XXXX

add title page

(from regulations)

XXXX

add annex 1 xor 2 (formular de inscriere a temei lucrarii)

(from regulations with signatures, scan)

XXXX

add Annex 6/6bis - copyright declaration

(from regulations with signature, scan)

XXXX

add Annex 5/5bis - academic honesty declaration.

(from regulations with signatures, scan)

Cuprins

[Introducere 19](#_Toc43461887)

[1 Introducere in metodologia agilă 21](#_Toc43461888)

[1.1 Metodologia de lucru agil 21](#_Toc43461889)

[1.2 Platforme agile 22](#_Toc43461890)

[1.2.1 Jira 22](#_Toc43461891)

[1.2.2 Bitrix24 22](#_Toc43461892)

[1.2.3 ClickUp 23](#_Toc43461893)

[2 Instrumente software folosite 24](#_Toc43461894)

[2.1 Baza de date 24](#_Toc43461895)

[2.1.1 MySQL 24](#_Toc43461896)

[2.2 Serverul 24](#_Toc43461897)

[2.2.1 Java 24](#_Toc43461898)

[2.2.2 SpringBoot 24](#_Toc43461899)

[2.2.3 Hibernate 25](#_Toc43461900)

[2.3 Interfața grafică 26](#_Toc43461901)

[2.3.1 JavaScript, HTML & CSS 26](#_Toc43461902)

[2.3.2 ReactJS 26](#_Toc43461903)

[3 Interfața grafică 27](#_Toc43461904)

[3.1 Structura fișierelor 27](#_Toc43461905)

[3.2 Starea globală a aplicației 29](#_Toc43461906)

[3.3 Punctele de intrare ale aplicației 30](#_Toc43461907)

[3.4 Securizarea aplicației 31](#_Toc43461908)

[3.5 Structura generală a paginilor 35](#_Toc43461909)

[3.5.1 AppLayout 35](#_Toc43461910)

[3.5.2 NavBar 36](#_Toc43461911)

[3.5.3 Footer 37](#_Toc43461912)

[3.6 Pagina de start 38](#_Toc43461913)

[3.7 Pagina proiectelor personale 38](#_Toc43461914)

[3.7.1 Afișarea proiectelor 38](#_Toc43461915)

[3.7.2 Managementul sarcinilor de lucru într-un proiect 38](#_Toc43461916)

[3.8 Pagina sarcinilor de lucru personale 38](#_Toc43461917)

[3.9 Pagina notițelor personale 38](#_Toc43461918)

[3.10 Pagina de raportare 38](#_Toc43461919)

[4 Serverul 38](#_Toc43461920)

[Concluzii 39](#_Toc43461921)

[Bibliografie 41](#_Toc43461922)

[Annex 1 43](#_Toc43461923)

[Annex 2 45](#_Toc43461924)

List of figures

[Figura 4.1: image template ro - XXXX schimba titlul 35](#_Toc12795531)

[Figure 4.1: image template eng XXXX change title 35](#_Toc12795532)

List of tables

Tabelul 4.1: table template ro XXXX schimba titlul 35

Table 4.1: table template eng XXXX change title 35

List of equations

[Ecuaţia 4.1: equation template ro XXXX schimba titlul 36](#_Toc12795535)

[Equation 4.1: equation template eng XXXX change title 36](#_Toc12795536)

List of acronyms and abbreviations

API – Application Programming Interface

CSS – Cascading Style Sheets

HTML – Hyper Text Markup Language

HQL – Hibernate Query Language

JDBC – Java Database Conectivity

JSX – JavaScript XML

REST –Representational state transfer

Npm – Node Package Manager

DOM – Document Object Model

# Introducere

Lucrarea de față își propune să proiecteze o platformă web care să poată gestiona proiectele, sarcinile de lucru aferente, facilitarea unui bun control al tuturor necesităților tehnice și facilitarea unei comunicări eficiente între conducere, angajați și elevi.

Alegerea temei a fost influențată de importanța unei comunicări eficiente în cadrul unei echipe, în cadrul unui departament și în cadrul unei organizații în general. Pentru a obține o comunicare eficientă, în facultatea noastră, consider că este nevoie, în primul rând, de o modalitate ușoară și accesibilă tuturor, în special studenților, pentru a raporta problemele întâmpinate către responsabilii facultății. În al doilea rând este nevoie de un sistem de gestiune a tuturor departamentelor, proiectelor și sarcinilor de lucru care să permită o mai bună viziune de ansamblu, atât pentru cei care se ocupă cu mentenanță, cât și pentru conducerea facultății.

Urmând câteva idei și principii din metodologiile de lucru ale echipelor de dezvoltatori, am dorit ca aplicația să permită șefilor de departament sau conducerii, să asigneze câteva sarcini de lucru, pentru angajații care se ocupă de mentenanță sau de partea tehnică. Aceștia, la rândul lor, vor putea să înregistreze progresul făcut în cadrul unei sarcini, să introducă un număr de ore lucrate, să aproximeze numărul de ore necesare pentru îndeplinirea sarcinii. Toate aceste detalii vor fi vizibile pentru manager. Astfel, aplicația va oferi o viziune clară de ansamblu, asupra modului în care evoluează un anumit proiect sau sarcina de lucru, ceea ce va permite o mai buna prioritizare și eventuale întrajutorări intre membrii echipei.

Pe lângă posibilitatea de management și gestiune al proiectelor și sarcinilor de lucru, aplicația trebuie să permită studenților facultății să raporteze anumite probleme sau nereguli întâmpinate în timpul folosirii oricărei resurse puse la dispoziție de facultate. Imediat ce este raportată problema, un mail cu conținutul mesajului studentului ar trebui să fie trimis automat către șeful departamentului, acesta trebuind să verifice problema și să decidă mai departe ce este de facut.

Platforma finala va conține 2 sub-aplicații, cu funcționare distinctă, dar care vor interacționa în strânsă legătură. Pe deoparte va exista aplicația server, numita si backend, ce are rolul de a interacționa cu baza de date, prin operații de tip creare, update sau ștergere. Serverul va trebui să pună la dispoziție astfel câteva puncte finale, care atunci când sunt invocate, să producă modificările dorite asupra bazei de date.

Pe de altă parte va exista interfața cu utilizatorul care sa permită modificarea bazei de date si a informațiilor din aceasta într-un mod facil si prietenos pentru utilizatorul de rând. Interfața cu utilizatorul, prin acțiunile efectuate de către utilizator va invoca acele puncte finale ale serverului, pentru crearea, modificarea sau ștergerea unei informații din baza de date.

În ceea ce privește tehnologiile folosite, aplicația va folosi MySQL ca sistem de gestiune al bazei de date, un server care să gestioneze interacțiunea dintre baza de date și interfața cu utilizatorul construit folosind Java ca limbaj de programare, împreună cu instrumentele specializate de dezvoltare Spring Boot și Hibernate. Interfața cu utilizatorul va fi elaborată folosind ReactJS, o librărie specială de Java Script care permite crearea de GUI-uri dinamice întru-un mod facil. Pe lângă Java Script, se vor folosi JSX, un limbaj descriptiv foarte asemănător cu HTML, dar și CSS pentru stilizare si aranjare in pagină.

# Introducere in metodologia agilă

## Metodologia de lucru agil

Metodologia de lucru agil este un model de gestionare si organizare a unui proiect de tip software, care susține o dezvoltare și reglementare din aproape in aproape. Practic o aplicație întreaga este spartă în probleme mai mici, ce trebuie tratate pe rând in vederea livrării celui mai bun produs posibil.

