

Final Project Group

By Yang Liu, Nuo Xu, Weibo Dai

Abstract

For the final project, we basically built a full-stack twitter-like web application and deployed the app on aws-eks.

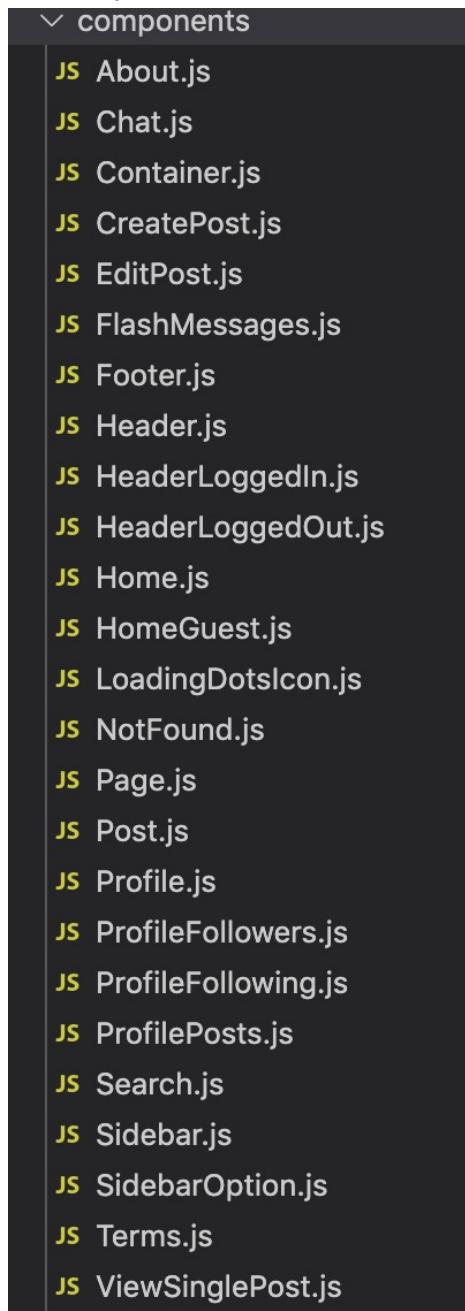
1. Basic application structure

The twitter-like application contains three tech-stacks, one front-end react app, one back-end redis app and one back-end node.js app. We also have the deployment files to deploy the app on both local minikube and on eks. We can see the project folder structure as follows.

```
DEOPSFINALPROJECT
  └── backend
      ├── controllers
      ├── models
      └── .gitignore
    └── app.js
    └── db.js
    └── Dockerfile
    └── exec.sh
    └── package-lock.json
    └── package.json
    └── Procfile
    └── README.md
    └── router.js
    └── test.js
  └── deployment files
      └── create EKS / tf-provision-eks-cl...
          └── manually deploy-yaml
      └── provision-tf
          └── back-depl.tf
          └── front-depl.tf
  └── frontend
      ├── app
      ├── .dockerignore
      ├── .gitignore
      ├── default.conf
      ├── Dockerfile
      ├── exec.sh
      ├── generateHtml.js
      ├── netlify.toml
      ├── package-lock.json
      ├── package.json
      ├── previewDist.js
      └── webpack.config.js
  └── readme.md
```

1. Front-end react service

For the front end service, we used react framework. We defined multiple components to load pages and defined routers in the Main.js to connect to the back-end service.



2.Back-end redis and Node.js service

In the node.js backend service, we matched the front-end router to the back-end services in the router.js file. Then we defined three controllers to handle the requests. The logics on how the requests should be handled are written in models file, after the definition of each model.

Inside the Post.js file, we created the redis server to cache the front-end input data. Before caching the data, we did the validation check, so that only valid data could be cached into the redis server. Only non-empty, good words would be accepted.

```
Post.prototype.cleanUp = function () {
  if (typeof this.data.title != 'string') {
    this.data.title = '';
  }
  if (typeof this.data.body != 'string') {
    this.data.body = '';
  }

  // get rid of any bogus properties
  this.data = [
    title: sanitizeHTML(this.data.title.trim(), {
      allowedTags: [],
      allowedAttributes: {}
    }),
    body: sanitizeHTML(this.data.body.trim(), {
      allowedTags: [],
      allowedAttributes: {}
    }),
    createdDate: new Date(),
    author: ObjectID(this.userid),
  ];
};

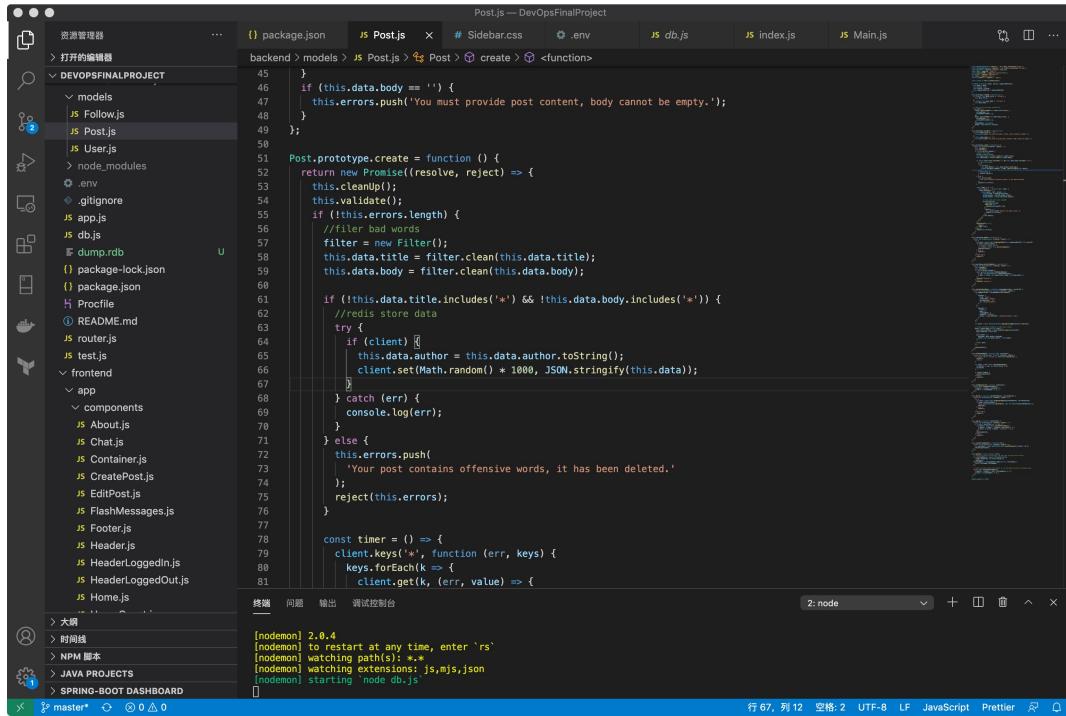
Post.prototype.validate = function () {
  if (this.data.title == '') {
    this.errors.push('You must provide a title, title cannot be empty.');
  }
  if (this.data.body == '') {
    this.errors.push('You must provide post content, body cannot be empty.');
  }
};
```

