CptS 422 Project: Deliverable 3

BATS: Business Analytics Tracking Service

Austin Marino, Nicholas Kent, Joseph Cunningham, Cole Bennett

Table of Contents

Implementation

Source Code

Deliverable Three: https://github.com/austinmm/WSU_CptS_422_BATS/tree/Deliverable_Three

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.

Unit Testing

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

Integration 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

Instanbul

JavaScript test coverage tool that instruments ES5 and ES2015+ JavaScript code with line counters, so that you can track how well your unit-tests exercise your codebase.

MySQL

The MySQL framework simplifies the process of connecting to a SQL database and querying it.
https://www.mysql.com

Docker

We are using Docker to create MySQL containers for out integration testing environment.
https://www.docker.com

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

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

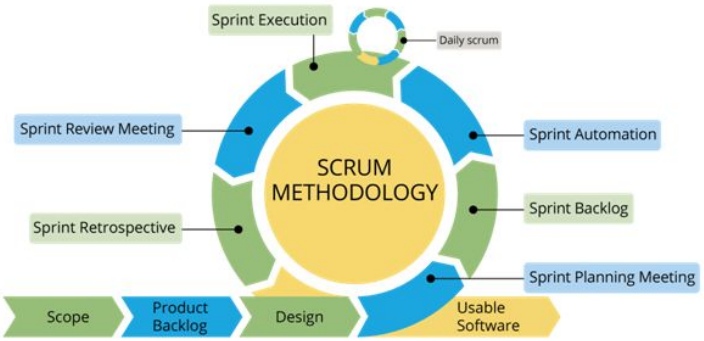
GitHub

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

Software Process

For deliverable three we continued 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 is easy to understand and facilitates team collaboration and organization as we will work in weekly 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 three, we have only had one sprint, which was two weeks long, and consisted of researching integration testing techniques for NodeJS and Express, and improving our existing unit tests from deliverable two. 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 submitting deliverable three.



Unit Test Improvements

Justification

The primary changes we made to our existing unit tests from deliverable two were composed of mostly covering our edge cases. This included making an increased number of calls to all our endpoints with a wider variety of inputs, some valid and others not. This allowed us to ensure that our routes were appropriately handling all types of request inputs and parameters and return the correct response values and status codes. The unit tests which were improved reside in the tagsTests.js and tokensTests.js files. See Austin Marino's role assignment explanation for more information on how the improvements were made.

Deliverable 2 Unit Test Coverage

91.8% overall coverage for *server/routes*. This was very good for our second deliverable, however our branch coverage was quite low for tags.js and overall we had improvements to make.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	91.1	83.05	95.24	92.55	
server	96.55	75	100	96.55	
app.js	96.55	75	100	96.55	27
server/lib	85	77.78	87.5	89.47	
db.js	85	77.78	87.5	89.47	64,65,66,92
server/routes	91.8	86.49	100	92.56	
base.js	100	100	100	100	
tags.js	85.11	66.67	100	86.96	8,9,11,12,49,50
tokens.js	94.34	93.75	100	94.34	13,14,15

Deliverable 3 Unit Test Coverage Improvements

100% overall coverage for *server/routes*. Note that we worked on improving all coverage criteria including statement, branch, function, and line coverage.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	94.36	93.22	90.91	95.34	
server	84.85	75	50	84.85	
app.js	84.85	75	50	84.85	27,47,48,49,50
server/lib	85	83.33	87.5	89.47	
db.js	85	83.33	87.5	89.47	64,65,66,92
server/routes	100	100	100	100	
base.js	100	100	100	100	
tags.js	100	100	100	100	
tokens.js	100	100	100	100	

Integration Testing

Strategy

We decided to use the **bottom-up** integration strategy. Due to the nature of how Express works and how we designed our application, we do not have traditional classes. We are treating each route in our RESTful API as an individual component, which need to be integrated to satisfy our defined user scenarios. We are treating the db.js JavaScript module as a component as well. Another reason for going with bottom-up is that we do not have an overall entrypoint into our

system which would serve as the top-level for integration. That logic is handled in the Express framework, and therefore each of our routes can be viewed as separate entry points. For this reason, the top-down approach would not fit in as well with our architecture and bottom-up is our primary strategy.

