**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**

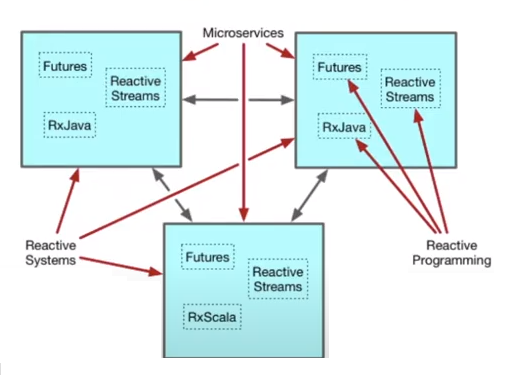
Reactive Manifesto is a document created in response to companies trying to cope with changes in the software landscape. There were multiple groups independently developed similar patterns for solving similar problems. Aspects of a Reactive Manifesto were individually recognized by these groups. The 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].

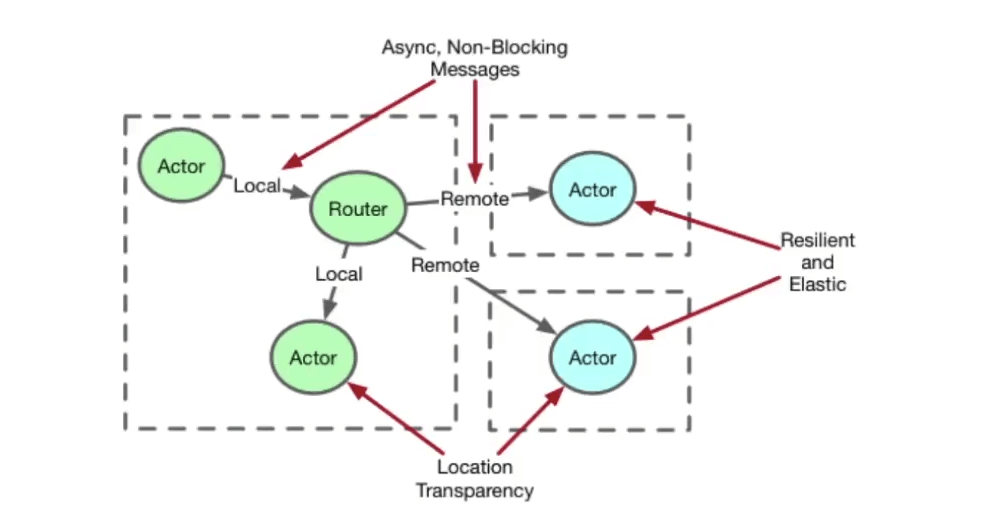
**How can we build a system that is not reactive using Reactive programming?** All you have to do is build a system and deploy it onto one node. If it is deployed onto one node then there is no way to be resilient. If that node crashes the whole system is lost. Furthermore, if you build the system in 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**

Domain Driven Design is an architectural approach, used to design large systems. The guidelines of domain driven design are highly compatible with the ones of reactive architecture. Many of the goals of the two are really similar and that’s why the two terms are often named together. Domain Driven Design originates from a book called in the same way where the goal is creating a software implementation that is based on an evolving model that is understood by the domain experts. One of the key goals of domain driven design is to **take a large system (or a large domain) and break it into smaller pieces** 🡪 the problem that we have with large domains is that there are several rules and for this they are hard to manage 🡪 break the system into smaller pieces give us a way to determine boundaries between those smaller pieces within the large domain. reactive microservices specifically, have a similar goal: they need to be separated along clear boundaries.

In the case of reactive microservices those boundaries need to be asynchronous: each microservice has to have a clearly defined API and a specific set of responsibilities. If we don't know what the responsibilities of the microservice are then it's going to be very hard to build it and design it properly. **The trick comes when we try to determine what those boundaries are**. There's no clearly defined technique that will just give us the right answer all of the time however this is **where domain driven design can help us**. It does give us a set of guidelines and a set of techniques that we can use to try to help us break larger domains into smaller domains. Because of that we can take that logic and apply it to microservices to come out with something similar. If you look at reactive frameworks such as Lagom for example Lagom on is built with domain driven design in mind. It leverages some of its terminology.

