UNIVERSITY OF REGINA

ENSE 477: SOFTWARE CAPSTONE PROJECT

SOFTWARE SYSTEMS ENGINEERING

Workshop Enterprise Resource Planning Suite Testing Plan

AUTHORS
Jonathan Wells
200328640
Konstantin Kharitonov
200354502

SUPERVISOR
Karim Naqvi
M.A.Sc., P.Eng.

April 9, 2019

Contents

List of Figures

1 Introduction

The Enterprise Resource Planning Suite is a workshop management application designed to service the engineering workshop at the University of Regina main campus. The system was created to serve as a modern way of processing job requests sent into the shop detailed in a form called the workorder, and manage those projects from the start of the project till their completion. Each workorder has its deadlines stored onto the program's calendar, allowing for the workshop manager to review the schedule for all workorders.

1.1 Objectives

During development of this web application, the primary focus is to ensure the requests made from the frontend of the application are being properly handled by the backend and the proper data is sent back and properly displayed for the users to access. Each test outlined in this document is part of ensuring that task. The frontend should send the proper CRUD requests and loads the proper information when testing, and the backend should process the requests in the proper CRUD format.

1.2 Tasks

Each section will first be tested separately in the first stage of testing, to ensure that each section has been built properly and efficiently. Since the frontend was developed in the Vue.js environment, this secion has access to Vue Test Utils, the frameworks every own unit testing library. This library allows for unit tests to be created inside a testing program which creates expect and assert statements to analyse different statements throughout the program. These tests are run through the Jest testing framework that runs each test parallel with each other. The backend, Postman is a software that allows for manual API integration testing. The program can mannually send and receive CRUD requests from the backend, ensuring that proper data is being transfered. For this particular section, Postman sends requests with data stored as JSON objects, and with the proper integration, the objects can then be stored directly onto the database. Postman can also request data from the database which will appear as JSON objects if properly received. Standard response codes are used to notify the status of each request.

Once all tests are completed, then the backend and frontend section will be ready for combined integration. During this process, automated integration testing will be used to ensure each endpoint from the backend is being accessed by the proper modules in the frontend.

1.3 Scope

Testing of the ERP suite has three main stages that need to be completed, with each being able to be divided into further substages. With the frontend testing section, each page has its own set of individual tests with regards to viewing the page on multiple devices, requesting data, and portraying necessary data. Once each separate page can be individually loaded, the scope grows to the entire frontend and how the pages load their data going from one page to the other.

The second stage, and the stage with the most amount of testing, is the backend portion, as it has the most amount of objects and endpoints which must use manual and automated tests. Each endpoint is related to a specific page in the frontend, whether it be the workorder, inventory or project management page. As such, manual tests are set up so that each section is working as intended, building up so that each page can cross share data. Once each section is ready for integration with the front end, the final tests will scale the entire program, testing whether or not the program is able to function properly at real time.

2 Testing Strategy

As discussed previously, tests are split into three main testing sections: backend exclusive, frontend exclusive, backend exclusive, and integrated. Each section has their own set of specific tests to ensrue proper functionality.

2.1 Backend Testing

Since the backend handles the direct conenction to the database, the section receives the most amount of testing to insure that each endpoint routed correctly, each request references the necessary data and that each section is only accessed with proper authentication.

2.1.1 Unit Testing

For unit tests in the system, each section of the backend (authentication, workorders, inventory and project management) each use a set of tests using the CRUD methodology. The primary tool used for these unit tests is Postman, a program that creates a direct connection to the backend and can send request using a JSON objects. The response sent back to the program is in the form of Http status codes depending on what the status of the request is. While there is a test for every single endpoint in the database, these tests all share the same format for each type of CRUD request. When the server is run, it listens at a specific URL that postman can then send the following requests to

2.1.1.1 GET all request There are two different kind of GET tests in the system. The first is a general requests for all entries in a table. These tests expect each table object in the table and will display each item in a list. Each object is labelled with their unique identifier used for distinguishing each entry. This id will allow for objects to integrate with each other. The id is generated when the object is created and not when it would be reference, and as such the unique identifier is stored onto the database. Each request is retrieved from the database as a JSON object when successful.

