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 It was originally made to help test small programs written in C, which lacks its own dedicated testing framework, however it can be used to test programs written in many languages, provided that they run in the terminal and make use of standard IO streams (`stdin`, `stdout` and `stderr`).

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 `test-runner.js` and the `tests` directory into the root directories of the projects where you wish to use it.

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.

Your Download Should Come With the Following Files:

- `test-runner.js` - the main script to be run
- a `.gitignore` file
- a `LICENSE` file
- a `tests` directory containing:
    - `testconfig.json` - a file containing settings used for building and testing.
    - `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.

Release History

v1.0.1 Documentation Fixes (latest) v1.0.0 Full Release v0.9.2 License Change
v0.9.1 Licensing and Incomplete Documentation
v0.9.0 Code Pre-release

Improvements, Contributing and Bug Reports

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 you from adding other files 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.

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, and provide as much useful information as possible, including all output and any error messages from Node.js or elsewhere. If you are able to provide any files used to allow maintainers to attempt to replicate the problem, that would be much appreciated, however not a requirement.

How to Use test-runner.js

Contents

1 - Setting Up your Project for Testing

Copy test-runner.js and the `tests` directory, including its contents, in the root directory of your project. 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

Test settings are configured by modifying values in the `tests/testconfig.json` file. The following settings are available:

WARNING: Ensure that you enter settings correctly in `tests/testconfig.json` as the value in each of the key-value pairs, like `"key":value`, noting the semicolon between the key and the value. Also ensure that the the data type you enter for each value matches the data types specified below.

2.1 - Show/Hide License Information

Set `"showLicenseInfo"` to `true` to print license information every time the script is run.
Set `"showLicenseInfo"` to `false` to skip printing license information when the script is run.

`"showLicenseInfo"` should always be set to a boolean value.

2.2 - Verbosity Settings

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 the `stdout` and `stderr` of a build or setup command if it fails. Also print out the test numbers 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

Set `"build"` to an array of zero or more commands which should be run to build (or compile) your program. All commands should be written as non-empty strings, wrapped in double quotes (this is a

requirement of JSON files). These commands will be run once, in the order they are written in the array, before running any tests.
If your program does not have a build or compiling requirement, say for example, a JavaScript or Python script, set `"build"` to an empty array.

`"build"` should always be set to an array containing zero or more non-empty string elements.

2.4 - Test Setup Commands

Set `"setup"` to an array of zero or more commands which should be run to prepare the environment for each test. This could include creating files, removing files previously created by your program, or resetting the content of files. All commands should be written as non-empty strings, wrapped in double quotes (this is a requirement of JSON files). These commands will be run once, in the order they are written in the array, before each test.
If your tests do not require any actions to be taken before each test, or between tests, set `"setup"` to an empty array.

`"setup"` should always be set to an array containing zero or more non-empty string elements.

2.5 - Run Program Command

Set `"run"` to the command which should be used to run your program. The command should be written as a non-empty string wrapped in double quotes (this is a requirement of JSON files). The command will be run once for each test. When run by the script, the command will be affixed with a command to use the test's input file as your program's `stdin`. For example, the command `cmd` will be run by your program for test 1 as `cmd < test1.in.txt`.

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

2.6 Run Timeout

Set `"runTimeout"` to the maximum time in seconds which your program should take to run. If your program takes longer than this to run in a test, it will be killed and the test will be counted as a failure. The timeout will not be applied to any of the build commands, setup commands, or the time it takes for the script to check the output of your program.

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

2.7 Specify a File to be Checked

Set `"checkFile"` to be the name or path of a file which will be checked as part of checking the result of a test. The name or path should be written as a string wrapped in double quotes (this is a requirement of JSON files). After each test is run, this file will be loaded, and its data compared to a test file containing the expected data.
If there is no file to check, set `"checkFile"` to an empty string.

`"checkFile"` should always be set to a string.

2.8 Returning the Final Score as the Exit Code

To facilitate usage of this script in CI pipelines and other programs or scripts, you may choose for the script to return the percentage score as the exit code of the process.

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

- An exit code of `101` 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 `"scoreExitCode"` is set to `false`:

- An exit code of `1` 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 `0` will be returned if the script was able to run successfully and run all tests, regardless of the results from the tests.

2.9 - Example `testconfig.json` file

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

```
{
    "showLicense": true,
    "verbose": 1,
    "build": [],
    "setup": ["rm output.file","echo \"1\n2\n3\n4\n5\n\" > input.file"],
    "run": "deno run --allow-read --allow-write testProgram.js",
    "runTimeout": 5,
    "checkFile": "output.file",
    "scoreExitCode": false
}
```

3 - Writing your Tests

Each test consists of a minimum of 2, up to a maximum of 4 files. All tests consist of a stdin file, and one or more of a stdout file, a stderr file, and a file that should be created by the program, You may also create a file to signal to the test-runner.js that a non-zero exit code is expected for this test. All test files should be placed inside the `tests` directory, and tests should be numbered sequentially, beginning at 1, with no gaps between test numbers.

