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.

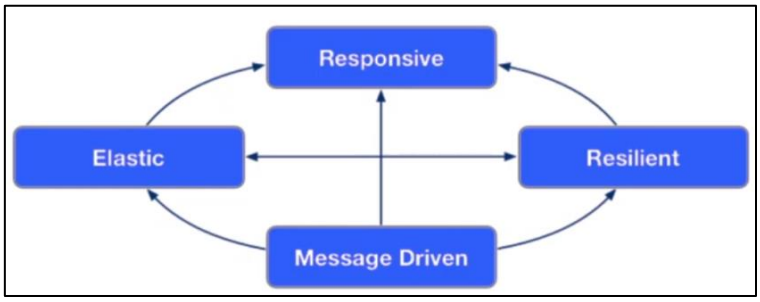
Why do we need reactive architectures? The software installations increase day by day. It is really common to go to a website/application and see "application in maintenance, please try again later" → developers may run updates or databases scripts on the software, we still need to do those operations 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**.

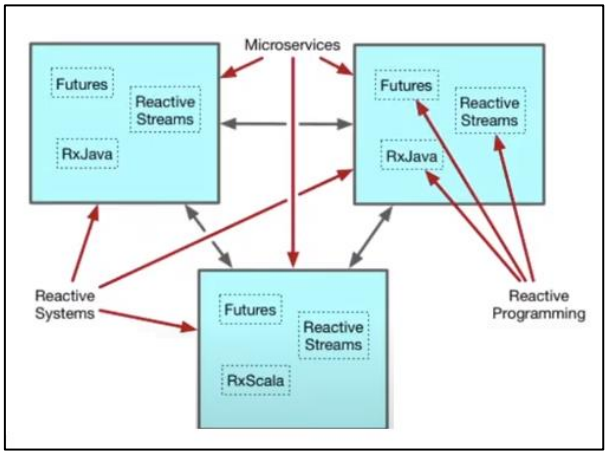
THE REACTIVE PRINCIPLES

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.



REACTIVE PROGRAMMING



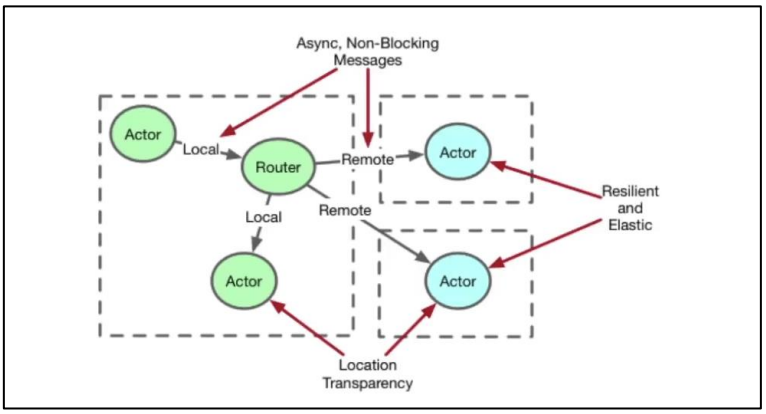
Reactive Systems apply the reactive principles at the architectural level, and they are represented by **microservices** (the 3 box) which are components designed at the architectural level to be reactive. Inside every microservice there are other elements: Futures, Reactive Streams, RxScala, RxJava → 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 a restaurant when a kitchen is preparing an order, there are certain things they care about. They need to know what's on the order. They need to know did the person order the reactive ribs did they order the steak. What did they order? Those types of details are important. They need to know anything about 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. It makes no difference to the kitchen what the price is. That matters to the customer 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 of detail where we don't need to provide the kitchen with all of that extra information they don't need. So, it's important 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 do this, and you'll get the right answer every time. There's nothing that works like that but **there are some guidelines**. We want to consider human culture and interaction. If you look at our restaurant there are **different areas** of the 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 the server takes or there's the order that is used for inventory management. Those two orders are different therefore they may exist in a different bounded context. Look for varying or unnecessary information. Employee ID is very important in an employee but it's meaningless in a customer. So that again might imply that those pieces of information belong in different bounded contexts. Another thing to realize is that **strongly separated bounded contexts will usually result in smooth workflows**. If you find yourself in a situation where you're trying to implement a particular workflow and it's awkward, that may

suggest that you've misunderstood your domain. Or you've broken up your bounded context incorrectly. If you find a bounded context has too many dependencies, again it may be over complicated. You may have drawn your lines incorrectly. That's an area where you want to explore in a little more detail and try and figure out if you'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

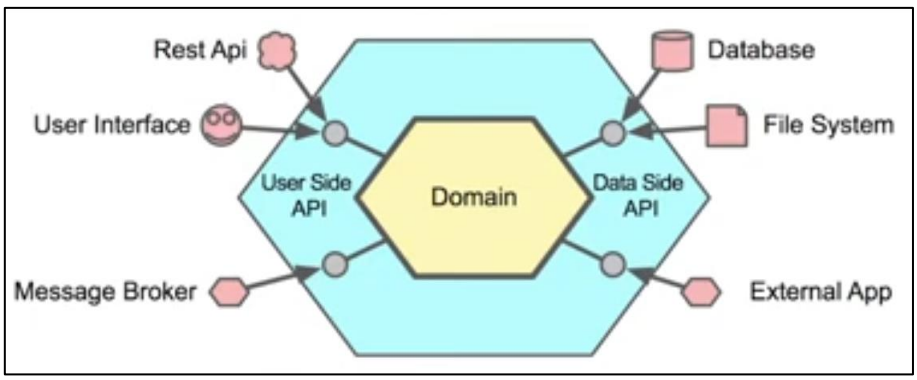
all the work. We want services to typically be fairly thin layers over a very specific piece of business logic rather than something that just does everything. Some other examples of things you might do inside of a service: So we have the Email Sender, that's one example. Something like maybe if you 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

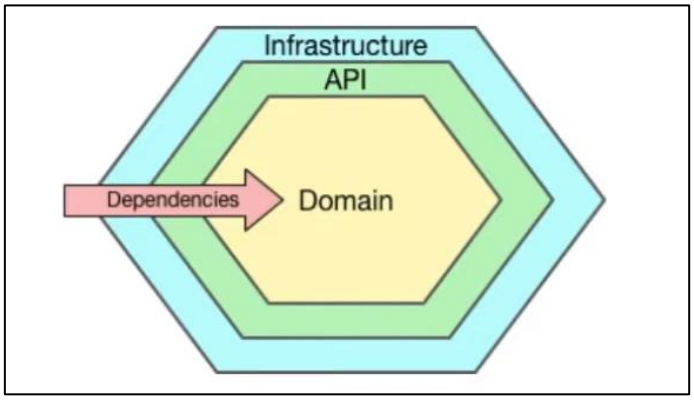
factories and repositories: factories and repositories are related. We talked about the fact that factories are the C and repositories are the RUD. Because they're so closely related often times people just combine them. And you end up with a repository that has all of the Create, Read, Update and Delete 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

A particular technique that is often combined with domain-driven design is called **hexagonal architecture**. Hexagonal architecture is not directly related to domain driven design. You can use domain driven design without using hexagonal architecture however **it is very compatible with domain driven design**. Hexagonal architecture is also known as ports and adapters. It's an alternative to the layered or n-tier architecture that you may or may not be familiar with. The **n-tiered architecture** is the idea that you **separate your application into different layers**. Usually with a database layer at the bottom and there's some kind of user interface layer at the top with kind of a domain layer sandwich somewhere in the middle. The number of layers varies depending on who you talk to, but you know it's somewhere between three and five or seven or something like that. **The idea is though that you have the user interface on one end, the database on the other end, and a bunch of other layers in the middle with the domain being one of the important ones**. **Hexagonal architecture takes a sort of a different approach** here. The idea here is that instead of the domain being in the middle of kind of that sandwich of the n-tiered architecture, **the domain is at the core of something** that it more closely resembles in this case a hexagon.



