

Galvanize

Internal Design Document

Nov 23, 2019

Version 3.0

pH14 Solutions

Andrea Tamez

Braxton J Hall

Christopher Powroznik

Cindy Hsu

Kwangsoo Yeo

Masahiro Toyomura

Table of Contents

Table of Contents	1
Introduction	2
Goals	3
Assumptions	3
Programming Environment	3
Production and Test Environments	4
Deploying Changes	4
Data Design	4
API Design	5
Software Architecture	7
Program Hierarchy Diagrams	7
Performance (Timing)	9
Capacity	9
Security & Authentication	9
Algorithms	10
Notable Trade-offs	10
Notable Risks	10
User Interface	11
Login Screen	11
Candidate Management Screen (Updating Candidate Info)	12
Candidate Management Screen (Creating New Candidate)	13
Scheduling Screen	14

Introduction

The purpose of this document is to explain the internal design, the architecture and the trade offs in a concise manner. We will describe the choices we made and why we made those decisions.

Galvanize would like to improve the process of scheduling interviews for new candidates and the interviewers. The project will result in a web based tool for administrator to set-up interviews between candidates and interviewers. The tool will allow a streamlined approach of booking rooms and interviewers with various preference options. The system will also notify all attendees to the interviews of their schedules and facilitate the entire process.

Goals

Create a web-based system that will facilitate scheduling of Candidates and Interviewers at Galvanize. This is comprised of the following parts:

- React front-end for interviewers and interviewees to submit availabilities and coordinate schedules
- An administrative dashboard for facilitating room registration and user coordination
- Node/Express backend with adapters that connect to the frontend, and a Controller for interfacing with Microsoft Graph

Assumptions

- All employees has a Microsoft Outlook User Profile
- All Candidates have an email address and browser that supports ES5
- Users at Galvanize have internet access

Programming Environment

Programming Tools & Languages

1. React v16.10 (Front-End)
2. Docker v19.03.2 (Backend)
3. Node v10.16.3 (Backend)
4. Express v4.17.1 (Front-End)
5. TypeScript v3.6.3 (Front-End and Backend)

Database

1. DynamoDB hosted by AWS (Backend)

IDE

1. JetBrains IDEs

Source Control

1. GitHub

Deployment

1. AWS (Amazon Web Services)
2. GCP (Google Cloud Platform)

UML Tools

1. Lucid Chart

Production and Test Environments

We are using Continuous Integration and Testing for this project. On every push to our development and master branches, a new build will be initiated for the project, along with a new run of the test suite, and this build will be deployed to a live link should it be successful. These builds are saved as Docker images, meaning their respective containers will be able to run in any environment with Docker installed. Should the project need to be hosted on another static server, we have written a docker-compose configuration such that the entire system can be deployed with a single command after cloning.

For testing, we will be using Mocha to run the tests, with assertions from Chai. We will be including a full unit test suite to cover the backend, with one separate test file for each class measuring correctness. We will be using statement coverage as our metric for how validated our system is, with a minimum of 95% statement coverage in our finished system.

Deploying Changes

Pushing to production is a simple manner. There is a .yaml file in the directory of the project. This can be hooked into by any cloud service such as AWS or Google. It would run the compiling of all the Docker Containers and push to their respective services. This means that patching and maintenance could be coded, tested locally and automatically pushed to production, by just pushing to master. There is also a development server for internal use. This can be redeployed by pushing to the develop branch. The project would follow standard git flow, requiring that master could only be updated from the develop branch, to avoid misaligned deployments.

