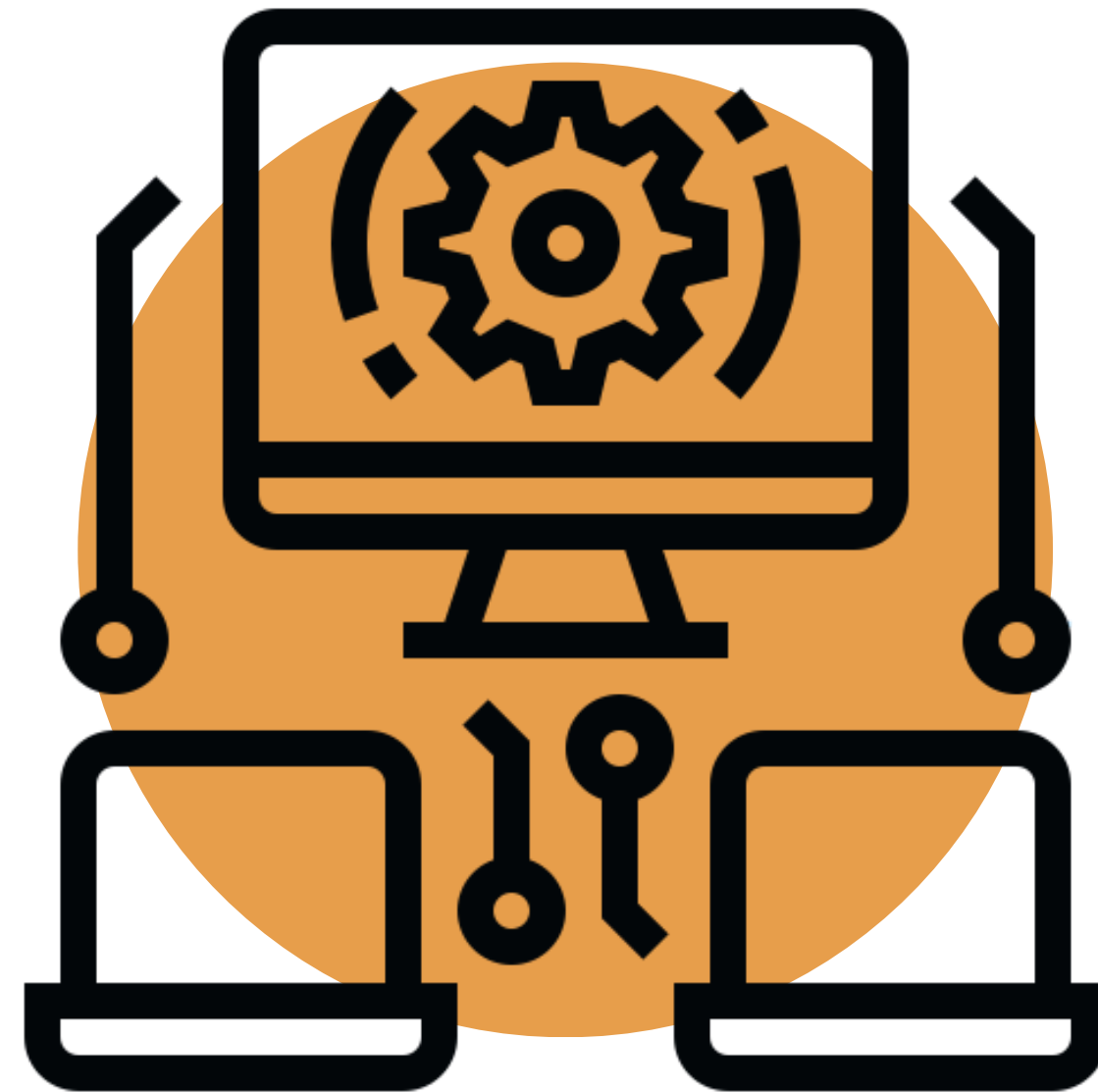


Lesson 3

Principles of Cloud Native design - 12 Factor apps



Socialize keywords of the reading below

"Principles of Cloud Native design - 12 Factor apps"



Cloud Native Architecture

Refers to the software model for building, deploying, and managing modern applications in cloud computing environments, focusing on scalability, flexibility, and resilience.

