CptS 422 Project: Deliverable 2

BATS: Business Analytics Tracking Service

Austin Marino, Nicholas Kent, Joseph Cunningham, Cole Bennett

Table of Contents

Implementation

Source Code

Deliverable Two: https://github.com/austinmm/WSU_CptS_422_BATS/tree/Deliverable_Two

Languages Used

Frontend

- **Languages**: Html/CSS, JavaScript
- **Reasoning**: This will be necessary for simple demo purposes of how anyone could leverage our analytics API and setup proper tagging for components on their frontend.

Backend

- **Languages**: Node.JS (JavaScript)
- **Reasoning**: JavaScript is the language we used to build and deploy our server. Node makes running a server with simple tasks extremely easy and will help us speed up development so we can focus on features and proper design.

Testing

- **Language**: Node.JS (JavaScript)
- **Reasoning**: It only made sense to use the same language as the code we are testing.

Frameworks/Technologies Used

Mocha, Chai, Sinon

Mocha is the test framework we used for Node.JS for testing. Chai is used for asserting tests which our Ci/CD pipeline will use to check that all tests pass. Sinon allows us to stub and mock, simplifying the process of writing our tests.
https://github.com/sinonjs/sinon (Mocking & Stubbing)
https://github.com/mochajs/mocha (JavaScript testing framework)
https://github.com/chaijs/chai (JavaScript assertion library for testing)

Express

Express is the most common framework users use with Node JS. It further simplifies creating a server instance, managing it, and deploying it.
https://expressjs.com

MySQL

The MySQL framework simplifies the process of connecting to a SQL database and querying it. We created a relational database on hosted on Azure due to ease of use and getting slightly more sophisticated information based off of our data without needing to setup a local database.
https://www.mysql.com
https://azure.microsoft.com/en-us

Other Development Tools Used

NPM

NPM is a package manager for JavaScript and the runtime environment Node.js. We used this package manager to maintain our project dependencies.
https://www.npmjs.com

Postman

Postman is a great tool for manual testing. It allowed us to easily invoke our endpoints and also has the potential to run automated test scripts for future deliverables.
https://www.getpostman.com

Visual Studio Code

There are many capable IDEs to choose from but we mostly used Visual Studio Code to due its many extensions/add-ons and developer friendly environment.
https://code.visualstudio.com

GitHub

We utilized GitHub for version control and pull requests to perform code reviews for merging change requests into our master branch.
https://github.com/austinmm/WSU_CptS_422_BATS

Test Outcomes

Token and Tag Endpoints

The majority of the testing performed on our token and tag endpoints involve testing status codes and response values, and verification through assertions. This testing strategy discovered numerous faults in our system, examples include:

- DELETE /api/tokens

        - The response codes for this endpoint were not working as expected due to inadequate implementation or invalid conditional checks. Thus, the unit testing revealed numerous changes that need to be made in order for the endpoint to work as expected.
- POST /api/tags/:name
    - While trying to write tests for this route, it was found that the logic in this route was more complex than necessary, so we simplified it by reducing the number of queries it executes.
    - It still contains three calls to executeQuery(), so we had to extend our initial strategy of testing routes with single executeQuery() calls to include stubbing sequential calls to executeQuery() for each test case. For example in the first test case we provide expected values for executeQuery() for each of the three calls so that we can cover all branches in the logic.
    - In addition, writing the tests for this route exposed a few bugs of not handling different possible return values from executeQuery(). This prompted us to handle cases where values returned from executeQuery() are undefined (null) or empty.

Middleware

The middleware was not easy to test because it does not emit any status code or response to test for. Instead, we had to pass in mocked response and request object values and check how they were altered after passing through middleware. To perform this kind of testing, we had to change up how our middleware was structured; since its original form would have been almost impossible to test. These required structural changes ended up working out great as it made our code easier to understand and work off of.

In addition to structural changes, the middleware test also exposed vulnerabilities and bugs that had to be fixed. One of the first issues brought to light was how we check for Bearer.

Authentication

The conditions written were not catching all possible authentication styles and values and therefore required more specific cases to catch all scenarios thrown its way. Additionally, the main feature of flagging requests as correctly authenticated or not was not performing its job correctly with each request. This was due to a lack of processing performed on fundamental response values. So we added additional checks and string conversions to ensure no values were being handled/registered incorrectly.

