Reactive Programming with Vert.x

Embrace asynchronous to build responsive systems

Clement Escoffier
Principal Software Engineer, Red Hat

Reactive

The new gold rush ?

Reactive system, reactive manifesto, reactive extension, reactive programming, reactive Spring, reactive streams...

Scalability, Asynchronous, Back-Pressure, Spreadsheet, Non-Blocking, Actor, Agent...

Reactive ?

Oxford dictionary

1 - Showing a response to a stimulus

1.1 (*Physiology)* Showing an immune response to a specific antigen

1.2 (of a disease or illness) caused by a reaction to something: *'reactive depression'*

2 - Acting in response to a situation rather than creating or controlling it

Reactive Architecture / Software

Application to software

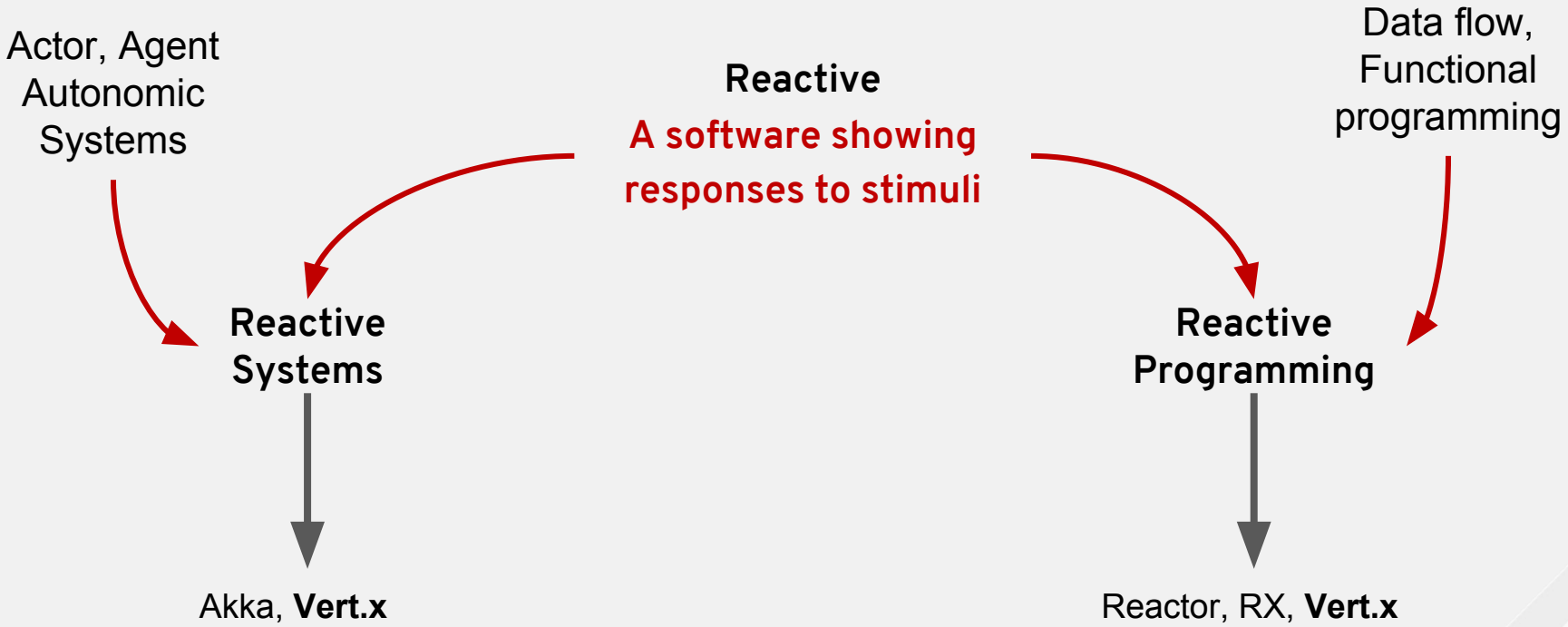
A software showing responses to stimuli

- Events, Messages, Requests, Failures, Measures, Availability…
- The end of the flow of control ?

Is it new?

- Actors, Object-oriented programming…
- IOT, Streaming platform, complex event processing, event sourcing…

The 2+1* parts of the reactive spectrum



Eclipse Vert.x

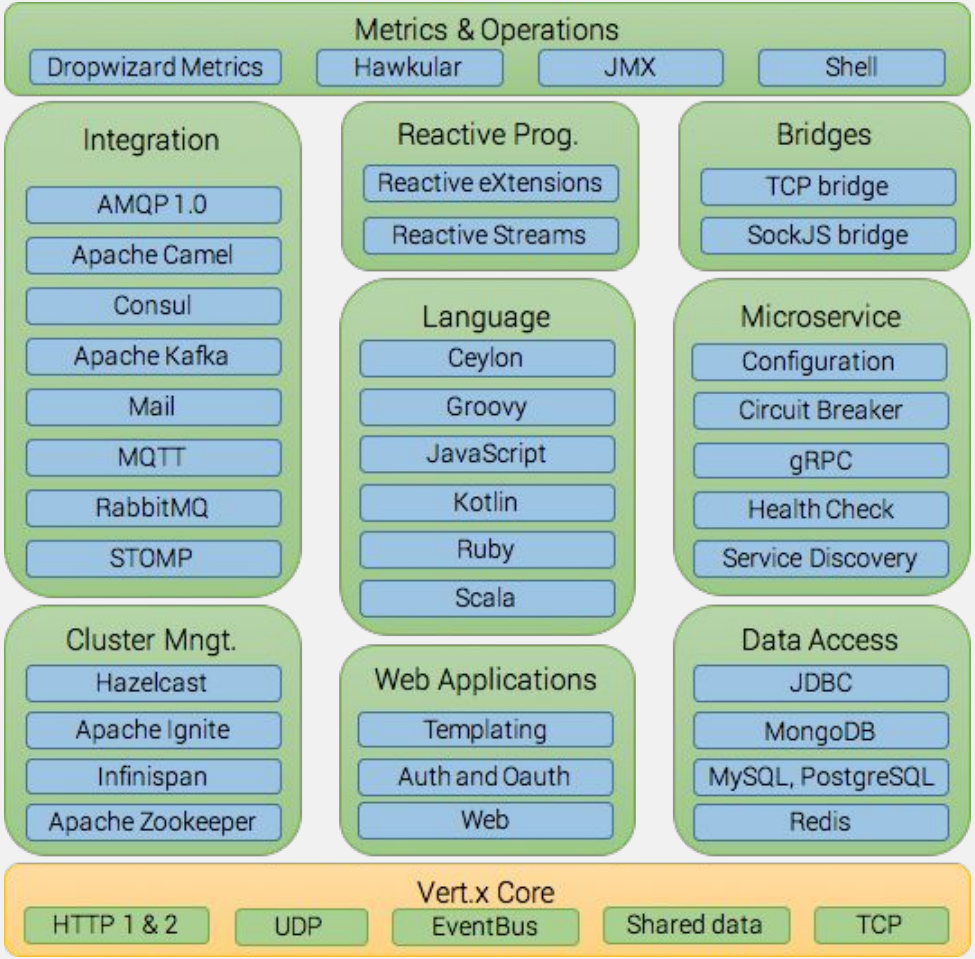
Vert.x is a toolkit to build distributed and reactive systems