WARNING: Ensure that your test files are placed in the `tests` directory, they are named exactly as specified, and tests are numbered sequentially, beginning at 1, with no gaps between test numbers, or test-runner.js will not be able to find them.

3.1 - Input Test File

The first file to create is the input file. The contents of this file should be exactly what is passed to the `stdin` of your program. It should be named `testX.in.txt`, where `X` is the number of your test, e.g. `test1.in.txt`. This file is required for every test you write. If your program prompts for inputs multiple times, each input should be placed on a separate line, with exactly one newline character separating each input. If no input is needed, leave the file empty.

3.2 - Output Test File

One of the files you can create for your program to be checked against is an output file. This should contatin exactly what is expected in the `stdout` of your program for a given test's input, including any newlines and whitespace that is output. If nothing is expected on `stdout`, you may create this file and leave it empty. This file should be named `testX.out.txt`, where `X` is the number of your test, e.g. `test1.out.txt`. If this file is not included, your program's `stdout` will not be checked.

3.3 - Error Test File

One of the files you can create for your program to be checked against is an error file. This should contatin exactly what is expected in the `stderr` of your program for a given test's input, including any newlines and whitespace that is output. If nothing is expected on `stderr`, you may create this file and leave it empty. This file should be named `testX.err.txt`, where `X` is the number of your test, e.g. `test1.err.txt`. If this file is not included, your program's `stderr` will not be checked.

3.4 - File Output File

One of the files you can create for your program to be checked against is a file whose contents exactly match the expected contents of a file which your program will create for a given test's input. The name of the file should begin as `testX.file`, where `X` is the number of your test. A file extension of your choice may be affixed, however this must match the file extension of the filename specified at `"checkFile"` in `tests/testconfig.json`. An example for a `pdf` file may look like `test1.file.pdf`. Only include this file if your program is expected to output a file during that test.

3.5 - Expected Error File

One of the files you can create for your program to be checked against is a file signalling to test-runner.js that your program should run into an error while running, and return a non-zero exit code. If this file is included in your test, and your program finishes normally with an exit code of `0`, the test will fail. This file should be named `testX.error`, where `X` is the number of your test, e.g. `test1.error`.
If this file is not included, test-runner.js will expect your program to finish normally with an exit code of `0`. Any non-zero exit code will cause the test to fail.

3.6 - Example `tests` Directory

Your `tests` directory may look something like the following:

```
test1.in.txt
test1.out.txt
test2.err.txt
test2.error
test2.in.txt
```

```
test2.out.txt
test3.file.json
test3.in.txt
test3.out.txt
testconfig.json
```

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 your `tests` directory should be located, and run the following command:

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

4.2 - Sequence of Events when test-runner.js is run

test-runner.js will begin by checking that your `tests/config.json` is valid, and your test files are valid.
It will then run the specified build commands.
After building, it will run each of your tests, one by one, running the specified setup commands before each test.
After running your tests, it will report the number of tests passed, then exit.

4.3 - 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 higher (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:

- **Some of your settings are invalid.**
  - The data types of some values in `tests/config.json` are incorrect.
  - Double check the values in `tests/config.json` (see 2 - Configuring Test Settings)
- **One or more of your build commands are invalid.**
  - One or more of the commands specified in the `"build"` array of `tests/config.json` is either an empty string or not a string.
  - Double check that all the commands are valid commands and are stored as strings in the array (see 2.3 - Build Commands).
- **One or more of your setup commands are invalid.**
  - One or more of the commands specified in the `"build"` array of `tests/config.json` is either an empty string or not a string.
  - Double check that all the commands are valid commands and are stored as strings in the array (see 2.4 - Test Setup Commands).

- **Your run command is invalid.**
  - The command saved as the value for `"run"` in `tests/config.json` is either an empty string or not a string
  - Double check that the specified run command is valid, and is stored as a string (see 2.5 - Run Program Command).
- **No tests could be found.**
  - test-runner.js could not find your test files.
  - You should write at least one test before running test-runner.js.
  - Ensure that your test files are named correctly, and are numbered sequentially beginning with 1. As part of this, there should be a file named `test1.in.txt` (see 3 - Writing Your Tests).
- **Some of your tests do not have an expected stdout, stderr, or file output file.**
  - test-runner.js found an input file for a test, but could not find an expected stdout, stderr, or file output file for that test.
  - At verbose level 1 or higher, test-runner.js will print exactly which tests have missing files.
  - Ensure that your test files are named correctly, and are numbered sequentially beginning with 1 (see 3 - Writing Your 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 test setup with [cmd].**
  - [cmd] encountered an error while running.
  - At verbose level 1 or higher, test-runner.js will print the `stdout` and `stderr` from the setup 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 .js files), you may have encountered a bug. It will be appreciated if you re-run test-runner.js at verbose level 2, create a bug report at https://github.com/max8539/test-runner/issues, describe what you did before the error occured, and include the entire output and Node.js error message in your report.

4.4 - 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.
- **The final score shows less tests than the number which I wrote**
  - Ensure that your test files are named correctly.
  - Ensure that your tests are numbered sequentially, beginning at 1, and there are no gaps between test numbers (see 3 - Writing your Tests).