Quick Start Guide
Skip to documentation

test-runner.js

A script to automate building and testing of your programs

General information

test-runner.js is a script which can be used to automate testing of programs. It was originally made to help test small programs written in C, however it can be used to test programs written in many languages. It can be especially useful for testing programs written in languages which lack a dedicated testing framework.

One of the main goals of test-runner.js was to make the script portable, and as user-friendly to non-JavaScript developers as possible. As such, test-runner.js avoids the use of npm and external packages entirely, only using packages which are built in to Node.js.

It is recommended to download (or `git clone`) this repository and save it to a central location in your user's filesystem, then copy the `test-runner.js` file into the root directories of the projects where you wish to use it. You may also wish to copy `exampletestconfig.json` to your project and use it as a starting point when creating your configuration file if you are unfamiliar with JSON syntax.

test-runner.js is run using Node.js. If not already installed on your system, you may download it from the officlal Node.js website, or consider using a version manager like nvm.

test-runner.js should work on most Linux and Unix-like systems. It will also likely work on macOS. test-runner.js does not currently work on Windows due to differences in command line syntax between Unix-like systems and Windows.

NOTICE to System Administrators: test-runner.js will run user-specified commands on your system. If the Node.js runtime is installed on your system as a set-uid or set-gid executable, users who run test-runner.js may be able to use it to run commands with a privileged level of access to your system.

Your Download Should Come With the Following Files:

- `test-runner.js` - the main script to be run
- `exampletestconfig.json` - an example configuration file
- a `.gitignore` file
- a `LICENSE` file
- a `tests` directory containing:
    - `exampletest1.in.txt` and `exampletest1.out.txt` - a sample test input and test output file.

- `README.md` and `README.pdf`, both of which contain the same information.

If any files are missing, or have since been updated to a newer version, you can re-download them from https://github.com/max8539/test-runner, or run `git pull` if using Git to maintain the repository on your computer.

v2.x Release History

v2.0.0 (latest)

Improvements and Contributing

You are welcome to suggest improvements to test-runner.js. The best way to do so is through the Issues Board at https://github.com/max8539/test-runner/issues.
While suggestions are much appreciated, there is no guarantee every suggestion will be acted upon.

If you would like to take up an issue and implement the changes (especially ones given the thumbs-up label), you are welcome to fork the repository, implement the changes, and create a pull request.

WARNING: Be careful not to accidentally add files from other projects to the repository. While the `.gitignore` should prevent other files from being added automatically if it is there, you should double check the files you are commiting before you make the commit. It is extremely difficult, if not impossible, to remove references to a file in Git's history once a commit has been made, especially if you have since pushed that commit to an online repository.

Reporting Bugs

If you believe you have found a bug, please report it by creating an issue at https://github.com/max8539/test-runner/issues with the label bug-report. Please provide a description of what you did in the lead-up to encountereing the bug. If you are able to provide any other files used to help maintainers reproduce the problem, that would be much appreciated, however not a requirement.

Quick Start Guide

This quick start guide walks through testing a simple program written in the C programming language, compiled with gcc, and only makes use of terminal input and output (`stdin` and `stdout` only). If you are testing programs or scripts written in other languages, much of the setup will be the same, but the `build` command and the `run` command in each test should be changed to use the correct compilers, interpreters and/or runtimes.

1. Copy `test-runner.js` to the root directory of your project.
2. Create a file named `testconfig.json`. Copy the below template and place it in the file:

```
{
    "build": ["exit 0"],
    "tests": [
        {
            "run_cmd": "exit 0",
```

```
            "stdin_file": "",
            "stdout_file": "",
        }
    ],
}
```