We set the waiting time of storing data callback function to be 10 seconds, which means that after 10 seconds of user input, all input requests would be handled by redis server and persisted into mongoDB database.

```
const timer = () => {
  client.keys('*', function (err, keys) {
    keys.forEach(k => {
      client.get(k, (err, value) => [
        const dataDB = JSON.parse(value);
        dataDB.author = dataDB.author.trim();
        dataDB.author = ObjectID(dataDB.author);

        // real save post into mongoDB
        postsCollection
          .insertOne(dataDB)
          .then(info => {
            resolve(info.ops[0]._id);
          })
          .catch(e => {
            this.errors.push('Please try again later.');
            reject(this.errors);
          });
        client.del(k);
      ]);
    });
  });
  setInterval(() => {
    timer();
  }, 1000 * 10);
} else {
  reject(this.errors);
});
};
```

The data are then fetched from local array and data will also be stored into the mongodb database server.



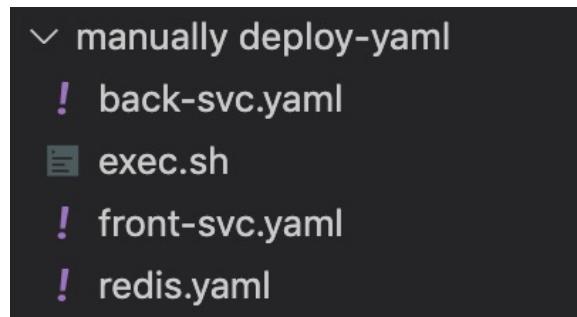
A screenshot of a code editor (VS Code) showing a project structure for a Node.js application. The project is named 'DEVOPTFINALPROJECT'. The 'models' folder contains 'Follow.js' and 'Post.js'. The 'Post.js' file is open and shows code for a 'Post' model. It includes validation logic for 'body' and 'title', and a method to store data to MongoDB using 'nodemon'. The code editor interface shows tabs for package.json, Post.js, Sidebar.css, .env, db.js, index.js, and Main.js. Below the code editor is a terminal window showing the output of 'nodemon' starting the application. The terminal output includes: [nodemon] 2.0.4, [nodemon] to restart at any time, enter `rs` [nodemon] watching path(s): ** [nodemon] watching extensions: js,mjs,json [nodemon] starting `node db.js`

We defined the mongodb server in db.js. We then exports the backend service to mongodb server and build the connection in .env file.

3. Deployment

For deployment, we mainly provide two options.

The first one is to manually deploy the app on local minikube. It contains three yaml files pointing to three services. For each service, it contains both creating deployment and start service parts.



The image shows three terminal windows side-by-side, each displaying a YAML configuration file for a Kubernetes service.

- front-depl.tf**: A Terraform configuration file defining a Service named "final-back-svc". It uses a LoadBalancer type with port 8080, protocol TCP, and targetPort 8080. It also defines a Deployment named "final-back" with one replica, selecting pods labeled "app: final-back". The deployment uses a RollingUpdate strategy with maxUnavailable set to 1 and maxSurge set to 1. The template section specifies a container named "final-back" using the image "merphylau/final-be:latest" and exposing port 8080.
- back-svc.yaml**: A Kubernetes Service YAML file. It defines a Service named "final-back-svc" with a LoadBalancer type, port 8080, protocol TCP, and targetPort 8080. It uses a selector for pods labeled "app: final-back".
- front-depl.tf**: Another Terraform configuration file defining a Service named "redis". It uses a LoadBalancer type with port 6379, protocol TCP, and targetPort 6379. It also defines a Deployment named "node-redis" with one replica, selecting pods labeled "app: node-redis". The deployment uses a RollingUpdate strategy with maxUnavailable set to 1 and maxSurge set to 1. The template section specifies a container named "redis" using the image "redis:latest" and exposing port 6379.
- back-svc.yaml**: A second Kubernetes Service YAML file. It defines a Service named "redis" with a LoadBalancer type, port 6379, protocol TCP, and targetPort 6379. It uses a selector for pods labeled "app: node-redis".
- redis.yaml**: A third Kubernetes Service YAML file. It defines a Service named "node-redis" with a LoadBalancer type, port 6379, protocol TCP, and targetPort 6379. It uses a selector for pods labeled "app: node-redis".

