Build a simulation of the current OMS Model

**Primary Goal:**

The purpose of this use case is to look at the process and flow of login verification in the current OMS Model and allow specific Order Management Home Pages depending on the User's admin status. If a user is an admin, the user can Create, Update, and Search Orders and agents can only Search Orders.

The Order Management Homepage gives a new layer of visibility so that Users can see what actions they can perform based on their status as an admin.

**Stakeholder:**

Admins seeking to place and update orders as well as agents looking to check placed orders.

**Scope:**

To securely log in to an application and access a page with specific actions.

**Assumptions:**

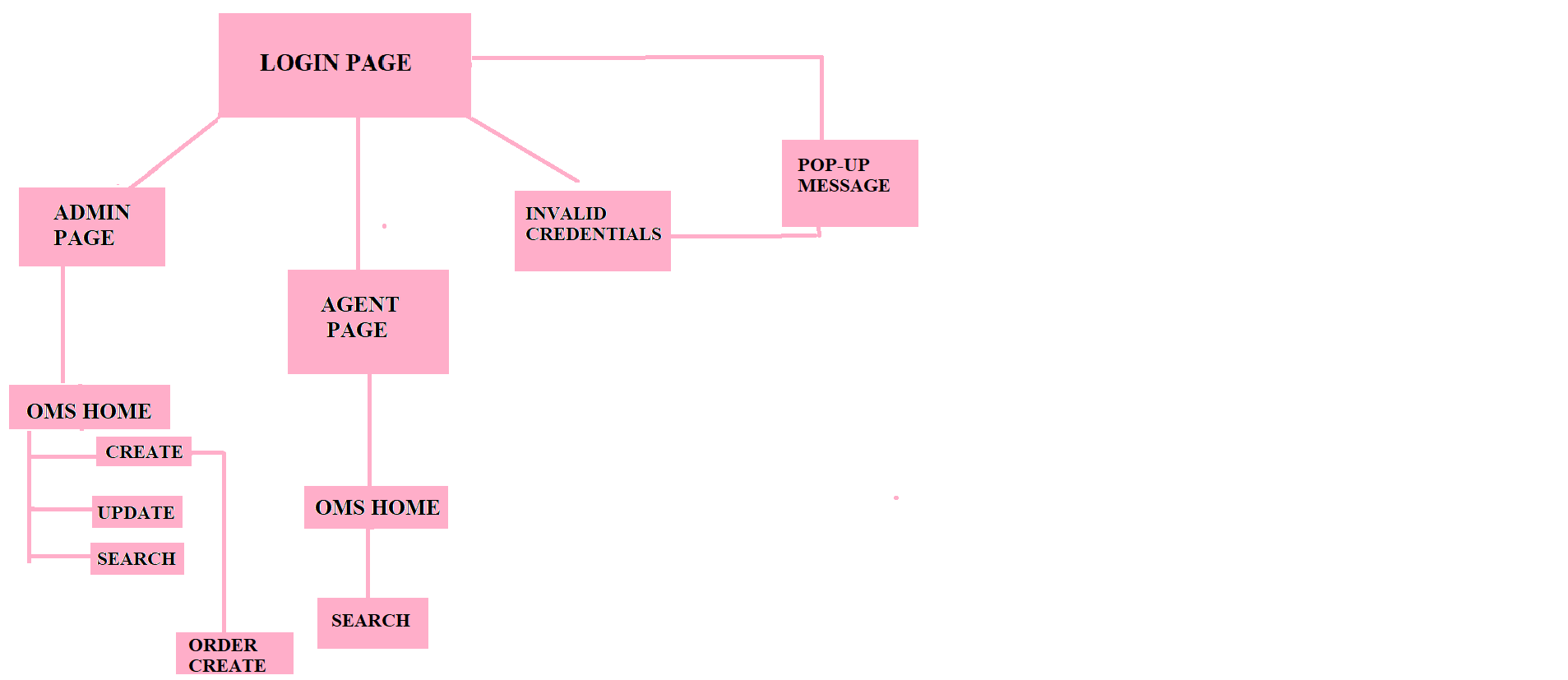
-> There are only admins and agents.

