

MINISTERUL EDUCAȚIEI AL REPUBLICII MOLDOVA  
UNIVERSITATEA DE STAT „ALECU RUSSO” DIN BĂLȚI  
FACULTATEA DE ȘTIINȚE REALE, ECONOMICE ȘI ALE MEDIULUI  
CATEDRA DE MATEMATICĂ ȘI INFORMATICĂ

**UTILIZAREA REACT JS SI REDUX LA CREAREA  
PĂRȚII FRONT END A APLICAȚIEI  
TEZĂ DE LICENȚĂ**

**Autor:**

Student al grupei IS31Z

Popa Dan

---

**Conducători științifici:**

Sergiu CHILAT

lector univ., magistru

---

**BĂLȚI, 2019**

Controlată:

Data \_\_\_\_\_

Conducător științific: Sergiu Chilat, (lector univ., magistru)

\_\_\_\_\_  
(semnătura)

Aprobată

și recomandată pentru susținere

la ședința Catedrei de \_\_\_\_\_

Proces verbal nr. \_\_\_\_\_ din \_\_\_\_\_

Șeful catedrei \_\_\_\_\_

dr., conf. univ. \_\_\_\_\_

Aprobat:  
Şeful de catedră \_\_\_\_\_  
” ” \_\_\_\_\_ 201 \_\_\_\_\_

### Graficul calendaristic de executare a tezei de licenţă

Tema tezei de licenţă **Utilizarea react js si redux la crearea părții front end a aplicației**

confirmată prin ordinul rectorului USARB

nr. \_\_\_\_\_ din „ \_\_\_\_\_ ”

Termenul limită de prezentare a tezei de licenţă la Catedra de matematică şi informatică

„ \_\_\_\_\_ ” \_\_\_\_\_ 2019

### Etapele executării tezei de licenţă:

Etapele	Termenul de realizare	Viza de executare
1. Stabilirea temei; fixarea obiectivelor; selectarea surselor de informare		
2. Investigația cadrului teoretic al cercetării; expunerea cadrului teoretic al cercetării		
3. Întocmirea problemei cercetării; stabilirea tipului de cercetare		
4. Specificarea unităților studiate		
5. Stabilirea și aprobarea planului tezei de licență		
6. Alegerea metodelor de cercetare; stabilirea tehnicilor și procedeele de lucru, etc.		
7. Elaborarea prototipului aplicației bazate pe react si redux		
8. Elaborarea și prezentarea capitolului I		
9. Elaborarea și prezentarea capitolului II		
10. Elaborarea aplicației și prezentarea capitolului III		
11. Redactarea tezei, rezolvarea aspectelor de grafică și design la calculator		
12. Susținerea prealabilă a tezei		

Student (a) \_\_\_\_\_  
(semnătura)

Conducătorul științific \_\_\_\_\_  
(semnătura)

## ADNOTARE

Popa Dan

### UTILIZAREA REACT JS SI REDUX LA CREAREA PĂRȚII FRONT END A APLICAȚIEI

Teză de licență. Bălți, 2019

*Structura tezei:* introducere, trei capitole, concluzii generale, bibliografie din 26 titluri, 44 pagini, 12 figuri, 1 tabel.

*Cuvintele cheie:* javascript, front end, react, redux, state, CRUD, view library.

*Domeniul de studii:* tehnologii javascript la elaborarea aplicațiilor web.

*Scopul cercetării:* constă în cercetarea bibliotecilor redux și react, analiza posibilităților lor și elaborarea unei aplicații web.

*Obiectivele lucrării:*

1. Analiza resurselor informative: literatură de specialitate, comunitățile și tutorialele;
2. Analiza generală a bibliotecii redux (metode de utilizare, compabilitate, actualitate);
3. Analiza generală a bibliotecii react (metode de a crea componente);
4. Analiza tehnologiilor și soluțiilor alternative;
5. Proiectarea aplicației de păstrare a linkurilor ce se grupează automat după domeniu.

*Noutatea și originalitatea științifică* a lucrării: biblioteca redux a apărut în anul 2015 iar react în 2013 primind rapid un suport puternic de la comunitate, în câțiva ani evoluând rapid. La moment aceste două biblioteci nu sunt cercetate suficient, iar în limba română documentație practic nu există.

*Semnificația teoretică* a cercetării: datorită faptului că în limba română nu există careva documentație, suportul teoretic al acestei lucrări poate fi utilizat de programatorii care doresc să studieze bibliotecile redux și react.

*Valoarea aplicativă a lucrării* constă în faptul că a fost elaborată o aplicație cu cod documentat, care, de asemenea poate servi ca suport practic pentru programatorii javascript și typescript.

## ANNOTATION

Popa Dan

### THE USE OF REACT JS AND REDUX FOR THE CREATION OF THE FRONT END PART OF AN APPLICATION

Bachelor thesis. Bălți, 2019

*Bachelor thesis:* Introduction, 3 chapters, general conclusions and recommendations, bibliography of 26 titles, 44 pages of basic text, 12 figures, 1 table.

*Keywords:* Javascript, front end, react, redux, state, CRUD, view library.

*Area of study:* Javascript technologies for the elaboration of web applications.

*The purpose of the research:* consists in researching the redux and react libraries, the analysis of their possibilities and in the elaboration of a web application.

*Objectives:*

1. Analysis of information resources: specialized literature, communities and tutorials;
2. General analysis of the possibilities of the Redux library(use methods, compatibility, actuality);
3. General analysis of the possibilities of the React library(methods to create components);
4. Analysis of the technologies and alternative solutions;
5. Developing of an application which stores the links and groups them automatically by domain name.

*Novelty and scientific originality:* Redux library was created in 2015 and React in 2013, both receiving a strong support from the community, progressing rapidly in the following years. At the current moment these two libraries are not researched enough, and there si almost no documentation in Romanian.

*Theoretical significance:* because there is almost no documentation in Romanian, the theoretical support may be used by programmers who wants to study Redux and React libraries.

*Application value:* consists in the development of a documented code application that can also serve as practical support for Javascript and Typescript programmers.

## Anexa 1. Lista de abreviaturi

DRY – Don't Repeat Yourself;

CoC – Convention over Configuration;

WeT – Write Everything Twice;

ORM – Object / Relational Mapping;

CRUD – Create, Read, Update, Delete;

IDE – Integrated Development Enviroment;

URN – Uniform Resource Name.

JWT – Json Web Token

UX – User Experience

API – Application Programming Interface

## CUPRINS

INTRODUCERE .....	4
1. PREZENTAREA BIBLIOTECILOR REDUX ȘI REACT.....	6
1.1. Scurt istoric.....	6
1.2. Motivele utilizării React Js.....	6
1.3. Bazele React JS .....	7
1.4. Motivele utilizării Redux.....	8
1.5. Bazele Redux-ului .....	8
1.6. Integrarea Redux-ului în aplicațiile React.....	11
1.7. Ecosistema Redux-ului.....	13
2. ANALIZA TEHNOLOGIILOR DE MANAGEMENT A STĂRII GLOBALE ÎN JS.....	15
2.1. Mobx și Mobx-state-tree .....	16
2.2. Vuex .....	18
2.3. NgRx.....	20
3. ELABORAREA APLICAȚIEI ”BOOKY” .....	23
3.1. Spațiul de lucru Visual Studio Code .....	23
3.2. Tehnologii necesare la dezvoltarea aplicației.....	24
3.3. Crearea aplicației Booky .....	29
3.4. Testarea aplicației elaborate .....	37
CONCLUZII .....	39
BIBLIOGRAFIE .....	41

## INTRODUCERE

La momentul actual, nu este rezolvată problema de păstrare a stării în partea front end și există mai multe metode de abordare, fiecare metodă diferențiindu-se prin idei și problemele ce încearcă să le rezolve. Autorul își propune să selecteze o librărie populară, interesantă care a influențat felul cum aplicațiile front end sunt scrise și care, nu este studiat în curricula universitară.

De multe ori, pastrarea starii unei aplicatii este o problemă majoră, din cauza că există o mulțime de factori care trebuie luați în considerare. Un caz aparte sunt aplicațiile unde componentele au nevoie să împartă starea între ele. Autorul își propune să cerceteze și să argumenteze selectarea bibliotecii Redux la dezvoltarea unei aplicații web.

*Actualitatea temei:* tema cercetată este una actuală, deoarece discuțiile privind cele mai bune tehnologii de programare încă persistă. Redux și React sunt niște tehnologii actuale, fiind unele din cele mai utilizate tehnologii în spațiul web.

*Utilitatea practică:* a fost elaborată o aplicație cu cod documentat, care, de asemenea poate servi ca suport practic pentru programatorii Javascript. Această lucrare este utilă prin prezentarea multor aspecte ale implementării Redux-ului și React-ului și diferitor tehnologii în atingerea scopului comun – crearea aplicației *Booky*. Partea teoretică conține în sine cunoștințele necesare despre biblioteca Redux și React, avantaje, dezavantaje, exemple teoretice de utilizare și principii de programare pe care le utilizează. Partea practică este utilă pentru programatori, care încă nu s-au stabilit cu alegerea metodei de a păstra starea într-o aplicație bazată pe React.js. Se va examina un exemplu care conține în sine toate principiile, standardele de programare și lucrul cu diferite module a bibliotecii Redux.

*Scopul tezei* de astăzi constă în cercetarea bibliotecii Redux, analiza posibilităților combinării Redux-ului cu React și elaborarea unei aplicații web.

Pentru atingerea scopului au fost stabilite următoarele *obiective* :

1. Analiza resurselor informative cum ar fi: literatură de specialitate, comunitățile și tutorialele;
2. Analiza generală a posibilităților Redux (a paternelor, standardelor și principiilor structurale);
3. Analiza generală a bibliotecii React (metode de a crea componente)
4. Proiectarea arhitecturii și logicii aplicației (ex componente în React);
5. Proiectarea aplicației de păstrare a linkurilor ce se grupează automat după domeniu

Teza este structurată în trei capitole:



Primul capitol “Prezentarea bibliotecii Redux și React” conține șapte subcapitole:

- “Scurt istoric” – va conține o prezentare succintă a istoriei bibliotecii Redux;
- Motivele utilizării React.Js – se va vorbi despre avantajele utilizării librăriei React
- Bazele React.JS – enumerarea elementelor de bază pentru a construi o interfață în React
- “Motivele utilizării Redux” – se va vorbi despre avantajele folosirii Redux-ului
- “Bazele Redux-ului” – începând cu studierea sau cercetarea unei biblioteci va fi nevoie de a înțelege cum de a o folosi;
- “Integrarea Redux-ului în aplicațiile React” – se va vorbi despre cea mai bună metodă de a integra Redux și React;
- “Ecosistema Redux-ului” – subcapitolul dat va conține folosirea extensiilor pentru Redux;

În al doilea capitol “Analiza tehnologiilor de management a stării globale în js” vor fi analizate alternativele și soluțiile existente la combinarea Redux și React, unde vor fi enumerate avantajele și dezavantajele fiecărei opțiuni. Capitolul va conține trei subcapitole:

- “Mobx și Mobx-state-tree” – o analiză a celei mai populare alternative pentru Redux
- “Vuex” – se va vorbi despre ce mai populară librărie pentru managementul stării în Vue.JS
- “Ngrx” – se va vorbi despre ce mai populară librărie pentru managementul stării în Angular

În al treilea capitol “Elaborarea aplicației Booky” vor fi analizați pașii de elaborare a aplicației web. Capitolul va conține patru subcapitole:

- “Spațiul de lucru Visual Studio Code” – analiza posibilităților IDE-ului dat: structura, componentele de bază și posibilitățile oferite. Cum se configurează spațiul de lucru.
- “Tehnologii necesare la dezvoltarea aplicației” – subcapitolul dat va fi expus în trei subcapitole care vor reda tehnologiile necesare pentru a dezvolta aplicația propusă
- “Crearea aplicației Booky” – subcapitolul dat va fi expus în patru subcapitole care vor reda tehnologiile și ideile aplicate în aplicația finală.
- “Testarea aplicației” – se va discuta despre acțiunile întreprinse pentru a asigura rularea aplicației fără greșeli

Teza este expusă pe 40 pagini, conține 26 resurse bibliografice, 12 figuri și 1 tabel.

