Fulfilling a RESTful Commitment

Test Suite Generation for Swagger-based REST APIs

Noah Dietz

Computer Science Department

College of Engineering

California Polytechnic State University

June, 2016

Table of Contents

Abstract
1 Introduction
    1.1 The Ecosystem
    1.2 The Framework
    1.3 The Motivation
    1.4 The Project
2 Contract-based Development
    2.1 Standardize Expectations
    2.3 A Proposed Cycle
3 Project Development
    3.1 Technology Stack
        3.1.1 Handlebars.js
        3.1.2 Mocha.js
        3.1.3 Async.js
    3.2 Design of Project
        3.2.1 JavaScript API
        3.2.2 Swagger-node Integration
        3.3.3 Parsing Hierarchy
    3.4 Development Cycle
    3.5 Individual Contributions
        3.5.1 Data Population
        3.5.2 Sequence Test Generation
4 Lessons Learned
    4.1 API Descriptions are Extremely Rich
    4.2 Testing Data-Driven APIs Comes With Baggage
    4.3 Asynchronous Nature of HTTP Requests
Conclusion
Generated Code Samples

Abstract

An application programming interface (API) can be the most integral part of a product or service. APIs drive businesses and innovation in all aspects of technology and they rely on descriptive frameworks to detail what they have to offer. These descriptive frameworks are a rich source of information on a variety of levels, such as usage, best-practices and even basic implications towards the underlying implementation. The motivation of this project during my internship was to exploit this wealth of information in order to create a description-based, or contract-based, unit test generation tool specifically for Swagger-based REST APIs and to encourage contract-based development practices. Beyond my internship, I have developed this tool as my senior project with the goal of broadening its use-case beyond unit testing and into integration testing. Thus, swagger-test-templates is a Node.js based tool designed to consume Swagger-described APIs and generate comprehensive unit tests and specified integration test scenarios.

1 Introduction

1.1 The Ecosystem

The Representational State Transfer (REST) architecture is one of the most popular design patterns for developing web services today. The essence of this design paradigm is centered around the stateless transfer of data explicitly through the use of HTTP methods[1]. Entire systems are exposed via REST APIs for developers to consume and exploit in a plethora of applications and environments. The resources consumed by developers are organized by “directory structure-like URIs”[1] or endpoints, with functionality derived from an endpoint’s supported REST operations. These include GET, DELETE, HEAD, POST and PUT, each of which are designed to be used for specific parts of an API’s implementation. However, with a robust set of resources comes an extensive list of endpoints, their operations and their specific usages. This becomes a difficult artifact to maintain without a standardized descriptive framework.

1.2 The Framework

The goal of an API description framework is to provide a standardized method of defining your API in order to reduce development time of the API, make collaborative development more succinct and, most importantly, to define a contract that developers consuming the API can rely upon for their apps. There are a large variety of description frameworks freely available for use, each having their own unique advantages and disadvantages. The framework of choice in this project was Swagger, a YAML-based API

[1] http://www.ibm.com/developerworks/library/ws-restful/

documentation architecture[2]. Since the development of this project, the open-source project of Swagger has been renamed Open API, but for the purpose of this paper, it will be referred to as Swagger. It is an architecture that is ideal for contract-driven development rather than post-implementation documentation. The Swagger framework has blossomed into a large community with a variety of tools built around its ability to provide rich definitions of APIs, from client library generation and server-code stubbing to live documentation and API development environments. This is what makes it the perfect tool for those looking to build an API from the inside out, with the API definition being the guiding factor throughout the development process.

1.3 The Motivation

This project was intended to join the Swagger ecosystem of tools which lacks any comprehensive test suite generation. The absence of such a tool means that an API developer must make a plethora of choices about their testing framework at the inception of their project that will affect them as their API grows. As such, testing can be rushed or ignored completely so as to save time and resources. When you have a community of developers or a business relying on the contract you've defined in a Swagger document, it's imperative that the API be well-tested. This is the concept of contract-first development, a process based on test-driven development of APIs, addressed in further detail later in this paper. The overarching motivation is that we want to encourage frequent testing, early and throughout the development of APIs through the exploitation of Swagger descriptions.

[2] "Swagger." – The World's Most Popular Framework for APIs. N.p., n.d. Web. 03 May 2016. <http://swagger.io/>.

1.4 The Project

Introducing, swagger-test-templates, a Node.js module designed to consume Swagger APIs and generate a comprehensive and customizable test suite. The majority of the project was developed by a team that I was a part of at Apigee for an internship. The team consisted of myself and another intern doing the development, with two full-time employees of Apigee acting as our coaches. As my senior project, I have forked the original project and extended the functionality of the module to further our overarching motivation. I will discuss the development of the project as a whole, the contributions I made during my senior project and what I learned about API development pedagogy.

2 Contract-based Development

Referring to an API description framework or document as a “contract” is not a new idea. With the many different API description frameworks comes the same idea of defining a contract that developers can depend on. On the API Blueprint website, probably the second most popular API description framework, they view such frameworks as “a contract for an API.[3]” By consuming an API, a developer is agreeing to the set usage defined in the description, providing the proper parameters and expecting only the defined responses.

2.1 Standardize Expectations

By building an API with the description-first approach, all involved parties will be on a level playing field. This means that the front-end developers know how to format their user input into requests. The backend developers know what information they have to work with from the client and what they must return as a response. Externally, all client apps that might leverage your API for their own core functionality know exactly what they can and can’t get from your API, and you as the developer supporting them know exactly to what they have access. API Blueprint says it best, when you’ve defined your API in a framework, “everybody can test whether the implementation is living up to the expectations set in the contract.[2]”

2.3 A Proposed Cycle

All of what has been said can be worked into a viable development cycle. The API description becomes the guiding force of the project’s progress. All discussion happens around

[3] "API Blueprint | API Blueprint." API Blueprint | API Blueprint. N.p., n.d. Web. 03 May 2016. <https://apiblueprint.org/>.

this document and until a resolution is reached, no code is written for the implementation. Following a decision on the API resources to be added, the tests should be created. This will give the consumers of the API (frontend developers, other application components, etc.) something to base their own concrete usage off of, while also encouraging the backend developers to test their implementation constantly. Once the tests are in place, the actual backend implementation can be written. Testing should happen often and throughout this part in order to ensure adherence to the API's description. Now what I've just described sounds a lot like test-driven development, and it is. The difference is that there is a live artifact guiding production and testing. Rather than referencing a static test document for a single piece of functionality, we use a single, ever-changing document that implies test scenarios as well as implementation requirements. This is contract-based development.

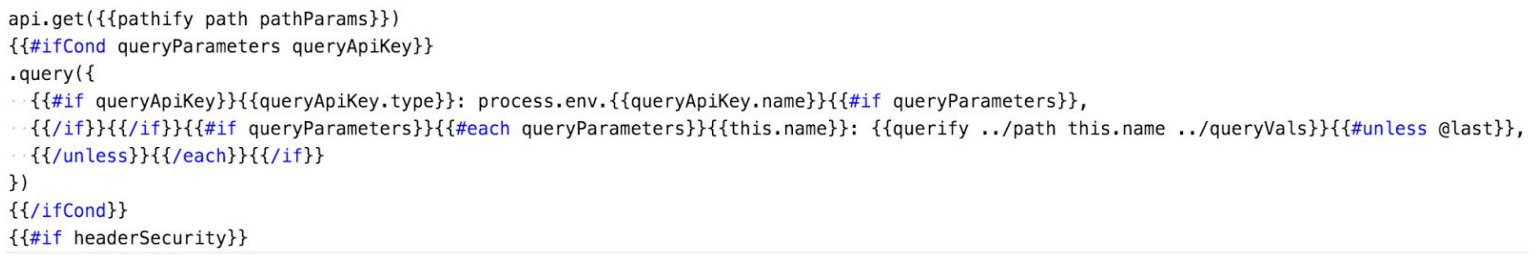
3 Project Development

3.1 Technology Stack

Swagger-test-templates is a Node.js project developed purely from JavaScript. We utilized a variety of node modules in our implementation, but there were two that were integral in the design of the project, Handlebars.js and Mocha.js.