The idea here is that the domain is core. It is absolutely the most important thing in your application, therefore it should be at the center. What you have then is that **you have ports that are exposed by the domain**. These ports act as essentially an API. They are per a preferred way of communicating with the domain. Then you have a series of we'll call **infrastructure adapters**. And those adapters communicate with that those ports, they communicate through that API. There are different sides to the hexagon. On one side we have something like the sort of the user side API and on the other side we have the data side API. If we actually were to flip this around on its on its side, then what you'd see is the user side API at the top, and the data side API at the bottom. At that point it starts to resemble the n-tiered architecture. In a lot of ways **it's just a different way of presenting the idea of the n-tiered architecture**. However, there are some additional rules that go along with hexagonal architecture that aren't necessarily talked about as much in the n-tiered architecture. Those **additional rules** provide some benefit.



We mentioned before that one of the ways you sometimes see this drawn is as an onion. Here we've kind of combined the onion and the hexagon into a drawing. Again, you see the domain at the center. Outside of that domain we have another layer which is the API. The domain exposes an API which is another layer outside. Then outside of that we have an infrastructure layer. The idea here is again the domain is the center of the onion, the API provides the ports and it's another layer, and then the infrastructure provides the adaptors which communicate with the ports. **One of the keys here is the dependencies**: **the outer layers are allowed to depend on the inner layers but the reverse is not true**. The infrastructure can depend on the API and the API can depend on the domain, but the domain has no knowledge that the API or the infrastructure exists. This has very interesting consequences. What it does is it means that **hexagonal architecture ensures a proper separation of infrastructure from domain**. Therefore, the domain has no knowledge of the infrastructure. It doesn't know about the database. It doesn't know about the user interface. It doesn't know about any of those things because it's not allowed. **Those rules prevent the concerns about databases, user interfaces, things like that, from bleeding into the domain**. This can be enforced with **packages**. What I will often do in a project is I'll create a domain package, an infrastructure package and an API package. Then I'll do searches and things like that to ensure that my domain package never refers to anything in my infrastructure package for example. But you can also do this using project structure. If you set these up as projects with dependencies then you can ensure that you can't have the domain depend on the API or the infrastructure because you

simply don't create that dependency. And if you did, it would create a circular dependency and then it wouldn't compile anyway. What this does is **it allows your domain to be portable**. It means that it becomes much more easy to swap out pieces of your infrastructure without having to affect your domain. **You can go ahead and swap to a different database implementation**. It doesn't have to be a different database, it might just be a different table structure within the same database. But you can swap to different structures without having to affect the domain. The domain doesn't know that you're using SQL. It doesn't know that you're using Cassandra. Changing those things becomes irrelevant from a domain perspective. The other thing that it does is **it allows you to make changes in the domain without affecting the API or the infrastructure potentially**. I've had situations where I've used this technique and I've realized that the way that we have done things in the domain was fundamentally flawed. We went ahead and rewrote the entire domain of a particular microservice without having to change anything in the API or infrastructure. Because you have this clean separation it means that you can make these kinds of substitutions and external clients of your application don't have to know about them. They don't know that anything has changed because you have this nice level of isolation.

Systems designed around hexagonal architecture can therefore be very flexible. You can change them in a lot of different ways and have a minimal impact on clients. That's a that's a very powerful technique. In terms of Akka and Lagom, some of our reactive tools, they actually leverage hexagonal architecture very well. What they do is they take the concepts of hexagonal architecture, and they abstract many of your infrastructure needs. So, persistence, distribution and communication are all built into the toolkits. Those are all infrastructure concerns. They're built into the toolkits in such a way that in many cases you can swap them out using different plugins. What this means is that you can now focus on the domain rather than having to worry too much about the infrastructure. You can worry less about the persistence, the distribution, the communication, because those are all taken care of by the toolkits. And now you can just focus on building the domain which is really at the end of the day, what we're there to do anyway.

3. REACTIVE MICROSERVICES

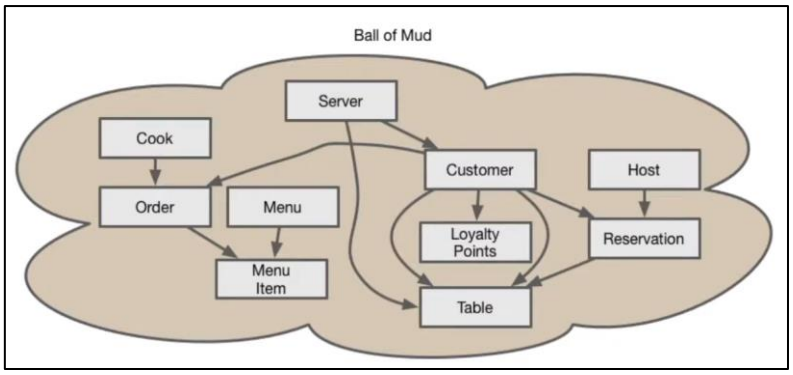
In the early days of software development, a single developer, or a small group, would build an application to support a small number of users. In this environment building a monolithic style application made sense. It was the simplest thing we could do. However, over the years, software complexity has grown. Today, multiple teams are working with many different technologies, supporting millions of users. In this environment, the monolithic application becomes a burden, rather than a benefit. In the modern world of software development, we need to find techniques that allow us to isolate the inherent complexity, not just of our software, but of our development processes as well. This is where microservices enter the picture. In this course we will have a look at the contrast between monolithic applications and microservices. We will journey down the path, showing that as we move from a monolith to a set of microservices, we are introducing more isolation into the system. And behind that isolation is a core set of principles to guide us. Understanding, and following these principles allows us to build systems that are truly Reactive.

THE SOFTWARE SPECTRUM

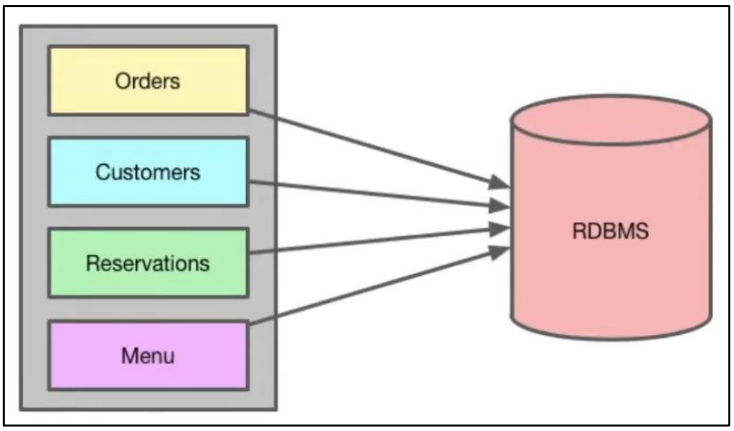
We've talked a lot about building reactive microservices up to this point, but we haven't really defined what a microservice is. And that's a bit of a problem because when people start building microservices for the first time, that's usually one of the big questions they ask. How big is a microservice? How do I decide what's appropriate there? We often talk about it in black and white terms: you're either building monoliths or you're building microservices. But that's not really fair because the reality is **monoliths** and **microservices** don't exist in kind of a black and white situation. They exist more on a spectrum with monoliths on one end, microservices on the other. Then **most applications that get built, live somewhere in-between**. The thing that we have to recognize is that **monoliths and microservices both have advantages and disadvantages**. No matter which one we pick we're going to be giving up one thing in order to gain something else. It's then a big part of our job to look at the differences between them and look at the benefits or disadvantages to each and try to balance them. In doing so we place ourselves somewhere on that spectrum. In order to do that though we need to really understand the difference and that's what this next chapter is going to be about. It's really gonna be about trying to understand the difference between monoliths and microservices. What makes a good microservice system and how we can try to balance those things.

MONOLITHS

Our software spectrum has monoliths at one end and microservices at the other. Let's first take a look at the worst-case scenario for a monolith.



