MINISTERUL E­­­DUCAŢIEI AL REPUBLICII MOLDOVA

UNIVERSITATEA DE STAT „ALECU RUSSO” DIN BĂLŢI

FACULTATEA DE ŞTIINŢE 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:

Șeful de catedră\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

”\_\_” \_\_\_\_\_\_\_\_\_\_201\_\_\_

**Graficul calendaristic de executare a tezei de an**

Tema tezei de licență **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 licență la Catedra de matematică și informatică\_

„\_\_\_\_” 2018

**Etapele executării tezei de licență:**

|  |  |  |
| --- | --- | --- |
| 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 procedeelor 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, 2018

*Structura tezei:* introducere, două capitole, concluzii generale, bibliografie din 4 titluri, 24 pagini, 17 figuri

*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 bibliotecii redux, analiza posibilităților ei 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. Proiectarea aplicației;
4. Elaborarea propriei aplicații din domeniul cosmosului;
5. Testarea aplicației.

*Noutatea și originalitatea științifică* a lucrării: biblioteca redux a apărut în anul 2015 primind rapid un suport puternic de la comunitate, în 3 ani evoluând rapid. La moment biblioteca nu este cercetată 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 biblioteca redux.

*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.

**ANNOTATION**

*english*

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.

**CUPRINS**

[INTRODUCERE 4](#_Toc1821354)

[1. PREZENTAREA BIBLIOTECII REDUX 6](#_Toc1821355)

[1.3. Bazele Redux-ului 7](#_Toc1821356)

[1.3.1. Acțiuni 7](#_Toc1821357)

[1.3.2. Reduceri 8](#_Toc1821358)

[1.3.3. Store 9](#_Toc1821359)

[1.4. Integrarea Redux-ului în aplicațiile React 11](#_Toc1821360)

[1.5. Ecosistema Redux-ului 13](#_Toc1821361)

[2. TEHNOLOGII NECESARE LA DEZVOLTAREA APLICAȚIEI 14](#_Toc1821362)

[2.1. Mediul de rulare a aplicației 14](#_Toc1821363)

[2.2. 16](#_Toc1821364)

[3. ELABORAREA APLICAȚIEI ”BOOKY” 16](#_Toc1821365)

[3.1. Spațiul de lucru Visual Studio Code 16](#_Toc1821366)

[3.2. Mediul de rulare 18](#_Toc1821367)

[3.3. Asd 18](#_Toc1821368)

[3.4. sdfsf 18](#_Toc1821369)

[CONCLUZII 19](#_Toc1821370)

[BIBLIOGRAFIE 20](#_Toc1821371)

INTRODUCERE

La momentul actual, nu este rezolvata problema de păstrare a stării în partea front end si există mai multe medode de abordare, fiecare metodă diferențiindu-se prin idei si problemele ce încearca 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 sa impartă 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 si React sunt niște tehnologii actuale, fiind unele din cele mai utilizate tehnologii in 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 implimentării Redux-ului și diferitor tehnologii în atingerea scopului comun – crearea aplicației *Booky.* Partea teoretică conține în sine cunoștințele necesare despre biblioteca Redux , 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 intro aplicație bazata pe React js. Se va examina un exemplu care conține în sine toate principiile, standardele de programare și lucrul cu diferite module abibliotecii Redux.

*Scopul* tezei de an constă în cercetarea bibliotecii Redux, analiza posibilităților analiza posibilităților combinarii 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, standartelor și principiilor structurare);
3. Proiectarea arhitecturii și logicii aplicației (ex componente in React);
4. Elaborarea propriei aplicații din domeniul cosmosului;
5. Testare aplicației.

Teza e structurată în trei capitole:

Primul capitol “Prezentarea bibliotecii Redux” conține cinci subcapitole:

* “Scurt istoric” – va conține o prezentare succintă a istoriei bibliotecii Redux;
* “Motivele utilizării Redux” – se va vorbi despre avantajele folosirei Redux-ului
* “Bazele Redux-ului” – începînd 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 si React;
* “Ecosistema Redux-ului” – subcapitolul dat va conține folosirea extensiilor pentu Redux;

În al doilea capitol “Elaborarea aplicației Scroll The Universerse” vor fi analizați pașii de elaborare a aplicației web. Capitolul va conține două 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.
* “Crearea aplicației Scroll The Universe” – subcapitolul dat va fi expus în șapte subcapitole care vor reda tehnologiile și ideile aplicate în aplicația finală.