Each requests made to the server will respond with a HTTP status code based on the success of the request. A successful request would return a status code of 200 OK for a successful get request. Upon a successful request, every JSON object will appear in the response window. The following figure showcases how the response is structured.


```
{
    "id": 3,
    "billable": false,
    "duration": 1,
    "notes": "Fixing kettle.",
    "startTime": "2019-03-28T10:00:00",
    "endTime": "2019-03-28T10:30:00",
    "timeTypeId": 1,
    "timeType": {
        "id": 1,
        "name": "Shop",
        "costPerHour": 10
    },
    "workorder": null
},
```


Figure 1: A time entry in the from of a JSON response from the server. For a GET all request, all objects in the database will show

2.1.1.2 GET single request This request is very similar to a GET all request from above, but instead of retrieving all entries, the request only receives a single object, based on the id of the specific object. Using figure 1 as an example, a request would be made with the specific identifier of the object, which is 3, and the server can send the response as long as the server is capable to.

2.1.1.3 POST request A post request spends a JSON object to the server such that it the object can be added to the database. The data is filled out according to the structure of the object, though not each field is required to be filled out. The following image showcases a JSON request being sent to the server.

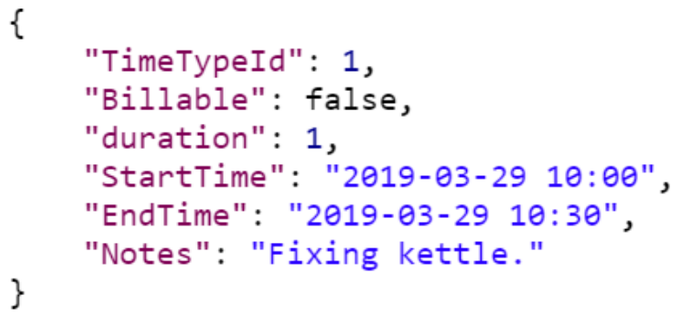

Figure 2: A time entry JSON request to be sent to the server.

When a POST request was successful in being added to the database, a status code of 201 Created will be sent back and the object created will showcase much like a get request. Notice that the POST request did not have each field included. Certain fields that have embedded fields do not need to be specified, but rather referenced as they are already pre-populated at the time of the server being live. For example using figure 2, the field TimeTypeId references the identifier in the TimeType table in the database and retrieves all of the data in that table which is aligned with that id. Since in the example TimeTypeId is 1, all of the fields in the TimeType table associated with that id will load onto the JSON object. How it is stored can be viewed using figure 1 shown earlier.

2.1.1.4 PUT request When a current entry in the database needs to be altered or edited in any capacity, a PUT request is sent to the system. This requests consists of a JSON object filled with the data that is to be changed and the id of the particular object in the database that will be changed. The request is sent to the specific id of the object, so that the changes are made to that specific entry. A successful return will send back a 204 No Content response, meaning that there was a successful response, but there the chandged data is not shown by the request. To view the changes, a GET request is required.

2.1.1.5 DELETE request To delete an entry from the database, a DELETE request is sent to the system specifically by the id of the object to be deleted. On a successful response, the database will respond with a 200 OK and the JSON object, signalling that the object was successfully deleted and removed from the database.

The unit tests for this server will be responsible for testing the business logic and validation behind each endpoint. This business logic will be located in the services layer where each API controller action will have a service. Each of these services will have a number of unit tests that fall under two categories; Architectural tests and Functionality tests.

The architectural tests are primarily there to ensure each service has proper and valid dependencies being injected. For example, with the plans to use an inversion of control strategy for the validators for each service, the architectural tests would be needed to verify that the proper validators are being registered and called.

