# Deliverable 3: Chapter 3

Edit New Page Jump to bottom

Jack edited this page now · 23 revisions

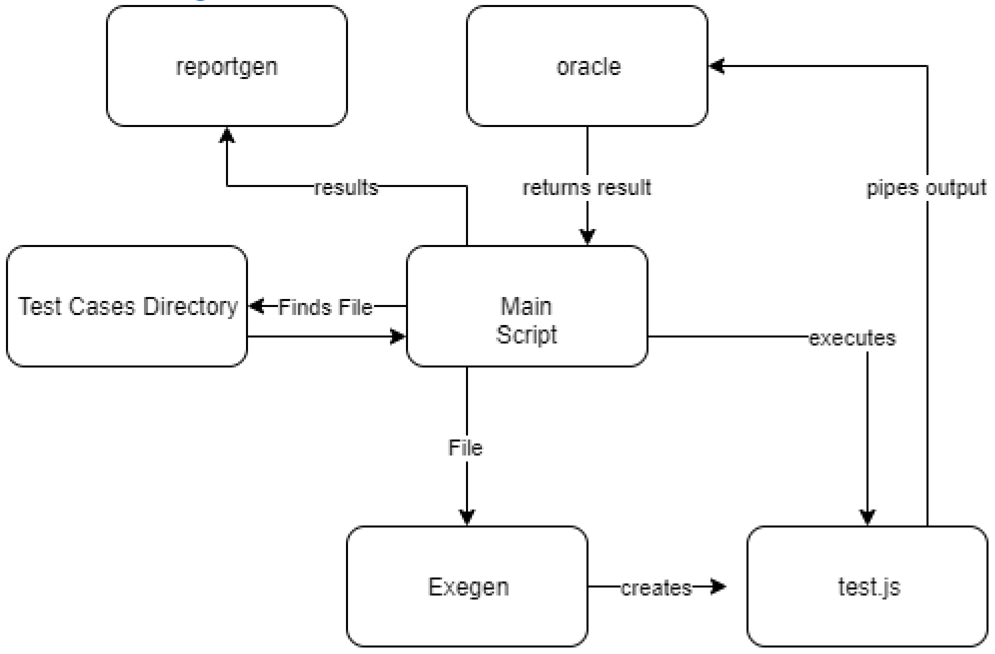
## Architecture

Our testing framework focuses on a cyclical process of running tests on each test case one at a time, adding each to the output file. As such, the parent script, `scripts/runAllTests.py` does most of the heavy lifting. After resetting and opening the output report file, it loops through each file in the `testCases/` directory; passing each file to `scripts/minigen.py` to be generated into an executable testable file within `testCasesExecutables/` .

