**Course Overview**

Hi everyone, my name is Rag Dhiman and welcome to my course titled Microservices Architecture. I'm an author of Pluralsight, but I've also been designing and developing software at a commercial level for well over 15 years and recently, I've had hands-on experience at implementing Microservices Architecture. In this course, we will look at the main design principles that make up this style of architecture. We will look at exactly what these design principles are and what they imply when it comes to designing your software. For years, people have developed software using a style which is the opposite of microservices and this has unfortunately resulted in large, rigid applications and services, and over time, these large, rigid applications and services and APIs become so complex that they are very difficult to change and deploy without introducing risk. The phrase monolith architecture is sometimes used to describe this large and unwieldy software architecture style and microservices architecture is the solution to avoid a monolith-type style. Microservices architecture and its design principles enable us to develop software that is easy and risk-free to change and deploy, and it let's us make softer architecture that is made up of small components that are each individually scalable, allowing us to scan out specific components that require improved performance. This also reduces risk during deployment, because we only have to deploy components that have changed instead of deploying the entire software architecture each time. There are many other design benefits that microservices architecture style brings to software. In this course, learn exactly what each one of those are and exactly which design principles to use. So, by the end of the course, you will learn all the design principles required to implement this architecture style and the exact approach for each one. I hope you will join me on this journey at Pluralsight to learn the design principles for microservices architecture with the course titled Microservices Architecture.

**Introduction**

Hi, this is Rag Dhiman from Pluralsight, and welcome to the Microservices Architecture course. And in this course you will learn how to architect complex software using Microservices architecture, so that it's flexible, scalable, and a competitive piece of software. And after completing this course, you will automatically start to think differently about software design and your architectural skills will become even stronger with a core set of design principles that will age you to architect better software. Okay, this course is split up into four modules. This first module will focus on introducing microservices and on introducing the design principles associated with microservices architecture. The second module will then look at the design approach that can be taken in order to implement the microservices design principles. On completion of the second module, how to implement a Microservice, it will become clear. The third module of the course will focus on the technology that can be used to implement microservices. Many types of technology are suggested, because the architectural theory in this course can apply to any platform. The final module of the course covers how you can move forwards with microservices, both in a Brownfield situation, and in a Greenfield situation. A Brownfield situation is where you're going to move an existing system forward with microservices architecture, and a Greenfield situation, you're basically developing a new system from scratch using microservices architecture. Okay, so in this module, we'll start off first by introducing microservices, and we'll also look at what was before microservices. We will then look at the emergence of microservices, why they are so successful and useful now,and then we'll conclude the module by looking at the design principles that are associated with microservices architecture.

**What Is a Service?**

Okay, we're going to start off by introducing the microservice architecture. The first thing we'll do is we'll question what a service is, and then we'll move on to introducing the microservices architecture itself. We will finish off this section of the module by looking at monolithic architecture. This is the architecture that came before microservices. Okay, let's start off by answering the question of, what is a service? Okay, a service is a piece of software, which basically provides functionality to other pieces of software within your system. It basically provides a service to other pieces of software. The other pieces of software could be anything from a website to a mobile app, or a desktop app, or even another service which uses another service in order to carry out a particular type of functionality. And, the service basically provides functionality to these applications. So, for example in the shopping website context, when a user places an order on the website, the website talks to the service, and the service actually carries out the creation, the update, the deletion, and the retrieval of what is from the database, so it provides functionality to the website application. And the communication between these software components and the service normally happen over a network using some kind of communication protocol. For example, a mobile app might communicate to a service via the internet. A system which uses a service or multiple services in this fashion is known to have a service oriented architecture, and this is normally abbreviated as S-O-A or SOA, and the main idea behind SOA is instead of using packet modules within each client application, I instead use a service to provide functionality to my client applications, and this allows me to have many client applications using the same functionality. And in the future, I can have newer or different types of clients connecting to the same service reusing that functionality. And as a software architecture, SOA has been successful. It allows us to scale up our software when demand increases, by enabling us to have a copy of the service on multiple servers, so when the traffic comes in, a load balancer will redirect that request to a specific instance of the service, and we can have multiple instances of the service, so when the demand increases, we just increase the number of instances of the service running across servers. We have already mentioned the fact that service oriented architecture provides reusability, reusability of functionality. So for example, the function to create an order on a website could be the same functionality which is triggered by a mobile app on our service. So, it's the same code creating an order for both the website and the mobile application. It allows us to reuse functionality. Another key characteristic of service oriented architecture is, is the idea of having standardized contracts or interfaces. When our client application called a service, it called a service by calling a method. The signature of that method normally doesn't change when the service changes, so we can upgrade our service without having to upgrade our clients, as long as the contract and the interface, either signature of the method doesn't change, we do not have to upgrade our clients when we upgrade our service. Another key characteristic of a service is, is the fact that they are stateless. So, when a request comes in from a website to our service, that instance of the service does not have to remember the previous request from that specific customer, that specific client. It basically has all the information from the request, that it needs in order to retrieve all the data associated with previous request within the service, so a service does not have to remember the previous call the client has made to that particular instance of the service. It's stateless, therefore any instance of the service can honor any income request from a client, because it does not have to remember any previous interaction with any other instance of a service. Now that we know what a service is, and what the service oriented architecture is, we can start introducing the microservices architecture. The microservices architecture is basically an improved version of service oriented architecture, and therefore it shares all the key characteristics of the service oriented architecture, of scalability, reusability, and standardized contracts in interface for backwards compatibility, and the idea of having a service that stateless.

**Microservices Introduction**

In this next section of the module, we're going to now introduce the microservices architecture. Microservices architecture is basically service oriented architecture done well. After years of doing service oriented architecture people have realized what service oriented architecture should be.And this is basically what microservices architecture is. It's an evolution of service oriented architecture. Microservices basically introduced a new set of additional design principles which teach you how to size a service correctly, because there was no guidance in the past on how to size a service and what to include in a service. Traditional service oriented architecture resulted in monolithic large services. And, because of the size of the service, these services became inefficient to scale up and to change in a reliable way. Smaller services, i.e. microservices. basically provide services which are more efficiently scalable, which are flexible. and we can provide high performance in the areas where performance is required. An application which is based on microservices architecture is normally an application which is powered by multiple microservices. And, each one of these microservices will provide a set of functions, a set of related functions to specific part of the application. A microservice normally provides a set of related functions to applications, to client applications, and client services, because the microservice normally has a single focus. It does one thing and it does it well. Microservice architecture also uses lightweight communication mechanism between clients and services, and service to service. The communication mechanism has to be lightweight and quick, because when you carry out a transaction within a microservices architecture system, the transaction will be a distributed transaction which is completed by multiple services. Therefore, the services need to communicate to each other in a quick and efficient way over the network. It needs to be a lightweight fast communication mechanism. The application interface for a microservice, either way you talk to a microservice, also needs to be technology agnostic. This basically means the service needs to use an open communication protocol so that it does not dictate the technology that the client application needs to use. And by using open communication protocols, for example like HTTP rest, we could easily have a .NET client application which talks to a Java based microservice. In a monolithic service, you're also likely to have a central database in order to share data between applications and services. In microservices architecture, each microservice has its own data storage. Another key characteristic of a microservice is that it is independently changeable. I can upgrade, enhance, or fix a specific microservice without changing any of the clients or any of the other services within the system. And because microservices are independently changeable, they also need to be independently deployable by modifying one microservice. I should be able to then deploy that change within my system independently from everything else, without deploying anything else. We've already mentioned the fact that when you make a transaction within a microservices architecture system, the transaction is most likely to be completed by multiple services, multiple services which are distributed, and therefore your transaction is also a distributed transaction. And because a microservices architecture system has so many moving parts, there's a need for centralized tooling for management of the microservices. You need something at all which will help you manage and see the health of your system, because there are so many moving parts. Okay, now let's have a look at a high-level architecture diagram for a microservices system. This is an example of a typical ecommerce system and as you can see, on the left hand side, a shopping website is running in the customer's browser. The browser connects to our shopping websites via the internet and our shopping website might be an ASP.NET MVC website which is running on the IIS. All the processing required for all the interactions with the website is actually carried out by a number of microservices which are running in the background. Each microservice has a single focus or a single set of related functions and each microservice also has its own data storage and it's also independently changeable and deployable. So for example, I could upgrade the orders service without upgrading any other part of my system. There might also be multiple instances for each type of microservice. For example, if the orders service is in demand, we might have several instances of the orders service in order to satisfy demand, and in order to direct a request from the shopping website to the correct instance of an orders service, we have an API Gateway. We will cover the API Gateway in more detail later on, but for now, think of it as something that manages and routes a request to the correct microservice within our system. So in this example, when the customer places an order, the shopping website might use multiple services and multiple functions within those services in order to satisfy the transaction.And this is why in a microservices architecture, a transaction is normally a distributed transaction, because the transaction is actually satisfied by multiple pieces of software, i.e., our microservices, in order to complete the transaction. And that's why the communication between these microservices needs to be super fast and lightweight in order to complete the transaction quickly. Throughout the rest of the course, you will see how design principles can be used to architect a microservices architecture like this.

**The Monolithic**

Okay, in this part of the module, we're going to look at the monolithic type of system. This is basically the type of system that came before the microservices movement. These are large systems that almost do the opposite of what microservices are trying to achieve. Okay, so the typical monolithic system is basically your typical enterprise application, and this application might be in the form of a large website with all the modules packaged in together into one package, or it could be in the form of a service which talks to a website and the service itself is a large service with all the modules packaged together as one executable. The key characteristic is as you add stuff to your application, it keeps growing. There's no restriction in size. There's no division. There's always one package which basically contains everything, and therefore, you also end up with a large codebase, and because the codebase is so large, it might also take the team longer to develop new functionality within the application. It might be the code is so intertwined, that it's difficult to make a change without affecting other parts of the system, and therefore, testing takes longer. Deployment of a large system can also be challenging, because even for a small bug fix you are having to deploy a new version of the entire system, and therefore, that creates greater risk. And because there's so much code and so much intertwined code, there might be functionality in one of our modules within our overall package that might be useful to an external system, but because it's hidden within a monolithic application, it might not be exposed via the service, and you might have features in there which are not accessible. And because it's one large codebase, we're also stuck with one technology stack that might be a new technology. For example the promotions part of our system could do with using, but it's a new technology which is different to our current technologies stack, but because promotions is part of the overall package, we can't use that new technology within the promotions module. It makes our overall system less competitive, because we can't easily adopt new technologies which might give us a competitive edge. And because all the code is in one large package, we might also have high levels of coupling, which basically means if I change one part of the system, it might affect another part of the system, because the code is intertwined. And this kind of coupling might be present between modules, and it might also be present between different services. So, one large service might be intertwined with another large service, because if we change the signature of one service, it affects the other service. And because it is one package, if a part of the system fails, it could affect the whole system. For example, if the account processes gets stuck, it might affect the functionality of the overall service. It might degrade the performance of the overall service. Even scaling up this service to meet demand is quite inefficient if let's say for example, the orders aspect of the system is in demand, we would have to create a copy of the whole package, of the whole service in order to scale up just the order section. This basically means we need to buy more powerful servers every time we need to scale up, because we are forced to scale the entire service up instead of just the part which needs scaling up. And because the footprint of the application is so large, we might actually need to buy powerful resource in order to run our entire application. We might have to place the service on a single server each time. When we do make a code change to the actual service, the time to compile the entire application will also be longer, because there's just more code, and there's more unit tests to run against the entire code base. The only one advantage a monolithic system has over a microservices system is the fact that you can run the entire code base on one machine, so when I'm developing and testing, I could probably replicate the entire environment on my machine, because it is just one thing to replicate and configure. Okay, here we have another example of a monolithic system. In this example we have a website application which packages in all the other modules from our system within one package. So, this could be an ASP.NET MVC site, where the website itself is the UI layer. In the business layer, you have your Accounts, Orders, Promotions, Inventory, and Products namespaces, which have classes related to each section. So, even though you're using namespaces to divide the code within our package, the code might be still quite intertwined, basically coupled, so changing one aspect might affect another aspect. So, change in Account might negatively affect Orders. For example you might change the signature of one of the methods in the Accounts class, and because you've changed that signature, you may have to reflect the code in one of the Orders classes, or in one of the Promotions classes. So, this means a change that was only actually relevant to the Accounts part of the system is now impacting other parts of the system. The application might be further coupled by having a one database. So for example, schema change within the database, for example changing the datatype on one column might result in several areas of the application requiring refactoring. You can probably see now why development times are longer with a monolithic system, and why deployment is such a challenge. There is just greater risk.

