Scalable Web Systems

Integration Part 1

Overview

1. What is integration?
2. Ideal Integration Technology
3. The Shared Database
4. Synchronous vs Asynchronous
5. Orchestration vs Choreography
6. Remote Procedure Calls
7. REST
8. Implementing Asynchronous Event-Based Collaboration
9. Services as State Machines
10. Reactive Extensions

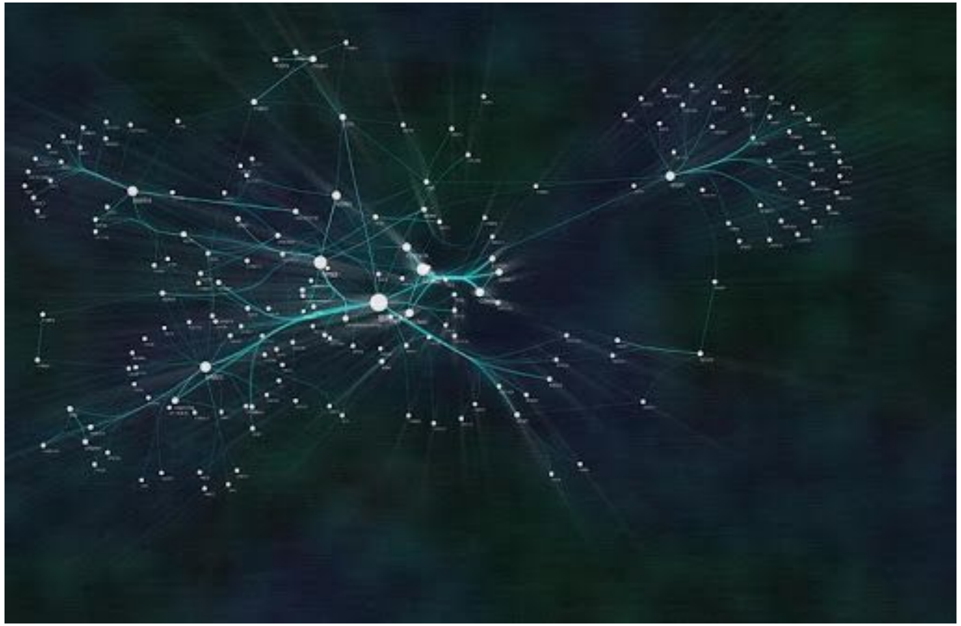
What is Integration?

What is integration?

We need a way for microservices to communicate and collaborate in order to architect an evolutionary system.

Integration consists of 2 important parts:

1. The choice of technology used to allow microservices to communicate with each other.
2. The architecture of how communication is conducted - synchronous vs asynchronous.



Ideal Integration Technology

There is a bewildering array of options out there for how one microservice can talk to another. But which is the right one: SOAP? XML-RPC? REST? Protocol buffers?

We will talk about these soon, but before we do, let's think about what we need to think about and how it relates to scaling an application.

These are *principles of integration*.

#1 Avoid Breaking Changes
Pick integration technology that ensures that changes to a microservice will not impact consumers.

#2 Keep Your APIs Technology-Agnostic
Avoid integration technology that dictates what technology stacks that can be used to implement microservices.

#3 Make Your Service Simple for Consumers
Allow clients full freedom in their technology choice and make it easy to use.

#4 Hide Internal Implementation Details
Consumers bound to internal implementation increases coupling. If an internal change to a microservice requires a consumer to change - the wrong decision was made.

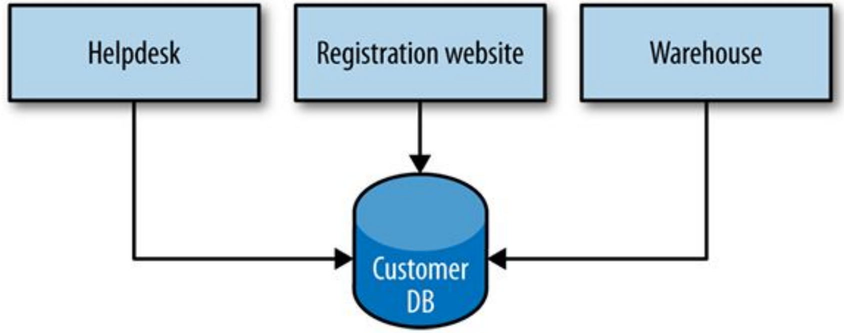
The Shared Database

#1 Avoid Breaking Changes.
#2 Keep Your APIs Technology-Agnostic
#4 Hide Internal Implementation Details

The most common/traditional form of integration is database (DB) integration.