-> Admins can Create, Update, and Search.

-> Agents can Search placed Orders.

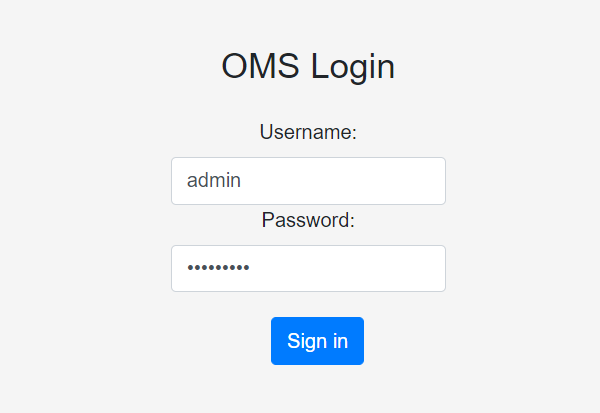
**Workflow:**

*General Workflow Diagram*

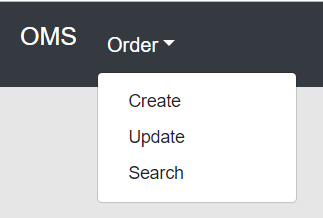


***Admin Flow:***

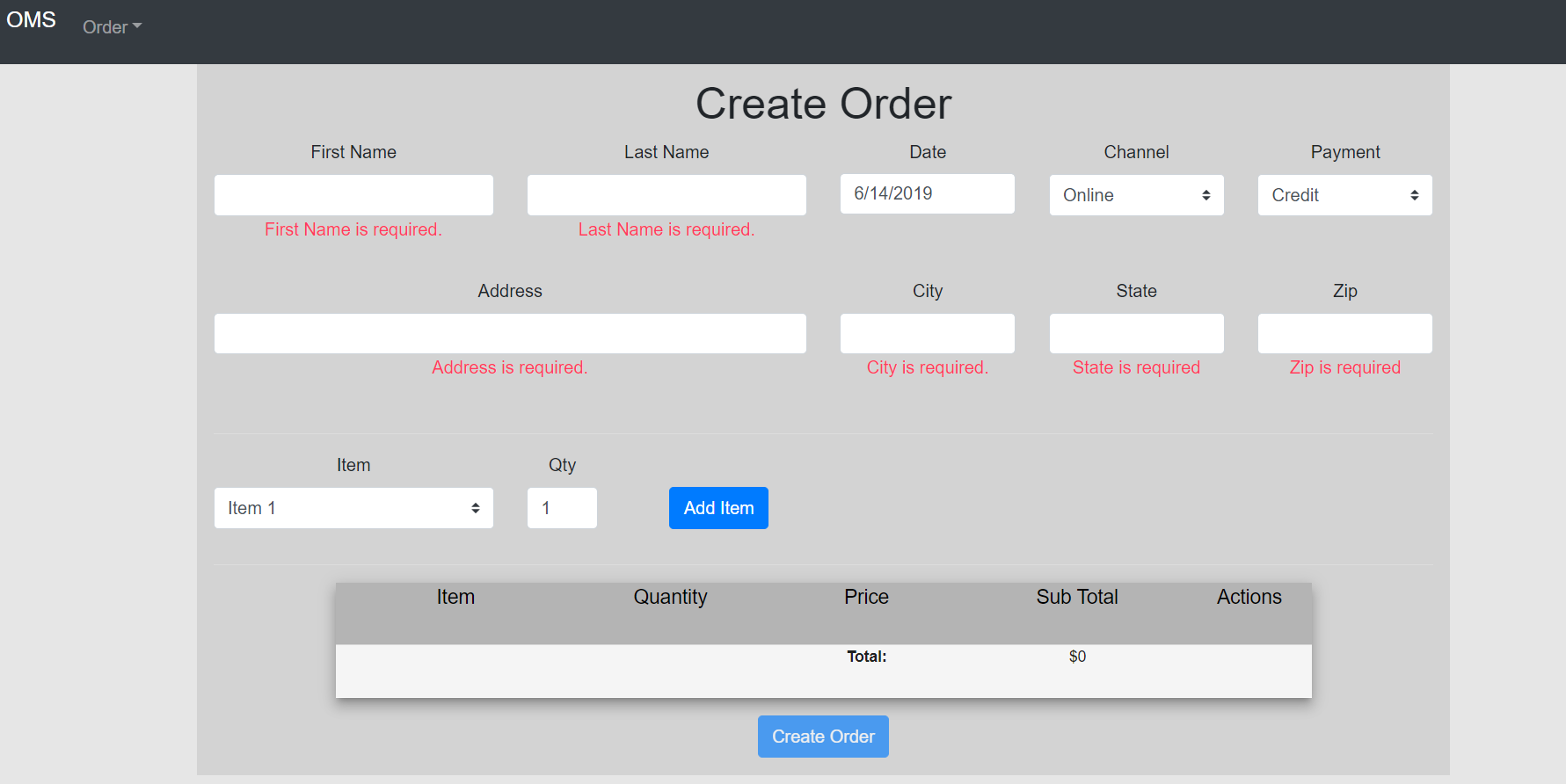
The first step for an Admin attempting to log onto OMS is to go through the login page. For our demo purposes, admins have the username admin and password Admin!123.