- **Asynchronous Non-Blocking development model**
- Simplified concurrency (**event loop**)
- Microservice, Web applications, IOT, API Gateway, high-volume event processing, full-blown backend message bus

Eclipse Vert.x Ecosystem

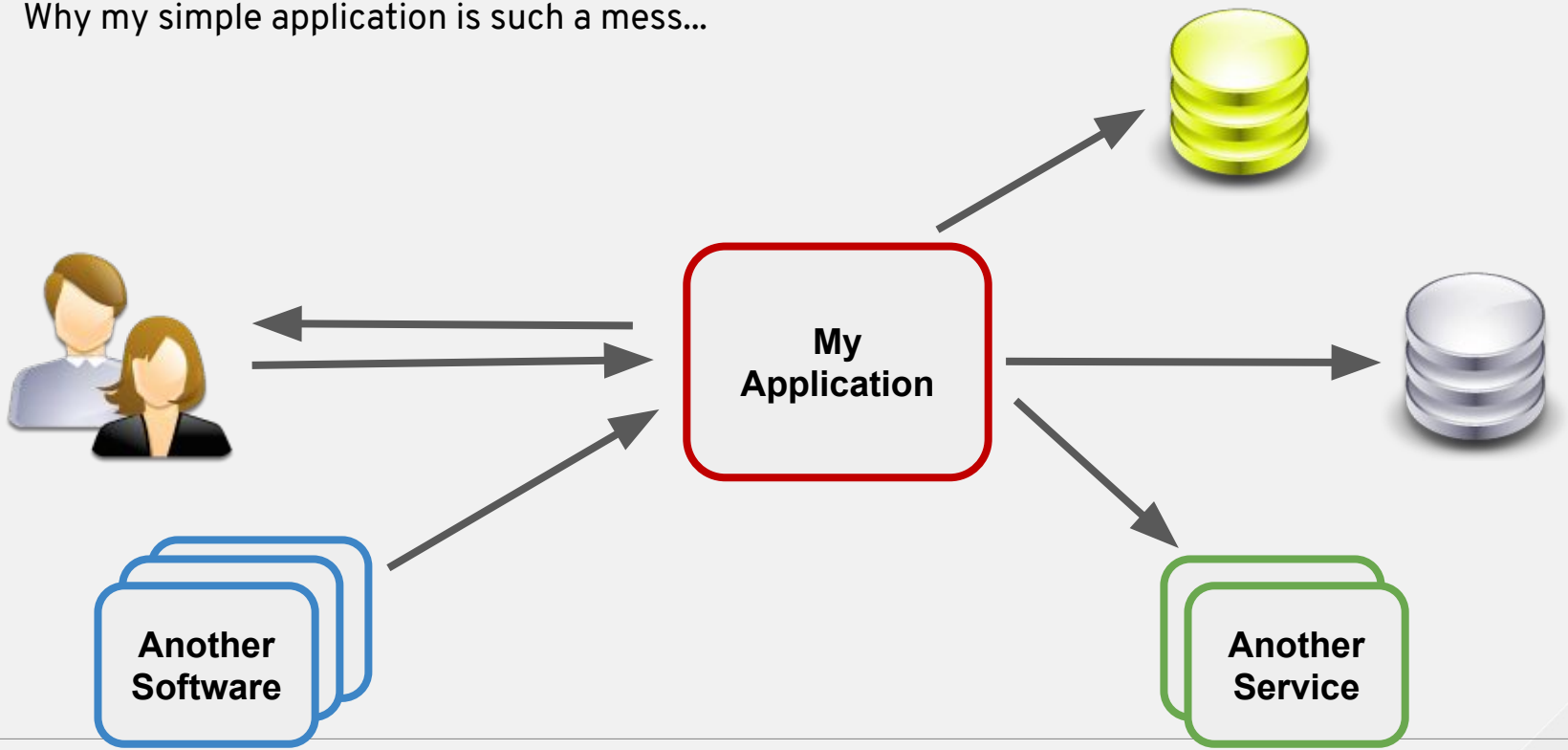
Build reactive systems

- Polyglot
- Integrable
- Embeddable
- Pragmatic
- Freedom



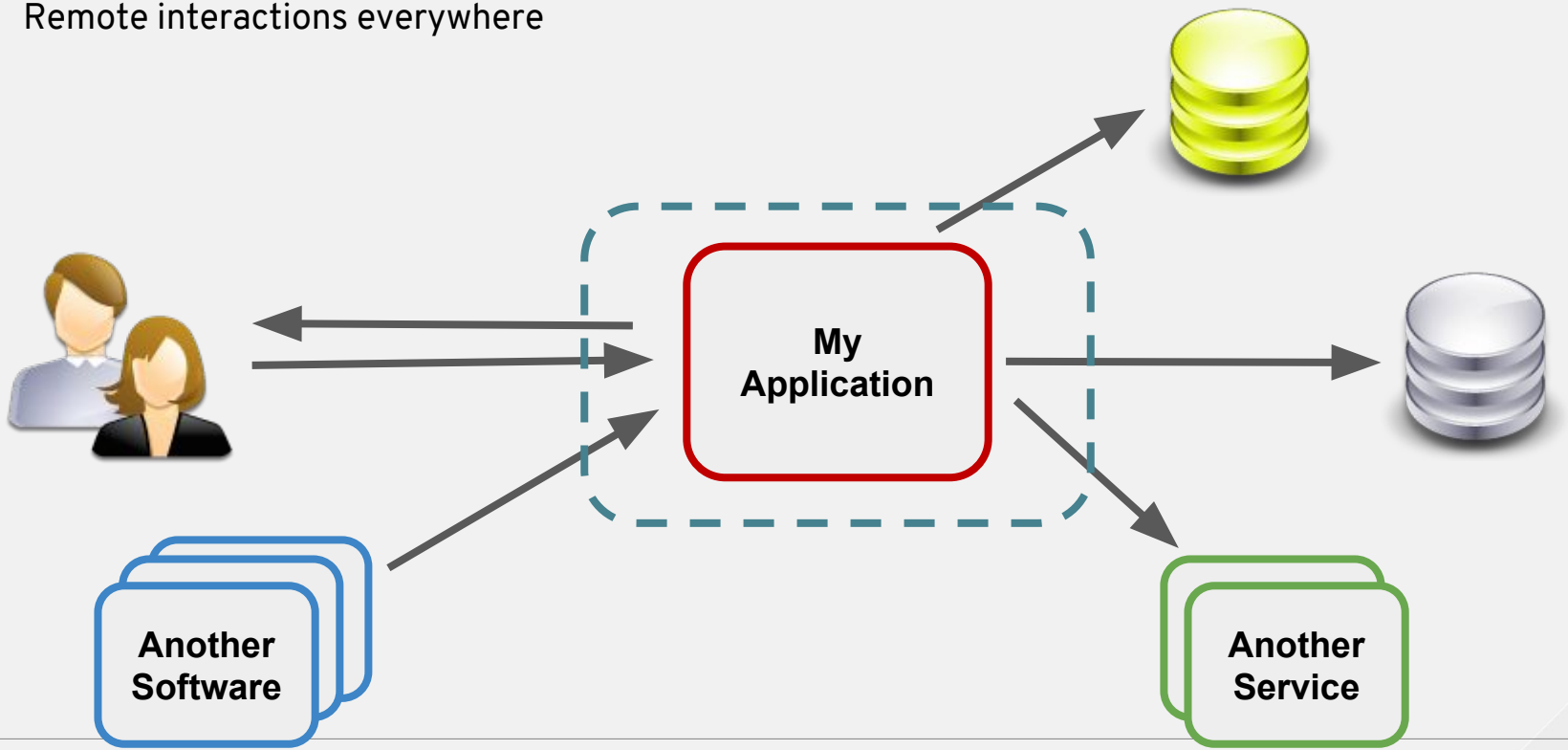
Modern software is not autonomous

Why my simple application is such a mess...



