**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](#_Toc201092526)

[1.1. Aplicația propusă 4](#_Toc201092527)

[1.2. Tehnologii folosite 4](#_Toc201092528)

[2. Structura Backend 5](#_Toc201092529)

[2.1. Structura bazei de date 5](#_Toc201092530)

[2.1.1. Colecția *Users* 5](#_Toc201092531)

[2.1.2. Colecția *Todos* 6](#_Toc201092532)

[2.1.3. Colecția *Todo\_Lists* 7](#_Toc201092533)

[2.1.4. Colecția *Events* 8](#_Toc201092534)

[2.2. Structura API-ului 8](#_Toc201092535)

[2.2.1. Gestionarea autentificării 10](#_Toc201092536)

[2.2.2. Gestionarea todo-urilor 10](#_Toc201092537)

[2.2.3. Gestionarea listelor todo 11](#_Toc201092538)

[2.2.4. Gestionarea calendarului 11](#_Toc201092539)

[3. Structura Frontend 12](#_Toc201092540)

[3.1. Punctul de intrare *main.jsx* 12](#_Toc201092541)

[3.2. Fișierul principal *App.jsx* 13](#_Toc201092542)

[3.3. Componente 14](#_Toc201092543)

[3.3.1. Componenta *ContextMenu* 14](#_Toc201092544)

[3.3.2. Componenta *Modal* 15](#_Toc201092545)

[3.3.3. Componenta *Navbar* 17](#_Toc201092546)

[3.3.4. Contextul de autentificare *AuthContext* 18](#_Toc201092547)

[3.3.5. Componenta *PrivateRoute* 20](#_Toc201092548)

[3.3.6. Componenta *TodoList* 20](#_Toc201092549)

[3.3.7. Componenta *WeeklyCalendar* 22](#_Toc201092550)

[3.3.8. Componenta *Pomodoro* 25](#_Toc201092551)

[3.4. Pagini 29](#_Toc201092552)

[3.2.1. Pagina *AuthPage* 29](#_Toc201092553)

[3.2.2. Pagina *TodoPage* 31](#_Toc201092554)

[3.2.1. Pagina *WeeklyCalendarPage* 34](#_Toc201092555)

[3.2.2. Pagina *PomodoroPage* 40](#_Toc201092556)

[4. Manual de utilizare 41](#_Toc201092557)

[4.1. Înregistrarea și autentificarea 41](#_Toc201092558)

[4.2. Interfața pentru gestionarea listelor *todo* 42](#_Toc201092559)

[4.3. Interfața pentru gestionarea calendarului 44](#_Toc201092560)

[4.4. Interfața pentru gestionarea cronometrului pomodoro 46](#_Toc201092561)

[Bibliografie 47](#_Toc201092562)

[Resurse preluate 47](#_Toc201092563)

# Introducere

## Aplicația propusă

Aplicația implementată în această lucrare reprezintă un sistem de gestionare a timpului, menit să ajute în sporirea productivității utilizatorului. Aceasta oferă acces în primul rând la o interfață prin intermediul căreia se facilitează procesul de observare și urmărire al sarcinilor utilizatorului sub forma listelor de tip *todo*. De asemenea, este pus la dispoziția utilizatorului un calendar săptămânal care ușurează planificarea timpului, precum și un cronometru de tip *Pomodoro* care poate ajuta utilizatorul atât în concentrarea pe sarcina de lucru curentă prin reducerea distragerilor, cât și în luarea de pauze semnificative, care să ajute în productivitate.

Am ales să abordez acest subiect în lucrare deoarece am simțit că am nevoie de o asemenea aplicație pentru gestionarea propriului meu timp, negăsind, după numeroase încercări, nicio aplicație existentă care combină toate aceste elemente într-un mod intuitiv, ușor de folosit și cu un aspect plăcut care nu distrage atenția utilizatorului. Astfel, am putut crea o aplicație despre care știu în mod sigur faptul că rezolvă o problemă din viața reală, aceasta fiind o problemă întâlnită și de mine însumi.

## Tehnologii folosite

Pentru implementarea *backend*-ului este folosit mediul de execuție *open-source* Node.js, în cadrul căreia se utilizează Express, un *framework* minimalist și flexibil, care oferă un set robust de unelte pentru dezvoltarea aplicațiilor web.

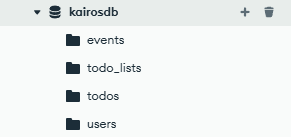
Baza de date a aplicației este implementată prin MongoDB, o bază de date de tip *NoSQL* orientată pe documente, care stochează datele într-un format *JSON* flexibil, găzduită (*hosted*) pe platforma MongoDB Atlas.