We also created a exec.sh file to automatically run all the services on local minikube.

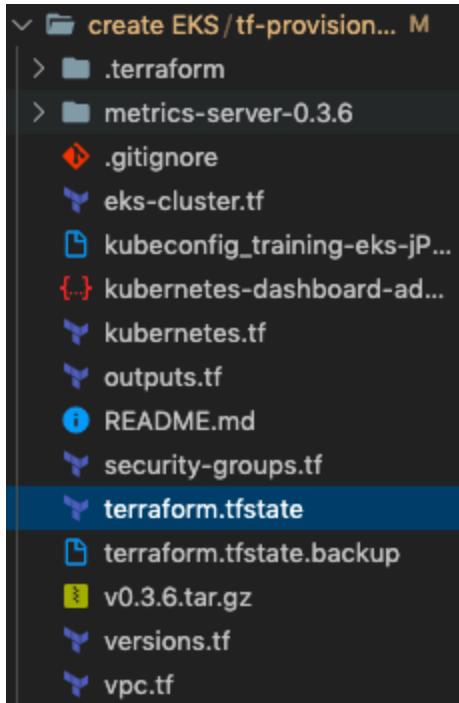
The image shows a terminal window with a shell script named "exec.sh". The script contains the following commands:

```

1  kubectl delete deployment --all
2  kubectl delete service --all
3
4  kubectl apply -f front-svc.yaml
5  kubectl apply -f redis.yaml
6  kubectl apply -f back-svc.yaml

```

The second option is to deploy the app on eks. We create the EKS using terraform files as following structure:

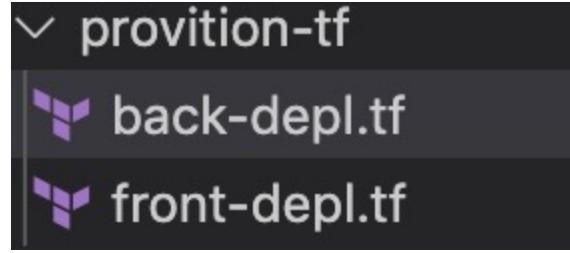


And we deploy the pods using the exec.sh file as following:

```
exec.sh — DevOpsFinalProject
front-depl.tf      exec.sh  X

deployment files > manually deploy-yaml > exec.sh
1  kubectl delete deployment --all
2  kubectl delete service --all
3
4  kubectl apply -f front-svc.yaml
5  kubectl apply -f redis.yaml
6  kubectl apply -f back-svc.yaml
```

We also created the provision-tf file, it contains two terraform file, namely back-depl.tf and front-depl-tf, but we have no time to test it.



```

back-depl.tf — DevOpsFinalProject
  front-depl.tf ×  back-depl.tf ×
deployment files > provition-tf > back-depl.tf > final-back > {
  1 provider "kubernetes" {}
  2
  3   references
  4   resource "kubernetes_deployment" "final-back" {
  5     metadata {
  6       name = "final-back"
  7       labels = {
  8         test = "final-back"
  9       }
  10
  11     spec [
  12       replicas = 2
  13
  14       selector {
  15         match_labels = {
  16           test = "final-back"
  17         }
  18
  19       template {
  20         metadata {
  21           labels = {
  22             test = "final-back"
  23           }
  24
  25         }
  26
  27         spec {
  28           container {
  29             image = "merphylau/final-be"
  30             name = "final-back"
  31
  32             resources {
  33               limits {
  34                 cpu    = "0.5"
  35                 memory = "512Mi"
  36               }
  37               requests {
  38                 cpu    = "250m"
  39                 memory = "50Mi"
  40               }
  41             }
  42           }
  43         }
  44       }
  45     ]
  46   }
  47 }
```

```

front-depl.tf ×  back-depl.tf ×
deployment files > provition-tf > front-depl.tf > final-front > {} s
  1 provider "kubernetes" {}
  2
  3   references
  4   resource "kubernetes_deployment" "final-front" {
  5     metadata {
  6       name = "final-front"
  7       labels = {
  8         test = "final-front"
  9       }
  10
  11     spec [
  12       replicas = 3
  13
  14       selector {
  15         match_labels = {
  16           test = "final-front"
  17         }
  18
  19       template {
  20         metadata {
  21           labels = {
  22             test = "final-front"
  23           }
  24
  25         }
  26
  27       spec {
  28         container {
  29           image = "merphylau/final-fe"
  30             name = "final-front"
  31
  32             resources {
  33               limits {
  34                 cpu    = "0.5"
  35                 memory = "512Mi"
  36               }
  37               requests {
  38                 cpu    = "250m"
  39                 memory = "50Mi"
  40               }
  41             }
  42
  43           }
  44         }
  45       }
  46     ]
  47   }
  48 }
```

2. Functionalities of the application

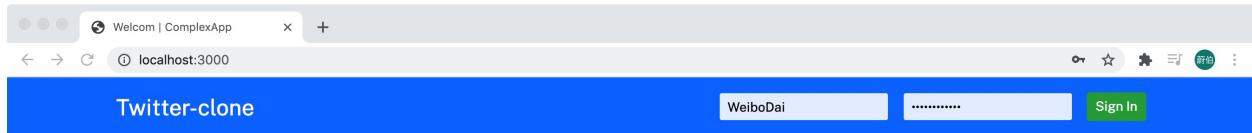
The application basically supports several main functions as following:

1. user login logout, signup
2. user writing tweets and post the tweet(user cannot post offensive tweets)
3. user follow other users and see who follows the user

4. user search for favourite tweets in following users

Below we would show a brief demo of how the applications would work on each functionality.

First, if the user has not signed up before, he or she needs to sign up first. The user should only enter letter or number as the user name, the email should also follow the email format, and the password should be longer than 12 characters. If the user did not follow the previous rules, he would be blocked. See the below screenshots as examples.



