**1. Introducere**

Tema abordată în această lucrare de licență constă în implementarea unei aplicații de gestionare a task-urilor, care permite organizarea eficientă a mai multe tipuri de activități, utilizând principii reactive alături de alte concepte compatibile. Aceasta face parte din domeniul dezvoltării web, cu accent pe subdomeniul programării reactive.

Într-un context în care productivitatea și echilibrul personal devin priorități, o aplicație intuitivă, vizual atractivă și capabilă să înțeleagă obiceiurile utilizatorului reprezintă un plus major. Într-un timp în care societatea pare să se miște accelerat și nevoile oamenilor sunt într-o continuă schimbare, stresul cotidian începe să apară și el din ce în ce mai des în rândul tuturor. Lipsa de organizare creează, astfel, un haos pe care mulți nu reușeșc să îl gestioneze, iar stresul și problemele de acest gen se agravează pe baza lui.

Motivația personală provine din două locuri. În primul rând, pornește din interesul crescut față de interacțiunea dintre interfețele moderne și serviciile scalabile, dar și din dorința de a înțelege și a aplica conceptele de programare reactivă într-un proiect integral, de la idee la implementare. În al doilea rând, fiind o astfel de persoană care este constant sabotată de propria-i lipsă de disciplină, dezvoltarea unei astfel de aplicații reprezintă un adjuvant în viața mea personală.

Mai mult, interfața prezintă o tematică estetică aparte, gândită atent pentru a da o notă de personalitate aplicației NovaPlanner. Atât numele cât și stilul vizual sunt inspirate de stilul de design ”Frutiger Aero”, apărut în mijlocul anilor 2000, caracterizat de tematica optimistă a armonizării dintre tehnologie și natură. Acum când protejarea climatului și a mediului înconjurător devine din ce în ce mai relevant și mai important, am considerat că este important ca acest stil unic și nostalgic să revină în atenția consumatorului de rând pentru a-i aminti de importanța acestui fapt, pe lângă, desigur, stilul plăcut vizual contruit pe baza ei.

Contribuția proprie se regăsește în toatr etapele realizării lucrării: proiectarea arhitecturii aplicației, implementarea logicii backend cu ASP.NET Core și MongoDB, dezvoltarea interfeței cu React.js și integrarea unui serviciu de inteligență artificială prin Semantic Kernel pentru personalizarea experienței utilizatorului.

Lucrarea este structurată pe 5 capitole:

* Capitolul 1 – Introducerea, unde se prezintă toate detaliile introductive spre înțelegerea temei și a ideii lucrării,
* Capitolul 2 – Preliminarii, unde se descrie baza tehnică și situația actuală care susțin necesitatea realizării lucrării,
* Capitolul 3 – Proiectarea și arhitectura aplicației, care detaliază clar și cursiv baza pe care s-a desfășurat implementarea proiectului, justificând alegerile luate,
* Capitolul 4 – Implementarea și dezvoltarea aplicației, unde se explică pas cu pas fluxurile existente ale aplicației și modul în care au fost create,
* Capitolul 5 – Concluzii, care oferă sinteza tuturor rezultatelor obținute și încheie structura lucrării.

Urmărim, astfel, prin prezenta lucrare nu doar realizarea unei aplicații complete și funcționale, ci și explorarea modului în care paradigma aceasta de programare și inteligența artificială pot coexista într-un produs modern, util și adaptabil nevoile reale are utilizatorilor.

**2. Preliminarii**

**2.1 Noțiunile științifice și tehnice de la baza temei**

Programarea reactive a devenit utilizată tot mai frecvent pentru dezvoltarea aplicațiilor moderne. Dispunând de capacitate mai mare de a gestiona într-un mod eficient fluxuri de date asincrone și alte evenimente care pot apărea imprevizibil, devine o opțiune preferată în rândul programatorilor. Abordarea aceasta are la bază modelul de publicare-abonare (sau *publish-subscribe*) care permite componentelor aplicației dezvoltate să reacționeze la modificările apărute la date sau la starea sistemului, de aici și numele. Comparativ cu ceea ce numim programare imperativă tradițională, controlul nu mai este explicit, iar execuția își pierde liniaritatea, programarea reactive adoptând în schimb un mod declarativ de a defini relațiile dintre date și comportamente.

Ecosistemul .NET reprezintă un stâlp de rezistență în susținerea acest tip de abordare prin implementarea de tehnologii precum *Reactive Extensions (Rx.NET)* sau de pattern-uri arhitecturale ca *Model-View-ViewModel (MVVM)* sau *Observer Pattern*, care promovează fluxuri de date unidirecționale sau actualizări în timp real. Integrarea dintre un backend care are la bază ASP.NET și un frontend construit cu React.js și alte framework-uri JavaScript creează posibilitatea realizării unor interfețe responsive, care reflectă imediat modificările de stare ale aplicației.

Pe lângă aceste tehnologii mai apare și MongoDB care, deși nu reprezintă în sine o componentă reactive prin natura ei, se integrează ușor într-un flux bazat pe această paradigmă și devine, astfel, încă un mod de susținere al ei în aplicațiile moderne. Fiind o soluție scalabilă NoSQL orientată pe documente în schimbul tabelelor tradiționale, aceasta favorizează programarea reactivă prin intermediul driverelor asincrone și prin modalitatea de gestiune a operațiilor I/O. MongoDB poate fi integrată în .NET folosind pattern-uri asincrone (*async/await*) sau librării cum este cea menționată mai sus, anume *Rx.NET,* fiind eficient permisă propagarea modificărilor de date către frontend în acest mod. Astfel, datele conținute în baza de date pot fi consumate într-un mod reactiv și contribuie la sistemul designat să răspundă în timp real la acțiunile utilizatorului.