The key here that we need to realize is that domain driven design can be used in the absence of a reactive microservice or a reactive architecture. And you can build reactive architectures without domain driven design but because the two are very compatible--and they are, they do have similar goals--you'll often find them use together.

**WHAT IS A DOMAIN**

**A domain refers to a business or idea that we are trying to model**. Typically, when we build software there is some sort of concept or some sort of business that we're trying to model, and that is our domain**. Experts in the domain are therefore people who understand that business or that idea not necessarily the software**.

So, if you're building banking software, for example, then you're gonna have, you know, your bankers are gonna be domain experts. Your domain first off would be banking. You will have your domain experts will be bankers, tellers, any of the bank managers are potentially domain

experts. So, these are all people with expertise in the domain not necessarily the software.

In our restaurant example then it's going to be people like your servers and your restaurant managers, your cooks, your chefs, your dishwashers. All of those people are potential domain experts.

So, **the goal of domain driven design then is to build a model that the domain experts can understand**. We're trying to build a model that we can show to these experts who, again remember they don't understand the software, we want to be able to take that model, present it to them and have them understand what we're talking about. This means that we need some sort of way of communicating.

**It's important to understand that the model is not the software**. That's a common misunderstanding. People will look at the software and say that "well this is my model." The model is not the software. **The model represents our understanding of the domain**. **The software is an implementation of the model**. We can implement that model in other ways, it doesn't have to be in software (e.g. using diagrams, writing documentation). The software should be implemented in such a way that it reflects the model. And as a result of that, if we talk about the model, we are talking about the software.

Since the two terms are correlated, we need a language that both the domain experts and the software developers can understand and can communicate with. Now this language is what we call the **ubiquitous language**. In domain driven design it is **a common language that enables communication between the domain experts and the developers**. Terminology in the ubiquitous language comes from the domain experts. We go and we talk to the domain experts as developers, and we **try to understand the words that they use, and we use those words in our model**. In turn we of course become domain experts because we are slowly learning about the actual domain.

Now words originate in the domain and are then used in software. We want to try to avoid going in the other direction. **We want to try to avoid taking software terms and introducing them into the ubiquitous language and forcing our domain experts to use them**. There are exceptions to that; sometimes that becomes necessary. Maybe there's a there's a particular abstraction that the domain experts don't have a word for, but we need it in the software for some reason. And so, we have to introduce that and then maybe we should talk to the domain experts about it. So, it's always very important to keep that communication going and to do it in such a way that it allows you to maintain a conversation. If we start using highly technical terms like databases and database terms, maybe event bus and entity, and all of these types of things, our domain experts are gonna get lost because they don't know what these things are.

They know words like order, server, menu, cook, dishes. These are words that somebody who works in a restaurant can understand.

The idea here is that the domain experts and the software developers should be able to have a conversation about the software without resorting to software terms. We can achieve that if we stick to the ubiquitous language and if we follow the rules of domain driven design.

**DECOMPOSING THE DOMAIN**

In this section we are going to talk about **how to break large domain into smaller pieces**. The first thing to realize is of course that business domains are often large and complicated. It's pretty rare that we find a business domain that is really tight and easy to understand without breaking it down any further because those domains contain ideas, actions, rules that interact in very complex ways.

If we look at our restaurant domain you might initially think "a restaurant, pretty simple, you got servers, you got cooks, you got these different people in the restaurant; that's not that hard. But

when you actually start digging into it, you realize there's a lot of objects that are involved here. You've got menus, you've got orders, cooks, servers, reservations, you've got dishwashers, drinks, plates, tips, bills, cash, credit, debit, delivery, this goes on and on. That's not even talking about things like inventory management or any of these other complexities, you know your payroll, all of those types of things. Obviously, this can get really big, really fast. If we were to try

to model this as one domain it would get pretty muddled.