Remember Writing?

Follow your interests.
Hear what people are talking about.
Join the conversation.

Username

You must provide a valid email address.

Email

Password

Sign up for Twitter



Remember Writing?

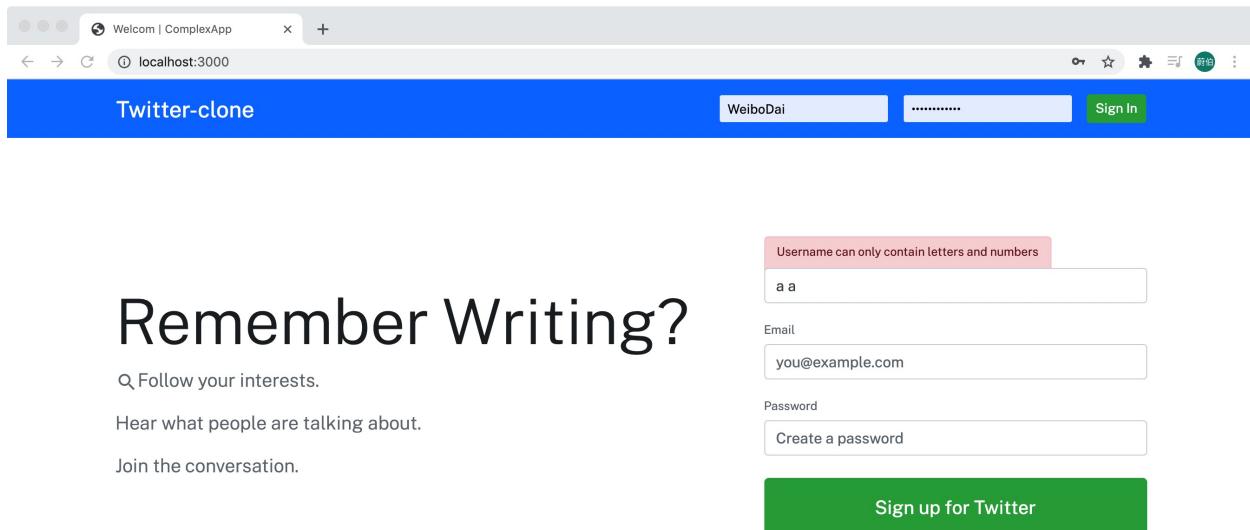
Follow your interests.
Hear what people are talking about.
Join the conversation.

Username

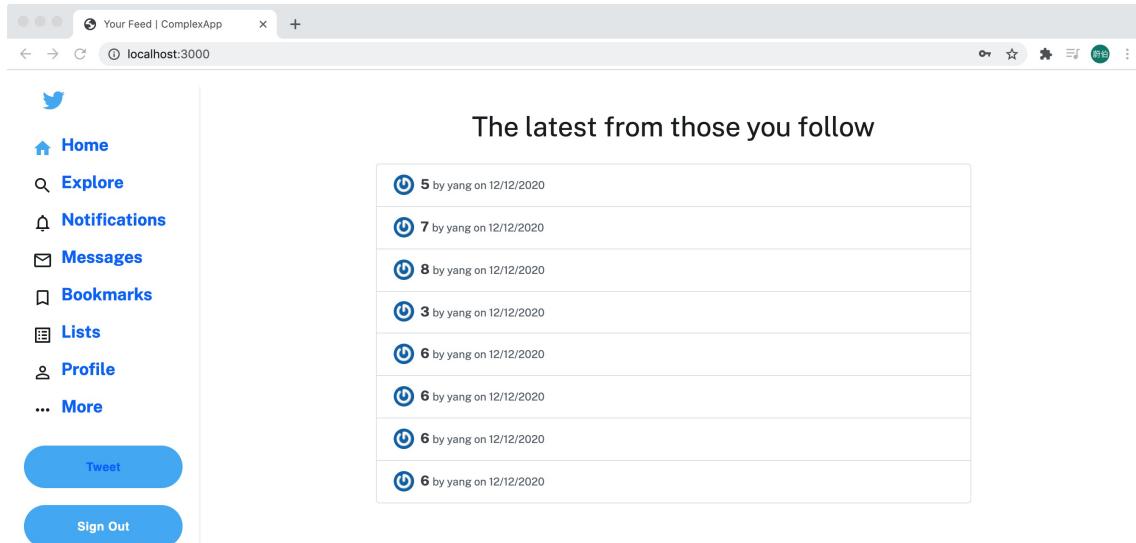
Email

Password must be at least 12 characters.

Sign up for Twitter



After the user is signed up, he could login and see the main page as below.