The functionality tests are the tests that verify that the services are behaving expectedly. These tests will verify that the object mapping from the incoming data transfer object (DTO) to the actual database model occurs without a change in data, that the validators perform correctly and ensure the correctness of the data, that any generated data is generated properly and assigned properly, and that the repository actions behave properly and return the expected object.

2.1.1.6 FakeItEasy The implementation plan for these unit tests is to utilize FakeItEasy with AutoFixture. FakeItEasy is a mocking or faking framework that is utilize to create fake instances of the objects and models that needs to be tested. AutoFixture is used to further simplify the arrange phase of a unit test by handling all of the dependencies of the SUT (system under test). It can also be used to generate fake data for models and objects. When used in conjunction with FakeItEasy, it creates an auto-mocking container that manages the entire arrange phase of the test more or less automatically.

When the unit test is written without the auto-mocking container, any object or dependency in the SUT needs to be set up manually, which also means the tests are liable to break every time anything in

the system is changed. The auto-mocking container ensures the tests aren't fragile by automatically mocking up all dependencies within the system. If any new dependencies are added to the system, the test will automatically create a mocked instance for it. For the arrange-act-assert model of unit testing, this is generally the entire arrange phase, leaving only the actual testing portion to be written.

2.1.2 Integration Testing

Once each separate endpoint is tested by sending CRUD requests, the next set of tests look to see if the JSON object that require the use other objects that are created on run time are integrated correctly. For example, a time entry has a field regarding workorders and whether or not any are associated with the particular entry. When there are associated workorders, a POST request should be able to include the workoder id and be able to reference said workorder by said id. When a GET request is sent to view the time entry, the workorder that is referenced should appear as well. Integrated object such as these are to be tested such that each table can properly store and showcase these integrations.

2.1.3 Performance Testing

After the competition of testing how each endpoint is integrated, further tests to indicate how quickly a request can be processed are done. While this type of tests are more thoroughly done after the competition of both frontend and backend testing, these tests backend exclusive performance tests are done to ensure that there exists a steady connection to the database and that requests a before being processed quickly without extended periods of latency before the requests are even processed by the frontend.

2.2 Frontend Testing

For the frontend section of the project, testing in general consists of visually inspecting whether or not certain elements of the frontend are being processed correctly as well as whether or not each page is formatted correctly. A large concern of the frontend is how the data will appear on the page once it is loaded from the backend. For Vue based project, Jest is a tool that allows the developer to create asserts to ensure that the data the frontend is expecting is in the proper format. This is done so that a particular page will expect and only allow specific data.

2.2.1 Unit Testing

To test the appearance and styling of a particular page, the frontend is run on a development server provided by the Node Package Manager, or npm for short. This development server allows each page in the frontend to be loaded as a locally hosted webpage accessible by a web browser. While programming, the server will automatically update the current page being worked on, showcasing which changes are being made in realtime. By having an environment that can show results instantaneously, a particular page can be evaluated on the spot without having to switch from development mode to testing mode. Each error or bug can be fixed much more quickly. Whenever a page cannot be properly generated due to a particular bug or issue, the error is showcased on the page itself. As such, most of the testing is done to ensure that the page is loaded correctly in the way it was intended.

2.2.2 Integrated Testing

These tests are similar to the unit tests in that all testing can be done simultaneously with development to ensure that each page in the frontend can direct and redirect the user to the proper destination page and that the navigation is smooth, logical and feasible.

2.2.3 Automated UI Testing

These tests are currently not in the inital development schema but are to be implemented for future testing to ensure that the frontend can be automated smoothly and efficiently. Each test ensures that each page is possible to navigate to and each page loads all necessary information upon runtime.

2.3 Integrated Testing

Once both frontend and backend sections are tested fully, the front end and backend sections are integrated to gether such that the frontend can send requests to the backend and the backend can respond back. Essentially, the frontend has to be able to replace the functionality of Postman.

