Bats: Bash Automated Testing System

Bats is a TAP-compliant testing framework for Bash. It provides a simple way to verify that the UNIX programs you write behave as expected.

A Bats test file is a Bash script with special syntax for defining test cases. Under the hood, each test case is just a function with a description.

```
#!/usr/bin/env bats

@test "addition using bc" {
  result="$(echo 2+2 | bc)"
  [ "$result" -eq 4 ]
}

@test "addition using dc" {
  result="$(echo 2 2+p | dc)"
  [ "$result" -eq 4 ]
}
```

Bats is most useful when testing software written in Bash, but you can use it to test any UNIX program.

Test cases consist of standard shell commands. Bats makes use of Bash's `errexit` (`set -e`) option when running test cases. If every command in the test case exits with a `0` status code (success), the test passes. In this way, each line is an assertion of truth.

Running tests

To run your tests, invoke the `bats` interpreter with a path to a test file. The file's test cases are run sequentially and in isolation. If all the test cases pass, `bats` exits with a `0` status code. If there are any failures, `bats` exits with a `1` status code.

When you run Bats from a terminal, you'll see output as each test is performed, with a check-mark next to the test's name if it passes or an "X" if it fails.

```
$ bats addition.bats
 ✓ addition using bc
 ✓ addition using dc

2 tests, 0 failures
```

If Bats is not connected to a terminal—in other words, if you run it from a continuous integration system or redirect its output to a file—the results are displayed in human-readable, machine-parsable TAP format. You can force TAP output from a terminal by invoking Bats with the `--tap` option.

```
$ bats --tap addition.bats
1..2
ok 1 addition using bc
ok 2 addition using dc
```

Test suites

You can invoke the `bats` interpreter with multiple test file arguments, or with a path to a directory containing multiple `.bats` files. Bats will run each test file individually and aggregate the results. If any test case fails, `bats` exits with a `1` status code.

Writing tests

Each Bats test file is evaulated n+1 times, where *n* is the number of test cases in the file. The first run counts the number of test cases, then iterates over the test cases and executes each one in its own process.

For details about exactly how Bats evaluates test files, see Bats Evaluation Process on the wiki.

The *run* helper

Many Bats tests need to run a command and then make assertions about its exit status and output. Bats includes a `run` helper that invokes its arguments as a command, saves the exit status and output into special global variables, and then returns with a `0` status code so you can continue to make assertions in your test case.

For example, let's say you're testing that the `foo` command, when passed a nonexistent filename, exits with a `1` status code and prints an error message.

```
@test "invoking foo with a nonexistent file prints an error" {
  run foo nonexistent_filename
  [ "$status" -eq 1 ]
  [ "$output" = "foo: no such file 'nonexistent_filename'" ]
}
```

The `$status` variable contains the status code of the command, and the `$output` variable contains the combined contents of the command's standard output and standard error streams.

A third special variable, the `$lines` array, is available for easily accessing individual lines of output. For example, if you want to test that invoking `foo` without any arguments prints usage information on the first line:

```
@test "invoking foo without arguments prints usage" {
  run foo
  [ "$status" -eq 1 ]
  [ "${lines[0]}" = "usage: foo <filename>" ]
}
```

The *load* command

You may want to share common code across multiple test files. Bats includes a convenient `load` command for sourcing a Bash source file relative to the location of the current test file. For example, if you have a Bats test in `test/foo.bats`, the command

```
load test_helper
```

will source the script `test/test_helper.bash` in your test file. This can be useful for sharing functions to set up your environment or load fixtures.

The *skip* command

Tests can be skipped by using the `skip` command at the point in a test you wish to skip.

```
@test "A test I don't want to execute for now" {
  skip
  run foo
  [ "$status" -eq 0 ]
}
```

Optionally, you may include a reason for skipping:

```
@test "A test I don't want to execute for now" {
  skip "This command will return zero soon, but not now"
  run foo
  [ "$status" -eq 0 ]
}
```

Or you can skip conditionally:

```
@test "A test which should run" {
  if [ foo != bar ]; then
    skip "foo isn't bar"
  fi

  run foo
  [ "$status" -eq 0 ]
}
```

Setup and teardown functions

You can define special `setup` and `teardown` functions which run before and after each test case, respectively. Use these to load fixtures, set up your environment, and clean up when you're done.

Code outside of test cases

You can include code in your test file outside of `@test` functions. For example, this may be useful if you want to check for dependencies and fail immediately if they're not present. However, any output that you print in code outside of `@test`, `setup` or `teardown` functions must be redirected to `stderr` (`>&2`). Otherwise, the output may cause Bats to fail by polluting the TAP stream on `stdout`.

Special variables

There are several global variables you can use to introspect on Bats tests:

- `$BATS_TEST_FILENAME` is the fully expanded path to the Bats test file.
- `$BATS_TEST_DIRNAME` is the directory in which the Bats test file is located.
- `$BATS_TEST_NAMES` is an array of function names for each test case.
- `$BATS_TEST_NAME` is the name of the function containing the current test case.
- `$BATS_TEST_DESCRIPTION` is the description of the current test case.
- `$BATS_TEST_NUMBER` is the (1-based) index of the current test case in the test file.
- `$BATS_TMPDIR` is the location to a directory that may be used to store temporary files.

Installing Bats from source

Check out a copy of the Bats repository. Then, either add the Bats `bin` directory to your `$PATH`, or run the provided `install.sh` command with the location to the prefix in which you want to install Bats. For example, to install Bats into `/usr/local`,

```
$ git clone https://github.com/sstephenson/bats.git
$ cd bats
$ ./install.sh /usr/local
```

Note that you may need to run `install.sh` with `sudo` if you do not have permission to write to the installation prefix.

Support

The Bats source code repository is hosted on GitHub. There you can file bugs on the issue tracker or submit tested pull requests for review.

For real-world examples from open-source projects using Bats, see Projects Using Bats on the wiki.

To learn how to set up your editor for Bats syntax highlighting, see Syntax Highlighting on the wiki.

Version history

0.3.1 (October 28, 2013)

- Fixed an incompatibility with the pretty formatter in certain environments such as tmux.
- Fixed a bug where the pretty formatter would crash if the first line of a test file's output was invalid TAP.

0.3.0 (October 21, 2013)

- Improved formatting for tests run from a terminal. Failing tests are now colored in red, and the total number of failing tests is displayed at the end of the test run. When Bats is not connected to a terminal (e.g. in CI runs), or when invoked with the `--tap` flag, output is displayed in standard TAP format.
- Added the ability to skip tests using the `skip` command.
- Added a message to failing test case output indicating the file and line number of the statement that caused the test to fail.
- Added "ad-hoc" test suite support. You can now invoke `bats` with multiple filename or directory arguments to run all the specified tests in aggregate.
- Added support for test files with Windows line endings.
- Fixed regular expression warnings from certain versions of Bash.
- Fixed a bug running tests containing lines that begin with `-e`.

0.2.0 (November 16, 2012)

- Added test suite support. The `bats` command accepts a directory name containing multiple test files to be run in aggregate.
- Added the ability to count the number of test cases in a file or suite by passing the `-c` flag to `bats`.
- Preprocessed sources are cached between test case runs in the same file for better performance.

0.1.0 (December 30, 2011)

- Initial public release.