
Test Plan

for

Galvanize

Version 1.0

pH14 Solutions

Andrea Tamez

Braxton J Hall

Christopher Powroznik

Cindy Hsu

Kwangsoo Yeo

Masahiro Toyomura

Revision History

[Nov 7] ver. 1.0 - First revision.

Table of Contents

Revision History	1
Table of Contents	1
Introduction	3
Overview	3
Scope	3
Requirements Test Coverage	4
Functional Tests	4
Non-Functional Tests	4
Performance & Capacity Testing	4
Testing Methodology	4
Separation of Duties	5
Automation	5
Test Types	5
Static Testing	5
Interface Testing	6
Monitoring Testing	6
Dynamic Testing	6
Docker Automate	6
White Box & Black Box Testing	7
White Box	7
Black Box	7
Test Environment	7
Test Cycles	8
The Adapter Test Cycle - Black Box Suite, Regression Suite, Integration Suite	8
Authentication in the Adapter Tests and Security	9
The Backend Test Cycle - White Box Suite	9

The Frontend Test Cycle - Interactive Tests, Integration Tests	10
The Acceptance Test Cycle - End to End Tests	11
Advanced Considerations	12
Data Generation	12
Security and Vulnerabilities	12
Data Masking and Sensitivity	12
Omissions	13
Tooling	13
Microsoft Azure Data Storage	13
Reporting Tools	14
Dependencies	14
Risks	14
Test Plan Approval	15
Appendix	15
Glossary	15
Scripting Details	16
Required Tools	16
Before Running the Application	16
Configuring the Environment	17
Running the Application	17
Running the Tests	17
Requirement Input, Outputs, Happy & Unhappy Paths	17
Functional Requirements	17
Frontend Acceptance Criteria	22
Authentication	22
Candidate Management Screen	23
Scheduling	24
Rooms	26
About	26

Introduction

The **Galvanize Scheduling Tool** is a web-based tool that enhances the interview scheduling process at Galvanize. The Human Resources department at Galvanize currently handles all scheduling manually through Outlook tools and communicating with candidates and interviewers directly. This tool automates some of these processes.

First, an Administrator (hiring manager or human resources staff) logs into the web portal using their Microsoft account, and provides a list of possible interviews with specified interview minutes. Once the “send invitation” button for a candidate is clicked, Outlook is invoked, and an interview invitation email is sent to the corresponding candidate. After a candidate submits their availabilities for visiting the office, the system automatically finds a list of suitable schedules for everyone involved. In cases where an interviewer would like to conduct their interviews with colleagues, the colleagues’ calendars are taken into account in the generated schedules. The tool includes all of the necessary management functions for this use case to be fulfilled.

Overview

The purpose of this document is to present the plan for testing the **Galvanize Scheduling Tool**. The plan will cover the test cycles, implementation, infrastructure, and technique. The types of tests—including static, interface, monitoring and dynamic—will be elaborated on, but for a more literal view, samples of test input and outputs can be found in the appendix. In addition, we will touch upon the methodologies used in our whitebox and blackbox tests, test automation, the approach for user acceptance testing, and risks related to execution of the plan.

Scope

Our testing suites will be measuring all functional requirements (the system’s expected functionality) and non-functional requirements (system attributes such as security, reliability and performance). We will be thoroughly testing these requirements through a set of test cycles, which each interrogate a subset of the project microservices, and their integration. These microservices (or constituent parts of the overall project) consists of the:

1. **Frontend:** The web user interface through which users will interact with the project
2. **Backend:** A web hosted server for managing the business logic of the project
3. **DynamoDB database:** A web hosted database for storing the project's information

Requirements Test Coverage

In order to write applicable tests, we categorized different user stories into functional, non-functional and performance & capacity testing. This separation is listed below:

Functional Tests

- FT001. The application must create schedules.
- FT002. The application must integrate schedules with employees' Outlook calendars.
- FT003. The application must invoke Outlook to email invitations to Candidates.
- FT004. The application must send and receive availability forms to and from Candidates.
- FT005. The application must allow for the creation and removal of user profiles.
- FT006. The application must support an administrator profile type.
- FT007. The application must support configuring the available rooms in the office.
- FT008. The system should check for the rooms available in Outlook before scheduling.
- FT009. The project result must consider the schedules of all the interviewers involved.

For detailed examples of each of Function Tests' inputs and outputs, please refer to the included appendices.

Non-Functional Tests

- NFT001. The project result must be a web based tool.
- NFT002. The project result must be hosted on AWS.

Performance & Capacity Testing

- PCT001. The project result must be able to support a minimum of 5 concurrent users.
- PCT002. The project result must have APIs and a webpage with have a response time of less than 2 seconds.

Testing Methodology

We start off with the SAT (System Acceptance Testing). We test on our development servers by sending a message and awaiting a response to ensure that the server is running. After the initial SAT, we move on to test the implementations of Adapter by creating unit tests. On this phase, the Adapter is tested alongside the Backend implementations. Furthermore we do more SAT such as performance and capacity. Finally we end up testing our Frontend with User Acceptance Testing. Each of these phases are described in greater detail throughout this document.

Separation of Duties

As we are using test-driven development and wrote the actual assertions and uses of the adapters from the Adapter Test Cycle before implementation (in a black box form), it is not a concern that test will be biased toward a possibly functionally incorrect implementation. As the Adapter and Backend tests are fully automated, every developer on the project can run them to completion before every commit (as is required before merges to the development branch). Thus, developers will be able to see if their build breaks functionality, even if it is in other parts of the system. Testing was divided among group members, by splitting them between working on the adapter tests and the backend tests. Christopher, Cindy and Andrea worked on the adapter tests and Braxton, Kwangsoo and Masahiro worked on the backend tests. Both automated test suites were built in a pair-programming environment with the respective assigned teammates.

