International
Institute of Information
Technology Bangalore

Final Project - Gameroom: A team building web application

CS 816-Software Production Engineering

Under Prof B Thangaraju
Dasari Surya Sai Venkatesh (IMT2017012)
George T. Abraham (IMT2017019)
Kaushal Mittal (IMT2017024)

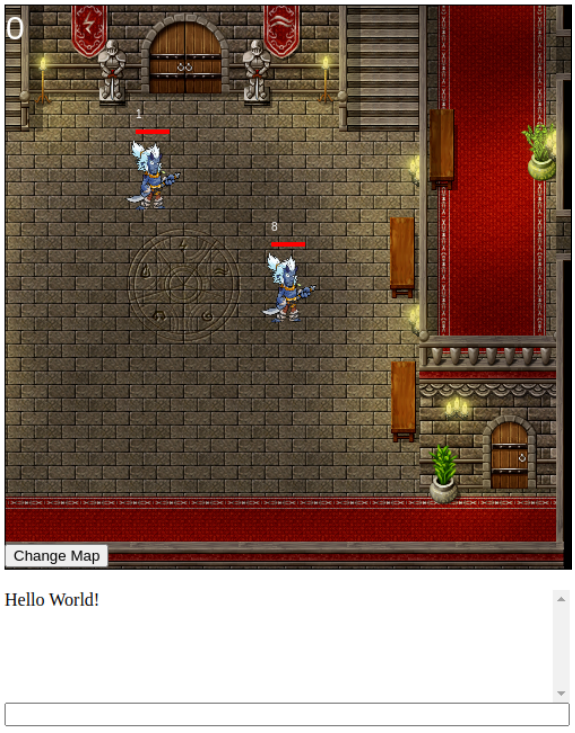
Table of Contents

1. Abstract

For any company to perform efficiently, the teams and its individual members must be working efficiently and this requires teamwork skills and proper communication between the members. Keeping this in mind, most companies conduct team building activities and games on a regular basis to build and maintain this skill. However, in remote times these activities cannot be conducted at a physical location.This brings a need for a web application that allows to conduct these team building activities remotely.

So, we propose Gameroom, a web application that provides simple team building games and lays emphasis on the communication aspect of team building.This implemented using “client-server” architecture.It primarily uses Javascript for both client, server side and Node.js in the server. Communication between the client and server was primarily handled using the javascript library socket.io.

In the following sections,we will get into the details of functionalities of the application itself followed by the DevOps pipeline starting from building to the continuous deployment and the logging.



Github Link to project
https://github.com/vazravasu/Treasure_Hunt

Link to the deployed game
http://52.255.169.252:2000/ (or)
http://localhost:2000

DockerHub Link
https://hub.docker.com/repository/docker/vazravasu/treasurehunt

(Image:GameRoom client side game screen)

2. Introduction:

2.1 Overview

While there are many websites that allow for team building activities, they are lacking in bringing all the players together as a team. Gameroom differs from them due to the reason that the game/puzzle offered by it can only be solved with effective communication between team members.The players can explore the map on their own and on finding necessary clues/keys can come together,discuss and try to solve the puzzle(In the game Treasure Hunt ,players must find a series of objects that together are used to open a treasure chest).

Disclaimer: Currently, Gameroom only runs a single game - TreasureHunt. Due to time constraints, most of the basic required functionalities such as chat functionality, inventories (item management for items) and some basic combat have been implemented but a story/puzzle could not be designed.

2.2 Features:

Gameroom supports the following features:

- Player login (using Mongoose and MongoDB)
- Realtime multiplayer game features (coded in Javascript and NodeJS)
- Realtime chat support (using socket.io)
- Game session logs (using log4js)
- Visualisation of Logs using ELK

2.3 Integration and Deployment Approach:

Since the application and its features/functionalities are very modular, they are subject to continuous changes with respect to feature additions. Therefore it becomes necessary to have a mechanism/pipeline that automates the steps starting from building the artifacts to deploying it on a server. Hence,we use the DevOps approach and DevOps tools to achieve the same.

DevOps tools allow us to perform code version control, testing and building systems, configuration management and deploying the application in an automated way.

3.System Configuration

3.1. Operating system

- Ubuntu 18.04.04
- Bionic Beaver.
- v5.3 based Linux kernel

3.2. CPU and RAM

1 Core CPU, 1GB RAM on Azure (Infrastructure as a service)

3.3. Programming Languages, Frameworks and Libraries

- JavaScript and NodeJS
- socket.io library
- HTML
- Canvas (for rendering game on the HTML page)

3.4. Database

- MongoDB Online
- Mongoose Library for MongoDB.

3.5. Dependency resolvers

npm (is a package manager for the JavaScript programming language)

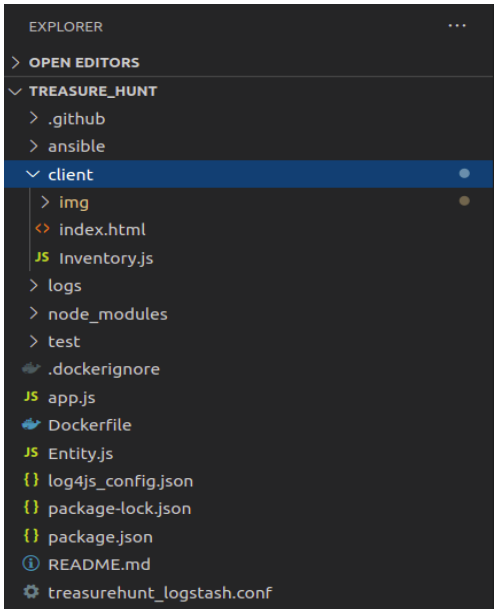
3.6. DevOps Tools

● Source Control Management - GitHub
● Continuous Integration - Github Actions
● Containerization - Docker
● Continuous deployment - Ansible
● Continuous monitoring - ELK Stack (Elastic Search, Logstash, Kibana)

4.Application Overview

Here we take a brief look at the file structure and basic principles behind the application.

