Food-Flow Testing Instructions (Group 15)

Overview

Food-Flow is a web-based platform designed to address the challenges of food waste and insecurity in Singapore, by facilitating the responsible redistribution of surplus food. It enables organizations to donate typically discarded excess food items to users in need for free, through listings and reservations.

Food-Flow consists of 11 microservices that are orchestrated together using Docker, Kubernetes, and WebSocket. These microservices are split into six front-end microservices that serve as the backbone of the system, with the remaining five microservices catered towards supporting reservation operations.

These microservices include:

- Front-End Microservices:
  - Front-End Microservice (Main UI for Food-Flow platform)
  - Envoy Microservice (For inter-microservice communications)
  - Authentication Microservice (For user authentication and CRUD operations)
  - Listing Microservice (To support food listing operations)
  - Amazon Web Services Microservice (Calls AWS API to access cloud services)
  - Federated Learning Microservice (For distributed machine learning service)
- Reservation Microservices:
  - Kafka Microservice (For handling food reservations)
  - ZooKeeper Microservice (Manages Kafka microservice)
  - React Reservation Microservice (Kafka producer, handles food reservation requests)
  - Reservation Database Microservice (Kafka consumer, backbone of food reservation system)
  - Reservation Socket Microservice (To enable real-time communications)

Presentation and Demo

To see a walkthrough of the Food-Flow system and our presentation, visit this link (https://youtu.be/9H4AzImgLTo), or scan the QR code below!

Prerequisites

- System Requirements

    - This project requires a significant amount of CPU and RAM to run, ensure that your device has at least 8GB of free RAM with a multi-core CPU (at least 3 cores)
- Required Software
    - Ensure these software are installed on the machine you are using to run Food-Flow
        - Docker (https://docs.docker.com/engine/install/)
        - Docker Desktop (https://docs.docker.com/desktop/install/)
        - Kubernetes (https://kubernetes.io/docs/tasks/tools/)
        - Minikube (https://minikube.sigs.k8s.io/docs/start/)
- Used Ports
    - Food-Flow uses some local ports for running the application, ensure that these ports are kept free and exposed in your firewall settings
        - Port 3000 (for Front-End Microservice)
        - Port 9900 (for Envoy Microservice)

Project Usage

> NOTE: Each microservice is not designed to be executed independently as they are dependent on the services provided by one another. However, some microservice may still be executed independently but may not work as intended. To do so, build the `Dockerfile` in the respective microservice's folder, and run the created image.

The entire Food-Flow system has been containerized into a Docker image. To run the Food-Flow system locally on your environment, we will run the main Docker image that contains the required `.yaml` files to create the Kubernetes cluster. Running the entire Food-Flow project is recommended to ensure that everything works as intended. To run Food-Flow, following the instructions below!

To run the Food-Flow system locally on your environment, run the main Docker image that contains the required `.yaml` files to create the Kubernetes cluster.

1. Ensure that your Kubernetes namespace `food-flow` is cleared

    ```
    kubectl delete namespace food-flow
    ```

2. Remove any existing `minikube` configurations, if any

    ```
    minikube delete
    ```

3. To handle its storage needs, a database has been pre-configured and is running on the cloud. However, if you intend to use your own database, follow the following instructions:
    - Create a SQL database
    - Import Food-Flow's database into your new database!

        ```
        mysql -u<username> -p <your database> < foodflow.sql
        ```

    - Modify the `food-flow` Docker image's `secrets.yaml` file with your database connection details (URL, host and password), ensuring that they are all base64 encoded.

3. Run the `food-flow` Docker image!
    - For Windows OS devices

```
docker run -it --rm -v /var/run/docker.sock:/var/run/docker.sock -v $HOME/.kube/config:/root/.kube/config skyish/food-flow:wir
```

- For MacOS devices

```
docker run -it --rm -v /var/run/docker.sock:/var/run/docker.sock -v $HOME/.kube/config:/root/.kube/config skyish/food-flow:mac
```

4. Once an interactive terminal within the Docker container has been obtained, ensure that all the containers in the `food-flow` namespace are running. This process may take a while as the project consists of 11 different microservices that need to be executed together with other load-balancing and scaling operations!

```
# Within Docker container
kubectl get all -n food-flow
```

