**12 Factor App Principles and Cloud-Native Microservices**

12-factor app principles are set of principles for building scalable and performant, independent, and most resilient enterprise applications. It establishes the general principles and guidelines for creating robust enterprise applications. 12-factor app principles got very popular as it aligns with Microservice principles.

Below are the 12-factor principles.

* Codebase (One codebase tracked in revision control, but many deploys)
* Dependencies (Explicitly declare and isolate the dependencies)
* Config (Store configurations in an environment)
* Backing Services (treat backing resources as attached resources)
* Build, release, and 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 the 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-time processes)

## Codebase (One Codebase Tracked in Revision Control, Many Deploys)

12-factor app advocates that every application should have its own codebase (repos). Multiple codebases for multiple versions must be avoided. Please do note that having branches would be fine.

i.e. For all the deployment environments, there should be only one repo but not multiple.

Multiple apps sharing the same code are a violation of the twelve-factor. Here you should opt-in for shared libraries.

From the 12-factor app perspective app, deploy meaning the running instance of an app like production, staging, QA, etc. Additionally, every developer has a copy of the app running in their local development environment, each of which also qualifies as a deploy.

Different versions (the version is like a code change that is available in one environment but not in other) may be active in multiple deploys.

**Microservices:**In Microservices, every service should have its own codebase. Having an independent codebase helps you to easy CI/CD process for your applications.

## Dependencies (Explicitly Declare and Isolate the Dependencies)

It talks about managing the dependencies externally using dependency management tools instead of adding them to your codebase.

From the perspective of the java, you can think of **Gradle** as a dependency manager. You will mention all the dependencies in the **build.gradle** file and your application will download all the mentioned dependencies from the maven repository or various other repositories.

You also need to consider the dependencies from the **operating system/ execution environment** perspective as well.

**Microservices:** All the application packages will be managed through package(dependency) managers like **sbt**, **maven**.

* For non-containerized environments, you can go for configuration management tools like chef, ansible, etc. to install system-level dependencies.
* For a containerized environment, you can go for **Docker file**.

## Config (Store Configurations in an Environment)

Anything that varies between the deployment environments is considered a configuration. This includes:

* Database connections and credentials, system integration endpoints
* Credentials to external services such as Amazon S3 or Twitter or any other external apps
* Application-specific information like IP Addresses, ports, and hostnames, etc.

**You should not hardcode any configuration values as constants in the codebase. This is a direct violation of 12-factor app principles.**

12-factor app principles suggest storing the configuration values in the environment variables.

It advocates the strict separation between the code and configurations.

The code must be the same irrespective of where the application is being deployed to.

**Microservices:**Externalize the configurations from the application. In a microservice environment, you can manage the configurations for your applications from a source control like Azure GIT / GitHub (spring-cloud-config) and use the environment variables to store the sensitive information.

## Backing Services (Treat Backing Resources as Attached Resources)

As per 12 factor app principles, a backing service is an application/service that our application consumes over the network as part of its normal operation.

**Databases**, **Message Brokers**, **External Cache Service**s, and any other external systems that the application communicates is treated as **Backing Service**.

**12-factor app** can automatically swap the application from one provider to another without making any further modifications to the code base. Let us say, you would like to change the database server from **MySQL to Oracle**. To do so, you should not make any code changes to your application. Only configuration change should be able to take care of it.

By following the **interfaced based programming**, the application allows to swap the provider dynamically without impact on the system. **Plug-in based implementation** also helps you to support multiple providers.

## Build, Release, and Run (Strictly Separate Build and Run Stages)

The application must have a strict separation between the build, release, and run stages. Let us understand each stage in more detail.

**Build stage: It builds and** transforms the code into an executable bundle/ build package.

**Release stage:** It gets the build package from the build stage and combines with the configurations of the deployment environment and makes your application ready to run.

**Run stage:** It is like running your application in the execution environment.

**Microservices:**

You can use CI/CD tools to automate the build and deployment process. **Docker images make it easy to separate the build, release, and run stages more efficiently.**

## Processes (Execute the App as One or More Stateless Processes)

The app is executed inside the execution environment as a process. An **app can have one or more instances/processes** to meet the user/customer demands.