3.1.1 Handlebars.js

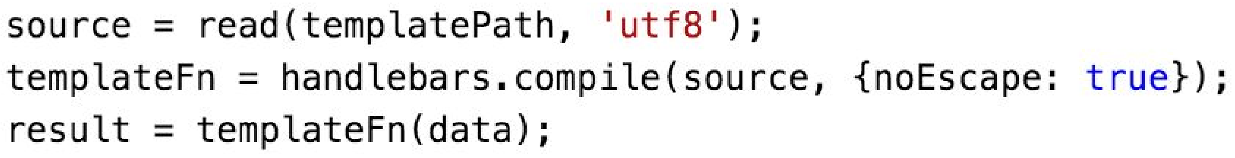
This library is an immensely powerful code templating and generation library. It is the core of the swagger-test-templates module. By defining a set of templates in *.handlebars* files, we are able to exploit a set of “compilers” exposed in the JavaScript API. An example of our templates is shown below, in Figure 1.



```
api.get({{pathify path pathParams}})
{{#ifCond queryParameters queryApiKey}}
.query({
  {{#if queryApiKey}}{{queryApiKey.type}}: process.env.{{queryApiKey.name}}{{#if queryParameters}},
  {{/if}}{{/if}}{{#if queryParameters}}{{#each queryParameters}}{{this.name}}: {{querify ../path this.name ../queryVals}}{{#unless @last}},
  {{/unless}}{{/each}}{{/if}}
})
{{/ifCond}}
{{#if headerSecurity}}
```

Figure 1. Example of Handlebars.js template

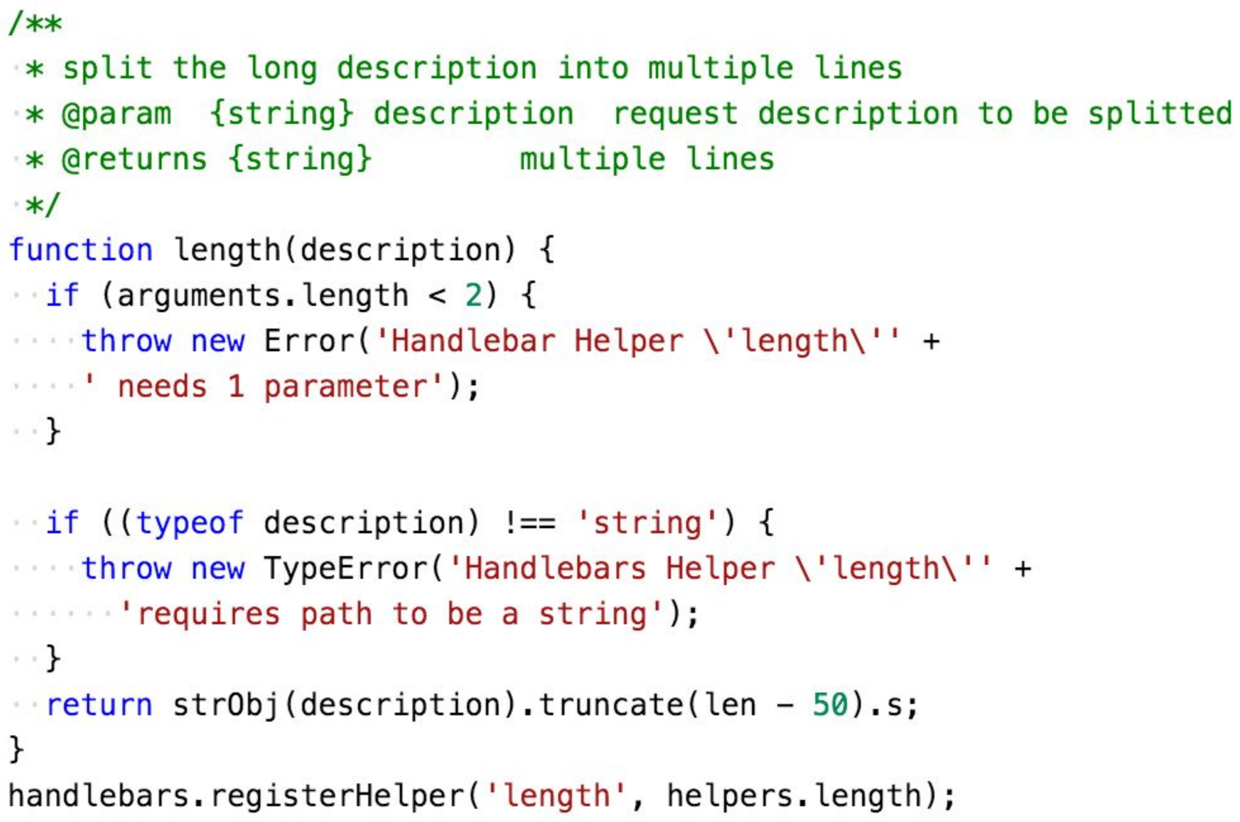
Using Handlebars.js allows us to statically define the general structure of the resulting generated code while dynamically defining the code that is specific to the API being parsed. The templates are extremely useful, but as the project grew, they became increasingly difficult to maintain. We implemented linting tests to ensure that the generated code was formatted in a standardized manner and was readable. Linting tests are a way to enforce a specific style and also prevent simple code formatting errors that could create errors at runtime (i.e. multi-line strings that aren’t properly broken up). Strict linting tests combined with dynamic code generation and a multitude of different templates meant we spent a lot of time curating the templates between development iterations as we implemented new features. Using the static templates in the project was easy.



```
source = read(templatePath, 'utf8');
templateFn = handlebars.compile(source, {noEscape: true});
result = templateFn(data);
```

Figure 2. Example of Handlebars.js JavaScript API

As shown in Figure 2, the developer can merely compile the template at any point prior to use and it produces a function that can be reused and maps any given data into the corresponding variables in the template. The data is passed in as a simple JavaScript object and the output is a string-form of the generated code, which can be written to file later. Not only is the code generation process with Handlebars.js quite simple, it is also quite customizable. The library supports custom helper functions for use in the templates. This allowed us to write JavaScript functions that could be used at generation-time to reformat code, populate data supplied at runtime or implement complex conditional logic. Figure 3 is an example of how we registered a helper function to break up long strings into multiple lines based on a configurable line length.



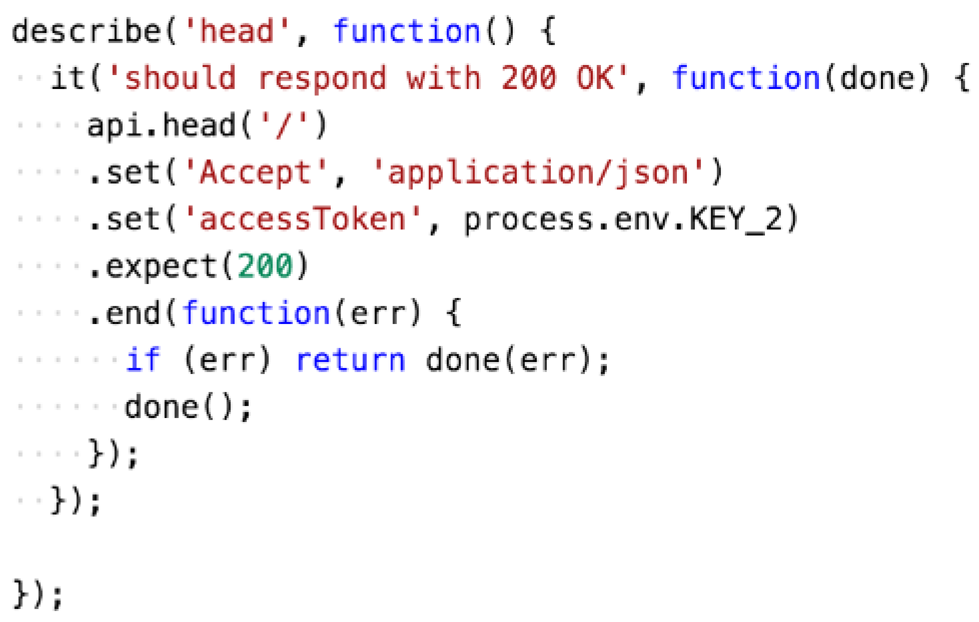
```
/**
 * split the long description into multiple lines
 * @param  {string} description  request description to be splitted
 * @returns {string}        multiple lines
 */
function length(description) {
  if (arguments.length < 2) {
    throw new Error('Handlebar Helper \'length\'' +
    ' needs 1 parameter');
  }

  if ((typeof description) !== 'string') {
    throw new TypeError('Handlebars Helper \'length\'' +
      'requires path to be a string');
  }
  return strObj(description).truncate(len - 50).s;
}
handlebars.registerHelper('length', helpers.length);
```

Figure 3. Example of Handlebars.js custom helper function

We took full advantage of this, registering several helper functions. While we did have approximately thirty templates to maintain, this tool would not have been possible without Handlebars.js.