În dezvoltarea aplicațiilor software, se întâmplă frecvent să apară întârzieri datorită problemelor neprevăzute ce pot apărea sau datorită modificării dorințelor clientului. Acestea din urma apar deoarece este dificil la început să se descrie cu exactitate toate cerințele unei aplicații. De aceea sunt necesare întâlniri recurente intre dezvoltatorii aplicației si client pentru a discuta eventuale probleme întâmpinate si direcțiile de urmat pe viitor.

Metodologia agilă de lucru promovează împărțirea unei probleme în subprobleme care să fie apoi planificate pe durate scurte, pentru a putea trata cât mai bine fiecare impediment întâmpinat. Planificând un proiect doar pe termen scurt permite o flexibilitate foarte mare în ceea ce privește dezvoltarea aplicației. Aceste planificări pe termen scurt se împart in cicluri de dezvoltare numite sprinturi sau iterații. De regula la începutul fiecărui sprint, există un număr de ședințe fixate, în vederea planificării sarcinilor de lucru. Acestea se negociază intre dezvoltatori si client in funcție de dificultatea sarcinilor de lucru, timpul alocat, dar și în funcție de bugetul disponibil.

|  |
| --- |
|  |
| Figura 1.1Metodologia de lucru agil |

De regulă când vine vorba despre etapele de lucru ale metodologiei agile, putem împărții tot procesul in 6 pași:

* Discutarea cerințelor clientului cu dezvoltatorii și stabilirea specificațiilor aplicației de comun acord.
* Planificarea bugetului si perioadei de dezvoltare din punct de vedere al cantității de lucru dar și a timpului.
* Crearea arhitecturii, logicii interne și realizarea unor scheme de urmărit pentru dezvoltarea interfețelor de lucru cu utilizatorii, la nivel conceptual.
* Dezvoltarea propriu zisă, urmând planul de lucru stabilit.
* Testarea aplicației sau a modificărilor aduse dea lungul sprintului și darea in folosință a noii versiuni.
* Ședințe cu scopul de a analiza ce a fost bine, ce a fost mai puțin bine, ce se poate schimba și cum. Toate acestea se bazează pe monitorizarea din timpul sprintului, și au în vedere îmbunătățirea modului in care lucrează echipa si eventuale ajustări pentru viitor.

## Platforme agile

### Jira

Jira este probabil cel mai utilizat instrument software de monitorizare a progresului unui proiect, folosit de către echipele de dezvoltatori de aplicații care lucrează în sistem agil. Dintre facilitățile oferite de către Jira reamintim:

* Planificarea sprintului, incluzând aici descrierea sarcinilor de lucru precum și împărțirea lor între membrii echipei.
* Urmărirea progresului fiecărei sarcini de lucru, în vederea prioritizării lor sau pentru întrajutorarea între membrii echipei.
* Urmărirea progresului pentru întregul proiect în vederea livrării unui produs finit cât mai apropiat de cerințele clientului.
* Crearea de rapoarte în timp real sau la final de sprint pentru îmbunătățirea performanței echipei și pentru ajustarea sarcinilor de lucru pe viitor.
* Integrarea cu alte instrumente de dezvoltare deja bine cunoscute și folosite la scară larga în întreaga lume, cum ar fi BitBucket sau Confluence.

În ceea ce privește prețurile Jira este gratis pentru o echipa de până la 10 persoane, având la dispoziție doar anumite facilități. În varianta gratis Jira oferă doar 2 GB de stocare online și nu pune la dispoziție suport direct, permisiuni avansate sau acces anonim, iar integrarea este permisă doar pentru anumite instrumente, contra cost. Varianta plătită oferă suport 24/7, spațiu de stocare nelimitat online, acces anonim, securitate sporita, informații speciale pentru administrator, permisiuni avansate si multe altele.

### Bitrix24

Bitrix24 este o soluție de management al proiectelor, sarcinilor si sub-sarcinilor de lucru. Oferă suport pentru un număr nelimitat de proiecte. Oferă facilități asemănătoare cu cele mentionate în cadrul punctului 1.1, dintre care reamintim: urmărirea progresului pentru sarcinile de lucru si pentru proiecte, managementul documentelor aferente unui proiect, instrumente de colaborare intre membrii echipei, precum mesagerie instant videoconferință. Este posibilă stocarea informațiilor atât online cat si pe propriul server.

Bitrix24 este o soluție gratis pentru echipe de maxim 12 persoane. În cazul în care echipele sunt mai mari, este disponibil un abonament lunar de 99$ pentru un număr nelimitat de persoane.

În plus Bitrix24 este o soluție open source code și open API.

### ClickUp

ClickUp este un alt instrument de management și gestionare pentru proiecte și sarcini de lucru, care în varianta gratis oferă un număr nelimitat de utilizatori, număr nelimitat de proiecte și sarcini de lucru, suport 24/7. Pe lângă aceste facilități, variantele plătite oferă spațiu de stocare nelimitat, pentru fișierele echipei, posibilitatea adăugării invitațiilor care să aibă permisiuni temporare și restrânse, setarea permisiunilor avansate, dar și posibilitatea alegerii și schimbării paletei de culori, precum și a modului de aranjare in pagina.

# Instrumente software folosite

## Baza de date

### MySQL

MySQL este un sistem de gestiune a bazelor de date relaționale, produs de compania suedeza MySQL AB și distribuit sub Licența Publică Generală GNU. Este cel mai popular SGBD open-source la ora actuală, fiind o componentă cheie a stivei LAMP (Linux, Apache, MySQL, PHP).[1]

Deși este folosit foarte des împreună cu limbajul de programare PHP, cu MySQL se pot construi aplicații în orice limbaj major. Există multe scheme API disponibile pentru MySQL ce permit scrierea aplicațiilor în numeroase limbaje de programare pentru accesarea bazelor de date MySQL, cum ar fi: C, C++, C#, Java, Perl, PHP, Python, FreeBasic, etc., fiecare dintre acestea folosind un tip specific API.[1]

În ceea ce privește aplicația noastră, folosind Java ca limbaj de programare pentru partea de server, vom avea nevoie de API-ul JDBC care oferă o interfață standard pentru conectarea și interacțiunea cu orice bază de date de tip relațional.

## Serverul

Dintre instrumentele software folosite in dezvoltarea serverului amintim:

* Java – Limbaj de programare
* SpringBoot – framework de Java folosit in configurarea întregii aplicații de server
* JPA – Java Persistence API, folosit pentru interacțiunea cu baza de date

### Java

Java este pe deoparte un limbaj de programare de nivel înalt, orientat pe obiecte, similar cu C++, dar care oferă facilități mai avansate și mai usor de folosit. În plus Java este independent de platformă, orice aplicație sau cod Java compilat poate rula pe toate platformele, cu ajutorul JVM, având astfel avantajul portabilității.

Este un limbaj de programare foarte sigur deoarece tot codul Java este ulterior transformat in bytecode, acesta din urma fiind imposibil de înțeles de către un om.

### SpringBoot

SpringBoot este un framework Java, open-source, care facilitează crearea de micro-servicii pentru dezvoltatori. Micro serviciile sunt o arhitectura care ajuta la crearea și livrarea de servicii care lucrează independent. Practic fiecare serviciu rulează pe un proces separat și de sine stătător.