Automation

Each automated test suite has an npm script that automatically starts up the necessary components and executes the tests. We use docker-compose for container orchestration of the Backend and Database. Certain configuration details are pulled from the .env file inside the repository at run-time, allowing for setup flexibility. This includes the secret key used during adapter testing to work around the Microsoft OAuth process, which normally requires a browser (this is described in detail in 'Adapter Test Cycle' on page 9). A Git Hook script will be used to automatically run the test suites on every commit, so developers do not accidentally push broken builds.

Test Types

Static Testing

Our static tests are linting of our Typescript code to ensure readability, best practices and scalability, and to avoid common syntactic errors. We want to ensure that future developers at Galvanize are able to understand and adhere to standard coding practices to ensure the best development of the project. We run a linting at the beginning of the build cycle and will not allow building if errors are present within the linting. We use ESLint with the Typescript Parser plugin to lint the Typescript code; all of this is installed via the npm install (described in the included appendices below). This is enforced by our Typescript compiler and each of our IDEs are set up to display the linting in real time. We provide the necessary configuration files in terms of a `tsconfig.json` and an ESLint configuration file, so this is possible in any Typescript enabled editor.

Interface Testing

To test if our application's Frontend can make requests or configure data in the Backend correctly, we primarily test the Adapter because the Adapter handles the communication between the Frontend and Backend. Therefore, given a specific input into a function in the adapter, it will expect an output with a specific format. We test for adding, getting, deleting, and updating data and test for unexpected information or edge cases in the input and expect correct handling for each of these cases.

Our application uses services such as Microsoft Graph APIs and DynamoDB. We create separate tests that handles requests to Microsoft Graph or DynamoDB. This way we can differentiate, if errors would happen, between implementation in the adapter, API calls to Microsoft Graph, or specific database errors. These tests are described in detail below in the Backend Test Cycle header.

Monitoring Testing

Our test monitoring consists of running '0x' (a library for performing dynamic analysis) to create a flame graph of our tests. This allows us to monitor the stack, runtimes and bottlenecks of our adapters and backend. This is incredibly useful during our stress tests, which will isolate which part of our system is causing the greatest latency and resource usage. This is run via an npm script that will wrap our testing script.

We will also include a script which generates requests to retrieve data from our system and send them concurrently. By increasing the number of requests over time, coupled with a 0x profile, we will be able to find the capacity of our system, and validate PCT001 and PCT002.

Dynamic Testing

Dynamic testing consists of our traditional tests that are testing our runtime functionality. This consists of both of our major test cycles, 'The Adapter Test Cycle' and 'The Backend Test Cycle', which are under their respective headers below in this document. Our system will only be deployed when these tests all pass.

Docker Automate

Docker automated test will allow us to run our tests in containers as well as to isolate each of the test we have in the development and deployment. The makes it so every dependency related to a test has a clean environment at the beginning of each cycle and there is no cross-over data or progressive differences between each run of the tests.

White Box & Black Box Testing

White Box

MSGraph: We will test specifically the required API calls which are used in the application. Since all the requests to MSGraph are done within one class called MSGraphController, we created unit tests which will test the functionalities our system requires. These include connections to MSGraph client, getting data out from MSGraph and errors given an incorrect input.

DynamoDB: Similarly to how we test MSGraph APIs, we also have a class handling database requests called DynamoDBController. We create unit tests testing each of the functions related to our database.

If both MSGraph and DynamoDB tests are working correctly, it means both services are ready and applicable for integration tests.

Black Box

The Adapter tests are essentially the black box tests of our system from the perspective of the Frontend. The adapters call generic functionality routes and receive a specific

output which later is sent to the frontend. This shows that our adapter receives inputs and outputs from both Frontend and Backend. Therefore, testing our adapters will mean testing the integration between Frontend and Backend.

Test Environment

The automated test environment requires a small set of dependencies (described in the Tools/Dependencies header below) to execute. Most importantly, Mocha which runs tests and generates reports on correctness and coverage. Further, a test instance of Microsoft Office 365 and Azure was created and maintained for the purpose of testing (both automated and interactive). This test instance must be live for the tests to run correctly. This test instance is described in detail in the Tooling/Microsoft Azure Data Storage header below.

Our test environment is separated into 2 different run-times and suites. One suite is used to test the adapter implementations and the other is used to test the backend functionality. The reason for the independence of both suites is because the adapters act as a pseudo end-to-end test, that mimics functionality that would happen on the Frontend. These are more generally touching our entire setup including http requests, database functionality and middleware. These tests are more general in nature. On the other hand, the Backend tests consist of specific unit tests and functionality on our Backend. Since it is difficult to test specific functions and areas, through the adapters, this test suite is used to perform more robust testing on backend functionality.

Test Cycles

The Adapter Test Cycle - *Black Box Suite, Regression Suite, Integration Suite*

The Adapter test cycle is used to measure both the integration of all of the projects' microservices (barring the React frontend), as well as regressions between releases. In this test cycle, the tests model the frontend's usage of the adapter. The adapter test cycle begins by spinning up a backend container as well as a DynamoDB container in Docker. From there, the test suite continues with authenticating a login with the MSGraph server; this login token is then saved and used to maintain the session as it would be in the frontend. This process is further described below. As well, to make the tests repeatable, the suite will clear the schedules of all Interviewers exposed from the test Microsoft Azure instance.

Each test file in our Adapter test-suite follows the discrete requests of a Use Case as defined in the Use Cases header of the Business Requirements Document. The test suite within each test will add and remove mock data, as well as retrieve data from the test instance of Microsoft Azure, and finally invoke Outlook emails. The test suite will also only touch a development database running locally, thus production data is not touched. The test suite covers all of the major frontend functionality.