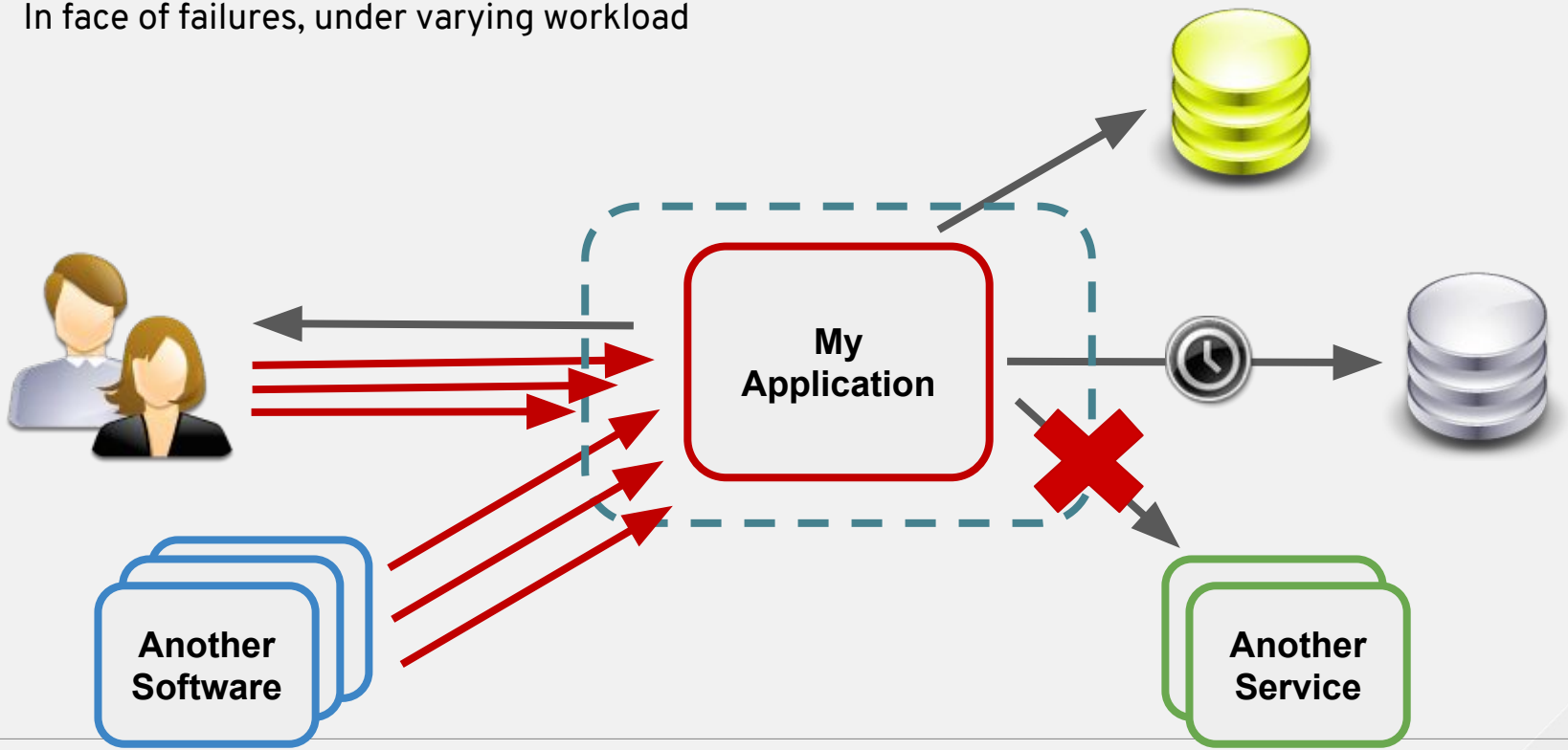
Modern software is not autonomous

Remote interactions everywhere



Need for responsiveness

In face of failures, under varying workload

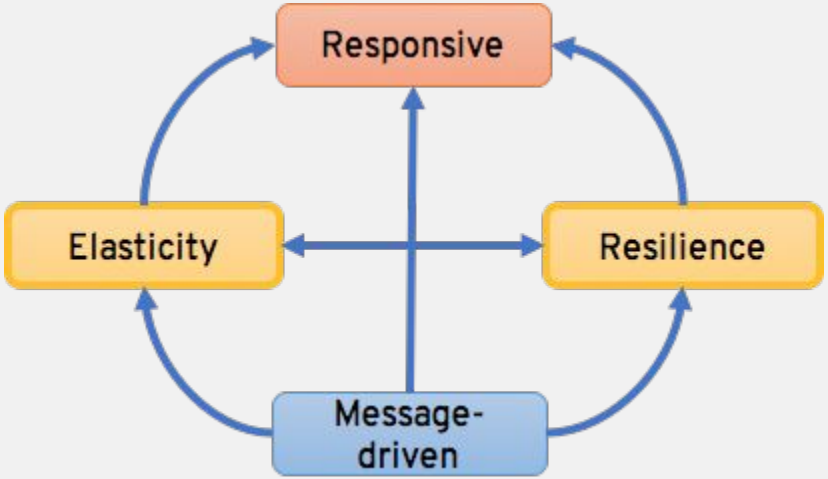


Reactive Systems => Responsive Systems

Reactive Manifesto

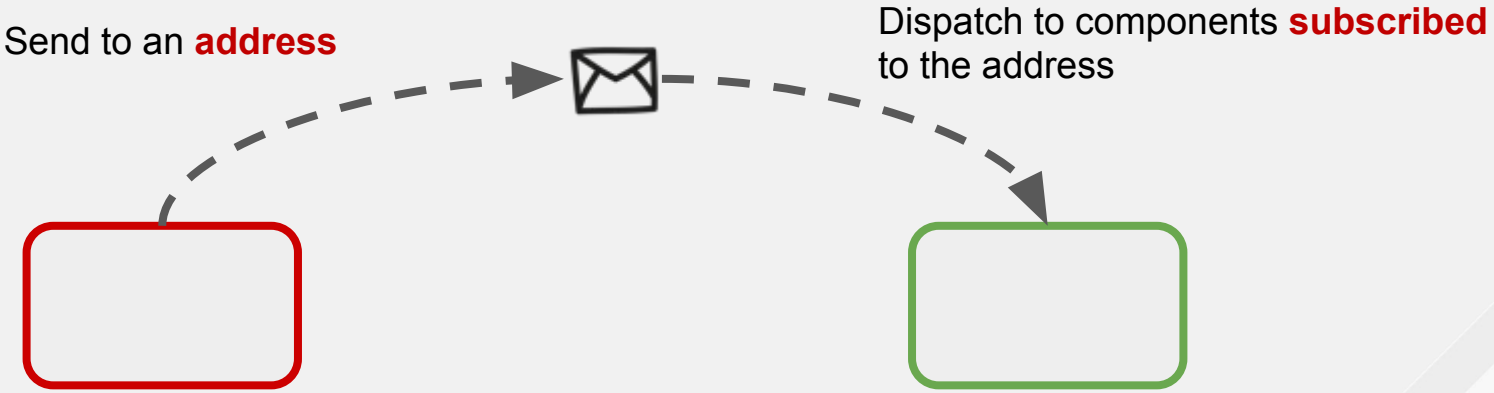
http://www.reactivemanifesto.org/

Reactive Systems are an architecture style focusing on **responsiveness**



Asynchronous message passing

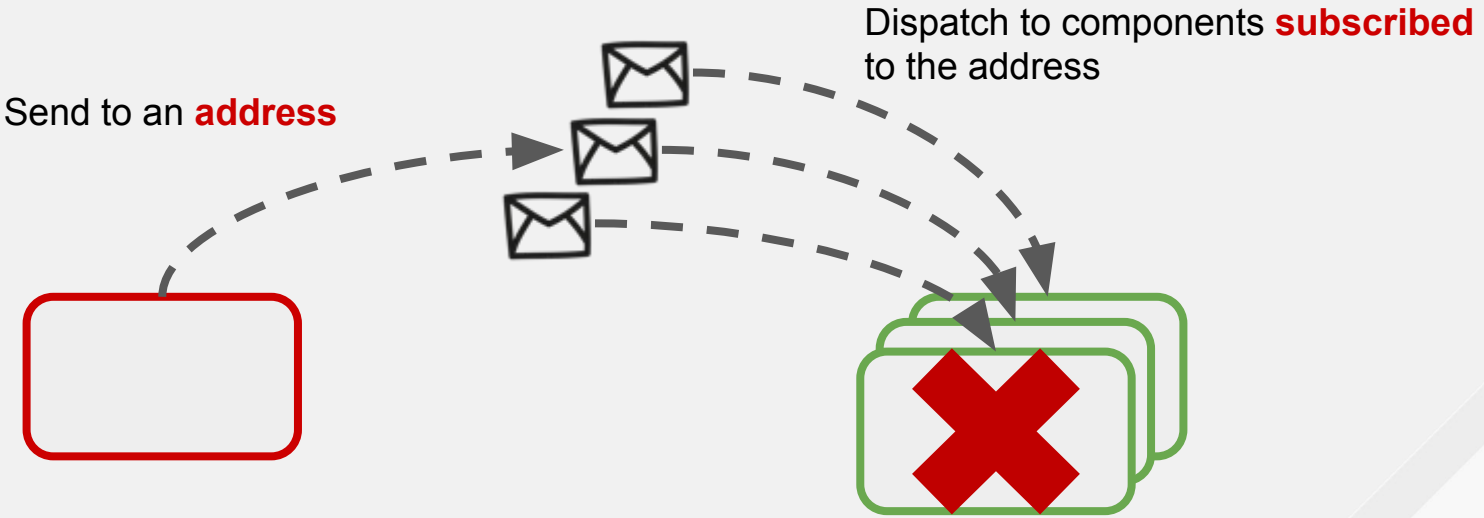
Components interacts using **messages**



Asynchronous message passing => Elasticity

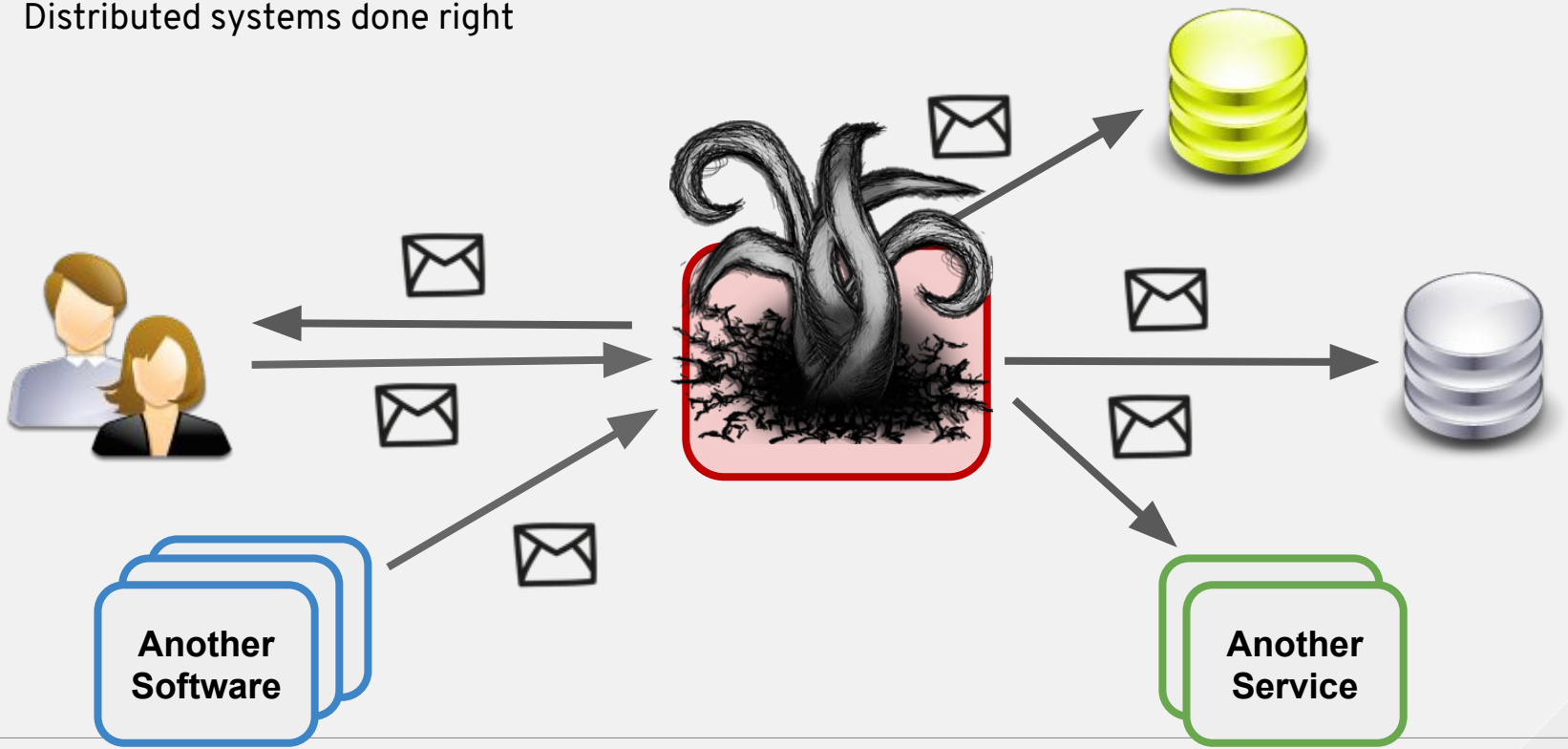
Messages allows **elasticity**

Resilience is not only about failures, it's also about **self-healing**



So, it's simple, right ?

Distributed systems done right



Pragmatic reactive systems

And that's what Vert.x offers to you

Development model => Embrace **asynchronous**

Simplified concurrency => **Event-loop**, not thread-based

I/O

- **Non-blocking I/O**, if you can't isolate