Data Design

Since we are integrating with the Office 365 environment, we are able to use the data stores of the Microsoft Graph service. These are broken down into the following interfaces.

```
interface groups {
    "id": string,
    "displayName": string
}
interface members {
    "id": string,
    "firstName": string,
    "lastName": string,
    "email": string
}
interface rooms {
    "id": string,
    "name": string,
    "email": string,
    "capacity": number
}
interface schedule {
    "scheduleId": string;
    "availabilityView": string;
    "scheduleItems": [{
        "isPrivate": boolean,
        "status": string,
        "subject": string,
        "location": string,
        "start": {
            "dateTime": string,
            "timeZone": string
        },
        "end": {
            "dateTime": string,
            "timeZone": string
        }
    }];
    "workingHours": {
        "daysOfWeek": [string],
        "startTime": string,
        "endTime": string,
        "timeZone": {
            "name": string
        }
    }
}
```

The interfaces displayed above are a small subset of the complete interfaces, but contain the members we intend to use. For more information on these interfaces, you can visit the Microsoft Graph resource pages for

- User
 - <https://docs.microsoft.com/en-us/graph/api/resources/user?view=graph-rest-1.0>
- Group
 - <https://docs.microsoft.com/en-us/graph/api/group-list?view=graph-rest-1.0&tabs=javascript>
 - <https://docs.microsoft.com/en-us/graph/api/group-list-members?view=graph-rest-1.0&tabs=javascript>
- Room
 - <https://docs.microsoft.com/en-us/graph/api/resources/room?view=graph-rest-beta>
- Schedule
 - <https://docs.microsoft.com/en-us/graph/api/calendar-getschedule?view=graph-rest-1.0&tabs=javascript>

API Design

Candidate Routes

```
submitAvailability(candidateID: string, availability: IAvailability):  
Promise<IAPIResponse>  
getCandidateByID(candidateID: string): Promise<IAPIResponse<ICandidate>>;
```

Staff Routes

```
createCandidate(token: string, candidate: ICandidate):  
Promise<IAPIResponse<ICandidate>>;  
sendAvailabilityEmail(token: string, candidate: ICandidate): Promise<IAPIResponse>;  
getSchedules(token: string, options: IGetSchedulesOptions):  
Promise<IAPIResponse<ISchedule[]>>;  
confirmSchedule(token: string, schedule: ISchedule): Promise<IAPIResponse>;  
cancelSchedule(token: string, candidate: ICandidate): Promise<IAPIResponse>;  
getCandidates(token: string): Promise<IAPIResponse<ICandidate[]>>;  
deleteCandidate(token: string, candidate: ICandidate): Promise<IAPIResponse>;  
updateCandidate(token: string, candidate: ICandidate): Promise<IAPIResponse>;  
getInterviewers(token: string, groupName: string):  
Promise<IAPIResponse<IInterviewer[]>>;  
getRooms(token: string): Promise<IAPIResponse<IRoom[]>>;  
toggleEligibility(token: string, room: IRoom): Promise<IAPIResponse>;
```

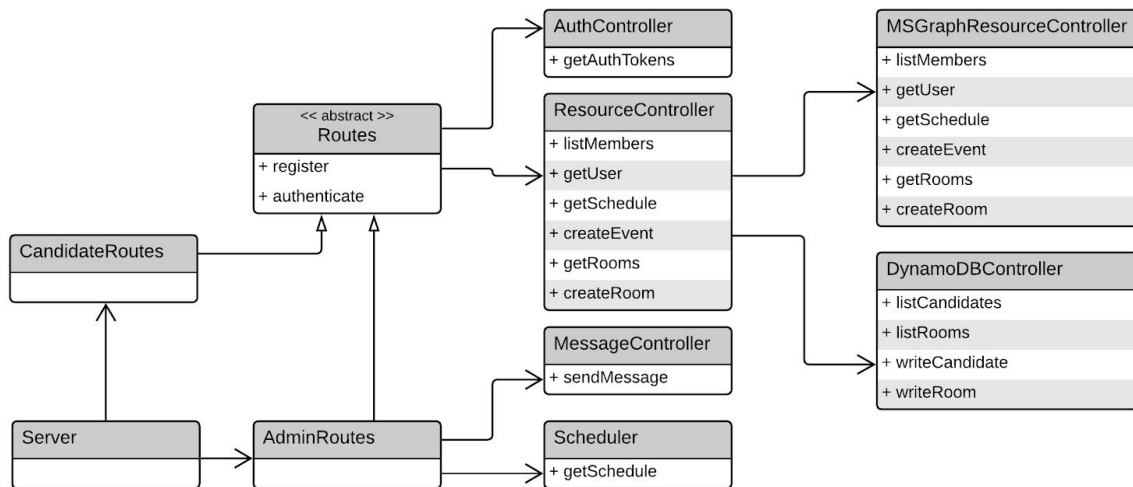
Miscellaneous

```
loginRedirectURL(): string  
checkToken(token: string): Promise<IAPIResponse<boolean>>;  
health(): Promise<IAPIResponse<boolean>>;  
logout(token: string): Promise<IAPIResponse>;  
urls: {[key: string]: string};  
fullURLs: {[key: string]: string};
```

