Initial:

Index.css:

Add rule

\*{

  margin: 0;

}

To make things center align horizontally and vertically:

.app{

  display: grid;

  place-items: center;

  height: 100vh;

}

Width using flex:

.sidebar{

    display: flex;

    flex-direction: column;

    flex: 0.35; /\*sidebar takes 35% of screen\*/

}

How to break down layout:



Icon use as button:

 <IconButton>

                    <DonutLarge />

                  </IconButton>

Automatic space between contents:

Use justify content:

.sidebar\_\_header{

    display: flex;

    justify-content: space-between;

    padding: 20px;

}

Code to make search bar thing:

<div className="sidebar\_\_search">

                    <div className="sidebar\_\_searchContainer">

                      <SearchOutlined/>

                      <input type="text" placeholder="Search or start new chat"/>

                    </div>

            </div>

.sidebar\_\_search{

    display: flex;

    align-items: center;

    background-color: #f6f6f6;

    padding: 10px;

    height: 39px;

}

.sidebar\_\_searchContainer{

    display: flex;

    align-items: center;

    width: 100%;

    height: 35px;

    border-radius: 20px;

}

.sidebar\_\_searchContainer > .MuiSvgIcon-root{

    color: grey;

    padding: 10px;

}

.sidebar\_\_searchContainer > input{

    border: none;

    margin-left: 10px;

    outline-width: 0;

}

To take full space:

Use flex: 1

For scroll:

Overflow: scroll

Simple card:



        <div className="sidebarChat">

            <Avatar/>

            <div className="sidebarChat\_\_info">

                <h2>Room name</h2>

                <p>This is the last room</p>

            </div>

        </div>

.sidebarChat{

    display:flex;

    padding: 20px;

    cursor: pointer;

    border-bottom: 1px solid #f6f6f6;

}

.sidebarChat:hover{

    background-color: #ebebeb;

}

.sidebar\_\_chats{

    flex:1;

    background-color: white;

    overflow: scroll;

}

.sidebarChat\_\_info{

    margin-left: 15px;

}

.sidebarChat\_\_info > h2{

    font-size: 16px;

    margin-bottom: 8px;

}

To make content full width as possible:

Use flex 1 (to take all possible vertical width)

Example:



.chat\_\_headerInfo{

    flex: 1;

    padding-left: 20px;

}

More than One class:

<p className="chat\_\_message chat\_\_reciever">

Use form where enter is required:



            <div className="chat\_\_footer">

                <InsertEmoticon/>

                <form>

                    <input placeholder="Type a message" type="text"/>

                    <button type="submit">Send a message</button>

                </form>

                <MicNone/>

            </div>

In css, as we only need enter functionality and don’t need to show button, so we hide it

.chat\_\_footer {

    display: flex;

    justify-content: space-between;

    align-items: center;

    height: 60px;

    border-top: 1px solid lightgray;

}

.chat\_\_footer > form {

    flex: 1;

    display: flex;

}

.chat\_\_footer > form > input {

    flex: 1;

    border-radius: 30px;

    padding: 10px;

    border: none;

}

.chat\_\_footer > form > button {

    display: none;

}

Material Ui Icon class:

.sidebar\_\_searchContainer > .MuiSvgIcon-root

MongoDb backend:

Separate folder for backend

Npm init (entry point -> server.js)

Query Selector:



Code:

div.about\_\_text::before{

    content: "<div>";

    color:#515152;

    font-weight: normal;

    font-family: 'La Belle Aurore', cursive;

    font-size: 18px;

    margin-left: -2%;

    z-index: 100;

    line-height: 0px;

}

div.about\_\_text::after{

    content: "</div>";

    color:#515152;

    font-weight: normal;

    font-family: 'La Belle Aurore', cursive;

    font-size: 18px;

    margin-left: -2%;

}

.about\_\_text > h2::before{

    content: "<h2>";

    margin-left: -2%;

    color:#515152;

    font-size: 16px;

    font-weight: normal;

    font-family: 'La Belle Aurore', cursive;

    font-size: 18px;