As per 12-factor principles, the application should not store the data in in-memory, and it must be saved to a data store and must be used from there later.

As far as the state is concerned, your application should store the state in the database instead of in memory of the process.

Avoid using sticky sessions as it is a violation of 12-factor app principles.

If you must store the session information, then you can choose **Redis** or **Memcached** or **any other external cache provider** based on your requirements.

By following these, your app can be highly scalable without any impact on the system.

**Microservices:**By adopting the stateless nature of REST, your services can be horizontally scalable as per the needs without any impact on the system. If your system still requires maintaining the state then you can use the attached resources (Redis, Memcached, or datastore) to store the state instead of in-memory of the process.

## Port Binding (Export Services via Port Binding)

The twelve-factor app is completely self-contained and doesn’t rely on runtime injection of a webserver into the execution environment to create a web-facing service. The web app exports HTTP as a service by binding to a port and listening to requests coming in on that port.

In short, this is all about having your application as a standalone instead of deploying them into any of the external web servers.

**Microservices:**Spring boot is one example of this one. Spring boot by default comes with embedded tomcat, jetty, or undertow.

## Concurrency (Scale Out via the Process Model)

This talks about scaling the application. Twelve-factor app principles suggest **running your application as multiple processes/instances instead of running in one large system.**

You can still opt-in for threads to improve the concurrent handling of the requests.

In a nutshell, twelve-factor app principles suggest to opt-in for horizontal scaling instead of vertical scaling.

(**Vertical scaling** - Add additional hardware to the system

**Horizontal scaling** - Add additional instances/processes of the application)

**Microservices:**By adopting **containerization**, applications can be scaled horizontally as per the demands/needs.

## Disposability (Maximize the Robustness with Fast Startup and Graceful Shutdown)

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

When the application is shutting down or starting, an instance should not impact the application state.

**Graceful shutdowns are very important. The system must ensure the correct state.**

The system should not be impacted when new instances are added or removing the existing instances as per need. **This is also known as system disposability.**

Systems do crash due to various reasons. The system should ensure that the impact would be minimal, and the application should be stored in a valid state.

**Microservices:**By adopting containerization into the deployment process of microservices, your application implicitly follows this principle to a maximum extent. Docker containers can be started or stopped instantly. Storing requests, state, or session data in queues or other backing services ensures that a request is handled seamlessly in the event of a container crash.

## Dev/Prod Parity (Keep the gap between Development, Staging, and Production as Similar as Possible)

The twelve-factor app principles suggest keeping the gap between development and production environment as minimal as possible. This reduces the risks of showing up bugs in a specific environment.

The twelve-factor developer resists the urge to use different backing services between development and production environments.

## Logs (Treat Logs as Event Streams)

Logs are important in troubleshooting the production issues or understanding user behaviour.

Logs provide visibility into the behaviour of a running application.

Twelve-factor app principles suggest separating the log generation and processing the log's information.

From the application, logs will be written as standard output and the execution environment takes care of capture, storage, curation, and archival of the logs.

**Microservices:**In Microservices, observability is the first-class citizen. Observability can be achieved by using **APM tools** (ELK, Dynatrace, and other tools) or **Log Aggregations Tools** like Splunk, logs, etc.

By following the above-mentioned guidelines all you need to debug an issue is to go to **the central dashboard of your tool and search for it**.

## Admin Processes (Run Admin/Management Tasks as One-time Processes)

There are several one-time processes as part of the application deployment like data migration, executing one-time scripts in a specific environment.

Twelve-factor principles suggest keeping such administrative tasks as part of the application codebase in the repository. **By doing so, one-time scripts follow the same process defined for your codebase.**

Ensure that **one-time scripts are automated** so that you don’t need to worry about executing them manually before releasing the build.

Twelve-factor principles also suggest using the **built-in tool of the execution environment to run those scripts on production servers**.

**Microservices:**Containerization also helps here to run the one-time processes as a task and shutdown automatically one done with the implementation.