Project Final Report

Sanil Shah | UNI: ss4924

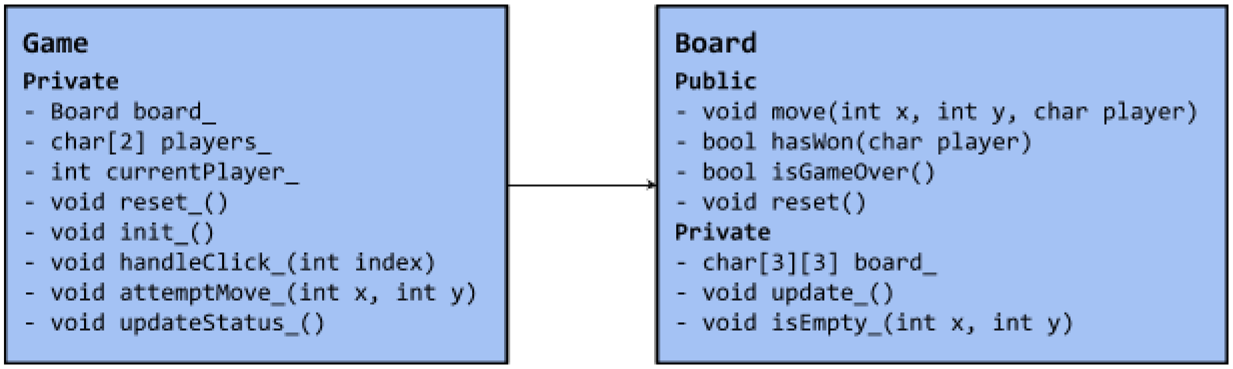
Analysis and evaluation of JavaScript testing frameworks

Overview

This project was worked on independently. It builds on the topic that I explored while researching and writing my midterm paper. This project involves developing an understanding of several JavaScript testing frameworks to determine how easy they are to use. In order to do so, a JavaScript application that will interact with DOM elements so as to mimic a real world application is created to act as the program under test. Unit tests for this application will then be implemented using three different test frameworks. The project will aim to analyze the developer effort and test coverage obtained from implementing unit tests in each of the frameworks to better understand the most effective framework or tool to use.

Project Details

The program under test is a simple tic-tac-toe web application where a two human players can play against each other on the same computer. The source code for the application can be found here and the game can actually be played at this demo page. The application has been written to correctly implement the rules of tic-tac-toe. Players will not be able to make a move once the game has been won or tied. Squares that are full may not be overwritten and the application correctly alternates between two players and identifies when a single player has won. A simple architecture diagram of the tic-tac-toe game can be seen below:



Note that while all methods on the Game class are private, interactions with those methods occurs through click handlers on various DOM elements. These click handlers are added when the Game object is created.

The two classes above as well as interactions with the DOM is then tested using three different test frameworks. This project concentrates on Mocha, Jasmine and Jest since they seem to be the most commonly adopted test frameworks today according to the survey carried out during the midterm paper. While implementing the tests, time taken to write the unit tests, setup the test framework, and run the tests were measured for each of the frameworks. Further, code coverage, lines of test code and number of google searches needed to implement a complete test suite was also measured in order to get a better understanding of the complexity of using these testing frameworks.

This project report compares the differences in productivity and developer effort from implementing tests in each of these test frameworks. Most new JavaScript developers shy away from testing because of the setup time needed to get these frameworks up and running and properly integrated with their code, so this project will measure and evaluate this metric as well. As a first time user for each of these test frameworks, my implementing them from scratch for a new application acts as an unbiased benchmark.

System Evaluation

Since this project evaluates JavaScript testing frameworks that have been developed by others, I will answer the questions listed in that section of the assignment prompt.

- ***Systems being evaluated:*** Jasmine, Mocha and Jest testing frameworks
- ***Why they were chosen:*** I chose these three testing frameworks based on the research done for the midterm paper. The developer survey conducted for the midterm paper indicated that these three test frameworks were the most popularly used ones for JavaScript testing. Further, I chose frameworks that I had never used before to provide unbiased results for set up times and to understand the difficulty of learning to use these frameworks
- ***Novel or unique:*** There are plenty of articles and blog posts comparing the functionality offered by various JavaScript testing frameworks. They also offer simplistic examples on how to write basic tests. However, there doesn't seem to be an in depth analysis of the actual time it takes to setup and configure these tests for a small to mid-sized project that is more than a playful example. This project provides a benchmark to help compare setup and implementation times for the three test frameworks mentioned above. This will provide a deeper understanding of which framework is easiest to use and hence

help improve developer productivity and test coverage by decreasing the reliance on manual testing.