## 1. PREZENTAREA BIBLIOTECILOR REDUX ȘI REACT

### 1.1. Scurt istoric

În această lucrare se va vorbi despre două biblioteci: React și Redux. Această combinație nu este una aleatorie, Redux-ul fiind prima bibliotecă la care se gandesc developerii ce scriu aplicații pe React cand au nevoie să utilizeze aceeași stare în diferite părți ale aplicației. Redux este o librărie JavaScript open-source pentru managementul stării aplicației. Aceasta librărie a fost creată de Dan Abramov si Andrew Clark în 2015. Dan Abramov a început lucrul la această librărie când se pregătea de conferinta *React Europe 2015*.

Abramov menționează, *“Eu încercam să fac o implementare a Flux conceptului care îmi va permite să schimb logica și îmi va permite să calatoresc în timp cu starea”*.

Abramov a observat similaritatea dintre paternul Flux si a unei funcție de reducere, adăugând:

*“Mă gândeam despre paternul Flux ca o operație de reducere într-o perioadă de timp... store-urile acumulează stare în răspuns la aceste acțiuni”*.

O caracterizare minimală ar suna așa:

- Redux e o librărie pentru aplicațiile JavaScript;
- Redux e un container predictibil pentru stare;
- Redux este liber (*open source*).

Trebuie de menționat că Redux nu are nici o relație cu React. Putem scrie aplicații Redux cu React, Angular, Ember, jQuery, JavaScript vanilla sau orice alta librărie JavaScript. Trebuie de mai menționat încă că Redux se combină în special cu asa librării ca React sau Deku deoarece ele permit descrierea UI ca niște funcții care reprezintă starea, iar Redux emite reînnoirile stării în răspuns la acțiuni.

După apariția Redux-ului, framework-urile ca Angular si Vue au preluat rapid idei din această bibliotecă si au introdus așa alternative ca NgRx pentru Angular și Vuex pentru Vue.

### 1.2. Motivele utilizării React Js

Fiind un instrument modern pentru crearea aplicațiilor, React facilitează crearea părții front-end prin oferirea a unui API relativ simplu ce permite crearea UI-ului într-un mod declarativ prin utilizarea limbajului Javascript sau orice alt limbaj ce se transpilează în Javascript ca Typescript, ReasonML, etc. Principalele caracteristici ale React.Js sunt:

- Este destul de neopinionat și putem alege instrumentele și librăriile adăugătoare preferate, începând de la Typescript terminând cu GraphQL și tot ce rămâne în mijloc: axios, d3, rxjs, etc.

- Este doar Javascript. Când scriem cod în React nu trebuie să ne bazăm pe un limbaj nou de templating sau să căutăm în documentație toată ziua ce metode avem accesibile. Totul e Javascript și chiar dacă dorim să folosim ultimele opțiuni experimentale ale limbajului e posibil prin folosirea a așa instrumente ca webpack, parcel și de genul.

- Învățând React în același timp se învață și Javascript deoarece majoritatea codului ce se scrie e chiar doar Javascript, eliminând cazurile dependenței developerilor de un singur framework pentru rezolvarea tuturor problemelor fără a cunoaște elementele de bază a limbajului în care scriu.

- Curba de învățare este nu chiar atât de acută. Majoritatea developerilor pot crea mici prototipuri în decurs de câteva ore de la prima încercare, deși sunt instrumente care oferă opțiuni mai bune aici: Vue.js fiind unul dintre ele.

- React este popular și folosit în cadrul a multor companii ceea ce oferă o comunitate variată și mare care asigură menținere unei ecosisteme sănătoase cu librării complementare actuale, deși Angular are o popularitate mai mare în aplicațiile enterprise.

### 1.3. Bazele React JS

O aplicație React simplă arată în felul următor:

```
function App () {  
  const [welcomeText, setWelcomeText] = useState("Welcome 1")  
  const handleClick = () => setWelcomeText("Welcome 2")  
  return <button onClick={handleClick}>{welcomeText}</button>  
}  
ReactDOM.render(  
  <App />,  
  document.getElementById("root")  
);
```

Se poate observa că aplicația este compusă dintr-un component, unicul component prezent în exemplu fiind componentul App. Fiecare component poate avea o stare internă, care definește dacă componentul are nevoie de re-render sau nu. În componentul App este prezent o variabilă de stare: welcomeText. Când această variabilă se schimbă atunci React decide că e timpul de făcut un rerender pentru a actualiza informația cu datele noi. În exemplul dat la apăsarea butonului se invocă funcția handleClick care schimbă valoarea variabilei welcomeText cu textul "Welcome 2".

Acest simplu exemplu este destul pentru a înțelege și vedea cum e metoda de a scrie aplicații în React, majoritatea codului fiind combinații a doar diferite componente cu diferite stări.

#### 1.4. **Motivele utilizării Redux**

În ultimii ani cerințele pentru aplicațiile single-page au devenit din ce în ce mai complicate, iar aplicația trebuie să aibă grijă de tot mai multă stare. Starea poate să conțină răspunsurile de pe server cât și date create local. Starea UI la fel e cu mult mai complexă ca înainte, având necesitatea de a dirija așa lucruri ca taburile selectate, spinners, paginația și așa mai departe.

De dirijat așa stare este complicat. Se ajunge la un moment când nu mai este clar ce se întâmplă în aplicație și nu mai stim ce se întâmplă cu starea. Când se ajunge la așa moment e greu de reprodus bug-urile și de le rezolvat.

Redux încearcă să facă mutarea stării predictabilă, adăugând niște restricții când și cum e necesar de actualizat.

#### 1.5. **Bazele Redux-ului**

Redux se ocupă cu interacțiunile stării și poate fi comparat cu un sistem de transmitere a mesajelor. Ca și în OOP, Redux inversează responsabilitatea de control de la apelant la recipient – UI nu manipulează direct starea ci transmite mesaje și lasă starea să reacționeze la ele.

Redux-ul poate fi descris prin 3 principii fundamentale:

- sursă unică de adevăr
- Starea e doar pentru citire
- Schimbările sunt făcute doar cu funcții pure

Sursa unică de adevăr înseamnă că întreaga stare a aplicației este păstrată într-un singur obiect.

Aceasta face mai simplu de inspectat și corectat aplicația iar starea poate fi persistată pentru dezvoltare la rapidă.

Unica metodă de a schimba starea este prin emiterea unei acțiuni ce este un simplu obiect care descrie ce s-a întâmplat. Aceasta asigură că nici vederile, nici callback-urile de rețea nu vor înscrie în stare direct, ci doar dorința lor de a transforma starea. Deoarece toate schimbările sunt centralizate și se execută una câte una în ordine, nu apar probleme de genul când aceeași parte a stării e schimbată în același timp.

Funcțiile pure sunt niște funcții în care valoarea returnată depinde doar de valorile de intrare, fără side efecte. Aceasta asigură că codul rămâne clar și ușor de înțeles. În cazul Redux-

ului, reducerii sunt funcțiile pure ce schimbă starea. Diferența la reduceri e că la input primesc starea anterioară și acțiunea, iar la output returnează starea nouă. Deci, putem observa că Redux-ul e împărțit în trei părți distincte: store, acțiuni și reduceri.

## Acțiuni

Acțiunile sunt datele ce sunt transmise de la aplicație la store. Ele sunt unica sursă de informație pentru store. O acțiune este un obiect JavaScript unde este prezență proprietatea *type*. Un exemplu de acțiune ar fi:

```
{  
  type: 'ADD_USER',  
  name: 'Andrei'  
}
```

În afară de tipul acțiunii, structura ei o decidem noi singuri, deși sunt și recomandări. Aceste recomandări sună în felul următor:

- Tipul este o constantă de tip String;
- Proprietatea *payload* este locul unde punem datele;
- Există proprietatea *error* de tip Boolean dacă este vreo eroare;
- Poate să existe proprietatea *meta* pentru informații adaugatoare ce nu fac parte din *payload*;

De ținut cont de recomandările astea decide fiecare singur, Redux-ul încercă să fie puțin opinionat, permițând dezvoltatorilor să experimenteze.

Pe lângă acțiunile propriu zise sunt și creatori de acțiuni, ce sunt doar niște funcții ce creează acțiuni. Un exemplu simplu ar fi:

```
function addUser(name) {  
  return {  
    type: 'ADD_USER',  
    name  
  }  
}
```

Sau folosind sintaxa JavaScript nouă:

```
const addUser = user => ({  
  type: 'ADD_USER',  
  name  
})
```

## Reduceri

Un reducer decide cum fiecare acțiune afectează starea globală și este constituit dintr-o funcție ce primește 2 parametri: starea curentă și acțiunea iar ceea ce este returnat din funcție va constitui starea nouă cu schimbări.

*(previousState, action) => newState*

Este foarte important ca reducerii să fie întotdeauna funcții pure. Având aceleași argumente, reducerul trebuie să calculeze starea viitoare și să o returneze. Fără surprize. Fără side efecte. Fără API chemări. Fără mutații. Doar o calculare.

Fie este necesar de un reducer care gestionează o listă de nume. O implementare a reducer-ului poate fi în felul următor:

```
const initialState = []
function namesReducer(state = initialState, action) {
  switch(action.type) {
    case 'ADD_NAME':
      return [...state, action.name]
    case 'REMOVE_NAME':
      return state.filter(name => name !== action.name)
    default:
      return state
  }
}
```

În exemplul dat am început prin specificarea unei stări inițiale. Asta este modalitatea de a returna starea inițială a aplicației. Atribuirea state la initialState în parametrii funcției este o metodă relativ nouă în JavaScript de a atribui unui parametru valori inițiale dacă paramentrul este *undefined*. O alternativă la așa atribuire este de controlat tipul parametrului în funcție manual și de returnat starea inițială dacă tipul e *undefined*.

```
if (typeof state === 'undefined') {
  return initialState
}
```

După asta avem un simplu switch statement care în dependență de tipul acțiunii calculează starea nouă și o returnează. În cazul dat sunt prezente trei cazuri posibile: ADD\_NAME, REMOVE\_NAME și default care respectiv adaugă un nume, șterge un nume și returnează aceeași stare. Se poate observa că starea nu este mutată și este folosit *object spread operator* pentru a crea o stare nouă.

## Store

În paragrafele anterioare a fost definit că acțiunile reprezintă ce s-a întâmplat, iar reducerii reînnoirea stării bazându-se pe aceste acțiuni.

Store este obiectul care le adună toate împreună. Store-ul are următoarele responsabilități:

- Ține starea aplicației
- Permite accesul la stare
- Permite ca starea să fie reînnoită
- Înregistrează ascultători(*listeners*)

Este ușor de creat un store dacă este prezent deja un reducer. Fie există un reducer în `reducer.js`

Un exemplu de a crea *store*-ul ar fi:

```
import { createStore } from 'redux'
import reducer from './reducer'
const store = createStore(reducer)
```

E posibil de specificat starea inițială ca al doilea argument la *createStore*. Este folositor pentru a popula starea cu starea de pe server.

```
const store = createStore(todoApp, state)
```

Este necesar de menționat că se folosește doar un singur store în aplicație, ce diferă de flux, unde putem avea mai multe. În cazul *redux*-ului când avem nevoie de împărțit data, vom utiliza compoziția de reduceri. Compoziția de reduceri este așa des utilizată că *Redux*-ul oferă o funcție ajutătoare: *combineReducers*. Un exemplu de așa combinare este:

```
import { combineReducers } from 'redux'
import todos from './todos'
import counter from './counter'
export default combineReducers({
  todos,
  counter
})
```

### 1.6. Integrarea *Redux*-ului în aplicațiile *React*

Cel mai des *Redux*-ul e folosit împreună cu *React*, din cauza asta o explicație cum se integrează aceste două librării este indispensabilă.