3.1.2 Mocha.js

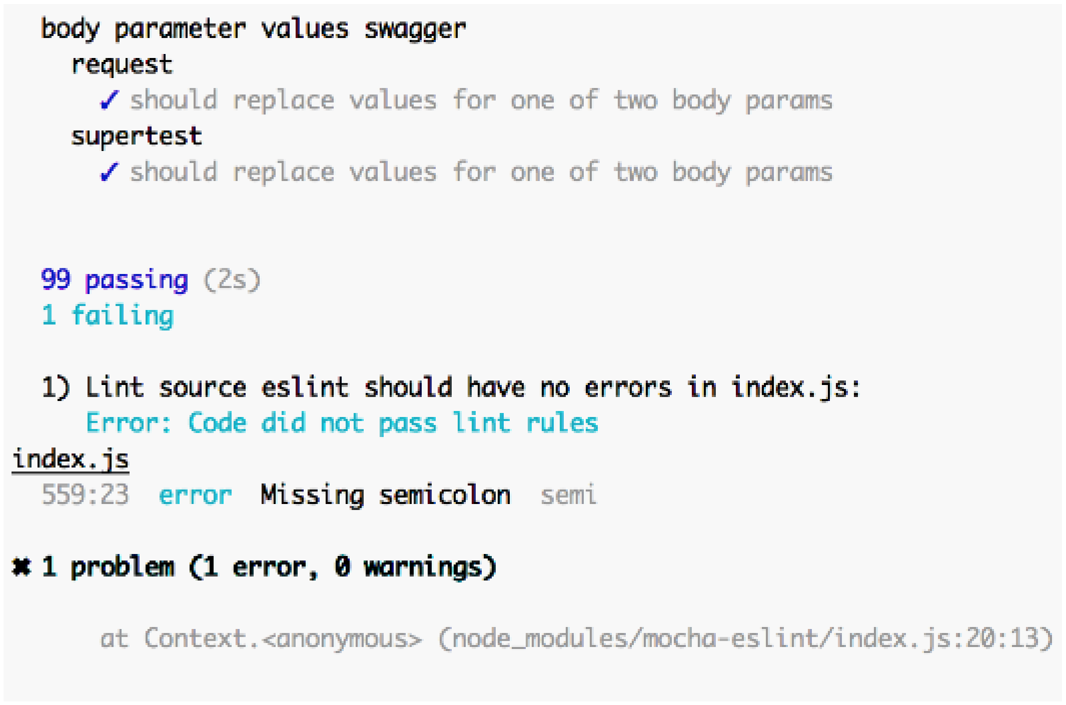
Creating a testing framework to run tests and track results is a non-trivial task, which is why we used Mocha.js. This framework is highly customizable, extremely easy to use and produces a rich, descriptive report of test results. Mocha.js was not only the foundation of our generated test suite, but also our own test cases. The highlight of Mocha.js with respect to testing HTTP APIs is the asynchronous nature of it’s test runner. We can define a set of unit tests to be run asynchronously, each with their own callback that allow it to serially report the results. This is crucial for making asynchronous HTTP requests, so as to not block. The simplicity of making tests asynchronous with Mocha.js is shown below in Figure 4, which contains example output from swagger-test-templates.



```
describe('head', function() {
  it('should respond with 200 OK', function(done) {
    api.head('/')
    .set('Accept', 'application/json')
    .set('accessToken', process.env.KEY_2)
    .expect(200)
    .end(function(err) {
      if (err) return done(err);
      done();
    });
  });

});
```

Figure 4. Example of a Mocha.js unit test

The *done* callback used throughout this example is what allows this test to run asynchronously. Simply removing the *done* callback from the test makes it synchronous. In the same example, you can see how intuitive it is to use this framework. It’s quite simple to provide rich descriptions for your tests along with an extremely organized test hierarchy manifested in the multiple



```
  body parameter values swagger
    request
      ✓ should replace values for one of two body params
    supertest
      ✓ should replace values for one of two body params


  99 passing (2s)
  1 failing

  1) Lint source eslint should have no errors in index.js:
     Error: Code did not pass lint rules
index.js
  559:23  error  Missing semicolon  semi

✖ 1 problem (1 error, 0 warnings)

      at Context.<anonymous> (node_modules/mocha-eslint/index.js:20:13)
```

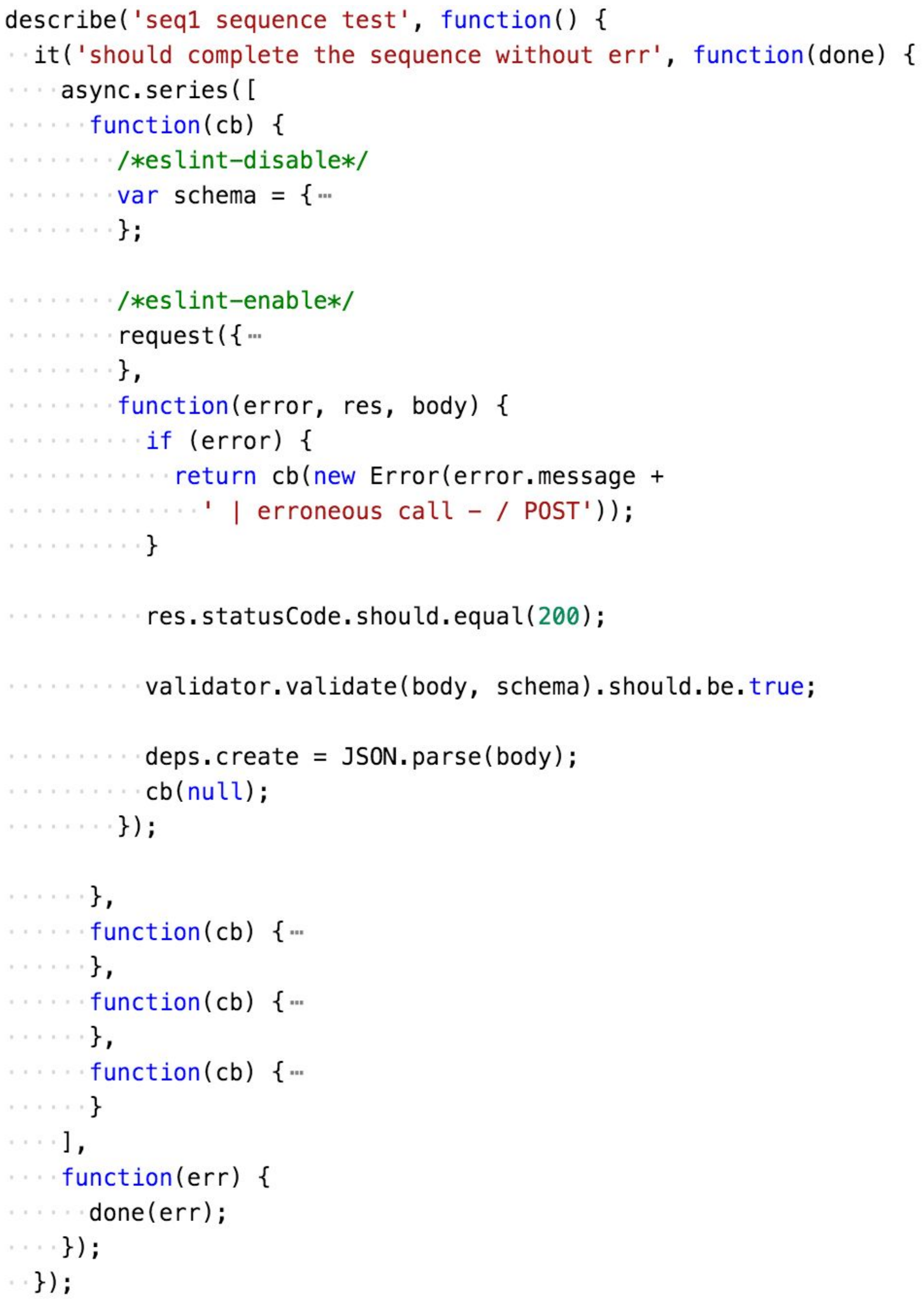
Figure 5. Example of Mocha.js test report output