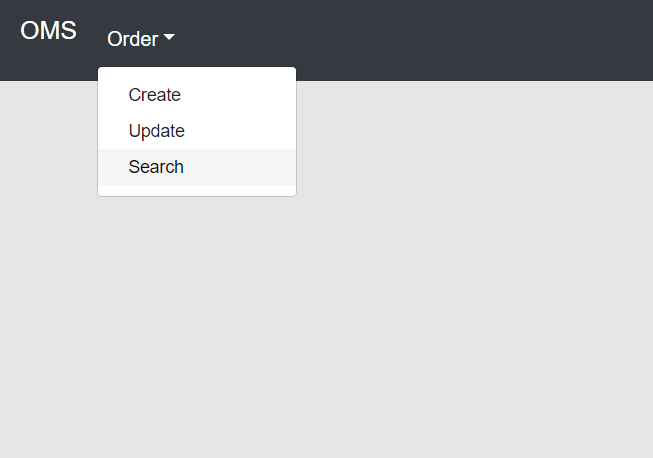
After successfully logging on, Admins are shown the options to Create, Update, and Search Orders.

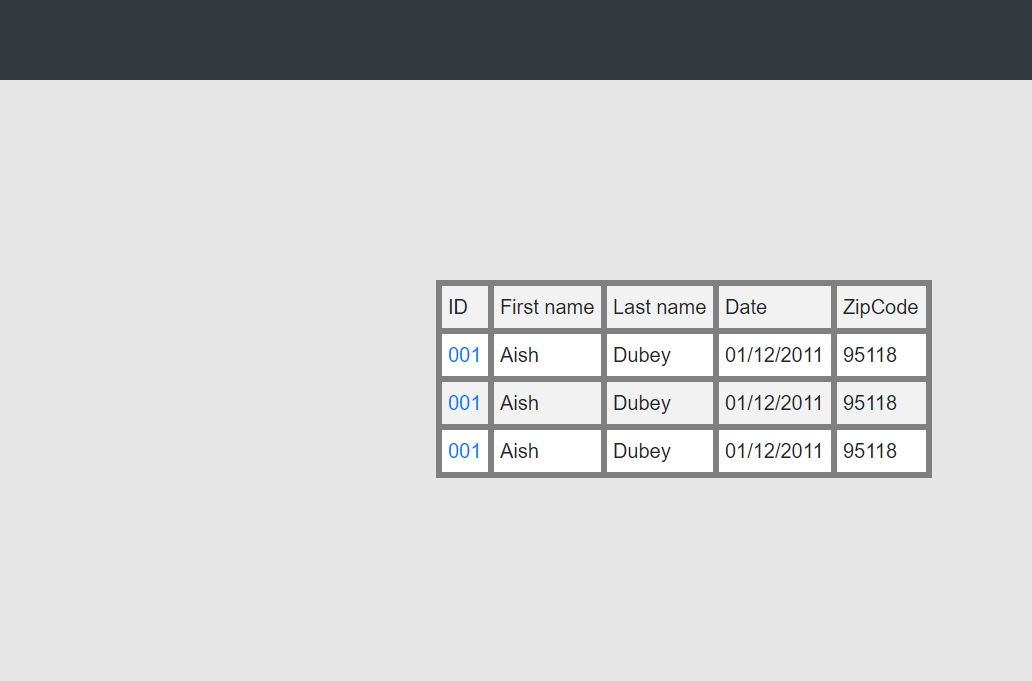


Clicking on Create allows Admins to create orders.



By clicking on Search the admin can search the orders that have been placed.

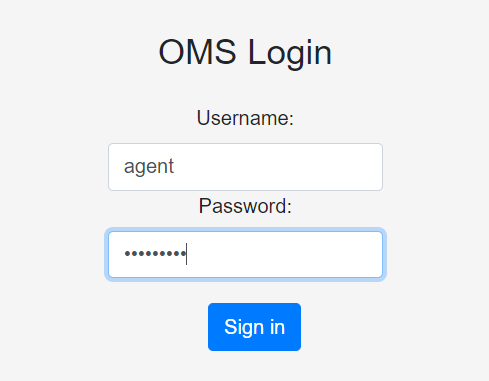
****

****

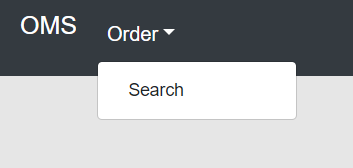
