Deliverable: Final

Edit | New Page

Jump to bottom

Jack edited this page now · 16 revisions

2-2 Javascript Unit Testing Framework

Table of Contents:

- Introduction
- How to use
- High-Level Description
- Project Progress Documentation
  - Deliverable 1
  - Deliverable 2
  - Deliverable 3
  - Deliverable 4
  - Deliverable 5
- Self Evaluation
- Project Critique

Introduction

Unit testing is the process of testing the functions, methods, and objects within a given project by comparing the accuracy of individual function calls against their target values. It is exceptionally useful because, while unit tests alone do not prove that a software is bug-free, they can be used to ensure that correctness of core functions has been achieved for a number of predictable scenarios, and can be run after any modification, ensuring that a change to the code does not accidentally introduce new errors without being noticed. In some software development schedules, unit tests might be designed before the functions are, so that tests can be run on even partially completed code to see how that code is progressing towards completion. This has the advantage of emphasizing efficient code that achieves clear project goals & requirements; it also has the potential disadvantage of making programmers write code to pass the tests, rather than write holistically good code. Nonetheless, when used correctly, unit testing is a tool to improve a software's reliability and robustness, and as such is an excellent addition to any acceptance testing process.

For our project, team 2-2 elected to create a `javascript` unit testing framework, and to utilize the `nightscout` repository for its source material to create tests for. This repository provided some unique challenges;

1. None of us in team 2-2 had any experience in `javascript` whatsoever.
2. The `nightscout` repository utilized object-oriented programming, making purely functional unit tests difficult to find source material for.

How To Use

0. Clone this repository to a safe location. While this project is intended to be run in `linux`, it can be run successfully on `Windows` as well without issues.

Any other unusual machine configurations, such as `Macintosh` or `WSL`, are expected to work but not thoroughly tested; their most likely outcome is that they will succeed at every stage *except* for the automatic opening of the final report file. In such a case, the file can be found in `TestAutomation/reports/testReport.html`. Moreover, for any non-ubuntu users, some of the following directions may have to be modified to work correctly on your device.

1. To run the tests, first install nodejs.

This can be done with by following the instuctions here: https://tecadmin.net/install-latest-nodejs-npm-on-ubuntu/

2. Ensure that `nodejs` is installed correctly; your computer may need to restart, `sudo apt update` or `sudo apt upgrade -y`
3. Ensure that the file to be tested is a `javascript` file that is fully unit-testable, and that it is located in the `project/src` directory

3b. Ensure that all node dependencies for your tested javascript file are installed correctly. In our example case, `levels.js` requires `language.js`, and `language.js` requires the js extension `lodash`

3c. Therefore, for this test environment, ensure that lodash is installed with `npm i lodash`

4. Ensure that test cases exist for the tested file, are located in `testCases/`, and that they have been written according to the Test Case Template.

   Template:
   `TestID:%`
   `Requirement:%`
   `Component(filename):%`
   `Expected Output:%`
   `(optional *)call:`
   `(optional *)param:`

5. cd into the TestAutomation subdirectory. The locations where you run the scripts from is important, as all relative paths assume that TestAutomation is the source directory

6. execute the command `python3 scripts/runAllTests.py`. This should run the tests, generate the report, and open it in your web browser. Tests should either Pass, Fail, or Error; A Pass or Fail indicate that the test executed and returned a value, whereas the Error indicates that there is something wrong with your test case, or alternatively, a crashing error in your tested function.

High-level Description

The 2-2 Testing Framework is designed utilizing `python` as the main scripting language, with `javascript` as the actual execution language.

The main project structure is thus:

1. scripts/runAllTests.py (rAT) : main controller & report generator
2. scripts/oracle_exegen.py (o_e) : generates single .js file for each test case
3. oracles/oracle.js (o/o) : calls the test case js file and checks its output against a target value

Full Execution Flow:

- On initial execution, rAT opens a new report, overriding any existing report from previous generations, and formats it.
- Next, rAT cycles through each file in the testCases directory
  - For each file, it opens it, parses the contents into instructions,and passes those instructions to o_e.
  - o_e then creates a new test.js based on the file contents, overriding any existing test.js, then returns to rAT
  - rAT next calls o/o via NPM
  - NPM intializes a javascript vm, which then executes o/o
  - o/o calls test.js, executes it, and recieves the return values
  - o/o interprets the results, then outputs its conclusion (Pass/Fail) to the console and exits
  - rAT intercepts the console output, receiving the Pass/Fail, and writes these results to the report file
- Finally, rAT saves the report and asks the operating system to open it in the user's default web browser.