For example, to expose the tests FT001 and FT004, JavaScript Object representations of their respective inputs are sent to the Adapter, and the responses are waited upon. After receiving responses for both Happy and Unhappy paths for each respective test, Chai is used to assert that the responses from the system are correct. Details on the inputs and expected outputs of these tests are described in the included appendices.

Upon completion, the test suite will kill the Docker containers it had opened. Because the containers are generated and removed every time the tests are run, and the schedules are cleared in the test instance of Azure, running the tests is idempotent; multiple executions will not interfere with each other.

In summary, The Adapter Test Cycle can be described as follows:

1. Create containers for a backend and database
2. Authenticate and save a token with MSGraph
3. Clear the schedules of all data points in Azure that will be exposed by the tests
4. Run the tests in Mocha with assertions from Chai
5. Kill and remove the backend and database containers

The acceptance criteria for the Adapter Test Cycle is 100% functional correctness as reported by the automated test suite.

Authentication in the Adapter Tests and Security

A large hurdle with testing the adapters is authenticating with the MSGraph APIs. In the frontend, in order to authenticate, the user will be redirected to a Microsoft login portal. This login portal is not under our control and we only specify an endpoint that authentication information is sent to after login. Because of this we are not able to automate logging in and getting a user token. The user token is necessary for most of the adapters to work. Because of this drawback, we had created a special function in our backend called `createTestToken`, this is a special route in our backend, that can create login tokens with MSGraph given a secret key that is stored only in the local environment and not hard coded into the repository. There is an additional function, which saves the authentication token in our database, so that the adapters may check

against it in the testSuite. Because of the security concerns of this functionality we disable it in production, which is controlled by an uneditable environment variable.

The Backend Test Cycle - *White Box Suite*

Similar to the adapter tests, the backend test cycle starts by spinning up a Docker container though Dockerode running the dynamodb microservice image. Similarly to the Adapter Test Cycle, this was done to allow the tests to run with a fresh database every run; the suite is idempotent. Using a database inside a Docker container makes clean up after the tests easy because the test container can just be deleted. Next, an authenticated token with MSGraph is generated and saved. As the suite runs in the backend, the security hurdle with extra test routes as described in the Adapter Test Cycle does not apply; the functionalities under test exist passed the authentication layer. Using this token, the suite will clear the schedules of all Interviewers exposed from the test Microsoft Azure instance.

The suite includes three major sets of tests: The DatabaseController.spec, measuring the functional correctness of the backend's interactions with the database; The MSGraphController.spec, measuring the functional correctness of the backend's interactions with Microsoft Graph, Azure and Outlook; The ResourceFacade.spec, measuring the functional correctness of the backend's ability to integrate data from both sources. The backend test suite is run using Mocha and Chai, and uses predefined arrays of mock data to test backend functionality such as inserting and deleting candidates and rooms from the database, as well as retrieving Interviewers, Interviewer Schedules and Room Schedules from Microsoft Graph.

For example, in the ResourceFacade.spec automated test suite, to complete FT005, FT007, FT008, JavaScript object representation of each tests' inputs are sent to the Backend microservices, and Chai is used to assert that each tests correct output is returned by the system. Descriptions of these tests' inputs and outputs are as described in the included appendices.

Upon completion, the DynamoDB container is removed.

In summary, the Backend Test Cycle can be described as follows:

1. Create container for the DynamoDB
2. Authenticate a token with Microsoft Graph
3. Clear the schedules of all data points in Azure that will be exposed by the tests
4. Run the tests in Mocha with assertions from Chai

- a. The DatabaseController.spec sub-suite
 - b. The MSGraphController.spec sub-suite
 - c. The ResourceFacade.spec sub-suite
5. Kill and remove the database container

As these tests were written along side actual implementation, they can be seen as white box tests, and increasing demands of our acceptance criteria. The acceptance criteria therefore for the Backend Test Cycle is 100% functional correctness as reported by Mocha, as well as achieving above 95% statement coverage within the controllers directory of the backend.

The Frontend Test Cycle - *Interactive Tests, Integration Tests*

Unlike the previous two test cycles, the Frontend Test Cycle is not automated, but instead interactive. Before testing the Frontend and its integration with the other two microservices, new Docker containers running images for the Frontend, Backend, and Database must run. By running a new container, the Database made fresh, and no previous interactions interfere with the cycle.

Just by ensuring that no page of the Frontend is accessible without proper login and authentication achieved by inputting proper Microsoft Office 365 credentials, we expose FT006.

From here, a tester would interact with the constituent parts of the Frontend to ensure that not only they respond as normal in the Frontend, but as well the effects outside the system are seen. For example, to test FT003, a new Candidate will be added to the system given the tester's email address. The tester would then proceed to check their email and ensure that they received one from the system to validate its correctness.

The tester would then fill out an availability form on behalf of the fictional Candidate (exposing FT004), and click through creating a Schedule, and then go to Microsoft's Azure platform to ensure that the schedule was in fact made (exposing FT002).

The test would repeat this from the beginning, giving Unhappy Path type data, like a Candidate with no email, an Availability Form with no times specified, and a Schedule with no Interviewers specified, so as to ensure the Unhappy Path outputs of each Functional Test. On completion, the tester need only log out, and kill and remove the Docker containers.

For full examples on interactive test procedures, refer to the included appendices below.

In summary, the Frontend Test Cycle can be described as follows:

1. Create a container for each microservice
2. Login to the Frontend web portal
3. Interact with the web page
4. Verify that emails were sent and schedules were created
5. Logout
6. Kill and remove containers

The acceptance criteria for the Frontend Test Cycle are exhaustive and described in detail in the included appendices.

