GROUP 10 ***-***
Sushma Tacholi Kudai (017519060)
Shobhita Agrawal (017552795)
Kush Bindal (017441359)

Github link - https://github.com/sushma1311/lab2

LAB2

Part 1 -
Integrating Docker and Kubernetes with Lab1 Code

1. Dockerizing Backend and Frontend Applications

Created separate Dockerfiles for the backend (Django) and frontend (React):

- **Backend**:
  - Used a Python base image to set up the Django backend.
  - Installed all dependencies from requirements.txt.
  - Exposed port 8000 for the backend server.
  - Ran the Django development server inside the container.
- **Frontend**:
  - Used Node.js as the base image to set up the React frontend.
  - Installed dependencies, built the app, and served static files using serve.
  - Exposed port 3000.

2. Managing Backend and Frontend with Docker Compose
To streamline running both the backend and frontend locally, we used a docker-compose.yml file to define services for both applications:

- **Backend**: Configured to run on port 8000.
- **Frontend**: Configured to run on port 3000 and connected to the backend using the API URL.

Commands Used:
Build and start services:
docker-compose up --build
Verify containers are running:
docker ps

3. Deploying Backend with Kubernetes
We set up Kubernetes configurations for the backend to manage its deployment.

- **Deployment**:
  - Created a backend-deployment.yml file to define the backend pod and its container.

    - Used the Docker image of the backend and exposed port 8000.
    - Ensured consistent running of the container.
- **Service**:
    - Created a backend-service.yml file to expose the backend deployment via a Kubernetes NodePort.

<u>**Commands Used**</u>:

Start minikube:

```
minikube start
```

Enable Docker's local Kubernetes environment:

```
eval $(minikube docker-env)
```

Build the Docker image for the backend and load it into Minikube:

```
docker build -t backend:latest ./backend
minikube image load backend:latest
```

Deploy the backend using Kubernetes:

```
kubectl apply -f backend-deployment.yml
kubectl apply -f backend-service.yml
```

Check the status of pods and services:

```
kubectl get pods
kubectl get services
```

Start Kubernetes backed:

```
minikube service backend-service
```

Integration of Frontend and Backend:

Get the url from backend-service and use it in .env.

The frontend, running locally, communicated with the backend using the environment variable in the .env file:

```
REACT_APP_API_BASE_URL=backend url here
```

Start the frontend:

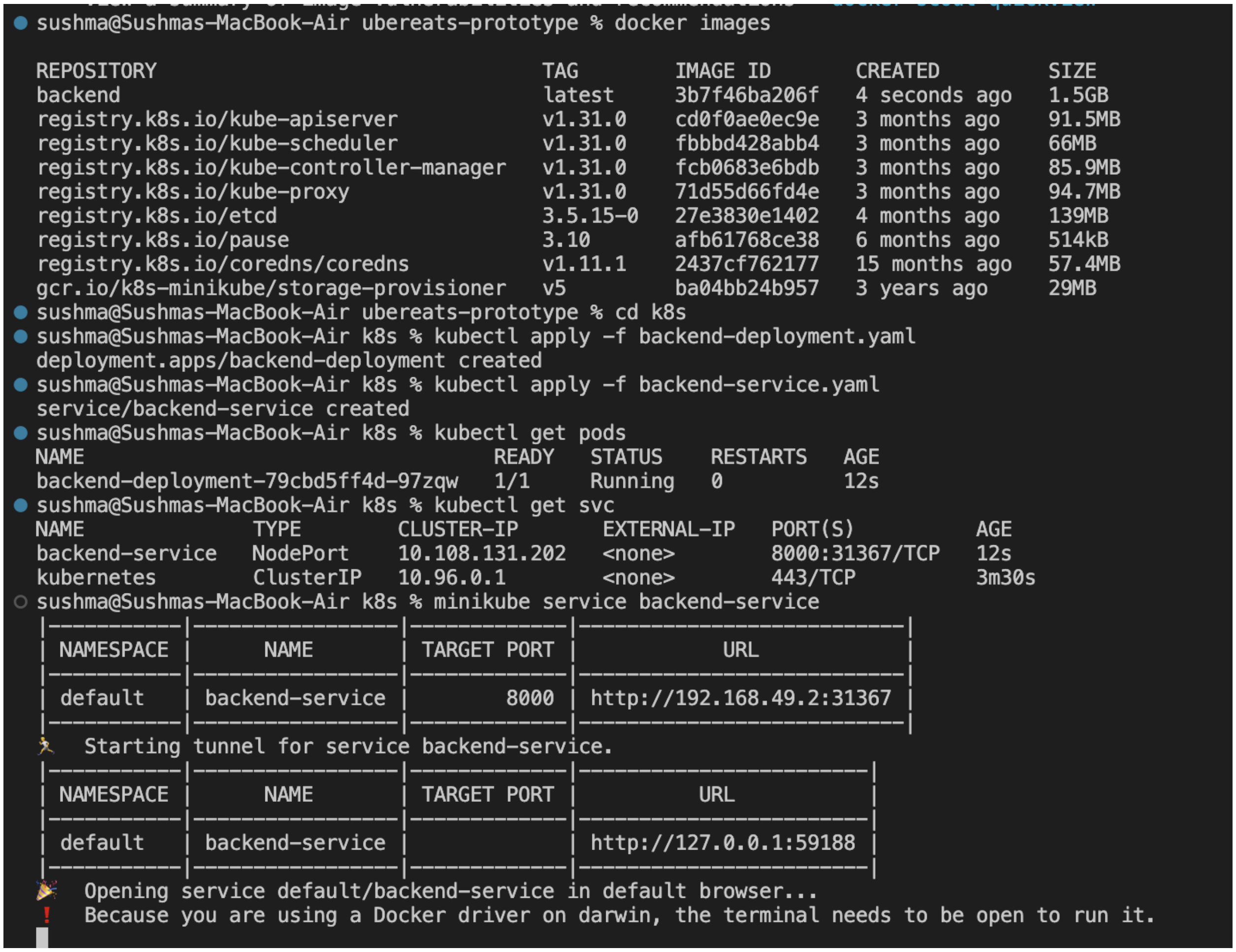
```
npm start
```

Verified communication with the backend by testing login,signup and order management functionalities.