**Emergence of Microservices**

Okay, in this section of the module, we're going to look at the emergence of microservices, basically why a microservice architecture is useful now, and what are the benefits. One of the reasons for the microservices architecture now is, is the need to respond to change quickly. The software market is really competitive nowadays. If your product can't provide a feature that's in demand, it will lose market share very quickly. And, this is where microservices can split a large system into parts, so we can upgrade and enhance individual parts in line with the market needs. So not only do we need to change part of a system quickly, we also need to change them in a reliable way in order to keep the market happy, and microservices provide this reliability by having your system in many parts, so if one part of the system breaks, it won't break the entire system. There is also need for business domain driven design. The architecture of our application needs to match the organization structure or the structure of the business functions within the organization. So, if for example Accounts are having a massive overhaul of their software, we can just change the account service without affecting any of the services related to other departments. There won't be any issues between departments if one department decides to change their software, because the software architecture matches the organization structure. Another reason why microservices architecture is now possible, is because we now have automated test tools. We've already seen that in a microservices architecture our transactions are distributed, and therefore a transaction will be processed by several services before it's complete. Therefore the integration between those services needs to be tested. And testing these microservices together manually might be quite a complex task, but the good news is these automated tests automatically test the integration between our microservices, and this is why microservices architecture is now possible, because we have automated test tools which test integration between services. Release and deployment of microservices can also be complex, because remember we now have multiple services, i.e. multiple pieces of software that we need to copy to servers or onto the cloud, and it might be that for each service type, we also have multiple instances that we need to operate at the same time. The good news is for release and deployment, we also have tools, centralized tools, which can carry out this function. Another reason why so many moving parts is no longer an issue any more, is the fact that we can host them using on demand technology. We can basically request things like virtual machines in order to host our microservices on demand. We basically no longer need to deploy our software to physical servers. We can instead just have a server which provides a cloud of virtual machines, and we can basically just clone these virtual machines and say deploy our microservices onto these virtual machines on demand, requesting them in software without having to do any physical work in order to configure and deploy the service onto a new host. On demand hosting is even now even more simpler. With cloud services on the internet, you can now basically spin up a machine in the cloud without even owning a physical box, and basically host your virtual machines, with your microservices running in them in a cloud service, which provides all the functional you need to manage and monitor you microservices. Another reason for the need to adopt microservices architecture is, is the need to adopt new technology, because our system is now in several moving parts, we can easily change one part, i.e. a microservice from one technology stack to another technology stack in order to get a competitive edge. Another advancement in technology which makes microservices possible is the asynchronous communication technology. In our microservices architecture, when we use the distributed transactions, the distributed transactions might use several services. in order to complete. Using asynchronous communication, the distributed transaction does not have to wait for individual services to complete their tasks before it's complete. We will look at asynchronous communication technology in more detail in the technology module later on. Another reason why microservices architecture are possible now is the fact that we have simpler server side and client side technology. There are a number of technologies we can choose from, both at the server side and at the client site. And, there are many open communication protocols which allow the server side and client side technology stocks to work together quite happily. Okay, let's now highlight the key benefits of the microservices architecture. Okay, so one of the key benefits is that microservices have shorter development times, because the system is split up into smaller moving parts. You can work on a moving part individually. You can have teams working on different parts concurrently, and because microservices are small in size, and they have a single focus, and the team have less to worry about in terms of scope, they know the one thing they're working on has a certain scope, and there's no need to worry about the entire system, as long as they honor the contracts between services. And because these services are loosely coupled, developers can rework, change, and deploy individual components, without deploying or affecting the entire system, and therefore deployment is more reliable and faster. Shorter development times and reliable and faster deployment also enable frequent updates. As we've already briefly mentioned, frequent updates can give you a competitive edge in the marketplace. The microservices architecture also allows us to decouple changeable parts. For example if we know, UI for our system, our user interface for our system changes quite often, if it uses the microservices architecture, the UI is most likely decoupled from all the services in the background, and therefore you can change it independently from all the services. Microservices architecture also increases security. In a monolithic system, you might have one central database, with one system accessing that database, and therefore all you need to do is hack that one system in order to gain access to the data. In the microservices architecture each microservice has its own database, and each microservice can also have its own security mechanism, therefore making the data distributed, and making the data even more secure. Microservices architecture also offers you increased uptime, because when it comes to upgrading the system, you will probably deploy one microservice at the time without affecting rest of the system. And because the system is split up into business domains and business functions, when a problem arises, we can probably quickly identify which service is responsible for that specific business function, and therefore resolve the problem within that microservice. Microservices architecture also make the system highly scalable, and it gives the system better performance. When there's a specific part of the system which is in demand, we can just scale that specific part up instead of scaling the whole system up. So if for example the inventory service is in demand, we can create many instances of that microservice without duplicating the entire system. We can also give the ownership of a microservice to a particular development team, so that there's better ownership and knowledge about the microservice. We've already briefly mentioned that microservices allow us to use the right technology for specific parts in the system. And because each microservice is separate from the other microservice, they don't share databases, and they have their own codebase, you can easily have microservices being worked on concurrently by distributed teams. In the next section of the module, we'll start looking at the design principles that enable microservices, and enable these benefits that we get from microservices.

**Design Principles Introduction**

In this section of the module, we will start looking at the design principles that make a service a microservice. I've summarize the design principles into six design principles. So, in order for your service to be a microservice, the service needs to have high cohesion, it needs to be autonomous, it must be business domain centric, and it must have resilience, and it also must be observable, and automation should also be used throughout the development process. In the next few sections of the module, we will cover each one of these design principles in detail, and learn exactly what they mean.

**High Cohesion**

Okay, so what do we mean when we say, a microservice must have high cohesion? We are basically saying the microservice's content and functionality, in terms of input and output, must be coherent. It basically must have a single focus, and the thing it does, it should do well within that single focus. So for example you might have a service, a microservice, which has a single focus of calculating postage. So, all the inputs and outputs from this microservice are solely focused on something around to do with calculating postage. And this idea of a microservice having a single focus or a single responsibility, is actually taken from the solid coding principles. And the single responsibility principle basically states that a class can only change for one reason, and this same principle is applied to microservices, it’s a useful principle, because it allows us to control the size of the service, and we will not accidentally create a monolithic service by attaching other behaviors into the microservice, which are not actually related. Our Postage service for example, only has one reason to change. It only changes if something to do with the postage calculation or the postage logic changes. We then enhance or upgrade our microservice with that new additional functionality. And the reason for change, i.e. the responsibility, normally represents a business function or a business domain. Our postage calculating microservice is a business function, and an Account service to do with the accounts department, might represent a business domain. High cohesion is also like the encapsulation principle from our old programming principles. We take all data and functionality that's related, and we package it into one package, which is the microservice, because the high cohesion principle controls the size of the microservice, and the scope of the contents of the microservice. The microservice is easily rewritable, as we are likely to have less of an attachment to a smaller codebase. And obviously, there will be fewer lines of code to rewrite because the microservice will be so small. And overall, if all our microservices have high cohesion, it makes our overall system highly scalable, flexible, and reliable. System is more scalable, because we can scale up individual microservices which represents a specific business function or a business domain which is in demand, instead of scaling up the whole system. And at the same time the system is more flexible, because we can change and upgrade or change the functionality of specific business functions or business domains within our system. And then we have reliability, because overall we are changing specific small parts within the system without affecting other parts within the system.

**Autonomous**

Microservices should also be autonomous. By autonomous we mean, a microservice should not be subject to change because of an external system it interacts with, or an external system that interacts with it. we are basically saying there should be loose coupling between the microservices and between the microservices and the clients that use the microservices. And by loose coupling we mean, a change to a microservice should not force other microservices to change or other clients to change. This means microservices must honor contracts and interfaces to other services and other clients, which basically means the way the input and output are formatted for a microservice should not change between versions, because that might break any other services trying to interact with that microservice, using those inputs and outputs. Like a website, a microservice should also be stateless. There should be no need to remember previous interactions that clients might of had with this service or other service instances in order to carry out the current request. And because microservices honor contracts and interfaces to other services and clients, they should be independently changeable, and independently deployable. They should just slot back into the system after a change or an enhancement even though it has a newer version than any of the other components within the system. This also ensures our service is always backwards compatible. Having clear defined contracts between services also means that microservices can be concurrently developed by several teams. Because there's a clear definition of the input and output of a microservice, separate teams can work on separate microservices. As long as they honor the contracts,development should go okay.

**Business Domain Centric**

A microservice should also be business domain centric, and by this we mean that a service should represent a business function. For example, in the organization you might have an accounts department that has a lot of accounting software and accounting functionality. And this account department might end up having code which results in a microservice to do with accounting functionality. You might also have specific business functions, for example calculating postage which might also end up within a microservice as a microservice itself. The overall idea is, is to have a microservice represent a business function or a business domain, i.e. a part of the organization, because this helps scope the service and control the size of the service. This is an idea which is taken from domain driven design. You basically define a bounded context, which basically contains all the functionality which is related to a specific part of the business, to a business domain, or a business function. And you define the bounded context by defining boundaries and seams within the code. You basically highlight the areas where related functionality exist. So for example on our diagram on the left, you have code which is related to the accounts department and give code which is related to the orders department, and the contents within each section becomes the bounded context, and overall this will eventually become our microservice. There will be times when code relates to two different bounded contexts, and this is where we need to shuffle the code around, so the code ends up in the right place, where it makes sense and it belongs, in terms of business function or business domain. We need to aim for high cohesion, remember? Making our microservices business domain centric also makes our microservices responsive to business change, so as the business changes, or the organization changes, or functions within the business change, our microservices can change in the same way. Because our system is broken up into individual parts, which are business domain centric, we can also change those parts which relate to specific parts in the organization which are changing.

