**UNIVERSITATEA DIN ORADEA**

**FACULTATEA DE INFORMATICĂ ȘI ŞTIINŢE**

**PROGRAMUL DE STUDIU: INFORMATICĂ**

**FORMA DE ÎNVĂŢĂMÂNT: CU FRECVENŢĂ**

**DEZVOLTAREA UNEI APLICAȚII CARE STIMULEAZĂ PRODUCTIVITATEA: KAIROS**

**COORDONATORŞTIINŢIFIC:**

**lECT. UNIV. DR. HOREA OROS**

**ABSOLVENTĂ:**

**Kéry Imola-Vivien**

**ORADEA**

**2025**

Cuprins

[1. Introducere 4](#_Toc201012726)

[1.1. Structura lucrării 4](#_Toc201012727)

[1.2. Aplicația propusă 4](#_Toc201012728)

[1.3. Tehnologii și resurse folosite 4](#_Toc201012729)

[2. Structura Backend 4](#_Toc201012730)

[2.1. Structura bazei de date 4](#_Toc201012731)

[2.1.1. Modelul *User* 4](#_Toc201012732)

[2.1.2. Modelul *Todo* 4](#_Toc201012733)

[2.1.3. Modelul *TodoList* 4](#_Toc201012734)

[2.1.4. Modelul *Event* 4](#_Toc201012735)

[2.2. Structura API-ului 4](#_Toc201012736)

[2.2.1. Gestionarea autentificării 4](#_Toc201012737)

[2.2.2. Gestionarea todo-urilor 4](#_Toc201012738)

[2.2.3. Gestionarea listelor todo 4](#_Toc201012739)

[2.2.4. Gestionarea calendarului 4](#_Toc201012740)

[3. Structura Frontend 4](#_Toc201012741)

[3.1. Componente 4](#_Toc201012742)

[3.1.1. Componenta *ContextMenu* 4](#_Toc201012743)

[3.1.2. Componenta *Modal* 6](#_Toc201012744)

[3.1.3. Componenta *Navbar* 8](#_Toc201012745)

[3.1.4. Contextul de autentificare *AuthContext* 9](#_Toc201012746)

[3.1.5. Componenta *PrivateRoute* 10](#_Toc201012747)

[3.1.6. Componenta *TodoList* 11](#_Toc201012748)

[3.2. Pagini 13](#_Toc201012749)

[3.2.1. Pagina *AuthPage* 13](#_Toc201012750)

[3.2.2. Pagina *TodoPage* 14](#_Toc201012751)

[3.2.1. Pagina *WeeklyCalendarPage* 18](#_Toc201012752)

[3.2.2. Pagina *PomodoroPage* 18](#_Toc201012753)

[4. Manual de utilizare 19](#_Toc201012754)

[4.1. Înregistrarea și autentificarea 19](#_Toc201012755)

[4.2. Interfața pentru gestionarea listelor todo 19](#_Toc201012756)

[4.3. Interfața pentru gestionarea calendarului 19](#_Toc201012757)

[4.4. Interfața pentru gestionarea cronometrului pomodoro 19](#_Toc201012758)

[Bibliografie 20](#_Toc201012759)

# Introducere

## Structura lucrării

## Aplicația propusă

## Tehnologii și resurse folosite

# Structura Backend

## Structura bazei de date

### Modelul *User*

### Modelul *Todo*

### Modelul *TodoList*

### Modelul *Event*

## Structura API-ului

### Gestionarea autentificării

### Gestionarea todo-urilor

### Gestionarea listelor todo

### Gestionarea calendarului

# Structura Frontend

## Componente

### Componenta *ContextMenu*

Componenta *ContextMenu* reprezintă un meniu contextual personalizabil, a cărui poziționare este determinată de coordonatele cursorului în momentul unui click dreapta. Scopul acestei componente este de a oferi utilizatorului o listă de acțiuni relevante pentru contextul în care s-a efectuat interacțiunea, precum ștergerea sau redenumirea unui element.