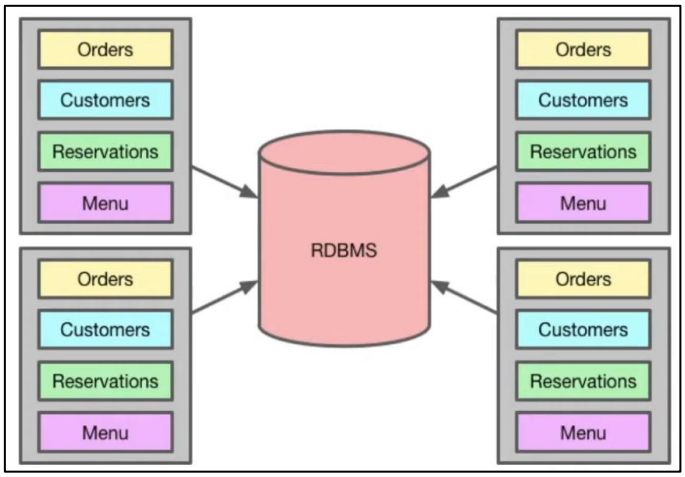
Now of course this does represent the worst-case scenario. Many monoliths don't look like this but there are some that do. So, if you've built or seen one that looks like this, this may look familiar. But on the other hand, if you exist somewhere else on the spectrum with your monolith, then your monolith may look very different from this and that's okay. But this is the worst-case scenario. Basically, we call it the **ball of mud**. It's also often referred to as spaghetti code. Essentially what we end up with in this case is there's **no clear isolation in the application**. If you look at this application you can't divide it into subsections or anything like that. The entire application, **everything depends on everything else**. We have a lot of complex dependencies. **Because of these complex dependencies**, because of the lack of isolation, **this is a very difficult application to understand and an even harder application to modify**. These are the types of applications that over time start to stagnate and it becomes more and more expensive to make changes, and that's obviously a problem. When we see this kind of application, when we have this monolithic ball of mud, **we're tempted to clean it up**. So how do we clean up the monolithic ball of mud so that it's not quite so ugly?



To clean up the ball of mud we introduce isolation into the application. We divide the application along clear domain boundaries. You can see that here we've taken our ball of mud where everything depended on everything else and we've said okay there is a particular section of the application that deals with orders. There is a section of the application that deals with

customers. There's one for reservations. There's one for the menu. So these different sections the application may exist as packages or libraries or some other manner of separation, but the point is that we have created very clear domain boundaries within our application. Libraries are a pretty common way to do this.

What are the characteristics of a monolith? First off, **a monolith is deployed as a single unit**. That's kind of one of the key elements that makes it a monolith. You deploy everything as a single unit. You may deploy that monolith multiple times but within each instance of the monolith everything is a single unit. **They have a single shared database typically**. So, **they'll have usually a relational database, but it doesn't technically have to be one**, but it is usually a relational database. And **they communicate usually with synchronous method calls**. So, this is a request--response scenario. You send a message, you expect a response immediately. **There's often deep coupling between libraries and components**. A lot of the time that's through the database. So, you'll have different parts of your application all relying on the same data from the database. They go to the database, they may do two joins on the database, there may be views established, whatever, but they will all go to the database to get the data that they need. That creates deep coupling. They have big bang style releases. So, **when you go to release this application it's kind of an all-or-nothing**. And what ends up happening is you'll get those situations where you have: okay everybody it's release day, so nobody check-in any code because we're doing a release, and we want to make sure nothing moves. So, the world kind of stops while you're doing that release, everybody focuses on that release. There's a lot of communication going on between teams about what's in their release and what isn't. There are often long cycle times. These can be weeks to months sometimes longer. And then teams have to carefully synchronize features and releases. So again, in that Big Bang style release you'll get a lot of the teams talking to each other saying "hey I got my stuff in, did you get your stuff in?" They are all trying to figure out whether all of the necessary pieces are going to be in that release before they release it.



So how do we scale these monoliths? **Multiple copies of the monolith are deployed**. **Each copy is fully independent. They usually don't communicate with each other**. You don't know that there are multiple copies deployed or at least each monolith doesn't know that there are other copies of itself deployed. They're fully independent. All of the dependency, **all of the**

communication between monolith instances happens in the database. So, the database provides consistency between the instances.

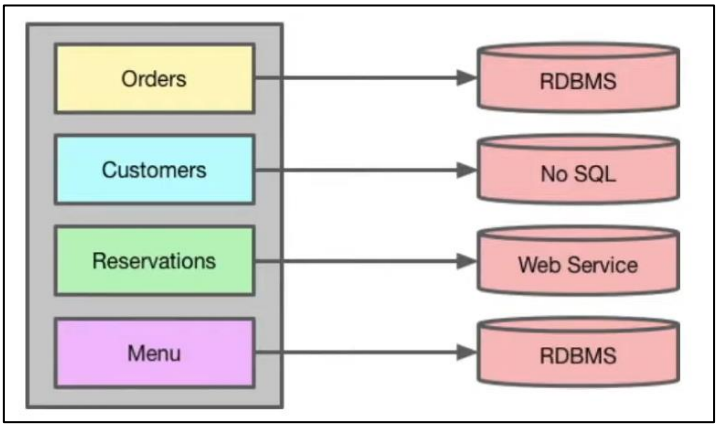
What are the advantages of the monolith? It gives **easy cross module refactoring**. If you want to change something that is dependent on by another module you just do a refactor. You can often do that using your IDE tools and that's completely fine. So that's very easy to do. It's also **easy to maintain consistency**. Again, the database acts as your consistency boundary so as long as your database supports consistency then your application will support it. This is very easy to maintain consistent a data set. You can always guarantee, for example, that if you write some data and then don't make any other modifications, you'll be able to read that data back. So, it's very easy to maintain consistency in a monolith. You have **a single deployment process**. When you want to deploy, yes you have this Big Bang style release, but you're only doing it once with one thing. And everybody knows how to do it or at least a few people are gonna know how to do it. You've only got one thing to monitor. You just need to know is your monolith running? I suppose you also need to know is your database running? But from an application perspective, all you're monitoring is the monolith. So, there's one thing to monitor, one approach to monitoring it that's very very simple. And it has a relatively **simple scalability model**. So again, the scalability model is just deploy more copies of it.

What are the disadvantages of the monolith? Let's talk about what the trade-offs are. **We're limited by the maximum size of a single physical machine**. Because of the nature of a monolith as that monolith grows, it's going to occupy more memory, it's gonna need more CPU, it's gonna need more system resources. Eventually it can grow to the point where it doesn't fit on the physical machine that you've got and then you need to get a bigger physical machine. As you continue to grow you can eventually hit the point where you actually can't get a bigger physical machine. Or the other factor is big physical machines are often more expensive than small physical machines. So, you'll end up with a cost issue with it as well as you go to these bigger and bigger machines they become more and more expensive. **It also only scales as far as the database will allow**. Yes, you can deploy multiple copies of the monolith, you can put them on multiple machines in order to continue to scale up, but often these are relying on a relational database which probably only runs a single instance in order to maintain consistency. You might have multiple instances but there's often a master/slave sort of scenario and because of that, that master only scales to a certain point. Again, it's limited by the size of a single physical machine and so you can hit that limit and then you have a problem. If you're wondering whether that can happen, whether you're ever going to hit the situation where your database is gonna run out of system resources, I can tell you it does. I've worked with clients that have hit that point where they're going "we cannot add any more load to this database because it will fall over and we can't get a bigger machine." So that does happen. **Components must be scaled as a group**. Even though certain components of your application may require less resources, it doesn't matter. You have to scale everything at the same time and as a result you may have unnecessary use of resources. **The deep coupling leads to inflexibility**. Because you have everything depending on everything else, often through the database, it becomes very hard to modify things. For example, you want to change the structure of a table because you've realized that it's kind of inefficient and there's probably a better way

to restructure it for a particular use case. The problem is there's a bunch of other things that are also accessing the data from that table. So, it becomes hard to make that modification because even though it may be more efficient for one thing it may be less efficient for something else. At the same, time you now have to track down all of those usages of that database table. So, it becomes inflexible in that sense. **Development is typically slow**. Because change is difficult due to the inflexibility and deep coupling, it means that change becomes difficult and build times start to become long. **Serious failures in one component often bring down the whole monolith**. So, what will end up happening is you'll have a particular failure, the monolith will go down. But then when the when one instance of the monolith goes down, it redistributes the load to the other instances only they can't handle the load either. So now they start to collapse this as well is called a cascading failure. There are some advantages to a monolith which we outlined but there's disadvantages as well. When we build our applications, we have to think about that and we have to decide what are we going to balance, what are we going to trade in order to get the right advantages or disadvantages?

