

# Jessica Mathias Lab Report 2 - Yelp-Prototype-Redux-MongoDB-Kafka

Youtube Link : <https://youtu.be/x3qwBG6Id2I>

Github Link: <https://github.com/mathiasjess/Yelp-Lab2-Redux-Node-Kafka-MongoDB>

## 1. Introduction

The objective of lab 2 is to build on the yelp prototype application created in lab 1. This lab is designed to use redux to showcase the change of state in every component. MongoDB which is a non-relational DB is used as the database to store customer and restaurant data. Kafka messaging system is used as the middleware to stream messages between the frontend and backend. The application is hosted on AWS EC2. This lab focuses on using pagination, and sorting of results for the current functionalities, adding “all users and following” for a customer and messaging system for a restaurant and customer

## 2. System Design

The system design is described as a combination of assumptions, functionalities, architecture and technology

A detailed description of the application is given as follows: -

### Assumptions

- Every user and restaurant owner has a unique ID and role assigned to them
- Images are stored in the backend and the name of the image is stored in the database
- All API's are validated using passport and JWT
- Bcryptjs is used to encrypt passwords
- Every input field for the customer and restaurant has proper validations
- MongoDB with pooling as a connection is used
- ESLint is used to beautify the code

### Functionalities

#### Restaurant:

- The restaurant owner can register using his restaurant and location
- The owner can update his profile, add/edit dishes and add/edit events
- The owner can view and update all orders to his restaurant
- The owner can click and view on a customer's profile
- The owner can message a customer

#### Customer

- Customer can register using his name
- The customer can update his profile image and all other basic profile details
- The customer can search for events, sort events in order of the date and register for events
- The customer can search for restaurants based on restaurant name, cuisine, type of delivery or location

- The customer can order food from the restaurant
- The customer can write reviews from the restaurant
- The customer can view the list of all yelp users, can search for another user, filter the user based on location and zipcode and follow another user
- The user can reply to a restaurant only if the restaurant messages first, the user can also see a history of messages

## API's

The API's from the lab1 functionalities are not added since they are the same. Only API's for additional functionalities such as messaging and "users tab" are present here

#	API	Method	Usage
1.	/getchats /getrestaurantconversations /getcustomerconversations /getallusers /getotheruserprofile	GET	To get all chats, restaurant specific Chats, customer chats, get all yelp Users and fetch profile details for Other users
2.	/addfollowing	POST	To add the customer to following list

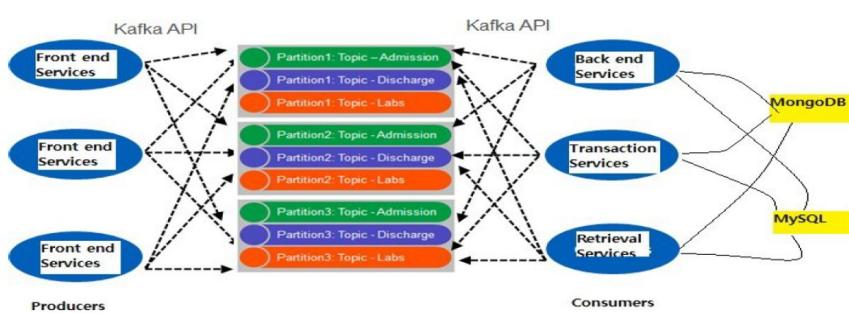
## Architecture

The architecture is split into 3 parts

FrontEnd: The front end is composed of React, Redux, HTML5, Javascript and CSS. Redux is used to store and update the state of all components

Backend: The backend consists of kafka backend and node backend. The request from the API comes to the node server, the node server then directs the request to the kafka server based on the topic. A topic was created for each of the functionality

## Kafka Backend



Kafka which is a distributed streaming system, fetches the data from the producers and adds it to the kafka cluster queue. The kafka consumer receives data, reads/writes to the DB depending on the request and the result is sent back to the node server as a callback

The figure above depicts Kafka's implementation on the backend server. This is used against multiple simultaneous requests to achieve high throughput. Because of replication, it also promotes fault tolerance. The kafka cluster is handled by the zookeeper. The replication factor and partitions are set to one because of the users' limitations.

Database: The cloud version of MongoDB known as MongoDB Atlas is used to store data for the customer and restaurant. Pooling is used. Since MongoDB is a non relational database, retrieval of data is faster than relational databases

Socket.io is used to implement the messaging system between a customer and restaurant. The advantage of socket.io is that it updates the chats in real time without having to refresh the page

## Deployment

The deployment of this application is done using AWS EC2.

EC2 Instance: <http://ec2-52-53-126-104.us-west-1.compute.amazonaws.com:3000>

The screenshot shows the AWS EC2 Instances details page for instance i-0062bcd42429e9526. The instance is running an Amazon Linux AMI (Inferred) and has a public IP of 52.55.126.104. It is associated with a VPC subnet and a VPC ID of vpc-4b4b552c. The instance type is t2.medium. The AWS Compute Optimizer is opt-in, and monitoring is disabled.