4.1 File Structure:



(Image 2: file structure of the application)

- Server Side
  - Entity.js
  - App.js
- Client Side
  - /client
- Package and Config Files
  - ansible/, node modules/, ...

4.1.1 Server Side:

Here, “app.js” serves as the entrypoint for the application to run on the server.It is responsible for calling necessary dependencies such as mongoose (for online user database), socket.io(for player chat),log4js (for logging) and express (for file communication to client). It also manages the players and relevant function calls,objects from “Entity.js”. Entity.js is the actual file that contains Player classes and handles communication from client to server.Both app.js and Entity.js are server side code.

4.1.2 Client side:

The folder “client” contains an index.html file which is the .html page a client sees on connecting to the website.It also contains Inventory.js which handles player inventory (we can store keys,potions etc.). Additionally,it also contains a folder called “img” which contains .png images of the player,map etc. Whenever,client wants to render these images,they are sent from server to client.

4.1.3 Package Dependencies and Config files:

- "package.json" is a .json file that specifies the package dependencies,entry point and other relevant details.On running the command npm install, it downloads all the relevant packages and puts it into the folder called "node_modules".
- "log4js_config.json" specifies the configuration and syntax of the logs created. It appends the logs to both the console and the file "./logs/TreasureHunt.log"

4.2 Code Workflow Outline:

Now that we have an idea of the contents of each file,we can take a brief look at how the code functions.An in depth explanation will be done in Section 6.

The server section of the code is concerned with the game logic and making sure that it maintains an updated list of player positions,inventory items and maintaining scores etc. When a player/user connects to the website,it makes a request to the root directory and the server sends the ./client/index.html page to the user.The user opens this html page,gets authenticated and starts the game.

The backbone of the multiplayer game is the "socket.io" package that allows a user to emit(send) a message or receive a message.This package is used from both the client and server side to pass necessary information.

```
socket.emit('addToChat','The player ' + data.username + ' is not online.');
```

socket.emit has 2 parameters, the type of the message and the actual data

```
socket.on('sendMsgToServer',function(data){
       for(var i in SOCKET_LIST){
           SOCKET_LIST[i].emit('addToChat',player.username + ': ' + data);
       }
 };
```

Socket.on takes 2 parameters, the type of message and function to apply on receiving the data

Whenever a client performs an action/input ,a message is sent via socket to the server.The server which maintains all player info receives this message and runs in game logic to update position,health,scores etc.On having an updated array of complete player information ,this array is sent from server back to client so that client can render the game on the .html page using canvas.

5.Software Development Life Cycle

5.1 Installations

Our application primarily uses Javascript,NodeJS and npm(Node package Manager).

To install NodeJS,run the following command:

```
$ sudo apt install nodejs
```

To install npm(Node Package Manager):

```
$ sudo apt install npm
```

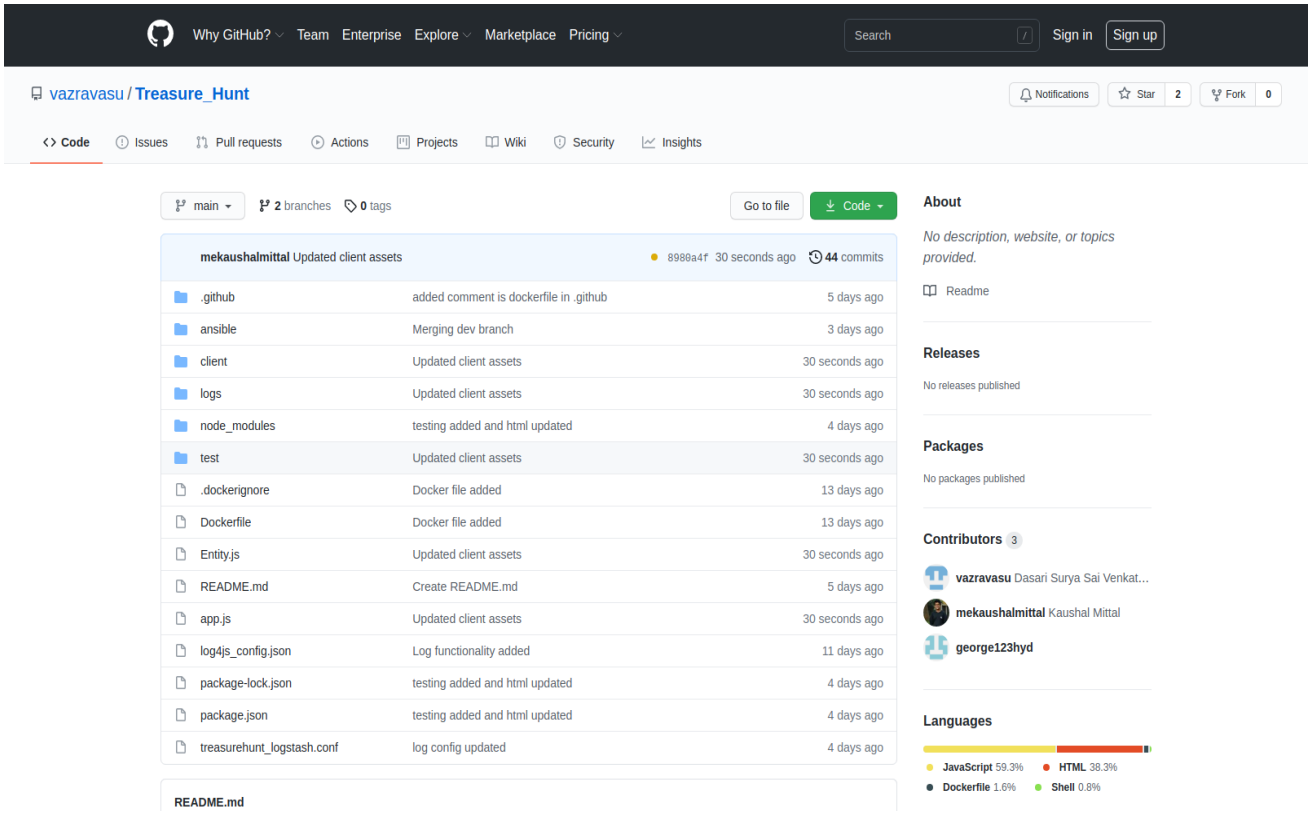
Other dependencies such as socket.io, mongoose, log4js etc. are mentioned in the package.json file.

```
{
 "name": "something",
 "version": "1.0.0",
 "main": "app.js",
 "dependencies": {
   "express": "^4.17.1",
   "log4js": "^6.3.0",
   "mongoose": "^5.12.7",
   "socket.io": "^4.0.1"
 },
 "scripts": {
   "start": "node app.js",
   "test": "mocha || true"
 },
 "engines": {
   "node": "10.19.0"
 },
 "author": "",
 "license": "ISC",
 "description": "",
 "devDependencies": {
   "chai": "^4.3.4",
   "mocha": "^8.4.0"
 }
}
```