**Resilience**

Another key design principle for microservices is resilience. We basically need to embrace failure when it happens. Failure might be in the form of another service not responding to your service. or might be a connection line to another system which has going down, or it might be a third party system which fails to respond. Whatever the type of failure, our microservice needs to embrace that failure by degrading the functionality within our microservice, or by using default functionality. An example of degrading functionality might be a scenario where we have a user interface microservice which basically draws a HTML page for available orders and promotions, but for whatever reason, the Promotions microservice is down and fails to respond. So, our user interface microservice basically chooses to degrade that functionality, and it chooses not to display the promotions on the page. An example of default functionality might be, a postage microservice. If the postage microservice goes down and doesn't respond, and the Orders microservice relies on the Postage microservice, when it detects the fact that the Postage microservice is not responding, the Orders microservice might use a default postage rate for the order instead of retrieving one from the Postage microservice. Another way of making microservices more resilient is by having multiple instances of microservices, so they register themselves as they start up, and if any of them fail, they deregister themselves, so our system, or our load balancers, et cetera, are only ever a way of fully functioning microservices. We also need to be aware there are different types of failures. So for example that might be exceptions or errors within a microservice. There might be delays in replying to a request, and there might also be complete unavailability of microservice, and this is where we need to work out if we did need to degrade functionality, or if we need to default functionality. Failures are also not just limited to the software itself. You might have network failures. And remember, we're using distributed-transactions here, where one transaction might go across the network and use several services before it actually completes. And therefore again, we need to make our microservices resilient to network delays or unavailability. We also need to ensure that when our microservices are called, and the input they receive as part of their request, that we can validate that input. And, this might be input from services, or from clients. We need to ensure that our microservices are resilient and can validate incoming data, and they don't basically fall over,because they've received something in an incorrect format.

**Observable**

Another key design principle that we need to build into our microservices architecture is, is the idea of our system being observable. We need a way to be able to observe our system's health in terms of system status, in terms of logs, i.e. activity currently happening in the system, and errors that are currently happening in the system. And, this type of monitoring and logging needs to be centralized, so that there's one place where we need to go to in order to view this information regarding the system's health. And, we need this level of monitoring and logging in a centralized place,because we now have distributed transactions. In order for a transaction to complete, it must go across the network and use several services.Therefore, knowing the health of the system is vital, and this kind of data would also be useful for quick problem solving. Because the whole system is distributed, and there's a lot going on, we need a quick way of working out where our potential problem possibly lies. And because we are also using automated tools for deployment, which means our deployment will be very quick, we also need a quick way of getting feedback in response to deployment, so if there are any issues we can clearly see from a centralized place. This data collected can also be used for capacity planning and also for scaling up of our system. So, when we can see that there's a clear demand somewhere within our system for a specific microservice, we can scale that area up. Orm when we know we've got let's say, 100 million customers going live soon, we can use this data to work out exactly how we can plan the capacity of our system in the future. We can also use this data to work out what parts of our system are actually used, and we can also build in some measures, in terms of logging in order to measure specific things which are related to the business, for example the number of sales on a daily basis. We can all log this to our central logging system, and view this from a centralized place.

**Automation**

We also need to feature automation in our microservice architecture, automation in the form of tools, for example tools to reduce testing. Automated testing will reduce the amount of time required for manual regression testing, and the time taken to test integration between services and clients,and also the time taken to set up test environments. Remember, in a microservices architecture, our system is made up of several moving parts, and therefore testing can be quite complex, and this is what we need testing tools to automate some of that testing. We need tools, automated testing tools, which gives quick feedback. So, as soon as I change a microservice and check that we have code in, into our sort control system, I want these test automation tools to provide me feedback on integration, to give me confidence that my change integrate with the entire system. And this type of testing, this type of automated testing which test for integration, is known as continuous integration. As well as automation tools to help with testing, we need automation tools to help with deployment, a tool which basically provides a pipeline to deployment. It gives our microservice a deployment release status, so when you check a change in, the test pass, and then the deployment status is at ready, then the tool knows that this build of the microservice is now ready for deployment. So, not only does this tool provide a pipeline, or the status for each deployable build of a microservice, it also provides a way of actually physically moving the build to the target machine, or the target cloud system, so the physical deployment of the software will be all automatic and therefore will be reliable, because it's preconfigured with the target, where the software needs to go, and it will be configured and tested once, and therefore it should work every time. The idea of using automation tools for deployment falls under a category called continuous deployment. So in order to have automation, we need to use continuous integration tools and continuous deployment tools, because in a microservices architecture, we have a system which is a distributed system and there are multiple instances of the services. This is a much more complex system and the way to organize across our system and you know with such a distributed system, manual integration testing would be too time-consuming and manual deployment will be too time-consuming and unreliable.

**Summary**

Okay, in this module, we introduced what microservices are. We first started off by looking at services themselves and we looked at how microservices have evolved from services. We then went on to see what came before microservices. This is the monolithic type of system, a system with a large codebase, a system that's normally one hold large package. We also looked at why the microservices architecture is relevant now and the main benefit of the microservices architecture in today's world. We then concluded the module by looking at the design principles that make a system of microservices architecture. We concluded that microservice needs to have high cohesion, it needs to be autonomous, and it must be business domain centric, and it must have some kind of resilience in there, and the microservice architecture must be observable, and there needs to be some degree of automation within the microservices architecture. In the next module, we will look at the design approach you need to take in order to apply this design principles to your architecture in order to make it a microservices architecture.

**Microservices Design**

**Introduction**

Hi, this is Rag Dhiman from Pluralsight. Welcome to the Microservices Design module. In this module, we will look at the design approach for each one of the microservices design principles. We will start off by reminding ourselves what the design principles are and for each one, we will look at the design approach we can take to implement the principle. Let's remind ourselves what the design principles are. A microservice must have high cohesion. It must do a single thing and it should do it well. It should have a single focus. This helps control the size of the microservice. A microservice should also be autonomous. It should be independently changeable and deployable because this lets us upgrade specific parts of our system without bringing risks to any other parts. Microservices also should be business domain centric. A microservice should represent a business function or represent a business domain. This is because microservices should be in line with the overall organization structure. This will mean if specific departments change and their systems change, the change does not affect other departments in their systems, i.e. their microservices.Microservices should also have resilience. They should embrace failure and default or degrade functionality when failure occurs. This is so our overall system doesn't fail because one of the microservices has stopped working because it's waiting for a response from a third party system or another service which has gone down. Microservices system should also be observable. We should be able to see the overall system health and they should be centralized logging and monitoring. This is because the microservices architecture can be complex because there are so many moving parts,therefore, we need a way of quickly problem solving when things go wrong. Microservices architecture should also have automation. We should have tools for testing and feedback and tools for deployment. This again is because microservices architecture can be complex. There can be many microservices and many instances of microservices and we need tools to test the integration between these microservices and to help with the deployment of these microservices into the production environment. In the remaining parts of the module, we will look at each one of theseprinciples in detail and the design approach we need to take in order to implement each one of these design principles.

**High Cohesion**

Let's start off with the high cohesion design principle. How do we implement a microservice with high cohesion? The first thing we need to do is we need to identify a single focus for our microservice. This might be in the form of a business function. For example for an account system, generate an invoice, a function which has clear inputs and outputs. Or it might be in the form of a business domain where the microservices focus is in the form of creating, retrieving, and updating, deleting data related to a specific part of the organization, for example, the accounts department. The key thing is we do not crowd our microservice with both types of focus in one microservice. For example the diagram on the left here. Our account service does all the CRUD operations to deal with data for the accounts department, and they also generate invoices for the accounts department. The generate invoice business function here, although it's associated with the accounts department, it should really have its own microservice because it has its own clear inputs and outputs. It should be its own microservice with its own database to store the invoicing data. We can now create and retrieve our general accounts data using our accounts service which has a separate database. Therefore, if we make changes to the accounts database schema, like for example adding a new column, this does not affect our accounts invoicing functionality. We must also be ready to split our microservices further if for example our inputs and outputs on a microservice change. Our business function of generating invoice might have an additional requirement of generating an invoice and sending it via EDI, via electronic data interchange. And therefore instead of building that functionality into the existing microservice, we should really create a separate microservice which carries out the extra functionality. The two microservices can then work together in order to achieve the goal of sending an invoice via electronic data interchange. The EDI invoicing service can then also store the EDI formatted data in a separate database and then in the future, if the EDI interface have changes, we only have to change one of our microservices and then there's no risk of breaking the old account invoicing service. What we need to do is when we're looking at the single focus of a microservice is a to avoid the thinking of it's kind of the same business function and then coupling multiple business functions into one microservice. A microservice should only have one reason for it to change. This is because we want to avoid the change in one business function breaking another business function. And in our EDI example, we don't want the EDI format change to change our general invoicing functionality. We also need to avoid getting lazy because creating an extra microservice or splitting an existing microservice will require extra effort but you've got to remember the overall objective. Our overall objective is to have a system which is scalable, flexible, and reliable. We want our system to be in separate parts so we can enhance and deploy specific parts and the laziness to create multiple microservices will only result in a system which is basically a monolithic system and it will have all the disadvantages which we've discussed earlier that had to do with monolithic systems. So don't be afraid to create many services. And remember, you won't create thousands of microservices overnight. This will probably happen in an incremental way and therefore we will have time to learn how to maintain and monitor these microservices. Overall to ensure your microservices have high cohesion, continuously question the design of microservices and this can be done in code reviews or peer reviews where you question if a new microservice has one reason to change.

