Final Project Group

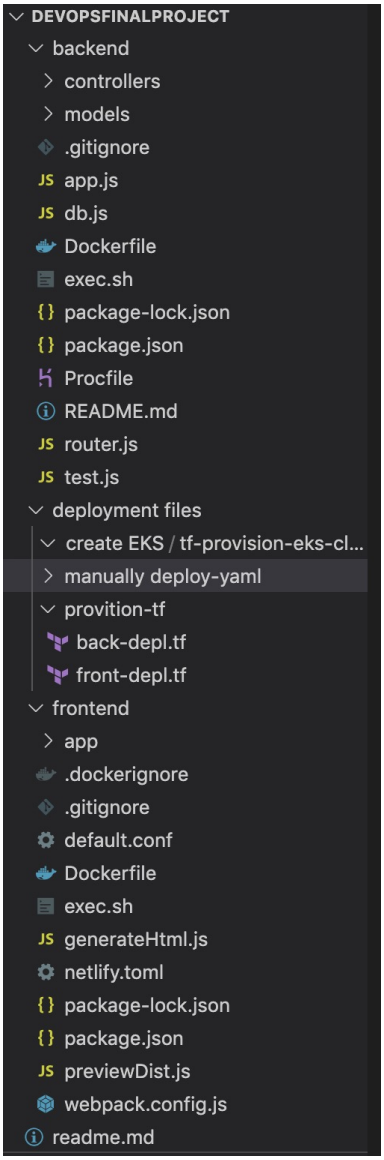
By Yang Liu, Nuo Xu, Weibo Dai

Abstract

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

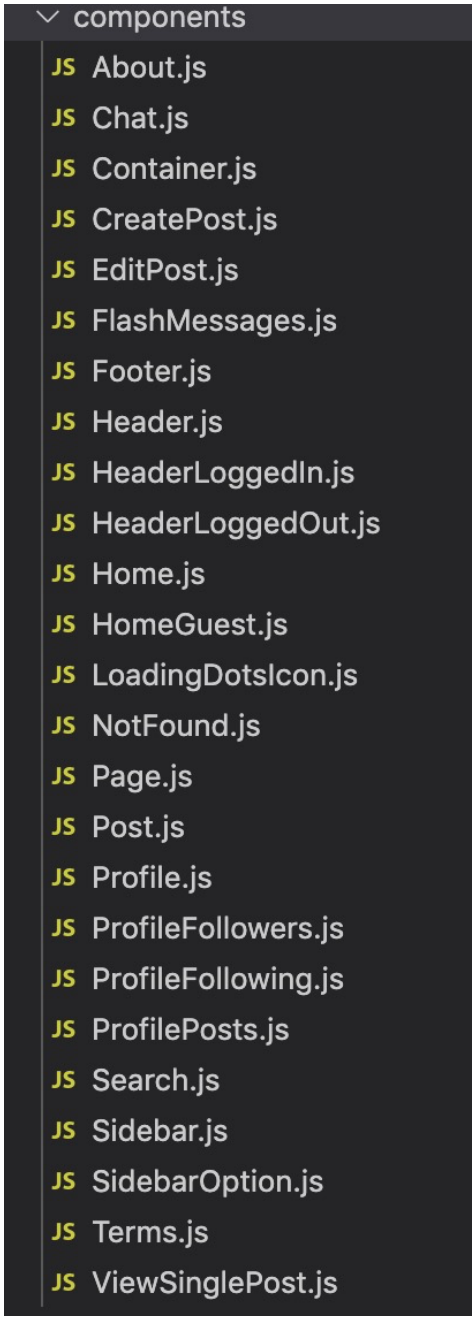
1.Basic application structure

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.



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.

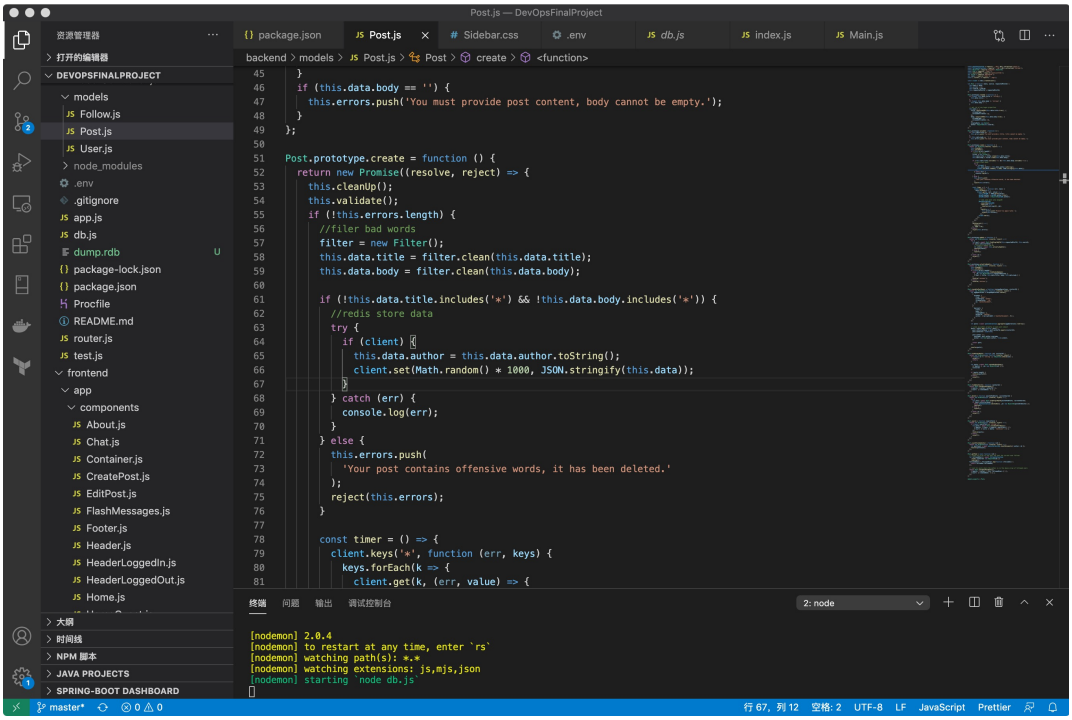
Inside the Post.js file, we created the redis server to cache the front-end input data. Before caching the data, we did the validation check, so that only valid data could be cached into the redis server. Only non-empty, good words would be accepted.

```
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.

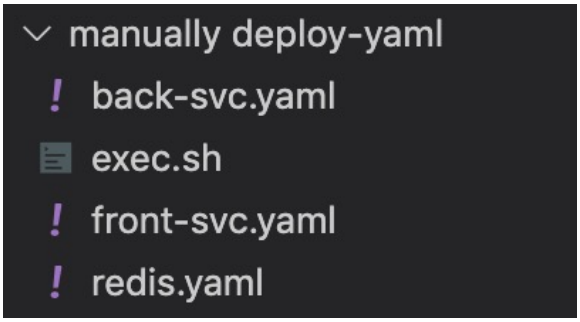


We defined the mongodb server in db.js. We then exports the backend service to mongodb server and build the connection in.env file.

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.



front-depl.tf | back-svc.yaml

deployment files > manually deploy-yaml > back-svc.yaml > {} s