**2.2 Status quo**

În momentul de față, domeniul gestiunii de sarcini personale sau profesionale are un monopol bine definit în industria software, acaparat de multe soluții comerciale sau open-source, precum Todoist, Habitica, OpenProject sau Microsoft To Do. Aceste aplicații variază de la simple liste de sarcini menite să funcționeze ca un adjuvant în organizarea timpului personal al individului până la platforme complexe de management al proiectelor folosite în mediul corporate, iar utilizatorii au la îndemână opțiuni de urmărire a progresului, planificare zilnică și colaborare.

În ultimii ani, trendul observabil s-a bazat pe accentuarea interactivității, a personalizării și a inteligenței contextuale, aspecte care au dus la adoptarea unor arhitecturi reactive, microservicii și tehnologii bazate pe cloud. În ciuda acestui fapt, multe aplicații rămân dependente de actualizări manuale sau de sincronizări periodice, iar potențialul deplin al fluxurilor de date reactive rămâne încă neexploatat. Acest context deschide oportunitatea unei aplicații care se bazează pe principii reactive în toate layerele de dezvoltare (proiectare, comunicare și prezentare) să ofere o experiență mai fluidă, mai adaptivă și mai scalabilă pentru utilizatori.

**2.3 Scopul lucrării**

În contextul stabilit în secțiunile 2.1 și 2.2, obiectivul principal al lucrării este reprezentat de intenția explorării și aplicării principiilor programării reactive în vederea dezvoltării unei aplicații web pentru organizarea personală. *NovaPlanner*, un manager de task-uri, devine obiectul principal al studiului propus pentru a evidenția modul în care această paradigmă poate îmbunătăți atât structura codului, cât și experiența utilizatorului. În primplan sunt următoarele:

* un flux de date reactiv între interfață și serverul ASP.NET,
* actualizare în timp real a informațiilor afișate, fiind eliminată necesitatea reîncărcării paginii,
* analiza impactului pe care îl are arhitectura reactive asupra scalabilității și întreținerii aplicației,
* interfață prietenoasă, intuitivă și interactivă care reflectă organizarea și progresul utilizatorului, inspirată estetic de stilul retro ”Frutiger Aero”.

Deși punctul culminant al acestei lucrări este în mod evident reprezentat de dezvoltarea integrală a unei aplicații funcționale, ideea aflată la baza ei include evidențierea avantajelor oferite de adoptarea programării reactive într-un context practic, relevant pentru utilizatorii finali.

**3. Proiectarea și arhitectura aplicației**

**3.1 Arhitectura generală**

Pentru a avea o separare cât mai clară între componentele aplicației web, am ales dezvoltarea pe două niveluri principale, de frontend și de backend, creând astfel o arhitectură de tip client-server. Aceasta separă clar responsabilitățile dintre partea de interfață, respectiv logica de business împreună cu baza de date, facilitând dezvoltarea independentă a celor două constituente principale, dar și scalabilitatea și mentenabilitatea pe termen lung. Împărțirea clară în acest fel mi-a permis navigarea printre fișiere, oferind posibilitatea unui proces mai fluid de implementare și o transparență mai bună a codului.

Pentru partea de frontend, framework-ul React.js a stat la baza interfețelor reactive, fiind utilizat împreună cu Vite sub forma unui bundler pentru o dezvoltare rapidă și eficientă. Node.js, deși nu contribuie la logica aplicației, creeză mediul de rulare pentru cele două, fiind necesar pentru gestiunea pachetelor din acest ecosistem JavaScript. Pentru navigarea între pagini am folosit React Router, iar stilizarea interfețelor a fost realizată cu fișiere personalizate de CSS, integrând însă și mici componente de Bootstrap 5(icon-urile), de React-Bootstrap (care, spre deosebire de Bootstrap clasic, nu se mai folosește de jQuery), dar și React Date Picker, o componentă open-source. Apelurile către backend sunt efectuate de Axios, configurat printr-o instanță dedicată care injectează automat token-ul JWT, folosit pentru identificarea utilizatorului conectat. Stările locale și efectele reactive sunt gestionate de către *useState* și *useEffect*, hook-uri dezvoltate pentru a înlocui necesitatea claselor în React.

Pe partea de backend, aplicația se bazează în întregime pe ASP.NET, framework-ul robust și performant dezvoltat de Microsoft pentru dezvoltarea de API-uri REST. Acesta este integrat cu MongoDB pentru baza de date, care este flexibil pentru modelarea datelor și ideal în tipul de aplicație cu cerințe dinamice, cum este cazul celei dezvoltate. Pe lângă toate acestea, prin intermediul Azure Kernel Semantics, am integrat cu ASP.NET și Azure AI Foundry pentru asistentul AI al organizatorului.

**3.2 Structurarea soluției**

Fiecare dintre cele două componente are propriul folder în care este organizată. Aceste directoare principale au fost autogenerate la crearea proiectului în aplicația Visual Studio 2022, IDE-ul pe care l-am folosit în implementare. Frontend-ul se află în *organizer.client*, conținând aplicația React, iar backend-ul în *Organizer.Server*, care conține aplicația ASP.NET Core. Fiecare dintre aceste directoare conține la rândul său alte subdirectoare, menite să mențină organizarea clară a componentelor proiectului.