- HTTP, TCP, RPC => Virtual address
- Messaging

Asynchronous development model

Asynchronous development models

Async programming

- **Exists since the early days of computing**
- Better usage of hardware resource, avoid blocking threads

Approaches

- **Callbacks**
- Future / Promise (single value, many read, single write)
- Data streams
- Data flow variables (cell)
- Continuation
- Co-Routines

Asynchronous development model

Callbacks

Synchronous

```
public int compute(int a, int b) {
   return ...;
}
```

```
int res = compute(1, 2);
```

Asynchronous

```
public void compute(int a, int b,
   Handler<Integer> handler) {
      int i = ...;
      handler.handle(i);
}
```

```
compute(1, 2, res -> {
   // Called with the result
});
```

Asynchronous development model

Web server example

```
vertx.createHttpServer()
  .requestHandler(req ->
      req.response().end(Json.encode(list)))
  .listen(8080, hopefullySuccessful -> {
      if (hopefullySuccessful.succeeded()) {
          System.out.println("server started");
      } else {
          System.out.println("D'oh !");
      }
  });
```

Callbacks lead to ....

Reality check....

```
client.getConnection(conn -> {
  if (conn.failed()) {/* failure handling */}
  else {
     SQLConnection connection = conn.result();
     connection.query("SELECT * from PRODUCTS",
        rs -> {
           if (rs.failed()) {/* failure handling */}
           else {
              List<JsonArray> lines =    rs.result().getResults();
              for (JsonArray l : lines) {  System.out.println(new Product(l)); }
              connection.close(
                 done -> {
                  if (done.failed()) {/* failure handling */}
                });
        }
     });
  }
});
```

Reactive Programming

Reactive programming - let’s rewind….

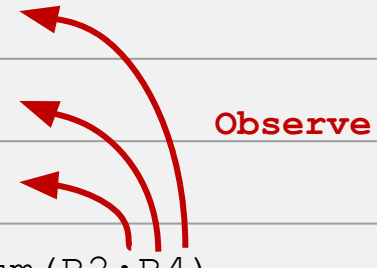
Do we have Excel users in the room ?

My Expense Report	
Lunch	15$
Coffee	25$
Drinks	45$
Total	85$

Reactive programming - let’s rewind….

Do we have Excel users in the room ?