The Acceptance Test Cycle - *End to End Tests*

The final end to end test will occur when the project is hosted on AWS (as per request of the sponsor) and integrated with a real DynamoDB (as per the suggestion of the sponsor).

This cycle will follow the same steps as the Frontend Test Cycle, however also ensuring that our system integrate with actual Amazon services. This is done to expose NFT001 and NFT002.

This cycle is considered accepted by the same criteria as the Frontend Test Cycle as described in the included appendices below.

Advanced Considerations

Data Generation

Since we cannot integrate with Galvanize's system yet, we set up our own Microsoft Enterprise account to generate requests to MSGraph. Also we assume which fields will be used by Galvanize to set up mock data in Office 365. The mock data in Outlook 365 is all manually inputted. Our tests will be using the manually added data. Though adding the data was done manually, this will only ever be done once, and getting and using data will be achieved with API calls and the data is added directly in Microsoft Azure Portal; therefore security is not a concern in generating mock data.

Another concern related to data is, since the main feature of the application is a schedule creation that relies on time, it means we should choose a “time” that is sufficiently ahead into the future (ex. 2023) so unexpected results can be avoided in future tests.

Security and Vulnerabilities

Our main concern with vulnerabilities was actually the vulnerabilities of our dependencies, as all sensitive information is stored in them, and we are as secure as they are.

Snyk is an open source library that helps in monitoring our project. It is a tool that helps in searching for vulnerabilities and outdated sources in a project which also includes imported open source libraries. After a project builds with Snyk, it will display us a list of vulnerabilities in our current project weighted by risk. Furthermore Snyk can be configured within a Docker image, meaning that every build can be tested against Snyk in an automated fashion on Continuous Integration and vulnerabilities can be isolated as soon as they are introduced.

Data Masking and Sensitivity

This was a major focus for the project. We do not want to store any personal data that is already inside of the the MSGraph organization. In the database that we run within the project, the references to staff and rooms are only stored via primary keys. No data is copied into our database, and the primary keys cannot be used without an authenticated token. This means that any compromise to our database will not reveal any internal information of Galvanize. However, the data of candidates is stored in our database, meaning their specific data does not meet the same security standards of MSGraph.

To further remove ourselves from potential data security risks, all tests were with a test instance of Microsoft Office 365. No real sensitive data was ever exposed in the testing process.

Omissions

- **Accessibility Testing:** Usually this is done to ensure that the application is usable by people with disabilities like hearing, color blindness, old age or other disadvantaged groups.

- Usability Testing: This is executed to verify how easy is to use something by conducting user studies.

We will be omitting both of these tests, as they were not outlined as specific goals in the non functional requirements, and due to time constraints.

Tooling

Microsoft Azure Data Storage

Our application uses Microsoft Azure Data Storage, and the data is used to implement some of the tests. First we need a login instance to be able to run the tests. After logging in, we test our implementation with a 25-day free trial instance of Microsoft Azure Data Storage (25 days later, might have to end up switching instance). This instance is not linked with any of Galvanize's storage and purely using pre-arranged mocked data. So currently our application does not contain any security bridges since it is not integrated with Galvanize's system.

The tests are implemented in such a way assuming that Galvanize also uses the main necessary features in Microsoft Azure Data storage to schedule an interview. Failure may happen when integrated with Galvanize's system and re-implementation will be needed if incompatibility with Galvanize's system should occur. The tests for the main feature in our app, scheduling interviews, are implemented with dates that are about four years into the future (around year 2023). Therefore, the tests can be run without errors if such a scheduling date defined in the tests schedules to the past.

Reporting Tools

We use GitHub issues to report any bugs and organize all of our tasks. This will help us to focus on our collaboration between the team. At the same time it is also a way for us to organize different types of issues and manage which ones have more priority than others.

Dependencies

- chai: 4.2.0 - Used for assertions inside the automated test suites.
- dockerode: 3.0.2 - Programmatically spawns Docker containers for the testing process.

- dotenv: 8.2.0 - Used to read .env configuration files.
- mocha: 6.2.2 - Runs the test suite and generates correctness and coverage reports.
- source-map-support: 0.5.13 - Reports Typescript line errors after compilation.
- Docker: 19.03.4 - Runs microservice containers.
- docker-compose: 1.24.1 - Used to coordinate Docker containers.
- 0x: 4.9.1 - Profiles system at run time.
- axios: 0.19.0 - Used to procedurally generate API calls to our own backend.
- Snyk: 1.244.0 - Used to find vulnerabilities in our dependencies and project.

Risks

- We are testing right now on mock data, which is necessarily a subset of all possible inputs. As with any test suite, we cannot prove every corner case it covered.
- While the tests suites are mostly idempotent, because scheduling is based on time, and to consecutive runs of the suite are necessarily at different times, the automated tests are never truly repeatable. While we have taken measures as described above to combat this, it is no less a small risk.
- We incur risks with every new dependency that we include, as if they report incorrectly, our automated test results will report incorrectly.

Test Plan Approval

Name	Role	Signature	Date
Asem Ghaleb	Facilitator		
Christopher Powroznik	Project Owner		
Cindy Hsu	Test Owner		
Peter	Client		

Appendix

Glossary

Git Hook: are scripts that Git executes automatically before or after events such as: commit, push and receive.

API: is a system of rules that will allow two or more entities to transmit information in this case a client and a server.

AWS: is a cloud computing platform that offers database storage and API's..

Docker: is an application platform as a service that helps to build and run applications with containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels.

Dockerized: is the process of converting an application to run within a Docker container.