SpringBoot aduce cu sine multe avantaje dintre care reamintim:

* Configurații minime prin simpla adăugare a adnotațiilor
* Oferă un server deja integrat, ce nu mai necesita prea multe configurări
* Creează un mediu de dezvoltare care în timp este ușor de scalat
* Gestionare ușoară a dependințelor
* Procesare rapida si management facil pentru endpoint-urile de tip REST

### Hibernate

Hibernate este un instrument software folosit in maparea dintre obiecte si entitățile asociate dintr-o bază de date. Practic prin Hibernate se pot trata toate entitățile din baza de date, folosind clase, iar interacțiunea cu sau dintre acestea folosind metode.

Conectorul Java (JDBC) oferă o interfață standard pentru conectarea și interacțiunea cu orice bază de date de tip relațional, permițând scrierea de interogări direct în codul de Java. Totuși pentru simplificare și mentenanță acest gen de abordare nu este deseori preferat. Soluțiile au fost Hibernate Query Language (HQL), care este un limbaj de scriere de interogări asemănător cu SQL, dar si Java persistence API (JPA), acesta din urma fiind o implementare a Hibernate, care pune la dispoziția dezvoltatorilor de aplicații câteva metode deja implementate, facilitând interacțiunea cu baza de date.

## Interfața grafică

### JavaScript, HTML & CSS

JavaScript este un limbaj de programare, cu o sintaxa asemănătoare cu cea a limbajului C, orientat pe obiect, folosit împreună cu HTML(Hyper Text Markup Language) și CSS(Cascading Style Sheets) în dezvoltarea aplicațiilor web. Acesta este folosit pentru a defini anumite funcționalități în pagina creată. Prin JavaScript se implementează practic o pagina web dinamică și interactivă. Acțiuni precum apăsarea unui buton, afișarea unor informații la intrarea mausului într-o anumită zona sau trimiterea de formulare către server spre procesare, toate se realizează folosind capabilitățile oferite de JavaScript.

HTML este un limbaj descriptiv, care folosind taguri, se ocupa de prezentarea conținutului într-o structură organizată. Acesta oferă dezvoltatorilor web taguri pentru titluri, paragrafe, linkuri, butoane și multe altele. Pe lângă HTML este folosit CSS in descrierea modului in care pagina arată. Acesta reprezintă un limbaj cu rol în formatarea tagurilor dintr-un document HTML, folosind stiluri fie standardizate precum Boostrap, fie stiluri proprii create de dezvoltator.

Toate aceste 3 limbaje, sunt interdependente și sunt standardizate pentru orice browser web, fiind tehnologiile de baza in realizarea celor mai multe site-uri. Rolul fiecăruia, precum și legătura dintre ele se poate observa și în diagrama următoare:

|  |
| --- |
|  |
| Figura 2.3.1: Interdependența dintre JavaScript, HTML și CSS |

### ReactJS

ReactJS sau simplu React, este o librărie de JavaScript, open-source foarte folosită în lumea aplicațiilor web, datorită simplității pe care o oferă în crearea unor site-uri dinamice, ușor de scalat și de menținut.

Permite crearea de componente care își mențin propria stare internă. Orice schimbare a proprietăților sau a stării unei componente duce la reevaluarea structurii ce trebuie afișată. În acest fel este extrem de ușor să modularizăm aplicația, precum și să creăm o pagină web dinamică.

Logica de adaptare a stării interne sau a proprietăților unei componente este implementată folosind JavaScript, limbaj care oferă posibilitatea de lucru cu obiecte nenumărate și foarte complexe. Astfel este mult mai facilă crearea unei aplicații web de scară largă, menținând un cod curat și reutilizabil. De asemenea trecerea acestor aplicații web către platforme mobile este posibilă folosind React Native.

# Interfața grafică

## Structura fișierelor

Pe nivelul de top aplicația de frontend are următoarele fișiere:

* node\_modules
* public
* src
* .gitignore
* package.json
* package-lock.json
* README.md

Aplicația de frontend rulează peste un server de Node.js. Acesta este un server, open-source, ce oferă un mediu de execuție pentru programele JavaScript. Odată instalat Node.js, se poate folosi Node Package Manager pentru instalarea, respectiv ștergerea librăriilor din proiect. Folosind comanda „npm install [nume\_librărie]”, se pot adăuga toate librăriile dorite. La rularea acestei comenzi, managerul de pachete caută online librăria specificată și in cazul în care aceasta este găsită, este adăugată în fișierul node\_modules. Toate aceste dependințe necesare proiectului, sunt adăugate apoi automat în package.json, fișier ce conține toate informațiile relevante pentru managerul de pachete.   
 Package.json este un fișier generat automat, fiind situat în directorul rădăcină al proiectului și conține toate informațiile necesare managerului de pachete în gestionarea dependințelor, cum ar fi: numele, versiunea sau alte setări specifice librăriei dorite. Pe lângă informațiile de identificare a dependințelor, package.json mai poate conține și o descriere a proiectului, numărul versiunii curente sau comenzi predefinite de către dezvoltatori, prin pare se pot trimite către execuție diferite instrucțiuni sau chiar scripturi întregi.

Package-lock.json este de asemenea un fișier generat automat de către managerul de pachete, în care sunt descrise sub formă de arbore toate dependințele folosite. Prin acest fișier este descrisă o singură reprezentare a setărilor făcute pentru o librărie, prin care se asigură faptul că toți cei care lucrează la un proiect, au exact aceleași librarii cu aceleași versiuni.   
 Package-lock.json mai oferă și posibilitatea de revenire la versiuni anterioare ale librăriilor, în cazul in care după schimbarea versiunii se constată că rezultatul nu este cel dorit. Prin menținerea acestor informații se optimizează și procesul de înnoire a versiunii unei dependințe, managerul de pachete instalând doar ce este in plus față.

Fișierul README.md este generat automat la crearea aplicației inițiale de React. Acesta conține descrieri sumare pentru câteva comenzi disponibile pentru dezvoltatori, cum ar fi:

* npm start – folosită pentru pornirea serverului de dezvoltare pentru aplicația de frontend
* npm test – lansează in execuție testele unitare
* npm run build – pregătește aplicația pentru distribuție, optimizând procesele și aranjând toate fișierele necesare in directorul „build”

Fișierul .gitignore este creat automat la integrarea cu una din multele soluții disponibile pentru versionare cum ar fi GitHub, BitBucket, GitLab, etc. In acesta trebuie menționate toate fișierele ce nu se doresc a fi incluse în versionare. În cazul de față în .gitignore sunt menționate directoarele /node\_modules, /coverage și /build.

Directorul public este cel care conține toate fișierele ce pot fi accesate sau văzute direct în browser. Aplicația de React nu se poate vedea direct in browser sau in instrumentele pentru dezvoltatori. De aceea este necesar directorul /public. În acesta se regăsește singurul document de HTML autentic și anume index.html.  
 În index.html se afla rădăcina aplicației*: <div id="root"></div>.* Peste aceasta este afișată, prin intermediul React, fiecare componenta care trebuie să apară în pagină la un moment dat. ReactJS preia codul ce descrie o componenta și îl convertește într-un întreg arbore HTML numit si DOM( Document Object Model).

