**Back-end of the search bar:**

why did we implement a search bar? it's a very crucial technology that must be implemented in any app and to make it a user friendly app or the user will ignore the app. That being said we decided to make the search bar stunning on both the front-end and back-end.

So how did we do it, we used react native styling for the front-end with a placeholder to make it easier for the user to identify the search bar. This styling consists of a view, a text Input and a button all styled according to the main theme of the app which is blue and grey.

For the back-end of the search bar we used node.js and mongo db to access the database and retrieve whatever the user searched for and present it in a new view screen made only for search purposes and separate it from the main screen where all the services can be found, if the user searched for a service that is not found the search screen will not load anything and notify the user that the service is not found.

this search feature was added to search for services. The search key words are meant for only the title of the service (not the description). The search is not case-sensitive, the user doesn't have to type the exact words to match up to a title but he can write a small keyword and the server will find out if this keyword is included in the title of the service.

e.g.: if a user type the word "shoes" the server will output all the services that have "shoes" in the title or more than this word.

The search route is found in the services routes (/routes/services.js):

// @route GET api/service/search

// @desc search for a service

// @access Private

// @errors noservice error

router.get(

'/search',

passport.authenticate('jwt', { session: false }),

serviceController.search

);

The search controller is found in the services controller (/controllers/serviceController/search.js):

const mongoose = require('mongoose');

// Models

const Service = mongoose.model('service');

module.exports = search = (req, res) => {

const errors = {};

Service.find({ name: new RegExp('^' + req.params.name + '$', 'i') })

.sort({ date: -1 })

.then((services) => {

if (services.length === 0) {

errors.noservices = 'No services found';

return res.status(404).json(errors);

}

return res.json(services);

})

.catch((err) => {

errors.error = 'Error fetching services from database';

res.status(500).json({ ...errors, ...err });

});

};

class SearchResultsScreen extends Component {

static navigationOptions = ({ navigation }) => ({

headerTitle: 'Search Results',

headerStyle: {

backgroundColor: colors.primary,

},

headerTitleStyle: {

color: colors.white,

},

});

render() {

const { navigation, searchedServices, getSearchedServicesLoading } = this.props;

return (

<View style={styles.wrapper}>

{searchedServices && (

<ServicesList

services={searchedServices}

loading={getSearchedServicesLoading}

navigation={navigation}

/>

)}

</View>

);

}

}

**Back-end of the chatting system:**

To make the user experience more engaging and responsive we decided to make a chatting system where the asker can speak with the owner and vice versa about the service that the owner posted and discuss everything about the service and how the owner want the service to be implemented.

Though the chatting system will be between the owner and the asker, the admins of the app can detect any inappropriate language and any agreement made outside the service will also be detected by the admins.

The chatting system is established between two users only, it's a private chat between the service owner and the service helper.

const express = require('express');

const router = express.Router();

const passport = require('passport');

const ChatController = require('../controllers/chatController/index');

// @route POST api/profile/id1+id2

// @desc Open chat between 2 users

// @access Private

// @errors noprofiles error

router.post(

'/:id1+:id2',

passport.authenticate('jwt', { session: false }),

ChatController.openChat

);

module.exports = router;

We used socket.io package to take information from the client and send it to the server then the server distributes the message to the client on the other end.

const io = socket(server);

io.on('connection', function(socket){ // waiting for connection with a client

console.log('socket connection made and the socket id is ' + socket.id);

socket.on('chat', function(data){

// waiting for a data to be send from a client

io.sockets.emit('chat', data)

// send the message back from the server to the rest of the clients

});

});

The chat history will be stored in mongoDB database and it consists of first user id, second user id, message of the first user and message of the second user.

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const ChatSchema = new Schema({

userID1: {

type: mongoose.Schema.Types.ObjectId,

ref: 'user',

required: true

},

msg1: {

type: String,

required: true

},

userID2: {

type: mongoose.Schema.Types.ObjectId,

ref: 'user',

required: true

},

msg2: {

type: String,

required: true

}

});

const Chat = mongoose.model('Chat', ChatSchema);

module.exports = Chat;

As for the front-end of the chatting system we decided to make it more modern and user-friendly so we made the current user messages with the green color and the message of the user on the other end with the grey color, also we used our normal app theme color which is blue and grey for the input text and the send button.

The send button position is not affected by the appearance of the keyboard on the screen and its position will be moved upward until the end of the keyboard.

The messages that are typed by both users have the ability to scroll down on its own so that the user does not have to scroll it down and became a frustrating experience for him.

<View style={styles.wrapper}>

<ScrollView ref={ref => this.scrollView = ref}

onContentSizeChange={(contentWidth, contentHeight)=>{

this.scrollView.scrollToEnd({animated: true});

}}

>

<View>{chatHistory}</View>

<View>{chatMessages}</View>

</ScrollView>

<KeyboardAvoidingView behavior="padding" keyboardVerticalOffset={85}>

<ChatInputItem

autoCorrect={false}

value={this.state.chatMessage}

onPress={() => this.submitChatMessage()}

onChangeText={(chatMessage) => {

this.setState({ chatMessage });

}}

/>

</KeyboardAvoidingView>

</View>