Teza este expusă pe 24 pagini, conține 4 resurse bibliografice și 8 figuri.

1. PREZENTAREA BIBLIOTECII REDUX
   1. Scurt istoric

În această lucrare se va vorbi despre o bibliotecă numită Redux. 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 librarie când se pregătea de conferinta *React Europe 2015*.

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

Abramov a observat similaritatea dintre paternul Flux si a unei functie 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 aşa:

* 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 reînnoirele 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. Motivele utilizării Redux

În ultimii ani cerințele pentru aplicațiile single-page au devenit din ce în ce mai complicate, iar codul nostru trebuie sa aibă grija 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 asa mai departe.

De dirijat așa stare e complicat. Se ajunge la un moment când nu mai e clar ce se întâmpla 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 faca mutarea stării predictabilă, adăugând niște restricții când și cum e necesar de actualizat.

* 1. Bazele Redux-ului

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

Redux-ul poate fi descris prin 3 principii fundamentale:

* sursă unică de adevăr
* Starea e doar pentru citire
* Schimbările sunt facute 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 si corectat aplicația iar starea poate fi persistată pentru dezvoltare lai 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 in 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 aceași parte a stării e schimbată în același timp.

Funcțiile pure sunt niște funcții în care valoarea returnată depinde doar de valorile de intrare, fara side efecte. Aceata asigură că codul rămâne clar si ușor de înțeles. În cazul Redux-ului, reducerii sunt funțiile pure ce schimbă starea. Diferența la reduceri e că la input primesc starea anterioară și actiunea, iar la output returnează starea noua. Deci, putem observa ca Redux-ul e împarțit in trei părți distincte: store, acțiuni și reduceri.

* + 1. Acțiuni

Acțiunile sunt datele ce sunt transmise de la aplicație la store. Ele sunt unica sursă de informație pentru store. O actiune este un obiect JavaScript unde este prezenă 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 recomandari sună în felul următor:

* Tipul este o constantă de tip String;
* Proprietatea *payload* este locul unde punem datele;
* Există proprietatea *error* de tip Boolean daca 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 si creatori de acțiuni, ce sunt doar niște funcții ce creaza acțiuni. Un exemplu simplu ar fi:

*function addUser(name) {*

*return {*

*type: ‘ADD\_USER’,*

*name*

*}*

*}*

*Sau folosind sintaxa JavaScript noua:*

*const addUser = user => ({*

*type: ‘ADD\_USER’,*

*name*

*})*

* + 1. Reduceri

Reducerii specifică cum starea aplicației se schimbă în răspuns la acțiunile trimise la *store.* Un reducer este o funcție pura care primește un starea precedentă și o acțioune, și returneaza starea următoare.

*(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. Fara side efecte. Fara API chemări. Fără mutații. Doar o calculare.

Fie avem nevoie de un reducer care gestionează o listă de nume. O implementare a reducerului 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 șansa noastră de a returna starea inițială a aplicației. Atribuirea state la initialState în parametrii funcției e o metodă relativ nouă in 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 nostru avem trei cazuri posibile: ADD\_NAME, REMOVE\_NAME si default care respectiv adaugă un nume, șterge un nume și returnează aceeași stare. Observați că noi nu mutăm starea ci folosim *object spread operator* pentru a crea una noua.

* + 1. Store

În paragrafele anterioare am definit că acțiunile reprezintă ce s-a întâmplat, iar reducerii reînoirea 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 reînoită
* Înregistrează ascultători(*listeners*)

Este ușor de creat un store dacă avem deja un reducer. Fie avem 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.* E folositor pentru a popula starea cu starea de pe server.

*const store = createStore(todoApp, state)*

*W*

Mai trebuie de menționat că avem doar un singur store în aplicație, ce diferă de flux, unde putem avea mai multe. În cazul redux-ului când avem nevoie să inpărțim data vom utiliza compoziția de reduceri. Compoziția de reduceri e 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. Integrarea Redux-ului în aplicațiile React

Cel mai des Redux-ul e folosit împreună cu React, din cauza asta e necesar de vorbit mai în detalii cum de le integrat.

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* >= 0.14 și se instalează de obicei prin npm folosind comanda *npm install react-redux*. Asta presupunând că folosim 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 aviabile doar pe npm.

După instalare, prima ce e nevoie să facem e să facem accesibilă tot store-ul Reduxu-lui în React. Aici vine prima parte majoră a librăriei *react-redux*, care este *Provider*. Providerul este un component React care servește doar o singur scop: de a furniza store-ul la comonentele copii. Deoarece Providerul furnizeaza store-ul doar la copii lui, un lucru bun e de a include toată aplicația în Provider.