Immutable Infrastructure

Servers for hosting cloud-native applications remain unchanged after deployment, allowing for predictable and automated deployment processes.

Microservices

Small, independent software components that work collectively as complete cloud-native software, promoting modularity, scalability, and agility.



12 Factor Application

A set of principles for building microservices-based cloud-native applications, emphasizing modularity, scalability, and agility.

Dependencies

Explicitly declaring and isolating project dependencies, including packages, platforms, and SDKs, to enhance compatibility and reproducibility.

Application Configuration

Storing configuration in the environment rather than hard-coding it in the source code, ensuring dynamic modifiability and consistency across environments.

Logs

Treating logs as event streams, writing log entries to stdout and stderr, and decoupling applications from log storage, processing, and analysis.

Reading: "Principles of Cloud Native design - 12 Factor apps"

Cloud Native architecture

Organizations the world over are working through application modernization programs of various sizes. Business systems are not only complex but there is a demand now for them to be responsive, provide innovative features, embrace rapid change , provide resilience and scale. These systems are expected to accelerate business velocity and growth through strategic transformations. Organizations plan to achieve these through their app modernization programs. A critical part of these application modernization programs is the decision to refactor or rebuild existing applications amongst other considerations such as replatform, replace, lift and shift etc. Designing applications in the new cloud native world requires thinking through the underlying principles, patterns and best practices of systems architecture.





Traditional architectures optimized for fixed, high-cost infrastructure with lower number of components. Monolithic architectures were popular, often due to the cost of physical resources and the slow velocity in which applications were developed and deployed. Cloud-native architectures take advantage of the distributed, scalable and loosely coupled nature of public, private and hybrid cloud environments. Cloud-native architecture focuses on achieving resilience and throughput through horizontal scaling, distributed processing, and automating the replacement of failed components. This architecture treats the underlying infrastructure as disposable. The infrastructure can be provisioned in minutes and resized, scaled, or destroyed on demand – via automation. The architecture favors the development of small, independent, loosely coupled services which can be delivered quickly. A loosely coupled architecture is an application design strategy in which the various parts of an application are developed, deployed and operated independently of each other.



The Cloud Native Computing Foundation - CNCF provides an official definition of Cloud Native -

“Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.”

