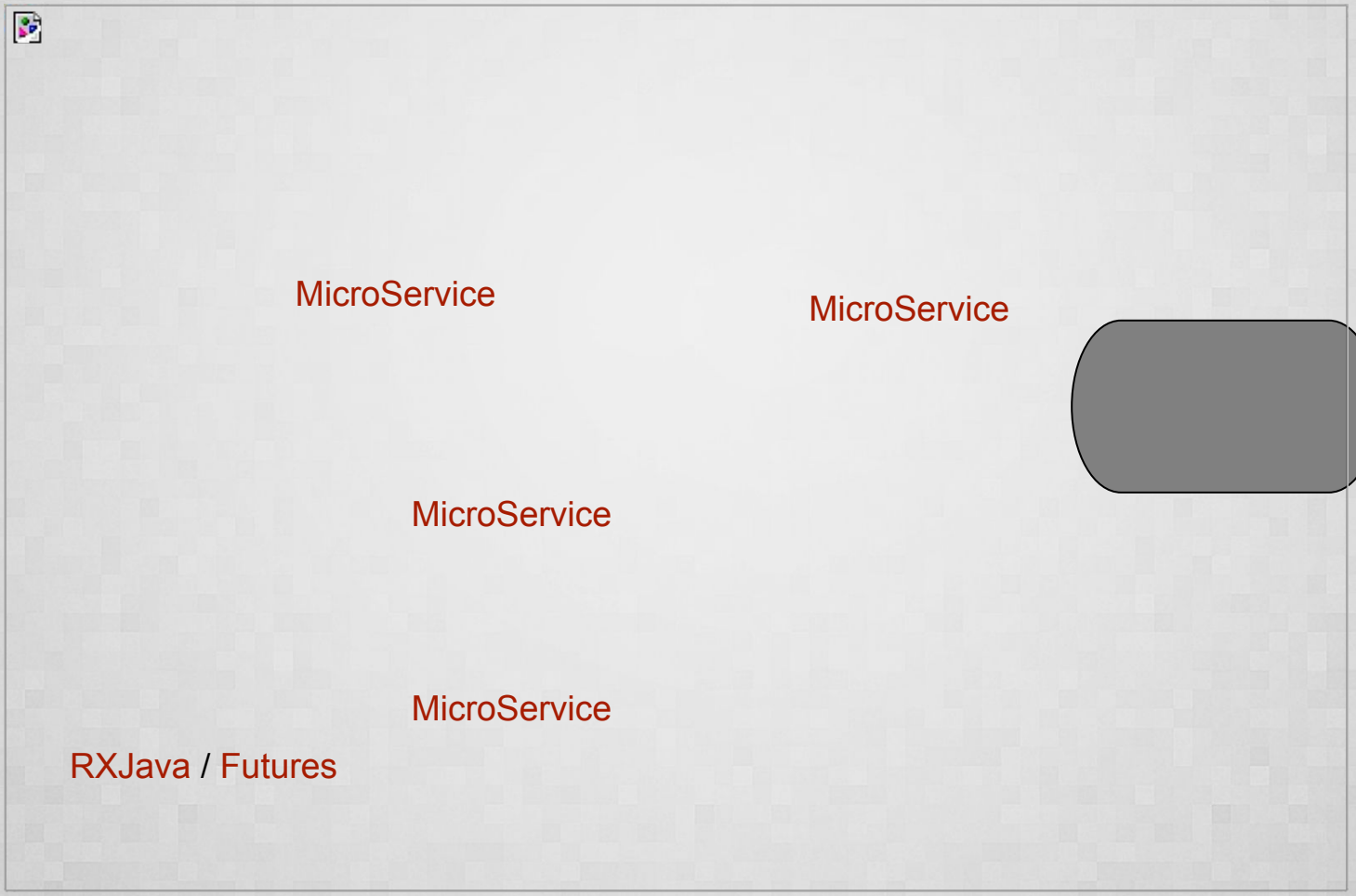
Fullstack
Developers

Resilience

Task Managing

- We need to manage thousands of trucks
- We are not utilizing our multicores properly
- Current implementation is Tuck Manager with concurrenthashmap - a lot of locking
- Current problem - if we have an exception while connecting to truck, we have a hard time handling it.

