**Deep Dive on cloud Native Apps and 15-Factor methodology**

These are the standards cloud native apps and 12-factor, 15-factor these are standards that any microservice developer as to know. Because following these standards only we are going to build many concepts inside the microservice.

What is cloud native application? If anyone non-technical person, ask we will give this answer

* We might be hearing this jargon or this buzzword a lot inside our project or inside some blogs podcasts or YouTube videos. So, lets try to understand what is this cloud native applications and how they are related to the microservices.
* **The layman definition** – Cloud-native applications are software applications designed specifically to leverage cloud computing principles and take full advantage of clouds-native technologies and services. These applications are built and optimized to run in cloud environments, utilizing the cloud advantages like scalability, elasticity, and flexibility offered by the cloud.
* In simple words, cloud native applications are built for cloud environments so that the organizations can take complete advantage of cloud provider services and technologies.

**Now let's try to understand the official definition of cloud native applications from the Cloud Native in technical manner.**

* So, if you see here, cloud native technologies empower organizations to build and run scalable applications in modern dynamic environments such as public, private and hybrid clouds. So, the cloud environment can be anything, it can be your own data centre, it can be GCP, azure, or it can be a hybrid combination of private cloud and public cloud. Regardless of what type of cloud computing you're trying to use; our cloud native applications will work.
* So how this is possible is when we are trying to build cloud native applications, we will try to leverage technologies like containers, service meshes, microservices, immutable infrastructure and declarative APIs. So, all these technologies make our applications to run on any cloud so we can’t get vendor lock in with any cloud environment.
* On top of these cloud advantages. These techniques also enable developing loosely coupled systems that are resilient, manageable and observable. Resilient means they can withstand any failures manageable means we already know they are easy to manage, and observable means we'll get to know everything about our applications, how it is working, are there any issues.
* So, with all these new techniques, along with the robust automation, these cloud native applications, they allow engineers to develop high impact changes frequently and predictably with minimal toil. So, this will give freedom to the organizations and the developers if they want to make a small change or if they want to make a small enhancement, they can do that very quickly without any defects or regression issues being introduced. Because they are all loosely coupled, and it is very easy to test them to develop new enhancements.

**Important characteristics of cloud-native applications:**

1. The very first main character of Cloud Native application is **microservice**. When we are building microservice based applications, which are like loosely coupled and smaller in nature, then that gives a flexibility to develop them parallelly and deploy and scale independently.

So, cloud native application is a broader topic and under that topic **microservices** is one of the important features.

1. After building microservices like separating our business logic, we will obviously **containerize our applications** with the help of **Docker** or any other containerization software. These are typically packaged and deployed using Docker containers.

These containers provide a lightweight and consistent environment for running applications and making them highly portable across different cloud platforms and infrastructure.

With the help of these containers, only the code will work very similarly regardless of which cloud environment we are trying to deploy. We take this container, we try to deploy inside our local system, and we deploy inside the AWS, GCP, Azure Cloud in all the places it is going to work in very similar manner.

Whereas this is not the scenario with a monolithic application. Our monolithic application won't give such kind of flexibility. we need to put a lot of efforts to bring that consistency across all the cloud platforms.

1. The next character of cloud native applications is it provides **scalability and elasticity**. Since we are building our applications based upon microservices and with the help of containers, they can be easily scaled horizontally allowing them to handle any kind of traffic that comes towards our applications.  
   So, adding more instances of services is going to be super easy and this can be achieved automatically with the help of container orchestration platforms like **Kubernetes**.
2. Moving on to the next principle, these cloud native applications, they **follow DevOps practices**, by embracing all these DevOps principles. They promote a collaboration culture between the development and operations teams. Whenever these cloud native applications are being built, they will not be any blame game between the developers and operations team when they are building cloud native application.

Because they are following these DevOps practices and with the help of these DevOps practices, they will incorporate continuous integration, continuous delivery and automated deployment pipelines to streamline the software development and deployment process.  
So, these cloud native applications, they will give complete flexibility to the organizations, whether they want to do continuous integration only or whether they want to go with the continuous delivery or if they want to go with the continuous deployment as well.