Fie avem deja creat store-ul Redux si avem aplicația react inportată *<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 am furnizat store-ul redux la aplicația noastră, putem conecta componentele noastre 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 putem vedea un exemplu de folosire a 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 primeste doi parametri: *mapStateToProps* și *mapDispatchToProps* ce sunt funcții pure, prima primește starea necesară din store, iar a doua definește actiunile 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, putem observa ca connect e 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. E 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, utilizînd store.subscribe() doar că nu e metoda recomandată. React Redux face optimizări care sunt greu de facut cu mâna și codul necesar de scris e unul mai succint și mai clar.

* 1. Ecosistema Redux-ului

Redux e o librărie mică, dar din cauza unui API bine gândit a dus la creearea 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 creearea 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 complete e 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 creearea store-uli. Un exemplu de conectare a middleware-ulilor 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 noștri

applyMiddleware(

axiosMiddleware(client)

)

)

1. TEHNOLOGII NECESARE LA DEZVOLTAREA APLICAȚIEI
   1. Mediul de rulare a aplicației

Exista 3 feluri de medii de rulare a aplicației:

* Sistemul de operare host – sistemul de operare ce e instalat ca sistemul de bază pe calculator si comunică cu partea hardware direct, fară virtualizări și folosirea a hypervisore.
* Mașini virtuale – emularea la un sistem de operare ce rulează complet izolat, dublând toată funcționalitatea unui întreg sistem de operare.
* Containere – rularea aplicațiilor intr-un mediu sandbox fără penalitatea mașinilor virtuale

În ultimii ani containerele au primit o popularitate imensă, rezolvând problema cu inconsistența, ce sunt prezente în rularea aplicațiilor pe sistemele de operare host și rezolvând problema utilizării a unui număr mare de resurse prezente în rularea aplicațiilor în mașinile virtuale. O implementare populară a conceptelor de containere e Docker. Docker permite folosirea unui mecanizm de izolare si de abstractizare a aplicației prin creearea a mai multor containere ce se comportă predictabil de pe un sistem pe altul, permițând un deployment usor și predictabil fără surprize. Docker lucrează pe toate sistemele de operare, în special el lucrează foarte bine pe Linux si Mac os din cauza că docker-ul foloseste intensiv API-urile low level din linux pentru a izola procesele. Pe Windows pentru a lucra cu cu Docker e necesar de activat opțiunea ”Hyper-V Containers” ce e un hypervisor nativ care permite crearea mașinilor virtuale în Windows și e aviabil doar începând cu versiunile de Windows Pro.

Pentru a lucra cu Docker avem nevoie de două tipuri de file-uri prezente în aplicația noastră:

* Dockerfile-uri
* Docker-compose

Dockerfile-urile constituie instrucțiile necesare pentru a rula un container. E necesar de avut un Dockerfile pentru fiecare parte a aplicației: un exemplu bun fiind având un Dockerfile aparte pentru baza de date, un Dockerfile pentru partea server și un Dockerfile pentru partea client. Un alt exemplu bun ar fi dacă aplicația noastră constă din microservicii sa avem un Dockerfile pentru fiecare microserviciu. Dockerfile-ul e un file text ce conține o listă de comenzi necesare pentru a creea o imagine. Comenzile sunt aproape identice cu cele din linux ceea ce ne permite să nu învățăm o sintaxă nouă. E o metodă simpla de a automatiza procesul de creeare. Un exemplu de Dockerfile e:

FROM ubuntu

RUN echo "deb http://us.archive.ubuntu.com/ubuntu/ precise universe" >> /etc/apt/sources.list

RUN apt-get -y update

RUN apt-get install -y g++

RUN apt-get install -y erlang-dev erlang-manpages erlang-base-hipe erlang-eunit erlang-nox erlang-xmerl erlang-inets

RUN apt-get install -y libmozjs185-dev libicu-dev libcurl4-gnutls-dev libtool wget

RUN cd /tmp ; wget http://www.bizdirusa.com/mirrors/apache/couchdb/source/1.3.1/apache-couchdb-1.3.1.tar.gz

RUN cd /tmp && tar xvzf apache-couchdb-1.3.1.tar.gz

RUN apt-get install -y make

RUN cd /tmp/apache-couchdb-\* ; ./configure && make install

RUN printf "[httpd]\nport = 8101\nbind\_address = 0.0.0.0" > /usr/local/etc/couchdb/local.d/docker.ini

EXPOSE 8101

CMD ["/usr/local/bin/couchdb"]

În exemplul dat sunt arătate instrucțiunile necesare pentru a instala și rula o bază de date numită “Couch DB”. Sunt folosite trei comenzi de baza: FROM, RUN, EXPOSE și CMD.