3. Inside the array (square brackers) next to `"build":`, enter the command used to compile your code. The build setting should look something like `"build": ["gcc code.c -o program"]`.
4. Inside `"tests"`, next to `"run"_cmd:`, enter the command used to run the executable produced by the compiler. The setting should look something like `"run_cmd": "./program"`.
5. Create a file in the root directory of your project. Inside this file, write out the terminal input to be given to your program, ensuring that any whitespace and newlines are consistent with what your program expects. If your program prompts for inputs multiple times, each input should be placed on a separate line. Save the file, then enter the file's name next to `"stdin_file:"`. The setting should look something like `"stdin_file: "input1.txt"`.
6. Create a file in the root directory of your project. Inside this file, write out the exact terminal ourput you expect from your program when given the input file you created in the previous step. including any whitespace and newlines which your program is expected to print. If lines are printed to prompt for input, these should be included too. Save the file, then enter the file's name next to `"stdout_file:"`. The setting should look something like `"stdout_file: "output1.txt"`.
7. Save `testconfig.json`.
8. Open a terminal, switch to the root directory of your project, and run the command `node test-runner.js`. If you have done everything correctly, and your code was compiled successfully, you should be given a score out of 1. Did your program pass your test?
9. Continue to write tests by adding new items to the `"tests"` array in `testconfig.json` and creating more input and output files, to comprehensively test your program. An item in the array is the entire set of attributes enclosed in curly braces `{}`. You should also copy the curly braces to enclose each test object. A test object will look something like below:

```
{
    "run_cmd": "exit 0",
    "stdin_file": "",
    "stdout_file": "",
}
```

10. Re-run test-runner.js to test your program with your new tests. For more advanced testing options, see 3 - Writing Your Tests in the full documentation below.

If you encounter any errors produced by test-runner.js, see 4.3 - Troubleshooting Error Messages for troubleshooting help.

test-runner.js Documentation

Contents

1 - Setting Up your Project for Testing

Copy `test-runner.js` to the root directory of your project. You may also wish to copy `exampletestconfig.json` to your project and use it as a starting point when creating your configuration file if you are unvamiliary with JSON syntax. Other files in the repository need not be copied.

Install Node.js if it is not already installed on your system.

2 - Configuring Test Settings in `testconfig.json`

Test settings are configured by modifying attributes (keys) and values in the `testconfig.json` file. The settings available are described below.

Note that major updates to test-runner.js may add additional attributes in `testconfig.json`, or change the data types of existing attributes. If you notice test-runner.js running into errors while reading your settings, you should check back here for any recent changes.

Ensure that you enter the names of attributes exactly as shown below, and the data type of values is correct. Attribues and values should be entered as `"attribute": value`. Always use double quotes `""` for string values.

2.1 - Summary of Configuration Attributes

Name	Required?	Description
`"verbose"`	Optional	Set the amount of information to be shown in the terminal.
`"build"`	Optional	Specify an array of commands to build or compile your program.
`"first_failure_exit"`	Optional	If `true`, stops testing if any test fails.
`"score_exit_code"`	Optional	If `true`, returns the percentage score through the exit code.
`"tests"`	Required	Specify an array of test objects to define the tests that are run. See 3 - Writing Your Tests.

2.2 - Verbosity Settings

Attribute: `"verbose"` (optional, default: `1`)

Set `"verbose"` to the level of verbosity (information printed) that you wish to see. There are three levels to choose from:

- `0` - Print only the final result after running all tests, or an error message if tests could not be run.
- `1` - Print everything in level 0. Print out the stages of the script as they are being executed. Print out more information about errors that are encountered. Also print out the test names as they are run, and print out information about why your program did not pass certain tests, if this occurs.
- `2` - Print everything in levels 0 and 1. Print out the shell commands that are run by the scrript, as they are executed in different stages of the script, and print out the stages of test checking as they happen.

`"verbose"` should always be set to a numeric value.

2.3 - Build Commands

Attribute: `"build"` (optional, skipped if not present)

Set `"build"` to an array of one or more commands which should be run to build (or compile) your program. All commands should be written as non-empty strings. These commands will be run once, in the order they are written in the array, before running any tests.

If present, `"build"` should always be set to an array containing one or more non-empty string elements.

If `"build"` is not present, no build commands will be run. This may be appropriate for programs that do not have a build or compiling requirement, such as a JavaScript or Python script.

2.4 Run All Tests, or Stop when a Test Fails

Attribute: `"first_failure_exit"` (optional, default: `false`)

Determine if all tests should be run, or if testing should stop as soon a a single test fails.

if `"first_failure_exit"` is set to `true`, testing will stop immediately if any test fails, and an error message will be printed.

If `"first_failure_exit"` is set to `false`, all tests will be run and a score will be given at the end.

