12-Factor Apps

John Zaccone
john.zaccone@ibm.com

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

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.

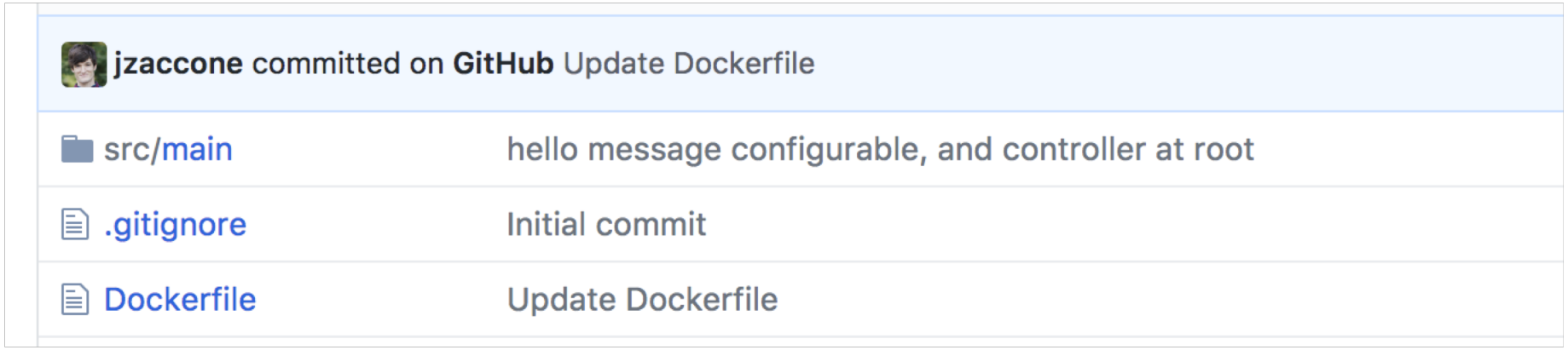
The 12 Factors

I. Codebase
II. Dependencies
III. Config
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

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



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)

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

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

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

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

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

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

Scale out via the process model. Processes are **first-class citizens**

Why?
Follow the Unix model for scaling, which is simple and reliable

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
./my_second_pro
status=$?
if [ $status -ne 0
  echo "Failed t
  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

Maximize robustness with fast startup and graceful shutdown

Why?

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

How?

- No 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

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

Treat logs as event streams

How?

- Write logs to stdout (Docker does by default)
- Centralizes logs using ELK 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

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

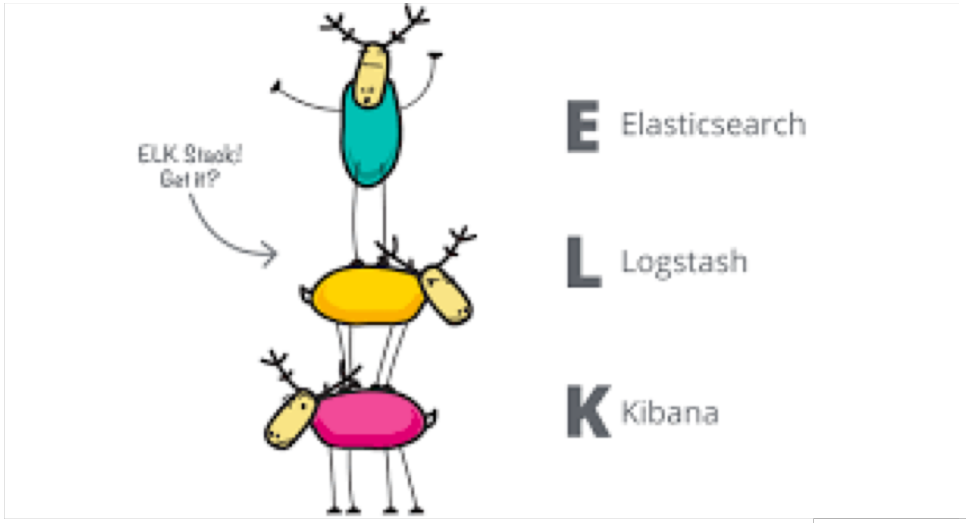
Inherent Benefits

- 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



The 12 Factors Github Example

ibm.biz/12-factor

Questions?