
# Assignment 7

CSCI 5409: Advanced Topics in Cloud Computing

## Group 13 - Cloud Evangelists


| | |
|---|---|
| Asmita Chaudhari | asmita.chaudhari@dal.ca |
| Malav Jani | malav.jani@dal.ca |
| Naitik Prajapati | naitik.prajapati@dal.ca |
| Tapan Prajapati | tapan.prajapati@dal.ca |
| Parth Parmar | parth.parmar@dal.ca |
| Samkit Shah | sm611862@dal.ca |


July 28, 2020

# Contents

# List of Figures

# List of Tables

# 1 Overview

The document outlines information about implementing angular and Node.js based web application using serverless [1] component to handle user requests to manage the job and part information for three different companies (x, y, and z). The application backend is mainly developed using three serverless components namely Amazon API gateway [2] to create and manage API endpoints, Amazon Lambda [3], event-driven function works as middleware between database and API gateway, and Amazon DynamoDB [4] for storing the data. It also discusses usage of docker [5] for containerization, docker hub [6] to host the containerized image and development of the applications on Amazon Elastic Beanstalk [7], to automatically scale application up and down based on a specific need.

To fulfill the assignment requirement, the frontend of the application is extended from the previous assignment 6 [8]. However, backend of the company x and y has been created from scratch using above mentioned serverless computing components. Further, the application is developed using agile methodology by keeping track of tasks and their progress using Trello [9], software management using FCS GitLab [10], and the collaboration with team members with the help of Microsoft Teams. [11]

## 1.1 Company X

Company X provides the facility of managing the job information. It provides a dedicated user interface to view, add, update, and delete the job information. Additionally, it allows to search for successful order information. The job information is represented by a set of triplets (jobName, partID, qty). The app communicates with a NoSQL database hosted on Amazon DynamoDB, which provides key-value and document database services by making it easy to set up, operate, and scale databases in the cloud.

- Application URL [1]
- Code Repository [2]

The following operations can be performed using the application of Company X:

1. View all available jobs.
2. Add a new job information by providing job name, part id, and quantity. Please note that it enforces below set of validations while adding a new job:

[1] `http://companyxa7-env.eba-pmezmaa7.us-east-1.elasticbeanstalk.com/jobs`
[2] `https://git.cs.dal.ca/chaudhari/company-x-a7`

- Job name, part Id, and quantity are required fields.
- Job name and part Id should be unique.
- Quantity should be 0 or more.

3. Search information by jobname about successful orders.
4. View information about a specific job.
5. View part orders.
6. Update quantity for a specific job. It only allows to update quantity information and that should not be less than 0.
7. Allows to delete a job information.
8. Handles all the corner cases and guides user to recover from an undesired state.

### 1.1.1 User Interface

The section discusses the various features accessed using the User Interface.

Feature 1: *Home* - Display a list of services offered by company X.
`http://companyxa7-env.eba-pmezmaa7.us-east-1.elasticbeanstalk.com/jobs`

Feature 2: *Get All jobs* - Illustrates list of available jobs.
`http://companyxa7-env.eba-pmezmaa7.us-east-1.elasticbeanstalk.com/jobs/displayjobs`

Feature 3: *Get Job* - Asks for Job name and Part Id to search for a specific job record
`http://companyxa7-env.eba-pmezmaa7.us-east-1.elasticbeanstalk.com/jobs/getOneJob`

Feature 4: *Add Job* - Asks for unique Job name, part id along with quantity to create a new job.
`http://companyxa7-env.eba-pmezmaa7.us-east-1.elasticbeanstalk.com/jobs/addjobs`

Feature 5: *Edit Job* - Enable user to update the quantity of job by selecting a job from available list
`http://companyxa7-env.eba-pmezmaa7.us-east-1.elasticbeanstalk.com/jobs/displayjobs`

Feature 6: *Delete Job* - Allows user to delete the job by selecting a particular job from available list
`http://companyxa7-env.eba-pmezmaa7.us-east-1.elasticbeanstalk.com/jobs/displayjobs`

Feature 7: *View Orders* - Shows list of part order with user Id.
`http://companyxa7-env.eba-pmezmaa7.us-east-1.elasticbeanstalk.com/jobs/searchjob`

### 1.1.2 Serverless APIs

The section discusses the various serveless endpoints exposed by the company X:

Endpoint 1: *Jobs - GET* - The API endpoint returns information about all available job in the database.
`https://qcutoolbyi.execute-api.us-east-1.amazonaws.com/Dev/api/jobs`

Endpoint 2: *Jobs - PUT* - Use this endpoint to update the quantity of a specific job record. Pass a JSON object in the body of a PUT request adhering to the expected schema. The endpoint updates the quantity if the JOB information exists, otherwise, it returns an appropriate error message.
`https://qcutoolbyi.execute-api.us-east-1.amazonaws.com/Dev/api/jobs`

Endpoint 3: *Jobs - POST* - Use this endpoint to create a new resource (job information). Pass a JSON object in the body of a POST request as "jobName": "nameOfJob", "partId": IdOfPart, "qty": quantity. The object should follow the expected schema, otherwise, the endpoint returns an error. If a similar record already exists, the endpoint reverts with an appropriate error message.
`https://qcutoolbyi.execute-api.us-east-1.amazonaws.com/Dev/api/jobs`

Endpoint 4: *Jobs - DELETE* - Use this endpoint to delete the quantity of a specific job record. Pass a JSON object in the body of a DELETE request adhering to the expected schema. The endpoint deletes the job if the JOB information exists, otherwise, it returns an appropriate error message.
`https://qcutoolbyi.execute-api.us-east-1.amazonaws.com/Dev/api/jobs`

Endpoint 5: *Ajob - GET* - The API endpoint is useful to fetch job record based on the Jobname and PartID. Note: This Endpoint is internal endpoint and you might get incorrect response while using it directly.
`https://qcutoolbyi.execute-api.us-east-1.amazonaws.com/Dev/api/ajob`

Endpoint 6: *GetallOrders - GET* - The API endpoint returns all the orders available in the database.
`https://qcutoolbyi.execute-api.us-east-1.amazonaws.com/Dev/api/getallorders`

Endpoint 7: *Partorders - POST* - Use this endpoint to create a new resource (part order information). Pass a JSON object in the body of a POST request as "partId": IdOfPart, "jobName": "NameOfJob", "userId": userId ,"qty": quantityOfParts. The object should follow the expected schema, otherwise, the endpoint returns an error. If a similar record already exists, the endpoint reverts with an appropriate error message.
`https://qcutoolbyi.execute-api.us-east-1.amazonaws.com/Dev/api/partorders`

Endpoint 8: *Searchjobs - GET* - The endpoint is useful to search job from part order information. It expects jobname to search the record.
`https://qcutoolbyi.execute-api.us-east-1.amazonaws.com/Dev/api/searchjobs`

## 1.2 Company Y

Company Y provides the facility of managing the part information. It provides a dedicated user interface to view, add, and update part information. It additionally allows a user to search for successful order information. The part information is represented by a set of

triplets (partId, partName, qoh), where partId is a primary key. The app communicates with a NoSQL database created on Amazon DynamoDB.

- Application URL [3]
- Code Repository [4]

The following operations can be performed using the application of Company Y:

1. View all available part information.
2. Add a new part information by providing part id, part name, and quantity-on-hand (qoh). Please note that it enforces below set of validations while adding a new job:
   - Part id, part name, and quantity-on-hand (qoh) are required fields.
   - Part Id should be unique.
   - Quantity-on-hand (qoh) should be 0 or more.
3. Search information about successful orders.
4. View information about specific part.
5. View part orders.
6. Update quantity for a specific part. It allows to update qoh and part name information.
7. Handles all the corner cases and guides user to recover from an undesired state.

### 1.2.1 User Interface

The section discusses the various features accessed using the User Interface.

Feature 1: *View Parts* - Display a list of available parts information along with search functionality.
*http://companyyui-env-1.eba-kffumsec.us-east-1.elasticbeanstalk.com/parts*

Feature 2: *Edit Parts* - Allows user to update the quantity of part by selecting part from the available part list,
*http://companyyui-env-1.eba-kffumsec.us-east-1.elasticbeanstalk.com/parts*

Feature 3: *View Order* - Shows list of part order with user Id.
*http://companyyui-env-1.eba-kffumsec.us-east-1.elasticbeanstalk.com/vieworders*

Feature 4: *Search Parts* - Find specific Parts based on Part Id.
*http://companyyui-env-1.eba-kffumsec.us-east-1.elasticbeanstalk.com/parts*

[3] http://companyyui-env-1.eba-kffumsec.us-east-1.elasticbeanstalk.com/parts
[4] https://git.cs.dal.ca/mjani/company-y-a7

### 1.2.2 Serverless APIs

The section discusses the various serveless endpoints exposed by the company X:

Endpoint 1: *Partorders - GET* - The API endpoint returns information about all available partorders in the database.
*https://mjq7oae6m9.execute-api.us-east-1.amazonaws.com/companyY/api/partorders*

Endpoint 2: *Partorders - POST* - Use this endpoint to create a new resource (part order information). Pass a JSON object in the body of a POST request as "partId": IdOfPart, "jobName": "NameOfJob", "userId": userId ,"qty": quantityOfParts. The object should follow the expected schema, otherwise, the endpoint returns an error. If a similar record already exists, the endpoint reverts with an appropriate error message.
*https://mjq7oae6m9.execute-api.us-east-1.amazonaws.com/companyY/api/partorders*

Endpoint 3: *Parts - GET* - The API endpoint returns information about all available part in the database.
*https://mjq7oae6m9.execute-api.us-east-1.amazonaws.com/companyY/api/parts*

Endpoint 4: *Parts - POST* - Use this endpoint to create a new resource (part information). Pass a JSON object in the body of a POST request as "partId": IdOfPart, "partName": "NameOfPart", "qoh": quantityOfParts. The object should follow the expected schema, otherwise, the endpoint returns an error. If a similar record already exists, the endpoint reverts with an appropriate error message.
*https://mjq7oae6m9.execute-api.us-east-1.amazonaws.com/companyY/api/parts*

Endpoint 5: *Parts - PUT* - Use this endpoint to update the quantity of a specific part record. Pass a JSON object in the body of a PUT request adhering to the expected schema. The endpoint updates the quantity if the part information exists, otherwise, it returns an appropriate error message.
*https://mjq7oae6m9.execute-api.us-east-1.amazonaws.com/companyY/api/parts*

## 1.3 Company Z

Company Z works with both job and part information. It provides a dedicated user interface to view list of jobs, search part information, place order, and login to the system. The app communicates with a NoSQL database created on Amazon DynamoDB.

- Application URL [5]
- Code Repository [6]

The following operations can be performed using the application:

1. Search Job by job name.
2. Get all parts associated with job.

[5] `http://companyza7-env.eba-59sjk5ev.us-east-1.elasticbeanstalk.com/home`
[6] `https://git.cs.dal.ca/tprajapati/company-z-a7`

3. Place order, for that user first need to log into the system.

4. Handles all the corner cases and guides user to recover from an undesired state.

### 1.3.1 User Interface

The section discusses the various features accessed using the User Interface.

Feature 1: *Home* - Displays list of available job with search functionality.
`http://companyza7-env.eba-59sjk5ev.us-east-1.elasticbeanstalk.com/home`

Feature 2: *View Part information and place order* - Displays information about required part for a job and option to place order. Note: Provide valid jobname in the url.
`http://companyza7-env.eba-59sjk5ev.us-east-1.elasticbeanstalk.com/parts/:jobName`

Feature 3: *Login* - Ask for valid credentials to login to the system.
`http://companyza7-env.eba-59sjk5ev.us-east-1.elasticbeanstalk.com/login`

### 1.3.2 Node.js - Express.js APIs

The section discusses the various serveless endpoints exposed by the company Z:

Endpoint 1: *JobParts - POST* - Use this endpoint to create a new resource (order information). Pass a JSON object in the body of a POST request as "partId": IdOfPart, "jobName": "NameOfJob", "userID": userID,"qty": quantityOfParts, "results": "OrderStatus". The object should follow the expected schema, otherwise, the endpoint returns an error. If a similar record already exists, the endpoint reverts with an appropriate error message.
`http://companyzendpoints-env.eba-z3p9pymd.us-east-1.elasticbeanstalk.com/api/jobparts`

Endpoint 2: *Searches - POST* - Use this endpoint to create a new resource (search information). Pass a JSON object in the body of a POST request as "jobName":"NameOfJob". The object should follow the expected schema, otherwise, the endpoint returns an error. If a similar record already exists, the endpoint reverts with an appropriate error message.
`http://companyzendpoints-env.eba-z3p9pymd.us-east-1.elasticbeanstalk.com/api/searches`

Endpoint 3: *JobOrder - GET* - The endpoint is useful to fetch information about a specific order. It expects two parameters, namely userid and jobname. It then returns a specific job information in case a matching record exist, otherwise it returns an error message.
`http://companyzendpoints-env.eba-z3p9pymd.us-east-1.elasticbeanstalk.com/:userid/:jobName`

Endpoint 4: *Authenticate - POST* - The endpoint is useful to authenticate user. It returns JSON Web token if the provided credentials are valid.
`http://companyzendpoints-env.eba-z3p9pymd.us-east-1.elasticbeanstalk.com/api/jobparts/authenticate`

# 2 High-level Architecture

This section gives an overview about the high-level architecture of all three application (company x, y, and z) and front and backend directory organization.

## 2.1 Generalized Architecture for Company X and Y

Figure 1 illustrates a high-level architecture for company X and Y. It is divided into four main verticals: 1) Users, 2) Front-end, 3) Backend, and 4) Database. The diagrams in 1 and 2 are created using LucidChart. [12]

Users mainly interact with the front-end which is served by various browsers (I.e., Google chrome, Firefox, and Safari) of different set of devices. (I.e., Desktop and Laptop).

The front-end of the application is containerized using Docker and deployed on Elastic Beanstalk cloud platform as a managed application, while back-end is deployed as lambda function which interact with Amazon API gateway to get user requests and based on user request it perform operation on DynamoDB.

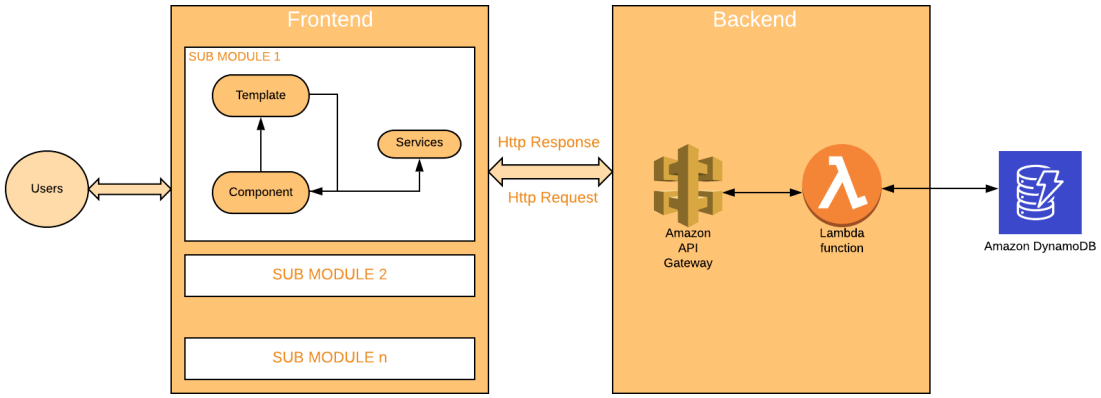


Figure 1: High Level Architecture for Companies X and Y

When user request for a service, Http request comes to the Amazon API gateway. It will trigger the specific endpoint of lambda function based on the request. Lambda function process request by creating database connection and send response back to the API gateway, which will further response to the client.

The front-end part is logically divided into sub-modules providing different features of our application. (I.e., Jobs Module or Parts Module) A single module contains a set of different components and their associated templates. The component contains a presentation logic for an individual template and there is a bi-directional communication between those

two. The component delegates the business logic to the service layer which is handled by various front-end services. A service is responsible for performing business logic and communicating with the backend using the HTTP Module.

The last pillar of the architecture is a NoSQL database created on Amazon DynamoDB to manage different tables for the company X and Y.

## 2.2 Generalized Architecture for Company Z

The architecture of company z is same as we had defined in assignment 6 [8], and the details are carry-forwared from it. Figure 2 illustrates a high-level architecture for company Z. It is divided into four main verticals: 1) Users, 2) Front-end, 3) Backend, and 4) Database.

Users mainly interact with the front-end which is served by various browsers (I.e., Google chrome, Firefox, and Safari) of different set of devices. (I.e., Desktop and Laptop).

The front-end and backend are containerized using Docker and deployed on the Elastic Beanstalk cloud platform as a managed application.

The front-end part is logically divided into sub-modules providing different features of our application. (I.e., Jobs Module or Parts Module) A single module contains a set of different components and their associated templates. The component contains a presentation logic for an individual template and there is a bi-directional communication between those two. The component delegates the business logic to the service layer which is handled by various front-end services. A service is responsible for performing business logic and communicating with the backend using the HTTP Module.

The backend of our application follows a model-view-controller pattern and it is logically divided per resource entity. (I.e., Job, Order, or User) The controller receives HTTP requests from the front-end services. It then communicates with the route authenticator to verify whether it contains a valid access token or not. If authenticated, it checks corresponding schema validator to verify the received data. If either of the preceding two processes fail, it sends an appropriate http response back to the client. If successful, the controller delegates the call to the corresponding service. The service is responsible for performing business logic and communicating with the database. After the process completion, it returns a resource back to the controller.

The last pillar of the architecture is a NoSQL database created on Amazon DynamoDB to manage different tables for the company Z.

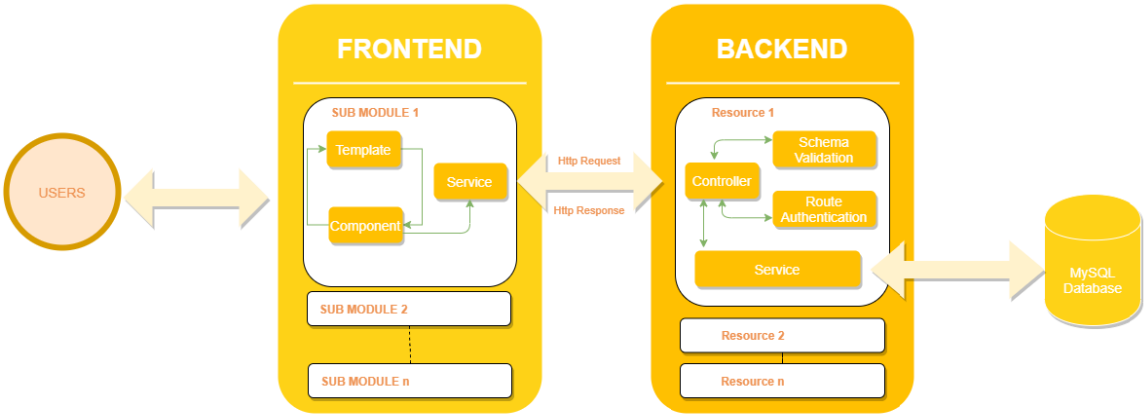


Figure 2: High Level Architecture for Company Z

## 2.3 Backend Organization

The backend of the company x and y is in the form of lambda functions deployed in different student account. So, in order to easily access the lambda function, we kept it under the server directory of each company.

- Company X Serverless Lambda Functions [7]
- Company Y Serverless Lambda Functions [8]

[7] https://git.cs.dal.ca/mjani/company-y-a7/-/tree/master/server
[8] https://git.cs.dal.ca/chaudhari/company-x-a7/-/tree/master/server

# 3 Database

The requirement for this assignment is to use serverless computing [1] for at least each of company X and Y, which should support the same functionality as assignment 6. [8] To use serverless computing for Company X and Company Y, we have used DynamoDB [4] provided by AWS. Tables managed by Company X and Company Y are created in the AWS DynamoDB. All the tables are independent of each other, moreover, companies have no dependencies for the data stored in individual table.

Apart from that, tables for Company Z are remained on Amazon RDS [13] as used in the assignment 6. These tables are also independent from tables of Company X and Company Y.

## 3.1 Configuring AWS DynamoDB: Tables of Companies X and Y

DynamoDB is a NoSQL database. In other words, it does not use concept of rows and columns. Instead, data is stored in the form of objects that contains attributes that can be of any type (an object also).

### 3.1.1 DynamoDB Setup

This section describes steps to create a table on DynamoDB. AWS provides a single database instance of DynamoDB, where you can create all the tables.

Step 1: *Access AWS Console* - Login to AWS Management Console [14] using your credentials which takes you to the dashboard. Find DynamoDB under Database and click to open DynamoDB Dashboard as shown in figure 3.

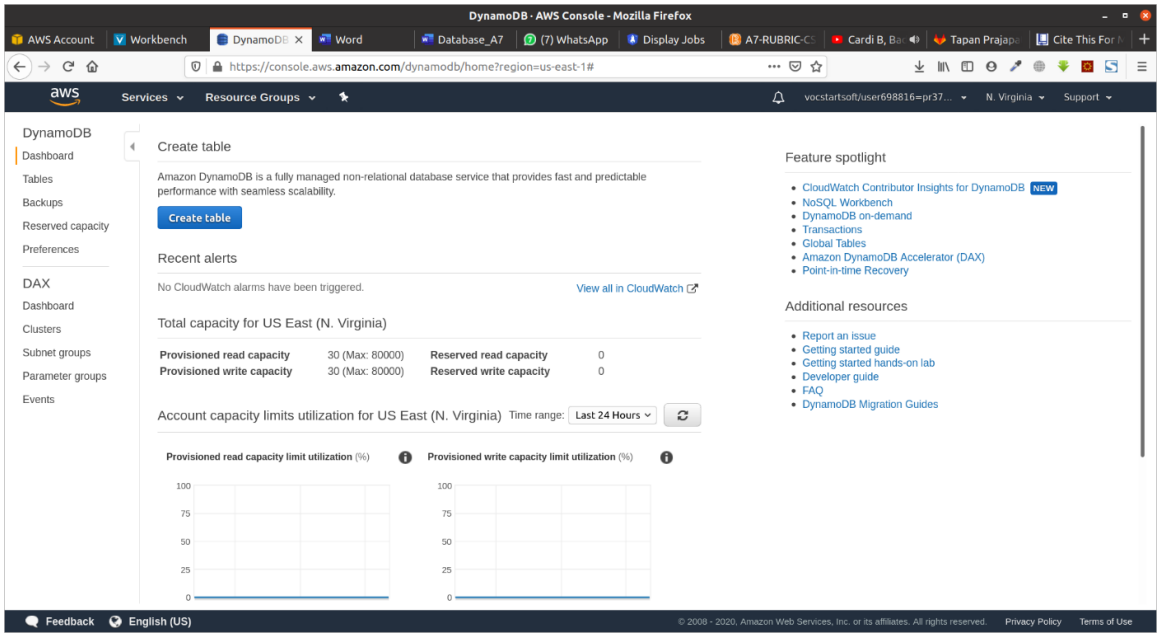


Figure 3: AWS DynamoDb Dashboard

Step 2: *Create Table* - Click on *Create table* button as shown in figure 3. It opens up a form as illustrated in figure 4 Enter name of table and primary key. If there is a need to use two attributes as primary key, click on checkbox Add sort key and enter attribute name. Select type of primary Key and sort key from dropdown on right of both key fields. Leave everything else default. Now click on *Create* button in the bottom. It will show overview of created table after successful creation of the table (refer figure 5).

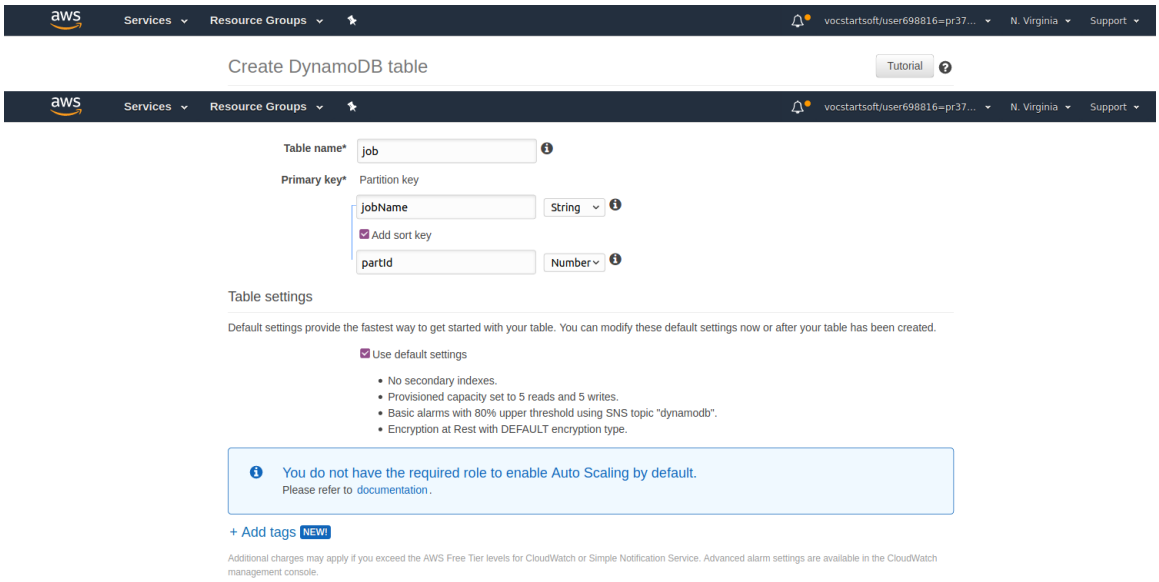


Figure 4: Create DynamoDb table

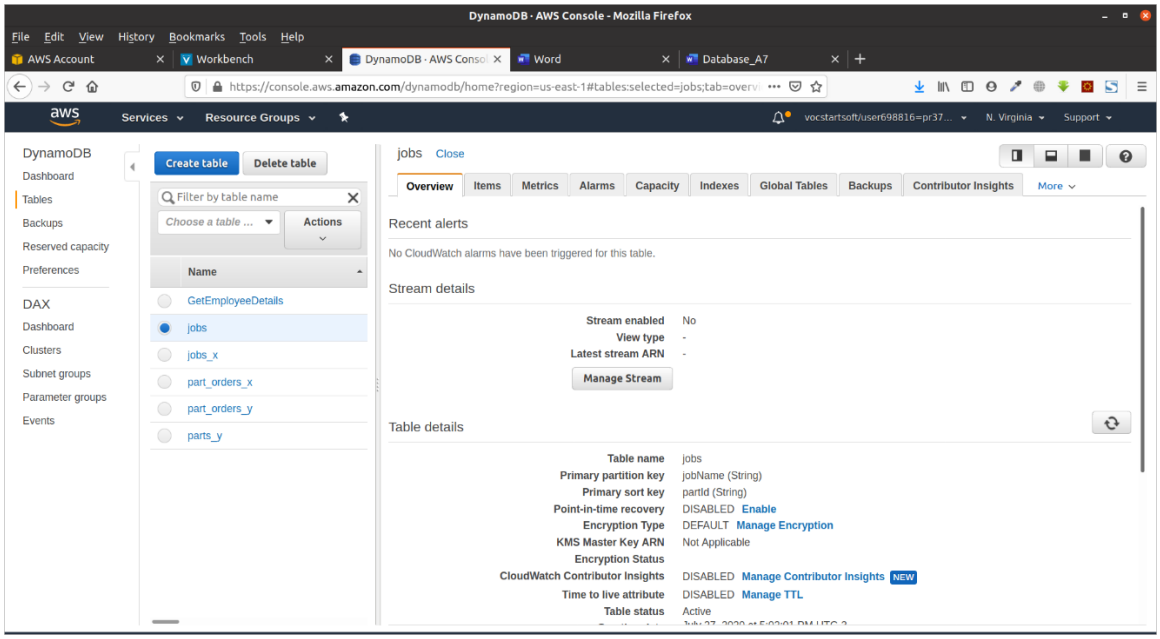


Figure 5: Overview - DynamoDB table

Step 3: *Add Items* - From the overview section, click on *Items* tab from the top and click on *Create item* as shown in figure 6 . Primary Key and Sort Key attribute will be available with unassigned values. New attribute can be added by clicking on add circle on left of available attributes. Select Append and choose data type. Enter name and value of new attribute. Once item is ready to be created, click

on *Save* button on bottom as shown in figure 7

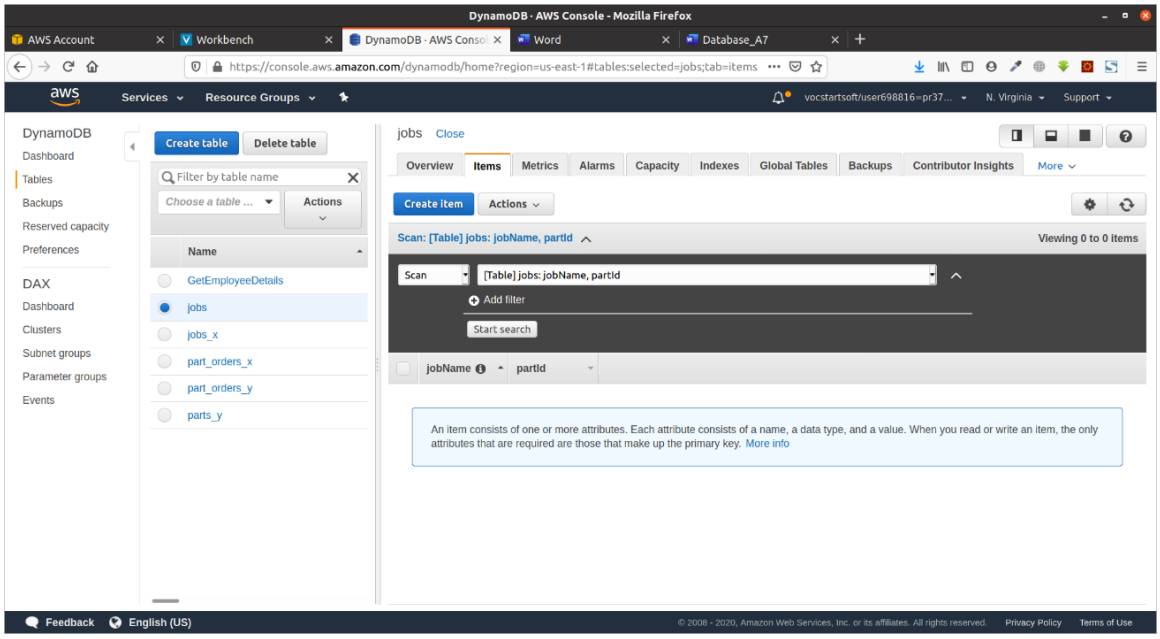


Figure 6: Items tab - DynamoDB table

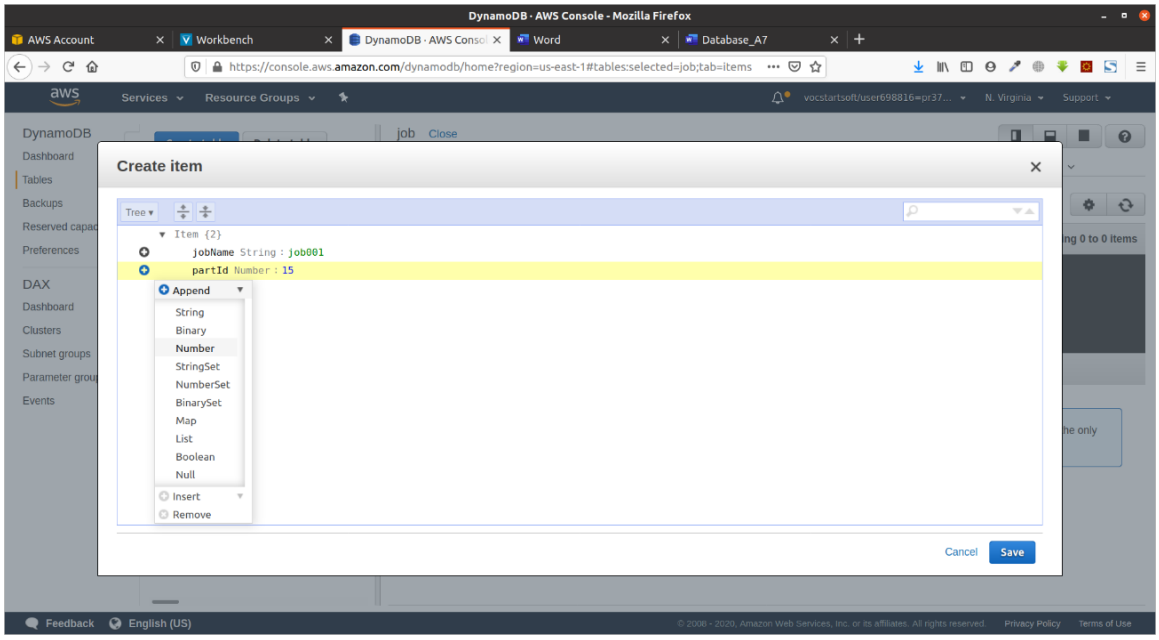


Figure 7: Create Item - DynamoDB table

Table 1 depicts the list of tables created on AWS DynamoDB for companies X and Y.

| Company | Tables | Primary Key | Sort Key |
|---|---|---|---|
| Company X | jobs_x | jobName | partId |
| | part_orders_x | partOrdersId | |
| Company Y | parts_y | partId | |
| | part_orders_y | partOrdersId | |

Table 1: AWS DynamoDB - Tables for Companies X and Y

Figures 8, 9, 10, and 11 illustrate screenshots of the tables created with the initial data. As all the tables are created using steps mentioned above, no script is used to create table and insert data.

