Clinic-Connect

Frontend

# How to add favicon in React?

Ans)

First create the logo, convert that to .ico file type, save that as favicon.ico basically overwrite the existing favicon.ico in the react project.

If you want to resize/restrict the favicon size, you can do it in manifest.json, and change it

"icons": [

    {

      "src": "favicon.ico",

      "sizes": "64x64 32x32 24x24 16x16",

      "type": "image/x-icon"

    },

Now it should automatically render in the tab. Else just stop and start the project.

# How to add images in React?

Ans)

Basically store all the images under the public folder of the react.

Create a new folder called images and store our project images in there.

<div className='header-nav-logo' >

                    <img src='/images/ClinicConnectlogo.png' alt="ClinicConnect-logo"/>

                </div>

Now if you are creating a web application there will be a primary navbar, which is available for both unauthenticated (Forbidden) and authenticated user.

And the other navbar which is only visible when the user’s logs in to the platform.

# PRIMARY HEADER

This is comething that we might use frequently in all the pages. So it better to create a component called <Header/>

In React in order to use <a> </a> anchor tag, we need to use <NavLink> or <Link>, during runtime this <NavLink> or <Link> will be turned into <a> tag by the react

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

<ul>

                        <li><NavLink to="/" >Home</NavLink></li>

                        <li><NavLink to="/links" >Pages</NavLink></li>

                        <li><NavLink to="/doctors" >Doctors</NavLink></li>

                        <li><NavLink to="/aboutus" >About Us</NavLink></li>

                    </ul>

Since we are planning to create a responsive frontend, so we need to make Header responsive for all devices.

CODE:

<header>

            <nav className='header-nav' >

                <div className='header-nav-logo' >

                    <img src='/images/ClinicConnectlogo.png' alt="ClinicConnect-logo"/>

                </div>

                <div className='header-nav-links' >

                    <ul>

                        <li><NavLink to="/" >Home</NavLink></li>

                        <li><NavLink to="/links" >Pages</NavLink></li>

                        <li><NavLink to="/doctors" >Doctors</NavLink></li>

                        <li><NavLink to="/aboutus" >About Us</NavLink></li>

                    </ul>

                </div>

                <button className='header-btn-mobile-nav' onClick={handleClick} >

                    <i className='icon-mobile-nav' name='menu'><img src="/images/icons/hamburger-menu.png" alt="click" /></i>

                    <i className='icon-mobile-nav' name='close' ><img src="/images/icons/close-menu.png" alt="click"/></i>

                </button>

            </nav>

        </header>

const handleClick = () => {

        const headerEl = document.querySelector(".header-nav");

        headerEl.classList.toggle("nav-open");

    }

So when the button is clicked, it calls the handleClick method, which takes no parameter when called. So it traverse through the above code and finds .header-nav.

.toggle() is basically off if on and on if off. If the button is clicked then toggle will close the nav bar or open the navbar depending upon the state of the nav bar.

Making the Responisve header/ NavBar is the most difficult part when it comes to creating the responsive layout for project.

But before moving to that we should adhere to some best practices in order minimize our effort on making responsive design

They are:

INDEX.CSS

\* {

  padding: 0;

  margin: 0;

  box-sizing: border-box;

}

html{

  font-size: 62.5%;

}

body{

  line-height: 1;

  font-weight: 400;

  font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;

}

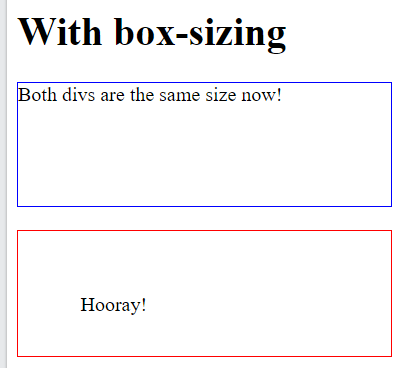
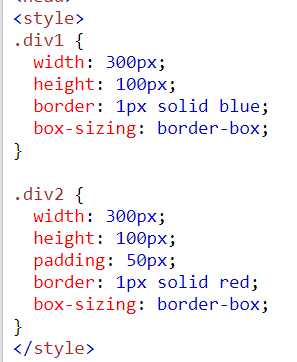
Basically 1rem = 16px by default, but we want to work with 1rem as 10px. So that’s why we do

Fontsize = 62.5% , basically ( 10/16) \*100 .

What’s Border-Box ?

The box-sizing property allows us to include the padding and border in an element's total width and height.

If you set box-sizing: border-box; on an element, padding and border are included in the width and height:



APP.CSS

.container{

    max-width: 121rem;

    margin: 0 auto;

}

No it is easier to make our layout responsive to change in pixels.

header{

    /\* background-color: linear-gradient(to right, rgba(190, 85, 85, 0), rgba(255,0,0,1)); Will not work\*/

    background: linear-gradient(to right, rgba(242, 245, 245, 0.7), rgb(230, 228, 228));

}

.header-nav{

    margin:0 auto;

    display: flex;

    justify-content: space-between;

    max-width: 121rem;

    padding: 1rem 1rem;

    position: relative;

}

.header-nav-logo img{

    height: 6rem;

    width: 60%;

}

.header-nav-links ul {

    list-style-type: none;

    display: flex;

    gap: 2rem;

    padding-top: 1.5rem;

    font-size: 2.4rem;

}

.header-nav-links ul li a{

    color: #135D66;

    font-weight: 400;

    text-decoration: none;

}

.header-nav-links ul li a:hover{

    color: #135D66;

    font-weight: 600;

    text-decoration-line: underline;

}

.header-btn-mobile-nav{

    display: none;

    font-size: 2rem;

    border: none;

    background: none;

    margin: 1rem;

    width: 4rem;

    height: 4rem;

    padding-top:1rem;

}

/\* hiding the close button \*/

.icon-mobile-nav[name="close"]{

    display: none;

}

The name = “ close” is used to uniquely identify a class

Inorder make our design responsive we need to find the breakpoints. Basically where our design is not properly aligned/ too congested/ too big/too small.

Finding breakpoint in a design can be done in trail and error method. We use browser developer tools to find the breakpoints in out design.

For our project we found a breakpoint at 35em, em is nothing but rem, both are essentionally the same.

What is 35em mean?

35em means 35 x 16px = 560, why 16px for multiplying ?, because 1em or 1rem = 16px

At 560px in width there was some flaw in the UI so it has to corrected, so we mark this point as breakpoint we divide this point 16px in order to get em.

**NOTE: The bottom-most style in the css file will given higher prefernce than the top-most style, if both of them are deaeling with same attributes.**

@media (max-width:35em){

    .header-btn-mobile-nav{

        display: block;

        position:absolute;

        top:0;

        right:0;

        z-index:9999;

    }

    .header-nav-links{

        background: linear-gradient(to right, rgba(242, 245, 245, 1), rgb(188, 185, 185));

        position: absolute;

        top:0;

        left:0;

        width: 100%;

        height:100vh;

        display: flex;

        align-items: center;

        justify-content: center;

        transition: all 0.5s;

        opacity: 0;

        pointer-events: none;

        visibility: hidden;

    }

    .nav-open .header-nav-links{

        opacity: 1;

        pointer-events: auto;

        visibility: visible;

    }

    .nav-open .icon-mobile-nav[name="close"]{

        display: block;

    }

    .nav-open .icon-mobile-nav[name="menu"]{

        display: none;

    }

    .header-nav-links ul{

        gap:4.8rem;

        flex-direction: column;

    }

    .header-nav-links ul li{

        font-size: 2.25rem;

        text-align: center;

    }

    .header-nav-links ul li:hover{

        color: #135D66;

        font-weight: 600;

    }

}

You set make the mobile-nav to display none. As it will not render the entire links part of the nav bar, so we must do this

.header-nav-links{

        /\* styles \*/

        opacity: 0;

        pointer-events: none;

        visibility: hidden;

}

.nav-open .header-nav-links{

opacity: 1;

        pointer-events: auto;

        visibility: visible;

    }

    .nav-open .icon-mobile-nav[name="close"]{

        display: block;

    }

    .nav-open .icon-mobile-nav[name="menu"]{

        display: none;

    }

Seeting opacity = 0, is like making them transparent, visibility = hidden is hidding them, and pointer events = none, is no matter what ever you clik or scroll, its not going to react.

But if the the button is clicked then the navopen classname will be added to the div, no the div will visible, and you can do click actions etc on it.

This was about the primary header/Navbar. Once we go to the Login Process we can discuss about the usernavbar

# Now the Footer

Again since we will be using the footer in several pages, we can create a componet and export and use it everywhere in our website

Creating a footer componet is not that difficult, but designing & aligning it is quite difficult

footer {

    background-color: #135D66;

    display: flex;

    justify-content: center;

    align-items: center;

    color: aliceblue;

    height: 4rem;

    font-size: 1.4rem;

    position: fixed;

    right: 0;

    left: 0;

    bottom: 0;

    width: 100%; /\* Ensure it spans the full width \*/

}

This will make the footer stick to the bottom of the view port. But if you want to create a footer that will change its position according to there webpage current content, then we should create a container/last-container and accoding to the device we need the increase and decrease the size of the last-container.

This last-container is used only for the design correction of the footer, since you cant solve this issue from footer styling itself, that why we need a container which can oush the footer down when the viewport is huge more than 2000px.

I havent implement this solution in the poject, this is something that striked me on the goo.

But what did was I increased the margin bottom of the container previous to the footer which is not the best practice ofcourse.

# Now time for UserNav

UserNav should contain few links to travel within the account and also a logout button.

These are necessary.

CODE:

import React from 'react'

import { useNavigate,NavLink } from 'react-router-dom'

import axios from 'axios';

import '../../App.css';

import './Usernav.css';

export default function Usernav() {

    // is used to navigate / redirect when logged out

    const navigate = useNavigate();

    // triggered when the user press the logout button

    const handleLogout = async(e) => {

        e.preventDefault();

        const jwtToken = localStorage.getItem('bmdv3ujwtToken');

        try{

            const response = await axios.post('http://localhost:8080/logoutuser',{

                headers:{

                    Authorization: `Bearer ${jwtToken}`

                }

            });

            console.log('Logout Successful',response.data);

            localStorage.removeItem('bmdv3uid');

            localStorage.removeItem('bmdv3uname');

            localStorage.removeItem('bmdv3uemail');

            localStorage.removeItem('bmdv3uphone');

            localStorage.removeItem('bmdv3ujwtToken');

            setTimeout(() => {

                navigate('/');

            }, 3000); // Wait for 3 seconds before navigating to /loginpage

        }catch(error){

            console.log('Error in log out', error);

        }

    };

    // showing user account details

    const uid = localStorage.getItem('bmdv3uid');

    const uname = localStorage.getItem('bmdv3uname');

    const uemail = localStorage.getItem('bmdv3uemail');

    const uphone = localStorage.getItem('bmdv3uphone');

    // add jwt in the UserHomePage

  return (

    <>

        <nav className='userlinks-nav'>

            <div className=' container userlinks'>

                <ul>

                    <li> <NavLink to={`/user/${uname}/history`}>History</NavLink> </li>

                    <li><NavLink to={`/user/${uname}/MakeAppointment`} >Make-Appointment</NavLink></li>

                    <li>

                        <button onClick={handleLogout} ><img src="/images/icons/logout.png" alt="logout"/></button>

                    </li>

                </ul>

            </div>

        </nav>

        <section className='userdetails-nav'>

            <div className='container usernav-container'>

                <div className='useraccount'>

                    <ul>

                        <li><NavLink to={`/user/${uname}`}>Home</NavLink></li>

                        <li><b>User ID:</b> {uid}</li>

                        <li><b>Username:</b> {uname}</li>

                        <li><b>Email:</b> {uemail}</li>

                        <li><b>Phone:</b> {uphone}</li>

                    </ul>

                </div>

            </div>

        </section>

    </>

  )

}

Once the logout is successfully, we remove all the stored data about the user from the localstorage. This will make sure that the user cant use the back button by the browser to access the contents, so the user has to login again.

