

12-Factor Apps

WW Developer Advocacy Team

IBM Developer



What is 12 Factor?



Created by developers of the Heroku platform who have witnessed the scalability of thousands of apps

<https://12factor.net>

[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

What is 12 Factor?

Industry best practices for building apps that are:

- **Portable** between on-prem, cloud and hybrid cloud
- **Horizontally Scalable.** Reliable and elastic scaling w/o changing code
- **Automated.** Because ain't nobody got time for that
- **Traceable and Observable** in a microservices environment
- **Robust.** Design for failure, recover quickly.

The 12 Factors

- | | | | |
|------|---------------------|-------|-----------------|
| I. | Codebase | VII. | Port Binding |
| II. | Dependencies | VIII. | Concurrency |
| III. | Config | IX. | Disposability |
| IV. | Backing Services | X. | Dev/Prod Parity |
| V. | Build, Release, Run | XI. | Logs |
| VI. | Processes | XII. | Admin processes |

I. Codebase

I. **Codebase**





- II. Dependencies
- III. Configuration
- IV. Backing services
- V. Build, release, run
- VI. Processes
- VII. Port binding
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes

- One codebase tracked in source code management (SCM) with versioning
- Multiple deployments from the same codebase
- Best practice is to automate deployment
 - Popular tools include Gradle and Jenkins

I. Codebase

Code for a single application should be in a single code base

- Track running applications back to a single commit
- Use Dockerfile Maven, Gradle, or npm to manage external dependencies
- Version pinning! Don't use '.latest'
- No changing code in production

 jzaccone committed on GitHub Update Dockerfile	
 src/main	hello message configurable, and controller at root
 .gitignore	Initial commit
 Dockerfile	Update Dockerfile

II. Dependencies

- I. Codebase
- II. Dependencies**
- III. Configuration
- IV. Backing services
- V. Build, release, run
- VI. Processes
- VII. Port binding
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes

- Explicitly declare and isolate dependencies
- Typically language-specific
 - Node.js: Node Package Manager (NPM)
 - Liberty: Feature manager
 - Ruby: Bundler
 - Java EE: Application resources
- Never rely on system-wide dependencies

II. Dependencies

Explicitly declare and isolate dependencies. AKA: Remove system dependencies

How?

- Step 1: Explicitly declare dependencies (Dockerfile)
- Step 2: Isolate dependencies to prevent system dependencies from leaking in (containers)

```
1 FROM openjdk:8-jdk-alpine
2 EXPOSE 8080
3 WORKDIR /data
4 CMD java -jar *.jar
5 COPY target/*.jar /data/
```


III. Configuration

- I. Codebase
- II. Dependencies
- III. Configuration**
- IV. Backing services
- V. Build, release, run
- VI. Processes
- VII. Port binding
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes

- Store configuration in the environment
- Separate configuration from source
- Enables the same code to be deployed to different environments

III. Config

Store config in the environment (not in the code).

How?

- Inject config as environment variables (language agnostic)
- ConfigMap in Kubernetes does this

```
$ docker run -e POSTGRES_PASSWORD=abcd postgres
```

IV. Backing Services

- I. Codebase
- II. Dependencies
- III. Configuration
- IV. Backing services**
- V. Build, release, run
- VI. Processes
- VII. Port binding
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes

- Treat backing services as attached resources:
 - Databases
 - Messaging systems
 - LDAP servers
 - Others
- Local and remote resources should be treated identically
 - Possible locations for run time and resources:
 - In the same process
 - On the same host
 - On different hosts in the same data center
 - In different data centers

IV. Backing Services

Treat backing resources as attached services. Swap out resources.

How?

- Pass in URLs via config (see III.)
- K8s built-in DNS allows for easy service discovery

```
services:  
  
account-api:  
  build:  
    context: ./compute-interest-api  
  environment:  
    DATABASE_URL: http://account-database  
  
account-database:  
  image: jzaccone/account-database
```

V. Build, Release, Run

- I. Codebase
- II. Dependencies
- III. Configuration
- IV. Backing services
- V. Build, release, run**
- VI. Processes
- VII. Port binding
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes

- Strictly separate build and run stages
- The runtime code should not be modifiable as there is no way to put those changes back into the build stage

V. Build, Release, Run

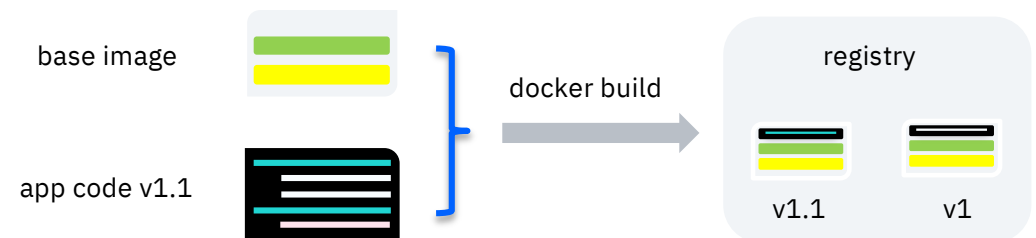
Strictly separate build and run stages.