1. The next character of native applications is, they are **resilient and fault tolerance in nature.** Whatever applications that we are going to develop with the help of cloud native principles, we can withstand any kind of failures which will make them resilient, and they will utilize techniques such as distributed architecture, load balancing and automated fail recovery to ensure high availability and fault tolerance.  
   One example that is, think like we have a microservice by following the cloud native application principles, we will deploy this microservice in multiple locations. Even if one of the locations has a downtime due to some power outage or due to some internet issue, the microservice will continue to work from the other location that you have deployed. And at the same time, for some reason, if one of our microservice instance is not working platforms like Kubernetes, they can automatically shut down that microservice instance and bring up a new instance.  
   This way we are ensuring the failure recovery automatically and bringing fault tolerance inside your applications.
2. The next character of cloud Native applications, which is they **utilize cloud native services** to a great extent. Like cloud native applications means they are developed to largely leverage the cloud environments.  
   When an organization or an application uses these cloud native services provided, they don't have to focus on infrastructure because all these services will be monitored and maintained by the cloud platform provider itself. With that, the developers and the organizations they can simply focus more on the application logic and enhancing the business logic with a very less focus on the infrastructure components.

Like we can see these are all the important characters of cloud native applications. Whenever we see an application is following all these characters, then we can simply assume that is a cloud native application.

**Difference between cloud-native and traditional applications:**

1. The very first difference of cloud native application and traditional applications is, cloud-native applications will have **predictable behaviour**, whereas with traditional applications, the behaviour is going to be **unpredictable**.  
   For example, inside microservice environment, if there is an issue, we can easily track that were the issue is because all your business logic is loosely coupled and separated into multiple microservices, so, you can easily predict where the exception happens.  
   Whereas inside monolithic application, since all our business logic clubbed together, we never know where the exception happens until developer put a lot of efforts by debugging all the lines of code that we have inside our application.
2. The next difference that we have here is cloud native applications they are not **OS dependent**. They will abstract the operating system, whereas traditional enterprise applications, they are dependent on **operating system**.  
   This is because inside cloud native applications we are going to adopt docker containers and Docker images where they will abstract the operating system, and they will work in any operating system.
3. The next difference that we have here is, cloud native applications they are **rightly sized**, and they are following the right capacity, and they work in an independent manner.  
   Whereas with traditional applications they are **oversized capacity** because we have all our business logic inside a single application or inside a single codebase. That's why most of the times traditional enterprise applications, they are dependent on each other.
4. Moving on to the next difference cloud native applications they **support continuous delivery** with the help of DevOps principles and automation, whereas traditional enterprise applications, they follow **waterfall development**. They won't support Agile style of working inside any organization.
5. Then the last difference that we have here is Cloud Native Application supports **rapid recovery and automated scalability**. Inside cloud native applications if one of the microservice instance is not working properly Kubernetes, they can automatically create a new instance and try to recover automatically and scale our applications automatically based upon the incoming traffic.  
   Whereas with traditional applications, since they are not going to use Docker containers, obviously they cannot rely on the platforms like Kubernetes with that reasons with the traditional enterprise applications, **the recovery is going to be super slow** and at the same time there won't be any automated style of recovery and scalability.

From these differences It is very clear that Cloud Native application is a clear winner compared to the traditional enterprise applications.

There are good guidelines and principles that we can follow while building these cloud native applications or microservices. So, let's try to understand what those principles and guidelines are.

**Development principles of cloud native applications: 12 Factors and Beyond**: <https://12factor.net/>

Many of us who is trying to build cloud native applications or microservices will have a basic question, which is how to get succeeded in building a better cloud native application or a better microservices applications. Are there any guiding principles that can be considered for the same?

To answer this question, yes, we do have standards, so let's try to understand them.