It is simple and fast to set up and easy to work with. **Advantage:** if you want to make a change in your data communication, then change it! Visible to all microservices!

But, it violates #1, #2, and #4 as the ideal choice for integration technology. Quick and easy is not always the ideal choice and can lead to long term consequences.



Do not do this!

Synchronous vs Asynchronous

One of the most important decisions to make in how services collaborate!

Yes, it seems arbitrary - but, it will dictate your entire architecture.

Synchronous Communication: a call is made to a remote server and blocks until the operation completes.

Asynchronous Communication: the caller doesn’t wait for the operation to complete before returning, and *may not even care* whether or not the operation completes at all!

Synchronous vs Asynchronous

Synchronous

- Easier to reason about
- Know when things completed or not

Asynchronous

- Can be hard to reason about
- Very useful for long running jobs and useful for low latency / user response time

These two different modes of communication can enable two different *idiomatic* styles of collaboration.

- **Request/Response:** a client initiates a request and waits for the response. Both modes of communication can work with this.
- **Event-Based:** collaboration is inverted, a client says *this thing happened* and expects other "parties" to know what to do. Logic is decentralized. Highly decoupled. Events that are important to consumers are subscribed to. Asynchronous communication is better at this one.

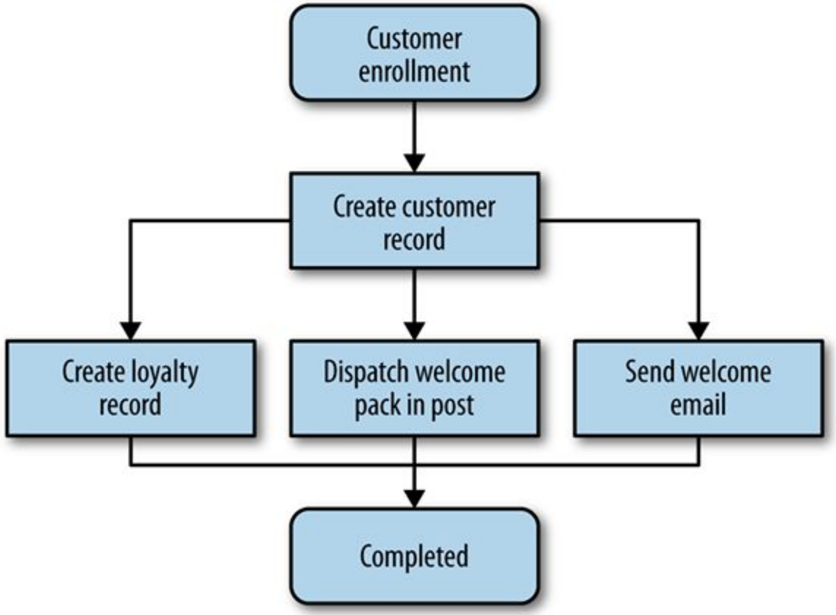
Orchestration vs Choreography

Example:

1. A new record is created in the loyalty points bank for the customer.
2. Postal system sends out a welcome pack.
3. A welcome email is sent to the customer.

Orchestration: we rely on a central brain to guide and drive the process.

Choreography: we inform each part of the system of its job, and let it work out the details, like dancers all finding their way and reacting to others around them in a ballet.



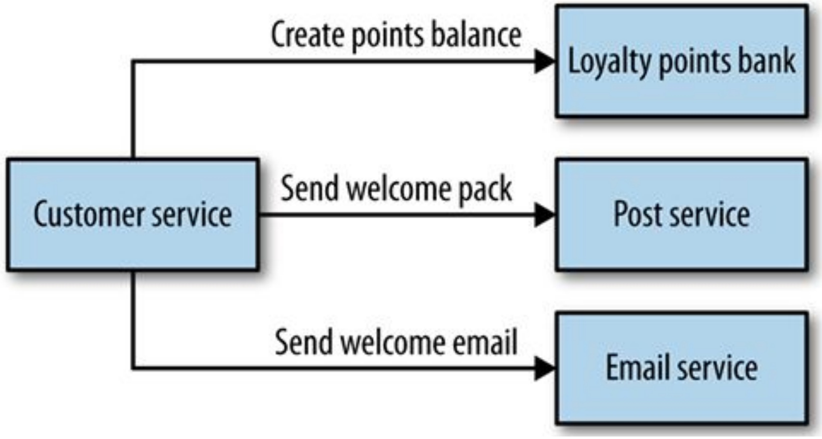
Orchestration vs Choreography

Example:

1. A new record is created in the loyalty points bank for the customer.
2. Postal system sends out a welcome pack.
3. A welcome email is sent to the customer.