În *organizer.client*, fișierele pe care le-am creat sunt împărțite în subfolderele *api*, *components*, *pages* și *styles*, delimitând concret funcționalitatea fiecăruia. Pe același principiu, pe partea de backend în *Organizer.Server* există *Controllers*, *Models*, *Plugins* și *Services.* Aceste decizii de organizare a proiectului nu au fost luate doar din prisma convenției clasice, ci reflectă clar arhitectura modernă client-server. Această structură avantajează flexibilitatea dezvoltării și permite adoptarea celor mai bune practici de implementare în acest mediu.

În Anexa 1 se poate regăsi structura întegrală a aplicației, așa cum este încărcată pe platforma Github.

**3.3 Frontend**

**3.3.1 Paginile principale și interfața**

Incluse în folder-ul *pages*, fișierele JavaScript XML (.jsx) găzduiesc paginile funcționale ale aplicației NovaPlanner. *Login* și *Register* servesc ca pagini incipiente în fluxul vizual al aplicației, fiind primele interacțiuni pe care le are utilizatorul cu aplicația. Ele sunt folosite pentru autentificare și pentru crearea unui cont în cadrul organizatorului. *Dashboard* este pagina centrală, nucleul care incorporează componentele principale ale interfeței. Aici am dezvoltat ce vede utilizatorul odată trecut de ecranul de login, anume toate task-urile și butonul care integrează asistentul AI. Paginile *ForgotPassword* și *ResetPassword* sunt folosite pentru recuperarea contului în caz de pierdere a parolei, iar secțiunea *ChangePassword* reprezintă pagina de schimbare a parolei de către utilizator. Cum a fost menționat în secțiunea 3.1, comunicarea dintre pagini are loc cu sistemul de rute și stări locale creat de *useState* și *useEffect* care răspund de reacția la modificările de date sau la acțiunea utilizatorului.

**3.3.2 Componentele reutilizabile și stilizarea**

Directorul *components* conține componentele reutilizabile ale aplicației, anume *TaskItem*, *GoalWithCheckpoints* sau *ToDoList*. Acestea oferă modularitate, permițând afișarea datelor într-un mod coerent și ușor de întreținut. Separarea aceasta a avut loc ca și consecință a amplitudinii fișierului *Dashboard*, care acum cuprinde referințe la componentele actuale. Fișierele conțin elementele afișate pe dashboard; task-urile și obiectivele sunt intermediate prin lista to-do, care este afișată pe pagina principală.

Pe partea de stilizare folosim fișierele CSS cu cod personalizat aflate în *styles* pentru fiecare pagină a aplicației web. Prin acestea am conferit stilul estetic ”Frutiger Aero”, modificând elementele paginii, inclusiv cele deja standardizate cum ar fi icon-urile Bootstrap, butoanele React-Bootstrap și React Date Picker-ul împreună cu alte inputuri. Astfel, se creează atmosfera retro-modernă plăcută vizual.

**3.3.3 Integrarea cu backend-ul**

Integrarea cu backend-ul este conținută în folder-ul *api*, unde fișierele .jsx realizează comunicarea cu logica din spatele aplicației. *AxiosInstance* este instanța comună pe care am definit-o pentru comunicarea cu serverul prin trimiterea de request-uri la care se adaugă în mod automat token-ul identificator pentru utilizator. *UserService* și *AssistantService* sunt servicii în care am definit funcții clare pentru apelurile API necesare în fluxul de autentificare, respectiv cel de interacțiune cu asistentul inteligent.

**3.4 Backend**

**3.4.1 Controlerele și rutele API**

*TaskController, UserController* și *AssistantController* sunt fișierele create în folder-ul *Controllers* și se ocupă de endpoint-urile RESTful. Operațiile CRUD (Create, Read, Update, Delete) pentru task-uri sunt gestionate de *TasksController*, *UserController* este responsabil de toate funcționalitățile din fluxul de autentificare (înregistrare, autentificare, resetarea parolei și validarea token-urilor JWT), iar *AssistantController* este cel care expune ruta prin care frontend-ul trimite task-urile zilnice către asistentul AI pentru sugestii. Toate aceste controlere respectă convențiile REST de creare a resurselor, formând o structură organizată și extensibilă a API-ului.

**3.4.2 Servicii și logică**

Controlerele sunt fiecare legate de serviciul propriu, care sunt referențiate din directorul *Services*. *TaskServices* conține toată logica pentru gestiunea task-urilor și alte mici funcționalități relevante, *UserService* se ocupă de logica de autentificare, cu toate măsurile de securitate și altele adiacente, iar *SemanticKernelService* deține responsabilitatea în interacțiunea cu asistentul AI și evaluarea planului zilnic al utilizatorului. Pe lângă acestea, în folder există și fișierul *MongoDBService* care oferă acces centralizat și reutilizabil la colecțiile din baza de date.

**3.4.3 Plugin-uri**

Folderul *Plugins* este cel care conține *TaskAnalysisPlugin*, fișierul care se ocupă de formatarea KPI-urilor spre a fi mai apoi trimise prin prompt-ul din controler mai departe la modelul creat în Azure AI Foundry.

**3.4.4 Modelele de date**

În *Models* putem regăsi modelele de date (entitățile), care descriu forma și conținutul informației gestionate de aplicația web. Ele fac legătura dintre backend și baza de date din MongoDB, dar sunt necesare și pentru validarea și organizarea datelor în cadrul logicii proiectului.