Structura de *frontend* a aplicației este implementată prin biblioteca React, utilizată extensiv pentru crearea de interfețe de utilizator interactive și dinamice. Aceasta permite dezvoltarea modulară a aplicației prin componente reutilizabile.

O versiune a acestei aplicații este găzduită și pe Render, o platformă *cloud* care oferă un mediu de găzduire rapid și ușor de folosit, la adresa <https://kairos-frontend-fas7.onrender.com>.

# Structura Backend

## Structura bazei de date

Baza de date a aplicației este implementată prin MongoDB și are următoarea structură de colecții:

* *events –* ansamblul evenimentelor create de utilizatori;
* *todo\_lists –* ansamblul listelor todo create de utilizatori;
* *todos –* ansamblul taskurilor create de utilizatori;

Figura 2.1:  
Structura bazei de date

* *users –* datele utilizatorilor înregistrați.

Fiecare document din fiecare colecție enumerată are un câmp *\_id*, generat automat, care este un identificator unic al documentului respectiv.

### Colecția *Users*

Colecția *Users* este folosită pentru a stoca informații despre utilizatorii înregistrați. Aceasta folosește următoarea schema Mongoose pentru a valida și structura datele stocate:

const *userSchema* = new *mongoose*.Schema(

{

googleId: {

type: *String*,

required: true,

unique: true,

},

email: {

type: *String*,

required: true,

unique: true,

lowercase: true,

trim: true,

},

name: {

type: *String*,

trim: true,

},

},

{ timestamps: true }

);

Câmpul *googleId* conține un identificator unic generat de Google pentru fiecare utilizator, necesar pentru autentificare. De asemenea, este stocată adresa de email și numele utilizatorului. Prin utilizarea *flag*-ului *timestamps: true* se creează automat câmpurile *createdAt* și *updatedAt*, pentru a urmări când a fost creat și când a avut loc ultima actualizarea a documentului.

### Colecția *Todos*

Colecția *Todos* este utilizată pentru stocarea task-urilor individuale ale utilizatorilor, folosind următoarea schema pentru structurarea și validarea datelor din documente:

const *todoSchema* = new *mongoose*.Schema(

{

todoListId: {

type: *mongoose*.*Schema*.*Types*.*ObjectId*,

ref: "TodoList",

required: true,

},

text: {

type: *String*,

required: true,

trim: true,

},

done: {

type: *Boolean*,

default: false,

},

},

{ timestamps: true }

);

Fiecare document din această colecție este asociat unei liste din colecția *Todo\_Lists* prin intermediul câmpului *todoListId* care face referire la lista corespunzătoare. De asemenea sunt stocate textul aferent taskului, precum și starea de finalizare a acestuia.

### Colecția *Todo\_Lists*

Documentele din colecția *Todo\_Lists*, perechea colecției *Todos*, stochează informațiile despre listele todo ale utilizatorilor, prin intermediul următoarei schema:

const *todoListSchema* = new *mongoose*.Schema(

{

userId: {

type: *mongoose*.*Schema*.*Types*.*ObjectId*,

ref: "User",

required: true,

},

title: {

type: *String*,

required: true,

trim: true,

},

},

{ timestamps: true, toJSON: { virtuals: true }, toObject: { virtuals: true } }

);

Câmpul *userId* face referire la un utilizator din colecția *Users*, ținând evidență de creatorul listei, după care se stochează și titlul acesteia.

De asemenea, fiecare document conține un câmp virtual *todos*, definit pentru a permite accesul la toate task-urile asociate acestei liste. Fiind un câmp virtual, acesta nu este salvat în baza de date, dar poate fi accesat pentru a returna toate task-urile care au câmpul *todoListId* identic cu câmpul *\_id* al listei curente.

*todoListSchema*.virtual("todos", {

ref: "Todo",

localField: "\_id",

foreignField: "todoListId",

});

### Colecția *Events*

Colecția *Events* este folosită pentru a stoca informațiile referitoare la evenimentele calendaristice create de utilizatori, utilizând următoarea schema:

const *eventSchema* = new *mongoose*.Schema(

{

userId: {

type: *mongoose*.*Schema*.*Types*.*ObjectId*,

ref: "User",

required: true,

},

title: {

type: *String*,

required: true,

trim: true,

},

startTime: {

type: *Date*,

required: true,

},

endTime: {

type: *Date*,

required: true,

},

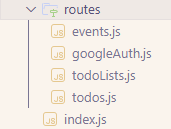
},

{ timestamps: true }

);

Precum și documentele din colecția *Todo\_Lists*, documentele din această colecție fac referire la un utilizator din colecția *Users* prin intermediul câmpului *userId*. De asemenea sunt salvate titlul evenimentului calendaristic, precum și datele și timpul de început și final al fiecărui eveniment.