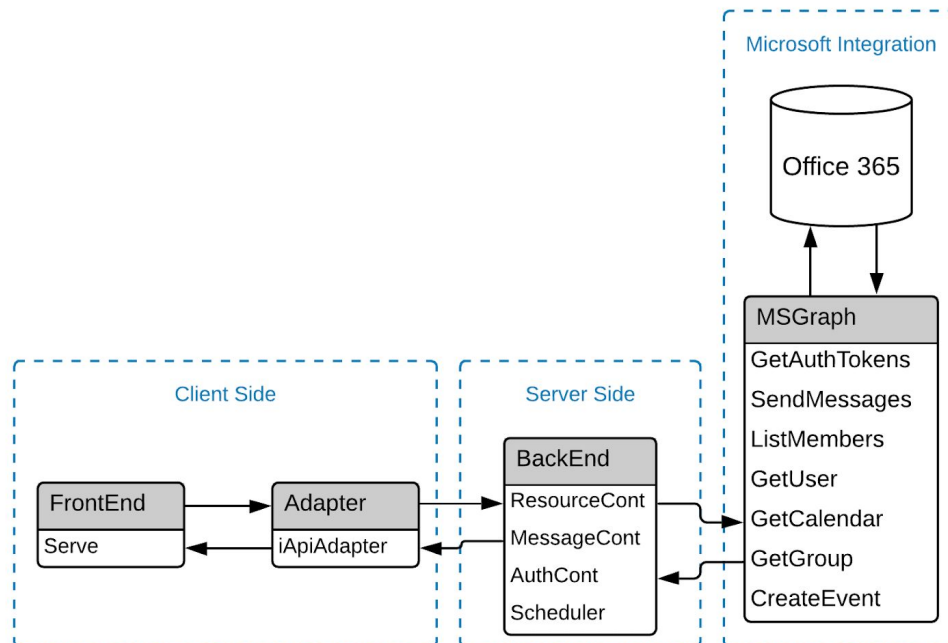
Software Architecture

Program Hierarchy Diagrams

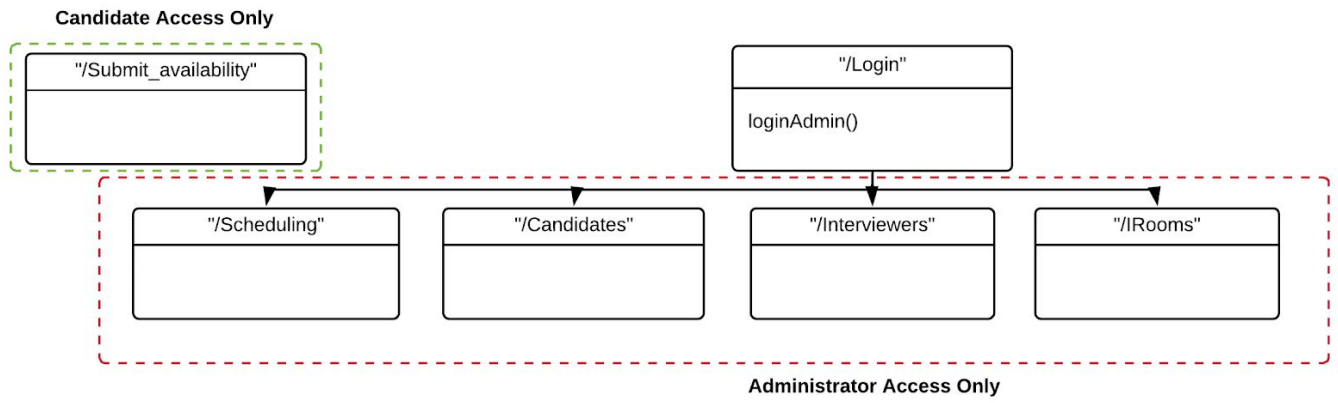
Backend Class Diagram



Deployment Diagram



Frontend Resources Diagram



Performance (Timing)