Have our customer service act as the “central brain”. On creation it talks to the loyalty points bank, email service, and postal service through a series of request/response.

This work flow can be easily modeled directly in code using synchronous communication.



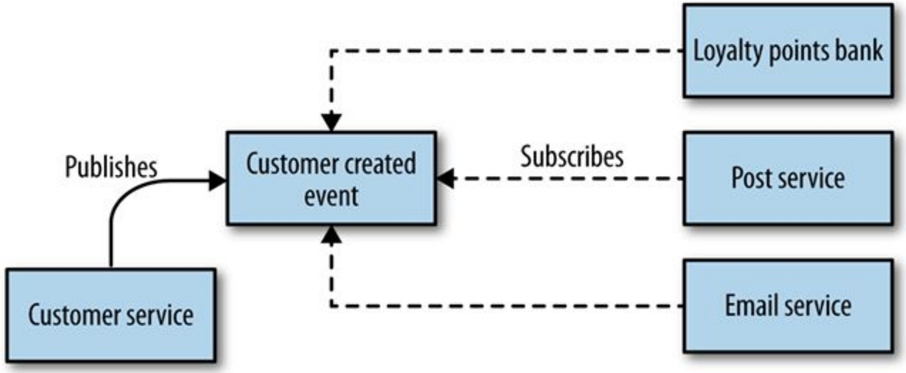
Orchestration vs Choreography

Example:

1. A new record is created in the loyalty points bank for the customer.
2. Postal system sends out a welcome pack.
3. A welcome email is sent to the customer.

Customer service emits events in an asynchronous manner, “customer created”.

The email service, postal service, and loyalty points bank subscribe to these events and react accordingly.



Remote Procedure Calls (RPC)

Remote procedure call refers to the technique of making a local call and having it execute on a remote service somewhere.

A definition of the calls are written and server/client stubs are generated.

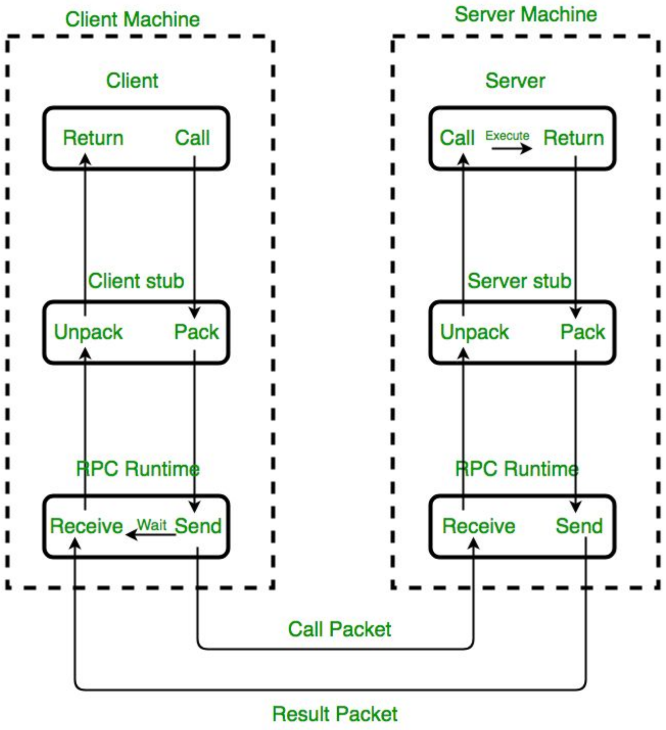
Many of these technologies are binary in nature

Java RMI, Thrift, or protocol buffers

But, they may not be

SOAP uses XML and some

implementations are network protocol specific



Implementation of RPC mechanism

RPC Pros and Cons

Pros

Remote calls “look like” local calls.

No need to worry about communication protocols - simply write the definition and fill in the blanks.

Easy to ramp up developers as they need not know about the intricate details of the network - they can still think in terms of objects and methods.

Cons

RPC hides too much. Naive use of RPC may have performance penalties. In fact, you may not be able to get around it.