<u>Code files in github</u> -

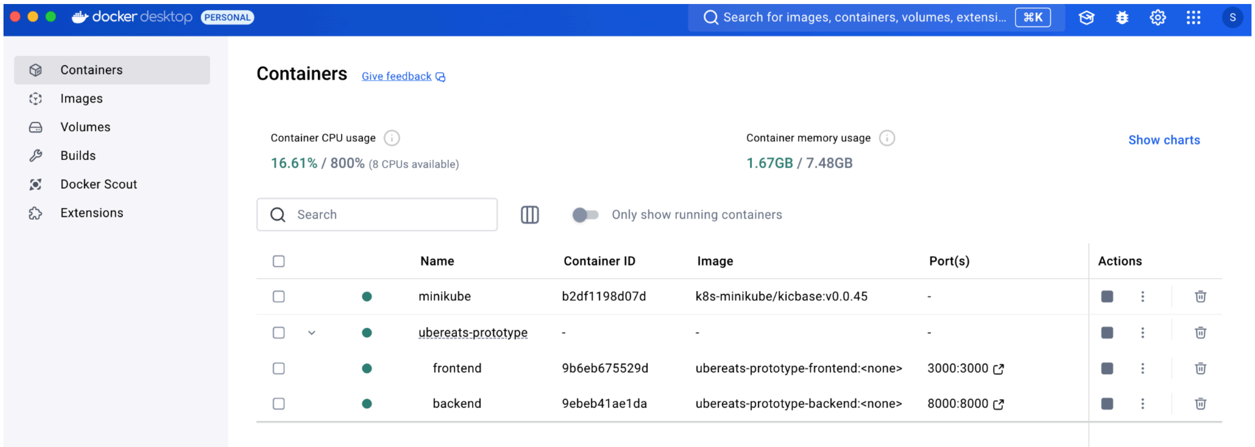
backend / Dockerfile
frontend / Dockerfile
docker-compose.yaml file

In minikube



```
● sushma@Sushmas-MacBook-Air ubereats-prototype % docker images

  REPOSITORY                               TAG        IMAGE ID       CREATED          SIZE
  backend                                  latest     3b7f46ba206f   4 seconds ago    1.5GB
  registry.k8s.io/kube-apiserver           v1.31.0    cd0f0ae0ec9e   3 months ago     91.5MB
  registry.k8s.io/kube-scheduler           v1.31.0    fbbbd428abb4   3 months ago     66MB
  registry.k8s.io/kube-controller-manager  v1.31.0    fcb0683e6bdb   3 months ago     85.9MB
  registry.k8s.io/kube-proxy               v1.31.0    71d55d66fd4e   3 months ago     94.7MB
  registry.k8s.io/etcd                     3.5.15-0   27e3830e1402   4 months ago     139MB
  registry.k8s.io/pause                    3.10       afb61768ce38   6 months ago     514kB
  registry.k8s.io/coredns/coredns          v1.11.1    2437cf762177   15 months ago    57.4MB
  gcr.io/k8s-minikube/storage-provisioner  v5         ba04bb24b957   3 years ago      29MB
● sushma@Sushmas-MacBook-Air ubereats-prototype % cd k8s
● sushma@Sushmas-MacBook-Air k8s % kubectl apply -f backend-deployment.yaml
  deployment.apps/backend-deployment created
● sushma@Sushmas-MacBook-Air k8s % kubectl apply -f backend-service.yaml
  service/backend-service created
● sushma@Sushmas-MacBook-Air k8s % kubectl get pods
  NAME                                  READY   STATUS    RESTARTS   AGE
  backend-deployment-79cbd5ff4d-97zqw   1/1     Running   0          12s
● sushma@Sushmas-MacBook-Air k8s % kubectl get svc
  NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
  backend-service   NodePort    10.108.131.202   <none>        8000:31367/TCP   12s
  kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP          3m30s
○ sushma@Sushmas-MacBook-Air k8s % minikube service backend-service
  |-----------|-----------------|-------------|---------------------------|
  | NAMESPACE |      NAME       | TARGET PORT |            URL            |
  |-----------|-----------------|-------------|---------------------------|
  | default   | backend-service |        8000 | http://192.168.49.2:31367 |
  |-----------|-----------------|-------------|---------------------------|
  🏃  Starting tunnel for service backend-service.
  |-----------|-----------------|-------------|------------------------|
  | NAMESPACE |      NAME       | TARGET PORT |          URL           |
  |-----------|-----------------|-------------|------------------------|
  | default   | backend-service |             | http://127.0.0.1:59188 |
  |-----------|-----------------|-------------|------------------------|
  🎉  Opening service default/backend-service in default browser...
  ❗  Because you are using a Docker driver on darwin, the terminal needs to be open to run it.
```



Containers

Container CPU usage
16.61% / 800% (8 CPUs available)

Container memory usage
1.67GB / 7.48GB

□		Name	Container ID	Image	Port(s)	Actions
□	●	minikube	b2df1198d07d	k8s-minikube/kicbase:v0.0.45	-	
□	●	ubereats-prototype	-	-	-	
□	●	frontend	9b6eb675529d	ubereats-prototype-frontend:<none>	3000:3000	
□	●	backend	9ebeb41ae1da	ubereats-prototype-backend:<none>	8000:8000	

We also pushed our images to docker hub

Tag images:

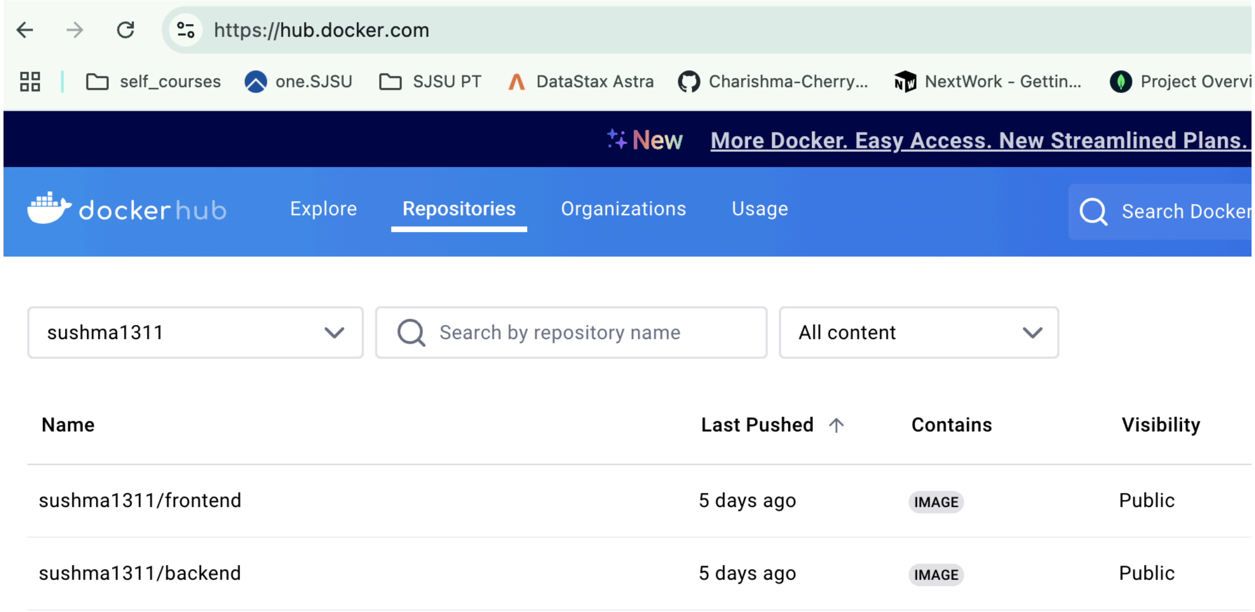
```
docker tag ubereats-prototype-backend sushma1311/backend:latest
docker tag ubereats-prototype-frontend sushma1311/frontend:latest
```

