The 12 Factor App

2012

4 min read • 815 words

This talk recaps the 12 Factor App methodology, which is a set of best practices for building scalable, maintainable, and portable web applications.

Introduction

The Twelve-Factor App methodology represents a comprehensive approach to building software-as-a-service applications that prioritizes consistency, scalability, and deployment flexibility across different environments. Born from practical experience rather than academic theory, this methodology emerged from real-world patterns observed in production systems.

> Kenneth's presentation of the 12-factor methodology at Heroku helped establish these principles as industry standards. The methodology codified practices that Heroku discovered through hosting thousands of applications, making implicit knowledge explicit.

I. Codebase

The first principle establishes a fundamental constraint: maintain a single codebase tracked in version control but deployed across multiple environments. This seemingly simple requirement creates clarity about application boundaries —if multiple codebases exist, you're dealing with a distributed system rather

than a single application. The principle enforces architectural clarity while enabling consistent deployment patterns across development, staging, and production environments.

II. Dependencies

Dependency management must be explicit and isolated, eliminating reliance on system-wide packages or tools that might vary between environments. This principle ensures true portability by making all dependencies visible and manageable through the application's dependency declaration. No more hoping that the right version of a library happens to be installed system-wide—everything the application needs should be explicitly stated and automatically installable.

III. Config

Configuration data—database connections, service credentials, environment-specific settings—must be stored in the environment rather than in code. This externalization makes applications truly portable across different deployment contexts without requiring code changes or rebuilds.

> The configuration principle fundamentally changed how developers think about application deployment. By externalizing configuration, applications became truly portable across environments, a concept that seems obvious now but was revolutionary when first articulated.

This separation enables the same build to run in development, staging, and production with only environmental differences, dramatically reducing deployment complexity and configuration drift between environments.

IV. Backing Services

Backing services—databases, message queues, mail services, caching systems—should be treated as attached resources, accessible via URLs or configuration stored in the environment. The application should make no distinction between

local services (like a local PostgreSQL database) and third-party services (like Amazon S3). This abstraction enables seamless service swapping and promotes loose coupling between the application and its dependencies.

V. Build, Release, Run

The deployment pipeline must strictly separate three stages: build (transform code into executable bundle), release (combine build with configuration), and run (execute the application in the target environment). This separation enables clean rollbacks, parallel development, and robust release management. Each release becomes an immutable artifact that can be deployed consistently across environments or rolled back when issues arise.

VI. Processes

Applications should execute as one or more stateless processes that share nothing and persist data only to stateful backing services. Process isolation enables horizontal scaling and fault tolerance—if a process crashes, it can be restarted cleanly without affecting other processes or corrupting shared state. This statelessness is fundamental to building applications that can scale elastically with demand.

VII. Port Binding

Applications should be completely self-contained and expose services by binding to ports rather than relying on external web servers for execution. This principle makes applications portable across different runtime environments and enables them to become backing services for other applications. The application becomes its own web server, eliminating deployment dependencies on specific server configurations.

VIII. Concurrency

Scale applications horizontally using the Unix process model, assigning different workloads to different process types (web processes, worker processes, etc.). Rather than threading within a single large process, applications should

embrace process-based concurrency managed by the operating system. This approach enables fine-grained resource allocation and scaling strategies that match workload characteristics.

IX. Disposability

Processes should be disposable, capable of starting or stopping on short notice while maintaining system robustness. Fast startup times enable rapid scaling and deployment, while graceful shutdown ensures that in-flight work completes properly. This disposability is essential for fault tolerance, enabling systems to recover quickly from hardware failures or deployment updates.

X. Dev/Prod Parity

Development, staging, and production environments should be kept as similar as possible to minimize gaps in time, personnel, and tools. This parity reduces deployment surprises and ensures that testing accurately reflects production behavior. The closer these environments mirror each other, the more confident teams can be in their deployments.

XI. Logs

Applications should treat logs as continuous event streams, writing to stdout without concern for their routing or storage. This separation of concerns allows the execution environment to handle log aggregation, routing, and analysis using specialized tools. The application focuses on generating meaningful events while the infrastructure handles log management.

XII. Admin Processes

Administrative and maintenance tasks should run as one-off processes in the same environment as regular app processes, using identical code and configuration. This ensures that admin tasks have access to the same resources and behave consistently with the main application, preventing subtle bugs that arise from environment differences.

For more details, visit 12factor.net.