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 opțiunile disponibile se declanșează funcția *onClick* specifică opțiunii, după care meniul se închide 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>

Adițional, paginile care implementează această componentă permit închiderea meniului prin intermediul unui click în afara acestuia, oriunde pe pagina părinte.

useEffect(() => {

const *handleClick* = () => {

if (*contextMenu*.*visible*) {

setContextMenu({ ...*contextMenu*, visible: false });

}

};

*window*.addEventListener("click", *handleClick*);

return () => *window*.removeEventListener("click", *handleClick*);

}, [*contextMenu*]);

Acest *useEffect*, declanșat în urma apariției meniului contextual, adaugă funcția *handleClick* ca un ascultător de eveniment global asupra întregii ferestre, așteptând pentru orice click, pe urma căruia dacă meniul contextual este vizibil, acesta va fi ascuns. Ascultătorul este eliminat ori la actualizarea efectului, ori la demontarea paginii, pentru a evita scurgerile de memorie.

### 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 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.

Următoarea regulă CSS este responsabilă pentru afișarea fundalului *modal-backdrop*, folosit pentru a bloca interacțiunea cu restul paginii până când utilizatorul oferă un răspuns ferestrei modale:

*.modal-backdrop* {

position: fixed;

inset: 0;

background: *rgba*(0, 0, 0, 0.4);

display: flex;

align-items: center;

justify-content: center;

z-index: 999;

padding: 1rem;

box-sizing: border-box;

}

Proprietățile *position: fixed* și *inset: 0* se asigură că elementul acoperă întregul viewport, indiferent de scroll-ul curent, iar *z-index: 999* poziționează acest fundal semi-transparent deasupra conținutului principal al paginii.

### 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>

);

Acest mecanism de autentificare oferă o soluție reactivă și scalabilă pentru gestionarea stării de autentificare a utilizatorului, fiind ideal pentru aplicații React care necesită controlul accesului la resurse pe baza stării de autentificare, persistarea sesiunilor și o interfață coerentă între utilizator și sistem.

### 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.

### Componenta *TodoList*

Componenta *TodoList* este responsabilă afișarea și gestionarea unei liste de taskuri, permițând utilizatorului să interacționeze cu elementele acesteia. Această componentă este esențială sistemului de gestionare a sarcinilor, oferind o interfață simplă și eficientă pentru manipularea listelor.

Componenta primește următoarele proprietăți de la componenta părinte:

* *list* – obiectul care reprezintă o listă, conține titlul listei, precum și o listă de taskuri;
* *onToggleTodo –* funcția care permite modificarea stării de finalizare a unui task;
* *onAddTodo, onDeleteTodo –* funcțiile care permit adăugarea, respectiv ștergerea unui task;
* *onRightClick –* funcția care deschide meniul contextual aferent listei în urma unui click dreapta.

const *TodoList* = ({

*list*,

*onToggleTodo*,

*onAddTodo*,

*onDeleteTodo*,

*onRightClick*,

}) => {

const [*newTodoText*, *setNewTodoText*] = useState("");

const *handleSubmit* = (*e*) => { /\* ... \*/ };

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

};

Apelată în urma apăsării butonului pentru adăugare sau a tastei *Enter*, funcția *handleSubmit* verifică întâi validitatea textului introdus, apoi adaugă în listă un task nou prin apelarea funcției *onAddTodo* corespunzătoare din componenta părinte, golind pe urmă inputul pentru a pregăti aplicația pentru o nouă adăugare.

const [*newTodoText*, *setNewTodoText*] = useState("");

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

*e*.preventDefault();

if (!*newTodoText*.trim()) return;

if (*onAddTodo*) {

onAddTodo(*list*.*\_id*, *newTodoText*);

setNewTodoText("");

}

};

Lista de taskuri este redată utilizatorului sub forma unui card Bootstrap, ce conține o listă *<ul>* unde fiecare task este reprezentat printr-un element de listă. Fiecare task este stilizat în funcție de starea sa de finalizare prin intermediul claselor *todo* și *todo-done*, cea din urmă aplicând un efect *strikethrough* textului. În dreapta fiecărui element se află și un buton utilizat pentru ștergerea taskului, funcționalitatea acestuia fiind implementată în componenta părinte, precum și schimbarea stării de finalizare prin click simplu.