Networks are not reliable. Things fail - much harder to do with RPC.

Because of this you simply can’t design a remote API the same way you design a local one.

It is tightly coupled to a technology and can break easily if the API is changed!

REST

REpresentational State Transfer (REST) is an architectural style inspired by the Web.

Most important is the concept of resources.

How a resource is shown externally is completely decoupled from how it is stored internally.

REST itself doesn’t really talk about underlying protocols, although it is most commonly used over HTTP.

Some of the features that HTTP gives us as part of the specification, such as verbs, make implementing REST over HTTP easier, whereas with other protocols you’ll have to handle these features yourself.

REST and HTTP

HTTP itself defines some useful capabilities that play very well with the REST style. For example, the HTTP verbs (e.g., GET, POST, and PUT) already have well understood meanings in the HTTP specification as to how they should work with resources.

The REST architectural style actually tells us that methods should behave the same way on all resources, and the HTTP specification happens to define a bunch of methods we can use.

HTTP also brings a large ecosystem of supporting tools and technology.

Hypermedia and the Engine of Application State

Another principle introduced in REST that can help us avoid the coupling between client and server is the concept of hypermedia as the engine of application state.

Hypermedia is a concept whereby a piece of content contains links to various other pieces of content in a variety of formats (e.g., text, images, sounds).

With hypermedia controls, we are trying to achieve the same level of smarts for our electronic consumers.

```
<album>
  <name>Give Blood</name>
  <link rel="/artist" href="/artist/theBrakes" /> ❶
  <description>
    Awesome, short, brutish, funny and loud. Must buy!
  </description>
  <link rel="/instantpurchase" href="/instantPurchase/1234" /> ❷
</album>
```

In this document, we have two hypermedia controls. The client reading such a document needs to know that a control with a relation of artist is where it needs to navigate to get information about the artist, and that instant purchase is part of the protocol used to purchase the album.

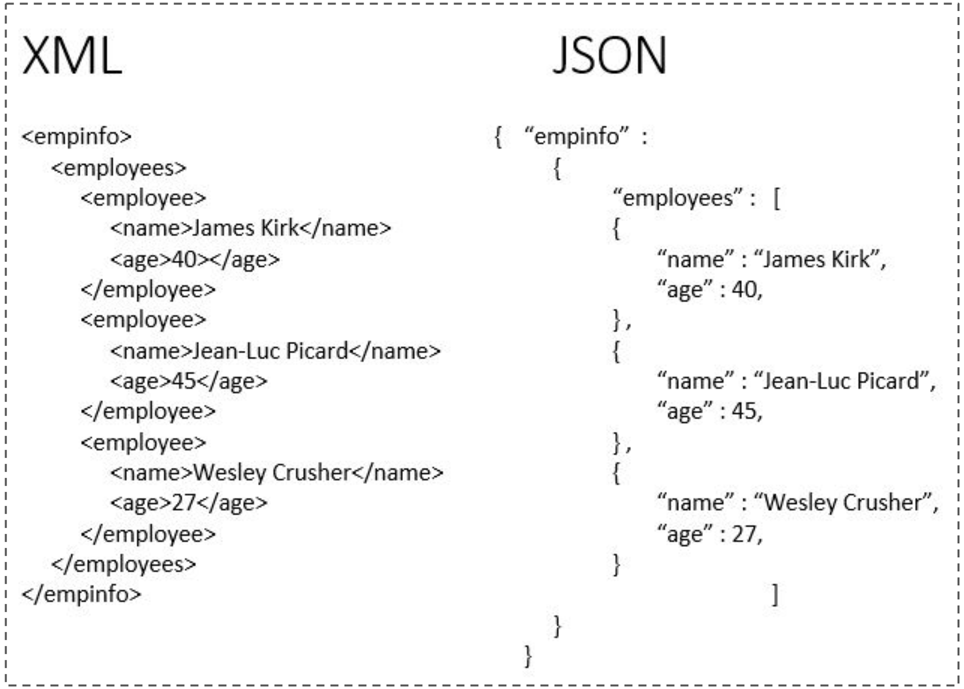
JSON, XML, or something else?

The use of standard textual formats gives clients a lot of flexibility as to how they consume resources.

REST over HTTP lets us use a variety of formats.

XML used to be the defacto standard, however, JSON has taken over more recently.