**Autonomous: Loosely Coupled**

For our microservices to be autonomous, i.e. independently deployable and changeable, our microservices should be loosely coupled. Basically our microservices should depend on each other in a minimal way and they should have the least amount of knowledge about each other. This can be achieved by having our microservices communicate to each other over the network. A bit like how machines are connected to each other over a network. They're not physically connected to each other directly but they use the network in order to talk to each other. This communication can be synchronous when a microservice calls another microservice and then wait for a reply. The good thing with synchronous communication is that we know our communication is successful because we receive a response in response to our request. To make synchronous communication decoupled, the service that's being called should respond straight away even before it's completed the actual task for the request. This is so the microservice that's making the request can carry on with a list of whilst the task is being processed. Once the task is complete, the service which has completed the task called back the original microservice that made the request in order to notify that the task is complete. And in order for this to work, the original request from the microservice should include a callback address. This is so that the microservice that has completed the task knows exactly which microservice to call back in order to notify that the task is completed. To make our microservices architecture even more decoupled, we could instead use asynchronous communication. This is why instead of making request, you instead publish events and these events are then handled by a message broker. Then other services listen out for these events and carry out the task. This is the most decoupled way of communication. Instead of microservices talking to each other directly, they instead publish events in the form of messages on a message queue, then interested microservices then pick up these messages, i.e. events, from the message queue and process them. Then when the task is complete, the microservice which has completed the task will publish another event in the form of a message in the message queue which the original microservice will pick up in order to learn the fact that the task has been completed. In order to ensure our microservices are further decoupled in terms of technology, we should use open communication protocols so that we have a technology agnostic API. Basically one technology stack doesn't force other microservices to have the same technology stack. For example, by using the open communication protocols like REST over HTTP, we could have a .NET based service talk to a Java based service. To further decouple our microservices, we should also avoid the use of client libraries. This is where the consumer of your microservice requires the implementation of a client library which you provided in order for that consumer to talk to your microservice. Client libraries increase coupling because if your microservice changes and it changes its client library, the consuming microservice will also need to change the implementation of the client library. The use of client libraries can also force the use of the specific technology platform of the consuming end. Microservices that talk to each other also should have contracts between the services. This is basically fixed and agreed interfaces between the two services. This is so when microservices change, i.e. they are enhanced or modified, the consumers don't have to change because the way of talking to the service still remains the same. This is in terms of method signatures and the format of the data that's exchanged, and when exchanging data, always use shared models. Shared models of the data, they are unlikely to change when the microservices are enhanced. The shared models should be different from the internal representation of the data within the microservice. So for example, if we add an extra field to our order internal model,a field that's not actually required by the account service, we do not have this extra field to our order shared model and the account service will carry on working because that contract remained intact and it doesn't have any surprises in terms of the new piece of data that it doesn't understand. So always keep the internal representation of data separate from data that's exchanged by using shared models to transfer the data. Agreed contract and interfaces are also important so that when multiple teams are working on different microservices, they clearly know the inputs and outputs for each microservice. To further keep our microservices decoupled, we should also avoid chatty exchanges between two microservices. Too many exchanges between two microservices in order to complete the task further couples the two microservices because the change at one end might change the sequence of the exchanges at one end and therefore forcing the other end to also change in order to cope with the change in exchanges.The sharing between two microservices should also be avoided especially when it comes to sharing things like databases. Sharing a database might seem like a good idea in order to share data quickly between two microservices but a change in the shared database will result in probably both microservices having to be changed in order to cater for the new schema change, and therefore you'll have to deploy two microservices instead of one because of one change. It also forces our microservices to use the same database technology. If each microservice has its own database, we can use specific database technology that's suitable for that specific microservice. Data instead should be shared by microservices calling each other and sending the data. We should also minimize the use of shared libraries within microservices because a bug fix in a shared library might mean we have to redeploy two microservices because they carry the same bug. If there is a demand for a functionality in the form of a shared library, maybe that function itself should be a microservice which could serve other services. In the next section of the module, we will look at how versioning can be used to make our microservices more autonomous.

**Autonomous: Ownership and Versioning**

Another way of making microservices autonomous is by having them owned by a team. If small teams are responsible for a microservice, i.e. they own the microservice, there'll be better knowledge retention about the microservice and also you can make the team responsible to make the microservice autonomous. It's the team responsibility to design the microservice so that it's independently changeable and independently deployable. The teams can also be responsible for agreeing to contract between the microservices or the microservices that interact with their microservices. If a team owns a specific microservice, they will also maintain the contract so that future changes don't break contracts with other microservices. Teams that own specific microservices should also be made responsible for the long-term maintenance of that microservice. Ownership of the microservice will also encourage the team to collaborate more with other teams in order to communicate contract requirements and communicate data requirements. Team ownership will also encourage concurrent development. Multiple teams can work on different microservices at the same time and agree with all interactions between them, between the teams. We have already mentioned the fact that autonomous microservices must be independently changeable and independently deployable. This means when you create a new version of a microservice, we need to think about our versioning strategy for that microservice. The key thing is when we create a new version of a microservice that we avoid breaking changes. Other services that consume our microservice must not break because we've changed something in the new version of the microservice. Our primary goal should be that all new changes and enhancements are backwards compatible. Microservice that use to use our microservice that's been upgraded should carry on working without any change. We should honor the original contract that was agreed. And in order to ensure that your new microservice doesn't break any existing contract, you should use integration test. Integration test which basically test the changed microservice for inputs and outputs and shared models basically testing to see if the original contract is still intact. Sometimes however you can't avoid breaking changes maybe because you've rewritten the microservice and its inputs and outputs have substantially changed. This is when you need to have a versioning strategy. So if your new version of a microservice includes breaking changes, you could have concurrent versions of the microservice running. You could have an old version and the new version running at the same time. This is specially required when you have no control over the consuming microservices and you have to have the old version of your microservice in place in order to ensure that the consuming microservices carry on working. Over time, you can then slowly negotiate the migration of the consuming microservices from the usage of the old microservice to the usage of the new microservice. To make transparent what version of the microservice are backwards compatible, we should use semantic versioning where the version number is made of three numbers, Major.Minor.Patch. You increment the major number if the new version of the microservice is not backwards compatible and you only increment the minor number if the new version of the microservice is actually backwards compatible and you increase the patch number if the only change in the new version of the microservice is the defect fix and the overall microservice is still backwards compatible. Sometimes we may choose to include both the old code and the new code in the new version of the microservice. This is where we can have coexisting endpoints so you have the original endpoint which points at the original code, the old version of the code, and you have a new endpoint which point to the new version of the code. Your consumers can then slowly migrate over time from using the old endpoint to using the new endpoint. You could even have a new version of a microservice which has the old endpoint, but the old endpoint is basically a wrapper for the new endpoint. So although existing consumers can code the old endpoint, the old endpoint basically redirects the code to the new endpoint. It's basically a wrapper for the new endpoint. So overall when we say microservices need to independently changeable and deployable, we need to ensure that when we do create a new version of a microservice, it does not break existing contracts and existing microservices that consume our new microservice.

**Business Domain Centric**

The business domain centric design principle for microservices basically suggests our microservice should present a business function or a business domain and the approach should be initially you define these business domains in a coarse manner. And these business domains will most likely represent a department or an area within the organization. We then need to further split these areas into possible business functions or business areas. It might be that the business domain is so simple that one microservice can contain all the functionality required for that business domain and further splitting is not required, and therefore we need to review the benefits of splitting the microservice further. A key thing to remember at the same time is the other design principle of a microservice having high cohesion, a microservice must do one thing and do it well. It must have a single focus and only one reason for it to change. It's ideal for our microservice as components, maps to a different component and functions within the organization, so when certain parts of the organization change, we know which specific microservices to change. Also when mapping our microservices to our organization, it's also key to agree a common language. So if for example we have a business function to deal with EDI invoicing, we must ensure we call our microservice to deal with that functionality, the EDI invoicing microservice. When splitting our microservices,we also need to remember that we can split microservices in order to have microservices just for data in order to create, retrieve, update, and delete data and/or we can have microservices which specifically carry out specific function, a business function. We must also be ready to fix incorrect boundaries. For example if you have a microservice which needs to be split further, you must be ready to split that microservice. On the same hand, if you have two microservices which are more or less doing the same function, you should be comfortable with the fact that you can merge two microservices into one microservice. When splitting our system into microservices, we also need to think about the inputs and outputs for each microservice because this is the explicit interface of the microservice to the outside world, and later on, these are the contract that exist between our microservices. Sometimes we may need to split our system by technical boundaries. This might be because we need a special microservice for example to access archive data or we might need a special microservice in order to improve performance in a specific area. So overall our microservices should really present business functions or business domains within the organization, but sometimes there might be exceptions of where you've got a microservice for some kind of technical reason.

**Resilience**

Another key design principle for microservices is resilience, especially when in a microservices architecture, we have so many moving parts, and if one specific microservice failed, we can't have the entire system going down because of one failure. So during the design stage of our microservice,we need to design our microservices for all known failures. The next question is, what failures are known to us? The known failures are basically failures of downstream systems. Systems which basically our microservice relies on in order to carry out a specific task. These might be internal services or external services. So for example in our diagram, we have our order service which relies on the postage service in order to calculate the postage for an order. So at design stage, we question what happens if the postage service goes down and we can't get the postage rate from the service? This is where we need to design our orders service to either degrade functionality on failure detection or to default functionality on failure detection. In our system, when someone's placing an order and the postage service goes down, we might choose not to display the postage cost and instead display a warning that postage at this stage cannot be displayed. We basically degrade the functionality of the system or we can default the functionality. So instead of using the postage rate from the postage service, we instead use standard rate. Basically default the functionality. So when we design our system to handle failure, another thing we need to do is to design our system so that it fails fast. Because there's no point degrading functionality or defaulting functionality if our transaction takes a lot longer to complete. A hanging transaction or a delayed transaction might be as bad as seeing an error in the screen, so instead we need the system to fail fast and recover fast. So as far as the user's concerned, nothing has actually happened. And in order to fail fast within our system, we need to use timeouts. Timeouts are commonly used between connected systems. Timeouts basically allow us to say if something doesn't respond within the given time, assume that they have failed. They basically allow us to timeout our request after a certain threshold. So for example in our postage service example, if we have a timeout of five seconds, if the postage service doesn't reply within those five seconds, we know the postage service is down and therefore we can fail fast. We can then quickly degrade or default our functionality. We should try and use timeout functionality everywhere between service to service communication and service to other systems communication. We also should specify a standard timeout length for our overall system so the standard timeout length between services to services and services to other systems, and we should only adjust the timeout length between two systems if there's an exception. But we should try and use a standard timeout length throughout our system. Remember, timeouts will not only help with downstream systems going down but it'll also help with network outages and network latencies. Network issues is also another type of failure and luckily for us timeouts can be used to cater both for network issues and for downstream systems going down. In that system, we should also continuously monitor our timeouts and log our timeouts. This will help us see that within our system, we have downstream systems that are not always responding or we have the occasional network issues. It makes all our issues transparent if we are monitoring our timeouts and logging them. So not only will monitoring timeout and logging timeout make our system health more transparent, but long term when it comes to problem solving, any specific behaviors of our system, logging timeouts will help us work out if specific behaviors are related to timeout issues. In the next two sections of the module, we will look at monitoring and logging.

**Observable: Centralized Monitoring**

Another key design principle for microservices architecture is that our microservices are observable. With the microservices architecture, our system will consist of multiple microservices and multiple instances of microservices. Therefore, it's important that we consider system health in a transparent way, and the one way of doing this is to have a centralized monitoring system with the new architecture. Something that produces monitoring data in real time. So as soon as there's an issue, you can start seeing this issue in your centralized monitoring system and you need to monitor the health of the host which runs your service in terms of CPU usage, memory usage, and disk usage. And you also need to monitor the service itself. You can do this by exposing metrics within your service. Things like response times, so the time it takes for a service to apply to your service. You also need to monitor things like timeouts. So if a service doesn't respond within a given amount of time, and therefore it has failed to save your service, you need to log this in your monitoring system. As well as monitoring the number of timeout errors, you should also monitor the number of exceptions or errors that happen across your microservices. Basically monitor anything that indicates the current health of the system.You can also expand your central monitoring system to include business data related metrics. For example, the number of orders you've currently taken or the average time from basket to checkout. We need a central monitoring tool that not only collects data but also aggregates that monitoring data so that at high level, we can see trends and history with the options to drill down into the detail if required. We also need our monitoring tool to be able to visualize trends so that we can see patterns and spot potential problems. We also need our monitoring tool to have the ability to compare data across service because our microservices will live on multiple servers and therefore the ability to compare data for different servers is key. We also need our monitoring tool to have the ability to trigger alerts if one of our specific measures exceeds a threshold. So for example on a given day, if our central monitoring tool detects the fact that we've had more than one timeout during the day, it might send an alert to all interested parties so that they can react and fix the issue. In the next section on the module, we will look at how centralized logging can be used to make our microservices architecture more observable.