So, what we do in domain driven design is we take our large domains, and we separate them into sub domains. Our **sub domains are created by grouping related ideas and actions and rules into a separate sub domain**. If we look at our restaurant again, we had all those different ideas but some of them are related. For example, we can look at reservation: table, customer, time, location, those are all specifically related to creating a reservation. We might look at that and say well perhaps there's a reservations sub domain that deals just with creating reservations within our restaurant. We can kind of extract that out into a separate set of terms and a separate ubiquitous language. In fact, we can extract it out into a separate model. What **we may realize though is that some parts of that may exist in multiple subdomains**. Customer, for example, is probably going to exist in other subdomains as well. It's not strictly limited to reservations. We have to recognize the fact that some of these concepts can actually cross multiple subdomains. It's also important to realize that **those shared concepts such as customer may not be identical initially**. Because of that it's important to avoid the temptation to abstract. We may have a temptation to take the concept of customer and say that well we just have one concept of customer. And it always looks the same no matter what don't sub domain we're working in. But the reality is customer in say the reservation sub domain may have details that are unimportant in other parts of the business. Or it may not need details that are important in other parts of the business.

Because of that each **sub domain essentially ends up having its own ubiquitous language and model**. The language and model for a sub domain is what we call a **bounded context**. So as a result, we can say that the model of a customer in reservations is probably not the same as a model of a customer in other parts of the business. This is your bounded context: it's the combination of the model and ubiquitous language and subdomains or bounded context usually map in kind of a one-to-one relationship. You'll often see those terms used interchangeably. The most common term you'll see is bounded context. These are a good starting point for building reactive microservices. When we want to build a microservice, it's often a good idea to start looking at our bounded context. That may not be where we stop. We may take an individual bounded context and break it up into multiple microservices.

But at the very least we probably don't want to have all of our bounded contexts combined into a single service because that's building a monolith. Let’s think about how we can build micro services around a bounded context. Let's try to understand those bounded context a little bit better. From one bounded context to the next the meaning of a word may change dramatically.

In our restaurant, for example, we're talking to a server, the term order has a very specific meaning. When you talk to the server it means one thing. However, if you talk to somebody else in the restaurant, for example, the person who manages inventory, then order means something different. When you're talking to a server, an order is something that is very much client facing. The customer comes in, they place an order, and the restaurant provides them with food. In

that case, servers represent the customer. Whereas the restaurant represents the supplier for that order. Now if you flip it around and talk about inventory management, the roles actually change up completely; now the restaurant is the customer and there is some external supplier who is supplying the inventory supplies. It takes on a very different meaning depending on who you talk to in the restaurant. It's still an order and it may share some similarity with the other order but it's important to understand that **they are not the same order. You can't model them the same because within the context of the restaurant they mean very different things**.

**We also have to observe how the details of the model change**. In arestaurant when a kitchen is preparing an order, there are certain things theycare about. They need to know what's on the order. They need to know didthe person order the reactive ribs did they order the steak. Whatdid they order? Those types of details are important. They need to know anythingabout special instructions: did they want that steak medium-rare? That's important.There are other things they don't care about: they don't care about the price. Itmakes no difference to the kitchen what the price is. That matters to thecustomer obviously and it matters to the server or whoever's collecting the money.But, from the kitchen’s perspective, it's irrelevant. That's an area ofdetail where we don't need to provide the kitchen with all of that extrainformation they don't need. So, it'simportant to understand that **those details change depending on what part of the business you're working in**.