Există o întreagă librărie pentru a integra ușor React cu Redux: *react-redux*, și asta este și metoda recomandată și oficială. *React-redux* lucrează cu versiunile de *React*  $\geq 0.14$  și se instalează de obicei prin npm folosind comanda *npm install react-redux*. Asta presupunând că se folosește npm cu un *module bundler* ca Webpack, Browserify, Rollup sau Parcel. Dacă din oarecare motiv nu dorim să folosim npm, putem include biblioteca ca un singur fișier UMD care face un obiect *ReactRedux* accesibil global. Folosirea unui fișier UMD nu este recomandat pentru orice aplicație serioasă deoarece majoritatea librăriilor ce sunt complementare Redux-ului sunt avabile doar pe npm.

După instalare, prima ce este nevoie de făcut este de făcut accesibil tot store-ul Redux-ului în React. Aici vine prima parte majoră a librăriei *react-redux*, care este *Provider*. Providerul este un component React care servește doar un singur scop: de a furniza store-ul la comonentele copii. Deoarece Providerul furnizează store-ul doar la copii lui, un lucru bun este de a include toată aplicația în Provider.

Fie avem deja creat store-ul Redux și avem aplicația React importată `<App />`. Un exemplu de a conecta Providerul cu store la aplicația React ar fi:

```
ReactDOM.render(  
  <Provider store={store}>  
    <App />  
  </Provider>,  
  document.getElementById('root')  
)
```

Acum că a fost furnizat store-ul redux la aplicație, se poate de conectat componentele existente la ea. Componentele nu pot interacționa direct cu store-ul, ele pot doar primi starea necesară sau prin propagarea unei acțiuni să schimbe starea. Cu asta se ocupă *connect*-ul.

Mai jos se poate observa un exemplu de folosire a funcției *connect*.

```
import {connect} from 'react-redux'  
  
const Users = props => {  
  return (  
    <div>  
      { props.users }  
      <span onClick={props.addUser}> Add User </span>  
    </div>  
  )  
}  
  
const mapStateToProps = state => {
```



```

    return {
      users : state.users
    }
  }
  const mapDispatchToProps = dispatch => {
    return {
      addUser : () => dispatch({
        type : 'ADD_USER'
      })
    }
  }
  export default connect(
    mapStateToProps,
    mapDispatchToProps
  )(Users)

```

În exemplul dat, *Users* este un component React. *Connect* este o funcție care primește doi parametri: *mapStateToProps* și *mapDispatchToProps* ce sunt funcții pure, prima primește starea necesară din store, iar a doua definește acțiunile posibile. Funcția *connect* returnează la rândul sau o funcție ce primește un component React ce returnează același component doar că înserat cu datele noi din redux. Deci, se poate observa că *connect* este o funcție de nivel înalt (higher order function), un concept des utilizat în React.

Libraria *react-redux* încurajează separarea componentelor React în două grupuri, componente prezentaționale și componente container. Componentele prezentaționale au grijă cum lucrurile arată (stilurile, markup) și nu știu de existența a redux, iar cele containere au grijă cum lucrurile lucrează (data fetching, reînoirea stării), știu de existența a redux, pot primi stare din redux și pot emite acțiuni. Este un concept simplu ce permite izolarea grijilor și face codul mult mai citeț, unde marcajul și stilurile nu sunt amestecate cu restul lucrurilor.

Lucrurile prezentate pot fi făcute și fara librăria *react-redux*, utilizând *store.subscribe()* doar că nu e metoda recomandată. React Redux face optimizări care sunt greu de facut cu mâna și codul necesar de scris este unul mai succint și mai clar.

## 1.7. Ecosistema Redux-ului

Redux e o librărie mică, dar din cauza unui API bine gândit a dus la creerea unui ecosistem bogat de unelte și extensii. Ele nu sunt obligatorii de utilizat, dar aceste unelte și extensii fac implementarea aplicației să fie mai simplă.

Cele mai utilizate biblioteci utilizate pentru a face Redux-ul mai bun sunt:

- Reselect – se ocupă cu crearea selectorilor memorizați pentru livrarea stării mai eficient
- Normalizr – normalizează JSON-ul după o schema
- Selectorator- o abstractizare peste Reselect pentru selectori des folosiți
- Redux-offline – persistă store-ul pentru aplicații Offline-First, cu support pentru optimistic UI
- Redux-thunk – permite folosirea promisiunilor în acțiuni
- Redux-saga – are grijă de logica async prin folosirea funcțiilor generatoare
- Redux-beacon – integrează Redux-ul cu orice serviciu analitic

Acestea sunt doar cele mai folosite unelte din ecosistema Redux-ului, lista completă este cu mult mai mare.

Middleware-urile în Redux sunt extensii third-pary ce se conectează între propagarea unei acțiuni, și momentul când acțiunea ajunge la reducer. De obicei middleware-urile se folosesc pentru logging, crash reporting și comunicarea cu un API async. Un avantaj al Redux-ului față de alte librării este multitudinea de middlewar-uri disponibile, una din cele mai mari la prezent. Unele din cele mai utilizate middleware-uri sunt `redux-axios-middleware`, `redux-api-middleware`, `redux-socket.io` etc.

Middleware-urile se conectează prin folosirea funcției *applyMiddleware* la crearea store-ului. Un exemplu de conectare a middleware-urilor a fi:

```
import {createStore, applyMiddleware} from 'redux';
import axios from 'axios';
import axiosMiddleware from 'redux-axios-middleware';
const client = axios.create({
  baseURL: 'http://localhost:8080/api',
  responseType: 'json'
});
let store = createStore(
  reducers, //reducerii existenți
  applyMiddleware(
    axiosMiddleware(client)
  )
)
```

## 2. ANALIZA TEHNOLOGIILOR DE MANAGEMENT A STĂRII GLOBALE ÎN JS

În capitolul unu se vorbește despre Redux ca fiind soluția pentru managementul stării globale în Javascript, deși există multe alte soluții atât pentru React cât și pentru restul framework-urilor ce există pentru platforma dată ce rezolvă aceeași problemă de păstrare și management a stării globale a aplicației. În majoritatea cazurilor unghiul de abordare a problemei este diferit pentru fiecare soluție ceea ce crează o multitudine de posibilități pentru fiecare echipă sau individual să își găsească soluția perfectă ce satisface cerințelor necesare. De multe ori aceste soluții, tehnologii, biblioteci noi apărute sunt o evoluție a tehnologiilor vechi existente, rezolvând niște puncte slabe a tehnologiilor anterioare, un exemplu ar fi cum Redux este doar o evoluție a Flux, rezolvând punctele slabe a flux-ului ce sunt complexitatea utilizării mai multe store-uri și complexitatea creării și emiterii acțiunilor. O similară analogie poate fi făcută despre framework-urile AngularJS și Angular, Angular fiind o evoluție ce rezolvă problemele existente în AngularJS, deși aceasă afirmație poate fi contrazisă prin prezentarea argumentului că Angular este o rescriere de la zero și nu are nimic comun cu AngularJS. Folosirea acestor noi soluții apărute ce rezolvă doar niște puncte slabe față de predecesori poate fi un lucru ce nu este destul pentru a investi timpul în învățarea și experimentarea cu acestei tehnologii, motive fiind că programatorii se simt confortabili cu ceea ce deja există, nedorința de a ieși din zona de confort, deadline-urile prezente în companie, convingerea toatei echipe că se merită de încercat, etc., aceste toate puncte fiind valide și cu sens doar că ceea ce pentru o echipă nu prezintă mult pentru altă echipă poate prezenta o îmbunătățire majoră și poate această nouă tehnologie apărută va sta la baza la ceva mai bun în viitor.

Dintre toate soluțiile existente se merită să se vorbească mai în detalii despre trei soluții în particular: Mobx, Vuex și Ngrx, două din ele fiind adaptări și integrări ce se bazează pe ideile precedente existente în paternul flux, iar una din aceste soluții este o încercare de a aduce ceva nou ca rezultat simplificând lucrul programatorilor într-un fel prin eliminarea boilerplate-ului necesar prezent în restul soluțiilor.

Un parametru important ce afectează alegerea unei anumite soluții este popularitatea ei și adoptarea soluției de comunitate. Acest parametru deseori este complicat de obținut și analizat din cauza că nu chiar există un număr exact ce determină popularitatea sau adoptarea, este posibil de comparat numărul de steluțe pe github ale bibliotecilor comparate, este posibil la fel de comparat numărul de descărcări lunare, zilnice sau anuale ale fiecărei librării de pe npm, deși chiar și aceste două metode nu sunt de obicei de ajuns, pot apărea încă mulți alți factori ce pot influența rezultatul final. Un factor exemplu ar putea fii numărul și calitatea articolelor existente

pe web ce vorbesc despre biblioteca anumită, un alt factor poate fi activitatea developerilor pe rețele de socializare sau bloguri personale ce explică de ce a fost aleasă direcția sau conceptele cheie.

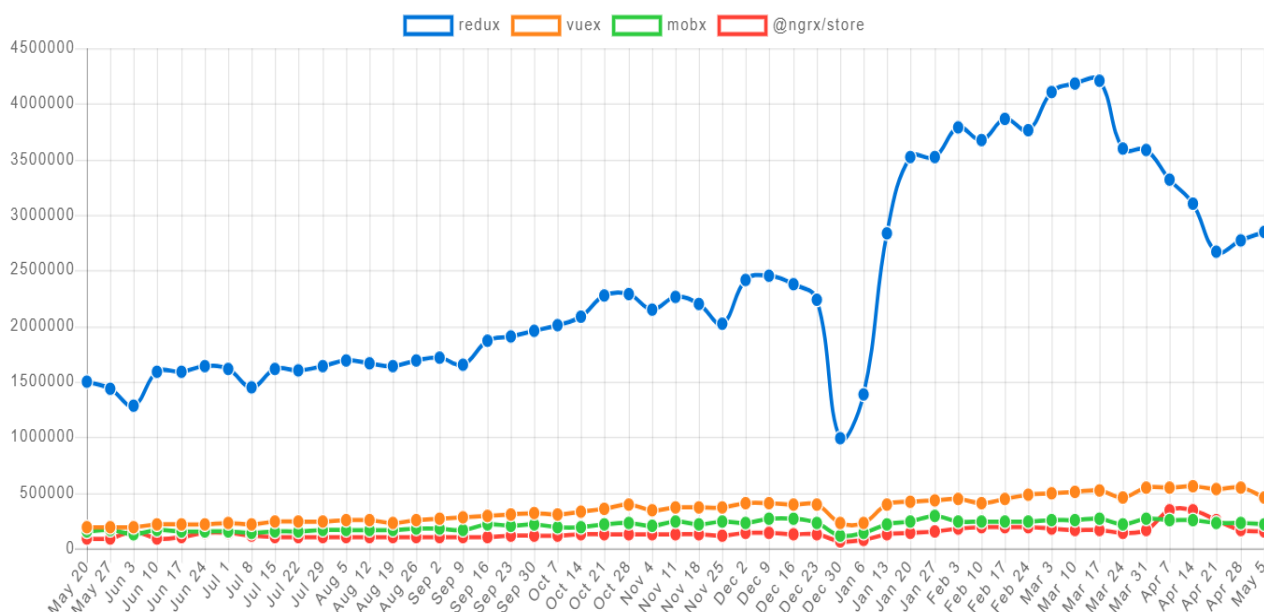


Fig 2.1. Numărul de descărcări anuale ale celor 4 librării discutate

În figura 2.1. se poate observa că dacă de analizat popularitatea librărilor după numărul de descărcări, redux este pe primul loc, fiind descărcat de mai multe ori ca restul librărilor combinate la un loc, deși enviroment-ul în care se necesită utilizarea a unei librării de management a stării joacă un rol important: de exemplu în aplicațiile Angular, Ngrx ar trebui să fie utilizat mai mult, iar în aplicațiile pe Vue, Vuex ar trebui să fie pe primul loc.