- ***User community:*** This study is primarily targeted at JavaScript developers who heavily rely on manual testing today and have no form of unit or integration tests. This could be beginners or intermediate developers as indicated by the survey in my midterm paper. The value offered by this project is two-fold. First, it provides more robust examples by actually creating a mid-sized project which will then be sufficiently tested using all three frameworks. These can be used as a starting point for developers looking to test their code. Secondly, the analysis of the time taken to setup and configure the frameworks, the number of web searches required to get them working, the ease of implementation, and time taken to run the test suites can be used by these developers to determine which test framework is easiest to use and most suitable for their projects.
- ***Benchmarks and metrics:*** The primary metrics collected are as follows:
    - Setup time - This is essentially the time it takes to get the most basic unit test running.
    - Implementation time - This is the time it takes to write a complete suite of tests for the tic-tac-toe program described above.
    - Running time - This measures the time it takes to execute the full suite of tests.
    - Coverage - This measures code coverage where possible using the framework itself if it is built in, or using a third party tool to measure coverage. This will also include the complexity of configuring a code coverage measuring tool to work with these test frameworks.
    - Search queries - I attempted to keep track of the number of Google searches I used in implementing unit tests in each of these frameworks. This will provide another data point to indicate the level of difficulty in using these frameworks.
    - Lines of test code - This measures the number of lines of test code written to create a complete suite of unit tests for each of the test frameworks.

Unit Test Implementation

Jest

The first test framework that I worked with was Jest. I started by writing unit tests for the Board class. In order to use Jest, I had to first install Node on my machine. Next, I installed the Jest package on my computer using `npm` which is the Node Package Manager tool. I had to configure the framework by modifying the package.json file to use Jest as my test framework. Next, I spent around 25 minutes setting up a single simple unit test for the Jest framework. This took particularly long since I had to learn how to use Node specific features like `require` and `modules.export`, which I wasn't familiar with. This took slightly longer than expected because ES6 also offers features like `import` and `exports` which accomplish similar functionality, however, those did not work correctly with Node. Since this was the first JavaScript test framework that I was setting up, it took 8 different Google searches to correctly configure and have a working test that could be run from the command line using `jest` or `npm test`.

Part of the complexity of running unit tests using Jest, was that the source code needed to be updated with things like `module.export` as recommended by their quick start guide. This is a Node specific feature which cannot run in a non Node environment like a browser. Since I needed the source code to continue working independently in a browser for the tic-tac-toe web application I had to come up with some hacks in order to ensure that Node specific lines of code were only executed when Node was present. These fixes can be seen in the source code of the board.js and game.js files around the `module.export` statements at the bottom and the `require` statement at the top of each of the classes. Further, I realized that the classes need to be isolated into their own closures in order to prevent duplicate requires or namespace collision. While figuring these hacks out took some time, the same findings applied to the other two testing frameworks, and hence those were relatively easier to setup.

Once I had the setup part figured out, the process of writing a complete test suite began. Jest is an intuitive language to understand and the APIs offered to write unit tests, implement test assertions, and create testing mocks is rather easy to use. The process of writing a full suite of tests was relatively straightforward and took me 3 hours to complete. While this seems long, this was mostly because I needed to figure out which tests to write, how to test manipulation to the DOM as well as how to measure code coverage for Jest. Additional time was spent figuring out how to handle dependencies between the Game and Board class, and the ability to mock out the Board constructor while testing the Game class. I used around 12 different Google searches to complete writing a full suite of tests using the Jest framework. I found that most of what I was looking for was available in the Jest API documentation or could be found as an answer on StackOverflow. This made Jest easy to work with once the setup was complete.

