12 Factors App with Docker On AWS
About Me: Ma Bowen

- ThoughtWorks Senior Consultant
- Web/RoR/Java/Scala Developer, 3 years DevOps
- Book Translation <Scala Cookbook>
- AWS Certified Associate Solution Architect
12facters.net
Methodology for building Web Apps

- Use declarative formats for setup automation
- Maximum portability between execution environments
- Suitable for deployment on modern cloud platforms
- keep environment consistence, continuous deployment
- Scale with few changes to tooling/architecture etc
One Codebase, Multiple Deploys
Explicitly declare & isolate dependencies

- Ruby Gemfile, e.g `bundle install --path=vendor/bundle`
- Debian/RPM
Store Config in Environment

- **Test**
  - DB_HOST: testdb
  - DB_USR: readwrite
  - DB_PASS: 123456

- **Staging**
  - DB_HOST: stgdb
  - DB_USR: stgrw
  - DB_PASS: password

- **Production**
  - DB_HOST: proddb
  - DB_USR: prodrw
  - DB_PASS: password
Backing services as attached resources
Build release run
Processes

The app is executed in the execution environment as one or more processes. Twelve-factor processes are stateless and share-nothing.
Port Binding

The twelve-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service.
Twelve-factor app processes should never daemonize or write PID files. Instead, rely on the operating system’s process manager (such as Upstart, a distributed process manager on a cloud).
Disposability

Maximize robustness with fast startup and graceful shutdown
Dev/prod parity

"Perhaps your machine is the only one where it works?"

It works on my machine
Logs

Treat logs as event streams, a twelve-factor app never concerns itself with routing or storage of its output stream.
Admin processes

ASG
Scheduled action

Instance

One Off task

Run admin/management tasks as one-off processes
How do we apply this on AWS with Docker
### Some Context

<table>
<thead>
<tr>
<th>Years Ago</th>
<th>Now</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev</td>
<td>Ops</td>
</tr>
<tr>
<td>8 teams</td>
<td>40+ teams</td>
</tr>
<tr>
<td>monoliths</td>
<td>micro services (decommissioning)</td>
</tr>
<tr>
<td>2 Data Centers</td>
<td>2 DC + 100 AWS accounts</td>
</tr>
<tr>
<td>Ops Deploying</td>
<td>TMI &amp;&amp; Continuous Delivery</td>
</tr>
</tbody>
</table>
Some Glossaries

- **AMI**: Amazon Machine Image
- **ELB**: Elastic Load Balancer
- **ASG**: Auto Scaling Group
- **Cloudwatch**: AWS Monitoring Service
- **CloudFormation**: Manage AWS resources with JSON template
- **Newrelic**: Application Monitoring
- **Splunk**: Enterprise Log Aggregator
Continuous Delivery Before

ThoughtWorks

#164 was successful – Manual run from the stage: **Deploy to Production**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test and Package</td>
<td></td>
</tr>
<tr>
<td>Publish RPM</td>
<td>✔️</td>
</tr>
<tr>
<td>Test Scala</td>
<td>✔️</td>
</tr>
<tr>
<td>Upload dependency file</td>
<td>✔️</td>
</tr>
<tr>
<td>Aminate</td>
<td>✔️</td>
</tr>
<tr>
<td>Aminate</td>
<td>✔️</td>
</tr>
<tr>
<td>Backup Staging DB</td>
<td>✔️</td>
</tr>
<tr>
<td>Backup Staging Database</td>
<td>✔️</td>
</tr>
<tr>
<td>Deploy to Staging</td>
<td>✔️</td>
</tr>
<tr>
<td>Deploy to E2E</td>
<td>✔️</td>
</tr>
<tr>
<td>Backup Production DB</td>
<td>✔️</td>
</tr>
<tr>
<td>Backup Production Database</td>
<td></td>
</tr>
<tr>
<td>Deploy to Production</td>
<td>✔️</td>
</tr>
<tr>
<td>Deploy to Production</td>
<td>✔️</td>
</tr>
</tbody>
</table>

### Build result summary

**Details**

- **Completed**: 15 Mar 2016, 5:04:14 PM
- **Duration**: 25 minutes
- **Labels**: None

- **New failures**: 0
- **Existing failures**: 0

### Code commits

- Write a comment…
Processes
Cons

- Packaging twice, RPM/AMI
- Duplicated effort for automate deployment
- Not good for succession plan
- Dev/Test/Staging/Prod different
Docker – FTW
Expected

Commit ➔ CI ➔ Docker Image ➔ App

- 0-downtime deployment
- auto-rollback
- logging
- monitoring
AIM of shipper

- Standardising and simplify the way we deploy
- Portable between teams and account
shipper.yml

```
app:
  name: app
  image: app
  environment:
    SOME_ENV: "some_env"
  health_check:
    path: /diagnostic/status/heartbeat
    port: 9090

aws:
  instances:
    type: t2.micro
  load_balancer:
    scheme: internet-facing
    listeners:
      443:
        to: 80
        protocol: https
  splunk:
    host: splunk
    index: app
```
<table>
<thead>
<tr>
<th>Any web application</th>
<th>Auto-scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>any web-app framework</td>
<td>multiple servers</td>
</tr>
<tr>
<td>any programming language</td>
<td>load-balancing</td>
</tr>
<tr>
<td>any Linux variant</td>
<td>support for scaling schedules</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Splunk support</th>
<th>Zero-downtime deployments</th>
</tr>
</thead>
<tbody>
<tr>
<td>captures stdout/stderr</td>
<td>safe upgrades</td>
</tr>
<tr>
<td>no app support required</td>
<td>safe config changes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CloudWatch support</th>
<th>New Relic support</th>
</tr>
</thead>
<tbody>
<tr>
<td>alerts you when service is</td>
<td>application monitoring</td>
</tr>
<tr>
<td>down via email or web-hook</td>
<td>system monitoring</td>
</tr>
<tr>
<td></td>
<td>deployment notification</td>
</tr>
</tbody>
</table>
generic AMI

requests

$CONFIG

ngxin

<app/>

log collector

New Relic

Splunk

docker-registry

supporting services

your application
Actual Process
Immutable Deployment (1/2)
Immutable Deployment (2/2)
Docker V2 registry
Deployed 70+ systems
Next Step

- Support batched jobs
- ECS/ECR
- Swarm/Kubernetes
Fin

iambowen.github.io

iambowen.m@gmail.com