SERVICE ORIENTED ARCHITECTURE

When we talked about monoliths, we saw that one of the ways that you can improve the monoliths' situation rather than having a large ball of mud kind of scenario, is that you can introduce an additional isolation. You do that by separating your monolith along clear domain boundaries usually in the form of different libraries for different areas of the domain. There's another way that we can introduce additional isolation and that's using something called **Service Oriented Architecture**.

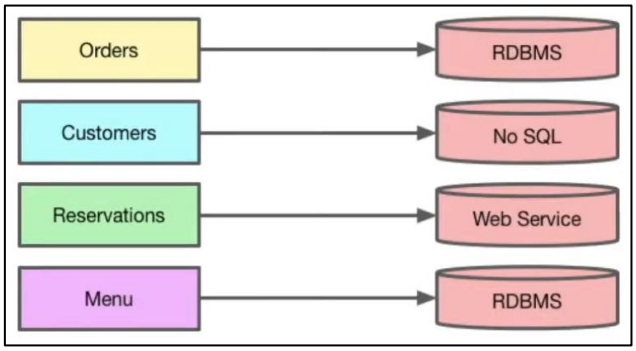


In a Service Oriented Architecture **each of those domain boundaries**, each of those libraries, **represents what we call a service**. So, you can see here we have an Order service, a Customer service, a Reservation service and a Menu service. The idea with Service Oriented Architecture is that **those services don't share a database**. **Each service will have its own database**. **Anybody who wants information about that service has to go through the services API. They're not allowed to go directly to the database**. So, for example in our Reservation service we might need Customer information. Now if that's the case we can't go directly to the Customers database. What we have to do is we have to go through the Customer API. We have to communicate directly through the library and then that will provide us information rather

than going to the database. Now what **this does is it creates additional isolation because now each independent service can have its own database**. It doesn't even have to be the same type of database. For example, in this case, we see that Orders is a relational database but Customers is using some sort of No SQL database-- maybe Cassandra or MongoDB or something like that. So now we can actually create additional isolation in that sense. And **the other thing is there's now no coupling between the different parts of the application and the database**. So, the Reservation system isn't coupled to the Customers database. Which means that if we want to modify the Customer database in some way -- we want to change the table structure or something like that -- we're free to do so. The only thing we have to worry about is the Customers service. We don't have to worry about the Reservation service and how it accesses that data. This reduces the coupling. Some people will say that Service Oriented Architecture is just microservices; it's just another word, so why do we need that other word. The reality is that with Service Oriented Architecture, it doesn't talk about deployment. So, while it talks about the fact that services don't share a database, they have to communicate through their API's, it doesn't say how they can be deployed. As a result, some people when they build using Service Oriented Architecture, they build it in a monolithic style like we've drawn here, where they still deploy all of those services as one application. But then those services communicate through some clearly defined API. On the other hand, some people choose to go the other route where each service is deployed as an independent application in which case it more closely resembles microservices. So, it's true that if you are in the latter case, if you are deploying each of your services independently, then there is very little difference between Service Oriented Architecture and microservices. But if you're on the other side and you're deploying things in a more monolithic approach, then there actually is a fairly significant difference. We're gonna say for now that Service Oriented Architecture isn't necessarily the same as microservices although in some cases it may be. From there we can start talking about well what makes it a micro service.

MICROSERVICES

We started with an application that had very little isolation. We had a lot of complex dependencies, everything depended on everything else. We introduced some isolation in the form of libraries that were based around domain concepts. And then we later isolated it even further by giving each of those libraries its own database. That brought us in the transition from monoliths to Service Oriented Architecture. Now we're going to take the next step and move into Microservices.

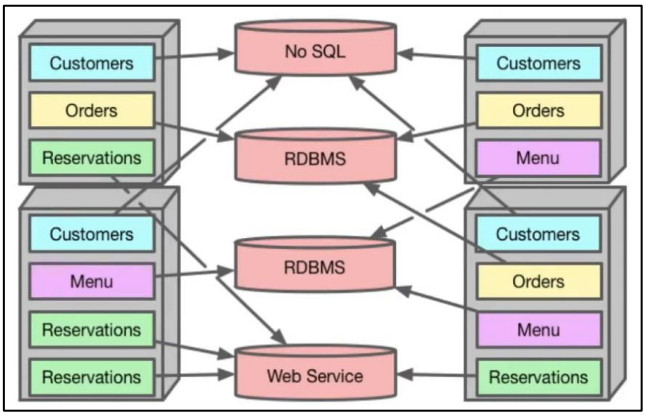


So **Microservices are a subset of Service Oriented Architecture**. The difference between them is **Service Oriented Architecture doesn't dictate any requirements around deployment**. You can deploy Service Oriented Architectures as a monolith, or you can deploy each of your services independently. If you deploy them independently that's when you start to build microservices. **Micro services require that each of those services are independently deployed**. This means that they can be put on to many different machines. You can have any number of copies of individual services depending on your requirements. We still keep all of the rules around Service Oriented Architecture: maintaining our own datastore, ensuring that our services communicate only through a clearly defined API, all of those things still apply. We've just added that extra requirement that we have to deploy the individual services independently. By doing this it means that **microservices are independent and self-governing**. If you look at our Reservation service and our Customer service, certainly our Reservation service has a dependency on Customers. But our Customer service has no dependency on Reservations. As a result, we can safely deploy our Customer service without giving it any knowledge of any of the other pieces of our application. It probably doesn't need to know about Orders. It probably doesn't need to know about Reservations. And it certainly doesn't need to know about the Menu. It can be completely independent from all of those things. It governs itself in that case.

What are the characteristics of a microservice? **Each service is deployed independently**. That is one of the key differences compared to a Service Oriented Architecture. You have **multiple independent databases** which it inherits from SOA. **Communication is synchronous or asynchronous**. So, it's not uncommon in a microservice system to see a synchronous communication where you have a request-response; you expect the response immediately. But it's also not uncommon to see a more asynchronous approach where you maybe send a request and then you don't wait for a response. Either one never comes or if it does come you're not waiting around for it. It'll come in an asynchronous fashion. That's what you're doing when you're building with reactive microservices. All of these things, in addition to other things, provide **loose coupling between the components**. Because we don't have database dependencies, we don't have synchronous communication dependencies. We don't have even necessarily shared code in some cases. That means that we are creating something where everything is much more loosely coupled. That can be a benefit and we'll talk about that shortly. We get rapid deployments. You know it's not uncommon to see micro-services that are deployed more frequently than once a week. But sometimes they might be deployed once a day or even multiple times a day. Sometimes that **deployment is even continuous** where you check your code in and after some automated suite of tests it automatically goes to production without any further intervention from yourself. **Team's release features when they're ready**. We get rid of that Big Bang style release where everybody has to communicate and make sure all their work is in sync. That goes away. Now teams work on features and release them as they are ready to do so. **Teams are often organized around a DevOps approach**. So, the microservices team will handle the application from developing the code all the way to maintaining it in production. We get rid of this idea of throwing it over the wall to the Ops team; which is not uncommon in monolithic approaches where you'll have the dev team does the

development and then the ops team maintains it in production. And those teams may or may not communicate. That goes away in the microservice approach because one team is responsible for everything.

How do you scale a microservice application? **Each micro-service is scaled independently**. **You can have as many copies of a microservice as are required**. So, if for example we decide that we need 20 copies of our Order service and only three copies of our Reservation service, we can do that. That means that the nodes in our cluster, the individual pieces of hardware, will host 0 or more copies of our application.



In this diagram the cubes represent hardware. You can see that in some cases we actually have two copies of Reservations deployed on one machine. We have 4 copies of the Customer service, but we only have three copies of the Menu service. We are free now to make those types of decisions and to deploy our application as required. Technically these applications don't even have to be deployed in the same data center. We could in some cases deploy them in different data centers if necessary.

