# 3. Frontend

## Componente comune

### **Componenta *ContextMenu***

Componenta *ContextMenu* implementează un meniu contextual personalizat, a cărui poziționare este determinată de coordonatele cursorului în momentul declanșării evenimentului asociat, de regulă, un click dreapta. Scopul acestei componente este de a oferi utilizatorului o listă de acțiuni relevante pentru contextul în care s-a efectuat interacțiunea.

Aceasta primește următoarele proprietăți:

* *x, y* – coordonatele la care trebuie randat meniul;
* *onClose* – funcția apelată la închiderea meniului;
* *options* – un array de obiecte, fiecare reprezentând o opțiune din meniu, cu componentele:
  + *label* – textul afișat utilizatorului;
  + *onClick* – funcția apelată în momentul selectării opțiunii.

const ContextMenu = ({ x, y, onClose, options }) => {

return ( /\*...\*/ );

};

Meniul este redat sub forma unei liste *<ul>* poziționate în funcție de coordonatele *x* și *y*. La apăsarea pe oricare dintre opțiuni, se declanșează atât funcția *onClick* specifică opțiunii, cât și închiderea meniului prin intermediul funcției *onClose*.

<ul

className="context-menu"

style={{ top: y, left: x, }}

onClick={onClose}

>

{options.map((option, index) => (

<li

key={index}

onClick={option.onClick}

>

{option.label}

</li>

))}

</ul>

### **Componenta *Modal***

Componenta *Modal* are rolul de a afișa o fereastră suprapusă peste conținutul aplicației, utilizată pentru confirmarea sau anularea unor acțiuni importante, precum și introducerea de către utilizator a unor informații suplimentare pentru crearea sau modificarea unor elemente vizuale. Aceasta reprezintă un mecanism de interacțiune temporar, care blochează accesul la restul interfeței până când utilizatorul oferă un răspuns explicit.

Proprietățile primite de această componentă sunt:

* *isOpen* – indicator boolean care controlează afișarea ferestrei modale, pentru evitarea prezenței inutile a acestei componente în DOM;
* *title* – titlul afișat în partea superioară a ferestrei;
* *children* – conținutul personalizat al corpului componentei, transmis din componenta părinte;
* *onCancel*, *onConfirm* – funcțiile apelate la anularea, respectiv confirmarea acțiunii;
* *confirmText, cancelText* – textele afișate pe butoanele de acțiune, cu valorile implicite “OK” pentru confirmare și “Cancel” pentru anulare.

const *Modal* = ({

*isOpen*, *title*, *children*, *onCancel*, *onConfirm*,

*confirmText* = "OK",

*cancelText* = "Cancel",

}) => {

if (!*isOpen*) return null;

useEffect(() => { /\*…\*/ }, [*isOpen*, *onCancel*]);

return ( /\*…\*/ );

};

Structura vizuală a acestei componente este compusă dintr-un fundal semi-transparent care acoperă întreaga pagină și o fereastră centrală care conține: titlul, zona de conținut și secțiunea pentru acțiuni, cu două butoane distincte pentru confirmarea sau anularea acțiunii.

<div *className*="modal-backdrop">

<div *className*="modal">

<h4>{*title*}</h4>

<div *className*="modal-body">{*children*}</div>

<div *className*="modal-actions">

<button *className*="btn" *onClick*={*onCancel*}> {*cancelText*} </button>

                    <button *className*="btn" *onClick*={*onConfirm*}> {*confirmText*} </button>

                </div>

            </div>

</div>

Un aspect semnificativ al acestei componente este utilizarea unui *useEffect* care ascultă apăsările de taste la nivel global, cât timp fereastra modală este deschisă:

useEffect(() => {

const *handleKeyDown* = (*e*) => {

if (*e*.*key* === "Enter") onConfirm();

if (*e*.*key* === "Escape") onCancel();

};

if (*isOpen*) *window*.addEventListener("keydown", *handleKeyDown*);

return () => *window*.removeEventListener("keydown", *handleKeyDown*);

}, [*isOpen*, *onCancel*]);

Apăsarea tastei *Enter* apelează aceeași funcție care este apelată de către butonul de confirmare, precum și apăsarea tastei *Escape* apelează similar funcția de anulare. Astfel, componenta oferă un control intuitiv utilizatorului, permițând navigarea atât prin mouse, cât și prin tastatură.