Since all of our routes and helper functions are unit tested individually with high degrees of isolation, we can write tests for verifying integration between our components (i.e. routes). We came up with integration test cases which pair various routes together for testing to ensure consistency in the underlying storage (our relational database). Integrating these pairs of components on the bottom-level of our integration allows us to then test higher levels of integration such as creating an account, followed by creating tags, and finally deleting tags. In addition, our integration tests are a mixture of white-box and black-box testing (i.e. gray-box testing) as it is necessary to view the control flows of the logic in each of our routes, and we are interacting with other routes through a local mocked HTTP calls.

Integration Test Cases

Token Endpoints

- Integration Test for Creating an Account
    - POST api/tokens/ (Create Account):
        - Request Values: {body: {organization: “test org”}}
        - Response Values: {organization: “test org”, token: <account_token>}
    - GET api/tokens/:account_token (Check for Account Existence in DB)
        - Request Values: N/A
        - Response Values: {organization: “test org”, token: <account_token>}
- Integration Test for Deleting an Account
    - First generated two random accounts in tokens table and saved one of the account’s token value (account_token).
    - DELETE api/tokens/ (Delete Account):
        - Request Values: {{header: {authentification: “Bearer <account_token>}}
        - Response Values: N/A
    - Then queried tokens table and ensured that there now existed one less account then before the DELETE route was called.

Tag Endpoints

- Integration Test for Creating a Tag
    - POST api/tags/
        - Request Values: {name: “test”}
        - Response Values: {name: “test”}
    - GET api/tags/
        - Request Values: N/A

        - Response Values: {name: “test”}
- Integration Test for Updating/Inserting a Tag
    - First, created a tag with existing values already in it (name = “custom.tag”, value = “test”, interaction = “ButtonClick”)
    - POST api/tags/ (Updating)
        - Request Values: {name: “custom.tag”, value: “testing”, interaction: “ImageSelected”)
        - Response Values: {name: “custom.tag”, value: “testing”, interaction: “ImageSelected”)
    - GET api/tags
        - Request Values: N/A
        - Response Values: {name: “custom.tag”, value: “testing”, interaction: “ImageSelected”)

Test Outcomes

Bugs Discovered

- When we were updating our unit test coverage for our tags endpoints, we uncovered a bug that we hadn't caught with our previous test coverage. What we found was when we didn't set the 'value' field for a tag entity in the JSON body portion of our request to the POST api/tags/:name endpoint, an exception would occur. This bug was fixed by changing a variable declaration from a constant to a variable.

Software Documentation Improvements

- We added comments at the beginning of each test file, briefly describing what the tests in a specific file are for. Related files:
    - baseTests.js
    - tagsTests.js
    - tokensTests.js
    - tagsIntegrationTests.js
    - tokensIntegrationTests.js
- Also, we uploaded all of our previous deliverable reports to the docs folder in our repository.

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 from deliverable two. Overall we completed integration tests to test how our routes work together to verify the user scenarios that we

defined in our first deliverable. The frameworks we selected worked well for this deliverable as well and we had no issues writing additional test cases for this level of testing. In addition, in this deliverable we incorporated the GitHub Actions tool into our repository to automatically create our testing environment, install dependencies, and run our tests. This allowed us to check if changes made in our commits passed our tests every time we made a push to the repository, adding a layer of verification to our pull request workflow to help ensure the master branch contains working code.

Role Assignments

Joseph Cunningham

- During the third deliverable, I was responsible for creating integration tests for the tags table in our database. The integration tests I wrote covered both the POST and GET of inserting/Updating a tag and Creating a tag.
- Inserting/Updating a tag
    - This function tests if a tag with variables such as “interactions” and “value” are POSTed correctly. If so, it then makes a GET request to retrieve the tag and then verifies whether or not the information is the same between the two
- Creating a tag
    - First, this function tests whether or not if there are actually any tags in the database. Then it tests if a tag is successfully created by making a POST request and then retrieving that tag with a GET request. If both are successful, it then checks to see if the tag is the same.