You should wait until all respective pods are up and running before proceeding. An example of an expected output is provided below:

```
/app # kubectl get all -n food-flow
NAME                                                       READY
pod/aws-s3-listing-deployment-64895d6868-rjm4f             1/1
pod/aws-s3-listing-deployment-64895d6868-s2mkz             1/1
pod/federated-flask-deployment-6d96d5d5f8-6c7pr            1/1
pod/federated-flask-deployment-6d96d5d5f8-tlldh            1/1
pod/kafka-deployment-1-f57fc5fbc-flsxx                     1/1
pod/kafka-deployment-2-6f5769c7c5-kf2sz                    1/1
pod/react-auth-deployment-6b84556ddf-hc64l                 1/1
pod/react-auth-deployment-6b84556ddf-vzctj                 1/1
pod/react-envoy-deployment-6556f5fd57-fzh7x                1/1
pod/react-frontend-deployment-8466d49c9-4vl2x              1/1
pod/react-frontend-deployment-8466d49c9-htb7t              1/1
pod/react-listing-deployment-6fdccb46ff-54th8              1/1
pod/react-listing-deployment-6fdccb46ff-llmdq              1/1
pod/react-reservation-deployment-6bb674fd69-4l85l          1/1
pod/react-reservation-deployment-6bb674fd69-5s4lk          1/1
pod/reservation-database-deployment-7ff47c6cc5-2k4fj       1/1
pod/reservation-database-deployment-7ff47c6cc5-4p2j4       1/1
pod/reservation-socket-deployment-6fd569cb79-7n8r4         1/1
pod/reservation-socket-deployment-6fd569cb79-j2qbg         1/1
pod/zookeeper-deployment-1-7989c6c445-vtxsw                1/1
pod/zookeeper-deployment-2-7989c6c445-p92zn                1/1

NAME                                     TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)
service/aws-s3-listing-service           ClusterIP      10.109.119.33    <none>        5001/TCP
service/federated-flask-service          ClusterIP      10.111.180.72    <none>        80/TCP
service/kafka-service-1                  LoadBalancer   10.104.148.45    localhost     9092:31079/TCP,29092:32098/TCP
service/kafka-service-2                  LoadBalancer   10.106.23.248    localhost     9092:31894/TCP,39092:32645/TCP
service/react-auth-service               ClusterIP      10.110.72.235    <none>        5000/TCP
service/react-envoy-service              LoadBalancer   10.106.152.230   localhost     9900:30804/TCP,9901:30532/TCP
service/react-frontend-service           LoadBalancer   10.96.102.193    localhost     3000:32241/TCP
service/react-listing-service            ClusterIP      10.99.112.80     <none>        5002/TCP
service/react-reservation-service        ClusterIP      10.100.44.246    <none>        5003/TCP
service/reservation-socket-service       LoadBalancer   10.107.145.32    localhost     8282:30626/TCP
service/zookeeper-service-1              LoadBalancer   10.101.138.251   localhost     22181:31424/TCP
service/zookeeper-service-2              LoadBalancer   10.100.180.172   localhost     32181:32064/TCP

NAME                                              READY   UP-TO-DATE   AVAILABLE
deployment.apps/aws-s3-listing-deployment         2/2     2            2
deployment.apps/federated-flask-deployment        2/2     2            2
deployment.apps/kafka-deployment-1                1/1     1            1
deployment.apps/kafka-deployment-2                1/1     1            1
deployment.apps/react-auth-deployment             2/2     2            2
deployment.apps/react-envoy-deployment            1/1     1            1
deployment.apps/react-frontend-deployment         2/2     2            2
deployment.apps/react-listing-deployment          2/2     2            2
deployment.apps/react-reservation-deployment      2/2     2            2
deployment.apps/reservation-database-deployment   2/2     2            2
deployment.apps/reservation-socket-deployment     2/2     2            2
deployment.apps/zookeeper-deployment-1            1/1     1            1
deployment.apps/zookeeper-deployment-2            1/1     1            1

NAME                                                         DESIRED   CURRENT   READY
replicaset.apps/aws-s3-listing-deployment-64895d6868         2         2         2
replicaset.apps/federated-flask-deployment-6d96d5d5f8        2         2         2
replicaset.apps/kafka-deployment-1-f57fc5fbc                 1         1         1
replicaset.apps/kafka-deployment-2-6f5769c7c5                1         1         1
replicaset.apps/react-auth-deployment-6b84556ddf              2         2         2
replicaset.apps/react-envoy-deployment-6556f5fd57             1         1         1
replicaset.apps/react-frontend-deployment-8466d49c9           2         2         2
replicaset.apps/react-listing-deployment-6fdccb46ff           2         2         2
replicaset.apps/react-reservation-deployment-6bb674fd69       2         2         2
replicaset.apps/reservation-database-deployment-7ff47c6cc5    2         2         2
replicaset.apps/reservation-socket-deployment-6fd569cb79      2         2         2
replicaset.apps/zookeeper-deployment-1-7989c6c445             1         1         1
replicaset.apps/zookeeper-deployment-2-7989c6c445             1         1         1

NAME                                                            REFERENCE                                TARGETS   MINPODS   MAXPODS
horizontalpodautoscaler.autoscaling/federated-flask-deployment  Deployment/federated-flask-deployment   0%/80%    2         5
horizontalpodautoscaler.autoscaling/react-frontend-deployment   Deployment/react-frontend-deployment    21%/80%   2         8
```

