
Share IT
CONNECTING KNOWLEDGE, SOLUTIONS,
AND EXPERIENCE
4 OCTOBER, 2019 / CĒSIS CONCERT HALL

Share IT

Case study:
Sharing deployment code across

- **family of sites,**
- **environments,**
- **regions**
- **and CMS versions**

2019
Ragnar Kurm
Wunder

October 4, 2019 | Cēsis Concert Hall

Ragnar Kurm

Professional profile:

- 26 years of coding experience
- 8 years of drupal experience
- Experienced in ca 20 programming languages

Human profile:

- Human values
- Basic values of life
- Patterns of mind
- Education
- Teaching yoga

The problem

- A company has a family of sites
- Sites share a lot of deployment code
- But sites vary according to environments, regions and CMS versions
- Any modification to the deployment is very expensive (time-wise)

The cause

There was a lot of deployment code copy-pasted

(code duplication)

Remedy

Deduplication

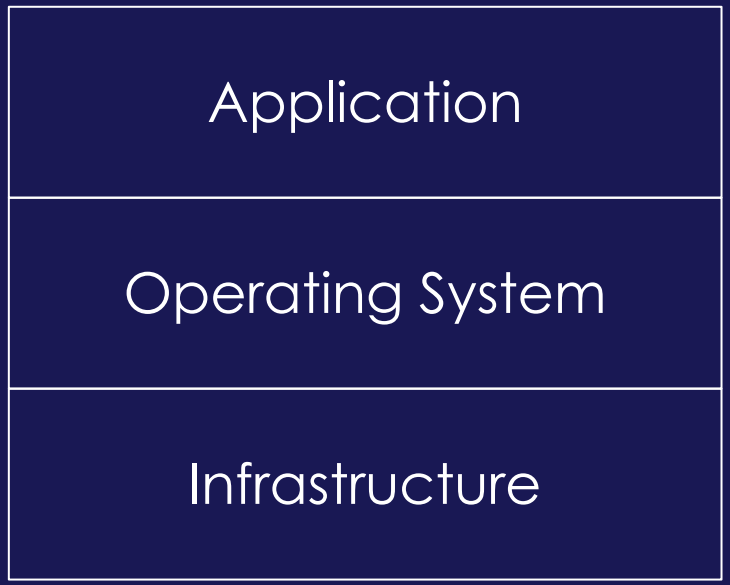
The challenge: its hairy stuff

Technical Context

- **Hosting:** AWS EC2 (disposable virtualhosts, not containers)
- **CI/CD:** TeamCity

The solution

Levels of code-sharing



Application level

Shared code is almost always as modules of a bigger software (or at least it should be strived to be).

This is beautifully solved by https://12factor.net/codebase:

Multiple apps sharing the same code is a violation of twelve-factor. The solution here is to factor shared code into libraries which can be included through the dependency manager.

Operating system level

A lot of miscellaneous stuff:

- Installing Operating system packages
- Configuring configuring packages
- Dealing with filesystem setup (mounting, permissions)
- Detecting leader host in load balancer context
- Running cron in a right way

Many of those items vary across multiple orthogonal categories:

- Type of environment (production, staging, development)
- Regions (EU, US, AU)
- Software versions (D7, D8)

Infrastructure level

???

Well … then … what's the recipe?

On a gross level the strategy would be this:

- Remove any code duplication
- Enable code sharing during deployment
- Cut code into Lego-pieces
- Organize Lego-pieces into layers
- Choose right layers for a particular deployment
- Execute layers in right order
- Populate filesystem in right order
- Add layers to the application as well
- Use environment variables to pass basic configuration parameters
- Bootstrap the whole process
- Profit

Code deduplication

- Find all similar code usages.
- Combine them into proper shell script.

It is joy to see one script handling all the sites.

For example, a script might do following:

- Install Postfix
- Configure it

And all emails will be handled properly in all your hosts.

Rinse and repeat.

Code sharing

There are two options

OS packages (RPM, DEB, …)
Suggested for those who are already familiar, comfortable and experienced creating packages. Allows more beautiful setup. Overhead managing all those packages + having a private package repository.

Repo (GitLab, GitHub, BitBucket, CodeCommit, …)
Might be easier setup. Challenge might be how to integrate the repo during deployment.

Carve the right pieces

Put each function into a separate script.