2.3.1 Unit Testing

Similar to the frontend unit tests, these tests can be done during development, to ensure that the particular page has a stable connection to the database, is able to get a response when requesting objects from the database, and is able to properly display that data where needed. The development server will showcase the result by loading the particular page and if developed correctly, the data received will display as intended. For example, when a time entry requested is properly received from the database, the data is mapped such that the entry can appear on a calendar in the page where it was requested. Each time a new entry is made, then the page will load that entry in the calendar if successful.

2.3.2 Performance Testing

One the integration tests are all complete, the system will then undergo vigorous performance testing to ensure that each section of the site is able to load its data from the server as efficiently and as quickly as possible. Each page must be able to work as quickly as can be so that the user does not have to experience any latency during production. Each task and feature should be optimized by this effort. Once these tests are complete, the program is ready for release, but performance tests are never truly finished, as the program should run efficiently at all times and any bugs found during use can be patched quickly.

3 Test Schedule

As stated in the previous sections, the plan to test each endpoint of the backend first separately, progressing into endpoint integration testing. After such, frontend page unit and integration testing is to be done before full backend and frontend integration testing is completed. Once each section of testing is completed, the program is intended to undergo stress testing and production testing, with the application being constantly improved and worked on as bugs are found.

3.1 Features to be Tested

3.1.1 Backend

The most important aspects of the program will be undergoing testing as part of the first release. For the backend, it is essential that every endpoint is fully fleshed out and optimized, and as such each endpoint has an associated unit test. Each endpoint must be able to be accessed and data can be requested or sent over.

Endpoints with inherited data is also a feature that will be tested, such that each can reference and load the correct data from the database when requested. This is done so that when features that require data from multiple sources, such as workorders and the materials that will be used, and any time tracking information that it may come with.

3.1.2 Frontend

On the frontend, the most important features are the ability to create and view workorders, create and view materials and create and view time entries. Each page should have its full ability to create and edit entries, and have each page related to the feature properly implemented at run time. As such, these main pages are the top priority of testing in the frontend section.
The side navigation bars which appear on each page of the frontend section also will be tested to ensure that each page can be accessed from these menus. This means that regardless of the page that the user is currently on, they are still able to navigate to any page of the application without having to backtrack.

3.2 Features not to be Tested

The features that are in currently in development but will not see full implementation in the first release will not be tested. These main features are frontend features such as the report generation due to the implementation of these features not being able to be fully implemented and operational during the first release of the application. In the next release, once these features are implemented to their full potential, they will be added to the testing plan. Other features that will not be tested include retrieving pre populated data from the database. Since this data is constant and is only used by the system when it is referenced by an editable endpoint, the test to see if the data exist already is tested during the endpoint unit tests.

Testing the compatibility of the site to work on a mobile device will also not be currently tested as the application is originally designed to have to function primarily on a desktop or laptop. Since the application is run using a full scale web browser, the application was not originally intended to be used on a mobile device.

4 Dependencies

The system requires the use of a full scale web browser that supports javascript functionality and will not function on Internet Explorer 8 or below. This is due to the software requirements of ASP.NET and Vue.Js, the frameworks that the application was designed with. However, the application will be able to run on any type of personal computer operating system, as it is only limited by the type of browser the machine is using. The system does require a steady internet connection, as the frontend is constantly communicating with the backend, which will be hosted on a server that the application must have a steady connection to.

5 Risks and Assumptions

The main assumption that is made while using this system is that the client is completely authorized to access all of the workorders and associated data to the engineering workshop. To reduce the risk of unauthorized use, the system does require a valid user login information before granting access. Another main assumption about the system is that all data that is currently entered to the system is

relevant to the workshop in some capacity. It is the responsibility of the client to ensure that all data that is added by the client is done in a correct and proper manner.