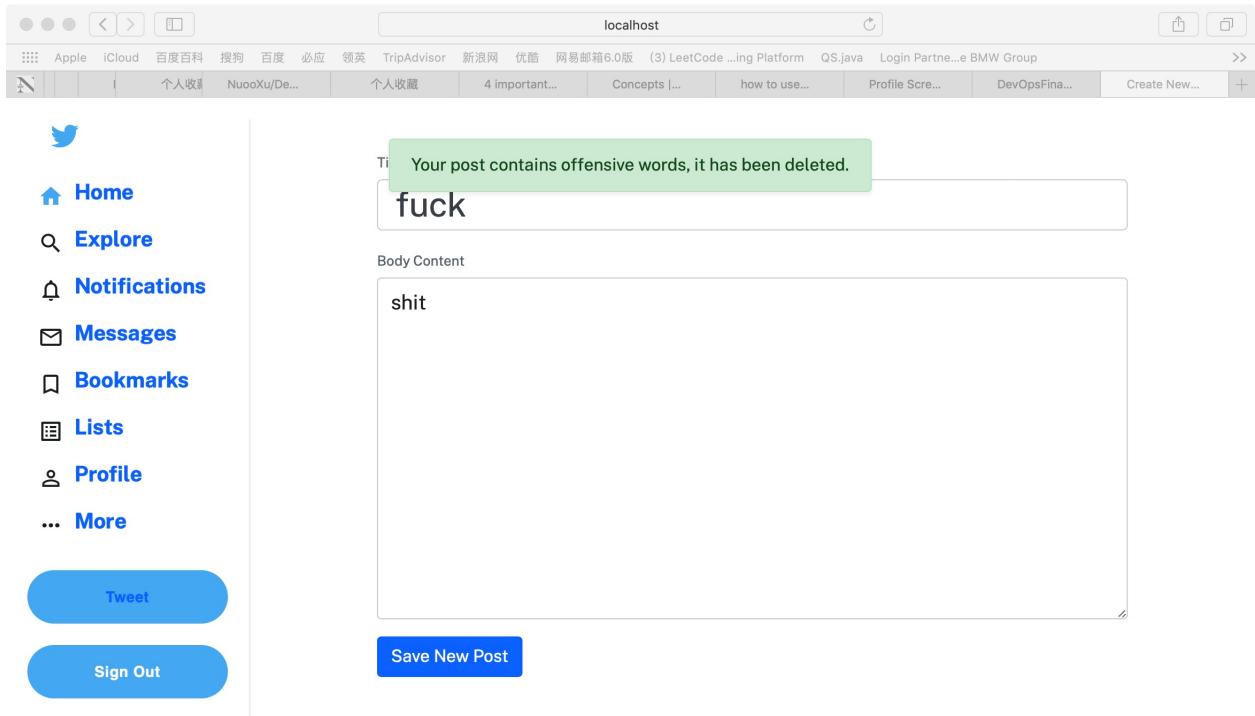


In the main page the user could see the latest tweets from his liked tweeter.

Second, in the main page, the user could write posts by clicking tweet button on the left column.

The screenshot shows a web browser window with the URL `localhost:3000/create-post`. On the left, there is a sidebar with a Twitter icon and links: Home, Explore, Notifications, Messages, Bookmarks, Lists, Profile, and More. Below these are two blue rounded rectangular buttons: 'Tweet' and 'Sign Out'. The main content area has a 'Title' field containing 'A good Day' and a 'Body Content' field containing 'Today is a good day!!!!'. At the bottom right of the main area is a blue 'Save New Post' button.

User are not allowed to write bad words in the posts.



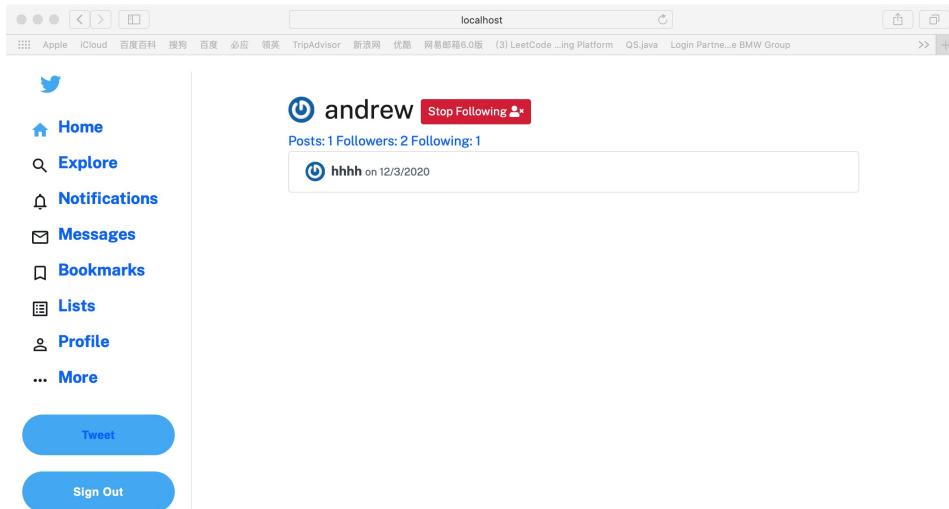
The content would first be stored in the redis database, after 10 seconds, the post request would be handled and the posts would be stored in the database. We would then see the content on profile.

The screenshot shows a web browser window titled "Profile Screen | ComplexApp" with the URL "localhost:3000/profile/weibodai". The page has a sidebar on the left with navigation links: Home, Explore, Notifications, Messages, Bookmarks, Lists, Profile, and More. Below these are two blue buttons: "Tweet" and "Sign Out". The main content area displays the profile of "weibodai". It includes a profile icon, the name "weibodai", and statistics: Posts: 3 Followers: 0 Following: 1. Below this, there are three tweet cards: "A good Day on 12/13/2020", "A good Day on 12/13/2020", and "asdasddsdasa on 12/13/2020".