1. Back in 2012, the engineering team at Heroku Cloud Platform, they introduced 12 factor methodology. So, these are the 12 different development principles that are aimed at guiding the developers in designing and developing cloud native applications.
2. These 12 different principles under the 12-factor methodology are the result of the Heroku team expertise from many, years building these cloud native applications. When we follow these 12-factor methodology, our applications will have advantages like ready for cloud platform deployment regardless of whatever cloud platform that we choose.
3. We can seamlessly deploy our cloud native applications. And at the core of this cloud native applications, they are going to support scalability and elasticity. And very similarly, these cloud native systems that we are going to develop with the help of 12 factor methodology, they allow system portability, which means our applications can run across different systems and environments without any issues. And at last, these applications will support continuous deployment and agility. So, in the initial days, everyone is recommended to follow these 12-factor methodology.
4. But later, after few years since the technology is getting evolved day by day, a person with the name Kevin Hoffman, he expanded these 12-factor methodology by adding three more principles.
5. And in his book with the name **Beyond the 12 Factory app**, he revised all these 12-factor methodology. And on top of that, he also introduced three more development principles.
6. And with that, a new 15 factor methodology came into picture. And this is the latest one right now everyone is referring to.

**15-Factor Methodology:**

So, these are the 15 methodologies are the principles or guidelines that any developer or organization should follow while building cloud native applications.

Without following these standards, they can't tell their applications as cloud native applications because they will end up with some problems since they are not following these standards. These standards are built based upon the development experience of cloud native applications.

Kevin Hoffman, he wrote a book with the name Beyond the Twelve-Factor App. And inside this book he added three more principles or guidelines. And with that this became 15 factor methodology.

In 2012, Early cloud pioneer Heroku developed the 12 Factor app, a set of rules and guidelines for helping organizations for building cloud native applications. It served as an excellent starting point, but as a technology changes day by day, some areas need revisiting to accommodate the current best practices. This practical book expands on the original guidelines to help us to build applications that not only function in the cloud but also thrive.

1. One code base, one application
2. API first
3. Dependency management
4. Design, build, release, run
5. Configuration, credentials and code
6. Log
7. Disposability
8. Backing services
9. Environment parity
10. Administrative processes
11. Port binding
12. Stateless processes
13. Concurrency
14. Telemetry
15. Authentication and authorization

Like we can see here, the very first principle are guideline that we have from the 15-factor methodology is:

1. **One code base, one application:** One code base for one application. So, this is going to be super simple. As per this guideline, we need to make sure there is one to one correspondence between an application and its code base, which means each application are a microservice should have its own dedicated code base.

Inside this course we build three different microservice. Off course, we are trying to upload all the code related to this microservice inside a single GitHub repo, so that it is convenient to access the same.

But in real world we need to make sure every microservice is having its own GitHub repo or its own code base inside the versioning system. If there is any code which is common for all the microservice, then such common code should be manage separately.

As a library, we can also deploy all these common code as a separate standalone service which will serve as a backing service for the other applications. Since it is possible to track each code base in its own code repository, this is going to provide a lot of flexibility and brings organization of your code in a cleaner manner.

Once we have separate code base for all our microservice, please make sure, regardless of how many deployments we are doing in different, different environment, we should always make sure we are building and converting our code base into our Docker container or a Docker package or any other artifact. Only once.

As per these guidelines, it is unnecessary to rebuild the code base for each environment deployment that we are going to make.

In this methodology, a deployment refers to an operational instance of the application. Multiple deployments can exist across different environments, all leveraging the same application artifact. It is unnecessary to rebuild the codebase for each environment-specific development. Instead, any factor that vary between deployments, such as configuration settings, should be maintained externally from the application codebase.

1. **API first:** Eeverything we are going to develop as REST APIs. So, even this cloud native guideline, which is API first, it is also recommended to always adopt and have a mindset of API first, which means right from design of your cloud native applications.  
   You should always think like to write as much logic as possible with the help of APIs. Only if you design and develop your business logic with the help of APIs.  
   This will give flexibility that most of our business logic can be invoked by the other APIs or by the other microservices as a backing service, and this also provide an advantage like different, different teams. They can work on different, different APIs.  
   Apart from these advantages, when we follow these API first approach, it is also going to give flexibility like inside our deployment pipeline we can write some testable integrations with other systems, and if the integration and testing is working, then only we can do the deployment.   
   On top of that, since we are going to separate all your services with the help of APIs, we can always do some internal modifications behind our API implementation without impacting other applications or teams that rely on our APIs.

That's why, while we are developing cloud native services or applications, always have a mindset that API first.