Instance ID	Public IPv4 address	Private IPv4 addresses
i-0062bcd42429e9526	52.55.126.104 [open address]	172.31.21.112

Instance state	Public IPv4 DNS	Private IPv4 DNS
Running	ec2-52-53-126-104.us-west-1.compute.amazonaws.com [open address]	ip-172-51-21-112.us-west-1.compute.internal

Instance type	Elastic IP addresses	VPC ID
t2.medium	-	vpc-4b4b552c

IAM Role	Subnet ID
-	subnet-caed67ac

Platform	AMI ID	Monitoring
Amazon Linux (Inferred)	ami-0e4035ae3f70c400f	disabled

### 3. Results

#### Customer Role:

##### 1. Registration

The customer registers to the website using his first name, last name, email ID and password. Once registered, he/she is rerouted to the login page. The password is authenticated using bcrypt.

The screenshot shows a web browser window with the URL `http://localhost:3000/register/customerregister`. A modal dialog box displays the message "User registration successful". Below the modal, there is a placeholder for a profile picture with a red ribbon banner. The browser's address bar shows the same registration URL. To the right of the browser, the Visual Studio Code editor is open, displaying the `customerRegistration.js` file. The code handles a POST request for customer registration, generating a salt and hash for the password, and saving the user to the database. It also logs the registration details and returns a success response.

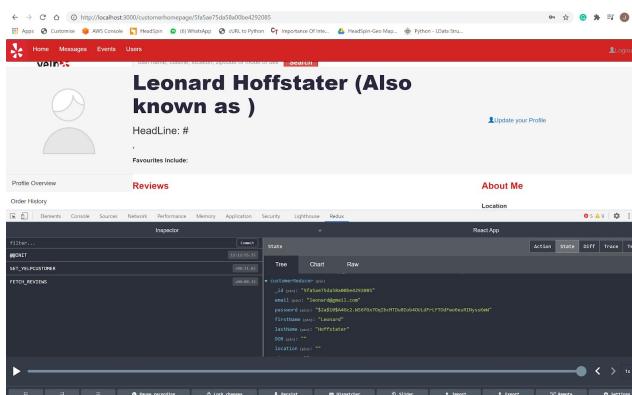
##### 2. Login

The user logs in using his email ID and password. On successful authentication, the passport authorises the user, a JWT token is created and sent to the front end. This JWT token is used to authenticate future requests from the user.

The screenshot shows a web browser window with the URL `http://localhost:3000/login/customerlogin`. A modal dialog box displays the message "User login successful". Below the modal, there is a placeholder for a profile picture with a red ribbon banner. The browser's address bar shows the login URL. To the right of the browser, the Visual Studio Code editor is open, displaying the `customerLogin.js` file. The code handles a POST request for user login, checks the credentials against the database, generates a JWT token, and returns it to the front end. The code also includes middleware for handling errors and logging.

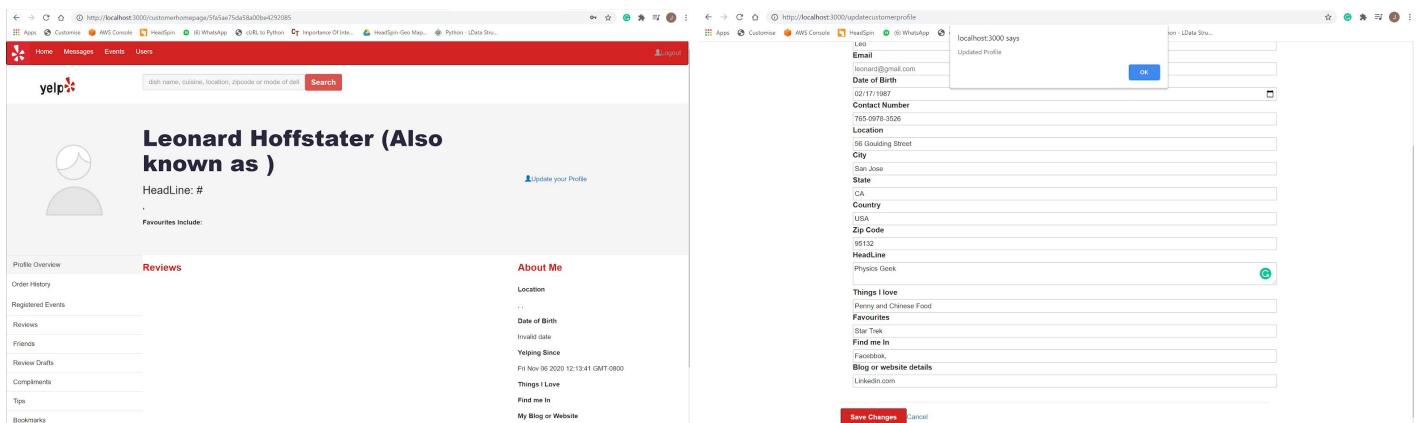
The screenshot displays a developer's workspace. On the left, a terminal window shows Node.js code for a Kafka consumer, specifically handling customer registration requests. On the right, a browser window shows a user profile for 'Leonard Hoffstater'. The profile includes sections for 'Profile Overview', 'Reviews', and 'About Me'. The 'About Me' section contains fields like 'Headline', 'Location', and 'Bio'. The browser's developer tools are open, showing the network tab with several requests and responses, including one for 'customerProfile'.