În directorul /src se află fiecare componenta de React creată și fiecare funcție de procesare a datelor aduse din server. Dintre fișierele importante ale acestui director amintim:

* index.js – punctul de intrare al aplicației de React.
* app – director care conține fișierul de rutare.
* store.js – fișierul care creează un container global pentru starea aplicației.
* history.js – face posibila navigarea utilizatorului între diversele rute ale aplicatiei.
* utils – director ce conține funcții și componente necesare in mai multe zone ale aplicatiei, cum ar fi tabelul de prezentare adaptabil in funcție de structura informației, diverse ferestre de dialog, etc.
* shared – în acest director sunt descrise componentele comune tuturor paginilor aplicației precum structura generală și încadrarea elementelor, bara cu instrumente, dar și meniurile aferente acesteia.
* Pages – aici se află fiecare componentă de React ce definește o pagină a aplicatiei.
* Actions și reducers – fișiere aferente folosirii librăriei open-source redux, componentă ce este necesară pentru crearea și manipularea stării globale a aplicației

## Starea globală a aplicației

Prin intermediul librăriei React, fiecare componenta creată are o stare și proprietăți interne care o definesc în mod activ, atât din punct de vedere al conținutului informațional cât și din punct de vedere al modului de prezentare. Aceste proprietăți sunt fie aduse de la server, fie moștenite de la o componentă părinte și pot fi folosite local sau transmise mai departe componentelor copii. Însă pot exista situații în care aceeași informație este necesară în 2 componente care nu sunt legate printr-o relație de tip părinte-copil. În această situație intervine librăria redux, care creează o stare globală pentru aplicație. Fiecare componentă poate modifica sau accesa această stare generală, distribuirea de date intre componente care nu sunt in relație părinte-copil fiind foarte simplă. În ceea ce privește starea globală, aceasta este menținută într-un container special numit si *Store*

Interacțiunea dintre Redux și componentele create este mediată de *acțiuni* și *reduceri*. *Reducerii* sunt componente care definesc modul în care starea globală se schimbă în funcție de *acțiunea* invocată.

|  |
| --- |
|  |
| Figura 2.3.1: Ciclul redux |

În momentul in care o componentă din interfața grafică dorește să modifice sau să adauge o informație în starea globală, aceasta trebuie să invoce o acțiune. Parametru obligatoriu pentru fiecare acțiune este tipul acesteia, care are rol de cheie unică. Pe lângă tipul acțiunii se pot transmite și alți parametrii cum ar fi id-uri pentru cererile către server sau chiar obiecte întregi ce urmează a fi salvate.  
 Pe baza tipului transmis acțiunii, redux caută în reducători o rutină de deservire. Odată găsit reducătorul, acesta modifică starea globală prin adăugarea, ștergerea sau adaptarea datelor. Ulterior orice schimbare în containerul redux pentru stare, provoacă o reevaluare a componentelor conectate la acesta.

În directorul /reducer sunt definiți reducătorii:

* LoginReducer
* EmployeeReducer
* ProjectReducer
* ReportReducer
* TaskReducer
* TODOReducer
* SnackReducer

În fiecare din acești reduceri sunt definite diferite metode de răspuns pentru invocarea acțiunilor, în funcție de tipul acestora. Toți sunt apoi invocați in componenta Reducer.js, care folosind metoda combineReducers preia toți reducătorii și îi unește în vederea folosirii lor.

##### Declararea și combinarea reducerilor

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | **import** { combineReducers } from 'redux';  **import** { employeeReducer } from './EmployeeReducer';  **import** { taskReducer } from './TaskReducer';  **import** { projectReducer } from './ProjectReducer';  **import** { loginReducer } from './LoginReducer';  **import** { todoReducer } from './TODOReducer';  **import** { reportReducer } from './ReportReducer';  **import** { snackReducer } from './SnackReducer';  **export** **default** combineReducers({  employeeObj: employeeReducer,  taskObj: taskReducer,  projectObj: projectReducer,  loginObj: loginReducer,  todoObj: todoReducer,  reportObj: reportReducer,  snackObj: snackReducer  }) |

Fiecare reducer definit trebuie importat în componenta Reducer.js(vezi liniile 2-9. De asemenea se importă și funcția combineReducers(linia 1) oferită de redux. Aceasta primește ca argument un obiect, care conține fiecare reducer importat. Rezultatul acestei metode este exportat către containerul stării globale, care este definit in fișierul /store.js:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **import** { createStore, applyMiddleware } from 'redux'  **import** rootReducer from './reducers/Reducer'  **import** thunk from 'redux-thunk'  **export** **const** store = createStore(rootReducer, applyMiddleware(thunk)) |

Funcția createStore preia obiectul rootReducer, ce conține toți reducerii definiți, împreună cu un optimizator pentru container și la fiecare pornire a aplicației creează un store nou, gata de utilizare.

## Punctele de intrare ale aplicației

În momentul inițial singurul lucru ce poate fi afișat in pagină este div-ul cu id-ul “root” din fișierul /public/index.html. Acesta este punctul de start al aplicației. Peste acest div este randată fiecare componentă de React creată. În directorul /src, directorul sursă pentru aplicația React, se află fișierul index.js. Aici se importă, printre altele, librăria „react-doom” și componenta „App”.

Folosind metoda render, pusă la dispoziție de librăria react-dom, este randată componenta App:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ReactDOM.render(  <Provider store={store}>  <Router history={history}>  <App />  </Router>  </Provider>,  document.getElementById('root')  ); |

Metoda render necesită două argumente: o componentă sau succesiune de componente, aflate in relația părinte-copil și un element din /index.html peste care să se construiască arborele cu elementele construite in React ce vor fi afișate. Acest element este preluat prin metoda getElementById, care primește ca parametru id-ul corespunzător.

Componentele Provider și Router sunt cele care fac disponibile întregii aplicații containerul ce conține starea globala, respectiv elementul definitoriu pentru istoric, care face posibilă navigarea între diferitele rute ale aplicației.

Prima componentă randată în arborele paginii este componenta App din directorul /app. Aceasta are rolul de a verifica in primă instanță dacă utilizatorul este autentificat. Astfel se decid ce rute sunt disponibile pentru vizualizare. Stabilirea rutelor se face prin componentele Switch și Route puse la dispoziția dezvoltatorilor de către librăria „react-router-dom”.

##### Exemplu de rutare in cazul în care utilizatorul este neautentificat:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | < Switch >  <Route path="/report">  <ReportPage />  </Route>  <Route path="/">  <LoginPage />  </Route>  </Switch > |

Fiecare pagină definită de către dezvoltator are asignată o rută specifică, prin care se poate face o singură redirecționare. Orice altă rută introdusă în adresa url, care nu este predefinită în acest fel, va fi redirectată folosind cea mai asemănătoare definiție existentă

.

## Securizarea aplicației

În momentul accesării aplicației primul pas executat în componenta App.js este verificarea autentificării. În funcție de starea autentificării se decide ce rute sunt disponibile:

|  |
| --- |
|  |
| Figura 3.4.1: Atribuirea rutelor în funcție de starea autentificării |

După cum se poate observa, în cazul in care utilizatorul nu este autentificat ruta corespunzătoare este „/”, care este asignată componentei LoginPage(vezi 3.3.1.1.1).

|  |
| --- |
|  |
| Figura 3.4.2: Componenta LoginPage |

Cele 2 câmpuri se completează cu datele de autentificare cunoscute, iar inputul utilizatorului este stocat într-un obiect inițial gol:

|  |  |
| --- | --- |
| 1  2  3  4 | **const** [loginData, setLoginData] = useState({  username: '',  password: ''  }) |

Metoda useState, este un *hook* pus la dispoziția dezvoltatorilor de către React. *Hook*-urile sunt metode specifice care permit utilizatorului să modifice sau să creeze noi obiecte care să fie păstrate în starea internă a componentei.