1. **Dependency Management:** As per these guidelines, it is important or crucial to explicitly declare all our dependencies of an application inside a single manifest file and post that we should also ensure that these dependencies are accessible to a dependency manager which can download all of them from a central repository.

In case of java applications like we know we have two robust tools like Maven and Gradle that follow the same standard. We as a developer, we only provide our dependencies inside the pom.xml or build.gradle and behind the scenes my build manager like Maven and Gradle.  
It is going to pull all the dependencies from a centralized repository. Once this download is completed during the packaging of our application as a Docker image or a Docker container, all the dependency libraries are going to package as a single artifact.  
By following this approach, we maintain a clear and controlled dependency management process for our application. If we don't follow these pom.xml or build.gradle and follow the very old approach like downloading all the dependencies manually and adding them to the classpath of our web application, which we used to do before Maven and Gradle. That is going to make our process complex because we are not going to have a single application. We are going to have 100 different microservice. And doing all that manual setup every time inside a microservice is going to make the process complex. That's why always follow this dependency management principle.

1. **Design, build, release, run:** The recommendation is our code base must progress from design to production by following the following stages.

The very first stage is design stage. Inside this design stage, we need to determine all the required technologies and dependencies and tools for a specific application.

If we are building a microservice during the design stage itself, we need to determine all the technologies, dependencies and tools needed for our microservice.

So here inside these design stage, it also includes the development and unit testing. So, once the design, development and testing are completed next, we need to move on to the build stage.

Inside this build stage, we need to compile and package the code base with the required dependencies. By creating an immutable artifact, every build artifact should have its own unique identification number, just like how we maintain versions like 1.02.03.0. Very similarly, each of our build stage should have its own unique identification and at the same time, whatever code base package we have generated, it should be immutable.

They should not be a scenario where someone is trying to change the content of the package code base manually that is not recommended.

After this build stage, our code base package is going to be ready for the release stage. Inside this release stage, we need to combine the code base package with the deployment configurations based upon the environment.

Suppose if we are trying to release this build package into a production environment, we should combine our code base with the production related configurations like database credentials, folder structure, any server related properties or microservice related properties.

So, all those details we need to club, and we need to prepare an immutable release component with its own unique identifier, just like how we maintain inside versioning like 6.1.5 or we can also follow the date and timestamp as a unique identifier.

So, inside a central repository we should store all our codebase artifacts along with the deployment configurations so that in future, if we are looking for a rollback to a specific version, it is going to make our life super easy.

And at last, once this release stage is completed, our application along with the configuration, is ready to be deployed and run as an microservice or as an application.

So that's why inside the last stage, which is run stage, we need to run the application in the designated environment using a specific release. So, whenever we are following these guidelines, we need to make sure that all these stages, they are maintaining strict separation. We should not try to club them and there is no runtime code modifications are allowed to prevent mismatches with the build stage.

Like once the build stage is completed, we should not do any modifications inside the code, at the runtime, inside our release stage or run stage.

That will bring some mismatches between our build stage and the next stages that we have. In other words, if I use the same immutable build and release artifact, I should always get the same kind of behaviour. So, the reproducibility is going to be super, super easy if you follow this principle.

1. **Configuration, credentials and code:** Configuration according to the 15-factor methodology? Configurations are the elements that are prone to change between the deployments. If there is a property that is going to differ from one environment to environment, we call all these kinds of properties or elements as configurations.

So, as per these 15-factor methodology, we should never club these configurations with our code base. We should have an ability to modify application configurations independently or without the need to rebuild the application for every environment.

The examples of these configurations are like our database properties or our message system properties, credentials for accessing third party APIs are feature flags, so all these kinds of configurations, they are confidential in nature and at the same time they're going to be differ from one environment to other environment.

So, if we take our database username and password, it is a confidential and at the same time the database credentials for dev, environment and QA environment and production environment are not going to be same.

So, for all such type of configurations, we need to make sure we are maintaining all of them in a separate codebase and at the same time we need to make sure we are not exposing any sensitive information while maintaining all these configurations in a separate codebase.

If needed, we need to provide all the sensitive configurations by following some externalization standards which we are going to discuss inside this course. In simple words, to follow this principle, our configuration should not be embedded within the code or tracked in the same code base except for the default configurations. So always the default configurations can be bundled within the application.