## 2.1. Mobx și Mobx-state-tree

Prima librărie ce apare pe gând când e necesar de comparat Redux-ul este Mobx. Mobx este a doua după popularitate librărie de management a stării globale pentru programatorii ce folosesc React deși ca și Redux-ul, Mobx nu depinde de React în nici un fel, lucrând independent și ar trebui să lucreze cu majoritatea soluțiilor ce există pe web la momentul dat, incluzând Angular, Vue și chiar soluții nu chiar așa cunoscute ca Svelte, LitComponent, Preact, etc. Mobx are idei comune cu Redux ca folosirea acțiunilor pentru a schimba starea aplicației și one way data flow. Principala deosebire a MobX față de Redux este că Mobx folosește principiile de object oriented programming iar Redux încearcă să folosească principiile de functional programming. Un store în Mobx arată în felul următor:

```
import { observable, computed, action } from "mobx";  
  
export default class GroceryStore {  
  @observable groceries = [];
```

```

    @action
    add(g) {
        this.groceries.push(g);
    }
    @action
    delete(name) {
        this.groceries.remove(name)
    }
    @computed
    get numOfGroceries() {
        return this.groceries.length;
    }
}

```

Exemplul dat conține toate părțile necesare într-o singură clasă: obiectul propriu zis, acțiunile și valorile calculate. Se poate observa că *groceries* este o valoare *observable* ceea ce face această valoare reactivă, luând exemplul de la rxjs ce permite observarea schimbărilor acestei valori. Se mai poate observa că sunt folosiți decoratorii pentru a specifica tipul valorilor în clasă. Acțiunile sunt niște funcții decorate cu *@action* și au scop de a modifica valorile decorate cu *@observable*. Funcțiile *computed* sunt calea de a păstra numărul minim de valori *@observable*, în schimb calculând valorile noi ce se derivă din starea existentă. Concluzia despre Mobx deci este că Mobx folosește Observables pentru a urmări schimbările din store, schimbarea stării este mai comodă față de redux din cauza eliminării necesității de a ne asigura că starea primită ca parametru nu e schimbată, deși prin folosirea a așa biblioteci ca immer această problemă dispare și în redux. Altă observare e că prin folosirea a Observable și claselor este cu mult mai ușor și clar de învățat cum de utilizat Mobx, iar amplasarea stării aplicației lângă acțiunile ce modifică această stare simplifică semnificativ lucrul pentru orice programator.

Pe lângă toate astea mai există și Mobx State Tree, o librărie complementară la Mobx ce are ca scop de a introduce o structură opionată pentru a crea o experiență consistentă și ajută mult la începerea unui proiect nou prin oferirea a practicilor recomandate ca reguli. Conceptul principal al Mobx-state-tree este conceptul de arbore. Acest arbore consistă din obiecte mutable dar în același timp protejate ce sunt îmbogățite cu informație adăugătoare. În alte cuvinte; fiecare arbore are o formă(informația despre tipuri) și starea(data). Aceasta înseamnă că pentru a face tot să lucreze trebuie de descris cum acest arbore este structurat, prin acest mod Mobx-state-tree va fi capabil să genereze toate limitele necesare și să faciliteze evitarea erorilor de folosirea tipurilor greșite. La fel este posibil de folosit tipurile din Typescript folosind o integrare. Modul de a scrie

valorile observabile, acțiunile și valorile computeate este diferit față de Mobx și poate fi prezentat prin exemplul ce urmează:

```
const GroceryStore = types
  .model("GroceryStore ", {
    groceries: types.array()
  })
  .views(self => {
    return {
      get numOfGroceries() {
        return this.groceries.length;
      }
    }
  })
  .actions(self => {
    return {
      add(g) {
        this.groceries.push(g);
      },
      delete(name) {
        this.groceries.remove(name)
      }
    }
  })
```

Se poate observa că modelul în cazul dat îndeplinește rolul de a specifica valorile observabile, doar că mai apar și tipurile, în cazul dat *groceries* fiind de tip *array*. Desigur dacă se utilizează Typescript este posibil de folosit tipurile implicite ale limbajului. View-urile îndeplinesc rolul de valorile computed iar acțiunile ca și acțiunile în Mobx îndeplinesc aceeași funcție. Folosind această structură e posibil de obține păstrarea stării într-un arbore imutabil dar în același timp permiterea accesării și schimbării obiectelor într-o formă mutabilă și acest lucru este implementat ca la schimbarea oricărei valori într-o formă mutabilă generează automat un *snapshot* ce în schimb formează un arbore nou fără a modifica arborele vechi.

## 2.2. Vuex

Vuex este un patrn de management a stării + librărie special pentru Vue.js și este puternic influențată de paternul Flux. Această librărie oferă toate caracteristicile ce le oferă și

Redux, adăugând încă opțiuni ce fac folosirea acestei librării în aplicațiile Vue.js cu mult mai comode decât restul librăriilor existente. Practic orice interacțiune cu această librărie este mai simplă și mai plăcută decât cu restul soluțiilor existente, acțiunile, store-ul și valorile computed fiind grupate în același loc și prezența unui API excelent pentru a interacționa cu elementele menționate. Fiind influențat de paternul flux, există ideea de one way data flow(fig. 2.1), existând acțiuni care schimbă starea iar starea se propagă la componentele Vue.

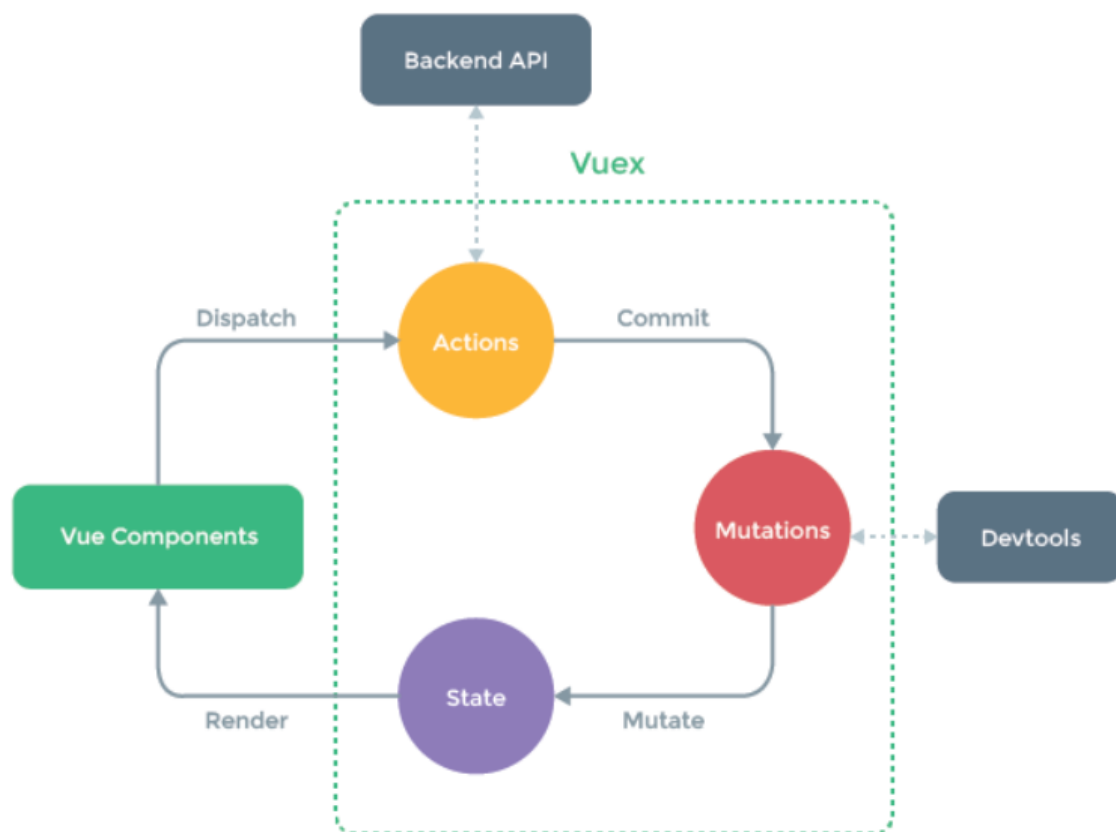


Fig. 2.1. Vuex ilustrat

Se poate observa încă prezența a mutațiilor în fig. 2.1. despre care nu se menționează în paragraful anterior. Mutațiile sunt de fapt unica modalitate de a schimba starea aplicației în Vuex, iar acțiunile doar apelează aceste mutații. Diferența între acțiuni și mutații este că mutațiile nu se ocupă cu business logic ci doar modifică starea. Acțiunile la rândul său implementează acest business logic și sunt capabile să apeleze mai multe mutații dintr-odata făcând codul mai ușor de înțeles și mai clar. Un exemplu de store Vuex ar putea fi:

```

const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment(state) {

```

```

        state.count++
    }
},
actions: {
    increment (context) {
        context.commit('increment')
    }
}
})

```

Se poate observa cum acțiunea *increment* nu schimbă starea direct, ci delegând operația dată către mutația *increment*, acțiunea în cazul dat având posibilitatea de a executa cod async, de exemplu fetch-ingul de date folosind un API careva în timp ce mutația schimbă doar starea sincronizat.

### 2.3. NgRx

NgRx este soluția de păstrare a stării globale pentru aplicațiile Angular, este inspirată de Redux și este pentru Angular ce Vuex este pentru Vue: o soluție opinionată ce încearcă să se integreze perfect cu framework-ul respectiv, în cazul la NgRx este folosită metoda de a interacționa cu componentele și starea ca în Angular, folosind valori observabile și folosind Rxjs. Existența Rxjs nu înseamnă că alte soluții nu există și nu sunt recomandate de folosit, ci doar că scopul principal al Rxjs este de a se integra bine cu Angular, defapt toate soluțiile menționate anterior pot fi folosite cu Angular pentru a provisiona o soluție de păstrare și interacționare a stării globale: Redux, Mobx, Mobx-state-tree fără probleme.

NgRx constă din aceleași acțiuni, reduceri și store doar că implementat diferit. Un exemplu de acțiune arată în felul următor:

```

import { Action } from '@ngrx/store';
export class Login implements Action {
    readonly type = '[Login Page] Login';
    constructor(public payload: { username: string; password: string }) {}
}

```

Se poate observa că orice acțiune trebuie să fie extinsă de la clasa de bază Action, tipul fiind un șir de caractere ce descrie acțiunea mai detaliat și permite asocierea ei cu reducerul în viitor, iar în constructor se specifică forma parametrului ce poate fi folosit pentru propagarea acțiunii. Folosirea claselor pentru a forma o acțiune permite o metodă sigură de a construi acțiunea cu



tipurile necesare. La instanțierea acestei clase vom primi sugestii și corectări adecvate dacă folosim o formă greșită a parametrilor.