**Observable: Centralized Logging**

Another way to make our microservices more observable is to have centralized logging. The difference between centralized logging and centralized monitoring is when we are monitoring, we are collecting numbers, and the counts, i.e. measures, when we are logging, we are actually recording information. Detailed information about events. And because they record detailed information about events, they tell a story. They allow us to see what's happened, why it happened, and how it happened. In a complex microservices architecture, we have multiple microservices and multiple instances of microservices. This kind of information is key for problem solving and specially when most transactions within this type of system are distributed transactions. We need a way, a centralized way, of seeing the whole story, the life story of a transaction, and its key to create these log entries when our microservices start up or when our microservices shut down. It's also a key to create these log entries when there are code path milestones. For example when you receive requests or when you make responses or when you make decisions within the code. Only key milestones which will help us problem solve later on. We also need to treat timeout errors, exceptions and errors as events that we need to log. The information we record regarding a specific event should also be structured and that structure should be consistent across our system. So for example our structure log entry might contain level information to state if that log entry is for information or it's information regarding an error or it's debug information or it's statistics that's been recorded as a log entry event. The structure log entry must also have a date and a time so we know when the event happened. We should also include the correlation ID in our structured logging so that we can trace distributed transactions across our logs.We should also include a host name within our structured log so that we know exactly where the log entry came from. We should also include the service name and the service instance so we know exactly which microservice made the log entry. And finally we should also include a message in our structure logging which is the key information which is associated with the event. So for example for an error, this might be the call stack or details about an exception. The key thing is we keep our structured logging format consistent. A consistent format will allow us to query the logging information. We can basically then search for specific patterns and specific issues using a centralized logging tool. Another key aspect of centralized logging within our microservices architecture is to make our distributed transactions more traceable. We briefly mentioned the idea of a correlation ID. A correlation ID is a unique ID which is assigned to every transaction. So when the transaction becomes distributed across services, we can follow that transaction across our services using logging information. The correlation ID is basically passed from service to service. Basically all services, that process, that specific transaction receive that correlation ID so that they can log in the events associated with that transaction to our centralized logs. In the next section of the module, we will be looking at the automation design principles for microservices.

**Automation: CI Tools**

Another key design principle for microservices architecture is automation. One aspect of automation in the microservices context is the use of tools to provide automatic testing and feedback for software changes. These tools are normally known as continuous integration tools. These continuous integration tools basically work with our source control system. They automatically test our software after we check in and change to the software into the source control system. They test our software by basically running unit test and integration test that we've written. Unit tests and integration tests are basically bits of code which is designed to test our production code. They basically test that a change or enhancement in the code doesn't break any of the existing requirements. So in order to fully benefit from using continuous integration tools, unit tests and integration tests should be implemented. So continuous integration tools basically allow us to ensure that our code compiled and that our tests passed and that our changed software integrates with anything else that might use our software. These tools also provide us quick feedback, so if a change to a microservice breaks the microservice itself or breaks anything else which might use the microservice, the continuous integration tool will give us quick feedback so that we can fix the issue quickly. In a microservices system with so many moving parts, this kind of information on the quality of integration is super useful. Quick feedback also causes an urgency to fix things quickly and issues don't pile up because automatic feedback is sent to the team and they can quickly address the issue. In fact, the team culture should be to stop any development until all issues reported by the continuous integration tool have been fixed. The continuous integration tool can also be used to build our software as part of the test and this build can be used by the testing to test a software and it can also be used for deployment. In the next section of the module, we'll look at how deployment tools can be used to take a build created by the continuous integration tools and have them deployed automatically.

**Automation: CD Tools**

Another aspect of automation when it comes to microservices is the use of automatic deployment tools and they're known as continuous deployment tools and they basically automate our software deployment. Under the microservices system where you might have multiple microservices and multiple instances on different servers on a number of different servers, this type of software is super useful. Initially, you have to take time out to configure the tool so the tool knows exactly what software to take from where and where to place the software and how to configure the software in the target location. However this configuration is only required once and the configuration is reused on future releases so when a new version of your software is available, you can use the same configuration to redeploy it again automatically. This type of deployment tool normally works with a continuous integration tool which creates the build that this tool will deploy and as long as all the continuous integration test passed, the new version of the software is deployable after check-in once the test have passed. As long as your production environment does not physically change, continuous deployment tools can be used at any time to reliably release your software into production. They can be configured to upgrade websites, web services, to upgrade databases. So when you are developing your microservices, ensure that continuous deployment tools are part of the plan. Continuous deployment tools can also give your company a competitive edge because you can get new versions of your software out to the market very quickly and in a reliable way. And because of reliable deployment, the customer will also have a better experience. So when designing your microservices, you need to keep in mind what the end deployment configuration for your microservices will be because we need to translate this configuration into a continuous deployment tool.

**Summary**

In this module, we learned the design approaches we can take in order to implement the design principles of the microservices architecture. In order for our microservices to have high cohesion, we must keep splitting our microservices until they have only one reason to change. We also learned in order to make our microservices autonomous, our system should be loosely coupled and that we should have a versioning strategy to ensure that our microservices are always backwards compatible. We also learned in order to make microservice more autonomous, i.e. independently changeable and deployable, it's best if the ownership of a microservice is given to a team so that they can ensure that the microservice is designed to be independently changeable and deployable. We also learned in order to make our microservices business domain centric, we must first define our microservices as coarse grain business domains. We then further subgroup them into functions and areas, i.e. further divide them into microservices to represent those functions and areas. We also learned in order to make our microservices more resilient, we must design them for known failures and we must design them in a way where they fail fast and recover fast. We also learned in order to make our microservices more observable so that the system health is more transparent, we should use tools for real time centralized monitoring and tools for centralized structured logging. We also learned the fact that in order to introduce automation into a microservices architecture in order to speed up testing and deployment, we should use continuous integration tools and continuous deployment tools. In the next module, we look at the technology that's available to make all these possible.

**Technology for Microservices**

**Introduction**

Hi this is Rag Dhiman from Pluralsight and welcome to the technology for microservices module. And in this module, we will look at technologies that help us enable a microservices architecture. We look at communication technologies for microservices and we look at hosting platforms for microservices, and we also look at technologies that make microservices observable. We then we want to looking at technologies that help with performance and then we conclude the module by looking at automation tools.

**Synchronous Communication**

In this section of the module, we will look at communication technologies that can be used for microservices. We will look at synchronous communication technologies and we will look at asynchronous communication technologies. So when using synchronous communication between our microservices, we basically makes requests and then wait for responses. And this kind of communication can happen between our client applications and our services or between service to service communication. Or between the service and an external system. So in our example here, service one makes a call to service two. And then service two receives that request as an instruction and then based on that instruction, service two carries out some work and then once that work's complete service two responds to service one. And as you can see this communication is synchronous because when service one makes call to service two, it has to wait for service two to complete its work before it will actually receive response. One of the technology types that can be used to make this kind of communication is called remote procedure call. Remote procedure call implementations basically allow you to create distributed client-server programs. And remote procedure call libraries normally hide the fact that you're communicating and making calls over a network. And when coding remote procedure calls, it's almost like coding a call to an internal method within the client itself. The remote procedure call libraries basically shield all the detail regarding the network protocols and the communication protocols. It appears if you're making a call to a local functional method, But in fact you're actually calling a remote method or function on a remote service. However, the problem with RPC is that it's sensitive to change. Any changes at the server end. So in our example, if we change service two will mean that service one will break changes like changing the method signatures at the server end will break all client applications. Another technology that can be used for request response at synchronous communication is HTTP. HTTP is a communication protocol that were used to using over the web. So when you request pages in a web browser, it's basically using HTTP communication in order to talk to the Web server and retrieve the webpage. And the same communication protocol can be used between microservices to make a request response synchronous communication calls. And because it works over the web, it's also firewall friendly. It's basically a communication protocol that network architectures are used to dealing with because it's used over the Internet. And therefore things like firewalls can be configured quite easily to let HTTP traffic through. You can even have RPC calls made using HTTP. Another type of communication protocol that can be used over HTTP to provide request response synchronous communication is REST And when using REST, the entities within our system can basically be accessed using endpoints URLs which basically maps to our entities within our system. In this example, I'm using an end point to retrieve an account record with an ID of 23 from service two by using this URL. Not only can I retrieve records, but I can also create retrieve update and delete records using HTTP verbs POST, PUT, GET, and DELETE which basically map to current operations. So in our current example, I can actually create an account using POST. And I can update an account using PUT. And I can retrieve an account using GET. And I can also delete an account using DELETE. REST also provides natural decoupling because a data returned is always in JSON or XML format which is normally different to the internal representation of that entity. REST is also an open communication protocol in that it doesn't dictate what technologies they used at the client end or at the server end. Both the natural decoupling and open communication protocol characteristics of REST make it ideal for microservices. Another key feature of REST that can be used is called HATEOS, this is basically a technique of including links to related resources and responses you receive. So for example, if I use a HTTP post call to create an account, the response I'll get from service two will include a link, a link to the newly created account so that I can use HTTP GET in order to retrieve the account using that link provided. Under the microservices architecture this is ideal because this means that communication will be less chatty and therefore even more decoupled. Because a communication we do make will result in responses that include extra information that will help us work with related entities. The main issues with synchronous communication are that both parties have to be available during the communication. Service one basically has to wait for service two to respond before service one can carry on doing other stuff. And in the microservices architecture, we have distributed transactions using many services. A slowdown in one service would slow the entire transaction down. It also makes the performance of our system and our transactions subject to the network quality because it might not be the service that's running slowly, it might be that the network is running slowly. Therefore responses from services arrive slowly. The other problem we have with synchronous communication is because the service talks directly to another service or a system, it must know the location of that service. However this problem can be resolved by using service registration and discovery patterns as well as other components on the network like load balances. In the next section of the module, we will look at asynchronous communication and how asynchronous communication can be used to avoid some of the problems associated with synchronous communication.

**Asynchronous Communication**

In this section of the module, we will look at how asynchronous communication can be used with microservices. So asynchronous communication in a microservices context basically means event based communication. So when a service needs another service to carry out a task, instead of connecting directly to that service the service basically creates an event and services that can carry that task out will automatically pick that event up and therefore in this type of event based asynchronous communication, there's no need for the client and the server to connect. It basically decouples the client and the service. And this type of communication normally uses a message queuing protocol, where the events created by the service are seen as messages where the service creating these messages is seen as a publisher and the service which carries out tasks in response to these messages is seen as a subscriber. And these messages are normally stored and buffered by a message broker until a subscriber picks these messages up. And the subscriber and the publisher only actually know of the message broker and therefore they are decoupled from each other .And therefore the subscriber and the publisher also don't need to know each other's location. Under the microservices architecture, this kind communication protocol will be perfect because we can spawn new versions of microservices and new instances of microservices, and there's no need for each of them to know each other's location because they all communicate using a message queuing protocol. You can also have multiple subscribers acting on the same message. For example in a microservices system, you might have a service which is just recently changed the type of data and it needs to inform all of the services to refresh the cash for that data. We also have many vendors that provide a message queuing protocols. We have Microsoft with Microsoft Message Queuing and we have RabbitMQ, and we also have ATOM, which uses HTTP to propagate events. The problem with event based asynchronous communication is  that it's complicated, especially when you're using distributed transactions and you're using events in order to complete a distributed transaction. Another problem is your system relies on a message broker. This is another additional component within the system that your system relies on and another potential point of failure. Visibility of transactions can also be a problem. So not only are your transactions distributed, but the transactions are not completing in asynchronous way you'll also need to learn new tools and techniques to manage the messaging queue. You will need to ensure the messaging queue is handled properly and that the performance of the system is not affected if you have any queuing issues. You'll find a microservices system you will end up using both synchronous communication and asynchronous communication. In the next section of the module, we'll look at how we can host our microservices.