Redux is used to store the state of the user once they have logged in

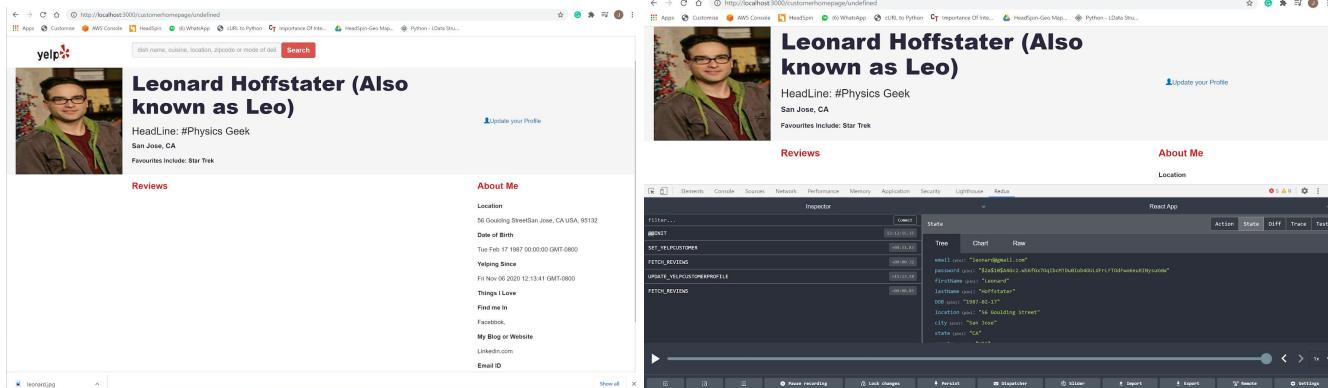


### 3. Profile Page

Once the user is logged in, he will be able to see his profile page. The user can then update his profile picture, basic details, his favourites, contact information and other profile details

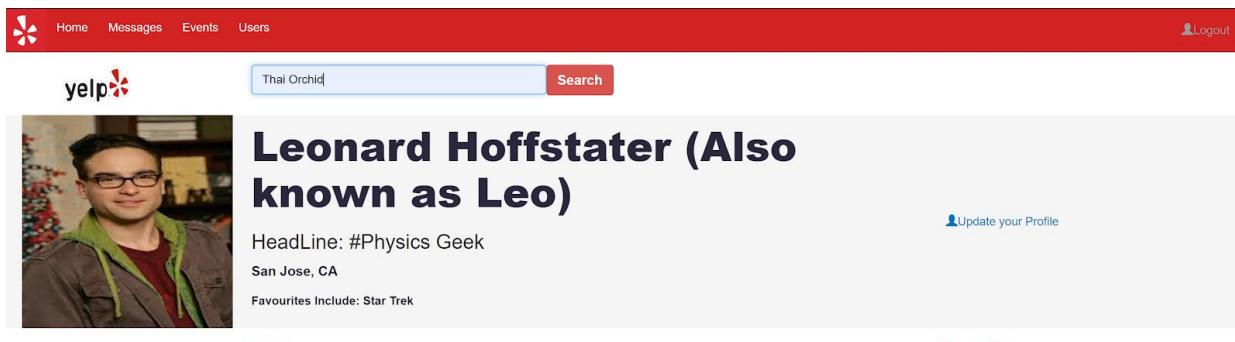


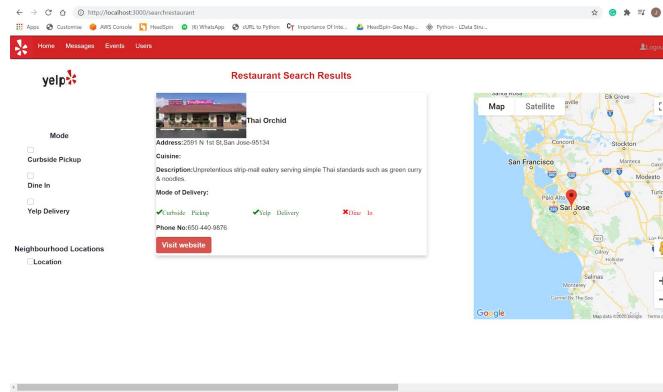
The reducer `UPDATE_YELPCUSTOMERPROFILE` is used to update the redux store with the user's updated profile information