describe() and *it()* blocks. The *describe()* blocks define the various levels of test organization. The *it()* blocks define individual tests, and would appear as the leaves on the test suite tree. The resulting report, as shown in Figure 5, provides insightful information on successes and failures. Being such a robust framework, Mocha.js gave us a solid foundation upon which we could generate a rich and descriptive test suite from an equally informative Swagger API description.

3.1.3 Async.js

While continuation passing and asynchronous programming is commonplace in JavaScript, especially in Node.js, it introduces a challenge in implementing a series of interdependent, asynchronous calls. Used in the development of integration test generation, Async.js provides a robust API for organizing collections of asynchronous tasks. I leveraged the Async.series utility to synchronize the execution of HTTP request sequences. It was incorporated into the Mocha.js framework by wrapping the series in a test clause, evaluating the result at each step of the series, with errors exiting early and failing the test.

So, as seen in Figure 6, it was really a matter of wrapping several HTTP requests in a single function call, this is the power of Async.js and it’s why I chose to use it.



```
describe('seq1 sequence test', function() {
  it('should complete the sequence without err', function(done) {
    async.series([
      function(cb) {
        /*eslint-disable*/
        var schema = {⋯
        };

        /*eslint-enable*/
        request({⋯
        },
        function(error, res, body) {
          if (error) {
            return cb(new Error(error.message +
              ' | erroneous call - / POST'));
          }

          res.statusCode.should.equal(200);

          validator.validate(body, schema).should.be.true;

          deps.create = JSON.parse(body);
          cb(null);
        });

      },
      function(cb) {⋯
      },
      function(cb) {⋯
      },
      function(cb) {⋯
      }
    ],
    function(err) {
      done(err);
    });
  });
```

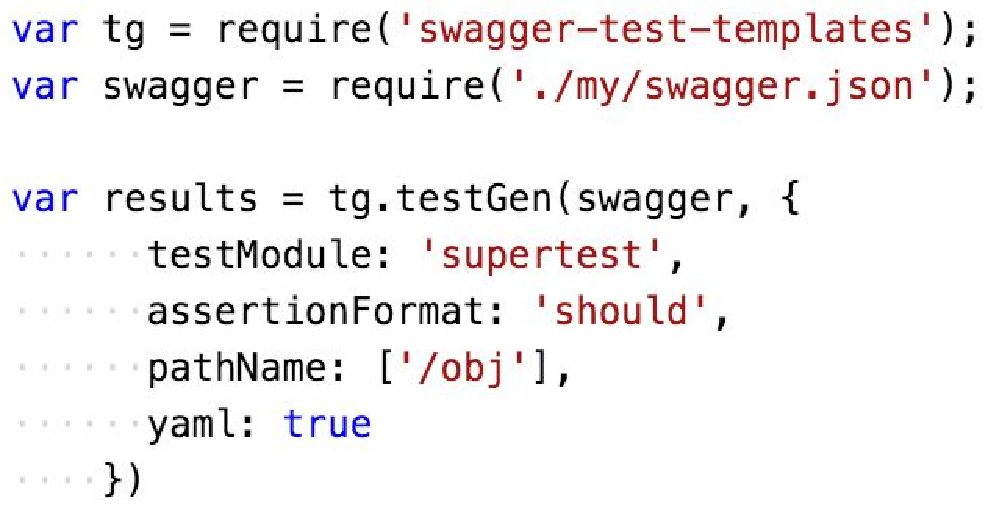
Figure 6. Usage of Async.js to synchronize an integration test

3.2 Design of Project

The swagger-test-templates tool was released as a standalone Node.js module with a Javascript API that could be used in any Node.js project. However, we decided to leverage the already existing ecosystem of tools built around the Swagger framework, specifically swagger-node. As such, there were two usage patterns we designed around, a JavaScript API and a command line interface (CLI), exposed in the swagger-node project. Examples of the generated output from swagger-test-templates can be found at the end of this document.

3.2.1 JavaScript API

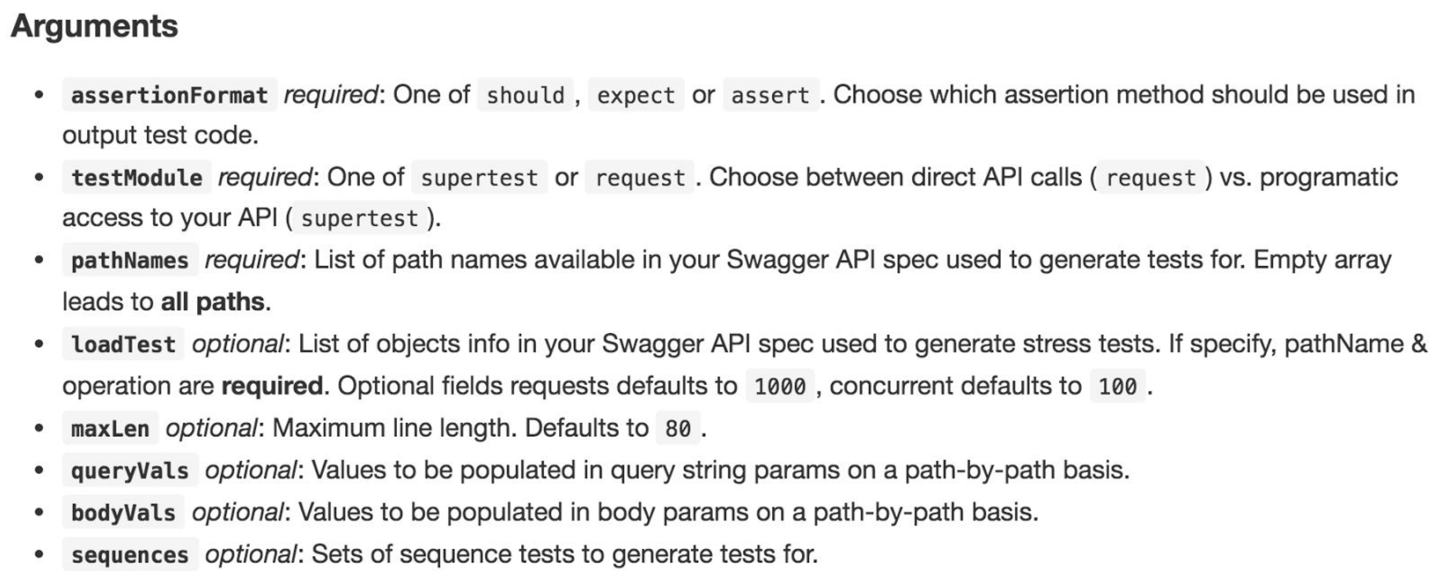
After the additions I made for my senior project, the JavaScript API is more robust, with respect to available options, than the CLI in swagger-node, because I do not have complete control over that project. Swagger-test-templates can be installed and imported like any other node module and exposes a single function. It’s usage is exemplified below in Figure 7.



```
var tg = require('swagger-test-templates');
var swagger = require('./my/swagger.json');

var results = tg.testGen(swagger, {
      testModule: 'supertest',
      assertionFormat: 'should',
      pathName: ['/obj'],
      yaml: true
    })
```

Figure 7. Example usage of swagger-test-templates JavaScript API

Notice the import of a Swagger document in JSON form. This is the REST API the module parses and generates tests for. The fact that it is represented as a JSON object is a drawback of the JavaScript API. Swagger documents are usually written in YAML, contain remote references, and benefit from validation of the document prior to translation to JSON. The object passed as the second argument is the configuration for generation, including the desired request serving module, assertion library to use, and many other options, listed here.