### **Componenta *Navbar***

Componenta *Navbar* reprezintă bara de navigație a aplicației și funcționează ca un meniu dinamic, care se adaptează contextului aplicației și al utilizatorului. Aceasta ghidează utilizatorul printre funcționalitățile principale, în funcție de autentificare și ruta curentă.

Lista *navItems* definește etichetele și rutele pentru secțiunile aplicației:

const *navItems* = [

{ label: "To-Do Lists", path: "/todos" },

{ label: "Calendar", path: "/calendar/week" },

{ label: "Pomodoro", path: "/pomodoro" },

{ label: "Stats", path: "/stats" },

];

Aceste elemente sunt afișate în bara de navigație doar dacă utilizatorul este autentificat și nu se află pe pagina de login. Fiecare link este generat dinamic, prin intermediul funcției *map*, iar proprietatea *isActive* este utilizată pentru a evidenția vizual pagina activă printr-un stil condițional:

{!isLoginPage && user && (

<div className="navbar-nav ms-auto d-flex flex-row gap-4">

{navItems.map(({ label, path }) => (

<NavLink className="nav-link" key={path} to={path}

style={({ isActive }) => ({

backgroundColor: isActive ? "var(--overlay15)" : "",

borderRadius: isActive ? "0.8rem" : "",

})}

> {label} </NavLink>

))}

<NavLink> {*/\*Log out\*/*} </NavLink>

</div>

)}

Un link suplimentar, “Log Out”, este afișat in partea dreaptă a meniului și permite deconectarea utilizatorului prin setarea acestuia ca fiind *null* și redirecționarea către pagina de autentificare, închizând sesiunea curentă.

<NavLink *className*="nav-link" *to*="#"

*onClick*={(*e*) => {

*e*.preventDefault();

setUser(null);

navigate("/");

}}

> Log Out </NavLink>

### **Contextul de autentificare *AuthContext***

Contextul de autentificare *AuthContext* oferă o soluție centralizată pentru gestionarea stării de autentificare a utilizatorului în cadrul aplicației, având rolul de a înfășura întreaga aplicație, cu scopul de a furniza contextul autentificării tuturor componentelor descendente.

Acesta este creat cu ajutorul metodei *createContext()* și este expus prin intermediul a două entități: componenta *AuthProvider* și hook-ul personalizat *useAuth*. În interiorul acestei componente este definită o stare locală numită *user* care stochează datele utilizatorului autentificat, precum și o stare booleană, *loading*, ce indică dacă aplicația a finalizat procesul de încărcare a datelor.

const *AuthContext* = createContext();

export const *AuthProvider* = ({ *children* }) => {

    const [*user*, *setUser*] = useState(null);

    const [*loading*, *setLoading*] = useState(true);

    useEffect(() => { /\*…\*/ }, []);

    useEffect(() => { /\*…\*/ }, [*user*]);

    return ( /\*…\*/ );

};

export const *useAuth* = () => useContext(*AuthContext*);

La montarea inițială a componentei, prin metoda *useEffect* cu dependență vidă, se încarcă preluarea informațiilor salvate anterior în *localStorage*. Dacă datele sunt valide, ele sunt parsate și introduse în starea locală prin *setUser*. În cazul în care conținutul nu poate fi interpretat, acesta este eliminat din *localStorage*, iar procesul de autentificare este invalidat.

useEffect(() => {

try {

const *storedUser* = *localStorage*.getItem("user");

if (*storedUser*) setUser(*JSON*.parse(*storedUser*));

} catch (*err*) {

*console*.error("Failed to parse stored user:", *err*);

*localStorage*.removeItem("user");

}

setLoading(false);

}, []);

Un al doilea *useEffect*, cu dependența *user*, monitorizează modificările asupra utilizatorului autentificat. Dacă există un utilizator activ, acesta este salvat în *localStorage*, permițând persistarea sesiunii între reîncărcări ale aplicației. În caz contrar, de exemplu în urma unui logout, datele sunt șterse pentru a menține coerența între starea aplicației și memoria locală.

useEffect(() => {

if (*user*) {

*localStorage*.setItem("user", *JSON*.stringify(*user*));