My Expense Report	
Lunch	15$
Coffee	25$
Drinks	45$
Total	`=sum(B2:B4)`

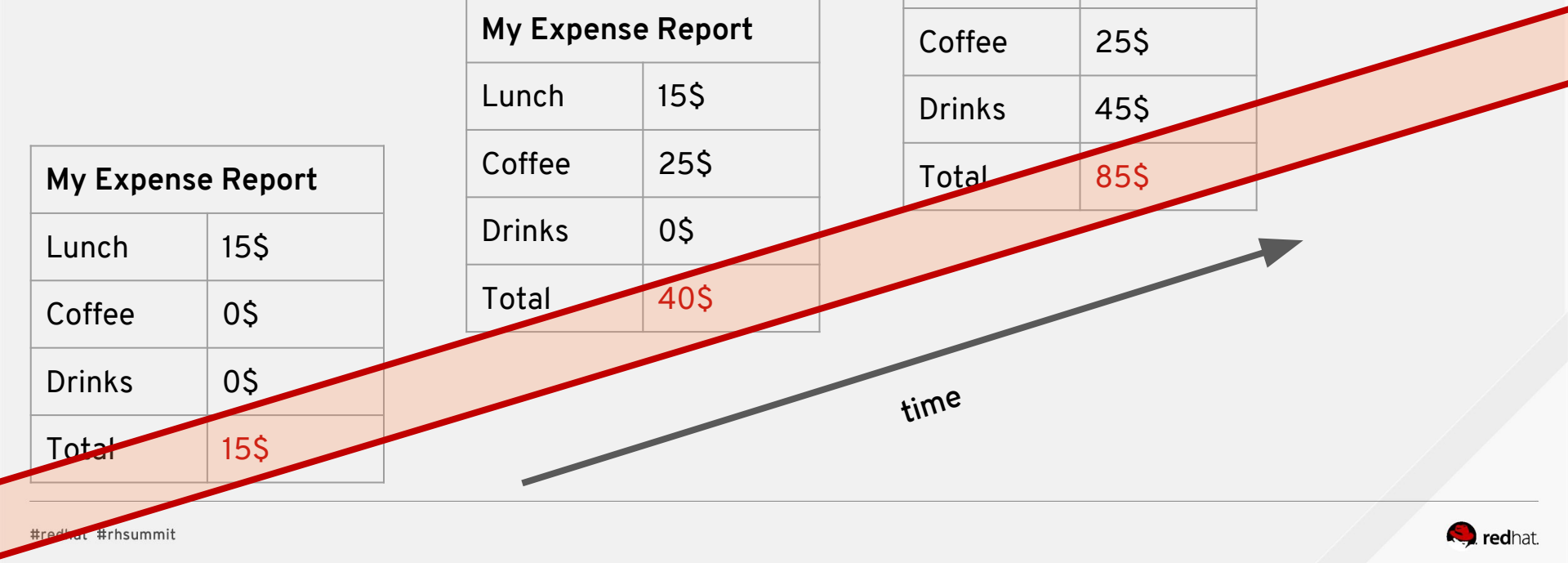


Observables

My Expense Report	
Lunch	15$
Coffee	0$
Drinks	0$
Total	15$

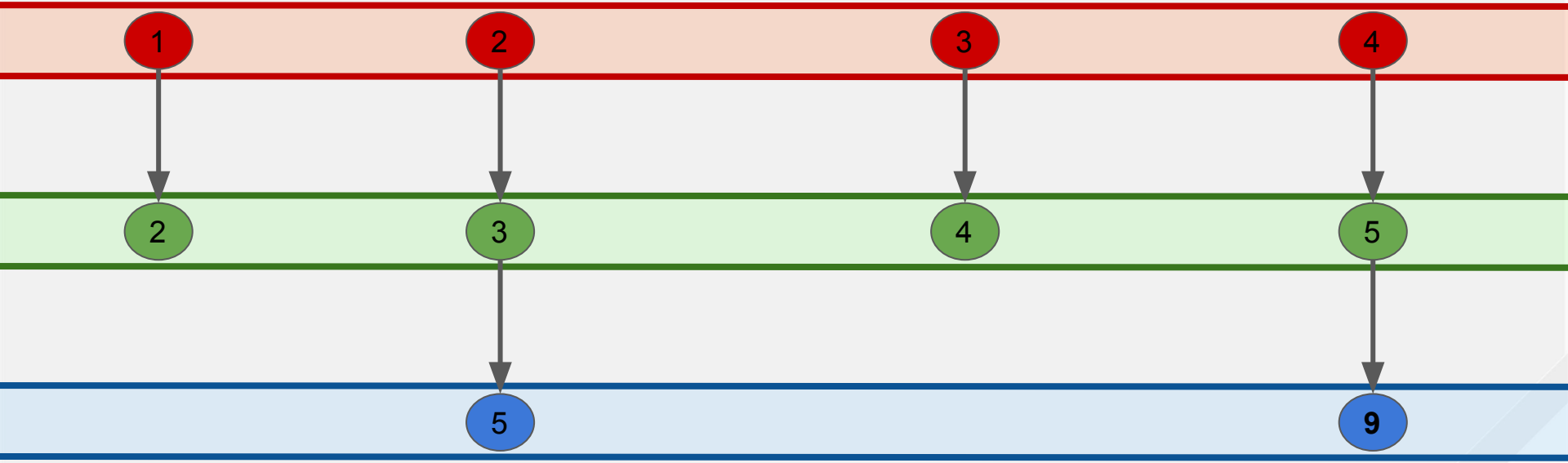
My Expense Report	
Lunch	15$
Coffee	25$
Drinks	0$
Total	40$

My Expense Report	
Lunch	15$
Coffee	25$
Drinks	45$
Total	85$



Reactive Programming

Observable and Subscriber



Reactive Extension - RX Java

```
Observable<Integer> obs1 = Observable.range(1, 10);

Observable<Integer> obs2 = obs1.map(i -> i + 1);

Observable<Integer> obs3 = obs2.window(2)
   .flatMap(MathObservable::sumInteger);

obs3.subscribe(
   i -> System.out.println("Computed " + i)
);
```

Reactive types

Observables

- Bounded or unbounded stream of values
- Data, Error, End of Stream

```
observable.subscribe(
   val -> { /* new value */ },
   error -> { /* failure */ },
   () -> { /* end of data */ }
);
```

Singles

- Stream of one value
- Data, Error

```
single.subscribe(
   val -> { /* the value */ },
   error -> { /* failure */ }
);
```

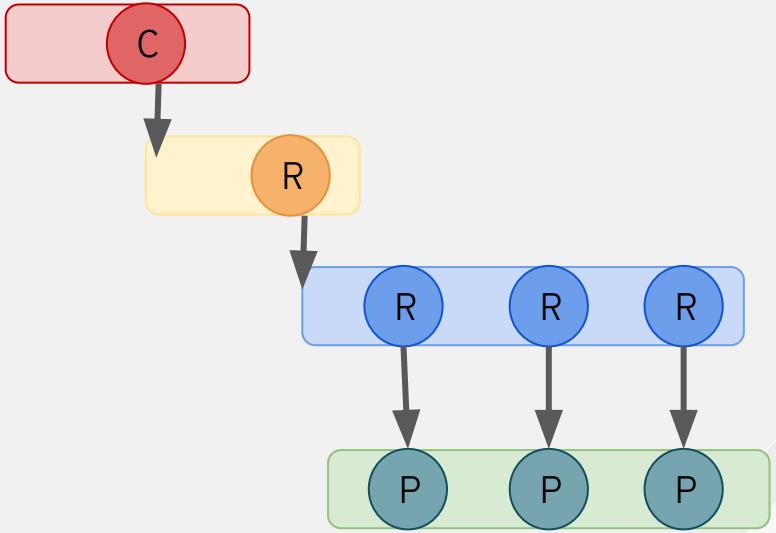
Completables

- Stream without a value
- Completion, Error

```
completable.subscribe(
   () -> { /* completed */ },
   error -> { /* failure */ }
);
```