}

.about\_\_text > h2::after{

    content: "</h2>";

    margin-left: -2%;

    color:#515152;

    font-size: 16px;

    font-weight: normal;

    font-family: 'La Belle Aurore', cursive;

    font-size: 18px;

}

.about\_\_text > p::before{

    content: "<p>";

    margin-left: -2%;

    line-height: 30px;

    color: #515152;

    font-family: 'La Belle Aurore', cursive;

    font-size: 18px;

}

.about\_\_text > p::after{

    content: "</p>";

    margin-left: -2%;

    line-height: 30px;

    color: #515152;

    font-family: 'La Belle Aurore', cursive;

    font-size: 18px;

}

Select Element of Particular Class:

div.about\_\_text{

    content: "<div>";

    color:#515152;

    font-weight: normal;

    font-family: 'La Belle Aurore', cursive;

    font-size: 18px;

    margin-left: -2%;

    z-index: 100;

    line-height: 0px;

}

Selecting div of class about\_\_text

Button:

import { Link } from 'react-router-dom';

                        <Link to="contact">

                            <button id="btn">

                                Contact Us

                            </button>

                        </Link>

Whatsapp backend:

In backend folder

Npm init

In package.json -> update it by adding start line in under scripts dict

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "start": "node server.js"

  },

After that, we can either install node modules

Npm install

Or we can directly install dependencies which also install node modules

npm i express mongoose

Then create server.js file

MongoDb:

Reference link:

[Get Started with Atlas — MongoDB Atlas](https://docs.atlas.mongodb.com/getting-started/)

It is a no sql db

After that go to mongodb and create new project and free cluster

Then create api

Structure of server.js

//importing

//app config

//middleware

//DB config

//mongodb stuff

//api routes

//listen

Now to do importing like react in server.js file

Add type:module in package.json

{

  "name": "whatsapp-backend",

  "version": "1.0.0",

  "description": "",

  "main": "server.js",

  "type": "module",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "start": "node server.js"

  },

First dummy check api on some base url or endpoint (/) and get response hello world when status is ok (200) on port 9000

//importing

import express from 'express'

//app config

const app = express()

const port = process.env.PORT || 9000

//middleware

//DB config

//mongodb stuff

//api routes

app.get('/', (req, res)=>res.status(200).send('hello world'))

//listen

app.listen(port, ()=>console.log(`Listening on localhost: ${port}`))

now to run this server

install nodemon

npm I –g nodemon

Then write

nodemon server.js

If everything goes right, then server will listen on port 9000 and cmd will return

[nodemon] 2.0.8

[nodemon] to restart at any time, enter `rs`

[nodemon] watching path(s): \*.\*

[nodemon] watching extensions: js,mjs,json

[nodemon] starting `node server.js`

(node:1592) ExperimentalWarning: The ESM module loader is experimental.

Listening on localhost: 9000

Now to get hello world, go to postmen, and get this url and it will return hello world

<http://localhost:9000/>

Now again go to mongodb

Now create user by going onto sidepanel under database access. Make sure to copy password if it is autogenerated

Then go to network access and allow access from current ip address or otherwise from any where

Now after that click on connect in cluster-> connect your application

Copy provided link and again come back in the application

Now its time to connect to db

It would be like

mongodb+srv://admin:<password>@cluster0.kgp7k.mongodb.net/myFirstDatabase?retryWrites=true&w=majority

Replace password and if required then dbname

import mongoose from 'mongoose'

//DB config

const connection\_url = 'mongodb+srv://admin:WrKNg2Jfc5vUkVLw@cluster0.kgp7k.mongodb.net/myFirstDatabase?retryWrites=true&w=majority'

mongoose.connect(connection\_url,{

    useCreateIndex:true,

    useNewUrlParser:true,

    useUnifiedTopology:true

})

