Better Bash

Unit and Integration Testing
C.J. Jameson

@cjcjameson

Pivotal

I work at Pivotal, I’m a Toolsmith, I tweet some

@cjcjameson

Customer = Engineer

Reliability & predictable outcomes
Good bug reports
Eager to make tradeoffs

Tradeoffs?

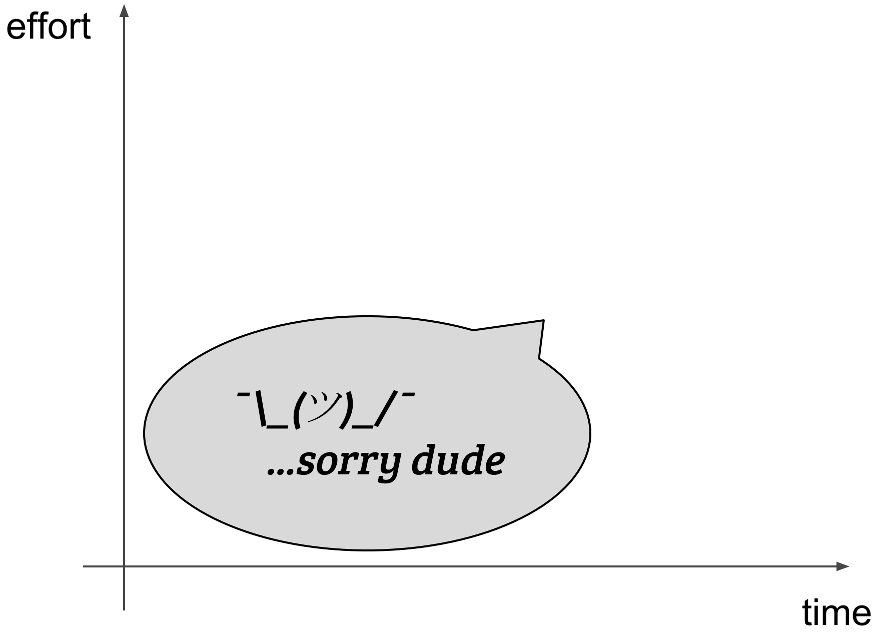
Customization vs. Reuse

User experience vs. Exposing internals

YAGNI

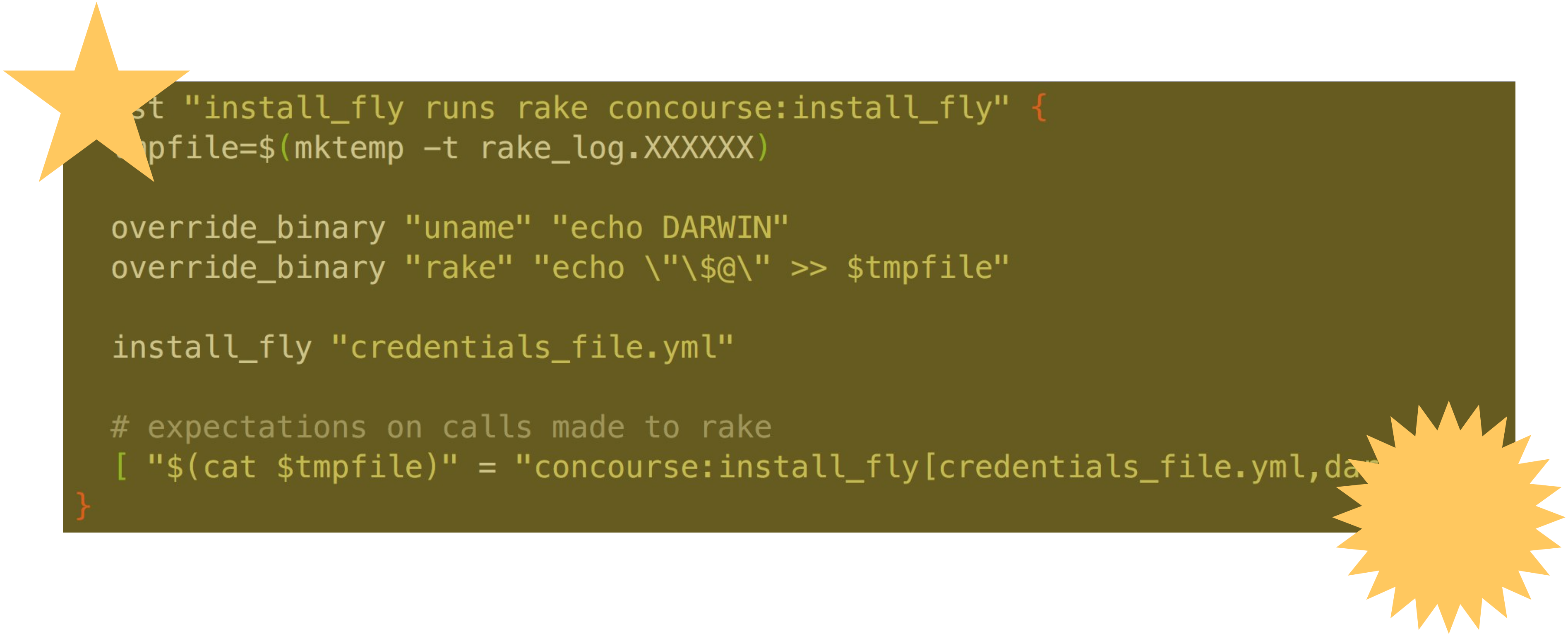
The tradeoff with your Bash scripts

How good does this need to be?