Arguments

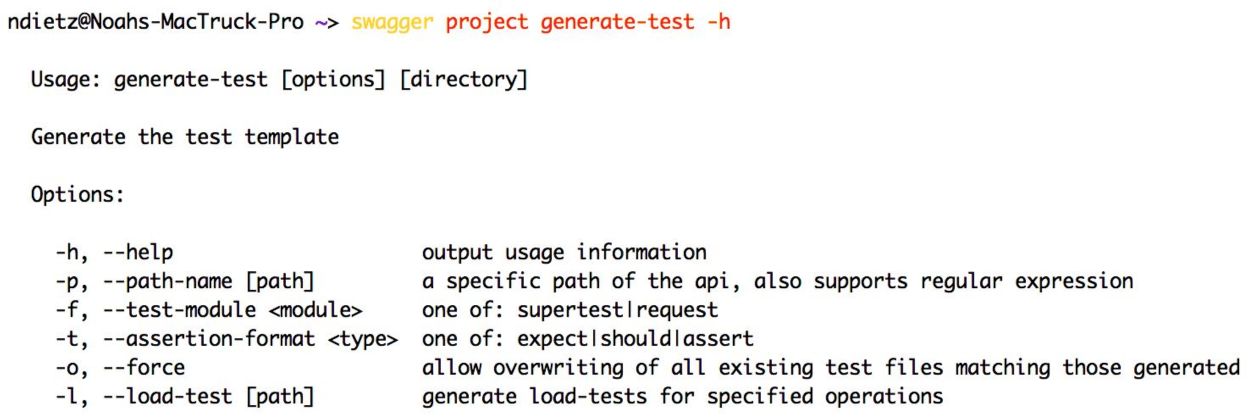
- `assertionFormat` *required*: One of `should`, `expect` or `assert`. Choose which assertion method should be used in output test code.
- `testModule` *required*: One of `supertest` or `request`. Choose between direct API calls (`request`) vs. programatic access to your API (`supertest`).
- `pathNames` *required*: List of path names available in your Swagger API spec used to generate tests for. Empty array leads to **all paths**.
- `loadTest` *optional*: List of objects info in your Swagger API spec used to generate stress tests. If specify, pathName & operation are **required**. Optional fields requests defaults to `1000`, concurrent defaults to `100`.
- `maxLen` *optional*: Maximum line length. Defaults to `80`.
- `queryVals` *optional*: Values to be populated in query string params on a path-by-path basis.
- `bodyVals` *optional*: Values to be populated in body params on a path-by-path basis.
- `sequences` *optional*: Sets of sequence tests to generate tests for.

Figure 8. Available options for the JavaScript API

As you can see in Figure 8, the JavaScript API supports heavy customization of the generated test files, but still has its drawbacks such as its verbose configuration object. The *sequences* and all of the data population options can end up being huge objects and become clumsy to work with in a Node.js script, especially when trying to regenerate specific path/operation tests.

3.2.2 Swagger-node Integration

The aforementioned faults of the JavaScript API are remedied with the integration of swagger-test-templates into the swagger-node CLI. Swagger-node provides tooling that allows for easy implementation and iteration of a purely Node.js server implementation of your Swagger-described REST API. Along with setting up middleware and routing services, it has live validation of your Swagger document and can translate to JSON format. The project has a CLI that among other things, allows you to generate a test suite or individual test files with swagger-test-templates. The only drawback is that the CLI doesn't expose as many options for test customization. It's usage and available options are shown below in Figure 9.



```
ndietz@Noahs-MacTruck-Pro ~> swagger project generate-test -h

  Usage: generate-test [options] [directory]

  Generate the test template

  Options:

    -h, --help                    output usage information
    -p, --path-name [path]        a specific path of the api, also supports regular expression
    -f, --test-module <module>    one of: supertest|request
    -t, --assertion-format <type> one of: expect|should|assert
    -o, --force                   allow overwriting of all existing test files matching those generated
    -l, --load-test [path]        generate load-tests for specified operations
```

Figure 9. Demo of swagger-node CLI usage of swagger-test-templates

Regardless of the fact that there are less options, the new features would be easily integrated and exposed in such a format. The added benefit of integrating with swagger-node is the existence of an already established developer community, in a project that encourages the contract-based development process. Thus, we know the context in which swagger-test-templates is used with swagger-node, a context that aligns with our desired API development practices.

3.3.3 Parsing Hierarchy

While there are two different usages for this module, there is only one implementation. Input for test generation, as we know, is a JSON object representation of our Swagger document. To traverse this document, we looked at the REST organizational pattern as a hierarchy. The top level of information is the global space, whose attributes apply to each child where the child does not specify a value for that attribute. This is to say, at any parent in the traversal tree, it's properties are cascading, when applicable. Beyond the global space, we get into the generation of individual test files, based on a single REST resource, known as the "path" level. Each path has a set of supported REST operations, making the "operation" level, which also make up a sublevel within the resource's test file as a new *describe()* block. Within an operation, there are a set of expected responses and their schema. The "response" level is the final level of the traversal. An individual *it()* block is made for each defined response, which is an individual test. This traversal hierarchy creates a comprehensive unit-test suite, and is visualized below in Figure 10 with an example of an API's traversal tree.

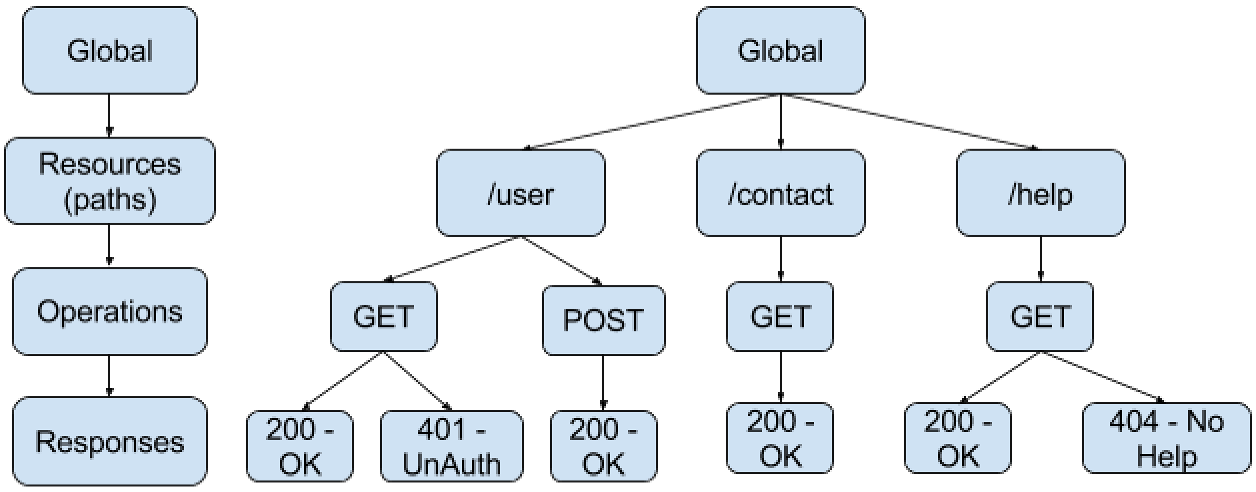


Figure 10. Swagger-test-templates traversal hierarchy

With this traversal structure, we were able to define the path of execution, identify the information at each level that was pertinent, and implement piece by piece, adding on more detailed information to our output as we iterated through the tree. This was the basis for our development cycle as well.