2.5 Returning the Final Score as the Exit Code

Attribute `"score_exit_code"` (optional, default: `false`)

If `"score_exit_code"` is set to `true`:

- An exit code of `128` will be returned if an error was encountered outside of running tests on your program, such as if a build or setup command encountered an error.
- If all tests were run and the script runs successfully, the exit code will be set as `100 - [percentage of tests passed]`. This is set so that an exit code of `0`, which conventionally indicates a program running successfully, corresponds to passing all tests. The actual percentage score can be found by calculating `100 - [exit code]`.

If `"score_exit_code"` is set to `false`:

- An exit code of `128` will be returned if an error was encountered outside of running tests on your program, such as if a build or setup command encountered an error.
- An exit code of `1` will be returned if the script was able to run successfully and one or more tests failed.
- An exit code of `0` will be returned if the script was able to run successfully and run all tests, regardless of the results from the tests.

This attribute is ignored if `"first-failure-exit"` is set to `true`.

2.6 - Specifying Your Tests

Attribute `"tests"` (required)

Set `"tests"` to be an array of one or more test objects. See 3 - Writing Your Tests for information about attributes in test objects.

You can create multiple tests by placing a comma `,` after the first test block (after the `}`), then place the next test block (from `{` to `}`) after the comma.

`"tests"` should always be set to an array containing at least one test object.

2.7 - Example `testconfig.json` file

Your `testconfig.json` file may look something like this:

```
{
    "verbose": 1,
    "build": ["exit 0"],
    "first_failure_exit": false,
    "score_exit_code": false,
    "tests": [
```

```
        {
            "name": "Example test 1",
            "run_cmd": "exit 0",
            "stdin_file": "stdini1.txt",
            "stdout_file": "stdout1.txt"
        },
        {
            "name": "Example test 2",
            "run_cmd": "exit 0",
            "stdin_file": "stdini2.txt",
            "stdout_file": "stdout2.txt"
        }
    ]
}
```

3 - Writing Your Tests

Each test is specified as an object enclused in curly braces `{}` in the `tests` array of `testconfig.json` with the attributes specified below. Each test is specified separately, and the values of attribues can vary between tests.

Ensure that you enter the names of attributes exactly as shown below, and the data type of values is correct. Attribues and values should be entered as `"attribute": value`. Always use double quotes `""` for string values.

3.1 - Summary of Test Object Attributes

Name	Required?	Description
`"name"`	Optional	Specify a name for your test.
`"expect_error"`	Optional	If `true`, the test expects an error from your program.
`"before_cmds"`	Optional	Specify one or more commands to be run before your program is run.
`"run_cmd"`	Required	Specify the command to run your program.
`"run_timeout"`	Optional	Set a timeout in seconds for your program.
`"stdin_file"`	Optional	Specify a file whose contents will be passed to `stdin` of your program.
`"stdout_file"`	Optional	Specify a file whose contents will be checked against the `stdout` of your program.
`"stderr_file"`	Optional	Specify a file whose contents will be checked against the `stderr` of your program
`"files"`	Optional	Specify one or more files produced by your program, and files containing the content that you expect. See 3.10 - Specify Files to be Checked

Name	Required?	Description
`"after_cmds"`	Optional	Specify one or more commands to be run after your program is run.

3.2 - Naming Your Test

Attribute: `"name"` (optional, a default name is generated if not present)

Set `"name"` to be the name of your test, which will be displayed as the test is run, and in error messages. If a name is not provided, the tests will be numbered, and a default name like "Test 3" will be generated instead.

If present, `"name"` should always be set to a non-empty string.

3.3 - Expecting an Error from Your Program

Attribute: `"expect_error"` (optional, default: `false`)

Set `"exoect_error"` to `true` if you expect your program to return a non-zero exit code when run in this test. Your test will now pass if your program encounters an error, provided that other checks in the test also pass. The test will fail if the program exits normally with an exit code of `0`.

If you expect your program to finish normally, set `"expect_error"` to `false`. The test will now fail if your program encounters an error.

3.4 - Run Commands Before Running Your Programs

Attribute: `"before_cmds"` (optional, skipped if not present)

Set `"before_cmds"` to an array of one or more commands which should be run before running your program in the test. These can be useful for performing any setup needed before runing your test. All commands should be written as non-empty strings. These commands will be run once, in the order they are written in the array. If any command fails, all testing will stop and an error message will be shown.