6 Tools

The main tool that used for testing is Postman, an API listening program that allows the testing environment to exists. The program acts as a requests hub, which will send requests directly to the backend and will await proper HTTP responses back, showcasing the result in a response window. It allows JSON objects to be sent to specific endpoints on the backend that will then populate the database during development. This allows for bakcend only development without the need of the frontend pages to be fully developed, allowing the refinement and proper implementation of each endpoint. The following figure showcases the program and how a request is made.

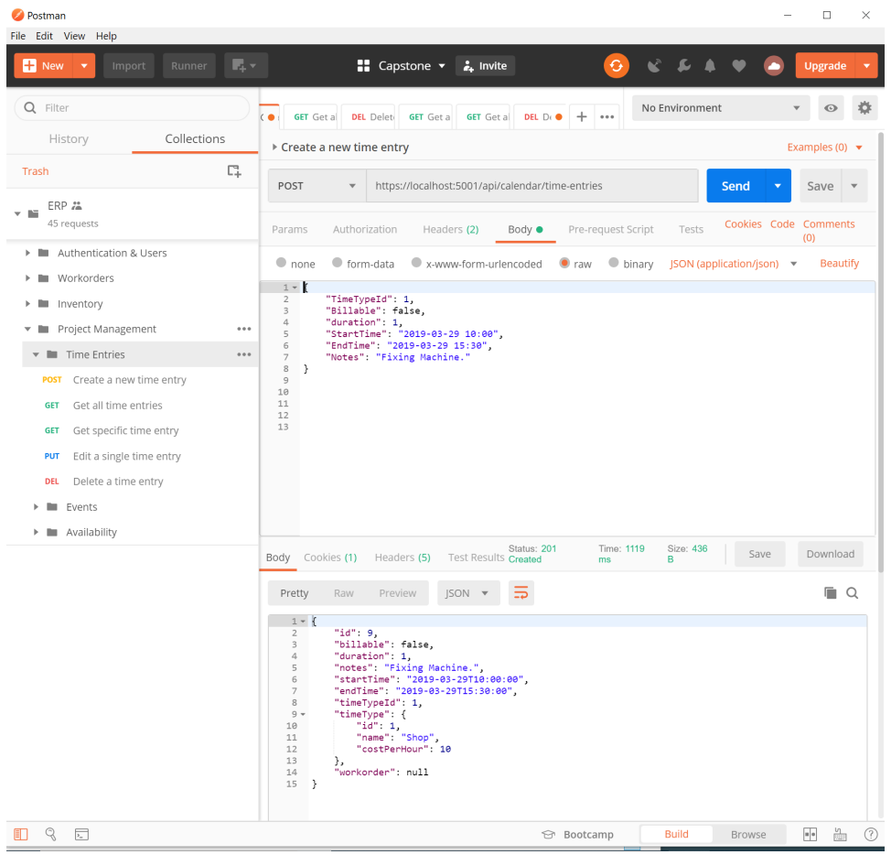

Figure 3: Sending a POST request in Postman

On the frontend side, the main testing tool is the one used alongside development which is the Vue command line interface run serve, which allows the frontend to be run on a development server. This server allows for any changes during development of the frontend to appear instantaneously rather than having to reload the page after every change. This allows for quicker error checking and page evaluation, which in turn means that unit tests for the frontend can be done immediately after a feature is implemented. In total, the frontend does not require many tools for testing, however the use of this development server does allow for more tests to be completed more quickly. The following figure showcases the development server running. Notice the URL of where the frontend is being run.

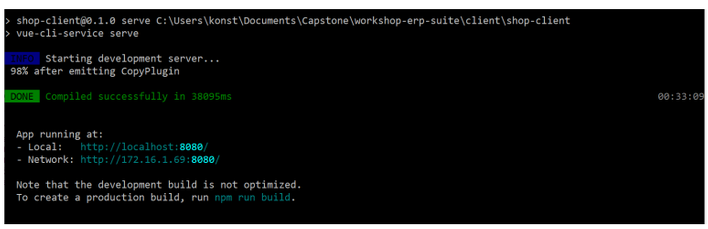

Figure 4: Development Server in use