Microsoft Graph: A developers' API platform to connect to the data that drives productivity. It delegates calls to different Office 365 Cloud services via one single endpoint

Snyk: integrates securing open source into existing workflow of a developer by doing audits to make sure everything is secure.

Statement Coverage: A measure of the number of statements within a codebase that are executed by a test suite over how many there are total in the codebase. This can be used to gauge how exhaustive the test suite is.

Chai: is an assertion library it makes testing easier by giving you assertions that you can run against your code.

Mocha: JavaScript test framework for Node.js programs featuring browser support, asynchronous testing, test coverage reports and use of any assertion library.

Azure: is a cloud computing service created by Microsoft for building, testing, deploying and managing applications and services through Microsoft-managed data centers.

DynamoDB: non relational database service offered by amazon that supports key-value and document data structures.

Scripting Details

<https://github.com/braxtonhall/galvanize-scheduling/tree/docs>

Required Tools

1. Node v10 and latest version of npm(node package manager)
2. Docker 19.03.4
3. Latest version of git (optional)

Before Running the Application

1. Copy the repo by running `git clone`
<https://github.com/braxtonhall/galvanize-scheduling.git>
or downloading the zip file manually.
2. Run Docker and have it running.
3. Get an image of DynamoDB in Docker by running `docker pull amazon/dynamodb-local`.

Configuring the Environment

1. Create a file called `.env` and place it in the root of the directory. (`touch .env`)
2. Copy the content of `.env.sample` into `.env` file. (`cat .env.sample > .env`)
3. Modify `.env` file with necessary information. (below are the necessary modifications)
 - a. `DB_HOST_PERSIST_DIR=<path to local db storage>`
 - b. `OAUTH_APP_ID=<id of MSGraph client>`
 - c. `OAUTH_APP_PASSWORD=<secret of MSGraph client>`

Running the Application

1. Run `npm run restart` in the root of the directory to start the server. (Must complete all of the above)
 - a. It may take a while on first run.
 - b. Create db... **done**

Create backend ...**done**

Create frontend ...**done**

Should appear if done correctly.

2. Navigate to <http://localhost:3000> to see main webpage.

Running the Tests

1. Have the instance of the server running up.
2. Navigate to <http://localhost:3000> and login with your credentials once.
 - a. Needs to be done once since DB needs a token for API calls.
3. Run `npm run test:backend` to test the Backend.
4. Run `npm run test:integration` to run Adapter/Integration tests.

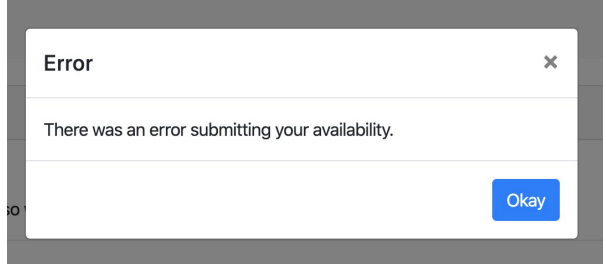
Requirement Inputs, Outputs, Happy & Unhappy Paths

Functional Requirements

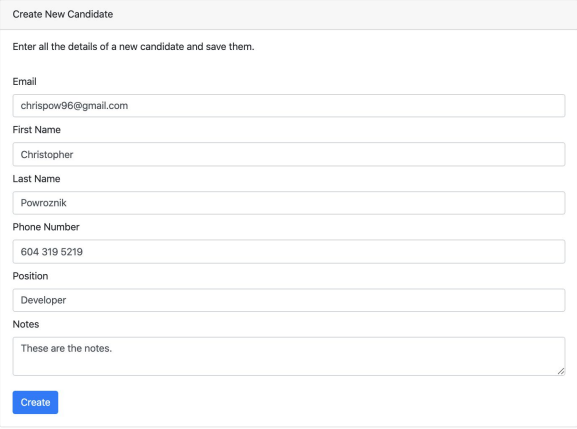
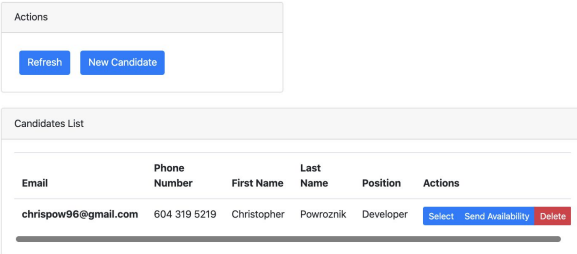
FT001. The application must create schedules. FT002. The application must integrate schedules with employees' Outlook calendars.	
Happy Input	Happy Output (Acceptance Criteria)
Given a Candidate with an availability, an Interviewer with an open calendar that matches the availability, and rooms with an open schedule.	Applicable schedules will be generated and displayed where the times available in the candidate availability, the calendar of the interviewer and the rooms schedule have an overlapping time.
Unhappy Input	Unhappy Output (Acceptance Criteria)
Given a candidate that hasn't submitted an availability, an interviewer without a calendar in outlook, or a room missing a schedule in the MSGraph API	A schedule will not be given and a message displaying this error appears.

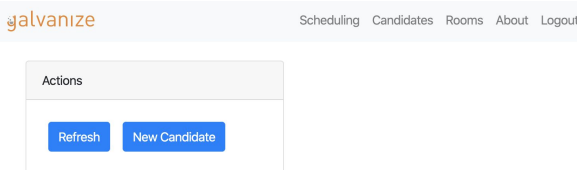
FT003. The application must invoke Outlook to email invitations to Candidates.	
Happy Input	Happy Output (Acceptance Criteria)
Given a candidate that has an email, and an admin that has pressed the button to	An email with a URL to the availability submission screen is sent to the

send an availability.	candidates email.
Unhappy Input	Unhappy Output (Acceptance Criteria)
Given a candidate that has an invalid email, and an admin that has pressed the button.	An email will fail to send, since it does not exist. This error is sent back to the admin.