There are pros and cons of both formats. For example, XML is more of a standard that defines the **link** control (like hypermedia), but JSON is more free-form and less verbose.


XML

```
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40></age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

JSON

```
{   "empinfo" :
      {
          "employees" :  [
          {
              "name" : "James Kirk",
              "age" : 40,
          },
          {
              "name" : "Jean-Luc Picard",
              "age" : 45,
          },
          {
              "name" : "Wesley Crusher",
              "age" : 27,
          }
                          ]
      }
  }
```

Beware Too Much Convenience

As REST has become more popular, so too have the frameworks that help us create RESTFul web services. However, some of these tools trade off too much in terms of short-term gain for long-term pain.

Example: Some frameworks make it very easy to couple the database with in-process objects that are exposed externally.

This violates the following key principles of integration technology:
#1 Avoid Breaking Changes
#2 Keep Your APIs Technology-Agnostic
#4 Hide Internal Implementation Details

Downsides to REST over HTTP

- You are pretty much on your own to define the application protocol over HTTP
- Some web server frameworks do not support all HTTP verbs well
- Performance may also be an issue - binary is probably the best
- HTTP is not great for low-latency communication in contrast with alternative protocols that are build directly on top of Transmission Control Protocol (TCP). WebSockets is a good example of using TCP directly.
- Server-to-server communication may prefer a lower latency protocol - HTTP is simple, but it is bulky.