**How do we determine bounded contexts?** **How do we decide what is a bounded context and what isn't? Where do we decide to draw those those boundaries?** There's no universal answer. I can't just say we'll always dothis, and you'll get the right answer every time. There's nothing that workslike that but **there are some guidelines**. We want to consider human culture andinteraction. If you look at our restaurant there are **different areas** ofthe restaurant that are **handled by different groups of people**. **That starts to suggest a natural division and it might imply a separate bounded context**. Look for changes in the ubiquitous language. **If the language or meaning of that language changes it may suggest a new context**. Again, when we're talking about order, there's the order that theserver takes or there's the order that is used for inventorymanagement. Those two orders are different therefore they may exist in adifferent bounded context. Look for varying or unnecessary information.Employee ID is very important in an employee but it's meaningless in acustomer. So that again might imply that those pieces of informationbelong in different bounded contexts. Another thing to realize is that **strongly separated bounded contexts will usually result in smooth workflows**. Ifyou find yourself in a situation where you're trying to implement a particularworkflow and it's awkward, that may suggest that you've misunderstood yourdomain. Or you've broken up your bounded context incorrectly. If you find abounded context has too many dependencies, again it may be overcomplicated. You may have drawn your lines incorrectly. That's an areawhere you want to explore in a little more detail and try and figure out ifyou've done something wrong.

Traditionally domain driven design focused on the objects within the domain. In more recent times the techniques have evolved a little bit and now we talked a lot about event **first domain driven design which places the focus on the activities or events that occur in the domain**. **Rather than looking at those objects we look at what is happening to those objects** or what they are doing. Customer makes a reservation, that's an event or an activity. "Server places an

order" is an activity or an event. "Food is served to the customer"; these are all activities or events that occur in our domain. Using event **first domain driven design we look to define the activities first, then we group those activities to find logical system boundaries**. There is a particular technique called "event storming."

When defining our activities (or events), it can be useful to use a common notation to help keep your activities clear. For this exercise we will use **subject-verb-object** notation. Our subject represents whoever is doing the particular activity. Our verb is the action being done. Our object is the target of that action, i.e. the object that the action is being applied to. For example, “Host creates a Reservation”. Here, our subject is "Host", our verb is "creates" and our object is "Reservation". When defining these activities, it is important to think about how the business would operate in the absence of software.

Once we've separated our bounded context into these nice clean boundaries, we have a bit of a job ahead of us which is maintaining those clear boundaries, **maintaining the purity of those bounded contexts**. So, we need a technique or a set of techniques that allow us to do that.

One way that we do that is with something that we call an **anti-corruption layer**. When we have our bounded context, it's important to recognize that each bounded context may have domain concepts that are unique. If we take as example the reservations context and our customer context. Now there are certain aspects of a customer that are unique to a customer. They don't matter to the reservations address as an example of that. In addition, there are concepts that are not always compatible from one context to the next. What happens sometimes is you may end up with a concept, for example, that is called one thing in the reservations context and is called something different in the customer context. Or you may end up with situations where something is not compatible in such a way that it requires a translation of some kind. Maybe its recorded in different units or something like that and we need to translate it. So, the point is that you can end up with situations where things are not compatible between these bounded contexts. What we don't want is just come up with sort of an abstraction or some way to make it the same across all bounded contexts because what that does is it causes one bounded context to bleed into another. For example, if we just said well, we have a customer representation that includes all the details: it includes address, phone number, when they became a customer, you know, lots of details about that customer.

We could just use that representation of a customer everywhere and then we don't have to worry about that. But that creates **coupling that we want to try to avoid**. For example, if we were to do that and then we were to go in and change the structure of an address, maybe it was represented one way and then we realize is a better way to represent it, we include additional information or we remove some information. Well, if we've used that everywhere then now we need to go and we need to update the reservations context even though the reservations context doesn't care about the address. So, now we've created an **unnecessary coupling**, and we want to avoid that.

**The anti-corruption** layer allows us to avoid that. What it does is it looks at whatever that customer context representation is, and it translates it into a representation that is unique to the reservation service. It **strips out the unnecessary information** like address. If there's any kind of language translation that has to happen it does that. As a result, the reservations context only deals with the pure representation that it cares about. This prevents bounded context from leaking into each other, but it can also allow our bounded context to stand alone. If we have, for example, a **caching layer** inside of that anti-corruption layer, then what can happen is if the customer context disappears for some reason, then the anti-corruption layer still may have a cache of that information. And still may be able to operate despite the fact that the customer context is gone now.