Network Requests

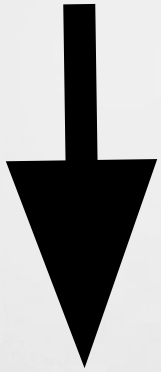


AKKA

- Akka is an open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM.
- Akka supports multiple programming models for concurrency, but it emphasizes actor-based concurrency, with inspiration drawn from Erlang.
- Language bindings exist for both Java and Scala. Akka is written in Scala and, as of Scala 2.10, Akka's actor implementation is included as part of the Scala standard library

Solution to moore's' law

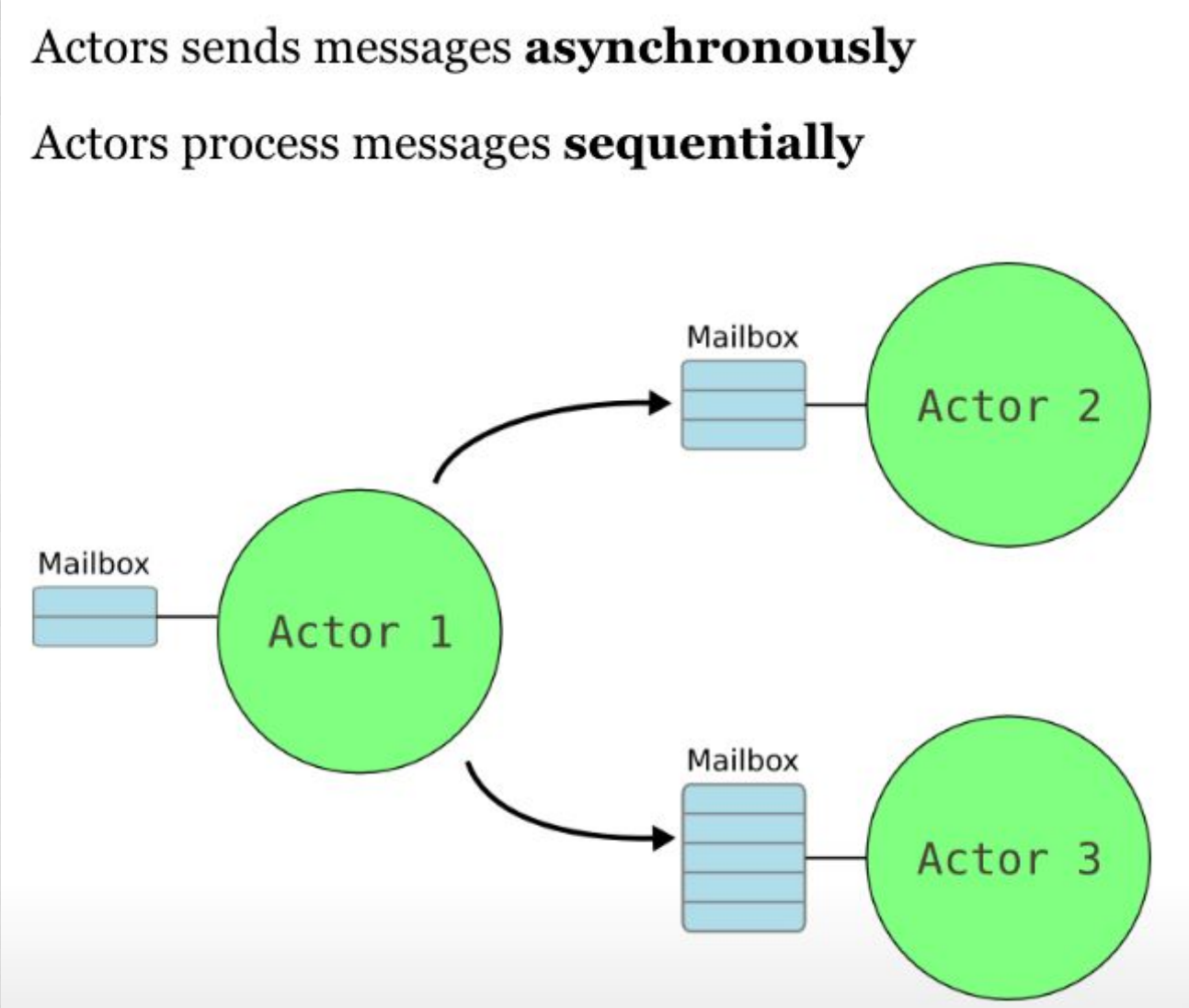
Multi Core
Cluster Computation



Scale OUT and not UP
Scale Horizontal not Vertical

Conclusion of Amdahl's Law for Concurrency

NEVER BLOCK



Actor messages

Actor messages - code example

```
class TruckActor(truckId : String) extends Actor{
 var driver : Option[Driver] = None
 var status : Option[TruckStatus] = None

 override def receive: Receive = {
  case message : Driver => driver = Some(message)
  case IsTruckAvailable => sender() ! status.getOrElse(TruckStatus(false)).isTruckReady
 }
}

/*
this object is for testing only
*/
object TruckActorExample extends App{
 val system = ActorSystem("CarSystem")
 val assignTruck : ActorRef = system.actorOf(Props(new TruckActor("1234")))