On running the command : "**npm install**" ,all the required dependencies are added to the folder node_modules.

5.2 Source Control Management (Git)

In our application,since features were added more frequently, a Source Code Management (SCM) tool was absolutely necessary. For our project we have used GitHub as the Source Control Management Tool.



Below is a list of the few basic commands for Github :

- **git clone <URL>** This command clones the entire repository from the git url
- **git pull** Update local version of code to the one with the remote repository.
- **git add <files>** To add changes to staging area from the working directory
- **git commit -m "commit message"** This command is used to commit the changes with a message giving the information about that commit.
- **git push** To push latest commit to remote repository

5.3 Testing:

For testing the code,we have used Mocha Chai .Mocha and Chai are used together generally for unit testing of Javascript code and are Javascript Frameworks.

Mocha calls functions that are executed in a specified order and is used to describe test suites and test batches. Along with mocha,we also use Chai which is an assertion library. Assertion library is used to compare expected value to the actual result of the functions.
To run the tests run the command : npm test

```
const assert = require('chai').assert
const myent  = require('../Entity')
const myapp  = require('../app')

//-----------------Check if string-------------------
describe("Entity Test ", function(){
   it('Testing entity class', function(){
       let res = myent();
       assert.typeOf(res, 'string');
   });
});

// ----------------Check if = MY Class-----------------
describe("App Test ", function(){
   it('Testing App class', function(){
       let res = myapp();
       assert.equal(res, 'MY Class');
   });
});
```

(Image:Unit test code using mocha and chai)

```
Entity Test
  1) Testing Entity class

App Test
  ✓ Testing App class

App Test
  2) Testing App class


2 passing (8ms)
2 failing

1) Entity Test
     Testing Entity class:

    AssertionError: expected 'Entity Class' to equal 'MY Class'
    + expected - actual

    -Entity Class
    +MY Class
```

(Image: Unit testing results.Tells how many test cases passed and failed)

Disclaimer: Since our code heavily relies on message triggered lambda functions and complicated data structures as parameters to these functions, we have applied testing on Dummy functions present within the code.

5.4 CI/CD Pipeline(GitHub actions)

Github Actions is used for pipelining the complete project step by step. Push or pull requests trigger our github Actions but only for the main branch.

We can also use work_flow dispatch to run this workflow manually from the Actions tab. A workflow run is made up of one or more jobs that can run sequentially or in parallel. The workflow contains 2 jobs called "build" and “deploy”.

```
name: CI
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

Build steps

- Login to docker hub
- Build a docker image using Dockerfile
- Push the built docker image to Docker Hub

```
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: Login to Docker Hub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKER_HUB_USERNAME }}
          password: ${{ secrets.DOCKER_HUB_ACCESS_TOKEN }}
      ...
      - name: Build and push
        uses: docker/build-push-action@v2
        with:
          file: ./Dockerfile
          push: true
          tags: ${{ secrets.DOCKER_HUB_USERNAME }}/treasurehunt:latest
      - name: Image digest
        run: echo ${{ steps.docker_build.outputs.digest }}
```

Deploy step

- It triggers Ansible playbook which has been described in the next section

```
  deploy:

      needs: build
      runs-on: ubuntu-latest

      steps:
      - uses: actions/checkout@v2
      - uses: ./.github/actions/ansible
        env:
          SSH_PASSWORD: ${{ secrets.SSH_PASSWORD }}
```

Just how Jenkins shows time to complete each step, Github Actions also shows time to each step. For example, https://github.com/vazravasu/Treasure_Hunt/runs/2568903404

5.5 Docker

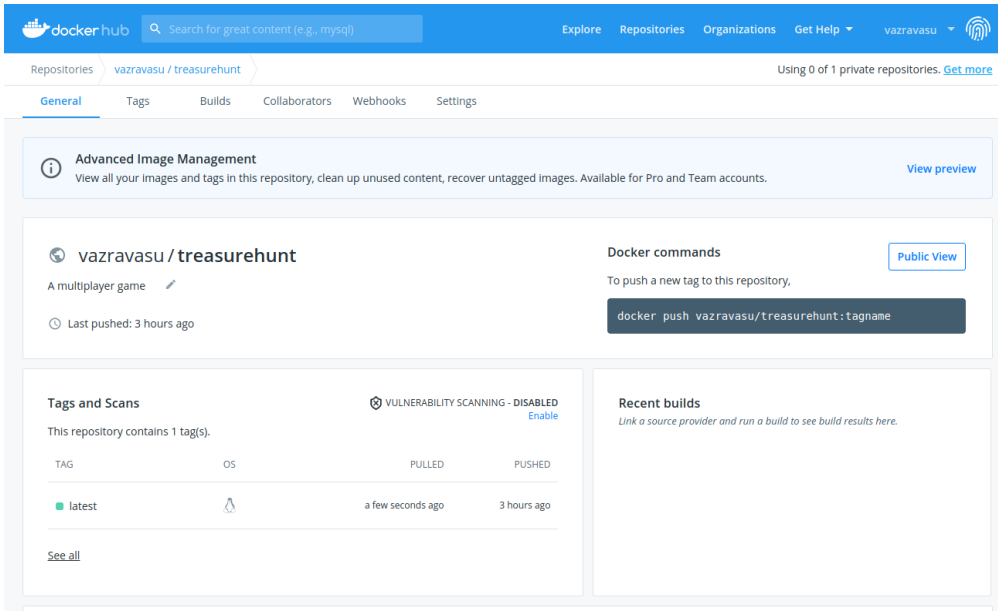
Using docker plugin we will containerize the code and push to docker hub. Then using Ansible we will deploy a docker image on the server. Containers are widely used in industry mainly in the cloud infrastructure; it makes packing and sharing applications very easy. In our application we use the following dockerfile.

Docker File

```
FROM node:14
WORKDIR /usr/src/app

COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 2000

CMD [ "npm", "start" ]
```

5.5.1 DockerHub

Every time a new change has been pushed to GitHub, the corresponding docker image is also pushed into our dockerHub



5.6 Ansible

Ansible uses open ssh to execute the playbooks, become command helps to execute the command as root. The plays are described in the playbook or the yaml file.
Ansible playbook (.yaml file)

Command to Remove previous image and container. Then pull and install docker image from DockerHub.

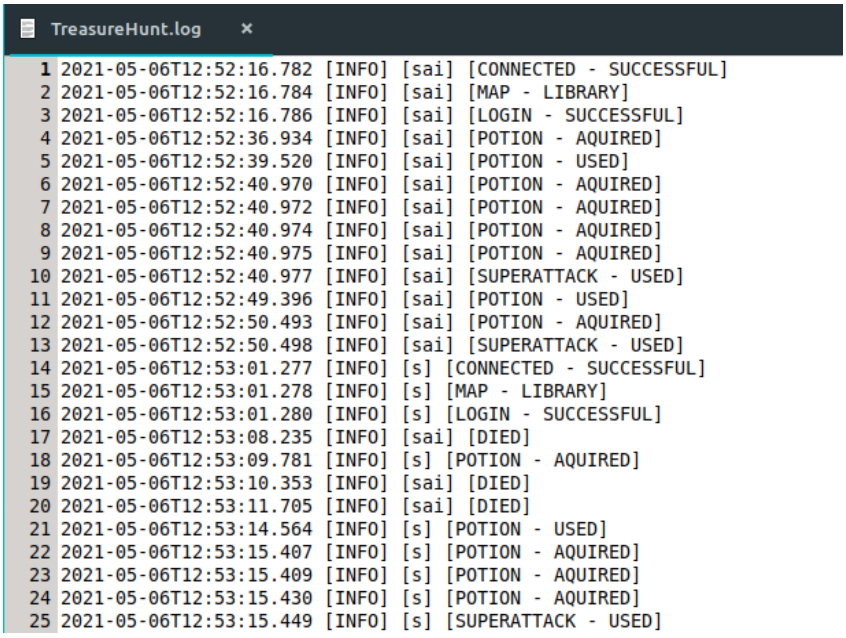
```
---
- hosts: all
  become: true
  tasks:
    - name: Stop the running docker
      shell: docker stop treasurehunt
    - name: Remove the docker
      shell: docker rm -f treasurehunt
    - name: Remove image of the docker
      shell: docker image rm -f vazravasu/treasurehunt:latest
    - name: Download the latest docker and run
      shell: docker run -d -p 2000:2000 --name treasurehunt
vazravasu/treasurehunt:latest
```

5.7 Logging

We use log4js which is similar to log4j in java.
[Log Config]

```
{
  "appenders": {

       "consoleAppender": {
           "type": "console", "layout": {
               "type": "pattern", "pattern": "%d [%p] %m"
       ----------------------------------
       "fileAppender": {
           "type": "file", "filename":"./logs/TreasureHunt.log",
           "layout": {
               "type":"pattern", "pattern": "%d [%p] %m"
-----------------------------------------
   "categories": {
       "default": {
           "appenders": ["consoleAppender", "fileAppender"],
           "level": "info"
```

[Image: Log file]



5.8 ELK monitoring

So now we have logs for our applications, but analysing logs line by line for a particular feature or bug will be not feasible if logs are very large. That is where ELK stack comes. ELK is combination of 3 things - ElasticSearch, Logstash and Kibana

Logstash is used to take the gathered logs, and parse each log into json using grok patterns after the parsing logstash will send parsed logs to elasticsearch where it can be stored. As the name suggests Elastic search is used for query processing. Kibana queries elastic search for the data and creates appealing visualisations. Now we will learn how to use ELK to visualise our logs.

First we create an account on http://cloud.elastic.co and create a deployment.with ELK stack. Obtain the cloud id and cloud auth for the deployment.
log config looks like this. Setup the output of the above obtained cloud id and auth.

Elastic search and Kibana both are RAM and CPU intensive and if you are not using a high specification machine then my recommendation is to use Elasticsearch cloud. They also provide 14 days free trial also.

1. Create a account in http://cloud.elastic.co
2. Click on Create a new deployment.
3. Select Elastic Stack.
4. Just change the deployment name and leave other settings as it is.
5. Then you will be prompted to download the deployment login credentials, download it.
6. Wait for the deployment to be created.

Now we will create a logstash config file where we will describe 3 things

- Input
- Parsing
- Output

In our case logstash config file will look like

Input Section