Hook-ul useState, primeste ca parametru un obiect sau o variabilă care să definească starea inițială. Rezultatul întors de această metodă constă într-o variabilă prin care se poate accesa valoarea curentă din starea internă, în cazul de fată „loginData”, dar și o funcție care permite modificarea acestei valori. Cele 2 inputuri afișate sunt componente TextField, adaptate pentru situația de față:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <TextField  className="form-fields"  id="username"  label="User Name"  variant="outlined"  value={loginData.username}  onChange={(event) => {  setLoginData({ ...loginData, username: event.target.value })  }}  /> |

După cum se poate observa pe linia 6 în codul descris anterior, valoarea primului input este preluată din starea internă al componentei părinte LoginPage, loginData.username. În momentul în care in zona de input se introduc de la tastatură caractere, proprietatea „onChange” a componentei TextField, se activează și apelează funcția „setLoginData”. Aceasta primește ca parametrii starea internă curentă și o proprietate, având cheia „username” cu valoarea nou introdusă de la tastatură. Trimiterea stării interne curente este necesară deoarece orice apel al funcției „setLoginData”, suprascrie complet datele. Astfel este necesară trimiterea stării curente, împreună cu proprietatea ce se dorește a fi modificată.

După completarea formularului cu datele necesare, utilizatorul se poate autentifica prin apăsarea butonului „Log In”. Datele sunt trimise către server, care verifică dacă in baza de date există un utilizator care să corespundă câmpului „username” trimis. Dacă este găsit un astfel de utilizator se verifică potrivirea parolei introduse și în caz de succes se trimite un obiect aferent utilizatorului găsit.

##### Apelarea acțiunii „logIn”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <Button  variant="contained"  className="primary-buttons"  onClick={() => props.logIn('LOG\_IN', loginData)}  >  Log in  </Button> |

La acționarea butonului „Log In”, prin proprietatea „onClick” se face un apel către acțiunea predefinită „logIn”, aceasta primind ca parametrii tipul acțiunii și obiectul cu informațiile introduse de utilizator. Props este o variabilă prin care se pot accesa proprietățile componentei pentru care este definită:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **const** LoginPage = (props) => {  ...  }  **const** mapStateToProps = (state) => ({  loginObj: state.loginObj  });  **const** mapDispatchToProps = {  logIn  };  **export** **default** connect(mapStateToProps, mapDispatchToProps)(LoginPage); |

Se poate observa pe linia 1 definiția componentei funcționale „LoginPage”, care primește ca parametru variabila props. Funcția „mapStateToProps” (liniile 4-6) are rolul de a asigna proprietăților interne ale componentei, accesibile prin variabila props, și obiectul „loginObj” din starea globală. Metoda „mapDispatchToProps” (liniile 7-9) leagă acțiunile predefinite, de care avem nevoie, de variabila props, astfel încât să le putem invoca tot prin props. În cazul de față avem nevoie doar de acțiunea „logIn”. Ulterior pe linia 10 se poate observa cum în momentul exportului spre folosire a componentei, se apelează mai întâi metoda connect, din librăria „react-redux”, care primește ca parametrii cele 2 funcții create pentru maparea stării globale și a acțiunilor peste proprietățile interne ale componentei.

În cazul de față avem nevoie doar de acțiunea „logIn”, pe care prin conectarea la proprietățile interne ale componentei o vom apela trimițând 2 parametrii: tipul actiunii „LOG\_IN” și obiectul „loginData” , necesar autentificării. (Vezi 3.4.1.1.1 linia 4). Implementarea propriu zisă a acestei acțiuni se poate observa in codul următor preluat din fișierul „/actions/CommonActions.js”:

|  |  |
| --- | --- |
| 1  2  3  4 | **export** **const** logIn = (type = "LOG\_IN", loginData) => ({  type: type,  loginData: loginData  }) |

Odată invocată această acțiune, Redux începe să caute în reducerii declarați (Vezi 3.2.1.1.1), folosindu-se de tipul oferit, o rutină care să trateze apelul funcției. În „LoginReducer” se regăsește codul care tratează invocarea acțiunii „logIn”:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **case** 'LOG\_IN': {  Axios.post('http://localhost:8081/employees/loginEmployee/', action.loginData)  .then((response) => {  **if** (response.data !== '') {  localStorage.setItem('loggedIn', **true**)  localStorage.setItem('loggedUserId', response.data.id)  history.push('/')  **return** state  } **else** {  localStorage.setItem('loggedIn', **false**)  history.push('/login')  alert("Wrong username or password!")  **return** state  }  })  .**catch**((error) => {  alert("Something went wrong")  localStorage.setItem('loggedIn', **false**)  history.push('/login');  console.log(error)  })  **return** state  } |

Folosind librăria „axios”, se realizează apelul către serverul api, la adresa specifică, împreună cu datele necesare pentru autentificare. Axios este o librărie de JavaScript, cu ajutorul căreia se pot realiza cereri de tip HTTP, către punctele finale definite in server. Cererile de regulă sunt însoțite si de informații suplimentare, necesare primirii datelor căutate, cum ar fi id-uri, diferite denumiri, etc. În cazul de față se trimit datele necesare autentificării („loginData”) către punctul „http://localhost:8081/employees/loginEmployee/”. Aceasta din urmă este adresa specifică serverului, prin care se tratează cererea de autentificare a utilizatorului. Serverul trimite un răspuns către frontend, iar in caz de succes se stochează în „localStorage” rezultatul autentificării: un câmp „loggedIn” cu valoarea de adevăr „true” și un câmp „loggedUserId” care reprezintă id-ul utilizatorului ce tocmai s-a autentificat. În cazul în care răspunsul serverului există, dar nu este un obiect sau conexiunea dintre frontend și backend nu s-a putut stabili in „localStorage” se salvează doar variabila „loggedIn” cu valoarea de adevăr „false”. Componenta „App” fiind conectată la store, odată cu modificarea stării globale, va fi reevaluată.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **let** isLoggedIn = localStorage.getItem('loggedIn')  **if** (isLoggedIn === 'true') {  **return** (  < Switch >  <Route path="/tasks">  <Tasks />  </Route>  <Route path="/">  <HomePage />  </Route>  ...  </Switch >  )  } **else** {  **return** (  < Switch >  ...  </Switch >  )  } |

Valoarea variabilei „isLoggedIn” este modificată cu noua valoare stocată in „localStorage” (vezi linia 1), iar apoi în funcție de aceasta se decide care set de rute va fi disponibil pentru utilizator.

## Structura generală a paginilor

### AppLayout

Structura generală a fiecărei pagini este menținută folosind componenta „AppLayout” din directorul „/shared”, ce conține resurse necesare tuturor paginilor. AppLayout definește structura generală, preluând componentele ce sunt trimise atunci când este utilizată:

##### Utilizarea componentei AppLayout

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | const HomePage = (props) => {  ...  return (  <AppLayout title="Home Page">  ...  <div className="home-page">  ...  </div>  </AppLayout>  )  } |

Toate componentele ce definesc paginile aplicației(vezi Figura 3.4.1), se folosesc de AppLayout, în aceeași manieră. În „HomePage” importăm „AppLayout”, iar apoi folosim componenta, trimițând ceea ce trebuie să afișăm in fiecare pagină în parte. Se observă pe linia 4 că pe lângă componentele de afișat trimise, mai adăugăm și proprietatea „title”, în cazul de față „Home Page”. Toate componentele, respectiv proprietățile trimise vor fi accesibile în componenta „AppLayout”, prin variabila props:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | const AppLayout = (props) => {  ...  return (  <div>  <NavBar title={props.title}></NavBar>  <Container...">  {props.children}  </Container>  <SnackBar.../>  </div>  )  } |

Se remarcă pe linia 1, faptul ca „AppLayout” așteptă să primească proprietăți, prin existența variabilei „props” ca argument. Ulterior, pe linia 5 observăm cum proprietatea „title”, accesibilă prin props, este trimisă mai departe către componenta „NavBar”. Aceasta este componenta care definește bara de instrumente a aplicației.   
 Componentele trimise pentru afișare (cele care în componenta „HomePage” se află între cele 2 tag-uri „AppLayout”) sunt accesibile prin proprietatea „children”. Întrucât „props.children” este o expresie da JavaScript, este necesar să marcăm acest lucru, pentru compilator, prin încadrarea expresiei între acolade.