Why?

Rollbacks, elastic scaling without a new build

How?

- Use Docker images as your handoff between build and run
- Tag images with version. Trace back to single commit (see I. Codebase)
- Single command rollbacks in Kubernetes



VI. Process

- I. Codebase
- II. Dependencies
- III. Configuration
- IV. Backing services
- V. Build, release, run
- VI. Processes**
- VII. Port binding
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes

- Run the application as one or more stateless processes
- Do not rely on session affinity, also called sticky sessions
- State should be stored in a stateful backing service external to the process

VI. Process

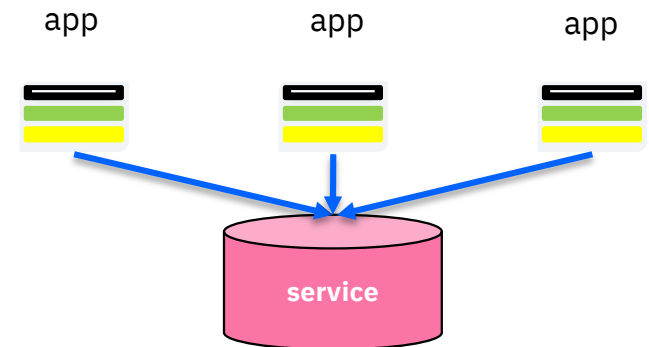
Execute app as stateless process

Why?

Stateless enables horizontal scaling

How?

- Remove sticky sessions
- Need state? Store in volume or external data service
- Use persistent volumes in Kubernetes for network wide storage



VII. Port Binding

- I. Codebase
- II. Dependencies
- III. Configuration
- IV. Backing services
- V. Build, release, run
- VI. Processes
- VII. Port binding**
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes

- Export services with port binding
- Web app binds to an HTTP port and listens for requests coming in on that port

VII. Port Binding

Export services via port binding. Apps should be self-contained.

Why?

Avoid “Works on my machine”

How?

- Web server dependency should be included inside the Docker Image
- To expose ports from containers use the `—publish` flag

VIII. Concurrency

- I. Codebase
- II. Dependencies
- III. Configuration
- IV. Backing services
- V. Build, release, run
- VI. Processes
- VII. Port binding
- VIII. Concurrency**
- IX. Disposability
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes

- Scale out, not up, using the process model
To add capacity, run more instances
- There are limits to how far an individual process can scale
- Stateless applications make scaling simple

VIII. Concurrency

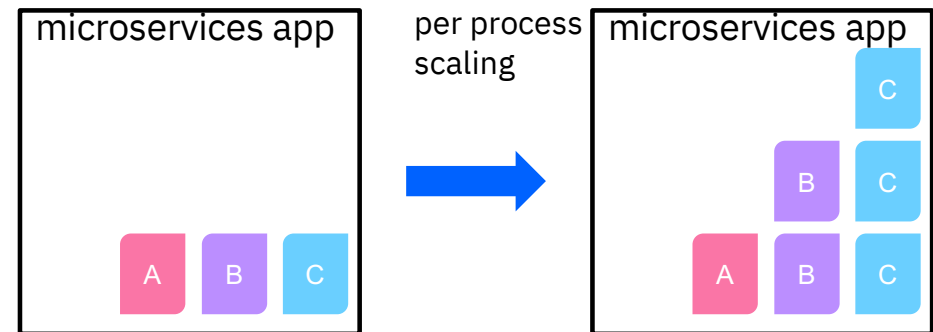
Scale out via the process model.

Processes are **first-class citizens**

How?

- Scale by creating more processes
- **Docker**: really just a process running in isolation
- **Kubernetes**: Acts as process manager: scales by creating more pods

Don't put process managers in your containers



Bad Example

```
# Start the first process
./my_first_process -D
status=$?
if [ $status -ne 0 ]; then
    echo "Failed to start my_first_process: $status"
    exit $status
fi

# Start the second process
./my_second_process -D
status=$?
if [ $status -ne 0 ]; then
    echo "Failed to start my_second_process: $status"
    exit $status
fi
```

```
FROM ubuntu:latest
COPY my_first_process my_first_process
COPY my_second_process my_second_process
COPY my_wrapper_script.sh my_wrapper_script.sh
CMD ./my_wrapper_script.sh
```

Containers should be a single process!