3.4 Development Cycle

As a team of two developers, and while I was working on my own, we followed a strict workflow. With git as our choice of version control for the project, we used a branch and merge strategy. For us, this meant that the master branch was the source of truth and was always functional. Any feature addition would be implemented in its own branch, tested, reviewed as a pull request and merged, with every outstanding branch rebasing on top of the newly merged commits. We combined this workflow with a test-driven development approach. We would start by writing mock output for the feature to be added, which we would diff against the actual output, as the expected results, for testing. Following the mock output, we would write the test harness for it, and begin implementation. We added a continuous integration tool to our branch and merge workflow, known as Travis CI. Upon opening a pull request or pushing a commit to a branch, Travis is configured to build our project and run the given test suite remotely, reporting any errors to the repository, so we know what branches are safe to merge.

Our workflow, combined with our design of the traversal hierarchy, made for an extremely efficient development cycle. With the main path of execution defined, we were able to implement a basic version of the test generation for a very limited set of REST properties. Once this was functional, we could branch off and independently develop new features, expanding our support for the entire Swagger spec, gathering more information from each traversal level.

3.5 Beyond the Internship

As I said earlier, the design and foundational development of this project was done in a team as part of a summer internship. For my senior project, I followed the same design pattern and workflow, but on my own fork of the main project. The state of the project prior to my continued development for school was that it was strictly a unit-test generation tool. Post-generation, the tests could be run, but would surely fail if there were any required request parameters. The output contained what we called “obvious indicators,” which highlighted parts of a request that needed to be fulfilled by the developer, post-generation. I personally addressed this shortcoming by implementing a configurable data population feature. After that, I tackled a bigger issue by implementing integration test, or “sequence” test, generation. Both of these features have been committed to my master branch, but not back to the original repository, one that I don’t exclusively control. The original repository was made aware of my developments through the issues I opened there to introduce my ideas.

3.5.1 Data Population

The Data Population feature essentially allows users to specify values that some set of variables used in their API calls, keyed by their names in the Swagger document. When a named variable appeared during test generation, instead of being given an obvious indicator, it would be given the values specified in the configuration. This feature applies to multiple request parameter types, including path, query string and body parameters. Each are independently defined and implemented. The query string and body parameters are specified in the configuration on a path-by-path basis, with each path having their own object containing their parameters and corresponding values. The path parameter value population usage is slightly different in that a single variable could be present in multiple resources, so the values are merely provided on a name-by-name basis and applied whenever found. The path parameter value population was also an open source contribution, so it was not part of my senior project.

But as the inspiration for the other parameter sections, I felt it should be introduced. Here is the usage for both body and query string parameter value population.

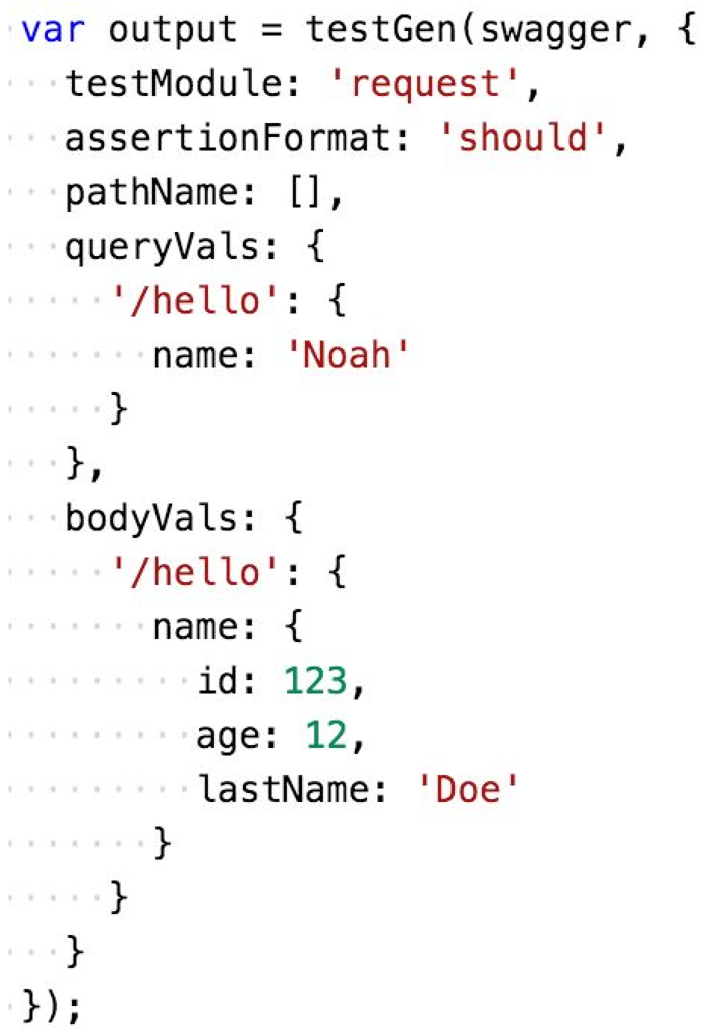
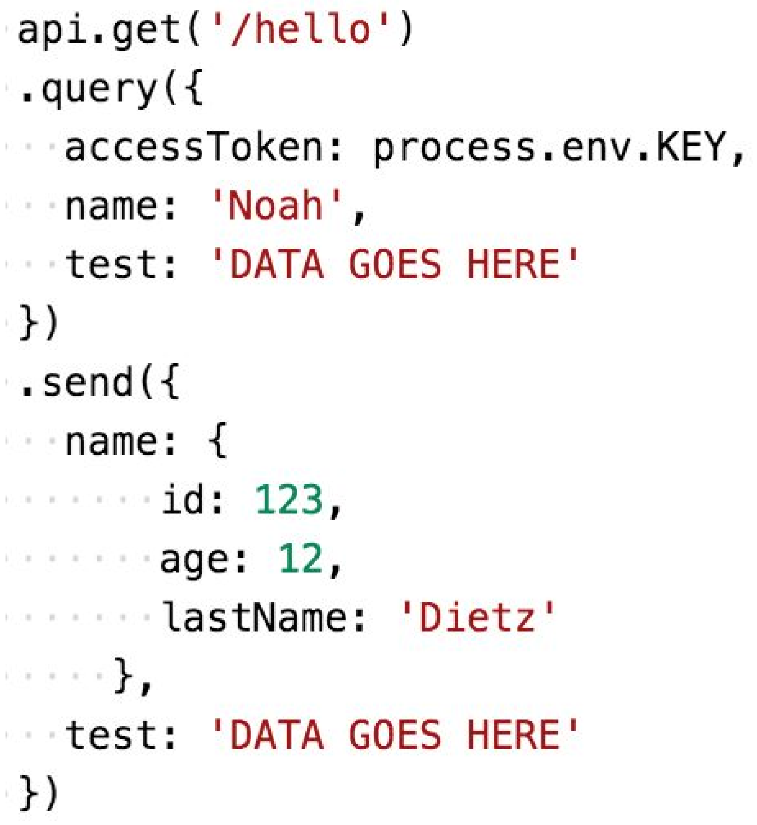

```
var output = testGen(swagger, {
  testModule: 'request',
  assertionFormat: 'should',
  pathName: [],
  queryVals: {
    '/hello': {
      name: 'Noah'
    }
  },
  bodyVals: {
    '/hello': {
      name: {
        id: 123,
        age: 12,
        lastName: 'Doe'
      }
    }
  }
});
```