Fiecare pagină folosind „AppLayout” pentru afișare, va păstra aceeași structură generală, având prezente și cele 2 componente, „NavBar” și „SnackBar”.

### NavBar

Componenta „NavBar” are 3 elemente distincte, vizibile și în Figura 3.5.1 :

* AppMenu – meniul general al aplicației
* Titlul paginii – specific fiecarei secțiuni
* AccountMenu - Meniul contului de utilizator

|  |
| --- |
|  |
| Figura 3.5.1: Bara de instrumente |

#### AppMenu

Componenta „AppMenu” este importată în interiorul fișierului „/NavBar.js” și utilizată cu rolul de meniu general al aplicației. Mecanismul de deschidere/închidere al meniului este bazat pe hook-ul „useState”, care setează elementul de ancoră pentru secțiunea ce trebuie să se deschidă.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <IconButton  ...  onClick={handleClick}  >  <MenuIcon />  </IconButton> |

Icoana meniului are rol dublu: de prezentare și de buton. La acționarea butonului se activează proprietatea „onClick”, care face apel către funcția „handleClick”.

|  |  |
| --- | --- |
| 1  2  3 | **const** handleClick = event => {  setAnchorEl(event.currentTarget);  }; |

Aceasta preia event-ul trimis implicit, și setează elementul de ancoră pentru meniul ce va fi deschis.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | <Menu  ...  anchorEl={anchorEl}  open={Boolean(anchorEl)}  onClose={handleClose}  >  {  isLoggedIn === 'true' &&  <MenuItem onClick={() => {  history.push('/');  setAnchorEl(**null**);  }}>  <ListItemIcon>  <HomeIcon fontSize="small" />  </ListItemIcon>  <ListItemText primary="Home Page" />  </MenuItem>  ...  }  <MenuItem onClick={() => {  history.push('/report');  setAnchorEl(**null**);  }}>  <ListItemIcon>  <ReportIcon fontSize="small" />  </ListItemIcon>  <ListItemText primary="Report a problem" />  </MenuItem>  </Menu> |

Starea meniului de deschis sau închis este definită de existența elementului de ancoră. La momentul inițial, acesta este null, iar expresia de pe linia 4 este evaluată ca fiind „false”. În momentul în care se acționează butonul meniului, se setează elementul de ancoră și această expresie devine „true”, situație în care meniul se deschide. Pe linia 3 poate remarca și setarea elementului de ancoră pentru meniu, trimițând proprietății „anchorEl”, variabila asociată.

Pentru afișarea diferitelor opțiuni ale aplicației este necesară verificarea stării de autentificare, pentru a permite sau nu vizualizarea resurselor. Acest lucru se realizează prin verificarea stării de adevăr pentru variabila stocata în „localStorage”, „isLoggedIn” (vezi linia 8).  
 În funcție de starea acestei variabile se decide ce opțiuni sun afișate. Tot ce se află în blocul delimitat de liniile 7-19, va fi disponibil utilizatorului doar daca este autentificat, pe când ce se află după acest bloc, linia 20 și mai departe, nu depinde de starea de autentificare a utiliatorului.

În momentul în care este acționată vreuna din opțiunile meniului, se realizează 2 operatii:

* Redirecționarea către pagina aferentă opțiunii alese, prin instrucțiunea „history.push()”(vezi liniile 10 și 21), aplicația având rutele predefinite în componenta „App”.
* Setarea elementului de ancora pentru meniu, cu valoarea „null”, prin instructiunea „setAnchorEl(null)”(vezi liniile 11 și 22).

#### Titlul paginii

Prin invocarea componentei „AppLayout” în fiecare pagină și prin trimiterea proprietății „title”, către „NavBar”, se stabilește titlul specific fiecărei sectiuni din aplicație:

|  |  |
| --- | --- |
| 1  2  3 | <Typography ...>  {props.title}  </Typography> |

Pentru asta, în componenta „NavBar”, este importată „Typography” din librăria „material-ui” și se utilizează împreună cu proprietatea „title”.

### SnackBar

„SnackBar” este componenta aplicației responsabilă pentru a transmite mesaje de informare în funcție de răspunsurile trimise de server înapoi, la diferitele acțiuni ale utilizatorului. De fiecare dată când utilizatorul adaugă, modifică sau șterge date, toate cererile sunt trimise către server. În funcție de răspunsul serverului folosind starea globală a aplicației se activează componenta „SnackBar”, afișând un mesaj de succes sau eroare.

|  |
| --- |
|  |
| Figura 3.5.1: Activarea mesajelor de informare |

Vom exemplifica modul de activare a mesajelor de informare pentru situația în care se adaugă o nouă sarcină de lucru.  
 Atunci când se trimit datele către server, se folosește o acțiune predefinită care caută in reduceri folosind tipul „SAVE\_TASK”. În rutina de tratare, ce se află în fișierul „TaskReducer”, se realizează o cerere de tip „POST” către url-ul aferent serverului. Serverul primește datele și trimite un răspuns. În funcție de acest răspuns se activează o acțiune responsabilă de modificarea stării globale pentru mesajele de informare:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Axios.post(...)  .then((response) => {  ...  store.dispatch(callSnack('SUCCESS', 'Task was saved!'));  })  .**catch**((error) => {  store.dispatch(callSnack('FAILURE', 'Something went wrong!'));  ...  }) |

În acest caz neavând proprietăți interne acțiunea „callSnack” este invocată direct prin metoda „dispatch()” aferentă variabilei „store”, cea care conține starea globală a aplicației. Această metodă primește ca si argument o acțiune pe care o va invoca și ulterior va începe procesul de căutare a rutinei de tratare în reducerii declarați. Acțiunea „callSnack”, la rândul ei primește ca parametrii tipul, după care se va face căutarea și un mesaj pentru utilizator care să-l informeze cu privire la răspunsul serverului(vezi liniile 4 respectiv 7 din codul expus).

În interiorul fișierului „SnackReducer” se găsește reducerul care tratează activarea acțiunii „callSnack” cu tipul „SUCCESS”:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **case** 'SUCCESS': {  **return** Object.assign({}, state, {  snack: {  open: **true**,  severity: 'success',  message: action.message  }  });  } |

În codul de mai sus, pe linia 3, în variabila „snack” se introduc câmpurile „open”, „severity” și „message”. Starea globală fiind modificată, se va declanșa reevaluarea tuturor componentelor care sunt legate la variabila „snackObj” din store(vezi 3.2.1.1.1 linia 17).