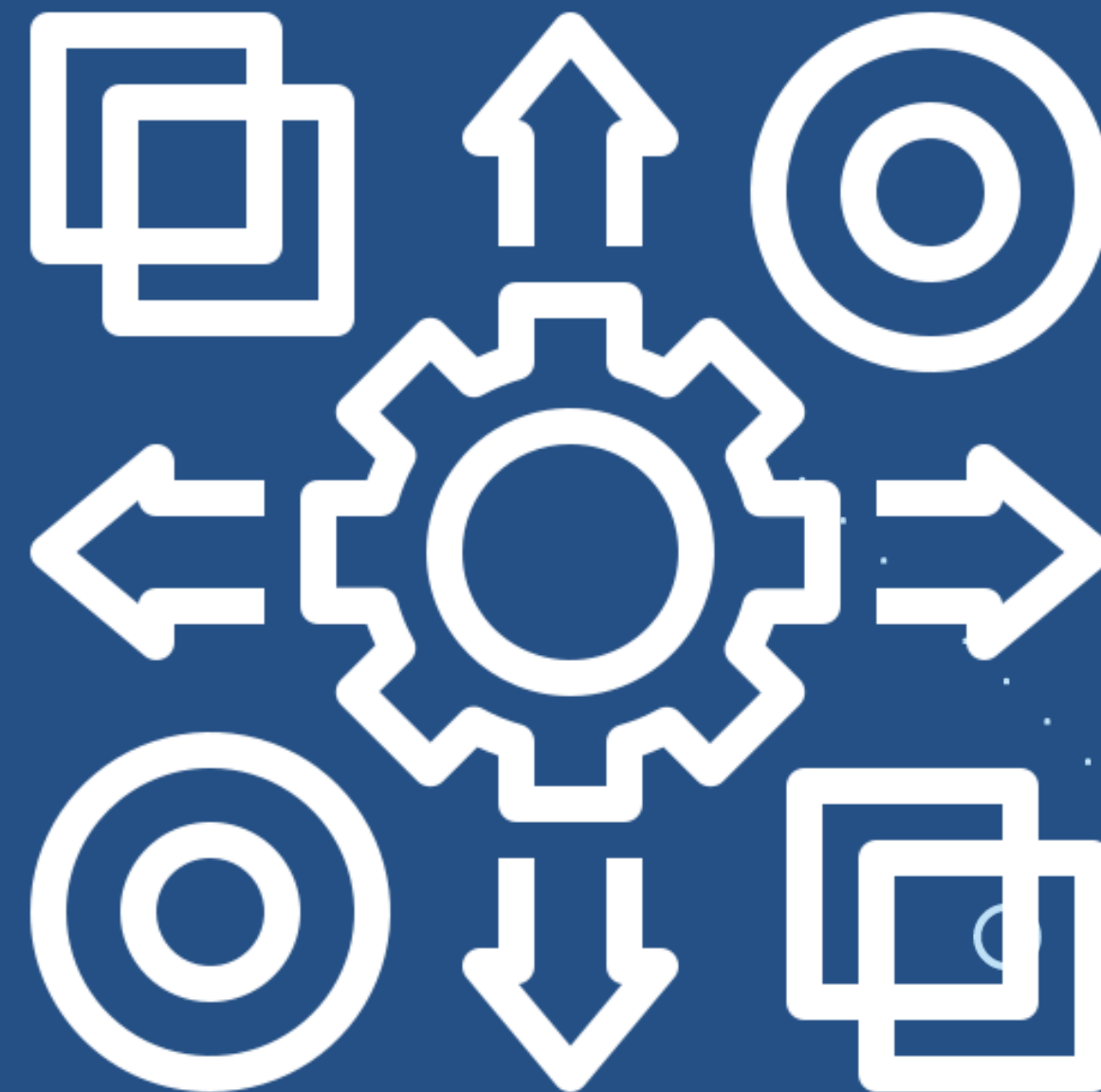
“These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.”

A widely accepted methodology to build cloud native applications is the 12 factor application.



The Twelve Factors

The twelve-factor app principles are a collection of best practices for building microservices-based cloud-native applications. These applications are modular, scalable, and agile. They are designed to perform at web scale and provide high resiliency. These principles are applied to cloud-native applications and are programming language and platform agnostic. The 12 Factors were published in 2012 by Adam Wiggins, founder of Heroku. It is hosted here . This methodology presents a thought process on how to develop applications and is complementary to other methodologies and thought processes such as the reactive manifesto.



The 12 factors are

Presionar cada factor para ver su contenido



Codebase

One codebase tracked in revision control, many deploys.

Dependencies

Explicitly declare and isolate dependencies.

Configuration

Store configuration in the environment.

Backing Services

Treat backing services as attached resources.

Build, release, run

Strictly separate build and run stages.

Processes

Execute the app as one or more stateless processes.



Port binding

Export services via port binding.

Concurrency

Scale out via the process model.

Disposability

Maximize robustness with fast startup and graceful shutdown.

Dev/prod parity

Keep development, staging, and production as similar as possible.

Logs

Treat logs as event streams



Admin processes

Run admin/management tasks as one-off processes

Taken from: <https://pradeepi.com/blog/12-factor-cloud-native-apps/>

Codebase

“A twelve-factor app is always tracked in a version control system... A codebase is any single repo (in a centralized revision control system like Subversion), or any set of repos who share a root commit (in a decentralized revision control system like Git).”




This principle advocates a single codebase tracked in revision control with many deployments across multiple environments. There can only be one codebase per microservice. The codebase must be managed by a version control system. Various deploys are generated from this codebase, each one for a different environment development, staging, production and maybe others. This looks too simplistic and nonsensical but if you think about it in the world of SOA and microservices we have to think through our version control strategies . Each service should be maintained as its own codebase and should be version controlled independently.