Figure 11. Example usage of body & query string value population

The example in Figure 11 specifies that for the '/hello' resource, the "name" parameter in the query string will take the value 'Noah', while the parameter of the same name in the body will take the corresponding object as it's value. A subsequent request could also contain variables that were not assigned a value, shown below in Figure 12.



```
api.get('/hello')
.query({
  accessToken: process.env.KEY,
  name: 'Noah',
  test: 'DATA GOES HERE'
})
.send({
  name: {
      id: 123,
      age: 12,
      lastName: 'Dietz'
    },
  test: 'DATA GOES HERE'
})
```

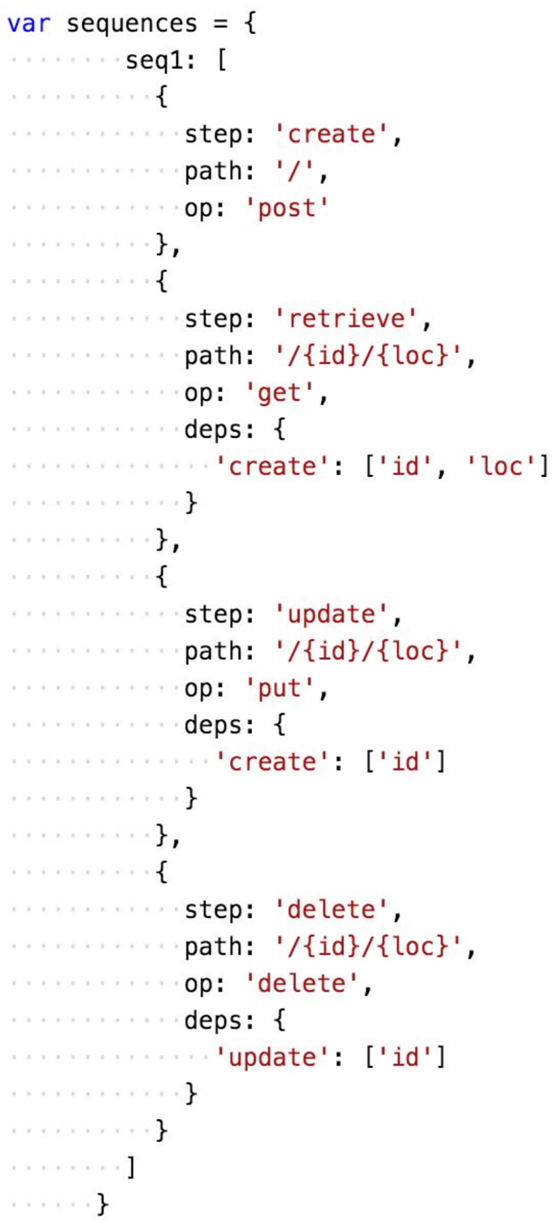
Figure 12. Example of resulting data population in generated test request

This data population feature was developed using the Handlebars.js custom helper functions. Each parameter type has their own helper function that is used in the templates. This feature wasn't originally implemented during the inception of the project for a simple reason. As a team, we felt that generating mock data or allowing for configurable parameter values would be too "magical." In other words, it would make it harder for the user to determine what work was actually being done, in case it needed to be debugged. The intention was to create an easier way to develop tests, but still have a high-touch relationship with the tests. This is important when you need to debug your API implementation, understanding what the test does, how it works and what the source of the failure could be. However, it came up in discussion with the developer community that such a data population feature would be quite useful and cut down on development time for large APIs. Thus, I made it an optional feature, leaving the original functionality intact, but allowing for the developer to harness the power of our comprehensive traversal pattern.

3.5.2 Sequence Test Generation

What I am referring to with “sequence” testing is commonly known as integration testing. Integration testing is a testing paradigm that aims to reflect the real use-cases of a product. In the case of APIs, this is a sequence of API calls that would replicate a user’s consumption of your API. For example, a website’s user might create a new profile, retrieve their user information, update it, then after some use, delete the profile. This specific example is a usage paradigm commonly known in the REST world as Create-Retrieve-Update-Delete (CRUD). This was the guiding idea for the sequence test generation feature. Such real use-cases are important to test, because they help define the overall user experience of your product. The use of an API won’t be strictly limited to a single request here and there. It will manifest itself as a series of calls that detail a user’s journey within the application. Identifying and ensuring their stability is going a step beyond contract-based development and even touching on behavioral-driven development practices. This allows other organizations within a business to weigh-in on development of an API, which goes a long way towards securing its success and that of dependent applications or services.

Sequence test generation introduced a big challenge in handling inter-test dependencies among asynchronous requests where the use of a response value from one or more requests was a necessary parameter of tests later in the sequence. For example, when a new piece of data is created, it might have an ID associated with it and be contained in the response of the request that created it. This ID might be needed later to retrieve, update or delete the newly created data, thus introducing an inter-test dependency. To solve this this problem, I instantiated a global object, keyed by the sequence step, that each completed API call updates with the response object. The dependencies object is visible to all requests in the sequence, and with the knowledge that one will finish before another is started, we can be sure that any necessary values will be in place prior to the start of a dependent test. The usage of the sequence test generation feature is crucial to the use this shared response object. Figure 13 is



```
var sequences = {
        seq1: [
          {
            step: 'create',
            path: '/',
            op: 'post'
          },
          {
            step: 'retrieve',
            path: '/{id}/{loc}',
            op: 'get',
            deps: {
              'create': ['id', 'loc']
            }
          },
          {
            step: 'update',
            path: '/{id}/{loc}',
            op: 'put',
            deps: {
              'create': ['id']
            }
          },
          {
            step: 'delete',
            path: '/{id}/{loc}',
            op: 'delete',
            deps: {
              'update': ['id']
            }
          }
        ]
      }
```

an example.

Figure 13. Usage of sequence/integration test generation

The *step* property is used by that request in the sequence to key it's response object in the shared object and used by dependent steps to access that response. The *path* and *op* properties identify the API resource and the desired operation to use for the step of the sequence. Finally, the most important property, *deps* identifies the part(s) of the sequence that this step depends upon to complete its own request along with the parameters from the response object that are necessary. A user might define multiple sequences or just a single one. The implementation for this feature was done with an ideal scenario in mind, meaning that each request should return a successful response and use Content-Type JSON.

Developing this feature was challenging in that it defied the traversal path we had designed the entire project around. As mentioned earlier, the ideal scenario requires that each request return a successful response and use a specific Content-Type. This could possibly leave parts of an API sequence untested. Instead of comprehensively navigating the tree, it picks and chooses the resources defined in each step of the sequence. Thus, in order to leverage the existing code, I had to forgo some of the levels of the hierarchy and the information gleaned from them. New templates were created, as the structure of these tests were drastically different from that of the unit tests. There is a single *describe()* level with a nested *it()* clause containing the Async.series. Each defined sequence gets its own file. In the end, the tests are still quite rich and descriptive, even having forgone some levels of information.

After developing sequence test generation, I had to be sure that the Async.series structure I used would actually work. This meant I had to create a real API, implement it and test it. Now, because I built the feature around the CRUD use-case, I had to implement a database system in order to write and read data via API calls. As an aside from the original project, I decided to implement a basic database service daemon for MongoDB. This node module, released as leash.js, was simply wrapped around the code that started a Node.js server in order to ensure a MongoDB service was started. With this in place, I could develop a Swagger API, generate test cases (both unit and sequence) and then implement the backend logic without the struggle of maintaining a database instance at the same time. While this might have been a distraction from the main project, it was crucial for the validation of my proposed sequence testing implementation, which worked as expected.

4 Lessons Learned

Over the duration of my internship and the continued development in my senior project, I gained an immense amount of knowledge. From raw JavaScript skills, to API design, I was exposed to a wide variety of information. While my technical skills and knowledge grew tremendously, the most impactful pieces I found in non-coding lessons.