**How is that anti-corruption layer implemented?** A common way to implement it is as an **abstract interface**. The abstract interface represents sort of a pure domain representation of the data. Then we have implementation of that interface which does the necessary translation. And that's sort of an infrastructure component that does that translation. So, this is in the situation where we have pure bounded contexts, our reservations context, our customer context, are fairly pure. We understand their boundaries fairly well. That's not always the case. Sometimes you have to interface with a legacy system. Now with the legacy system, the domain, may be messy or may be unclear. Nobody may have taken the time to understand the domain of that legacy system so in this case, an anti-corruption layer has the job of keeping the mess of that legacy system out of your pure bounded context. It prevents your domain from having to deal with that mess. Now in this case we have our anti-corruption layer in two places. We have it both in the reservations context and in the legacy system. You can see it in both places. Technically, you could put it in either place or both. **What I would generally recommend is anytime a pure microservice or a pure bounded context has to communicate with another external service of some, kind it should do so through an anti-corruption layer**. The reservations context should always have an anti-corruption layer to communicate with an external service. Now on the flipside, in your legacy service what you

may want to do is because of the complexity of it, you may want to expose an API that is just for the reservations context. Which then has a bit more of a pure representation. It's not totally pure but it's a bit more cleaned up from what your messy legacy system would typically be exposing. So, you might expose that in the legacy system in order to help but even when you do that I would still recommend having the anti-corruption layer on the reservations context side as well.

A context map. It's a way of visualizing bounded contexts and the relationships between them.

You can draw arrows to represent the relationship between different bounded contexts, saying it's a dependency, what kind of dependency is it.

**DOMAIN ACTIVITIES**

Within a domain, there's actually many different types of activities that we could be dealing with. We're gonna go into some specific categories of activities.

One of the types of activities that we deal with in domain driven design is called a **command**: it represents a request to perform an action. It's important to understand that because it's a request, **it's not something that has happened yet**. It is something that we are asking to happen in the future. And again, because it's a request **it could potentially be rejected**; we could choose not to proceed with whatever that request is. Commands are typically delivered to a specific destination. They usually have a specific recipient in mind whether that's a specific micro service or something else. And **when it is received, it will cause a change to the state of the domain**. After the command has been completed, the domain won't be in the same state that it was prior to issuing that command. Some examples of commands would be to "add an item to an order," "to pay a bill," "prepare a meal." Those are all requests to perform an action. We are not saying that the meal has already been prepared, we are asking someone to prepare the meal. The person could turn around and reject us and say "no I'm not gonna cook that" for any number of reasons.

In addition to commands, we also have **events**: a type of activity that occurs in the domain but now it represents an action that happened the past. A command is something that we are requesting to happen in the future. **An event is something that happened in the past**. Because it happened in the past, **it can't be rejected**. You can't say that event never happened because it did. It is part of history at this point. You can choose to ignore it. You can choose to do nothing with it. But you can't say that it never happened. So, you can't really reject it; **you can just choose to ignore it** or whatever you want to do. Events are often broadcast to many destinations. **So where a command is usually message going to a specific recipient, an event is often just sent to anybody who cares**. You will broadcast it to many different microservices or many different destinations. What an event does, where a command will cause a change in the domain, **an event actually records a change in the domain**, a change to the state. **It is therefore often a result of a command**. If you look at our examples, here we have "an item was added to an order" as opposed to the command which was "add item to an order." You'll also notice that where commands are always worded in such a way that you can see that it's a command, events are always worded as past tense. You can see that it's an event. "An item was added to an order," that's something that already happened. "A bill was paid" as opposed to "pay a bill." Again "a bill was paid" as something that happened in the past. "A meal was prepared." Again the critical thing here is that events happened in the past. They are indisputable. You cannot argue with the fact that it has already happened.