Pivotal

Here's my shiny tool…?



```
st "install_fly runs rake concourse:install_fly" {
 pfile=$(mktemp -t rake_log.XXXXXX)

  override_binary "uname" "echo DARWIN"
  override_binary "rake" "echo \"\$@\" >> $tmpfile"

  install_fly "credentials_file.yml"

  # expectations on calls made to rake
  [ "$(cat $tmpfile)" = "concourse:install_fly[credentials_file.yml,da
}
```

@cjcjameson

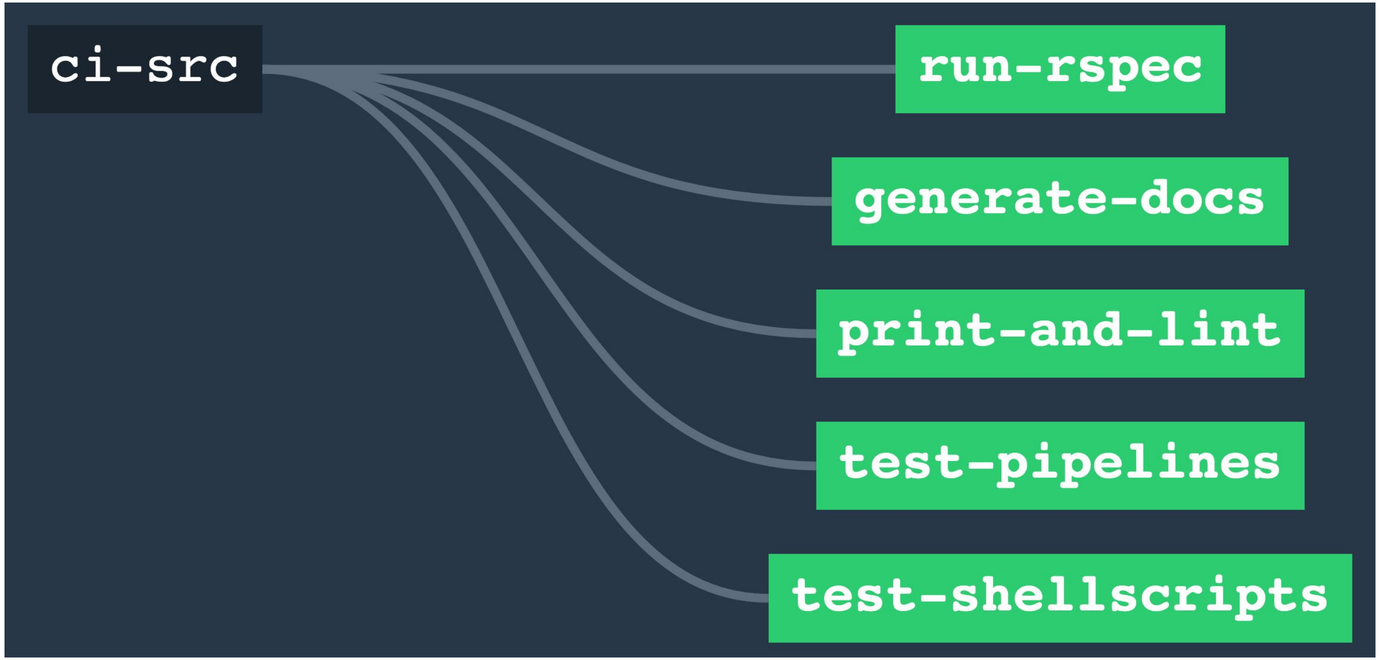
Pivotal

When would anyone write a test themselves?