## 4. Restaurant Search

The user should be able to search for a restaurant using restaurant Name, dish Name, location, type of delivery and cuisine





The user is further able to filter his search results based on type of delivery and neighbourhood locations. The search feature also used google maps to show the locations of the nearby restaurants

On clicking the restaurant, the user should be redirected to the restaurant's site  
The redux store is updated with the restaurant's information

## 5. Orders Tab

User should be able to see the order history along with the date and order details. Pagination is used to show 3 orders per page

The user should be able to filter the orders based on delivery type, and delivery status(order received and order preparing)

The screenshot shows a developer's workspace. On the left, a terminal window displays Node.js code for handling Kafka messages related to customer order history. On the right, a browser window shows a list of orders from a restaurant named "Cocina Del Charro". The orders are sorted by date, with the most recent at the top. Each order card includes the order ID, date, total price, delivery status, and type.

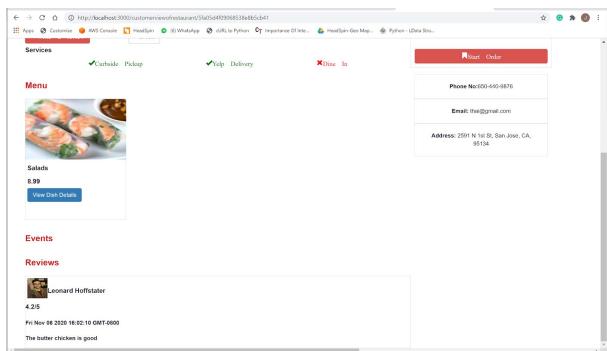
The orders can be sorted in ascending and descending order based on date

The redux store is updated to show the customer order history

This screenshot illustrates the state of the application's redux store. It shows two separate instances of the customer order history page. The first instance displays orders for "Thai Orchid" and the second for "Cocina Del Charro". Both pages show the same data as the main screenshot, with orders sorted by date. To the right, the developer tools Network tab is open, showing the raw JSON data for one of the orders, which includes the order ID, date, total price, delivery status, and type.

## 6. Restaurant Page

Once the user views the restaurant page, he can add reviews



The user can choose the dishes and add to cart. Take out options can be selected as pickup or delivery. The cart handling is done by redux. Pagination is implemented.

The order is successfully placed and shows up in the customer's order history

## 7. Events

The user can see events from all restaurants on a single page. The user can also sort the order of events in ascending and descending order of the date

The redux store is populated with all the events  
Click and view on details

The screenshot displays a web application interface with multiple windows open. On the left, a browser window shows the main events page with a list of events: "Taste of the Mediterranean" (Host: Cocina Del Chavo, Date: Wed Oct 21 2020 17:00:00 GMT-0700), "Karaoke Nights" (Host: Cocina Del Chavo), and "Event2" (Host: Dinh Da Dash, Date: Wed Nov 04 2020 16:00:00 GMT-0800). Below this is a DevTools screenshot showing the React component tree for the event list. On the right, another browser window shows the event details for "Taste of the Mediterranean". A third browser window on the far right shows a success message: "Registered successfully for event". At the bottom, a code editor window in VS Code shows the source code for the "registerForEvent" function in the "EventDetails.js" file, which handles button clicks to register for specific events.

## 8. Users Tab

The user should be able to see all the yelp users. Pagination is implemented

Redux stores the state of all users

The user should be able to search for another user using his first name or nickname. (Here we are searching for Chandler Bing using Chan)

The results can be filtered using location  
On clicking the user's name, the customer profile page should render

The user can follow another user and can filter the user results based on following

```
const kafka = require('kafka-node');
const http = require('http');
const express = require('express');
const bodyParser = require('body-parser');

const config = {
    brokers: ['localhost:9092'],
    logLevel: 'info'
};

const client = new kafka.KafkaClient(config);
const consumer = client.createConsumer({
    topics: ['getfollower'],
    autoCommit: true,
    autoOffsetReset: 'earliest'
}, { 'callback': handleRequest });
consumer.on('error', console.error);

function handleRequest(err, req, res) {
    if (err) {
        console.error(`Error: ${err}`);
        res.json({ error: 'Internal Server Error' });
        return;
    }
    const { followerId } = req.query;
    const followerDetails = {
        followerId,
        followerName: 'Leonard Hoffstaetter',
        followerEmail: 'l.hoffstaetter@kafka.com',
        followerPhone: '+43 664 75508000123456789'
    };
    res.json(followerDetails);
}

const app = express();
app.use(bodyParser.json());
app.get('/followers/:followerId', handleRequest);
app.listen(3001, () => {
    console.log('Kafka follower API listening on port 3001');
});
```