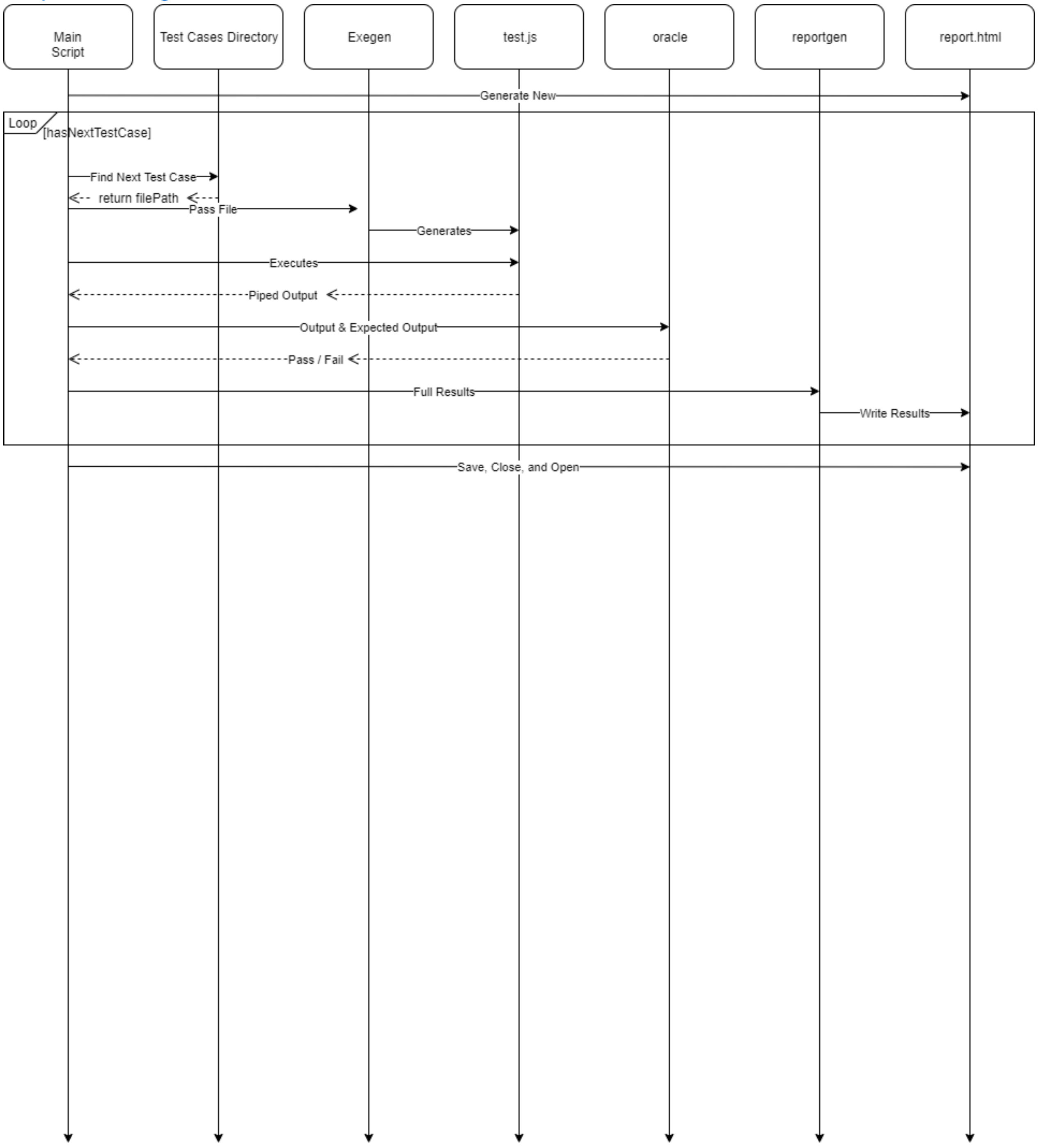
Next, `runAllTests` sends a command via subprocess, asking the operating system to `npm start` with a pipe to catch the output. `npm` has been configured to interpret the command `npm start` as `node testCasesExecutables/test.js` ; this is how the standalone generated test file is executed, as javascript requires a virtual environment such as a web browser, or Node.js, in order to be executable; javascript alone is not runnable without such an environment.

The console outputs are caught by the subprocess pipe, and sent onwards to `oracles/oracle.py` alongside the test-case-determined target value. The `oracle` determines if the test has been passed or failed, and returns that judgement back to `runAllTests` , where the results are formatted and appended to the growing report. Once this loop has been completed for all test case files, the execution and logging are complete; the report file `reports/testReport.html` is opened automatically, and the process is finished.

- Structural diagram



- Sequence diagram



# How to Run

## Prerequisites:

As this project is javascript-based, there is one significant prerequisite installation before any tests can be executed; `Node.js`. `Node`, alongside its in-built package execution manager `npm`, makes up an environment in which javascript can be executed. This is dissimilar to most languages, where they have a central compiler and executor that is core to the langauge; in javascript, the code is merely written standalone, trusting that a web-browser or environment will come along and execute it automatically when the (likely html) file embedding it is loaded. Our other prerequisite is the ubiquitous `Python 3`. Given that this framework is intended to be run on Linux, therefore, the only concern is in downloading and installing `Node`, which can be done with the simple commands `sudo apt install nodejs` followed by a `sudo apt update`, and possibly additionally a system restart. We have managed with continuous development to remove any possible `npm i` or `pip install` s as requirements - so after this you should be good to go.

## 1: Initialization

Your first step is to ensure that any files which you want to test are present in the `project/src/` directory. These files must be `.js`, and additionally, they must export a unit-testable function in order to be testable in this framework.