Log in to Docker Hub:

```
docker login
```

Push images:

```
docker push sushma1311/backend:latest
docker push sushma1311/frontend:latest
```



Name	Last Pushed	Contains	Visibility
sushma1311/frontend	5 days ago	IMAGE	Public
sushma1311/backend	5 days ago	IMAGE	Public

Part 2 -

Integrate Kafka for publishing and consuming events related to order processing. (locally)

1. Installed Kafka and Python Dependencies

```
pip install confluent-kafka
```

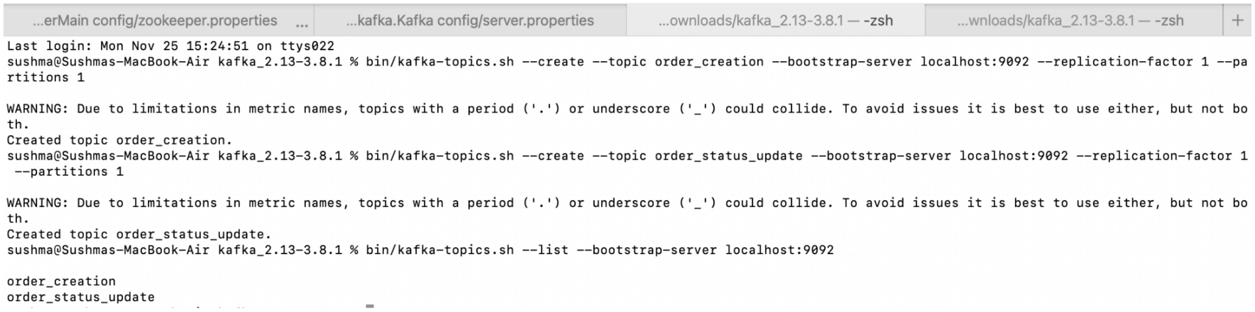
Started Zookeeper:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Started kafka server:

```
bin/kafka-server-start.sh config/server.properties
```

Created topics:


...erMain config/zookeeper.properties ...
...kafka.Kafka config/server.properties
...ownloads/kafka_2.13-3.8.1 — -zsh
...wnloads/kafka_2.13-3.8.1 — -zsh
+

```
Last login: Mon Nov 25 15:24:51 on ttys022
sushma@Sushmas-MacBook-Air kafka_2.13-3.8.1 % bin/kafka-topics.sh --create --topic order_creation --bootstrap-server localhost:9092 --replication-factor 1 --pa
rtitions 1

WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issues it is best to use either, but not bo
th.
Created topic order_creation.
sushma@Sushmas-MacBook-Air kafka_2.13-3.8.1 % bin/kafka-topics.sh --create --topic order_status_update --bootstrap-server localhost:9092 --replication-factor 1
 --partitions 1

WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issues it is best to use either, but not bo
th.
Created topic order_status_update.
sushma@Sushmas-MacBook-Air kafka_2.13-3.8.1 % bin/kafka-topics.sh --list --bootstrap-server localhost:9092

order_creation
order_status_update
```

2. Publishing Events to Kafka

In the views.py file, added the following:

Created a Kafka Producer:

Created a function kafka_producer that initializes and returns a Kafka producer. This producer connects to the Kafka server.

```
def kafka_producer():
return Producer({'bootstrap.servers': 'localhost:9092'})
```

Implemented publish_order_event:

This function takes a Kafka topic (order_creation) and a dictionary of event data, converts it to JSON, and sends it to the specified Kafka topic.

```
def publish_order_event(topic, order_data):
    producer = kafka_producer()
    producer.produce(topic, json.dumps(order_data))
    producer.flush()
```

Integrated Kafka Publishing in the Order Workflow:

In the FinalizeOrderView class, after an order is finalized, call the publish_order_event function to publish the event to Kafka.

```
order_event = {
    "order_id": order.id,
    "restaurant_id": order.restaurant.id,
    "user_id": user.id,
    "items": order.items,
    "total_price": str(order.total_price),
    "delivery_option": order.delivery_option,
    "delivery_address": order.delivery_address.address if delivery_address else None
}
publish_order_event("order_creation", order_event)
```

When an order is placed by the user, the order data is stored in the database. After storing, the order details are published to the Kafka topic order_creation for further processing.

3. Consuming Events from Kafka

In the order_consumer.py file, implemented the following:

Created a Kafka Consumer:
Initialized a Kafka consumer with configurations to connect to the Kafka broker, specify a consumer group, and set the offset reset policy.

```
def kafka_consumer():
    return Consumer({
        'bootstrap.servers': 'localhost:9092',
        'group.id': 'restaurant-service',
        'auto.offset.reset': 'earliest'
    })
```

Implemented Event Handlers:

Two functions were added to handle different types of events:

1. **process_order_event**:

Processes events from the order_creation topic (e.g., when an order is created).
Fetches the order and restaurant details.
Logs the event and ensures the status is handled properly ('New').

2. **process_order_status_update_event**:

Processes events from the order_status_update topic (when an order status is updated).
Updates the order status in the database.

```
def process_order_event(event_data):

    try:
        order_id = event_data.get('order_id')
        restaurant_id = event_data.get('restaurant_id')
        order = Order.objects.get(id=order_id)
        restaurant = Restaurant.objects.get(id=restaurant_id)
        print(f"Order Received: ID {order_id}, Restaurant {restaurant.name}")
        # Ensure status remains 'New'
        if order.order_status == 'New':
            print(f"Order {order_id} remains '{order.order_status}'.")
        order.save()
    except Exception as e:
        print(f"Error processing order: {str(e)}")

def process_order_status_update_event(event_data):
    try:
        order_id = event_data.get('order_id')
        new_status = event_data.get('status')
        order = Order.objects.get(id=order_id)
        order.order_status = new_status
        order.save()
        print(f"Order {order_id} status updated to {new_status}")
    except Exception as e:
        print(f"Error processing order status update: {str(e)}")
```