```yaml
apiVersion: v1
kind: Service
metadata:
  name: final-back-svc
spec:
  type: LoadBalancer
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: final-back
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: final-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: final-back
  minReadySeconds: 15
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: final-back
    spec:
      containers:
        - name: final-back
          image: merphylau/final-be:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
```

front-depl.tf | back-svc.yaml

deployment files > manually deploy-yaml > back-svc.yaml > {} s

```yaml
apiVersion: v1
kind: Service
metadata:
  name: final-back-svc
spec:
  type: LoadBalancer
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: final-back
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: final-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: final-back
  minReadySeconds: 15
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: final-back
    spec:
      containers:
        - name: final-back
          image: merphylau/final-be:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
```

front-depl.tf | redis.yaml

deployment files > manually deploy-yaml > redis.yaml >

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    app: node-redis
spec:
  type: LoadBalancer
  ports:
    - port: 6379
      protocol: TCP
      targetPort: 6379
  selector:
    app: node-redis
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-redis
spec:
  selector:
    matchLabels:
      app: node-redis
  replicas: 1
  template:
    metadata:
      labels:
        app: node-redis
    spec:
      containers:
        - name: redis
          image: redis:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 6379
```

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

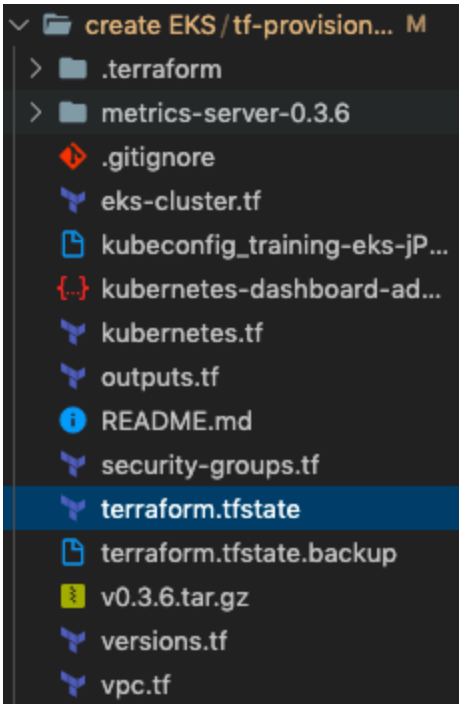
exec.sh — DevOpsFinalP

front-depl.tf | exec.sh

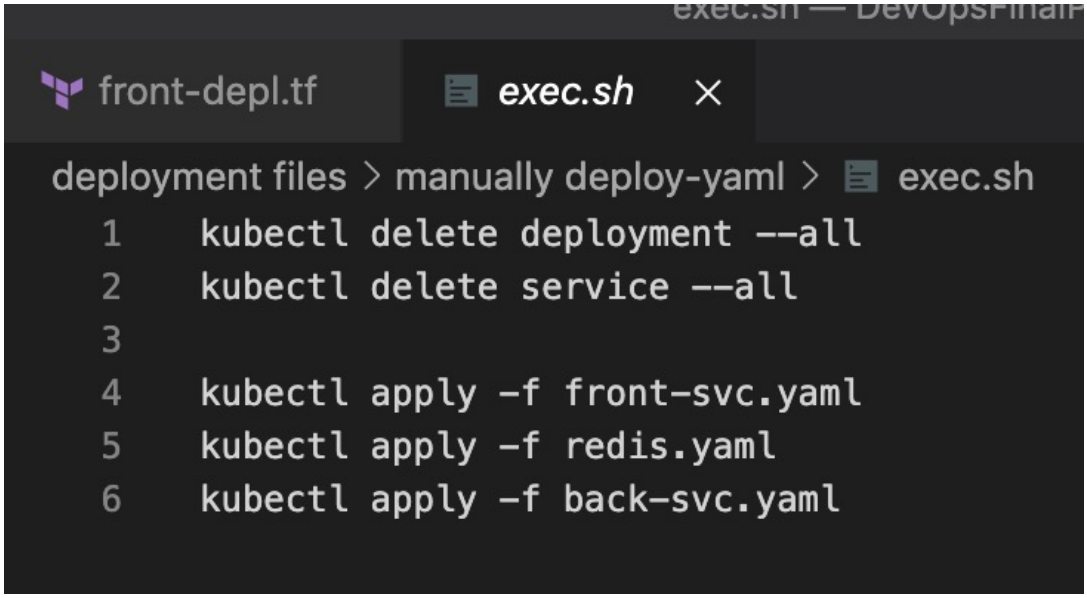
deployment files > manually deploy-yaml > exec.sh

```
kubectl delete deployment --all
kubectl delete service --all

kubectl apply -f front-svc.yaml
kubectl apply -f redis.yaml
kubectl apply -f back-svc.yaml
```

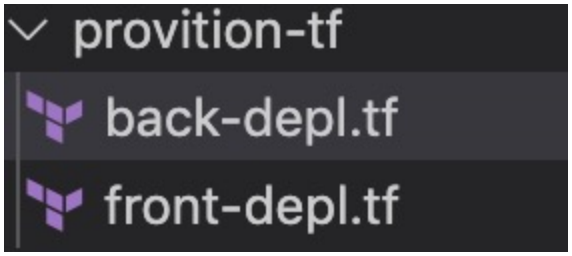
The second option is to deploy the app on eks. We create the EKS using terraform files as following structure:



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



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.



front-depl.tf | back-depl.tf

deployment files > provition-tf > back-depl.tf > final-back

```
provider "kubernetes" {}

1 references
resource "kubernetes_deployment" "final-back" {
  metadata {
    name = "final-back"
    labels = {
      test = "final-back"
    }
  }

  spec {
    replicas = 2

    selector {
      match_labels = {
        test = "final-back"
      }
    }

    template {
      metadata {
        labels = {
          test = "final-back"
        }
      }

      spec {
        container {
          image = "merphylau/final-be"
          name  = "final-back"

          resources {
            limits {
              cpu    = "0.5"
              memory = "512Mi"
            }
            requests {
              cpu    = "250m"
              memory = "50Mi"
            }
          }
        }
      }
    }
  }
}
```

front-depl.tf | back-depl.tf

deployment files > provition-tf > front-depl.tf > final-front

```
provider "kubernetes" {}

1 references
resource "kubernetes_deployment" "final-front" {
  metadata {
    name = "final-front"
    labels = {
      test = "final-front"
    }
  }

  spec {
    replicas = 3

    selector {
      match_labels = {
        test = "final-front"
      }
    }

    template {
      metadata {
        labels = {
          test = "final-front"
        }
      }

      spec {
        container {
          image = "merphylau/final-fe"
          name  = "final-front"

          resources {
            limits {
              cpu    = "0.5"
              memory = "512Mi"
            }
            requests {
              cpu    = "250m"
              memory = "50Mi"
            }
          }


        }
      }
    }
  }
}
```