```
input {
  file {
    path => "log file-path"
    start_position => "beginning"
  }
}
```

Parsing Section

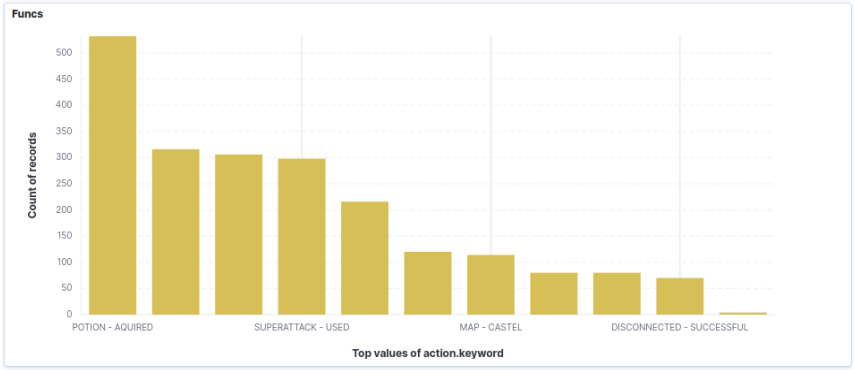
```
filter {
  grok {
    match => [
      "message", "%{TIMESTAMP_ISO8601:timestamp_string}
\[%{GREEDYDATA:logger}\] \[%{GREEDYDATA:username}\]
\[%{GREEDYDATA:action}\]"
    ]
  }

  date{
       match => ["timestamp_string", "ISO8601"]
  }

  mutate {
    remove_field => [timestamp_string]
  }
}
```

Output Section

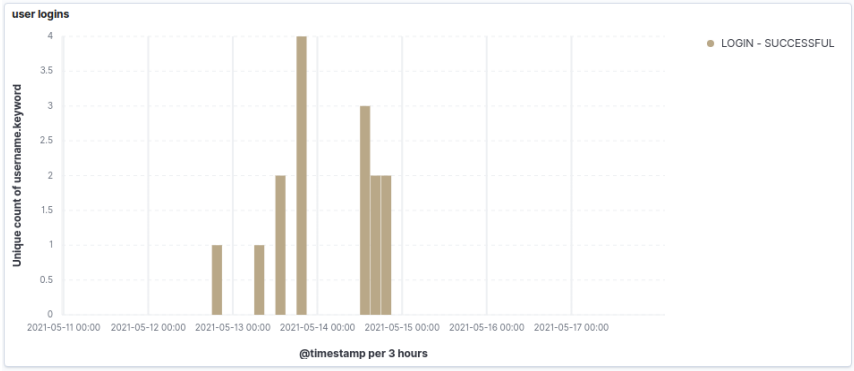
```
output {
  elasticsearch {
    index => "treasurehunt"
    cloud_id => "cloud-id"
    cloud_auth => "auth"
  }