În interiorul componentei „AppLayout” se leagă variabila din starea globală ce conține informațiile legate de procesul de afișare a mesajului de informare, de proprietățile interne:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **const** mapStateToProps = (state) => ({  snackObj: state.snackObj,  });  ...  **export** **default** connect(mapStateToProps, mapDispatchToProps)(AppLayout); |

Toate câmpurile setate în variabila „snack” din „snackObj”, sunt trimise ca proprietăți componentei „SnackBar”, din interiorul „AppLayout”:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <SnackBar  open={snack && snack.open}  severity={snack && snack.severity}  message={snack && snack.message}  /> |

Ulterior în interiorul fișierului „SnackBar” se importă din librăria „@material-ui/core”, componenta „Snackbar”, care se folosește de aceste proprietăți pentru stabilirea tipului de informare (succes sau eșec) respectiv a mesajului ce trebuie să ajungă la utilizator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | return (  <Snackbar  anchorOrigin={{  vertical: 'bottom',  horizontal: 'left',  }}  open={props.open}  autoHideDuration={6000}  onClose={handleClose}  action={  <IconButton ... onClick={handleClose}>  <CloseIcon fontSize="small" />  </IconButton>  }  >  <Alert onClose={handleClose} severity={props.severity}>  {props.message}  </Alert>  </Snackbar>  ); |

Liniile 3-6 setează elementul de ancoră pentru mesaj în partea de jos, în stânga ecranului. Prin „props.open” se trimite starea de deschis sau închis pentru componenta SnackBar. Pe linia 8 se setează un timp de afișare egal cu 6 secunde (acest timp trebuie trimis în milisecunde). Pentru momentul în care trebuie încetată afișarea mesajului, fie datorită expirării timpului setat, fie datorită acționarii butonului de închidere a notificării, este implementată metoda „handleClose” și trimisă proprietății „onClose” a „Snackbar”.   
 Liniile 10-15 desemnează o singură acțiune pentru notificări și anume posibilitatea de încetare a afișării prin acționarea butonului de închidere. De asemenea la apăsarea acestui buton se folosește „handleClose” pentru a determina cursul de acțiune pentru încetarea afișării. Liniile 16-18 este definită notificarea propriu zisă prin componenta „Alert”. Aceasta primește ca proprietăți metoda de încetarea a afișării „handleClose”, câmpul „severity” care definește tipul mesajului (succes sau eroare) și mesajul pentru utilizator („props.message”).

|  |
| --- |
|  |
| Figura 3.5.1: Notificarea de succes |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **const** handleClose = (event, reason) => {  **if** (reason === 'clickaway') {  **return**;  }  props.closeSnack()  }; |

În metoda „handleClose” se verifică mai întâi dacă motivul declanșării este apăsarea mausului în altă zonă, moment în care nu se dorește încetarea afișării. Pentru orice alt motiv (acționarea butonului de închidere, sau expirarea timpului prestabilit) se invocă acțiunea „closeSnack”, care după același model setează câmpul „open” cu valoarea booleană „false”.

## Pagina de start

După procesul de autentificare utilizatorul este redirecționat pe ruta „/” care în componenta „App” este atribuită paginii de start. Această pagina este implementată în componenta „HomePage”, din fișierul cu același nume.

La accesarea acestei pagini utilizatorul este întâmpinat cu un mesaj personalizat în funcție de contul folosit la autentificare:

|  |
| --- |
|  |
| Figura 3.5.1: Mesaj de întâmpinare |

Prin procesul descris în secțiunea 3.4.1.1.1 după autentificare, în localStorage este salvat id-ul utilizatorului autentificat. În interiorul componentei „HomePage” acest id este preluat și cu ajutorul lui se realizează o cerere către server pentru aducerea datelor aferente utilizatorului corespunzător. Această cerere este obținută prin intermediul acțiunii „getDataById”, care primește ca parametrii tipul „GET\_EMPLOYEE\_BY\_ID” și id-ul utilizatorului autentificat:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **const** loggedUserId = localStorage.getItem('loggedUserId')  useEffect(() => {  props.getDataById('GET\_EMPLOYEE\_BY\_ID', loggedUserId)  }, [loggedUserId]); |

Pe linia 3 se observă hook-ul „useEffect”, oferit de React, care are ca scop activarea unui bloc de instrucțiuni și reevaluarea conținutului de afișat, în cazul în care vreuna din variabilele de testat și-au modificat valoarea. În cazul de față variabila de testat este chiar „loggedUserId”, care se schimbă doar odată în întreg ciclul de viață al componentei și anume la început când se inițializează. Odată cu stocarea valorii id-ului utilizatorului autentificat, în variabila „loggedUserId”, se declanșează hook-ul „useEffect”, în interiorul căruia se invocă acțiunea precizată.

Rutina de tratare specifică tipului „GET\_EMPLOYEE\_BY\_ID”, se regăsește in reducerul „EmployeeReducer”. În acest caz, folosind „Axios”, se realizează o cerere de tip „GET” către server, id-ul necesar fiind trimis ca parametru în url:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **let** url = `http://localhost:8081/employees/getEmployee/${id}/`;  Axios.get(url)  .then((response) => {  store.dispatch(gotData(response.data, 'GOT\_EMPLOYEE\_BY\_ID'));  })  .**catch**((error) => {  console.log(error);  }); |

Pe linia 1 se construiește url-ul necesar, alipind la finalul său id-ul primit. Ulterior se realizează cererea către backend (vezi linia 2) și dacă serverul răspunde trimițând datele utilizatorului, acestea sunt pasate, pe linia 4, acțiunii „gotData” împreună cu tipul „GOT\_EMPLOYEE\_BY\_ID”. Această acțiune tratează situația în care s-au adus datele, adăugând în starea globală un obiect denumit „loggedEmployee”, care conține toate datele corespunzătoare utilizatorului autentificat:

|  |  |
| --- | --- |
| 1  2  3 | **case** 'GOT\_EMPLOYEE\_BY\_ID': {  **return** Object.assign({}, state, { loggedEmployee: action.data });  } |

În cazul în care toate aceste date sunt aduse, în interiorul componentei „HomePage”, este folosit prenumele utilizatorului autentificat pentru crearea mesajului de întâmpinare personalizat.

Pe lângă acest mesaj, pe pagina de pornire se mai poate observa un set sumar de instrucțiuni si linkuri, cu privire la celelalte pagini ale aplicației:

|  |
| --- |
|  |
| Figura 3.5.1: Intrucțiuni și linkuri din pagina de start |

## Pagina proiectelor personale

Pagina proiectelor personale este o secțiune în care utilizatorul poate vedea toate proiectele în care este angrenat, să gestioneze sarcinile de lucru pentru fiecare în parte, dar și să creeze un proiect nou cu sarcini de lucru noi la nevoie.