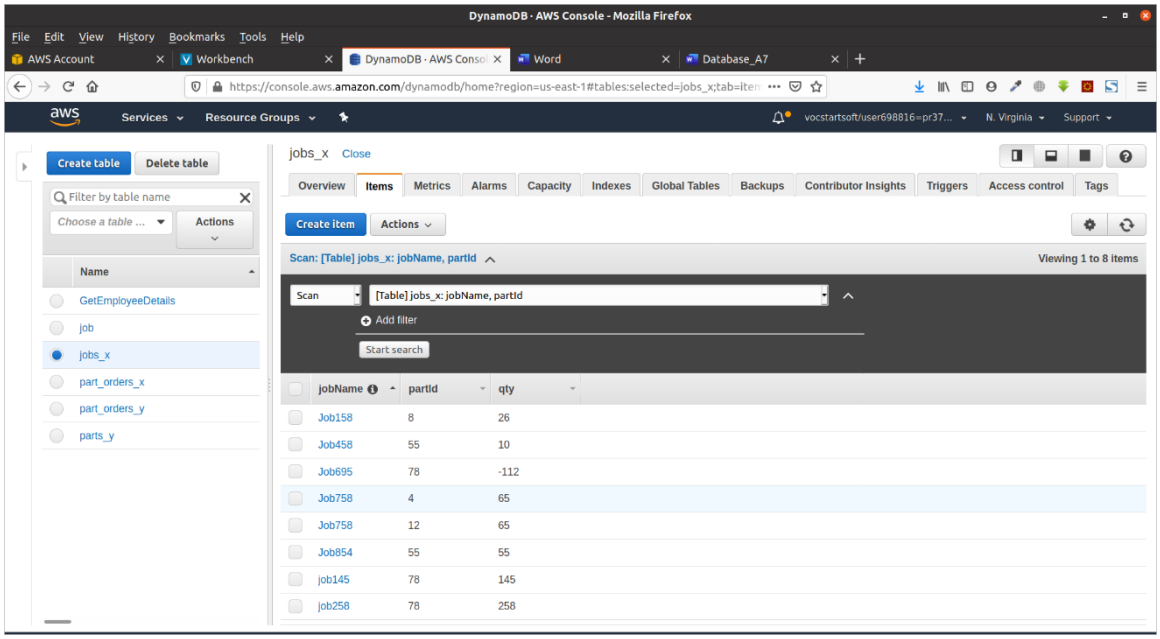


Figure 8: Company X - jobs_x table

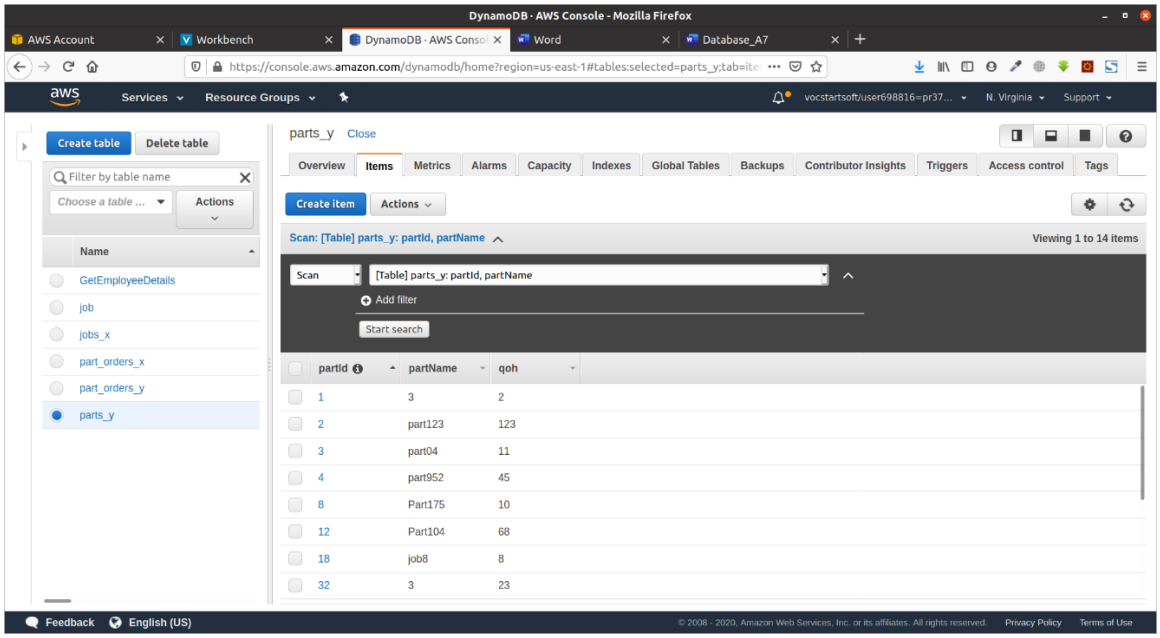


Figure 9: Company Y - parts_y table

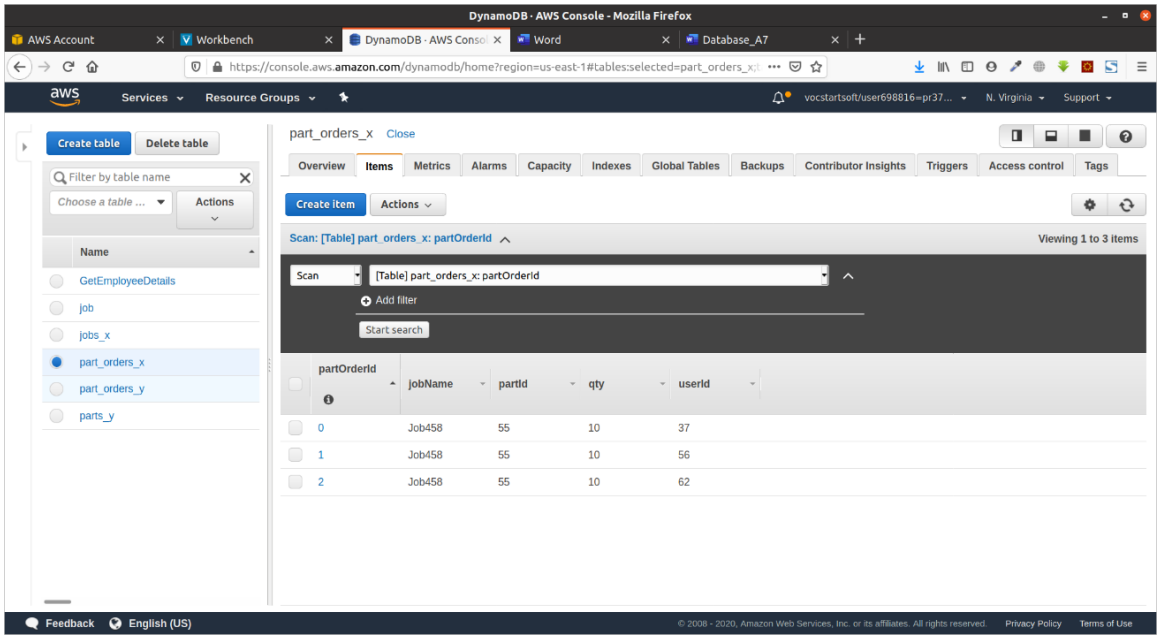


Figure 10: Company X - part_orders_x table

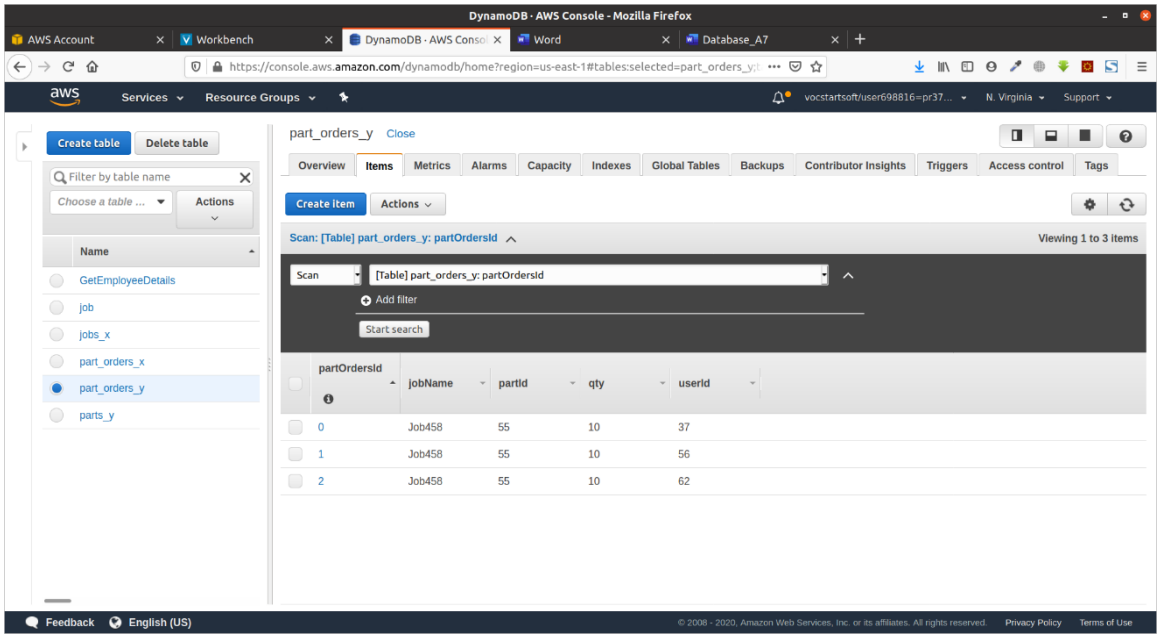


Figure 11: Company Y - part_orders_y table

## 3.2 Amazon RDS: Company Z Tables

After moving Company X and Company Y to serverless computing, tables of only Company Z are left in Amazon RDS database. Figure 12 shows tables for Company Z.

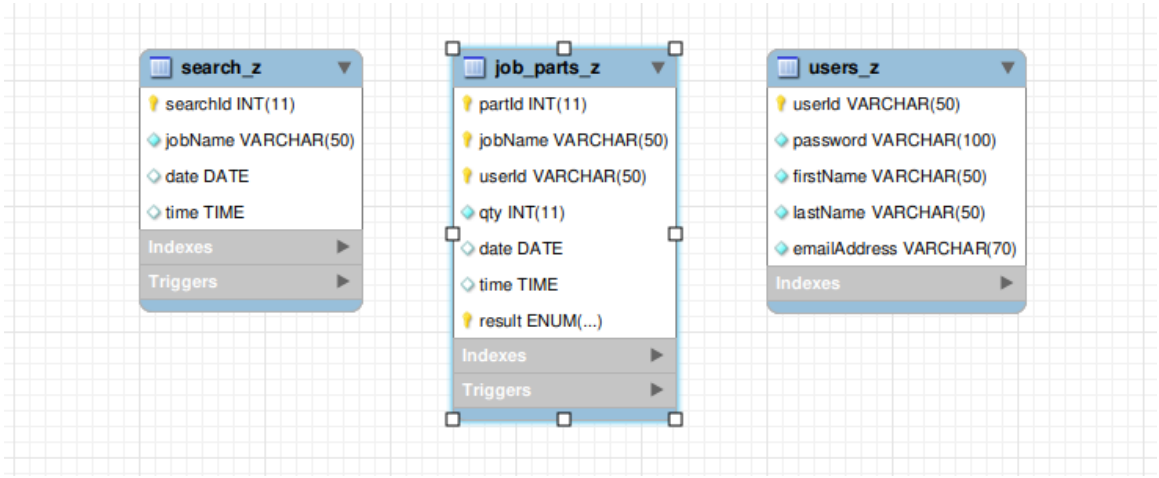


Figure 12: Company Z - Database tables

Please refer Appendix A to see the scripts for creating tables of company z. Also, the scripts files are available in Git Repository of Company Z at /sql_scripts. [9]

[9] https://git.cs.dal.ca/tprajapati/company-z-a7/-/tree/master/sql_scripts

# 4 Front-end: All Companies

The section provides a detailed description about front-end for all three companies (company X, company Y, and company Z). User Interface of all three companies is similar to the interface used in assignment 6. As the functional requirements for assignment 7 was similar to the assignment 6, our group has reused the code base. We have used AWS Elastic Beanstalk [7] for deploying the containerized front-end as a managed web application. Below sections provides a detailed explanation about areas modified to accommodate back-end change, containerization using docker, and deploying the Docker image on AWS.

## 4.1 Front-end Changes

The functional requirement for this assignment was same as previous assignment, although change in back-end led us to change parts of the front-end. As we are using Dynamo DB for back-end, many validation functionalities were managed at the front-end side. There was some modification carried out in order to serve the functionality and provide efficient server throughput.

For maintaining the primary key constraint for *partId* column in data table, we have validated the data before passing it to the server. At server side, there was no such functionality of sending a proper response to front-end, thus we have managed it by checking the available data with user input. Moreover, the front-end for all three companies are accessing serverless APIs created for companies X and Y.

## 4.2 Containerization and deployment process for front-end

This section gives a detailed overview of creating an AWS Beanstalk environment, Docker image, and hosting image on Docker hub. [6] Similar steps were followed for all the three companies and a generalized description is provided in the following sections.

### 4.2.1 Hosting Docker Image

This subsection covers a detailed workflow about uploading the Docker image on Docker hub repository. Deploying the image on the Docker hub is done with the help ngx-deploy-docker package. [15] This is a npm package used by Angular CLI for deploying the application. Repository on Docker hub is automatically created and image is pushed in it using CLI. For creating the Docker image, system uses Dockerfile present in the project. Docker file content can be found in Appendix B. Also the repository name will be taken from angular.json file.

For deploying the image on docker hub, please execute the below command:

```
ng deploy
```

The below list of screenshots showcases the successful creation and deployment of images on docker hub for company Y. All the steps would be same for remaining two companies.

Figures 13 and 14 shows the output of creating a docker image for the front-end of company Y and pushing the generated image to the docker hub.

```
dracula@Midgard:/mnt/Data/Google Drive/Dalhousie/Summer 20/Cloud Computing 5409/Assingnments/Assignment-7/company-y-a7$ sudo ng deploy
 Building "company-y-ui-a7". Configuration: "production".
Generating ES5 bundles for differential loading...
ES5 bundle generation complete.

chunk {0} runtime-es2015.0dae8cbc97194c7caed4.js (runtime) 1.45 kB [entry] [rendered]
chunk {0} runtime-es5.0dae8cbc97194c7caed4.js (runtime) 1.45 kB [entry] [rendered]
chunk {2} polyfills-es2015.497d09372e98279300ff.js (polyfills) 36.8 kB [initial] [rendered]
chunk {3} polyfills-es5.525842d25cafd1f6f850.js (polyfills-es5) 130 kB [initial] [rendered]
chunk {1} main-es2015.217ef8b21024ae7ae149.js (main) 522 kB [initial] [rendered]
chunk {1} main-es5.217ef8b21024ae7ae149.js (main) 626 kB [initial] [rendered]
chunk {4} styles.e0f45488eb6be5a7126d.css (styles) 281 kB [initial] [rendered]
Date: 2020-07-27T17:27:14.749Z - Hash: 3a02e4c37fceddd5fce8 - Time: 48996ms


 Executing Docker Build...
Sending build context to Docker daemon  5.621MB

Step 1/6 : FROM nginx:alpine
 ---> 7d0cdcc60a96
Step 2/6 : LABEL version="1.0.0"
 ---> Using cache
 ---> 328120aee63b
Step 3/6 : COPY nginx.conf /etc/nginx/nginx.conf
 ---> Using cache
 ---> 83bccace75f7
Step 4/6 : WORKDIR /usr/share/nginx/html

 ---> Using cache
 ---> 5983ba2555b1
Step 5/6 : COPY dist/company-y-ui-a7 .
```

Figure 13: Docker Image build - 1

```
 ---> Running in 4f854e4cdfe9

Removing intermediate container 4f854e4cdfe9
 ---> 0344e650db12

Successfully built 0344e650db12

Successfully tagged malavjani/company-y-ui-a7:latest

✔ Docker Build was successfully
🚀 Publishing image to registry
The push refers to repository [docker.io/malavjani/company-y-ui-a7]

ec6194987f5f: Preparing
b36b8f1e308c: Preparing
a181cbf898a0: Preparing
570fc47f2558: Preparing
5d17421f1571: Preparing
7bb2a9d37337: Preparing
3e207b409db3: Preparing
7bb2a9d37337: Waiting
3e207b409db3: Waiting

570fc47f2558: Mounted from malavjani/company-y-ui

a181cbf898a0: Mounted from malavjani/company-y-ui

5d17421f1571: Mounted from malavjani/company-y-ui
```

Figure 14: Docker Image build - 2

Figure 15 shows the pushed image on the docker hub account.

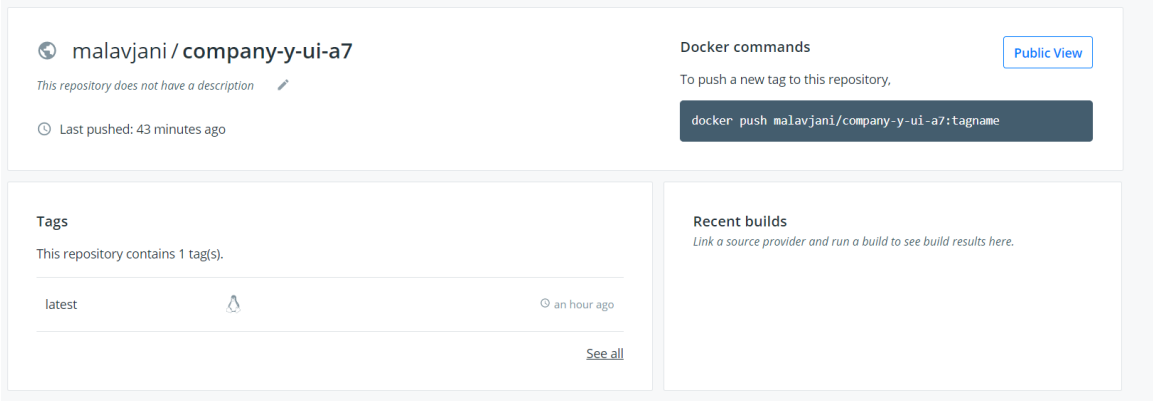


Figure 15: Docker Image on Docker Hub

### 4.2.2 Creating environment on Elastic Beanstalk

Step 1: Initially, after login into the AWS console, a list of all services will be displayed from which Elastic Beanstalk should be selected in order to create web service as shown in figure 16.

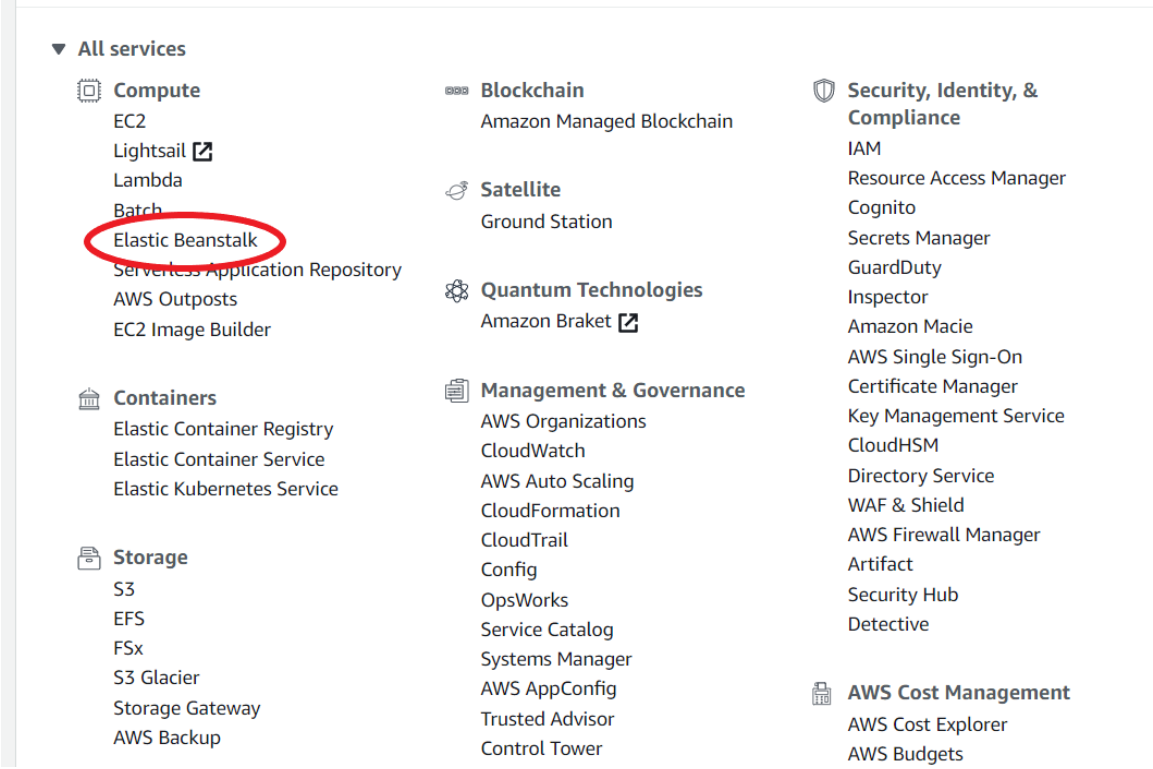


Figure 16: AWS Console - Elastic Beanstalk

Step 2: After getting into the dashboard, click on Create a new application button to create a new application as shown in figure 17.

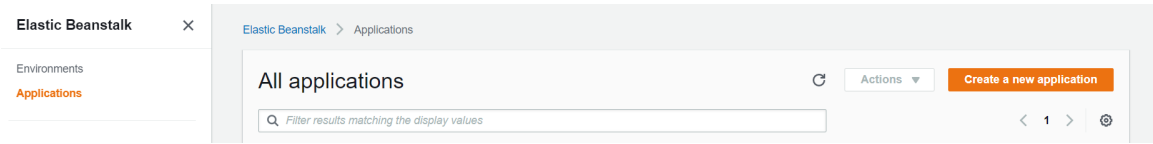


Figure 17: Create new Elastic Beanstalk application

Step 3: A small form will be prompted, which will ask for some information, and after filling the form click on Create button as shown in figure 18.

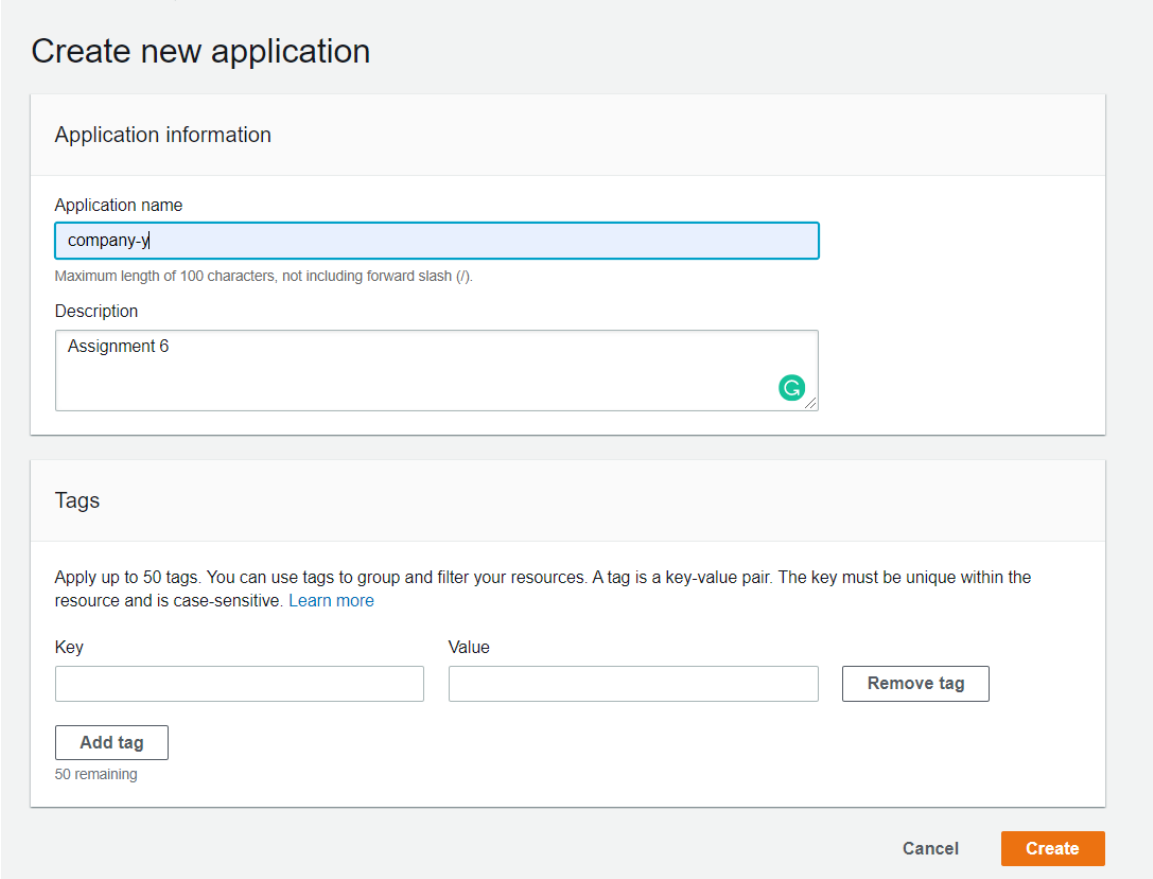


Figure 18: Create new Elastic Beanstalk application Form

Step 4: After the creation of the application, we need to create a new environment and which can be performed using create new environment button.

Step 5: Form present in figure 19 will be displayed which will ask for information related to the environment you wish to create.

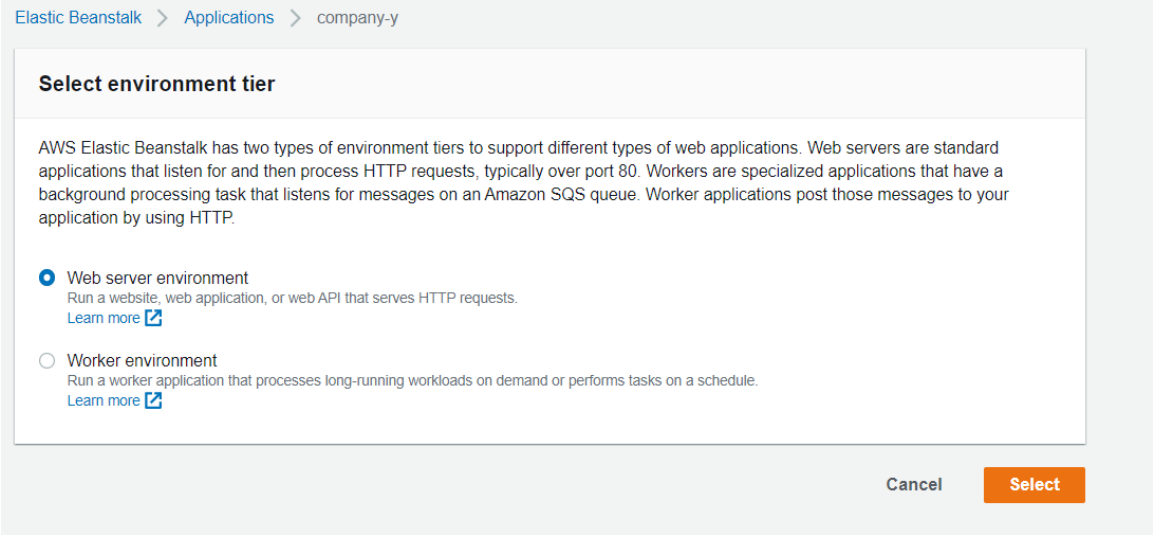


Figure 19: Select Environment Tier

Step 6: The second step will include filling name and some description related to the environment as shown in figure 20.

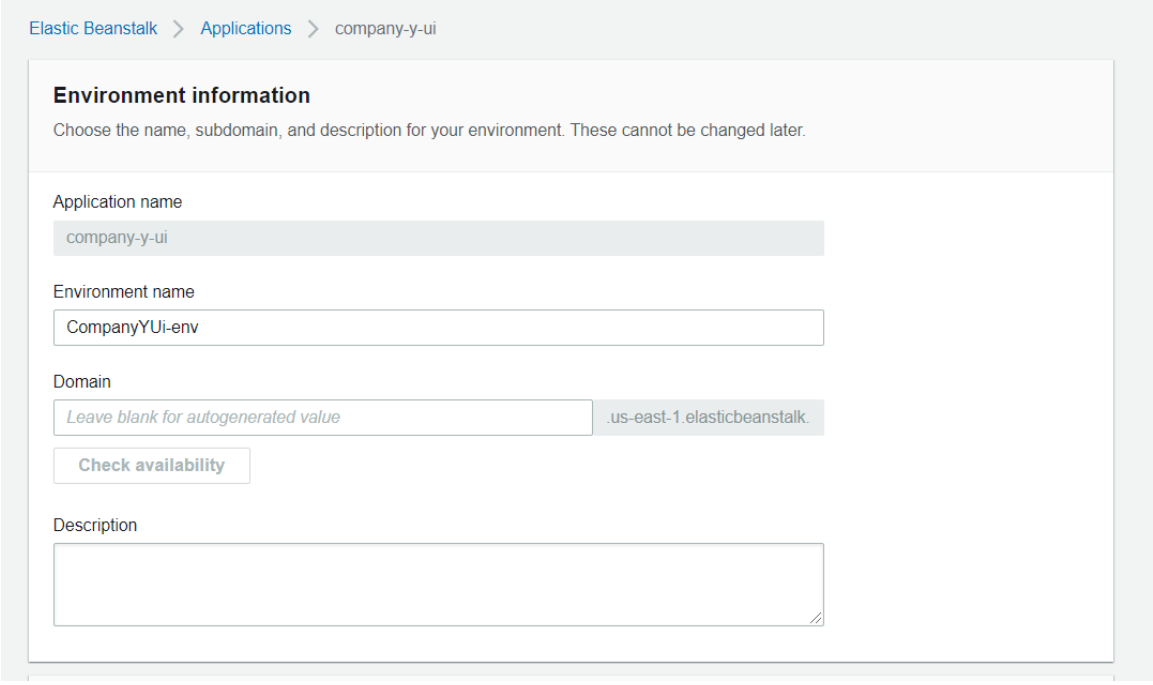


Figure 20: Environment Information

Step 7: As in this project, Docker is being used, and we need to select Docker into the platform list as shown in figure 21.

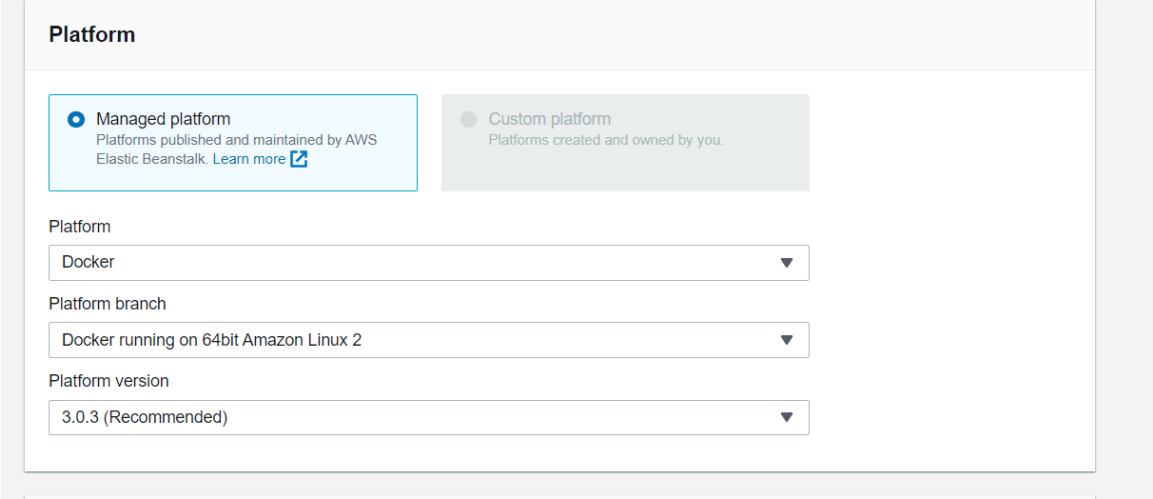


Figure 21: Platform

Step 8: After clicking on create the environment, the process of creating a new environment will start.

Step 9: After the successful creation of the environment still, the service will not be accessible as we need to give the path to the repository to EB as shown in figure 22.

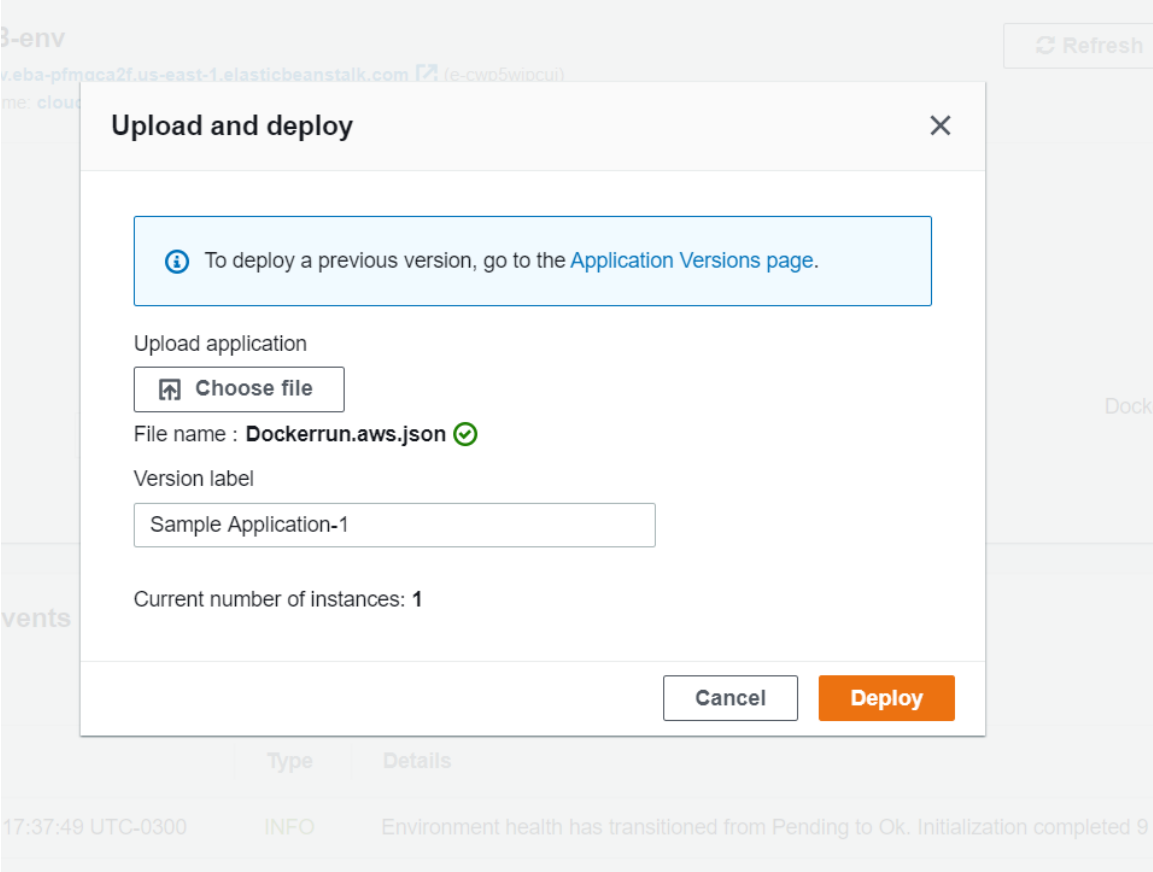


Figure 22: Upload and Deploy

Step 10: After uploading the JSON file which points to the Docker repository it will successfully create the EB service and deploy the image successfully. JSON file can be seen under Appendix B section. Figures 23 demonstrate all the three companies' successful deployment.

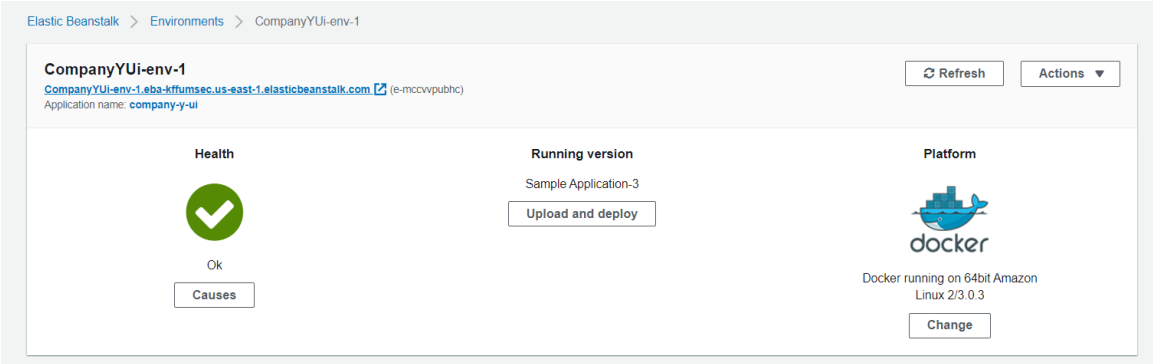


Figure 23: Company Y - EB Environment

# 5 Backend Implementation

This section gives an overview about the development of the serverless application for company X and Y. It includes the task related to create an Amazon API Gateway, Lambda function and connect Lambda function with DynamoDB Database.