<ul *className*="list-group ">

{*list*.*todos*.map((*todo*) => (

<li *key*={*todo*.*\_id*}

*className*={`list-group-item d-flex ${

*todo*.*done* ? "todo-done" : "todo"

}`}

*onClick*={() => *onToggleTodo* && onToggleTodo(*todo*.*\_id*)}

>

<span>{*todo*.*text*}</span>

<button *className*="btn delete-button"

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

*e*.stopPropagation();

*onDeleteTodo* && onDeleteTodo(*list*.*\_id*, *todo*.*\_id*);

}}

> <EraserIcon *size*="18" *color*="#011627" /> </button>

</li>

))}

<li *className*="list-group-item"> { <form> /\* Adăugare \*/ <form/> } </li>

</ul>

De asemenea, componenta permite adăugarea unui task nou prin intermediul unui formular text, al cărui buton de *submit* apelează funcția *handleSubmit* definită anterior. Acest buton este activat doar atunci când câmpul text nu este gol, iar apăsarea lui poate fi înlocuită prin apăsarea tastei *Enter*.

<form *onSubmit*={*handleSubmit*} *className*="d-flex">

<input *type*="text" *className*="form-control todo-input"

*placeholder*="Add new todo" *value*={*newTodoText*}

*onChange*={(*e*) => setNewTodoText(*e*.*target*.*value*)}

/>

<button *className*="btn add-task-btn" *type*="submit"

*disabled*={!*newTodoText*.trim()}

> <MagicWandIcon *size*="30" *color*="#011627" /> </button>

</form>

### Componenta *WeeklyCalendar*

Componenta *WeeklyCalendar* reprezintă un calendar săptămânal interactiv folosit pentru gestionarea și vizualizarea evenimentelor stabilite de către utilizator, cu scopul de a ține evidența timpului acestuia.

Componenta primește următoarele proprietăți de la componenta părinte:

* *startDate –* data de început a săptămânii care trebuie afișată;
* *events –* lista evenimentelor care trebuie afișate;
* *onCellClick –* funcția apelată în momentul în care utilizatorul face click pe una din celulele calendarului;
* *onRightClickEvent –* funcția apelată atunci când utilizatorul face click dreapta pe un eveniment;
* *onDragStart, onDrop –* funcțiile apelate la tragerea și plasarea unui eveniment din calendar, folosite pentru a permite mutarea evenimentului în altă oră sau zi;
* *onResizeStart –* funcția apelată în momentul redimensionării unui eveniment pentru a schimba durata acestuia;

const *WeeklyCalendar* = ({

*startDate*, *events*,

*onCellClick*, *onRightClickEvent*,

*onDragStart*, *onDrop*,

*onResizeStart*,

}) => { /\* ... \*/ };

Header-ul calendarului afișează numele zilelor săptămânii urmat de numărul zilei, folosind metoda *format* din modulul *date-fns*.

<div *className*="calendar-header">

<div *className*="time-column-header" />

{*days*.map((*day*, *i*) => (

<div *className*="day-header text-center" *key*={*i*}> {format(*day*, "EEE dd")} </div>

))}

</div>

</div>

Pe urmă se creează câte un rând pentru fiecare oră din zi. Orele sunt afișate în formatul “01:00”, utilizând metoda *padStart* pentru a asigura faptul că acestea sunt randate cu două cifre.

{*hours*.map((*hour*) => (

<div *className*="time-row" *key*={*hour*}>

<div *className*="time-label text-end">

{`${String(*hour*).padStart(2, "0")}:00`}

</div>

{ /\* … \*/ }

</div>

))}

În continuare se creează celule interactive pentru fiecare oră din fiecare zi, atribuind fiecăreia funcțiile necesare pentru a permite utilizatorului să adauge sau să modifice evenimentele din calendar.