The customized NavLink uses formated string, this is basically used to embedd a value inside the string and $ is used to do it

const response = await axios.post('http://localhost:8080/logoutuser',{

                headers:{

                    Authorization: `Bearer ${jwtToken}`

                }

            });

This is something that we must follow, if you want to send a token to the server. Then we must follow this rule.

The localstorage stores all the basic information about the user such as account\_number, name , mobile number, jwttoken.

Once the user clicks logout button. All these information will be earsed, so the user need to login freshly in order to access his/her account.

Similarly for the doctors we can do, only thing is variable name change.

# APP.js the routing table

App.js is going to be the mock routing table inside the website

import { BrowserRouter,Routes ,Route } from 'react-router-dom'

export default function App() {

  return (

    <>

        <BrowserRouter>

          <Routes>

            <Route path="/" element={<IndexPage/>}/>

            <Route path="\*" element={<PageNotFound/>} />

            <Route path="/links" element={<LinksPage/>}/>

            <Route path="/aboutus" element={<AboutPage/>} />

            <Route path="/doctors" element={<DoctorListPage/>} />

            <Route path="/createaccount" element={<CreateAccountPage/>} />

            <Route path="/userlogin" element={<UserLoginPage/>}/>

            <Route path="/doctorlogin" element={<DoctorLoginPage/>}/>

            <Route path="/user/:uname" element={<UserHomePage/>} />

            <Route path="/user/:uname/history" element={<UserPrevPage/>} />

            <Route path="/user/:uname/MakeAppointment" element={<BookAppointement/>} />

            <Route path="/doctor/:dname" element={<DoctorHomePage/>}/>

            <Route path="/doctor/:dname/history" element={<DoctorPrevPage/>}/>

          </Routes>

        </BrowserRouter>

    </>

  )

}

# GENERAL PAGES

# IndexPage.js

The backgound image

<div className='background-index-img'>

          <div className='background-index-overlay'>

            <section className='container index-intro-container' >

              <main>

                <div className='intro-content' >

                  <h1>Booking and Handling appointments made easy with ClinicConnect.</h1>

                  <p>Our platform redefines the appointment creation and management experience. Enjoy a streamlined, user-friendly system that simplifies the entire process, ensuring efficiency and ease for all users.</p>

                  <button onClick={handleGotopages} >Make Appointemnt</button>

                </div>

              </main>

            </section>

          </div>

        </div>

.background-links-img {

    background-image: url("/public/images/medical-links-img.jpg");

    background-size: cover;

    background-position: center;

    background-repeat: no-repeat;

}

.background-links-overlay{

    background-image: linear-gradient(to bottom, rgba(115, 234, 194, 0.377), rgba(194, 206, 202, 0.381));

}

const Navigate = useNavigate();

  const handleGotopages=(e)=>{

    Navigate('/links');

  }

<div className='background-index-services'>

          <section className='container index-services-container'>

            <div className='services-grid'>

              <div>

                <h2>Appointment</h2>

                <p>Any problems or discomfort experiencing? Make an appointment with our experts to get back on track.</p>

                <button onClick={handleGotopages}>Appointment Now</button>

              </div>

              <div>

                <h2><img src="/images/icons/emergency-call.png" alt=" "/> Emergency Cases.</h2>

                <h3>+99-55-66-88-526</h3>

                <p>You can call us for Emergency and make an appointment out of our clinic</p>

                <button onClick={handleGotopages} >Appointment Now</button>

              </div>

              <div>

                <h2><img src="/images/icons/opening-hours.png"  alt=" "/> Opening Hours.</h2>

                <p>Mon-Fri &emsp;  &emsp;  &emsp; &emsp;   8:00-21:00</p>

                <p>Saturday&emsp;   &emsp;   &emsp; &emsp; 8:00-21:00</p>

                <p>Sunday &emsp;    &emsp;  &emsp;  &emsp;&ensp;8:00-21:00</p>

              </div>

            </div>

          </section>

        </div>

# CREATE ACCOUNT PAGE.js

Why do we need useState() ?

The React useState Hook allows us to track state in a function component. State generally refers to data or properties that need to be tracking in an application.

What is Hook in react?

Hooks are functions that connects react state and lifecycle features from a funcitonal component. Hooks decreased the popularity of using class based components.

 //Users Datatypes

    const [username,setUsername] = useState('');

    const [password,setPassword] = useState('');

    const [email,setEmail] = useState('');

    const [phone,setPhone] = useState('');

    const [role,setRole] = useState('user');

    const handleUsernameChange = (e) => {

        setUsername(e.target.value);

    };

    const handlePasswordChange = (e) => {

        setPassword(e.target.value);

    };

    const handleEmailChange = (e) => {

        setEmail(e.target.value);

    };

    const handlePhoneChange = (e) => {

        setPhone(e.target.value);

    };

handleSubmit is going to be async, if you have backend

const handleSubmit = async(e) => {

        e.preventDefault();

        console.log('Username:', username);

        console.log('Password:', password);

        console.log('Email:', email);

        console.log('Phone:', phone);

        try{

            const response = await axios.post('http://localhost:8080/Createaccount',{

                uname: username,

                upassword: password,

                uemail: email,

                uphone: phone,

                rolee: role,

            });

            if(response.data){

                setCreateresponse(true);

                setCreateresponsemsg('Account created successfully!');

            }else{

                setCreateresponse(true);

                setCreateresponsemsg('Account with username already exists');

            }

        }catch(error){

            console.log('Error Creating account',error);

            setCreateresponse(true);

            if (error.message.includes('ERR\_CONNECTION\_REFUSED')) {

                setCreateresponsemsg('Cannot register your account, Server Down');

            } else {

                setCreateresponsemsg('Server Down, We will be back online soon');

            }

        }

        // Reset form fields

        setUsername('');

        setPassword('');

        setEmail('');

        setPhone('');

    };

Now its going to await it will gets some response from the backend

The Form:

We need a value, and if it’s a input field, then we need a onchange which points to a method,

Required make these field madatory

The input field:

<input

Type=””

Value=””

onChange=””

required / or not

/>

<form onSubmit={handleSubmit}>

                            <label>

                                Username:

                                <input

                                    type="text"

                                    value={username}

                                    onChange={handleUsernameChange}

                                    required

                                />

                            </label>

                            <label>

                                Password:

                                <input

                                    type="password"

                                    value={password}

                                    onChange={handlePasswordChange}

                                    required

                                />

                            </label>

                            <label>

                                Email:

                                <input

                                    type="text"

                                    value={email}

                                    onChange={handleEmailChange}

                                    required

                                />

                            </label>

                            <label>

                                PhoneNo:

                                <input

                                    type="text"

                                    value={phone}

                                    onChange={handlePhoneChange}

                                    required

                                />

                            </label>

                            <button type="submit">Create Account</button>

                            {createresponse && (

                                <p>

                                {createresponsemsg}

                                </p>

                            )}

                        </form>

Finally

<form onSubmit={handleSubmit} >

<button type=”submit”>submit</button>

</form>

Is the format for submitting the form, using this no need to manually use the mouse to click submit. Just do Enter.

# USER LOGIN PAGE

<>

        <Header/>

        <div className='background-form-img'>

            <div className='background-form-overlay'>

                <section className='container account-container'>

                    <div className='account-form'>

                        <img src="/images/ClinicConnectlogo.png" alt="clinicconnect logo" />

                        <h1>Users Login</h1>

                        <form onSubmit={handleSubmit}>

                            <label>

                                Username:

                                <input

                                    type="text"

                                    value={username}

                                    onChange={handleUsernameChange}

                                    required

                                />

                            </label>

                            <label>

                                Password:

                                <input

                                    type="password"

                                    value={password}

                                    onChange={handlePasswordChange}

                                    required

                                />

                            </label>

                            <button type="submit">Login</button>

                            {uloginflag && <p>{uloginmsg} </p>}

                        </form>

                    </div>

                </section>

            </div>

        </div>

        <Footer/>

For postmapping:

 const handleSubmit = async (e) => {

        e.preventDefault();

        console.log('Username : ', username);

        console.log('Password : ', password);

        try {

            const response = await axios.post('http://localhost:8080/Userslogin', {

                uname: username,

                upassword: password,

            });

            console.log('Response data: ', response.data);

            if (response.data) {

                const { uid, uname, uemail, uphone, token, islogged } = response.data;

                setUloginflag(islogged);

                localStorage.setItem('bmdv3ujwtToken', token);

                localStorage.setItem('bmdv3uid', uid);

                localStorage.setItem('bmdv3uname', uname);

                localStorage.setItem('bmdv3uphone', uphone);

                localStorage.setItem('bmdv3uemail', uemail);

                console.log('Users Jwt token : ', token);

                setUloginmsg('login success');

                setTimeout(() => {

                    navigate(`/user/${uname}`);

                }, 1500);

            } else {

                setUloginflag(true);

                setUloginmsg('error in Backend');

                console.log('error in Backend');

            }

        } catch (error) {

            if (error.response && error.response.status === 401) {

                setUloginflag(true);

                setUloginmsg('Unauthorized: Invalid Credentials');

                console.log('Unauthorized: Invalid Credentials', error);

            } else {

                setUloginflag(true);

                if (error.message.includes('ERR\_CONNECTION\_REFUSED')) {

                    setUloginmsg('Server Down, We will be back online soon');

                } else {

                    setUloginmsg('Server Down, We will be back online soon');

                }

            }

        }

    };

We are storing all the user details in localstorage, so that once the user is logged in we can show their account infromations.