  stdout {
    codec => rubydebug
  }
}
```

We create a dashboard where we can have multiple visualisations based on the index pattern and those visualisations can be updated in real time. Creating a Dashboard in Kibana, you can drop the fields in the middle to create visualisation about them.

We create a dashboard to see all visualisation in one place.



Top image shows all the functionalities being used like potions Acquired, Player Deaths.



Bottom Image shows number of logins vs time

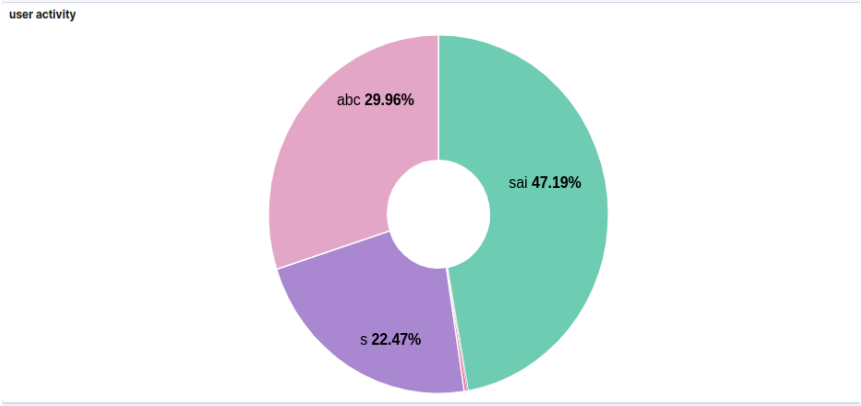
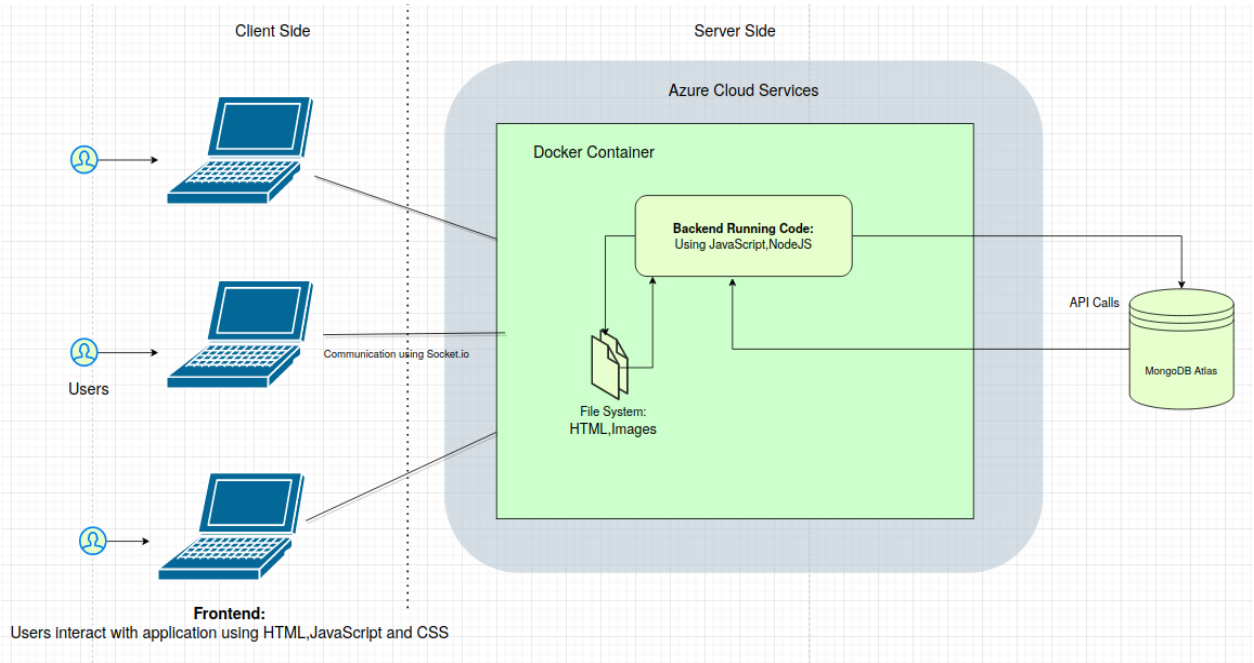


Image shows user activity based on the amount of logs generated by them

6.Experimental Support

6.1 Architecture Diagram



(Image:High level Overview of the Web Application Architecture)

This section provides a high level overview of the Application Architecture.Each user connects to the server using a browser and sees the frontend display as seen in the Results section(Section 7). User authentication is done using MongoDB Atlas which provides an online DB service. Upon user Authentication,the users can start using the application and communicate with the server.The server runs all the required functions and stores the log files in the Docker Container.The entire backend section of the application runs in a single Docker container deployed in an Azure server.

6.2 Running the Application:

- First install npm (it is generally included with node js installation)
- Then run "**npm install**", This will gather all dependencies required for the application
- Then run "**npm start**" to begin the server
- You access the game at "**localhost:2000**"

6.3 Code Walkthrough:

Before going through the specifics,we need to remember that client side is just used for user input and rendering whereas Server side code is for game logic and functionality.So communication between the both is done via messages in the format:
("Message type",Content of message). Let us go through a basic example of socket messages so that further sections can be understood easier.

```
socket.emit("Login Attempt",{username:"george",password:"dummy"});
```

Where "Login Attempt" is message type and the dictionary with user details is the actual content(data) of the message. This information is sent from the user side html page to the server.

```
socket.on("Login Attempt",ValidateLogIn(data));
```

When the server receives this message,using the Message type ,it can identify what appropriate function is to be called on the data.
These basic concepts form the backbone of the entire application.

6.3.1 Server setup and client webpage:

Express is a minimal Node.js web application framework that provides features for developing web and mobile applications. It Allows to set up middlewares to respond to HTTP Requests.

In our app.js code, express is used for file communication

```
var express = require('express');
const { addListener } = require('process');
var serv = require('http').Server(app);
app.get('/', function (req, res){
        res.sendFile(__dirname + '/client/index.html');
});

app.use('/client', express.static(__dirname + '/client'));
serv.listen(process.env.PORT || 2000);
```

Using express for file communication and sending the necessary client files in app.js

This app starts a server and listens on port 2000 for connections. The app responds by sending the index.html (or) the game page when the request is made to the root URL(http://52.255.169.252:2000/) by the user. After this, the user can start using the Gameroom web app on index.html from the client side. Similarly, when a request is made to the client directory from the URl,it sends files and folders corresponding to the request.This is necessary for sending the map and game character images etc.

6.3.2 User Authentication:

It is necessary to have user accounts credentials stored in some format if we want to have user Authentication.For security reasons,it is always better to have the user credential data files in a database rather than an accessible text file on the server. For this reason,we use MongoDB.

MongoDB is a document based(NoSQL) database system that supports querying and indexing features.In our application,we use a free online MongoDB database server called MongoDB Atlas.

Additionally, we also use Mongoose on top of MongoDB. MongooseJS is an Object Document Mapper (ODM) that translates MongoDB database documents to objects in the program.

```
const mongoose = require('mongoose');
const uri = “...”
mongoose.connect(
       ...
);
const accountSchema = new mongoose.Schema({
       username: {type:String, required:true},
       password: String
const account = mongoose.model('account', accountSchema);
```

(Linking the mongooseJs to the online database and creating an “account” object that can be used for DB queries on the user account database)

Now, a user inputs the user credentials(in index.html) and sends the required information to the server.

```
signDivSignIn.onclick = function() {
    socket.emit('signIn', {username:signDivUsername.value, password:signDivPassword.value});
}
```

In app.js (server side), upon receiving a user login request,it runs a database query to validate the user.

```
isValidPassword = async function(data, cb) {
       const res = await account.find({username:data.username,
password:data.password},function(err, res) {...} ) }
```

For communicating both the player info as well as providing the chat functionality,we use socket.io. Socket.IO is a JavaScript library that is utilized in real time web applications and allows for bi-directional communication(from server to client and vice-versa).

6.3.3 Player and Entity class(Inventory, player and bullet)

Since,the entire logic of handling real time player movements and inventory is a huge chunk of the code,we will give just a very brief Idea of the approach used to achieve real time player input and updates.While app.js handles all the dependencies and the basic logic of the application,the specific implementations of the Entities(players etc.) is handled by Entity.js

Both the server side and client side contain a dictionary called Player.List where we store player info such as coordinates,score,map location and inventory.Each player is uniquely identifiable using an id. Whenever a player provides key or mouse input,the script inside the index.html page sends a message regarding the same to the server.