## Structura API-ului

API-ul, sau serverul, aplicației este structurat astfel:

* directorul *routes* – conține rutele adiționale folosite de server, împărțite pe mai multe fișiere *.js* pentru a nu supraîncărca fișierul principal;
* fișierul *index.js* – fișierul principal pentru rularea serverului.

Figură 2.2:  
Structura de fișiere a API-ului

*Entry-point*-ul serverului, fișierul *index.js*, creează întâi o instanță a aplicației Express și definește portul acesteia.

const *app* = express();

const *port* = *process*.*env*.*PORT* || 3000;

Urmată de o configurare *CORS* (*Cross-Origin Resource Sharing*) pentru a permite accesul doar de la anumite origini, acestea fiind *frontend*-ul local al aplicației sau cel hostat pe Render, protejând astfel serverul de solicitări neautorizate din alte surse. *Flag-*ul *credentials: true* este folosit pentru partajarea *cookie*-urilor de sesiune între server și client.

const *allowedOrigins* = [ /\* fronten local \*/, /\* frontend hostat \*/ ];

*app*.use(

cors({

origin: function (*origin*, *callback*) {

if (!*origin* || *allowedOrigins*.includes(*origin*)) {

callback(null, true);

} else {

callback(new Error("CORS not allowed for this origin"));

}

},

credentials: true,

})

);

Rutele adiționale sunt importate din fișierele corespunzătoare din directorul *routes*.

*app*.use("/api/auth", *authRoutes*);

*app*.use("/api/todos", *todoRoutes*);

*app*.use("/api/todo-lists", *todoListRoutes*);

*app*.use("/api/events", *eventRoutes*);

Pe urmă se definește ruta de rădăcină *index* (*/*), trimițând utilizatorul către pagina principală (*index.html*).

*app*.get("/", (*req*, *res*) => {

*res*.sendFile(*path*.join(*\_\_dirname*, "../client/dist/index.html"));

});

La final se stabilește conexiunea la baza de date folosind Mongoose, după care serverul începe să asculte pe portul specificat.

*mongoose*

.connect(*process*.*env*.*MONGO\_URI*)

.then(() => {

*app*.listen(*port*, () => {

*console*.log(`Running on http://localhost:${*port*}`);

}); })

.catch((*err*) => *console*.error("MongoDB connection error: ", *err*));

### Gestionarea autentificării

Autentificarea este gestionată prin librăria oficială Google, *google-auth-library*. Fișierul *googleAuth.js* definește un *router* Express folosit la autentificarea utilizatorului prin Google OAuth2.

Mai întâi se efectuează verificarea token-ului Google prin metoda *verifyIdToken*, stocând pe urmă datele utilizatorului: ID-ul unic Google, emailul și numele.

const *ticket* = await *client*.verifyIdToken({

idToken: *token*,

audience: *process*.*env*.*GOOGLE\_CLIENT\_ID*,

});

const *payload* = *ticket*.getPayload();

const { *sub*, *email*, *name* } = *payload*;

După care se efectuează o căutare a acestui utilizator în baza de date. În cazul în care nu se găsește niciun utilizator cu aceste date, este creat unul nou.

let *user* = await *User*.findOne({ googleId: *sub* });

if (!*user*) {

*user* = await *User*.create({

googleId: *sub*,

*email*,

*name*,

});

}

### Gestionarea todo-urilor

Fișierul *todos.js* conține rutele necesare operațiilor *CRUD* (*Create, Read, Update, Delete*) pentru gestionarea task-urilor. Acestea sunt:

* *POST /* – folosită la crearea de taskuri noi, așteptând parametrii *todoListId* și *text* în corpul cererii;
* *GET /list/:id* – returnează toate task-urile conținute în lista identificată prin *id;*
* *PATCH /:id* – utilizată pentru a schimba starea de finalizare a task-ului identificat prin *id;*
* *DELETE /:id* – permite ștergerea task-ului identificat prin *id.*

### Gestionarea listelor todo

Fișierul *todoLists.js* oferă rutele folosite pentru gestionarea listelor utilizatorilor:

* *POST /* – utilizată pentru a crea o listă nouă, preluând datele din corpul cererii și creând pe urmă o nouă listă goală;
* *GET /user/:id* – returnează toate listele todo ale utilizatorului identificat prin *id*, utilizând metoda *populate* pentru a popula listele;

const *lists* = await *TodoList*.find({ userId: *req*.*params*.*id* })

.populate("todos")

.exec();

* *PUT /:id* – permite actualizarea unei liste, identificată prin *id*, noile date fiind preluate din corpul cererii
* *DELETE /:id* – șterge o listă identificată pe baza *id*.