Lastly, I spent some time setting up the code coverage tool to measure the effectiveness of the test suite that I had written in Jest. This took a little over 15 minutes and was incredibly easy to do since Jest comes with a built in coverage tool, and didn't require any additional downloads. A few quite searches provided me with the required changes to the package.json file to start running the code coverage tool with each run of the test suite. This also helped me find gaps in my test suite, and lines that hadn't been sufficiently tested, which lead to my adding a few additional unit tests to cover those edge cases.

Overall, the process of writing unit tests using Jest, as well as setting up the framework to capture code coverage data, took a little under 4 hours and approximately 20 Google searches. The Jest API is extremely well documented and there is an active developer community which provides answers to any questions and issues that may arise.

Jasmine

The second test framework that I wrote tests for was Jasmine. Since I had already setup tests using Jest, I had already encountered and solved a lot of the issues that I would have faced while working with Jasmine as well. Node was already downloaded from when I setup the Jest tests, so I simply had to install the Jasmine testing framework using `npm`. Once this was complete, the ability to copy paste code while making minimal changes to conform to the Jasmine API, meant that I had my first Jasmine test running in 9 minutes. It only took 3 Google searches to get these tests up and running and be able to execute them from the command line using the `npm test` or `jasmine` commands.

Once the basic test was written, the remaining tests were copied over from the Jest test suite. This was mostly easy to do since the syntax for the two APIs is remarkable similar. The primary difference is that Jasmine doesn't run in a browser environment so the tests that dealt with DOM interactions needed to be updated to use `jsdom` instead. JSDOM is a tool that allows Jasmine and Mocha unit tests to mimic a browser environment and tests DOM interactions. A majority of my time while porting the test suite over to Jasmine, was being able to find a way to test DOM interactions and setting up JSDOM with Jasmine. This was eventually accomplished by running `npm install --save-dev jsdom` to install the jsdom package.

I used 17 Google searches to figure out how to implement a full suite of tests using Jasmine. A majority of these searches was to figure out how to test DOM manipulation, JSDOM API and setup documentation as well as Jasmine documentation. One other major difference between Jest and Jasmine was that there was no easy way to mock out the Board class constructor, while writing tests for the Game class. A new Board object is created as part of the constructor in the Game class. The Game class constructor also calls the `init_()` method which in turn calls the `updateStatus_()`. This calls methods on the Board class like `board.hasWon()` and `board.isGameOver()` which need to be mocked out before the Game is created at all.

This wasn't a problem in Jest since it allowed me to mock out the Board constructor itself. I was unable to find similar functionality for Jasmine. The correct way to really fix this issue is to refactor the Game constructor to ensure that it had no side effects like calling various Board methods. However, doing so would force me to update the Jest tests as well so instead I resorted to the next best alternative which was to make the Game constructor accept a Board object that it would use. This allowed me to create and pass in a fake Board object to the Game constructor, who's methods could then be monitored while testing the Game class. This entire process took a while to figure out and hence writing all the tests for Jasmine took 2 hours and 30 minutes even though a lot of the tests could simply be copy pasted from Jest and modified to fit the Jasmine API.

Once I had the complete test suite using Jasmine, I attempted to add code coverage to the Jasmine tests as well. Unlike Jest, Jasmine did not come with a built in code coverage tool, so I had to integrate with another tool called Istanbul. I initially tried using Karma which was the first result that popped up during Google searches, however I had a hard time getting it to work with PhantomJS or Chrome which were browser environments. I found myself having to download more and more Node packages and realized that there had to be an easier way. Additional Google searching lead me to use Istanbul, which was relatively quick to setup and didn't require any additional packages other than 'istanbul'. I modified the package.json file to run the coverage tool during the test run. It took some tweaking to point the coverage tool to the correct source code folder but once I had it running, it showed that I had 96% code coverage which is the same as that from Jest. The lines of code that could not be tested for the branches around that hack that I mentioned earlier, which was used to ensure that Node specific APIs are not called in the browser.

Overall, Jasmine tests took me a little over 3 hours to complete, including implementing the unit tests and setting up the code coverage tool. While porting over tests was incredibly easy since it mostly involved copy pasting code, a majority of the time was spent figuring out how to test DOM interactions using JSDOM, mocking out the Board constructor correctly and then setting up code coverage measurement using Istanbul. The Jasmine API documentation is also pretty clear, however I found myself having to search for various things as I encountered the kinds of issues that I described above. Meanwhile, Jest mostly works out of the box and didn't require any sort of integration with any other tools like JSDOM that was needed for Jasmine.