What are the advantages to the microservice system? **Individual services can be deployed and scaled as needed**. Again, you can have 20 copies of the Order service and only three of the Reservation service if that's what's required. You also get **increased availability because serious failures are typically isolated to a single service**. For example, if our Reservation service failed for some reason, we might still get a cascading failure where one instance of the Reservation service fails. And when it fails it redistributes load to the other instances and then those instances in turn fail because they now can't handle the additional load either. That is still possible with the microservice setup but when that happens, we've only lost the Reservation service. We didn't lose the Order service, we didn't lose the customer service, we didn't lose the Menu service. As a result, it's entirely possible that a large part of our business can continue to operate. People can still place orders. They can still come and eat in our restaurant. We only have lost the ability to create Reservations. We also have an **insulation and decoupling which provides more flexibility to evolve within a module**. Within any one of these services you're free to evolve the underlying code as long as you maintain that external API. As long as you keep the API the same whatever happens under the hood doesn't matter. If you want to change databases, that's fine. If you want to restructure the database, that's cool too. If you want to completely rewrite the underlying code of the application, no problem. I can honestly say that

I have done all of these things in a microservice architecture. As long as you maintain that API nobody knows that you've made those changes. Because you don't have to worry about who else might be accessing your database or accessing your code it becomes very easy to do that. **It also supports multiple platforms and languages**. It's very easy to build your applications so that you have some of your microservices written say in Java and then later on if you want to introduce Scala you can do that. If you want to have some of them written in C# or Python that's fine too. There's nothing that restricts us to a particular language. It's also nothing that restricts us to a particular database. If you want to have part of your application -- maybe is well suited to a relational database -- so you deploy that using a relational database. Maybe another part would be better suited to something like Cassandra and that's fine. If you want to do that there's no problem there.

Now **microservices do often come with an organizational change**. If you're within a large existing organization this may not be quite the way people are used to building things. One of the things that you'll find is that teams tend to operate more independently. You don't have these big bang style releases so teams aren't constantly communicating with each other to synchronize on various tasks. They have a particular task. They do that task. They deploy that change. And then they move on. This results in shorter release cycles. So it's not uncommon to see microservices team releases perform releases multiple times a day. You might have five releases in a day as opposed to having to schedule them at a certain time every month. This results in less cross team coordination. You're not again constantly going to the other teams and saying "hey I've got my changes for this feature, they're going in, are you ready for that?" You don't have to have that kind of communication. You just release when you're ready. These changes can facilitate an increase in productivity if done right. It can result in teams that are able to release new features, sometimes even within hours of getting information about that feature. You make the changes, you release. If you have a bug that needs to be fixed, you fix the bug, you release. You don't have to coordinate with a bunch of other people in order to get that bug fix out the door. This can result in an increase in productivity but it's not free.

We talked about the fact that monoliths have advantages and disadvantages, so do microservices. Because you now have many different microservices possibly written in different languages being deployed, you may have complex deployment systems and complex monitoring approaches. It's with a big bang style monolith release you just had one process. Granted it was a large process and it was complicated but there was only one of them. Now with microservices each individual deployment is usually smaller. It's not that big bang but there's many of them. So overall **the complexity could actually be larger**. It's just hidden across multiple services. You also have **multiple deployment and monitoring approaches because each of these things may have different monitoring requirements**. They use different databases. They may be written in different languages. They may be deployed on different hardware. The way you monitor them might not be the same and so that becomes more complex. As well, **cross service refactoring becomes harder**. When you were in a monolith and you wanted to do a cross service refactoring, you just refactored and deploy, that was it. Sometimes the automated tools could do that for you. That's harder to do with a microservices because these things won't be deployed at the same time. As a result when you

make a change to the API, for example, you have to continue to support the old API; **you have to version your API so that anybody who's using that old version can continue to work and then you can slowly migrate them over to the new version**. Once they're migrated over to the new version, well then you can go back and delete the old version but you know that takes time, that's harder than it was in a monolith. The other thing is that there's an organizational change that comes in here as well. We talked about this a little bit, but the fact of the matter is large organizations that have been building monoliths for a long period of time have built up structures around that. They may have Quality Assurance groups that need to test the monolith in a certain way. They may expect deployments to happen on a certain schedule. They may have various things that are arranged around that schedule. When you start trying to introduce microservices and you want to deploy maybe in a continuous fashion and you want to do a lot of these other interesting things, the organization may not be ready for that. So, we have to be very careful when we introduce microservices into an organization that we don't try to do it all at once. Start small. Introduce things a little bit at a time. And allow the organization to slowly adapt to the new process before trying to introduce the next thing.

THE RESPONSIBILITY OF MICROSERVICES

Hopefully at this point we have a better idea of what makes a microservice, but we still have a little bit of a question of well how do we decide how big a microservice is? We haven't really answered that question. Now we're going to start to answer that question a little bit more. A big part of understanding where to draw the lines between your microservices is all about understanding responsibility. To do that we want to go back to Robert C Martin, also known as Uncle Bob. He came up with something that he calls the "**single responsibility principle**." In actual fact, when he came up with this, he wasn't talking about microservices he was talking about classes in an object-oriented system. What he said was that **a class should have only one reason to change**: that's the single responsibility principle. But this idea works well for microservices. **The idea then is that a microservice should have a single responsibility**. It should have only one reason to change. An example of that would be if you have a microservice for managing Accounts. Or if you have a microservice in our Reactive Barbecue for dealing with Orders. Or a microservice for the Menu. These things have only one responsibility. The Menu service deals only with the Menu. It doesn't do Menus and Orders. If you do Menus and Orders then it's not a single responsibility. What this does is it means that a change to the internals of one microservice shouldn't necessitate a change to another microservice. If I go ahead and I want to completely rewrite the Reservation service that shouldn't require me to also make changes to the Customer service or to the Order service or something else. If I want to completely gut the Customer service, which the Reservation service does depend on the Customer service, but again as long as I maintain the external API for that service, then nobody needs to know that I made any changes. That's a big part of how the single responsibility principle applies to microservices. How do we decide where to draw those lines? How do we decide when to build our microservices and wear the proper responsibilities are? This is where we go back to domain driven design. **Bounded Contexts are an excellent place to start building microservices**. It doesn't mean we stop there. We'll get in to cases later in the course where we can actually figure out ways to further subdivide even a Bounded Context. But we

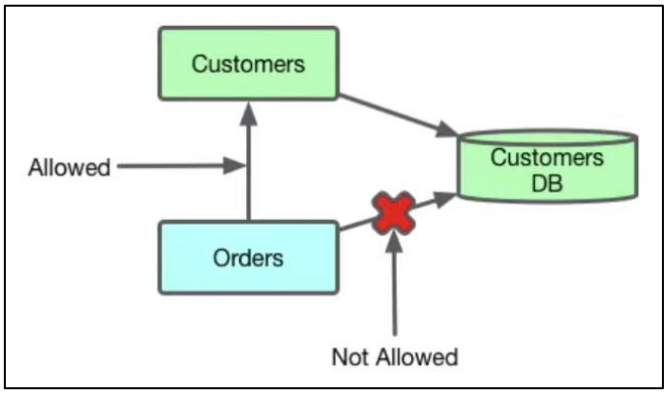
start by saying that **a Bounded Context could be a microservice**. Bounded Contexts define a context in which a specific model applies and so this gives us an opportunity to say okay this Bounded Context has a specific responsibility and exposes a specific API so let's leverage that as a microservice. Again, further subdivision of the Bounded Context is possible but we won't worry about that at this stage in the course.

PRINCIPLES OF ISOLATION

If you look at our journey so far through understanding microservices and reactive microservices you'll see that we've gone through various stages of introducing more and more isolation. We've been trying to figure out responsibilities and this brings us to really what I think is in many ways **the core of what reactive microservices is about and that's about isolation**. It really is about finding ways to create isolation between different microservices. So, **when you want to answer the question of how big should a microservice be, you're really asking the wrong question**. **The right question is more about how can I isolate my microservices?** If you can figure out how to isolate them appropriately, then you'll probably find a way to make them a good size. Let's talk about some of the principles of isolation, some of the ways that we can isolate our applications.