Despite these disadvantages, REST over HTTP is a sensible default choice for service-to-service communication because often the advantages far outweigh the disadvantages.

Implementing Asynchronous Event-Based Collab.

#1 Technology Choices

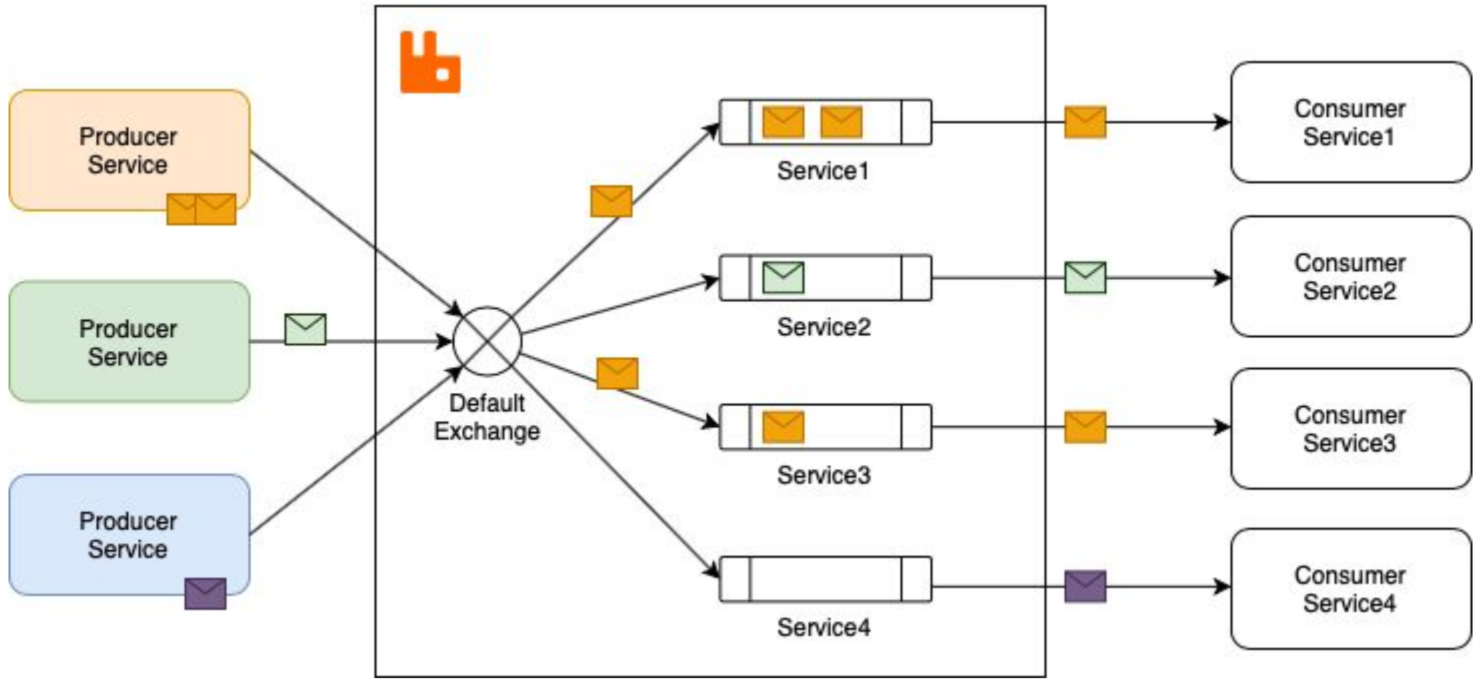
1. Need a way for microservices to emit events
2. Need a way for consumers to know those events occurred.

It is useful to use a “message broker” (e.g., RabbitMQ) to allow services to “publish” events and “subscribe” to events.

These systems are designed to scale and be resilient, but it doesn’t come for free.

There are several things to consider (see the book). It is often useful to simply use an asynchronous request/response model using JSON.

RabbitMQ Architecture



Implementing Asynchronous Event-Based Collab.

#2 Complexities of Asynchronous Architectures

Some of this asynchronous stuff seems fun, right?

Event-driven architectures seem to lead to significantly more decoupled, scalable systems. And they can. But these programming styles do lead to an increase in complexity.

How to handle the complexity?

1. Need to have good monitoring in place
2. Need a way to trace requests across microservice boundaries (correlation IDs)

Services as State Machines

Services should be built around bounded contexts.

Each service *owns* all logic associated with behavior in that context.

When we need to change the *state* contained in a service we communicate to that service and have that service make the change (no shared database!).

If the change of state of a service is leaked out of that service we lose **cohesion**!

The state of a service should only exist within that service and should not be changed outside of that service.

Reactive Extensions