Similarly for the DoctorLoginPage.js

# USER HOME PAGE

import React , { useState,useEffect }from 'react'

import {Navigate } from 'react-router-dom'

import axios from 'axios';

these are the essential hooks and libraries needed

<>

        <Header/>

        <Usernav/>

        <section className='container userhome-container'>

            <div className='userhome'>

                <h1>My Current Appointments</h1>

                {currentAppoFlag && <p className="totalappo">{currentAppointmentsMsg}</p>}

                <div className='appointmentshow'>

                    {currentAppointments.map((appointment, index) => (

                        <div key={index} className="appointmentcard">

                            <div className='beautify-card'>

                            </div>

                            <div className='appointmentdetails'>

                                <p><b>Appointment ID:</b> {appointment.aid}</p>

                                <p><b>Patient name:</b> {appointment.pname}</p>

                                <p><b>Patient age:</b> {appointment.page}</p>

                                {/\* <p><b>Patient Phone No:</b> {appointment.pphone}</p> \*/}

                                {/\* <p><b>Booking Name:</b> {appointment.auname}</p> \*/}

                                <p><b>Doctor name:</b> {appointment.adname}</p>

                                <p><b>Date:</b> {appointment.adate} <b>Time:</b> {appointment.aslot}</p>

                                <button type="button" onClick={() => handleCancel(appointment.aid, appointment.adname, appointment.adate, appointment.aslot)}>Cancel</button>

                            </div>

                        </div>

                    ))}

                </div>

            </div>

        </section>

        <Footer/>

    </>

Sytanx for readering an array of objects:

{

Arr.map((item, idx)=>{

<div key={idx}>

<p> item.something</p>

</div>

})

}

Is an anoynmous function.

If we want to do something for a particular card, then the card needs to have a button.

Lets say we have need to cancel appointment.

<button type="button" onClick={() => handleCancel(appointment.aid, appointment.adname,

appointment.adate,

appointment.aslot)}>Cancel</button>

is the syntax, for whatever we say early was for form submit button

if nothing needed to be passed to the backend, then we do something like this

const Navigate = useNavigate();

  const handleGotopages=(e)=>{

    Navigate('/links');

  }

<button onClick={handleGotopages} >Make Appointemnt</button>

The userhome page provides

Links to other pages

Fetch appointments and renders it

Handle cancel appointments

Make sure its protected

 const fetchAppointments = async () => {

    try {

        const response = await axios.post('http://localhost:8080/Userpage/ShowCurrentUserAppointments', {

            uid: uid,

            uname: uname,

        }, {

            headers: {

                Authorization: `Bearer ${jwtToken}`,

            },

        });

        if (response.data) {

            if (response.data.length === 0) {

                setCurrentAppoFlag(true);

                setCurrentAppointmentsMsg('No current appointments');

                console.log('No current appointments');

            } else {

                const tot = response.data.length;

                setCurrentAppoFlag(true);

                setCurrentAppointmentsMsg(`Total Current Appointments: ${tot}`);

                setCurrentAppointments(response.data);

                console.log('Current appointments:', response.data);

            }

        }

    } catch (error) {

        setCurrentAppoFlag(true);

        setCurrentAppointmentsMsg('Cannot fetch appointments');

        console.log('Cannot fetch appointments', error);

    }

};

const response = await axios.post('http://localhost:8080/Userpage/ShowCurrentUserAppointments', {

            uid: uid,

            uname: uname,

        }, {

            headers: {

                Authorization: `Bearer ${jwtToken}`,

            },

        });

A typicall request from the frontend with payload and authentication header.

const handleCancel = async (aid, adname, adate, aslot) => {

    try {

        await axios.post('http://localhost:8080/Userpage/CancelAppointment', {

            aid: aid,

            auid: uid,

            auname: uname,

            adname: adname,

            adate: adate,

            aslot: aslot,

        }, {

            headers: {

                Authorization: `Bearer ${jwtToken}`,

            },

        });

        fetchAppointments();

    } catch (error) {

        console.error('Error in canceling appointment:', error);

    }

};

The reason why fetchAppointment is used again, since a change in the number of items.

We can also remove fetchAppointment() from the function, but when you cancel the appointment, then you may need to reload the page in order to see the change. It wont happen in front of your eyes.

But if you wan to see the change then you need to use fetchAppointments() inside the cancel appointments

# Whats useEffect, why useEffect?

useEffect(() => {

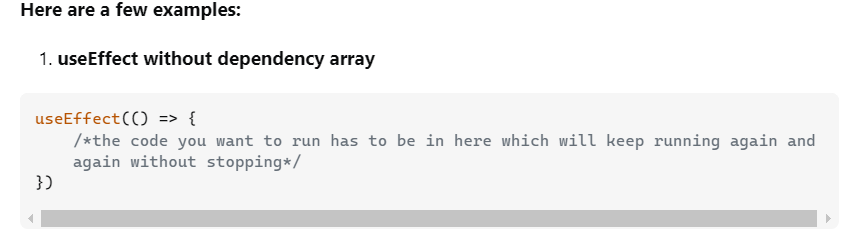
    fetchAppointments();

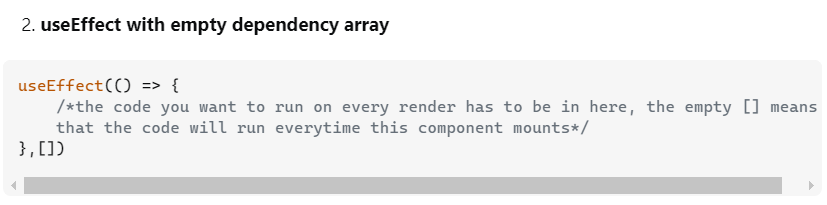
}, [uid, uname, jwtToken]);

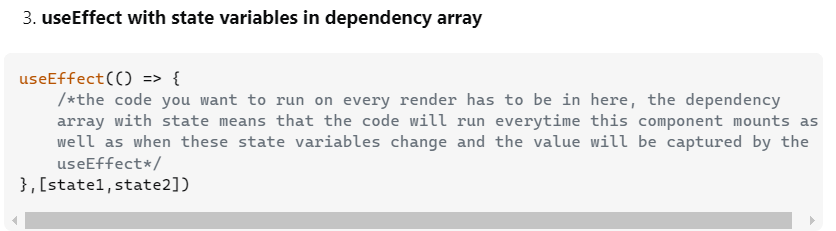
useEffect is one of the most important hooks in react and is a way to handle life cycle of the component in which it is present.

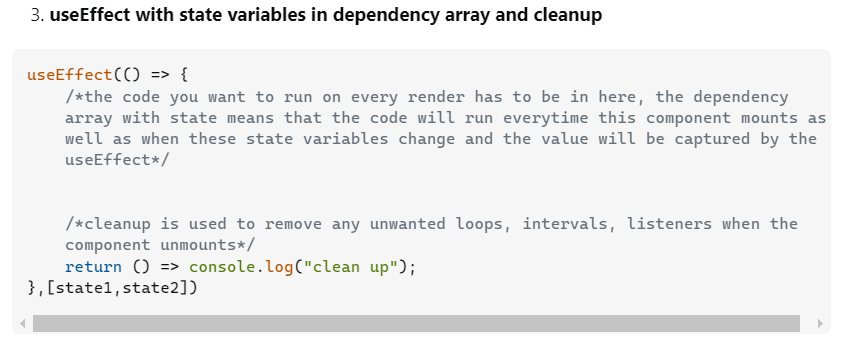
useEffect runs on every render of the component (i.e when a state variable changes) and can also run every time a specific value changes that is mentioned in it's dependency array.

useEffect also provides a very useful cleanup function which can be used to remove any active listeners when the component changes









A solid example will be :

import { useState, useEffect } from "react";

import ReactDOM from "react-dom/client";

function Counter() {

const [count, setCount] = useState(0);

const [calculation, setCalculation] = useState(0);

useEffect(() => {

setCalculation(() => count \* 2);

}, [count]); // <- add the count variable here

return (

<>

<p>Count: {count}</p>

<button onClick={() => setCount((c) => c + 1)}>+</button>

<p>Calculation: {calculation}</p>

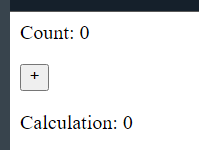
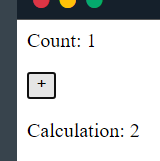
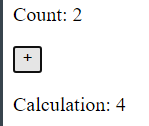
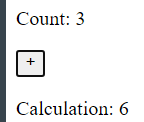
</>

);

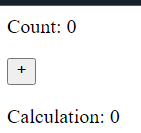
}

const root = ReactDOM.createRoot(document.getElementById('root'));

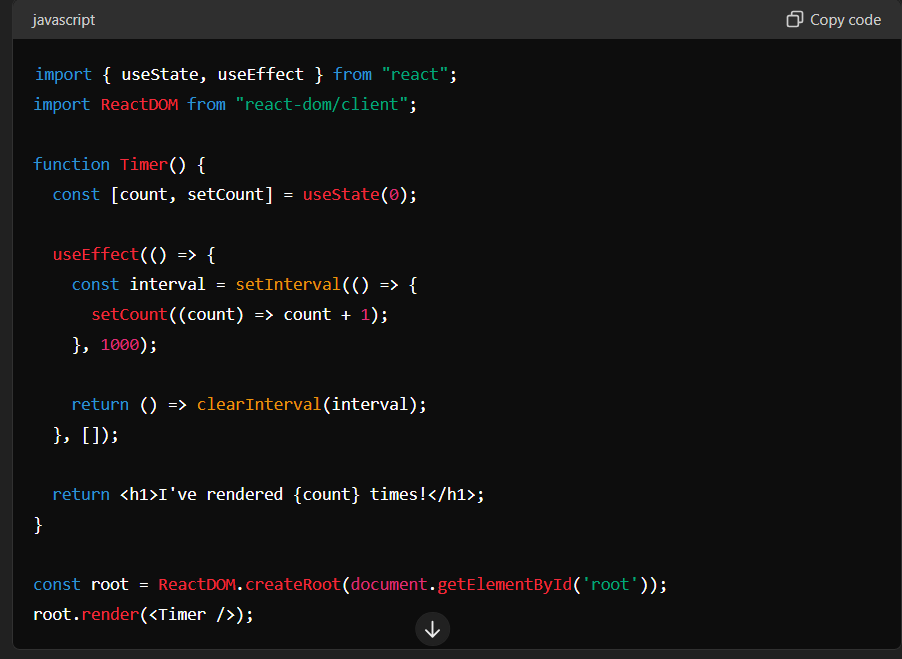
root.render(<Counter />);

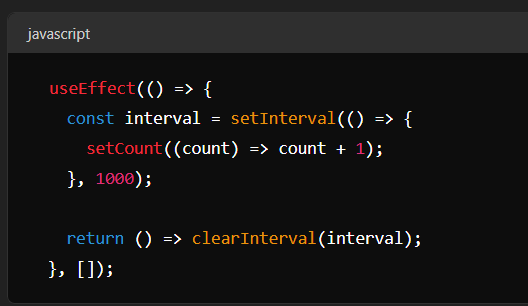
   

When I click reload



EX 2:





useEffect is used to handle side effects, which in this case is setting up an interval to update the count state variable every second.

setInterval is called with a delay of 1000 milliseconds (1 second). It repeatedly executes the callback function every second.

The callback function setCount((count) => count + 1) increments the count state variable by 1.

The return function () => clearInterval(interval) is a cleanup function that clears the interval when the component is unmounted or before the effect runs again. This prevents memory leaks.

The empty dependency array [] ensures that this effect runs only once, when the component mounts.

And for protecting the UserHomePage from unauthenticated users we can do something like this

if (!jwtToken) {

    return <Navigate to="/" />;

}

Remember when we login it creates a token and stores it in browser localstorage.