Mocha

The last test framework that I worked with was Mocha. Mocha and Jasmine are extremely similar and by this time I'd encountered and solved almost any issues that I would face while writing these unit tests. I started by downloading the Mocha package using `npm`. Mocha doesn't come with it's own assertion API like Jasmine and Jest however it works with the 'assert' API that comes installed with Node so I simply used that for test assertions. Developers can choose to use other assertion libraries like Chai if they wish to do so. It took 15 minutes to get the first test running using the `npm test` or `mocha` commands. A majority of the time was spent downloading Mocha itself and looking up the API for the Node assert library.

Once the simple test was set up, the remaining tests were essentially copy pasted from their Jasmine counterparts. Changes were made to ensure that the test complied with the Node assert API instead of the Jasmine API. DOM interactions were handled using JSDOM, similar to Jasmine. Since this was already installed, I did not need to worry about downloading another package. However, while Jasmine and Jest come with built-in APIs for creating mocks and spies, Mocha requires you to use a third party library to implement mocking functionality.

After some Google searching about implementing mocks in Mocha, I downloaded the Sinon library using `npm`. This was then easily integrated into the Mocha unit tests to achieve the same kind of behavior that was accomplished in Jest and Jasmine. Creating mocks using Sinon was extremely intuitive and more or less mimicked the APIs offered by Jest and Jasmine for the same kind of functionality.

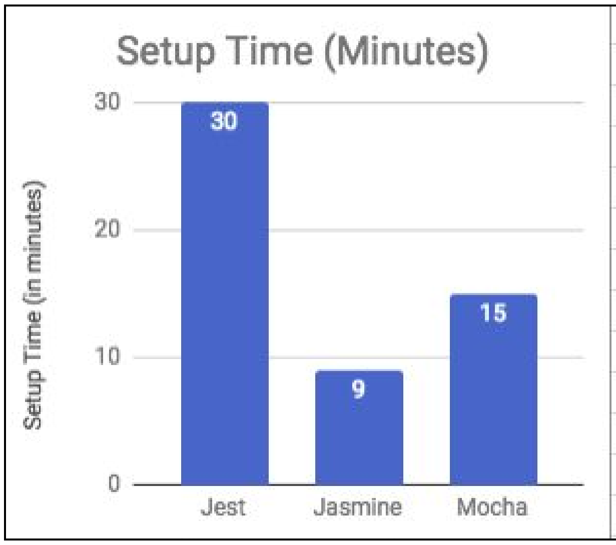
Since I was able to copy paste most of my tests from Jasmine, writing the full suite of Mocha tests barely took 40 minutes. The majority of this time was spent replacing the API calls to match the node assert API, and looking up how to use the Sinon library. It only took 4 Google searches as well since I had to simply look up the documentation to the Node assert and Sinon libraries. This was by far the fastest completion of a test suite but that was in a large part due to the fact that issues encountered and solved in Jest and Jasmine were not faced again with Mocha since they'd already been dealt with.

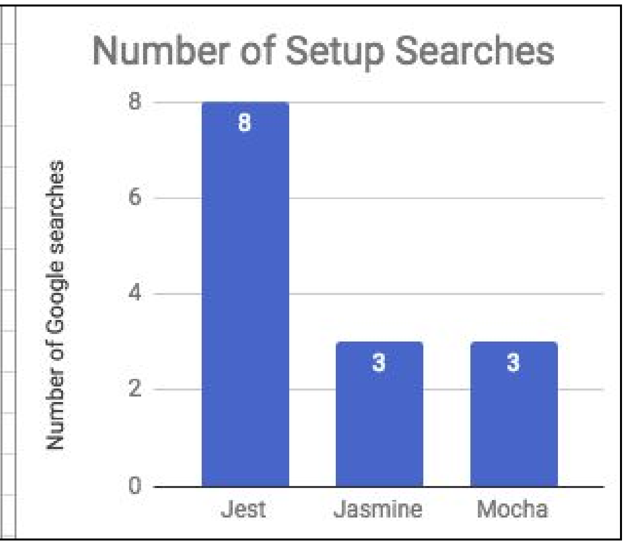
Once the test suite was complete, I quickly added coverage measurement the same way that I did for Jasmine. Once again, I used Istanbul and confirmed that the unit tests achieved 96% code coverage as expected since they were essentially the same tests. Overall the Mocha tests were implemented, and code coverage was setup in approximately one hour which was really quick compared to Jasmine and Jest.

