Implementing the Twelve-Factor App Methodology for Developing Cloud-Native Applications

By, Janakiram MSV

Executive Summary

Application development has gone through a fundamental shift in the recent past. Developers are under pressure to deliver software that runs in a variety of environments including enterprise data centers and cloud platforms. The modern software is expected to scale with no major changes to the code. Software has to be shipped rapidly to deliver new features and user experiences.

In the era of web, mobile, and device computing, application development needs to adopt a new methodology. The rise of cloud, containers, and microservices has led to a new paradigm of developing and deploying software. The twelve-factor app methodology advocates a set of patterns and practices for designing contemporary applications that take advantage of cloud infrastructure. Since these applications are born in the cloud to exploit the capabilities of modern infrastructure, they are often referred as cloud-native applications.

This report attempts to introduce the twelve-factor app methodology to developers focused on building contemporary, cloud-native applications.

Key Attributes of Cloud-Native Applications

Cloud-native applications are differentiated from their counterparts based on their packaging, deployment, configuration, and modularity. They have the following attributes:

- **Automated -** Cloud-native apps separate the code from configuration. The external configuration declared outside of the code encourages automation, which reduces the cost of deployment. It also delivers consistency and repeatability of deployments.
- **Portable -** Cloud-native apps have zero dependency on the underlying operating system and cloud infrastructure. They are easily ported across multiple environments with no changes to the code.
- **Agile -** The components or modules of cloud-native apps can be rapidly upgraded with minimal disruption to the system. They follow the principles of continuous deployment for moving the code from development environments to production.

- **Elastic -** Cloud-native apps can run in highly constrained environments with minimal resources or can be scaled to support a large number of users. They can be scaled-in and scaled-out with no changes to the application code.

The Twelve Factors

The twelve-factor app methodology was originally conceived to define the patterns for applications targeting PaaS offerings such as Heroku and Engine Yard. Given the resemblance between PaaS and container platforms, these principles are applicable for modern, cloud-native applications.

Below is a list of the patterns and practices advocated by this methodology.

1. Codebase

The very first principle of the methodology is related to versioning and tracking the code. The entire code base that belongs to the application must be stored in one repository such as Git or SVN. There may be multiple versions or branches but there must be a clear 1:1 correlation between the code, repo, and the application. The running instance of code is called as a deployment. There may be multiple deployments of the same codebase running in different environments such as staging and production.

Key takeaway - Maintain one and only one codebase for all the modules associated with the application.

2. Dependencies

An application must explicitly declare all the dependencies. It cannot make assumptions about the availability of dependencies. It is the job of the platform to ensure that all the dependencies declared by the app are available at runtime. Examples of such declarations include Ruby Gem files and a Node.js Package.json file. The clean separation between code and dependencies enables DevOps to do their job efficiently. The operations team can ensure that the dependencies are met while developers stay focused on the code.

Key takeaway - Applications cannot make assumptions about dependencies. Always explicitly declare the dependencies.

3. Configuration

This principle emphasizes on the clean separation of configuration and code. The separation can be achieved through environment variables. Since the application can retrieve the configuration at runtime, the same codebase can be seamlessly deployed in multiple environments with no changes. For example, the database connection string can be passed as an environment variable to the application, which can be different between development, testing, and production environments. Though configuration can be moved to files that are managed by the same version control system, environment variables are better suited as they are always expected at a well-known location. Configuration files have the risk of getting fragmented and duplicated.

Key takeaway - Store configuration in environment variables and read them at runtime.

4. Backing Services

The fourth factor deals with the backing services management. Backing services are external services, which an app depends on over a network connection. This can be a local database service such as MySQL or MongoDB or any other 3rd party service such Amazon RDS or mLab. For a twelve-factor app, the interface to connect to these various services should be defined in a standard way. In most cases it would be accessed over a URL with credentials stored in the config. There should not be any code changes when switching between the local and remote services. For example, a twelve-factor app should be able to switch between a local MySQL database and a remote one by simply changing the config.

Key takeaway - Treat backing services as attached resources.

5. Build, Release, Run

This principle emphasizes clean separation of the stages in the software lifecycle. The build stage must deal with building the binaries by combining the code, dependencies, and other assets. The outcome from this phase is an executable bundle. The release stage combines the output from the build stage with the configuration specific to an environment, making it ready for a set of target environments. The final stage, run, executes the app in the select environment by choosing the right release from the previous stage.

