baut Documentation

Release 0.1-beta

haikikyou

Dec 28, 2017

Table Of Contents

#!Baut

Baut (Bash Unit test Tool) is a unit testing tool runs on Bash and helps you to verify that the programs on Unix/Linux behave as expected.

Here is a example.

test_sample.sh

```
#: @BeforeAll
function setup_all() {
  echo "==> Called once at first"
}

#: @BeforeEach
function setup() {
  export PATH=/usr/local/bin:"$PATH"
}

#: @Test(The usage should be displayed when command line options are invalid)
function parse_cli_options() {
  run ./my.sh
  [[ "$result" =~ usage: ]]
}

#: @Test
#: @Ignore
function this_test_is_ignored() {
  echo "This test is ignored"
}

#: @AfterEach
function teardown() {
  echo "# Clean up a test."
}

#: @AfterAll
function teardown_all() {
  echo "==> Called once at last"
}
```

OK, we run tests.

```
$ baut run test_sample.sh
1 file, 1 test
[1] /Users/guest/workspace/baut/test_sample.sh
==> Called once at first
o The usage should be displayed when command line options are invalid
  # Clean up a test.
==> Called once at last
1 test, 1 ok, 0 failed, 0 skipped
```

```
1 file, 1 test, 1 ok, 0 failed, 0 skipped
Time: 0 hour, 0 minute, 0 second
```

CHAPTER 1

Introduction

Baut is a simple tool for unit test written with Bash. Baut helps you to verify that the programs on Unix/Linux behave as expected. Test file is just a Bash script, so you can write test code with your favorite editor as always. Baut does not need a special environment, **only Bash** is required.

Note: Baut uses features available only in newer version than Bash 4. If Bash on your machine is earlier than 4, you need to upgrade Bash to newer version than 4.

1.1 Test Code

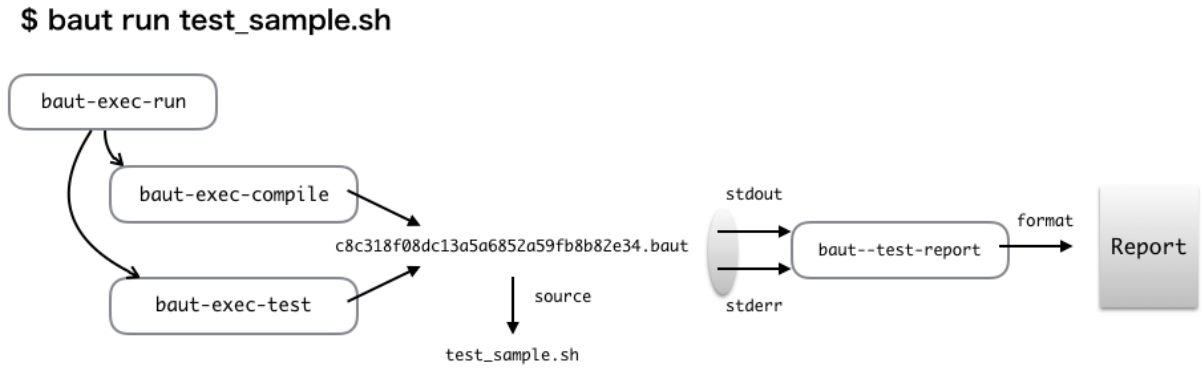
You can write test code with Bash on style like xUnit. Test file with Baut is the following:

```
#: @BeforeEach
setup() {
  touch flagfile
}

#: @Test
script_should_return_error_code_1() {
  run myscript.sh
  [ $status -eq 1 ]
}

#: @AfterEach
teardown() {
  rm flagfile
}
```

As described above, you can use conditional expressions with `[` or `[[` command to verify that the result of commands equals your expecting result. `[` and `[[` are buildin commands of Bash. You have only to write conditional expressions with them.

`@Test` is a directive and it means that the following function is a test. `@BeforeEach` means that the following function `setup` is a setup function and `setup` is called at the first of a test. The function `teardown` written after `@AfterEach` is a cleanup function and `teardown` is called at the last of a test.