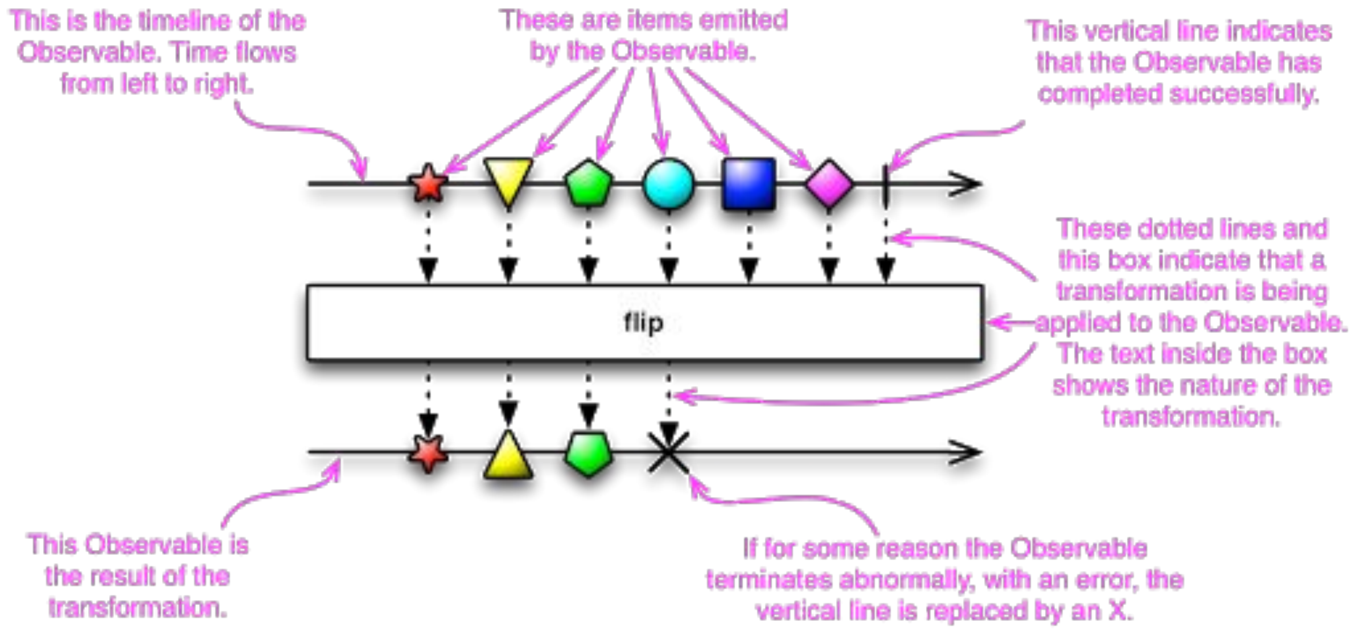
Reactive extensions are a mechanism to compose the results of multiple calls together and run operations on them. The calls themselves could be blocking or nonblocking calls.

Reactive extensions invert traditional flows. Rather than **asking for some data**, then performing operations on it, you **observe the outcome** of an operation (or set of operations) and react when something changes

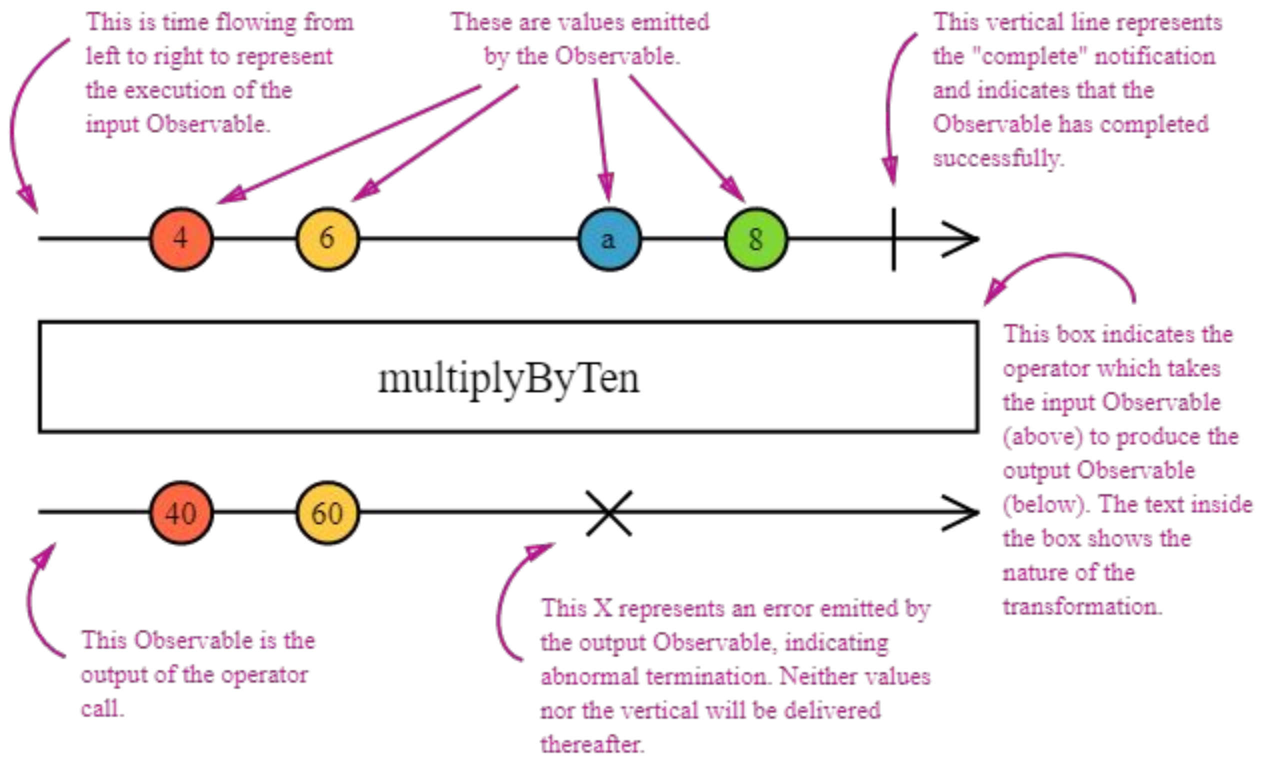
Reactive components received some active attention over the recent years.

The Scala programming language and its libraries and frameworks pioneered a lot of this work in recent years as it has functional programming at its heart.

Reactive Extensions


This is the timeline of the Observable. Time flows from left to right.
These are items emitted by the Observable.
This vertical line indicates that the Observable has completed successfully.
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.
flip
This Observable is the result of the transformation.
If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Reactive Extensions



This is time flowing from left to right to represent the execution of the input Observable.

These are values emitted by the Observable.

This vertical line represents the "complete" notification and indicates that the Observable has completed successfully.

This box indicates the operator which takes the input Observable (above) to produce the output Observable (below). The text inside the box shows the nature of the transformation.

This Observable is the output of the operator call.

This X represents an error emitted by the output Observable, indicating abnormal termination. Neither values nor the vertical will be delivered thereafter.