When running on a Mac device, you may encounter issues with Kubernetes' API as follows:

> **E1110 11:28:05.888283   47 memcache.go:265] couldn't get current server API group list: Get "http://localhost:8080/api?timeout=32s": dial tcp 127.0.0.1:8080 : connect: connection refused**
> **E1110 11:28:05.892096   47 memcache.go:265] couldn't get current server API group list: Get "http://localhost:8080/api?timeout=32s": dial tcp 127.0.0.1:8080 : connect: connection refused**

To fix this issue, perform the following steps:

- Edit the configuration in `/root/.kube/config`` (VIM is installed in the image)

```
vi /root/.kube/config
```

- Edit the "server" option in `/root/.kube/config` to the below:

```
server: https://kubernetes.docker.internal:6443
```

- Save and test the configuration:

```
kubectl get all -n food-flow
```

5. Enter the web application by opening up `http://localhost:3000` in your browser
6. Enjoy making a food listing and reserving food listings on Food-Flow!

Testing Instructions

Food-Flow has several functions that can be tested, as follows!

Authentication Microservice

The authentication microservice acts as a `gRPC` server which will receive translated web traffic from Envoy.

- In `/auth/login` : You can log in as either a donor or patron, with the following credentials:

```
Username: donorExample
Password: password123

Username: patronExample
Password: password123
```

- In `/auth/register` : You can register as a new user, which will be created in the database.
- Upon log in, in `/user-profile` : You can update your user details.

Logged in as Donor (Listing, Federated Learning, AWS Microservice)

The two main functions as a donor include:

1. My Listings

- In the "My Listings" page, listings will be fetched from the listing microservice, which will be filtered in the server based on availability statuses. Fetched listings will also fetch the image data from the AWS microservice, for viewing. You can:
    - View your available listings, to update or delete in the "view listings" page
    - View your reserved listings, to click "reserved", this will update the status in the database through the listing microservice.
    - View your collected listings.

2. Donate

In the donate page, the application will fetch the client model from the global server on rendering. This client model will classify the images based on freshness later on.

- Upload an image to begin. The first step verification occurs after clicking `Verify Image` , the image will be passed to the AWS microservice via gRPC, which will upload the image to an S3 bucket for verification. If the food is categorized under "Food and Beverages", a valid response will be returned.
    - Try uploading a food image and a non food image!
- If the verification of food succeeded, the second step is to classify the freshness (for food safety). The client model will classify the uploaded image based on "Fresh" or "Rotten", which will be shown on the frontend. The confidence level will also be shown in your web console. Make sure to wait a while to allow Tensorflow.js to work its magic behind the scenes!
    - If flagged as rotten, click "Looks fresh to me" to dispute. This will retrain the client model (**UNSAFE→SAFE**) , and upload the model and weights to the global server for federated averaging.
    - Try uploading a fresh and rotten food image!
- If the all food verifications succeeded, click next to populate your food details. When completed, the food listing details will be sent to the listing microservice which will store the listing into the database.

Logged in as patron (Reservation Cluster, Kafka, Federated Learning, Listing and AWS microservices)

The two main functions as patron include:

1. Available Listings

- In the available listings page, a WebSocket with the reservations database is established. Listings are fetched similar to how it was fetched for donors except a different `gRPC` method is called to fetch listings that do not belong to the user.
- You can view listings, reserve a listing, or report a listing.
    - When reserving a listing
        1. A `POST` reservations request will be sent to the reservation microservice, which acts as a producer.
        2. The microservice will then form a payload that is sent to the Kafka queue.
        3. The reservation database microservice acts as a consumer that consumes the payload to create the reservation details into the database.
        4. The outcome details will be passed from the websocket to the frontend service.
        5. Upon receiving a message from the websocket, a notification will be shown displaying a message that the reservation has been created.
    - "Reporting" a listing trains the client model (**SAFE→UNSAFE)** and uploads the model and weights to the global server for federated averaging

2. Reserved Listings

- Upon entering the reserved listings page, the same flow as the `POST` request is executed, but instead a `GET` request is sent. From the reservation details, listings will be fetched from the Listing microservice and images will be fetched from the AWS microservice.

- You can view listings or cancel a reservation.
    - When cancelling a reservation, a `DELETE` request is sent the reservation microservice, and the same flow of events is executed. When receiving a message from the websocket, a notification will be displayed.