 assignTruck ! Driver("chaim")
}
```

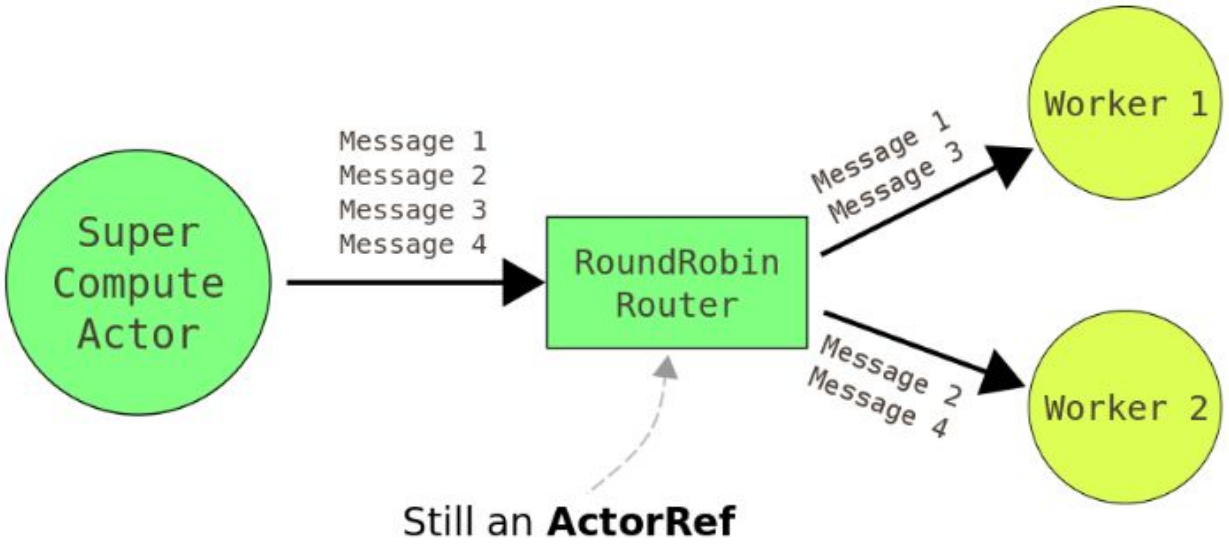
- We need to do a self test of each truck every day.
- The self test is very cpu intensive.
- We have one thread/actor that iterates over all trucks, and tests them.
- To solve the bottleneck we would like to load balance the work

Scenario - Truck Self Test

Routers

Routers are not actors, they are specifically designed for concurrency

Akka comes with different routers



Routers

```
class TruckComputations extends Actor{
 val router = context.actorOf(RoundRobinPool(10).props(Props[TruckCalculateAvailability]),"router1")

 override def receive: Receive = {
   case test : TruckSelfTest => router ! test
 }
}

class TruckCalculateAvailability extends Actor {
 override def receive: Receive = {
   case test: TruckSelfTest =>{
     autoTestTruck(test.truckId)
   }
 }
}

case class TruckSelfTest(truckId : String)

/* this object is for testing only */
object TruckComputationsExample extends App{
 val system = ActorSystem("TruckSystem")
 val computeTruck : ActorRef  = system.actorOf(Props[TruckComputations])
 computeTruck ! TruckSelfTest("id1234")
 computeTruck ! TruckSelfTest("id454")
}
```