Un reducer Ngrx este în tocmai ca un reducer în redux cu foarte mici deosebiri. Un exemplu poate fi:

```
export function reducer(  
  state = initialState,  
  action: Scoreboard.ActionsUnion  
): State {  
  switch (action.type) {  
    case Scoreboard.ActionTypes.IncrementHome: {  
      return {  
        ...state,  
        home: state.home + 1,  
      };  
    }  
  }  
}
```

Se poate observa că pe lângă adăugarea tipurilor din Typescript structura este foarte asemănătoare față de reducerii din Redux, reducerul fiind format dintr-o funcție ce primește ca parametru starea și acțiunea curentă, iar în dependență de acțiune returnează o stare nouă cu modificările noi. Acești reduceri suferă de aceeași problemă ca și Redux-ul prin faptul că starea primită ca parametru nu poate fi schimbată direct, ci o stare nouă trebuie returnată de fiecare dată, deși folosirea la așa biblioteci ca immer reduce complexitatea acestei probleme.

Comparând toate aceste soluții se pot forma niște concluzii, aceste concluzii neavând motivul de a identifica metoda, biblioteca sau ideea cea mai bună, ci de a constata că toate aceste biblioteci comparate au un lucru comun principal: încercarea de a elabora un instrument ce face posibil și ușor de a rezolva o problemă anumită, în cazul nostru problema management-ului stării globale. Nici o metodă nu este perfectă, fiecare având avantaje și dezavantaje iar fiecare echipă este nevoită să analizeze avantajele și dezavantajele prezente și să își formuleze o concluzie care soluție va fi cea mai benefică pentru proiectul în discuție și ce workflow va fi îmbunătățit prin utilizarea soluției anumite.

Toate aceste librării discutate pot fi reprezentate printr-un tabel ce poate fi utilizat pentru a prezenta informația de bază despre fiecare librărie: stelele pe Github care se iau mult în considerare pentru a determina popularitatea, fork-urile ce ajută la determinarea activității

comunității, data de creare pentru a forma o imagine completă și mărimea fiecărei librării ce pentru aplicațiile Front-end joacă un rol important și afectează experiența utilizatorului final:

Tabelul 2.1. Prezentarea Informației generale a librăriilor analizate

	Stele pe Github	Fork-uri	Creat	Mărime
redux	48527	12413	May 30, 2015	2.6KB
mobx	19316	1161	Mar 14, 2015	14.8KB
ngrx	4754	1204	Mar 2, 2017	4.3KB
vuex	20214	6480	Jul 16, 2015	3.1KB

### 3. ELABORAREA APLICAȚIEI "BOOKY"

#### 3.1. Spațiul de lucru Visual Studio Code

Pentru a elabora o aplicație web este necesar de câteva lucruri, cum ar fi:

- Mediu de rulare a aplicației;
- Un redactor de cod modern;
- Cunoștințele necesare.

Visual Studio Code – este un editor de cod liber(open source) pentru dezvoltarea software-ului și are suport inclus pentru JavaScript, TypeScript și Node.js, dar care poate fi folosit și pentru alte limbaje ca C++, C#, Java, Python, PHP, Go, etc.

Visual Studio Code este dezvoltat de Microsoft și lucrează pe Windows, Linux și macOS. Este bazat pe Electron, un framework ce se utilizează pentru rularea aplicațiilor Node.js pe desktop utilizând Blink layout engine, și la moment este unul din cele mai populare editoare de cod.

Posibilitățile de bază:

- Redactor text cu evidențierea sintaxei, funcția de autocompletare;
- Analizează codul și oferă posibilitatea corectării momentane;
- Navigarea rapidă prin ierarhia proiectului și liniilor de cod;
- Debugger;
- Snippet-uri
- Git control integrat
- Customizabil
- Există o multitudine de teme
- Gratis și open-source
- Hot-Keys.

Pentru a pregăti spațiul de lucru vom avea nevoie de unele extensii:

- ESLint;
- Jsdoc;
- DotENV.

Mai avem nevoie de instalat următoarele lucruri:

- Node.js
- Typescript
- Styled-components

- Create-react-app

După instalarea aplicațiilor date se intră în Visual Studio Code și se alege o mapă, iar în mapa selectată se rulează comanda din fig. 3.1.

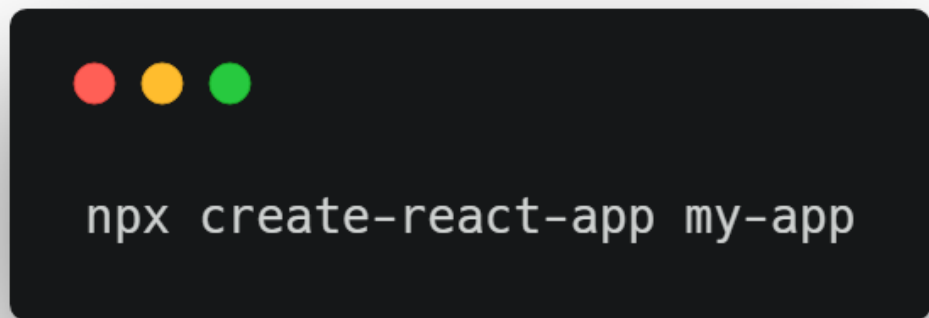
A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The text 'npx create-react-app my-app' is displayed in a light gray monospace font.

Fig. 3.1. Crearea unui proiect nou cu toate setările.

Se poate menționa că pentru executarea acestei comenzi este necesar de npm 5.2+.

O alternativă care lucrează cu versiunile de npm mai vechi ar fi rularea comenzii din fig. 3.2. pentru a instala package-ul global.

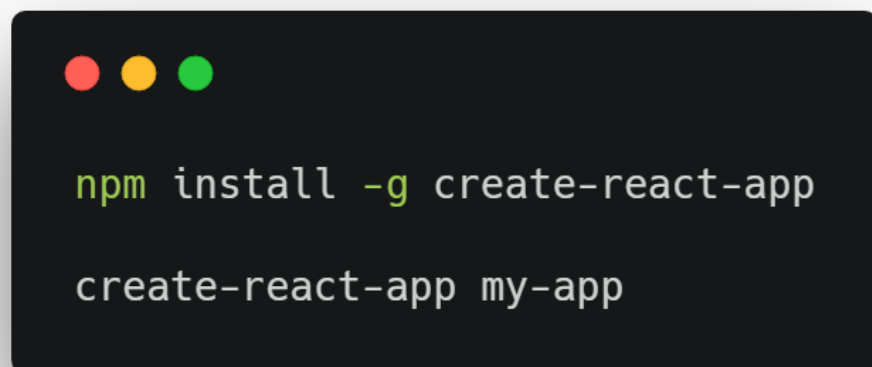
A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. Two lines of text are displayed in a light gray monospace font: 'npm install -g create-react-app' and 'create-react-app my-app'.

Fig. 3.2. Crearea unui proiect nou cu toate setările, comenzi alternative

După executarea comenzii în câteva clipe se vor crea toate dosarele și fișierele necesare, așa ca fișierele de configurare, atât și niște componente React.

### 3.2. Tehnologii necesare la dezvoltarea aplicației

## Node.JS

În majoritatea cazurilor elaborând o aplicație este necesar să ne bazăm pe cod ce a fost scris mai devreme de alți programatori, unde acest cod ne ajută la rezolvarea problemei noastre. Cea mai simplă metodă de a utiliza acest cod e de a utiliza principiul WET: de a copia codul direct în aplicație, deși această metodă este una ce aduce mai multe probleme decât beneficii. Folosind principiul WET apare necesitatea de a altera codul în mai multe locuri de fiecare dată când avem nevoie să schimbăm ceva și rata de greșeli logice e mai mare folosind acest principiu prin necesitatea de a menține cod în mai multe locuri deodată. O opțiune de a evita problemele enumerate e de folosi un alt principiu: principiul DRY. Folosind principiul DRY și grupând codul în module autonome ce rezolvă o problemă anumită elimină problemele enumerate și face mai ușor ce se întâmplă și în aplicația noastră. În Javascript aceste module se numesc package-uri și sunt folosite extensive, fiind normal de a avea zeci, sute sau chiar mii de package-uri într-o singură aplicație. Aceste package-uri lucrează atât pe Node.js cât și în browser, doar că în browser e necesar de folosit un bundler ca webpack sau parcel pentru a crea un bundle: un singur fișier Javascript cu toate modulele adunate pentru ca browserul să le înțeleagă. Aceste module sunt de obicei publicate pe npm ce prescurtarea la node package manager. Npm constituie din o aplicație de consolă și o bază de date online ce conține toată lista de package-uri disponibile. Pentru fiecare aplicație javascript ce folosește npm e necesar de avut un fișier package.json (fig. 3.3) ce definește lista de package-uri necesare și niște informații de baza despre proiect. Acest fișier package.json permite reproducerea și instalarea dependențelor necesare printr-o singură comandă.

```

1  {
2    "name": "prueba",
3    "version": "1.0.0",
4    "main": "index.html",
5    "scripts": {
6      "start": "lite-server"
7    },
8    "author": "",
9    "license": "ISC",
10   "keywords": [],
11   "description": "",
12   "devDependencies": {
13     "lite-server": "^2.4.0"
14   }
15 }
16

```

Fig. 3.3. Exemplu de fișier package.json

### Create React App

Create React App este un package javascript ce ajută enorm la definirea unor configurații și instrumente de bază pentru a începe dezvoltarea unei aplicații React.js. Acest package utilizează sub capotă Webpack, Babel, ESLint, Jest pentru a oferi o configurație care e ușor de instalat, e necesar doar de executat o comandă în consolă și totul va fi setat, una din avantajele care le oferă Create React App este zero configurație, totul fiind configurat de la început și posibilitatea de a face upgrade la versiunile noi cu ușurință din nou fără a schimba ceva configurații, doar executând o comandă în consolă. Această configurare include:

- Suport pentru React.js
- Suport pentru JSX
- Suport pentru ES6+ și Typescript
- CSS Autoprefixat
- Unit Tester integrat
- Live server de development
- Script pentru a face bundle la fișierele javascript, css și imaginilor pentru production cu hashuri și mape sursă
- Service worker implicit cu web app manifest ce satisface tuturor criteriilor la progressive web apps

În Create React App de asemenea este implementat code splitting cu ajutorul la operatorului `import()` ce este la momentul dat in stage 3 proposal și adaugă posibilitatea de a împărți codul în părți ce este descărcat de user în momentul potrivit. Un exemplu de folosirea a `import()` ar fi:

```
import React, { Component } from 'react';
class App extends Component {
  handleClick = () => {
    import('./moduleA')
      .then(({ moduleA }) => {
        // Use moduleA
      })
      .catch(err => {
        // Handle failure
      });
  };
  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Load</button>
      </div>
    );
  }
}
export default App;
```

În acest exemplu doar când se apasă pe buton se începe încărcarea modului A.

Dacă configurarea standardă nu este de ajuns, este posibil întotdeauna de ejectat și de schimbat configurarea cum dorim, pierzând abilitatea de a face ușor upgrade la versiunile noi dar primind control complet asupra configurării noastre. Putem schimba configurarea de a lucra cu server side rendering deși Next.js și Razzle sunt mai optimizate pentru așa ceva, putem face o integrare mai bună cu aplicațiile brownfield, de exemplu de integrat cu aplicațiile existente Django sau Sumphony, deși și în cazul dat alte package-uri ca nwb și Neutrino fac un lucru mai bun. Putem constata în final că Create React App e un package ce elimină configurarea într-un proiect bazat pe React.js și în special e bun pentru aplicațiile greenfield, acele aplicații care sunt elaborate de la început și nu au probleme de integrare cu cod anterior.