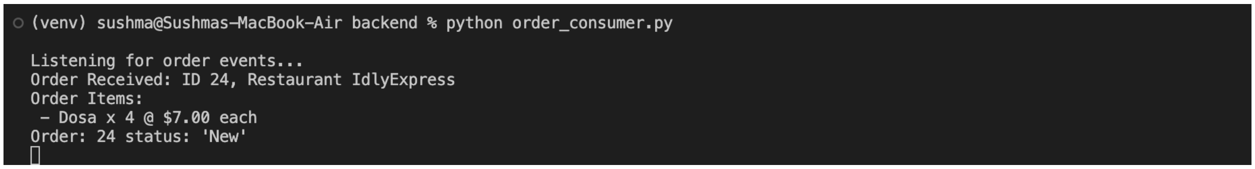
Created the Consumer Loop:

A function consume_order_events listens for events on the Kafka topics order_creation and order_status_update.
Based on the topic, it calls the respective processing function.

```
def consume_order_events():
    consumer = kafka_consumer()
    consumer.subscribe(['order_creation', 'order_status_update'])
    while True:
        message = consumer.poll(1.0)
        if message is None:
            continue
        if message.error():
            print(f"Consumer error: {message.error()}")
            continue
        topic = message.topic()
        event_data = json.loads(message.value().decode('utf-8'))
        if topic == 'order_creation':
            process_order_event(event_data)
        elif topic == 'order_status_update':
            process_order_status_update_event(event_data)
        consumer.commit()
```

The consumer listens to events on order_creation and order_status_update topics. When a message is received, it is processed and stored in the database.

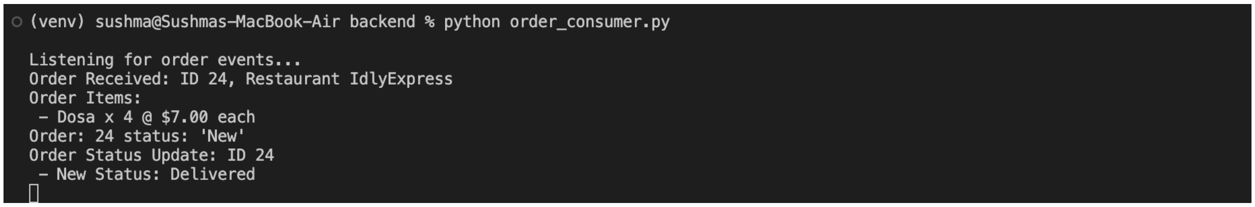
Order creation:



```
○ (venv) sushma@Sushmas-MacBook-Air backend % python order_consumer.py

  Listening for order events...
  Order Received: ID 24, Restaurant IdlyExpress
  Order Items:
   - Dosa x 4 @ $7.00 each
  Order: 24 status: 'New'
```

Order status updates:



```
○ (venv) sushma@Sushmas-MacBook-Air backend % python order_consumer.py

  Listening for order events...
  Order Received: ID 24, Restaurant IdlyExpress
  Order Items:
   - Dosa x 4 @ $7.00 each
  Order: 24 status: 'New'
  Order Status Update: ID 24
   - New Status: Delivered
```

<u>Part 3</u> -

Integrated **MongoDB** as the database for the backend, ensuring secure storage of data and session information.

1. Database Configuration

- Used MongoDB Atlas (a cloud-based database service) to host the database.
- Created a cluster on MongoDB Atlas and Added connection credentials to Django's settings.py.
- Configured the database connection in the settings.py file:

```
DATABASES = {

  'default': {
     'ENGINE': 'djangomi',
     'NAME': 'ubereats',  # Your database name
     'CLIENT': {
        'host':
'mongodb+srv://<username>:<password>@<cluster-url>/ubereats?retryWrites=true&w=majority',
        'username': '<your-username>',
        'password': '<your-password>',
        'authSource': 'admin',
     }
  }
}
```

- Used MongoDB to store not only data but also **sessions** by enabling the session engine:
  ```
  SESSION_ENGINE = 'django.contrib.sessions.backends.db'
  ```
- Installed django and pymongo packages to integrate MongoDB with Django and few other for password hashing
  ```
  pip install django pymongo dnspython bcrypt
  ```

2. Password Encryption

Configured Django to securely store encrypted passwords in MongoDB using hashing:

```
PASSWORD_HASHERS = [
   'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
   'django.contrib.auth.hashers.Argon2PasswordHasher',
   'django.contrib.auth.hashers.PBKDF2PasswordHasher',
   'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
]
```

During user signup, passwords were hashed automatically by Django's authentication system before being saved to MongoDB.

3. Storing Sessions in MongoDB

- Django was configured to use MongoDB to store user sessions. This enabled session-based authentication and persistence of user login sessions.

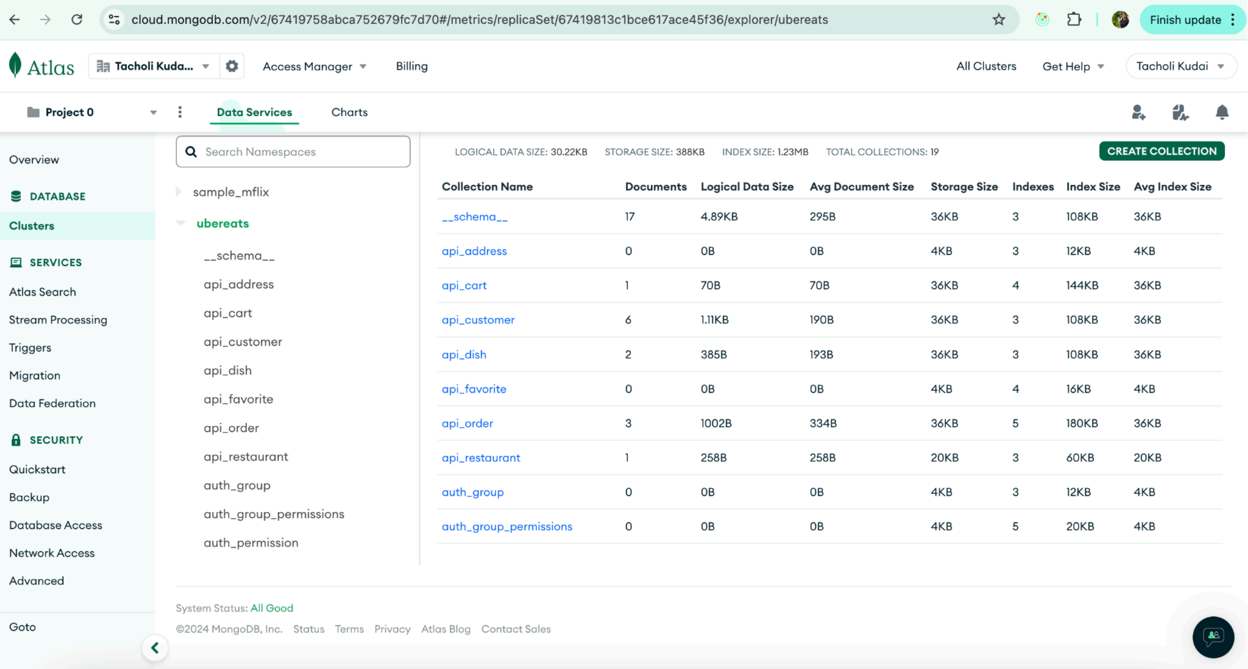
Configured in settings.py:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.db'
```

After making all changes in the backend settings.py