```
document.onkeydown = function(event) {
	if(event.keyCode === 68) {
		socket.emit('keyPress', {inputId:'right', state:true});
	}
```

***index.html** identifies user input and sends message*

The server on receiving this message updates the Player details in the list and broadcasts the Player.List dictionary to all the connected players.

```
socket.on('keyPress', function(data) {
	if(data.inputId === 'left'){
		player.pressingLeft = data.state;
	}
.

self.updateSpd = function() {
	if(self.pressingRight) {
		self.spdX = self.maxSpd;
	}
```

*The server updates the position,based on the input in the array of player info in **Entity.js***

On receiving the updated information,the script in the html page changes all the player details accordingly and is used for rendering in the final step.

```
socket.on('update',function(data){
	 for(var i = 0 ; i < data.player.length; i++){
		var pack = data.player[i];
		var p = Player.list[pack.id];
		if(p){ ...
				p.x = pack.x;
```

Corresponding update takes place in the list stored in index.html

Following this mechanism,we are able to provide basic real time multiplayer functionality.

6.3.4 Chat functionality

The chat window is seen on client html page.On entering the message in the chat-form (form in html that allows the user to input text), we send/emit the message to the server.

```
chatForm.onsubmit = function(event) {
               socket.emit('sendMsgToServer', chatInput.value);
}
```

On receiving the message and identifying the message type as “sendMsgToServer”, the server sends the message to all the other players using an array called SOCKET_LIST that contains socket objects of all connected players.

```
socket.on('sendMsgToServer',function(data){
        for(var i in SOCKET_LIST){
       // emit to send message to all in the socket list, to identify message username
               SOCKET_LIST[i].emit('addToChat',player.username + ': ' + data);
        }
 });
```

On receiving the message from the server,the message is appended to the chat text window in the client side html page.

```
socket.on('addToChat', function(data){
       chatText.innerHTML += '<div>' + data + '</div>';
})
```

By this mechanism, we are able to provide chat functionality.

In the next section we look at rendering and drawing the assets and map on the HTML page

6.3.5 Rendering/Drawing on the HTML page

The final step is to render all the players,map and other game relevant information on the user screen.By following all the above steps,the user/client has a list/array of Player and Bullet Objects identifiable by a unique ID.These objects contain all relevant information such as coordinates of the players, inventory details, score etc.

To draw the game,we use Canvas from HTML.The HTML <canvas> element is used to draw graphics using JavaScript. Using Canvas, we can add text, images, boxes etc.

```
<div id="gameDiv" style="display:none;">
    <div id="game" style="position:absolute;width:500px;height:500px">

        <canvas id="ctx" width="500" height="500" style="position:absolute;border:1px solid
#000000;"></canvas>

        <canvas id="ctx-ui" width="500" height="500" style="position:absolute;border:1px
solid #000000;"></canvas>
...

</div>
</div>
```

Adding canvas element to the html page

Finally,we call a function that runs every '40 ms' to render these images on canvas using the array of player info.