### Gestionarea calendarului

Fișierul *events.js* conține rutele necesare gestionării evenimentelor calendaristice ale utilizatorilor:

* *GET /user/:userId* – returnează toate evenimentele unui utilizator identificat prin *userId* aferente unei săptămâni specificate de parametrul *weekStart*;

const *events* = await *Event*.find({

*userId*,

startTime: { $gte: *start*, $lt: *end* },

}).sort({ startTime: 1 });

* *POST /* – folosit pentru crearea unui eveniment nou, detaliile acestuia fiind transmise prin corpul cererii;
* *PUT /:id* – permite actualizarea unui eveniment identificat prin *id*, conținutul nou al acestuia fiind preluat din corpul cererii;
* *DELETE /:id* – utilizat pentru ștergerea unui eveniment identificat prin *id*.

# Structura Frontend

A screenshot of a computer

AI-generated content may be incorrect.Frontend-ul, sau client-ul, aplicației are următoarea structură de fișiere:

Figură 3.1:  
Structura de fișiere a client-ului

* directorul *dist* – fișierele construite ale aplicației;
* directorul *src* – codul sursă al aplicației:
  + directorul *assets* – fișierele statice, folosit pentru a stoca iconițele SVG folosite în aplicație;
  + directorul *components* – componentele reutilizabile;
  + directorul *context* – fișierele care gestionează starea globală a aplicației;
  + directorul *pages* – componentele de nivel înalt care includ mai multe componente din *components*;
  + fișierul *App.jsx* – fișierul principal al aplicației;
  + fișierul *main.jsx* – punctul de intrare al aplicației.

## Punctul de intrare *main.jsx*

Fișierul *main.jsx* reprezintă *entry-point*-ul aplicației, unde este montată aplicația în *DOM* (*Document Object Model*). Acest fișier face legătura dintre *index.html* și *App.jsx*. Tot aici este introdus *GoogleOAuthProvider*, pentru a permite componentelor interne utilizarea autentificării prin Google.

createRoot(*document*.getElementById("root")).render(

    <StrictMode>

        <GoogleOAuthProvider *clientId*={import.meta.*env*.*VITE\_GOOGLE\_CLIENT\_ID*}>

            <BrowserRouter>

                <AuthProvider>

                    <App />

                </AuthProvider>

            </BrowserRouter>

        </GoogleOAuthProvider>

    </StrictMode>

);

## Fișierul principal *App.jsx*

În acest fișier se află componenta principală a aplicației, numită *App*, care definește rutele și structura aplicației și afișează bara de navigație, pentru a fi disponibilă pe toate paginile aplicației. Rutele protejate sunt învelite de componenta *PrivateRoute*, pentru a asigurarea faptului că doar utilizatorii autentificați au acces la acestea.

Rutele aplicației sunt următoarele:

* *AuthPage* – pagina de autentificare;
* *TodoPage* – pagina listelor todo;
* *WeeklyCalendarPage* – pagina calendarului săptămânal;
* *PomodoroPage* – pagina cronometrului Pomodoro.

function App() {

    return (

        <>

            <Navbar />

            <Routes>

                <Route *path*="/" *element*={<AuthPage />} />

                <Route *path*="/todos" *element*={

                        <PrivateRoute>

                            <TodoPage />

                        </PrivateRoute>

                    }

                />

                <Route *path*="/calendar/week" *element*={

                        <PrivateRoute>

                            <WeeklyCalendarPage />

                        </PrivateRoute>

                    }

                />

                <Route *path*="/pomodoro" *element*={

                        <PrivateRoute>

                            <PomodoroPage />

                        </PrivateRoute>

                    }

                />

            </Routes>

        </>

    );

}

## 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",

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

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;

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, folosindu-se de referința *timerRef* care stochează ID-ul generat de *setInterval*, 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);

};

Pornirea cronometrului este declanșată de către un *useEffect* care reacționează la schimbările stării *isRunning*.

const [*isRunning*, *setIsRunning*] = useState(false);

useEffect(() => {

if (*isRunning*) startTimer();

return () => clearInterval(*timerRef*.*current*);

}, [*isRunning*]);

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);

};

Un al doilea *useEffect* este folosit pentru actualizarea cronometrului în cazul schimbării modului sau al timpilor din proprietatea *times*, pentru a reflecta noul interval de timp.

useEffect(() => {

setTimeLeft(getTimeForMode(*mode*));

}, [*mode*, *times*]);

Funcția *toggleTimer* comută starea cronometrului între activ și inactiv pentru implementarea butonului de start/pauză.

const *toggleTimer* = useCallback(() => {

setIsRunning((*prev*) => !*prev*);

}, []);

Cel de al treilea *useEffect* permite utilizatorului să pornească sau oprească cronometrul prin apăsarea tastei *Space*, ascultând la nivel global pentru apăsarea acesteia.

useEffect(() => {

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

if (*e*.*code* === "Space") {

*e*.preventDefault();

toggleTimer();

}

};

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

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

}, [*toggleTimer*]);

Funcția *resetTimer* efectuează o resetare totală, oprind cronometrul și revenind la starea inițială.

const *resetTimer* = () => {

clearInterval(*timerRef*.*current*);

setIsRunning(false);

setPomodoroCount(0);

setMode("work");

setTimeLeft(getTimeForMode("work"));

};

Din punct de vedere vizual, această componentă afișează mai întâi modul curent într-un mod condiționat, încadrat între două iconițe care reflectă acesta. Pe lângă verificarea faptului că modul curent este de lucru sau nu, se verifică și tipul de pauză (scurtă sau lungă).