Summary

For this deliverable, we divided up all of the functions in our source code evenly among the four of us and we all utilized the same frameworks. We have an NPM script (npm test), which calls

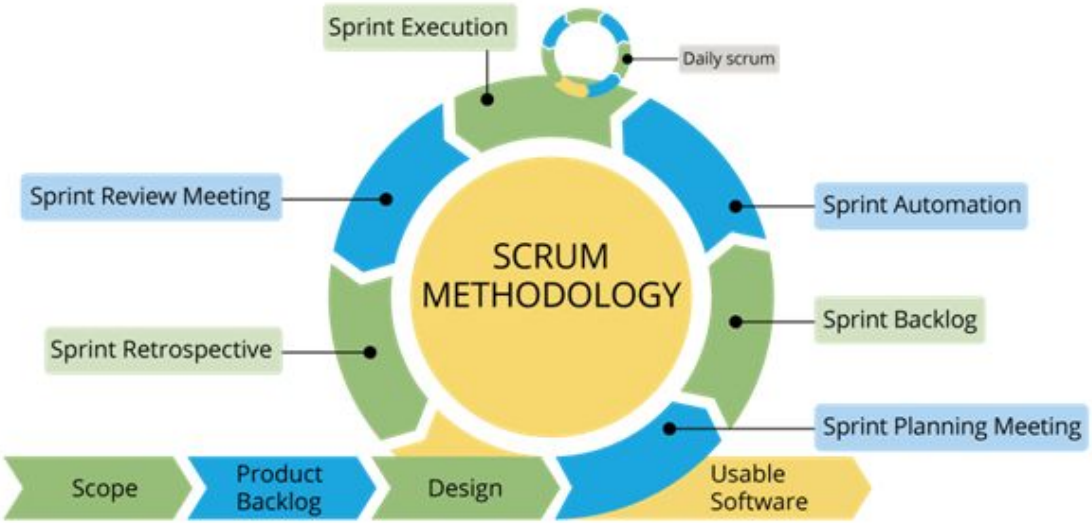
the Mocha testing framework to execute all tests in our server/test directory. Using this script, we were able to verify that all of our test cases passed during development.

Overall we did not have any issues and the frameworks we selected worked quite well for testing our Node server. The only challenge was researching how to properly use Sinon to stub out specific functions in external dependencies and in our own JavaScript modules.

Software Process

For this project, we've decided to utilize the Scrum methodology. Scrum is a type of agile process which means that this method is suitable for small projects such as this one and is also very flexible, allowing for the requirements to change if they need to. It's easy to understand and facilitates team collaboration and organization as we will work in week sprints and evaluate at the end of each one. We also perform code reviews via GitHub for each change made in our source code. This involves branching off of master for a feature or bug and then opening a pull request which must be reviewed and approved by all team members before being merged back to master.

For deliverable two we have only had one sprint which was two weeks long and consisted of us researching JavaScript testing frameworks and writing unit tests. Before the sprint started we had a meeting to organize it and divide tasks equally among the members of our team. At the end of the sprint we had another meeting for discussion and review before our deliverable was submitted.



Project Activities

Meetings

Our team conducted weekly meetings, every Monday from 5:00pm to 6:00 pm, where we discussed current issues that needed to be addressed and any future work that was required. During this time, we also conducted code reviews and assigned a future task for each member of the team to ensure that everyone was doing an equal share of the work. We documented who was assigned what tasked in our Slack channel which we regularly communicated on a nearly daily basis. Please visit here to view our messages on Slack.

10/7/19

Discussed deliverable two requirements and created this document to be filled out as we progress. Discussed what frameworks we will use for testing our Node.js app.

10/14/19

Sprint start. We split deliverable two work into tasks for each team member to work on for the next few weeks.

10/21/19

No meeting. Deliverable two work and messaging through Slack.

10/28/19

Went over deliverable two requirements and discussed what has to be completed by each team member before the due date.

10/30/19

Sprint end and deliverable two submission. We reviewed all of the tests written and read through our documentation.

Milestone Report

Joseph Cunningham

During the second deliverable, our group wrote unit tests to test the functions of our project.I was tasked with writing tests that interact with the database, specifically the tags table. All of the

time I spent during this deliverable was put into writing two GET /tags unit tests, which both work with the information in the table. In order to write the unit tests, I utilized a Javascript test framework named Mocha and a BDD/TDD assertion library named Chai. I first had to learn the framework in order to get a better understanding of what I was writing and how to write it, then i was able to successfully write the unit tests. All of the following tests were written in the file tagsTest.js:

- GET /tags
    - This test function checks to see if a list of all tags in the TAGS_TABLE are return successfully when no specific name is given.
- Get /tags/name
    - This test function checks to see if a specific tag in the TAGS_TABLE is returned when a name is given.