```
setInterval(function(){
        ...
        drawMap();
        drawScore();
        for(var i in Player.list)
                Player.list[i].draw();
        for(var i in Bullet.list)
                Bullet.list[i].draw();
    },40);
```

***drawMap**,**drawScore** and the **.draw** functions are used to render the respective images on the canvas.*

This is a complete overview of the steps involved in the web application.Upon running the application,we see results as seen in Section 7.

7.Results

7.1 Login and SignUp Page:

On entering the URL, we see the following Login Page.

- If you are a new user, you can choose the Sign-Up Option.
- After creating a new Account (or) using an existing account click on the Sign-In option.

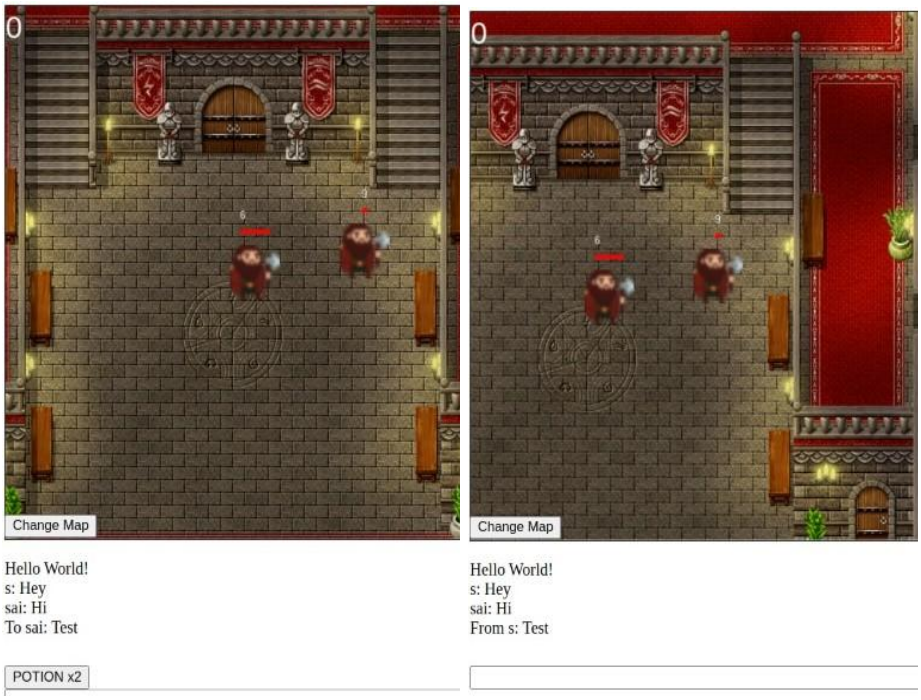
7.2 Game Screen:

After connecting successfully to the server,a player can start playing the game.
Below is the actual game screen.

Just below the game canvas, we can see the inventory and map functionalities.

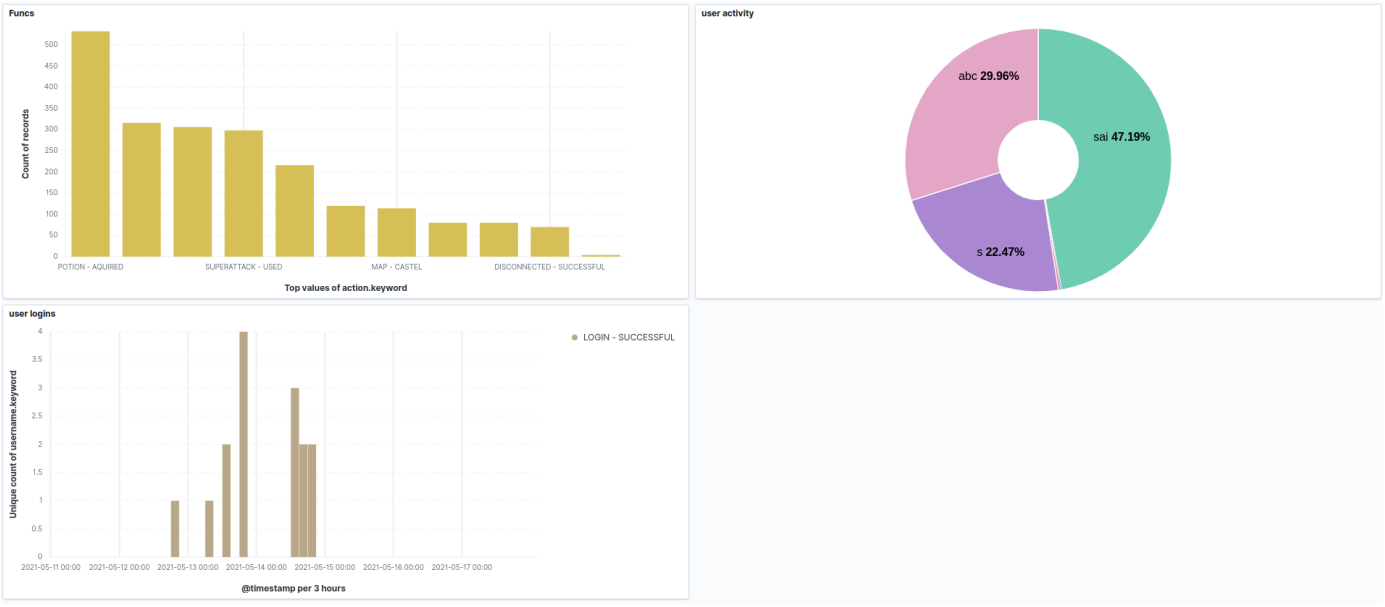
7.3 Chat functionality:

On successful connection with the server,players can send each other messages and are identifiable by usernames.



7.4 ELK Images

Dashboard created using Kibana in general ELK.



8.Scope for Future Work

Currently ,we have a randomly generated Inventory for Players.We need to design a puzzle using the provided functionalities such that players can come together,discover keys(useful inventory items) and use them together to solve the puzzle.

Since the game puts all the people joining the website in a single game/room it will defeat the point of having a puzzle when number of users is high.So,we need to create separate rooms for small groups of people.Need to add individual room functionalities that can be accessed using room ID and password.

Right now, log files are sent to a remote system via ssh connection.Then the ELK stack is applied on top of it.We need to add a mechanism to keep sending the logs to elasticsearch from the container directly on a real time basis.

9.Conclusion

Thus,we have built a Teamwork /Team Development focused Gameroom using DevOps tools for automated building to deployment. We have made use of the following DevOps tools: Github, GitHub Actions, Docker and Ansible. We also have added log files and associated Monitoring functionality using the ELK Stack.Since the entire pipeline runs in automated fashion and is triggered by a push to the GitHub repository,the deployment process is very quick.

On an average, it takes 3-4 minutes to completely run the entire pipeline and deploy to the Azure server. Using DevOps tools,we were able to experience faster,frequent and more error free updates to deployment.

References

[1] "Building Multiplayer Games With Node.Js And Socket.IO | Modern Web". 2021. *Modern Web*. https://modernweb.com/building-multiplayer-games-with-node-js-and-socket-io.

[2] "Empowering App Development For Developers | Docker". 2021. *Docker*. https://www.docker.com/.

[3] "Get Started". 2021. *Socket.IO*. https://socket.io/get-started/chat.

[4] "Github Actions Documentation - Github Docs". 2021. *Docs.Github.Com*. https://docs.github.com/en/actions.

[5] "How To Build A Multiplayer Browser Game (Part 1) | Hacker Noon". 2021. *Hackernoon.Com*. https://hackernoon.com/how-to-build-a-multiplayer-browser-game-4a793818c29b.

[6] "Introduction To Testing With Mocha And Chai | Codecademy". 2021. *Codecademy*. https://www.codecademy.com/articles/bapi-testing-intro.

[7] "Javascript Tutorial - Tutorialspoint". 2021. *Tutorialspoint.Com*. https://www.tutorialspoint.com/javascript/index.htm.

[8] "Mongoose V5.12.9: Getting Started". 2021. *Mongoosejs.Com*. https://mongoosejs.com/docs.