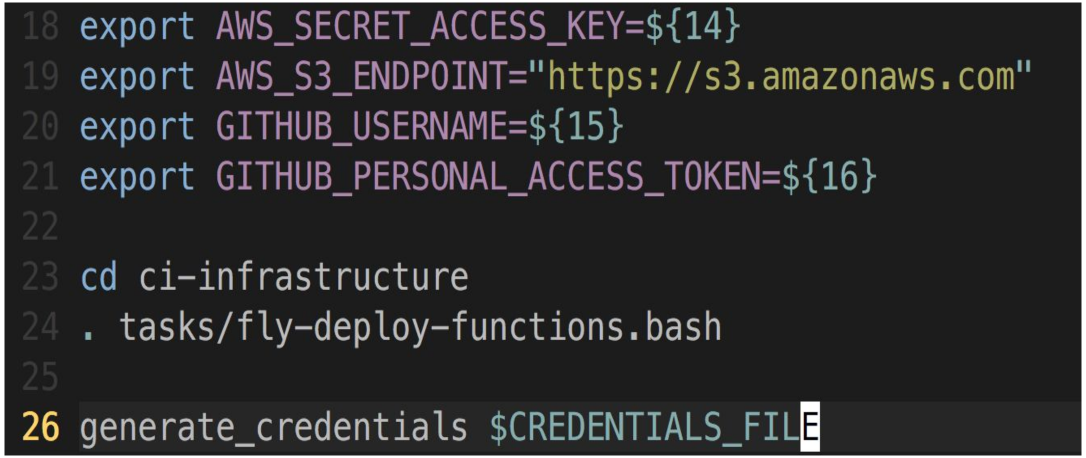
When would anyone ~~write~~ a test themselves?

When would anyone run the tests?

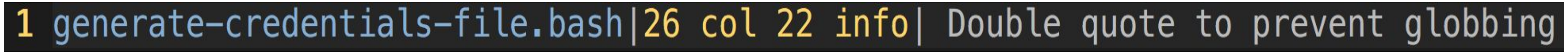
When would anyone run the tests?



When would anyone run the tests?



```
18 export AWS_SECRET_ACCESS_KEY=${14}
19 export AWS_S3_ENDPOINT="https://s3.amazonaws.com"
20 export GITHUB_USERNAME=${15}
21 export GITHUB_PERSONAL_ACCESS_TOKEN=${16}
22
23 cd ci-infrastructure
24 . tasks/fly-deploy-functions.bash
25
26 generate_credentials $CREDENTIALS_FILE
```



```
1 generate-credentials-file.bash|26 col 22 info| Double quote to prevent globbing
```

In general, unit tests...[1]

Find problems early

Make change easy

Make integration easy

Are documentation

Drive modular design

[1]https://en.wikipedia.org/wiki/Unit_testing

You’re writing an installer file

So your unit tests should help:

Make integration easy (esp. across distros)

Be documentation

You’re touching this code for the first and only time

So your unit tests should help:

Make change easy

Drive modular design

Be documentation

Pivotal

You’re treating the filesystem as a first-class object

So your unit tests should help:

Find problems early

Drive modular design

You’re writing a shared `common.bash` library

So your unit tests should help:

Make integration easy

Be documentation

You’re crafting a CLI user experience

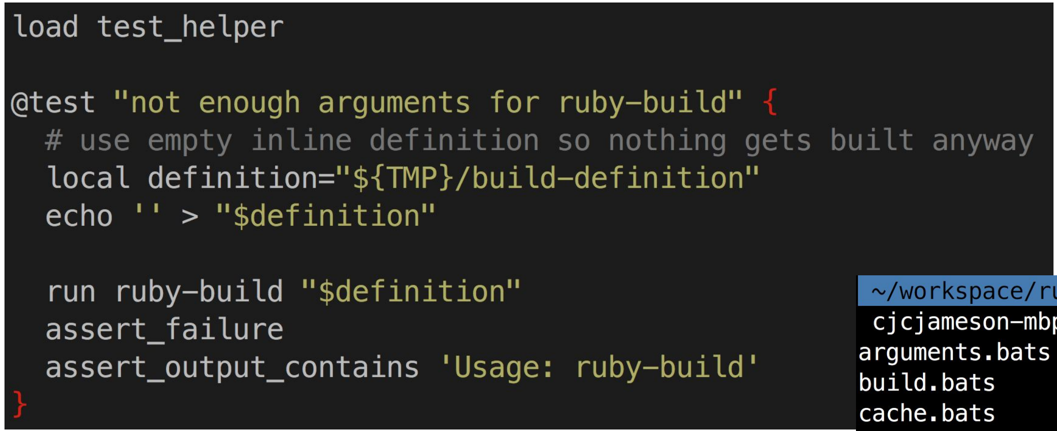
So your unit tests should help:

Find problems early

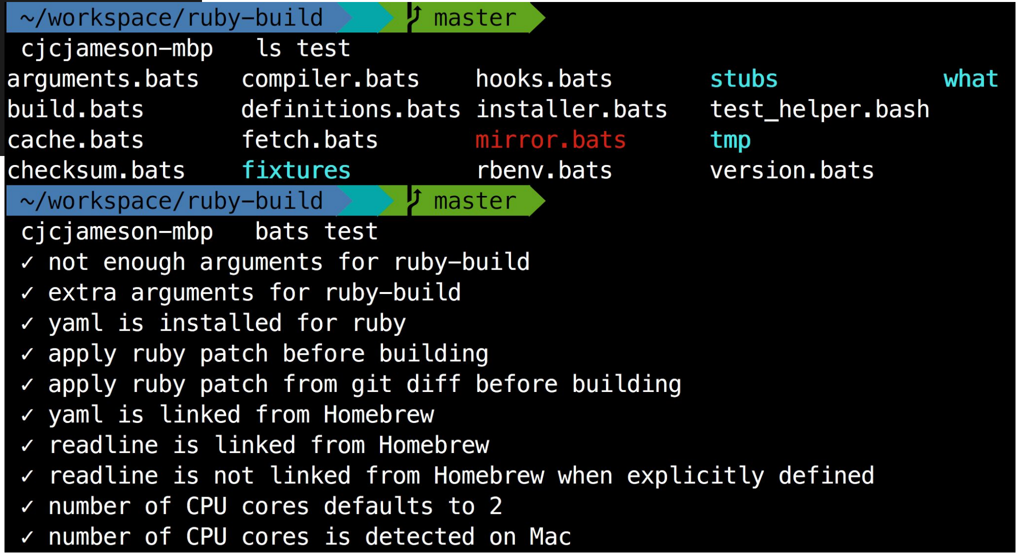
Make change easy

Pivotal

Example: ruby-build



```
load test_helper

@test "not enough arguments for ruby-build" {
  # use empty inline definition so nothing gets built anyway
  local definition="${TMP}/build-definition"
  echo '' > "$definition"

  run ruby-build "$definition"
  assert_failure
  assert_output_contains 'Usage: ruby-build'
}
```



```
 ~/workspace/ruby-build   master
 cjcjameson-mbp   ls test
arguments.bats    compiler.bats    hooks.bats       stubs            what
build.bats        definitions.bats installer.bats   test_helper.bash
cache.bats        fetch.bats       mirror.bats      tmp
checksum.bats     fixtures         rbenv.bats       version.bats
 ~/workspace/ruby-build   master
 cjcjameson-mbp   bats test
 ✓ not enough arguments for ruby-build
 ✓ extra arguments for ruby-build
 ✓ yaml is installed for ruby
 ✓ apply ruby patch before building
 ✓ apply ruby patch from git diff before building
 ✓ yaml is linked from Homebrew
 ✓ readline is linked from Homebrew
 ✓ readline is not linked from Homebrew when explicitly defined
 ✓ number of CPU cores defaults to 2
 ✓ number of CPU cores is detected on Mac
```