Putem înțelege ușor citind aceste comenzi că majoritatea comenzilor sunt instrucțiuni din linux, primul și ultimele două rânduri fiind instrucțiuni noi, FROM ubuntu însemnând că ne bazăm pe imaginea de bază cu ubuntu instalat, EXPOSE 8101 însemnând permiterea accesării aplicației ce rulează pe portul dat din afară, iar CMD ["/usr/local/bin/couchdb"] fiind comanda ce se apelează la rularea imaginii.

Pe lăngă Dockerfile-uri e recomandat de avut și un file docker-compose.yml la baza aplicației. Acest file e un file de configurație care unește Dockerfil-urile într-un tot întreg și permite configurarea a tuturor serviciilor în același loc. Un exemplu de docker-compose.yml file este:

version: '2'

services:

web:

build: .

ports:

- "5000:5000"

volumes:

- .:/code

networks:

- front-tier

- back-tier

redis:

image: redis

volumes:

- redis-data:/var/lib/redis

networks:

- back-tier

volumes:

redis-data:

driver: local

networks:

front-tier:

driver: bridge

back-tier:

driver: bridge

* 1. Biblioteci și Framework-uri

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 in 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 necesetiatea 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 întro 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. 2.1) ce definește lista de packag-uri necesare și niște informații de baza despre proiect. Acest file package.json permite reproducerea și instalarea dependențeleor necesare printr-o singură comandă.



Fig. 2.1. Exemplu de fișier package.json

* + 1. Create React App
    2. Styled Components

1. ELABORAREA APLICAȚIEI ”BOOKY”
   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-ul 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 e 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 popular editore 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;
* Debbugger;
* 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
* 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:



Fig. 2.1. Crearea unui proiect nou cu toate setările.

Putem menționa că pentru executarea acestei comenzi avem nevoie de npm 5.2+.

O alternativă care lucrează cu versiunile de npm mai vechi ar fi rularea comenzii



Fig. 2.2. Crearea unui proiect nou cu toate setările, comenzi alternative

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

* 1. Creearea aplicației Booky
     1. Ierarhia aplicației
     2. Comunicarea parții Back-end cu partea Front-end
     3. Acțiunile Redux
     4. Reducerii Redux

CONCLUZII

Lucrarea dată reprezintă o cercetare a bibliotecii Redux și utilizarea ei în cadrul aplicațiilor React. Astfel au fost punctate particularitățile librăriei, referințe la apariția sa istorică și prezentarea posibilitaților acesteia. Redux-ul este librărie JavaScript open-source pentru managementul stării aplicației.

Redux-ul se axează pe folosirea a următoarelor principii:

* Сât mai des posibil să ne focusăm la scrierea componentelor funcționale, iar deoarece aceste componente sunt pure și se vor returna aceleași reprezentări UI
* Întreaga stare a aplicației se păstrează intr-un singur store, ca un obiect JavaScript unde fiecare cheie e o parte distinctă a stării aplicației
* Acțiunile sunt declarative, ele descriu ce poți face în aplicație, nu cum de făcut. Acțiunile sunt date pure.
* Reducerii iau partea în care se descrie cum de făcut. Ei sunt niște funcții pure careiau starea curentă și acțiunea și returnează starea nouă.

În fine, Redux-ul ajută la crearea aplicațiilor ce se comportă consistent, rulează în diferite medii(client, serve, nativ) și care sunt ușoare de testat.

BIBLIOGRAFIE

1. Alex Bachuk*.* „An Introduction To Redux”. Publicat în Smashing Magazine 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] Disponibil: http://threedevsandamaybe.com/the-history-of-react-and-flux-with-dan-abramov/
3. Mark Piispanen „Modern architecture for large web applications” [on-line] https://jyx.jyu.fi/dspace/bitstream/handle/123456789/54129/URN:NBN:fi:jyu-201705272524.pdf?sequence=1
4. Miguel A, Moreno „A Basic React + Redux introductory tutorial” [on-line] https://hackernoon.com/a-basic-react-redux-introductory-tutorial-adcc681eeb5e