} else {

*localStorage*.removeItem("user");

}

}, [*user*]);

Valoarea expusă de *AuthContext.Provider* conține obiectul *user*, funcția de actualizare *setUser*, precum și starea de încărcare *loading*. Acestea pot fi accesate în orice componentă copil cu ajutorul hook-ului *useAuth*, care încorporează funcționalitatea *useContext(AuthContext)* într-o formă reutilizabilă și simplificată.

return (

<AuthContext.Provider *value*={{ *user*, *setUser*, *loading* }}>

{*children*}

</AuthContext.Provider>

);

### **Componenta *PrivateRoute***

Componenta *PrivateRoute* are rolul de a restricționa accesul la anumite secțiuni ale aplicației, permițând afișarea conținutului doar dacă utilizatorul este autentificat.

Aceasta primește o singură proprietate:

* *children* – componenta sau ruta care trebuie afișată condițional.

const *PrivateRoute* = ({ *children* }) => {

const { *user*, *loading* } = useAuth();

if (*loading*) return <p>Loading...</p>;

return *user* ? *children* : <Navigate *to*="/" />;

};

Componenta utilizează contextul de autentificare *useAuth* din *AuthContext* pentru a obține informațiile despre starea curentă a utilizatorului:

* *user* – obiectul de autentificare care conține datele utilizatorului;
* *loading* – indicator boolean ce semnalează dacă datele despre utilizator sunt încă în curs de încărcare.

Comportamentul logic al componentei începe prin verificarea indicatorului *loading*. Dacă acesta are valoare de adevăr *true*, atunci componenta returnează un mesaj temporar: “Loading…”. În caz contrar, dacă utilizatorul este autentificat, adică dacă obiectul *user* este definit, atunci conținutul din *children* este returnat și afișat. Dacă în schimb utilizatorul nu este autentificat, atunci acesta este redirecționat către pagina de autentificare (“/”).

Această abordare permite protejarea rutelor sensibile ale aplicației, oferind o soluție simplă, eficientă și reutilizabilă pentru controlul accesului bazat pe autentificare.

## Pagina *AuthPage*

Componenta *AuthPage* reprezintă interfața de autentificare a aplicației, oferind utilizatorului posibilitatea de a se conecta prin intermediul serviciului Google OAuth 2.0. Aceasta este responsabilă pentru inițierea procesului de autentificare, validarea token-ului JWT și setarea utilizatorului în contextul global.

În interiorul componentei sunt utilizate:

* *useNavigate()* – din *react-router-dom* pentru redirecționarea utilizatorului după autentificare;
* *useAuth()* – din contextul de autentificare pentru a actualiza starea globală *user*;
* *GoogleLogin* – din biblioteca *@react-oauth/google* pentru a declanșa procesul de autentificare cu Google.

const *AuthPage* = () => {

const *navigate* = useNavigate();

const { *setUser* } = useAuth();

return (

<div *className*="container auth-container">

<h3 *className*="text-center pb-2">Sign In</h3>

<GoogleLogin

*shape*="pill"

*onSuccess*={ /\*…\*/ }

*onError*={ /\*…\*/ }

/>

</div>

);

};

În momentul în care autentificarea este finalizată cu succes (*onSuccess*), este extras token-ul de autentificare *credentialResponse.credential*, care este transmis către serverul aplicației printr-un request *POST*. După primirea răspunsului de la server, datele utilizatorului sunt preluate și salvate în contextul global prin *setUser()*, iar utilizatorul este redirecționat către pagina principală a aplicației folosind *navigate()*. În cazul în care autentificarea eșuează (*onError*), se înregistrează o eroare în consolă, dar aplicația nu se întrerupe, pentru a oferi posibilitatea de reîncercare.

*onSuccess*={async (*credentialResponse*) => {

const *id\_token* = *credentialResponse*.*credential*;

const *res* = await fetch(`${import.meta.*env*.*VITE\_API\_BASE\_URL*}/auth/google`, {

method: "POST",

headers: { "Content-Type": "application/json" },

body: *JSON*.stringify({ token: *id\_token* }),

});

const *data* = await *res*.json();

setUser(*data*.*user*);

navigate("/todos");

}}

*onError*={() => { *console*.log("Login failed!"); }}