## 5.1 Serverless computing for company X and Y

We have implemented our backend for company X and Y using the serverless concepts. The concept of serverless computing is very suitable for the cloud-based application where you do not need to think of maintaining server. This application is implemented on the AWS services such as AWS API Gateway, Lambda function, and DynamoDB NoSQL Database. The further details are outlined in below sections.

We have created serverless application for the Nodejs. There are mainly three components to create a serverless application using AWS services.

1. Setup a new function on AWS Lambda to perform task.
2. Create an AWS API Gateway.
3. Connect the Lambda function with DynamoDB database.

Detailed steps to create a serverless application are illustrated further.

Step 1: Create a function from AWS Lambda [10] console as shown in figure 24

[10] `https://console.aws.amazon.com/lambda/`

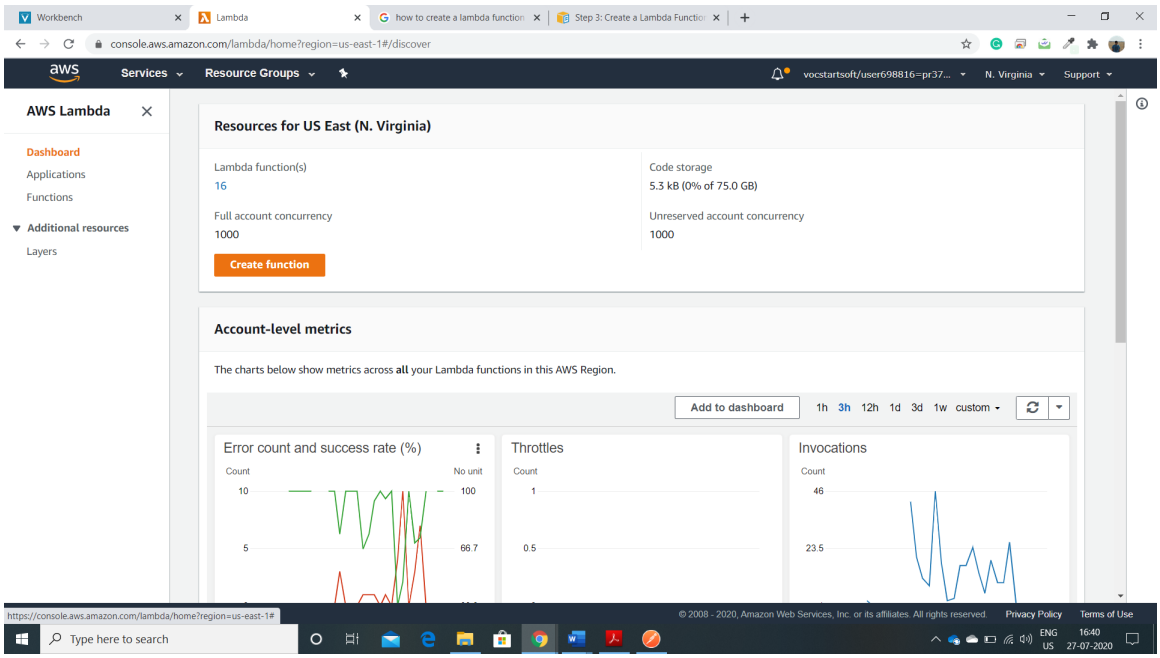


Figure 24: AWS Lambda Console

Step 2: On create function page, click on the Author from the scratch, enter the name of function and choose execution role. Here we are naming function as “getjobs” and choosing existing role. Click on the create function button as shown in figure 25.

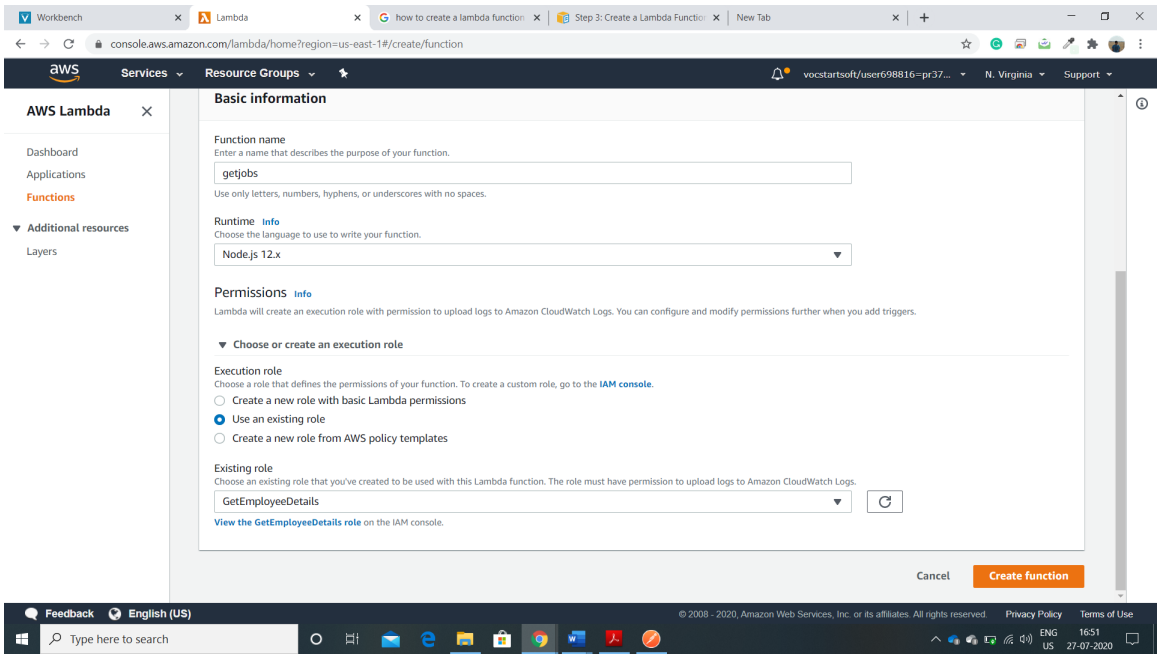


Figure 25: AWS Lambda Function Basic Information

Step 3: Getjobs function is created successfully as shown in figure 26.

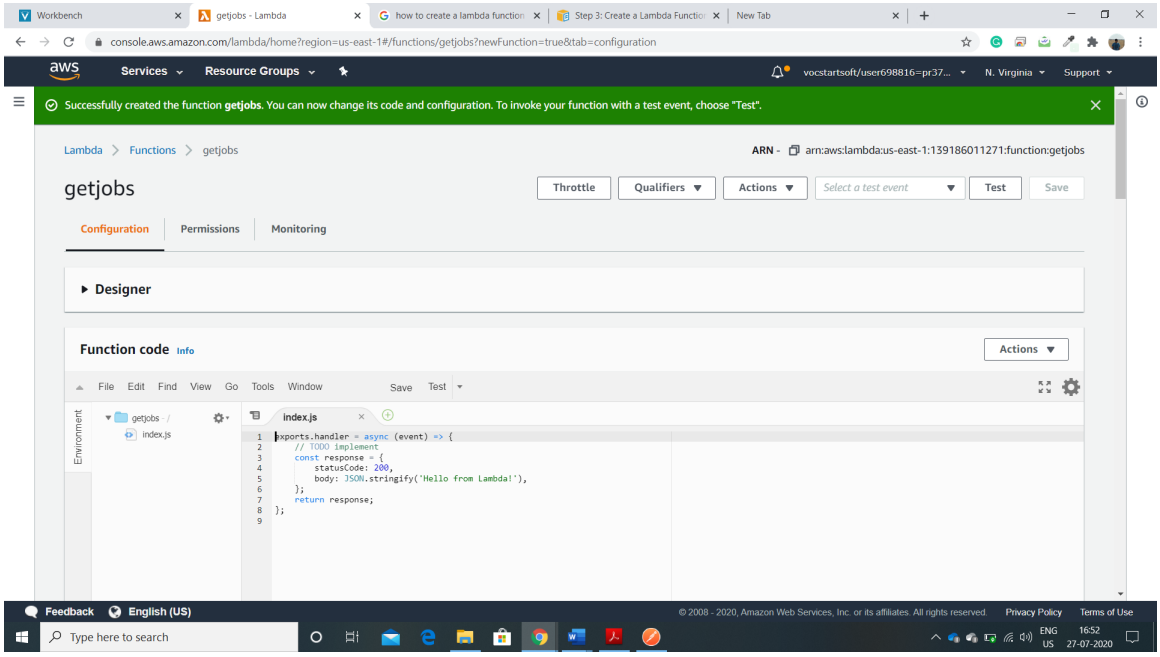


Figure 26: getjobs Lambda function

Step 4: Now we can Test our function. Click on the Test button and enter the name of the test event. You will get the result that lambda function is working as

shown in figure 27

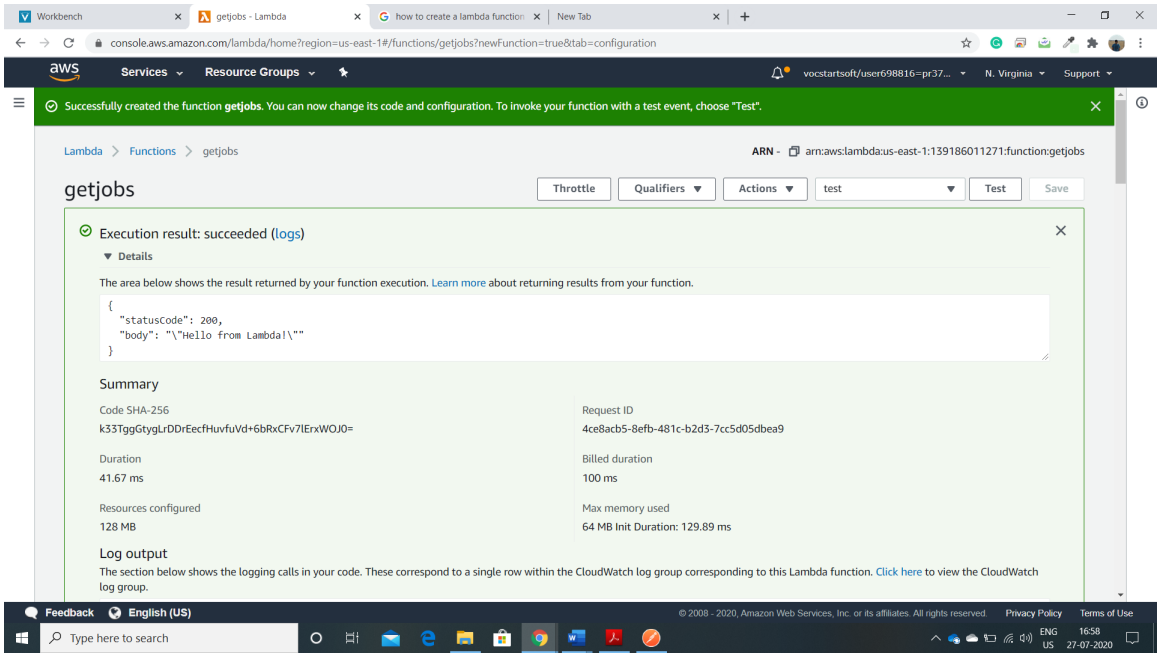


Figure 27: Testing getjobs lambda function

Step 5: You can write you own code into the function code. Here we have written code to get the data of jobs from the jobs_x table. It is shown in the figure 28. Here we have used scan function of the DynamoDB DocumentClient, which is able to fetch all the items available in the table named "jobs_x". you can see in the figure 28 that name of the table is stored in the variable. The schema of the table is shown in the figure 29.

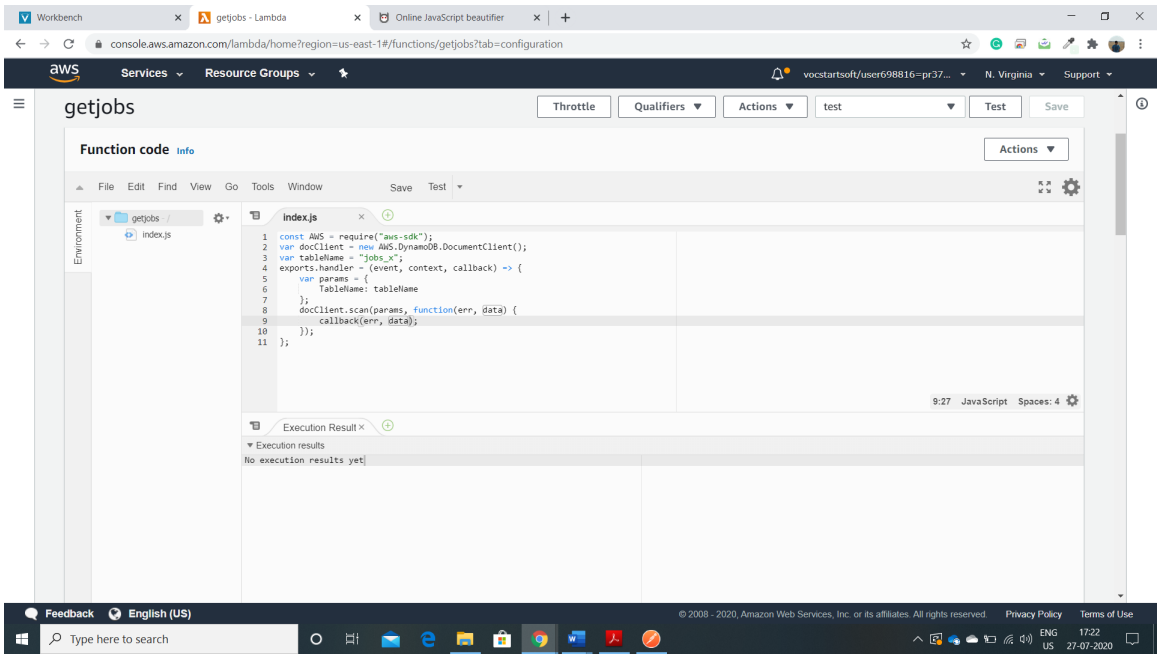


Figure 28: getjobs function code - retrieve from jobs_x

Step 6: Schema of table named "jobs_x" is shown in the figure 29. The detailed description of database is already illustrated in database part. Till now, we have successfully created AWS Lambda function and DynamoDB. Now we will create AWS API Gateway.

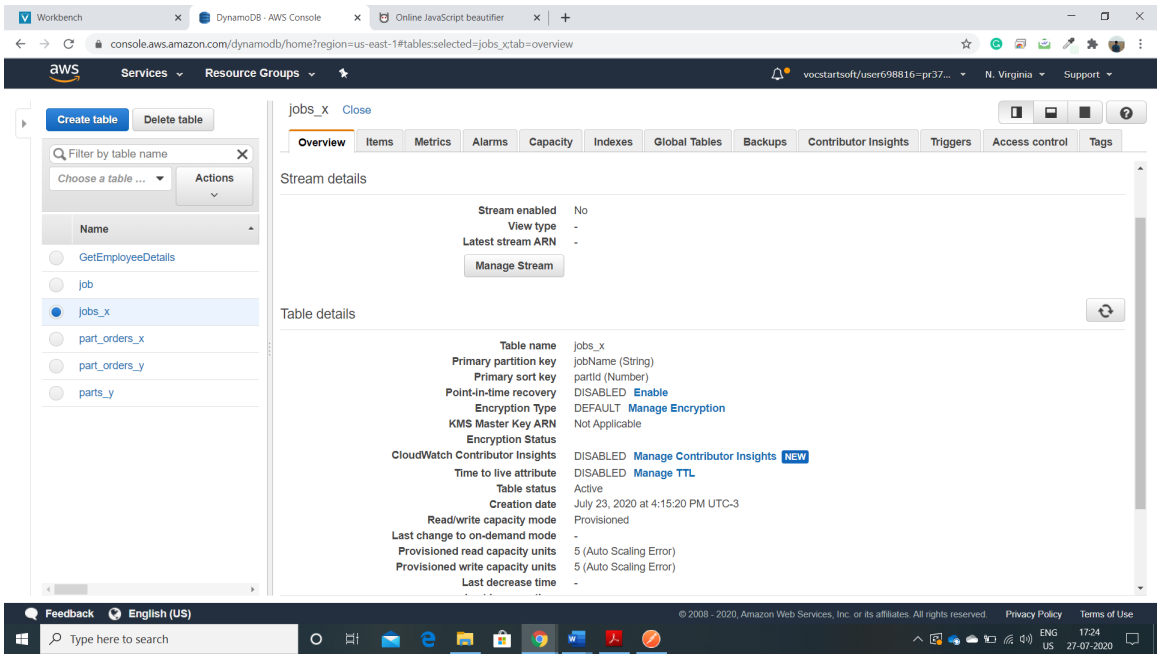


Figure 29: Schema - jobs_x table

Step 7: Go to AWS API Gateway under the Application Services in AWS Console as shown in the figure 30

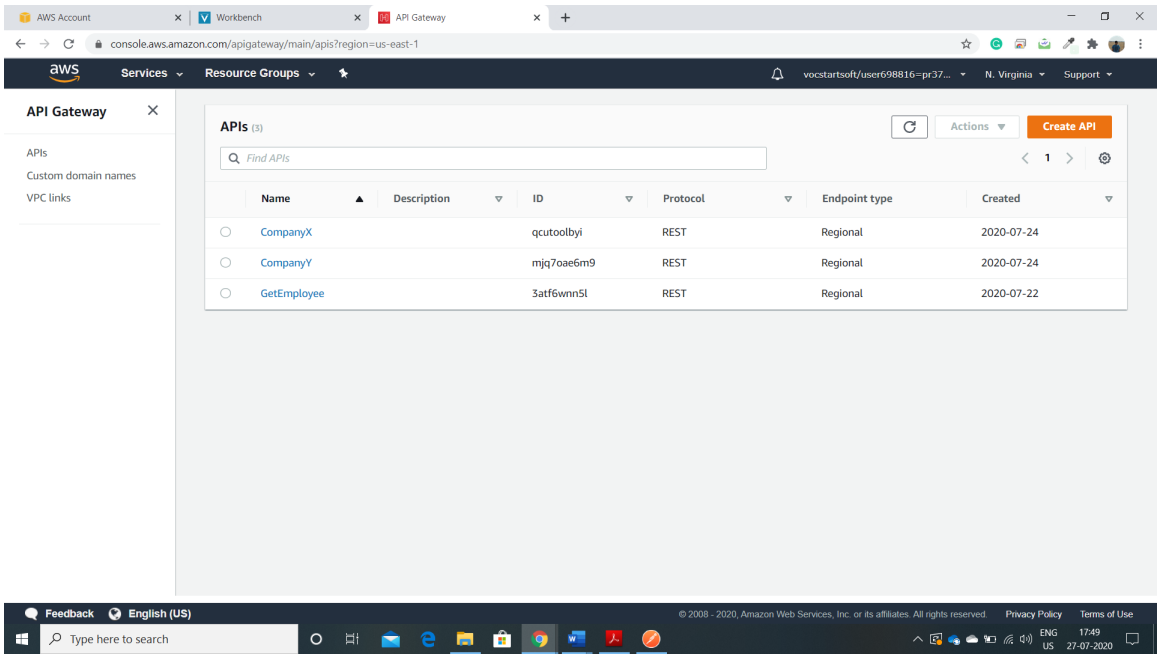


Figure 30: API Gateway Console

Step 8: Click on the create API button and select REST API build option as shown in

the figure 31

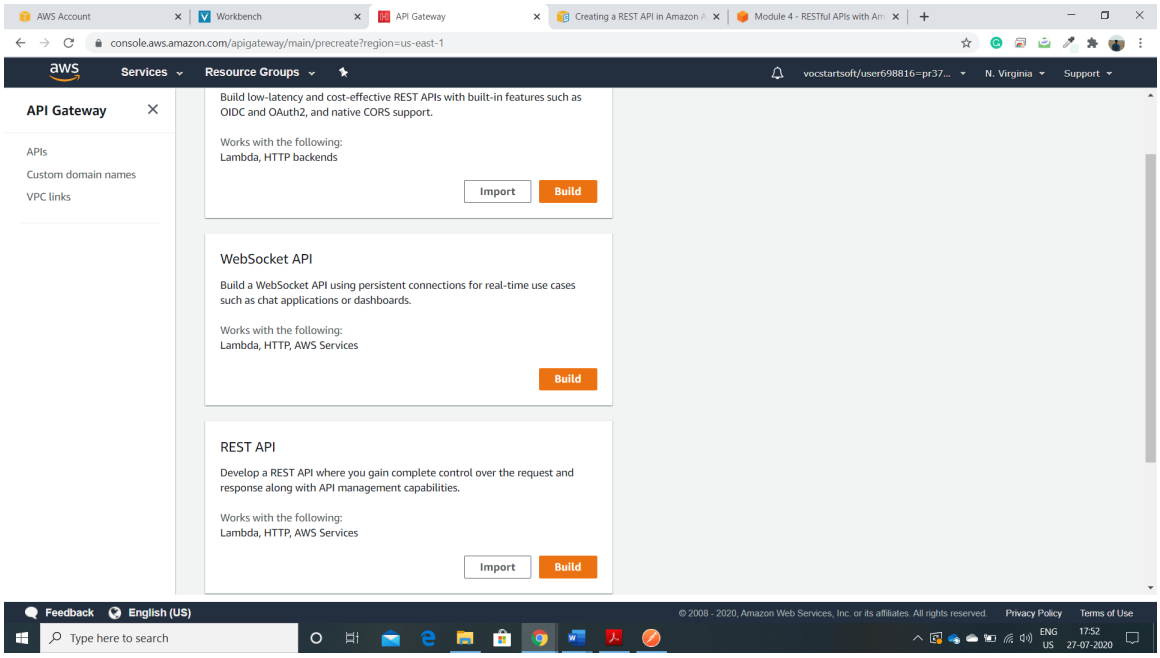


Figure 31: Create API Gateway

Step 9: Choose the protocol REST, select the option to create a new API and enter the name of API as shown in the figure 32

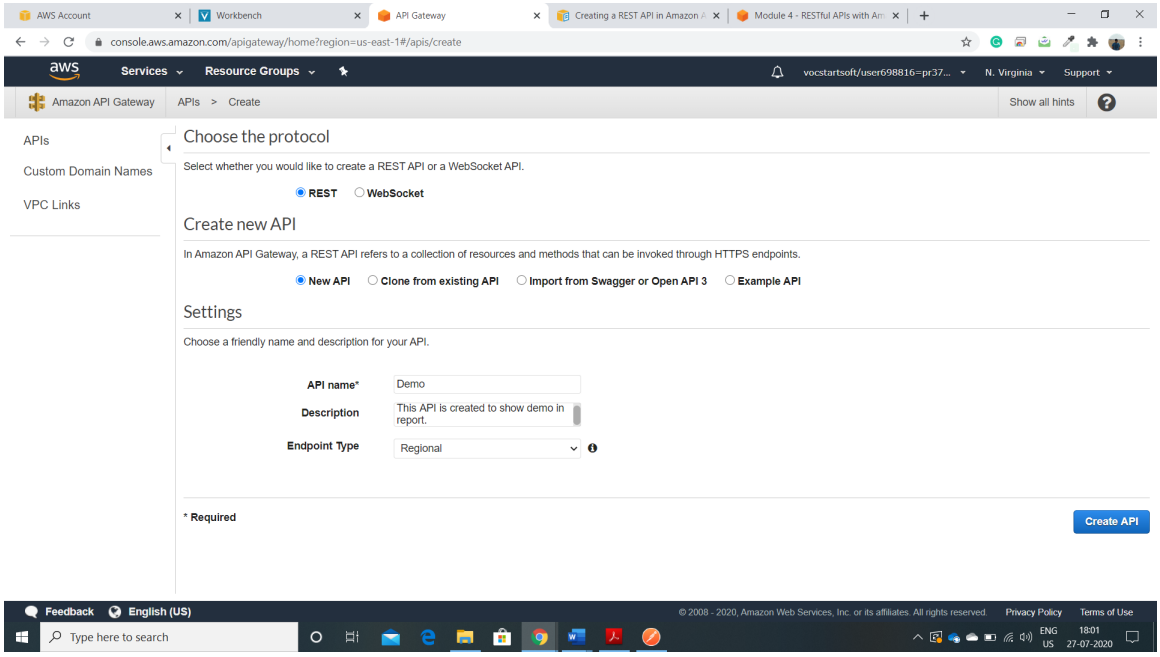


Figure 32: Protocol Selection

Step 10: After clicking on the create API button the dashboard will be displayed as shown in figure 33 Which means the API Gateway is created.

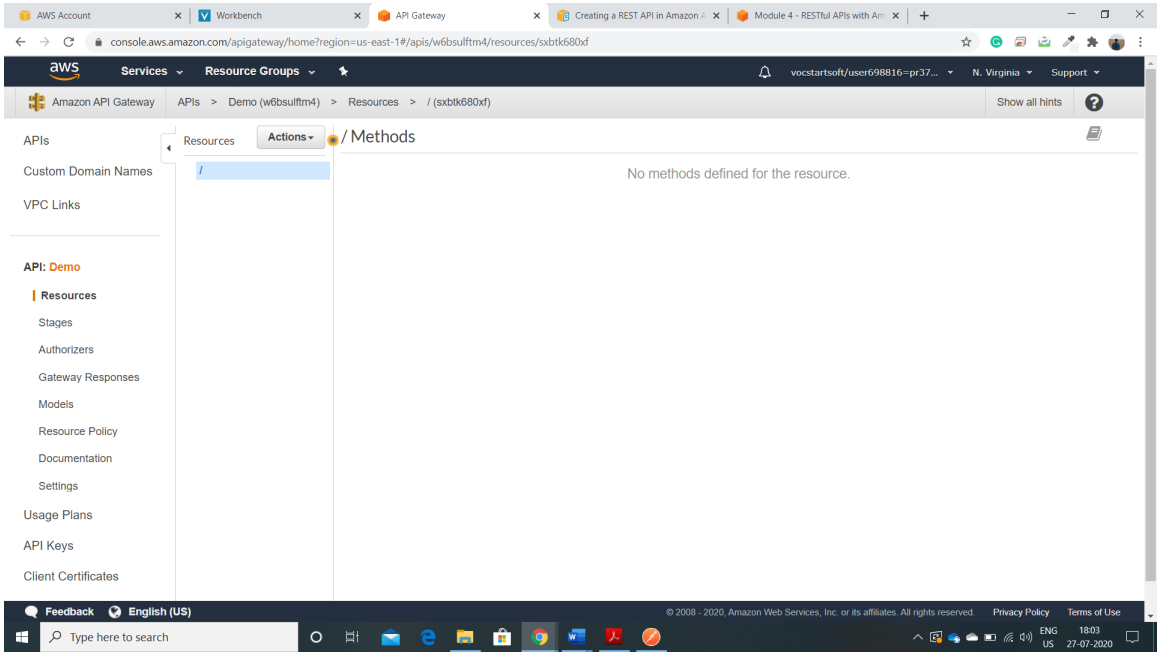


Figure 33: Created API Gateway Dashboard

Step 11: Go to action and click on the create resource button as shown in figure 34.

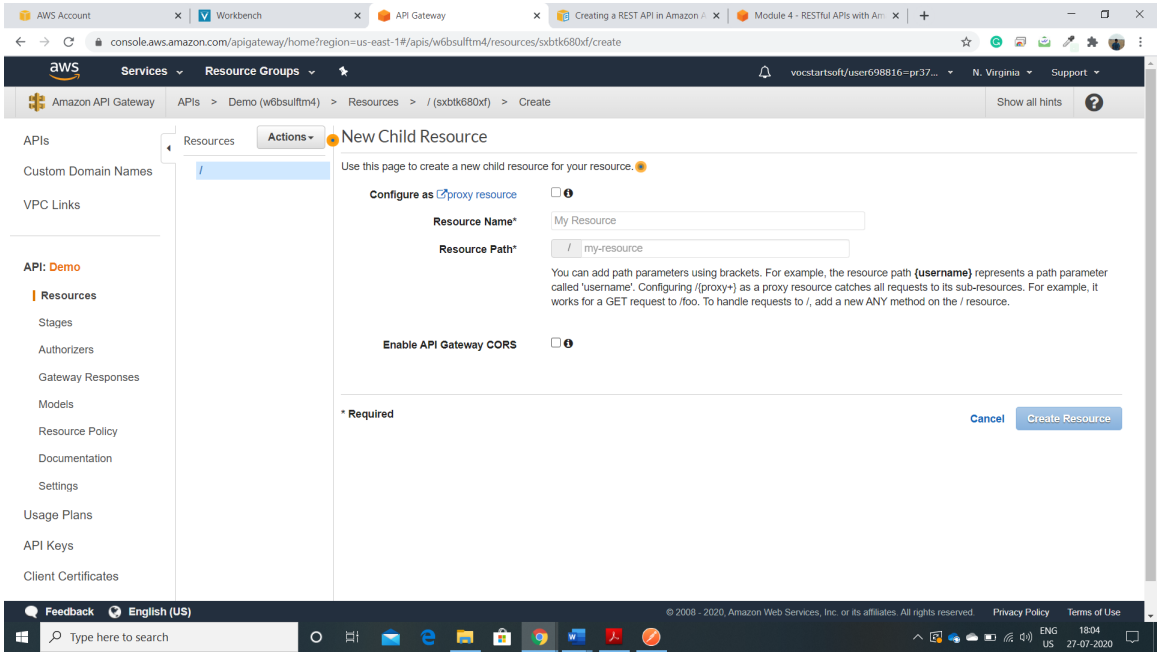


Figure 34: New Child Resource

Step 12: We have entered Resource Name as “job” and click on the create resource button. We have not added any method yet so it does not show any method as shown in figure 35.

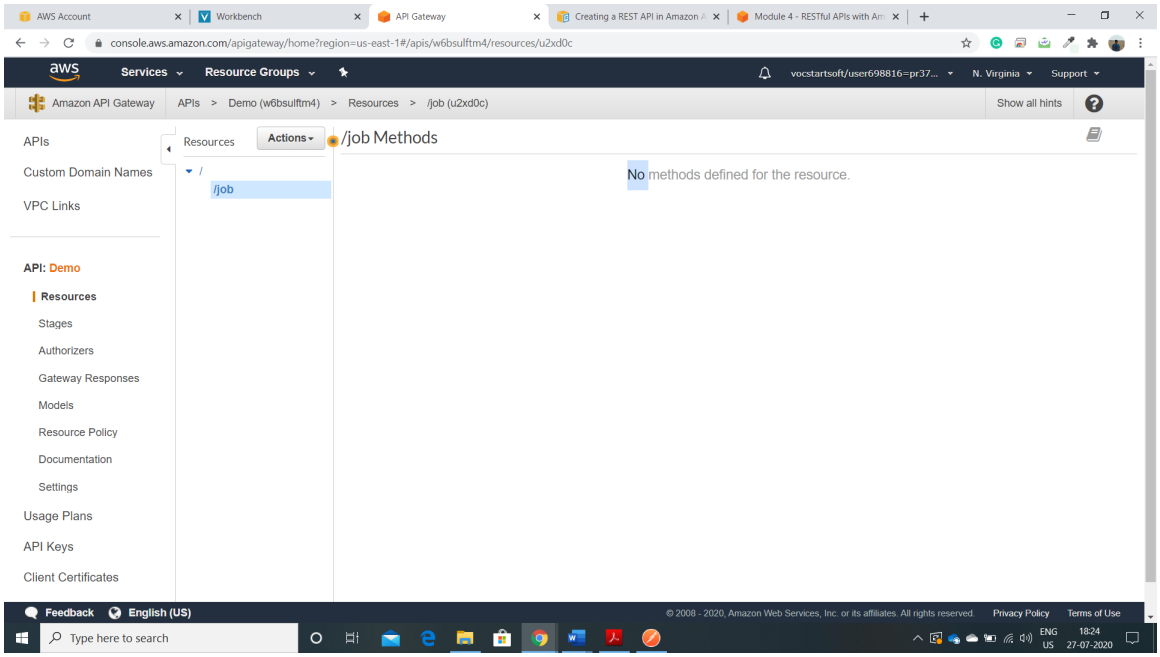


Figure 35: API Gateway - Job Resource

Step 13: Go to actions and create a new method. There are 8 options available. Choose GET method because we have created lambda function to get all jobs as shown in the figure 36.

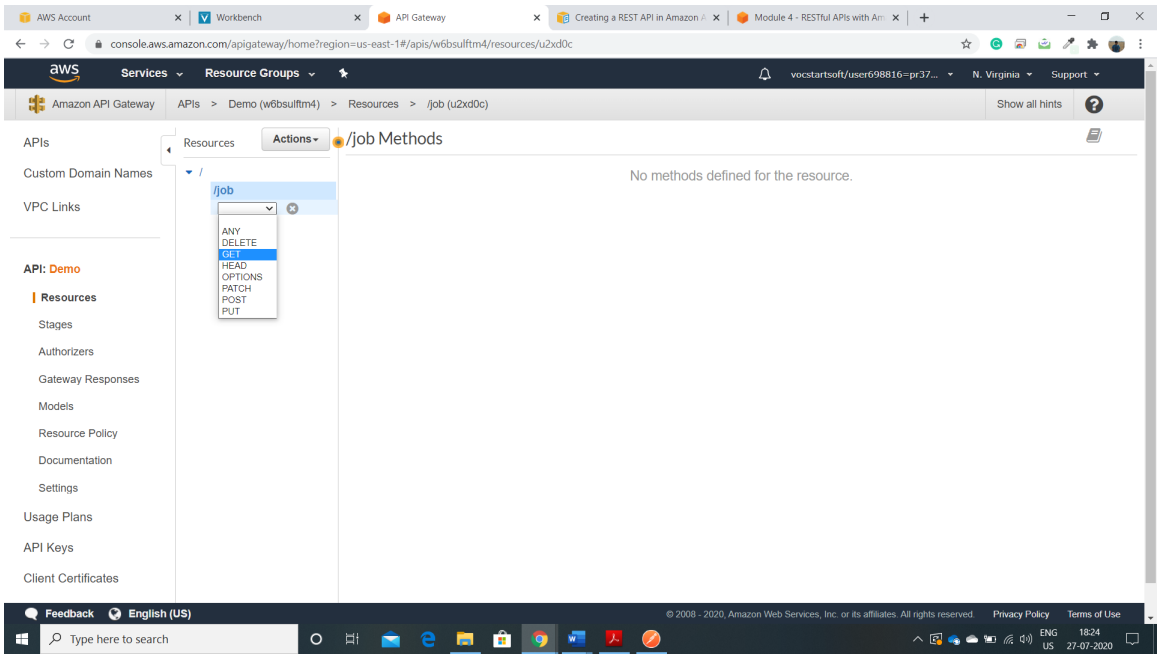


Figure 36: Resource options

Step 14: Now we need to enter the name of lambda function which you need to call, and click on the save function as shown in the figure 37.