### Afișarea proiectelor

Afișarea tuturor proiectelor pentru un utilizator se realizează prin componenta „Projects”, accesibilă pe ruta „/projects”. În această secțiune se poate ajunge fie prin linkul din pagina de start, fie prin folosirea meniului din bara aplicației.

|  |
| --- |
|  |
| Figura 3.5.1: Pagina proiectelor personale |

Componenta „Projects” este implementată folosind vechea metodă propusă de React, folosind clase. Folosind acest mod de implementare, se pot folosi metodele de ciclu de viață ale unei componente. Ciclul de viață al unei componente React este compus din 8 stadii succesive după cum urmează:

|  |
| --- |
|  |
| Figura 3.5.1: Ciclul de viață al unei componente React |

În componenta „Projects” este folosită metoda „componentDidMount” pentru a face o cerere de la server cu datele utilizatorului autentificat, în cazul in care acestea nu există deja în starea globală a aplicației:

|  |  |
| --- | --- |
| 1  2  3  4  5 | componentDidMount() {  **if** (**this**.props.employeeObj.loggedEmployee === **undefined**) {  **this**.props.getDataById('GET\_EMPLOYEE\_BY\_ID', loggedUserId)  }  } |

Pentru afișarea datelor se utilizează un tabel importat din librăria „material-table”, modificat în funcție de nevoi. Această librărie este importată și utilizată în componenta „EnhancedTable”, pe care o și folosim în secțiunea proiectelor personale:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <EnhancedTable  columns={header}  data={data}  name="projects"  /> |

Pentru a funcționa tabelul importat are nevoie de date prelucrate care să se potrivească cu capul de tabel. În continuare vom descrie procesul de prelucrare al datelor:

* Datele aduse de la server sunt reprezentate sub forma de JSON :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {  description: "Test Projeect"  expectedDeliveryDate: "06/05/2020"  id: **1**  name: "Project 1"  startDate: "03/04/2020"  ...  } |

* Acestea sunt stocate în variabila „assignedProjects” și folosind metoda „getFields”, se preiau doar cheile din obiectul salvat (description, name, etc.):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **function** getFields(project) {  **let** fields = [];  Object.entries(project).map(([key, value], index) => {  fields.push(key);  });  **return** fields;  } |

În interiorul metodei „getFields”, pentru toate câmpurile unui proiect se preiau perechile de cheie-valoare, cheia fiind adăugată în vectorul „fields”.

* Următorul pas este construirea propriu zisă a capului de tabel:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **let** fieldsToSend = fields && [fields[**0**], fields[**1**], fields[**3**], fields[**4**]];  **let** header = [];  fieldsToSend && fieldsToSend.map((field, index) => {  **let** columnHeader = {  title: \_.startCase(field),  field: field  }  header.push(columnHeader)  }) |

Pe linia 1 selectăm elementele de pe pozițiile 0, 1, 3 și 4 pentru afișare în capul de tabel. Ulterior pe liniile 4-7, fiecare coloană va trebui sa conțină un titlu, pentru afișare și un câmp, după care să se realizeze căutarea in datele de expus. Pe linia 8 capul de tabel procesat este adăugat in variabila „header” ce va fi trimisă tabelului.

* Ultima parte din preprocesarea tabelului constă în asignarea tuturor datelor ce privesc proiectele, variabilei „data”, care va conține datele reprezentate sub formă de JSON, cu cheile identice cu câmpul „field” din „header”.

Pe lângă afișarea datelor tabelul mai permite și sortarea informațiilor în ordine crescătoare sau descrescătoare, în funcție de o anumită coloană, folosirea câmpului „Search” pentru a căuta în tabel după anumite date sau denumiri, dar și 2 acțiuni care depinde de fiecare proiect în parte: ștergerea definitivă a unui proiect și gestionarea sarcinilor de lucru pentru un proiect. Aceste acțiuni sunt disponibile prin utilizarea butoanelor din dreptul fiecărui proiect:

|  |
| --- |
|  |
| Figura 3.5.1: Acțiunile corespunzătoare unui proiect |

Prin acționarea butonului de ștergere se deschide o fereastră de confirmare care permite fie anularea acțiunii fie ștergerea proiectului în cauză:

|  |
| --- |
|  |
| Figura 3.5.1: Acțiunile corespunzătoare unui proiect |

La apăsarea butonului „DELETE”, acțiunea „deleteProjectById” este invocată, primind tipul „DELETE\_PROJECT\_BY\_ID”, alături de id-ul proiectului ce se dorește a fi șters. Rutina de tratare pentru tipul menționat se află în „ProjectReducer”:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **case** 'DELETE\_PROJECT\_BY\_ID': {  ...  **let** url = `http://localhost:8081/projects/deleteProject/${id}`;  Axios.**delete**(url)  .then((response) => {  store.dispatch(callSnack('SUCCESS', 'Project was deleted!'));  store.dispatch(getDataById('GET\_EMPLOYEE\_BY\_ID', loggedUserId));  })  .**catch**((error) => {...});  **return** state;  } |

Pe linia 3 se construiește url-ul la care se va face cererea către server, adăugând la final id-ul proiectului ce se dorește a fi șters. Pe linia 4 se realizează cererea de tip „DELETE”, folosind Axios, urmând ca în cazul unui răspuns afirmativ să se activeze sistemul de notificare, cu mesajul „Project was deleted!” și să se activeze prin „store.dispatch()” acțiunea de aducere a datelor unui angajat, cu tot cu proiectele și sarcinile de lucru ale acestuia, pentru reactualizarea datelor din tabel.

Tot in această secțiune prin apăsarea butonului „NEW PROJECT” se poate crea un proiect nou prin completarea formularului ce se deschide:

|  |
| --- |
|  |
| Figura 3.5.1: Formular de adăugare a unui proiect |

Pentru crearea unui nou proiect, se creează în starea internă a componentei un obiect „newProject”:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | newProject: {  name: '',  description: '',  startDate: **new** Date(),  expectedDeliveryDate: **new** Date(),  }, |

Fiecare câmp din fereastra de adăugare a unui proiect, este legat de una din proprietățile obiectului „newProject”, după modelul următor:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <TextInputBoxWrapper  id="projectName"  label="Project Name"  value={**this**.state.newProject.name}  onChange={(event) => **this**.setState({  newProject: {  ...**this**.state.newProject,  name: event.target.value  }  })}  ...  /> |

Pe liniile 2 și 3 se setează id-ul precum și denumirea standard pentru un câmp. Mai departe pe linia 4 se stabilește de unde să-și ia acest câmp valoarea și anume din obiectul inițial gol, creat in starea internă a aplicației.   
 Liniile 5-9 descriu cum trebuie să răspundă câmpul în cauză, la introducerea unui input de la tastatură. De fiecare dată când se întâmplă acest lucru, se modifică în starea internă a aplicației câmpul „name” din obiectul „newProject” și implicit și valoarea pe care inputul nostru o afișează (vezi linia 4). În aceeași manieră lucrează și celelalte 3 câmpuri.

După completarea formularului, la apăsarea butonului „SAVE”, se activează acțiunea „saveProject”, cu tipul „SAVE\_PROJECT\_BY\_EMPLOYEE”, împreună cu obiectul „newProject”, dar și cu id-ul utilizatorului autentificat.   
 În aceeași manieră ca și până acum se caută rutina de tratare, după tipul trimis. Aceasta este prezentă în „ProjectReducer” și prin intermediul Axios se realizează cererea de tip „POST”, trimițând datele aferente proiectului nou, alături de id-ul utilizatorului autentificat. În cazul unui răspuns pozitiv de la server, se invocă acțiunea „getDataById” cu tipul „GET\_EMPLOYEE\_BY\_ID”, împreună cu id-ul utilizatorului autentificat, pentru reîmprospătarea datelor ce trebuie afișate, dar și acțiunea de activare a notificării de succes cu mesajul „Project was saved!”.(XXXX pune referință catre cod în anexă)

### Managementul sarcinilor de lucru într-un proiect

## Pagina sarcinilor de lucru personale

## Pagina notițelor personale

## Pagina de raportare

# Serverul

# Concluzii

# Bibliografie

Made by Mihai A

XXXX add here

1. <https://ro.wikipedia.org/wiki/MySQL>
2. ccc
3. ddd

# Annex 1

optional annexes as needed.

# Annex 2

optional annexes as needed