<div *className*="mode-container d-flex justify-content-center">

{*mode* === "work" ? (

<div *className*="d-flex gap-3 ">

<FlashIcon *size*="40" *color*="#011627" />

<h3 *className*="badge-text">Work Time</h3>

<FlashIcon *size*="40" *color*="#011627" />

</div>

) : (

<div *className*="d-flex gap-3 ">

<TulipIcon *size*="40" *color*="#011627" />

<h3 *className*="badge-text">

{*mode* === "short-break"

? "Break Time"

: "Long Break Time"}

</h3>

<TulipIcon *size*="40" *color*="#011627" />

</div>

)}

</div>

Urmează afișarea timpului rămas pe cronometru, precum și o reprezentarea vizuală a acestuia prin intermediul unei bare de progres. Lățimea acestei bare este calculată procentual pe baza timpului rămas, iar culoarea sa se schimbă în funcție de modul curent al cronometrului.

<div *className*="timer display-1 my-3">{formatTime(*timeLeft*)}</div>

<div *className*="progress" *style*={{ height: "20px" }}>

<div *className*={"progress-bar"} *role*="progressbar"

*style*={{ width: `${(*timeLeft* / getTimeForMode(*mode*)) \* 100}%`,

backgroundColor: *mode* === "work" ? "var(--prim)" : "var(--sec)",

}}

/>

</div>

În cele din urmă, se afișează două butoane de control: primul dintre acestea comută starea cronometrului, actualizându-și propria etichetă corespunzător stării actuale (*Pause / Start*), iar cel de al doilea este folosit pentru revenirea la setările inițiale ale cronometrului.

<div *className*="timer-btns my-4">

<button *className*="btn me-4" *onClick*={*toggleTimer*}>

{*isRunning* ? "Pause" : "Start"}

</button>

<button *className*="btn" *onClick*={*resetTimer*}>

Reset

</button>

</div>

## Pagini

Componentele prezentate anterior sunt folosite pentru a construi pagini care constituie interfața prin care utilizatorul poate interacționa cu aplicația.

### 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 *WeeklyCalendarPage* reprezintă implementarea unei interfețe vizuale pentru gestionarea, afișarea și interacțiunea cu un calendar săptămânal personalizat.

Pentru a asigura interactivitatea acestei componente sunt folosite următoarele stări principale:

* *currentWeekStart –* începutul săptămânii curente, stocat în acest mod pentru a permite utilizatorului să schimbe între săptămânile calendarului;
* *events –* lista evenimentelor aferente săptămânii curente;
* *modalData, modalMode, showAddEditModal, showDeleteModal* – folosite de către ferestrele modale responsabile de crearea/editarea și ștergerea evenimentelor;
* *draggedEvent, resizingEvent, isResizing* – folosite în implementarea funcțiilor de *drag & drop* și redimensionare.

Încărcarea evenimentelor se efectuează în mod analog paginii *TodoPage*, singurele diferențe fiind faptul că sunt încărcate doar evenimentele pentru săptămâna curentă, precum și faptul că această încărcare va fi redeclanșată în cazul modificării săptămânii.

useEffect(() => {

const *fetchEvents* = async () => {

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

try {

const *res* = await fetch(

`${import.meta.*env*.*VITE\_API\_BASE\_URL*}/events/user/${

*user*.*\_id*

}?weekStart=${*currentWeekStart*.toISOString()}`

);

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

setEvents(*data*);

} catch (*err*) {

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

} finally {

setLoading(false);

}

};

fetchEvents();

}, [*user*, *currentWeekStart*]);

Funcția *handleAddEditEvent* are un rol dublu, fiind responsabilă atât de crearea unui nou eveniment, cât și de actualizarea unui eveniment existent. Ea este implementată în acest mod, deoarece datele trimise către baza de date sunt aproape identice în cele două cazuri. Astfel, datele primite prin fereastra modală de adăugare sau cea de editare sunt trimise către server printr-o cerere *POST*, respectiv *PUT*, după care urmează actualizarea listei de evenimente locale, închiderea ferestrei modale și pregătirea stării *setModalData* pentru următoarea operație de creare sau editare.