FT004. The application must send and receive availability forms to and from Candidates.	
Happy Input	Happy Output (Acceptance Criteria)
Given a candidate that has clicked to the availability email sent before, the candidate fills his availability form and clicks send availability form.	A Thank You page will show up to the candidate.
Unhappy Input	Unhappy Output (Acceptance Criteria)
Given a candidate in the availability page, the candidate does not fill the form and click on send availability form.	Availability form will fail to send and display an error. 

FT005. The application must allow for the creation and removal of user profiles.	
Happy Input	Happy Output (Acceptance Criteria)
Given the minimum of an email of a candidate, and any combination of the following optional information... <ul style="list-style-type: none"> • First Name • Last Name • Phone Number • Position • Notes 	A user will be added to the candidates menu and be available for sending availability, updating or deleting.

	
Unhappy Input	Unhappy Output (Acceptance Criteria)
<p>Given a candidate with no email and any combination of the following optional information...</p> <ul style="list-style-type: none"> • First Name • Last Name • Phone Number • Position • Notes 	<p>The candidate will not be not be created, since the email is a required field.</p>

FT006. The application must support an administrator profile type.	
Happy Input	Happy Output (Acceptance Criteria)
<p>Given an administrator, the administrator logs in through the login portal of the application.</p>	
Unhappy Input	Unhappy Output (Acceptance Criteria)
<p>Given an administrator, the administrator fails to authenticate with Microsoft.</p>	<p>Microsoft authenticate page will redirect to the login page.</p>

FT007. The application must support configuring the available rooms in the office.

FT008.The system should check for the rooms available in Outlook before scheduling.

Happy Input

The MSGraph has inputted rooms from the Office 365 system, these rooms could exist or not exist inside the DynamoDB database. Before selecting generate schedules by the admin, the application generate schedules given the rooms have available time slots.

Candidates List					
Email	Phone Number	First Name	Last Name	Position	Actions
chrisspow96@gmail.com	604 319 5219	Christopher	Powroznik	Developer	Select

Interviewers				
ID	First Name	Last Name	Preference	Time Needed
272373f6-d0af-47d8-bd26-c916e257bb20	Peter	Parker	Alone	0 minutes
4c294c2d-c014-4538-a9e7-9fb1e6628ba0	Perry	Liao	Alone	0 minutes

Generate Schedules

Happy Output (Acceptance Criteria)

A list of rooms are shown indicating if they are available for interviews. Each room should be toggable between are available for interviews or not.

Unhappy Input

The MSGraph does not have access to any rooms in the Office 365 system.

Unhappy Output (Acceptance Criteria)

A list of possible schedules will not be shown and a message will display showing no rooms are available.

FT009. The project result must consider the schedules of all the interviewers involved.

Happy Input

A valid scheduling request of a candidate with an availability and a series of interview enabled rooms.

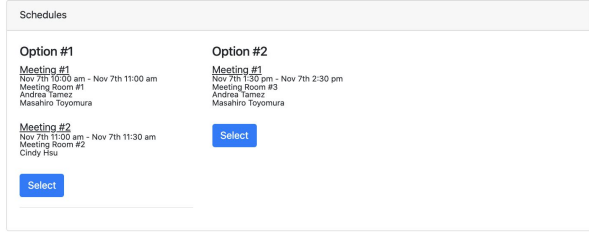
Candidates List					
Email	Phone Number	First Name	Last Name	Position	Actions
chrisspow96@gmail.com	604 319 5219	Christopher	Powroznik	Developer	Select

Interviewers				
ID	First Name	Last Name	Preference	Time Needed
272373f6-d0af-47d8-bd26-c916e257bb20	Peter	Parker	Peter Parker	30 minutes
4c294c2d-c014-4538-a9e7-9fb1e6628ba0	Perry	Liao	Alone	0 minutes

Generate Schedules

Happy Output (Acceptance Criteria)

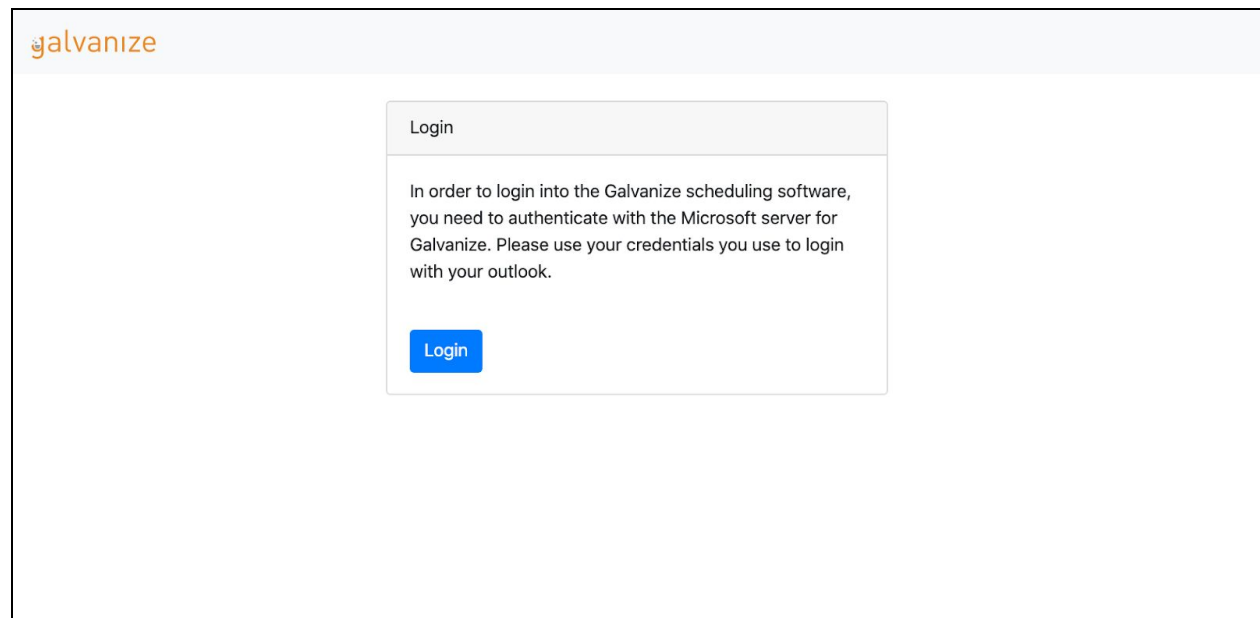
The Outlook calendars of each eligible interviewer is called and checked against the availability of the candidate and rooms. A series of schedules are returned that have valid overlap between the candidate, the interviewer calendar and room.