Deliverables

A note on the deliverables: They are accurate representations of the development process of this project, and as such provide insights and serve as time-capsules to the project as it evolved. Be wary of utilizing the deliverables for documentation, however, as they only represent the project as it was when they were written, not how it is now.

Deliverable 1

The objective for Chapter 1 was to clone the target project's repository, build the project, and run any existing tests that the original developers might have left behind. What we found was that the hardest part, by far, was figuring out what to to. The project structure of Nightscout includes a dockerfile, making it seem almost certain that building/deploying through Docker would be possible, but it immediately became clear that we didn't understand docker and don't know what any of the lines within the dockerfile do. So, we set about following the instructions on their github's readme/markdown file.

First step was installing NPM, which takes some time but is a simply 'sudo apt install npm'. However, this leaves you with a severely deprecated NPM; a new version of NPM and node JS has to be forced to install correctly;

- sudo npm cache clean -f

- sudo npm install -g n
- sudo n stable
- sudo npm install npm@latest -g

Next, MongoDB had to be installed, which could be done with a

- sudo apt install mongodb-clients

Finally, we discovered the 'setup.sh' file is actually congfigured to auto-install all the dependencies (and correct versions), as well as automatically running the build process once that is completed. A simple command installed and built the project in one fell swoop:

- ./setup.sh

or, since we've gone though all the trouble of doing all these steps manually, all that is left for a fully-manual build is a simple

- npm install

After installing all the pre-requisites, and using npm to install/build the project, part 1 was completed; we got a 'build succeeded' message in the console, at least. No idea how to access or run it, but it 'built'.

More ways to build/deploy the project were also tested. For one, we tried simply calling

- sudo apt install docker.io

By renaming 'dockerfile.example' to simply 'dockerfile', docker should able to recognize the project structure and run an automated deploy of the project? Seems docker has a few steps before it can do so, though.