```
python manage.py makemigrations
python manage.py migrate
Python mange.py runserver
```

Verified that the database was correctly integrated by checking the data in the MongoDB Atlas cluster dashboard.

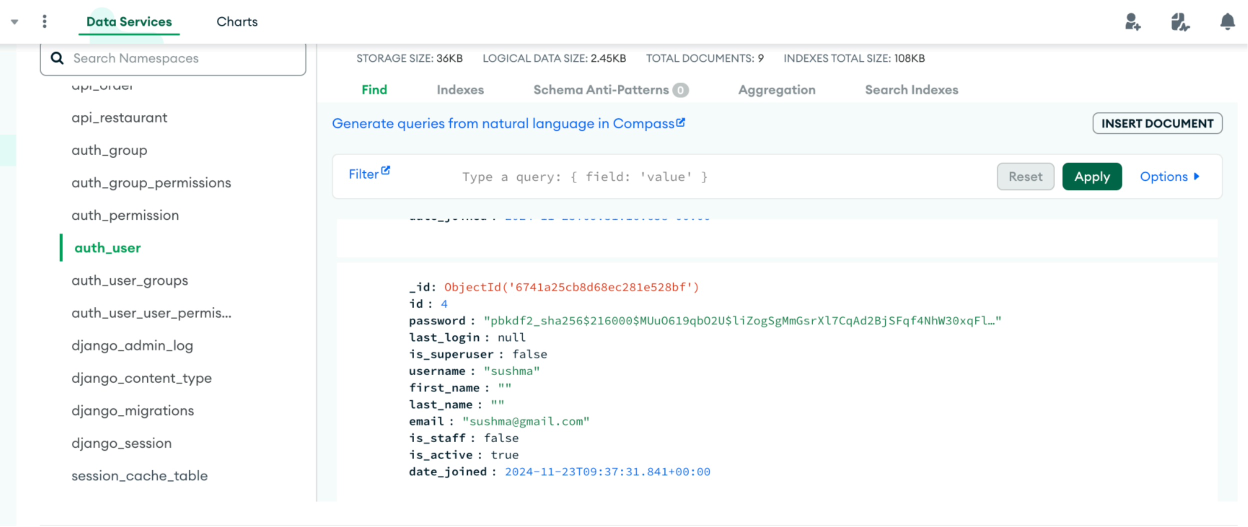


LOGICAL DATA SIZE: 30.22KB STORAGE SIZE: 388KB INDEX SIZE: 1.23MB TOTAL COLLECTIONS: 19

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
__schema__	17	4.89KB	295B	36KB	3	108KB	36KB
api_address	0	0B	0B	4KB	3	12KB	4KB
api_cart	1	70B	70B	36KB	4	144KB	36KB
api_customer	6	1.11KB	190B	36KB	3	108KB	36KB
api_dish	2	385B	193B	36KB	3	108KB	36KB
api_favorite	0	0B	0B	4KB	4	16KB	4KB
api_order	3	1002B	334B	36KB	5	180KB	36KB
api_restaurant	1	258B	258B	20KB	3	60KB	20KB
auth_group	0	0B	0B	4KB	3	12KB	4KB
auth_group_permissions	0	0B	0B	4KB	5	20KB	4KB

Tested signup and login functionalities. Data was successfully inserted into the MongoDB database.

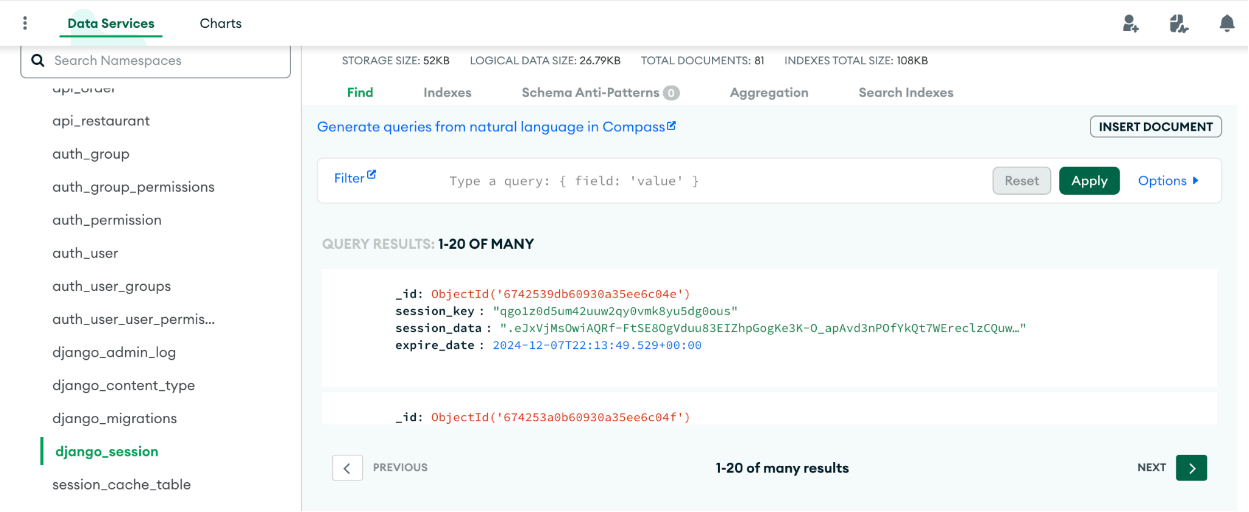
Passwords were securely hashed and stored.



STORAGE SIZE: 36KB LOGICAL DATA SIZE: 2.45KB TOTAL DOCUMENTS: 9 INDEXES TOTAL SIZE: 108KB

```
_id: ObjectId('6741a25cb8d68ec281e528bf')
id : 4
password : "pbkdf2_sha256$216000$MUuO619qbO2U$liZogSgMmGsrXl7CqAd2BjSFqf4NhW30xqFl…"
last_login : null
is_superuser : false
username : "sushma"
first_name : ""
last_name : ""
email : "sushma@gmail.com"
is_staff : false
is_active : true
date_joined : 2024-11-23T09:37:31.841+00:00
```

Sessions were created and persisted in MongoDB.



STORAGE SIZE: 52KB LOGICAL DATA SIZE: 26.79KB TOTAL DOCUMENTS: 81 INDEXES TOTAL SIZE: 108KB

QUERY RESULTS: 1-20 OF MANY