Examples

- `email_install_and_setup.sh`
- `cron_setup.sh`
- `shell_comfort.sh`
- `filesystems_mounting.sh`

Combining into layers

- Next, the scripts need to be organized into layers.
- Organizing means taking into account generality/specificity and grouping (including permutations) by category
- Only those layers are meaningful which contain scripts

Combining into layers – example

- Common
- Environment: production
- Environment: staging
- Environment: development
- Region: EU
- Region: US
- Region: AU
- Version: D7
- Version: D8
- Region-Version: EU-D8
- Region-Version: US-D8
- Environment-Region-Version: production-EU-D7

Choosing Layers for Deployment

When deploying for a host, choose right set of layers

Example

- Environment-Version: production-D8
- Environment: production
- Common
- Region: AU
- Version: D8

Execution order

- Execute from most general towards most specific
- Execution order does not matter between same-cardinality layers

Execution order – example

- Cardinality=0
  - Common
- Cardinality=1
  - Region: AU
  - Version: D8
  - Environment: production
- Cardinality=2
  - Environment-Version: production-D8
  - Version-Region: D8-AU
- Cardinality=3
  - Environment-Region-Version: production-AU-D8

Filesystem layering – structure of one

- A layer (or subfolder) represents filesystem root
- Populate files into that folder in same structure as they should end up in deployable host

Example

```
env-prod/etc/postfix/main.conf
env-prod/etc/postfix/master.conf
env-prod/usr/local/bin/postfix-setup.sh
env-prod/usr/local/deploy-bootstrap/1-env-prod.sh
```

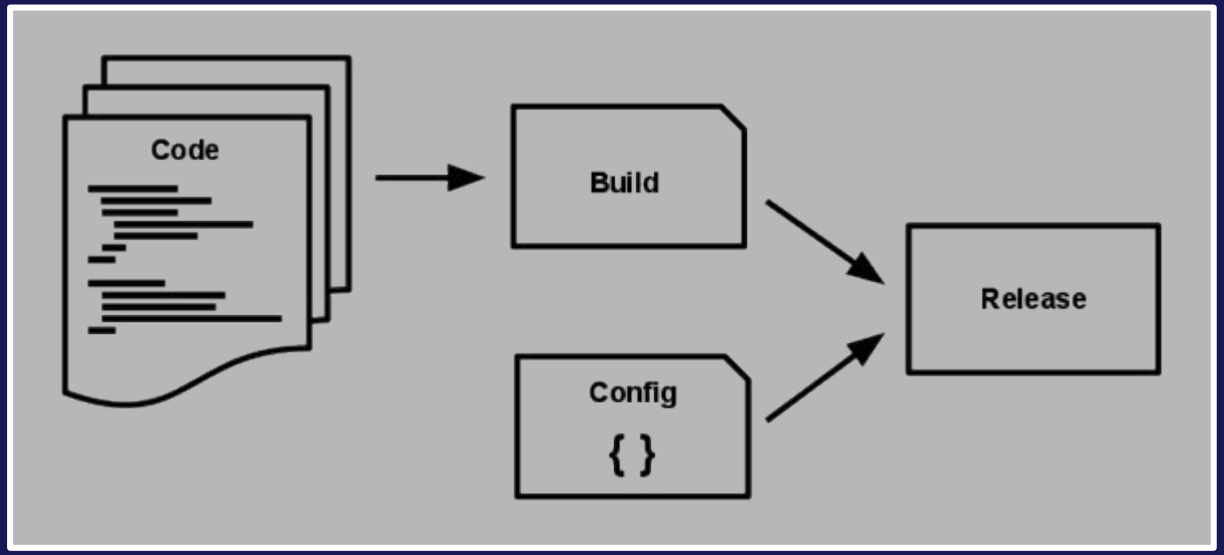
Filesystem layering – combining

- The concept is similar to Docker filesystem layers
- During deployment just copy contents of those folders into /
- This info resides in deployment system, not in a repo.
- Remember the ordering

Filesystem layering – combining – 12f

https://12factor.net/build-release-run

- Build
- Release
- Run



Filesystem layering – combining, ex.

Use: `cp -drv --preserve=mode dir/. /`

```
cp -r 00--common/. /
cp -r 01--region--au/. /
cp -r 01--version--D8/. /
cp -r 01--environment--production/. /
cp -r 02--environment-version--production-D8/. /
cp -r 02--version-region--d8-au/. /
cp -r 03--environment-region-version--production-au-d8/. /
```

Application layers

In practice there will also be a need to have layers on application side.