Pivotal

Example: Concourse’s `git-resource`

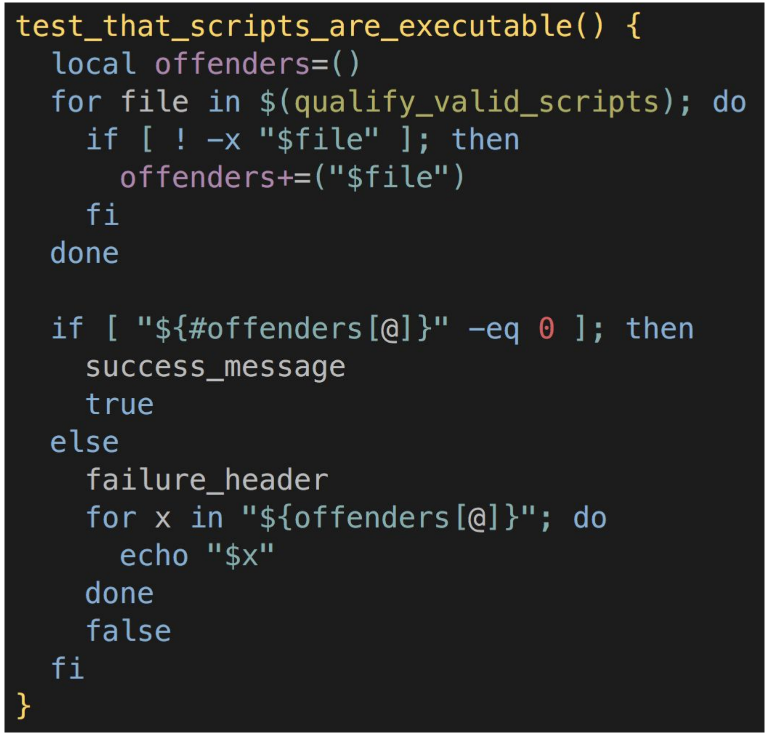
```
it_honors_the_depth_flag() {
  local repo=$(init_repo)
  local firstCommitRef=$(make_commit $repo)

  make_commit $repo

  local lastCommitRef=$(make_commit $repo)

  local dest=$TMPDIR/destination

  get_uri_at_depth "file://"$repo 1 $dest |  jq -e "
    .version == {ref: $(echo $lastCommitRef | jq -R .)}
  "

  test "$(git -C $dest rev-parse HEAD)" = $lastCommitRef
  test "$(git -C $dest rev-list --all --count)" = 1
}
```

```
run() {
  export TMPDIR=$(mktemp -d ${TMPDIR_ROOT}/git-tests.XXXXXX)

  echo -e 'running \e[33m'"$@"$'\e[0m...'
  eval "$@" 2>&1 | sed -e 's/^/  /g'
  echo ""
}

init_repo() {
  (
    set -e

    cd $(mktemp -d $TMPDIR/repo.XXXXXX)

    git init -q
```

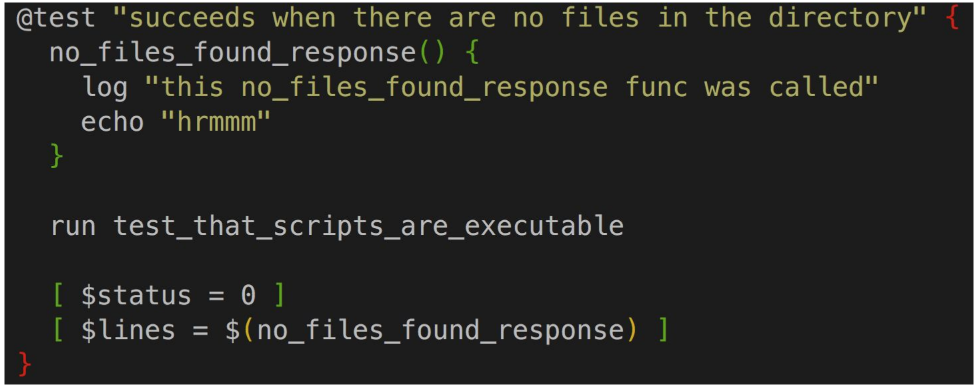
```
running it_honors_the_depth_flag...
  Switched to a new branch 'bogus'
  Switched to branch 'master'
  8d95f355e0f093f54a49116557c186b7c8480bbd
  Cloning into '/tmp/git-tests.HMiqIi/git-tests.85LJmS/destination'...
  Fetching master
  1e7cc29 commit 3 /tmp/git-tests.HMiqIi/git-tests.85LJmS/repo.GjEujS/some-file
  {
    "version": {
      "ref": "1e7cc29ebf60938e05b8d10e4a891fd48e74dcd3"
    },
    "metadata": [
      {
        "name": "commit",
        "value": "1e7cc29ebf60938e05b8d10e4a891fd48e74dcd3"
      },
      {
        "name": "author",
```

https://ci.concourse.ci/pipelines/resources/jobs/git-resource

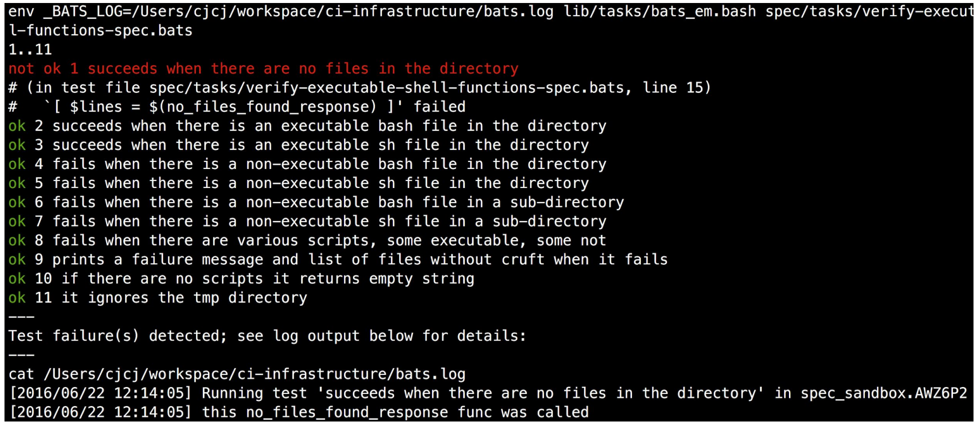
Example: “Check the executable bit”



```
test_that_scripts_are_executable() {
  local offenders=()
  for file in $(qualify_valid_scripts); do
    if [ ! -x "$file" ]; then
      offenders+=("$file")
    fi
  done

  if [ "${#offenders[@]}" -eq 0 ]; then
    success_message
    true
  else
    failure_header
    for x in "${offenders[@]}"; do
      echo "$x"
    done
    false
  fi
}
```