Home Messages Events Users

Logout



# Michael Scott (Also known as )

HeadLine: #  
San Jose,  
Favourites include: Travelling

## Reviews

Ratings: 4.25

 Thai Orchid

Date: Fri Nov 06 2020 16:02:10 GMT-0700

Comments: The butter chicken is good

## About Me

Location  
San Jose, USA

Date of Birth  
Invalid date

Yelping Since  
Mon Oct 19 2020 10:13:34 GMT-0700

Things I Love  
Find me in  
LinkedIn.com

The screenshot shows a web application interface. At the top, there is a red header bar with the text "Home", "Messages", "Events", and "Users". Below this, a white search interface has a red border. It features a "User Name or Nick Name" input field and a red "search" button. Above the search bar, there is a section titled "Filters" with three options: "All Users", "Location", and "Following", where "Following" is highlighted with a red box. The main content area has a red border and contains the heading "List of Yelp Users" in red text. Below it, the name "Michael Scott" is listed. At the bottom of the content area, there is a navigation bar with two arrows and a central page number "1".

## **Restaurant Role:**

## **1. Restaurant Login, Profile and Update Profile Details:**

The restaurant owner registers using the restaurant name, email, password and location



The screenshot shows the Yelp login interface. At the top, there's a red navigation bar with links for Home, Messages, Menu, Events, Orders, and Reviews. Below it, a large circular placeholder image of a restaurant building is centered. To the left of the placeholder, there are two buttons: 'Existing User?' and 'Restaurant Owner?'. On the right side of the placeholder, there's a 'Sign in to Yelp' button. Below the placeholder, there are input fields for 'Email' (containing 'cooking@gmail.com') and 'Password', followed by a 'User Log In' button.

Sign in to Yelp

New to Yelp? Sign Up

Existing User? Restaurant Owner?

cooking@gmail.com

password

User Log In

Code editor showing routes and logic for handling restaurant profile updates:

```

    // routes
    router.post('/updateRestaurantProfile', (req, res) => {
      // ... logic to handle post request ...
      res.json({status: 'Success', message: 'Restaurant Profile updated successfully.'})
    })
  }
}

// to handle post request call to update basic Restaurant Reference
router.post('/updateBasicRestaurantProfile', (req, res) => {
  // ... logic to handle post request ...
  res.json({status: 'Success', message: 'Basic Restaurant Profile updated successfully.'})
})

```

## 2. Update Restaurant Profile

The restaurant owner can update profile details such as basic details, restaurant pics, contact information etc

Form fields for updating restaurant profile:

- Address: 2301 N 1st St, San Jose
- Phone: 95134
- Email: Update Restaurant Profile
- Password: (hidden)
- Amenities and more:
  - Curbside Pickup
  - Dine In
  - Delivery

Code editor showing logic for updating restaurant user profile:

```

    // routes
    router.put('/updateRestaurantUserProfile', (req, res) => {
      const {username, email, password} = req.body
      const restaurantOwner = await RestaurantOwner.findById(req.params.id)
      if (!restaurantOwner) return res.status(404).json({message: 'Restaurant Owner not found'})
      if (password) {
        const salt = await bcrypt.genSalt()
        const hashedPassword = await bcrypt.hash(password, salt)
        restaurantOwner.password = hashedPassword
      }
      restaurantOwner.username = username
      restaurantOwner.email = email
      restaurantOwner.save()
      res.json({status: 'Success', message: 'Restaurant User Profile updated successfully.'})
    })
  }
}

```

## 3. View Reviews

The restaurant owner should be able to view reviews given by all the users.

Pagination is implemented.

The reviews are fetched from the redux store

Review details:

- Rating: 4.2/5
- User: Leonard Hoffstatter
- Date: Fri Nov 06 2020 16:02:10 GMT-0800
- Comments: The butter chicken is good

Code editor showing logic for handling reviews:

```

    // routes
    router.get('/getReviews', (req, res) => {
      const {id} = req.params
      const reviews = await Review.find({restaurantId: id})
      res.json({reviews})
    })
  }
}

// to handle post request to update basic Restaurant Reference
router.post('/updateBasicRestaurantProfile', (req, res) => {
  const {id} = req.params
  const {name, address, phone, email, password} = req.body
  const restaurant = await Restaurant.findById(id)
  if (!restaurant) return res.status(404).json({message: 'Restaurant not found'})
  if (password) {
    const salt = await bcrypt.genSalt()
    const hashedPassword = await bcrypt.hash(password, salt)
    restaurant.password = hashedPassword
  }
  restaurant.name = name
  restaurant.address = address
  restaurant.phone = phone
  restaurant.email = email
  restaurant.save()
  res.json({status: 'Success', message: 'Basic Restaurant Profile updated successfully.'})
})

```

## 4. Add Dishes to the Menu

The owner can add dishes to the Menu, the added dishes can then be seen in the menu tab.

## Pagination is implemented

The screenshot shows a web application interface for managing a restaurant's menu. On the left, a modal window titled "Description of the Dish" is open, prompting the user to enter a dish description ("Chicken caesar salad") and a price ("10.99"). On the right, the application's codebase is visible in a code editor, specifically the file `restaurantMenus.js`, which contains logic for handling dish additions to the menu.