The benefit of doing this kind of isolation is **it provides reduced coupling and increased scalability**. As we introduce more and more isolation we saw -- when going from monoliths to Service Oriented Architecture and then from Service Oriented Architectures to full blown microservices -- we saw benefits in terms of reducing coupling and increasing the scalability. How can we take that even further? **Reactive microservices look to be isolated in as many ways** as possible. Specifically, **we look at trying to isolate them in terms of State, Space, Time and Failure**. I'm going to talk about each of these individually.

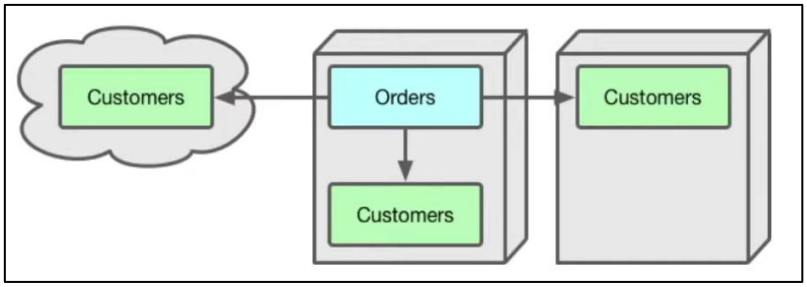
Let's start by talking about isolation of **State**. We've actually talked about this a lot already although we haven't really called it isolation of state. **The idea is all access to a microservice's state must go through its API**. You're not allowed to have backdoor access to the database.



So, our Order service may need Customer data for whatever reason. It can go to the Customer service and say hey I need information about this Customer. That's acceptable but it can't go directly to the Customer's database and just extract the information that way. That would

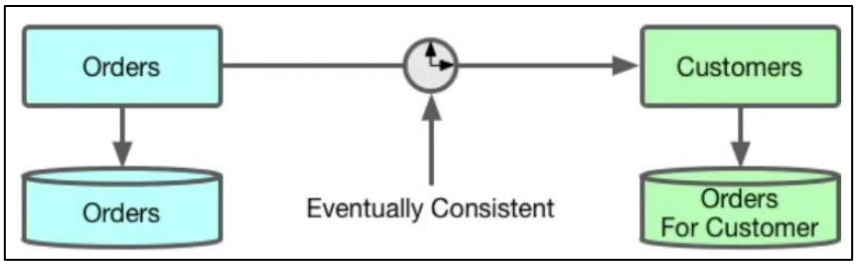
violate the isolation of state. What this does, this **isolation of state, allows the microservice to evolve internally without affecting the outside because we only go through the public facing API**. Again, we've mentioned this many times, as long we maintain that public-facing API, nobody needs to know what happens under the covers. Nobody needs to know that we have changed databases, restructured the database, nobody needs to know that we rewrote the domain code. All of that can evolve independently. That's the idea behind isolation of State.

What about isolation of **Space**? **Microservices should not care where other microservices are deployed**. This means that it's possible to move a microservice to another machine, possibly in a different data center, without any issues.



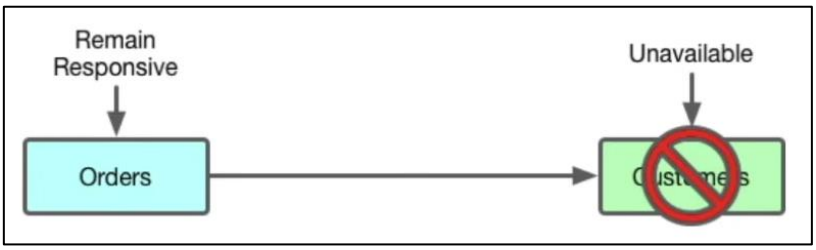
We see here in our diagram, we have the Order service, it has a dependency on Customers. But whether customers is deployed on the same piece of hardware or maybe a different piece of hardware, in the same data center or maybe a totally different data center, shouldn't matter to the Order service. Ideally, **it'll use a single method of communication that will work no matter where the Customer service is actually been deployed**. What this does is **it allows the service to be scaled up or down to meet demand. If we need to create additional copies of the service we can do that**. We can spin up 50 copies of the Customer service if necessary, across many different pieces of hardware without any issues. If it's possible to have it deployed in a separate data center, that's even better. Now it gives us amazing availability setups because now we can have multiple copies deployed across different data centers. Even if we lose an entire data center the application continues to operate. In some cases, due to latency issues, deploying to a different data center may not be possible, but if it is possible, it’s certainly going to provide some benefits. So, we should consider it at the very least. That's the idea behind isolation of Space.

What about isolation in **Time**? **Microservices shouldn't wait for each other. Requests ideally will be asynchronous and non-blocking**. This goes all the way back to our talk about the Reactive Principles which talked about asynchronous, message driven applications. This is really just an extension of that same idea. What this does is **it allows for more efficient use of resources. Rather than having a situation where we send a request, and then we block a thread while we wait for a response, what we do is we send a request and then we walk away**. That allows for more efficient use of resources because we don't have to occupy that thread and threads aren't free.



So, if we can free up the resources, and it's not just threads, we can free up things like memory, we can free up CPU, threads, all of these things. If we can free these up rather than holding a lock on them in some way, then that will allow for more efficient use of resources. I like to think of this in terms of people. In fact, I like to think about all of this stuff in terms of people. Can you imagine what your work day would be like if every time you went to ask somebody for help with something, you ask them to do something, and then you stood over their shoulder and watched until they were finished? Well, that's a very inefficient use of your time. It's probably kind of uncomfortable for them. But that's how we often build software. We have software that sends a request and then it sits there and does nothing while it waits for a response. **It would be far better if we could send the request, free up the resources and then move on to something else**. In addition to that **between microservices we expect eventual consistency**. **Within a microservice we can perhaps have strong consistency but between microservices we expect eventual consistency**. This provides increased scalability. Total consistency requires some form of central coordination and that limits scalability. We're going to get more into this later on so we're going to kind of leave it at that. But just keep in mind that **by allowing eventual consistency between microservices it gives us an increase in scalability** that we'll talk about later. So that's isolation in Time.

What about isolation of **Failure**? Again, **reactive microservices will also try to isolate failures**.



If we have a dependency between our Order service and our Customer service and our Customer service goes away; it fails for some reason. We don't want to also fail our Order service. Our Order service should remain responsive. The failure in one service should not cause another to fail. What this does is **it allows the system to remain operational in spite of failures**. The more that we can do this, the more robust our system is going to be. Ideally if we can get to the point where any one of our services can operate completely independently of all the others, then that leaves us in a really good state.

ISOLATION TECHNIQUES

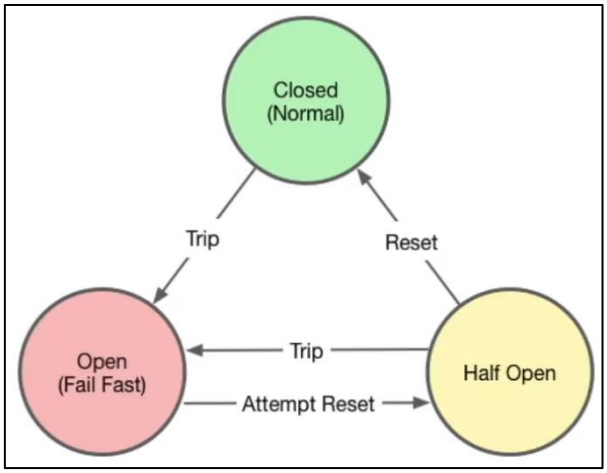
1) BULKHEADING

We've indicated that it's important to provide isolation in a number of different forms but failure being one of them. **A particular technique for isolating failure is something that we call bulkheading**. Bulkheading is a term that comes from shipbuilding. Bulkheads and ships are used to create separate watertight compartments in the hull of the ship. This means that **a failure in the hull will potentially flood one compartment, but the others are going to remain safe**. And as a result, the ship won't sink so this allows the ship to be resilient in the face of failure. **Bulk heading in terms of software is** a similar kind of concept. What we try to do is we **try to create failure zones within our application so that when a failure occurs its isolated within that zone and it's not going to propagate to other services**. If we have a microservice that can act as a failure zone, for example. Then that means that that microservice will fail but other services can continue to operate. The idea here is that **the overall system can remain operational even in the face of fairly significant failures; it just may have to operate in a degraded state**. Netflix is a really good example of this. The recently watched or my list features, when they go down that's due to a microservice being down. **What Netflix does in that case is it simply hides those features**. The rest of Netflix can continue to operate. You can continue to watch your movies or TV shows or whatever. You just can't access the more personalized aspects like to recently watched or my list. So that that's an example of bulkheading in a real world application. This is a pretty common technique to use when you're building with microservices.