*TaskItem* și *User* sunt modele persistente, definind structura documentelor stocate în bază. Pe lângă ele, *MongoDBSettings*, *JwtSettings* și *EmailSettings* conțin informații necesare pentru funcționalități gestionate de backend, fiind modele de configurare. Acestea nu sunt salvate în Mongo, deci sunt considerate nepersistente. Valorile lor sunt reținute prin diverse metode în interiorul aplicației, cum ar fi prin fișierul *appsettings.json* sau în spațiul de stocare secret al .NET, numit User Secrets. Prin acest mod, datele sunt preluate automat la pornirea aplicației și injectate prin Dependency Injection. Celelalte modele din director, *DailyTaskKpi*, *EvaluateRequest* și *TaskDto* sunt tot nepersistente, create pentru a fi utilizate doar temporar în cadrul procesării interne, folosite în principal în conexiunea cu AI-ul.

**3.5 Fluxul de date și reactivitatea**

Aplicația dezvoltată integrează principiile programării reactive prin intermediul unui mecanism de actualizare automată a interfeței în momentul în care se produce o schimbare, astfel lipsindu-ne de nevoia unui refresh manual. Pentru a înțelege cum funcționează, se consideră ca exemplu momentul în care un utilizator adaugă un task nou și ce se întamplă în spatele interfeței pe care o vede: se lansează un request de tip POST către backend, task-ul este validat și salvat, iar apoi *useState* și *useEffect* actualizează instantaneu interfața cu noua listă de sarcini care să îl afișeze pe partea de frontend. Are loc o reacție imediată și o experiență naturală pentru utilizator. Asemănător se petrec lucrurile și pentru asistentul AI, care, odată apelat de către utilizator, se declanșează automat pentru a oferi sugestia de optimizare. Acest lanț de acțiuni reactive contribuie semnificativ în experiența avută prin interacțiunea cu aplicația, fiind coerentă și eficientă.

**4. Implementarea și dezvoltarea aplicației**

**4.1 Integrările de bază**

Pornind de la zero, construcția aplicației NovaPlanner a urmărit pe tot parcursul procesului de dezvoltare o abordare incrementală și modulară, bazată pe structura stabilită cu separare clară dintre componentele sale principale. Proiectul a fost planificat având în vedere cerințele funcționale, stabilindu-se de la început că aplicația va fi una web, modernă, cu o interfață prietenoasă și capabilă să ofere suport pentru managementul zilnic al task-urilor, obiectivelor și activităților recurente, păstrând, însă, flexibilitate și adaptabilitate în dezvoltare. Funcționalitățile au fost dezvoltate rând pe rând, însă pasul cu care am început, piatra de temelie a întregii aplicații, a constat în modul în care am conectat componentele între ele.

În prima etapă am generat soluția în Visual Studio 2022, folosindu template-ul IDE-ului pentru o aplicație fullstack cu un proiect React pentru frontend și un proiect ASP.NET Core pentru backend. Acesta a definit automat directoarele pentru controlere, modele și servicii. Apoi, primul pas a reprezentat crearea configurației pentru MongoDB, realizată prin definirea unui model de configurare în *MongoDBSettings*, care conține string-ul de conexiune, numele bazei de date și denumirile colecțiilor din interiorul său. În *appsettings.json* am atribuit în mod clar valori fiecărui membru al modelului, urmând să le injectăm mai apoi în fișierul care gestionează această parte a backend-ului. Accesul se centralizează, astfel, în *MongoDBService*, care preia rolul de a crea conexiunea unică și stabilă cu baza de date. Fișierul folosește driverul oficial MongoDB pentru .NET, instanțiind un obiect MongoClient pe baza string-ului de conexiune local (adică cel default, pe portul 27017). Serviciul accesează apoi ”OrganizerDb” și oferă metoda generică *GetCollection<T>* prin care facem legătura între colecția din bază corespunzătoare cu valoarea din appsettings. Astfel, avem un model prin care celelalte servicii ale aplicației pot lucra direct cu datele din Mongo, fără a mai fi nevoie de a repeta logica de conectare în mai multe locuri.

În paralel, template-ul a inițializat și proiectul frontend cu Vite, soluția rapidă de build pentru aplicații React. La configurarea inițială am adăugat instanța personalizată Axios, prin care definim adresa de bază către backend (http://localhost:5211/api, tot pe portul default pe care template-ul a creat proiectul backend), permițând trimiterea de cereri HTTP către API-ul .NET Core. Astfel, se stabilește conexiunea dintre cele cele două componente principale ale proiectului, care pot comunica între ele pentru a obține sau trimite date.

**4.2 Autentificarea**

Înainte de a crea o interfață pentru autentificarea în aplicație, am început prin implementarea funcționalităților pe partea de backend. Fiind un aspect esențial al acestei aplicații web, accentul a fost pus pe crearea unui sistem sigur, scalabil și eficient. Acesta a fost implementat folosind *JSON Web Tokens(JWT)*, care este un standard în transmiterea securizată a informațiilor între client-server. La autentificare, serverul generează un token criptat cu o cheie secretă ce conține informații despre utilizatorul curent, fiind apoi transmis către client și stocat web în localStorage. Modificările aduse ulterior în *AxiosInstance* ne permit să atașăm în frontend acest token în mod automat la fiecare cerere ulterioară. Prin configurarea acestui interceptor, utilizatorul râmâne autentificat între sesiuni și poate accesa rutele protejate de sistemul de autentificare fără să reintroducă datele de login.