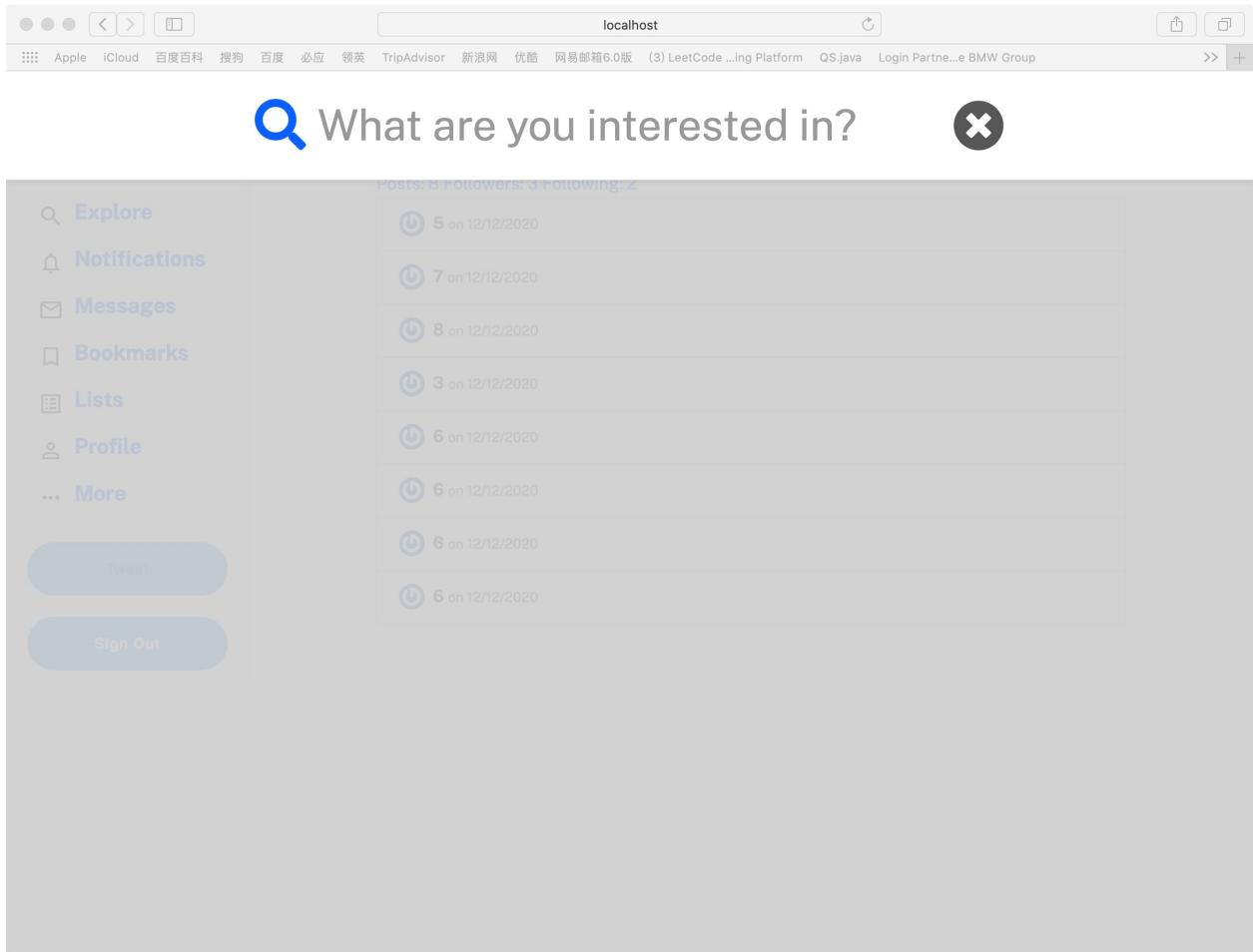
Third, the users could also add followers and see who is following them.

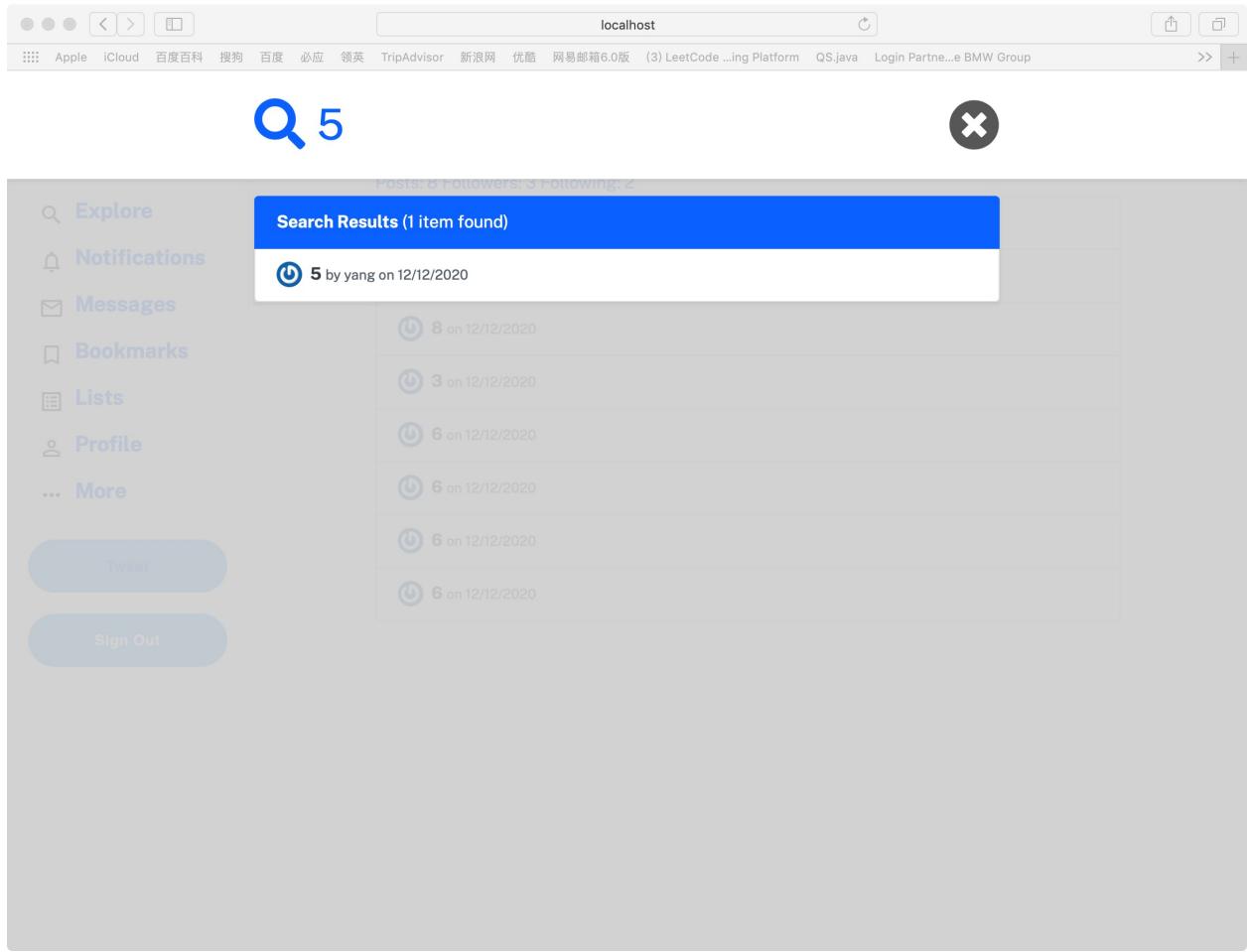
The screenshot shows a web browser window titled "localhost" with the URL "localhost:3000/profile/andrew". The page has a sidebar on the left with navigation links: Home, Explore, Notifications, Messages, Bookmarks, Lists, Profile, and More. Below these are two blue buttons: "Tweet" and "Sign Out". The main content area displays the profile of "andrew". It includes a profile icon, the name "andrew", and a "Follow" button. Statistics show: Posts: 1 Followers: 1 Following: 1. Below this, there is one tweet card: "hhhh on 12/3/2020".