And finally, we have **queries**: they represent a request for information about the domain. As you'd expect with a query, **you always expect a response**. With a command or an event, that's not necessarily the case. Commands you often expect a response, basically something like "yeah I got your command." You may not actually get any details in that response, it's just "yes I got that and I will do that." Events, you may not get any response. Sometimes you will get an acknowledgment just again something like "yes I got that." With queries though you're always expecting some sort of detail back. You don't just want a "yes I got your query." You want an answer to your question. So a response is always expected. **It's usually delivered to a specific destination**. You're usually going to like have a specific microservice, for example, saying please give me information about this domain entity. The other thing is that **queries should not alter the state of the domains**. When we do a query, if we do that query multiple times, we should always get the same response assuming nothing else has changed. We should never get a situation where we issue a query and that query changes the state in some way. If we do change the state in some way it's not a query anymore it's now a command. So, some examples of queries "get the details of an order." We're not changing the order, we're just getting the details of it. "Checking if a bill has been paid." Again we're not marking the bill as having been paid, we're just checking to see if it has been paid. We don't change the state in any way.

**In a reactive system, the commands, events and queries represent the messages** that we talked about when we talked about it being a message driven approach. In our reservations context, for example, we are going to have commands such as "make reservation." We are going to have events such as "reservation made." And we are going to have queries like "get reservation." They basically form the API of a bounded context or a microservice. As a result, because we're aiming for this asynchronous message driven approach, these are often issued in an asynchronous fashion. When you say "make reservation" and provide it with the details, you don't immediately get back a response saying that reservation is now completed what you get back is maybe an acknowledgement saying "yes I got your request and I will process it later." That allows it to be asynchronous. You may not even get that back. You may just issue the request fully asynchronously and get no response back. And then it's up to you to watch for something like a reservation made event coming out the other side which would then indicate that your command has been successfully processed. That's what we mean when we say asynchronous and message driven. We do this in such a way that we don't have to wait for a response right away.

**DOMAIN OBJECTS**

Let's have a look now what types of objects we are going to see when we start diving in more deeply into a bounded context. One of the objects that we're going to encounter is what we call a **value object**: it is defined by its attributes. **The nature of a value object it has to be immutable**. We can say that the value object has to be immutable because any change to it causes it to become a different object. These are typically state containers. They contain data that is often passed between different components in a reactive system, but they can contain business logic as well. If you want to have some logic that extracts some information out of the address, that logic can exist inside of the address object. **In a reactive system messages are implemented as value objects**. We talked about the message driven architecture of a reactive system: these are the messages. They are implemented as value objects and then they are passed between different components in your reactive system.

To contrast that, we also have something called **entities**: it is defined by a unique identity like an ID or a key. What we're saying here is with **an entity**, first off, entities, unlike value objects, **can change**. They are mutable. The attributes can change, and it can still maintain its identity. Entities are the single source of truth for a particular ID. Entities are often workhorses for mutable state but also for business logic. They often contain a lot of business logic that says things like when I want to change your particular attribute this is how I do it and these are the rules associated with that change. **In reactive systems** if we're building using Akka, **Actors are an excellent model for entities**. They have a unique identity in the form of an address. They are mutable. You can change their state in a safe way. They make excellent representations of entities.

There's also a specific type of entity which we call an **aggregate**. An aggregate is a collection of domain objects bound to a root entity. Here we see a Person that consists of a Name, an Address and a Phone Number. That Name, Address, Phone Number and Person make up an aggregate. The root entity here, and it is always an entity, is called the aggregate root. The aggregate root in this case is the Person. Objects in an aggregate can be treated as a single unit. You can say that a Person consists of a Name, an Address and a Phone Number and those are all part of a single larger whole. **Access to objects in the aggregate have to go through the aggregate root**. Remember, an entity is the single source of truth for any particular piece of data. Same thing with an aggregate. It is the single source of truth. If you want to know the truth about that Person you have to go to that aggregate root of Person. You can't go directly to the database because the database is not the source of truth. The aggregate is the source of truth. Transactions should not spend multiple aggregate roots. If you find yourself in a situation where you need to do a transaction that actually crosses aggregate roots, then you've either defined your aggregate roots incorrectly or there's a problem with your transaction and you might have to rethink it a little bit. **Aggregates actually make really good candidates for distribution in reactive systems**. When we start looking at other things later on such as cluster sharding in Akka, for example, that is a good place to start looking to distribute with cluster sharding is your aggregates. Lagom uses the concept of a persistent entity which is backed by a cluster sharding. It distributes those entities or those aggregates across a cluster. That's a good starting place to start distributing.