If present, `"before_cmds"` should always be set to an array containing one or more non-empty string elements.

If not present, no commands will be run before running your program.

3.5 - Running Your Program

Attribute: `"run_cmd"` (required)

Set `"run_cmd"` to the command used to run your program, including any arguments passed to the program. Commands should be written exactly as how you would enter it in a terminal at the project's root directory.

`"run_cmd"` should always be set to a non-empty string.

3.6: Setting a Timeout for Your Program

Attribute: `"run_timeout"` (optional, skipped if not present)

Set `"run_timeout"` to the number of seconds your program should be allowed to run before it is killed. This can be useful for stopping a program that enters an infinite loop or other non-terminating condition. If your program is killed, the test will be considered to be failed.

If present `"run_timeout"` should always be set to a numeric value.

If not present, no limit will be set on the amount of time your program can run.

3.7 - Specify Input for Your Program

Attribute: `"stdin_file"` (optional, skipped if not present)

Set `"stdin_file"` to the path to a file relative to the root directory of your project containing the input to be passed to the `stdin` of your program for this test (this is anything you type into the terminal while your program is running). If your program prompts for inputs multiple times, each input should be placed on a separate line, not including the prompt printed by your program, with exactly one newline character separating each input.

If present, `"stdin_file"` should always be set to a string.

If not present, no input will be passed to your program. If the program then waits for input, the test may not complete.

3.8 - Specify Expected Output from Your Program

Attribute: `"stdout_file"` (optional, skipped if not present)

Set `"stdout_file"` to the path to a file relative to the root directory of your project containing the output you expect from the `stdout` of your program for this test, including all whitespace and newlines (this is usually everything printed by your program to the terminal, including any prompts for input, unless you know that you are printing to `stedrr` instead). The `stdout` from your program will be checked after your program is ran, and it must exactly match the contents of the file provided for the test to pass.

If present, `"stdout_file"` should always be set to a string.

If not present, the `stdout` of your program will not be checked when the test is run.

3.9 - Specify Expected Error Output from Your Program

Attribute: `"stderr_file"` (optional, skipped if not present)

Set `"stderr_file"` to the path to a file relative to the root directory of your project containing the output you expect from the `stderr` of your program for this test, including all whitespace and newlines. The `stderr` from your program will be checked after your program is ran, and it must exactly match the contents of the file provided for the test to pass.

If present, `"stderr_file"` should always be set to a string.

If not present, the `stderr` of your program will not be checked when the test is run.

3.10 - Specify Files to be Checked

Attribute: `"files"` (optional, skipped if not present)

Set `"files"` to be an array of objects (enclosed by curly braces `{}`), each with the following attributes:

- `"program_file"`: The path from the root directory of your project to the file produced by your program
- `"check_file"`: The path from the root directory of your project to the file containing the expected contents of the file specified in `"program_file"` for this test.

Multiple such objects can be specified if multiple files produced by your program should be checked. The contents of each `"program_file"` specified must exactly match its corresponding `"check_file"` for the test to pass.

If present, `"files"` should always be specified as an array containing objects.

If not present, no files will be checked when running your test.

3.11 - Run Commands After Running Your Program

Attribute: `"after_cmds"` (optional, skipped if not present)

Set `"after_cmds"` to an array of one or more commands which should be run after running your program in the test. These can be useful for runing more advanced checks as part of your test. All commands should be written as non-empty strings. These commands will be run once, in the order they are written in the array. If any command fails, the test will be considered to have failed.

If present, `"after_cmds"` should always be set to an array containing one or more non-empty string elements.

If not present, no commands will be run after running your program.

3.12 - Example Test Object

A test object may look something like this:

```
{
    "name": "Example test 1",
    "expect_error": false,
    "before_cmds": ["exit 0"],
    "run_cmd": "exit 0",
    "run_timeout": 10,
    "stdin_file": "stdini1.txt",
    "stdout_file": "stdout1.txt",
    "stderr_file": "stderr1.txt",
    "files": [
        {
            "program_file": "file1a.txt",
            "check_file": "file1a.txt"
        },
        {
            "program_file": "file1b.txt",
            "check_file": "file1b.txt"
```

```
            }
        ],
        "after_cmds": ["exit 0"]
    }
```

3.13 - Order of Test Checks

In each test, checks are performed in the following order, regardless of what order they are specified in the test object:

1. Check if the program exits normally or encounters an error.
2. Check the program's `stdout`.
3. Check the program's `stderr`.
4. Check each of the files specified, in the order they are specified.
5. Run commands specified to run after the program is run, in the order they are specified,

4 - Running and Troubleshooting

4.1 - Running test-runner.js

Ensure that you have completed sections 1, 2 and 3 above before running test-runner.js.