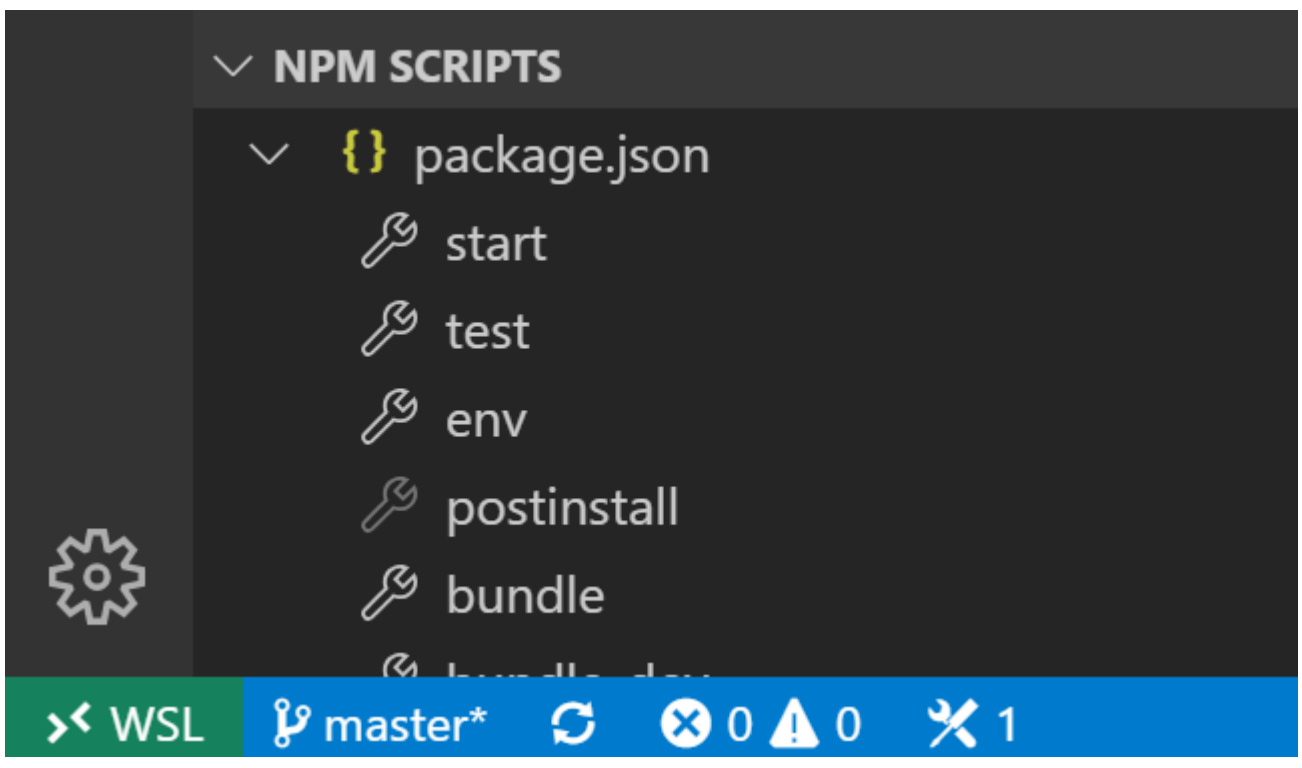
We attempted

- docker build .

But it ran into odd errors which suggest that the provided dockerfile is incomplete.

With so many ways to succeed, it seems clear the while this project is complex, it's quite well developed. If we understood *any* of the various frameworks which this project is compatible with, it would be extremely simple to deploy our own clone of the project. However, as a graduating senior who has never learned any of this stuff, except a tiny bit as a side note in our 'network security' class, it seems clear our education has failed us *completely* in this regard.

Next we tried to run the tests. We found some things that looked like tests, but being .js they didn't seem runnable on their own from the command line. Then, I realized that my IDE had automatically detected the tests somehow during the NPM build, and there was a clickable button called 'test' that ran the tests made by the Nightscout developers.



Running the tests ran the NPM test environment that had been developed.

```
    convertedOnTheFly: true }
      ✓ set a pill to the BWP with infos

    bridge
      ✓ be creatable
      ✓ set options from env
      ✓ store entries from share

    cage
      ✓ set a pill to the current cannula age
      ✓ set a pill to the current cannula age
      ✓ trigger a warning when cannula is 48 hours old

    client
Nightscout bundle ready
Nightscout report bundle ready
Application appears to be online
Authentication passed.
status isAuthenticated { apisecret: '',
  storeapisecret: false,
```

The majority of the tests passed, with those that failed probably being due to the fact that we have no idea how to deploy the server so any test that would check for connectivity or server status would (and should) return 'Fail'. Nonetheless, that most passed means that the build was successful, so that's good.

```
239 passing (1m)
19 failing
```

Ultimately, we achieved the objectives of the Chapter, but did so with a massive amount of confusion. Hopefully that was the intended experience.

Deliverable 2

Test Plan

Testing process

1. Find Testable Method
2. Import method to the TestAutomation/project/src/ directory
3. Modify method as needed to make it able to interact directly with the test environment.
4. Determine valid / invalid inputs for method
5. Write tests for method
6. Write test cases for tests in test environment
7. execute `./scripts/runAllTests.sh`

Requirements traceability

If any tests fail, requirements are not being met

Tested items

Any js file which has standalone functions are available to be tested. We will continue to add tests and test files until the project ends.

Testing schedule

Whenever tests are due, we will have done them by.

Test recording procedures

Utilization of recording frameworks within nodejs allows an html record of the test process to be generated and stored automatically on test run.

Hardware and software requirements

Hardware should be fairly permissive, but software is linux-based. Most shell scripts will work even in the WSL, but unfortunately the automatic file opener will run into issues if a cross-platform web-browser is not set up within WSL settings.

Constraints

Time and energy; we will write tests until we no longer have to; such does not result in a thoroughly tested software

System tests

As this project is unit testing, system tests are outside our purview. Sadly, as they seem pretty cool; but happily, as the system is rather complicated and as such would be pretty hard to write tests for.

Test Layout:

Test Case Files: Standard Layout

1. Test ID:

   ID is descriptive to the name of the file being tested e.g. 'units_test_16' is the 16th test of the 'units' file

2. Requirements being tested:

   short description of why this test exists

3. Component being tested:

   name of file (or files) containing source code being tested

4. Method being tested:

   name of function (or functions) which are called within the test

5. Function:

   summary of the features of the function(s) called

6. Input:

   input value

7. Expected output:

   expected return value note: this value is what the method *should* return; if it fails, blame the developers