Every developer clones their copy of the codebase to make changes or run locally. The platform will pull source code from a single repository and use this code to build a single deployable unit. Individual branches or tags are used based on the branching and release strategy. However, there is no one size fits all with codebase strategies and teams generally tend to move from Mono repo to Multi repos based on various factors. Mono Repo is where all Microservices are housed within a single repository. In multi repo each microservice codebase is tracked in its independent repo.



Dependencies




“A twelve-factor app never relies on the implicit existence of system-wide packages.” Dependencies used by a project and their versions must be explicitly declared and isolated from code. Explicitly stating versions results in lower compatibility issues across environments. This also results in better reproducibility of issues occurring in specific version combinations. Dependencies not only include packages but also platforms, SDK’s etc. Package dependencies can be managed using package management tools like Nuget, NPM etc. Container technology simplifies this further by explicitly specifying all installation steps. It also specifies versions of base images, so that image builds are idempotent. .Net core provides the project file as a container to declare all dependencies. It uses nuget as the package manager to download the necessary packages.




Application configuration

“Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires strict separation of config from code. Config varies substantially across deployments, code does not.”





Application configurations that differ across environments such as external dependencies, databases, credentials, ports etc are only manifested at runtime. This configuration should not be hard coded in the source code but should be externalized and dynamically modifiable from outside the application. There should be no hard-coded credentials and no configuration in the code. This ensures that the application is not modified to update configuration to deploy it across environments and is completely agnostic of the environment. This also ensures that sensitive information is not mixed in with code. The use of environment variables that can be injected when deploying an application in a specific environment is highly recommended. This ensures that the developer can focus on code with the assurance that the necessary configuration and credentials are available consistently across all environments. In addition to environment variables tools such as consul and vault enable configuration to be stored in a secure way across environments. An example of externalizing configuration is here.



Backing Services


“The code for a twelve-factor app makes no distinction between local and third-party services...Each distinct backing service is a resource.”

Databases, API's and other external systems that are accessed from the application are called resources. The application should be abstracted away from its external resources. These resources should be attached to the application in a loosely coupled manner. Backing services should be abstracted into individual components with clean interfaces. They should be replaceable by different instances without any impact on the application using the application configuration principle above.


- The generic implementation should allow backing services to be attached and detached at will.
- Administrators should be able to attach or detach these backing services to quickly replace failing services without the need for code changes or deployments.
- Other patterns such as circuit breaker , retries and fallback are also recommended when using these backing services.

Build, Release, Run

“The twelve-factor app uses strict separation between the build, release, and run stages.”




This principle is closely tied in with the previous principles. A single codebase is taken through the build process to produce a compiled artifact. The output of the build stage is combined with environment specific configuration information to produce another immutable artifact, a release. Each release is labelled uniquely. This immutable release is then delivered to an environment (development, staging , production, etc.) and run. If there are issues this gives us the ability to audit a specific release and roll back to a previously working release. All of these steps should be ideally performed by the CI/CD tools provided by the platform. This ensures that the build, release and run stages are performed in a consistent manner across all environments.



Processes

“Twelve-factor processes are stateless and share-nothing.”



All processes and components of the application must be stateless and share-nothing. An application can create and consume a transient state while handling a request or processing a transaction, but that state should all be gone once the client has been given a response. All long-lasting states must be external to the application and provided by backing services. Processes come and go, scale horizontally and vertically, and are highly disposable. This means that anything shared among processes could also vanish, potentially causing a cascading failure. This principle is key to characteristics such as fault-tolerance, resilience, scalability, and availability.

Port Binding


“The twelve-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service.”

A twelve-factor app is fully self-contained and does not depend on any runtime such as application servers, web servers etc to be available as a service. It is self-contained and exposes its functionality via a protocol that best fits it such as HTTP, MQTT, AMQP etc. A twelve-factor app must export the service by port-binding, meaning that the application also interfaces with the world via an endpoint. The port binding can be exported and configurable using the configuration principle above.


- An application using HTTP as the protocol might run as `http://localhost:5001` on a developer's workstation, and in QA it might run as `http://164.132.1.10:5000`, and in production as `http://service.company.com`.
- An application developed with exported port binding in mind supports this environment-specific port binding without having to change any code.