Let's take a look at an example in our services. In the reactive barbecue we're gonna have maybe an Order service and a Loyalty service and a Payment service. There are some dependencies here. Payments is going to depend on Orders. We need an Order in order to make a Payment. That probably makes sense. We are also going to have Orders depend on Loyalty. When you go to have your Order service produce a receipt you want the Loyalty information on there so maybe it pulls it out of the Loyalty service. But if Loyalty is down, we don't want to fail. What we would like to do is allow ways for the Order service to operate without Loyalty. Maybe when it generates that receipt it just ignores that information. It puts like information not available or something on the receipt or it just leaves it off the receipt. If the customer doesn't get their loyalty information, most customers probably won't even notice that. So, there's ways that we can do that now. On the other hand, if Loyalty is down, we absolutely don't want Payments to fail because Payments has no dependency on Loyalty whatsoever. So, if we did allow Orders to fail because of the lack of Loyalty information that would then mean that Payments would also fail because it can't access Orders. Now all of a sudden large part of our system have broken down. So that's the kind of situation that we want to avoid. We want to allow our systems to continue to operate possibly in a degraded state when their dependencies are unavailable. The reality is if we were gonna implement this system in a truly Reactive way there likely wouldn't even be a dependency from Orders to Loyalty. There may not be a dependency from Payments to Orders. There are ways that we can implement that to eliminate those dependencies. But assuming that we had this sort of setup, we at least want to make sure that we didn't bring down the entire system.

2) CIRCUIT BREAKERS

Another technique for isolating failures is what we call a circuit breaker. And of course this comes from the idea of an electrical circuit breaker. **We want to avoid those situations where we have things like cascading failures, and we want to avoid having retries that put additional load on a system**. So again, what happens when a service depends on another that is overloaded? What calls to that overloaded service may fail? So, you know we make a call to that service, we get something like a timeout because it's unable to respond immediately and so we get a failure. **The caller may not realize the service is under stress and it may retry**. It's not uncommon when you get that kind of failure to say well let's just retry that because we want to make sure we get a success. That's a pretty common strategy. The problem is that the retry makes the load worse. **Every time you retry, the problem gets worse**. And as a result, you can actually end up in the situation where you push your overloaded system over the edge which then in turn leads to the cascading failure. So, we want to avoid that. We don't want those retries to put additional load on an already overloaded system. But **how do we know whether that system is overloaded?** We really can't when we get a timeout. There's no guarantee that the system is overloaded. It's possible there's a network failure or a temporary network glitch and the retry maybe would have been successful. **There's no way for us to know the difference between a timeout because the system is overloaded versus a timeout because of something else happening in the system**. So, **we just need to be careful to avoid this kind of situation**. How do we do that? We do that using what we call a circuit breaker.



A circuit breaker is a way to avoid overloading a service. What **they do is they quarantine a failing service so that it can fail fas**t. In normal operations our circuit breaker operates in the closed state. That's the green circle here. In the closed state any request that gets sent just goes through to the external circuit and everything works as normal. However, **in the event that we get a failure then we move into the open state**. You can see that's the red circle and that happens when we trip the circuit breaker or when we get some kind of failure. What that does is **it'll now cause it to fail fast**. Now when we make calls through the circuit breaker we don't even bother trying to talk to the external service. We just immediately fail the call, we give it an exception or a message, something like you know **the circuit breaker is open so this call is going to fail**. What that does is it means that the external service now has a chance to recover. We're not going to continue to put more and more load on it. **It does mean that our retries**

don't exist or can't exist because any retry would just immediately go into that failing fast state and they would immediately fail. So, we lose any retry capability here but at the same time that's not necessarily a bad thing because again we don't want to overload the system. This works even when retries aren't involved because you make all calls through this circuit breaker. So, in the event that it's not a question of retries, it's simply that your service is producing data faster than the client service is able to keep up. In that case this will kick in as well. What it will do is it will trip the circuit breaker causing the calls to fail fast and as a result your application will stop basically beating on the external application that's already struggling. It allows the failing service time to recover without overloading it. **What about when the service does recover?** What happens? **Over time when you build these circuit breakers these circuit breakers usually have a timeout of some kind. Essentially what happens is after that period of time the circuit breaker attempts to reset itself**. At that point, **it goes into what we call the half-open state**. That's the yellow the yellow circle here. In the half-open state **the next request that comes through it will allow it to go to the external service**. It's only going to let one through, but it will let the next request go through if the next request succeeds. Then it will reset, and it will go into the closed state again and everything will operate as normal. But on the other hand, if the service is still overloaded or it's still causing problems, then it will trip the circuit breaker again and go back into the open state allowing again the service more time to recover. Akka and Lagom both feature circuit breakers. These are built-in features that you can leverage. Again, they operate basically by making those calls through a circuit breaker. So, calls to the external service go through the circuit breaker and then they include a timeout essentially or a reset time that will determine when it can move into the half-open state.

3) MESSAGE DRIVEN ARCHITECTURE

When we introduce the Reactive principles, we talked about the fact that Reactive systems are message driven; they're based on a message driven architecture. We talked about the those systems using asynchronous non-blocking messaging. Let's talk a little bit more about those messages and why they're important. **Asynchronous non-blocking messaging allows us to decouple our system in time and failure**. It is a way of isolating our system. It means that **services aren't dependent on the response from each other**. Again, I think this is a place where it's helpful to think about this in terms of people. If you try to go about your day to day life, your job, in a synchronous manner that means that if you need somebody to do something for you, you will go to them and you will say to them hey I need you to complete this task for me: can you gather this information or complete this report or write this code or whatever it is? You will ask them to do that task and then you're gonna stand over their shoulder and wait for them to finish it. You're not going to continue doing your own work because if you do then you're no longer being synchronous. You're gonna sit there and wait for them to complete that work. Once they have completed that work then you will take the results of that and go on about your business. If you think about that from a productivity perspective, it's extremely inefficient. It basically means that you have two people doing the job of one person. The fact that you are standing over their shoulder watching means that you are unable to do anything and therefore you might as well not even be there. We should just have one person doing all the work instead

of two. That's very inefficient. If you flip that around and approach that from an asynchronous non-blocking perspective. in that case, what you do is you go to the person, and you say hey I need you to complete this task. Once you're complete I'm gonna be at my desk. Can you send me a message or something, let me know that it's done, then I can continue on. Then you go back to your desk, and you continue to do whatever other work you have. You can check your email, you can do other meetings, you can do whatever it is you need to do. You've taken the problem and you've turned it into an asynchronous non-blocking problem. That's far more efficient. We know that from our day-to-day life. We know that that's a better way to do things. That's what we're trying to achieve with software. **We want that asynchronous non-blocking messaging because it is more efficient**. When it is asynchronous and non-blocking like that, **it allows us to free up things like memory resources. It allows us to free up CPU resources. We're not occupying threads**. All of these things can be put aside and used for other tasks. We're not stuck occupying you know even something like a database lock. We're not stuck locked on the database while we wait for something to complete. This frees us up to do a lot of other things. It also means that if a request to a service fails, the failure won't propagate. If you have a service that does a synchronous request then when that happens and you get a failure, you send a request to the other service, it doesn't respond. First off, you're stuck waiting for a response that isn't going to come. Eventually you're gonna timeout and then you're stuck failing. In the meantime, you've occupied all of these resources for something that never completed, which is it's a waste of time. **By making it asynchronous, you send the request, you hope that a response will come back at some point, maybe you keep checking occasionally, whatever the case may be. But you don't occupy those resources in the meantime. If that failure does occur**, then because of the fact that you've made it asynchronous, **you'll hopefully have a way to deal with that failure**. Maybe you do the request again or something at that point. But the point is you're not going to just bubble that failure back to whoever requested something originally from you because the client isn't waiting for a response. It continues to operate normally. **If other requests come in the meantime, you can just go ahead and do them**. You're not gonna have to worry about there not being enough memory because it's currently being occupied by a request that's stuck in limbo. You're not gonna have to worry about a lock on the database that you left open while you waited for this response. All of those types of things go away when you build things in an async or some non-blocking fashion. As a result, we become more isolated in both time and failure. Our services are less likely to fail. They become less brittle and they're not dependent on things to happen immediately. They allow for some time to happen and that's a benefit. It results in systems that are going to be more robust.