1.2 How it works?



At first, Baut parses the specified test files with `run` command and compiles them. Compilation is done for each file and compiled source files are also just Bash scripts. Next Baut executes the compiled source files and generates a test report. `baut-exec-test` script is in charge of test execution and `baut--test-report` script is in charge of generation of a test report. `baut-exec-test` writes output into the standard output and error output. Then `baut--test-report` receives output lines from standard input and convert to a test report format.

Baut runs tests on Bash with `-E` and `-u` options, uses `ERR` trap to detect whether there was a error in your test. So if you were to set `ERR` trap in your test scripts, Baut might not behave normally.

CHAPTER 2

Installation

Installation does not need anything special at all. Download source files from github and put it into an arbitary directory.

```
$ cd /path/to/a_directory
$ git clone https://github.com/moritoru81/baut
```

Set the path to `baut` script to `PATH`.

```
$ cat <<EOS >> ~/.bash_profile
export PATH="$(pwd)/baut/bin:\$PATH"
EOS
$ source ~/.bash_profile
```

You can also run `install.sh` to do it.

```
$ source baut/install.sh
```

Run `baut` for confirmation of installation, then you will see the usage of Baut.

```
$ baut help
```

CHAPTER 3

Writing Tests

A test is just a shell function. You can write multiple tests in a file. A test file is included by Baut and tests are executed in their written order in Baut's test running process.

The name of a test file must start with `test_` and end with `.sh`. Baut regards a file like `test_a.sh` as a test file.

3.1 Quick Start

```
$ mkdir test && cd test
$ baut init
$ ./run-test.sh
1 file, 3 tests
#1 /Users/guest/workspace/baut/test/test_sample.sh
x test_ng_sample
Not implemented
# Error(1) detected at the following:
#       13   #: @Test
#       14   test_ng_sample() {
#=>     15     fail "Not implemented"
#       16   }
#       17
o test_ok_sample
~ test_skip_sample # SKIP Good bye!
3 tests, 1 ok, 1 failed, 1 skipped

  1 file, 3 tests, 1 ok, 1 failed, 1 skipped
Time: 0 hour, 0 minute, 0 second
```

3.2 Test Function

To tell Baut that a shell function is a test function, you need to learn the following rules.

1. The function which name starts with `test_`.
2. `@Test` annotation before a function definition.

This example shows you how to write a unit test.

```
# (1) This is a test with 'test_' prefix.
function test_mytest() {
  run echo "mytest"
  [ "$result" = "mytest" ]
}

# (2) This is a test with 'test_' prefix.
# function keyword is not required.
test_mytest2() {
  run echo "mytest2"
  [ "$result" = "mytest2" ]
}

# (3) This is a test with '@Test' annotation.
#: @Test(mytest3 should be ok)
mytest3() {
  run echo "mytest3"
  [ "$result" = "mytest3" ]
}
```

All functions of the above are tests. Tests are executed in written order in a file.

3.3 Test Context

Each test runs in subshell. This means that you cannot read variables or functions defined in other tests.

Here is a example.

```
test_1() {
  MY_VAR=1
  [ $MY_VAR -eq 1 ]
}

test_2() {
  [ $MY_VAR -eq 1 ] # This test should be failed.
}
```

`MY_VAR` defined in `test_1` cannot be read from `test_2`, and `test_2` will fail.

To set up a test and clean up a test, you can use `@BeforeEach` and `@BeforeAfter` annotations. The functions specified with these annotations are executed before a test starts or after a test ends.

```
#: @BeforeEach
setup() {
  MY_VAR=1
  echo "hello" > flagfile
}

test_1() {
  run cat flagfile
  [ $MY_VAR -eq 1 ]
  [ "$result" = "hello" ]
```

```
}

test_2() {
  run cat flagfile
  [ $MY_VAR -eq 1 ]
  [ "${lines[0]}" = "hello" ]
}

#: @AfterEach
teardown() {
  MY_VAR=1
  rm flagfile
}
```

`setup` function is executed in the same context as `test_1` and `test_2`, so `MY_VAR` defined in `setup` is visible from `test_1` and `test_2`. `setup` and `teardown` functions are called for each test.