```
_id: ObjectId('6742539db60930a35ee6c04e')
session_key : "qgo1z0d5um42uuw2qy0vmk8yu5dg0ous"
session_data : ".eJxVjMsOwiAQRf-FtSE8OgVduu83EIZhpGogKe3K-O_apAvd3nPOfYkQt7WEreclzCQuw…"
expire_date : 2024-12-07T22:13:49.529+00:00
```

```
_id: ObjectId('674253a0b60930a35ee6c04f')
```

1-20 of many results

Updated backend code to make a few fields compatible with django. (DecimalField)

Part 4 -

Redux has been integrated into the React frontend to manage application-wide states such as user authentication, restaurant data.

In frontend folder :
npm install @reduxjs/toolkit react-redux redux-thunk

frontend/src/store/store.js
A Redux store was created in the file store.js using @reduxjs/toolkit's configureStore method. The store manages application states through reducers and middleware.

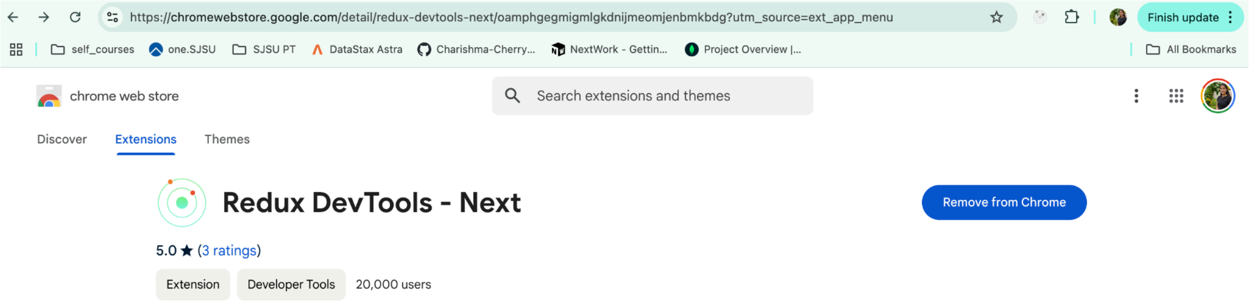
frontend/src/store/counterReducer.js
A reducer function is implemented to handle actions for User session token, Restaurant list and menu, Orders and order updates.

frontend/src/store/reducer.js
All reducers were combined into a single root reducer using combineReducers in reducer.js

frontend/src/store/selector.js
useSessionToken is a custom selector that fetches the sessionToken from the Redux store. This is used across components to verify user authentication

Redux DevTools is utilized to debug and monitor the Redux state in real-time.

1. Installed the Redux DevTools browser extension.
2. Integrate it with your Redux store using configureStore (already handled automatically by Redux Toolkit).



https://chromewebstore.google.com/detail/redux-devtools-next/oamphgegmigmlgkdnijmeomjenbmkbdg?utm_source=ext_app_menu

Redux DevTools - Next

Remove from Chrome

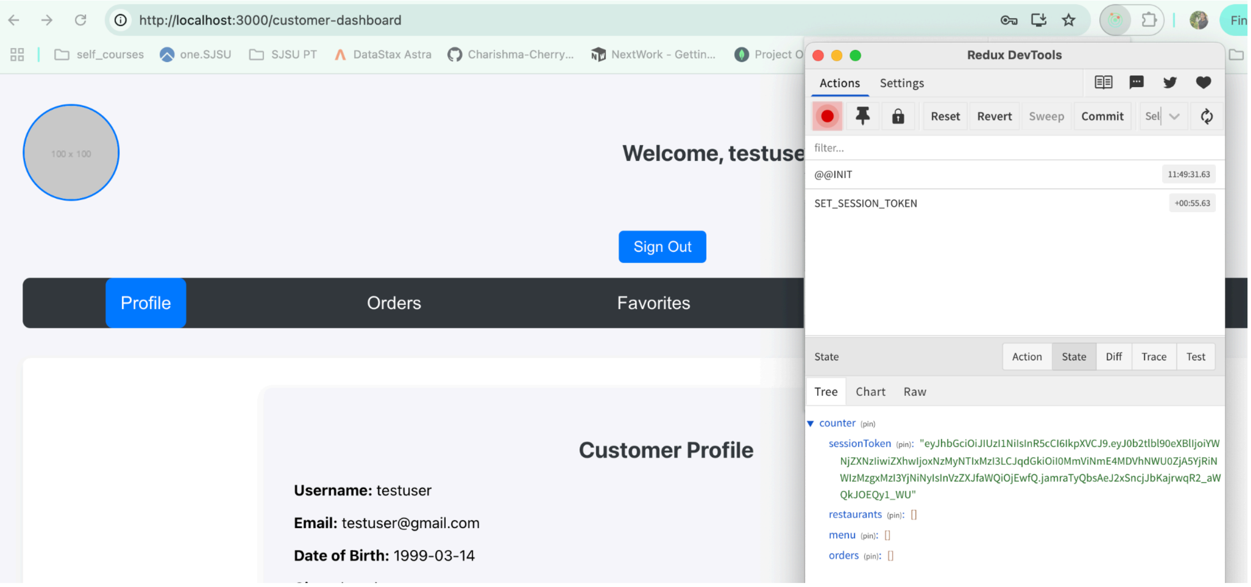
5.0 ★ (3 ratings)

Extension | Developer Tools | 20,000 users

- Created a central Redux store with an initial state containing properties like sessionToken, restaurants, menu, and orders.
- Defined a restaurantReducer to handle actions for setting and updating these properties in the state.
- Created action cases like SET_SESSION_TOKEN, SET_RESTAURANTS, SET_MENU, and UPDATE_ORDER_STATUS to manage state updates.
- Defined selectors like useSessionToken to access specific parts of the Redux state
- Dispatched actions to update the Redux store and used selectors to fetch data from the Redux store on different pages. Specifically handled user authentication (session token) and restaurant data management (menu and orders).

Used dispatch to store the session token in the Redux store when the user logs in.
Fetched the session token from the Redux store using useSessionToken for authentication.