`restaurantMenus.js` code snippet:

```

    router.post('/addDish', function (req, res) {
      var addDishObject = {
        restaurantId: req.body.restaurantId,
        dishIngredients: req.body.dishIngredients,
        dishDescription: req.body.dishDescription,
        dishPrice: req.body.dishPrice,
        dishCategory: req.body.dishCategory
      }
      kafka.make_request('updateMenu', addDishObject, function (err, results) {
        if (err) {
          console.log("Inside err");
          res.json({
            status: "error",
            msg: "System Error, try Again."
          })
        } else {
          console.log("Inside response");
          res.json({
            status: "success",
            msg: "Dish added successfully"
          })
        }
      })
    })
  }
}

```

Below the main application interface, there are two browser tabs showing the "Menu Page". The left tab shows the initial state with one dish listed. The right tab shows the state after the dish has been added, with the new dish now appearing in the list.

## 5. Orders Tab

The restaurant owner is able to view all orders by customers. The order details are stored in redux. Pagination is implemented

The screenshot shows the "Orders" tab of the application. It lists three customer orders with their details: Michael Scott (Cancelled Order), Leonard Hofstadter (Delivered), and Howard Wolowitz (New Order). The application uses Redux for state management, and the right-hand panel shows the React DevTools Inspector, which tracks the state transitions for these orders.

React DevTools Inspector state logs:

```

    ADD_ORDER
    SET_RESTAURANTOWNERPROFILE
    SET_RESTAURANTOWNER
    SET_RESTAURANTOWNERPROFILE
    ADD_ORDER
    SET_RESTAURANTOWNER
    SET_RESTAURANTOWNERPROFILE
    SET_VELPUSOWNER
    SET_VELPUSOWNER
    FETCH_REVIEWS
    SET_RESTAURANT
  
```

The orders can be filtered based on New order, delivered order or cancelled order.

Order status can be updated by the restaurant owner

## 6. Events Tab

The restaurant owner should be able to post events. All the events posted by the owner can be viewed in the restaurant tab. The owner can view the registered list for each event and only clicking on the registered user, it should lead him to the customer page

The left window shows a table titled "List of Users Registered for Event" with two rows:

Sl No.	Customer Name
1	Mike Scott
2	Leonard Hoffstatter

The right window shows a detailed user profile for "Leonard Hoffstatter (Also known as Leo)". Key details include:

- Headline:** #Physics Geek
- Location:** San Jose, CA
- Favourites include:** Star Trek
- Reviews:** Ratings: 4.25, Thai Orchid
- Date of Birth:** Mon Feb 16 1987 16:00:00 GMT-0800
- Comments:** The butter chicken is good

## 7. Messages Tab

The left window shows a user profile for "Leonard Hoffstatter (Also known as Leo)" with a picture, headline, location, reviews, and an "About Me" section.

The right window shows a "Real Time Chat" interface with a message from "Cocina Del Charro" at 2020-11-07 20:21:40: "Message H, your order is being prepared".

The top window shows a "Chat History" section with a message from "Cocina Del Charro": "Message: Hi, your order is being prepared" at 2020-11-07 20:21:40.

The bottom window shows a "Real Time Chat" interface with messages from "Cocina Del Charro" and "Leonard Hoffstatter".

The top part of the window shows the browser's Network tab with several requests listed, such as "GET /api/messages" and "POST /api/messages".

The bottom part shows a code editor with Node.js code for handling messages, including routes like "/api/messages" and "/api/messages/:id", and functions for creating, updating, and deleting messages.

## 4. Testing

## **1. Enzyme testing:**

Enzyme testing was done for 3 views namely customer registration, restaurant registrations and reviews from customers

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure with files like `CustomerReviews.js`, `CustomerReviews.test.js`, `RestaurantRegister.js`, `RestaurantRegister.test.js`, `RestaurantHomePage.js`, `RestaurantProfile.js`, `RestaurantRegister.test.js`, and `App.js`.
- Editor:** The main editor pane displays `CustomerReviews.test.js` and `RestaurantRegister.test.js`. Both files contain Jest test cases for their respective components.
- Terminal:** The terminal tab shows the command `1: node` and the output of running tests:

```
Test Suites: 4 passed, 4 total
Tests:
  5 passed, 5 total
Time: 10.394s
Ran all test suites related to changed files.
```
- Status Bar:** Shows "File 18, Col 3" and "Javascript (Babel)".
- Taskbar:** Shows various pinned icons including GitHub, LinkedIn, and a file icon.