Concurrency

“In the twelve-factor app, processes are a first class citizen...The process model truly shines when it comes time to scale out.”



Applications should scale out using the process model. Elastic scalability can be achieved by scaling out horizontally. Rules can be setup to dynamically scale the number of instances of the application/service based on load or other runtime telemetry. Stateless, share-nothing processes are well positioned to take full advantage of horizontal scaling and running multiple, concurrent instances.



Disponability

“The twelve-factor app’s processes are disposable, meaning they can be started or stopped at a moment’s notice.”

Processes are constantly created and killed on demand. An application’s processes should be disposable, and allow it to be started or stopped rapidly. An application cannot scale, deploy, release, or recover rapidly if it cannot start rapidly and shut down gracefully. Shutting down gracefully implies saving the state if necessary, and releasing the allocated computing resources. This is a key requirement due to the ephemeral nature of cloud native applications.

Dev/Prod Parity


“The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small.”

All environments should be maintained to be as similar as possible. This ensures that any environment specific issues are identified as early as possible.



Logs


" A twelve-factor app never concerns itself with routing or storage of its output stream."




Logs should be treated as event streams. Logs are a sequence of events emitted from an application in time-ordered sequence. A cloud-native application writes all of its log entries to stdout and stderr. You can use tools like the ELK stack (ElasticSearch, Logstash, and Kibana), Splunk etc to capture and analyze your log emissions. Applications should be decoupled from the knowledge of log storage, processing, and analysis. Logs can be directed anywhere. For example, they could be directed to a database in NoSQL, to another service, to a file in a repository, to a log-indexing-and-analysis system, or to a data-warehousing system.

Admin Processes

“Run admin/management tasks as one-off processes.”



Maintenance tasks, such as script execution for data migration, initial data seeding and cache warming should be automated and performed on time. These are executed in the run time environment and should be shipped with the release for the specific code base and configuration. This ensures that the maintenance tasks are performed on the same environment that the application is running on. This principle is key to the ability to ship the application with the maintenance tasks.



× Multiple choice: quizizz game.

LINK: https://quizizz.com/admin/quiz/65b7c67f95c97582e0e49086?source=quiz_share



1. Multiple Choice 45 seconds 1 point

1. What is the primary focus of Cloud Native Architecture?

☐ A) Fixed infrastructure ☐ B) Monolithic structures

☐ C) Scalability, flexibility, and resilience ☐ D) Low-cost development

2. Multiple Choice 45 seconds 1 point

2. What does "Immutable Infrastructure" mean in cloud-native applications?

☐ A) Ever-changing servers ☐ B) Unpredictable deployment processes

☐ C) Servers remain unchanged after deployment ☐ D) Manual updates to infrastructure

3. Multiple Choice 45 seconds 1 point

3. What are Microservices in the context of cloud-native applications?

☐ A) Large, interconnected software components ☐ B) Independent software components working together

☐ C) Single, monolithic applications ☐ D) Static elements of a cloud-native system

4. Multiple Choice 45 seconds 1 point

4. What is the purpose of the 12 Factor Application principles?

☐ A) Enhancing monolithic applications ☐ B) Building large-scale infrastructure

☐ C) Ensuring modularity and agility ☐ D) Restricting scalability

5. Multiple Choice 45 seconds 1 point

5. What does the "Dependencies" principle in a 12 Factor Application emphasize?

☐ A) Implicit reliance on system-wide packages ☐ B) Explicit declaration and isolation of dependencies

☐ C) Avoidance of version control systems ☐ D) Unspecified compatibility across environments

6. Multiple Choice 45 seconds 1 point

6. How does cloud-native architecture handle application configuration?

☐ A) Hard-coding configuration in the source code ☐ B) Storing configuration in a centralized repository

☐ C) Dynamic modifiability using environment variables ☐ D) Relying on implicit system-wide configuration

7. Multiple Choice 45 seconds 1 point

7. What is the recommended approach for handling logs in a cloud-native application?

☐ A) Storing logs as constants in the code ☐ B) Directing logs to a single, fixed location

☐ C) Treating logs as event streams and writing to stdout and stderr ☐ D) Ignoring log storage and analysis concerns.

INICIO