	
Unhappy Input	Unhappy Output (Acceptance Criteria)
The MSGraph API does not have access to the schedules of the interviewers or there are no interviewers in the request, that need allotted time.	There are no schedules returned and the user is prompted that an interviewer needs an allotted time.

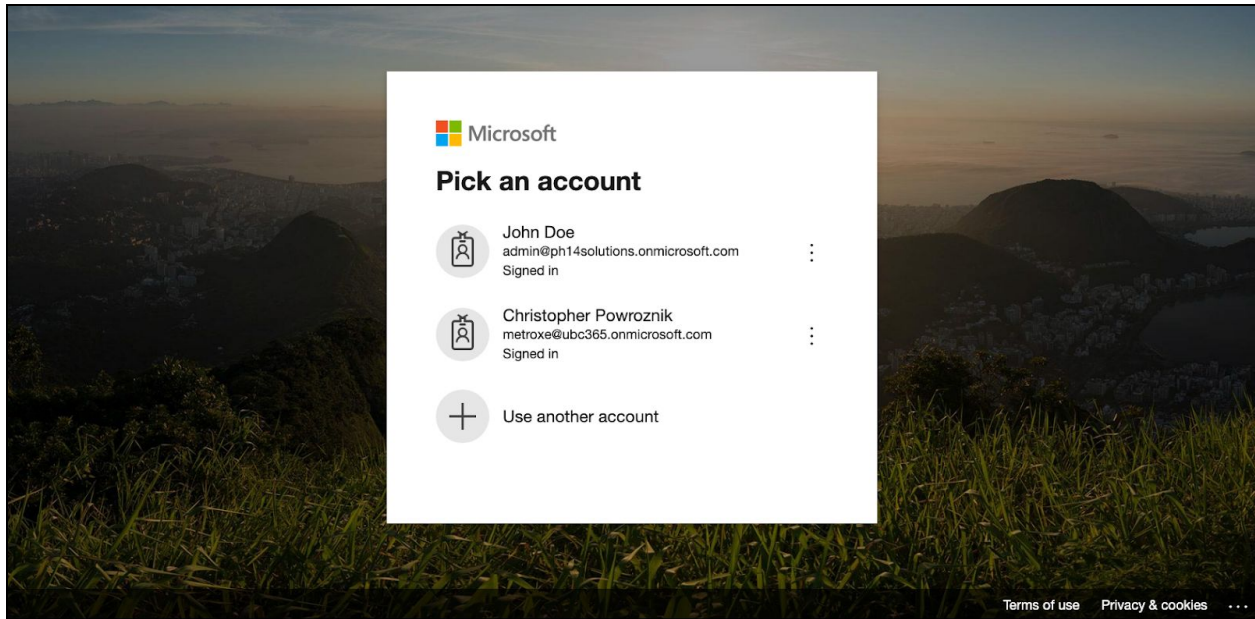
Frontend Acceptance Criteria

Authentication

- ❑ Going to any route on the website while not logged in will redirect to the login screen.



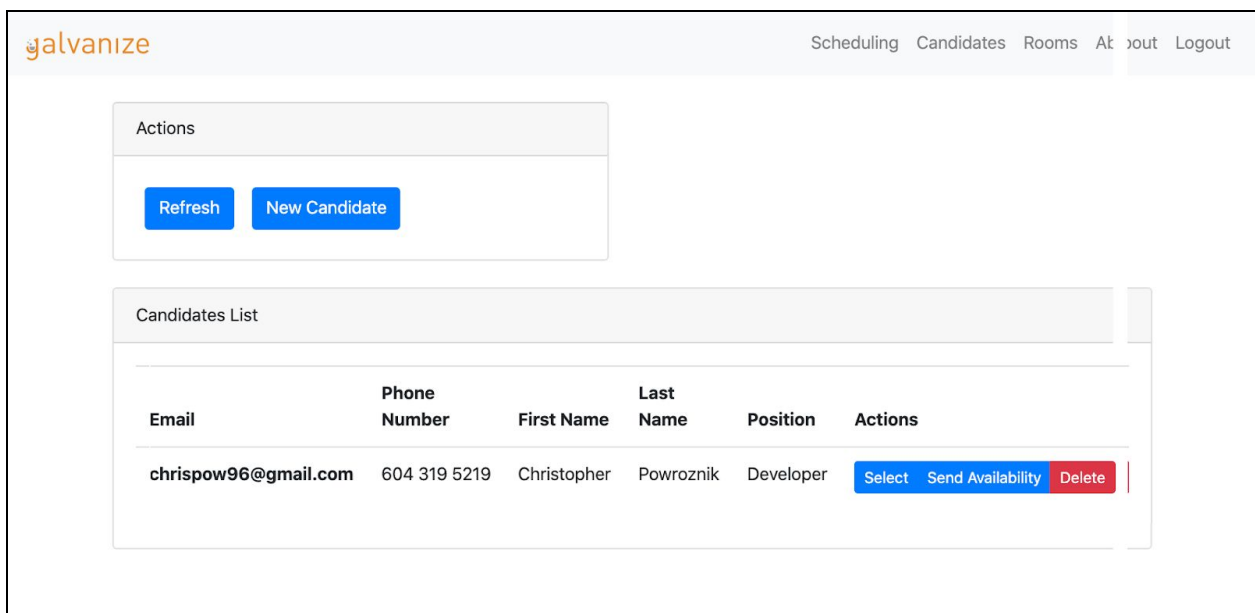
- ❑ The login screen should have a link to access the Microsoft Login portal, on returning it should redirect the user to the candidates menu. In the case where the credentials where the credentials are not correct, the user will be taken back to the login.