The search page will display the data in the form of a table where each ID will be linked to the view page giving the admin the entire summary of the order.

*Agent Flow:*

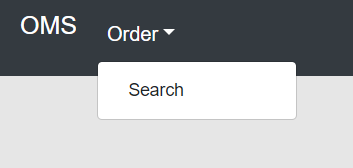
The first step for an Agent attempting to log onto OMS is to go through the login page. For our demo purposes, agents have the username agent and password Agent!123.



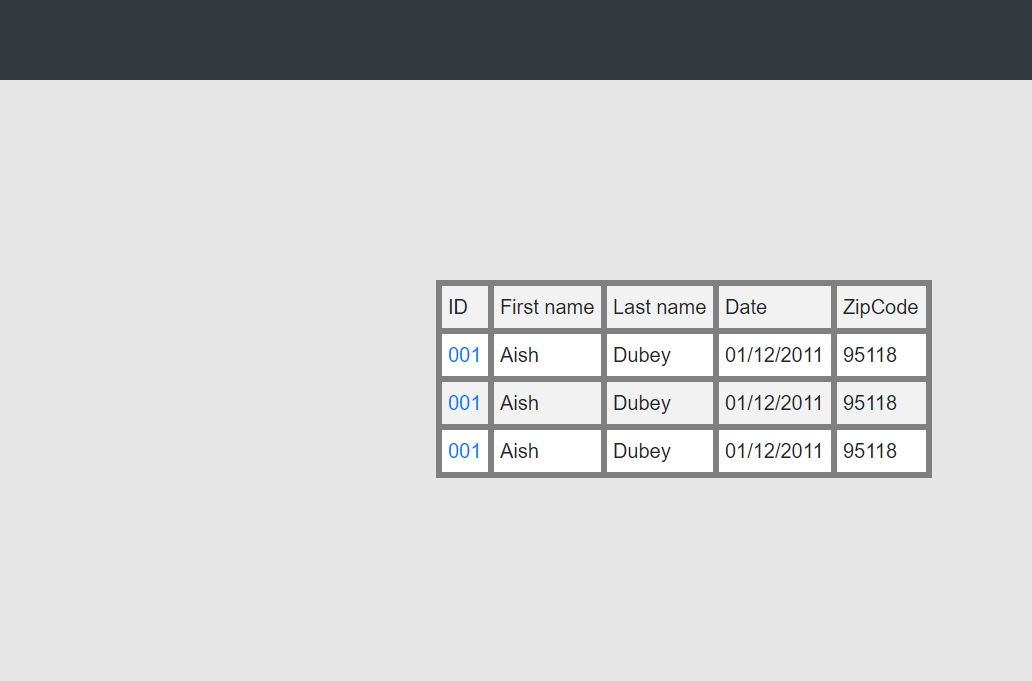
After successfully logging on, agents are shown the option to Search Orders.