Handling the asynchronous with reactive programming

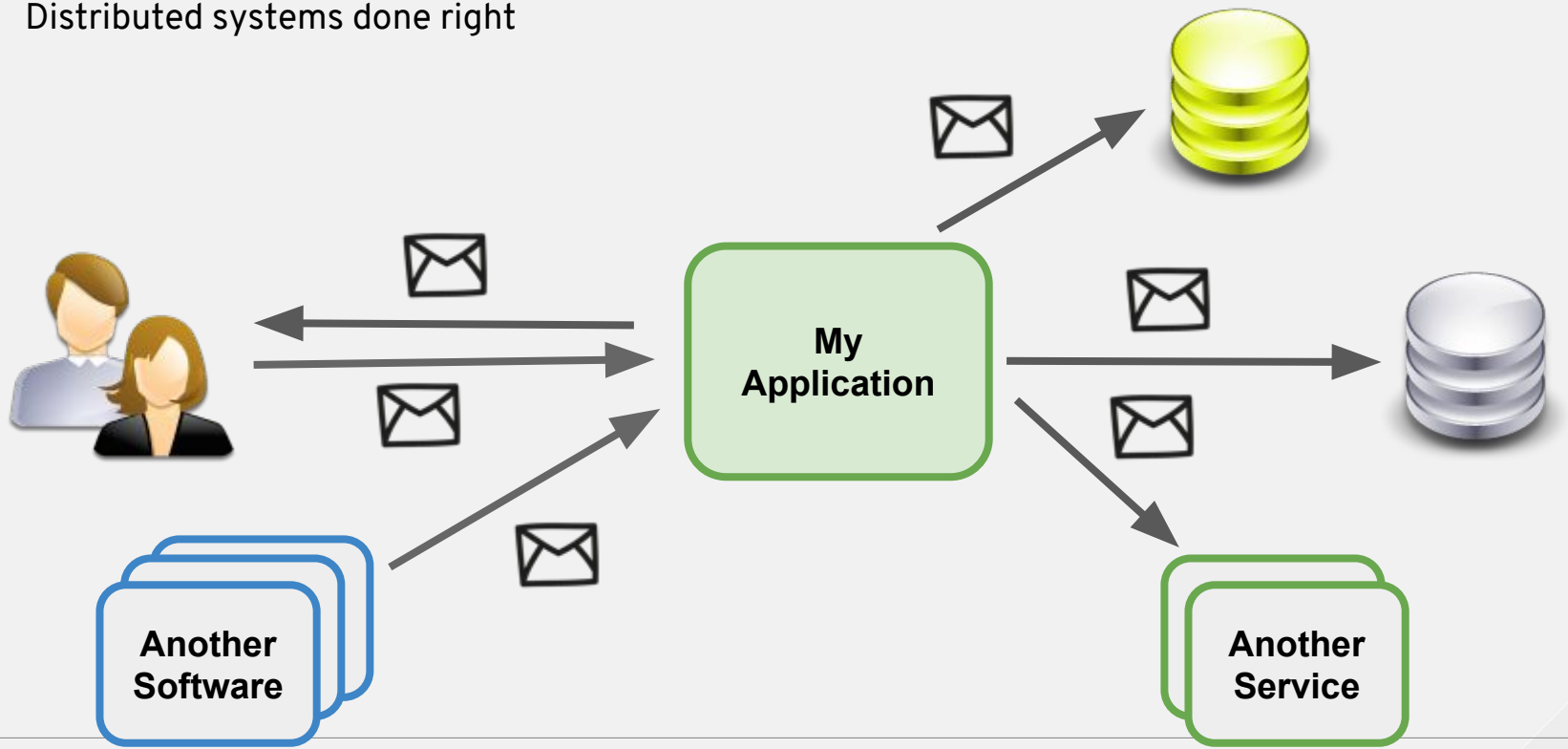
```
client.rxGetConnection()
 .flatMapObservable(conn ->
   conn
     .rxQueryStream("SELECT * from PRODUCTS")
     .flatMapObservable(SQLRowStream::toObservable)
     .doAfterTerminate(conn::close)
 )
 .map(Product::new)
 .subscribe(System.out::println);
```



Unleash your superpowers
Vert.x + RX

Taming the asynchronous

Distributed systems done right



Reactive Web Application

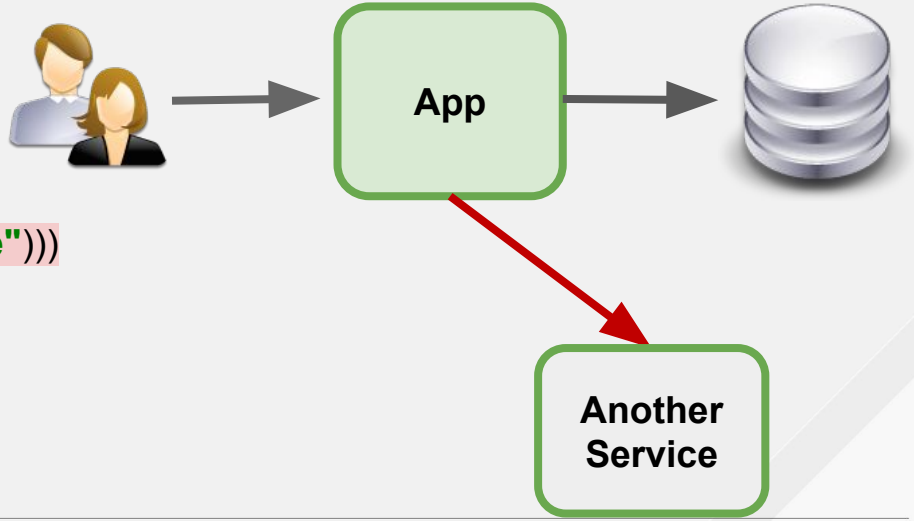
```
private void add(RoutingContext rc) {
  String name = rc.getBodyAsString();
  database.insert(name) // Single (async)
    .subscribe(
      () -> rc.response().setStatusCode(201).end(),
      rc::fail
    );
}

private void list(RoutingContext rc) {
  HttpServerResponse response = rc.response().setChunked(true);
  database.retrieve() // Observable (async)
    .subscribe(
      p -> response.write(Json.encode(p) +" \n\n"),
      rc::fail,
      response::end);
}
```