But anything that is going to change from environment to environment, we should not embed within our code base. If you try to embed it along with the code, then we need to generate the Docker image or the code base package for every environment which is against this 15-factor methodologies.

Like we can see here, first there will be a single code base which is going to hold all the business logic, which is not going to change from environment to environment. Using this single code base, during the build stage, we are going to generate a software package or a Docker image to this software package and Docker image while deploying to a specific environment like development, testing and production. We need to make sure at runtime we are injecting the configurations related to the specific environment.

Suppose if I am trying to deploy my microservice into a production environment at runtime or at deployment stage, I need to provide the production related configurations. This is one of the super important principles because since we are going to have hundreds of microservices and cloud native applications, it is always recommended to maintain the configurations related to them, which are going to change from environment to environment in a separate location.

1. **Log:** Inside a traditional application are a monolithic application, how the logs are handled. The application will write the logs into a file and folder location of our server.

So, if there are any issue comes inside our monolithic application, developers will go to that log location. They will try to open the log related to a specific date and timestamp and they will try to look at the logs.

What happened, what is the issue, what is the exception? But do we think this is going to work very similarly inside the microservice or cloud native applications?

Off course not, because we will have 100 differences of microservice. If there is any issue, are we going to look and search in all the 100 different servers or in all the 100 different locations where our applications are going to write the logs? That is not going to be a feasible option.

That's why as per this methodology, it is recommended that all the log routing and storage are not the applications concerned or cloud native applications are microservices they are not going to write the logs inside any log folder or inside any log location.

Instead, the application will simply redirect the logs to the standard output, treating them as a sequentially ordered events based upon the time.

The responsibility of the log storage and the rotation should be shifted to an external tool called log aggregator. This tool retrieves, gathers and provides access to the logs for the debugging purpose.

So, if we see here, there is an accounts microservice, loans microservice and cards microservice. They will simply print the logs to a standard output based upon the log statements and log framework that they use.

And behind the scenes, there will be a log aggregator tool that will keep on pulling for the logs, and all these logs will be aggregated in a single location so that the developer or operations team, they can search all the logs of all the microservice with the help of this log aggregator tool.

So, this is one of the guiding principles. we will feed all the logs from all the microservice into that tool and we are going to search all the logs of all the microservice from a single UI log aggregator tool. So, all these principles are very important, and we are going to use a lot of tools and solutions to implement and achieve all these guiding principles.

1. **Disposability:** Inside a traditional monolithic application, making sure that a single monolithic application is always running is a top priority, and there is no room for this monolithic application to terminate or to get stopped.

However, inside a cloud environment or inside cloud native applications or inside microservices, it is not going to critical and necessary as well because we will have hundreds of microservices, and they'll be having multiple instances running.

It is not possible to manually monitor and making sure and all these instances and microservices are running always.

That's why applications in the cloud are always considered as ephemeral, meaning that if a particular microservice or a cloud native application becomes unresponsive, it can be terminated and replaced with a new instance by platforms like Kubernetes automatically and at the same time, during high load periods or during high load traffic, additional instances of the applications can be spin up to handle the increased workload.

This concept of shutting down and creating new instances automatically is called application disposability, where the applications can be started or stopped as needed. But to make our applications disposability and manage our applications in the dynamic environments like cloud, it is crucial to design them for quick startup when new instances are required and for graceful shutdown when they are no longer needed.

This fast startup enables systems elasticity and ensuring robustness and resilience. Without these fast capabilities, performance and availability issues may arise like we discussed before, with the help of Spring boot framework and Docker containers, any microservice we can create and destroy within seconds.

Whereas with virtual machines and monolithic applications, it is going to take at least 10 to 15 minutes. So that's why we are following Docker containers along with the useful framework like Spring Boot.

And whenever a graceful shutdown is being involved for an application, the application should be capable of not accepting any new request and at the same time any ongoing requests should be processed successfully and then only it should exit. So, this process is going to be straightforward for a web application.

However, for worker process or any other type of process, it involves returning any pending jobs to the worker queues before exiting.