IX. Disposability

- I. Codebase
- II. Dependencies
- III. Configuration
- IV. Backing services
- V. Build, release, run
- VI. Processes
- VII. Port binding
- VIII. Concurrency
- IX. Disposability**
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes

- Maximize robustness with fast startup and efficient shutdown
- Application instances are disposable
- Application should handle shutdown signal or hardware failure with crash-only design
- Containers built on this principle

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

Why?

- Enables fast elastic scaling, robust production deployments. Recover quickly from failures.

How?

- multi-minute app startups!
- Docker enables fast startup: Union file system and image layers
- In best practice: Handle SIGTERM in main container process.

X. Dev/Prod Parity

- I. Codebase
- II. Dependencies
- III. Configuration
- IV. Backing services
- V. Build, release, run
- VI. Processes
- VII. Port binding
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity**
- XI. Logs
- XII. Admin processes

- Keep development, staging, and production as similar as possible
 - Use the same backing services in each environment
 - Minimize incompatible
 - Backing services
 - Tools
 - Platforms
- across environments

X. Dev/Prod Parity

Keep development, staging and production as similar as possible.

Minimize time gap, personnel gap and tools gap

How?

- **Time gap:**

Docker supports delivering code to production faster by enabling automation and reducing bugs caused by environmental drift.

- **Personnel gap:**

Dockerfile is the point of collaboration between devs and ops

- **Tools gap:**

Docker makes it very easy to spin up production resources locally by using `docker run ...`

XI. Logs

- I. Codebase
- II. Dependencies
- III. Configuration
- IV. Backing services
- V. Build, release, run
- VI. Processes
- VII. Port binding
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity
- XI. Logs**
- XII. Admin processes

- Treat logs as event streams
- Each process writes to stdout
 - Application should not write to specialized log files
 - Environment decides how to gather, aggregate, and persist stdout output
- Log tool example - ELK stack (Elasticsearch, Logstash, and Kibana)
 - Stream, store, search, and monitor logs

XI. Logs

Treat logs as event streams

How?

- Write logs to stdout (Docker does by default)
- Centralizes logs using ELK, EFK or [your tool stack here]

Don't write logs to disk!

Don't retroactively inspect logs! Use ELK to get search, alerts

Don't throw out logs! Save data and make data driven decisions

XII. Admin Processes

- I. Codebase
- II. Dependencies
- III. Configuration
- IV. Backing services
- V. Build, release, run
- VI. Processes
- VII. Port binding
- VIII. Concurrency
- IX. Disposability
- X. Dev/prod parity
- XI. Logs
- XII. Admin processes**

Run administrative and management tasks as single processes, such as these examples:

- Tasks for performing database migrations
- Debugging

XII. Admin Processes

Run admin/management tasks as one-off processes.

Don't treat them as special processes

How?

- Follow 12-factor for your admin processes (as much as applicable)
- Option to collocate in same source code repo if tightly coupled to another app
- “Enter” namespaces to run one-off commands via ``docker exec ...``

Docker + 12 Factors Summary

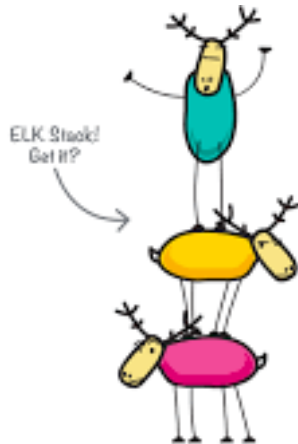
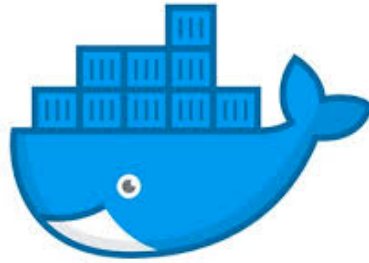
Inherent Benefits

- Eliminate environment drift
- Explicit and isolated dependencies via Dockerfile
- Explicit and automated dependencies in development environment
- Fast startup and deployments

Overlapping Best Practices

- Stateless processes
- Build your artifact (Docker Image), deploy many times.
- Configuration in environment
- URL defined backing resources

Tools to build 12-factor apps



E Elasticsearch

L Logstash

K Kibana



CLOUD NATIVE
COMPUTING FOUNDATION

The 12 Factors Github Example

`ibm.biz/12-factor`

Questions?