After following a twitter, we could see their tweets in the profile section. We could also stop following.

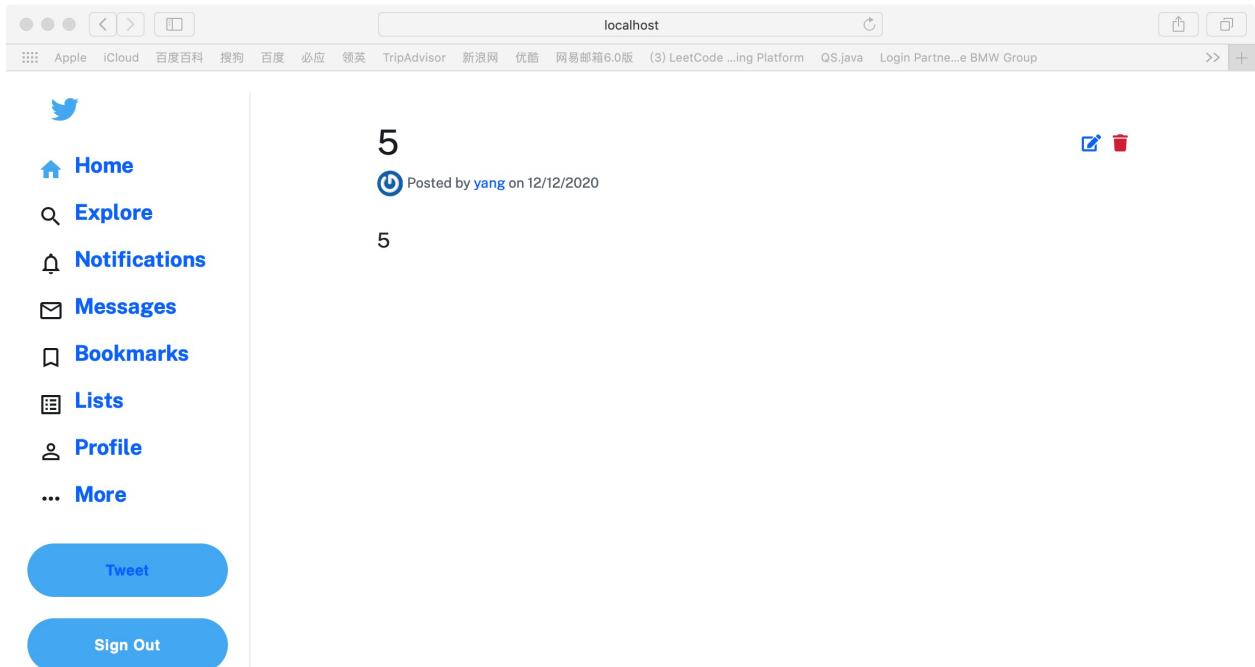


Fourth, we could also search for the tweets posted by the user's followers.





Then we can see the search results.



3. Deploy Run and Monitor the application.

Part1 Deploy and run the app on local machine.

1. Go to the front end app folder and run npm install
2. Run npm run dev
3. Add .env file under the back end folder
4. Go to back end app folder and run npm start
5. Go to http://localhost:3000/

Part2 Deploy and run the app on local minikube

1. Go to deployment files/manually deploy-yaml file
2. Run chmod +x exec.sh
3. Run ./exec.sh
4. Go to http://localhost:3000/?webapp=http://localhost:8080

Part3 Deploy the app on EKS

1. Go to the create-EKS file to create the EKS in the aws(terraform init, terraform plan, terraform apply)
2. After create the EKS, config the kubectl using aws eks --region \$(terraform output region) update-kubeconfig --name \$(terraform output cluster_name)
3. Then deploy the k8s metrics server using by run the following command

```
wget -O v0.3.6.tar.gz  
https://codeload.github.com/kubernetes-sigs/metrics-server/tar.gz/v0.3.6 && tar -xzf  
v0.3.6.tar.gz
```

```
kubectl apply -f metrics-server-0.3.6/deploy/1.8+/
```

```
kubectl get deployment metrics-server -n kube-system
```

```
Kubectl apply -f  
https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta8/aio/deploy/recommended.yaml
```

```
kubectl proxy
```

```
kubectl apply -f  
https://raw.githubusercontent.com/hashicorp/learn-terraform-provision-eks-cluster/master/kubernetes-dashboard-admin.rbac.yaml
```