Test Case Executables Interface: BDD

```
describe('Array', function() {
  before(function() {
    // ...
  });

  describe('#indexOf()', function() {
    context('when not present', function() {
```

```
      it('should not throw an error', function() {
        (function() {
          [1, 2, 3].indexOf(4);
        }.should.not.throw());
      });
      it('should return -1', function() {
        [1, 2, 3].indexOf(4).should.equal(-1);
      });
    });
    context('when present', function() {
      it('should return the index where the element first appears in the array', function() {
        [1, 2, 3].indexOf(3).should.equal(2);
      });
    });
  });
});
```

https://mochajs.org/#interfaces

Testing Framework Environment Details

Operating System and Testing Framework

- Linux
- NPM (node js package manager; essential for building and testing js code)
- Mocha (one of many available test case frameworks within nodejs; selected for being simple to use)
- Mochawesome (a lovely html file output extension to mocha)

Deliverable 3

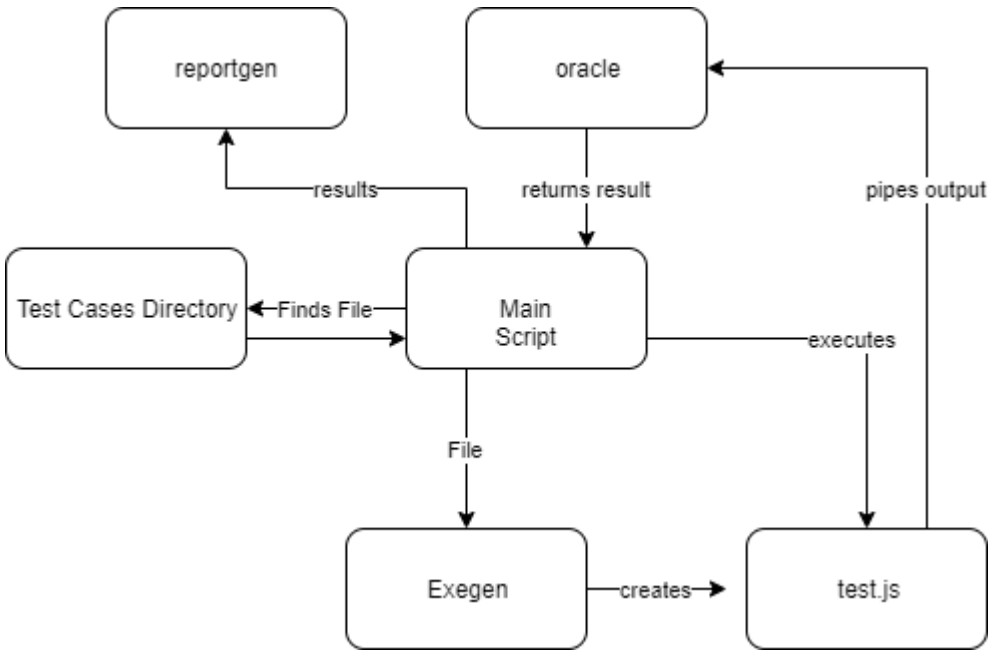
Architecture

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/`.