2. Functionalities of the application

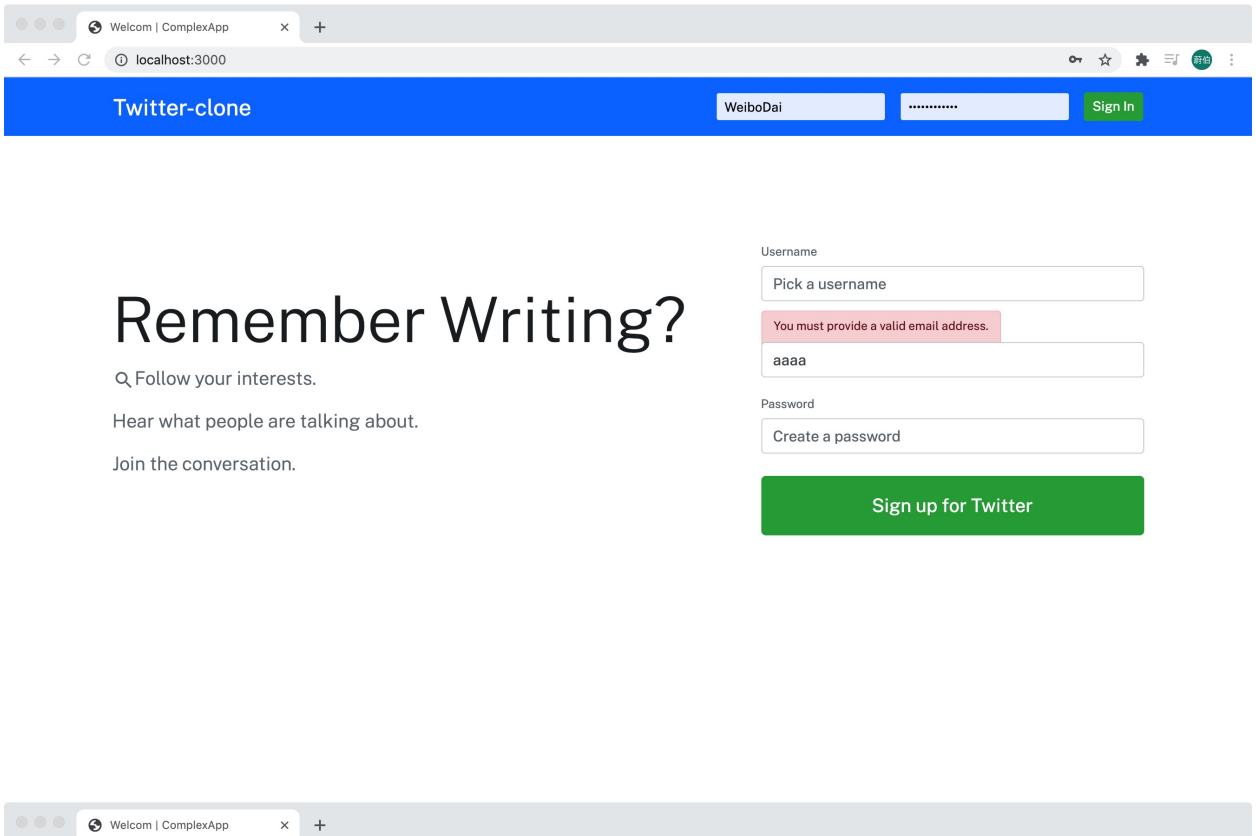
The application basically supports several main functions as following:

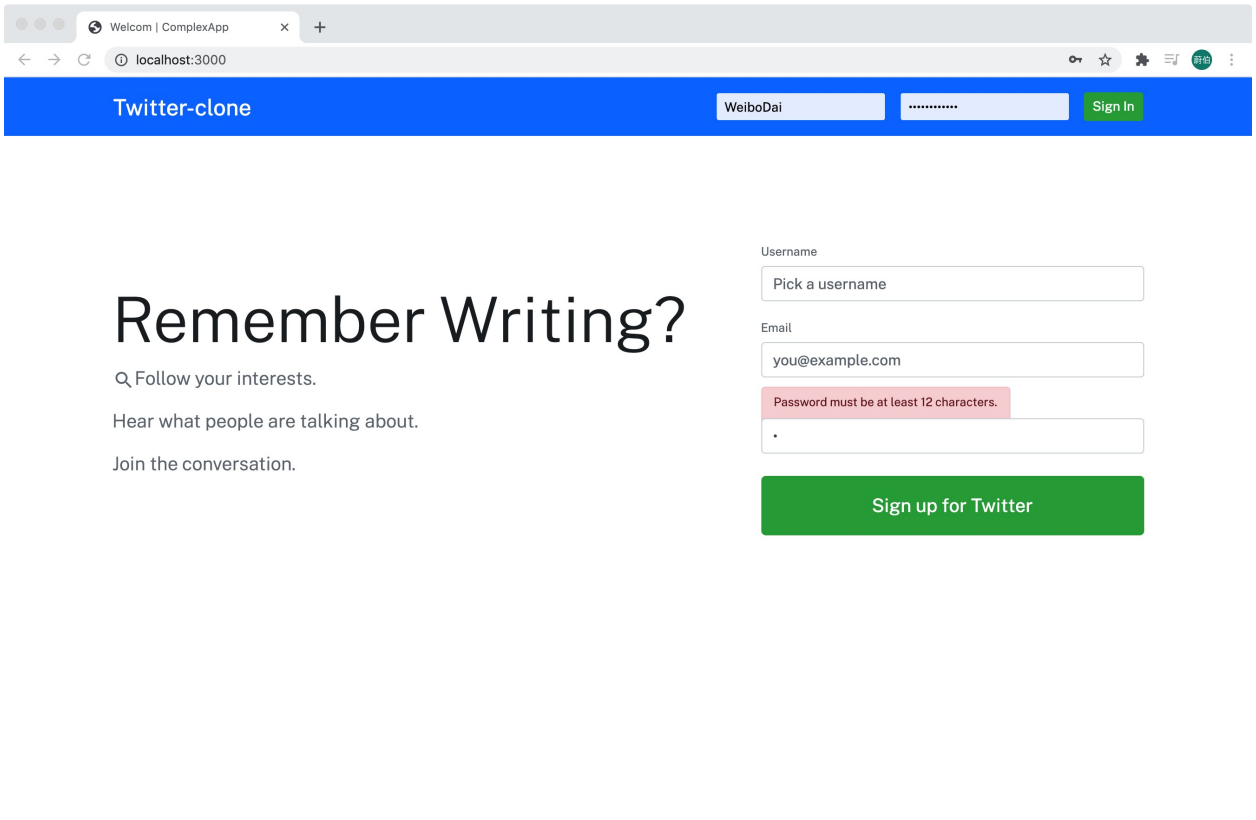
1. user login logout, signup
2. user writing tweets and post the tweet(user cannot post offensive tweets)
3. user follow other users and see who follows the user