```
dispatch({ type: 'SET_SESSION_TOKEN', payload: response.data.access});
```



Welcome, testuse

Customer Profile

Username: testuser

Email: testuser@gmail.com

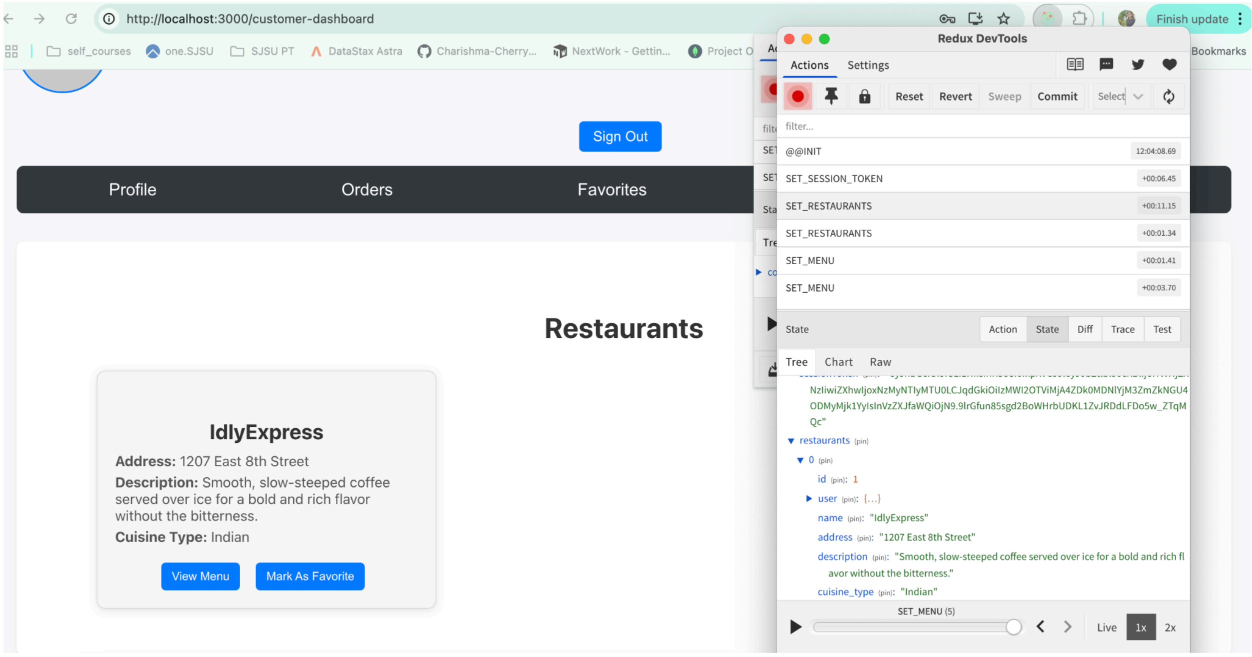
Date of Birth: 1999-03-14

```
@@INIT              11:49:31.63
SET_SESSION_TOKEN   +00:55.63
```

```
▼ counter (pin)
    sessionToken (pin): "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYW
      NjZXNzIiwiZXhwIjoxNzMyNTIxMzI3LCJqdGkiOiI0MmViNmE4MDVhNWU0ZjA5YjRiN
      WIzMzgxMzI3YjNiNyIsInVzZXJfaWQiOjEwfQ.jamraTyQbsAeJ2xSncjJbKajrwqR2_aW
      QkJOEQy1_WU"
    restaurants (pin): []
    menu (pin): []
    orders (pin): []
```

Dispatched the SET_RESTAURANTS action to store the fetched list of restaurants in the Redux state. Fetched the restaurants from the Redux state to display on the home page.

```
dispatch({ type: 'SET_RESTAURANTS', payload: response.data });
```



Restaurants

IdlyExpress

Address: 1207 East 8th Street

Description: Smooth, slow-steeped coffee served over ice for a bold and rich flavor without the bitterness.

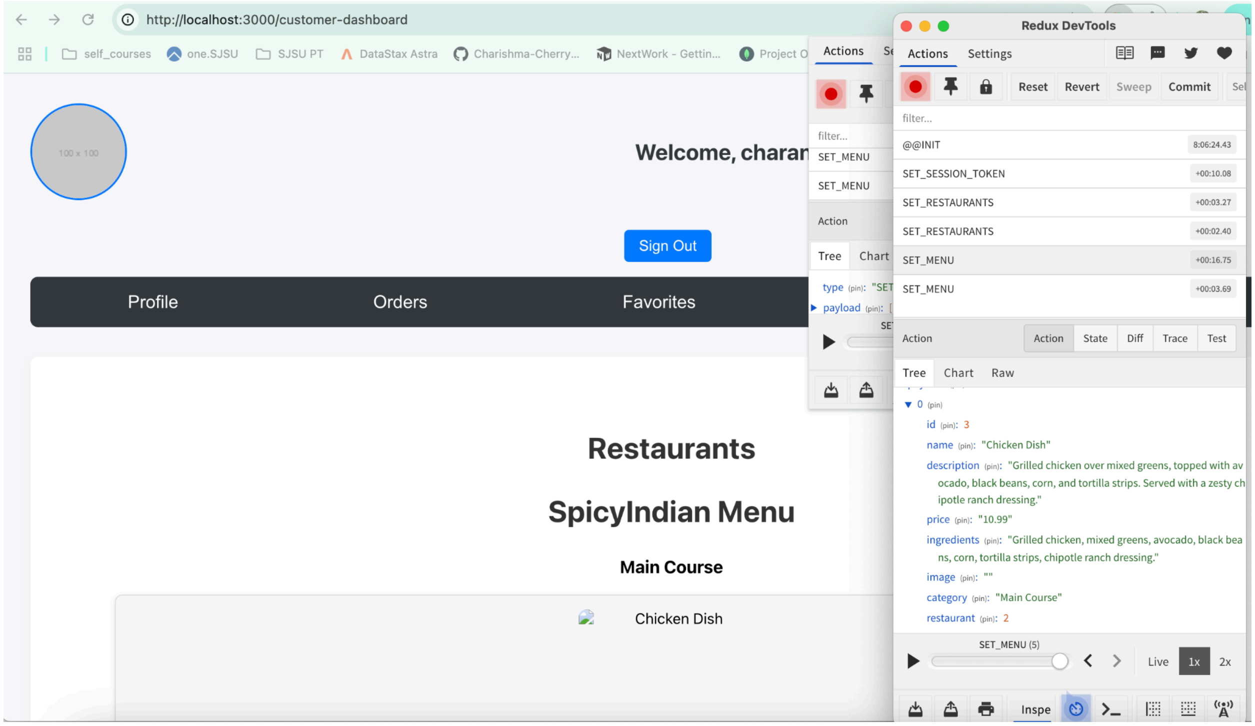
Cuisine Type: Indian

```
@@INIT              12:04:08.69
SET_SESSION_TOKEN   +00:06.45
SET_RESTAURANTS     +00:11.15
SET_RESTAURANTS     +00:01.34
SET_MENU            +00:01.41
SET_MENU            +00:03.70
```

```
      NzIiwiZXhwIjoxNzMyNTIyMTU0LCJqdGkiOiIzMWI2OTViMjA4ZDk0MDNlYjM3ZmZkNGU4
      ODMyMjk1YyIsInVzZXJfaWQiOjN9.9lrGfun85sgd2BoWHrbUDKL1ZvJRDdLFDo5w_ZTqM
      Qc"
  ▼ restaurants (pin)
    ▼ 0 (pin)
        id (pin): 1
      ▶ user (pin): {…}
        name (pin): "IdlyExpress"
        address (pin): "1207 East 8th Street"
        description (pin): "Smooth, slow-steeped coffee served over ice for a bold and rich fl
          avor without the bitterness."
        cuisine_type (pin): "Indian"
```

Dispatched the SET_MENU action to store the menu for a selected restaurant in the Redux state. Fetched the menu from the Redux store to display on the page.