Scenario - Exception Handling

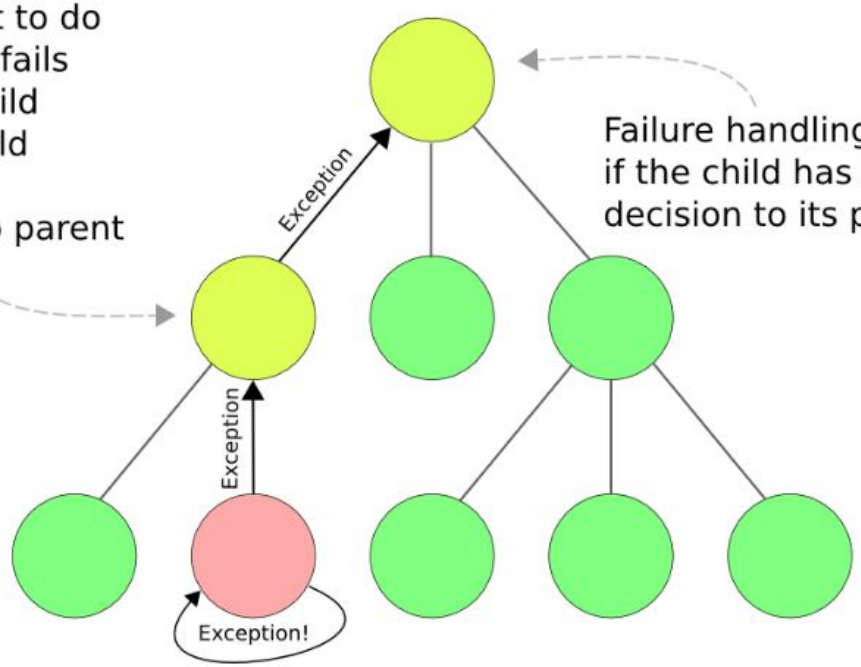
- When the connection to a truck fails - we lose the thread.
- Passing data from work thread to data object requires locking
- How to have logic flow on failures?

Scenario - Connectivity Issues

Supervision

Supervision through hierarchies

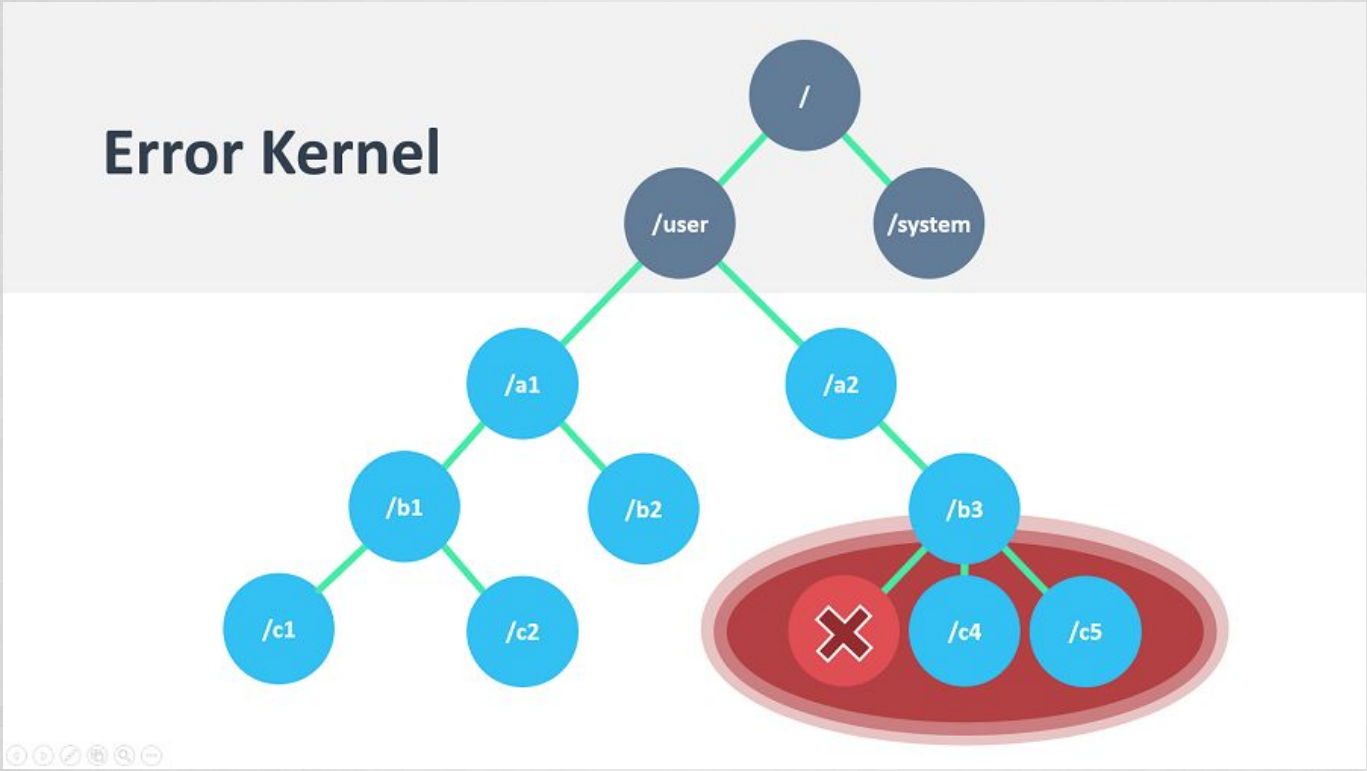
Decides what to do
when a child fails
- ***Resume*** child
- ***Restart*** child
- ***Stop*** child
- ***Escalate*** to parent