Pe partea de frontend, fluxul de autentificare este dezvoltat elegant și eficient în cadrul componentelor React. Pentru login există fișierul *Login.jsx*, care se ocupă de logica din interfață. Acesta folosește starea locală pentru a gestiona inputurile introduse de utilizator și *useNavigate* din *react-router-dom* pentru a redirecționa automat după autentificare. Funcția asincronă *handleLogin* este cea care se ocupă de trimiterea cererii HTTP POST către backend la trimiterea formularului, prin apelarea metodei *login* din *UserService*. Imediat după autentificare, se navighează spre dashboard fără a reîncărca pagina. Pentru înregistrare, *Register.jsx* oferă un formular complet de creare a contului, cu validare pentru formatul e-mail-ului implementată printr-un regex și mesaje de feedback pentru erori. *handleRegister* este funcția asincronă care se apelează metoda *register* din serviciul de frontend, care trimite HTTP POST către backend. Odată ce înregistrarea este completată cu succes, utilizatorul este redirecționat spre pagina de login. *ForgotPassword.jsx* și *ResetPassword.jsx* reprezintă fișierele care susțin procesul de resetare a parolei. Prima componentă oferă utilizatorului un formular simplu în care își poate introduce adresa de e-mail asociată contului. La apăsarea butonului de trimitere a formularului, *requestPasswordReset* este apelată din *UserService*, care comunică cu partea de backend pentru generarea token-ului de resetare și trimiterea sa pe adresa introdusă. Mesajul care apare în interfață este generic și nu ține cont dacă e-mail-ul este existent în baza de date sau nu, lucru care respectă principiile de securitate prin evitarea expunerii existenței conturilor. A doua componentă al acestui flux conține pagina care este accesată prin link-ul primit pe adresa de e-mail, conținând un token JWT temporar în URL. Acesta este extras prin *useLocation*, iar utilizatorul poate introduce parola nouă, care e trimisă cu token-ul la backend prin metoda *resetPassword*. În interfață se afișează un mesaj în funcție de caz. Pe lângă acest mecanism de resetare a parolei, NovaPlanner oferă utilizatorului autentificat și opțiunea de a-și schimba parola din interfață. Fișierul *ChangePassword.jsx* furnizează formularul în care utilizatorul poate introduce parola sa curentă, parola nouă și confirmarea acesteia. După validarea locală care se asigură că valorile ultimelor două câmpuri sunt identice, se trimite apelul POST la backend către ruta protejată corespondentă, fiind necesar token-ul JWT pentru autorizare.

De toată logica autentificării se ocupă serviciul pentru utilizator. Clasa *UserService* gestionează înregsitrarea, login-ul și cele două moduri prin care utilizatorul își poate reseta parola. La crearea unui cont în aplicație, datele de înregistrare sunt salvate în Mongo prin intermediul modelului *User*. În controlerul corespondent, endpoint-ul primește un obiect JSON cu numele de utilizator, e-mail și parolă. El apelează serviciul pentru validare și verifică existența utilizatorului pe baza e-mail-ului și a username-ului, ambele fiind obligatoriu unice. Parola este criptată cu algoritmul SHA-256, iar hash-ul este stocat în baza de date. În caz de succes sau eșuare a creării utilizatorului, se afișează un mesaj aferent cazului transmis de răspunsuri.

La autentificare, endpoint-ul primește identificatorul, care poate fi username-ul sau e-mail-ul, și parola. Parola introdusă de utilizator e criptată din nou, iar hash-ul obținut se compară cu cel stocat pentru username-ul și e-mail-ul respectiv, pentru a confirma validitatea datelor de înregistrare. Dacă verificarea este validă, utilizatorul primește un mesaj de succes și un token JWT valabil timp de o oră se generează și se semnează cu cheia secretă din fișierul de configurare prin algoritmul HMAC-SHA256, folosit în general pentru astfel de semnături digitale. El conține revendicări (*claims*), anume ID-ul, username-ul și email-ul, esențiale pentru identificarea utilizatorului de către server. Setările pentru JWT sunt gestionate de modelul nepersistent *JwtSettings*, care conține informații pentru cheia secretă, emițătorul (definit în serviciu ca ”OrganizerApp”) și timpul de expirare. În cazul contrar în care autentificarea eșuează, utilizatorul este înștiințat printr-un mesaj care apare în interfață.

Pentru schimbarea parolei, utilizatorul are cele două opțiuni: cazul în care dorește să o modifice manual sau cazul în care a pierdut accesul la aceasta. În primul caz, endpoint-ul, de data aceasta protejat, extrage din revendicările JWT *userId*, prin *HttpContext.User*. Când utilizatorul trimite parolele în aplicație, controlerul apelează serviciul, care, prin operația *ChangePasswordAsync* verifică dacă parola actuală este corectă, criptează parola nouă și o înlocuiește pe cea veche în baza de date cu noul hash. În funcție de răspunsul primit pe urma validărilor, se trimite un mesaj pentru utilizator ca să-l înștiinteze de reușită sau de eroare. În cel de al doilea caz, apelul POST la ruta pentru uitarea parolei primește email-ul introdus de utilizator. Dacă acesta există în baza de date, serverul generează un token aleator generat criptografic care expiră în timp de o oră, îl atribuie ca valoare membrului *PasswordResetToken* din model și îl trimite printr-un link de resetare pe e-mail-ul introdus. Ora trimiterii este și ea salvată în *ResetTokenExpiry*. Aceste informații permit validarea și expirarea corectă a token-ului pentru a evita posibile breșe de securitate. E-mail-ul este trimis prin *SmtpClient* care conține date configurate în User Secrets: adresa expeditorului, parola, hostul SMTP și portul pentru securizarea SSL. Utilizatorul poate accesa link-ul, iar în cazul în care token-ul este valid și nu a expirat, noua parolă este criptată și salvată în locul celei vechi; în cazul contrar se afișează mesajul de eroare aferent.