```
dispatch({ type: 'SET_MENU', payload: response.data.menu });
```



```
@@INIT                8:06:24.43
SET_SESSION_TOKEN     +00:10.08
SET_RESTAURANTS       +00:03.27
SET_RESTAURANTS       +00:02.40
SET_MENU              +00:16.75
SET_MENU              +00:03.69
```

```
▼ 0 (pin)
    id (pin): 3
    name (pin): "Chicken Dish"
    description (pin): "Grilled chicken over mixed greens, topped with av
      ocado, black beans, corn, and tortilla strips. Served with a zesty ch
      ipotle ranch dressing."
    price (pin): "10.99"
    ingredients (pin): "Grilled chicken, mixed greens, avocado, black bea
      ns, corn, tortilla strips, chipotle ranch dressing."
    image (pin): ""
    category (pin): "Main Course"
    restaurant (pin): 2
```

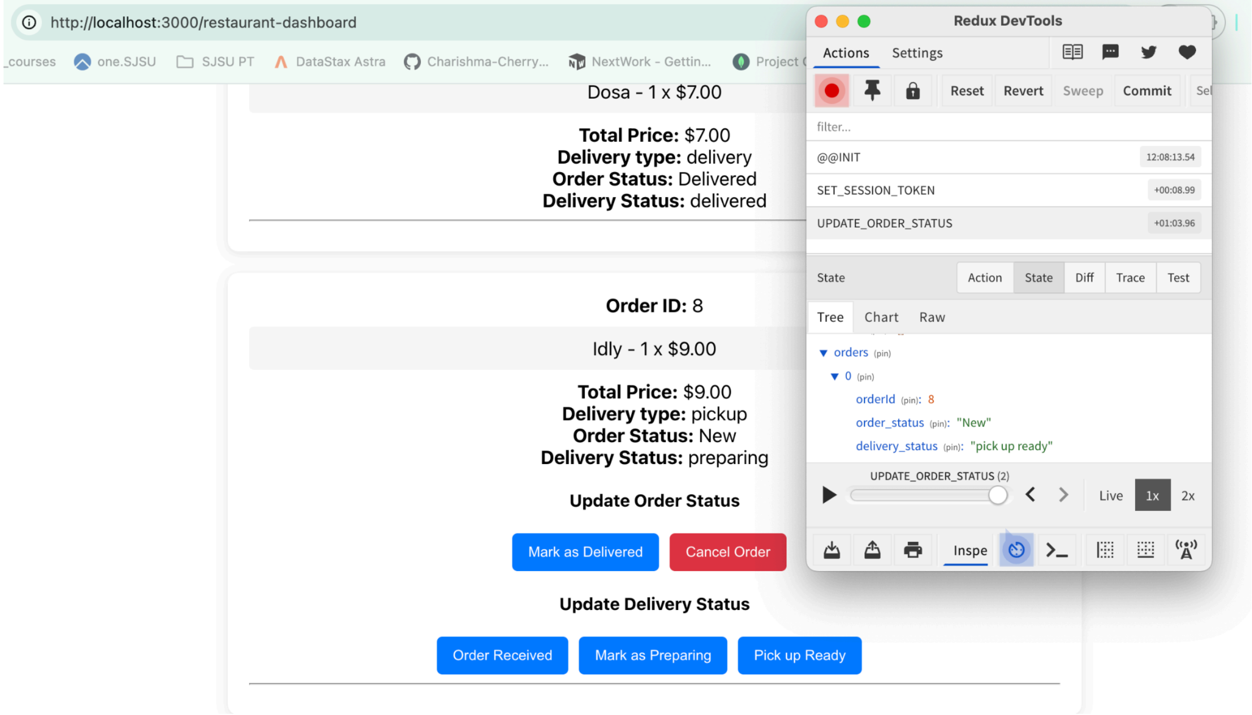
Dispatched the UPDATE_ORDER_STATUS action when the order status was updated, ensuring that the order status in the Redux state matched the server.

```
dispatch({

    type: 'UPDATE_ORDER_STATUS',

    payload: [{ orderId, newStatus, newDeliveryStatus }]

  });
```



Dosa - 1 x $7.00

Total Price: $7.00
Delivery type: delivery
Order Status: Delivered
Delivery Status: delivered

Order ID: 8

Idly - 1 x $9.00

Total Price: $9.00
Delivery type: pickup
Order Status: New
Delivery Status: preparing

Update Order Status

Update Delivery Status

Redux DevTools

```
@@INIT                12:08:13.54
SET_SESSION_TOKEN     +00:08.99
UPDATE_ORDER_STATUS   +01:03.96
```

```
▼ orders (pin)
  ▼ 0 (pin)
      orderId (pin): 8
      order_status (pin): "New"
      delivery_status (pin): "pick up ready"
```

Overview -

Docker was used to containerize both the **frontend (React)** and **backend (Django)** services. Created separate Dockerfile files for the **backend** and **frontend**.

Backend: Installed Python dependencies and ran Django using python manage.py runserver.

Frontend: Installed Node.js dependencies and built the React app for production using npm run build.

Leveraged docker-compose.yml to orchestrate and run both services together in isolated containers.
Benefits:

- Simplified local testing and development.
- Consistent environments across different machines.

Kubernetes Integration

Deployed the **backend** on Kubernetes (using Minikube for local testing):

- Created backend-deployment.yaml and backend-service.yaml.
- Exposed the backend using a **NodePort** service.

The React frontend was not deployed on Kubernetes. It ran locally via npm start and connected to the backend using the port-forwarded address (http://localhost:8000)

Kafka Integration

Kafka was set up locally using a multi-node Kafka and Zookeeper cluster.
Integrated Kafka into the backend:

- Producer: Published events (e.g., order_creation) to Kafka topics.
- Consumer: Processed events from topics such as order_creation and order_status_update.

Example:
Producer: Published order creation data after an order was placed.
Consumer: Monitored the Kafka topics to process order events.

Benefits:
Enabled asynchronous order processing and updates between microservices.

Integration of Redux into the Frontend

Redux was added to manage global state for authentication, restaurant data, and order states.

- Reducers: Centralized logic for updating the state.
- Selectors: Provided easy access to specific slices of state.
- Store: Configured using Redux Toolkit to manage the entire application's state.

Benefits of Redux

- State Centralization: Simplified managing session tokens, restaurant lists, menus, and order tracking.
- Improved Debugging: Leveraged Redux DevTools to track actions and state changes.