Example

```
app/deploy-layers/0--common/etc/apache/app.conf
app/deploy-layers/0--common/usr/local/bin/apache-setup.sh
app/deploy-layers/0--common/usr/local/bin/10--common.sh
```

Environment variables

Pass following info by environment variables by deployment system to host:

- Configuration / Per-deploy values
- Connectivity / Resource handles
- Secrets / Credentials

See also https://12factor.net/config

Environment variables – example

```
DB_HOST=db-jh45f.provider.com
DB_NAME=myapp
DB_PASS=nEs42KMsmnjRvSQbWMpxLSzn6HeUNVLq
DB_PORT=3306
DB_USER=joe
LISTEN_IF=0.0.0.0
LISTEN_PORT=8080
```

Ignite the fuze

```
for layer-bootstrapper in /usr/local/deploy-bootstrap/*.sh
do
  "$layer-bootstrapper"
done
```

Note, that each of bootstrapper script in turn executes scripts of specific function. Bootstrapper scripts do not carry out any task.

Practical tips

Logging

- Set your scripts to be verbose (`set -x`). Because it saves a lot of time. Spotting problems would be instantaneous.
- Many shell commands have verbose flag (`-v`), use that. Except when it produces excessive output. For example: `mv`, `cp`, `ln`, ...
- Avoid logging of secrets for security purposes
- Log exit code of scripts
- Log execution time of scripts

Logging – example

```
>>> install_composer_and_drush.sh >>>>>>>>>>>>>>>>>>>>>>>>>>>>
exit_code=0, duration=0

+ subshell email_persistency.sh

<<< email_persistency.sh <<<<<<<<<<<<<<<<<<<<<<<<<<<<
+ set -e
+ set -u
+ set -o pipefail
+ export PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin
+ PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin
+ readonly spool_old=/var/spool/mail
+ spool_old=/var/spool/mail
+ readonly spool_new=/persistent-storage/mail
+ spool_new=/persistent-storage/mail
+ [[ -L /var/spool/mail ]]
+ exit 0
>>> email_persistency.sh >>>>>>>>>>>>>>>>>>>>>>>>>>>>
exit_code=0, duration=0

+ subshell install_sendmail.sh

<<< install_sendmail.sh <<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

Logging script

/usr/local/bin/subshell.sh

```
function subshell {

  echo
  echo "<<< $@ <<<<<<<<<<<<<<<<<<<<<<<<<<<<"
  local begin=$(date +%s)
  $@
  status="$?"
  local end=$(date +%s)
  local duration=$(($end - $begin))
  echo ">>> $@ >>>>>>>>>>>>>>>>>>>>>>>>>>>>"
  echo "exit_code=$status, duration=$duration"
  echo

  # return the exit code of the script
  # but don't pollute logs by return statement
  {
    return "$status"
  } 2> /dev/null

}
```

/usr/local/bin/bootstrap.sh

```
#!/bin/bash

source subshell.sh

for layer-bootstrapper in
/usr/local/deploy-bootstrap/*.sh
do
  subshell "$layer-bootstrapper"
done
```

Name your scripts well

I mean bootstrap scripts.

Name your scripts so that you know exactly where they originate from. It helps to quickly pinpoint scripts, layers, repos. Also makes log reading easier. Script name should contain:

- A number which determines if the script originates from common deployment repo or from application repo
- A number which specifies cardinality
- A layer name

Example: `11-env-stg.sh`

Shell scripting

- Follow Google Shell Style Guide
  https://google.github.io/styleguide/shell.xml
- Start every script by these lines (verbatim):

```
#!/bin/bash

set -xeuo pipefail
export PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin
```

- Learn to quote everything by single quotes, double quotes and heredoc.
- Don’t put scripts into deploy files. Managing is hard.

Afterglow

- How does the result feel like?
- Considerations?
- Pros and cons?
- Experience of transfer of services: dev, stg, prod.
- Are we still happy with the result?
- How did you handle tasks before deploy repo?
- Did it pay off?
- Do we version deploy repo? Branching?
- How about accidental bug in deploy repo?
- How about adding intentional new feature?
- But but how do you … ?

SPECIAL THANKS TO OUR SUPPORTERS

cēsis

VALMIERA

valsts #196

1862 RTU CĒSU FILIĀLE

SUN GIS

ECHO TECH

TestDevLab

Kebbe it

Share IT

ORGANIZERS

SKOLA6

wunder

VALMIERAS ATTĪSTĪBAS AĢENTŪRA