Failure handling is delagated,
if the child has ***Escalated*** the
decision to its parent

Actor supervision is recursive, enabling delegation of failure handling

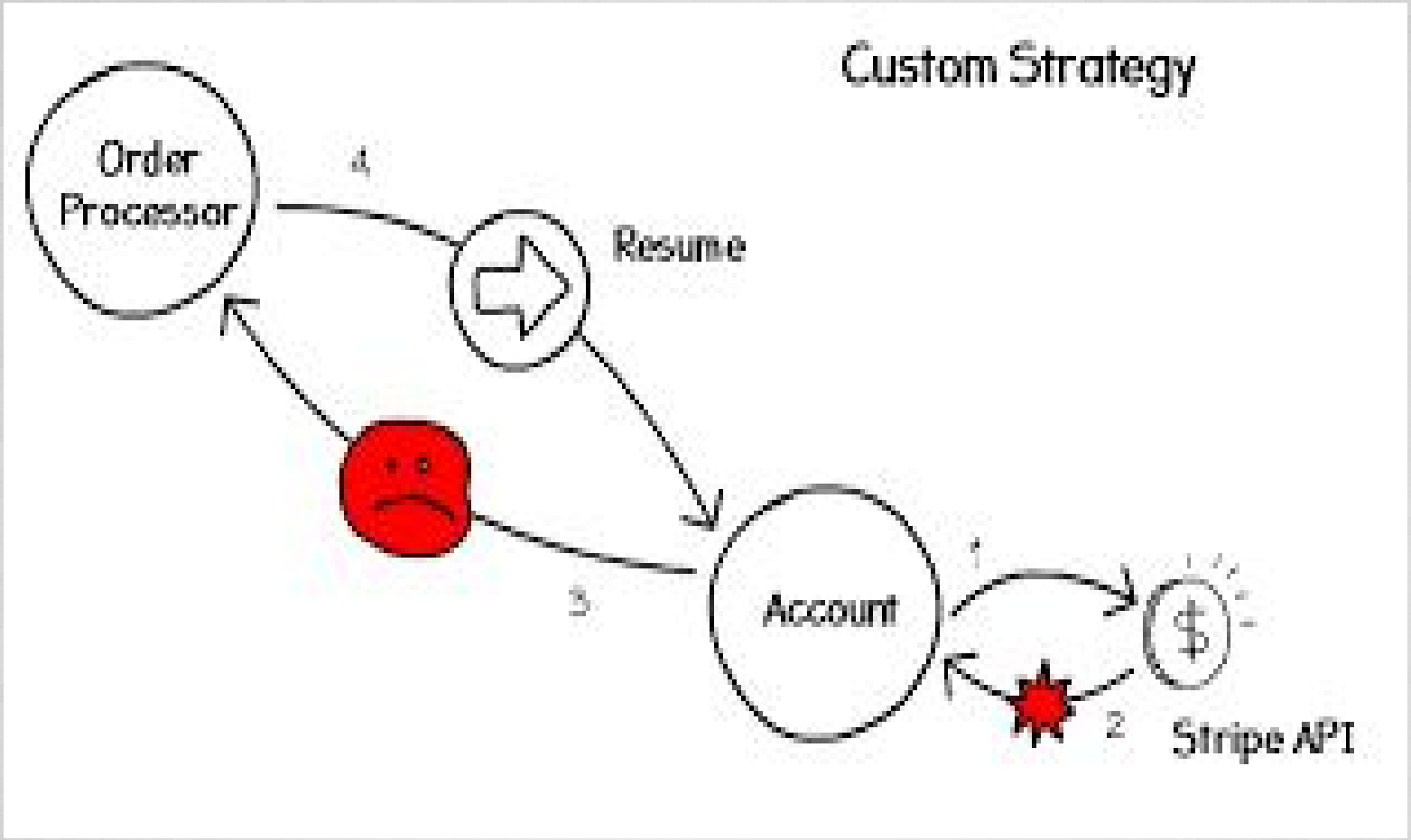
Kernel Error Pattern



Truck actor saves the state of the truck. We would like to communicate with the truck, but if there are any errors in the communication that we did not foresee, we do not want to loose and data about the truck

TruckStatusActor

```
class TruckStatusActor(truckIp : String) extends Actor {
 private var requestTimes : Int = 0
 private var speed : Int = 80
 private var mileage : Int = 0

 context.system.scheduler.schedule(0 seconds, 5 minutes)( self ! CalculateStatus)

 def connectToTruck() : Boolean = { ... }

 def calcStatus(): Unit ={
   requestTimes += 1
   if (!connectToTruck())
     throw new ConnectionException()
   mileage =    ...
   speed =    ...
   parent ! TruckTelemetry(mileage,speed)
 }

 override def receive: Receive = {
   case CalculateStatus => calcStatus
   case GetStatus => sender() ! TruckTelemetry(mileage,speed)
 }
}

object TruckStatusActor {
 case class TruckTelemetry(mileage : Int, speed : Int)
 case object GetStatus
 case object CalculateStatus
 }
```



Resume: If you choose to Resume, this probably means that you think of your child actor as a little bit of a drama queen. You decide that the exception was not so exceptional after all – the child actor or actors will simply resume processing messages as if nothing extraordinary had happened.

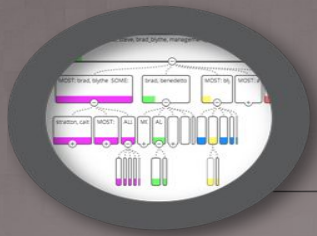