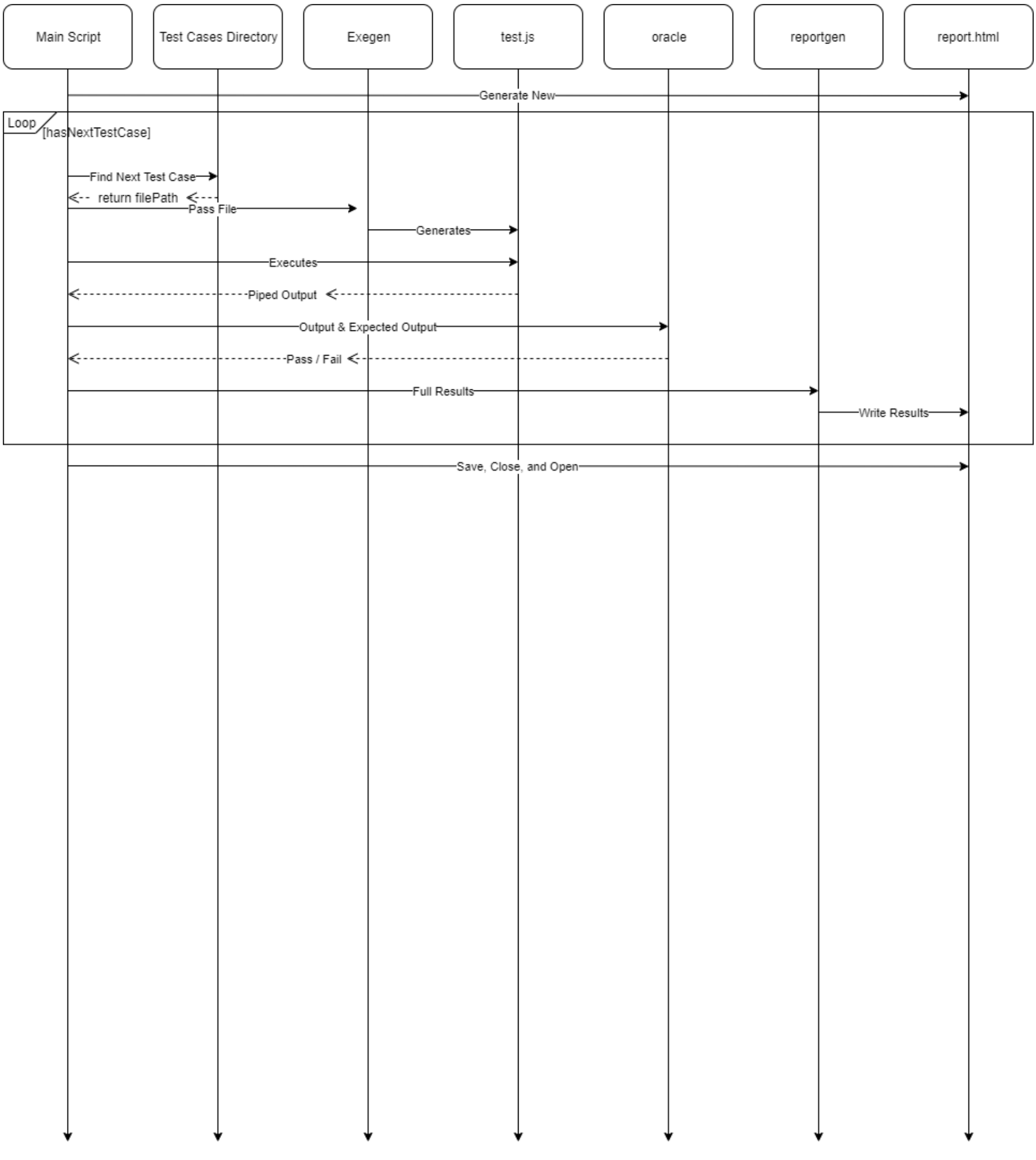
Next, `runAllTests` sends a command via subprocess, asking the operating system to `npm start` with a pipe to catch the output. `npm` has been configured to interpret the command `npm start` as `node testCasesExecutables/test.js`; this is how the standalone generated test file is executed, as javascript 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!

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.

Deliverable 4

Improvement through re-implementation

Deliverable 4 saw our unit testing framework tested against, and re-designed to work with, object-based components. We found this necessary because our project has a severe shortage of pure functions; nearly everything in the nightscout repository is, in fact, an object. Unit testing of functions is great and simple; you import the file, pass the function name, pass the value, and it returns an output that can either agree or disagree with your expectations. Object-based unit tests, however, require a few more steps to the process.

The testing scenario for an object goes as such:

1. Initialization: creating an object and giving it an initial state such that an internal function can be tested
2. Testing: calling the unit-test target function on the object
3. Post-processing: in the case of a function that returns an object, running the 'getter' necessary to actually receive the value produced in step 2.

Compare and contrast to the function testing scenario:

1. The whole thing: calling the function, which returns a value

Our original test-case pipeline was not well suited to the new process, as it only supported the one step; the 'middle' part of an O.O. test. Thus, redesigns were required.

Redesign 1

Our first variant involved no changes to the test cases or executable generator at all; it was merely hacking together a functioning test. In our test-case layout, we kept it as

`filename:`
`function name:`
`input:`
`output:`
but made allowances by adding a new parameter: `*args:`

thus, any non-function object calls could be placed like so:

`function name:function`
`input:inval`
`*args:.get()`

which would translate to, roughly: `function(inval).get()`

however, the *args were not an elegant solution when we realized some objects had to be inititialized in particular ways. For instance, how would we parse `functionName.input` vs `functionName(input)` vs `functionName.input()`? Clearly another design was required.