Cole Bennett

For deliverable 2, I wrote unit tests for two functions and two routes in the BATS API server. I utilized Mocha, Chai for assertions, and Sinon for stubbing. All of these tests involved stubbing out the MySQL library's createConnection() function, which is invoked in our db library to establish a MySQL database connection. In addition, our db's executeQuery() function which interacts with the MySQL library (which is stubbed out in these tests) is also stubbed out. The executeQuery() stub returns expected values that are defined for each test case so all paths of the functions/routes being tested can be covered. Here are the functions/routes I wrote unit tests for this deliverable:

- check_token_existence - baseTest.js
    - This function queries the database to check if the given token exists.
    - I had to check different cases where results from the mocked executeQuery() function are valid, non-existent, or malformed.
- check_organizational_existence - tokensTest.js
    - This function queries the database to check if there exists a token which is associated with the given organization name. Similar to my unit tests for check_token_existence, I mocked out executeQuery() providing different results to cover all cases.
- GET /tokens - tokensTest.js
    - This route retrieves all tokens and their information such as organization name and date issued.
    - I have two test cases for this route, which include checking if tokens are found (200 status code), and if no tokens are found (404 status code).
- POST /tags/{name} - tagsTest.js
    - This route creates a tag or updates an existing tag, and then inserts a new interaction associated with the created or updated tag.
    - executeQuery() is called multiple times in this route, so I had to mock a sequence of calls to executeQuery() to reach desired cases in my tests.

Apart from contributing these unit tests to the deliverable, I also helped clean up the source code (both tests and implementation). I found that the initial version of our tests included warning errors as they were trying to connect to a MySQL database, so I stubbed out the necessary functions in the MySQL node library to make our tests run more smoothly.

Nicholas Kent

The work for deliverable 2 was setting up unit tests and setting up the libraries for testing. Part of setting up the testing was also creating a template for the team to use to build their tests off of. Within the actual code the key factor was the fact that the tests needed to be for http calls so to make this work we needed to stub the functions and mock the final promise since the promise never properly could return. We setup unit tests for each individual component as well.

- createConnection - tokensTest.js
    - Stubbing the function that creates the initial connection that way the server won't try to connect to the database causing the application to fail.
- executeQuery - tokensTest.js
    - Stubbing the actual function for executing queries to the database. By stubbing this we can force the function to return the exact data we would expect and use it for assertions.
- checkOrganizationExistence - tokensTest.js
    - Stubbing this function and setting up the stub to react to two different scenarios to make sure it is properly tested and returns different values.

Austin Marino

Most of the work I did for deliverable 2 revolves around the design and implementation of white and black box testing of our software system. Being that our application acts primarily as a RESTful API, the majority of our testing required a black-box style testing of the responses of HTTP calls against what we expected them to return, both in value and status code. However, most of the unit test my team and I wrote involved mocking or directly invoking internal methods/functions that an API endpoint was reliant upon, and this is when most of our white box testing took place. By finding and mocking dependant methods for each endpoint, we were forced to understand the internals of our systems at times. I wrote unit tests for three critical endpoints and middleware systems in our application.

- **Middleware - baseTest.js**
    - The middleware in a Node.js application is any methods that are provided as arguments to router.use() or app.use(). When this is done, these methods are not automatically executed on every API call before the target endpoint. Because middleware acts and performs slightly different operations than a typical endpoint, it provided a tougher challenge to test. In our case, we use middleware to flag all incoming requests as either…
        - Authorized with a valid token,
        - Unauthorized with an invalid token, or

    - No/ Incorrect type of authorization provided

  In order to ensure our middleware was working as expected, I had to test if the flags, res.locals.token and res.locals.token_id, were being set appropriately given a variety of authorization values provided in an API call. To do this, I mocked all the database responses and ensured that all the middleware methods flagged the response's local parameters correctly using assert.equals().
- **get_authorization_token - baseTest.js**
  - This unit test was smaller compared to the other two I wrote. The 'get_authorization_token' method is a simple method that takes in the authorization information provided in the header of a request in a RESTful API calls and parses that information to verify its format follows the Bearer structure. If the authorization does, then the token provided by the client is returned; if not, an empty string is returned. Thus, for my testing executed this method with a variety of possible authorization types and ensured it returned what was expected.
- **DELETE /api/tokens - tokensTest.js**
  - For this unit test, I had to ensure that the endpoint for deleting tokens was working as expected. This required that I checked all four possible status code responses…
    - 204 from successful removal of the token
    - 403 from improper token provided
    - 401 from no token provided
    - 500 from an internal error

  This endpoint did require the mocking of some middleware and database methods as it requires Bearer authorization and the ability to delete entries using SQL.