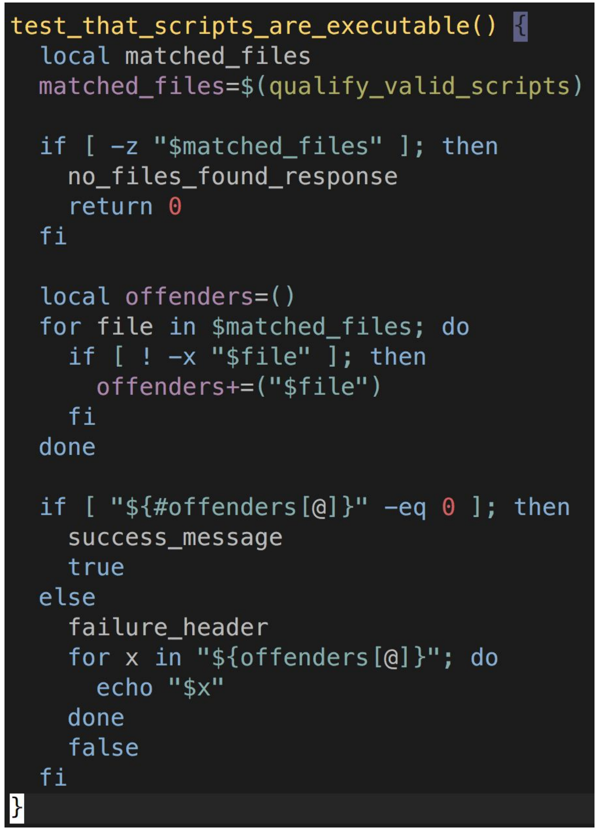
```
@test "succeeds when there are no files in the directory" {
  no_files_found_response() {
    log "this no_files_found_response func was called"
    echo "hrmmm"
  }

  run test_that_scripts_are_executable

  [ $status = 0 ]
  [ $lines = $(no_files_found_response) ]
}
```



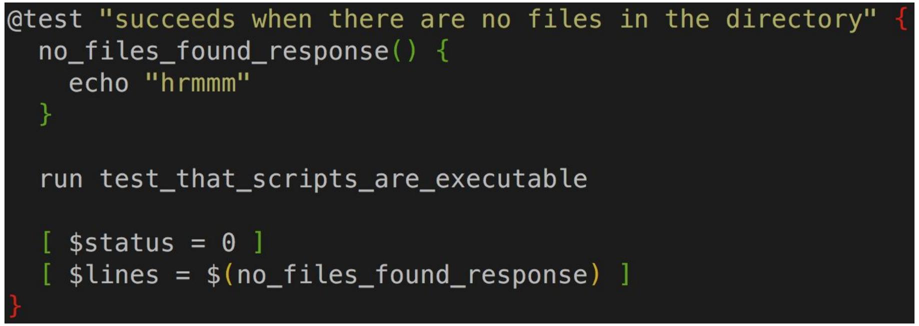
```
env _BATS_LOG=/Users/cjcj/workspace/ci-infrastructure/bats.log lib/tasks/bats_em.bash spec/tasks/verify-execut
l-functions-spec.bats
1..11
not ok 1 succeeds when there are no files in the directory
# (in test file spec/tasks/verify-executable-shell-functions-spec.bats, line 15)
#   `[ $lines = $(no_files_found_response) ]' failed
ok 2 succeeds when there is an executable bash file in the directory
ok 3 succeeds when there is an executable sh file in the directory
ok 4 fails when there is a non-executable bash file in the directory
ok 5 fails when there is a non-executable sh file in the directory
ok 6 fails when there is a non-executable bash file in a sub-directory
ok 7 fails when there is a non-executable sh file in a sub-directory
ok 8 fails when there are various scripts, some executable, some not
ok 9 prints a failure message and list of files without cruft when it fails
ok 10 if there are no scripts it returns empty string
ok 11 it ignores the tmp directory
---
Test failure(s) detected; see log output below for details:
---
cat /Users/cjcj/workspace/ci-infrastructure/bats.log
[2016/06/22 12:14:05] Running test 'succeeds when there are no files in the directory' in spec_sandbox.AWZ6P2
[2016/06/22 12:14:05] this no_files_found_response func was called
```