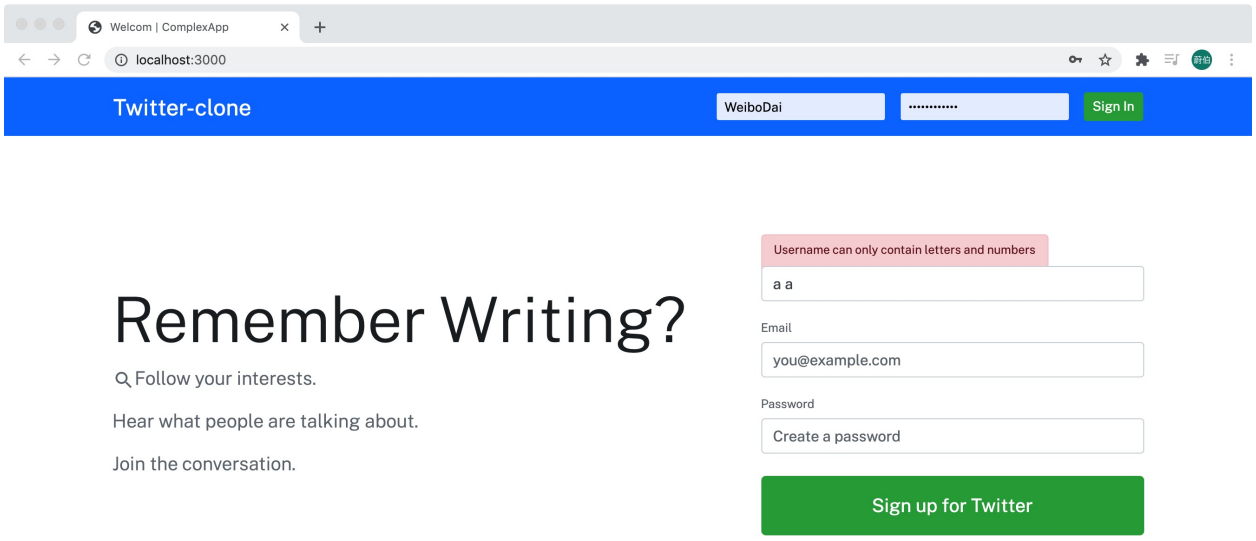
4. user search for favourite tweets in following users

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

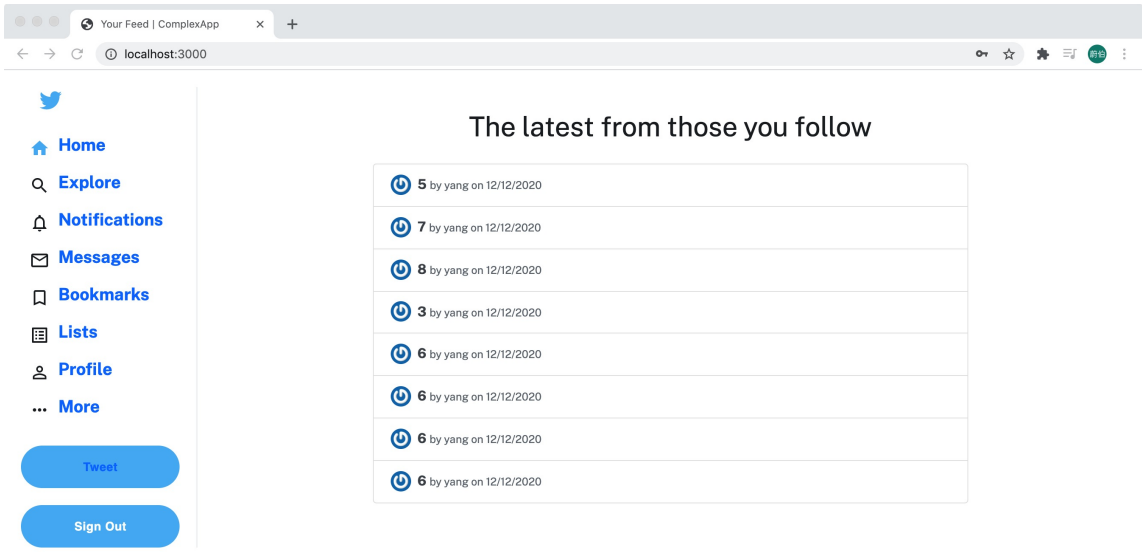
First, if the user has not signed up before, he or she needs to sign up first. The user should only enter letter or number as the user name, the email should also follow the email format, and the password should be longer than 12 characters. If the user did not follow the previous rules, he would be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


Welcom | ComplexApp
localhost:3000
Twitter-clone
WeiboDai
Sign In
Remember Writing?
Follow your interests.
Hear what people are talking about.
Join the conversation.
Username
Pick a username
You must provide a valid email address.
aaaa
Password
Create a password
Sign up for Twitter


Welcom | ComplexApp
localhost:3000
Twitter-clone
WeiboDai
Sign In
Remember Writing?
Follow your interests.
Hear what people are talking about.
Join the conversation.
Username
Pick a username
Email
you@example.com
Password must be at least 12 characters.
Sign up for Twitter