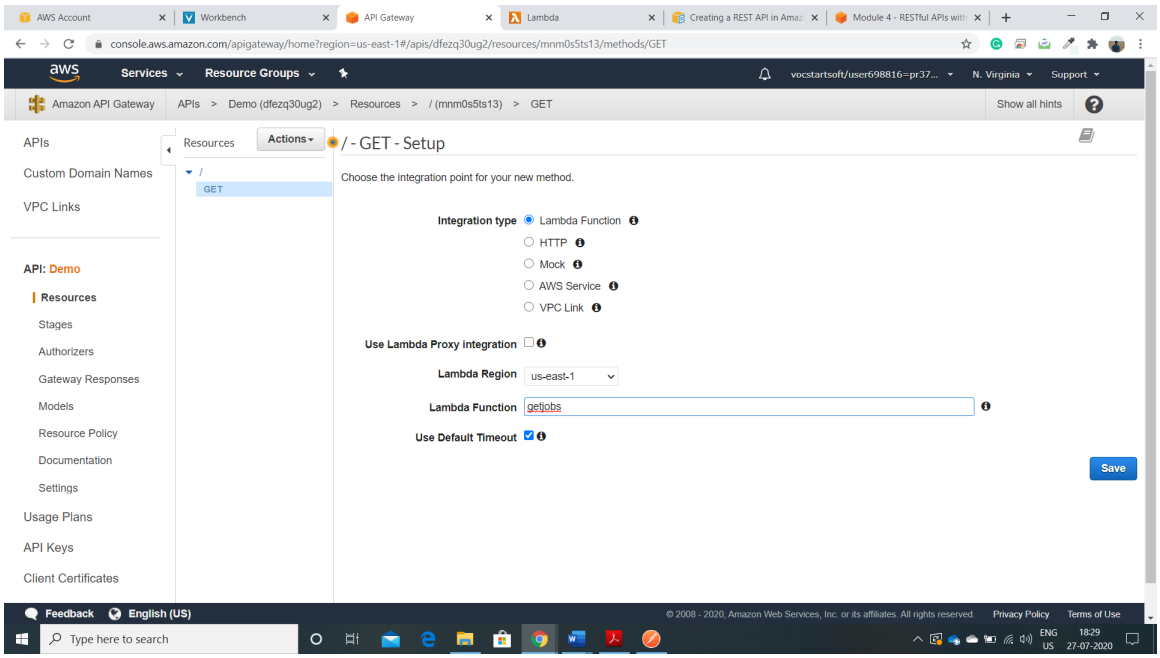


Figure 37: API Gateway - GET Setup

Step 15: The API Gateway is now attached with the lambda function named “getjobs”.

It is shown in the figure 38.

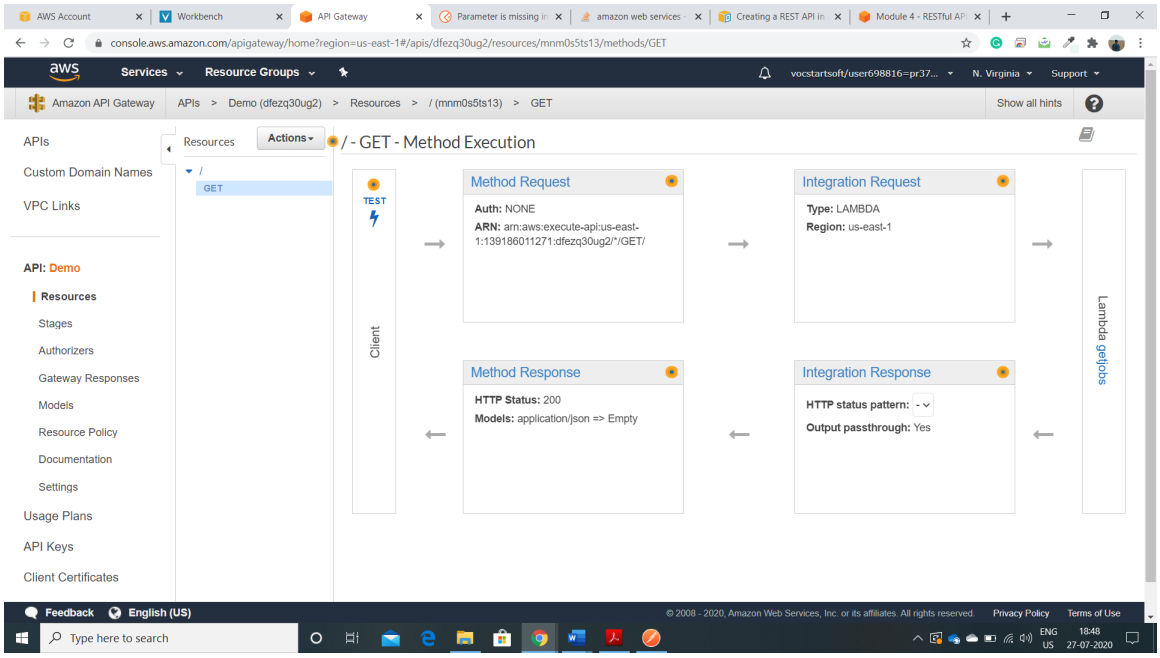


Figure 38: GET - Method Execution

Step 16: When we click on the Test button on the Client bar, the request will be generated and response will be displayed to user as shown in the figure 39.

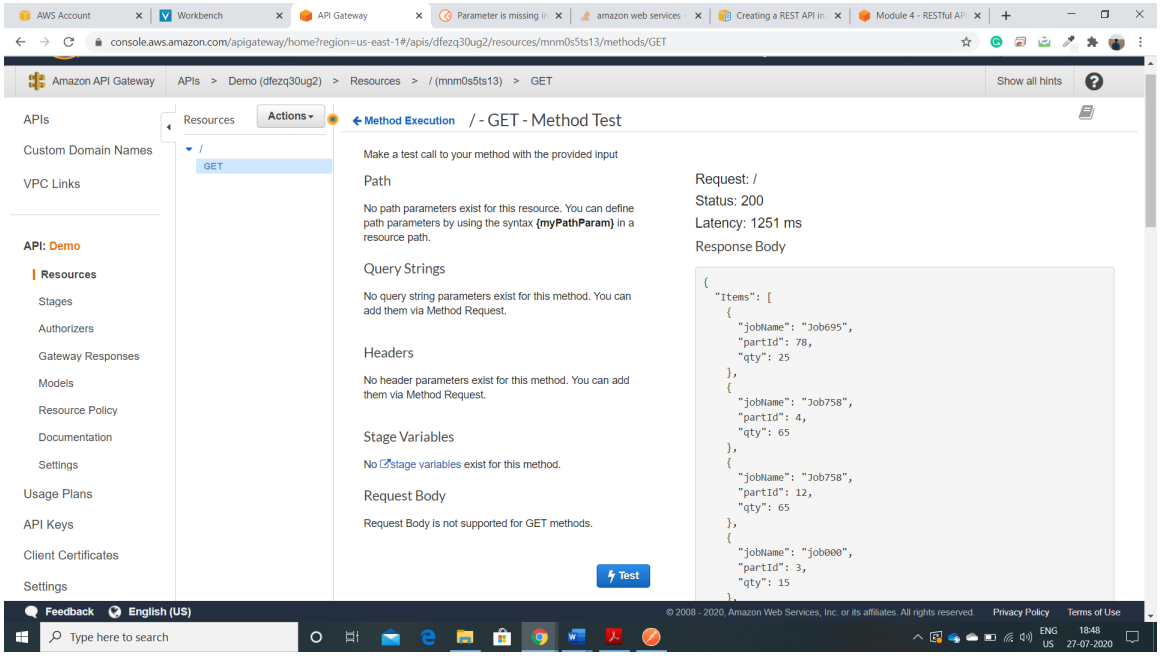


Figure 39: /GET - method test

You can see in the figure * that all the items from the table is fetched and returned in the response body.

Overall, through these all steps we have created one API Gateway named 'GET: /job' which will call the lambda function named "getjobs". The lambda function is implemented in a way that it connect with table in DynamoDB database and scan all the items and return to the user. Now we can deploy this API Gateway and with that URL we can test it through the POSTMAN as well.

Step 17: Go to Action and click on the deploy API. Enter the required details as shown in the figure 40

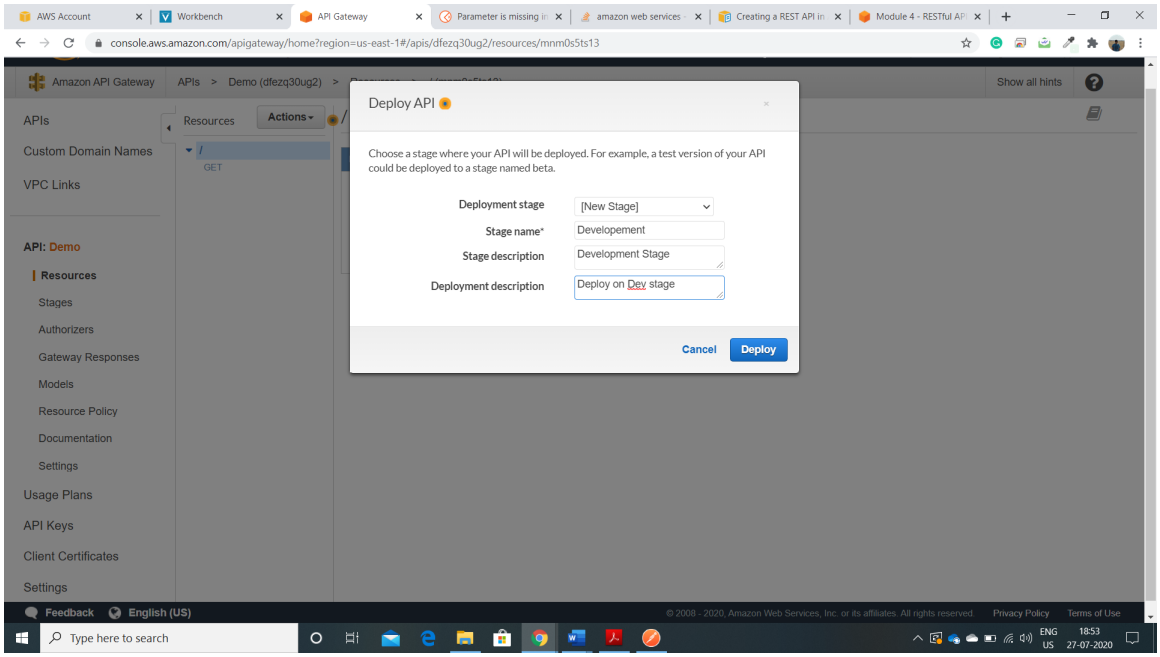


Figure 40: Deploy API

Step 18: Click on the deploy button and the API is deployed. The Invoke URL is generated, and you can access the endpoint as shown in figure 41.

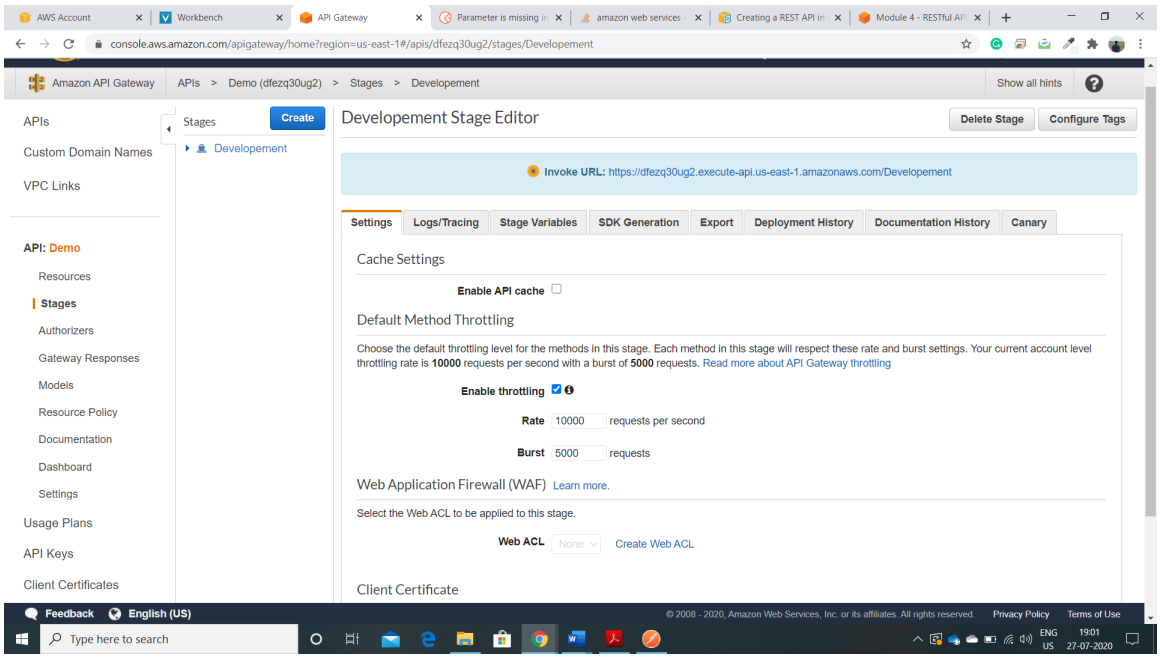


Figure 41: Development Stage Editor

Step 19: Checking the generated URL in the POSTMAN as shown in the figure 42.

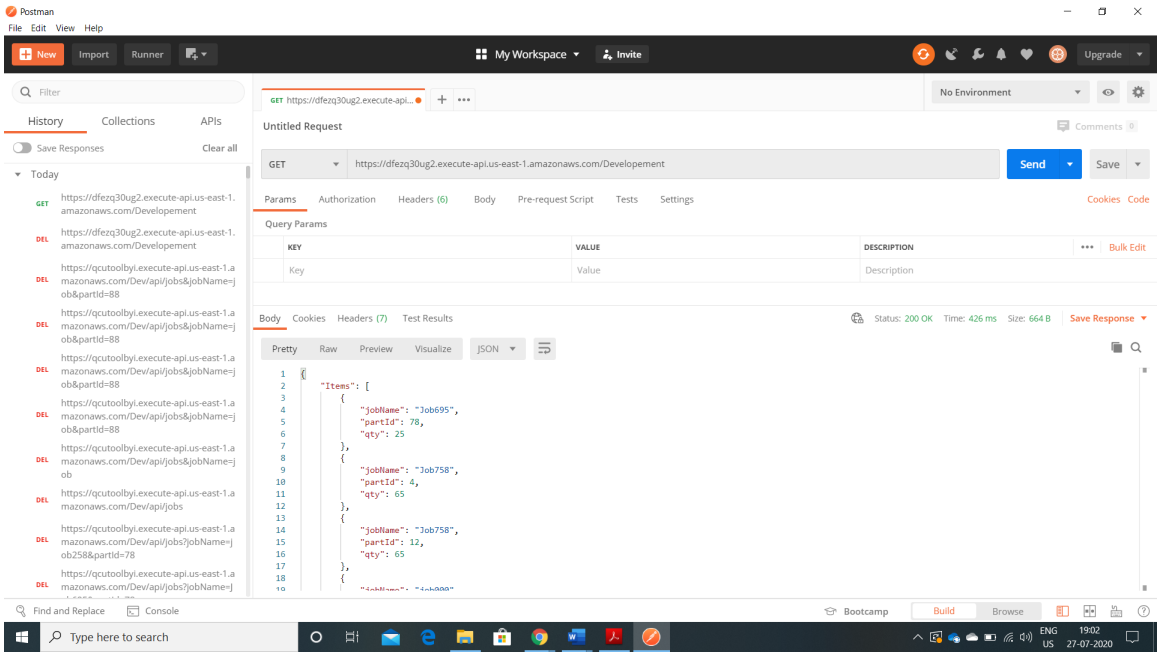


Figure 42: Testing URL in POSTMAN

### 5.1.1 Serverless backend for Company-X

This way we have created all the AWS API Gateway for the company X. The AWS API Gateway for the company X are as follows:

```
1.  /api
    1.1.  /ajob
        1.1.1. GET
        1.1.2. OPTIONS
    1.2.  /getallorders
        1.2.1. GET
        1.2.2. OPTIONS
    1.3.  /jobs
        1.3.1. DELETE
        1.3.2. GET
        1.3.3. OPTIONS
        1.3.4. POST
        1.3.5. PUT
    1.4.  /partorders
        1.4.1. GET
        1.4.2. OPTIONS
        1.4.3. POST
    1.5.  /searchjobs
        1.5.1. GET
        1.5.2. OPTIONS
```

All the API Gateways of the company-X are shown in the figure 43.

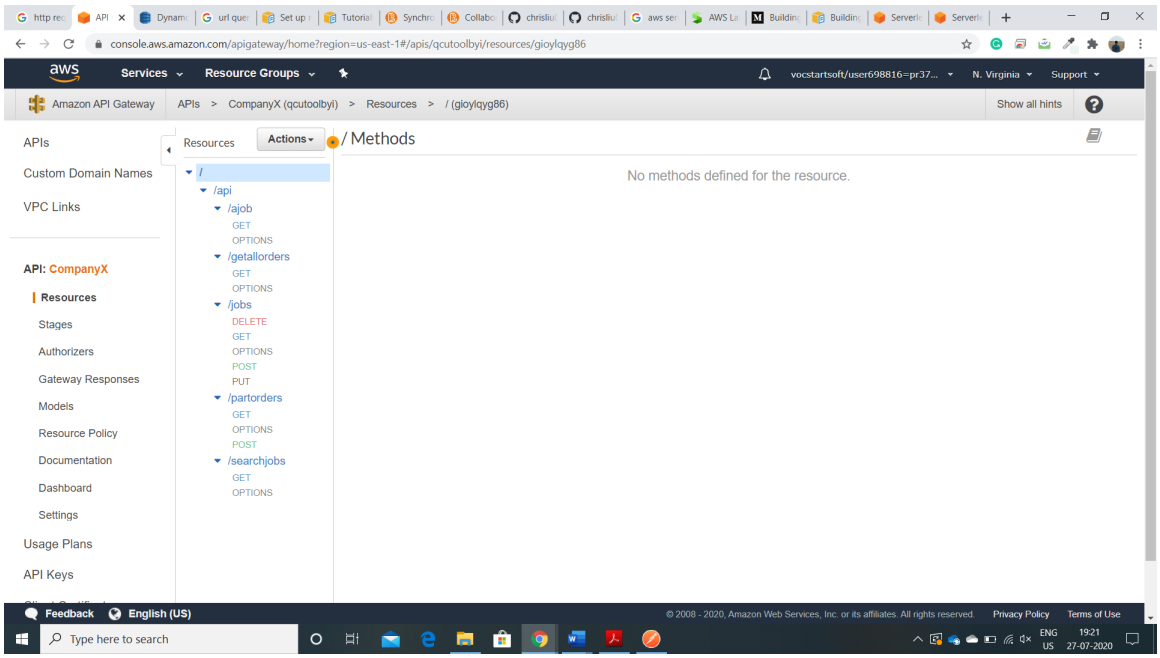


Figure 43: API Gateway - Company X

These API Gateways are attached with the lambda functions. The list of lambda functions are shown in the figure 44.

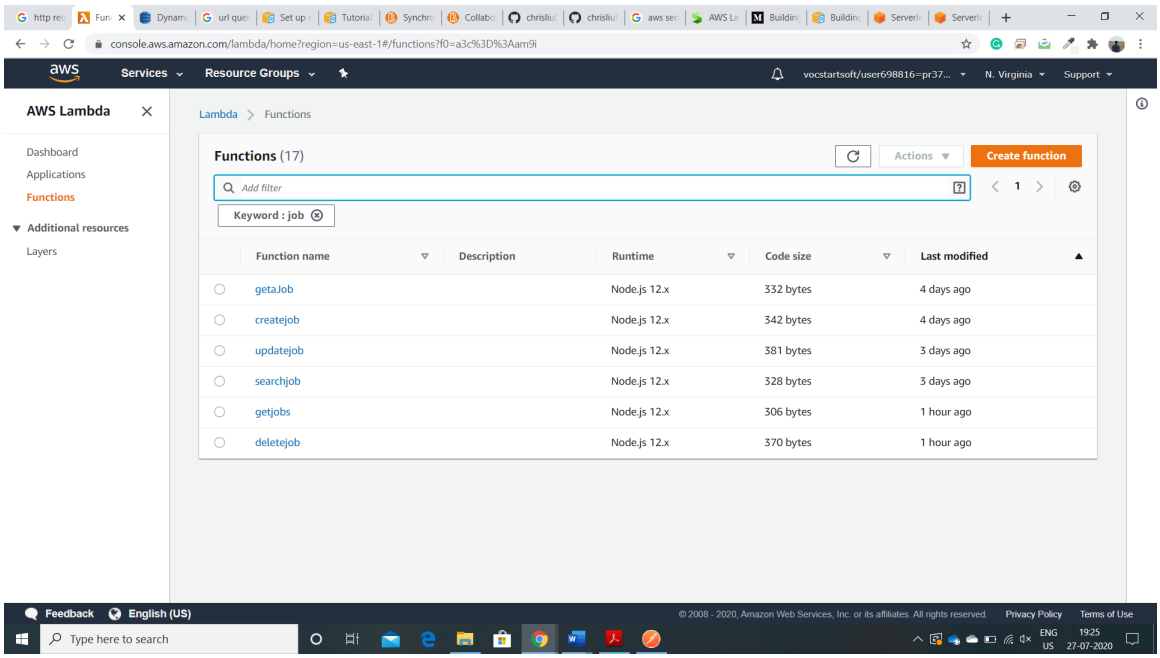


Figure 44: Lambda Functions - Company X

The API Gateway is deployed on the Development server as shown in the figure 45.

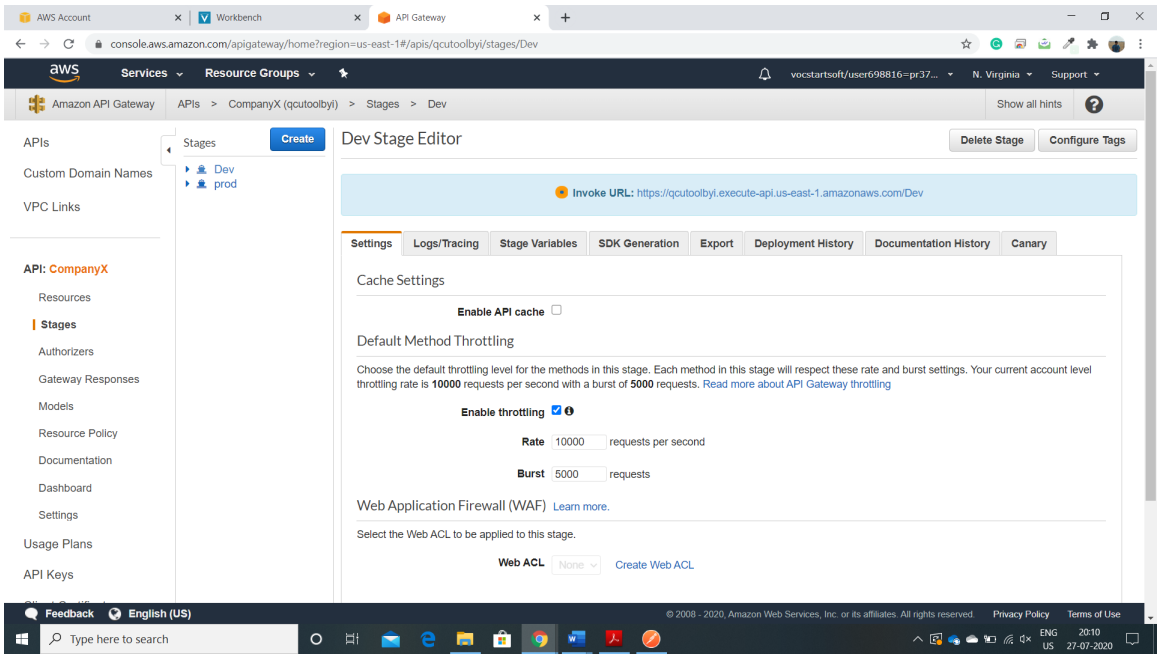


Figure 45: Deployment Stages - Company X

### 5.1.2 Serverless backend for Company-Y

We have created all the AWS API Gateway for the company Y. The AWS API Gateway for the company – Y are as follows:

```
1. /api
2.   /apart
       2.1. GET
       2.2. /{partId}
            2.2.1. GET
3.   /partorders
       3.1. GET
       3.2. OPTIONS
       3.3. POST
4.   /parts
       4.1. GET
       4.2. OPTIONS
       4.3. POST
       4.4. PUT
```

All the API Gateways of the company-Y are shown in the figure 46.

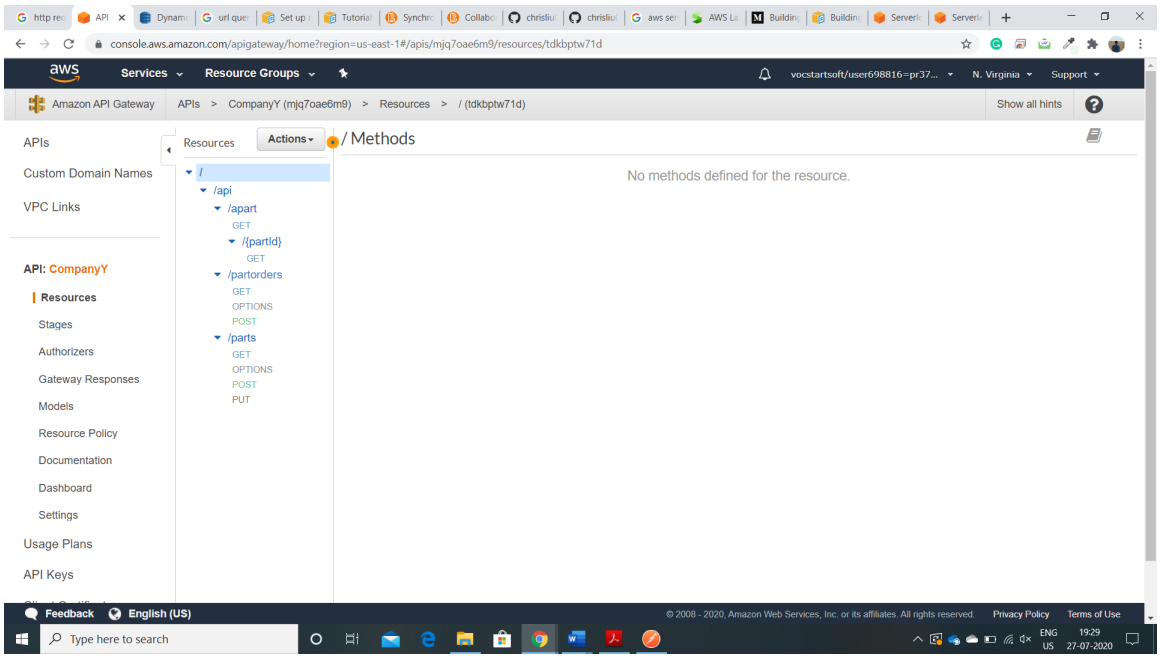


Figure 46: API Gateway - Company Y

These API Gateways are attached with the lambda functions. The list of lambda functions are shown in the figure 47.

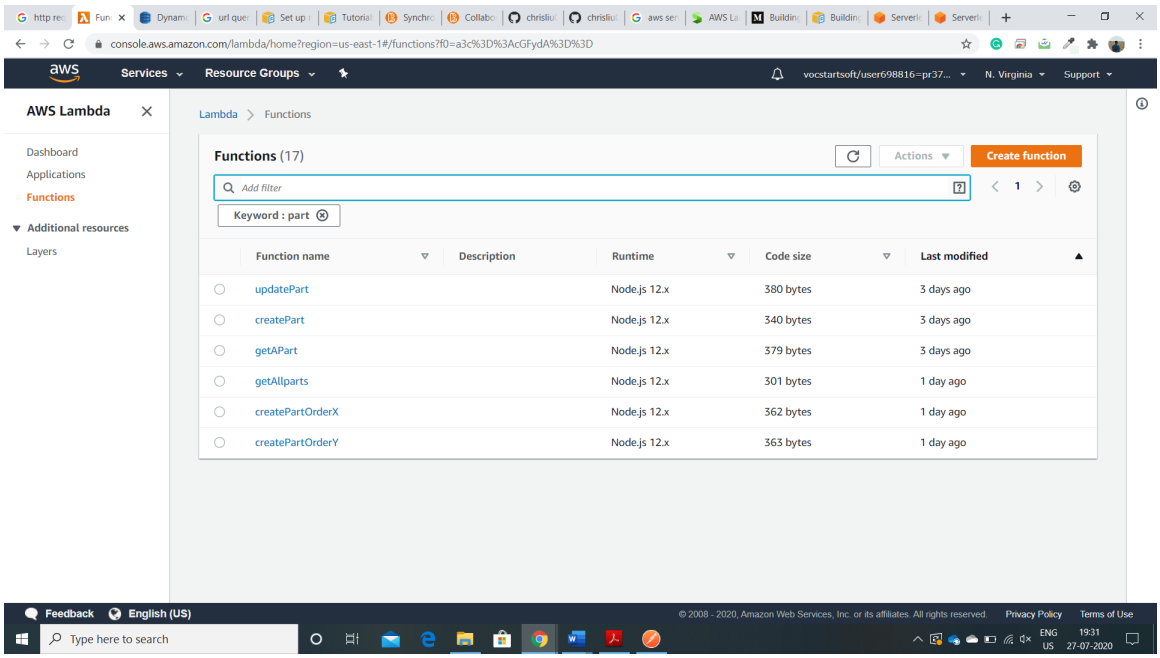


Figure 47: Lambda Functions - Company Y

The API Gateway is deployed on the Development server as shown in the figure 48.

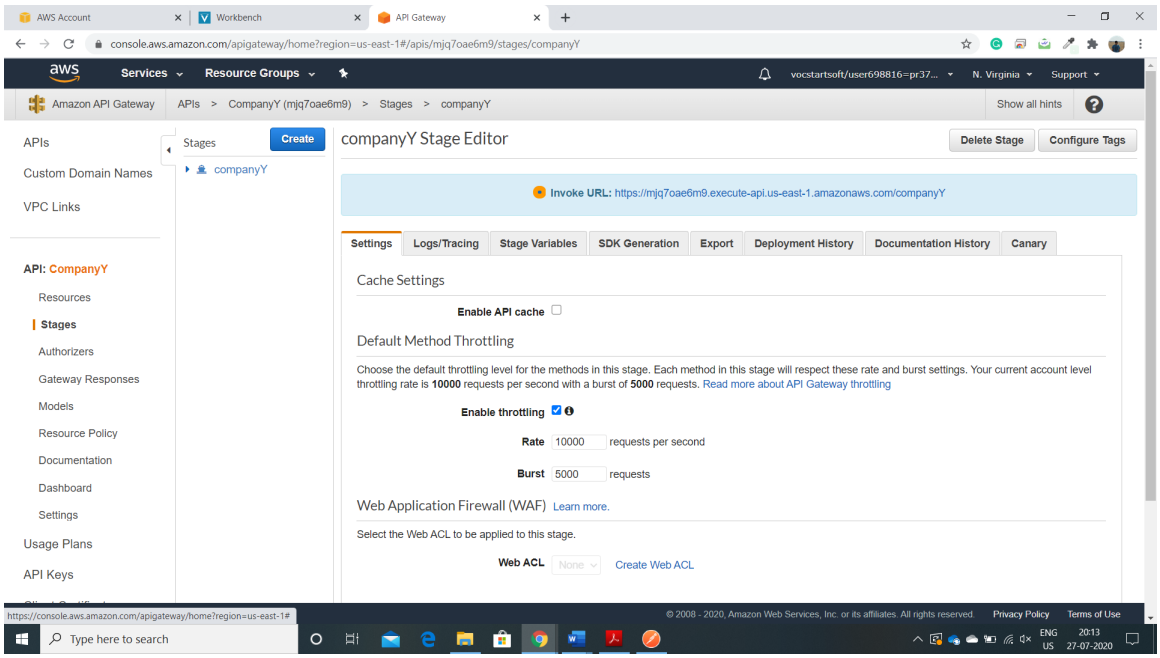


Figure 48: Deployment Stages - Company Y

## 5.2 Containerization of company Z

The section gives a brief overview about the process followed to containerize and deploy the backend of a company-Z. It provides generalized steps that is being followed by applications. We have reused the Elastic beanstalk for the company Z from the group assignment 6.

The containerization is performed using docker and the generated images are hosted on docker hub. The individual backend is deployed as a managed containerized application on Amazon Elastic Beanstalk.

### 5.2.1 Process to containerize and deploy Front-end

This section outlines the workflow to create AWS Elastic Beanstalk environment, creating Docker image, and hosting it to docker hub.

Step 1: *Hosting docker image* - This step gives a detailed explanation of workflow of deploying the docker image on Docker hub using ngx-deploy-docker npm package. This package is used for dockerizing angular application using Angular CLI. The image created using the CLI is directly pushed on the docker hub registry.

For deploying the image on docker hub, please execute the below command:

```
ng deploy
```

The command first creates a docker image using the Dockerfile present in the project. Content of the Docker File can be found in Appendix. After successful creation of the Docker image, the system will automatically create a repository on the account provided in angular.json file.

Figure 49 shows the output of creating a docker image for the frontend of company z and pushing the generated image to the docker hub.

```
dracula@Midgard:~/Desktop/Summer20/Cloud Computing 5409/Assingnments/Assignment-6/Implementation/company-z$ sudo ng deploy
📦 Building "company-z-ui". Configuration: "production".
Generating ES5 bundles for differential loading...
ES5 bundle generation complete.

chunk {2} polyfills-es2015.497d09372e98279300ff.js (polyfills) 36.8 kB [initial] [rendered]
chunk {3} polyfills-es5.525842d25cafd1f6f850.js (polyfills-es5) 130 kB [initial] [rendered]
chunk {1} main-es2015.76577bcb5e5ae75dda56.js (main) 482 kB [initial] [rendered]
chunk {1} main-es5.76577bcb5e5ae75dda56.js (main) 577 kB [initial] [rendered]
chunk {0} runtime-es2015.0dae8cbc97194c7caed4.js (runtime) 1.45 kB [entry] [rendered]
chunk {0} runtime-es5.0dae8cbc97194c7caed4.js (runtime) 1.45 kB [entry] [rendered]
chunk {4} styles.7eaf9928554c069e972c.css (styles) 219 kB [initial] [rendered]
chunk {scripts} scripts.fd6c5645829e82918226.js (scripts) 166 kB [entry] [rendered]
Date: 2020-07-13T15:10:09.166Z - Hash: 2116454e7c64b4640470 - Time: 31440ms


🚧 Executing Docker Build...
Sending build context to Docker daemon  6.474MB

Step 1/6 : FROM nginx:alpine

 ---> 7d0cdcc60a96
Step 2/6 : LABEL version="1.0.0"
 ---> Using cache
 ---> 1e9a627ad681
Step 3/6 : COPY nginx.conf /etc/nginx/nginx.conf

 ---> Using cache
 ---> 60d153c53ea2
Step 4/6 : WORKDIR /usr/share/nginx/html


 ---> Using cache
 ---> ea655bd13392
Step 5/6 : COPY dist/company-z-ui .

 ---> Using cache
 ---> cec18756ec5f
Step 6/6 : EXPOSE 8080

 ---> Using cache
 ---> 756611ea83fb

Successfully built 756611ea83fb

 ---> Using cache
 ---> cec18756ec5f
Step 6/6 : EXPOSE 8080

 ---> Using cache
 ---> 756611ea83fb

Successfully built 756611ea83fb

Successfully tagged tapanprajapati/company-z-ui:latest

✔️Docker Build was successfully
🚀 Publishing image to registry
The push refers to repository [docker.io/tapanprajapati/company-z-ui]

190025b1316b: Preparing
92a6a30c1b99: Preparing
a181cbf898a0: Preparing

570fc47f2558: Preparing
5d17421f1571: Preparing
7bb2a9d37337: Preparing
3e207b409db3: Preparing
7bb2a9d37337: Waiting
3e207b409db3: Waiting

a181cbf898a0: Layer already exists

5d17421f1571: Layer already exists

92a6a30c1b99: Layer already exists

570fc47f2558: Layer already exists

190025b1316b: Layer already exists

7bb2a9d37337: Layer already exists

3e207b409db3: Layer already exists

latest: digest: sha256:7417e1d9fa1d3fc136c40643e31ddf94bf93820fcfe8657b97384338f02c760d size: 1778

🎊 Successfully published image. Have a nice day.
```



Figure 49: Docker Image - Company Z front-end

Step 2: Created environment on Elastic Beanstalk for front-end as shown in figure 50.