There may be when you want to read variables from all tests, in that case you can use `@BeforeAll` or `@AfterAll` annotations. Variables, which are defined in the functions specified with these annotations, can be read from all test functions.

```
EVALUATED_ONCE="var"

#: @BeforeAll
setup_all() {
  GLOBAL_VAR="global"
}

test_3() {
  [ "$GLOBAL_VAR" = "global" ]
}

#: @AfterAll
teardown_all() {
  : # Nothing
}
```

`setup_all` function with `@BeforeAll` annotation is called only once before all tests start, and `teardown_all` function with `@AfterAll` annotation is called only once after all tests ends. These functions are executed in parent shell of tests, `GLOBAL_VAR` is visible from all tests. Outside of functions, `EVALUATED_ONCE` is also evaluated once with `source` command.

3.4 Commands

3.4.1 run

```
run <command>
```

`run` executes the specified command in subshell. You can get its output with `$result`, and get the exit status code with `$status`. And also you can use `$lines`, you can access each line with `${lines[0]}`.

```
test_run() {
  run echo "hoge"
  [ "$result" = "hoge" ]
  [ $status -eq 0 ]
```

```
  [ "${lines[0]}" = "hoge" ]
}
```

3.4.2 run2

```
run2 <command>
```

`run2` executes the specified command in subshell as `run`, but you can separately get its output with `$stdout` and `$stderr`. Then the exit status code can be read with `$status`. If you separately handle each line of output, you can access each line with `${stdout_lines[0]}` or `${stderr_lines[0]}`.

This is a small script.

```
# hello.sh
echo "hello"
echo "world" >&2
```

You can use `run2` as the following.

```
test_run() {
  run2 ./hello.sh
  [ "$stdout" = "hello" ]
  [ "${stdout_lines[0]}" = "hello" ]
  [ $status -eq 0 ]
  [ "$stderr" = "world" ]
  [ "${stderr_lines[0]}" = "world" ]
}
```

3.4.3 eval2

```
eval2 <command>
```

`eval2` executes the specifiled commans with `eval` command. You can get output or exit status code as `run2`.

```
test_eval2() {
  eval2 'echo "hello" >&2'
  [ $status -eq 0 ]
  [ "$stdout" = "" ]
  [ "$stderr" = "hello" ]
  [ "${stderr_lines[0]}" = "hello" ]
}
```

3.4.4 fail

```
fail [<text>]
```

`fail` makes a test fail.

```
test_fail() {
  fail "Not implemented"
}
```

3.4.5 skip

```
skip [<text>]
```

`skip` skips the rest codes after it.

```
test_skip() {
  if [ -e flagfile ]; then
    skip "found flagfile, so we skip."
  fi
  echo "If flagfile exists, not reach here."
}
```

3.4.6 wait_until

```
wait_until [-i|--interval <sec>] [-m|--retry-max <count>] <command>
```

Retry until the command ends successfully.

```
test_wait_until() {
  run myapp.sh start
  wait_until --retry-max 3 "[ -e my.pid ]"
  run myapp.sh stop
  wait_until --retry-max 3 "[ ! -e my.pid ]"
}
```

3.5 Annotations

An annotation line needs to start with `#:`, `#` is interpreted just as a comment.