Toate aceste procese sunt implementate în mod asincron pentru a menține performanța server-ului, iar rutele sensibile sunt protejate cu atributul [Authorize] pentru a avea siguranța că doar utilizatorii autentificați pot accesa datele, iar toate măsurile luate respectă bunele practici de securitate.

**4.3 Task-urile**

Componenta principală de frontend care se ocupă de gestionarea logicii pentru task-uri este chiar *Dashboard*. Ea reprezintă punctul focal al întregii aplicații, fiind concepută ca o interfață dinamică și intuitivă prin care utilizatorul gestionează toate tipurile de sarcini într-un mod organizat și cu distincții clare. Interfața prezintă patru tab-uri care duc spre fiecare tip principal de task: ”dailies” (sarcinile zilnice), ”to do” (task-uri generale), ”habits” (menite să ajute utilizatorul spre stabilirea unei rutine) și ”goals” care conține și tipul ”checkpoint” (obiective setate de către utilizator prin care își poate urmări progresul în atingerea unei ținte). Fiecare pagină conține cartonașe aparținând fiecărui task adăugat cu informațiile principale aferente, un buton care deschide pop-up-ul de adăugare a unei noi sarcini și un buton prin care se apelează asistentul inteligent. În bara de navigare, alături de tab-uri există și un buton de tip ”hamburger menu”, care îi oferă utilizatorului opțiunea de a accesa pagina de schimbare a parolei și opțiunea de logout din NovaPlanner.

Dincolo de aspectul vizual, logica de gestiune a frontend-ului este organizată în jurul a mai multor stări, gestionate prin hook-urile React. Fiecare stare are un rol specific în controlul comportamentului interfeței și în organizarea datelor afișate, contribuind astfel la o experiență fluentă și personalizată. Avem *activeView*, a cărui valori corespund categoriilor de task-uri menționate de tab-uri, valoarea implicită fiind setată pentru ”dailies”, reflectând presupunerea că utilizatorii își planifică în principal sarcinile zilnice. Atunci când starea aceasta este schimbată, este afișată lista respectivă de task-uri fără a mai reîncărca pagina.

Pentru încărcarea datelor, stările *tasks, loading* și *error* lucrează împreună pentru a administra fluxul de date provenit de la server. Prima stare conține o listă completă a task-urilor, *loading* este cea care semnalează că aplicația se află într-un proces de așteptare, iar *error* asigură posibilitatea de a rezolva eventualele probleme apărute în comunicarea cu backend-ul. Gestionarea eficientă este susținută de funcția *fetchTasks*, care trimite o solicitare GET la backend și normalizează fiecare task prin copierea id-ului într-un câmp *\_id* corespondent. În cazul erorii, aplicația actualizează starea *error* cu mesajul informativ de eroare care se va afișa în interfață. După obținerea datelor, obiectul *filteredTasks* le împarte pe categoriile existente prin funcții de filtrare după tip și, în cazul task-urilor zilnice, după data selectată în filtrul prezent în pagină (*selectedDate*). Acest sistem modular permite utilizatorului să comute între diferite tipuri de activități schimbând doar tab-ul și fără a încărca din nou datele din rețea, ceea ce contribuie semnificativ la performanță și uzabilitate.

Utilizatorul poate crea sau edita sarcini printr-un formular de tip pop-up. Există două funcții care să separe aceste două situații: *openAddPopup*, care ințializează pop-up-ul pentru modul de creareși *openEditPopup*, care îl setează în modul de editare, completând toate inputurile cu datele sarcinii selectate. În acest caz, aplicația comută automat *activeView* în funcție de tipul sarcinii ca fallback pentru a evita posibilitatea de suprascriere a tipului unei înregistrări. Afișarea formularului este controlată de starea *showPopup*, iar pentru a marca clar diferența între modul de creare și cel de editare ne folosim de starea booleană *isEditing*. Actualizarea datelor este susținută prin *handleChange*, care reflectă în starea locală exprimată de *currentTask* fiecare modificare din interfață. În momentul trimiterii formularului (fapt gestionat de *handleSubmit*), aplicația construiește un obiect *payload* cu informațiile necesare, incluzând validări contextuale. Spre exemplu, dacă tipul sarcinii este ”daily”, aplicația verifică dacă suma orelor estimate pentru toate task-urile zilnice din acea zi nu depășește 24 de ore, fiind o măsură importantă pentru a preveni supraîncărcarea și a încuraja planificarea realistă. După trimiterea datelor la backend cu apeluri POST sau PUT care depind de context, aplicația apelează din nou *fetchTasks* pentru a reflecta modificările, resetează formularul și închide pop-up-ul. Tot în *handleSubmit* se gestionează și cazul special creat de tipul ”checkpoint”, care face parte dintr-o ierarhie împreună cu tipul ”goal”. Funcționalitatea aceasta este marcată de prezența lui *parentGoalId*, care, atunci când este setat, interfața consideră automat că task-ul creat sau editat este etapă a obiectivului ”părinte”. Ștergerea este gestionată de *handleDelete*, o funcție simplă care oferă un mecanism de confirmare printr-o fereastră de confirmare și trimite un apel DELETE spre server, dar care gestionează și ștergerea de tip cascade pentru obiective și etape. Pe lângă toate acestea, am implementat și un mecanism de actualizare rapidă pentru bifarea unui task ca fiind completat și pentru creșterea streak-ului pentru rutine prin *handleQuickUpdate*. Acest mod de a trimite doar câmpurile modificare prin apel PUT optimizează performanța fiindcă evită trimiterea întregului obiect, în concordanță cu principiile reactive.