Redesign 2

This one didn't last very long, but it's here anyways. Part of the evolutionary design process is having dead branches to the family tree, after all. We considered that, perhaps, the issue was the type of test being run. For instance, if you're testing a function, it needs the `()`, and if you aren't, it doesn't. Simple, right? With that in mind, our next design was implemented.

`Method being tested:`
`Input:`
`Isobject:`

The idea was that `functionName(input)` could have an isobject of 0, and `functionName.input` could have an isobject of 1. But what about `functionName().input`, the function that returns an object? Or even `function1(input1).function2(input2)`, the function that returns an object of functions? A more dynamic approach was required, and as such this design never got past the theoretical stage.

Redesign 3

The final and current design, therefore, had to account for dynamic number of objects, parameters, and functions, in any order. After all, why would we go through all the effort of making the test cases accept `function().objcall` and not accept `objcall.function()`? We had to go full optional number of parameters. As such, we went with a fully * optional `call` line, and a fully * optional `param` line. The new implementation is like so:

`component:`
`(opt*)call:`
`(opt*)param:`

and can be combined in any permutation necessary. For instance, what if the file import is itself a constant?

`component:constant.js` (no lines following)

-> which generates into `component` when called

What if the import returns an object, the function is behind a subobject and takes no parameters, and the answer is only accessible by getting the 1st index of a method called on that object?

`component:crazyobjects.js`
`call:objmethd`
`call:function`
`param:`
`call:resultmethd[1]`

-> `crazyobjects.objmethod.function().resultmethd[1]`

While this loses some elegance, one must admit that testing this particular case has very little by way of "elegant" solutions. And these are exactly the kind of objects we're dealing with. Take `languages.js`, for example. To get it to translate a word, our test case looks like this:

`Component:language.js`
`param:`
`call:set`
`param:'fr'`
`call:translate`
`param:'Clock'`

which generates into this: `language().set('fr').translate('Clock')`

We hope that y'all agree when we say that, though we admit the redesign lacks the simplicity of a pure-function testing environment, it gains far more in functionality than it loses in elegance. Either way, it was necessary.

Deliverable 5

Breaking a few Eggs

We modified a few functions to make them fail the tests. This was simple enough to do; changing the indexes of warning levels, or making our rounding happen at the wrong stage during calculation. These all achieved the expected result; the tests failed and displayed as such nicely in the report.

Error 1: `language.js`
Injected a default language error, so that it no longer defaults to english. This error might never be discovered, as all ui elements default to english; but it may ask the translator to translate an element to the unknown default language, which now has invalid references.

Error 2: `levels.js`
Injected an indexing error, where an alert level is associated with the wrong numeric equivalent value. This error might represent a simple miscount, or an abandoned refactor of the error values that accidentally carried forward this change. It leads to urgent warnings mapping to-int and from-int differently, which could cause significant internal errors whenever a warnings level conversion occurs.

Error 3: `times.js`
Injected a mathematical error, where a `*` token is instead a `+` token. This is especially harmful to times because each time type has a custom set of functions, making this error specific to only one from-to conversion.

Error 4: `units.js`
Injected a rounding error, where without rounding the resulting output, converted values are left in un-rounded float, which may overflow the screen if there is only a small space in the ui to display the values.

Error 5: `levels.js`
Injected a non-key error, where a key not found now returns 'no alert' rather than 'unknown alert'. This is a stealthy error because it is not necessarily wrong in a vacuum, but because the method doesn't return what it is required/expected to do, it may lead to errors elsewhere.

With this deliverable complete, we are confident that our testing environment can accurately reflect the passing and failing of any valid test we could come up with. While this test framework offers only a drop in the bucket when compared to the functionality of other published testing frameworks, for this project we feel that it succeeds in its goal of unit testing based on imported files and hand-written test cases.

We are pleased with the outcomes, and we hope you are too.

2-2

Self-Critique

We had a hard time properly distributing tasks, largely due to lack of seeing the big picture at any one moment. It might be somewhat ironic for a project management class to suffer for this, but we have the excuse of being in the process of learning it all, so to speak. Ultimately, everything got done, but it came at the cost of frequently having all three members working on our own implementations of the same method or function, rather than distributing these requirements in a more efficient manner. We excuse this, however, by suggesting that by doing everything, we all learned how to do the entirety of the project as individuals, rather than a more distributed project in which we might not all understand everything.

Final Conclusions

The project was pretty good. We learned a lot, especially about testing, but also about ideal project structure, and implementation of javascript codebases. All of this lead to a lot of good new knowledge.