3.5.1 @BeforeAll

```
#: @BeforeAll
```

A function with `@BeforeAll` is executed **only once** before all tests start. You can specify this annotation for multiple functions, and those functions will be executed in written order.

```
# (1)
#: @BeforeAll
setup_all1() {
  GLOBAL_VAR1=10
}

# (2)
#: @BeforeAll
setup_all2() {
  export PATH=/usr/local/bin:"$PATH"
}
```

3.5.2 @BeforeEach

```
#: @BeforeEach
```

A function with `@BeforeEach` is executed before a test starts, the function is called **for each test**. You can specify this annotation for multiple functions, and those functions will be executed in written order.

```
#: @BeforeEach
setup1() {
  touch flagfile
}

#: @BeforeEach
setup2() {
  TEST_VAR2=20
}
```

3.5.3 @Test

```
#: @Test[(<text>)]
```

A function with `@Test` is regarded as a test. You can also tell Baut by writing a function name starts with `test_`. If you write `<text>` after `@Test` annotation, the text will be displayed as a test name in a test report.

```
#: @Test(This test should be absolutely passed)
test_passed() {
  [ 1 -eq 1 ]
}
```

Here is the result.

```
$ baut run test_sample.sh
1 file, 1 test
#1 /Users/guest/workspace/baut/test_hoge.sh
o This test should be absolutely passed
1 test, 1 ok, 0 failed, 0 skipped

1 file, 1 test, 1 ok, 0 failed, 0 skipped
Time: 0 hour, 0 minute, 0 second
```

3.5.4 @TODO

```
#: @TODO[(<text>)]
```

A function with `@TODO` is regarded as a test. If you write `<text>` after `@TODO` annotation, a result of a test will be displayed with `# TODO <text>` tag in a test report.

3.5.5 @Ignore

```
#: @Ignore
```

A test function with `@Ignore` is absolutelly ignored.

3.5.6 @Deprecated

```
#: @Deprecated[(<text>)]
```

A function with `@Deprecated` is regarded as a test. If you write `<text>` after `@Deprecated` annotation, a result of a test will be displayed with `# DEPRECATED <text>` tag in a test report.

3.5.7 @AfterEach

```
#: @AfterEach
```

A function with `@AfterEach` is executed after a test ends, the function is called **for each test**. You can specify this annotation for multiple functions, and those functions will be executed in written order.

```
#: @AfterEach
teardown() {
  rm flagfile ||:
}
```

3.5.8 @AfterAll

```
#: @AfterAll
```

A function with `@AfterAll` is executed **only once** after all tests ends. You can specify this annotation for multiple functions, and those functions will be executed in written order.

```
#: @AfterAll
teardown_all() {
  rm "$TMPDIR/*.tmp" ||:
}
```

3.6 Common Variables

`BAUT_TEST_FUNCTION_NAME`

`BAUT_TEST_FILE`

`BAUT_TEST_FUNCTIONS`

`before_all_functions` (Array)

`before_each_functions` (Array)

`after_all_functions` (Array)

`after_each_functions` (Array)

3.7 Other APIs

3.7.1 load

load <file> [<arg> . . .]

Loads the file with the specified arguments. This calls `source` command internally. If the file does not exist, it will abort. You can load the file multiple times.

load_if_exists <file> [<arg> . . .]

Loads the file with the specified arguments. This calls `source` command internally. If the file does not exist, it will return `1`. You can load the file multiple times.

require <file> [<arg> . . .]

Loads the file with the specified arguments. This calls `source` command internally. If the file does not exist, it will abort. You can load the file multiple times, but if the file has already been loaded, it will not be loaded again.

```
# Load configurations.
load "conf.sh" "arg1"
# At first, load optional settings. But if it does not be found, we load default␣
↪settings.
load_if_exists "options.sh" || load "default.sh"
# Load 'mylib' only once.
require "mylib.sh"
```

3.7.2 log

These functions can be used for debug, and you can control which level of message is output with `--d[0-4]` option or `BAUT_LOG_LEVEL` variable.