# USER PREVIOUS APPOINTMENTS

To fetch the preious appointments

const fetchAppointments = async () => {

        try {

            const response = await axios.post('http://localhost:8080/Userpage/ShowPreviousUserAppointments', {

                uid: uid,

                uname: uname,

            }, {

                headers: {

                    Authorization: `Bearer ${jwtToken}`,

                },

            });

            if (response.data) {

                if(response.data.length===0){

                    setPrevAppoFlag(true);

                    setPrevAppointmentsMsg('No Previous Appointments');

                    console.log('No previous appointments');

                }

                else{

                    const tot = response.data.length;

                    setPrevAppoFlag(true);

                    setPrevAppointmentsMsg(`Total Appointments: ${tot}`);

                    setPreviousAppointments(response.data);

                }

            }

        } catch (error) {

            console.log('Cant Fetch appointments', error);

        }

    };

    useEffect(() => {

        fetchAppointments();

    }, [uid, uname, jwtToken]);

Again few things repeating. So everytime we visit the userprevpage, the components mounts in the index.html by the react, the useEffect runs all the code nested inside it and also, if there is any change in the variables in the dependency array it will run the useEffect code again.

useEffect( annoymous function , dependency array) so 2 arguments for useEffect

# BOOK APPOINTMENTS

<form onSubmit={handleSubmit}>

                        <label>

                            Patient Name:

                            <input type="text" value={pname} onChange={handlePnameChange} required />

                        </label>

                        <br />

                        <label>

                            Patient age:

                            <input type="number" value={page} onChange={handlePageChange} required />

                        </label>

                        <br />

                        <label>

                            Patient Phone no:

                            <input type="text" value={pphone} onChange={handlePphoneChange} required />

                        </label>

                        <br />

                        <label>

                            Choose Doctor:

                            <select value={adid} onChange={handleAdidChange} required>

                                <option value="">Select Doc</option>

                                {doctorlist.map((doctor, index) => (

                                    <option key={index} value={doctor.did}>

                                        {doctor.dname}, {doctor.specialization}

                                    </option>

                                ))}

                            </select>

                        </label>

                        <br />

                        <label>

                            Appointment Date:

                            <input type="date" value={adate} onChange={handleAdateChange} required />

                        </label>

                        <br />

                        <label>

                            Appointment Slot:

                            <select value={aslot} onChange={handleAslotChange} required>

                                <option value="">Select Time</option>

                                <option value={doctorlist.find((doctor) => doctor.did === parseInt(adid))?.slot1}>

                                    {doctorlist.find((doctor) => doctor.did === parseInt(adid))?.slot1}

                                </option>

                                <option value={doctorlist.find((doctor) => doctor.did === parseInt(adid))?.slot2}>

                                    {doctorlist.find((doctor) => doctor.did === parseInt(adid))?.slot2}

                                </option>

                            </select>

                        </label>

                        <br />

                        <button type="submit">Book</button>

                        {appointmentflag && <p>{appointmentmsg}</p>}

                    </form>

So how to show timings according to the doctors

<label>

                            Choose Doctor:

                            <select value={adid} onChange={handleAdidChange} required>

                                <option value="">Select Doc</option>

                                {doctorlist.map((doctor, index) => (

                                    <option key={index} value={doctor.did}>

                                        {doctor.dname}, {doctor.specialization}

                                    </option>

                                ))}

                            </select>

                        </label>

useEffect(() => {

        const fetchDoctors = async () => {

            try {

                const response = await axios.get('http://localhost:8082/getDoctorList');

                setDoctorlist(response.data);

            } catch (error) {

                console.log('Error in fetching doctors: ', error);

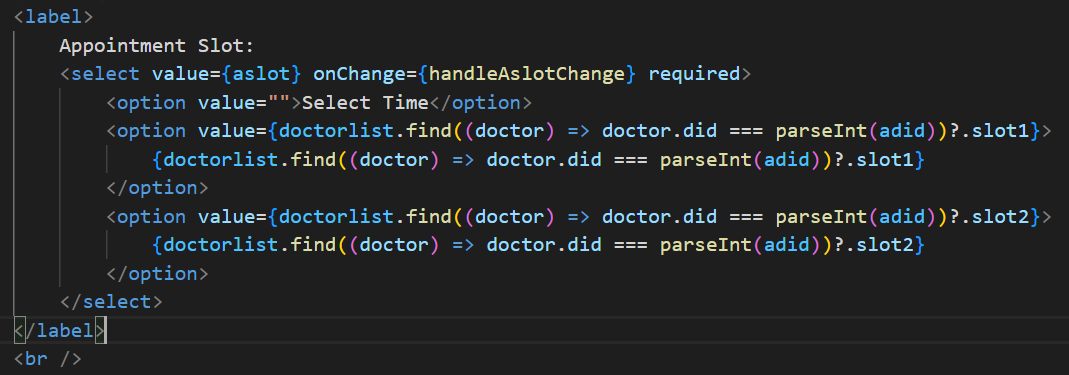
            }

        };

        fetchDoctors();

    }, []);

This is how to render doctors. Ofcourse there are other approaches also which Is more optimized for rendering, here we used axios inside the useEffect and there is an empty dependency array which means only once that is during the component mounting we can see the useEffect running.



Now this will filter according to the doctor we choose above.

NOTE: again all the endpoints are protected.

# DOCTOR HAME PAGE

Is just like the USER HOME PAGE. Same , only words will change.

# Doctor Prev Page.

Here it provides doctor with tools like, sorting and filtering

const sortAppointments = (order) => {

    console.log('clicked sort');

    const sortedAppointments = [...filteredAppointments].sort((a, b) => {

      const dateA = new Date(a.adate);

      const dateB = new Date(b.adate);

      if (dateA.getTime() === dateB.getTime()) {

        // Secondary sort by slot if dates are equal

        return a.aslot.localeCompare(b.aslot) \* (order === 'asc' ? 1 : -1);

      }

      return (dateA - dateB) \* (order === 'asc' ? 1 : -1);

    });

    setFilteredAppointments(sortedAppointments);

  };

So it first we create a variable and copy the useState array to it using [...asdasda]

Now we sort that

It done using annoymous functions .sort((a,b)=>{....}

Here we are sorting with respect to date, if date are same, then we sort wrt to slot

So just like compareable in java used for sorting objects we use this annoymous function to sort objects.

      return (dateA - dateB) \* (order === 'asc' ? 1 : -1);

for descending order we just multiply with -1, which will reduce the value, and now we do the ascending order for the array, it becomes the descending order.

# Handle Filtering

const handleFilterChange = (e) => {

    console.log('clicked filter');

    setFilterDate(e.target.value);

    if (e.target.value === '') {

      // Show all appointments if no date is selected

      setFilteredAppointments(previousAppointments);

    } else {

      // Filter appointments by the selected date

      const filtered = previousAppointments.filter(

        (appointment) => appointment.adate === e.target.value

      );

      setFilteredAppointments(filtered);

    }

  };



BACKEND

UserServer

# Configure Pom.xml and application.properties

Before we start building the backend we need some dependency

We can get that from mvn repository

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.mysql</groupId>  
 <artifactId>mysql-connector-j</artifactId>  
 <scope>runtime</scope>  
 </dependency>  
  
  
 <!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->  
 <dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-api</artifactId>  
 <version>0.12.5</version>  
 </dependency>  
  
 <!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl -->  
 <dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-impl</artifactId>  
 <version>0.12.5</version>  
 <scope>runtime</scope>  
 </dependency>  
  
 <!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson -->  
 <dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-jackson</artifactId>  
 <version>0.12.5</version>  
 <scope>runtime</scope>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-mail</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

After adding everything , you need do maven reload. Once you are done with it, do a dry run. Just go to the main class type hello world and run it to check everything is working fine. If everything is working fine then proceed to do your project.

Now, go to application in resources folder. You can either do in yml or properties type. Your wish

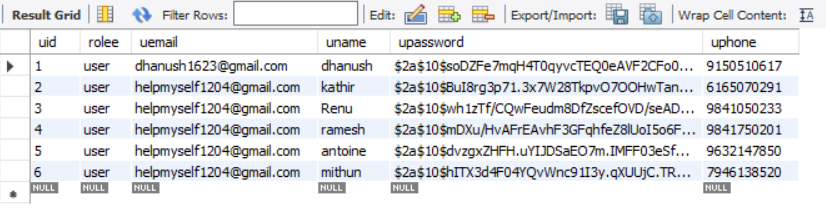
spring.application.name=UsersServer  
  
spring.datasource.url=jdbc:mysql://localhost:3306/bookmydoctorv3db  
spring.datasource.username=root  
spring.datasource.password=rrdd  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
  
spring.jpa.show-sql=true  
spring.jpa.properties.hibernate.format\_sql=true  
spring.jpa.database=mysql  
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect  
#spring.jpa.hibernate.ddl-auto=update  
  
  
  
spring.mail.host=smtp.gmail.com  
spring.mail.port=587  
spring.mail.username=dhanush0616@gmail.com  
spring.mail.password=hanz mzyu vqmc xptr  
spring.mail.properties.mail.smtp.auth=true  
spring.mail.properties.mail.smtp.starttls.enable=true  
  
spring.main.allow-bean-definition-overriding=true

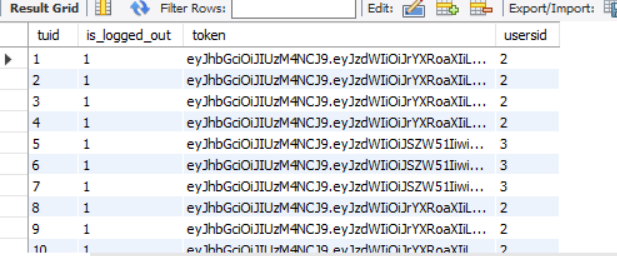
once the table are created in the first run then comment the ddl-auto = update.

# How to create Tables

Now how to create tables, we are using mysql, so mysql-connector-j is needed. We have it in our pom.xml

We are going to create 2 tables for user, one is Users and the another is Utoken for users security.





package com.example.UsersServer.model;  
  
  
import jakarta.persistence.\*;  
import org.springframework.security.core.GrantedAuthority;  
import org.springframework.security.core.authority.SimpleGrantedAuthority;  
import org.springframework.security.core.userdetails.UserDetails;  
  
import java.util.Collection;  
import java.util.List;  
  
@Entity  
@Table(name = "Users")  
public class Users implements UserDetails {  
 public Users(){}  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name = "uid")  
 private Integer uid;  
 @Column(name = "uname")  
 private String uname;  
 @Column(name = "upassword")  
 private String upassword;  
 @Column(name = "uemail")  
 private String uemail;  
 @Column(name = "uphone")  
 private String uphone;  
  
 @Column(name="rolee")  
 @Enumerated(EnumType.*STRING*)  
 private Role rolee;  
  
 @OneToMany(mappedBy = "user")  
 private List<Utoken> tokens;

// getters and setters for the fields  
  
 // implementing UserDetails  
 public String getUsername(){return uname;}  
 public String getPassword(){return upassword;}  
  
  
 @Override  
 public boolean isAccountNonExpired() {  
 return true;  
 }  
  
 @Override  
 public boolean isAccountNonLocked() {  
 return true;  
 }  
  
 @Override  
 public boolean isCredentialsNonExpired() {  
 return true;  
 }  
  
 @Override  
 public boolean isEnabled() {  
 return true;  
 }  
  
 @Override  
 public Collection<? extends GrantedAuthority> getAuthorities() {  
 return List.*of*(new SimpleGrantedAuthority(rolee.name()));  
 }  
}

We need to import jakarta.persistance.\* in order to use the JPA annotations in the code.