Pivotal

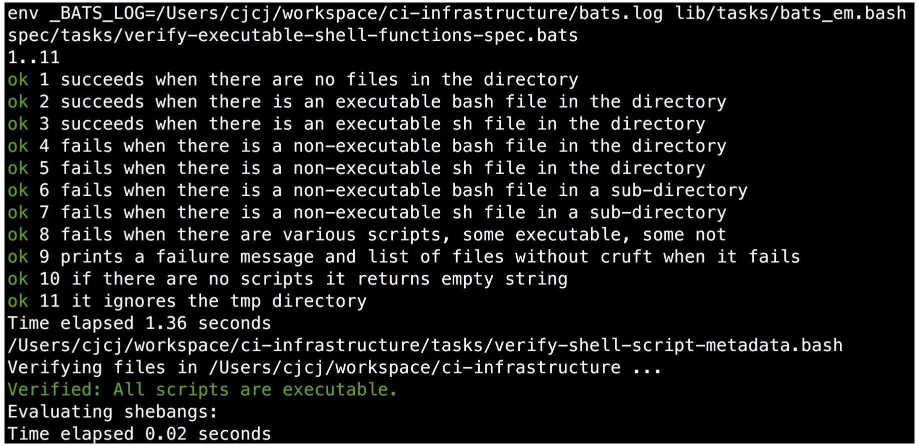
Example: “Check the executable bit”



```
test_that_scripts_are_executable() {
  local matched_files
  matched_files=$(qualify_valid_scripts)

  if [ -z "$matched_files" ]; then
    no_files_found_response
    return 0
  fi

  local offenders=()
  for file in $matched_files; do
    if [ ! -x "$file" ]; then
      offenders+=("$file")
    fi
  done

  if [ "${#offenders[@]}" -eq 0 ]; then
    success_message
    true
  else
    failure_header
    for x in "${offenders[@]}"; do
      echo "$x"
    done
    false
  fi
}
```

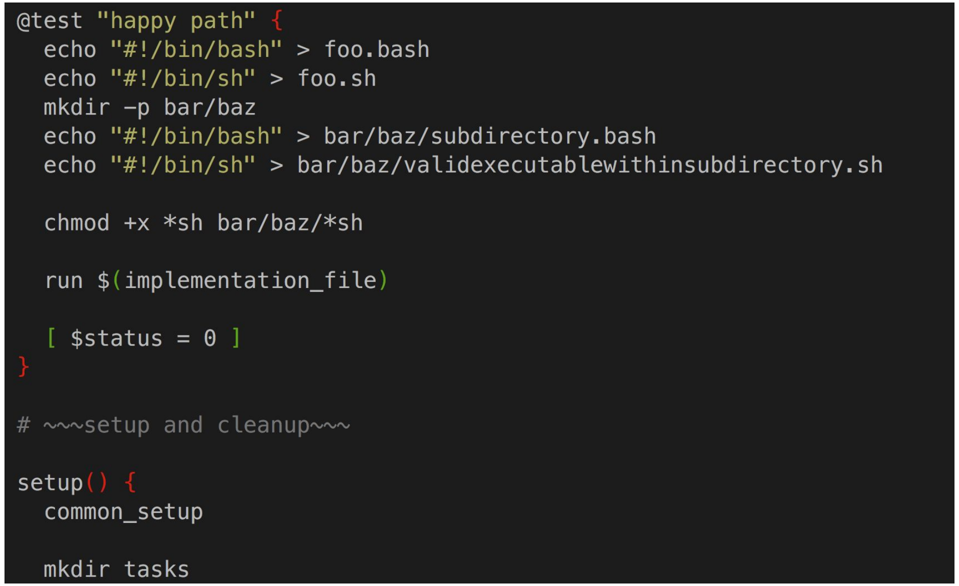


```
@test "succeeds when there are no files in the directory" {
  no_files_found_response() {
    echo "hrmmm"
  }

  run test_that_scripts_are_executable

  [ $status = 0 ]
  [ $lines = $(no_files_found_response) ]
}
```