Cole Bennett

- For deliverable three, I initially focused on setting up the testing environment to support integration tests. We decided to go the direction of using a real MySQL database instance for our routes to interact with in our integration tests, so I made the run-tests.sh script which cleans our testing environment, spins up a MySQL Docker container, runs all of our tests, and then releases all resources. Using containers allows us to better control our testing environment and achieve consistent and expected behavior. It also alleviates the hassle of manually installing MySQL locally or having to maintain and rely on a remotely hosted instance.
- I created the *Test Create Token with check_organizational_existence* integration test case in tokensIngrationTests.js. This essentially tests the integration between our database module and the *check_organizational_existence* function (which is defined in our base.js module). We already have covered this route in our unit tests, however, we stubbed out both the database module and the *check_organizational_existence* function

in those unit tests. This test verifies that the actual implementations work together as intended.
- We also needed a way to measure our test coverage, so I researched different JavaScript test coverage tools and settled on https://istanbul.js.org. This tool provides detailed coverage output (as shown in the unit testing section of this report).
- Also, I improved the documentation of our code by adding comments at the beginning of each test file to briefly explain their purpose.

Nicholas Kent

- For deliverable three I worked on setting up the integration tests regarding tagging. As the database gets more and more involved it is important to have proper integration testing to make sure that these vital components are able to work together.
- The key to making sure that these tests were working properly was by testing the Get routes and Post routes. The initial setup for these is very identical to that of the unit tests for tags and tokens however this time in order to test that the db works we needed a docker environment that allows us to temporarily setup a sql instance for us to test then scrap the container after.

Austin Marino

- For deliverable 3, I worked on a variety of integration and unit tests. In terms of unit tests, I was in charge of increasing the existing test coverage of our tags endpoints. Before making any adjustments, the code coverage was rated at…
    - Function Coverage: 100%
    - Lines Coverage: 86.96%
    - Branch Coverage: 66.67%
    - Statements Coverage: 85.11%

  I looked into how I could better the original unit test that we wrote for deliverable two and found that most of the improvements included checking for proper handling when improper values were passed to our tags endpoints. Thus, most of my additional unit tests included checking to ensure that the status codes were properly being returned when an issue arose. One exciting event that took place during my updates was I discovered a bug in our POST api/tags/:name endpoint that would cause an exception to occur if the user didn't passed a value attribute to the body of the request. This bug was fixed by changing a variable declaration from a constant to a variable. After making my updated, the code coverage was rated at…
    - Function Coverage: 100%
    - Lines Coverage: 100%
    - Branch Coverage: 100%
    - Statements Coverage: 100%

In terms of integration tests, I was tasked with creating two token route integration tests…

- POST api/tokens/ (Create Account):
    - For this integration test, I created a new account by calling the route above and providing the organization's name in the JSON body to the request. Once I had successfully created the account, which was verified by the request-response code, I then made a GET call to our api/tokens/:organization endpoint. By testing that these two routes could successfully create and then retrieve account details, I was able to ensure they were both successfully working in tandem with their SQL scripts and data handling. Ultimately this was a test to ensure both routes worked properly to maintain our DB successfully.
- DELETE api/tokens/ (Delete Account):
    - For this integration test, I pre-populated our DB with two accounts and stored the details of one of the accounts in a variable. I then made a call to the endpoint above with the proper Bearer authentication using the saved token from one of the existing accounts in the DB. Once I had successfully deleted the account, which was verified by the request-response code, I then executed a SQL command to pull all accounts in the DB and ensured there was now one less than there was prior. By testing that the DB was correctly being altered when the tokens DELETE endpoint was successfully executed, I was able to ensure that the route interacts properly with our DB.

Project Activities

Meetings

Our team conducted weekly meetings, every Monday from 3:00 pm to 4:00 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.

11/11/19

We reviewed best practices for integration testing with NodeJS and Express. All of our findings were documented for everyone to reference later on.

11/18/19

We did more integration testing research for Express applications and began to come up with out integration testing strategy.

11/25/19

In this meeting we wrote down all of the tasks that had to be completed for deliverable three and then broke up the work among all team members.

12/2/19

More integration testing work by all team members throughout the week. We also worked on our deliverable three report and improved our unit tests from deliverable two.

12/6/19

Sprint end and deliverable three submission. We reviewed all of the integration tests written and read through our documentation to ensure we had fulfilled all requirements for deliverable three.