**1. INTRODUCTION TO REACTIVE SYSTEMS**

Reactive Architecture aims to provide software that remains responsive in all situations. Reactive Systems build user confidence by ensuring that the application is available whenever the users need it. In this course we will discuss what it means to be Reactive. We will outline the principles that are used to build highly responsive systems.

**Why do we need reactive architectures?** The software installations increase day by day. It is really common to go to a website/application and see “application in maintenance, please try again later” 🡪 developers may run updates or databases scripts on the software, we still need to do those operations but blocking users for long time is no more acceptable 🡪 we need deploy phase, but we must find different ways to do it. A real change: modern users have come to rely on software to do their jobs and to live day-to-day; think about the effect of a server that people use every day offline for few hours! Another thing to consider is how long this software take to respond and how this impact our day-by0-day usage.

The result is that unresponsive software leaves users unsatisfied 🡪 “time is money”. If we have software that is slow and unavailable is useless, the **responsiveness is the key!** The primary goal of reactive architecture is to provide an experience that is responsive under all conditions.

**The Goal**: **the application must scale as the number of the users increases**; on the other hand we must be sure the application consumer only the resources that are necessary at a certain time 🡪 for instance, an e-commerce website will be more under pressure by users during vacations such as Christmas, in that period we want to use the maximum flow and in less-stressed periods we want to scale back because using all the resources we have is expensive when useless. When failures happen, they should have the minimum impact on the user, the ideal is that they do not have effect, the user does not even notice but there are scenarios where this is not possible. When that happens, we want that effect on the user to be as small as possible (e.g. the whole app not available has a greater effect than just a part of the app not available).

They way we achieve **scalability** and **failure handling** is by making sure that our system can be distributed across tens hundreds or even thousands of machines. That’s not going to solve all problems but if we can build our system to scale across these tens, hundreds or thousands of machines, it gives us a lot of benefits: scalability problems might be largely solved (using more machines to support additional load); failure handling is partially solved because we can lose machines but the remaining ones can continue to operate and hopefully continue to serve our clients’ needs.

Then we want to make sure we maintain a **consistent level of quality and responsiveness** despite all these things 🡪 the application’s behaviour under 10 or 10 thousand users should be quite the same, there will be of course little differences. **If we can maintain this consistent quality level, we have built a reactive system**.

**THE REACTIVE PRINCIPLES**

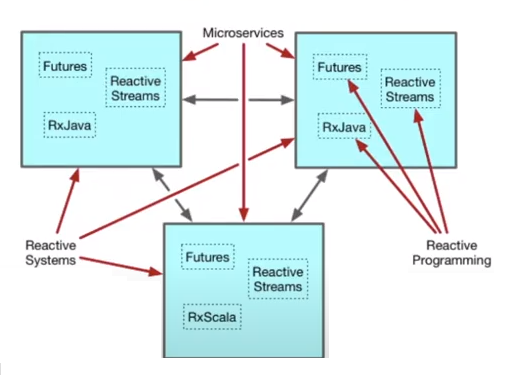
Reactive Manifesto is a document created in response to companies trying to cope with changes in the software landscape. There were multiple groups independently developed similar patterns for solving similar problems. Aspects of a Reactive Manifesto were individually recognized by these groups. The Reactive Manifesto attempts to bring these common ideas into a single set of principles:

* **Responsive**: a reactive system consistently responds in a timely fashion. The most important principle, all other principles point to this one. Responsiveness is the cornerstone of usability, it is the key component, the key principle. If you can create a responsive system without facing up resilience, elasticity and message driven it would be fine, but the reality is that we cannot do that.
* **Resilient**: a reactive system remains responsive, even when failure occurs. Resilience provides responsiveness despite any failure. This is achieved though a number of techniques including:
  + *Replication*: we have multiple copies.
  + *Isolation*: services can function on their own, they do not have external dependencies.
  + *Containment*: a consequence of isolation: if there is a failure it does not propagate to another service because it is isolated).
  + *Delegation*: recovery is managed by an external component because the system may be not enough reliable to handle the failure (if the system goes down, it cannot restart itself because it is down).
* **Elastic**: a reactive system remains responsive, despite changes to system load. If the system requires more computational power, it can scale up and if the system requires less computational power, it can scale down. If we have elasticity, we can exploit auto scaling techniques to support it: scale up/down. The goal is having zero contention and no central bottlenecks.
* **Message Driven**: a reactive system is built on a foundation of async non-blocking messages. Responsiveness, Resilience, Elasticity are supported by a Message Driven architecture. It provides loose coupling, isolation, location transparency. Resources are consumed only while active, then they are released.