Restart: The Restart directive causes Akka to create a new instance of your child actor or actors. The reasoning behind this is that you assume that the internal state of the child/children is corrupted in some way so that it can no longer process any further messages. By restarting the actor, you hope to put it into a clean state again.

Stop: You effectively kill the actor. It will not be restarted.

Escalate: If you choose to Escalate, you probably don't know how to deal with the failure at hand. You delegate the decision about what to do to your own parent actor, hoping they are wiser than you. If an actor escalates, they may very well be restarted themselves by their parent, as the parent will only decide about its own child actors.

Supervision

```
class TruckActor(truckId : String) extends Actor{
 var ipAddress : String = "127.0.0.1"
 var driver : Option[Driver] = None
 var status : Option[TruckStatus] = None
 var truckStatus : ActorRef = context.actorOf(Props(new TruckStatusActor(ipAddress)))
 var mileage : Int = 0
 var speed : Int = 0

 override def receive: Receive = {
   case message : Driver => driver = Some(message)
   case IsTruckAvailable => sender() ! status.getOrElse(TruckStatus(false)).isTruckReady
   case TruckTelemetry(mileage , speed ) => { this.speed = speed; this.mileage = mileage}
 }

 override val supervisorStrategy =
   OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
     case _: ConnectionException      => Restart //maybe connection parameters are incorrect
     case _: ArithmeticException      => Resume
     case _: NullPointerException     => Restart
     case _: IllegalArgumentException => Stop
     case _: Exception                => Escalate
   }
}
```