## Styled Components

Alegerea unui mod de a stiliza componentele în aplicațiile React este un lucru care orice echipă sau individual ce se ocupă cu dezvoltarea aplicațiilor pe React se confruntă, librăria React fiind neopinionată față de care opțiune este mai bună, existând mai multe soluții. Una dintre soluțiile cele mai populare este Styled Components și este soluția care este utilizată în acest proiect.

Styled Components este o metodă în React de a scrie CSS-in-JS, o metodă de a scrie cod CSS direct în Javascript, iar această metodă oferă următoarele beneficii:

- Izolarea regulilor și selectorilor. CSS are proprietăți care sunt inheritate automat de la părinte. De multe ori acest lucru duce la component ce arată diferit în diferite părți ale aplicației. Prin evitarea acestei inheritanțe e posibil de evitat acest lucru prin crearea unui component ce arată și funcționează la fel în orice context.
- Regulile CSS sunt automat prefixate pentru a suporta browserele vechi
- Ușurința de a transmite variabile din codul javascript către stiluri
- Eliminarea stilurilor nefolosite din buildul final
- Crearea stilurilor în tag de style eliminând necesitatea de a utiliza stilurile inline

Un avantaj major ce poate fi greu de înțeles de la prima vedere este că folosirea a CSS-in-JS ce este la rândul său o metodă de a scrie cod CSS în Javascript face ca codul CSS amestecat în Javascript să fie minim, ce sună contradictoriu dar prin oferirea unui API comod pentru a transmite variabile Javascript ce afectează stilurile generate elimină necesitate de a utiliza stilurile inline. Un exemplu de a afecta stilurile cu o variabilă Javascript fără a folosi Styled Components în aplicațiile React este:

```
function Component(props) {  
  const error = props.error;  
  return (  
    <div  
      className="panel"  
      style={{ backgroundColor: error ? "red" : "green" }}  
    >  
      {props.children}  
    </div>  
  )  
}
```

Se poate observa prezența atributului style chiar în structura html ce face codul greu de înțeles și modificat. Același component poate fi scris în Styled Components ca:



```
const Panel = styled(BasePanel)`
  background-color: ${props => props.error ? "red" : "green"}
`;

function Component(props) {
  return <Panel error={props.error} />
}
```

În acest exemplu se poate observa că toată manipulare cu CSS se petrece în același loc permițând o compoziție de componente ușor de înțeles fără stiluri inline.

Desigur folosirea a CSS-in-JS are și neajunsuri, unele din ele fiind compexitatea și timpul necesar pentru a învăța și convinge echipa că se merită de inclus stilurile în același fișier ca și codul Javascript.

### 3.3. Crearea aplicației Booky

#### Descrierea generală a aplicației Booky

Aplicația Booky este elaborată pentru web în limbajul de programare Typescript, folosind Nest.js, React.js, Mariadb, TypeORM, Docker, Nginx și Let's Encrypt. Aplicația reprezintă un serviciu de pastrare a linkurilor, diferența principală dintre restul aplicațiilor asemănătoare fiind că linkurile adăugate se grupează automat după numele de domeniu. Această aplicație îndeplinește următoarele funcții:

- Autentificarea utilizatorului;
- Afișarea unui landing page care explică de ce utilizatorul ar dori să utilizeze această aplicație
- Afișarea interfeței de adaugare a unui link nou
- Implementarea operațiilor CRUD necesare ca ștergerea linkului sau modificarea lui
- Afișarea grupurilor de linkuri

La prima lansare a aplicației utilizatorul e redirecționat la landing page (fig. 3.4) unde utilizatorul poate lua cunoștință cu posibilitățile aplicației și are posibilitatea de a se loga cu contul Google.

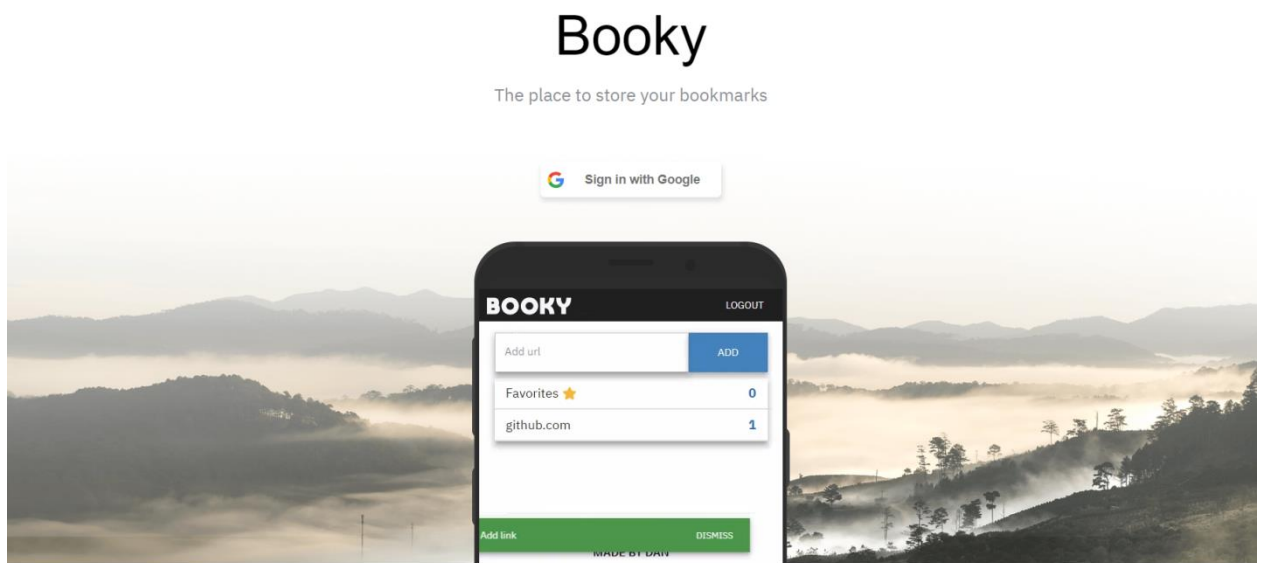


Fig. 3.4. Landing page

Dupa logare, utilizatorul este redirectionat la aplicația propriu zisă, unde îi este prezentată pagina principală (fig. 3.5) unde toate linkurile sunt grupate dupa domeniu, atât și forma de adăugare unui link nou.

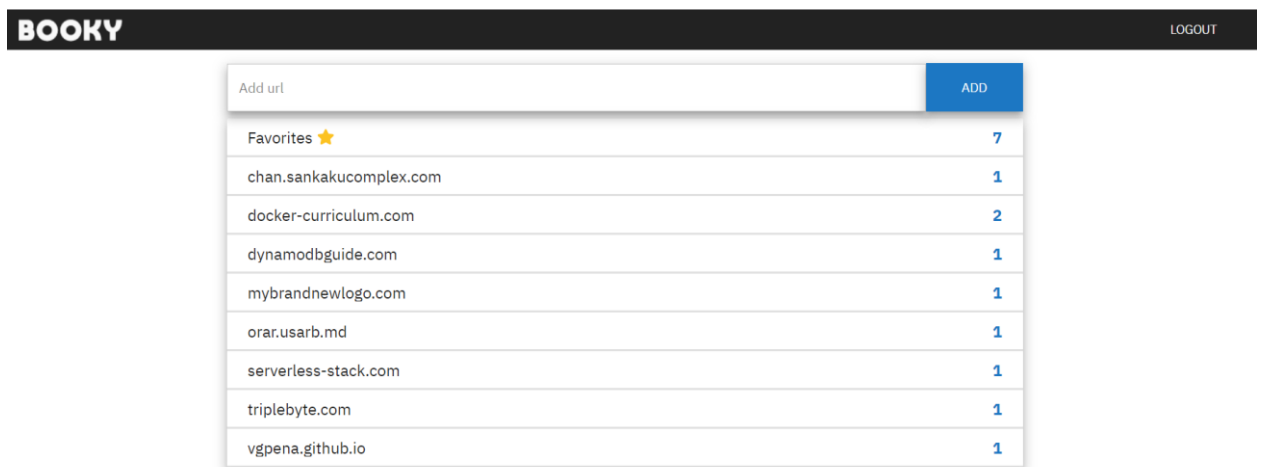


Fig. 3.5 Pagina principală

De aici utilizatorul poate accesa și viziona fiecare grup în particular (fig. 3.6) și viziona meniul de interacțiuni al fiecărui link (fig. 3.7). unde are posibilitatea de a adauga linkul respectiv în favorite și posibilitatea de a șterge linkul.

## ← docker-curriculum.com

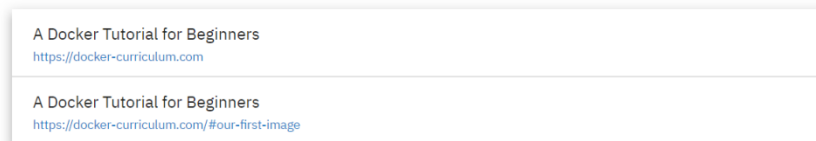


Fig. 3.6. View al unui grup

Orice view prezent ca de exemplu view-ul din figura 3.6 încearcă să reutilizeze datele și componentele la maxim. Datorită că păstrăm toate linkurile în același array în starea globală în redux avem posibilitatea să reutilizăm aceste date în toată aplicația fără a avea nevoie de a face un re-fetch, este de ajuns de extras doar link-urile ce aparțin acestei grupe din toată lista prezentă menționată. În aplicația dată este folosit un filtru obișnuit pentru filtrarea acestor link-uri la conectarea store-ului redux la component, deși dacă este nevoie de performanță maximă este posibil de utilizat o librărie de genul `reselect` ce elimină calculele repetate la fiecare re-render al componentului conectat, calculele având loc doar când chiar lista de linkuri este schimbată.

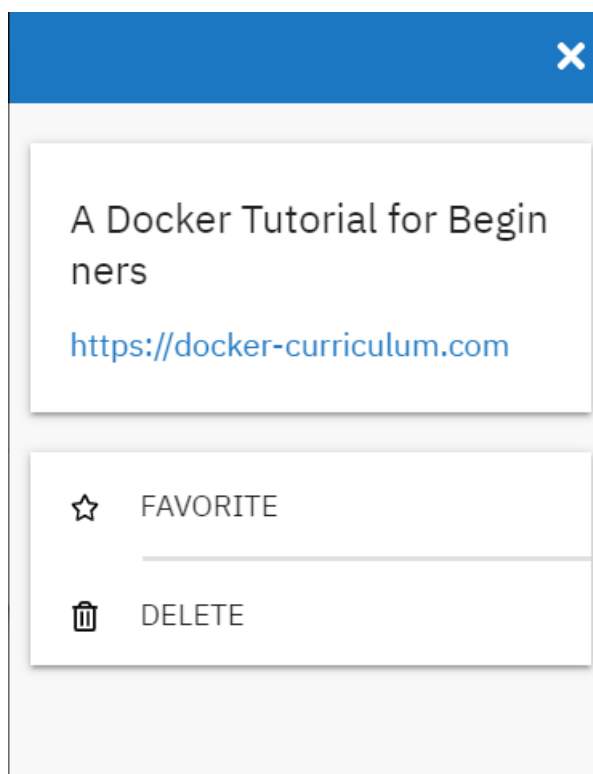


Fig. 3.7 Meniul de interacțiuni al linkului

## Ierarhia aplicației

Datorită utilizării tehnologiilor menționate mai sus se observă că structura este asemănătoare la majoritatea framework-urilor front-end, Angular, Vue.js și Svelte.js având structuri asemănătoare. Aplicația este împărțită în 2 mape: mapa public/ ce conține toate fișierele statice ca index.html, manifest.json, etc. și mapa src/ ce conține restul aplicației. Importanța separării este că webpack-ul nu procesează mapa public, doar copie tot conținutul în mapa de build în timp ce tot din mapa src/ e procesat într-un singur file. Procesarea a tot conținutului mapei src/ într-un singur file e varianta cea mai des întâlnită, deși pot apărea cazuri unde webpack-ul decide că unele filuri din mapa src/ merită de despartit ca imaginile prea mari pentru a fi convertite în base64 sau css generat folosind așa librării ca styled-components, css-modules sau css filuri simple.

