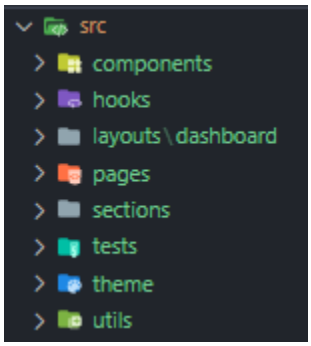
Documentación Prueba Tech Mahindra

En este documento se describe las funcionalidades y partes de código de un aplicativo creado en react para la gestión de usuarios.

El aplicativo está compuesto con las siguientes librerías:

- **Material UI:** Librería de componentes que permite reutilizar componentes con estilos ya creados, esta permite agilizar la creación de la interfaz.
- **Axios:** Para consumo de la API, traemos la información de los usuarios para la validación en el login y para consumir la información del dashboard.
- **Json-server:** Para simular la API con la información de los usuarios registrados y el dashboard.
- **React-apexcharts:** Para la creación del dashboard, se le pasan ciertos parámetros para crear los gráficos.
- **Lodash:** Se usa para simplificar el manejo y edición de objetos, arrays, etc. ya que este proporciona muchos métodos de utilidad para hacerlo.
- **Numeral:** Para formatear valores.
- **React-router-dom:** Para el manejo de las rutas y navegabilidad.

Estructura de carpetas:



En la carpeta ‘components’ tenemos 6 componentes de menor tamaño, estos son utilizados para el logo, el dashboard, los iconos, colores y navegación.

En la carpeta ‘hooks’ tenemos un custom Hook para el responsive de ambas páginas.

En la carpeta ‘layouts’ esta la configuración del dashboard, donde se agregan las opciones de la barra de navegación, los iconos, el header y estilos en general usando ‘styled’ de MUI.

En la carpeta ‘pages’ están las dos paginas que tenemos, la del login y la del dashboard, en el archivo del dashboard llamado ‘DashboardAppPage.js’ llamamos dos archivos para implementar los dos gráficos que tenemos ‘AppCurrentVisits’ y ‘AppWebsiteVisits’ alojados en sections y en el archivo para la pagina del login ‘LoginPage’ llamamos ‘LoginForm’ el cual contiene el componente del formulario con la función de validación.

En la carpeta ‘sections’ tenemos las dos secciones principales, dashboard y login, para la sección del dashboard, enviamos la información que encontramos en la API tanto para el grafico de torta como para el lineal. Para la sección del login, en este caso no enviamos la info ya que consumimos la API desde acá.

En la carpeta ‘tests’ encontramos 3 archivos con las pruebas de los 3 archivos principales. (Aun no terminado)

En la carpeta ‘theme’ están los estilos de todo el aplicativo, el cual llamamos en App.js como ThemeProvider para encerrar todo el aplicativo en este tema.

Por último, en la carpeta ‘utils’ tenemos unos estilos adicionales creados en formato de funciones, estos son llamados en diferentes partes del aplicativo y así pueden ser reutilizados.

Funcionalidades principales:

Empezando con el login, nos encontramos con un formulario básico, donde debemos poner correo y password, esta información viene desde la API. Si el usuario intenta acceder con un usuario que no se encuentra registrado saldrá un mensaje indicando que el ‘Correo o contraseña incorrectas’, de la siguiente manera:

Si el usuario inicia sesión con alguno de los usuarios que se presentan a continuación, podrá acceder de manera correcta al dashboard.

Para tener esta funcionalidad se creó el siguiente código:

```
const handleClick = async () => {
  try {
    const response = await axios.get(getUrl);
    response.data.map((user) => {    Array.prototype.map() expects a re
      if(user.email === userEmail && user.password === userPassword) {
        setShowError(false);
        localStorage.setItem("Email", userEmail);
        navigate('/dashboard', { replace: true });
      }
      if(user.email !== userEmail || user.password !== userPassword) {
        setShowError(true);
      }
    })
  } catch (error) {
    console.error(error);
    setShowError(true);
  }
};
```

Esta función la asignamos al botón ‘Login’, consumimos la API con axios, iterando el array con la información y con una condicional validamos si el usuario y contraseña que ingreso el usuario coinciden con la información que esta en nuestra API, de ser así con la función ‘navigate’ lo dirigimos a la página del dasboard, sino, activamos el estado del error en true para que se muestre el mensaje de error, de haber un error en el consumo de la API, lo capturamos en el catch y volvemos a activar el mensaje de error.