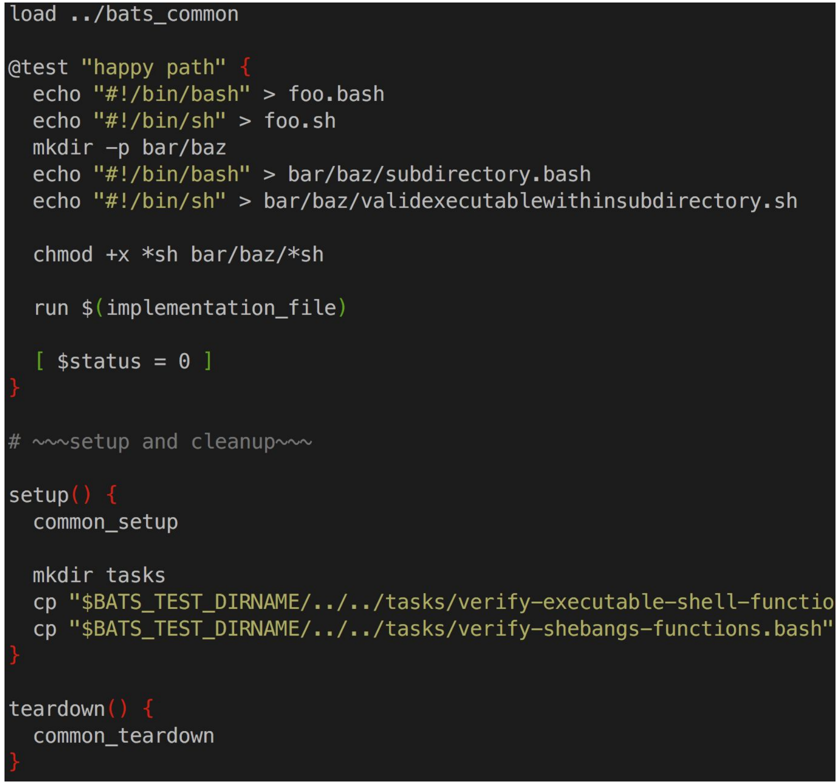


```
env _BATS_LOG=/Users/cjcj/workspace/ci-infrastructure/bats.log lib/tasks/bats_em.bash
spec/tasks/verify-executable-shell-functions-spec.bats
1..11
ok 1 succeeds when there are no files in the directory
ok 2 succeeds when there is an executable bash file in the directory
ok 3 succeeds when there is an executable sh file in the directory
ok 4 fails when there is a non-executable bash file in the directory
ok 5 fails when there is a non-executable sh file in the directory
ok 6 fails when there is a non-executable bash file in a sub-directory
ok 7 fails when there is a non-executable sh file in a sub-directory
ok 8 fails when there are various scripts, some executable, some not
ok 9 prints a failure message and list of files without cruft when it fails
ok 10 if there are no scripts it returns empty string
ok 11 it ignores the tmp directory
Time elapsed 1.36 seconds
/Users/cjcj/workspace/ci-infrastructure/tasks/verify-shell-script-metadata.bash
Verifying files in /Users/cjcj/workspace/ci-infrastructure ...
Verified: All scripts are executable.
Evaluating shebangs:
Time elapsed 0.02 seconds
```

Example: “Check the executable bit” - integration



```
@test "happy path" {
  echo "#!/bin/bash" > foo.bash
  echo "#!/bin/sh" > foo.sh
  mkdir -p bar/baz
  echo "#!/bin/bash" > bar/baz/subdirectory.bash
  echo "#!/bin/sh" > bar/baz/validexecutablewithinsubdirectory.sh

  chmod +x *sh bar/baz/*sh

  run $(implementation_file)

  [ $status = 0 ]
}

# ~~~setup and cleanup~~~

setup() {
  common_setup

  mkdir tasks
```



```
load ../bats_common

@test "happy path" {
  echo "#!/bin/bash" > foo.bash
  echo "#!/bin/sh" > foo.sh
  mkdir -p bar/baz
  echo "#!/bin/bash" > bar/baz/subdirectory.bash
  echo "#!/bin/sh" > bar/baz/validexecutablewithinsubdirectory.sh

  chmod +x *sh bar/baz/*sh

  run $(implementation_file)

  [ $status = 0 ]
}

# ~~~setup and cleanup~~~

setup() {
  common_setup

  mkdir tasks
  cp "$BATS_TEST_DIRNAME/../../tasks/verify-executable-shell-functio
  cp "$BATS_TEST_DIRNAME/../../tasks/verify-shebangs-functions.bash"
}

teardown() {
  common_teardown
}
```

@cjcjameson

Pivotal

Inside the BATS `run` command

https://github.com/sstephenson/bats

- Exit codes are swallowed
- `stdout` and `stderr` are swallowed
- Start by `source`ing in the file under test … so the file runs

Pivotal

Thanks!

@cjcjameson

Pivotal