După pașii necesari de configurare a mediului de lucru, vom avea afișată structura generată de Create React App. În mediul de lucru configurat ce este Visual Studio Code vom efectua crearea, ștergerea și accesarea de fișiere fără linia de comandă(fig 3.8.).

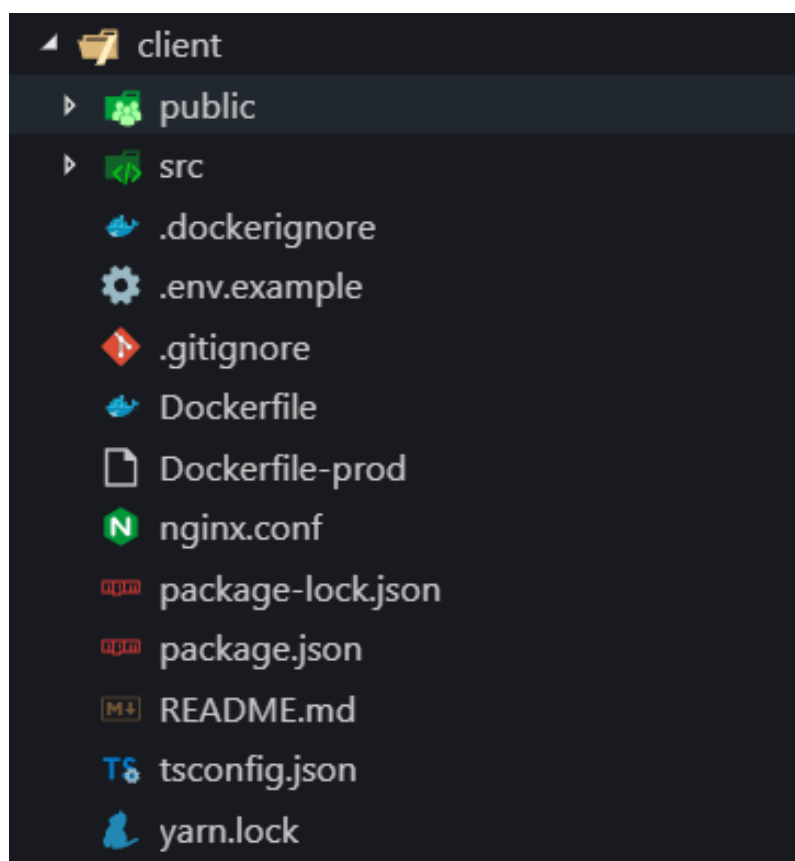


Fig. 3.8. Ierarhia dosarelor în aplicația React.

Conținutul și structura fișierelor în mapa src/ diferă de la proiect la proiect, fiecare echipă ia decizia cum să își structureze fișierele în această mapă, neexistând o soluție perfectă, fiecare soluție având careva dezavantaje. În aplicația dată mai putem observa abundența fișierilor de

configurare ca Dockerfile, .gitignore, tsconfig, etc., un lucru normal în toate proiectele și apare deoarece pe lângă codul propriu zis necesar pentru aplicație mai avem nevoie să folosim instrumente adăugătoare pentru a administra aplicația într-un mod rezonabil, de exemplu folosim docker pentru a face deployment-ul mai ușor pe server și folosim git pentru colaborare.

Mapa src/ la rândul său este constituită din mape ce despart componentele și funcțiile adăugătoare după tip(fig. 3.9). Părțile cele mai importante sunt mapele components și pages, mapa components conținând componente generice și fără business logic care se reutilizează în aplicație iar mapa pages conținând componente care emulează o pagină, combinând componente și adăugând business logic, restul mapelor existente fiind funcții și configurări ajutătoare.

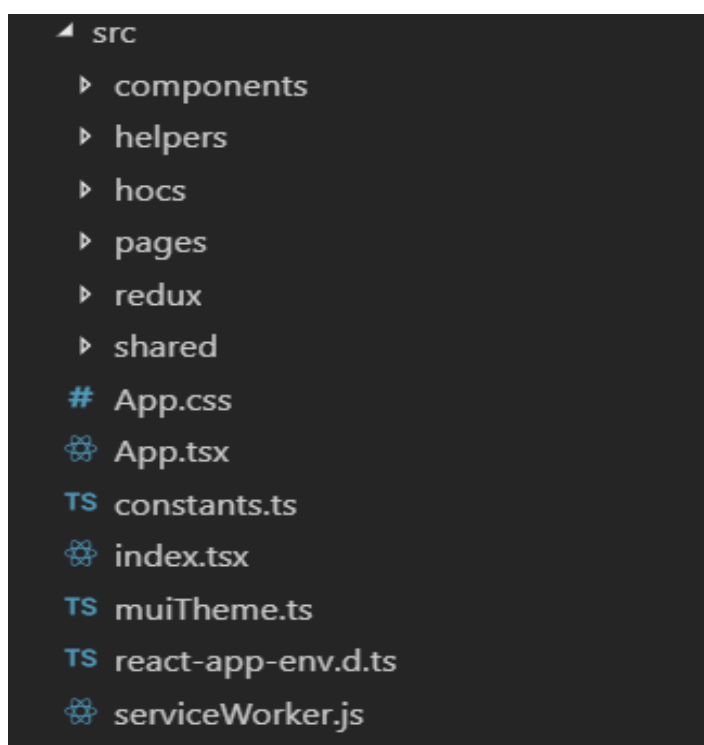


Fig. 3.9. Ierarhia dosarelor în mapa src/.

### Store-ul în aplicație

Este necesar de ales ce date de pastrat global, în cazul dat în redux. Este nevoie de o balanță aici, excesul de date care nu contribuie cu nimic și este local pentru un component anume nu are ce căuta în acest store, datele corecte fiind cele ce sunt întradevăr globale ca de exemplu utilizatorul logat sau niște setări globale și datele ce au nevoie de a fi transmise la mai multe componente dar dintr-un motiv sau altul este incomod. În aplicația dată datele ce sunt globale pentru aplicație sunt: cheia JWT, datele despre utilizator ca de exemplu id lui și statutul la loadingBar.

Loading bar-ul menționat este un middleware redux ce permite de a afișa o bară de încărcare ca pe Youtube ce se activează automat când are loc un fetch de date folosind un API

ceva. Știind că fiecare fetch e o promisiune javascript, middleware-ul își schimbă starea când este o promisiune în așteptare.

Cheia JWT este o metodă de a identifica dacă userul este logat. JWT este un standart care definește o cale de a transmite informativ securizat dintre parteneri folosind un obiect JSON, iar informația poate fi verificată dacă e originală ca rezultat a semnării ei cu o cheie privată. JWT este doar o metodă de a verifica dacă utilizatorul are permisiunile necesare și are neajunsuri ca și oricare alt standart. Unul dintre neajunsurile principale este că e dificil de revocat permisiunile după ce au fost eliberată cheia, necesitând un effort Pentru a păstra datele despre logare la următoarele vizite ale aplicației e posibil de folosit alt middleware numit *redux-persist* ce folosește localStorage pentru a salva și repopula automat datele la redeschiderea aplicației. Ca să fie folosit este necesar de specificat ce date urmează să fie salvate și să specificăm ce modalitate de salvare se va folosi(fig. 3.10).

```
import promiseMiddleware from "redux-promise-middleware";
import { composeWithDevTools } from "redux-devtools-extension";
import { loadingBarMiddleware } from "react-redux-loading-bar";

import { persistStore, persistReducer } from "redux-persist";
import storage from "redux-persist/lib/storage"; // defaults to localStorage for web and
import { combinedReducers } from "../reducers/index";

const persistConfig = {
  key: "root",
  storage,
  whitelist: ["jwt", "user"]
};

const persistedReducer = persistReducer(persistConfig, combinedReducers);

export const store = createStore(
  persistedReducer,
  composeWithDevTools(
    applyMiddleware(promiseMiddleware(), thunk, loadingBarMiddleware())
  )
);

export const persistor = persistStore(store);
```

Fig. 3.10 Redux-persist

Store-ul aplicației este constituit din câmpul jwt, user, groups, links, favoritesCount și loading bar(fig. 3.11). Câmpul *user* este valoarea calculată din extragerea informației din codul JWT, o metodă alternativă și în multe aspecte mult mai bună ar fi de calculat valoarea asta la extragerea codului jwt folosind biblioteci de genul *reselect* care permite definirea unei funcții la extragere care transformă într-un mod sau altul datele, în proces memorizând valoarea inițială și recalculând doar când valoarea inițială se schimbă. Folosirea acestei metode elimină problema

duplicării datelor în store-ul global și elimină problema menținerii acestor două câmpuri sincronizate. Câmpurile *grups* și *links* sunt folosite ca un fel de cache, permițând trecerea de la un view la altul prin afișarea informației primite anterior. Desigur se face un refetch de fiecare dată pentru datele noi, doar că datele își fac update doar când refetch-ul este finisat, în așa mod permițând schimbarea de view-uri și afișarea instanță a datelor deja prezente. Prin mici schimbări această metodă este posibil de facut să lucreze offline, utilizând integrări cu service workers pentru a face o aplicație offline first ce îmbunătățește semnificativ user experience final, ce este un factor important în aplicațiile moderne. Câmpul favoritesCount este o necesitate din cauza cum este necesar de afișat interfața aplicației și felul în care a fost implementat API pe partea backend. O soluție ar fi de a utiliza GraphQL și de a mitiga toate endpoint-urile fixe, deși GraphQL are și el limitările și provocările lui.

```
export interface CombinedReducers {  
  jwt: string | null;  
  user: any;  
  groups: Groups,  
  links: Links  
  favoritesCount: FavoritesCount  
  loadingBar: any  
}
```

Fig. 3.11 Interfața store-ului

### Utilizarea acțiunilor de a face fetch la date pentru a popula componentele și store-ul

O necesitate des întâlnită este fetch-ingul datelor folosind un REST API, păstrarea datelor primite în redux și reprezentarea lor într-un component sau altul. Un component ce demonstrează o metodă de a soluționa aceste probleme este:

```
const Component: React.FunctionComponent<Props> = (props: Props) => {  
  const [startedFetching, setStartedFetching] = React.useState(false);  
  React.useEffect(() => {  
    setStartedFetching(true);  
    if (props.path === "favorites") {  
      props.fetchFavorites();  
    } else {  
      props.fetchLinks(props.group);  
    }  
  })  
}
```

```

    }, []);
...
function mapListItems(links: Array<ILink>) {
    return links.map((link, id) => (
        <LinkListItem
            key={link.id}
            primary={link.title}
            secondary={link.url}
            onClick={onItemClick(link)}
            isFavorite={link.isFavorite}
        />
    ));
}
return (
    <List>{mapListItems(props.links.data)}</List>
);
}
function mapStateToProps(state: CombinedReducers, props: any) {
    if (props.path === "favorites") {
        return {
            links: filterFavoriteLinks(state.links)
        };
    } else {
        return {
            links: filterLinks(state.links, props.group)
        };
    }
}
export const LinksList = compose(
    connect(
        mapStateToProps,
        { fetchLinks, fetchFavorites }
    ),
    withSnackbar
)(Component) as React.FunctionComponent<{ [type: string]: any }>;

```



Se poate observa în exemplul prezentat că se exportă componentul *LinksList* care este un component conectat la store-ul Redux, ce se abonează la câmpul de date *links* cum se poate observa în funcția *mapStateToProps*. Când acest câmp va primi date noi, componentul va ști și va face automat un rerender la component pentru a utiliza datele noi. *FetchLinks* și *fetchFavorites* sunt niște funcții ce îndeplinesc rolul de acțiuni și e nevoie de le inclus ca parametri în funcția *connect* ce le decorează cu o altă funcție numită *dispatch* ce are ca scop anunțarea reduxului că o acțiune a avut loc ca reducerii să poată reacționa. Funcțiile decorate cu *dispatch* vor apărea ca convenție ca *props.funcție* în component ca o convenție, deși API-ul dat nu e cel mai intuitiv și se lucrează la mari schimbări în privința acestui API prin folosirea metodelor noi a utilizării stării în React, Vue, etc.