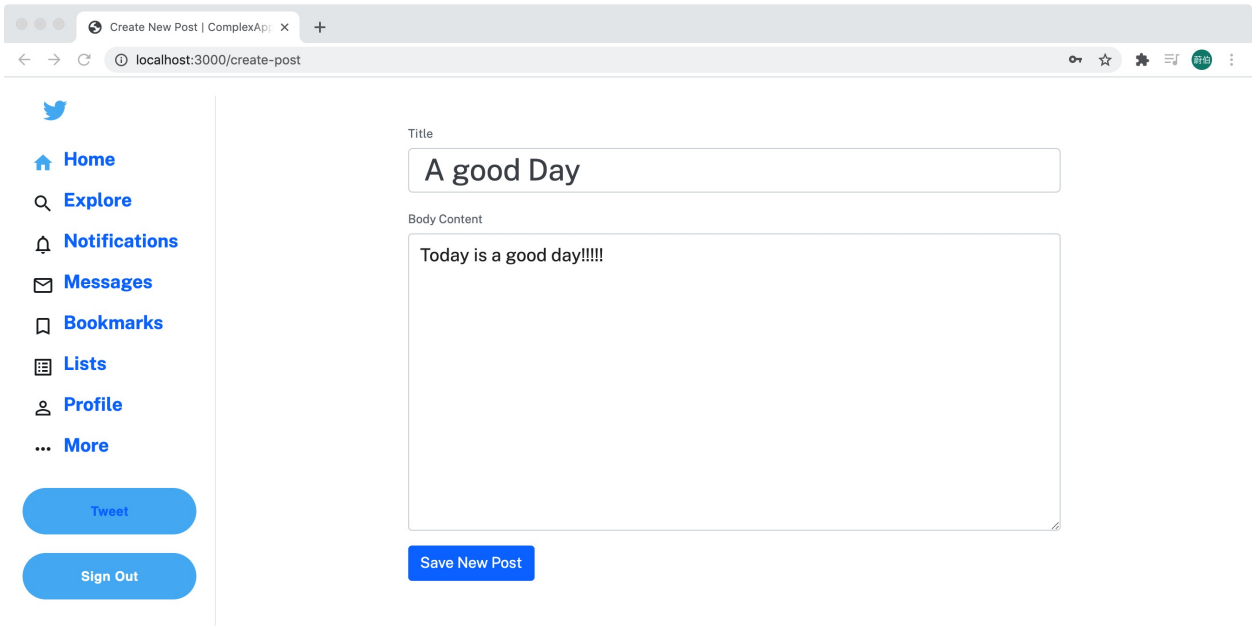


After the user is signed up, he could login and see the main page as below.

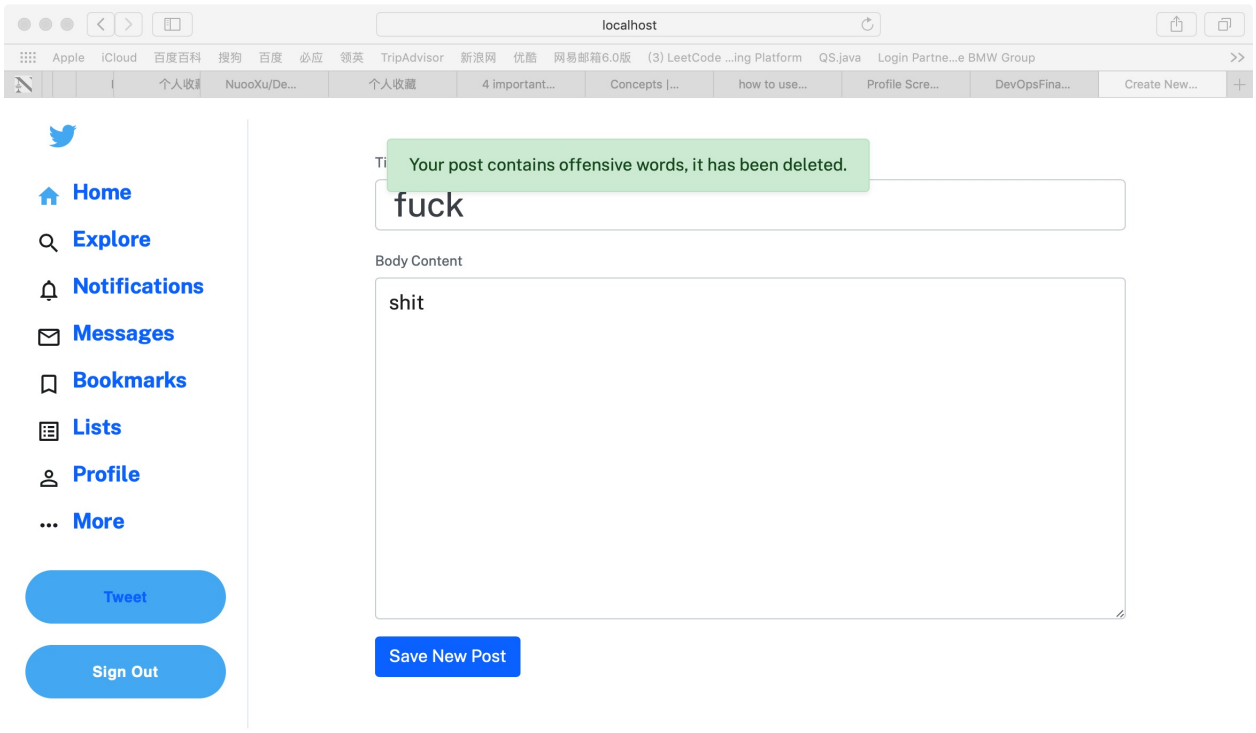


In the main page the user could see the latest tweets from his liked tweeter.

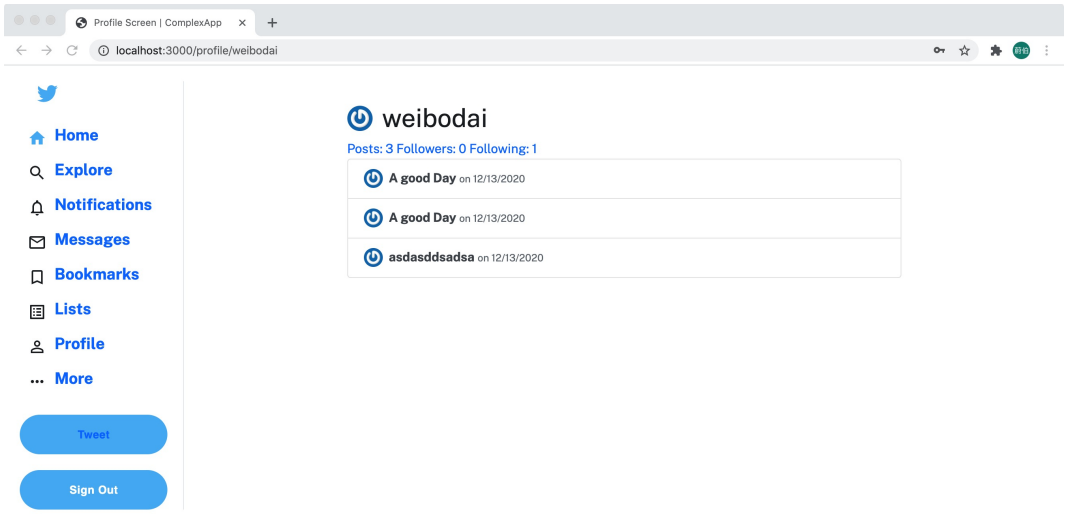
Second, in the main page, the user could write posts by clicking tweet button on the left column.