This conveys that always graceful shutdown is also important. We should not ignore the existing requests that are being processed by a particular instance or a microservice that we are trying to shut down. When we use Docker container along with an orchestrator like Kubernetes, they are going to inherently satisfy these disposable requirements.

That's why inside this course also we already started using Docker and by the end of this course, also going to introducing to Kubernetes and we will run all our microservices inside Kubernetes cluster as well.

1. **Backing services:** Our microservices may have dependency on many other external resources like database Smtp servers, FTP servers, caching systems, message brokers. So, all these external resource dependencies, we call them as backing resources.

So, we should always treat these backing resources as attached resources so that we can modify or replace them without needing to make any changes inside our application code.

For example, consider the use case of a database throughout our software development life cycle. Typically, different database is used in different stages, such as development, testing and production.

We should always treat these databases as an attached resource so that we can easily switch to a different service depending on the environment.

How this attachment is going to work through resource bonding, which involves providing necessary information like URL, username and password of the database through externalized configurations.

So, once we generate a Docker image, we should not be generating the Docker image again and again whenever we want to change to our different database, instead to the same code base, we need to provide these URL information, user information and password information from an external configuration.

1. **Environment parity:** As per this guiding principle, it is recommended to minimize the differences between various environments of our application and avoid any costly shortcuts. If our environments they look very similar, then our application also is going to work in a very similar manner.

Just to fix some issue, if we are making some changes inside an environment by using shortcuts, then the behaviour is going to be different environments.

That's why this environment parity recommends making sure all your environments are looking exactly as much as possible. When we use this environment parity, we are going to address three different types of gaps. The very first gap is time gap.

**Time Gap** - Usually the time it takes for a code change to be deployed can be significant, but this methodology encourages automation like adopting CI/CD pipelines and perform continuous deployment to reduce the time between the code development and production deployment.This will make sure always our environments are looking similar, and it is going to make developerslife easy to perform any debugging.

**people gap** - Usually developers, they create applications while platform operators they handle the deployment in production. To bridge this gap, a DevOps culture should be followed. And this DevOps culture, it promotes collaboration between developers and operators, fostering the philosophy, which is build it, we run it.  
When the people gap is reduced, then there will be more coordination and collaboration, which will help to make environments look very similar. And with that, the number of issues is going to be less.  
And even if an issue comes since the environment are looking same, debugging is going to be easy

**Tool gap** - Applications can always use backing services and sometimes they can differ from environment to environment. For a microservice, a developer might use an H2 database locally, but in production they might be using PostgreSQL. This is not recommended because to achieve the environment parity we need to make sure we are using the same type of tool and the same version of the backing service across all environments.  
If we maintain different type and versions of backing services in different environments, then definitely the developer will code his business logic that is going to work with a H2 database and the same code base may not work for the Postgres SQL. That's why it is very important to use the same backing services in all the environments.

1. **Administrative processes:** Many times, we will be having many management tasks required to support application such as database migration, any batch jobs to clean the data or to update the data.

So, all these maintenance tasks and management tasks, they should be treated as isolated process. Like application process. The code for these administrative or management tasks should be version controlled and packaged along with the application and executed within the same environment.  
Many times, developers, they will try to skip these administrative tasks running in the local dev or QA. For example, they don't run these batch jobs or data migrations, any maintenance tasks just to save some time and so that they can deploy the code into production quickly.

But that will sometimes bring surprising results in production. That's why we should make sure these management and administrative processes are also equally important, and they are treated as a separate isolated process, and they are also properly tracked using versioning system and packaged along with the application and deployed in each environment where the application is being deployed.

It is always advisable to consider these administrative tasks as independent microservices where they are executed only once and when they are not needed. We can discard them. If you try to put this administrative process inside our business logic, then we are unnecessarily carrying all these administrative tasks inside our microservice. Instead, if we keep our microservice business logic and these administrative tasks separately, once we are done with these administrative tasks, we can discard them so that our microservice alone can run continuously and serve its clients.

Alternatively, these administrative tasks, they can also be integrated directly into the application, which can be activated by calling a designated endpoint. But it is always better to deploy them as an independent microservice.

1. **Port binding:** As per this guideline, all the cloud native applications should be self-contained and expose their services through port binding. When we say self-contained, an application should not rely on an external server within the execution environment.