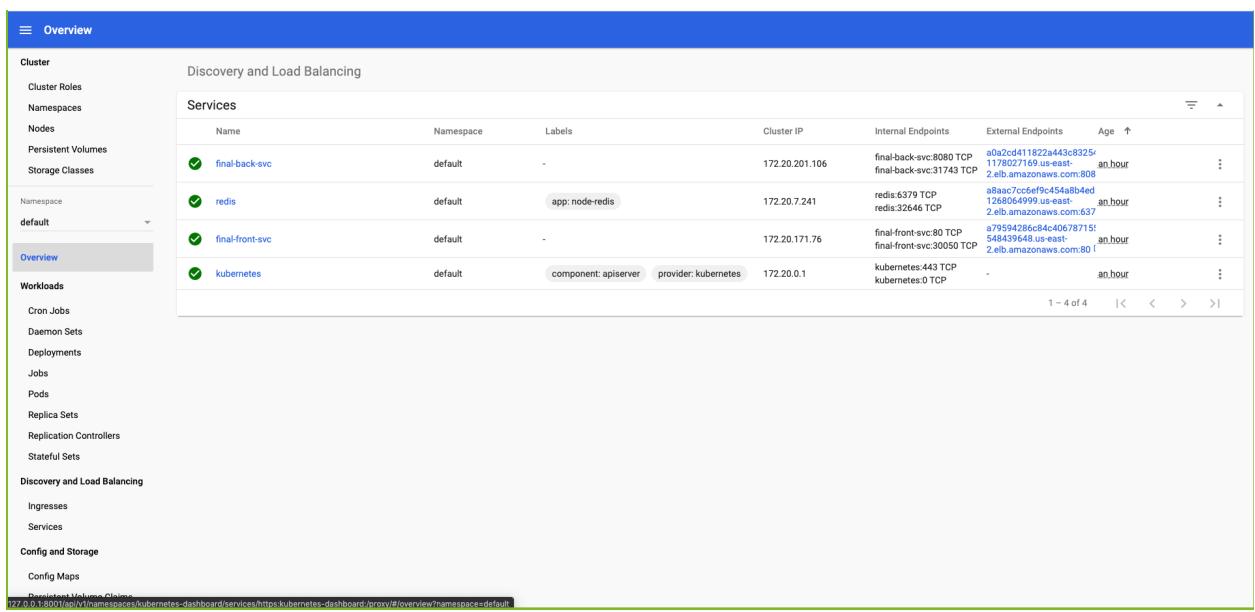
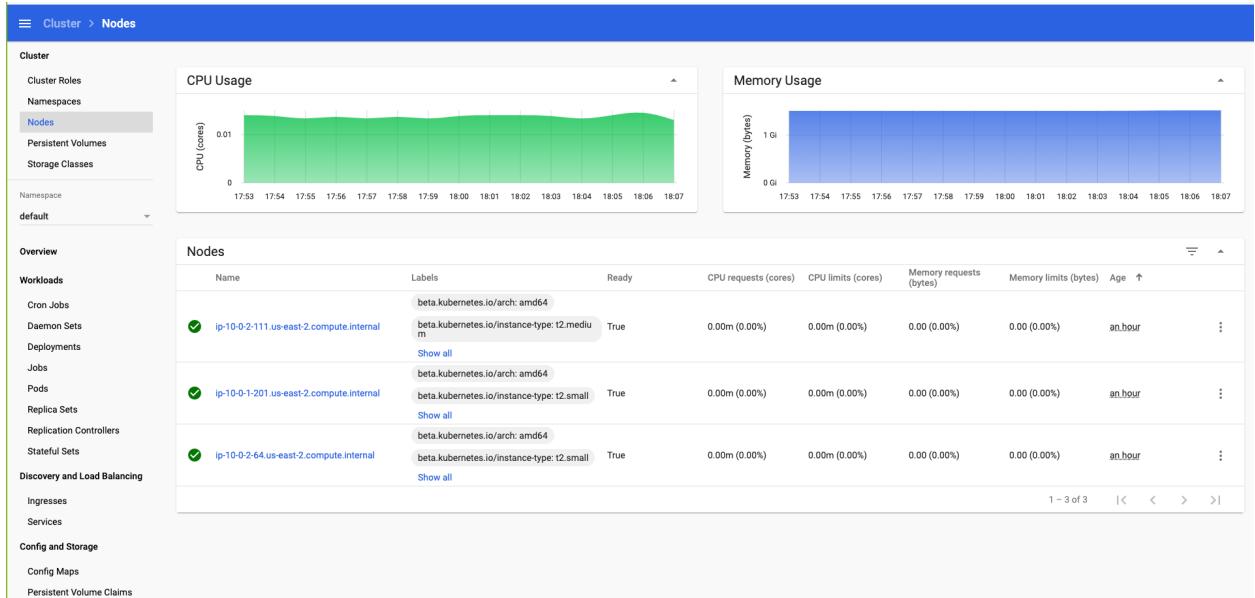
```
kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep service-controller-token | awk '{print $1}')
```

Then can see the monitor.

4. Go to deployment files/manually deploy-yaml file
5. Run chmod +x exec.sh
6. Run ./exec.sh
7. And the deployment is done

Part4 Monitor the App

Below we could see how much CPU and Memory the instance is consuming and services' endpoints using metrix.



4. References.

<https://learn.hashicorp.com/tutorials/terraform/eks>