As per the document docs the project will be able to achieve a response time of at most 2 seconds for each request. However, there are certain APIs from Microsoft Graph, which tend to have a long request time. In particular the **/resource/schedules** route is solving an NP-complete algorithm and would scale with the number of people in the company. However, we still believe with these limitations, our APIs will be under 2 seconds.

The application is stateless. One instance using 1 vcpu and 128mb of memory, will be able to run up to 80 concurrent requests, without dips in performance. This will be able to facilitate the 5 concurrent users that was a requirement. We are using Alpine for our Docker instance, meaning our container size is extremely small and will require minimal amount of memory. Because of this, the system would afford to be shut down between requests and rebooted by a load balancer to facilitate each request. This would save on costs, and wouldn't hinder response times dramatically. This would be possible with a Google cloud Run or a Kubernetes setup on AWS.

Capacity

In Production, the system will be almost entirely stateless. The advantage of this is scalability. Since every request doesn't require previous context, you could spin up any number of instances in concurrency and load balance between them. The bottleneck would be the latency of Microsoft Graph API calls, which are out of the control of this project. However, it would be unlikely for the system to have a load beyond what could be managed by a single instance. There would be over 99% uptime, with this setup. Our Single Point of Failure is the authentication system. Should the authentication stop working, the entire system would become unusable. However, authentication is being handled by Microsoft Graph, thus we manage to have their level of reliability by association.

The one place we do have state is in our DynamoDB, which maintains a current list of candidates, rooms, and their availability. We do not have to worry about capacity here because the database can handle millions of rows, and it will take a long time to reach a list of millions of candidates.

Security & Authentication

Microsoft Graph allows us to use the company credentials to generate authentication. This works using the [/authorize](#) api. This will allow us to login using the email and password from their Office 365 login. This will return a unique token for that user, than can be used to make the subsequent requests to the Microsoft Graph APIs. The token will be stored on the frontend and in our DynamoDB. When requests are required, the token will be passed to the adapters to be used on the backend, for the REST requests to Microsoft Graph. On logout, this token will be deleted and eventually expired from the Microsoft Graph service and DynamoDB.

Although the Candidate is *not* an authenticated user, they still provide some risk to the system. Should their availability request form URL be mistakenly leaked, the availability could be compromised, as after a submission the link is still usable. This means that a bad actor could repeatedly request loading of a candidate availability form in a denial of service attack.

In the availability form, no data from Galvanize is sent to the user. Lastly, should the Candidate accept an invitation from Galvanize, no sensitive data will be provided in the schedule except for room numbers, times of day, and the first names' of Interviewers.

Algorithms

We will be using a complex algorithm. Our `/resource/schedules` api will calculate viable schedules and return them. The algorithm supports cycles and chains in the preferences array, as the system will attempt to group everyone in the cycle/chain together into one interview.

The algorithm is a greedy algorithm that works in several steps.

1. Acquire all schedules for rooms and necessary interviewers within the Candidate's availability range
2. Find the overlapping time between each room and the Candidate Availability
3. Score each room based on four features
 - a. Longest available timeslot
 - b. Average available timeslot
 - c. Average Overlap Size with Interviewers
 - d. Capacity
4. Going down the scored list, one at a time a room is filled starting with the largest timeslot, followed by the closest timeslots.
5. If a group cannot fit in a room or has mismatching availability, members are ejected from the group one at a time and the meeting is reattempted.
6. The full algorithm is run 10 times, and the schedule with fewest change overs to other rooms, the schedule with the most interviewers actually scheduled, and a schedule that does the best on average in both regards, are returned.