By clicking on Search the agents can search the orders that have been placed.

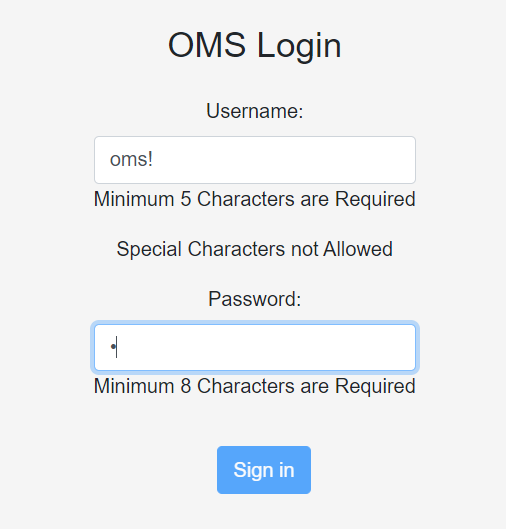


They can view the orders where the ID is the link where agents can view the summary report of the orders.

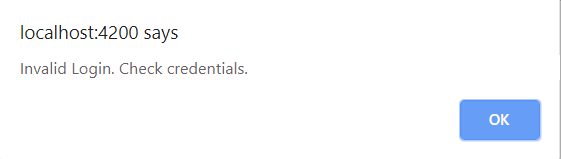
****

*Invalid Flow:*

There is a front-end login verification system that prevents invalid logins early. Usernames are alphanumeric and must have 5 characters at least. Passwords can contain special characters and have to have at least 8 characters. Both the username and password fields are case-sensitive. The "Sign in" button is disabled until a properly formatted username and password is given.



If an invalid login is given, a pop up stating "Invalid Login. Check credentials." is shown. Clicking on the "OK" button allows the user to attempt to log on again.



**Conclusion**

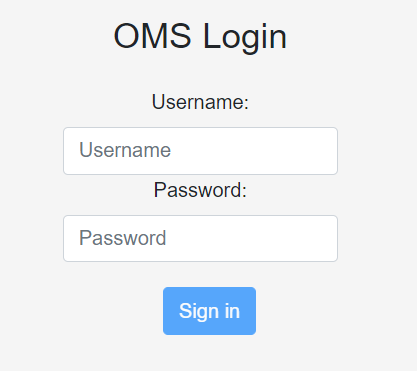
In conclusion, this current version allows for users to log in as admins or agents depending on the login username and password and shows different options for both types of viewers. Admins do not have restrictions and can Create Orders as well as Update Orders while Agents can only View and Search Orders.

Invalid logins are prevented early on the front end as improperly formatted login combinations cannot sign in. Logins that are properly formatted but are incorrect have a pop up show up.