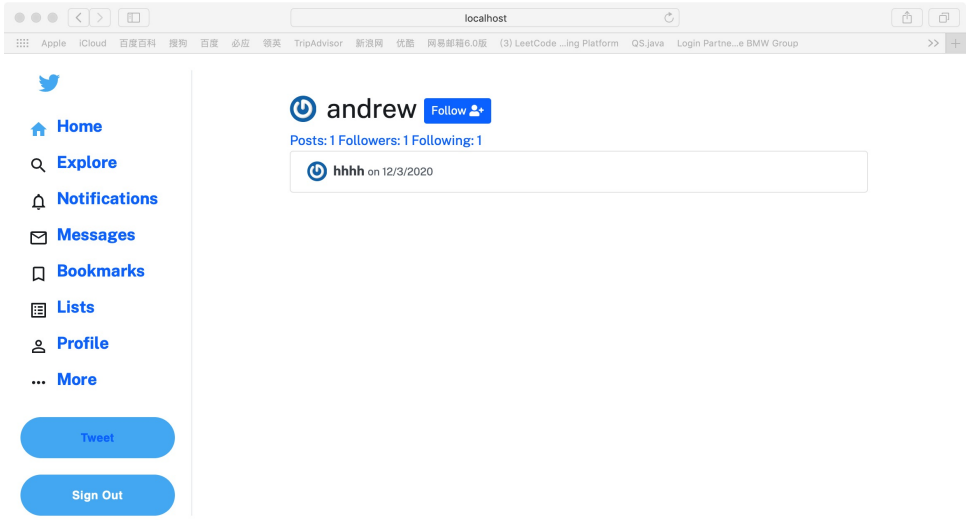
User are not allowed to write bad words in the posts.



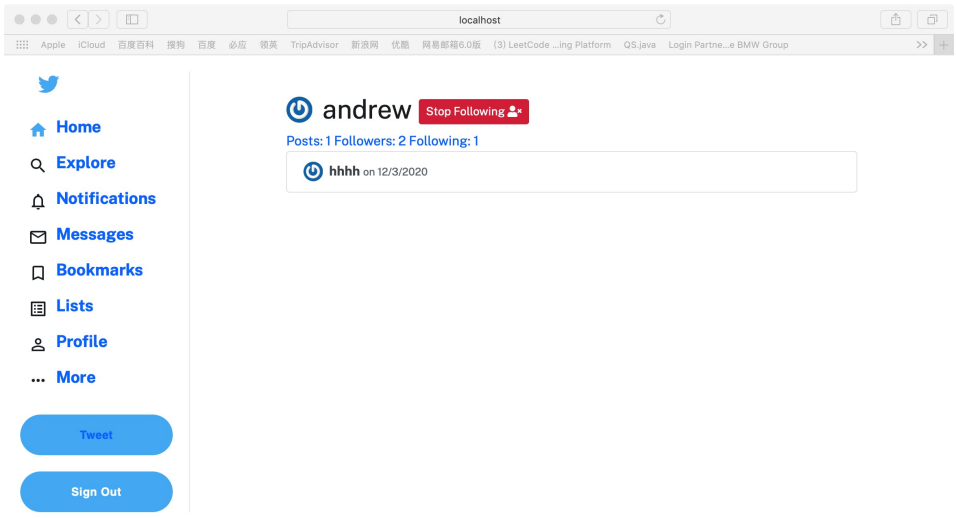
The content would first be stored in the redis database, after 10 seconds, the post request would be handled and the posts would be stored in the database. We would then see the content on profile.



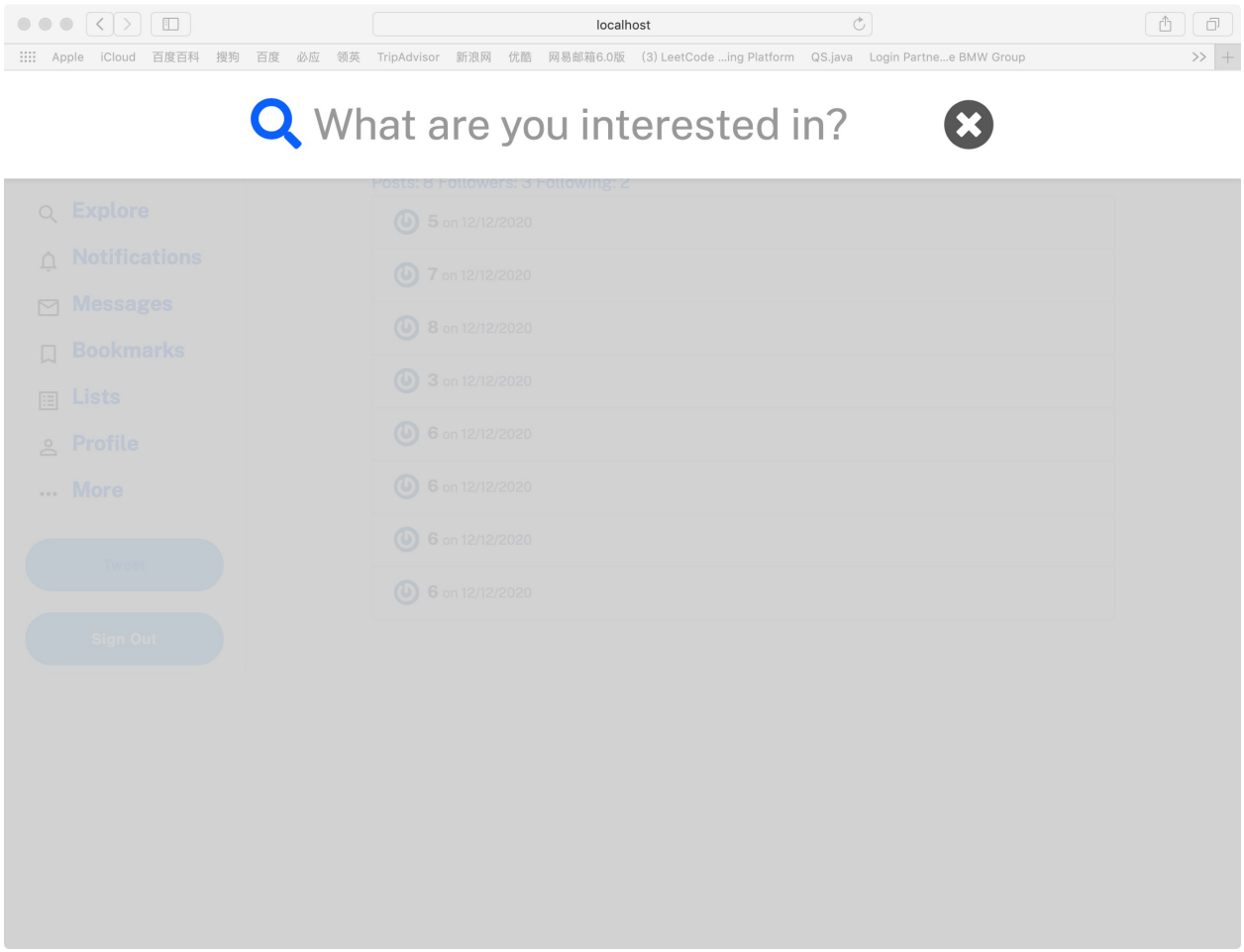
Third, the users could also add followers and see who is following them.