For instance, a traditional Java web application might typically run within a server container like Tomcat, Jetty or Undertow. We as a developer should manually deploy our Java code into these servers. In contrast to this, a cloud native application does not depend on the presence of a Tomcat server in the environment.

It manages a server as a dependency within itself. We are already using these inside our microservices, for example using spring boot, we are enabling the usage of an embedded server where an application incorporates the server within itself instead of relying on its availability in the execution environment.

Once the application is started using these self-contained servers, each application is going to map to its own server compared to traditional approach were deploying multiple applications in a single server. In simple words, we should not deploy multiple applications in a single server. Instead, every application should be deployed in a separate server and even that server also should be self-contained but not external to the application.

Once the application is started with its own self-contained server, the applications should expose its services to the outside world through port binding.

When we are trying to start our Docker image as a Docker container with the help of docker run command, we use the port forwarding or port mapping. So, using that port binding, we are exposing the microservice to the external network.

Once this application is exposed at a specific port, then any other service or any other clients they can invoke our microservice and these microservices are cloud native application is going to act as a backing service for another application.

So, this is very common practice within cloud native systems because many applications are many microservices they want to communicate with each other. In such scenarios we should make sure we are properly exposing our microservice by following these self-contained and port binding standards.

Never ever develop cloud native applications, which depends on an external Tomcat server or an external server. With such scenarios, we will end up with managing these external servers manually in all the locations where your cloud native applications and microservices are being deployed.

1. **Stateless processes:** We know that the cloud native applications are the microservices are developed with high scalability in mind. One of these key principles to achieve scalability is to design our applications as stateless process and adopting a shared nothing architecture. Like we can see here, there is a single microservice where it has multiple instances running. To make it simpler. Let's try to assume we have our accounts microservice. If it is receiving a lot of traffic, what we will do, we will scale our accounts microservice by onboarding multiple new instances of same accounts microservice. So, all these multiple instances of accounts, microservice they should follow stateless process, and they should not share anything between themselves.

This is important because when the traffic is low or when an instance is not working properly, we are going to destroy it, or we are going to recreate it. If the instance of a microservice is going to hold some data, there will be some data loss would occur and the business logic will be impacted.

That's why our application should be strictly stateless. However, sometimes there might be a requirement where our accounts microservice or a specific cloud native service, it must store some data, or it must store some user data. So, for all such scenarios, these instances are accounts microservice or cloud native service, they should use a backing service like a data store, for example we can use a database, or we can use a Redis cache to store all the caching related information. This will remain our application stateless and whenever they want to store something, they can store inside the database or any other data store.

Even a particular instance of accounts microservice is shut down and in future, after few minutes if a new instance Is coming up, there is no loss of information. Everything is stored inside a data store from where our new instance can read and execute the business logic.

So, that's why always make sure we are not storing anything inside our instance, such as user session related information or caching information, because we are going to lose all that information as soon as a particular instance is shut down.

That's why please make sure always every information is being stored inside the storage system and making our instances and cloud native applications as true stateless applications.

1. **Concurrency:** We discussed that by implementing stateless applications we can achieve the scalability. But scalability cannot be achieved solely by creating stateless applications. While statelessness is important, scalability also requires the ability to serve large number of users. This means our applications should support concurrent processing to handle multiple users simultaneously.

This means if we have ten different microservice instances are running inside our microservice network, they should not accept only ten different requests and a process sequentially one by one. Instead, they should be capable of processing lot of requests parallelly simultaneously to achieve this, according to the 15-factor methodology process play a very crucial role in application design.

Whenever we are getting a greater number of traffic, which means we need a greater number of processes to handle such traffic. So, in such scenarios we can horizontally scale our processes to distribute the workload across multiple processors on different machines. Inside Java we already have this concurrency inbuilt developed when we built our applications with the help of Java and JVM, concurrency is typically managed by the program itself with the help of multiple threads which are available from the thread pools.

And whenever we are trying to scale our application, we should never follow the vertical scalability instead, we should follow the horizontal scalability. We can see vertical scalability means we are going to increase the Ram memory and CPU processes for a mission.