Analysis and Comparison

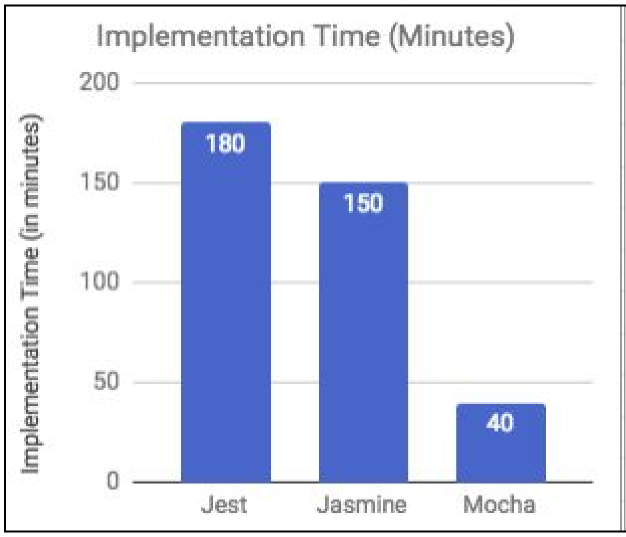
This section covers the various metrics collected throughout the process of implementing these tests. The raw data collected can be found here and will be described in detail below.

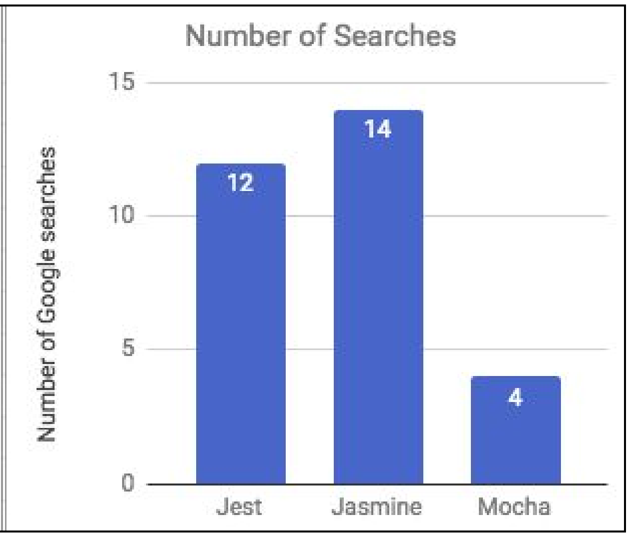
Setup Time: This measures the time from when I started to attempt to write tests using a particular test framework to when I had the first test running. Jest took the longest since it was the first test framework that I used. Problems encountered with Jest such as function scoping, issues with require and modules.export, would have also been faced with Jasmine and Mocha had they not already be solved. These results should take into consideration that there is a bias against Jest due to the fact that it was implemented first.





Implementation Time: As described earlier, this metric measures the amount of time it took to write the full suite of tests for both the Board and the Game class. Once again, Jest appears to take the longest since it was the framework that I worked with first. The test suite took surprisingly little time to implement in Mocha since I was essentially able to mostly copy paste the tests over from Jasmine and then rename a few APIs to work correctly in Mocha. All problems that would have been encountered during this process had been solved while writing the Jest and Jasmine tests.