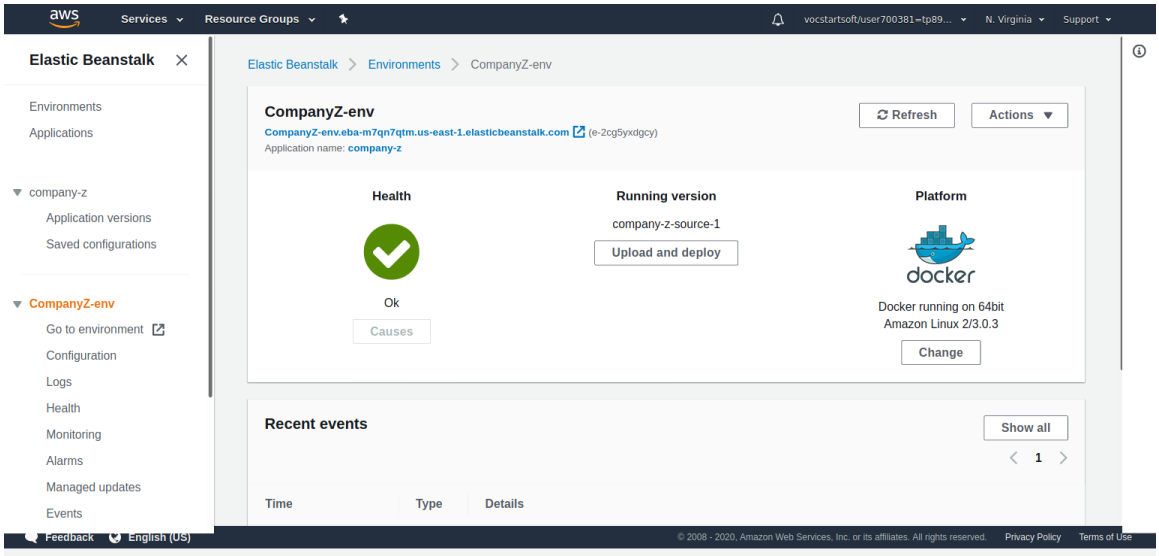


Figure 50: EB environment - Company Z front-end

Step 3: *Process to containerize and deploy backend* - The step outlines the workflow to build an image for the company-z application's backend using docker, host the build image on a remote repository, and create an environment on AWS Elastic beanstalk to run the docker image. Steps to build docker image and deployment on AWS Elastic has shown for the back end of company-z.

*App containerization on Local Machine* It containerize the application using docker. The local machine environment is already configured to build a docker image. To build an image for the company-z-endpoints Nodejs app (refer to figure 51), go to the root of the application directory, open terminal and run following command:

```
docker build -t "parthsw/company-z-endpoints" .
```

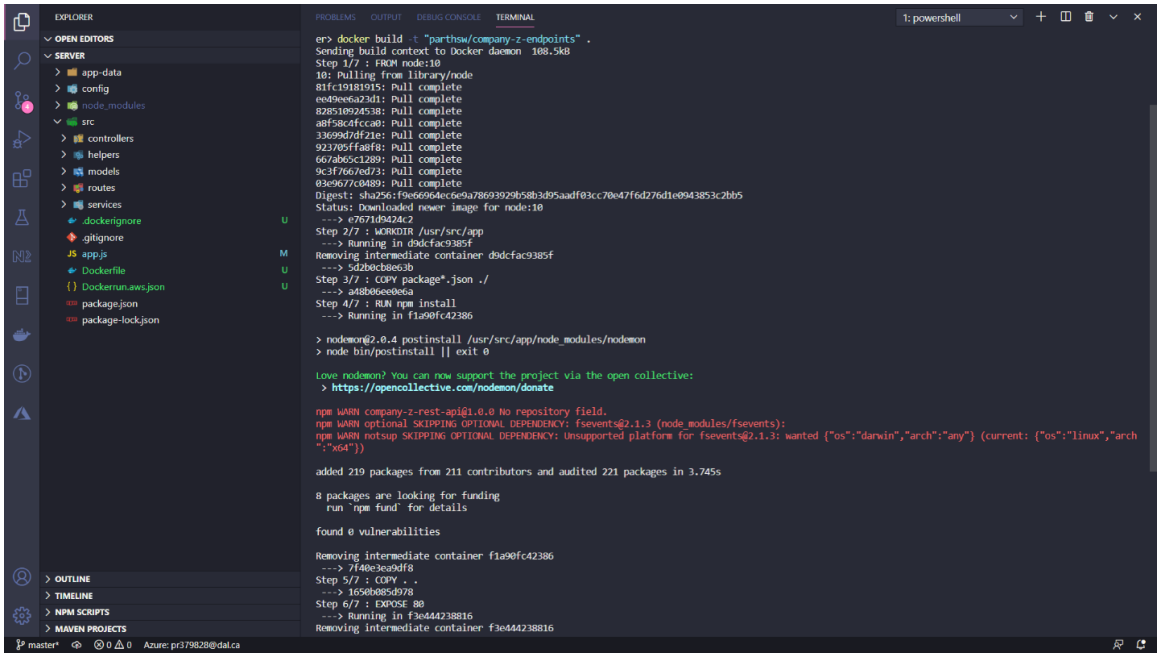


Figure 51: Building Docker Image - Company Z backend

Figure 52 illustrates the successful execution of building a docker image for the company-z-endpoints Node.js application.

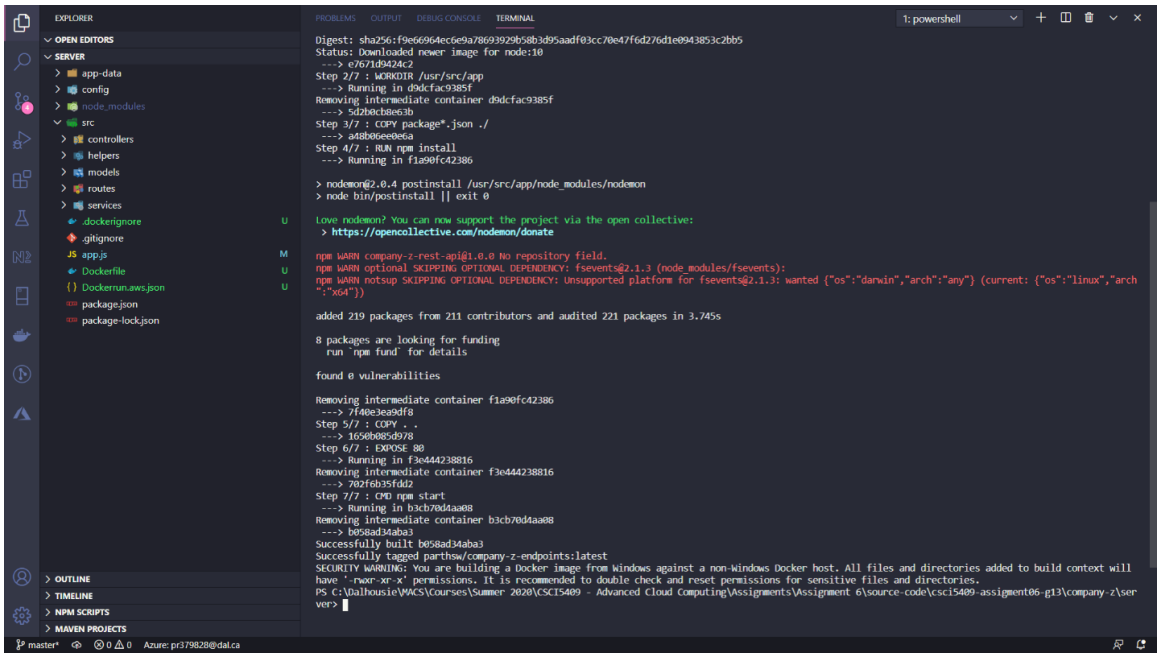


Figure 52: Docker Image Successful - Company Z backend

Please refer to figure 53 depicting the list of images available on the machine,

including recently created "parthsw/ company-z-endpoints".

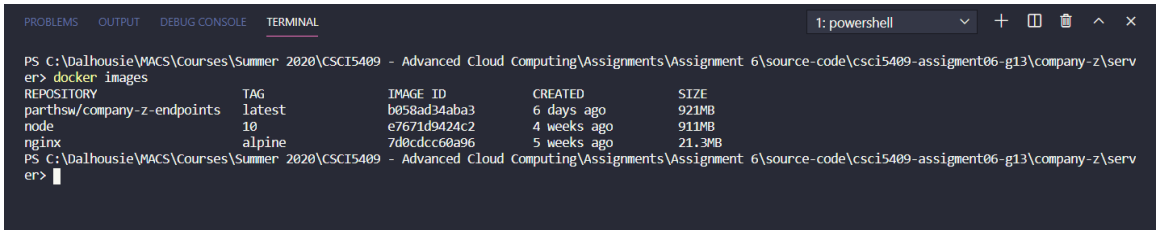


Figure 53: Docker Images

Step 4: *Hosting the docker image* - Once the docker image is created, the next step is to host it in a remote environment so that the Elastic Beanstalk application can easily pick the image from a remote location and run it inside a container.

Sign up on Docker Hub to manage the docker container application. Once successfully registered, create a public repository to maintain the docker image. As shown in figure 54, provide the name, description, and select Public in the Create repository form.

The next step is to push the image on the created repository. First, it requires a login to the Docker Hub. Please execute the below command, which asks you to enter username and password. See figure 55 showing the successful login to the Docker Hub.

```
docker login
```

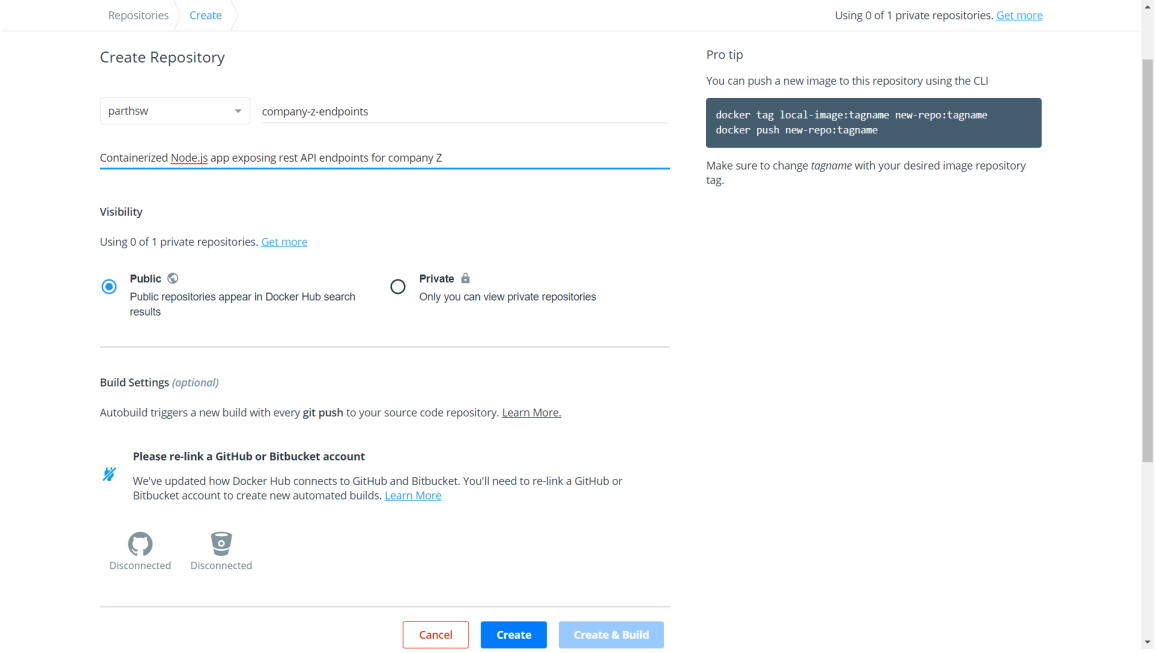


Figure 54: Docker Hub Repository for Company Z

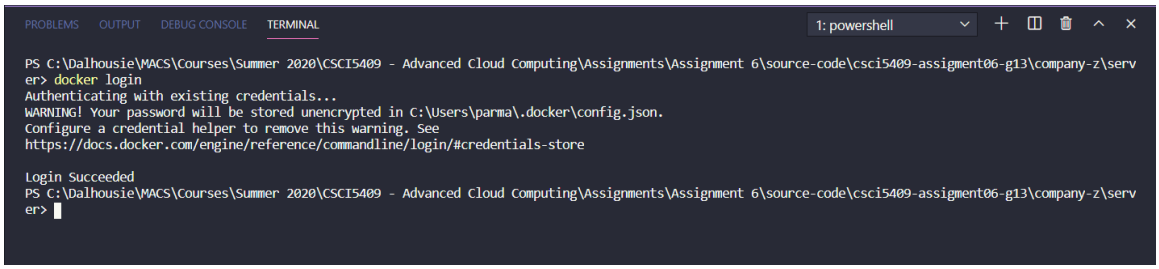


Figure 55: Docker Hub - Login Success

Once logged in, execute the below command to push the image to the remote repository, as shown in Figure 56. Figure 57 shows the docker image pushed on the Docker Hub repository.

```
PS C:\Dalhousie\MACS\Courses\Summer 2020\CSCI5409 - Advanced Cloud Computing\Assignments\Assignment 6\source-code\csci5409-assigment06-g13\company-z\serv
er> docker push "parthsw/company-z-endpoints"
The push refers to repository [docker.io/parthsw/company-z-endpoints]
883355103830: Pushed
d593eb890d84: Pushed
29ad5dab226a: Pushed
51dd4a1a1879: Pushed
d1e551c519b5: Mounted from library/node
70a3a00c2b44: Mounted from library/node
918a6facd6de: Mounted from library/node
bc17cd405095: Mounted from library/node
ee854067fbbd: Mounted from library/node
740ffea5d5c3: Mounted from library/node
eac9ead92b24: Mounted from library/node
23bca356262f: Mounted from library/node
8354d5896557: Mounted from library/node
latest: digest: sha256:f9bc485c7f19fbce8aca395fe6b16165b6c01ab3f8bbfed2edac96a34c7c7587 size: 3051
PS C:\Dalhousie\MACS\Courses\Summer 2020\CSCI5409 - Advanced Cloud Computing\Assignments\Assignment 6\source-code\csci5409-assigment06-g13\company-z\serv
er>
```

Figure 56: Pushing Image to Docker Hub

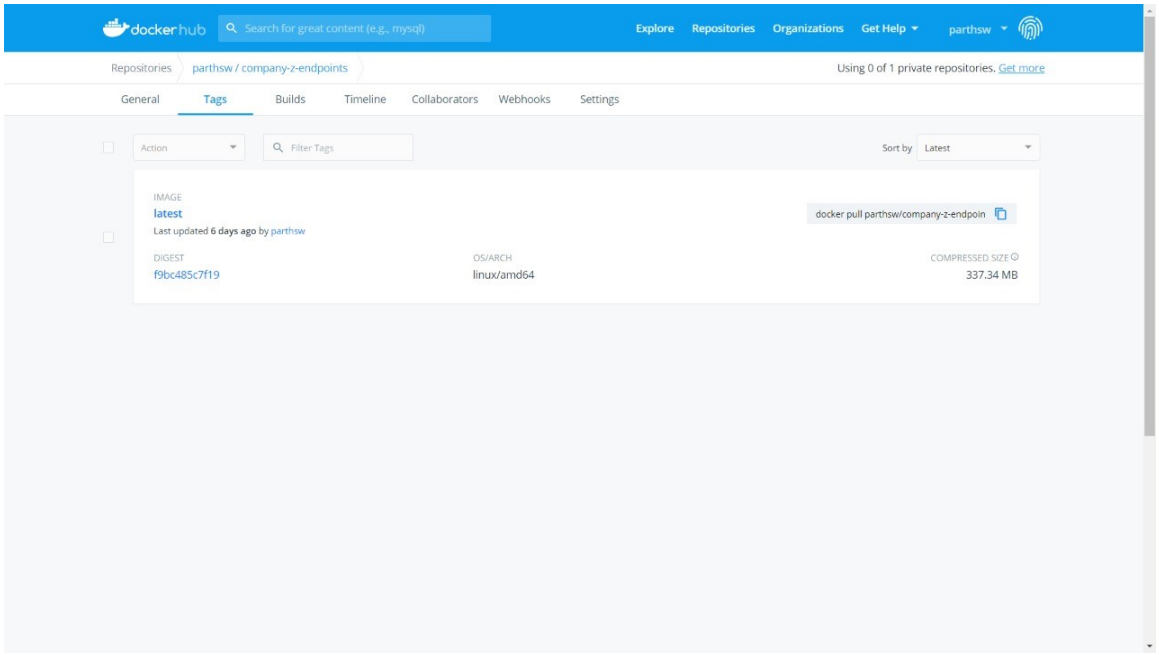


Figure 57: Docker Image on Docker Hub

Step 5: *AWS Elastic Beanstalk environment setup* - Figure 58 shows the environment of elastic beanstalk for the company z endpoints.

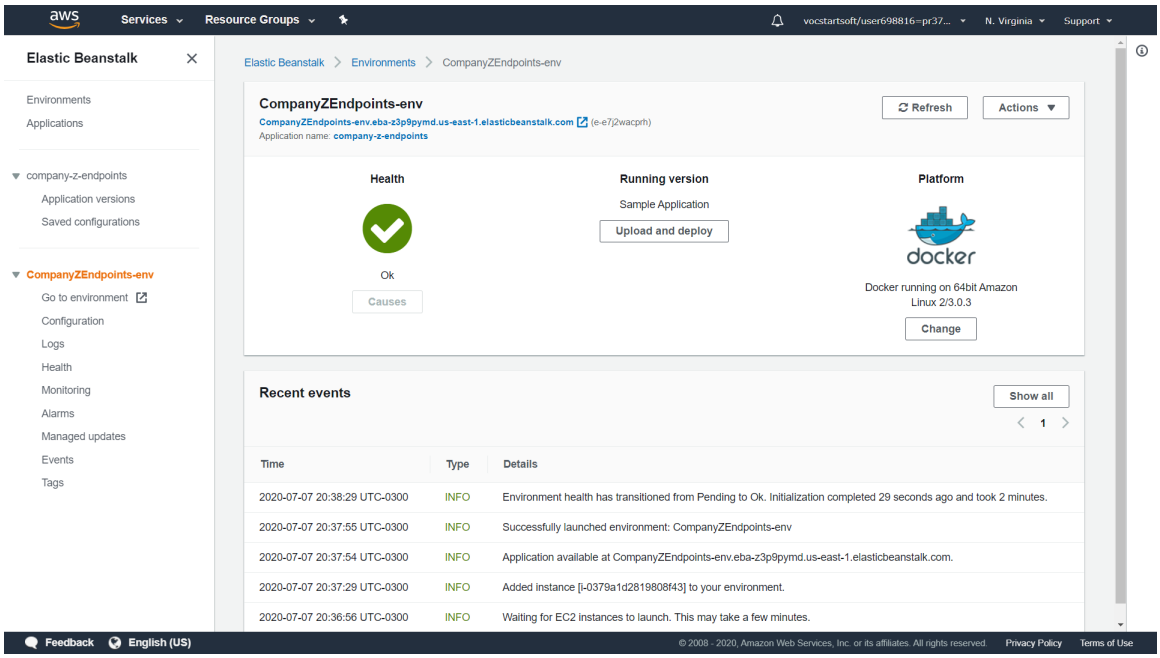


Figure 58: EB environment - Company Z backend

# 6 Test Cases

The section discusses various test cases performed with the developed applications. All testcases are executed using the Google Chrome browser.

Quick links for applications:

Company X [11] Company Y [12] Company Z [13]

## 6.1 Test Cases – Company X

Following are the list of tests carried out to test the different functionalities provided by the company X.

Test 1: *Landing Page* - Figure 59 shows the landing page of the user interface for company X. User has four options on the landing page namely, view jobs, add job, view specific job, and search job.

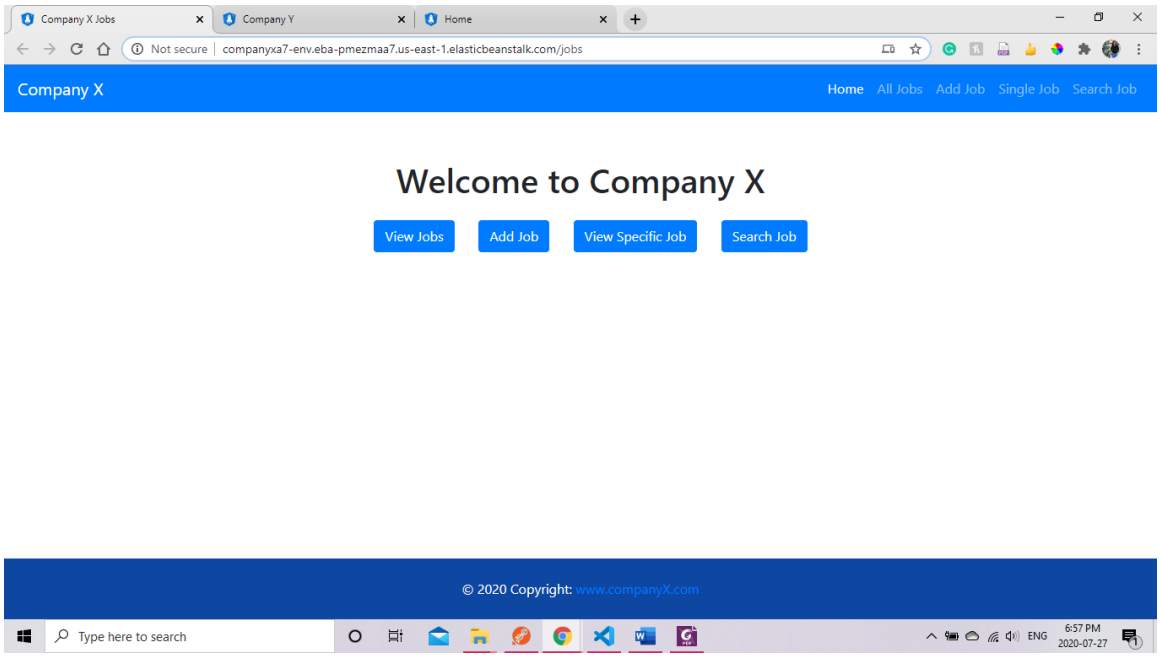


Figure 59: Landing Page Company X

Test 2: *View Job* - When user click on the view jobs then application will display all the jobs available as shown in the figure 60.

[11] http://companyxa7-env.eba-pmezmaa7.us-east-1.elasticbeanstalk.com/jobs
[12] http://companyyui-env-1.eba-kffumsec.us-east-1.elasticbeanstalk.com/parts
[13] http://companyza7-env.eba-59sjk5ev.us-east-1.elasticbeanstalk.com/home

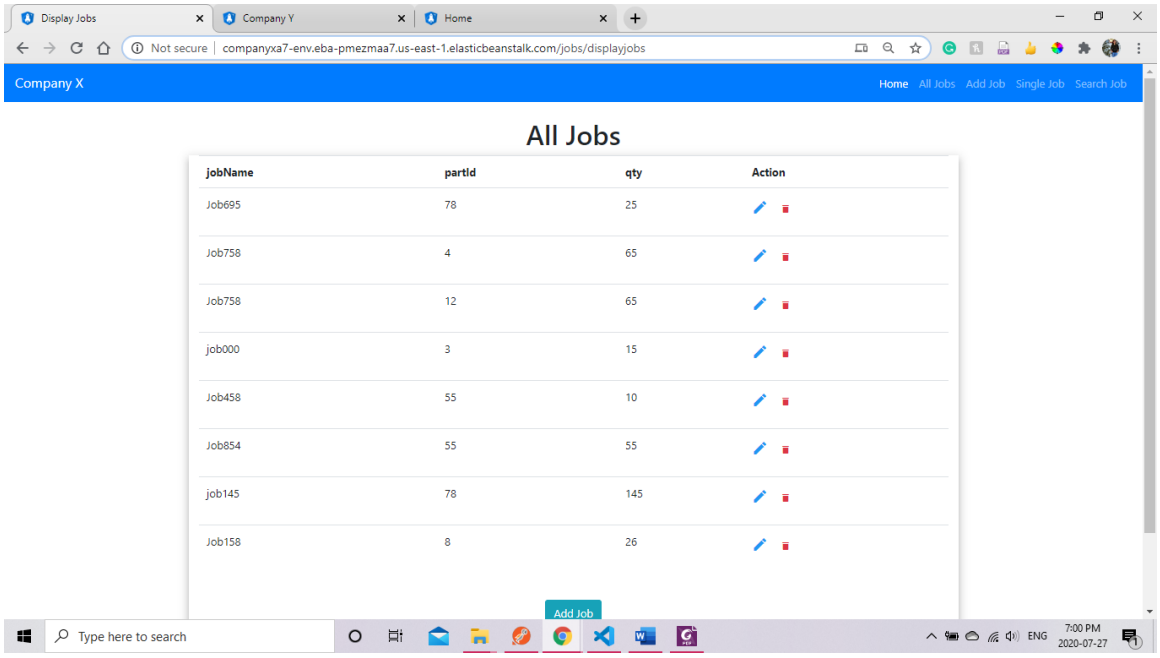


Figure 60: View all jobs

Test 3: *Get specific Job (Validation)* - User can search for a specific job as well. But user need to enter the jobname and part id for that. The search button will be deactivated until the user enters valid details as shown in figure 61.

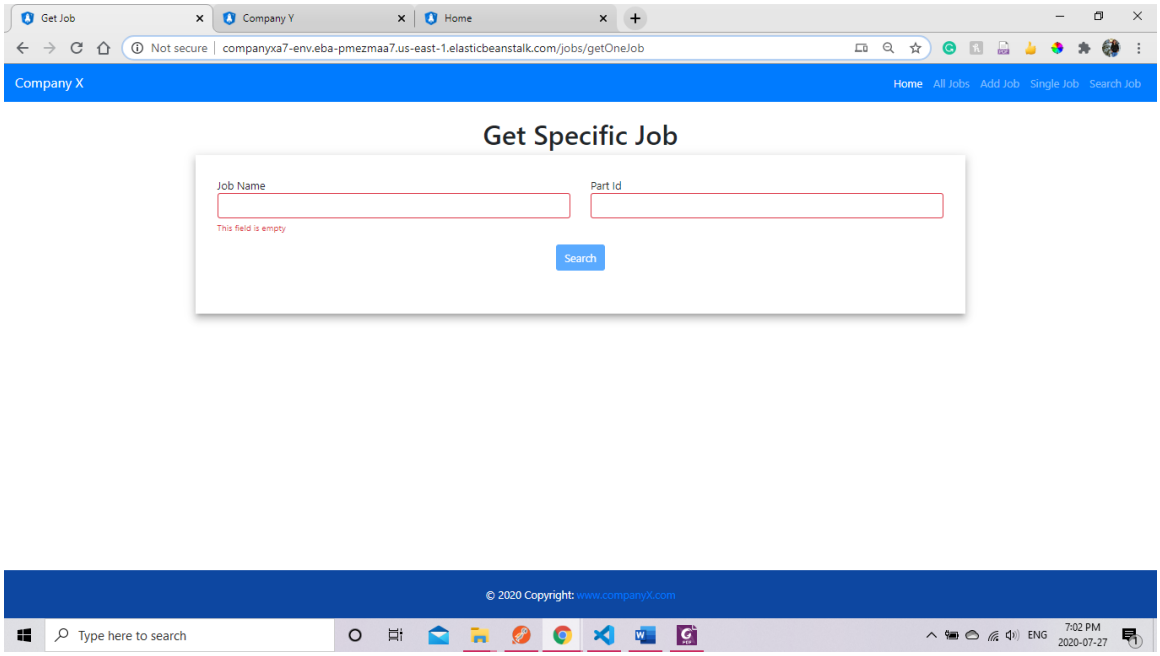


Figure 61: Get specific job Validation

Test 4: *Get specific job* - User enters the jobname as "job000" and partId as "3" and click on the search button as shown in figure 62.

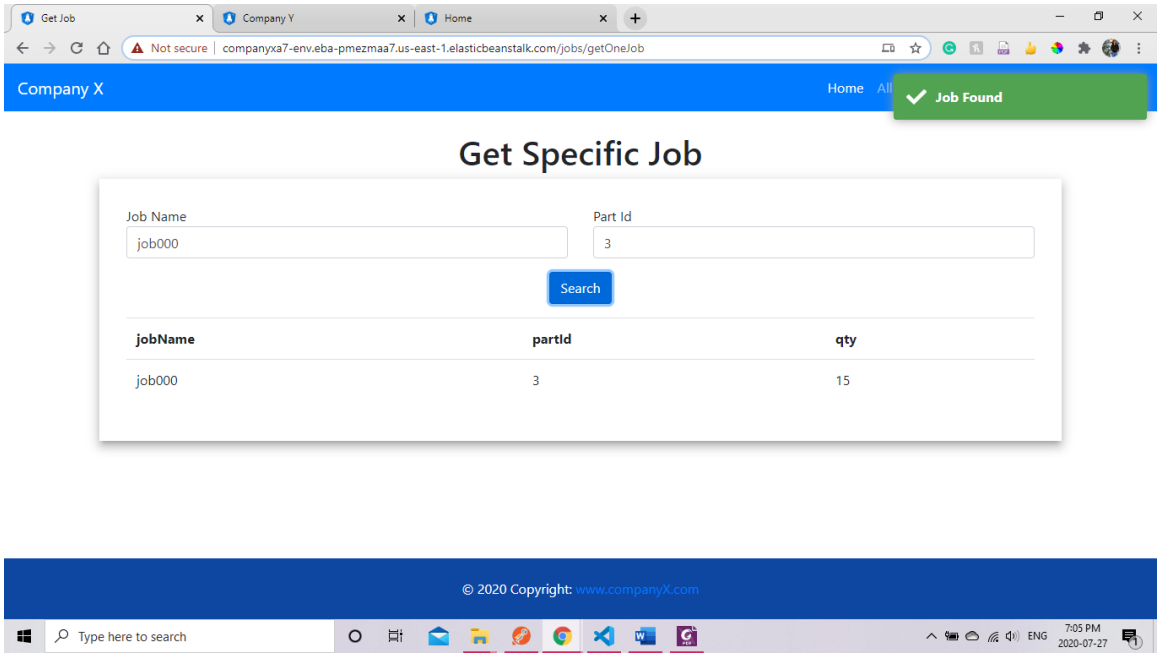


Figure 62: Get specific job

Test 5: *Add an existing job* - User can create a new job. But if user try to create job which already exists then application will show proper appropriate error message as shown in the figure 63.

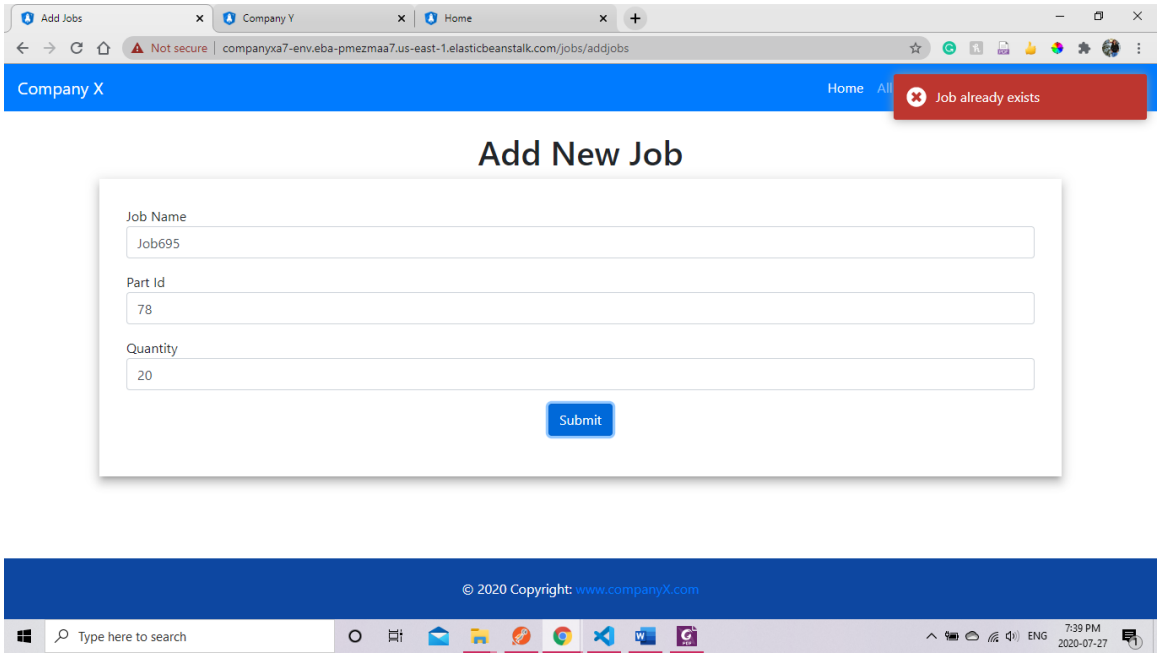


Figure 63: Add an existing job

User will get an error message as shown in Figure 6 while creating new job in which partId does not exist in database. Figure 64 shows the proper error message because partId : '77' does not exists in parts table.

