k. Wrote Unit Tests for their code.

BE developers wrote and executed unit tests for their code. BE also included a suite of Security tests to round out our prototype testing efforts.

BE Safe is an internet facing application utilizing common open-source technologies. We feel it is important in all development projects, especially those facing the internet, to be built securely. We had our security engineer derive a list of best practices specific to the technologies in use on this prototype and automate the testing for these best practices. More details on our approach to security testing is below.

BE Front End developers ran tests manually prior to propagating their code through the project. We also used end users to test and validate our front end features. Had this been a larger effort that released for production use BE would have automated the UI testing.

BE Backend developers wrote tests for our APIs using Frisby.js. BE utilized test driven development concepts whereby tests were developed and initially failed and then made to pass by writing the associated features. Those tests were rerun by developers prior to propagating their code through the project as well as executed by Circle CI during the build and release process.

It should be noted that BE testing identified an issue with the OpenFDA API whereby our tests and application were executing faster than the API could respond.

BE Safe API Unit Testing in Frisby.js

The BE Safe API unit tests were written in frisby.js, an extension of jasmine-node. Frisby.js is a simple, easy to use tool that allows the developer writing the scripts to call API endpoints and define what the expected response should look like. If the response from the API matches what's defined in the test, then the test passes, but if the response doesn't match the test's definition, then frisby.js returns a failure. For example, in the test shown below frisby.js will call the BE Safe Drug Recalls API with an expected status code of 200 OK. As long as the Drug Recalls API returns the 200 status code for a successfully executed HTTP GET, then the test will pass.

```
console.log('Testing Recalls GET')
frisby.create('Testing Recalls GET')
    .get('http://be-safe.elasticbeanstalk.com/#/api/drugs?brand_name=ibuprofen&search_type=recalls')
        .expectStatus(200)
.toss();
```

Figure 1. Firsby.js results

Frisby.js tests are executed on the command line, and tests in the same JavaScript file are all executed in quick succession when that file is called. The command line will log how many tests pass and fail. Should a test fail, frisby.js will log the reason the test failed. No log is provided for tests that pass. Below are examples of a successfully executed test and a failed test.

```
C:\Users\Jeff\Projects\unittests-3>npm test

> rppr.pdip@2.3.0 test C:\Users\Jeff\Projects\unittests-3
> jasmine-node ./tests/integration-tests_spec.js

Testing Open FDA Recalls GET
Testing Open FDA Adverse Reaction Events GET
Testing Recalls GET
Testing All Email Subscription and Unsubscription
....

Finished in 0.499 seconds
4 tests, 4 assertions, 0 failures, 0 skipped
```

Figure 2. Successfully Executed Test

```
  2) Frisby Test: Testing All Email Subscription POST
       [ PUT http://localhost:3000/api/drug/subscriptions/all?jefferson.baker@b
uchanan-edwards.com ]
   Message:
     Expected 404 to equal 200.
   Stacktrace:
     Error: Expected 404 to equal 200.
    at null.<anonymous> (C:\Users\Jeff\Projects\18f dev\node_modules\frisby\lib\
frisby.js:493:42)
    at null.<anonymous> (C:\Users\Jeff\Projects\18f dev\node_modules\frisby\lib\
frisby.js:1074:43)
    at Timer.listOnTimeout (timers.js:110:15)
```

Figure 3. Failed Test

There is one caveat to the unit test collection written for BE Safe. Frisby.js may return false negatives when the unit tests are executed in an environment where there is high latency. It is possible for the tests to time out in these situations and return an Internal Server error HTTP Status Code of 500. We believe that the quick succession of the calls to the Open FDA API may also be a contributing factor to these false failures.

Addendum A – Security Best Practices

BE built a prototype focused on engaging internet based people. Security is always important but becomes especially critical for an internet facing application. BE took some of our best practices for securing an internet application that uses the technologies we selected, documented them here

and built tests to validate our adherence to these standards. Should this application be slated for production use, we would revisit this list, update as necessary, and build more tests to validate our adherence to these practices.

Mandatory Practices

- The `eval` statement should not be used: It opens a backdoor that allows potential injection attacks. The following statements execute the `eval` command, and thus should also not be used.

```
new function(String)

setTimeout(String, 2)

setInterval(String, 2)
```

- The `.exec()` command should not be used. Instead use `.execFile()`.
- Cookies should have the `HttpOnly` and `secure` flag set in order to prevent cookie theft.
- The application or framework should not insert untrusted data into the DOM or allow an HTML escape before inserting.
- Cross Site Request Forgery protection should be enabled using `app.use(express.csrf())`.
- `express.bodyParser()` should be used with caution. It can be used to generate infinite amount of temporary files and crash the server.
- User input should be escaped wherever possible.
- When subscribing to an email system, users should be distributed a globally unique identifier so that they cannot be unsubscribed by other users.

Suggested Practices

- Strict mode is more secure and thus should be used when possible.
- The node application should not be ran with superuser permissions (for obvious reasons).
- Content Security Policy is an additional layer of security to help prevent attacks, including cross site scripting and data injection attacks. It should be enabled.
- The X-Powered-By header can be used for attackers to gain insight on the engine in which the application is built. It should be disabled using `app.disable("x-powered-by")`.
- `retire.js` is a node module that helps detect vulnerabilities in your application. It should be used if possible.
- Always check the expected type when validating the input.

- Input fuzzing should be incorporated into test suites to find issues.

- Hardened error handling is absolutely necessary, `try/catch`, `domain`, and `cluster` should be used.
- Inputs generated by the user should not be used for `regex`.