4) AUTONOMY

We build systems using the principles of isolation to achieve what we call autonomy. **Autonomy is basically the idea that each of our services that we build can operate independent of one another**. Why is that beneficial? Why do we want this autonomy of your services? We'll keep in mind that microservices can only guarantee their own behavior. They do that through their API. Each microservice presents an API to the world and it says this is how I will behave. If you ask me this question, I will give you this answer. They can't guarantee what

other services are going to do. They can't guarantee that those services are going to be running. They can't guarantee that they'll be responsive. They may be slow. They may have other issues. They can only guarantee what they will do. It's again, this is just like people, I can guarantee how I'm gonna behave. I think it's very useful when we think about distributed systems to think about people because people represent a natural distributed system. **Micro-services only guarantee their own behavior. Isolation allows those services to operate independent of one another**. When services are very tightly coupled, and they're not isolated they can't operate independently. They rely on other people to do their work and as a result they become brittle. Again, if we think about that in terms of people, if I can do all of my work completely independently, that's a very efficient process. On the other hand if my work relies on 50 other people in order to get it done, it's probably going to happen very slowly. And it's probably going to be very difficult to coordinate. All of that things are gonna start to fall apart very quickly. What happens if somebody gets sick and can't do their part of the job? Making each service autonomous helps to solve that problem.

How do we make services autonomous? **We need to make sure that they have enough information to resolve any conflicts and repair failures**. What this does is **it means that they don't require other services to be operational at all times**. There's a number of different ways that you can do that. You know you can use things like asynchronous messaging. You can use various techniques for introducing isolation into your application. But generally speaking, **the idea is you need to isolate those applications**. **You can use techniques like making copies of data so that in the event that an external dependency goes down the service can rely on a copy of the data rather than relying on the external service**. There's lots of things that can be done here but again the end goal is to make it so that each service can operate completely independently of every other service for at least a short period of time, maybe not forever. At some point they might need those other services to come back but if they can operate at least for a short period of time in a fully autonomous way that's very beneficial.

What are the benefits? **Autonomy allows for stronger scalability and availability**. If you think about it, fully autonomous services, these are services that literally have no external dependencies, can be scaled indefinitely. You can just keep creating copies of them. They have no dependencies so there's nothing preventing you from scaling forever. Now the reality is a **fully autonomous system is a pretty rare** possibly non-existent thing but the closer we can get to fully autonomous the better situation we're going to put ourselves in. In addition, **operating independently means that they can tolerate any amount of failure**. If it's a fully autonomous system, then the entire world can fail around them, and they will continue to operate. Again, that's probably an unachievable goal but the closer we can get to it the better. So that's kind of what we're trying to achieve we're trying to get as close to fully autonomous as possible recognizing that we're probably not going to get all the way there.

How do we do it well, communicating only through asynchronous messaging? Again, if the moment we introduce a synchronous message now we've eliminated autonomy we, now depend on that response. So, **communicating only through asynchronous messaging improves our autonomy**. **We need to maintain enough internal state for the service to function in isolation. This may mean making copies of data, for example**. Those copies incidentally are going to be eventually consistent rather than immediately consistent or strictly consistent. That's important because it again it allows us to create these additional copies of

the data so that in the event that an external dependency fails, we can fall back to that possibly slightly out of date information. If you think about this from a banking perspective, if you go to your bank and you want to know the balance of your account, what would you really like to know? Do you want to know the balance of your account right at this second or would you like to know the balance of the account ten minutes ago. If you need to know it right at this second, then that means that you can't really tolerate any failures. If you need to know it say 10 minutes ago then that means theoretically the system that updates your balance could have failed 9 minutes ago and you're still going to be ok. Now if you're standing at the till at a store trying to buy something with your bank card… 10 minutes ago is probably sufficient for most use cases. You don't necessarily need up to the millisecond information at that point. So that's a way to help achieve autonomy. We also want to avoid any kind of direct synchronous dependencies on external services. Again, direct synchronous dependencies means that you can't do your job until you get back that response, which means you can't be autonomous. There's just no way you can do it, you now have a direct dependency on somebody else to do their job. So, these are the types of things that we want to do but you know at the end of the day, it's all about finding ways to isolate your microservices. **The more isolation you can put in, the more autonomous your microservices are going to be**.

5) GATEWAY SERVICES

Micro-services provide us a lot of benefits particularly in the form of isolation. However, they also **introduced some complexity into our application**. We saw when we talked about the difference between monoliths and microservices, that whichever option you pick, you're going to have advantages and disadvantages. Well one of the disadvantages of microservices is they can create complexity, particularly in your client application. If you think about a monolith we would make a single request from the client to the monolith. That monolith then probably needed information from a number of different locations, different tables. What it would do is it would do a join, typically on those tables, in order to access that information and then it would present it to the client. Now that's a relatively simple process. The client just makes one request, deals one failure. But when we move into microservices that gets more complex because now that information is not contained within any one micro-service; that's potentially contained within multiple microservices. You can't just do a database join because that violates the principles of isolation. It means that you're no longer isolating your state. So, **what we need to do now is the client actually needs to make individual calls to each microservice**. **Then it needs to aggregate the information and then present it**. **That means that the client needs to know about each of the different micro-services**. It needs to know they exist. It needs to deal with potential failures from each of those microservices, which again is additional complexity that we don't want to deal with. So, what has happened now is that all that complexity that used to be handled by the monolith has moved into the client. Why is that a problem? Well, it's a problem because clients are usually harder to update. If it's something like an iPhone app or an Android app you know we can't guarantee that a user is going to update that app, they may not. It also means that in order for them to update that app we have to go through Apple or we have to go through Google or whoever in order to provide the update. It's no longer under our control. That means that we're stuck in this situation where if we want to

deprecate a microservice or move it or add an additional microservice we can't do that until we ensure that all the clients are up to date. We don't know how long that could take. That's a problem and it's a problem that we want to try to solve.
The question is **how can we manage these complex API's that may access many microservices?** We do that **using what we call a gateway service** or an API gateway service. API gateway services **are essentially just microservices, other microservices, that we put between our individual micro-services and our client**. **Our client now makes a request to the gateway service and then the Gateway service knows about all the individual services. that it needs to talk to in order to fulfill that request**. We could do something like have just a client application Gateway service that handles all the requests. We can also make it a little more specific to the individual purpose that we're trying to do, a little more domain oriented. We could, for example, for the drivers delivery application we could have a Delivery Gateway service, for example. So, they can be domain specific. They can be more general if necessary. But basically, what happens now is the client makes a request to the Gateway service. The Gateway service makes all the individual requests, aggregates the results, passes it back to the client. Then if a failure occurs on one of the individual services, the gateway service is responsible for figuring out how to deal with that failure. That may mean servicing that failure back to the client, but it may also mean that the Gateway service has some sort of logic that allows it to recover from that failure. For example, it might cache some of the data and fall back to a stale cache in the event that the data is not available. That's an option that we could use in the Gateway service. So basically, what we are doing here though is we're creating an additional layer of isolation between our client and our individual micro- services. As we've already established, more isolation generally is a good thing. That is one of the primary goals of being reactive is to make things more isolated. This gives us a way to do that.