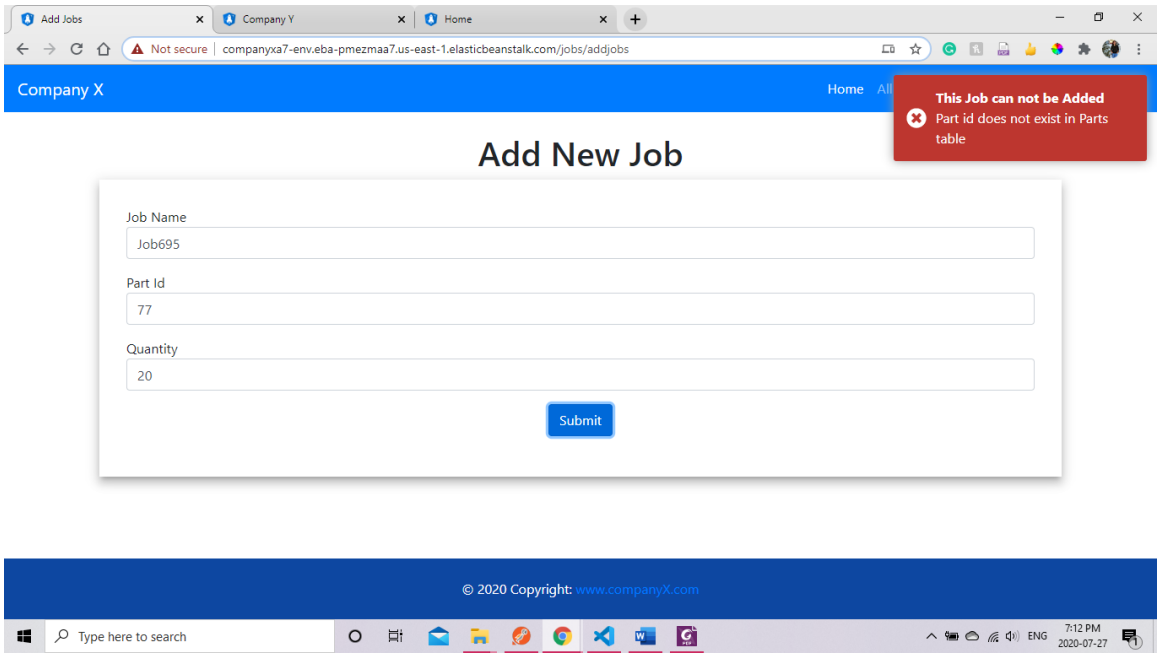


Figure 64: Add job - error

Test 6: *Create a new job* - User can create a new job as well. User need to provide the three inputs such as jobname, partId, and quantity as shown in Figure 65 . When user click on the submit button, application will check that does that partid exist or not. If the part id does not exist, then it will show error as shown in the Figure 64. But if it exists, and jobname and partid make a unique entry then it will create a new job. Application will show success message as shown in the Figure 66.

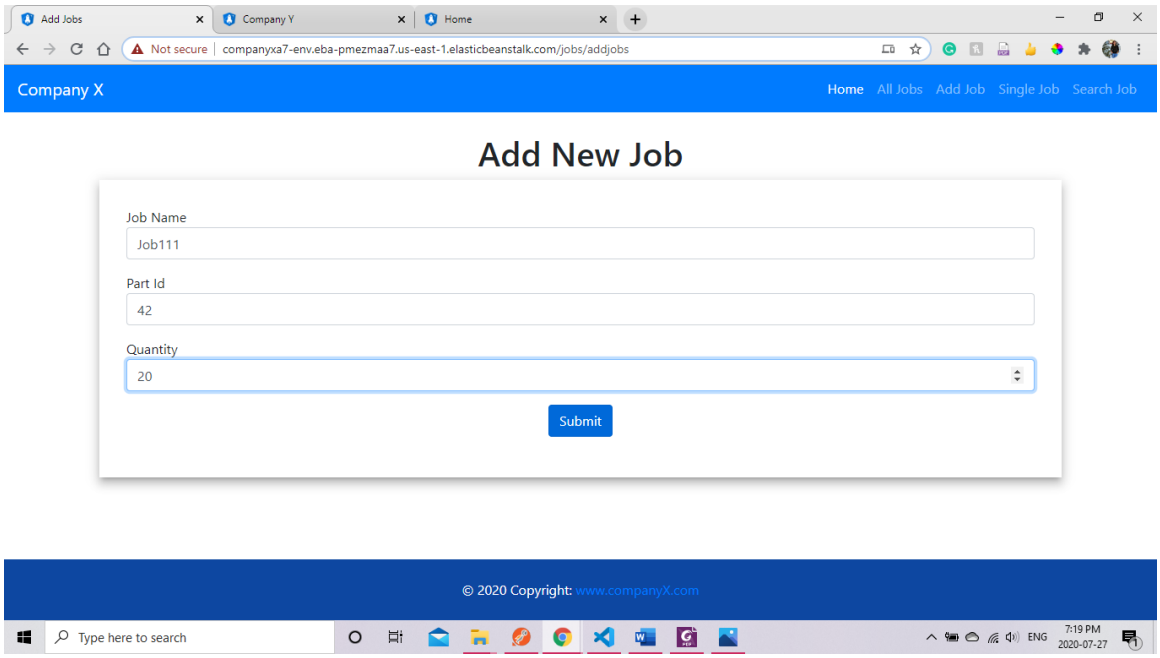


Figure 65: Create a New Job

When user click on the submit then it will also be added into all jobs list as shown in the figure 66.

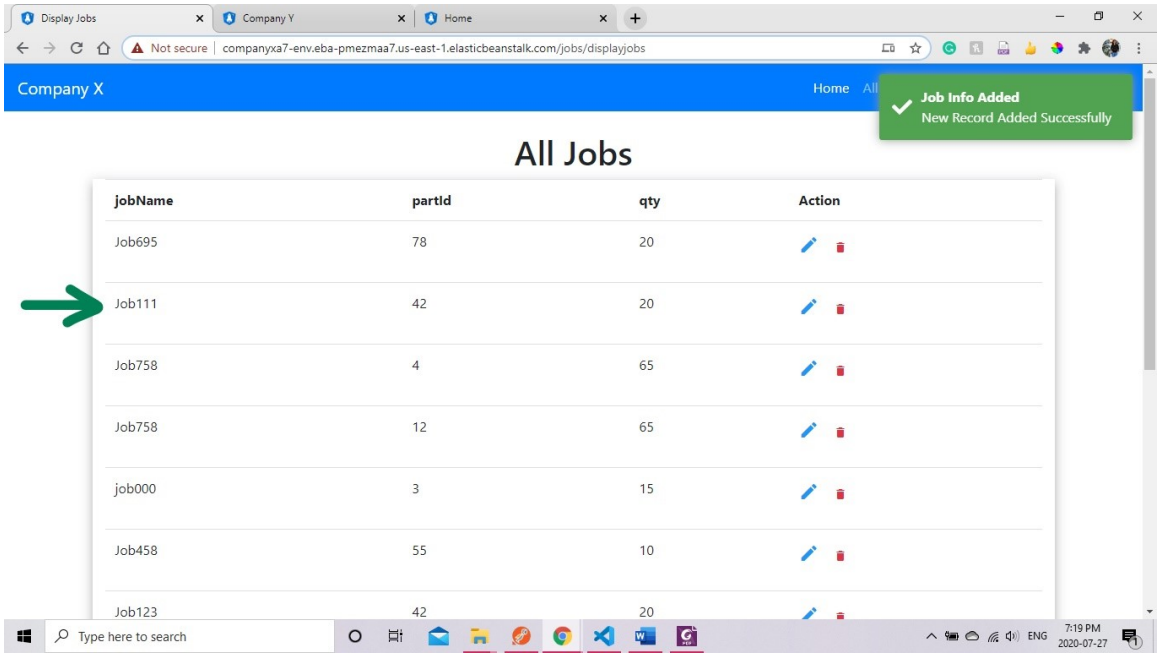


Figure 66: Job created successfully

Test 7: *Delete Job* - User can delete the job as well using the delete option. Delete option is available in the action column. Application will ask for the confirmation as shown in the figure 67.

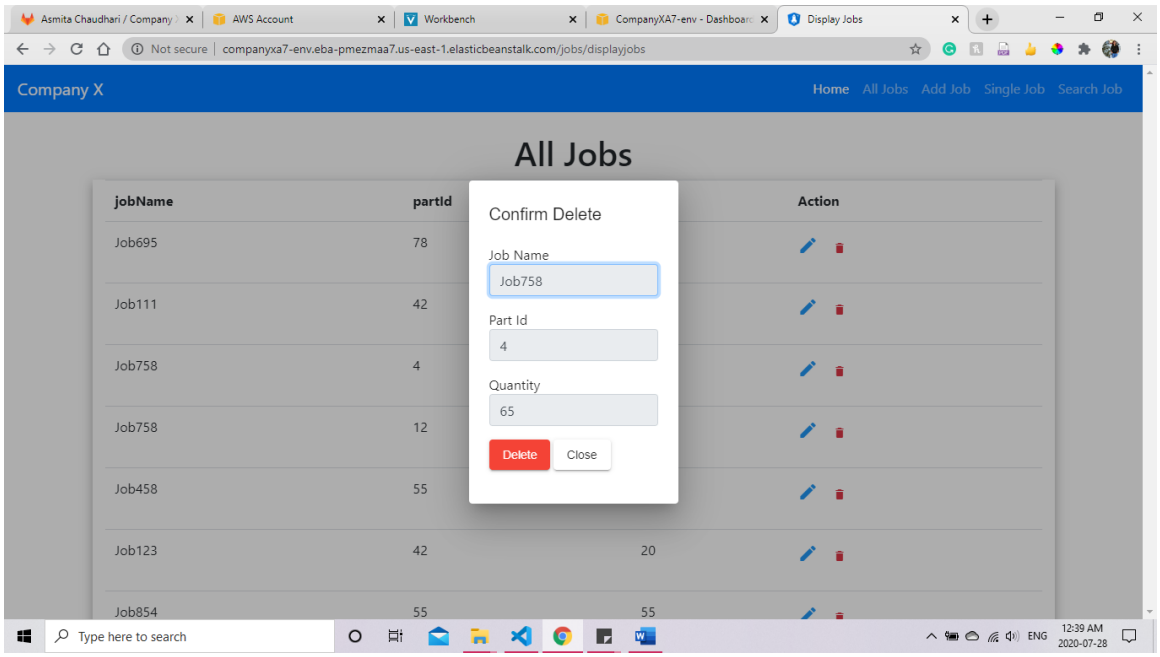


Figure 67: Delete Job

If user clicks on the delete button then it will be deleted, and the success message will be displayed as shown in the figure 68. You can notice in the Figure 68 that deleted job is not listed in all jobs.

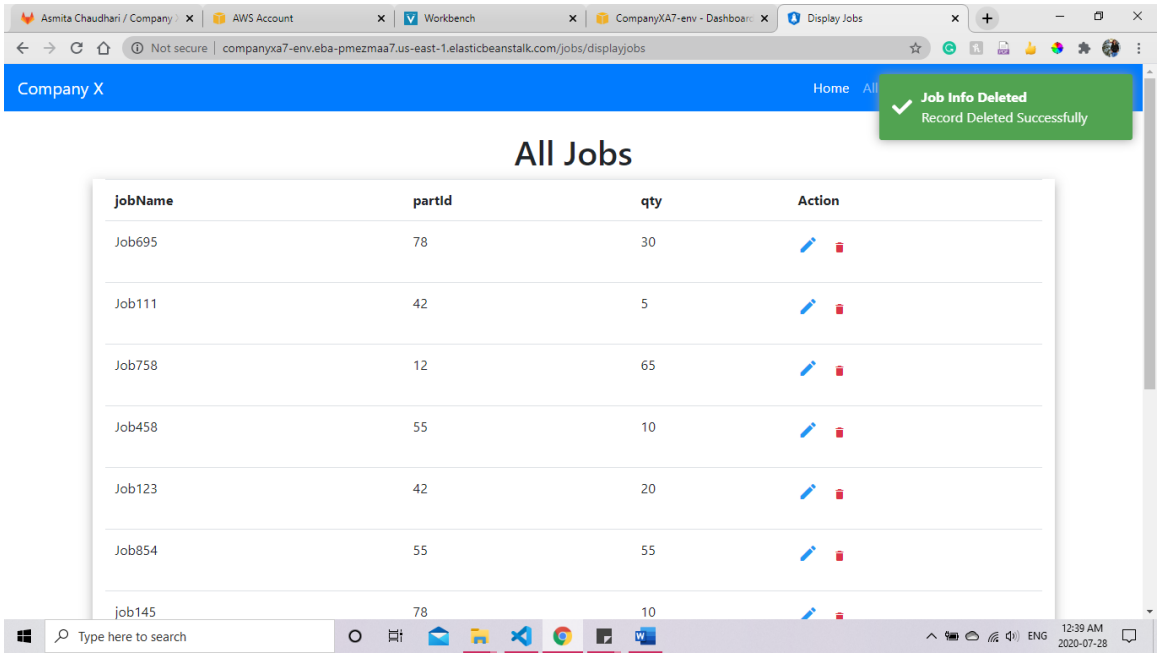


Figure 68: Job Deleted

Test 8: *Update Job Details* - User can edit the quantity of the existing job. User need to enter the new quantity as shown in the figure 69.

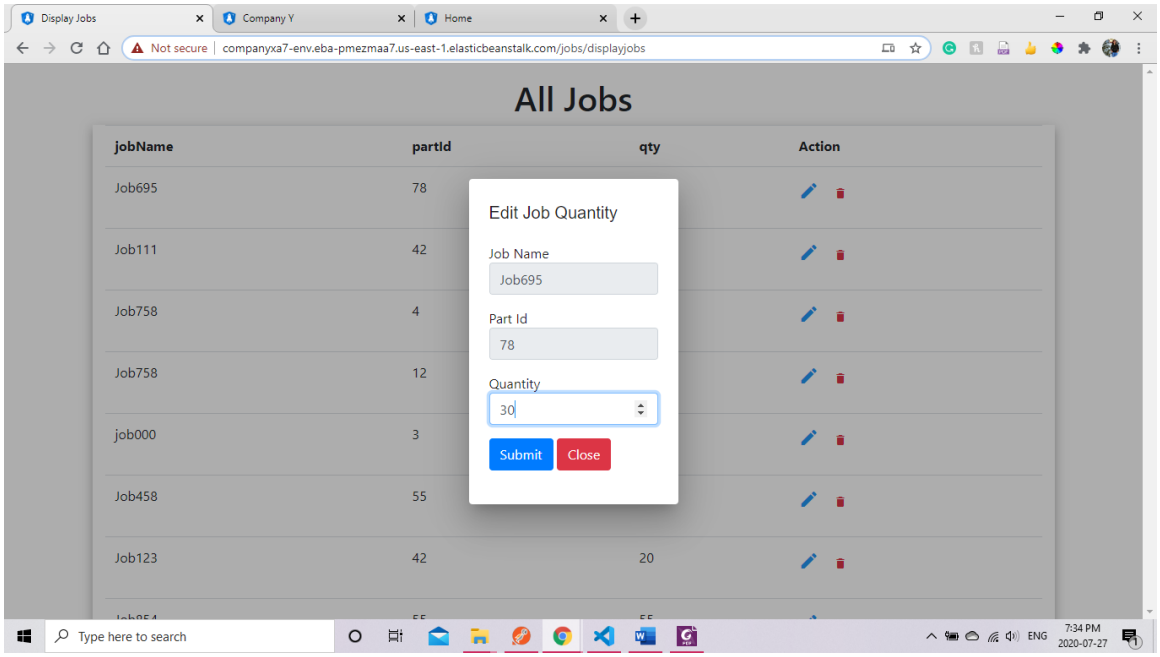


Figure 69: Update Job

When user click on the update button, it will update the quantity and success message will be displayed as shown in the figure 70.

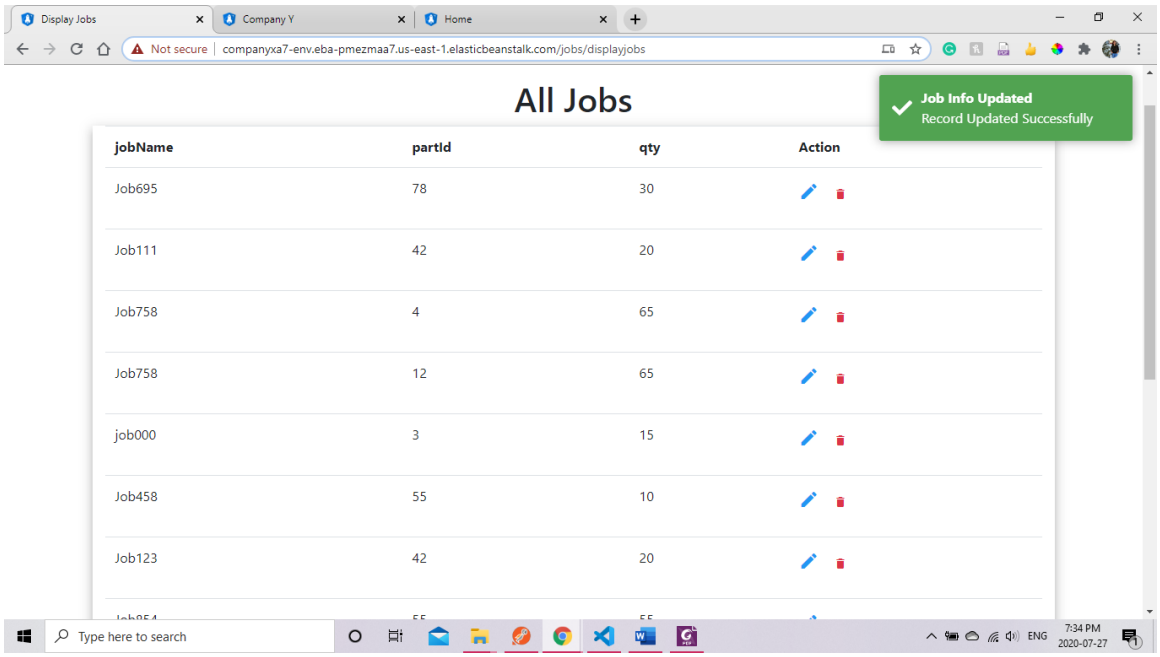


Figure 70: Job updated successfully

Test 9: *Search job (Found)* - User can search for the job using the jobname. User need to enter

the jobname and click on the search; application will show the list of jobs available with the name as shown in the figure 71.

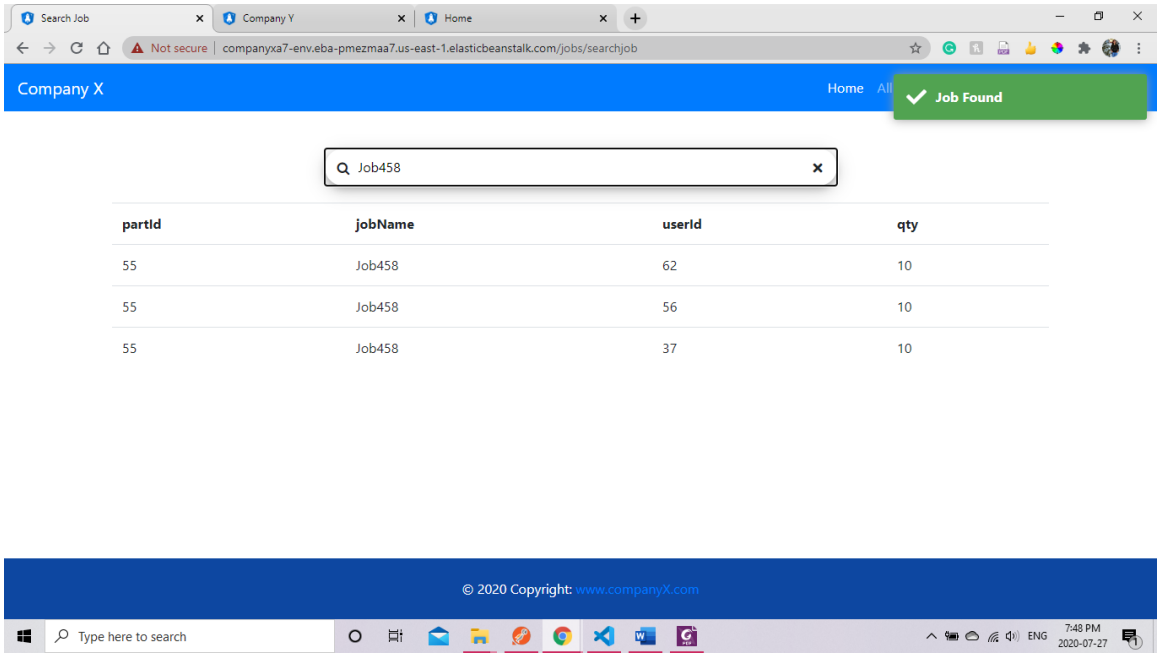


Figure 71: Search Job (Found)

Test 10: *Search Job (Not found)* - When user enters the job name which does not exist then application will show a proper error message as shown in the figure 72.

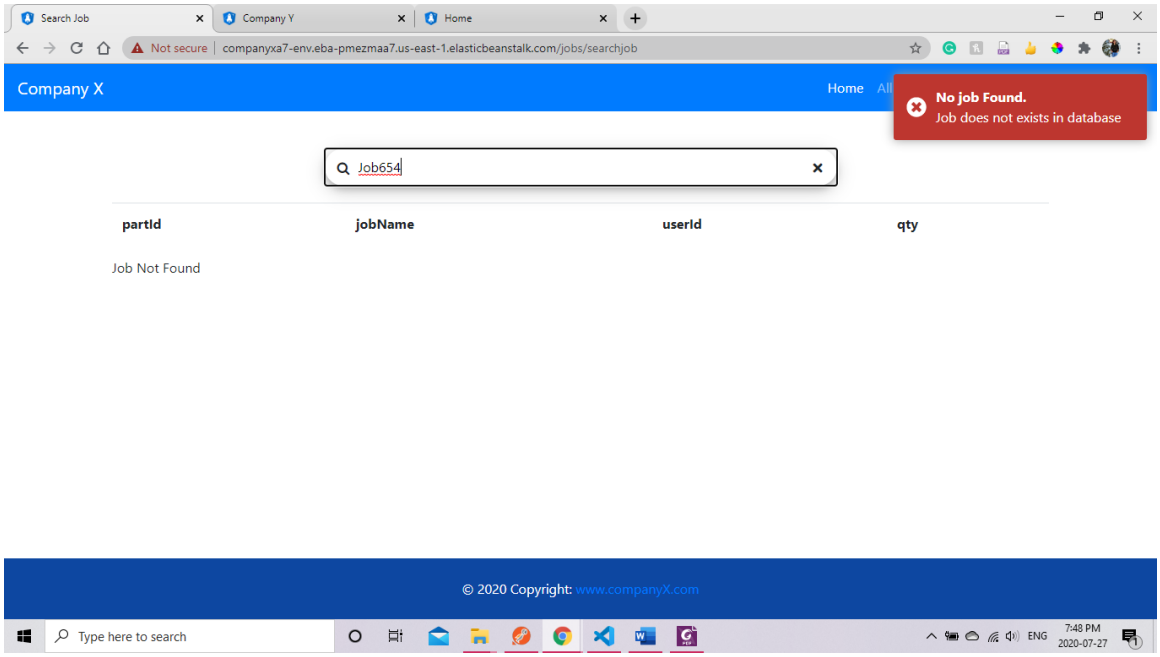


Figure 72: Search Job (Not Found)

## 6.2 Test Cases – Company Y

Following are the list of tests carried out to test the different functionalities provided by the company Y:

Test 1: *Viewing all the parts* - Application shows all the parts available from the table on the landing page of an application. It also shows the details of quantity available as shown in figure 73.

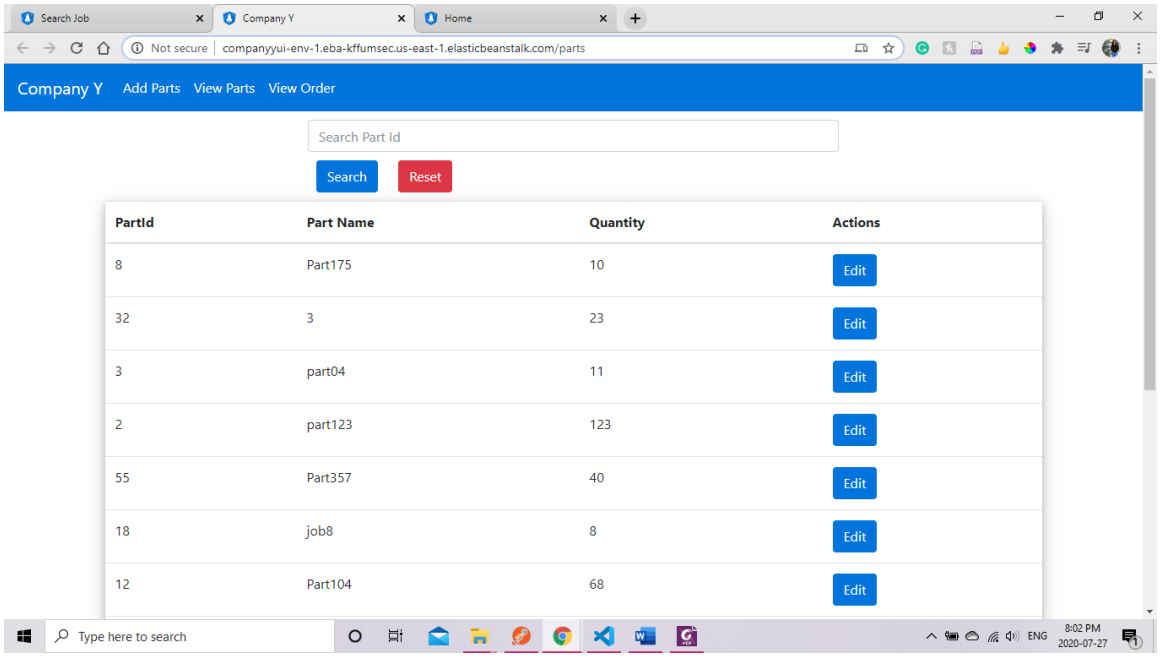


Figure 73: Listing all the parts

Test 2: *Part not found* - User can search for part from the search bar given on the landing page. If the part is not available, then application will show appropriate error message as shown in figure 74.

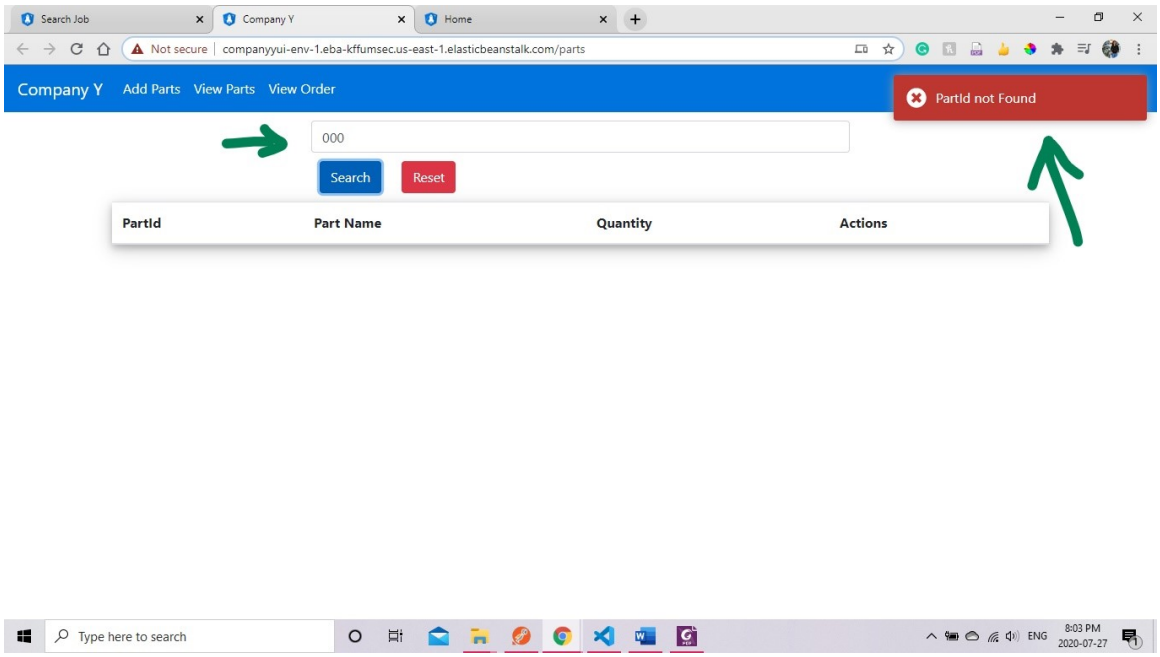


Figure 74: Search part (Not Found)

Test 3: *Search Part* - User can search for part using part id. Application will show the details of the part, which user searched as shown in the figure 75.

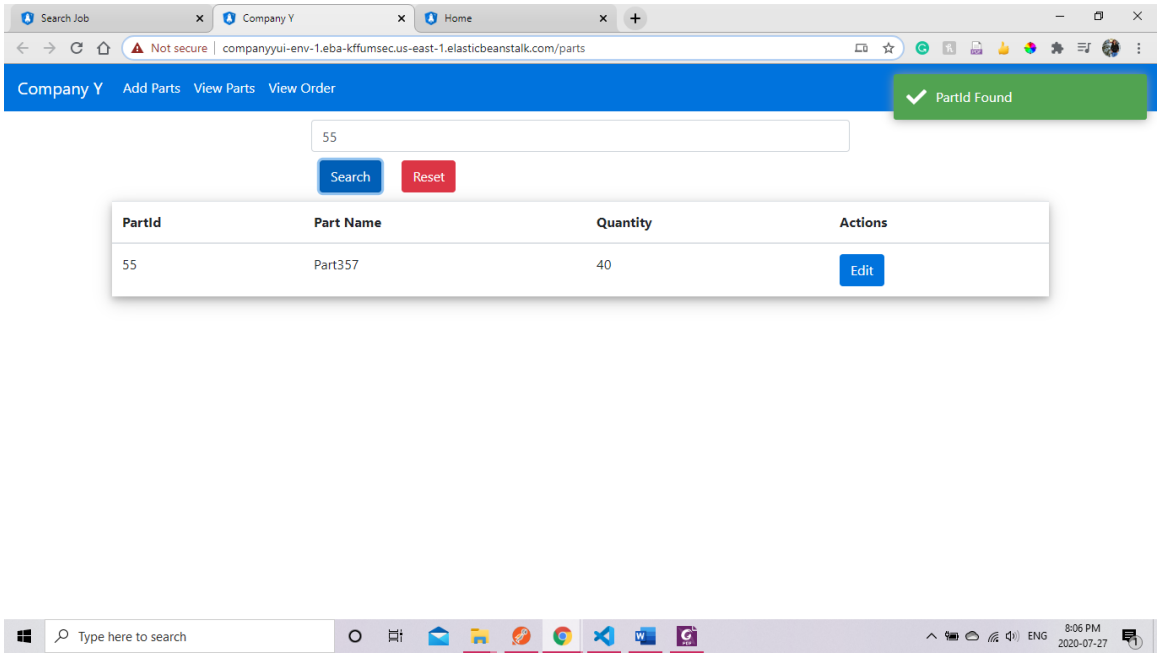


Figure 75: Search Part (Found)

Test 4: *Add an existing part* - User can add the details of the new part as well. If user add the part which already exists, then application will show the appropriate error message as shown in figure 76.

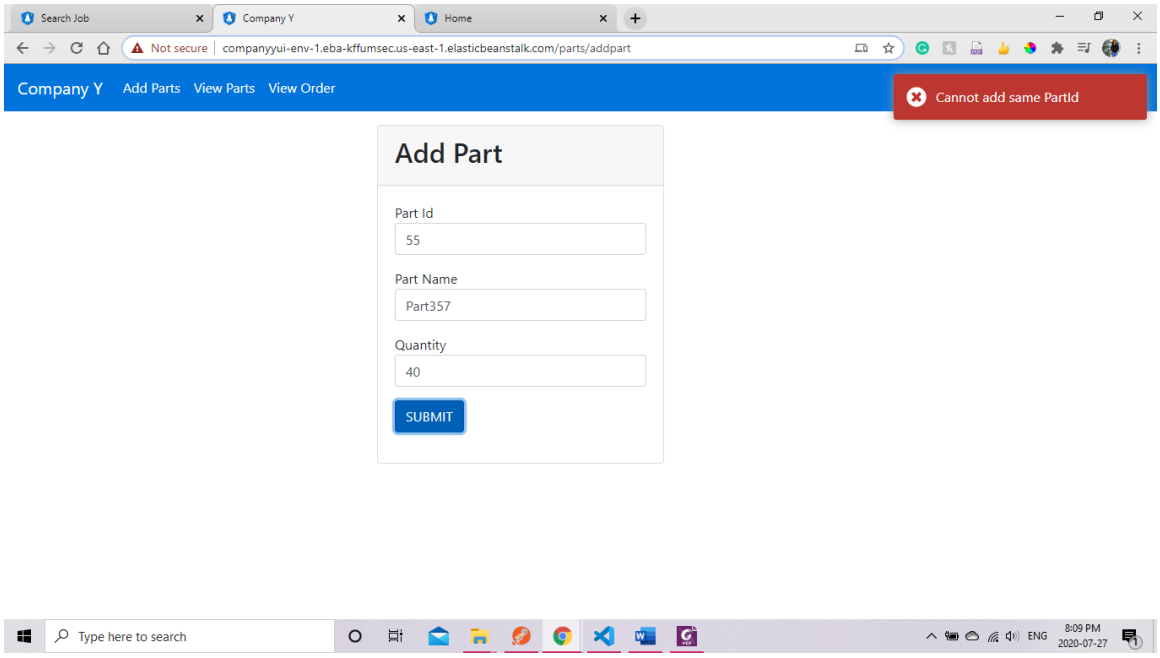


Figure 76: Add an existing part

Test 5: *Add invalid detail* - User must add the details of the part at the time of creating new part. When user does not enter all the fields required to create a part, then application will deactivate the submit button and show the proper appropriate error message as shown in figure 77.

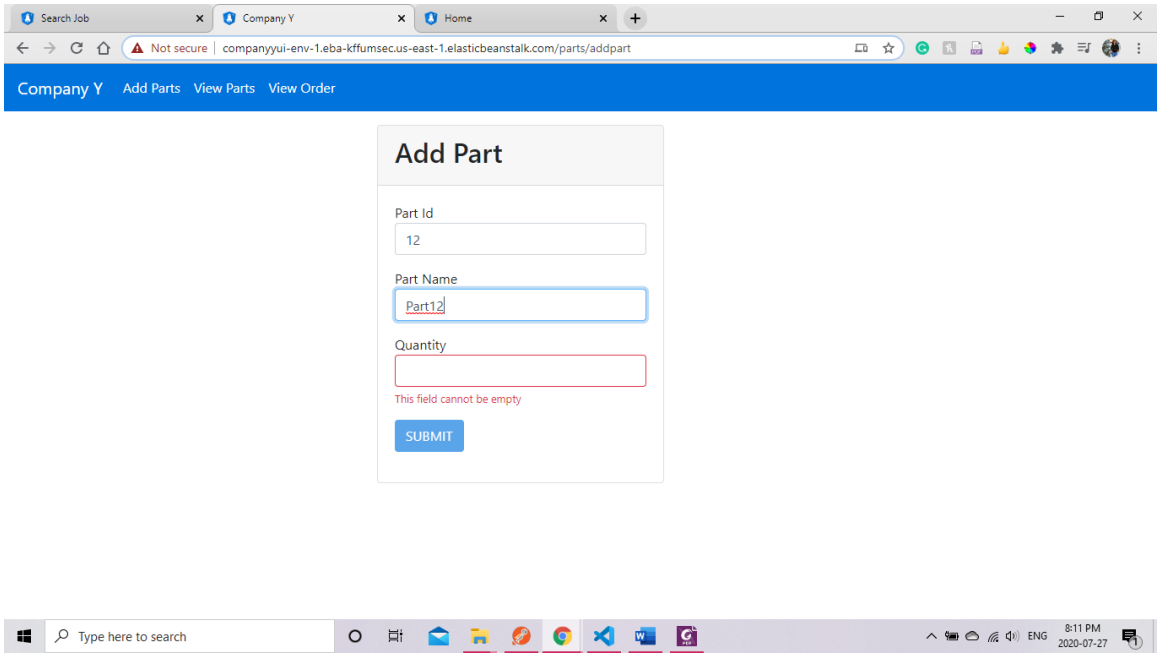


Figure 77: Add invalid details

Test 6: *Add a new part* - User enters valid details about part and click on the submit button. The data will be added in the database and shown in the list of parts. You can see in the figure 78 that the submit button is activated.