Technical Design

**Services and Components:**

**Login:**

****

File Structure:

omsui/oms/src/app/login:

├── `login.component.html` - HTML code to build the component and handle front-end verification

├── `login.component.ts` - LoginComponent class to handle input data and connect to VerifyLoginService

├── `login.component.css` - CSS code to handle the styling for the login page

├── `login.component.spec.ts` - File to handle the component

**Login.component.ts class structure:**

**Variables:**

User: keeps track of name

Yes: keeps track of password

**Constructor:**

Parameter: private \_verifyLoginService: VerifyLoginService

**Class Methods:**

**onSubmit(e):** Connects to button on login screen and does backend verification this.\_verifyLoginService.verifyBackend(e)

**Verify-login.service.ts**

**Constructor:**

Parameter: private http: HttpClient

**Class Methods:**

**verifyBackend(user):** Calls API and gets if the username and password combination is valid and if it is an admin user

verifyBackend(userL) {

return this.http

.post(this.apiURL, {

username: userL.value.user,

password: userL.value.yes

})

.subscribe(

data => {

this.redirect(data)

},

error => {

console.error("Error posting order!");

return false;

}

);

}

**redirect(data):** Takes result from API and redirect the login page to the Order menu page or shows an invalid login popup

redirect(data) {

if (data.isValid && data.isAdmin) {

console.log("here");

location.href = "./order";

} else if (data.isValid && !data.isAdmin) {

location.href = "./order-agent";

} else {

alert("Invalid Login. Check credentials.");

}

return true;

}

**Order-Create:**

File Structure:

omsui/oms/src/app/order-create:

├── `order-create.component.html` - HTML code to render the view of the component

├── `order-create.component.ts` - OrderCreateComponent class to handle the business logic of the component

├── `order-create.component.css` - CSS code to handle the styling

├── `order-create.service.ts` - Service class to retrieve and send data from database using http client api calls

**Order-create.component.ts class structure:**

**Constructor:** Makes an API call to populate the list of available items from the items table in the database. Initializes the order form and sets default values where appropriate.

Two form groups created using Angular FormBuilder.

itemForm: item, quantity, price, subtotal

orderForm: items: FormBuilderArray, channel, date, firstname, lastname, address, city, state, zip, payment, total

**Class Methods:**

**addItem():** Adds item, quantity, price, subtotal values to an Angular Material table and to items array

Ex: "items": [

{ "item": "Item 1", "quantity": "2", "price": 199, "subtotal": 398 },

{ "item": "Item 2", "quantity": "3", "price": 29, "subtotal": 87 },

{ "item": "Item 3", "quantity": "1", "price": 49, "subtotal": 49 }

],

**updateTotal():** Updates the total price of the order dynamically

**removeItem():** Removes an item row from the table and the items array

**createOrder():** Makes api call to order-create service class by sending the orderForm group json object

Ex: {

"items": [

{ "item": "Item 1", "quantity": "2", "price": 199, "subtotal": 398 },

{ "item": "Item 2", "quantity": "3", "price": 29, "subtotal": 87 },

{ "item": "Item 3", "quantity": "1", "price": 49, "subtotal": 49 }

],

"channel": "Online",

"date": "2019-06-12T05:00:00.000Z",

"firstname": "John",

"lastname": "Doe",

"address": "12345 Main St",

"city": "Frisco",

"state": "TX",

"zip": "75033",

"payment": "Credit",

"total": 534

}

**OrderComponent Service Class:**

Makes use of HttpClient to make api calls to the database.

Http.post - Send request for sending new data to the backend service

Http.get - Receive request to receive data from the backend service

**postOrder() -** Send orderForm json data using http.post request

**getItems() -** Receive items json data using http.get request

**Search:**

File Structure:

omsui/oms/src/app/search:

├── `search.component.html` - HTML code to design the look and feel of the search webpage.

├── `search.component.ts` - SearchComponent class to handle input data

├── `search.component.css` - CSS code to handle the styling for the search page

├── `search.component.spec.ts` - File to handle the component