This is all needed to connect to mongodb

Create another dbMessage.js file. In that we will write schema

import mongoose from 'mongoose'

//defining schema --> defining how data will be build

// 1.message

// 2.name

// 3.timestamp

const whatsappSchema = mongoose.Schema({

    message: String,

    name: String,

    timestamp: String,

recieved: Boolean,

})

export default mongoose.model('messageCollection', whatsappSchema)

As we can have different documents or collection in one cluster, so we provide collection name in mongoose.model. Now after schema, come back to server.js and use this collection to send messages

import Messages from 'dbMessage.js'

//middleware

app.use(express.json()) //To convert string to json

//DB config

const connection\_url = 'mongodb+srv://admin:WrKNg2Jfc5vUkVLw@cluster0.kgp7k.mongodb.net/myFirstDatabase?retryWrites=true&w=majority'

mongoose.connect(connection\_url,{

    useCreateIndex:true,

    useNewUrlParser:true,

    useUnifiedTopology:true

})

//mongodb stuff

//api routes

app.get('/', (req, res)=>res.status(200).send('hello world'))

app.post('/app/v1/messages/new', (req, res)=>{

    const dbMessage = req.body

    Messages.create(dbMessage, (err, data)=>{

        if (err){

            res.status(500).send(err)

        } else {

            res.status(201).send(data)

        }

    })

})

Now send message using postman

<http://localhost:9000/app/v1/messages/new>

Post

Body

Json

{

    "message": "Sending first message to mongodb",

    "name": "muneeb",

    "timestamp": "dummy timestamp",

    "recieved": **false**

}

And in response it will send data as we have mentioned in app.post

{

    "\_id": "60dc592991159b32b8899d78",

    "message": "Sending first message to mongodb",

    "name": "muneeb",

    "timestamp": "dummy timestamp",

    "recieved": **false**,

    "\_\_v": 0

}

When testing, always makesure that your server is running by using command nodemon server.js

Imp status response

Getting data or downloading data or getting ok -> 200

Error -> 500

Creating something in db or post somethinf -> 201

Now signup on pusher:

It makes mongodb realtime and we don’t need to refresh our app for latest feed

Frontend send msg to backend, pusher takes msg from backend and trigger frontend with the msg



When creating channel for project

Frontend -> react

Backend -> node.js

After creating, it will give us the node js code

We will put that code in server.js file in app config

Also we need to install pusher

Npm I pusher

To trigger pusher, we will add change stream in db config which will be triggered when any change in observed in mongodb

mongoose.connect(connection\_url,{

    useCreateIndex:true,

    useNewUrlParser:true,

    useUnifiedTopology:true

})

const db = mongoose.connection;

db.once('open', ()=>{

    console.log("DB is connected")

    const msgCollection = db.collection('messagecollections');//same name for collection

    const changeStream = msgCollection.watch();

    changeStream.on("change", (change)=>{

        console.log("A change occured", change)

    })

})

Store connection const db, when once db gets open we got identified

Now store our collection in msgCollection const

And change stream will watch change in msgCollection or in our collection

When any change happens, it will give us in console

Test it by sending msg in postmen, it will console the data in real time in CMD (console)

Now its time to trigger pusher when change occurs

 changeStream.on("change", (change)=>{

        console.log("A change occured", change)

        if (change.operationType == 'insert'){

            const messageDetails = chnage.fullDocument;

            pusher.trigger('messages', 'inserted', {

                name: messageDetails.name,

                message: messageDetails.message

            })

        } else {

            console.log('Error in triggering pusher')

        }

Adding headers to make app able to deploy on heruko

First way

//middleware

app.use(express.json()) //To convert string to json

app.use((req, res, next)=>{

    //headers for heruko

    //As access from anywhere, so no security in it

    res.setHeader("Access-Control-Allow-Origin", "\*"); //req from any end point

    res.setHeader("Access-Control-Allow-Headers", "\*"); //accept headers from everywhere

    next();

})

Second way:

Npm I cors

import cors from 'cors';

//middleware

app.use(express.json()) //To convert string to json

app.use(cors())//headers for heruko

Backend completed:

Now connect pusher to frontend so it triggers frontend when db changes

UseEffect:

It is used to run code inside it once when the app loads

In app.js, from getting started in pusher, paste this code under useEffect

  useEffect(()=>{

    const pusher = new Pusher('1da9a361a566d7dc83bc', {

      cluster: 'ap2'

    });

    const channel = pusher.subscribe('messages');

    channel.bind('inserted', (data)=> {

      alert(JSON.stringify(data));

    });

  }, [])

Install pusher-js in frontend

Npm I pusher-js

And import it

import Pusher from 'pusher-js'

Now when we post api from postman, we get data in form of alert 

Which means our app is sync with the mongodb backend

Create axios.js file where we place our backend url to fetch data or to use backend apis

Axios is used to fetch or to interact frontend with backend apis

import axios from 'axios'

const instance = axios.create({

    baseURL: "http://localhost:9000",

});

export default instance;

When deploy on heruko, we just need to change baseUrl

Coming back to app.js

import './App.css';

import {useEffect, useState} from 'react'

import Chat from './components/chat/Chat';

import Sidebar from './components/sidebar/Sidebar';

import Pusher from 'pusher-js'

import axios from './axios'

function App() {

  const [messages,setMessages] = useState([]);

  //useEffect for fetching

  useEffect(()=>{

    axios.get('/messages/sync')

      .then(response=>{

        setMessages(response.data)

      })

  }, [])

 useEffect(()=>{

    const pusher = new Pusher('1da9a361a566d7dc83bc', {

      cluster: 'ap2'

    });

    const channel = pusher.subscribe('messages');

    channel.bind('inserted', (newMessage)=> {

      alert(JSON.stringify(newMessage));

      setMessages([...messages, newMessage])

    });

    return ()=>{

      channel.unbind\_all(); // to unbind so it will no listen to all messages everytime, only listen new message

      channel.unsubscribe();

    };

  }, [messages]) //add message here so it updates with the message

  console.log(messages);

First useEffect to fetch messages from api. Now we need to set and update messages. For that useState is used and we update messages everytime by using setMessages.

Second useEffect is for pusher. Pusher give us the changed data which is an object. So what we do is kept all messages […messages] and also add new message and update using setMessage function

Pass message object to chat component so it can be used after destructuring

<Chat messages={messages}/>

In chat,

function Chat({messages}) {

    const [input, setInput] = useState('');

    const sendMessage = (e) => {

        e.preventDefault(); //avoiding from refresh

    }

Update chat body div as

            <div className="chat\_\_body">

                {messages.map((message) =>(

                    <p className={`chat\_\_message ${message.recieved && "chat\_\_reciever"}`}>

                        <span className="chat\_\_name">{message.name}</span>

                        {message.message}

                        <span className="chat\_\_timestamp">

                            {message.timestamp }

                        </span>

                    </p>

                ))}

            </div>

Now to post message to api and keep track of send message and input field:

 const [input, setInput] = useState('');

    const sendMessage = async (e) => {

        e.preventDefault(); //avoiding from refresh

    await axios.post('/messages/sync',{

        "message": input,

        "name": "muneeb",

        "timestamp": "Just now!",

        "recieved": true,

    });

    setInput('');

    }

            <div className="chat\_\_footer">

                <InsertEmoticon/>

                <form>

                    <input value={input} onChange={e => setInput(e.target.value)}placeholder="Type a message" type="text"/>

                    <button onClick={sendMessage} type="submit">Send a message</button>

                </form>

                <MicNone/>

            </div>

Use of classes based on condition:

 <p className={`chat\_\_message ${message.recieved && "chat\_\_reciever"}`}>

Use chat\_\_message class, if message.recieved is true, then use chat\_\_reciever class

Using two classes:

<p className="chat\_\_message chat\_\_reciever">