### 3.4. Testarea aplicației elaborate

Testarea aplicației este procesul de verificare dacă aplicația elaborată satisfacere tuturor criteriilor de calitate și toate acțiunile noi lucrează cum a fost planificat. La găsirea a careva defecte, aceste defecte sunt raportate pe Trello sau pe Github și sânt reproduse și fixate de persoana responsabilă. Testarea este repetată până nu se mai găsesc greșeli.

Pentru a asigura idempotența mediului de rulare a aplicației se folosește Docker, ce oferă același mediu în faza de development cât și în faza de deployment, de obicei fiind un distributiv linux ce conține toate package-urile necesare în sandbox ca node.js, npm, nginx, etc. Folosind Docker se elimină o listă întreagă de probleme și greșeli care pot apărea ca:

- Erorile din cauza folosirii unei versiuni noi a sistemului de operare, care schimbă un funcțional sau API intern, folosind Docker e posibil de specificat versiunea fixă a distributivului folosit.
- Erorile din cauza folosirii unui package de versiune diferită, de exemplu la fiecare versiune noua de Node.js careva API sunt schimbate și nu este rar să apară greșeli stranii și greu de reprodus în aplicațiaa rulată. Docker permite de specificat versiunea exactă a package-ului instalat în distributiv.
- Diferența între sistemele de operare la development și deployment. Pot apărea probleme dacă la development folosit un sistem de operare ca Windows iar la deployment un distirbutiv de linux ca Ubuntu de exemplu. Unele fișiere și librării au nevoie să fie compilate pentru fiecare sistem de operare aparte, uneori fiind chiar librării complet diferite pentru fiecare sistem de operare incompatibil unul cu altul.

Până la deplyment aplicația este testată local pentru a identifica problemele din timp până a ajunge la utilizator, folosind date de test, de obicei folosind o bază de date cu date de test ce simulează situațiile extreme.



## CONCLUZII

Managementul unei stări globale în aplicațiile frontend folosind React.Js a fost întotdeauna o provocare pentru programatori, . În deceniul 2 al secolului 21, tehnologia de management a stării globale numită Redux a încercat să rezolve această mare dilemă, astfel încât folosirea aceloși date în componente diferite, deseori în diferite părți ale DOM-ului nu mai este o problemă așa complicată.

Câteva dintre numeroasele beneficii ale folosirii Redux-ului în aplicațiile React.Js sunt:

- Rularea în orice enviroment: client server și native
- Testarea ușoară prin folosirea unei stări fixe fără mutații.
- Persistența datelor prin folosirea unui loc unic de păstrare a datelor.
- Prezența unui larg repozitoriu de addon-uri și extensii
- Existența a inspectorului ca addon pentru toate browserele populare ce permite de inspectat și analizat starea aplicației
- Actual și folosit în multe aplicații atât mari cât și mici

Librăria Redux, în combinație cu React și-a găsit și se folosește în diverse ii de activitate umană, de la aplicațiile web mici ce deservește câteva sute de persoane pe lună până la aplicațiile ce sunt dezvoltate să fie capabile să deservească de mii de ori mai mulți utilizatori, ca de exemplu aplicațiile ce le dezvoltă Facebook, Airbnb sau Reddit. Folosind și îmbunătățind paternul Flux, librăria dată a captivat interesul individualilor și echipelor ce s-au întâlnit cu problema management-ului stării, atrăgând contribuitori.

Toate obiectivele înaintate pentru îndeplinirea scopului au fost îndeplinite:

- Analiza resurselor informative: literatură de specialitate, comunitățile și tutorialele;
- Analiza generală a bibliotecii Redux și React (metode de utilizare, compabilitate, actualitate)
- Analiza tehnologiilor și soluțiilor alternative alternative
- Proiectarea aplicației de păstrare a linkurilor ce se grupează automat după domeniu
- Testarea și implementarea aplicației elaborate.

La aplicația elaborată Booky a fost folosită librăria react-redux pentru a face o conectare între aplicația React și store-ul pe redux, redux-persist pentru a păstra și rehidrata automat starea în local storage, thunk și redux-promise-middleware pentru a permite lucrul cu operațiile async ce permite lucrul cu serverul folosind un API și loading-bar-middleware pentru a adăuga automat o bară de încărcare de fiecare dată când o operație async este în îndeplinire.

Pentru implementarea părții back-end s-a folosit framework-ul Nest.js, un framework scris pe Typescript ce lucrează pe Node.js ce este inspirat de Angular cu mari asemănări cum lucrurile sunt implementate în dotnet core, iar MariaDb este folosită ca bază de date cu biblioteca TypeORM ce îndeplinește funcția de ORM.

Pentru comoditate este folosit Docker și Docker-compose pentru development și deployment mai ușor, ce este configurat ca să lucreze cu nginx și https folosind linuxserver/letsencrypt.

Aplicația elaborată în combinație cu teza curentă explorează tema management-ului stării globale în aplicațiile Javascript, în special atragând atenția la combinarea între librăriile Redux și React și este utilă pentru programatorii ce doresc să creeze aplicații complexe ce necesită utilizarea a unei forme de management a stării, mai ales pentru programatorii Javascript sau Typescript.

## BIBLIOGRAFIE

1. Alex Bachuk. *An Introduction To Redux* [on-line]. [citat 06.06.2016]. Disponibil: <https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/>
2. Dan Abramov. *The History of React and Flux with Dan Abramov* [on-line]. [citat 01.06.2017]. Disponibil: <http://threedevsandamaybe.com/the-history-of-react-and-flux-with-dan-abramov/>
3. Mark Piispanen. *Modern architecture for large web applications* [on-line]. [citat 04.06.2016]. Disponibil: <https://jyx.jyu.fi/dspace/bitstream/handle/123456789/54129/URN:NBN:fi:juu-201705272524.pdf?sequence=1>
4. Miguel A, Moreno. *A Basic React + Redux introductory tutorial* [on-line]. [citat 03.04.2016]. Disponibil: <https://hackernoon.com/a-basic-react-redux-introductory-tutorial-adcc681eeb5e>
5. Robin Wieruch. *The Road to learn React: Your journey to master plain yet pragmatic React.js*, Germania 2018. pag. 25-50.
6. Alex Banks. *Learning React: Functional Web Development with React and Redux*, United States of America 2017. pag. 56-67.
7. Azat Mardan. *React Quickly: Painless web apps with React, JSX, Redux, and GraphQL*, United States of America 2017, pag. 240-255.
8. Valentino Gagliardi. *React Redux Tutorial for Beginners: The Definitive Guide* [on-line]. [citat 28.01.2019]. Disponibil: <https://www.valentinog.com/blog/redux/>
9. Brian Troncone. *Redux Middleware: Behind the Scenes* [online]. [citat 20.07. 2015]. Disponibil: <http://briantroncone.com/?p=529>
10. Yang Hsing. *Getting Started With React Redux: An Intro* [on-line]. [citat 06.04.2016] Disponibil: <https://www.codementor.io/mz026/getting-started-with-react-redux-an-intro-8r6kurexf>
11. Will Becker. *Redux best practices* [on-line]. [citat 28.07.2015]. Disponibil: <https://medium.com/lexical-labs-engineering/redux-best-practices-64d59775802e>
12. Milo Mordaunt. *Handcrafting an Isomorphic Redux Application* [on-line]. [citat 02.05.2015]. Disponibil: <https://medium.com/front-end-developers/handcrafting-an-isomorphic-redux-application-with-love-40ada4468af4>
13. Zohaib Rauf. *Redux - Calling web service asynchronously* [on-line]. [citat: 21.03.2016] Disponibil: <https://zohaib.me/redux-call-service-async/>

14. Doron Sharon. *Implementing a smart Login Modal with Redux, reselect and ReactJS* [on-line]. [citat 24.10.20.2015]. Disponibil: <https://medium.com/@dorsha/implement-login-modal-with-redux-reselect-and-reactjs-668c468bcbe3>
15. Tero Parviainen. *A Comprehensive Guide to Test-First Development with Redux, React, and Immutable* [on-line]. [citat 10.09.2015]. Disponibil: <http://teropa.info/blog/2015/09/10/full-stack-redux-tutorial.html>
16. André Gardi. *State Management with React Hooks—No Redux or Context API* [on-line]. [citat 09.04.2019]. Disponibil: <https://medium.com/javascript-in-plain-english/state-management-with-react-hooks-no-redux-or-context-api-8b3035ceecf8>
17. James Wright. *Rolling your own Redux with React Hooks and Context* [on-line]. [citat 07.01.2019]. Disponibil: <https://medium.com/yld-engineering-blog/rolling-your-own-redux-with-react-hooks-and-context-bbcea18b1253>
18. Pran B. *React—Redux workflow in 4 steps—Beginner Friendly Guide* [on-line]. [citat 07.03.2019]. Disponibil: <https://hackernoon.com/https-medium-com-heyphb-react-redux-workflow-in-4-steps-beginner-friendly-guide-4aea9d56f5bd>
19. Roman Nguyen. *Architecting your React application. The development and business perspective of what to be aware of* [on-line]. [citat 14.01.2019]. Disponibil: <https://blog.usejournal.com/architecting-your-react-application-5af9cd65a891>
20. Ryan Florence. *The Suspense is Killing Redux* [on-line]. [citat 05.09.2018]. Disponibil: <https://medium.com/@ryanflorence/the-suspense-is-killing-redux-e888f9692430>
21. Dan Abramov. *Writing Resilient Components* [on-line]. [citat 16.04.2019]. Disponibil: <https://overreacted.io/writing-resilient-components/>
22. Nazare Emanuel Ioan. *How to use Redux in ReactJS with real-life examples* [on-line]. [citat 11.12.2018]. Disponibil: <https://medium.freecodecamp.org/how-to-use-redux-in-reactjs-with-real-life-examples-687ab4441b85>
23. Lusan Das. *The best way to architect your Redux app* [on-line]. [citat 19.06.2018]. Disponibil: <https://medium.freecodecamp.org/the-best-way-to-architect-your-redux-app-ad9bd16c8e2d>
24. Albert Parron. *Demystifying Redux* [on-line]. [citat 23.08. 2018]. Disponibil: <https://apiumhub.com/tech-blog-barcelona/demystifying-redux/>
25. Soham Kamani. *React-redux "connect" explained* [on-line]. [31.04.2017]. Disponibil: <https://www.sohamkamani.com/blog/2017/03/31/react-redux-connect-explained/>
26. Sooraj Chandran. *MobX vs Redux with React: A noob's comparison and questions* [on-line]. [01.01.2017]. Disponibil: <https://codeburst.io/mobx-vs-redux-with-react>