const *handleAddEditEvent* = async () => {

if (!*modalData*.*title*.trim()) return;

const *eventPayload* = {

...*modalData*,

startTime: new Date(*modalData*.*startTime*).toISOString(),

endTime: new Date(*modalData*.*endTime*).toISOString(),

};

try {

if (*modalMode* === "create") {

const *res* = await fetch( /\* POST \*/ );

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

setEvents((*prev*) => [...*prev*, *created*]);

} else {

const *res* = await fetch( /\* PUT \*/ );

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

setEvents((*prev*) =>

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

(*event*.*\_id* === *updated*.*\_id* ? *updated* : *event*))

);

}

} catch (*err*) {

*console*.error("Failed to save event:", *err*);

} finally {

setShowAddEditModal(false);

setModalData(null);

}

};

Ștergerea unui eveniment este efectuată de funcția *handleDeleteEvent*. Aceasta, similar cu ștergerile din pagina *TodoPage*, elimină evenimentele din baza de date prin intermediul unei cereri de tip *DELETE*, urmată de actualizarea stării local și ascunderea ferestrei modale, precum și a meniului contextual.

const *handleDeleteEvent* = async () => {

try { await fetch( /\* DELETE \*/ );

setEvents((*prev*) => *prev*.filter((*event*) => /\* ... \*/);

} catch (*err*) {

*console*.error("Failed to delete event:", *err*);

} finally {

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

setShowDeleteModal(false);

}

};

Funcționalitatea *drag & drop* folosită pentru relocarea evenimentelor în altă oră sau zi este implementată de funcțiile *handleDragStart* și *handleDrop*. Funcția responsabilă pentru *drop* calculează noile valori pentru începutul și lungimea evenimentului după plasarea acestuia în celula nouă, actualizând aceste date printr-o cerere *PUT* și pe urmă actualizând starea locală, în final pregătind starea *draggedEvent* pentru următoarea operație de mutare a unui eveniment.

const *handleDragStart* = (*e*, *event*) => { setDraggedEvent(*event*); };

const *handleDrop* = async (*dayIndex*, *hour*) => {

if (!*draggedEvent*) return;

const *originalStart* = new Date(*draggedEvent*.*startTime*);

const *originalEnd* = new Date(*draggedEvent*.*endTime*);

if (isNaN(*originalStart*) || isNaN(*originalEnd*)) { /\* ... \*/ return; }

const *newStart* = new Date(*currentWeekStart*);

*newStart*.setDate(*newStart*.getDate() + *dayIndex*);

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

const *duration* = *originalEnd*.getTime() - *originalStart*.getTime();

const *newEnd* = new Date(*newStart*.getTime() + *duration*);

if (isNaN(*newStart*) || isNaN(*newEnd*)) { /\* ... \*/ return; }

const *updatedEvent* = {

...*draggedEvent*,

startTime: *newStart*.toISOString(),

endTime: *newEnd*.toISOString(),

};

try {

const *res* = await fetch( /\* PUT \*/ );

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

setEvents((*prev*) =>

*prev*.map((*event*) => (*event*.*\_id* === *updated*.*\_id* ? *updated* : *event*))

);

} catch (*err*) {

*console*.error("Failed to update event:", *err*);

} finally {

setDraggedEvent(null);

}

};

Pentru redimensionarea evenimentelor este folosit următorul *useEffect*, care actualizează grafic block-ul evenimentului în timp real pe măsură ce utilizatorul trage partea inferioară a acestuia cu mouse-ul:

useEffect(() => {

if (!*resizingEvent*) return;

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

const *handleMouseUp* = async () => { /\* ... \*/ };

*document*.addEventListener("mousemove", *handleMouseMove*);

*document*.addEventListener("mouseup", *handleMouseUp*);

return () => {

*document*.removeEventListener("mousemove", *handleMouseMove*);

*document*.removeEventListener("mouseup", *handleMouseUp*);

};

}, [*resizingEvent*]);

Funcția *handleMouseMove* din cadrul acesteia este responsabilă de actualizarea vizuală *real-time* a evenimentului. Aceasta calculează poziția cursorului utilizatorului pentru a determina ora pe baza acestuia, setând pe urmă noua oră de sfârșit a evenimentului și actualizând starea locală a evenimentului dacă noua oră este validă (are loc după ora de început a evenimentului).