Randarea taskurilor este realizată dinamic pe baza datelor prefiltrate. Se folosește componenta *ToDoList*, care la rândul ei conține componentele *TaskItem* și *GoalWithCheckpoints* în funcție de *activeView*, fiindu-i transmise funcții de manipulare a datelor ca parametri: *onEdit*, *onDelete* și *onQuickUpdate*. Fiecare dintre cele două componente primește parametri pentru obiectul sau obiectele ce trebuie afișate, operații CRUD și tipul de task redat, iar comportamentul lor este ramificat condițional în funcție de *activeView* pentru a decide ce informații trebuie să apară în cartonaș. Ele includ și o secțiune de acțiuni care găzduiesc butoanele pentru operațiile CRUD. *ToDoList* mapează apoi fiecare task pentru a afișa componenta corespunzătoare, având astfel o logică separată și bine definită.

La fel ca și cartonașele cu task-uri, și inputurile din pop-up sunt afișate dinamic în funcție de tip. Randăm aceeași componentă de tip ”form” din *React Bootstrap* și pentru creare și pentru editare, așa că și titlurile și denumirea butonului sunt afișate dinamic în funcție de caz. Prin această modularitate păstrăm codul clar și curat, simplificând mentenanța și oferind compatibilitate cu toate tipurile de task-uri.

În backend, de implementarea fluxului se ocupa serviciul, controlerul și modelul corespondent. Modelul *TaskItem* conține toate câmpurile necesare pentru toate tipurile de task-uri, însă unele admit valori nule pentru a putea salva doar informațiile de care avem nevoie pentru fiecare tip. Clasa *TaskService* este cea care se asigură de logica de acces și de manipulare a datelor asociate task-urilor în MongoDB. Serviciul extrage informațiile din baza de date și identitatea utilizatorului din claims cu Dependency Injection. Extrage apoi ID-ul acestuia prin *GetUserIdFromClaims* și utilizarea *ClaimTypes.NameIdentifier* pentru siguranța că modificările care apar sunt doar asupra propriilor date. *GetByTypeAsync* preia din baza de date și filtrează task-urile după tip pentru utilizatorul identificat. La adăugarea unui task nou, serviciul adaugă automat ID-ul utilizatorului la obiectul *TaskItem* și, în funcție de tipul task-ului, anumite câmpuri vor fi inițializate (pentru ”habit” streak-ul și pentru ”goal” progresul vor primi ambele valoarea 0 inițial dacă nu sunt definite). Editarea se face într-un mod flexibil fiindca sunt actualizate doar câmpurile care au fost modificate. Acest lucru este gestionat printr-o listă de *update definitions*, care sunt combinate într-o singură operație atomică. Se evită în acest mod înlocuirea accidentală a datelor care nu au fost modificate. Ștergerea este gestionată de *DeleteAsync*, unde, deși în interfață utilizatorul își vede doar propriile task-uri, am implementat un fallback simplu care permite ștergerea doar după verificarea că acesta aparține lui, în funcție de ID. Concomitent, *TasksController* este punctul de acces REST care face legătura cu frontend-ul. Folosim și aici *[Authorize]* pentru protecția rutelor și apeluri GET (pentru returnarea task-urilor), POST (pentru adăugarea unui task nou), PUT (pentru actualizarea unui task) și DELETE (pentru ștergerea unei înregistrări).

**4.4 Asistentul AI**

În contextul unei aplicații moderne de tip task manager, simpla oferire a unei interfețe pentru adăugarea, editarea și filtrarea sarcinilor nu mai este suficientă pentru a sprijini utilizatorul în mod real în organizarea eficientă a timpului.Pentru a depăși acest nivel, aplicația NovaPlanner implementează un element inovator, anume asistentul inteligent bazat pe Semantic Kernel, un kit de dezvoltare open-source de la Microsoft, în combinație cu modelul lingvistic GPT-4. ”Nova”, așa cum l-am denumit în interfață, este capabil să analizeze datele istorice de productivitate și să ofere sugestii proactive privind planificarea zilnică.

Ce face posibilă interacțiunea cu modelul GPT-4 pe care l-am creat în Azure AI Foundry este clasa *SemanticKernelService*, care se ocupă de conectarea aplicației la Azure OpenAI. La inițializare, serviciul preia valorile de configurare (endpoint-ul, cheia și deployment-ul) din User Secrets, construind o instanță pentru obiectul *Kernel* pe care o expune printr-o proprietate publica astfel încât să fie utilizată de alte componente.