Orchestrating remote interactions

Sequential composition

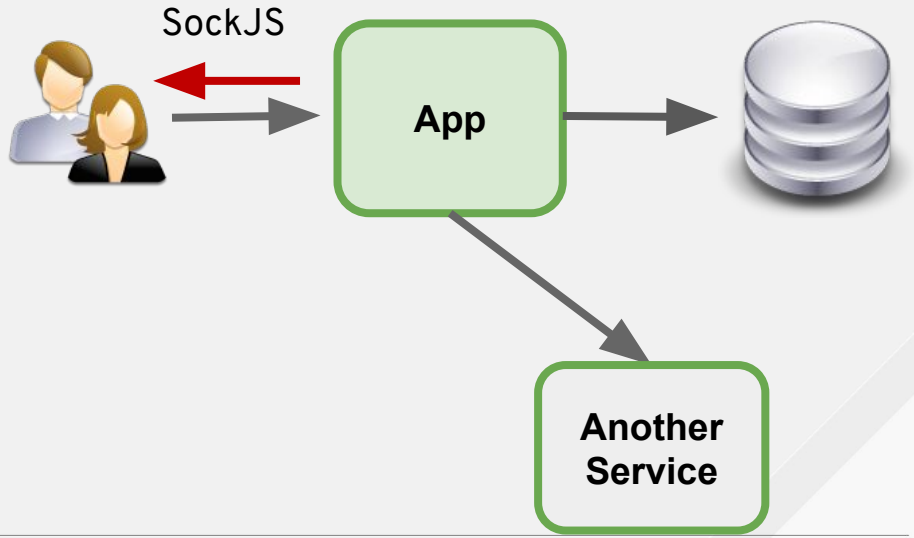
```
WebClient pricer = ...
HttpServerResponse response = rc.response().setChunked(true);
database.retrieve()
   .flatMapSingle(p ->
      webClient
         .get("/prices/" + p.getName())
         .rxSend()
         .map(HttpResponse::bodyAsJsonObject)
         .map(json -> p.setPrice(json.getDouble("price")))
   )
   .subscribe(
      p -> response.write(Json.encode(p) + " \n\n"),
      rc::fail,
      response::end);
```



Push data using event bus bridges

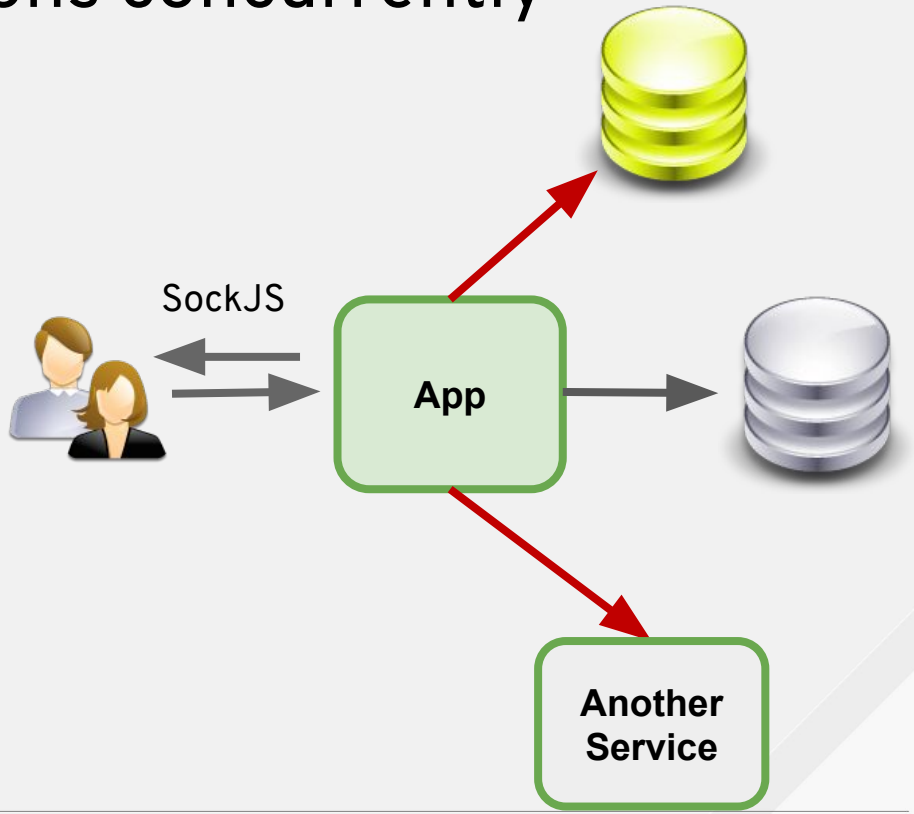
Web Socket, SSE...

```
String name = rc.getBodyAsString().trim();
database.insert(name)
   .flatMap(...)
   .subscribe(
      p -> {
         String json = Json.encode(p);
         rc.response().setStatusCode(201).end(json);
         vertx.eventBus().publish("products", json);
      },
      rc::fail);
```



Executing several operations concurrently

```
database.insert(name)
  .flatMap(p -> {
      Single<Product> price = getPriceForProduct(p);
      Single<Integer> audit = sendActionToAudit(p);
      return Single.zip(price, audit, (pr, a) -> pr);
  })
  .subscribe(
      p -> {
          String json = Json.encode(p);
          rc.response().setStatusCode(201).end(json);
          vertx.eventBus().publish("products", json);
      },
      rc::fail);
```



Vert.x + RX

RX-ified API

- *rx* methods are returning Single
- ReadStream provides a *toObservable* method
- Use RX operator to combine, chain, orchestrate asynchronous operations
- Use RX reactive types to be notified on messages (*Observable*)

Follows Vert.x execution model

- Single-threaded, Event loop
- Provide a RX scheduler

What you can do with it

- Messaging (event bus), HTTP 1 & 2 client and server, TCP client and server, File system
- Async data access (JDBC, MongoDB, Redis…)

The path to better systems

Is Reactive Programming all you need ?

Reactive Programming

- Provides an elegant way to deal with asynchronous operation
- Vert.x provides an execution model (event loop) + the different network and utilities bricks - all integrated with RX-apis

Other solutions

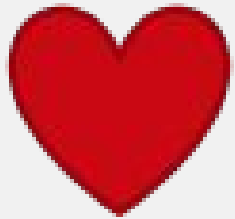
- Kotlin: Coroutine
- Java with Quasar: Continuation (vertx-sync)

It’s not enough !

- Reactive systems is not only about async
- **Resilience + Elasticity => Responsive**

All you need is (reactive) love

Reactive Systems + **Reactive Programming** =



Don’t let a framework lead, you are back in charge


clement.escoffier@redhat.com


@clementplop


@vertx_project


https://groups.google.com/forum/#!forum/vertx


https://developers.redhat.com/promotions/building-reactive-microservices-in-java

THANK YOU


plus.google.com/+RedHat


facebook.com/redhatinc


linkedin.com/company/red-hat


twitter.com/RedHatNews

youtube.com/user/RedHatVideos