Key takeaway - Strict separation of build, release, and run stages

6. Processes

The application is expected to run in an execution environment as one or more processes. Processes are completely stateless and share-nothing. The app cannot assume the presence of a local file, cache, or a database. Instead, it is expected to access it from a well-known location. An example of this includes moving files from a local file system to an object storage system like Amazon S3. Each process has to download its own copy to access the content. This encourages scalability in which the application can run simultaneously across multiple execution environments.

Key takeaway - Strict separation of stateful and stateless services

7. Port Binding

Cloud-native applications are exposed through a URL but cannot assume the port binding through which they are exposed to the external network. In most of the cases, there would be a proxy or a frontend server that forwards the requests to the backend application. Applications must bind themselves to a port that is externally declared in the configuration file. They should work in environments where they are exposed via external-facing services.

Key takeaway - Export services via port binding

8. Concurrency

Processes in twelve-factor apps are treated as first-class citizens. Each process should be able to scale, restart, or clone itself when needed. This loosely-coupled model enables scale-out architectures. For example, when the traffic increases to a web server, there may be multiple web processes responding to the requests. The worker processes responsible for dealing with the messages are scaled independent of the web processes. The web and worker processes may be connected via a message queuing system. This approach will improve the reliability and scalability of the application.

Key takeaway - Scale out via the process model

9. Disposability

The ninth factor emphasizes the robustness of an application through faster startup and shutdown methods. It is recommended that applications try to minimize the startup time for each process. This brings agility to the overall release process. Processes should shutdown gracefully against sudden death in the case of hardware failures. Applications can rely on robust queuing backends such as Beanstalkd and RabbitMQ that will return unfinished jobs back to the queue in the case of a failure.

Key takeaway - Maximize robustness with fast startup and graceful shutdown

10. Dev/Prod Parity

Twelve-factor apps are designed for continuous deployment by keeping the gap between development and production small. Consistency is key in bringing parity across dev, test, staging, and production environments. When deployment environments are similar, testing and developing gets much simpler. Consistent environments ensure that areas such as the infrastructure stack, config management processes, software and runtime versions and deployment tools are the same everywhere.

Since the test cases are applied on production-level data, this approach reduces the bugs in the production environment.

Key takeaway - Keep development, staging, and production as similar as possible

11. Logs

Logging plays an important role in debugging and maintaining the overall health of an application. At the same time, an application in itself shouldn't burden itself with the storage of logs. Instead, the logs should be treated as a continuous stream that is captured and stored by a separate service. Each running process may write its event stream, unbuffered, to stdout. During local development, the developer can view this stream in the foreground of their terminal to observe the app's behavior. In staging or production environments, the logs are routed to one or more final destinations for viewing and long-term archival.

Key takeaway - Treat logs as event streams

12. Admin Processes

Applications often deal with one-off administrative processes such as initializing a database, cleaning up the database, and migration of servers. These processes should be run in an identical environment as the regular long-running processes of the app. If they run against a specific release, they should use the same codebase and config as any process run in that release. Admin code must ship with application code to avoid synchronization issues. Twelve-factor methodology prefers languages that provide a REPL shell out of the box, which makes it easy to run one-off scripts.

Key takeaway - Run admin/management tasks as one-off processes

DreamFactory as a Twelve-Factor Application

DreamFactory is an API-first platform to develop data-driven applications. It exposes a variety of databases and data stores as standard REST APIs. DreamFactory is architected based on the twelve factors of modern application development.

DreamFactory is maintained as a single repository on Github that can be easily cloned or forked. Since the platform is built on top of PHP and Laravel frameworks, the dependencies are clearly defined in composer.json. The configuration is maintained in a set of environment variables, making it easy to deploy in multiple environments. DreamFactory can run in a variety of environments ranging from Raspberry Pi to the leading web-scale cloud platforms.

Developers can easily integrate DreamFactory with their containerized applications. The platform can be deployed on Docker Swarm, Kubernetes, and Mesosphere environments.

Summary

The twelve-factor app methodology helps developers with the best practices for building cloud-native applications. By adopting these principles, they can design, develop, and deploy applications that are reliable, portable, scalable, and extensible. These principles are highly relevant for the modern-day applications that target web, mobile, and device users.

DreamFactory can be integrated with the cloud-native applications for easier integration with legacy and modern data sources.