**But how do we determine aggregate roots?** Aggregate roots are very important in a domain driven design system but figuring out what they are is not always straightforward. There's a few problems. One is **the aggregate root can be different from one context to the next**. In Reservations we've decided that the aggregate root is going to be Reservation but if we build a Loyalty program while Loyalty has nothing to do with Reservations so the aggregate root there might be something different like a Customer for example. In addition, you can encounter situations where **a context may require multiple aggregate roots**. It's not common, it's far more common to see a single aggregate root per bounded context but it's not always the case. So, some questions that you have to consider are: Is the entity involved in most of the operations in that bounded context? In the Reservations context I think we can clearly say that almost every operation, probably every operation, is going to involve a Reservation in some way. It probably makes sense that that would be a candidate for our aggregate root. Another question to ask is: If you delete the entity does it require you to delete other entities? Well if I delete their Reservation then I don't really need the Time for the Reservation; that's not really that important. Arguably I don't really need the Customer for the reservation anymore either. You might say yeah but if you delete the Reservation, doesn't mean the person isn't still a Customer. That's true but it does mean that if you delete all of the Reservations for that Customer, there's still a Customer but they're not a Customer who has Reservations. So, from a purely Reservation context we can say that that particular Customer doesn't matter anymore because they don't have any Reservations. In fact, deleting the Reservation could actually delete the Customer. Will a transaction span multiple entities? If the answer to that question is yes then we can safely say that we have got the wrong aggregate root. Because again a transaction should not span multiple aggregate root. Those are the types of questions that we can use when trying to determine the aggregate root. That should help us build kind of a candidate list. Then from there it's it's going to be a matter of looking at that candidate list and saying okay it's that one. And that's a little bit instinctual, there's a bit of an art to that, rather than necessarily being something that we can do scientifically in all cases.

**DOMAIN ABSTRACTIONS**

In addition to the activities and objects that occur inside of our domain, there are certain abstractions that we leverage as well when building using domain driven design. These abstractions can be useful for a number of different reasons.

One of the abstractions that we use is what we call a **service**. We talked about the fact that we have entities and value objects. And that those entities and value objects can contain business logic as well as state. But sometimes you get cases where there's a particular piece of business

logic that doesn't really fit into an entity or a value object. For whatever reason you can't come up with an entity that makes sense for that particular piece of business logic. In that case this can be encapsulated by a service. Services have some special criteria: **The first is they should be stateless**. If you find yourself including state in it, it's not a service anymore. It's either an entity or a value object. They should be stateless. The other thing is that services are often used to abstract away something like an anti-corruption layer. In our example here we have our Abstract Email Sender. That doesn't contain any state. The state would be contained in an Email object of some kind. The job of the Email Sender is just to take that Email and send it. What we are doing in this case is we're creating an Abstract Email Sender and then there will be a Concrete implementation; and in this case an SMTP Email Sender. That SMTP Email Sender is an external service. We're putting an anti-corruption layer on top of it so that we're not communicating directly with it in our domain. That means that our domain doesn't need to know about SMTP. All it needs to know about is emails which makes sense because later on we might start out using an SMTP Email Sender but then later on we might change that up to something different. We might use a web service or something like that instead. This allows us to swap in those different implementations. Again that Email Sender is totally stateless but it