{*days*.map((*day*, *dayIndex*) => {

const *cellDate* = new Date(*day*);

*cellDate*.setHours(*hour*, 0, 0, 0);

return (

<div *className*="time-cell" *key*={*dayIndex*}

*onClick*={() => onCellClick(*dayIndex*, *hour*)}

*onDragOver*={(*e*) => *e*.preventDefault()}

*onDrop*={() => onDrop(*dayIndex*, *hour*)}

> { /\* … \*/ } </div>

);

})}

Urmează ca, iterând peste toate celulele calendarului și toate evenimentele săptămânii, să se verifice dacă evenimentul începe în aceeași zi și ora ca celula curentă din calendar. În caz afirmativ, se calculează durata evenimentului în minute, pentru a fi folosită în calcularea înălțimii block-ului folosit la afișarea acestuia și se returnează acest block pentru a fi afișat în celula corespunzătoare. În caz contrar, celula este lăsată goală prin returnarea obiectului *null*.

{*events*.map((*event*) => {

const *start* = typeof *event*.*startTime* === "string" ? parseISO(*event*.*startTime*) : *event*.*startTime*;

const *end* = typeof *event*.*endTime* === "string" ? parseISO(*event*.*endTime*) : *event*.*endTime*;

if ( isSameDay(*start*, *cellDate*) && isSameHour(*start*, *cellDate*)) {

const *durationMins* = differenceInMinutes(*end*, *start*);

const *blockHeight* = (*durationMins* / 60) \* *rowHeight*;

return ( { /\* … \*/ } );

}

return null;

}

Se atribuie toate funcțiile necesare interacțiunii cu evenimentul, acestea fiind afișarea meniului contextual în cazul unui click dreapta, precum și funcția apelată la mutarea evenimentului prin tragere. Din punct de vedere vizual, evenimentul este afișat sub forma unui block ce conține titlul acestuia, urmat de un mic control în partea inferioară care poate fi utilizat pentru redimensionare.

<div *key*={*event*.*\_id* || *event*.*title* + *start*.toISOString()} *className*="event-block" *draggable*

*style*={{ height: `${*blockHeight* - 1}px` }}

*onContextMenu*={(*e*) => {

*e*.preventDefault();

onRightClickEvent(*event*, *e*.*pageX*, *e*.*pageY*);

}}

*onDragStart*={(*e*) => onDragStart(*e*, *event*)}

>

<div *className*="event-content-wrapper">

<div *className*="event-block-content"> <p>{*event*.*title*}</p> </div>

<div *className*="resize-handle" *onMouseDown*={(*e*) => onResizeStart(*e*, *event*)} />

</div>

</div>

Un aspect important al componentei este următorul *useEffect*, care la montarea componentei efectuează un *scroll* automat al componentei, astfel încât utilizatorul să vadă evenimentele ce au loc după ora 6 dimineața, deoarece nu este foarte probabil ca programul acestuia să înceapă înainte de această oră:

const *scrollContainerRef* = useRef(null);

useEffect(() => {

if (*scrollContainerRef*.*current*) {

const *scrollToHour* = 6;

*scrollContainerRef*.*current*.*scrollTop* = *scrollToHour* \* *rowHeight*;

}

}, []);

### Componenta *Pomodoro*

Componenta *Pomodoro* implementează un cronometru interactiv folosit pentru tehnica de lucru Pomodoro, aceasta fiind o metodă de gestionare a timpului care presupune lucrul într-o sesiune concentrată de obicei de 25 de minute, urmată de o pauză scurtă. După patru sesiuni de lucru urmează o pauză mai lungă, de regulă de 15-30 de minute.