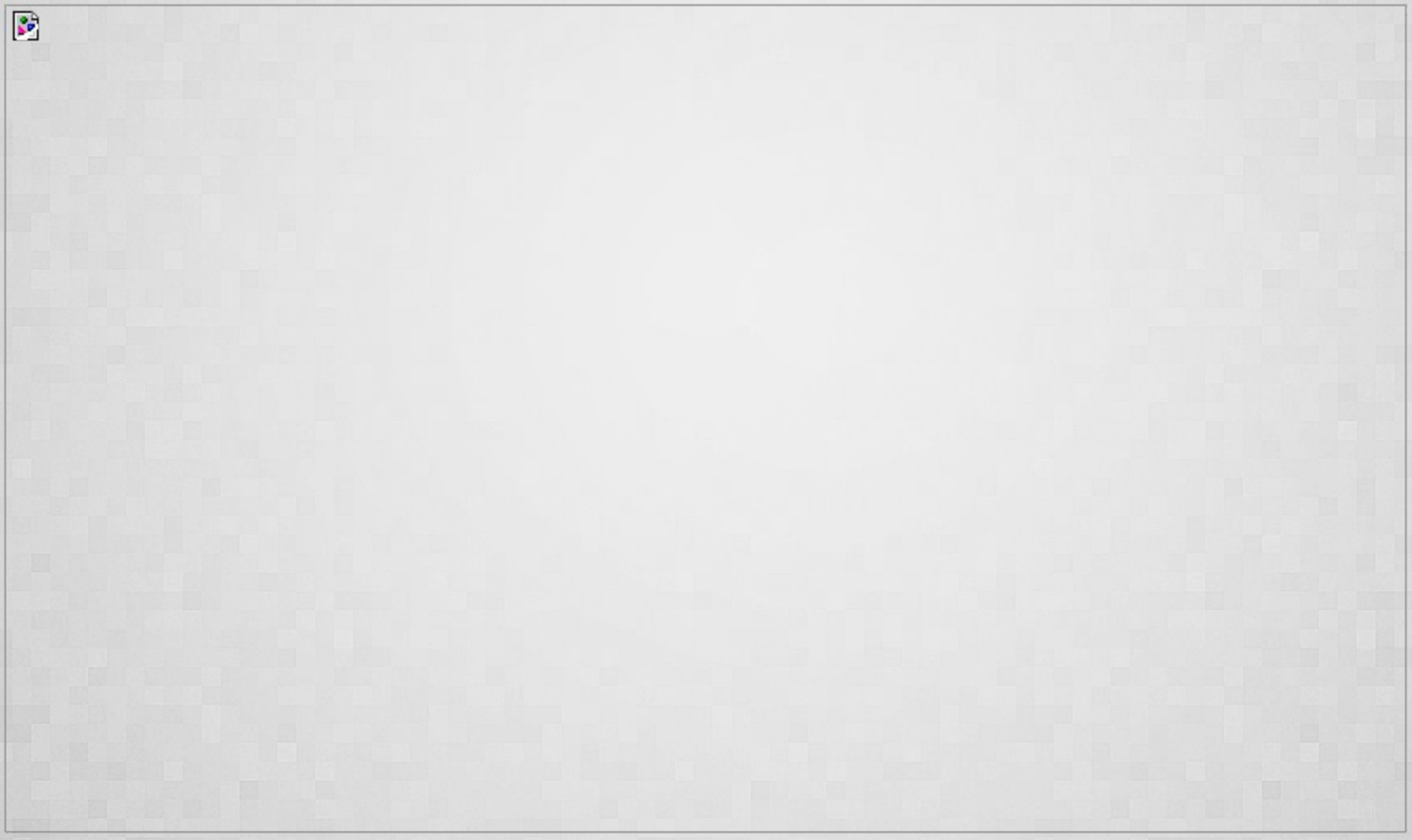
verview

Fullstack Developers

The journey to reactive ...

- Four Reactive Cornerstones - Recap
- Vehicle Fleet Tracker - overview of application
- Reactive refactoring
  - Breakdown of monolithic app to microservices
  - Salesforce CRM update Use Case
    (Emphasising Responsive (readable asynchronous programing))
    Salesforce multithreading - Scala Futures
  - Consolidate UI request (performance degradation due to microservices)
    - Observable patterns with option to abort - RXJava
  - Analytics performance boost, solution for thread exceptions
    - Resilience / Message driven - Akka

Network Requests



THANK YOU

Chaim Turkel
Email: chaim@tikalk.com
Tel: +972-053-550-6148