## 2. Mocha Chai Test

Mocha chai testing was implemented for 5 random API's

1. Restaurant Login
  2. Fetch restaurant profile for a particular customer
  3. Fetch all restaurant events
  4. Fetch customer profile
  5. Fetch order summary for a particular restaurant

```

    // This API is tested without the checkAuth
    describe("GET Customer Profile", () -> {
        let id = "5e76f527403d04642d";
        it("It should get customer profile details", (done) -> {
            chai.request('http://localhost:3001')
                .get(`customerprofile/${id}`)
                .end((err, res) => {
                    expect(res).to.be.ok;
                    expect(res.body.message).to.be.equal("Success");
                    expect(res.body.data).to.be.an("object");
                    expect(res.body.data.customerName).to.be.equal("John Doe");
                    expect(res.body.data.restaurantName).to.be.equal("Doe's Deli");
                    expect(res.body.data.orderCount).to.be.equal(45);
                    done();
                });
        });
    });

    // This API is tested without the checkAuth
    describe("GET All Restaurants", () -> {
        it("It should get all restaurants", (done) -> {
            chai.request('http://localhost:3001')
                .get(`customerprofile/getallrestaurants`)
                .end((err, res) => {
                    expect(res).to.be.ok;
                    expect(res.body.message).to.be.equal("Success");
                    expect(res.body.data).to.be.an("array");
                    expect(res.body.data.length).to.be.equal(1);
                    done();
                });
        });
    });

```

```

    // This API is tested without the checkAuth
    describe("GET Customer Order Summary", () -> {
        it("It should get customer order summary", (done) -> {
            chai.request('http://localhost:3001')
                .get(`customerprofile/getcustomerordersummary`)
                .end((err, res) => {
                    expect(res).to.be.ok;
                    expect(res.body.message).to.be.equal("Success");
                    expect(res.body.data).to.be.an("array");
                    expect(res.body.data.length).to.be.equal(1);
                    done();
                });
        });
    });

```

### 3. Performance testing using JMeter

Jmeter is used to test the restaurant login route

1. MongoDB JMeter testing was done using connection pooling and without connection pooling. For connection pooling the pool size was set to 100.
2. From the below results, the response time increases proportionately when the number of concurrent users increases.
3. When the pool size is equal to or lower than the concurrent users, the average response time is much faster. As the number of users increases about the pooling size, the response time increases exponentially
4. Time taken for requests to be served on a non pooled connection is more since it can serve only one server request at a time, this causes a performance bottleneck as the response time increases

API	Users	With connection Pooling(ms)	Without Connection Pooling(ms)
http://localhost:3001/restaurantloginroute/restaurantlogin	100	795	1835
	200	1986	2987

	300	3996	4864
	400	4844	5949
	500	6241	8009

### With Connection Pooling

100 users

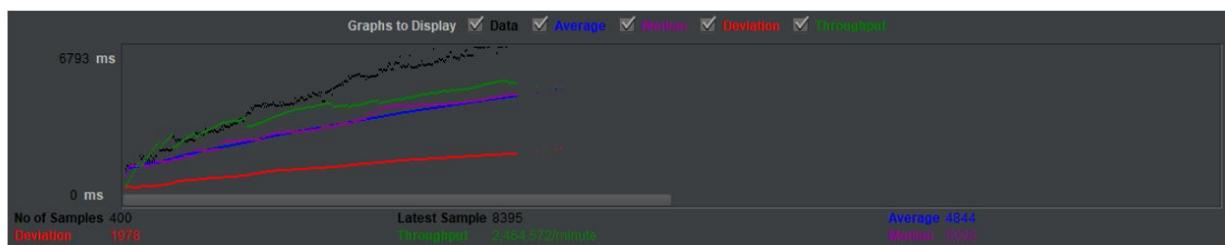
200 users



300 users



400 users



500 users



### Without using Connection Pooling

100 users



## 200 users



## 300 users



## 400 users



## 500 users



## 5) Git Commit History

The image displays three separate screenshots of GitHub commit histories, each showing a list of commits for a specific branch. The first screenshot shows the master branch commit history from November 8, 2020, to November 3, 2020. The second screenshot shows the master branch commit history from October 1, 2020, to October 21, 2020. The third screenshot shows the master branch commit history from October 21, 2020, to October 14, 2020. Each screenshot lists commits with author, message, date, and a link to the commit details.

**Screenshot 1 (Top):** Commit history for the master branch from Nov 8, 2020, to Nov 3, 2020.

- Commits on Nov 8, 2020:
  - kafka testing on cloud
  - Delete Kafka settings
  - kafka settings
- Commits on Nov 7, 2020:
  - Modified package.json to deploy to cloud
  - optimizing functionalities
- Commits on Nov 4, 2020:
  - Implemented Kafka for customer
- Commits on Nov 3, 2020:
  - Implemented JWT passport for restaurant APIs

**Screenshot 2 (Middle):** Commit history for the master branch from Oct 1, 2020, to Oct 21, 2020.

- Commits on Nov 1, 2020:
  - Added Kafka middleware
- Commits on Oct 31, 2020:
  - Implemented messaging system for customer
  - Implemented Chat system for restaurant
- Commits on Oct 28, 2020:
  - Implemented users tab with following
- Commits on Oct 27, 2020:
  - Implemented paging for customer order history and restaurant orders
- Commits on Oct 24, 2020:
  - Implemented restaurant order history
- Commits on Oct 21, 2020:
  - Front end for customer login, profile details and update profile with...
  - Implemented Pagination for restaurant pages