4.1 API Descriptions are Extremely Rich

The first and most important lesson learned was that API descriptions, specifically ones made within a framework, are a wealth of information. As a developer, the API description enables the creation of live documentation. This allows consumers of the API to make mock API calls directly in the documentation, because the expected request parameters and response definitions are known to the system. An example of this is Swagger UI, a dynamic, live-documentation builder that, when added to a project, helps serve testable documentation of your Swagger-based API. Similarly, an implementation of server logic can be generated for you by parsing a Swagger document. Swagger Code Generator is an open source project that does this and more, by also generating client libraries from your server. The existence of such a tooling environment is a testament to the richness of API description frameworks. One could design their entire API in a single document and generate a majority of the code that implements it.

4.2 Testing Data-Driven APIs Comes With Baggage

When I implemented a basic API to test my sequence test generation feature, I ran into an issue of test data collision. If there was a failure during the sequence test, any data created by that sequence was persistent in the database, even though the series had failed. This means that, while swagger-test-templates with integration testing is a very powerful tool, it is incomplete. The missing piece is database build-up and teardown during test runs. Without this, a data-driven API would have nondeterministic behavior over the course of development, assuming there was some data left behind by test cases. Ideally, the solution for this would involve instantiating a new or clean database instance for each sequence test, such that if one series fails, the others will not collide with the orphaned data, if they share API calls. This is an issue that all REST API testing frameworks have to deal with, but it was a new challenge for me, discovered through the development of my own API.

4.3 Asynchronous Nature of HTTP Requests

This was a lesson I learned quickly. At the beginning of my work with this project, I didn't fully understand asynchronous testing. Assuming each request would run in series just by calling them one after the other, I couldn't determine why tests that might have an inter-dependency would fail or succeed in a non-deterministic manner. It was then that I realized that testing requests over the network would take special consideration for how the tests are designed and run. For example, if one test creates a piece of data, but another running in parallel attempts to get all data expecting it to be empty, therein lies a race condition. To avoid this, tests need to be run either synchronously (with the help of Mocha.js) or in isolation (unique database/server instance for each test, which is a waste of resources).

Conclusion

The development of swagger-test-templates has been a gateway for me into the world of API design and development. Today, the project is still actively developed by a handful of open source contributors and myself. This tool is the first of it’s kind in the Swagger world, but surely won’t be alone for long. Already, there are other API contract validation and testing tools available for other description frameworks. I learned a lot about the open source community, what it takes to develop and release a successful open source project. To this date, swagger-test-templates has about 5,106 module downloads with 4 other node modules that depend on it to provide their own functionality. Credit also goes to the team I worked on for the summer at Apigee, including Prabhat Jha, Mohsen Azimi and Linjie Peng.

Generated Code Samples

I. Swagger document for target API

```
{
    "swagger": "2.0",
    "info": {
        "version": "0.0.0",
        "title": "Simple API"
    },
    "securityDefinitions": {
      "key": {
        "type": "apiKey",
        "in": "query",
        "name": "access_token"
      }
    },
    "security": [
      {
        "key": []
      }
    ],
    "paths": {
        "/": {
          "post": {
            "parameters": [
              {
                "name": "name",
                "in": "body",
                "type": "string"
              },
              {
                "name": "age",
                "in": "body",
                "type": "number"
              }
            ],
            "responses": {
              "200": {
                "description": "OK",
                "schema": {
                  "type": "object",
                  "properties": {
                    "id": {
                      "type": "number"
                    },
                    "loc": {
                      "type": "number"
                    }
                  }
                }
              }
          }
         }
        },
        "/{id}/{loc}": {
          "parameters": [
           {
            "name": "id",
            "in": "path",
            "type": "number"
           },
           {
            "name": "loc",
            "in": "path",
            "type": "number"
           },
           {
            "name": "id",
            "in": "query",
            "type": "number"
           }
          ],
          "get": {
            "responses": {
              "200": {
                "description": "OK"
              }
            }
          },
          "delete": {
            "responses": {
              "200": {
                "description": "OK"
              }
            }
          },
          "put": {
            "parameters": [
             {
              "name": "newId",
              "in": "body",
              "type": "number"
             }
            ],
            "responses": {
              "200": {
                "description": "OK",
                "schema": {
                 "type": "object",
                          "properties": {
                            "id": {
                              "type": "number"
                            }
                          }
                        }
                    }
                }
            }
        }
    }
}
```

II. Generated sequence test

```javascript
'use strict';
var async = require('async');
var chai = require('chai');
var ZSchema = require('z-schema');
var validator = new ZSchema({});
var supertest = require('supertest');
var api = supertest('http://localhost:10010'); // supertest init;
var deps = {};

chai.should();

require('dotenv').load();

describe('seq1 sequence test', function() {
  it('should complete the sequence without err', function(done) {
    async.series([
      function(cb) {
        /*eslint-disable*/
        var schema = {
          "type": "object",
          "properties": {
            "id": {
              "type": "number"
            },
            "loc": {
              "type": "number"
            }
          }
        };

        /*eslint-enable*/
        api.post('/')
        .query({
          accessToken: process.env.KEY
        })
        .set('Accept', 'application/json')
        .send({
          name: 'DATA GOES HERE',
          age: 'DATA GOES HERE'
        })
        .expect(200)
        .end(function(err, res) {
          if (err) {
            return cb(new Error(err.message +
              ' | erroneous call - / POST'));
          }

          validator.validate(res.body, schema).should.be.true;
        deps.create = res.body;
        cb(null);
      });

    },
    function(cb) {
      api.get('/' + deps.create.id + '/{loc PARAM GOES HERE}')
      .query({
        accessToken: process.env.KEY,
        id: deps.create.id
      })
      .set('Accept', 'application/json')
      .expect(200)
      .end(function(err, res) {
        if (err) {
          return cb(new Error(err.message +
            ' | erroneous step - /{id}/{loc} GET'));
        }

        res.body.should.equal(null); // non-json response or no schema

        deps.retrieve = res.body;
        cb(null);
      });
    },
    function(cb) {
      /*eslint-disable*/
      var schema = {
        "type": "object",
        "properties": {
          "id": {
            "type": "number"
          }
        }
      };

      /*eslint-enable*/
      api.put('/' + deps.create.id + '/{loc PARAM GOES HERE}')
      .query({
        accessToken: process.env.KEY,
        id: deps.create.id
      })
      .set('Accept', 'application/json')
      .send({
        newId: 'DATA GOES HERE'
      })
      .expect(200)
```

```
        .end(function(err, res) {
          if (err) {
            return cb(new Error(err.message +
              ' | erroneous call - /{id}/{loc} PUT'));
          }

          validator.validate(res.body, schema).should.be.true;

          deps.update = res.body;
          cb(null);
        });

      },
      function(cb) {
        api.del('/' + deps.update.id + '/{loc PARAM GOES HERE}')
        .query({
          accessToken: process.env.KEY,
          id: deps.update.id
        })
        .set('Accept', 'application/json')
        .expect(200)
        .end(function(err, res) {
          if (err) {
            return cb(new Error(err.message +
              ' | erroneous step - /{id}/{loc} DELETE'));
          }

          res.body.should.equal(null); // non-json response or no schema

          deps.delete = res.body;
          cb(null);
        });

      }
    ],
    function(err) {
      done(err);
    });
  });
});
```

III. Generated single path test

```javascript
'use strict';
var chai = require('chai');
var ZSchema = require('z-schema');
var validator = new ZSchema({});
var supertest = require('supertest');
var api = supertest('http://localhost:10010'); // supertest init;

chai.should();

require('dotenv').load();

describe('/', function() {
  describe('post', function() {
    it('should respond with 200 OK', function(done) {
      /*eslint-disable*/
      var schema = {
        "type": "object",
        "properties": {
          "id": {
            "type": "number"
          },
          "loc": {
            "type": "number"
          }
        }
      };

      /*eslint-enable*/
      api.post('/')
      .query({
        accessToken: process.env.KEY
      })
      .set('Accept', 'application/json')
      .send({
        name: 'DATA GOES HERE',
        age: 'DATA GOES HERE'
      })
      .expect(200)
      .end(function(err, res) {
        if (err) return done(err);

        validator.validate(res.body, schema).should.be.true;
        done();
      });
    });

  });
```