**Virtualization**

In this section of the module, we will look at how we can host a microservices. We will look at how virtualization can be used to host our microservices and how containers can be used, and also how you can self host your own microservices. We will also look at how there's a need to register and discover microservices when hosting them. Okay so one way of hosting microservices is  to use virtualization. This is basically using virtual machines as a host in order to run the microservice. You no longer need to run new microservices on physical machines, instead you can just spin up a virtual machine and install your microservice, and you've got a self-contained machine that's running your microservice. Virtual machines are basically entire instances of operating systems running on hardware which is simulated in software. And therefore on one physical machine, you can have multiple virtual machines running which act and behave as if they are physical machines themselves. The beauty of virtual machines is that you can clone them. So once you have your one virtual machine setup and you have your microservices installed, if you need multiple instances of that microservice, you can basically clone multiple copies of that virtual machine. And that's why virtual machines have been the foundation of many cloud platforms. You can now in fact, subscribe to cloud platform services and these types of services are known as platform as a service. And these types of services can provide you an entire infrastructure through cloud services in order to run your microservices architecture. You could run your entire microservices architecture in the cloud with vendors like Microsoft who have an Azure platform. And vendors like Amazon who provide Amazon Web services. Without actually owning a physical server, you could run your entire architecture in these cloud services. And within these services, you could have multiple virtual machines running your complete architecture. And as the demand increases for your microservices, you could spin up extra virtual machines in order to run extra instances of your microservices. There are also platforms out there which allow you to create your own cloud, a cloud that you host yourself. A cloud which is basically made up of virtual machines in order to create an infrastructure, but it's a cloud that you host on your own physical machines. The only downside of virtual machines is that they could be more efficient. They can take a lot of time to set up and they can take some time to load. And they also can take quite a bit of resource. You have to remember these are entire instances of operating systems, running in software simulated hardware on top of another operating system. They do however have some unique features. You can for example take a snapshot of a virtual machine and then that later date, you can restore that virtual machine back to the time when the snapshot was taken. We've already mentioned you can clone virtual machines. You could clone virtual machines that already have microservices running in them and therefore quickly creating multiple instances of that microservices when the demand for that microservices increases. The other good thing about visualization is that the technology itself is now standardized and it's very mature. There are plenty of virtualization platforms available now. With real good tooling, it's a very common thing these days to have your entire enterprise software running in a virtualized environment. In the next section of the module, we'll look at another virtualization technology which is slightly different to virtual machines. We will look at the container technology.

**Containers**

Another hosting option for microservices is the use of containers. And the containers are another type of virtualization. However unlike virtual machines, they do not run an entire operating system within the container. They run the minimal amount required to run your service. They are a good way of isolating services from each other. So that each service is more stable and more secure. It's also a common practice to only run one service, one microservice within a container. We've already mentioned the fact that they are different to virtual machines in that they don't contain an entire operating system. Instead they have the minimal to run a microservice and therefore they use less resource in a virtual machine. This is resource in the form of CPU usage, memory usage and disk usage from the host machine, and therefore we can probably have many more containers running on a host machine than you can have virtual machines running on the host machine. Containers also tend to run faster than virtual machines. And they tend to boot up a lot faster than virtual machines and because containers are so lightweight and fast, it's quicker to create new instances to meet demand. And because of these characteristics, containers are also seen the future of hosted applications, and cloud platform support for them is growing. So support from the likes of Azure and Amazon Web services is growing. Currently containers are only Linux-based, but in the future Windows support is on the horizon. As a technology, they're also not quite yet established as virtual machine technology. The technology for containers isn't quite standardized yet. And there's limited features and tooling support. Infrastructure support is also in its infancy, and they're quite complex to set up when compared to virtual machines. Over time, container technology will standardize and in the future, they will play a key part in microservices architecture. Examples of container technology are Docker, Rocker, and Glassware. In the next section of the module, we will look at what it means to self host microservices.

**Self Hosting**

In this section of the module, we will look at self hosting as an option for implementing your microservices architecture. If you already have a good IT infrastructure and IT staff, self hosting might be your primary option. Cloud hosting services can make the implementation of a microservices architecture a lot simpler because you control the whole thing via a portal and there's no need for physical machines or specialized staff members in order to carry out specific tasks. But if you already have a good IT infrastructure, you might choose to implement your own cloud. You might implement your own virtualization platform or implement your own containers. If the implementation of these technologies is too complex to start off with, you can always start off by implementing your microservices architecture initially using physical machines where you have a single service on a server or multiple services on a server. However long-term, if you want to reap the full benefits of using a microservices architecture, you will end up using some kind of cloud technology so that you can at least scale up the system on demand. However, if you choose to implement the cloud platform yourself and have a local cloud instead of using an external cloud service, like Microsoft Azure, there are some challenges. You'll have to think about the long-term maintenance of a cloud platform and the need for technician's. Technicians that specialize in supporting cloud platform.And you'll also have to train your existing staff and you'll probably need extra space for extra service that you'll need to purchase in order to host the cloud platform. You still might find that scaling is not as immediate as buying an external service because in order to scale up your cloud services,you still have to buy the physical machines. In the next section of the module, we'll look at how when you do scale up your system by creating new instances of microservices how those microservices register themselves and how clients can discover them.

**Registration and Discovery**

Okay, now we'll look at registry and discovery of microservices within your architecture. So basically when you create new instances of a microservices because you're scaling up how is your system aware that that microservices now available to carry that work? So when we need to connect to a microservice, we need to know where that microservice is, what host the microservice is running on? And what port it's listening on. And the version of that microservice. One way of storing this information and making this data available to your system is by having your service registry database. So when you implement new microservices or when you create new instances of microservices to meet demand, on startup of these microservices register themselves on this database. You could even make your system clever enough so that when a microservice stops responding, the system the registers that instance of the microservice from the service registry database. This makes the system almost kind of self heal. New client requests will not connect to the microservice instance that's experiencing problems. Service registration and service deregistration are only really an issue when you're hosting the microservices yourself. Cloud services like, Amazon Web services or Microsoft Azure, can make service registration or service deregistration quite straightforward and automatic. So in your self hosted system how can these microservices register themselves in this servicer is for database? Option one is they register themselves on start up or you can have a third-party piece of software which detects new instances of microservices and registers them on the registry database. Okay, now that all your services are registered in the service registry database, how do the clients learn off all the locations? The options are the clients either directly connect to the service registry database in order to learn the location of specific instances of microservices. Or you use some kind of gateway which all the clients connect to and it's the gateway that retrieves the location from the service registry database. The process of a clients connecting to the service registry database in order to find the location of the specific microservice is known as client-side discovery. When a gateway or a load balancer or a similar type of component is instead used to connect to the service registry database on behalf of the client in order to find the location of the microservice, this process is known as server-side discovery.

**Monitoring Tech**

In this section of the module, we will look at monitoring technologies and logging technologies that make our microservices observable. In the previous module, we looked at why monitoring microservices is important. And the good news is there are a number of central monitoring tools out there. We have tools like Nagios, PRTG and we even have some components within our system like load balancers that can also carry out some monitoring. There are also a number of online services like New Relic, that can tap into your system and monitor it. The key features we require from our centralized monitoring tool is the ability to gather metrics across service and functionality to aggregate these data and to visualize these data. We also want a tool that provides automatic or minimal configuration. So that when we do create new instances of our microservices they automatically, or with minimal configuration, start being monitored. A tool that also allows the use of client libraries to send metrics will also be useful. So that we can create custom client libraries that send data that we want to monitor. A monitoring tool that can send test transactions and monitor those test transactions is also useful. A monitoring tool that can also send alerts when something being monitored is not as expected is also useful. In a microservices system, the network is a key component especially when most transactions are distributed transactions which make use of the network. So therefore a monitoring tool that can also monitor the network is super useful. In our microservices architecture, we need to ensure that monitoring is standardized across the whole system and this can be achieved by using a central tool and also by ensuring that all our hosts, our virtual machines and containers are preconfigured with everything that's required in order for them to be monitored. We also need to ensure that monitoring is real time because in a complex architecture like a microservices architecture, if there's a problem we need to know immediately so that we can react in a proactive manner. In the next section, we will look at what technologies can be used for logging, logging events that are happening within our system.

**Logging Tech**

Okay now, we will look at all the logging technology that's available to make our microservices observable. In the previous module, we looked at why logging was important for microservices. Mainly why the logging of specific events within our system was important for problem solving. The good news is there's a number of centralized tools which act as a portal for logging data. You basically push the logs, add these tools and then these tools basically store the logging data in a central database. So when a specific event happens within our system, we basically send the log to the centralized tools and then the centralized tool will store all the data as a log entry into a database. And these centralized tools normally also provide a front-end which allows you to look at this data, this logging data, and also to query and find patterns within this logging data. So for example using the correlation ID or the transaction ID, I can look at the journey of a transaction through our system, through all the microservices it's gone through as a distributed transaction by creating a query for that transaction or correlation ID within the centralized tool. And there are a number of centralized logging tools which provide all these functionalities. Within the microservice itself, you also need to implement a client logging library which basically pushes the log against our centralized tool an example of this is something called Serilog. Serilog can be configured to push and raise events against our centralized logging tool so that the log gets stored centrally. The desired features for both the client logging libraries and the centralized logging tool is is that they support structured logging. We need our logs across the system to have structure so that they can be queried and they contain a minimal amount of information that's useful. And because our system is distributed with distributed transactions, we need a logging to work across service. We need our centralized tool to have the ability to accept data from multiple service. And we also need our centralized logging tool to be automatic or minimal in terms of configuration. So that when we create new instances of our microservices because there's an increase in demand, our microservices can instantly start logging data to our centralized tool. And because we're using distributed transactions within a microservices architecture, it's key that we can log a correlation or a context ID and that we can query a correlation or a context ID within our data. Across the system, we need to ensure that we standardize our logging and we achieve this by using a central tool and also by providing a template for our client libraries that actually carry out the logging and pushing of the logs to the central tool. So that next time we create a new microservice, we can take this template for the client logging library and implement it and therefore we can ensure that the format of the log, the structure of the log, is the same standard format used across the system.