**Screenshot 3 (Bottom):** Commit history for the master branch from Oct 21, 2020, to Oct 14, 2020.

- Commits on Oct 21, 2020:
  - Front end for customer login, profile details and update profile with...
  - Implemented Pagination for restaurant pages
- Commits on Oct 20, 2020:
  - Implemented redux for restaurant profile, menu, reviews and events
  - APIs for orders, events and reviews
- Commits on Oct 18, 2020:
  - Created APIs for restaurant registration, login and updating profile
- Commits on Oct 14, 2020:
  - Initial commit

## 6. Answer to the questions

### 1. Comparison of passport authentication with Lab1 authentication process.

In the first lab for the authentication process, bcrypt was used to encrypt the password and the token was stored in the localStorage. This method did not provide security to the API calls. With JWT, a token is

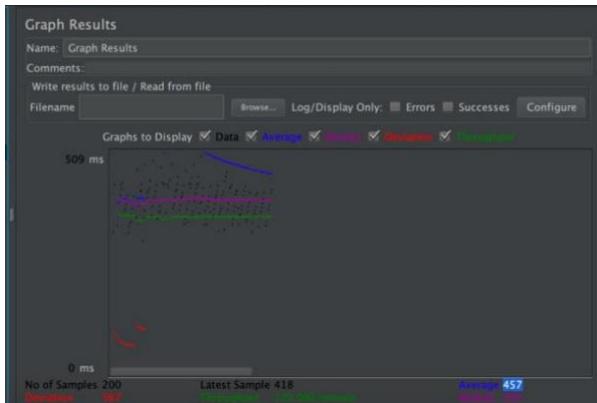
created during the login for a restaurant or customer. This token can then be used to authenticate the roots and make any requests. The middleware created using passport can be reused by multiple API's which helps to increase code reusability and decreases code complexity

## 2. Performance comparison of with and without Kafka.

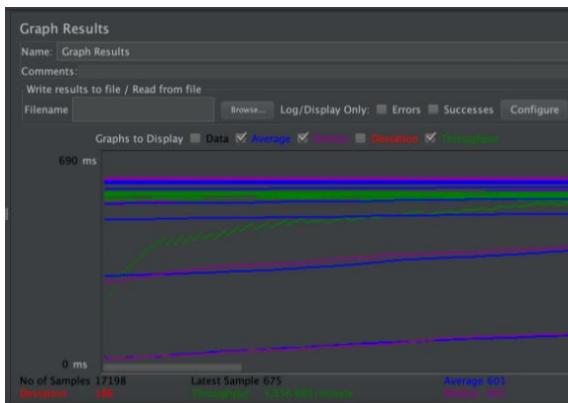
One API was used in order to compare the performance of response time with and without Kafka messaging system. The application was load tested with 200 users for 10 minutes. It was found that with kafka the average response time was < 0.4 sec whereas without Kafka the average response time was greater at >0.6 sec

The Kafka queues were able to deliver the backend messages, as shown, and also acted as a store buffer to escape bottlenecks from the database. Not only are responses enhanced, but Kafka will help the application scale in a distributed manner. As the available queue will pick up the message to prevent bottlenecks from being processed.

### With Kafka



### Without Kafka



## 3. If given an option to implement MySQL and MongoDB both in your application, specify which data of the applications will you store in MongoDB and MySQL respectively

MySQL which is a relational database is highly reliable when it comes to maintaining consistency in data. MongoDB on the other hand gradually becomes consistent. With this as consideration, critical data which is required for almost all API requests and needs consistency should be stored in the MySQL database.

In the yelp application, in the case of a customer, we need the custom ID, email and password across APIs to authenticate and fetch consequent requests. In the case of a restaurant we need the restaurant ID, name, email, password and location to make subsequent API requests. Hence these data fields with high dependency on them should be stored in the MySQL so that retrieval is quick and consistent

MongoDB works best when there is a need to store non structured data. Since the data is non relational, it fetches data faster. For example in the customer role, customer details can be stored in the mongoDB, orders, registered events, and messages can be stored in the MongoDB

In this application, MongoDB played a crucial role for me in creating the restaurant schema. WIth MongoDB it was easier to structure the events, menu items, reviews and orders for a particular restaurant under one schema. This made it easier to pull the data from MongoDB once and store it in the redux state. With one call, all the details for the restaurant can be fetched

In case of messaging system, it is very useful to store the chats in the MongoDB since the data can be non structured ie, array of text, images and videos can be stored in MongoDB

Another instance is that auto sharding will benefit from the inclusion of items in MongoDB for Items set. This will also suggest that sets of objects are essentially compatible (which could be a tradeoff considering this as non-critical data from the rest of the request)

### Note:

Added TA as contributor