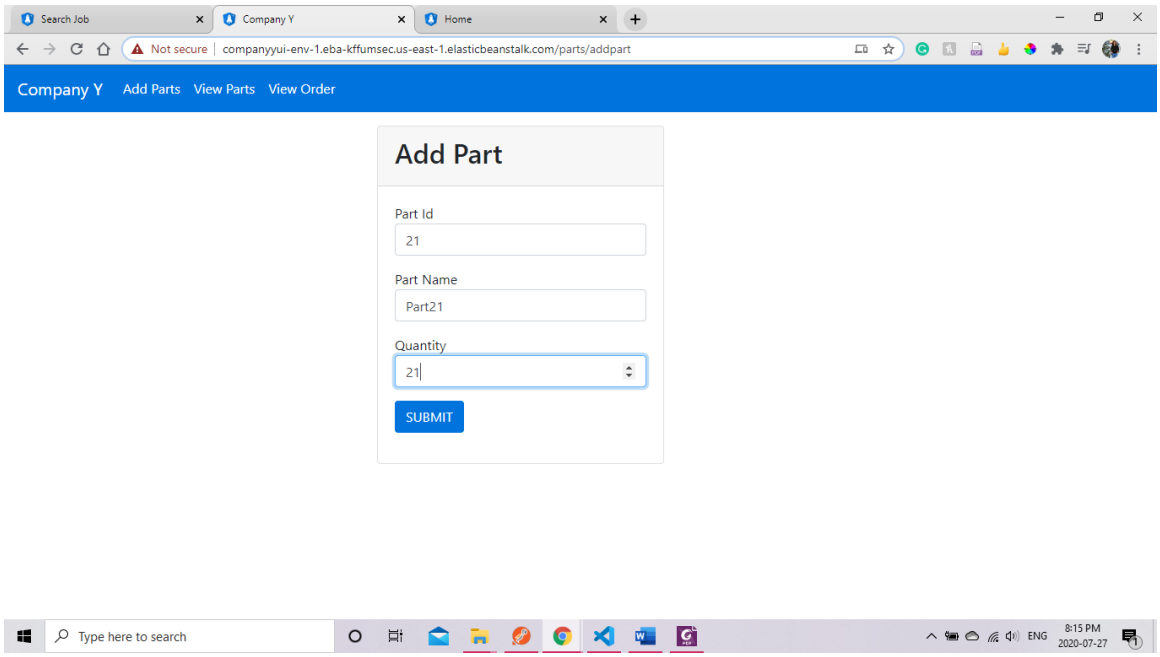


Figure 78: Add new part

New part is added and updated in the list of all parts. You can see that in the figure 79.

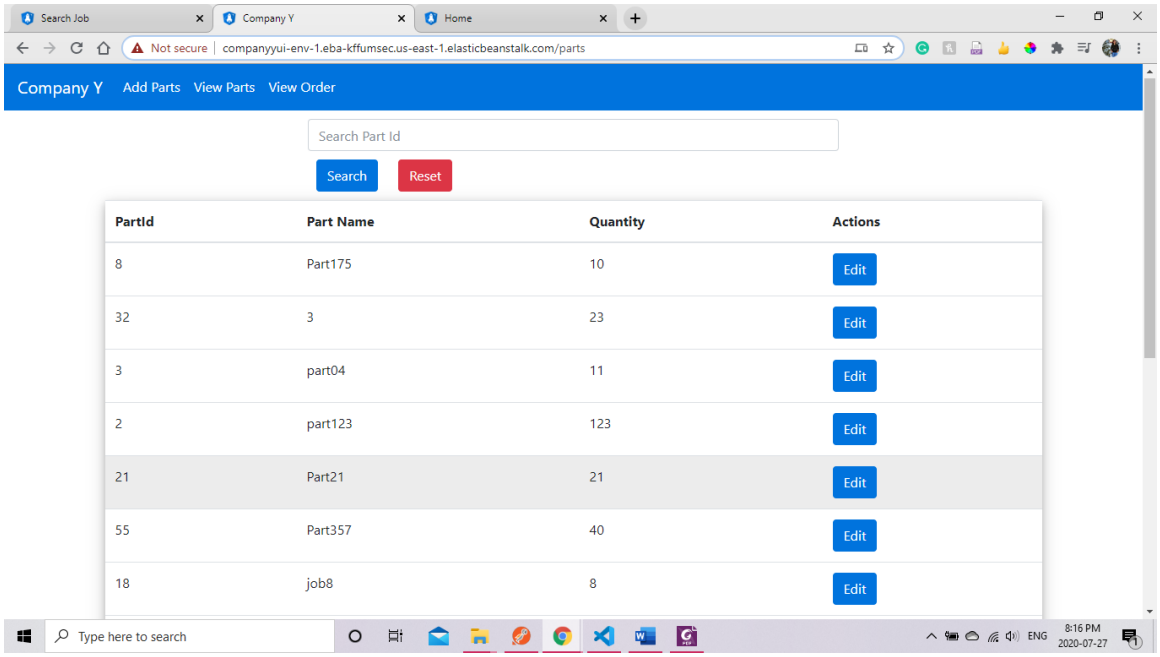


Figure 79: New part added successfully

Test 7: *Update part details (Invalid)*- User can only update the quantity of the part. If user tries to click on the update without entering all fields then application will deactivate the update button and show appropriate error message as shown in figure 80.

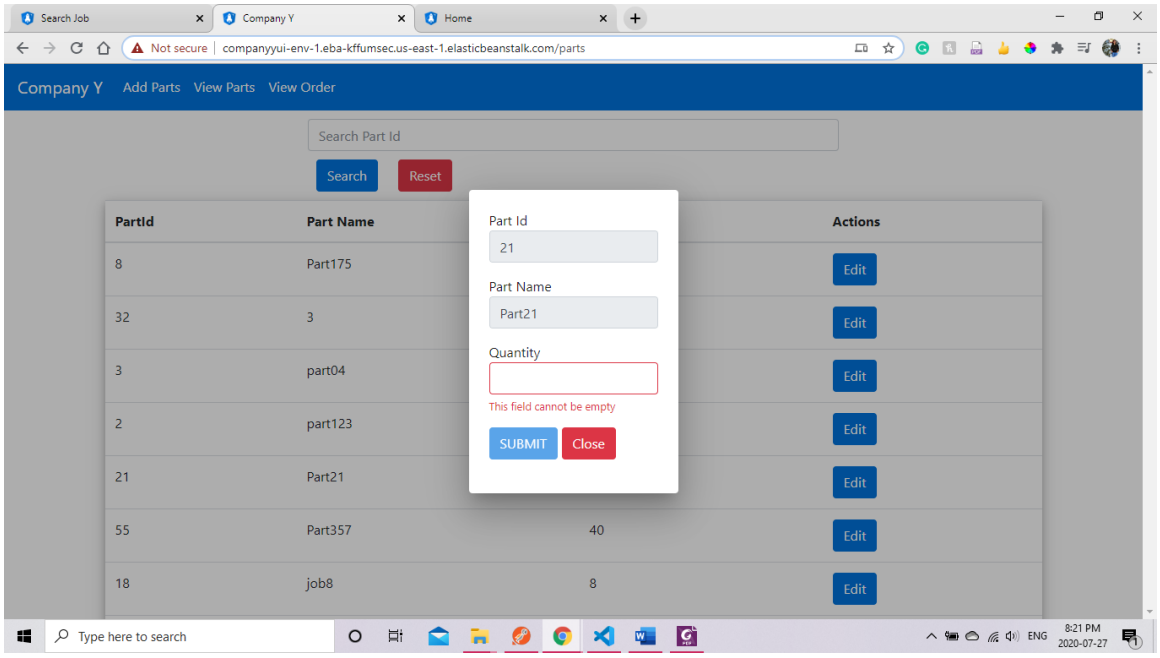


Figure 80: Update part details (Invalid)

Test 8: *Update part details* - User enters the new quantity of the part and click on the submit button as shown in figure 81. Application will show success message and updated quantity in the view all parts table as shown in the figure 82.

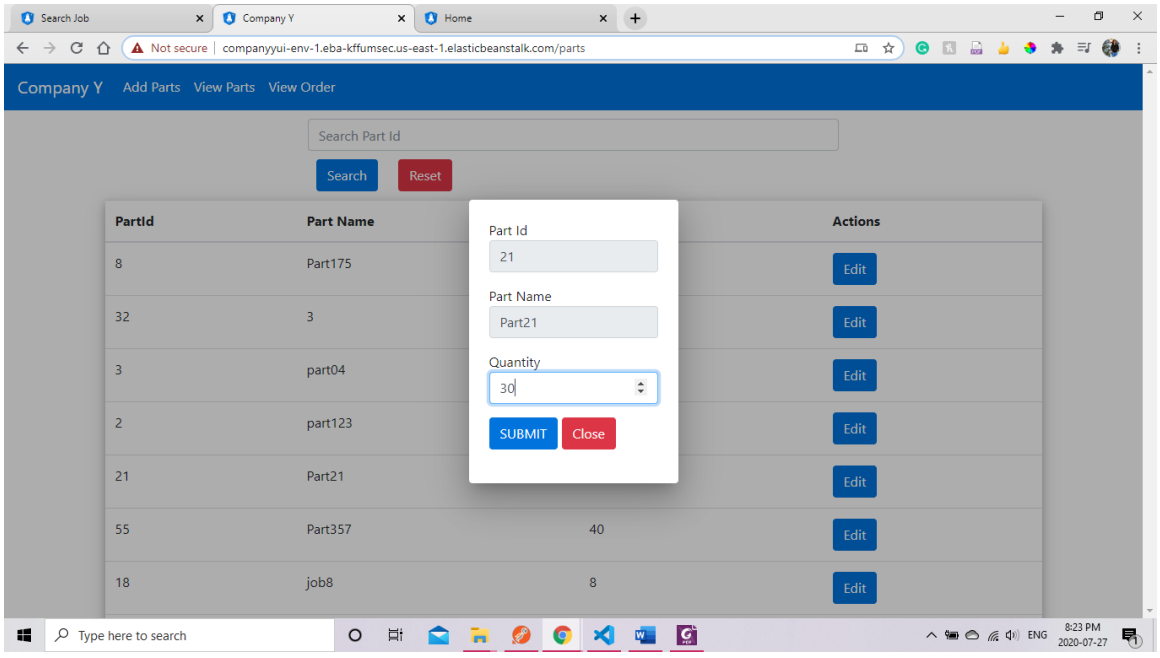


Figure 81: Update quantity of part

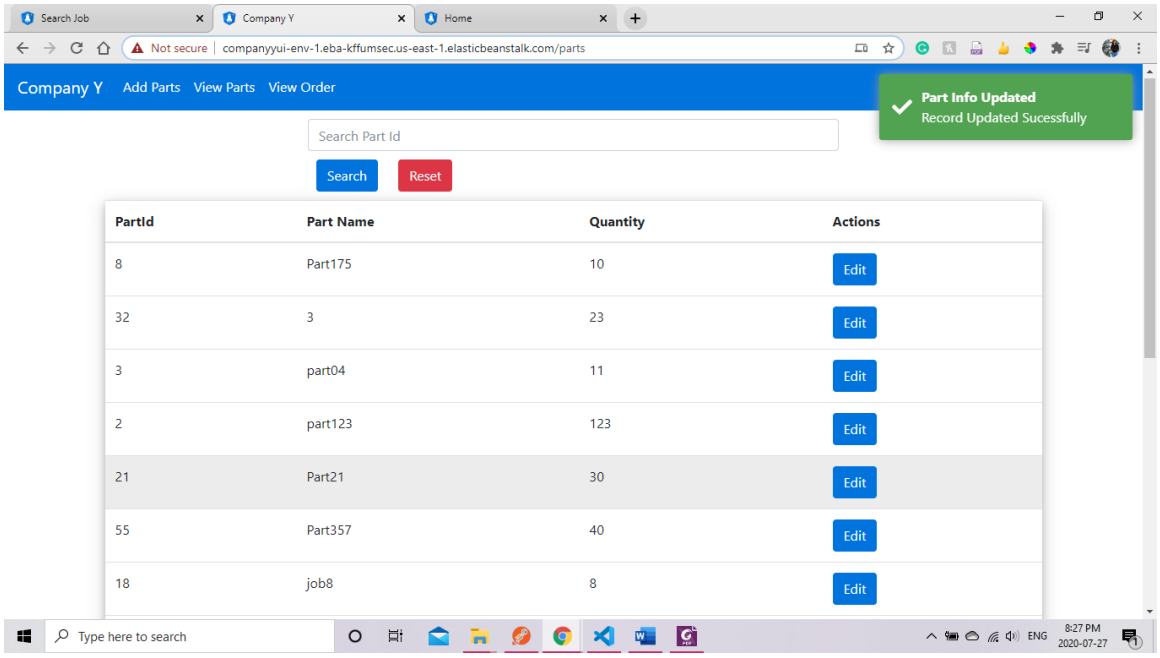


Figure 82: Quantity updated successfully

Test 9: *View all orders* - User can see all the orders from the view order page. User click on the view order button in the header and all the order will be displayed on the page as shown in the figure 83.

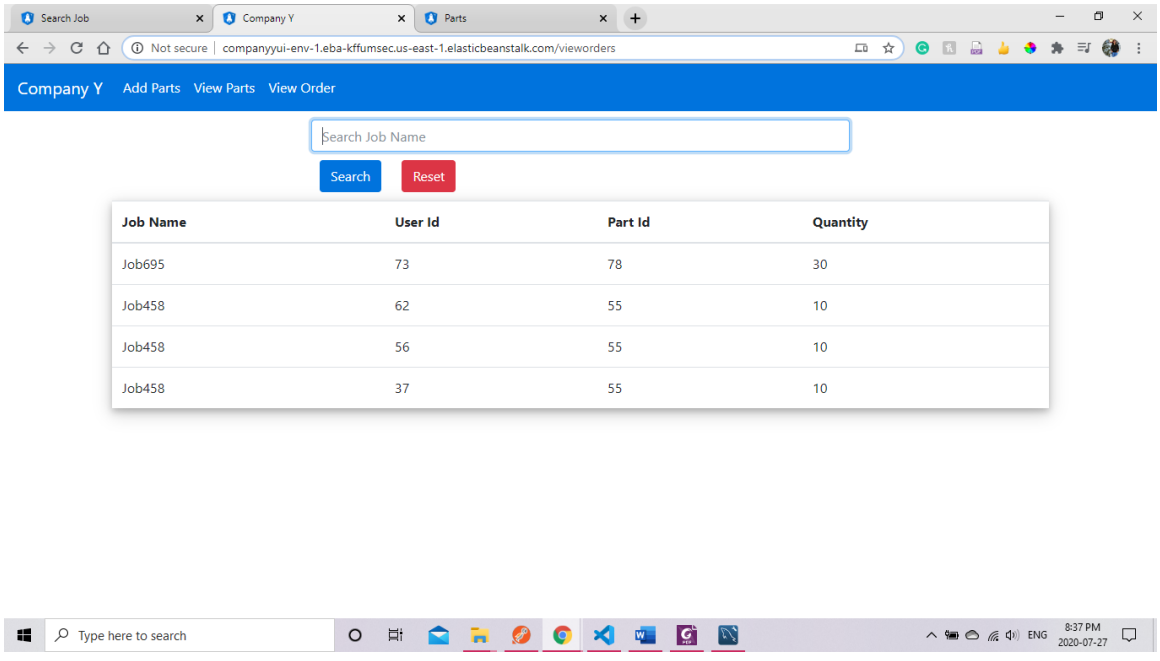


Figure 83: View all orders

Test 10: *Search order* - User can search the order using the jobname. User enter the name of job and click on the search button. Application will display searched order as shown in figure 84.

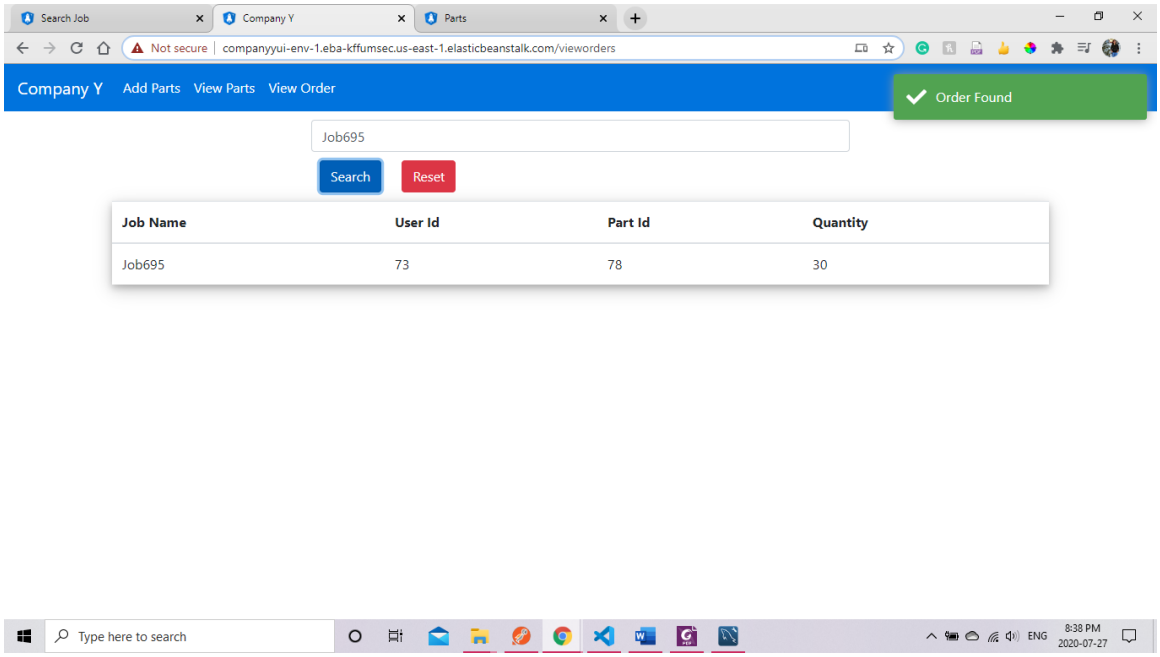


Figure 84: Search Orders

Test 11: *Search unavailable order* - If user enters the jobname which does not exist, then application will show appropriate error message as shown in the figure 85.

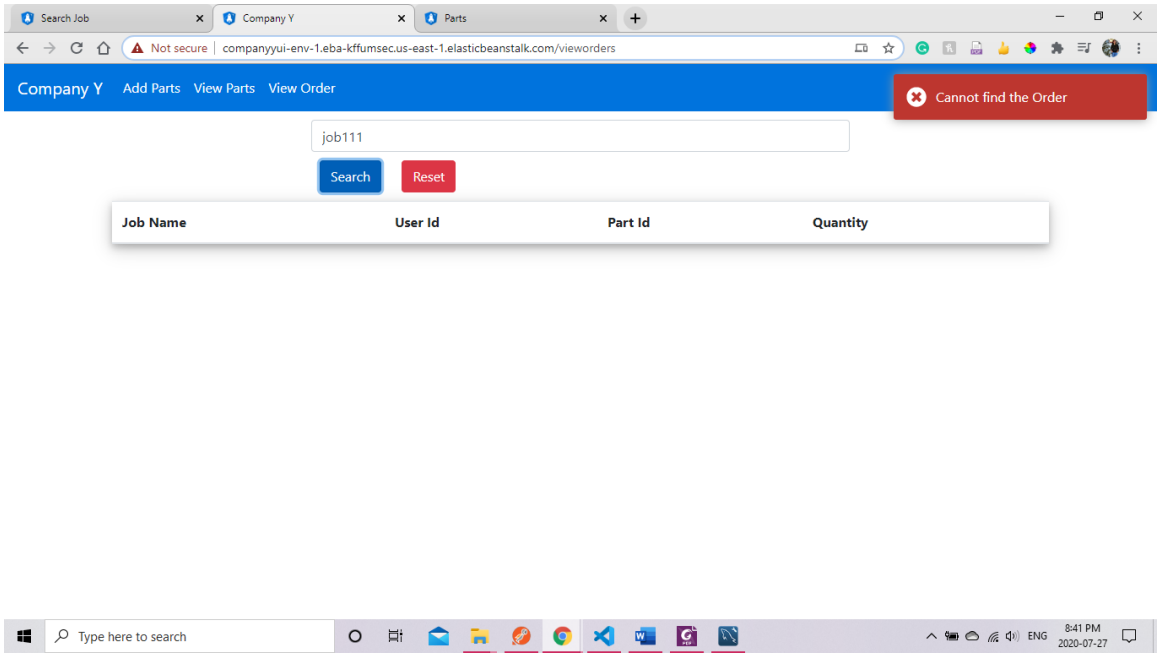


Figure 85: Search unavailable order

## 6.3 Test Cases – Company Z

Following are the list of tests carried out to test the different functionalities provided by the company Z:

Test 1: *Viewing all the jobs* - Application shows all available jobs from the table on the landing page of an application. Figure 86 shows a landing page displaying all available jobs.

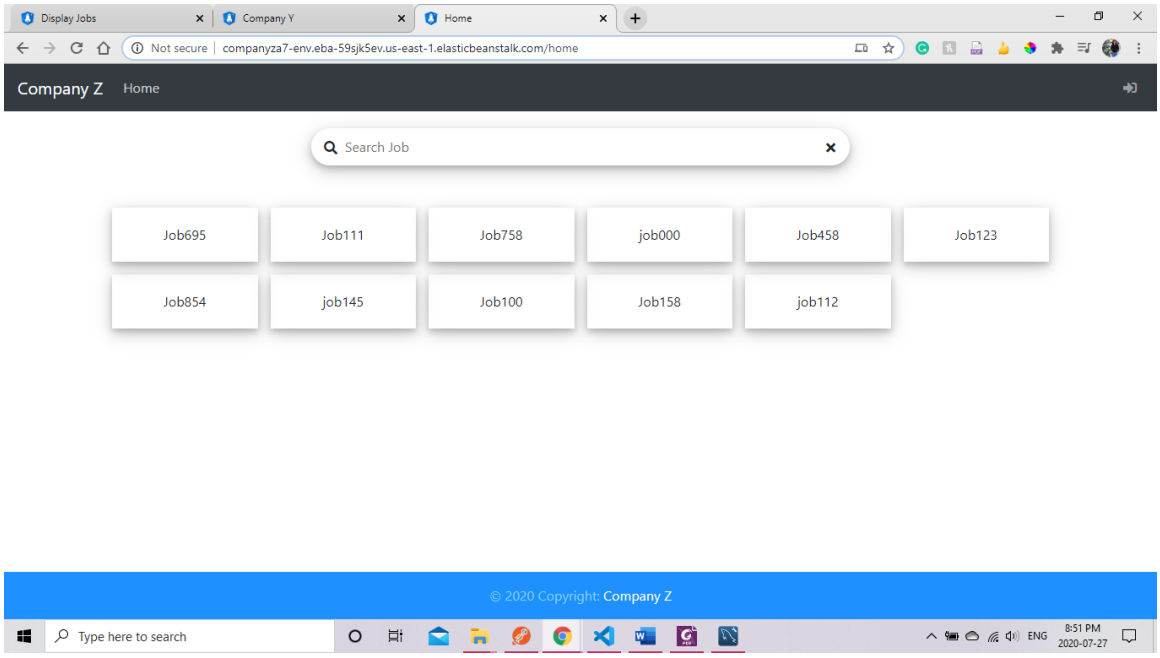


Figure 86: Viewing all jobs

Test 2: *Search unavailable job* - Go to the application landing page and enter the name of the job. If the job does not exist, then application will show appropriate error message as shown in figure 87.

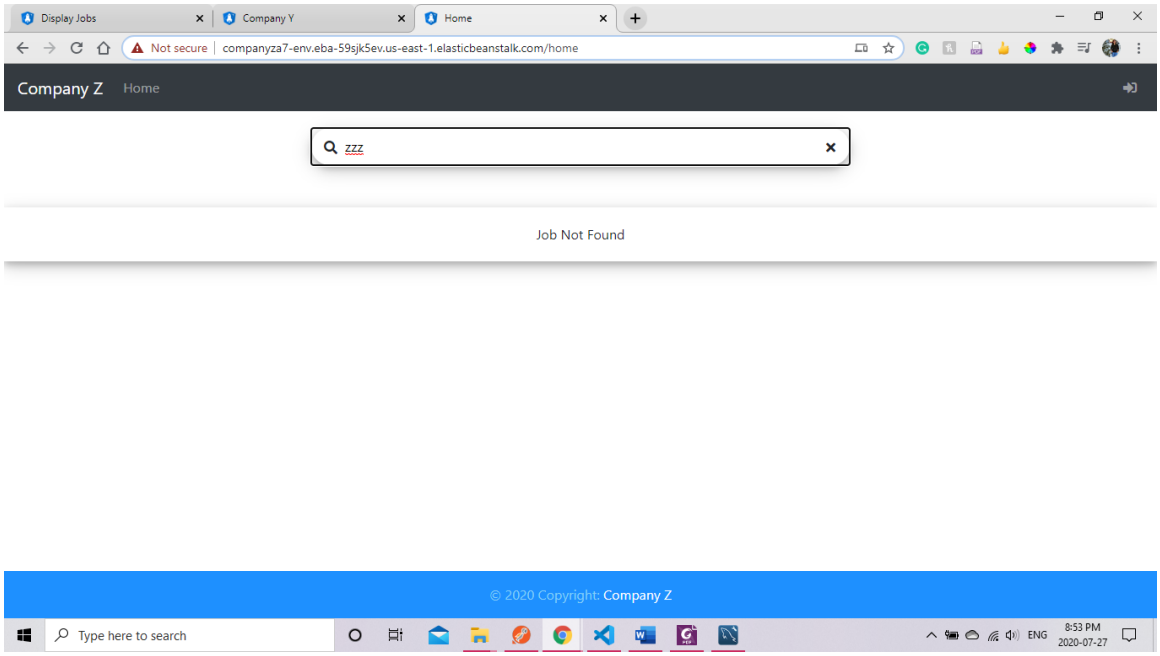


Figure 87: Search unavailable job

Test 3: *Search job* - Go to the application landing page and enter the name of the job which you want to find. Application will display the job if available as shown in figure 88.

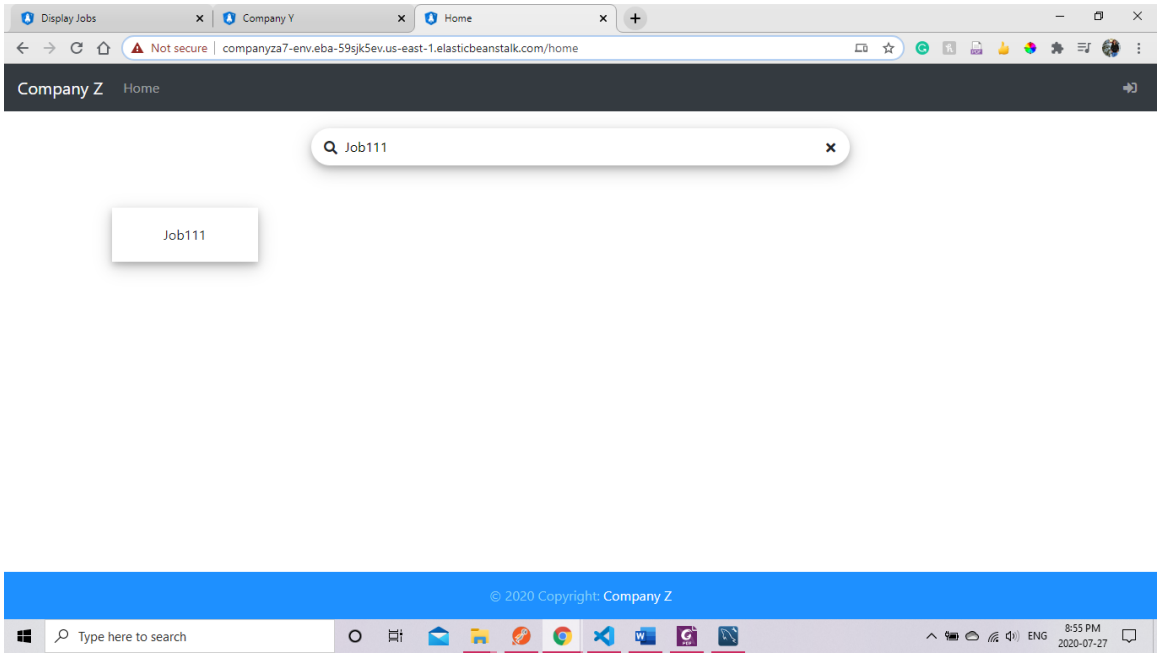


Figure 88: Search Job

Test 4: *View the job* - To view the details of the job you must click on the job name. Application will show you the details of the part and quantity as shown in figure 89.

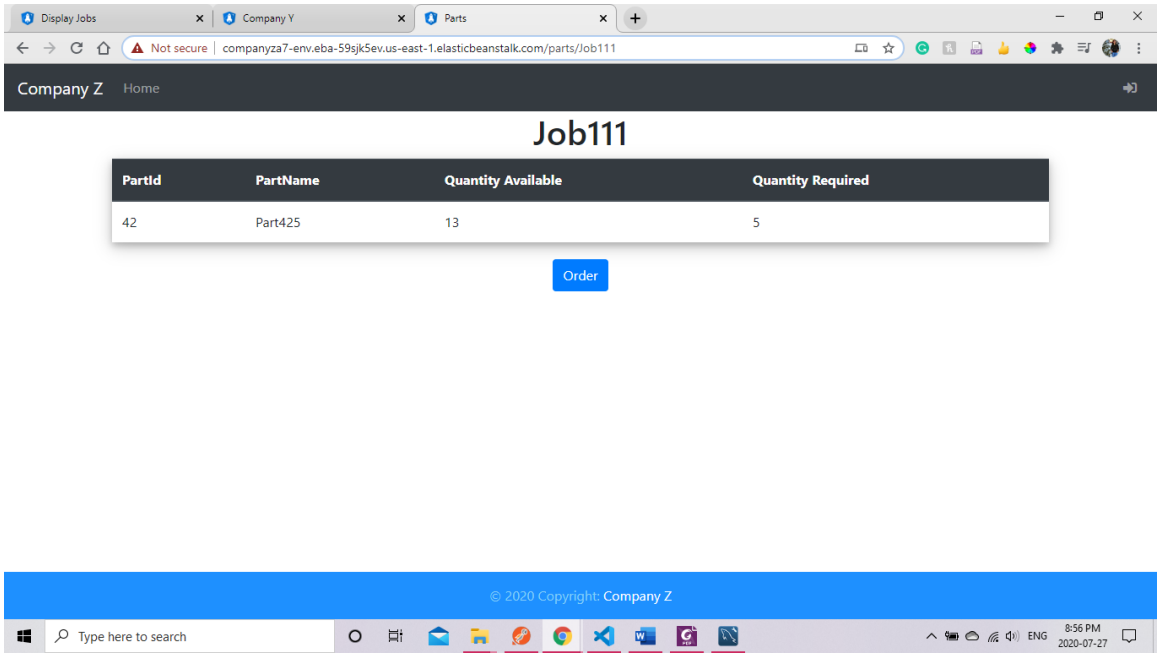


Figure 89: View the Job

Test 5: *Order parts* - To order the parts you need to click on the order button. As user needs to be authenticated first before ordering, application will show login page to the user as shown in figure 90.

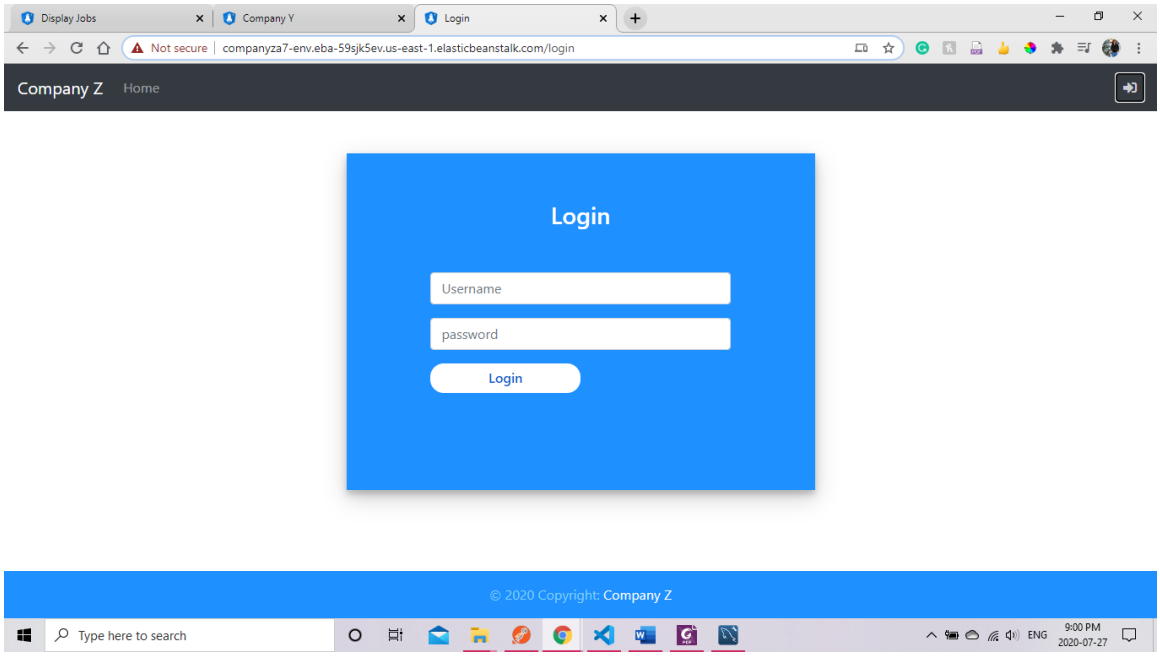


Figure 90: Login Page

Test 6: *Login Failed* - User enter the username and password, click on the login button. Application will check entered credential in the database. If it is not valid then it will show proper error message as shown in figure 91.