Syntax

```
log_trace <text>
log_debug <text>
log_info <text>
log_warn <text>
log_error <text>
```

Examples

```
log_trace "Level trace"
log_debug "Level debug"
log_info  "Level info"
log_warn  "Level warn"
log_error "Level error"
```

Here is a example in a test.

```
# test_log.sh
test_log() {
  run echo "sample"
  if [ $status -eq 0 ]; then
    log_info "status code is ok."
  else
    log_error "status code is not ok."
    fail
  fi
}
```

You can run tests with `--d[0-4]` log option, and this option must be put before `run` command.

```
$ baut --d1 run test_log.sh
1 file, 1 test
#1 /Users/guest/workspace/baut/test_log.sh
o test_log
2017-10-01 00:30:10 [INFO] test_log.sh:4 - status code is ok.
1 test, 1 ok, 0 failed, 0 skipped

1 file, 1 test, 1 ok, 0 failed, 0 skipped
Time: 0 hour, 0 minute, 0 second
```

3.7.3 trap

add_trap_callback <signame> <command>

Adds a command to a callback chain of signame. The function added later is executed first. In this example, `rm flagfile` is executed, and then `echo "done"`.

```
add_trap_callback "EXIT" echo "done"
add_trap_callback "EXIT" rm flagfile
```

reset_trap_callback [<signame> . . .]

Removes existing commands from callback chains of the specified signals. This function removes commands. but does not remove the already registered trap entries.

```
reset_trap_callback "EXIT" "ERR"
```

register_trap_ballback [<signame> . . .]

Registers traps of the specified signals. This function is usually used with `add_trap_callback` function.

```
add_trap_callback "EXIT" rm "flagfile"
register_trap_callback "EXIT"
```

unregister_trap_ballback [<signame> . . .]

Unregisters traps of the specified signals.

```
unregister_trap_callback "EXIT"
```

enable_trap [<signame> . . .]

Enables traps of the specified signals. This function just switches on/off of trap, the existing trap commands remain.

disable_trap [<signame> . . .]

Disables traps of the specified signals. This function just switches on/off of trap, the existing trap commands remain.

```
disable_trap "ERR"
{
  echo "do something"
}
enable_trap "ERR"
```

Note: DO NOT use `ERR` or `EXIT` by built-in 'trap' command in your test scripts. You can use `add_trap_callback` instead.

3.7.4 Others

hash_get <key> [<key> . . .]

Returns the value with the specified keys.

hash_set <key> [<key> . . .] <value>

Sets the value with the specified keys.

hash_delete <key> [<key> . . .]

Deletes the entry with the specified keys.

```
hash_set "namespace" "key" "value"
hash_get "namespace" "key" #=> value
hash_delete "namespace" "key"
```

get_comment_block <file> <ident>

Extracts the comment block with the specified ident from the file.

```
# test_my.sh
get_comment_block "$(__FILE__)" "HELP"  #=> This is a help comment.

#=begin HELP
#
# This is a help comment.
#
#=end HELP
```

self_comment_block <ident>

Extracts the comment block with the specified ident from the written file.

```
# test_my.sh
self_comment_block "HELP"  #=> This is a help comment.

#=begin HELP
#
# This is a help comment.
#
#=end HELP
```

CHAPTER 4

Running Tests

To run tests, you need to execute `run` command with files or directories.

```
$ baut run <file or directory> [<file or directory> ...]
```

`run` takes files or directories and executes them in order. If directories are passed, Baut will find test files under the directories. With `-r` option, Baut finds test target files recursively under the directories.

```
$ baut run -r directory
```

4.1 Commands

4.1.1 run(r)

You can run tests with `run` command, and its command takes some options.

`-r, --recursively`

Baut finds test target files recursively under the specified directories.

`-s, --stop-on-error`

Baut stops test process when a test fails.

`--no-color`

A test report will not be colored.

`--no-checksum`

Baut does not verify checksum of compiled test files.

`-f, --format [oneline|default|tap|cat]`

You can get a test report with various formats with this option. Default is `default`. `tap` is Test Anything Protocol. (For more detail, see https://testanything.org)

`-m, --match <regex>`

Executes only functions that match the specified value.

```
$ baut run -m "option" test_command.sh
```

`-i, --interactive`

Run tests in interactive mode.

4.1.2 compile(c)

`compile` converts a test script into the baut test file. This command is called in `run` command process, so you will not need to call explicitly `compile` command.

```
$ baut compile test_sample.sh > test_sample.baut
$ baut test test_sample.baut | baut report
```

4.1.3 report(R)

`report` command receives data from the starndard input, formats the result of test.

```
$ baut compile test_sample.sh > test_sample.baut
$ baut test test_sample.baut | baut report
```

`report` command can interpret the following messages as a special message.