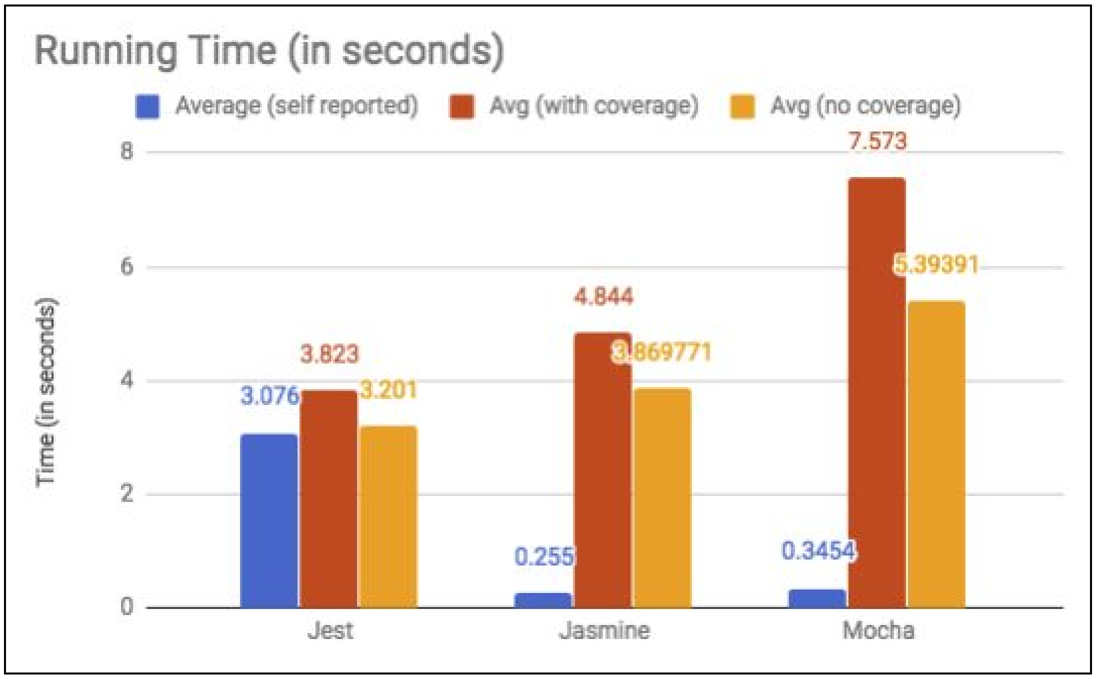




Running Time: This was the time taken to run the entire test suite. As mentioned in the README file on GitHub, the entire test suite could be run from the command line. Each of the test frameworks self reported the amount of time it took to run the tests. However I also wanted to measure the run time including the setup and teardown time for the framework itself so I wrote a script to run each of the test suites 5 times, and measure that average time taken to run the scripts. The findings here were a little surprising. The self reported run time differed greatly from the one measured by the script for Mocha and Jasmine. Jest was more honest about the amount of time it took to run the tests. I also measured the time taken to run the tests without the coverage calculations since I imagined that would add some time. The results are summarized in the table below (all values are in seconds).

	Run 1	Run 2	Run 3	Run 4	Run 5	Avg1	Avg2	Avg3
Jest	3.27	2.763	2.931	3.339	3.077	3.076	3.823	3.201
Jasmine	0.237	0.246	0.233	0.299	0.26	0.255	4.844	3.870
Mocha	0.393	0.495	0.253	0.248	0.338	0.345	7.573	5.394

Table Legend:
- Avg1 is the average running time of 5 runs as reported by the framework itself.
- Avg2 is the average running time of 5 runs as reported by the python tool. This includes running the code coverage tool as well.
- Avg3 is the average running time of 5 runs as reported by the python tool. This excludes the code coverage step and hence is significantly less that Avg2 for all the frameworks.



These above findings indicate that Jest seems to take the least amount of time to actually setup and execute the entire framework of tests. The difference is even larger when coverage measurement is included. This is in line with what I expected based on research conducted during the midterm paper. What I found most surprising here, was the large difference in the self reported and actual runtime for Mocha and Jasmine test suites.

Number of Google Searches: I measured the number of searches I had to use to answer various questions and address issues as they came up while implementing tests for each of these frameworks. Once again Jest had the most number of searches since it was the first test framework that I used. I compiled a list of all the results from my searches and the resulting list of articles and websites can be found here. A lot of searches eventually ended up pointing to API documentation for the framework itself which was extremely convenient. All these frameworks have a strong online community so finding answers to issues that others have encountered was relatively easy. Since all three frameworks use Node, once a solution had been found for one of the frameworks, it usually applied to the other two as well.