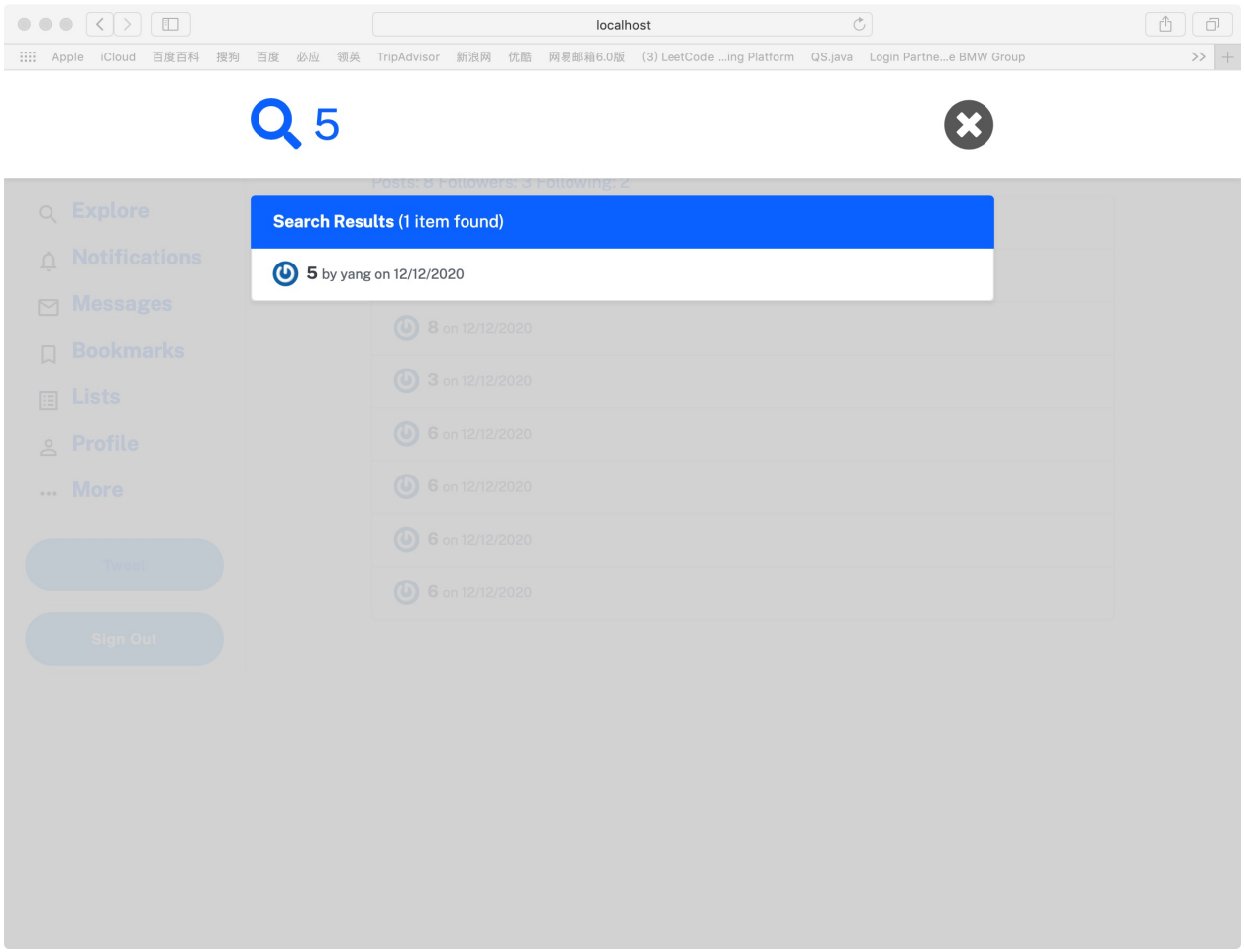


After following a twitter, we could see their tweets in the profile section. We could also stop following.

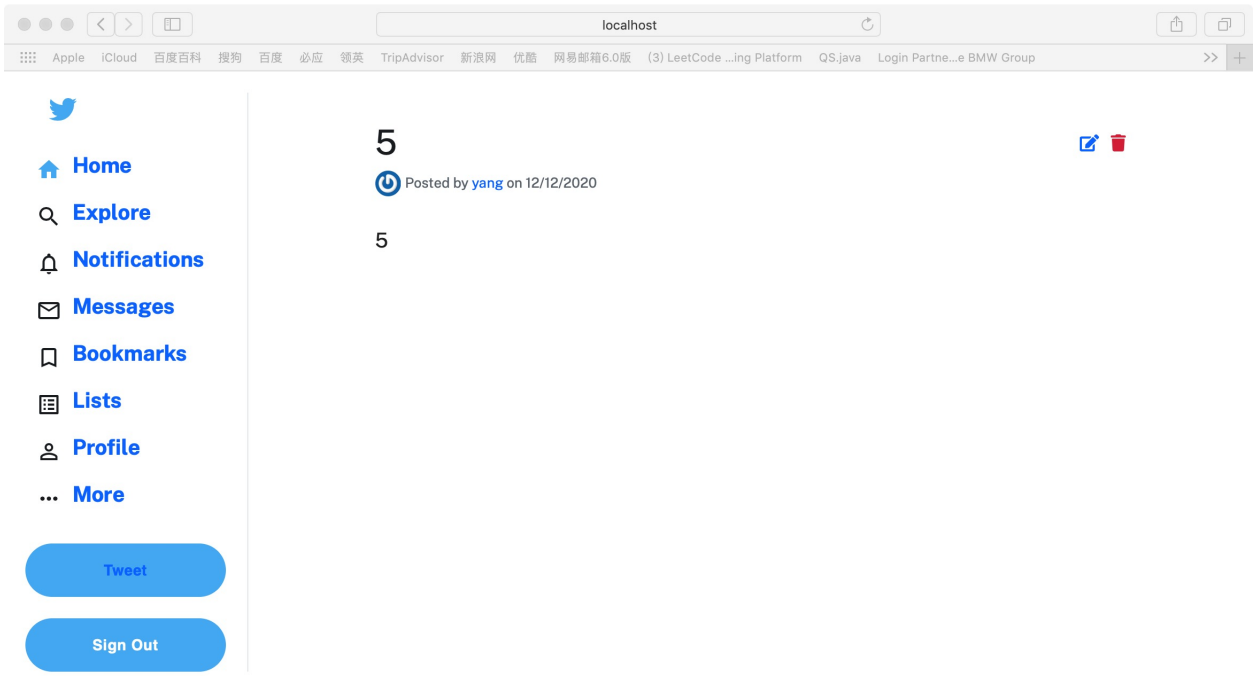


Fourth, we could also search for the tweets posted by the user's followers.