This will return a list of at most three suggestions which the administrator can click on to select the desired schedule.

Notable Trade-offs

The most notable trade-off in our application is how broke-up our data into two different storage locations, DynamoDB and Microsoft Graph. We chose to store our candidate and room information in DynamoDB because Microsoft Graph didn't allow for us to rank rooms so we created our own algorithm to make this possible. This improves the efficiency of our scheduling

algorithm. We rely on Microsoft Graph for representing the associations between interviewers and rooms - the graph operations make the algorithm easier to understand.

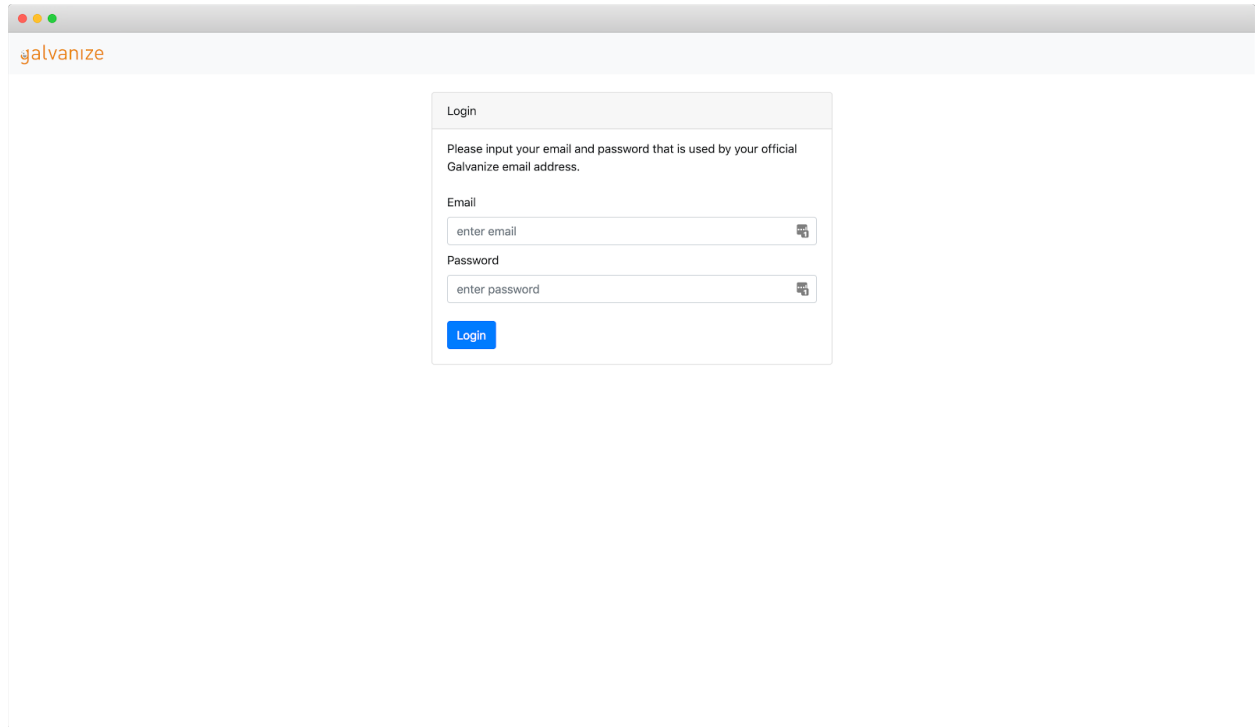
Notable Risks

1. The Microsoft Graph API has the potential to change, which would cause our program to have unpredictable behaviour. Although unlikely for most endpoints, this is most pertinent with the Place resource type, as that part of the API is currently in beta. Although even less likely, we also face the possibility of complete deprecation of Microsoft Graph.
2. As we are hosting the service of AWS in production, we are subject to their uptime. Should AWS go down in the region, so too would our system. Our DynamoDB is also hosted in AWS and will suffer from the same risks.
3. As our data is not stored in a database locally, we are not the only users of the Microsoft Graph data that our system depends on. Therefore, other parts of Galvanize could potentially interfere with the system by mutating data without our knowledge. This would lead to unpredictable behaviour.
4. As our implementation requires heavy use of Microsoft Graph directories, there is also the risk that Galvanize does not use these directories, which would directly impact the usability of the system.