@Entity tells us that, it’s a JPA entity

@Table(name=”Users”) is basically telling we want a table created with the name as Users

This class implements UserDetails, it because we when we are using for authentication and of user. The spring security accepts as UserDeatils and not as Users class because of abstraction.

Sometime we need to explicitly mention the default constructor also

@Id  
@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
@Column(name = "uid")  
private Integer uid;

Is used for creating primary Key that value is auto incremented

We can also set the column name as we like using @Column(name=”uid”)

@Column(name="rolee")  
@Enumerated(EnumType.*STRING*)  
private Role rolee;

Role is enumeration there is only 2 roles Users, Doctors, we hardly use this in our porject but this can widly be used for implementing RBAC. Anyway so that why the @Enumerated is used beacause to tell the database (mysql) that this not ordinary datatype like String.

package com.example.UsersServer.model;  
  
public enum Role {  
 *user*,  
 *doctor*}

All this Users, Role ,Utoken etc should be stored in a separate package called model. Its not necessary but it’s a good practice to do it.

@OneToMany(mappedBy = "user")  
private List<Utoken> tokens;

Is basically creating a foreign key. A account can have n number of tokens. So that’s why oneToMany

// implementing UserDetails  
public String getUsername(){return uname;}  
public String getPassword(){return upassword;}  
@Override  
public boolean isAccountNonExpired() {  
 return true;  
}  
@Override  
public boolean isAccountNonLocked() {  
 return true;  
}  
@Override  
public boolean isCredentialsNonExpired() {  
 return true;  
}  
@Override  
public boolean isEnabled() {  
 return true;  
}  
@Override  
public Collection<? extends GrantedAuthority> getAuthorities() {  
 return List.*of*(new SimpleGrantedAuthority(rolee.name()));  
}

All these methods are for Spring Security, no need to touch it mostly. Since the User class/ Entity is UserDetails type(implements) so it required to implement all those methods.

Now Utoken

package com.example.UsersServer.model;  
  
import jakarta.persistence.\*;  
  
@Entity  
@Table(name = "Utoken")  
public class Utoken {  
  
 public Utoken(){}  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name = "tuid")  
 private Integer tuid;  
 @Column(name = "is\_logged\_out")  
 private boolean logged\_out;  
 @ManyToOne  
 @JoinColumn(name="usersid")  
 private Users user;  
 @Column(name = "token")  
 private String token;  
  
 // getters and setters  
  
}

since key from the user is OnetoMany, so in Utoken its ManytoOne

in order to establish a foreign key fully, must be followed

@ManyToOne  
@JoinColumn(name="usersid")  
private Users user;

Here @JoinColumn acts as @Column, provide name to the field and aswell as joins Colum to the other foreign key

@OneToMany(mappedBy = "user")  
private List<Utoken> tokens;

We will be using this Utoken whenever we deal with security.

and we want to create Appointments table, so we also do it like this

@Entity  
@Table(name="Appointments")  
public class Appointments {  
  
  
 public Appointments(){}  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name="aid")  
 private Integer aid;  
  
 @Column(name="auid")  
 private Integer auid;  
  
 @Column(name="auname")  
 private String auname;  
  
 @Column(name="pname")  
 private String pname;  
  
 @Column(name="page")  
 private Integer page;  
  
 @Column(name="pphone")  
 private String pphone;  
  
 @Column(name="adid")  
 private Integer adid;  
  
 @Column(name="adname")  
 private String adname;  
  
 @Column(name="adate")  
 private String adate;  
  
 @Column(name="aslot")  
 private String aslot;  
  
 @Column(name="astatus")  
 private boolean astatus;  
  
 @Column(name="drating")  
 private Integer drating;  
  
 // getters and setters

}

Similarly for Schedules

@Entity  
@Table(name = "Schedules")  
public class Schedules {  
 public Schedules(){}  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name="sid")  
 private Integer sid;  
 @Column(name="sdid")  
 private Integer sdid;  
 @Column(name="sdname")  
 private String sdname;  
 @Column(name="date")  
 private String date;  
 @Column(name="slot")  
 private String slot;  
 @Column(name="numofpatients")  
 private int numofpatients;  
  
 // getters and setters

Its better to create separate classes for request and response.

Like don’t touch modal classes basically the one with @entity. It only used for database quering and internal processing in service layer. Create a separate classes for request and response from and for the frontend respectively.

# Creating Controllers

DEMOTESTLINKS controller

package com.example.UsersServer.controller;  
  
import org.springframework.http.ResponseEntity;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class DemoTestLinksController {  
 @GetMapping("/demo")  
 public ResponseEntity<String> demo(){  
 return ResponseEntity.*ok*("Hello from demo url");  
 }  
  
 @GetMapping("/doctoronly")  
 public ResponseEntity<String> doctoronly(){  
 return ResponseEntity.*ok*("Hello from Doctoronly url");  
 }  
  
 @GetMapping("/useronly")  
 public ResponseEntity<String> useronly(){  
 return ResponseEntity.*ok*("Hello from useronly url");  
 }  
}

if you want to make class as rest api, you need to do @RestController

Now why ResponseEntity<T> over String ?

Ans)

String generally returns the string, while

ResponseEntity represents the whole HTTP response: status code, headers, and body. As a result, we can use it to fully configure the HTTP response.If we want to use it, we have to return it from the endpoint; Spring takes care of the rest.

ResponseEntity is a generic type. Consequently, we can use any type as the response body:

Now, UserPageController

package com.example.UsersServer.controller;  
  
import com.example.UsersServer.model.Appointments;  
import com.example.UsersServer.model.Users;  
import com.example.UsersServer.service.AppointmentsService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
public class UserpageController {  
  
 @Autowired  
 private AppointmentsService appointmentsService;  
  
 /\*  
 \* Sends the appointment request to the appointmentservice;  
 \* \*/  
 @PostMapping("/Userpage/BookAppointment")  
 public boolean bookAppointment(@RequestBody Appointments apreq){  
 return appointmentsService.bookAppointment(apreq);  
 }  
  
 /\*  
 \* Will fetch all the appointments of the user , but should provide uid  
 \* \*/  
 @PostMapping("/Userpage/ShowCurrentUserAppointments")  
 public List<Appointments> showCurrentUserAppointments(@RequestBody Users user){  
 return appointmentsService.showCurrentUserAppointments(user);  
 }  
 @PostMapping("/Userpage/ShowPreviousUserAppointments")  
 public List<Appointments> showPreviousUserAppointments(@RequestBody Users user){  
 return appointmentsService.showPreviousUserAppointments(user);  
 }  
  
 // for cancelling the appointment  
 @PostMapping("/Userpage/CancelAppointment")  
 public void cancelAppointment(@RequestBody Appointments apreq){  
 appointmentsService.cancelAppointment(apreq);  
 }  
  
}

this is how we can create controllers, but still wait there is still create account, login and logout endpoint/controllers left to review, we will review once we start spring security. Now lets move to our Repository.

# Creating Repositories

Spring Data JPA or JPA stands for Java Persistence API, so before looking into that, we must know about ORM (Object Relation Mapping). So Object relation mapping is simply the process of persisting any java object directly into a database table

Usually, the name of the object being persisted becomes the name of the table, and each field within that object becomes a column. With the table setup, each row corresponds to a record in the application

Hibernate is one example of ORM. In short, JPA is the interface while hibernate is the implementation.

Straight to the point: JPA is just guidelines to implement ORM and there is no underlying code for the implementation. Pure Abstraction, Abstraction to the fullest

Spring Data JPA is not a JPA provider, it is a library/framework that adds an extra layer of abstraction on the top of our JPA provider line Hibernate.

package com.example.UsersServer.repository;  
  
  
import com.example.UsersServer.model.Users;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.data.jpa.repository.Query;  
import org.springframework.stereotype.Repository;  
  
import java.util.Optional;  
  
@Repository  
public interface UsersRepository extends JpaRepository<Users,Integer> {  
  
 // for security  
 Optional<Users> findByUname(String uname);  
  
 @Query("SELECT u.uemail FROM Users u WHERE u.uid = :uid")  
 String findUemailByUid(Integer uid);  
  
}

we may get the user with name = uname, or may we not get also, that’s why we use Optional of the type Users

By using @Query( you can manaully create queries and map it to the function declaration.)

Declaration: return type , name of the function, parameters

Definition: the body of the function.

package com.example.UsersServer.repository;  
  
import com.example.UsersServer.model.Utoken;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.data.jpa.repository.Query;  
  
import java.util.List;  
import java.util.Optional;  
  
public interface TokenUsersRepository extends JpaRepository<Utoken,Integer> {  
  
 @Query("""  
 select t from Utoken t inner join Users u on t.user.uid = u.uid  
 where t.user.uid = :uid and t.logged\_out = false   
 """)  
 List<Utoken> findAllTokensByUser(Integer uid);  
  
 Optional<Utoken> findByToken(String token);  
  
}

package com.example.UsersServer.repository;  
  
import com.example.UsersServer.model.Appointments;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.data.jpa.repository.Query;  
import org.springframework.stereotype.Repository;  
  
import java.util.List;  
  
@Repository  
public interface AppointmentsRepository extends JpaRepository<Appointments,Integer> {  
  
 List<Appointments> findByAuid(Integer auid);  
 void deleteByAid(Integer aid);  
  
 @Query("SELECT a FROM Appointments a WHERE a.auid = :auid AND a.astatus = true")  
 List<Appointments> findByAuidWhereAstatusIsTrue( Integer auid);  
  
 @Query("SELECT a FROM Appointments a WHERE a.auid = :auid AND a.astatus = false")  
 List<Appointments> findByAuidWhereAstatusIsFalse(Integer auid);  
  
  
}

SCHEDULE respository

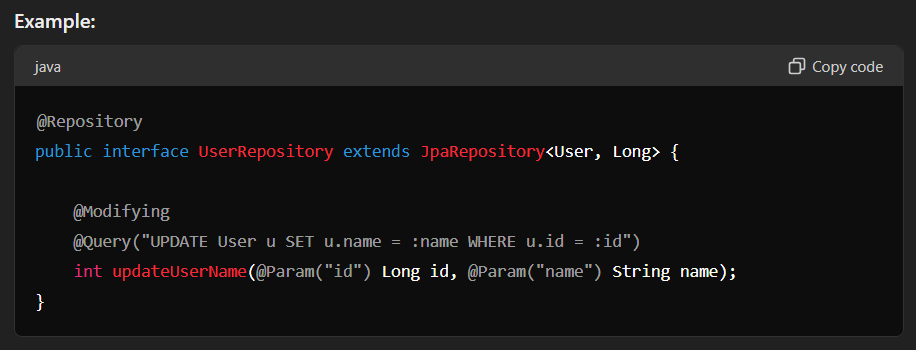
package com.example.UsersServer.repository;  
  
import com.example.UsersServer.model.Schedules;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.data.jpa.repository.Modifying;  
import org.springframework.data.jpa.repository.Query;  
import org.springframework.stereotype.Repository;  
import org.springframework.transaction.annotation.Transactional;  
  
@Repository  
public interface SchedulesRepository extends JpaRepository<Schedules,Integer> {  
  
 @Query("SELECT s.numofpatients FROM Schedules s WHERE s.sdname = :dname AND s.date = :date AND s.slot = :slot")  
 Integer findNumofPatientsByDnameDateSlot( String dname, String date, String slot);  
  
 @Modifying  
 @Transactional  
 @Query("UPDATE Schedules s SET s.numofpatients = s.numofpatients + 1 WHERE s.sdname = :dname AND s.date = :date AND s.slot = :slot")  
 void incrementNumofPatientsByOne( String dname, String date, String slot);  
  
 @Modifying  
 @Transactional  
 @Query("UPDATE Schedules s SET s.numofpatients = s.numofpatients - 1 WHERE s.sdname = :dname AND s.date = :date AND s.slot = :slot")  
 void decrementNumofPatientsByOne( String dname, String date, String slot);  
  
 @Query("SELECT COUNT(s) > 0 FROM Schedules s WHERE s.sdname = :dname AND s.date = :date AND s.slot = :slot")  
 boolean isSchedulePresent( String dname, String date, String slot);  
  
}

Now what is @Modifying and @Transactional ?