Code Coverage: I measured code coverage for tests written in all three test frameworks. This was accomplished using the built-in coverage tool for Jest, and using Istanbul for Jasmine and Mocha. Across the board I was able to achieve similar code coverage metrics of 96% line coverage since the tests were essentially exactly the same, just using different APIs.

Lines of Code: This was a somewhat uninteresting metric for this project at least. The complete suite of tests for each test framework used of very similar number of lines of code, since they mostly involved simply changing API calls from one suite to the other. The code is extremely similar in every other way so all tests were between 290 and 300 lines of code in all three frameworks.

Conclusion

The project was able to successfully implement a web application to test. Complete tests suites in three JavaScript testing frameworks, Jest, Jasmine and Mocha were implemented to provide close to 100% code coverage. Code coverage measurement tools were integrated with each of the three frameworks. The implementation times varied greatly because a single developer implemented tests in all three frameworks and was able to copy paste and carryover learnings from one framework onto the next. This greatly skewed the objective results and metrics measured in favor of Mocha which was implemented last. However, personally I would strongly recommend that beginner JavaScript developers use Jest by using this project as a guide to solve any configuration and setup issues that they may encounter. Jest offers and all in one package and does not require the developer to integrate with other JavaScript libraries like

Sinon, Istanbul or JSDOM. Further it comes with a built-in coverage tool, which makes code coverage measurement easier to set up. Finally, the ability to mock out a constructor proved to be extremely useful and saved me a lot of time, which I was unable to do in Jest or Mocha. Overall, even though the data shows that Jest took the longest time to implement, it makes up for it by offering a complete, feature rich testing framework with no external dependencies. Along with the robust developer documentation and strong online community, this makes Jest a remarkably easy to use testing tool.

What I Learned

I learned a lot about open source JavaScript testing frameworks through this project. I write a lot of JavaScript at work, and we write robust tests as well but those tests are written using internal proprietary testing tools due to which I had no experience with APIs available for JavaScript testing outside a work environment. I learned that Jest offers a fully integrated testing solution, while Mocha is the most customizable framework since it requires users to integrate with third party tools for assertion, mocks, fakes, DOM testing and such functionality. I found Jest to be the easiest to use since it came with all the functionality needed to write any of the tests that I wanted to. I also learned that a majority of my time was spent figuring out how to configure and setup these test frameworks, in a large part due to the fact that they are run using Node. Node isn't built to run in browser so it was a bit of a challenge figuring out how to write source files that could be tested using Node while simultaneously being run successfully in a browser. This required some unintuitive changes to the source code that I would not have otherwise expected. Through this process I realized why a lot of novice or beginner JavaScript developers continue to rely on manual testing. I hope that this project acts as a guide and sample for new JavaScript programmers trying to unit test their code.

Demo

For my demo I was able to show the following in class:

- A demo of the actual tic-tac-toe game. I demonstrated a few different scenarios that may occur such as either player winning, a tie game, attempting to play on an already occupied square and any other "error" scenarios. This demonstrated both that the code was well tested and the unit tests were effective in catching any bugs that may have existed before the demo.
- A demo of executing the Jest test suite. The time it takes to run these tests should be similar to the "running time" metric in my analysis of the various test frameworks .
- The Jasmine and Mocha test suites were still in progress when I demo-ed in class.

- I intentionally introduced a bug in one of the source files and re-ran the test suite to show that it failed.
- I modified one of the tests to be incorrect and re-ran the test suite to show that it failed.

Deliverables

- Source code for the tic-tac-toe web application
- Demo of the tic-tac-toe web application
- Source code for the unit tests in Jest
- Source code for the unit tests in Jasmine
- Source code for the unit tests in Mocha
- Instructions to install Node and run the above tests
- List of links from various Google searches
- Tool for calculating the actual running time of each test command
- Instructions to run the timing tool
- Raw data for metrics collected throughout this project
- Config file that shows how to configure and run the test frameworks using Node

Tools

The following third party and open source frameworks were used for this project.

- Node: https://nodejs.org/en/
- Jest: https://facebook.github.io/jest/
- Jasmine: https://jasmine.github.io/
- Mocha: https://mochajs.org/
- Sinon: http://sinonjs.org/
- JSDOM: https://github.com/jsdom/jsdom
- Istanbul: https://istanbul.js.org/