`#:RDY;<file_num>\t<test_num>`

This message is sent at the begin of all tests.

`#:STR;<baut_file>\t<test_file>\t<test_num>`

This message is sent at the begin of a test set.

`#:END;<baut_file>\t<test_file>`

This message is sent at the end of a test set.

`#:STRT;<test_function>[\t<alias_name>]`

This message is sent at the begin of a test.

`#:STRTDT;<todo>\t<test_function>[\t<alias_name>]`

This message is sent at the begin of a todo test.

`#:ENDT;<test_function>\t<exit_status>`

This message is sent at the end of a test.

`#:OK;<test_function>`

This message is sent when a test has ended successfully.

`#:ERR;<test_function>`

This message is sent when a test failed.

`#:SKP;<test_function>[\t<message>]`

This message is sent when a test has been skipped.

`#:DPR;<test_function>[\t<message>]`

This message is sent before a deprecated test begins.

`#:ERR0;<test_function>`

This message is sent when a critical error occurred.

`#:STP;<baut_fuke>\t<test_file>[\t<test_function>]`

This message is sent when test process is stopped.

This example shows a cycle of running test.

```
$ cat report.txt
#:RDY;1      1
#:STR;hoge.baut      hoge.sh 1
#:STRT;test_foo      alias_name
#:OK;test_foo
#:ENDT;test_foo      0
#:END;hoge.baut      hoge.sh
#:TIME;total test time
$ cat report.txt | baut report
1 file, 1 test
#1 hoge.sh
o alias_name
#$ 1 test, 1 ok, 0 failed, 0 skipped

  1 file, 1 test, 1 ok, 0 failed, 0 skipped
Time: total test time
```

CHAPTER 5

Customization

You can customize behavior and report format of Baut. Please see `libexec/baut--test-behavior` and `libexec/baut--test-report-default` and so on.

CHAPTER 6

Sample Templates

By default, Baut source tree includes some templates.

6.1 PostgreSQL

This template shows a test case like PostgresSQL regression test.

```
$ baut init -t postgresql postgresql
$ cd postgresql
$ tree
.
├── expected
│   ├── test_count.out
│   ├── test_from_file.out
│   ├── test_limit.out
│   ├── test_offset.out
│   └── test_where.out
├── run-test.sh
├── sql
│   └── test.sql
└── test_sample.sh
```

There are expected results under `expected` directory, and actual results are written under `results` directory.

```
$ cat test_sample.sh
#!/usr/bin/env bash

load "diff-helper.sh"

SQLDIR="$(__DIR__)/sql"

#: @BeforeAll
setup_all() {
  export PGDATABASE=sample
```

```
  dropdb --if-exists sample
  createdb --encoding=utf8 sample
  psql -c "create table users (id int primary key, name varchar(128) not null);"
  psql -c "insert into users select i , 'name-' || i from generate_series(1, 100) as␣
↪i;"
}

test_where() {
  run_diffx psql -c "select id, name from users where id = 1;"
}

test_count() {
  run_diffx psql -c "select count(*) from users;"
}

test_limit() {
  run_diffx psql -c "select id, name from users order by id limit 10;"
}

test_offset() {
  run_diffx psql -c "select id, name from users order by id limit 10 offset 50;"
}

test_from_file() {
  run_diffx psql -f "$SQLDIR"/test.sql
}

#: @AfterAll
after_all() {
  dropdb sample
}
```

`load 'diff-helper.sh'` loads the helper which enables the execution of `run_diff` or `run_diffx` commands. The execution result is as follows.

```
$ ./run-test.sh
1 file, 5 tests
#1 /Users/guest/workspace/baut/postgres/test_sample.sh
NOTICE:  database "sample" does not exist, skipping
CREATE TABLE
INSERT 0 100
o test_where
o test_count
o test_limit
o test_offset
o test_from_file
#$ 5 tests, 5 ok, 0 failed, 0 skipped

  1 file, 5 tests, 5 ok, 0 failed, 0 skipped
Time: 0 hour, 0 minute, 0 second
```

