THE 12-FACTOR APP METHODOLOGY EXPLAINED

Writing application code predates most cloud solutions—solutions that most application programmers write specifically for these days.

To handle this dissonance, the 12-Factor App methodology has emerged. The 12 factors is an approach that helps programmers write modern apps in a declarative way, using clear contracts deployed via cloud.

In this article, I'll introduce the 12-factor app methodology and offer a high-level summary of its principles.

What is the 12-Factor app methodology?

In 2012, programmers at Heroku debuted the 12-Factor app methodology. These programmers have developed and deployed hundreds of apps wrote this methodology, drawing on their experience of seeing SaaS apps "in the wild".

This group considers the methodology a triangulation of:

- Ideal practices to support app development
- The dynamics that occur as an app grows organically
- The relationship and collaboration between codebase developers

Their goals are two-fold:

- To help avoid software erosion costs
- To raise awareness of systemic problems they've observed in modern app development

The group points to two inspirational sources, Patterns of Enterprise Application Architecture and Refactoring, both by professional developer Martin Fowler.

And now, I'll introduce each of the 12 factors.

Principle I. Codebase

"One codebase tracked in revision control, many deploys"

Your code base should have a logical version control system that's easy to understand.

Every deployment should have its own code repository that can be deployed to multiple environments. Avoid housing multiple applications in the same repository. This makes version control almost impossible to understand and versions will get tangled up, resulting in non-value-added work.

Principle II. Dependencies

“Explicitly declare and isolate dependencies”

This principle maintains that you should never rely on the implicit existence of system-wide packages. Instead"

- Make sure that app-specific libraries are available
- Verify shelling out to the OS
- Verify that needed system libraries, like curl or ImageMagick, are available. (There's no guarantee that these exist on every system where the app could run in the future.)

Overall, a 12-factor app must be self-containing. The application should be isolated sufficiently to avoid interactions with conflicting libraries that are installed on the host machine.

Let's look at an example. In Python, you can achieve declaration and isolation by using Pip and Virtualenv respectively. To satisfy this principle, you must always use both declaration of dependency **and** isolation. Manage dependencies within the application, not from an external source. Dependencies should be hosted in a repository within the app

Principle III. Config

“Store config in the environment”

An application and its configuration should be completely independent. Further, storing configs constantly in code should be avoided entirely.

Your configurations should have a separate file and shouldn't be hosted within the code repository. A separate config file makes it easy to update the config values without touching the actual code base, eliminating the need for re-deployment of your applications when you change certain config values.

When configurations are in the environment, not the app, as variables, you can easily move it to another environment without touching the source code. Twelve-factor apps store configs as variables so that they are “unlikely to be checked into the repository” accidentally. Another bonus: then your configs are independent of language and OS.

Principle IV. Backing services

“Treat backing services as attached resources”

In a 12-factor app, any services that don't support the core app must be accessed as a service. These non-core essential services might include:

- Databases
- External storage
- Message queues
- Etc.

These can be treated as a resource. These should be accessed as a service via HTTP or similar request, then specified in the config. This way, the service's source can be changed without affecting the app's core code.

For example, an app that uses a message queuing system is best if it can easily change from RabbitMQ to ZeroMQ to ActiveMQ by only changing configuration information.

Principle V. Build, release, run

"Strictly separate build and run stages"

A 12-factor app is strict about separating the three stages of building, releasing, and running.

Start the build process by storing the app in source control, then build out its dependencies. Separating the config information means you can combine it with the build for the release stage—and then it's ready for the run stage. It's also important that each release have a unique ID.

Principle VI. Processes

"Execute the app as one or more stateless processes"

Store any data that is required to persist in a stateful backing service, such as databases. The idea is that the process is stateless and shares absolutely nothing.

While many developers are used to "sticky sessions", storing information in the session expecting the next request will be from the same service contradicts this methodology.

Principle VII. Port binding Principle

"Export services via port binding"

12-factor apps must always be independent from additional applications. Every function should be its own process—in full isolation.

In a traditional environment, we assume that different processes handle different functionalities. As such, it's easy to further assume these functionalities are available via a web protocol, such as HTTP, making it likely that apps will run behind web servers, like Apache or Tomcat. But this is counter to the 12-factor methodology.

Instead, add a web server library or similar to the core app. This means the app can wait requests on a defined port, whether that's HTTP or a different protocol.

Principle VIII. Concurrency

"Scale out via the process model"

A true 12-factor app is designed for scaling. Build your applications so that scaling in the cloud is seamless. When you develop the app to be concurrent, you can spin up new instances to the cloud effortlessly.

In order to add more capacity (start additional processes on additional machines), your app should be able to add more instances instead of more memory or CPU on the local machine.

Principle IX. Disposability

"Maximize robustness with fast startup and graceful shutdown"

The concept of disposable processes means that an application can die at any time, but it won't affect the user—the app can be replaced by other apps, or it can start right up again. Building disposability into your app ensures that the app shuts down gracefully: it should clean up all utilized resources and shut down smoothly.

When designed this way, the app comes back up again quickly. Likewise, when processes terminate, they should finish their current request, refuse any incoming request, and exit.

Principle X. Dev/prod parity

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

Applications deployed to development and production should have parity. Essentially, there should be only the slightest difference between both deployments.

A vast difference may lead to unintended compatibility issues between dev and production code. When building a 12-factor app, backing services between dev and prod must be the same; a difference here could cause significant issues down the line.

Principle XI. Logs

"Treat logs as event streams"

Unlike monolithic and traditional apps that store log information in a file, this principle maintains that you should stream logs to a chosen location—not simply dump them into a log file.

Typically, logs don't reside in the same place within cloud-based environments after every burn. As new processes start or your app crashes, the logs will be distributed across several cloud machines; they won't sit on a single machine or host.

Solve this issue by naming a common place for the logs to stream. In some instances, you can simply redirect Stdout to a file. More likely, however, you'll want to deploy a log router like Fluentd and save the logs to Hadoop or a specific service, like Splunk.

Principle XII. Admin processes

"Run admin/management tasks as one-off processes"

The final 12-factor app principle proposes separating administrative tasks from the rest of your application. These tasks might include migrating a database or inspecting records.

Though the admin processes are separate, you must continue to run them in the same environment

and against the base code and config of the app itself. Shipping the admin tasks code alongside the application prevents drift.

Related reading

- BMC DevOps Blog
- Agile vs Waterfall SDLCs: What's The Difference?
- Deployment Pipelines (CI/CD) in Software Engineering
- What Is Extreme Programming (XP)?
- How & Why To Become a Software Factory
- Top Conferences for Programming & Software Development