@Modiying

The @Modifying annotation is used in Spring Data JPA repository methods to indicate that the method is modifying the data in the database (such as INSERT, UPDATE, DELETE) and not just fetching data (SELECT). This annotation is necessary when you are executing a JPQL (Java Persistence Query Language) or native query that changes the state of the database.

We have used the jpql that is the sql inside @query is called the jpql



@Transactional

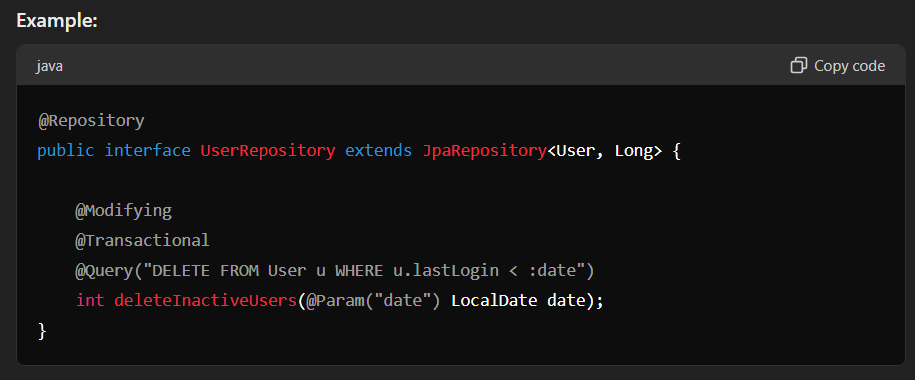
The @Transactional annotation is used to define the scope of a single database transaction. It can be applied at the method or class level and ensures that the method/class runs within a transaction context. If any unchecked exception is thrown, the transaction is rolled back.



In this example, the @Transactional annotation ensures that the updateUser method runs within a transactional context. If any exception occurs during the execution of updateUser, the transaction will be rolled back, and the changes will not be committed to the database.

Mostly @Modifying and Transactional are used together

Often, these annotations are used together to ensure that the modifying query runs within a transaction context, ensuring consistency and atomicity of the database operations.



In this example, @Modifying indicates that the query deletes data, and @Transactional ensures that the deletion happens within a transaction. If the method fails, the transaction will roll back, ensuring the database remains consistent.

Summary

@Modifying: Indicates that a query modifies the database (e.g., INSERT, UPDATE, DELETE).

@Transactional: Ensures that a method/class runs within a transaction context, providing consistency and rollback capabilities in case of exceptions.

# Spring Security and JWT implementation

You want to have these dependencies in your project

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>

<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-api</artifactId>  
 <version>0.12.5</version>  
</dependency>  
  
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl -->  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-impl</artifactId>  
 <version>0.12.5</version>  
 <scope>runtime</scope>  
</dependency>  
  
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson -->  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-jackson</artifactId>  
 <version>0.12.5</version>  
 <scope>runtime</scope>  
</dependency>

We start from Config

We start from CorsConfig

package com.example.UsersServer.config;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.servlet.config.annotation.CorsRegistry;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  
  
@Configuration  
public class CorsConfig implements WebMvcConfigurer {  
  
 @Override  
 public void addCorsMappings(CorsRegistry registry) {  
 registry.addMapping("/\*\*")  
 .allowedOrigins("http://localhost:3000")  
 .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")  
 .allowedHeaders("\*")  
 .allowCredentials(true)  
 .maxAge(3600);  
 }  
}

we want to understand what is @configuration

One of the most important annotations in spring is @Configuration annotation which indicates that the class has @Bean definition methods. So Spring container can process the class and generate Spring Beans to be used in the application.

First we learn with an example

// Java Program to Illustrate College Class

**package** ComponentAnnotation;

// Class

**public** **class** College {

    // Method

**public** **void** test()

    {

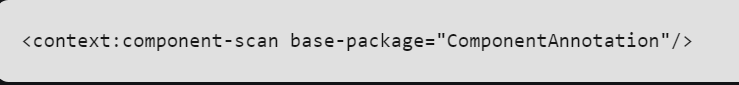
        // Print statement

        System.out.println("Test College Method");

    }

}

To create object we can use @Component, but we need to write this line inside beans.xml



We want to stop/reduace the use of beans.xml and focus in business login, so for that we can create a configuration class we can make it into configuration by specifying @Confifguration on thop of the class. This will make it a configuration class.

So now we don’t need to use Component and ComponetScan to create the bean.

// Java Program to Illustrate Configuration Class

**package** BeanAnnotation;

// Importing required classes

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

@Configuration

**public** **class** CollegeConfig {

      // Using Bean annotation to create

    // College class Bean

    @Bean

    // Here the method name is the

    // bean id/bean name

**public** College collegeBean() {

        // Return the College object

**return** **new** College();

    }

}

Similarly, if you have another class named Student and you want to create the bean for this Student class then you can create the bean using the @Bean annotation inside the configuration class just like that

// Java Program to Illustrate Application Class

**package** ComponentAnnotation;

// Importing required classes

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

// Application class

**public** **class** Main {

    // Main driver method

**public** **static** **void** main(String[] args)

    {

        // Use AnnotationConfigApplicationContext

        // instead of ClassPathXmlApplicationContext

        // because we are not using XML Configuration

        ApplicationContext context

            = **new** AnnotationConfigApplicationContext(

                CollegeConfig.**class**);

        // Getting the bean

        College college

            = context.getBean("collegeBean", College.**class**);

        // Invoking the method

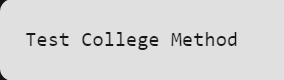
        // inside main() method

        college.test();

    }

}

OP:



This configuration is present in context.annotation package

Whats CORS?

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

Coming back to CorsConfig

@Configuration  
public class CorsConfig implements WebMvcConfigurer {  
  
 @Override  
 public void addCorsMappings(CorsRegistry registry) {  
 registry.addMapping("/\*\*")  
 .allowedOrigins("http://localhost:3000")  
 .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")  
 .allowedHeaders("\*")  
 .allowCredentials(true)  
 .maxAge(3600);  
 }  
}

public class CorsConfig implements WebMvcConfigurer: This class implements the WebMvcConfigurer interface, which provides callback methods to customize the Java-based configuration for Spring MVC. By implementing this interface, you can customize how Spring MVC is configured in your application.

public void addCorsMappings(CorsRegistry registry): This method is used to add CORS mappings to the application. The CorsRegistry parameter allows you to configure the CORS settings.

registry.addMapping("/\*\*"): This line configures CORS for all endpoints in the application. The /\*\* pattern matches all paths.

.allowedOrigins("http://localhost:3000"): This specifies that only requests from http://localhost:3000 are allowed. This is typically used during development when the front-end is served from a different origin (e.g., a React.js application running on localhost:3000).

.allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS"): This allows specific HTTP methods. In this case, GET, POST, PUT, DELETE, and OPTIONS methods are permitted.

.allowedHeaders("\*"): This allows all headers in the request. It means the client can send any headers in its requests.

.allowCredentials(true): This allows cookies and other credentials to be included in the requests. This is often necessary for sessions and authentication.

.maxAge(3600): This sets the maximum age (in seconds) for which the CORS configuration can be cached by clients. In this case, it is set to 3600 seconds (1 hour).

UserCustomLogoutHandler

package com.example.UsersServer.config;  
  
import com.example.UsersServer.model.Utoken;  
import com.example.UsersServer.repository.TokenUsersRepository;  
import jakarta.servlet.http.HttpServletRequest;  
import jakarta.servlet.http.HttpServletResponse;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.web.authentication.logout.LogoutHandler;  
  
@Configuration  
public class UserCustomLogoutHandler implements LogoutHandler {  
  
 @Autowired  
 private TokenUsersRepository tokenRepository;  
  
 @Override  
 public void logout(HttpServletRequest request,  
 HttpServletResponse response,  
 Authentication authentication) {  
 // get the Authorization header , you can check it in axios methods in frontend  
 /\*  
 \* Since we are logging out, lets say that we are giving back our token  
 \* \*/  
 String authHeader = request.getHeader("Authorization");  
  
 /\*  
 \* its checking for " Bearer ", if not present then just return, dont logout  
 \* \*/  
 if(authHeader == null || !authHeader.startsWith("Bearer ")) {  
 return;  
 }  
  
 /\*  
 \* extract the token from the "Bearer "  
 \* find the token in the database else return null, .orElse handle that null pointer exception  
 \* \*/  
 String token = authHeader.substring(7);  
 Utoken storedToken = tokenRepository.findByToken(token).orElse(null);  
  
 /\*  
 \* If you find the token, set logout for that token to true and save that token in the database.  
 \* \*/  
 if(storedToken != null) {  
 storedToken.setLogged\_out(true);  
 tokenRepository.save(storedToken);  
 }  
 }  
  
}

read the comments

UserServiceImp

package com.example.UsersServer.service;  
  
import com.example.UsersServer.repository.UsersRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.security.core.userdetails.UsernameNotFoundException;  
import org.springframework.stereotype.Service;  
  
@Service  
public class UserServiceImp implements UserDetailsService {  
 @Autowired  
 private UsersRepository usersRespository;  
  
 /\*   
 \* Remember we implemented UserDetails for the Users class(Entity in the model package)  
 \* The Spring security has many methods, but there is no methods that accepts( parameters) which is of the type Users  
 \* For each application the entity will change, so the spring security choose abstraction to be it parameters  
 \* here abstraction is the UserDetails, the spring security has methods that deals with UserDetails and not the entity itself.  
 \* So we need to get data as a form of UserDetails and not as a form of Users( The Entity) itself  
 \* so that's why we have implemented Users implements UserDetails  
 \* \*/  
 public UserDetails loadUserByUsername(String uname) throws UsernameNotFoundException {  
 return usersRespository.findByUname(uname)  
 .orElseThrow(()-> new UsernameNotFoundException("User not found"));  
 }  
}

Remember we implemented UserDetails for the Users class(Entity in the model package)

The Spring security has many methods, but there is no methods that accepts( parameters) which is of the type Users

For each application the entity will change, so the spring security choose abstraction to be it parameters

Here abstraction is the UserDetails, the spring security has methods that deals with UserDetails and not the entity itself.