6.2 MongoDB

```
$ baut init -t mongo mongo
$ cd mongo
```

```
$ tree
.
├── expected
│   └── test_query.out
├── run-test.sh
└── test_sample.sh
```

```
$ cat test_sample.sh
#!/usr/bin/env bash

load "diff-helper.sh"

DBPATH="$(__DIR__)/data"
LOGPATH="$(__DIR__)/logs"

mkdir -p "$LOGPATH" "$DBPATH"

#: @BeforeAll
setup_all() {
  log_warn "==> start mongod"
  mongod --fork --dbpath="$DBPATH" --logpath="$LOGPATH/mongod.log"
}


#: @AfterAll
after_all() {
  log_warn "==> shutdown mongod"
  mongo --quiet <<EOF
use admin;
db.shutdownServer();
EOF
  rm -rf "$DBPATH"
}

#: @BeforeEach
setup() {
  mongo --quiet <<EOF
use test;
for (var i = 0; i < 100; ++i) {
  db.users.insert({userid: i, username: "name-" + i});
}
EOF
}

#: @AfterEach
teardown() {
  mongo --quiet <<EOF
use test;
db.users.remove({});
EOF
}

test_query() {
  run_diffx mongo --quiet <<EOF
use test;
db.users.count();
EOF
}
```

```
$ ./run-test.sh
1 file, 1 test
#1 /Users/guest/workspace/baut/mongo/test_sample.sh
2017-11-04 10:32:43 [WARN] test_sample.sh:12 - ==> start mongod
about to fork child process, waiting until server is ready for connections.
forked process: 44517
child process started successfully, parent exiting
o test_query
  switched to db test
  WriteResult({ "nInserted" : 1 })
  switched to db test
  WriteResult({ "nRemoved" : 100 })
2017-11-04 10:32:47 [WARN] test_sample.sh:19 - ==> shutdown mongod
switched to db admin
server should be down...
2017-11-04T10:32:47.976+0900 I NETWORK  [thread1] trying reconnect to 127.0.0.1:27017␣
↪(127.0.0.1) failed
2017-11-04T10:32:47.976+0900 W NETWORK  [thread1] Failed to connect to 127.0.0.
↪1:27017, in(checking socket for error after poll), reason: Connection refused
2017-11-04T10:32:47.976+0900 I NETWORK  [thread1] reconnect 127.0.0.1:27017 (127.0.0.
↪1) failed failed
#$ 1 test, 1 ok, 0 failed, 0 skipped

  1 file, 1 test, 1 ok, 0 failed, 0 skipped
Time: 0 hour, 0 minute, 4 seconds
```

6.3 Redis

```
$ baut init -t redis redis
$ cd redis
$ cat test_sample.sh
#!/usr/bin/env bash


#: @BeforeAll
setup_all() {
  log_warn "==> boot redis"
  redis-server &>/dev/null &
  REDISPID="$!"
  sleep 1
}

test_set_get() {
  run redis-cli SET key value
  [ "$result" = "OK" ]
  [ $status -eq 0 ]

  run redis-cli GET key
  [ "${lines[0]}" = "value" ]
  [ $status -eq 0 ]
}

#: @AfterAll
after_all() {
```

```
  log_warn "==> shutdown redis"
  kill $REDISPID
}
```

```
$ ./run-test.sh
1 file, 1 test
#1 /Users/guest/workspace/baut/redis/test_sample.sh
2017-11-04 22:35:26 [WARN] test_sample.sh:6 - ==> boot redis
o test_set_get
2017-11-04 22:35:27 [WARN] test_sample.sh:26 - ==> shutdown redis
#$ 1 test, 1 ok, 0 failed, 0 skipped

  1 file, 1 test, 1 ok, 0 failed, 0 skipped
Time: 0 hour, 0 minute, 1 second
```