To some extent, this vertical scalability may work, but once we reach to a maximum CPU processors and maximum Ram, then we cannot scale our applications vertically.

That's why we should always follow the horizontal scalability inside the horizontal scalability. We are going to create multiple virtual machines with the same configurations as two GB Ram and two CPU.

And inside these virtual machines we can create containers, or we can create processes based upon our requirements. And there is no limit for our horizontal scalability. We can create any number of virtual machines, and we can deploy any number of containers inside them.

And usually, these processes which handles the traffic of the application, they are categorized based upon their respective types.

For example, all the web processes responsible for handling the Http requests. And just like web processes, we also have worker processes that execute some scheduled background jobs. So, by classifying all these processes and optimizing their concurrency, applications can effectively scale and handle increased workloads without having this concurrency ability inside our application and inside our programming language that we are going to use, our cloud Native applications or microservices. They are not going to be easily scaled.

1. **Telemetry:** Inside monolithic application we will have very limited number of applications, like 1 or 2 monitoring them, understanding their logs, metrics, performance related information is very easy because we must monitor only one server or one application or two applications or two servers.

Whereas with cloud native applications or microservices, we will be having multiple containers running, multiple services running, multiple servers will be running inside our organization. So how are we going to monitor them. For the same we have a concept called observability.

So, this observability is a fundamental characteristic of cloud native applications. Since we are going to have multiple applications inside the cloud it becomes essential to have access to the accurate and comprehensive data from each component of the system. If we have 100 different microservices, we should be able to access the accurate information and comprehensive information about all these microservices in a single place. So that it can enable remote monitoring of the systems behaviour and facilitate effective management, what kind of data we need for this effective management. Telemetry data such as logs, metrics, traces, health status and events. All this plays a very vital role in providing this visibility into our cloud Native applications.

If we try to understand how this telemetry is derived in Kevin Hoffman's analogy, he emphasizes the significance of telemetry by comparing with the applications to the space probes. So, he's trying to compare our cloud native applications or microservices to the space probes. Space probes are nothing, but the satellites are the space rockets that we send into the space for research. Space organizations like NASA, Isro.

They will use this telemetry data of the space probe to monitor and control remotely. The same concept applies to the cloud native applications as well. To effectively monitor and control applications remotely, we need various types of telemetry data, and these telemetry data are like detailed logs for troubleshooting metrics to measure performance, traces to understand request flows, health status to access system well-being and events to capture significant occurrences, by gathering and utilizing all these type of telemetry data, we can gain valuable insights about our applications and microservices and make informed decisions to manage them effectively from a remote location.

We are going to talk in detail about what is this telemetry, how to gather all this telemetry information into a single place inside this course.

But on a high level, what this guideline is recommending is please make sure our applications are feeding all this telemetry information to a centralized component. And from that centralized component, we should be able to monitor and control their behaviour.

1. **Authentication and authorization:** Now, moving on to the last guiding principle, which is authentication and authorization. So, security is a very critical aspect of any software system.

But many times, we see that this security is not receiving necessary emphasis it deserves. So, what this guiding principle is saying is, we need to follow a zero-trust approach, and we need to make sure, every communication and every interaction within the system and within the microservice network or cloud native systems is happening by following the security standards. When we talk about security, there are many things involved apart from authentication and authorization, which is the responsibility of the developer.

Apart from these authentication and authorization platforms team, they can follow Https protocol, they can have some SSL certificates; they can have some firewall protection. So, all these standards also our operations team, they are going to follow.

But from the development perspective, following these authentication and authorization is very important. So, what is an authentication, authentication enable us to track and identify who is the user trying to access our application. Once the authentication is completed, like with the help of username and password, we can then proceed to evaluating their permissions and determine if they have necessary authorization to perform a specific action.

So inside authentication will only check the identity of the end user, whereas inside the authorization that is going to happen after the authentication, we are going to validate if a specific end user has enough privileges to perform a specific action inside an application.

So, implementing these authentication and authorization is very important for cloud native applications and microservices. Inside this course there is a separate section focusing on the security and how to implement it inside the microservices. We are going to leverage the standards like OAuth 2.1 and OpenID Connect to enforce security inside our microservices.