**Scaling**

In this section of the module, we look at how we can meet the performance requirements of our microservices by using scaling, caching and API gateways. Okay so in a microservices system when the performance requirements increase or the performance degrade, we normally address this by scaling the system up and this normally happens by increasing the number of instances of the microservices that's in demand or by adding extra resource to the host that's running a particular service. So this might be in the form of increasing the number of CPU cores that are available to a service. Or the amount of memory that's available to a service, or the amount of disk or bandwidth that's available to a service. And the way you increase the number of instances of a service or the amount of resource available to a service might be automated or on-demand. With the automated option, the system will automatically detect that the system requires extra resource or extra instances of a particular service and it will automatically increase the system in these areas. With the on-demand option, it might be as simple as logging onto a portal and increasing the number of instances for a service and adding extra resource. Or the on-demand option might be as manual as adding extra hardware or software in order to meet the demand. Cloud based services, like Microsoft Azure and Amazon Web services, are known to provide auto scaling options. So when the demand increases, the system scales up in an automated way. Automated and on-demand scaling up is only possible because of technologies like virtualization and containers. Increasing the number of physical host service in order to scale up is nowadays deemed as the slower option. Virtualization and containers are seen as a quick way of increasing the number of host, virtual hosts, that are running our services. Once we've scaled up our system by increasing the number of instances of a particular service, the next thing we need is a load balancer. And this can be in the form of an API gateway, something which basically takes in all the incoming request for a service and then shares the requests between the instances of that service, i.e. balances the load across the system to improve performance. Later on, we we’ll also look the other things an API Gateway can do as well as load-balancing. So when do you scale up? You need to basically scale up when there are performance issues or a new monitoring data shows that you will have performance issues or when new capacity planning shows that there's going to be increase in usage in the future. It might be the automated scaling up can be configured to cover most these scenarios. In the next section, we will look at how caching can be used to improve our system's performance.

**Caching**

Another way to improve the performance of a microservices architected system is to use caching, caching of data. Caching is a way of basically detecting that multiple calls are asking for the same thing and instead of honoring each request, you honor one request, retrieve the data and then use the same data to satisfy your all other requests. And therefore caching improves performance by reducing the client calls to a service, service calls to a database and service to service calls. The ideal place to implement caching within a microservices architected system is that the API Gateway level or at the proxy server level, this way the caching implementation is invisible to everything else, and at this level, not only will it reduce the number of calls to our services and our databases, but it will also reduce the amount of traffic within our network. Caching can also be done at the client-side level, for example single page applications that download most the data they need and work with that data until they need to make a call back to the system. You could also implement caching at service level. So when service A called service B, it caches a response from service B. And service A might be configured to only refresh the data once a day because it's static data. In terms of requirements, we want our caching system to be simple to set up and manage because in a complex microservices architected system, caching needs to be simple and easy to implement. Another concern to do with caching is data leaks. We do not want to accidentally present the wrong data to the wrong call. In the next section of the module, we will look at how API gateways can help with improving microservices performance.

**API Gateway**

In this section of the module, we will look at how API gateways can be used to improve microservices performance. So API gateways are basically the central entry point into our system for client applications, and therefore they can be used to improve performance by having load-balancing functionality and by having caching functionality. And because they are the central point of entry, they simplify the implementation of load-balancing and caching. The rest of the system can be oblivious to the fact that we have load-balancing and caching implemented, that's what makes API gateways so useful. So overall, API gateways help with creating our central point into our system and exposing our services to clients. And also they provide one interface to many services. In the background, we could increase the number of microservices or move our microservices around in terms of location. But the client application won't realize, because their main entry point is the API Gateway and it hides all the detail in terms of the location of the microservices within the system. So it helps with dynamic location of services. API gateways can also be configured to route specific calls to a specific instance of a service. So for example, all calls from European clients might be directed to instances of microservices which are located in Europe. They can also be configured to look up the location of microservices in the service registry database in order to help with load-balancing. They can also be used to provide security for the overall system by providing authentication and authorization. They might have a dedicated service in the background which gives them all the data they need in terms of security. So having security built into the API Gateway is a central way of implementing security. But you might choose to have security and authorization both at the API Gateway level and at individual service-level. So as you can see, not only do API gateways help with performance but they can help simplify overall microservices architecture.

**Automation Tools**

In this section of the module, we're going to look at the automation tools that are available for our microservices architecture in the form of continuous integration tools and continuous deployment tools. We've seen in the previous module why we need continuous integration tools for our microservices architecture. Good news is there many continuous integration tools. From team foundation server, to TeamCity and many more. You just need to Google for continuous integration tools and you'll see how many there are. When making your selection, these are some of the desired features that you should look for. The tool should be cross-platform. So if you build microservices using Windows technologies or Java technologies, it has the relevant builders in order to produce a build. You also need to ensure that it integrates with your chosen source control system. So that when you check a code change in, it can detect this and trigger off a build. It also must have the functionality to send notifications. So for example if your integration tests fail it must have the ability to alert and send notifications. It's also nice to have a continuous integration tool that has some integration with the development ID. Another key thing is, when you use continuous integration tools you map microservices to individual CI build.This way a code change for a specific microservice only triggers the build for that specific service and this way if the build fails, you know the feedback is just received for a specific service and you know where the issue lies. The feedback is also received quite quickly because it's just one microservice that is build, the test and the build run a lot faster. As well as having a separate CI build for microservice is also worth having a separate code repository for a service. This way two microservices are not accidentally changed at the same time. Another advantage of mapping a microservice to a CI build is, we can place the microservice in a specific place which helps with continuous deployment because a microservice will also have its own database and a microservice maps to specific CI build. We can configure the CI build to autotest the database changes. And this way, the CI build can also already boost the microservice build and the database upgrade at the same time. If you choose to have one CI build for all you microservices, you will lose all those advantages we've just described. So you should avoid having one CI build for all your services. Instead each microservices should have its own CI build. And remember microservices should be independently changeable and deployable and having a specific CI build for each microservice helps this. And like there are many continuous integration tools, there are also many continuous deployment tools.And like with the continuous integration tools we need to ensure that our continuous deployment tool is cross-platform, so that it can help us deploy our microservices to a Windows server or for example a Linux server. Other design features for our continuous deployment tools are the option to have a central control panel. Having a distributed architecture like a microservices architecture can make the deployment of software quite complex.Having a central control panel just makes things a little bit more easier. We need a clear way of seeing what production environments we have and what test environments we have that we can deploy our software to and it must be a straightforward task to add extra deployment targets. So these are deployment targets where our software is delivered. So these might be virtual machines, containers or actual physical machines. And the task of deployment is always not straightforward, it's not always as simple as just copying files across, and therefore sometimes we need scripting to do additional add on tasks. It's also useful if our continuous deployment tool has support for build statuses. So for build failed during continuous integration due to failing tests, it knows how to highlight this and how not to include it in a production release list. And therefore it's ideal if our continuous deployment tool has some kind of integration with our continuous integration tool. So not only to see build statuses but also to know where to fetch the build from. We must also ensure that our continuous deployment tool can support multiple environments. So for example a production environment and the test environment. And we must also ensure that our continuous deployment tool can release to cloud-based hosted systems. For example Microsoft Azure and Amazon Web services. When it comes to microservices architecture, it's key that our continuous deployment tool is as flexible as it can be because with microservices, we could end up using any type of technology platform.

**Summary**

In this module, we've looked at the communication technologies that can be used with microservices. We looked at synchronous communication technologies and we've also looked at asynchronous communication technologies. And we concluded that in your microservices architecture, you'll probably end up using a combination of both. We also look technologies that we could use to host our microservices from virtual machines to containers, to self hosting. And we also looked at how we could have our microservices register themselves and be discovered. We concluded with the fact that in a microservices architecture, you're bound to create multiple instances of microservice as the demand increases. And therefore you need a way of registering these instances so that they can be discovered. And that in a self hosted system, registry and discovery is your responsibility. We also looked at the desired features for monitoring technology and logging technology that would make our microservices more observable. We also looked at the scaling and caching and API Gateway technology that's available to help with microservices performance. We also looked at the desired features for our automation tools that make our microservices architecture easy to test and deploy. In the next module, we will look at how we can move forward with the microservices architecture when in the Brownfield situation and when in a Greenfield situation.

**Moving Forward with Microservices**

**Introduction**

Hi, this is Rag Dhiman from Pluralsight, and welcome to the Moving Forwards in Microservices module. And in this module we will look at how you can move forwards with the microservices architecture when you are in a Brownfield situation, meaning that there's an existing system with an existing architecture that needs to be migrated to a microservices architecture. We will also look at how you can move forward with the microservices architecture in a Greenfield situation, in a situation where you're creating the system from scratch. We will then conclude the module by looking at microservices provisos, basically everything you need to understand and accept before you can move forwards with the microservices architecture.

**Brownfield Microservices Approach**

Okay we'll start off by looking at how we can move forward to the microservices architecture in a brownfield situation. We will first look at the initial approach we can take in order to ready our system for the migration over to a microservices architecture. We will then look at the actual migration of how we can migrate a monolithic system to a microservices architectured system. A monolithic system is also likely to have one large database, so we will also look at database migration, database migration from a monolithic database to a number of databases to support our microservices. We will then go on to look at the effect microservices architecture can have on transactions within our system and reporting within our system. Okay so in a brownfield situation, our existing system is likely to be a monolithic system that's organically grown, and it might seem too large to split into microservices. It's also likely to have one large business layer or domain layer where most of the business logic for the system exists. And overall the system probably lacks most of the microservices design principles we've covered in this course. So the question is, how can we start readying this system for a migration from the monolithic system to a microservices architectured system. The first key step is to start analyzing the code within the system. We then start identifying seams within the system. And this is done by identifying separation within the code which reflects the domains within our business. And within the code it might be clear to see specific classes, modules, and functions which relate to specific parts of the organization. And the separation might not be just related to domains within the organization, but functions within the actual organization. So for example you might see loads of code related to the accounts department within your system, but you might also see loads of code related to a specific function within the business. For example calculating postage cost. And the seams of both scenarios are valid. And by grouping that code in business domains and business functions, we're basically identifying bounded contexts within our system, an idea which is taken from domain driven design about grouping related code together. You'll also might find that seams within the system are not that clear at all. They're quite fuzzy because of overlapping code. So for example you might find a code, which is related to the stock department, overlaps significantly with code related to the orders department. So this is where we need to start readying that code by refactoring the code so that it's better split into bounded contexts. We can start off by creating a module for each bounded context, and then we can start moving our code around incrementally. So for example this might involve moving classes and functions which actually should be in the stock department related code away from the orders department code and moving into a module which represents the stock department. So for example in a .Net project, we might have an accounts project, an orders project, a stock project, each creating code libraries which are part of our overall solution. We then move the overlapping code by moving the overlapping functions and classes which are in the wrong place to the right module or library. Remember you can tidy up each section per release. You need to take your time because we need to ensure that we don't break the existing system. And to validate your code refactoring, it's important you run unit tests and integration tests to validate your change. And if you're not using unit test or integration test, it might be worth investing in unit test and integration test technology. The other key thing is after every release, we review the code again and refactor again and make our bounded contexts even more clearer. It's important to have clear seams and clear bounded contexts because in the future, these are our microservice boundaries. And in the next section of the module, we will start looking at how we can migrate these bounded contexts into actual microservices.