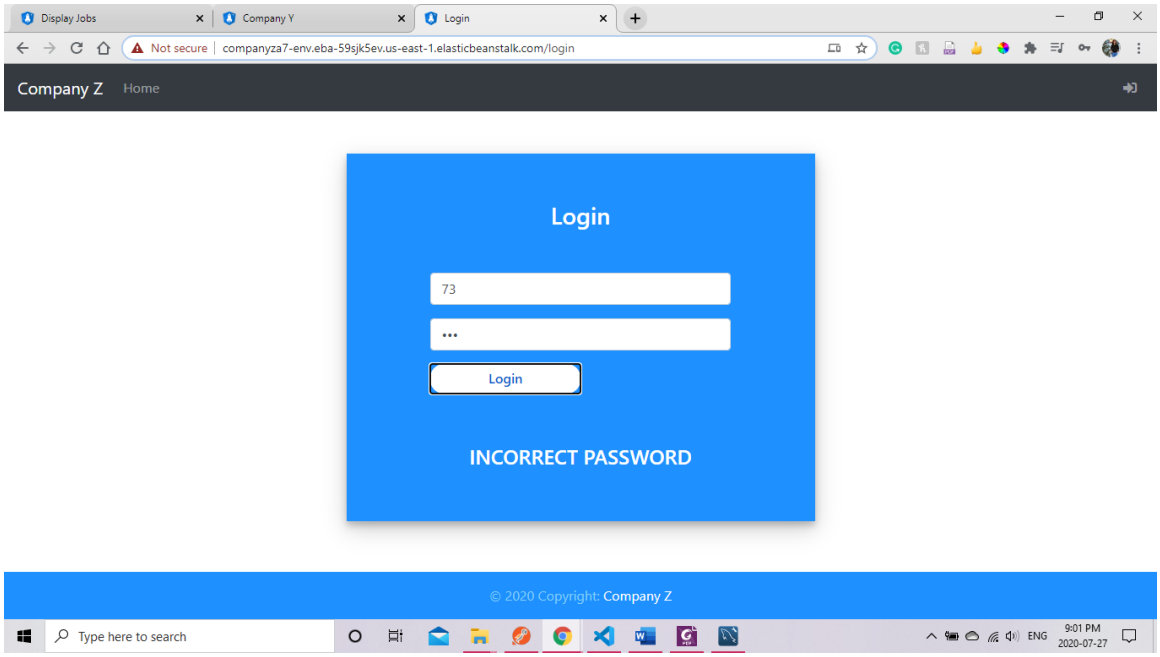


Figure 91: Login failed

Test 7: *Login Success* - User will enter the username and password and click on the login button. Application will check the credential in the database and if the credential is valid then show the landing page where user can order as shown in figure 92.

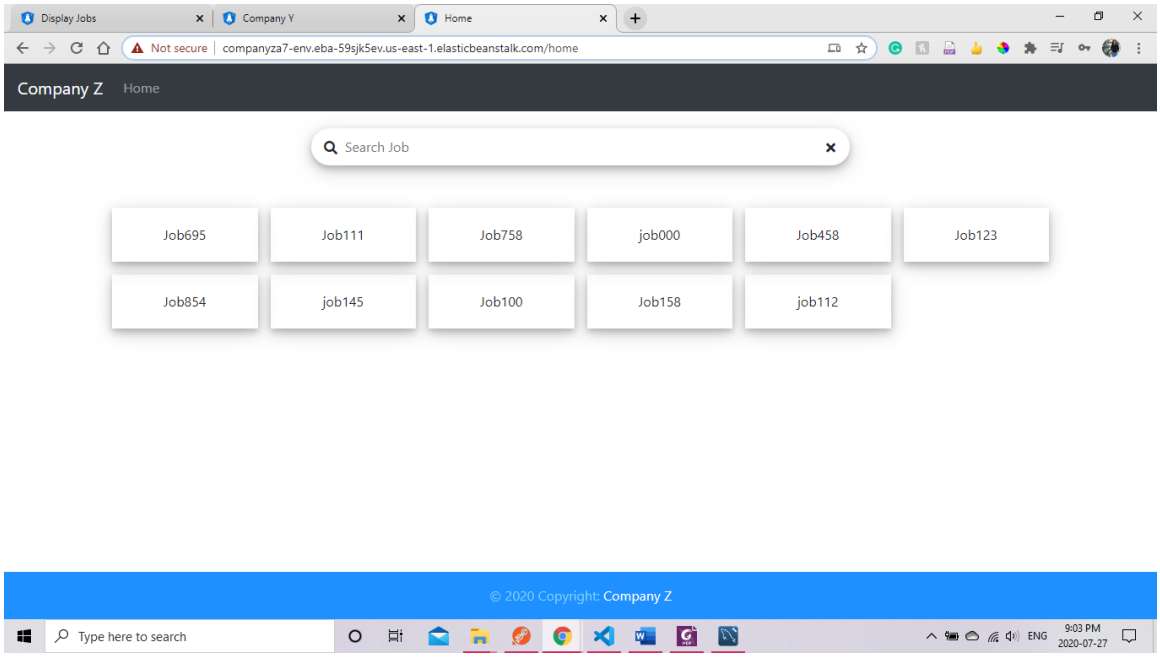


Figure 92: Login Succeed

Test 8: *Order Failed* - User will click on the order button. The required quantity is greater than the available quantity. Application will provide proper error message as shown in figure 93.

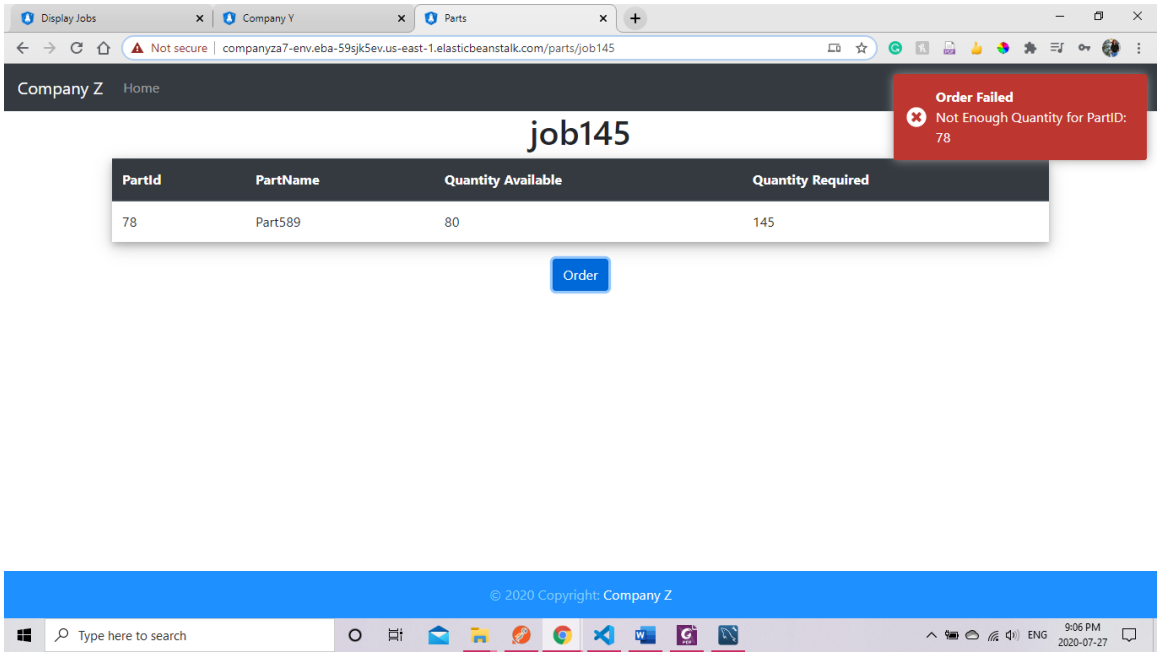


Figure 93: Order failed

Test 9: *Order success* - User clicks on the order when the quantity available is greater than the quantity required then order will be placed successfully then the success message will be shown as shown in the figure 94.

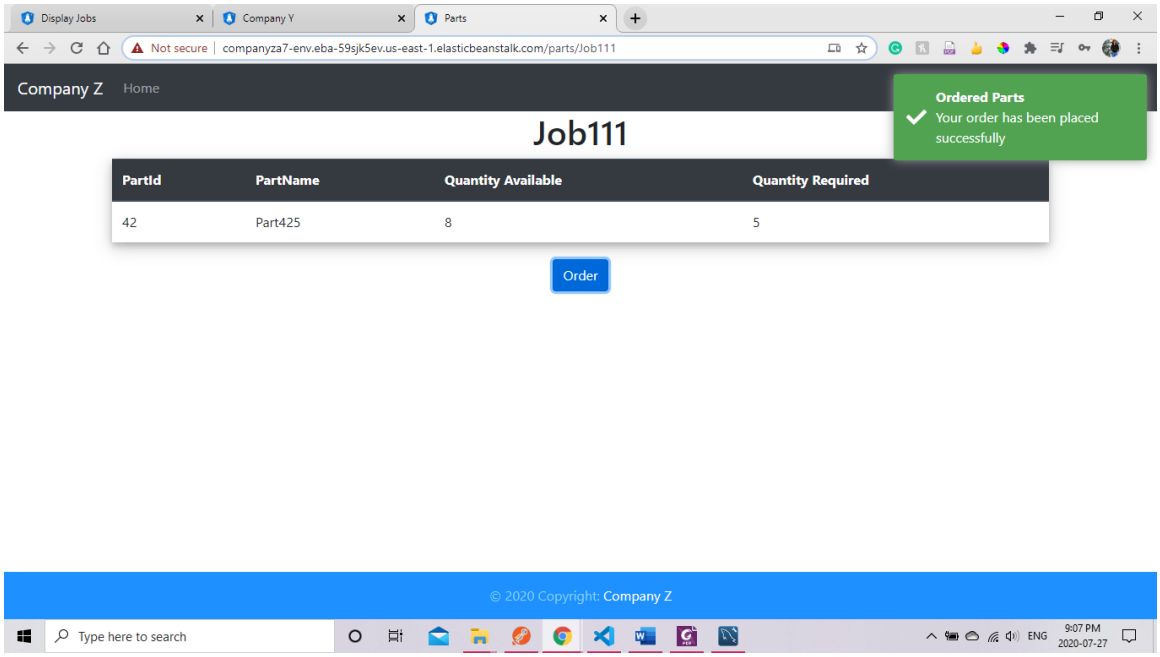


Figure 94: Order succeed

Test 10: *No parts found for job* - When user searched for a job which does not contain any part then the appropriate error message will be shown as shown in figure 95.

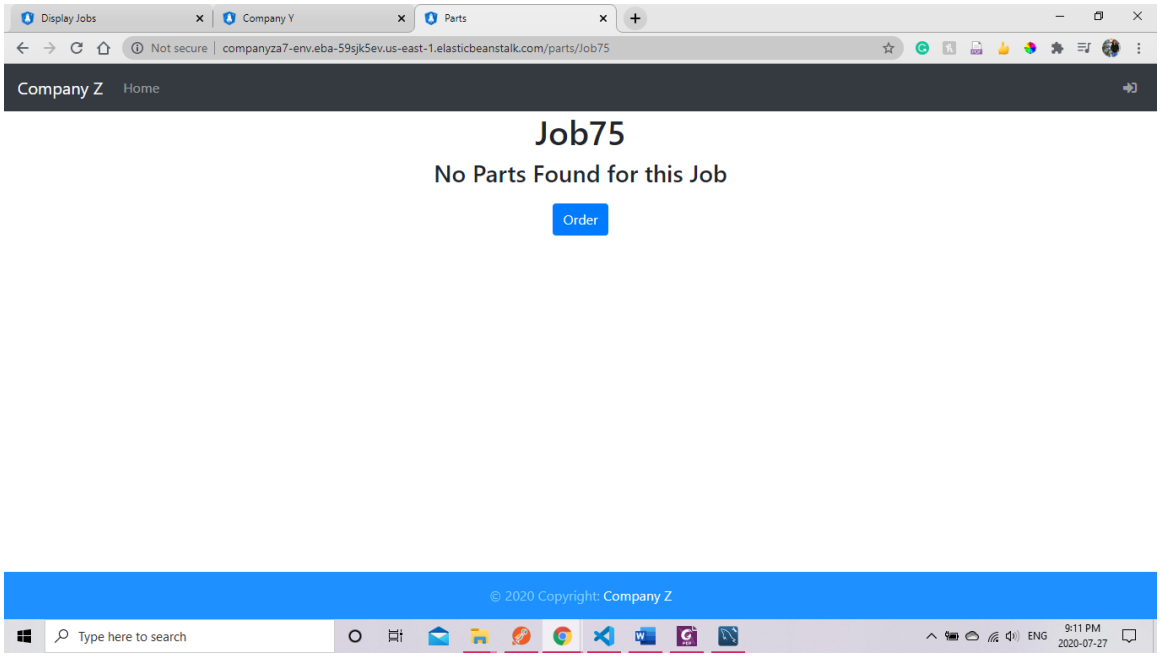


Figure 95: No parts found for job

Test 11: *Order already exist* - Figure 96 shows the job which has already ordered parts. When user will try to order parts for same job then system will show error message as shown in figure 96.

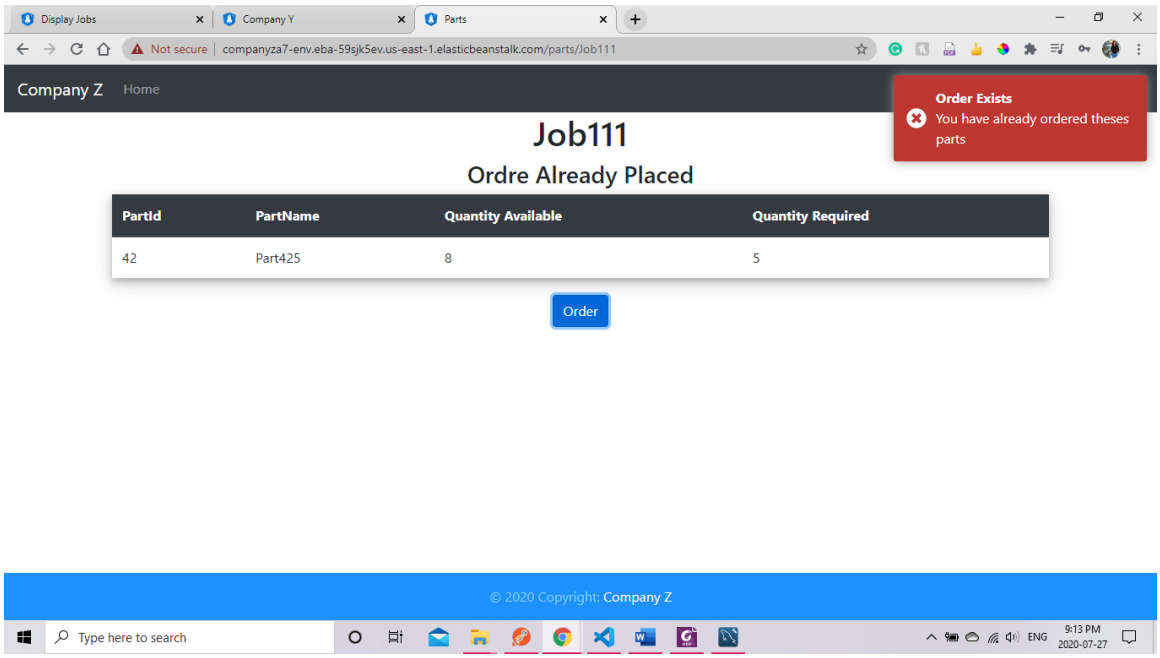


Figure 96: Order already exist

# 7 Deficiencies

The submission does not have any deficiency as it covers all the requirements as per the assignment 7 handout.

- The application for company x provides functionality to add, update, delete and view job information. It also allows to look up the successful orders by providing the job name.
- The application for company y provides functionality to add, update, and view part information. It also allows to look up the successful orders by providing the job name.
- The application for company z provides functionality to see list of job names, search for a job, view the associated part information for a selected job, and place an order for all the parts associated with a job. It also provides an authentication mechanism which is needed before placing an order. The application also stores the search information.
- The front part of all three applications are containerized using docker and deployed as a managed web app on Elastic Beanstalk cloud platform.
- The backend components of companies x and y are implemented with serverless computing using AWS API Gateway, Lambda, and DynamoDB.
- The tables for companies x and y are stored on AWS DynamoDB, whereas tables for company z are hosted on Amazon RDS. Also, the frontend of individual companies only communicates with the tables managed by themselves. For the cross company communication, they use API endpoints exposed by individual companies.

    1. Company X handles two tables: jobs_x and part_orders_x
    2. Company Y handles two tables: parts_y and part_orders_y
    3. Company Z handles three tables: search_z, users_z, and job_parts_z

# 8 Collaboration Tools

The section gives an overview about the various collaboration tools used by our group to successfully carry out the assignment 6. It discusses the usage of Trello [9], Microsoft Teams [11], and GitLab. [10]

## 8.1 Assignment Management using Trello

The section provides a brief overview about the Trello board of group 13, and it discusses various terminologies followed. To have a detailed look of user stories, task breakdown, and story assignment, please head over to the Trello board. Our group has thoroughly utilized Trello to manage the work of entire assignment 7. We have followed a template named Agile Sprint board available on Trello templates.

Figure 97 and 98 show the initial snapshot of our Trello board having main buckets such as Product Backlog, Sprint Backlog, In Progress, Done (Tasks), and Sprint – Complete.

The Product backlog contains the list of user stories envisioned at the start of assignment. As per the team's initial understanding, we divided the work into list of user stories and put them under *product backlog* bucket.

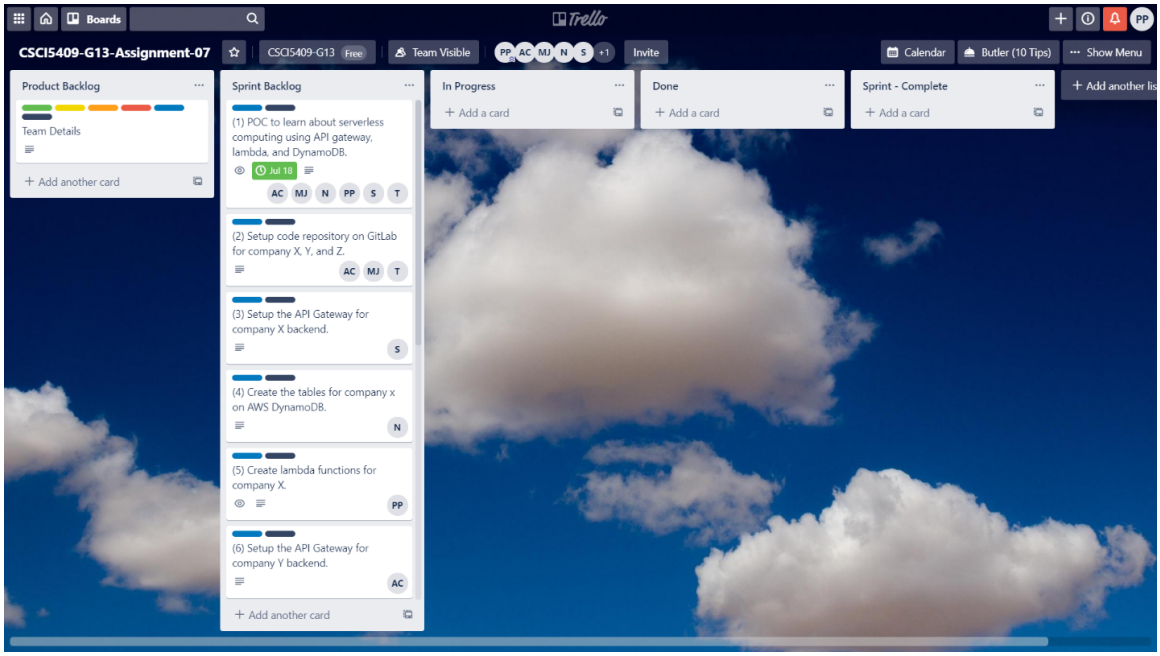


Figure 97: Sprint Backlog - User Stories 1


Figure 98: Sprint Backlog - User Stories 2

During the sprint planning meeting, we dragged all the user stories into *Sprint Backlog* bucket and assigned them to individual team members. The team members further assigned story points, due date, and created sub-tasks for individual user stories and assigned estimated hours to each task.

Team used the *In-Progress* bucket to move individual user story and tasks, as and when they started working on it.

*Done (Tasks)* bucket is used to move all the completed tasks and user stories.

*Sprint – Complete* holds the list of all completed user stories.

Figure 99 shows a snapshot for a user story created to perform POC to learn about serverless computing using API gateway, lambda, and DynamoDB. It includes the story requirements, assigned member, due date, story point, and labels. (I.e., New, User Story) All created user stories follow the similar template.

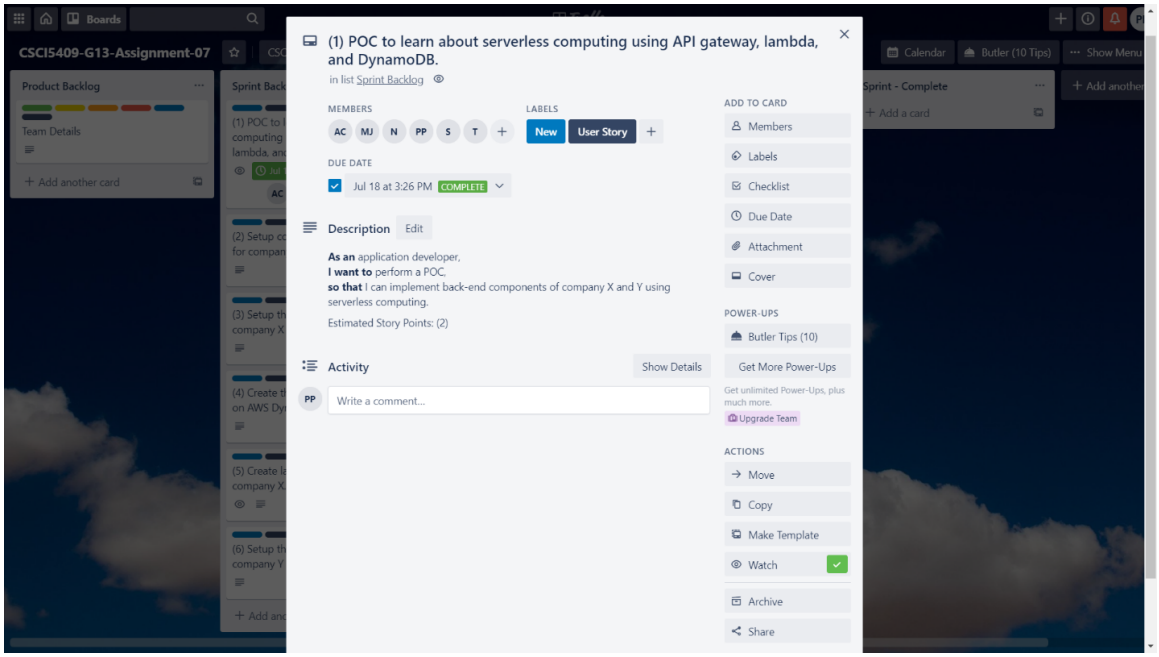


Figure 99: A sample user story

Figure 100 shows the snapshot of our Trello board from July 23, where the POC, API Gateway for company X and DynamoDB database setup user stories are completed. The team has started working on front end for company x, y, z.

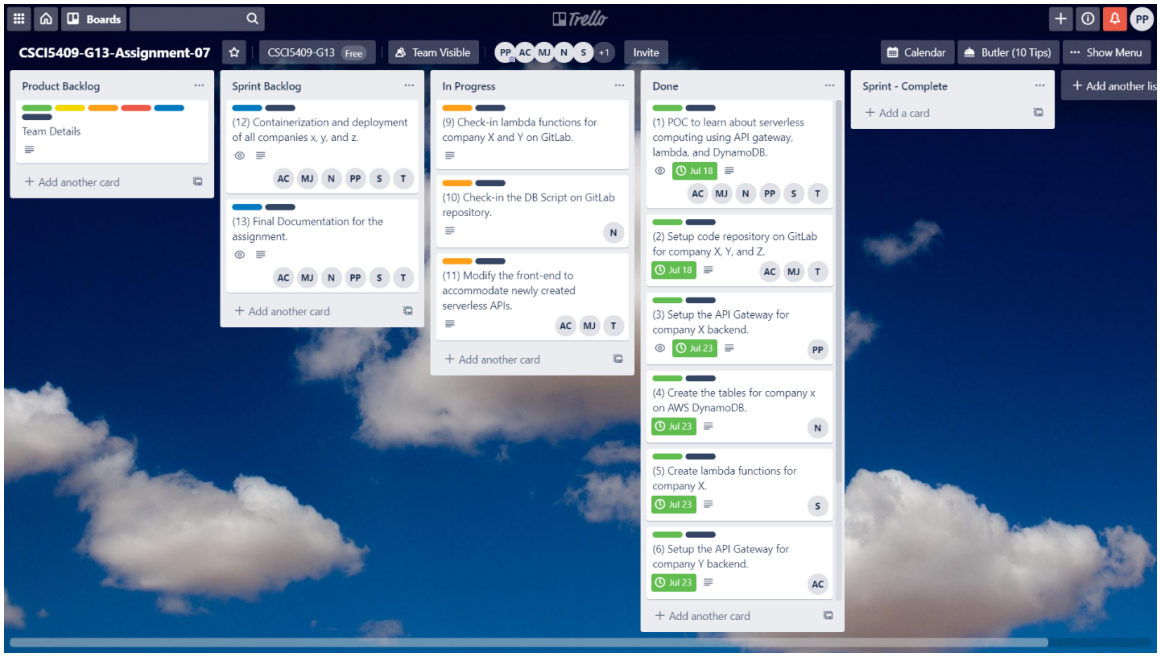


Figure 100: Trello board as of July 23

Figure 101 shows a snapshot of the Trello board at the time of writing this document, where check-in lambda function and the final documentation related user story is pending. The user story will be marked as done once we upload the document. All the completed tasks are moved under *Done (Tasks)* bucket, whereas, the *Sprint – Complete* hosts all the completed user stories.

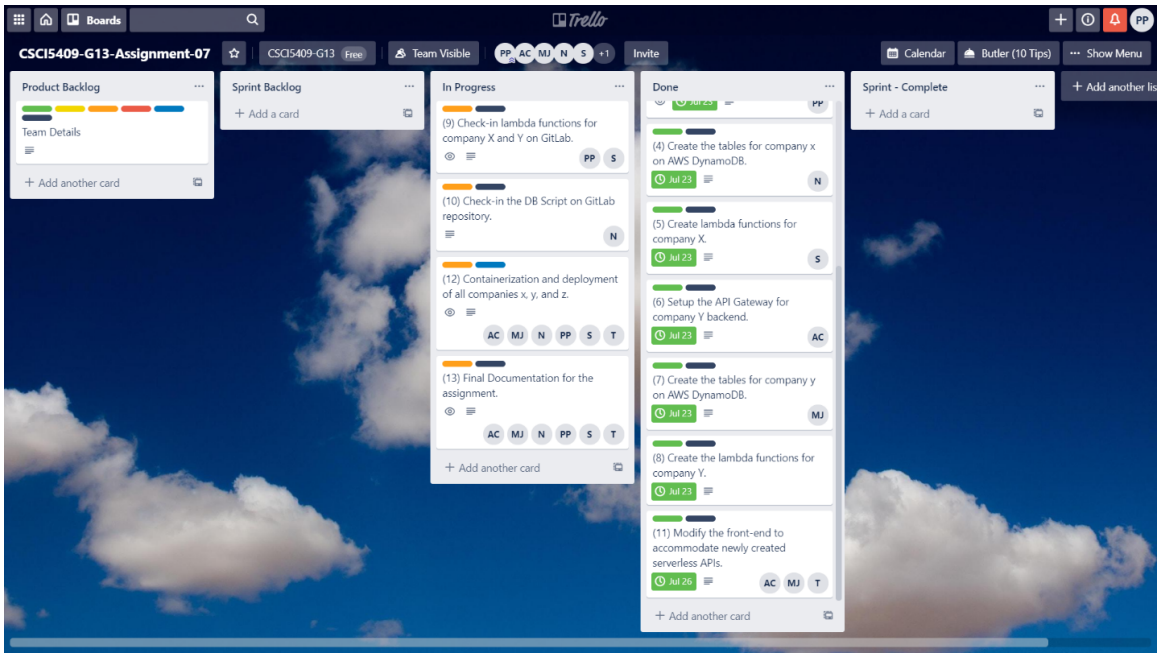


Figure 101: Trello board as of July 27

## 8.2 Microsoft Teams

Our group has primarily used the Microsoft Teams for sharing the information to all group members and storing the documents.

The information sharing examples would be sharing the database details and providing list of document deliverable.

It hosts various team documents such as, minutes of meetings, Trello board planning document, assignment planning document, and assignment document outline.

At the end of assignment, our group will also host the below set of final documents under a folder named Assignment 7 – final documents for individual team members, Home TA, and instructor.

1. A7-G13-Charter.pdf - Team Charter
2. A7-G13-CompXYZ.pdf – Main document

3. A7-G13-Minutes-of-Meetings.pdf – All minutes of meetings in a combined document
4. A7_TasksForm_G13.pdf - Showcasing the team and work distribution
5. A7-G13-Basic-Workflow-Tests.pdf - Screenshots showing successful workflow and all endpoints

Our team did not use Microsoft Teams for meetings as all group members live in the same apartment building, and therefore, the meetings were not conducted virtually.

### 8.3 FCS GitLab – Source code repository

The source code repository management has been done using the FCS GitLab. As our group has been further divided into two sub-teams responsible for handling serverless computing of individual companies x and y, three separate code repositories are created to handle the frontend and backend of individual companies.

Source Code Links:

Company X [14] Company Y [15] Company Z [16]

We did not need to create separate branches due to the individual repositories. Only two members were working on individual source code repository.

Please head over to the individual source code repositories for detailed description about each commits and contribution of individual team members.

The source code has lesser commits as only front-end part was slightly modified to accommodate changes of serverless functions. Majority of work was done over AWS accounts to create serveless backend components for companies X and Y.

[14] https://git.cs.dal.ca/chaudhari/company-x-a7
[15] https://git.cs.dal.ca/mjani/company-y-a7
[16] https://git.cs.dal.ca/tprajapati/company-z-a7

# 9 Minutes of Meeting

The section summarises the different meeting held by our group to successfully carry out the assignment 7.

Note: All the team members have attended all team meetings.

Tables 2, 3, 4, and 5 provide a high-level summary for each meeting held by our group. Please refer to a separate document *A7-G13-Minutes-of-Meetings.pdf* that is a combined document containing details of each meeting.

| Sr. No. | Meeting Date/Time | Major Items discussed |
|---|---|---|
| 1 | July 15, 2020 08:00 PM | 1. High-level requirements of the assignment.<br>2. Documented unknown areas.<br>3. Discussed about basic concepts of serverless computing.<br>4. Decided to use serverless computing for backend components for X and Y companies.<br>5. Assigned Team roles.<br>6. Went through the Team Charter requirements.<br>7. Setup a Trello board based on our understanding |

Table 2: July 15, 2020 08:00 PM - MOM

| Sr. No. | Meeting Date/Time | Major Items discussed |
|---|---|---|
| 1 | July 19, 2020 08:00 PM | 1. Team Discussed their individual experiences of creating POC using API gateway, lambda, and DynamoDB.<br>2. Team performed the backlog grooming using created Trello board.<br>3. Discussed individual stories and assigned them to team members.<br>4. Team members to create sub tasks for the user stories, assign estimated hours and due dates. Also, start working on the respective stories |

Table 3: July 19, 2020 08:00 PM - MOM

| Sr. No. | Meeting Date/Time | Major Items discussed |
|---|---|---|
| 1 | July 24, 2020 08:00 PM | 1. Discussed the serverless components created for company x and y.<br>2. Discussed the status of individual's work. (I.e., what is completed, what is pending, any risk items)<br>3. Listed out endpoints that needs to be changed in frontend for company x and y respectively.<br>4. Discussed and distributed the containerization and deployment work to individuals. Everyone should deploy their respective part frontend as a managed containerized application on Amazon Elastic Beanstalk.<br>5. Discussed about the document deliverables. |

Table 4: July 24, 2020 08:00 PM - MOM

| Sr. No. | Meeting Date/Time | Major Items discussed |
|---|---|---|
| 1 | July 27, 2020 11:30 AM | 1. Discussed pending changes with regards to the functionalities.<br>2. Discussed the outline for the final report.<br>3. Reviewed the task distribution.<br>4. Individual team members filled up the high-level points of their contribution. |

Table 5: July 27, 2020 11:30 AM - MOM

# References


[1] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava, "Serverless computing for container-based architectures", https://www.sciencedirect.com/science/article/abs/pii/S0167739X17316485, Jan 2018.

[2] "Amazon api gateway", https://aws.amazon.com/api-gateway/, 1989.

[3] "Aws lambda", https://aws.amazon.com/lambda/, 1989.

[4] Derek Rangel, "Dynamodb: everything you need to know about amazon web service's nosql database", https://aws.amazon.com/dynamodb/, 2015.

[5] "Docker - empowering app development for developers", https://www.docker.com/.

[6] "docker hub", https://hub.docker.com/.

[7] "aws elastic beanstalk - deploy web applications", https://aws.amazon.com/elasticbeanstalk/.

[8] N. Prajapati T. Prajapati P. Parmar S. Shah A. Chaudhari, M. Jani, "Csci 5409 - assignment 6", Jul 2020.

[9] "Trello - trello lets you work more collaboratively and get more done", https://trello.com/en-GB.

[10] "The first single application for the entire devops lifecycle", https://about.gitlab.com/.

[11] "Microsoft teams - free chat, video calling, collaboration", https://www.microsoft.com/en-ca/microsoft-365/microsoft-teams/free.

[12] "Online diagram software visual solution", https://www.lucidchart.com/pages/.

[13] "Rds: understanding animal research in medicine", https://aws.amazon.com/rds/, 2007.

[14] "Aws management console", https://aws.amazon.com/console/, 2015.

[15] "ngx-deploy-docker", https://www.npmjs.com/package/ngx-deploy-docker.

# Appendices

## A Database Scripts

1. Table: search_z

```
CREATE TABLE `search_z` (
`searchId` int(11) NOT NULL AUTO_INCREMENT,
`jobName` varchar(50) NOT NULL,
`date` date DEFAULT NULL,
`time` time DEFAULT NULL,
PRIMARY KEY (`searchId`),
KEY `jobName` (`jobName`)
)
```

2. Table: job_parts_z

```
CREATE TABLE `job_parts_z` (
`partId` int(11) NOT NULL,
`jobName` varchar(50) NOT NULL,
`userId` varchar(50) NOT NULL,
`qty` int(11) NOT NULL,
`date` date DEFAULT NULL,
`time` time DEFAULT NULL,
`result` enum('success','failure') NOT NULL,
PRIMARY KEY (`partId`,`jobName`,`userId`,`result`),
KEY `jobName` (`jobName`),
KEY `userId` (`userId`)
)
```

3. Table: users_z

```
CREATE TABLE `users_z` (
`userId` varchar(50) NOT NULL,
`password` varchar(100) NOT NULL,
`firstName` varchar(50) NOT NULL,
`lastName` varchar(50) NOT NULL,
```

```
`emailAddress` varchar(70) NOT NULL,
PRIMARY KEY (`userId`)
)
```

# B Front-end Scripts

1. angular.json for deployment

```
"deploy": {
   "builder": "ngx-deploy-docker:deploy",
   "options": {
     "account": "malavjani"
   }
 }
```

2. Dockerfile to build image

```
FROM nginx:alpine
LABEL version="1.0.0"

COPY nginx.conf /etc/nginx/nginx.conf

WORKDIR /usr/share/nginx/html
COPY dist/company-y-ui-a7 .

EXPOSE 8080
```

3. Dockerrun.aws.json to deploy on AWS EB environment

```
{
    "AWSEBDockerrunVersion": "1",
    "Image": {
        "Name": "malavjani/company-y-ui-a7"
    },
    "Ports": [
        {
```

```
                "ContainerPort": "8080"
            }
        ]
    }
```