does contain some valuable business logic. **It's important to note that creating systems that use too many services, it leads to an anemic domain**. When we start finding ourselves in a situation where we need to implement a service we should be very careful first at making sure that we're not missing an entity or a value object. What we don't want are services to be doing all the work. We want services to typically be fairly thin layers over a very specific piece of business logic rather than something that just does everything. Some other examples of things you might do inside of a service: So we have the Email Sender, that's one example. Something like maybe if you need to hash a password at some point then you might have a Password Hashing service. Your Password Hashing service is potentially stateless but it doesn't necessarily really belong in any of your entities or value objects. I suppose you could put it inside of your password itself. Although at the same time it would be nice to do an abstraction of some kind so that you can swap out different hashing algorithms if necessary. In that respect something that that would make a good use of a service as well.

Another type of abstraction that we use is a **factory**. When we go to create a new domain object often entities or aggregate roots, the logic to construct those domain objects may not be trivial. There may be a lot of work involved with creating that domain object. It may have to access external resources. For example when we create a new Reservation maybe we need to assign an unique identifier to it, maybe that requires us going to the database and looking at a table. So there may be database access we might have to look at, files or REST APIs. There's all sorts of complexity that may come along with building that new object. A factory allows us to abstract away that logic. It's again usually implemented as a domain interface with one or more concrete implementations. In this case, we have a Reservation Factory which knows about Reservations

and knows that it has a method that will allow you to create them. But it doesn't necessarily know the details of how it gets created. We leave that to the concrete implementation, in this case, the Cassandra Reservation Factory. We may go to Cassandra and look in a table in order to generate an ID or whatever needs to happen. That logic is going to exist in the concrete implementation. The factory is giving us an abstraction away from that concrete implementation which means that our domain doesn't need to know about Cassandra. All it needs to know about is Reservations. Again, that's a valuable abstraction. Factories incidentally if you're used to the term CRUD -- Create, Read, Update, Delete -- factories or the C in CRUD; they are the create. Repositories are similar to factories but instead of abstracting away creation they abstract away the retrieving of existing objects. Factories are used to get new objects. Repositories are used to get or modify existing objects. Again if we use the CRUD terminology -- Create, Read, Update and Delete -- then repositories are the RUD. They are the Read, the Update and the Delete. They often operate again as abstraction layers over top of databases but they can also work with files, REST API, etc. In our example here we have our Reservation Repository. And we have a specific implementation of it that is file based. So we're not using a database in this case, we're using a file system. Now that's an important distinction here which is the fact that oftentimes repositories are thought of as just a layer on top of the database. While they often are, they don't have to be a layer on top of the database. The other thing is that with repositories and factories, and even services, because we have this abstraction layer, we can substitute different implementations. A common thing that I will do is I'll build a repository. And I will have for example, a Cassandra version and an in-memory version. The in-memory version I'll use in tests but then the Cassandra version I will use in production. Or I've also used this in cases where I've had a MongoDB version, for example, which then later on we swapped in a SQL version. We actually wanted to change the database that we were using. So, you can swap in different implementations as required. Quick note on factories and repositories: factories and repositories are related. We talked about the fact that factories are the C and repositories are the RUD. Because they're so closely related often times people just combine them. And you end up with a repository that has all of the Create, Read, Update and Delete operations -- now we don't bother with factories in that case. Tools like Akka and Lagom are really powerful because they provide facilities that abstract away the need for repositories. When you're using Akka Persistent Actors or Lagom Persistent Entities, you don't need repositories or factories anymore. That's taken care of by the framework. The implementation of repositories in Akka or Lagom actually ends up being done by plugins. So rather than having to go ahead and implement that repository you just drop in a new plugin and it just works. That's a very powerful thing. Often times when you see systems built using Akka or Lagom you will often see that they don't bother with repositories. Having said that, repositories are still a very valuable technique that I would encourage you to get a good handle on even if you are planning to use tools like Akka or Lagom.

**HEXAGONAL ARCHITECTURE**

…