So we need to get data as a form of UserDetails and not as a form of Users( The Entity) itself so that's why we have implemented Users implements UserDetails

JwtService

We need to use the 3 jjwt – api, impl, jackson in order to create the jwt token

In this file pure cryptography is happening

package com.example.UsersServer.service;  
  
import com.example.UsersServer.model.Users;  
import com.example.UsersServer.repository.TokenUsersRepository;  
import io.jsonwebtoken.Claims;  
import io.jsonwebtoken.Jwts;  
import io.jsonwebtoken.io.Decoders;  
import io.jsonwebtoken.security.Keys;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.stereotype.Service;  
  
import javax.crypto.SecretKey;  
import java.util.Date;  
import java.util.function.Function;  
  
@Service  
public class JwtService {  
  
 // we need a 256 bit length secret key  
 private final String SECRET\_KEY = "4bb6d1dfbafb64a681139d1586b6f1160d18159afd57c8c79136d7490630407c";  
  
 @Autowired  
 private TokenUsersRepository tokenUsersRepository;  
  
  
 /\*  
 \* we getting the secert key and doing some decoding operation and turing the secretkey into byte array,  
 \* now whill be suitable for signing HMAC  
 \* there the HMAC used is HMAC SHA 256 why is you have mentioned that Keys.hmacShaKeyFor(..)  
 \* the Keys is from the jsonwebtoken package and it provides methods wihic sign token using SHA hashing alogorithm.  
 \* why 256 , is beacuse the secert key length is 256 bit, so that why it automatically sign with SHA256 algo.  
 \* \*/  
 private SecretKey getSigninKey() {  
 byte[] keyBytes = Decoders.*BASE64URL*.decode(SECRET\_KEY);  
 return Keys.*hmacShaKeyFor*(keyBytes);  
 }  
  
 // bunch of validating methods  
 public String extractUsername(String token) {  
 return extractClaim(token, Claims::getSubject);  
 }  
 private boolean isTokenExpired(String token) {  
 return extractExpiration(token).before(new Date());  
 }  
 private Date extractExpiration(String token) {  
 return extractClaim(token, Claims::getExpiration);  
 }  
 public <T> T extractClaim(String token, Function<Claims, T> resolver) {  
 Claims claims = extractAllClaims(token);  
 return resolver.apply(claims);  
 }  
 private Claims extractAllClaims(String token) {  
 return Jwts  
 .*parser*()  
 .verifyWith(getSigninKey())  
 .build()  
 .parseSignedClaims(token)  
 .getPayload();  
 }  
   
 public boolean isValidUser(String token, UserDetails user) {  
 String username = extractUsername(token);  
  
 boolean validToken = tokenUsersRepository  
 .findByToken(token)  
 .map(t -> !t.isLogged\_out())  
 .orElse(false);  
  
 return (username.equals(user.getUsername())) && !isTokenExpired(token) && validToken;  
 }  
  
 public String generateTokenforUser(Users user) {  
  
 String token = Jwts  
 .*builder*()  
 .subject(user.getUsername()) // add claim  
 .issuedAt(new Date(System.*currentTimeMillis*()))  
 .expiration(new Date(System.*currentTimeMillis*() + 120\*60\*1000 ))  
 .signWith(getSigninKey())  
 .compact();  
  
 return token;  
 }  
}

Authentication Service

package com.example.UsersServer.service;  
  
import com.example.UsersServer.model.AuthenticationResponse;  
import com.example.UsersServer.model.Users;  
import com.example.UsersServer.model.Utoken;  
import com.example.UsersServer.repository.TokenUsersRepository;  
import com.example.UsersServer.repository.UsersRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
  
@Service  
public class AuthenticationService {  
  
  
 @Autowired  
 private UsersRepository usersRepository;  
  
 @Autowired  
 private TokenUsersRepository tokenUsersRepository;  
  
 @Autowired  
 private JwtService jwtService;  
 @Autowired  
 private AuthenticationManager authenticationManager;  
 @Autowired  
 private PasswordEncoder passwordEncoder;  
  
 @Autowired  
 private EmailService emS;  
  
  
 /\*   
 \* passwordEncoder does BCrypt for us, check UserSecurityConfig  
 \* \*/  
 public boolean createaccount(Users register){  
 if(usersRepository.findByUname(register.getUname()).isPresent()){  
 System.*out*.println("User Already Exist");  
 return false;  
 }  
 register.setUpassword(passwordEncoder.encode(register.getUpassword()));  
 usersRepository.save(register);  
 System.*out*.println("User created Successfully");  
 emS.CreateAccountConfirmation(register);  
 return true;  
  
 }  
  
 /\*   
 \* Find the username and passsword from DB, this id done by the authencation manager.  
 \* authenticationManager it authenticates the user, like How? again abstraction.  
 \* you are creating an object of UsernamePasswordAuthenticationToken(setting it username and password)  
 \*   
 \* if username and password are fine, correct, then the token will be created for the given username  
 \*   
 \* now we revoke all the existing valid/timeout tokens. by this way we made sure that we solve the parallel/concurrent logins.  
 \* and we save the newly created token   
 \* \*/  
 public AuthenticationResponse authenticateuser(Users loginrequest){  
 // dont know whiy i used it here  
 authenticationManager.authenticate(  
 new UsernamePasswordAuthenticationToken(  
 loginrequest.getUsername(),loginrequest.getPassword()  
 )  
 );  
 // we dont know the users role, so we go fetch all the data from the given user and   
 // we pass it to the jwtService it will use the user role from it  
 Users user = usersRepository.findByUname(loginrequest.getUsername()).orElseThrow(); // used for fetching the user role  
 String jwt = jwtService.generateTokenforUser(user);  
 revokeAllTokenByUser(user);  
 saveUserToken(jwt,user);  
 return new AuthenticationResponse(jwt,true);  
 }  
  
 private void revokeAllTokenByUser(Users user){  
 List<Utoken> validTokens = tokenUsersRepository.findAllTokensByUser(user.getUid());  
 if(validTokens.isEmpty()){return ;}  
 validTokens.forEach(t->{  
 t.setLogged\_out(true);  
 });  
 tokenUsersRepository.saveAll(validTokens);  
 }  
  
 private void saveUserToken(String jwt,Users user){  
 Utoken token = new Utoken();  
 token.setToken(jwt);  
 token.setLogged\_out(false);  
 token.setUser(user);  
 tokenUsersRepository.save(token);  
 }  
  
  
 public void logoutuser(String token){  
 Utoken storedToken = tokenUsersRepository.findByToken(token).orElse(null);  
 if(storedToken != null){  
 storedToken.setLogged\_out(true);  
 tokenUsersRepository.save(storedToken);  
 }  
 }  
  
}

if you want to allow concurrent users, then we can remove the revokeAllToken in the authenticateuser method.

JwtFilter

package com.example.UsersServer.filter;  
  
import com.example.UsersServer.service.JwtService;  
import com.example.UsersServer.service.UserServiceImp;  
import jakarta.servlet.FilterChain;  
import jakarta.servlet.ServletException;  
import jakarta.servlet.http.HttpServletRequest;  
import jakarta.servlet.http.HttpServletResponse;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.lang.NonNull;  
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  
import org.springframework.security.core.context.SecurityContextHolder;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;  
import org.springframework.stereotype.Component;  
import org.springframework.web.filter.OncePerRequestFilter;  
  
import java.io.IOException;  
  
@Component  
public class JwtUserAuthenticationFilter extends OncePerRequestFilter {  
 @Autowired  
 private JwtService jwtService;  
  
 @Autowired  
 private UserServiceImp userServiceImp;  
  
 /\* this is a method of OncePerRequestFilter, which is filter  
 \* we check for "Bearer " token if it doesnt exist, then just return , basically telling the authorization is not compatiable  
 \* else we extract the token from the request header  
 \* we check whether the token is valid , then allow the request  
 \*   
 \* Example is Airport,  
 \* they check your flight ticket first,  
 \* lets say you carry a pretty big luggage, so you cant take it with you since it need to to the baggage part of the aeroplane  
 \* so the process is   
 \*   
 \* 1) first in the counter( ticket counter) the officer enters the details of the person and luggage   
 \* 2)then they create a barcode to distinguish the luggage  
 \* 3) finally they put that in converier belt and send it to the ground men  
 \* 4) they will put our luggage into the plane.  
 \*   
 \* bascially line 63 to 72 is analogous to this above mentioned example and process  
 \*   
 \* \*/  
 @Override  
 protected void doFilterInternal(  
 @NonNull HttpServletRequest request,  
 @NonNull HttpServletResponse response,  
 @NonNull FilterChain filterChain) throws ServletException, IOException {  
  
 String authHeader = request.getHeader("Authorization");  
 if(authHeader == null || !authHeader.startsWith("Bearer ")) {  
 filterChain.doFilter(request,response);  
 return;  
 }  
 String token = authHeader.substring(7);  
 String username = jwtService.extractUsername(token);  
  
 if(username != null && SecurityContextHolder.*getContext*().getAuthentication() ==null){  
 UserDetails userDetails = userServiceImp.loadUserByUsername(username);  
 if(jwtService.isValidUser(token,userDetails)){  
 UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(  
 userDetails,null,userDetails.getAuthorities()  
 );  
  
 authToken.setDetails(  
 new WebAuthenticationDetailsSource().buildDetails(request)  
 );  
 SecurityContextHolder.*getContext*().setAuthentication(authToken);  
 }  
 }  
 filterChain.doFilter(request,response);  
  
 }  
}

/\* this is a method of OncePerRequestFilter, which is filter

\* we check for "Bearer " token if it doesnt exist, then just return , basically telling the authorization is not compatiable

\* else we extract the token from the request header

\* we check whether the token is valid , then allow the request

\*

\* Example is Airport,

\* they check your flight ticket first,

\* lets say you carry a pretty big luggage, so you cant take it with you since it need to to the baggage part of the aeroplane

\* so the process is

\*

\* 1) first in the counter( ticket counter) the officer enters the details of the person and luggage

\* 2)then they create a barcode to distinguish the luggage

\* 3) finally they put that in converier belt and send it to the ground men

\* 4) they will put our luggage into the plane.