Normalmente esta función para el login de usuarios se hace mediante un post, y el back nos da la respuesta si el usuario esta registrado o no.

Adicional a esto, el correo del usuario lo guardamos en el localStorage, si el usuario recarga la pagina igual va a seguir logueado, y esto lo hacemos para traer mas datos del usuario mediante su correo.

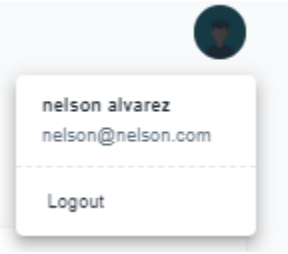
Ya estando logueados, traemos la información del usuario, nombre, correo y foto guardadas en la API. Para llamar esta info tenemos la siguiente funcionalidad:

```
function userLogin(user) {
  return user.email === localStorage.getItem("Email");
}

const name = async () => {
  try {
    const response = await axios.get(getUrl);
    let findResponse = response.data.find(user => userLogin(user));
    setUserName(findResponse.name);
    setUserEmail(findResponse.email);
    setUserPhoto(findResponse.photoURL);
  } catch (error) {
    console.error(error);
  }
};

useEffect(() => {
  name();
}, [])    React Hook useEffect has a missing dependency: 'name'. Eith
```

En la función ‘userLogin’ enviamos el usuario que coincide con el correo que esta en localStorage y de acuerdo con eso sacamos nombre, correo y foto guardando cada uno de estos en un estado y así llamarlos de esta manera:

```
<Typography variant="subtitle2" noWrap>
  {userName}
</Typography>
<Typography variant="body2" sx={{ color:
  {userEmail}
</Typography>
```

Y esto renderiza el siguiente componente:



Estando acá nos encontramos con el botón de logout, a este botón se le asigno la siguiente funcionalidad:

```
const handleLogout = () => {
  localStorage.removeItem("Email");
  navigate('/', { replace: true });
};
```

Acá eliminamos la información guardada en el localStorage y lo enviamos a la raíz de la página, la cual es el login.

Otra funcionalidad para el inicio de sesión del usuario, es que si no está logueado no podrá acceder a la pagina del dashboard, la función para esto es:

```
useEffect(() => {
  if (!localStorage.getItem("Email")) {
    navigate("/", { replace: true });
    return;
  }
  fetchData();
}, []);    React Hook useEffect has a mis
```

Esta función esta en el archivo ‘pages/DashboardAppPage’, si el usuario pone la ruta del dashboard sin estar logueado lo primero que se valida es que en el localStorage este guardado el correo, si no es así lo redirige a la raíz (Login). Para el Login es lo mismo si el usuario ya esta logueado.

Para el llamado de la información del dashboard tenemos el siguiente código:

```
const getUrlDateVisits = "http://localhost:3000/dateVisits";
const getUrlRegisteredUsers = "http://localhost:3000/registeredUsers";
const getUrlCurrentVisits = "http://localhost:3000/currentVisits";
```

```
const fetchData = async () => {
  try {
    const responseDateVisits = await axios.get(getUrlDateVisits);
    const responseRegisteredUsers = await axios.get(getUrlRegisteredUsers);
    const responseCurrentVisit = await axios.get(getUrlCurrentVisits);
    setRegisteredUsersState(responseRegisteredUsers.data);
    setCurrentVisitsState(responseCurrentVisit.data);
    responseDateVisits.data.map((date1) => {    Array.prototype.map() expects a return
      setDateState((prevArray) => [...prevArray, date1.date]);
    });
    if (responseDateVisits.status === 200 && responseRegisteredUsers.status === 200 &&
      setDataComplete(true);
    }
  } catch (error) {
    console.error(error);
  }
};
```

Esta función se llama en el useEffect de ‘pages/DashboardAppPage’, dentro de un try, para capturar los errores, consumimos con axios los 3 endpoints, uno para los datos de visitas, otro para los usuarios registrados (No son los mismo con los que se inicia sesión), y el ultimo para las visitas actuales.

Para los usuarios registrados (getUrlRegisteredUsers) y visitas actuales (getUrlCurrentVisits) se guardan directamente en el estado, para el caso de los datos de visita (getUrlDateVisits) se debe iterar la respuesta del endpoint ya que solo necesitamos la fecha guardada en un array, para esto dentro del estado usamos spread operator para guardar los datos previos y agregar los que vayan llegando de la iteración.