Singura proprietate primită de la componenta părinte este *times* – un obiect care conține timpii pentru diferitele faze ale tehnicii Pomodoro, folosit pentru a putea personaliza durata fiecărei faze.

const *Pomodoro* = ({ *times* }) => {

const *timerRef* = useRef(null);

/\* … \*/

return ( { /\* … \*/ }

};

Funcția *startTimer* implementează cronometrul propriu-zis, fiind responsabilă atât de pornirea acestuia, cât și de scăderea timpului în fiecare secundă. În momentul ajungerii la zero, se schimbă modul cronometrului din muncă în pauză, respectiv invers.

const *startTimer* = () => {

clearInterval(*timerRef*.*current*);

*timerRef*.*current* = setInterval(() => {

setTimeLeft((*prev*) => {

if (*prev* <= 1) {

clearInterval(*timerRef*.*current*);

switchMode();

return 0;

}

return *prev* - 1;

});

}, 1000);

};

Schimbarea dintre modurile cronometrului este efectuată de funcția *switchMode*. La apelarea acesteia se verifică întâi dacă ultima sesiune completată a fost una de lucru. În caz afirmativ, se verifică dacă aceasta a fost a patra astfel de sesiune, urmând o pauză lunga dacă da, iar în caz contrar o pauză scurtă. Dacă în schimb ultima sesiune a fost una de pauză, urmează una de lucru.

const *switchMode* = () => {

let *nextMode*;

if (*mode* === "work") {

const *newCount* = *pomodoroCount* + 1;

setPomodoroCount(*newCount*);

*nextMode* = *newCount* % 4 === 0 ? "long-break" : "short-break";

} else { *nextMode* = "work"; }

setMode(*nextMode*);

setTimeLeft(getTimeForMode(*nextMode*));

setIsRunning(false);

};

## Pagini

### Pagina *AuthPage*

Pagina *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!"); }}

### Pagina *TodoPage*

Pagina *TodoPage* este cea responsabilă pentru afișarea și gestionarea listelor de tip “to-do” asociate utilizatorului autentificat. Această pagină reunește multiple componente și mecanisme de stare pentru a oferi o interfață interactivă, funcțională și plăcută.

Pentru a asigura interactivitatea paginii sunt folosite numeroase stări, cele mai importante dintre acestea fiind:

* *lists* – ansamblul listelor todo;
* *contextMenu* – meniul contextual aferent fiecărei liste;
* starea de vizibilitate a următoarelor ferestre modale:
  + *showDeleteModal* – confirmarea ștergerii unei liste;
  + *showRenameModal* – redenumirea unei liste;
  + *showAddModal* – adăugarea unei liste noi.

În momentul autentificării utilizatorului, pagina preia datele acestuia din contextul global prin intermediul metodei *useAuth* din contextul de autentificare și încarcă din baza de date toate listele asociate acelui utilizator folosind ruta */todo-lists/user/:id*.

const { *user* } = useAuth();

const [*lists*, *setLists*] = useState([]);

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

useEffect(() => {

const *fetchLists* = async () => {

if (!*user*?.*\_id*) return;

try {

const *res* = await fetch(

`${import.meta.*env*.*VITE\_API\_BASE\_URL*}/todo-lists/user/${*user*.*\_id*}`

);

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

setLists(*data*);

} catch (*err*) {

*console*.error("Failed to fetch lists:", *err*);

} finally {

setLoading(false);

}

};

fetchLists();

}, [*user*]);

Conținutul paginii este organizat într-un grid cu doar trei coloane pentru a evidenția vizual listele utilizatorului fără aglomerare. Listele sunt reprezentate de *card*-uri Bootstrap stilizate care conțin câte o componentă *TodoList*, fiecare fiind capabilă să apeleze funcțiile de gestionare a listelor cu proprii parametrii, precum și afișarea propriului meniu contextual.

{*uniqueLists*.map((*list*) => (

<div *className*="col-md-4 mb-4" *key*={*list*.*\_id*}>

<TodoList *list*={*list*}

*onToggleTodo*={(*todoId*) =>

handleToggleTodo(*list*.*\_id*, *todoId*)

}

*onAddTodo*={(*listId*, *newTodoText*) =>

handleAddTodo(*listId*, *newTodoText*)

}

*onDeleteTodo*={(*listId*, *todoId*) =>

handleDeleteTodo(*listId*, *todoId*)

}

*onRightClick*={(*e*) => {

*e*.preventDefault();

setContextMenu({

visible: true,

x: *e*.*pageX*,

y: *e*.*pageY*,

listId: *list*.*\_id*,

});

}}

/>

</div>

))}

Pe urma afișării acestor carduri se mai adaugă încă un card adițional, fiind responsabil de afișarea ferestrei modale folosite pentru crearea unei liste noi.