**Brownfield Microservices Migration**

In the previous section of the module you saw how we ready our code, how we reorganize our code so that it's ready for migration over into microservices. In this section of the module you will see the steps we can take in order to migrate that code into actual microservices. Okay so when our code has been reorganized into bounded contexts, code related to a specific business domain or function is in one place, and there are clear boundaries with clear interfaces between each. So for example if there's a function within the orders section of the code that needs the accounts section of the code to do something, there are clear functions and methods defined that the order section is to call in order for the accounts section of the code to do something. There are clear interfaces between the two bounded contexts, and it's these bounded contexts that we can work into a microservice. And we might initially start off by just converting one into a microservice. This is so that we can get comfortable with the microservices architecture and the technologies it involves. For example message broker's central monitoring tools, central logging tools, continuous integration and deployment tools, as well as getting comfortable with other things like distributed transactions which actually complete all the network, and things like separate data stores for specific parts of the system. You could even make the old functionality and the new microservices functionality switchable, so if there are any issues, you can revert back to the old functionality. However the risk with this is you might end up with maintaining two versions of the same code. Okay so how do you prioritize what to split out first into a microservice? You might do this by risk. So the part of the system which has least impact, if there's an issue, you might choose to split that out into a microservice first. Or you might do this by technology. If there's a specific part of the system, that for example could do with the different type of technology in order to improve performance or provide extra features, you might choose to split that part of the system out first. Or you might do this by dependencies, converting the bounded contextswith the least amount of dependencies into a microservice first. So for example if the promotions bounded context is only ever called by the orders bounded context, it might be safer to convert the promotion bounded context into microservice first. The key thing to remember is you're going to take an incremental approach so that we can learn how to live with the microservices architecture. And because it's an incremental approach, we need to ensure the new microservices we create, they integrate with our existing monolithic system, and when the two types of architecture run alongside each other, we need to monitor both for impact. For example how is the monolithic system coping with having some of the transactions done over the network? You might choose to closely monitor operations that specifically talk to microservices. And you might review each time and improve the infrastructure in order to support the new distributed transactions which are done by the microservices. And by taking this incremental approach to converting the monolithic, eventually it will be converted to a complete microservices architectured system. In the next section of the module, we will look at how we can split our monolithic database into databases for each microservice.

**Brownfield Microservices Database Migration**

In this section of the module we will look at how we can split our monolithic database across our microservices so that each microservice has its own database. In the previous module we've already mentioned the fact that we want to avoid shared databases. We want our microservices to be independently changeable and independently deployable, and a shared database does not allow this. Okay so the question is how do we split our databases? The good news is it's the same approach as we took to split the code into bounded contexts. We basically identified seams in the databases, seams which relate to seams in the code. So for example we might have tables which relate to the accounts department functionality, and therefore that is a clear seam and we can split those tables out into a separate database. And in order to support the existing application, the monolithic application, you might have a data layer which can access multiple databases. Okay so what do we do when we have a table which links across seams? For example you might have a promotion which links to an order. Okay this is where the service, so for example our promotions service must provide methods which allow the orders service to fetch a specific promotions data from the promotions service. This kind of approach will allow microservices to fetch any data that's related to one of its entities from any other microservice. And like we refactored our code base into multiple code bases, we are basically doing the same with our database, we're refactoring the database into multiple databases. Even though we've refactored our database into multiple databases, we still have to worry about data referential integrity. So for example in our accounts microservice,if I delete the account of a customer, I also might want to delete related orders for that customer, orders that exist in the orders microservice, and I would do this by calling the orders microservice and calling a method which basically instructs the orders microservice to delete specific ordersrelated to a specific account ID. So data referential integrity can be maintained even though the data is split across microservices by having the microservices talk to each other and instruct each other. Okay so what if you have static data tables that's required by all microservices? The best thing to do with this kind of static data is is to place that static data within a configuration file and make that configuration file available to each microservice. The other option for static data tables is is to have a specific microservice just for them so other microservices can just basically call that service in order to fetch that static data. The same is valid for shared data, data that read and written to by multiple microservices, promote this data to its own microservice so that the other microservices can interact with it in order to read and write that data. In the next section of the module, we will look at how we need to think differently about transactions when migrating from a monolithic system to a microservices architectured system.

**Brownfield Microservices Transactions**

In this section of the module we'll look at how we need to view and handle transactions within our system differently when moving from a monolithic system to a microservices architectured system. Firstly transactions are useful to ensure data integrity. They basically allow us to create or update several records as part of one transaction. And if one of those updates or creates fails, we can roll the entire transaction back. The monolithic system transactions are fairly straightforward. You have one process which is updating and creating the records which are part of the transaction, and therefore the same process can either commit the transaction or roll the transaction back if there are any issues. However in a microservices system,transactions are complex because you now have several processes by several microservices which actually complete the one transaction. The transaction is now a distributed transaction which spans multiple microservices, and therefore it's complex to observe and to problem solve, and complex to roll back. In our example here we have a transaction of placing an order, and as you can see it starts off within the orders microservice,and then we create or update a record using the promotions microservice, and then we create and update a record using the accounts microservice,and then the transaction finishes off by creating or updating a record within the product's microservice. So if one of these microservices fails to update or create a record which is part of our distributed transaction, we will need to roll back the entire transaction. So what are the options for failed transactions? One option is we can try again later so the part of the transaction which has failed, we place back onto the queue for possibly another service to pick up. The transaction will then eventually complete. This however relies on the other instance of the microservice not failing the same parts of the transaction. Another option is we abort the entire transaction, we detect the fact that our transaction has failed, and then we issue an undo transaction. An undo transaction basically tells all other microservices to undo any creates or updates which they carried out as part of a specific transaction. The problem with this is who issues the undo transaction? And what if the undo transaction itself fails? One way of resolving our transaction issues is is to use a transaction manager software. Some transaction manager software use a method called two phase commit. And in this method, all the participating microservices first tell the transaction manager if they are okay to commit their part of the transaction, and if they all are, the transaction manager tells all participating microservices to commit the transaction. And if any of the microservices participating in the distributed transaction does not respond or responds with a no to committing, then the transaction manager tells all participating microservices to roll back the transaction. The problem with the use of a transaction manager is that we become heavily reliant on the transaction manager. And it also delays the processing of our transactions. And it potentially also might become a bottleneck within the system. The transaction manager can also be quite complex to implement. Another thing we need to worry about is is the fact when we run our new microservices alongside our monolithic system, our new microservice needs to tell the monolithic system that their part of the transaction is complete, and this can be achieved by placing a message in the message queue for the monolithic system. In the next section of the module, we will look at the effects of migrating to a microservices architecture on reporting within our system.

**Brownfield Microservices Reporting**

In this section of the module we're going to look at how we can support reporting within our system when we have a microservices architectured system. Microservices do complicate reporting. In a monolithic system you might have a few databases that you need to report from, but in a microservices architectured system, each microservice has its own database. The data you want to report on is basically split across multiple microservices. And by default there's no central database that you can extract the reporting data from. And there will be an inevitable need to join data across databases in order to provide the information your organization needs. And we will have to accept the fact that reporting within a microservices architectured system will be slower, and it will be complicated to develop an actual report. However there are some solutions which make things easier. We could for example have a dedicated reporting service which basically calls other microservices to collect the data and to consolidate the data so that it can be presented in a report. However this might not be adequate if we're reporting on large volumes of data, or if we need our reporting to be real time. Another option is that we have a data pump, a data pump which basically pumps the data to a central databasethat can be used for reporting purposes. How you implement the data dump will depend on your requirements. For example do you need the data in real time? If real time data is not a requirement, what you could do is you could have a consolidation environment where all the data from all the microservices is consolidated into a central database. This can be done for example using an overnightly job. So in order for a microservicesarchitecture to be successful, we need to ensure we have some kind of a reporting strategy.

**Greenfield Microservices**

In this section of the module we will look at how we can approach a microservices architecture in a Greenfield situation where basically we're creating a new system from scratch. Okay so a Greenfield situation will mostly likely involve a new project. It might be a new aspect of our existing system, or it might be a completely new system altogether. And as with all new systems, the requirements might evolve during the design phase, and there might be no clear indicator indicating what microservices are actually required for the overall solution. And this will mainly happen because this is a new system, and the business domain is not fully understood, and we have to get the domain experts involved in order to understand it. We will also find as we discuss and design with the domain experts the system boundaries will evolve. The initial boundaries you set will change, and that's why it's important in a Greenfield situation to understand and collect the requirement, and not focus too much on what microservices are required to address the overall problem. How well the design and discussions go will also depend on the team's experience. Is this the team's first microservices architectured system or have they got experience with microservices? The other key thing to understand is  the new system going to integrate with an existing system? And is the current system a monolithic system? Or do you already have an established microservices architectured system? The key thing to remember is we need to push for change. We need to change in order to apply our microservices principles if there's currently no microservices architectured system, The irony however is that because the requirements are evolving, and it's not quite clear to see what microservices are required, it's best to start off with a monolithic design. So we start off with a high level design and we allow the seams within our system to evolve. We define each high level area as a module. And as the interaction between these areas and seams becomes clearer, the boundaries within our system start to become clearer as well. And as our understanding of these modules, these areas becomes clearer, we refine and refactor our design. And if we need to further split our module, we split it further. Eventually these modules should represent business domains and business functions at a very finite granular level. And it's these modules that later on go on to become our microservices. If you find there's loads of code that needs to be shared between these modules, consider promoting those code libraries to a service. The key thing to remember is at each stage we review our microservice principles, and that we also keep the requirements in mind. We actually deliver what's required in terms of customer needs and demand.

**Microservices Provisos**

In this section of the module we'll look at microservices provisos, basically what you need to accept before you implement a microservices architecture. If it's your first microservices system, you need to accept the fact that there will be some initial expense. The development times will be longer. And there will be a need for training for tools and new skills to cope with a new microservices architectured system. We also need to accept the fact that it will take our team time to skill up to handle a distributed system. Only with experience will they learn how to handle distributed transactions and how to problem solve distributed transactions, as well as handling reporting in an environment where the data is not kept in one place. And because of the distributed nature of the system, you might also need to invest in extra testing resource in order to test for latency, performance, and resilience. This is important because remember our software is no longer a single process, instead it's a number of components which are distributed over a system, and they work together over the network. And therefore it's important to have extra resource who have the skills to test for network related issues, as well as investing in testing the network infrastructure. You might find you have to improve the network infrastructure in terms of security, performance, and reliance. And again this is important because our software heavily relies on the network in order to function. The components need to interact with each other in a timely manner in order for our software to function, therefore we need to ensure that our infrastructure is adequate with the complex distributed piece of software which is what our microservices architecture is. We need to ensure that we understand that there will be an overhead to manage these microservices. We need to constantly monitor our system for performance issues to avoid any problems. We also need to accept the fact that investing in cloud technologies will make the management of microservices a lot easier.We also need to accept the fact that applying the microservices design principles overall for an organization is a culture change.

**Summary**

In this module we looked at how we can architect a microservices system when in a Brownfield situation i.e. where there's an existing system that needs to be converted. We looked at how we need to reorganize our existing code base into bounded contexts which are then later on converted into microservices. We also looked at how we can create a microservices system in a Greenfield situation where basically we're creating a new system from scratch. We concluded with the fact that it's best to start off with a monolithic design which you refine and refactor into a microservices design. We concluded the module by looking at microservices provisos, basically everything we need to accept before we introduce a microservices architectured system. We concluded with the fact that not only is microservices a technical change for a company but it's also a huge cautious change for a company.