Activat în *Dashboard* prin funcția *handleEvaluate*, odată apelat prin butonul din UI extrage sarcinile zilnice curente și le transmite către un endpoint dedicat prin apel POST împreună cu ID-ul utilizatorului, așteptând feedback. *AssistantController* primește cererea și extrage istoricul task-urilor zilnice din ultimele 30 de zile prin metoda *GetDailiesForUserAsync* din serviciul pentru task-uri. Se agregă apoi informațiile la un set de KPI zilnici care conțin timpul estimat total pentru activitățile dintr-o zi, numărul de task-uri duse la final și numărul de task-uri rămase incomplete. Aceste KPI-uri sunt formatate într-un text coerent cu ajutorul plugin-ului *TaskAnalysisPlugin*, iar controlerul îl preia de acolo și construiește un prompt pe care îl transmite modelului prin metoda *InvokePromptAsync.* Se primește un feedback de la LLM sub forma unui text generat pe baza informațiilor transmise. Feedback-ul primit este trimis apoi spre frontnend, afișându-l pentru utilizator în timp real.

Această întreagă dezvoltare bazată pe Semantic Kernel este cea mai avantajoasă fiindcă presupune adaptabilitate la datele utilizatorului prin crearea acestor KPI-uri bazate pe obiceiurile sale personale, prompturile sunt dinamice și semantice în locul unor reguli rigide de validare, iar modularitatea și scalabilitatea ies în evidență prin faptul că kernel-ul este configurat într-un serviciu izolat, deci se pot face modificări ulterioare fără să fie afectat restul codului.

**4.4 Stilizarea**

Stilizarea urmărește îndeaproape tematica propusă pentru intefață. Toate alegerile cu privire la estetica vizuală au fost luate în mod deliberat pentru a susține stilul ales. ”Frutiger Aero” are anumite caracteristici estetice care îl fac ușor de identificat: suprafețe lucioase și transparente, paletă de culori bazată predominant pe nuanțe de verde și albastru, motive acvatice și bazate pe bule de săpun și un simț tehnologic inclinat spre futurism. Pe astfel de elemente se bazează și interfața creată pentru NovaPlanner. Butoanele, bara de navigație și cartonașele de task-uri sunt rotunjite și lucioase, unele fiind chiar perfect circulare. Umbrele și gradientele setate prin fișierele CSS conferă impresia unei bule de săpun pentru butonul de ștergere, cel de editare, cel de adăugare a unui *checkpoint* și cel de apelare a asistentului AI. Pictogramele alese sunt reprezentative pentru acest stil, pe fundal având, alături de un gradient pastelat cu accente verzi și albastre, o fotografie editată care inspiră exact tematica principală ”frutiger aero”: armonia dintre tehnologie și natură. Pe toate paginile există un efect vizual prin cale bule plutesc de jos în sus, creat cu ajutorul pseudoelementelor *::before*  și *::after* și a regulii CSS *@keyframes*, folosită pentru a controla pașii într-o secvență de animație. Aproape toate componentele conțin un efect de *hover*, care creează un aspect plăcut și prietenos în interfață. Stilizarea standard pentru inputuri a fost suprascrisă manual cu alte valori pentru a susține estetica vizuală, păstrând aceeași paletă de culori și același stil lucios și rotunjit. Iconițele Bootstrap au fost alese în detrimentul altora pentru simplicitate și minimalism, fiind personalizate prin diferite atribute pentru culoare. Animațiile la deschiderea pop-up-ului sau la navigarea între tab-uri au fost realizate prin integrarea bibliotecii Framer Motion, creânnd tranziții line și efecte vizuale care sporesc percepția de fluiditate a aplicației.

Anexa 2 prezintă o captură de ecran cu dashboard-ul aplicației, unde se poate observa vizual interfața.

**5. Concluzii**

În final, lucrarea de față demonstrează potențialul semnificativ al programării reactive în contextul dezvoltării de aplicații web moderne, evidențiind avantajele accestei paradigme în construirea unui sistem fluid, scalabil și interactiv. Aplicația NovaPlanner, subiectul central al lucrării, a servit drept exemplu practic și concret pentru integrarea fluxurilor de date reactive între frontend-ul construit cu React.js și backend-ul ASP.NET Core, sprijinit de o bază de date MongoDB și de un asistent AI integrat prin Semantic Kernel. Toate dezvoltările create au avut ca scop scoaterea în evidență a comportamentului adaptiv și al interfeței prietenoase care reflectă instantaneu schimbările de stare, totul contribuind la o experiență de utilizator modernă și eficientă. Prin arhitectura și toate tehnologiile adoptate, proiectul reușește să valideze conceptul de reactivitate și demonstrează cum poate fi transpusă această paradigmă într-un sistem concret de management al task-urilor. De asemenea, integrarea inteligenței artificiale aduce un plus de valoare aplicației prin menținerea sa la un nivel al domeniului apropiat de cel curent și prin crearea experienței personalizate și adaptive.

Concluzionăm astfel că lucrarea de licență își atinge obiectivul propus de a ilustra aplicabilitatea și beneficiile programării reactive, atât din perspectiva dezvoltării software, cât și din cea a experienței utilizatorului. Prin proiectul creat se demonstrează cum un design atent și o arhitectură bine gândită transformă un concept teoretic într-un produs funcțional și relevant, capabil să răspundă cerințelor utilizatorilor moderni. Rezultatele obținute în urma creării NovaPlanner confirmă faptul că programarea reactivă reprezintă nu doar un trend tehnologic, ci și o direcție viabilă pentru construirea de aplicații web robuste, performante și ușor de întreținut.