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

const *scrollContainer* = *document*.querySelector(".calendar-scrollable-body");

const *calendarTop* = *scrollContainer*.getBoundingClientRect().*top*;

const *pixelY* = *e*.*clientY* - *calendarTop* + *scrollContainer*.*scrollTop*;

const *hour* = *Math*.floor(*pixelY* / 40);

const *newEnd* = new Date(*resizingEvent*.*startTime*);

*newEnd*.setHours(*hour* + 1, 0, 0, 0);

if (*newEnd* > new Date(*resizingEvent*.*startTime*)) {

setResizingEvent((*prev*) => ({ ...*prev*, endTime: *newEnd*.toISOString(),}));

setEvents((*prev*) => *prev*.map((*event*) =>

*event*.*\_id* === *resizingEvent*.*\_id*

? { ...*event*, endTime: *newEnd*.toISOString() }

: *event*

));

}

};

Finalizarea procesului de redimensionare are loc prin intermediul funcției *handleMouseUp*, care transmite datele noi printr-o cerere *PUT*, pregătind pe urmă stările relevante pentru următoarea operație de redimensionare.

const *handleMouseUp* = async () => {

if (!*resizingEvent*) return;

try {

await fetch( /\* PUT \*/ );

} catch (*err*) {

*console*.error("Failed to resize evenet");

} finally {

setIsResizing(false);

setResizingEvent(null);

}

};

Vizual, pagina este redată sub forma unui *container* ce conține în primul rând anul aferent datei curente, urmat de datele de început și sfârșit ale săptămânii actuale, încadrate între două butoane ce permit schimbarea săptămânii actuale prin intermediul funcțiilor *subWeeks* și *addWeeks* din modulul *date-fns*.

<h6 *className*="text-center">{*currentWeekStart*.getFullYear()}</h6>

<div *className*="week-nav d-flex justify-content-center align-items-center gap-3 mb-3">

<button *className*="btn"

*onClick*={() => setCurrentWeekStart((*prev*) => subWeeks(*prev*, 1))}

> <LeftArrowIcon *size*="24" *color*="#2a272c" /> </button>

<h4>{*formattedWeek*}</h4>

<button *className*="btn"

*onClick*={() => setCurrentWeekStart((*prev*) => addWeeks(*prev*, 1))}

> <RightArrowIcon *size*="24" *color*="#2a272c" /> </button>

</div>

În cele din urmă se afișează componenta *WeeklyCalendar*, atașând toate funcțiile necesare funcționalității acesteia.

<WeeklyCalendar *startDate*={*currentWeekStart*} *events*={*events*}

*onCellClick*={(*dayIndex*, *hour*) => { handleCellClick(*dayIndex*, *hour*); }}

*onRightClickEvent*={(*event*, *x*, *y*) => {

setContextMenu({ visible: true, *x*, *y*, *event* });

}}

*onDragStart*={*handleDragStart*}

*onDrop*={*handleDrop*}

*onResizeStart*={*handleResizeStart*}

/>

### Pagina *PomodoroPage*

Pagina *PomodoroPage* este folosită pentru încadrarea și afișarea componentei *Pomodoro*, precum și a ferestrei modale folosite pentru gestionarea timpilor cronometrului.

La inițializarea paginii sunt setate întâi valorile implicite pentru timpii de lucru și cei de pauză:

const [*times*, *setTimes*] = useState({

work: 25,

shortBreak: 5,

longBreak: 15,

});

Structura vizuală a paginii este una simplă, majoritatea logicii cronometrului fiind implementată de către componenta *Pomodoro*. Aceasta este alcătuită dintr-un *container* care conține componenta *Pomodoro*, urmat de un buton folosit pentru modificarea timpilor impliciți de lucru și de pauză, atât scurtă cât și lungă, permițând personalizarea totală a cronometrului.

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

<div *className*="text-center mb-4">

<Pomodoro *times*={*times*} />

<button *className*="btn mt-4" *onClick*={() => setShowSettingsModal(true)}>

Settings

</button>

</div>

</div>

# Manual de utilizare

Acest capitol este menit să fie folosit ca un manual de utilizare vizual al aplicației propuse, detaliind cele mai importante funcționalități ale acesteia.

## Înregistrarea și autentificarea

A sign in box with text and a logo

AI-generated content may be incorrect.

Figură 4.1:  
Butonul de autentificare

Pagina de autentificare are un singur element interactiv, butonul de autentificare prin Google. În urma apăsării acestuia este afișată o fereastră *pop-up* pentru finalizarea înregistrării sau autentificării.

A screenshot of a computer

AI-generated content may be incorrect.

Figură 4.2:  
Fereastra pop-up pentru autentificare

După autentificare, un element comun al fiecărei pagini este bara de navigare aflată în partea superioară a paginii, care poate fi folosită pentru navigarea între paginile aplicației, precum și pentru delogare.



## Interfața pentru gestionarea listelor *todo*

Pagina responsabilă pentru afișarea listelor *todo* redă acestea sub forma unor liste plasate pe un *grid*, câte trei liste pe un rând.

A screenshot of a computer

AI-generated content may be incorrect.