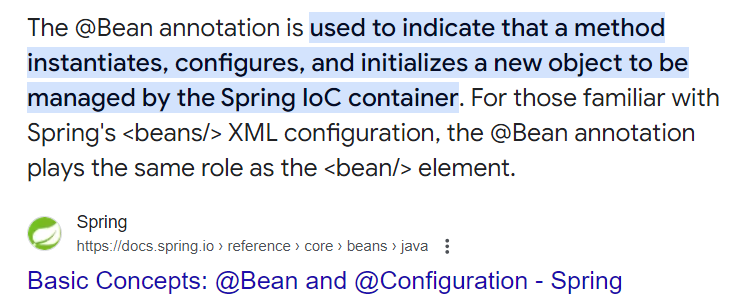
\*

\* bascially line 63 to 72 is analogous to this above mentioned example and process

\*

\* \*/

UserSecurityConfig



package com.example.UsersServer.config;  
  
import com.example.UsersServer.filter.JwtUserAuthenticationFilter;  
import com.example.UsersServer.service.UserServiceImp;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.http.HttpStatus;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;  
import org.springframework.security.config.http.SessionCreationPolicy;  
import org.springframework.security.core.context.SecurityContextHolder;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.security.web.SecurityFilterChain;  
import org.springframework.security.web.authentication.HttpStatusEntryPoint;  
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  
  
@Configuration  
public class UserSecurityConfig {  
  
  
 @Autowired  
 private UserServiceImp userServiceImp;  
  
 @Autowired  
 private JwtUserAuthenticationFilter jwtUserAuthenticationFilter;  
  
 @Autowired  
 private UserCustomLogoutHandler userlogoutHandler;  
  
 @Bean  
 public SecurityFilterChain usersecurityFilterChain(HttpSecurity http) throws Exception{  
  
 return http  
 .csrf(AbstractHttpConfigurer:: disable)  
 .authorizeHttpRequests(  
 req->req.requestMatchers("/Userslogin/\*\*","/Createaccount/\*\*").permitAll()  
 .requestMatchers("/doctoronly/\*\*").hasAnyAuthority("doctor")  
 .requestMatchers("/useronly/\*\*","/Userpage/\*\*").hasAnyAuthority("user")  
 .anyRequest()  
 .authenticated()  
 ).userDetailsService(userServiceImp)  
 .sessionManagement(session->session  
 .sessionCreationPolicy(SessionCreationPolicy.*STATELESS*))  
 .addFilterBefore(jwtUserAuthenticationFilter, UsernamePasswordAuthenticationFilter.class)  
 .exceptionHandling(  
 e->e.accessDeniedHandler(  
 (request,response,accessDeniedException)->response.setStatus(403)  
 ).authenticationEntryPoint(new HttpStatusEntryPoint(HttpStatus.*UNAUTHORIZED*)))  
 .logout(l->l  
 .logoutUrl("/logoutuser")  
 .addLogoutHandler(userlogoutHandler)  
 .logoutSuccessHandler((request, response, authentication) -> SecurityContextHolder.*clearContext*()  
 ))  
 .build();  
  
 }  
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
  
 @Bean  
 public AuthenticationManager authenticationManager(AuthenticationConfiguration configuration) throws Exception {  
 return configuration.getAuthenticationManager();  
 }  
  
  
}

SecurityFilterChain => Chain here means layers of filter

We are concerned about HttpSecurity

Now we are configuring the end points, is done using .requestMatcher()

Now we wanna tell spring that the userServiceImp is the UserServiceDetail that we wanna use.

Tell spring security that we want to have a stateless session.

We tell spring security that we add jwtFilter of the type UsernamePasswordAuthenticationFiler.class

We handle the addFilterBefore() with exception using exceptionHandler

Basically exception is going to be accessDenied, token timeout or invalid( accessDenied got them covered) the client/Browers doent understands the exception by the server so we Set the HttpStatusEntryPoint to be UNAUTHORIZED.(403)

Now, for Logout

You want to tell springsecurity the logouturl

You also want to tell springsecurity the logouthandler for logging out the user

You want to notify the the user has logout from the backend Server so

LogoutSuccessHandler and you use request , response and alsoe you remove(.clearContext()) that user from the backend SecurityContextHolder

SecurityContextHolder is a core component of Spring Security, used to store and access security-related information (such as the authentication details of the currently authenticated user) in a Spring Boot application. It holds the SecurityContext, which contains the Authentication object representing the currently authenticated principal.

Key Points:

SecurityContext:

The SecurityContext holds the Authentication object, which contains details about the authenticated user, including authorities/roles.

It's a thread-local container, meaning it is specific to the current thread of execution.

Authentication:

The Authentication object represents the principal (user) and includes credentials, authorities (roles), and other details.

Typically, this object is created during the authentication process and populated with user-specific information.

Common Methods:

SecurityContextHolder.getContext(): Retrieves the SecurityContext for the current thread.

SecurityContext.getAuthentication(): Retrieves the Authentication object from the SecurityContext.

SecurityContextHolder.setContext(SecurityContext context): Sets the SecurityContext for the current thread.

SecurityContextHolder.clearContext(): Clears the SecurityContext for the current thread.

# Email automation using JavaMailSender

spring.mail.host=smtp.gmail.com  
spring.mail.port=587  
spring.mail.username=dhanush0616@gmail.com  
spring.mail.password=hanz mzyu vqmc xptr  
spring.mail.properties.mail.smtp.auth=true  
spring.mail.properties.mail.smtp.starttls.enable=true

this is device dependent this properties is only valid for this particular device

package com.example.UsersServer.service;  
  
import com.example.UsersServer.model.Appointments;  
import com.example.UsersServer.model.Users;  
import com.example.UsersServer.repository.UsersRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.mail.SimpleMailMessage;  
import org.springframework.mail.javamail.JavaMailSender;  
import org.springframework.stereotype.Service;  
  
@Service  
public class EmailService {  
  
 @Value("${spring.mail.username}")  
 private String sender;  
  
 @Autowired  
 private JavaMailSender javaMailSender;  
  
 @Autowired  
 private UsersRepository usersRepository;  
  
 public void sendBookingConfirmation(Appointments appointments){  
 String usremail = usersRepository.findUemailByUid(appointments.getAuid());  
 try{  
 SimpleMailMessage mailAppointment = new SimpleMailMessage();  
 mailAppointment.setFrom("Dhanush SR<"+sender+">");  
 mailAppointment.setTo(usremail);  
 mailAppointment.setSubject("Your Appointment Confirmation");  
 String str = appointments.getAuname() + " your appointment to visit " + appointments.getAdname()+" on "+appointments.getAdate()+" at "+appointments.getAslot()+" is confirmed. "+" If you have any other queries feel free to contact us. \n"+" Thank You ";  
 mailAppointment.setText(str);  
 javaMailSender.send(mailAppointment);  
 System.*out*.println("Email Sent successfully");  
 }catch(Exception e){  
 System.*out*.println("error in sending email to " + usremail);  
 e.printStackTrace();  
 }  
 }  
  
 public void sendBookingCancellationConfirmation(Appointments appointments){  
 String usremail = usersRepository.findUemailByUid(appointments.getAuid());  
 try{  
 SimpleMailMessage mailAppointment = new SimpleMailMessage();  
 mailAppointment.setFrom("Dhanush SR<"+sender+">");  
 mailAppointment.setTo(usremail);  
 mailAppointment.setSubject("Your Cancellation Request for Appointment =" +appointments.getAid()+" is Done");  
 String msg = "Dear "+appointments.getAuname()+" your appointment with id = "+appointments.getAid()+ "and appointment details : "+appointments.getAdname()+" "+appointments.getAdate() + " "+appointments.getAslot() +" is Successfully Cancelled ";  
 mailAppointment.setText(msg);  
 javaMailSender.send(mailAppointment);  
 System.*out*.println("Email Sent successfully");  
 }catch(Exception e){  
 System.*out*.println("error in sending email to " + usremail);  
 e.printStackTrace();  
 }  
 }  
  
 public void CreateAccountConfirmation(Users register){  
 try{  
 SimpleMailMessage mailAppointment = new SimpleMailMessage();  
 mailAppointment.setFrom("Dhanush SR<"+sender+">");  
 mailAppointment.setTo(register.getUemail());  
 mailAppointment.setSubject("Your Account has been Created");  
 String msg = "Dear "+register.getUname()+" Your Account has been Created Successfullt" ;  
 javaMailSender.send(mailAppointment);  
 }catch(Exception e){  
 System.*out*.println("error in sending email to " + register.getUemail());  
 e.printStackTrace();  
 }  
 }  
  
}

2 things

SimpleMailMessage is used to contruct a simple mail message

Their methods:

Setfrom, setTo, setSubject,setText

javamailSender.send( simplemailmessage) is used to send it

# MainPage Controllers

package com.example.UsersServer.controller;  
  
import com.example.UsersServer.model.AuthenticationResponse;  
import com.example.UsersServer.model.Users;  
import com.example.UsersServer.model.response.LoginResponse;  
import com.example.UsersServer.repository.UsersRepository;  
import com.example.UsersServer.service.AuthenticationService;  
import jakarta.servlet.http.HttpServletRequest;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.security.core.userdetails.UsernameNotFoundException;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestBody;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.Optional;  
  
@RestController  
public class MainpageController {  
 @Autowired  
 private AuthenticationService authenticationService;  
  
 @Autowired  
 private UsersRepository usersRepository;  
  
 /\*  
 \* Endpoint for Login-user : First the user is authenticated  
 \* the response is constructed = {uid,uname,uphone,uemail,token,flag}  
 \* sent to the frontend for using this in various purposes like booking, canceling etc  
 \* \*/  
 @PostMapping("/Userslogin")  
 public LoginResponse loginuser(@RequestBody Users loginreq){  
 AuthenticationResponse ar = authenticationService.authenticateuser(loginreq);  
 System.*out*.println("Login Successful");  
 LoginResponse lgres = new LoginResponse();  
 Optional<Users> usr = usersRepository.findByUname(loginreq.getUname());  
 if(usr.isPresent()){  
 Users user = usr.get();  
 lgres.setUid(user.getUid()); // find uid  
 lgres.setUname(loginreq.getUname());  
 lgres.setUemail(user.getUemail()); // get uemail  
 lgres.setUphone(user.getUphone()); // get uphone  
 lgres.setToken(ar.getToken());  
 lgres.setIslogged(ar.getMessage());  
 }else{  
 throw new UsernameNotFoundException("User not found");  
  
 }  
 return lgres;  
 }  
  
 /\*  
 \* For Creating Account, return a flag value to the frontend  
 \* \*/  
 @PostMapping("/Createaccount")  
 public boolean createuser(@RequestBody Users loginreq){  
 boolean ar = authenticationService.createaccount(loginreq);  
 System.*out*.println("Account Created Successful");  
 return ar ;  
 }  
  
 /\*  
 \* For Logging out, return a flag value to the frontend, so that it redirects to the Mainpage of the website  
 \* \*/  
 @PostMapping("/logoutuser")  
 public String logoutuser(HttpServletRequest request){  
 String authHeader = request.getHeader("Authorization");  
 if (authHeader != null && authHeader.startsWith("Bearer ")) {  
 String token = authHeader.substring(7);  
 authenticationService.logoutuser(token);  
 return " User Logout Success";  
 } else {  
 return "Authorization header missing or invalid";  
 }  
 }  
  
  
}

NOTE: this for UsersServer, but the same thing , I mean the literal same thing is for DoctorServer.

The things that change are

1. securityConfig, the endpoints change , the role just swapped,
2. Table name changes
3. Fewer number of endpoints than Users