localhost
Apple iCloud 百度百科 搜狗 百度 必应 领英 TripAdvisor 新浪网 优酷 网易邮箱6.0版 (3) LeetCode ...ing Platform QS.java Login Partne...e BMW Group
What are you interested in?
Explore
Notifications
Messages
Bookmarks
Lists
Profile
More
Tweet
Sign Out
Posts: 8 Followers: 3 Following: 2
5 on 12/12/2020
7 on 12/12/2020
8 on 12/12/2020
3 on 12/12/2020
6 on 12/12/2020
6 on 12/12/2020
6 on 12/12/2020
6 on 12/12/2020



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

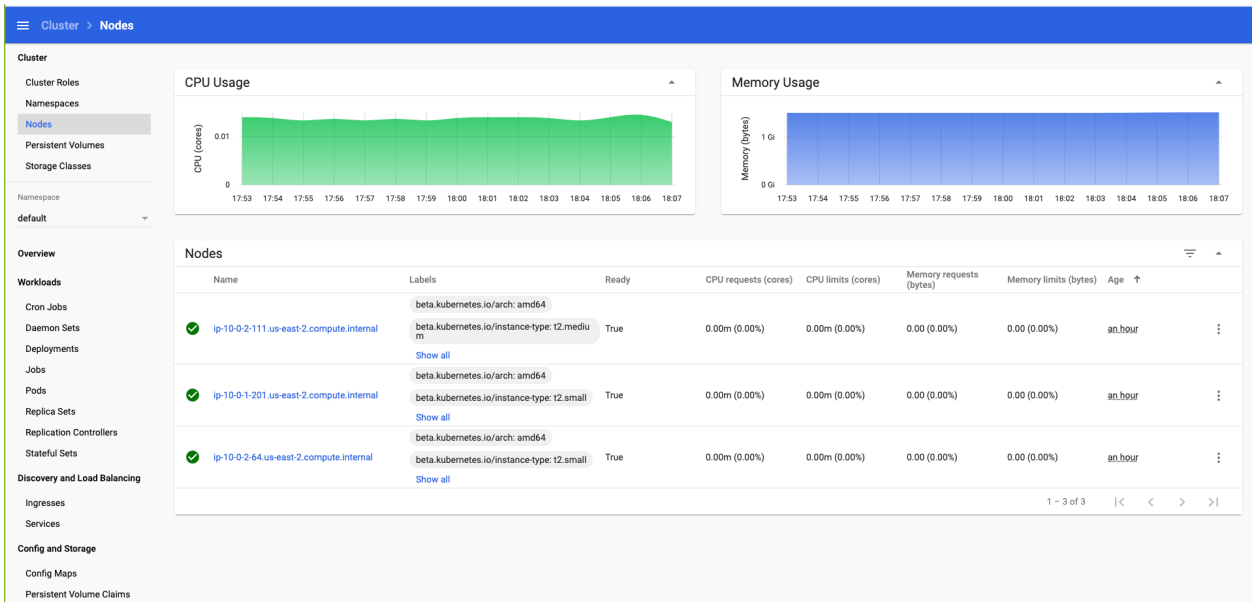
   kubectl apply -f
   https://raw.githubusercontent.com/hashicorp/learn-terraform-provision-eks-cluster/master/kubernetes-dashboard-admin.rbac.yaml

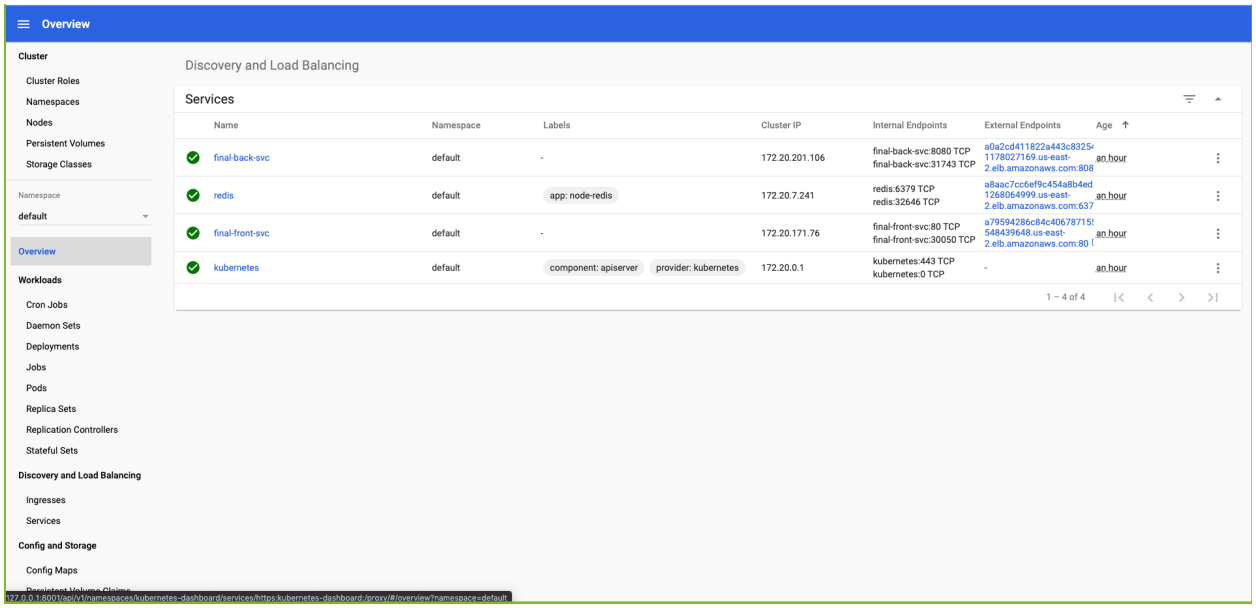
   kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep service-controller-token | awk '{print $1}'))

   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