Figură 4.3:  
Interfața paginii To-Do Lists

*Todo*-urile din fiecare listă pot fi marcate finalizate printr-un click asupra lor, sau șterse cu butonul de ștergere asociat fiecăruia, care apare în momentul în care cursorul este deasupra *task*-ului.

A brown rectangle

AI-generated content may be incorrect.

Figură 4.4:  
Controalele asociate unui task

Fiecare listă are în partea inferioară propriile controale pentru adăugarea *task*-urilor noi, prin intermediul unei casete text însoțite de un buton de confirmare. Adăugarea poate fi confirmată de asemenea și prin apăsarea tastei *Enter*.

A close up of a wall

AI-generated content may be incorrect.

Figură 4.5:  
Controalele pentru adăugarea unui task nou

Prin intermediul unui click dreapta poate fi afișat meniul contextual folosit pentru ștergerea sau redenumirea unei liste.

A close up of a computer screen

AI-generated content may be incorrect.

Figură 4.6:  
Meniul contextual

În urma apăsării oricărei opțiuni din acest meniu este deschisă o fereastră modală care cere confirmarea sau introducerea de date utilizatorului. Aceste ferestre, pe lângă butoanele de control, pot fi confirmate și prin apăsarea tastei *Enter*, precum și anulate prin apăsarea tastei *Escape*.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

Figură 4.7:  
Ferestrele modale

La finalul afișării tuturor listelor se găsește butonul folosit pentru a adăuga o listă nouă, deschizând de asemenea o fereastră modală pentru introducerea titlului acesteia.

A pixelated image of a feather pen

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

Figură 4.8:  
Crearea unei liste noi

## Interfața pentru gestionarea calendarului

Pagina prin care poate fi gestionat calendarul este afișată sub forma unui tabel, unde fiecare coloană reprezintă o zi, iar fiecare rând o oră a acelei zile. Calendarul poate fi *scroll*-at în sus sau în jos folosind *scroll wheel*-ul mouse-ului.

A screenshot of a computer

AI-generated content may be incorrect.

Figură 4.9:  
Pagina WeeklyCalendar

Săptămâna afișată poate fi modificată prin intermediul butoanelor de navigație din partea superioară a paginii.

A screenshot of a calendar

AI-generated content may be incorrect.

Apăsând pe oricare dintre celulele tabelului poate fi creat un eveniment nou, prin intermediul unei ferestre modale.

A screenshot of a computer

AI-generated content may be incorrect.

Figură 4.10:  
Adăugarea unui noi eveniment

Prin intermediul unui click dreapta asupra unui eveniment din calendar poate fi afișat și pe această pagină un meniu contextual, cu opțiuni și funcționalitate similară meniului din pagina *To-Do Lists*.

A screenshot of a computer screen

AI-generated content may be incorrect.

Figură 4.11:  
Meniul contextual

De asemenea, evenimentele pot fi repoziționate prin *drag & drop*, sau redimensionate prin tragerea controlului din partea inferioară a fiecărui eveniment.

A close-up of a sign

AI-generated content may be incorrect.

Figură 4.12:  
Un eveniment

## Interfața pentru gestionarea cronometrului pomodoro

Pagina *Pomodoro* afișează un cronometru, însoțit de controalele asociate.

A screenshot of a clock

AI-generated content may be incorrect.

Cronometrul poate și pornit prin apăsarea butonului *Start* sau a tastei *Space*. Butonul *Reset* este folosit pentru readucerea cronometrului la starea implicită. Butonul *Settings* deschide fereastra modală folosită pentru modificarea valorilor implicite ale cronometrului.

A screenshot of a computer screen

AI-generated content may be incorrect.

Figură 4.13:  
Fereastra de setări

# Bibliografie

1. Bootstrap documentation. (fără an). Preluat de pe https://getbootstrap.com/docs/4.1/getting-started/introduction/
2. Express API Reference. (fără an). Preluat de pe https://expressjs.com/en/5x/api.html
3. Google Cloud Documentation. (fără an). Preluat de pe https://cloud.google.com/docs
4. MDN Web Docs. (fără an). Preluat de pe https://developer.mozilla.org/en-US/
5. MongoDB Documentation. (fără an). Preluat de pe https://www.mongodb.com/docs/
6. Mongoose Documentation. (fără an). Preluat de pe https://mongoosejs.com/docs/
7. Node.js v24.2.0 Documentation. (fără an). Preluat de pe https://nodejs.org/docs/latest/api/
8. React Reference Overview. (fără an). Preluat de pe https://react.dev/reference/react

## Resurse preluate

Toate icoanele SVG folosite au fost preluate din următorul pachet gratis și *open-source*:  
<https://www.streamlinehq.com/icons/pixel>.