<div *className*="col-md-4 mb-4">

<div *className*="card new-todo-card"

*style*={{ cursor: "pointer" }}

*onClick*={() => setShowAddModal(true)}

>

<div *className*="card-body">

<span> <QuillIcon *size*="64" *color*="#011627" /> </span>

</div>

</div>

</div>

Funcția *handleToggleTodo* este apelată în urma unui click asupra unui task dintr-o listă. Aceasta se ocupă de comutarea stării *done* („finalizat”) a unui task, actualizând atât baza de date printr-o cere *PATCH* pe ruta */todos/:id*, precum și starea locală a aplicației pentru a reflecta schimbarea în interfața vizuală. Actualizarea celei din urmă este realizată prin parcurgerea tuturor listelor și taskurilor pentru a actualiza doar taskul corespunzător.

const *handleToggleTodo* = async (*listId*, *todoId*) => {

const *currentList* = *lists*.find((*list*) => *list*.*\_id* === *listId*);

const *currentTodo* = *currentList*?.*todos*.find((*todo*) => *todo*.*\_id* === *todoId*);

if (!*currentTodo*) return;

const *newDone* = !*currentTodo*.*done*;

try {

await fetch(`${import.meta.*env*.*VITE\_API\_BASE\_URL*}/todos/${*todoId*}`, {

method: "PATCH",

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

body: *JSON*.stringify({ done: *newDone* }),

});

setLists((*prev*) =>

*prev*.map((*list*) =>

*list*.*\_id* === *listId*

? {

...*list*,

todos: *list*.*todos*.map((*todo*) =>

*todo*.*\_id* === *todoId*

? { ...*todo*, done: *newDone* }

: *todo*

),

}

: *list*

)

);

} catch (*err*) {

*console*.error("Failed to toggle todo:", *err*);

}

};

În mod analog se execută și funcțiile *handleAddTodo* pentru adăugarea unui task nou prin intermediul unei cereri *POST* pe ruta */todos*, precum și *handleDeleteTodo* pentru ștergerea unui task existent cu o cerere *DELETE* pe ruta */todos/:id*, urmate de actualizarea stării locale.

Redenumirea unei liste este implementată prin funcția *handleRenameList*, care funcționează în mod similar funcțiilor anterioare de manipulare a taskurilor. După verificarea titlului nou introdus se trimite o cerere *PUT* către server pentru a actualiza numele listei în baza de date, pe ruta */todo-lists/:id*, urmată de actualizarea stării locale a listelor. În urma finalizării cu succes a actualizărilor se închide fereastra modală folosită și se golește inputul pentru a pregăti aplicația pentru o eventuală utilizare viitoare.

const *handleRenameList* = async (*listId*) => {

if (!*newTitle*.trim()) return;

try {

const *res* = await fetch(

`${import.meta.*env*.*VITE\_API\_BASE\_URL*}/todo-lists/${*listId*}`, {

method: "PUT",

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

body: *JSON*.stringify({ title: *newTitle* }),

}

);

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

setLists((*prev*) =>

*prev*.map((*list*) =>

*list*.*\_id* === *listId* ? { ...*list*, title: *newTitle* } : *list*

)

);

} catch (*err*) {

*console*.error("Failed to rename list:", *err*);

} finally {

setShowRenameModal(false);

setNewTitle("");

}

};

În mod analog se efectuează atât adăugarea unei liste noi prin funcția *handleAddList* printr-o cerere *POST* pe ruta */todo-lists*, cât și ștergerea unei liste prin funcția *handleDeleteList* printr-o cerere *DELETE* pe ruta */todo-lists/:id*.

### Pagina *WeeklyCalendarPage*

### Pagina *PomodoroPage*

# Manual de utilizare

## Înregistrarea și autentificarea

## Interfața pentru gestionarea listelor todo

## Interfața pentru gestionarea calendarului

## Interfața pentru gestionarea cronometrului pomodoro

# Bibliografie