Immagine che contiene linea, testo, Blu elettrico, diagramma

Descrizione generata automaticamente

**REACTIVE PROGRAMMING**



**Reactive Systems** apply the reactive principles at the architectural level, and they are represented by **microservices** (the 3 box) which are components designed at the architectural level to be reactive. Inside every microservice there are other elements: Futures, Reactive Streams, RxScala, RxJava 🡪 all these are programming techniques. **Reactive Programming** can be used to support the construction of Reactive Systems, it supports breaking problems into small, discrete steps. Those steps are executed in async/non-blocking fashion, usually via a callback mechanism. ATTENTION: use Reactive Programming does not mean we built a Reactive System; this does not mean we are using reactive architecture principles. All major architectural components (microservices) interact in a reactive way, and they communicate each other using async/non-blocking messages. [Microservices are an example of something that is built using Reactive Systems].

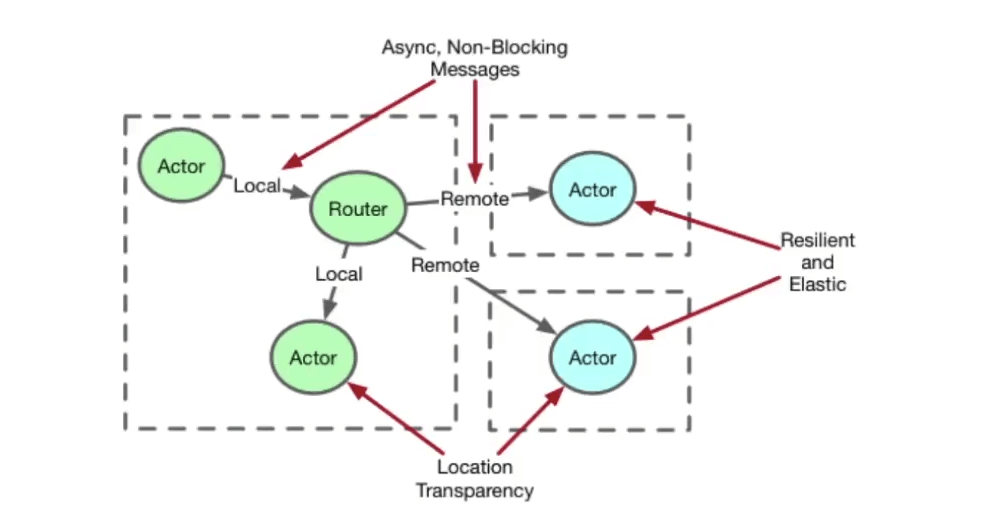
**How can we build a system that is not reactive using Reactive programming?** All you have to do is build a system and deploy it onto one node. If it is deployed onto one node then there is no way to be resilient. If that node crashes the whole system is lost. Furthermore, if you build the system in such a way that it leverages a local cache and there is no way to keep that local cache in sync with multiple nodes, then it becomes very difficult to scale up because when you scale up you now have a cash consistency problem.

**REACTIVE SYSTEMS AND THE ACTOR MODEL**

The **Actor Model** is a programming paradigm that supports construction of Reactive Systems. It is message-driven, all communications between actors are done using async/non-blocking messages. Abstractions provide elasticity and resilience. It can be used to build reactive software. On the JVM we have Akka that implements the actor model, Akka is the foundation of reactive tools like Lagom and Akka Streams.

All computation occurs inside of actors 🡪 in practical we have a combination of different actors, and all the computation will occur inside one of those actors or across many of those actors. Each actor has a unique address, and every actor communicates with the others only through asynchronous messages.

The message-driven nature of actors provides us something called “**Location Transparency**”: our actors communicate with the same technique regardless of location, this means that local versus remote is mostly about configuration 🡪 actors have no knowledge about the location of where the message is going to go. This allows actors to be both resilient and elastic: we can deploy those actors across multiple pieces of hardware which means they are resilient if one of those pieces of hardware fails; it also allows to be elastic because if we have a high/low level of load, we can simply add/remove routers on more pieces of hardware giving us elasticity.



Location Transparency should not be confused with **Transparency Remoting** which tries to take remote calls and make them look like local calls, it hides the fact that you are making remote calls and as a result it can hide potential failure scenarios. Location Transparency takes the opposite approach: it makes local calls look like remote calls. That means that you are always assuming that you are making remote calls which means we have to assume a remote failure scenario could occur (the message could not be delivered).

The Actor Model is important in many ways. There are many Reactive Programming tools. Most support only some of the Reactive Principles. You often have to combine different technologies to build a Reactive System (programming tools + …). The Actor Model provides facilities to support all of the Reactive Principles: message-driven by default, location transparency to support elasticity and resilience through distribution, elasticity and resilience provide responsiveness.

It is possible to build a Reactive System without Actors. Components are *added on* rather than being *built in*. Requires additional infrastructure such as: service registry, load balancer, message bus. Result will be Reactive at the large scale, not necessarily the small.

**2. DOMAIN DRIVEN DESIGN**

…