Second, you must ensure that your test cases in 'testCases/' are written correctly. The testcaseTemplate is as such:

```
TestID:%
Requirement:%
Component(filename):%
Method(functionName):%
Input:%
Output:%
```

Where the `%` must be substituted with the related detail.

## 2: Running the Tests

Assuming the test file is valid and the test cases are done correctly, all you have left to do is run the script!

First, ensure your terminal is located at the head of the `TestAutomation/` directory. If not, none of the scripts will path correctly; this location is specified in the project documentation, and is a hard requirement. Once you have successfully `cd TestAutomation`, simply type in `Python3 scripts/runAllTests.py`. After a minute or so, your report from running all of the tests specified will automatically open!

tests specified will automatically open.

## Modifications and Improvements

Deliverable 3 saw the rethinking and redesign of many core components of our testing system, mostly categorized by a removal of usage of existing frameworks, and a restructuring to support use of our own code in their place. We went from simple .sh scripts which called the configured frameworks to execute themselves, to a more verbose python script to fully control all steps in the new, much more manual, main software execution sequence. Each broad segment of our redesign is detailed below.

### Removal of Mocha Framework and Creation of Own Oracle

The Mocha Framework, as an extension to the core testing functionality of Node.js, was originally utilized for it's ease of use and quality of testing design. With it, we gained access to the ability to have a function call wrapped around a `.expects` in order to automatically utilize a built-in oracle with attractive formatting of results. This, when futher combined with the initial use of the mochawesome mocha extension, allowed for an `npm test` command to execute all test cases, compare their results, and generate it all into a well-formatted html file. However, while the purpose of this project is to ostensibly utilize all tools at our disposal in order to get the best results possible, this extensive utilization of frameworks was deemed by our team to be taking this concept too far; better to write our own code for this one, to prove that we can, and save the frameworks for future testing of real-world software projects.

Therefore, we configured a variant of `npm start` which simply executes an extant singular test file, and wrote our own oracle which is piped the console outputs of the test file's execution alongside a value representing the expected output, returning a pass/fail. While this process may be rudimentary when compared to the combined powers of `mocha` & `npm test`, we feel they better represent the spirit of the assignment in terms of writing our own code and functionalities from scratch.

A surprising advantage of this change, however, is significantly greater control of text execution outputs and piping. While mocha initially seemed a perfect framework, and mochawesome an attractively formatted report, they were nonetheless significantly limited in their customizability. So while they prove to be a simple 'catch-all' solution to testing for generic javascript code, they produced many outputs, such as automatic error handling, which got in the way of our own custom report requirements. Thus, formatted report generation is simpler with a handmade oracle, if only because our oracle outputs can now be designed specifically to work well with the specific report style that is intended for this project.

## Test-executable Generation Based on Interpretation of Test Case Files

Another major rework of our project since Deliverable 2 is the understanding, and correction of, our test case pipeline. Initially, our assumption had been that the verbose test-case syntax of mocha meant that our test cases and our test case executables could be the same file; in fact, in a real world testing situation, this would be the case. However, we had missed the fine print; in this project, our test case files must be simple txts, not pseudocode, and not real code. Thus, our readable test case files would have to be read, interpreted, and generated into executable code on a per-test basis. This change was not difficult, but represented a shift in our approach to our project; instead of achieving the goals of the project in the most effective and efficient way possible, we were now reading the details of a very explicitly specified software project. This may have been an intended lesson; getting the correct result is immaterial if the methodology does not match the project specifications.

## Removal of Mochawesome Framework and Piping of Executable-Run Outputs to Custom Report Generation

A further redesign came in the shape of fixing the layout of our reports generated from our test executions. When coming from an auto-generated report, moving to a self-made file left us with an initially raw output; the report went from something colorful and interactive, to a sequence of blocks of console output with no formatting and little design; this was largely due to the fact that our development cycle went as far as a minimum viable product as a report then focused on larger issues elsewhere. As we came back to report generation, therefore, design requirements had changed and our report would as well. While the new report is a significant improvement from our older working model, it still has many planned improvements before it can become framework-quality.

+ Add a custom sidebar

Clone this wiki locally

```
https://github.com/csci-362-02-2019/2-2.wiki.git
```