Note that overall the risk of using the system is low. Should Galvanize become dependant on it, even this dependency is a low risk. Should the system go down for whatever reason, the Microsoft Office 365 functionalities would be unaffected, and interviews could still be manually scheduled as they were before.

User Interface

Login Screen



The image shows a web browser window with the Galvanize logo in the top left corner. The main content area features a centered login form. The form has a title 'Login' and a message: 'Please input your email and password that is used by your official Galvanize email address.' Below this, there are two input fields: 'Email' with a placeholder 'enter email' and 'Password' with a placeholder 'enter password'. Both fields have a small icon on the right side. At the bottom of the form is a blue 'Login' button.

galvanize

Login

Please input your email and password that is used by your official Galvanize email address.

Email

enter email

Password

enter password

Login

Candidate Management Screen (Updating Candidate Info)

galvanize

CandidatesSchedulingInterviewersHuman ResourcesLogout

Actions

RefreshNew Candidate

Candidates List

ID	Email	Phone Number	First Name	Last Name	Position	
1	christopher@frameonesoftware.com	(604) 319-5219	Christopher	Powroznik	Lead Developer	SelectDelete
2	braxton.hall@alumni.ubc.ca	(604) 555-1234	Braxton	Hall	Musical Programmer	SelectDelete
3	kyeo@gmail.com	(604) 345-7890	Kwangsoo	Yeo	Instructor	SelectDelete

Edit Candidate

You may edit all the details of a candidate and save them.

ID

1

Email

christopher@frameonesoftware.com

First Name

Christopher

Last Name

Powroznik

Phone Number

(604) 319-5219

Position

Lead Developer

Notes

Possibly the greatest candidate of all time.

Update

Candidate Management Screen (Creating New Candidate)

galvanize

CandidatesSchedulingInterviewersHuman ResourcesLogout

Actions

Refresh

New Candidate

Candidates List

ID	Email	Phone Number	First Name	Last Name	Position	
1	christopher@frameonesoftware.com	(604) 319-5219	Christopher	Powroznik	Lead Developer	<div>SelectDelete</div>
2	braxton.hall@alumni.ubc.ca	(604) 555-1234	Braxton	Hall	Musical Programmer	<div>SelectDelete</div>
3	kyeo@gmail.com	(604) 345-7890	Kwangsoo	Yeo	Instructor	<div>SelectDelete</div>

Create New Candidate

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

Email

enter email

First Name

enter first name

Last Name

enter last name

Phone Number

enter phone number

Position

enter position candidate is applying for

Notes

enter any additional notes

Create

Scheduling Screen

galvanize

CandidatesSchedulingInterviewersHuman ResourcesLogout

Candidates List

ID	Email	Phone Number	First Name	Last Name	Position	
1	christopher@frameonesoftware.com	(604) 319-5219	Christopher	Powroznik	Lead Developer	Select
2	braxton.hall@alumni.ubc.ca	(604) 555-1234	Braxton	Hall	Musical Programmer	Select
3	kyeo@gmail.com	(604) 345-7890	Kwangsoo	Yeo	Instructor	Select

Interviewers

ID	First Name	Last Name	Preference	Time Needed
4	Andrea	Tamez	Masahiro Toyomura	60 minutes
5	Masahiro	Toyomura	Andrea Tamez	60 minutes
6	Cindy	Hsu	Cindy Hsu	30 minutes

Schedules

Option #1

Meeting #1
Oct 4th 10:00 am - Oct 4th 11:00 am
Meeting Room #1
Andrea Tamez
Masahiro Toyomura

Select

Option #2

Meeting #1
Oct 4th 1:30 pm - Oct 4th 2:30 pm
Meeting Room #3
Andrea Tamez
Masahiro Toyomura

Select