- ❑ Pressing 'Logout' should result in being redirected back to the login and access to any unauthenticated routes will be lost until the next login.

Candidate Management Screen

- ❑ Upon entering the screen refresh should be triggered automatically pulling all candidates into the menu.



- ❑ The 'Create New Candidate' form should be shown when pressing the 'New Candidate' button.

Create New Candidate

Enter all the details of a new candidate and save them.

Email

First Name

Last Name

Phone Number

Position

Notes

Create

- ❑ Entering a properly formatted email into the 'Create New Candidate' form and pressing 'Create' will result in a Candidate being created with 'N/A' as the input for all other fields besides email.
- ❑ Pressing 'Create' on the 'Create New Candidate' form with no information will not allow a candidate to be created.
- ❑ Entering a properly formatted email into the 'Create New Candidate' form, then filling out an optional amount of the proceeding fields will result in a candidate being created with 'N/A' in all input that were left blank and the input from the form shown for their respective labels.
- ❑ Pressing 'Select' on a candidate will open the 'Edit Candidate' menu with all the information pre-inflated from the candidate
- ❑ Editing any information and pressing 'Update' on the 'Edit Candidate' form will result in the Candidates information being changed.
- ❑ Pressing 'Send Availability' on a candidate will result in a toast notification showing the URL that has now been emailed to the candidate. An email is sent to the candidate's email address.
- ❑ Pressing 'Delete' will result in the candidate being deleted from the system.
- ❑ Pressing 'Refresh' will reset all open form and load the newest information on Candidates

Scheduling

- ❑ Upon entering the Scheduling page, the 'Candidates List' will be shown of all candidates in the system.

Candidates List					
Email	Phone Number	First Name	Last Name	Position	Actions
chrispow96@gmail.com	604 319 5219	Christopher	Powroznik	Developer	<button>Select</button>

- ❑ Pressing select on a candidate will bring up the list of interviewers and their respective options.

Interviewers				
ID	First Name	Last Name	Preference	Time Needed
272373f6-d0af-47d8-bd26-c916e257bb20	Peter	Parker	Alone ▾	0 minutes ▾
4c294c2d-c014-4538-a9e7-9fb1e6628ba0	Perry	Liao	Alone ▾	0 minutes ▾
<button>Generate Schedules</button>				

- ❑ Preferences should be selectable and the scheduling software should display a bias towards the preference by always showing the combination if the interviewers are available together.
- ❑ Putting a minute amount of 0 will result in the interviewer not getting an interview with the candidate despite preference.
- ❑ Putting in minute amounts greater than 0 on any interviewer and pressing 'Generate Schedules' will result in schedules being returned (only if the availability and the interviewers outlook calendar have an overlap in an available room).
- ❑ 'Schedules' should be displayed if a valid input from the 'Interviewers' section.

Schedules

Option #1 <u>Meeting #1</u> Nov 7th 10:00 am - Nov 7th 11:00 am Meeting Room #1 Andrea Tamez Masahiro Toyomura <u>Meeting #2</u> Nov 7th 11:00 am - Nov 7th 11:30 am Meeting Room #2 Cindy Hsu Select	Option #2 <u>Meeting #1</u> Nov 7th 1:30 pm - Nov 7th 2:30 pm Meeting Room #3 Andrea Tamez Masahiro Toyomura Select
--	--

- ❑ Selecting a schedule will result in a summary of everything selected so far.

Actions

Candidate
name: Christopher Powroznik
email: christopher@frameonesoftware.com
phone number: (604) 319-5219
position: Lead Developer

<u>Meeting #1</u> Nov 7th 10:00 am - Nov 7th 11:00 am Meeting Room #1 Andrea Tamez Masahiro Toyomura	<u>Meeting #2</u> Nov 7th 11:00 am - Nov 7th 11:30 am Meeting Room #2 Cindy Hsu
--	--

Schedule & Send Emails

- ❑ Pressing 'Schedule & Send Emails' will result in an email being sent to the candidate with the information on the meeting. The meeting will also be scheduled with all applicable interviewers and the rooms availability is updated.

Rooms

- ❑ Accessing the 'Rooms' page will trigger a refresh command which will pull all of the rooms. Each room will indicate if it is available for scheduling.
- ❑ Pressing the toggle will change of a room is available for scheduling or not.

About

- ❑ The 'About' page displays all of the developers names and links to their Githubs.
- ❑ The 'About' page will display buttons to each of the documents provided throughout the Galvanize project. Pressing the buttons will download the respective file.
- ❑ The file that is downloaded by pressing 'Development Documentation' is displayed on the 'About' page rendered from the same markdown that is downloaded via the button.