To run test-runner.js from a terminal, switch to the root directory of your project, where `test-runner.js` and `testconfig.json` should be located, and run the following command:

```
node test-runner.js
```

4.2 - Troubleshooting Error Messages

While error messages which causes test-runner.js to stop are always printed, useful error information is hidden when running at verbose level 0. Before troubleshooting the error, it may be useful to re-run test-runner.js at verbose level `1` or `2` (see 2.2 - Verbosity settings).

One of the following error messages may be output by test-runner.js if it runs into an error which causes it to stop:

- **testconfig.json could not be found.**
    - Your test configuration file could not be found
    - Ensure that your test configuratio file is named exactly `testconfig.json`, and it is located in the same directory as `test-runner.js`.
- **testconfig.json could not be read due to a JSON syntax error.**
    - There are one or more JSON syntax errors in `testconfig.json`, preventing the script from reading and parsing the file.
    - The easiest way to fix this problem is to open `testconfig.json` in a code editor which supports `JSON` syntax highlighting and fix any issues which it identifies.
- **[attribute name] is missing**
    - A required attribute in `testconfig.json` is missing.

    - See information for `"tests"` and `"run_cmd"`.
- **The value of [attribute name] is invalid**
    - The value you entered for the attribute shown is invalid. This is often due to an incorrect data type being provicded.
    - See 2 - Configuring Test Settings in `testconfig.json` and 3 - Writing Your Tests for information on data types for specific attributes.
    - If you do not intend for an optional attribute to be used, delete the attribute name and its value from `testconfig.json`.
- **[location]: One or more commands are invalid.**
    - One or more commands you have entered at the location shown is invalid.
    - Commands should be strings, enclosed in double quotes`""`. They must also not be empty.
- **[\location]: File {file path} does not exist.**
    - The file at the file path shown could not be found.
    - Ensure that the file exists, and that you have spelled the filename and all directory names in the path correctly.
- **No tests specified.**
    - You have set the `tests` attribute to an empty array.
    - You should write at least one test before running test-runner.js.
    - See 3 - Writing Your Tests for information about writing tests.
- **An error occured while building your program with [cmd].**
    - [cmd] encountered an error while building or compiling your program.
    - At verbose level 1 or higher, test-runner.js will print the `stdout` and `stderr` from the build command that failed, which will likely include warnings and/or errors from your code.
    - You most likely need to fix your code so that it will build or compile correctly.
    - Refer to troubleshooting for the specific command in question, if available.
- **An error occured while running command [cmd].**
    - [cmd] encountered an error while running.
    - Commands specified under `"before_run"` are expected to pass. If any of these commands fails, testing will stop.
    - At verbose level 1 or higher, test-runner.js will print the `stdout` and `stderr` from the command that failed.
    - Refer to troubleshooting for the specific command in question, if available.

If you encounter a runtime error generated by Node.js (you will know you have one if you see a traceback listing of functions files), you may have encountered a bug. It would be appreciated if you submit a bug report..

4.3 - Other Troubleshooting

- **I'm not sure why my tests are failing.**
    - Re-run test-runner.js at verbose level 1 or higher to get an explanation of why certain tests failed.
- **The output looks the same as what the test expects, but it's still failing.**
    - Ensure that spelling is correct, and the use of whitespace and newlines is consistent in both your program and your test files. They must match exactly. Even a difference of one bit or one character will cause the test to fail.

- **Commands to check the result of a test are causing the test to fail when they're not supposed to**
    - Re-run test-runner.js at verbose level 1 or higher to see the error messages of the commands that failed.
    - If the error is unrelated to your test, then there is likely another issue which is causing them to register a failure. Any command that runs into an error will cause test-runner.js to register a failure.
    - Refer to troubleshooting for the specific commands in question, if available.