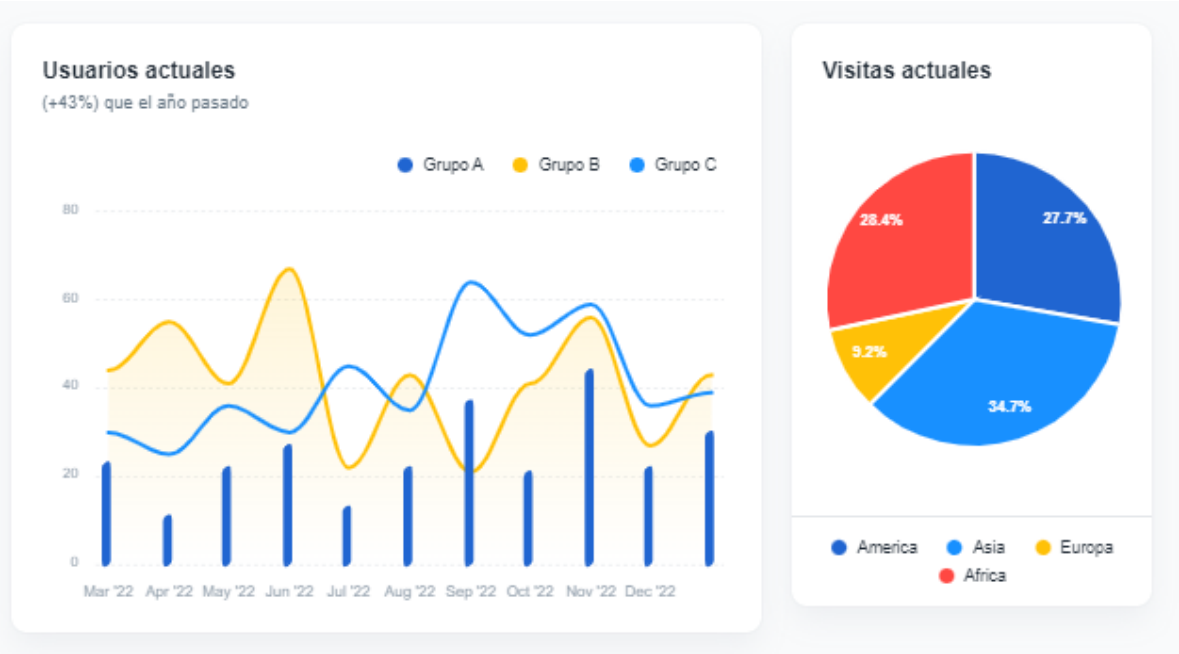
Agregamos una validación para saber si la respuesta de ambos enpoints fue 200 y así ya renderizar los datos en la interfaz de la siguiente manera:

```
{dataComplete ? (
  <Grid container spacing={3}>
    <Grid item xs={12} md={6} lg={8}>
      <AppWebsiteVisits
        title="Usuarios actuales"
        subheader="(+43%) que el año pasado"
        chartLabels={dateState}
        chartData={registeredUsersState}
      />
    </Grid>

    <Grid item xs={12} md={6} lg={4}>
      <AppCurrentVisits
        title="Visitas actuales"
        chartData={currentVisitsState}
        chartColors={[
          theme.palette.primary.main,
          theme.palette.info.main,
          theme.palette.warning.main,
          theme.palette.error.main,
        ]}
      />
    </Grid>
  </Grid>
) : (
  <div></div>
)}
```

Mientras que el estado dataComplete este en false se renderizara un vacío, apenas el estado cambie a true que es cuando ambas respuestas ya dieron estado 200, se renderizara el dashboard ya que si se renderiza antes la información va a estar incompleta.

Para el componente AppWebVisits enviamos dos propiedades, una con el estado ‘dateState’ y otra con el estado ‘registeredUsersState’ y para el componente AppCurrentVisits solo enviamos una propiedad con el estado ‘currentVisitsState’.

El componente se vería de la siguiente manera:



Al lado izquierdo el componente AppWebVisits, donde esta el grafico con filtros para los grupos y al lado derecho el componente AppWebVisits con una torta y al hacer hover en ambos componentes habrá un tooltip con más información detallada.