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 AN**

**Autor:**

Student al grupei IS31Z

Popa Dan

\_\_\_\_\_\_\_\_\_\_\_\_

**Conducători științifici:**

Mircea PETIC

dr., conf.

\_\_\_\_\_\_\_\_\_\_\_\_

Sergiu CHILAT

lector univ., magistru

\_\_\_\_\_\_\_\_\_\_\_\_

**BĂLȚI, 2018**

Controlată:

Data \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Conducător științific: Mircea Petic, (dr., conf.)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura)

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 an. Bălți, 2018

*Structura tezei:* introducere, trei capitole, concluzii generale, bibliografie din 20 titluri, 42 pagini, 17 figuri și un 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 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**

Țurcanu Marius-Dan

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

Bachelor thesis. Bălți, 2018

*Structure of thesis:* Introduction, 3 chapters, general conclusions and recommendations, bibliography of 20 titles, 42 pages of basic text, 17 figures, 1 table.

*Keywords:* Standards, paradigms, application in the field of articles (news), Ruby programming language, Ruby on Rails framework, MVC, Active Record, database.

*Area of study:* ruby technologies to develop web applications.

*The purpose of the research:* consists in analyzing the possibilities of the Ruby language, its Ruby on Rails framework, and the development of a web application with articles.

*Objectives:*

1. Analysis of information resources: specialized literature, communities and tutorials;
2. General analysis of the possibilities of Ruby on Rails (technologies, standards and programming principles);
3. Design the architecture and logic of the application (eg database);
4. Develop your own application in the field of articles;
5. Testing the application.

*Novelty and scientific originality*: although the Ruby language appeared in 1994, a more pronounced development took place in recent years, due to the emergence of the Ruby on Rails framework, which at the moment is not sufficiently researched, and in Romanian there is practically no documentation.

*Theoretical significance:* due to the fact that there is no documentation in Romanian, the theoretical support of this paper can be used by programmers who wants to study the Ruby language.

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

Anexa 1. Lista de abreviaturi

RoR – Ruby on Rails;

POLS – Principle of Least Surprise;

DRY – Don’t Repeat Yourself;

CoC – Convention over Configuration;

WeT – Write Everything Twice;

MVC – Model View Controller;

ORM – Object / Relational Mapping;

CRUD – Ceate, Read, Update, Delete;

UML – Unified Modeling Language;

AR – Acive Record;

BD – Baze de Date;

IDE – Integrated Development Enviroment;

URN – Uniform Resource Name.

**CUPRINS**

[INTRODUCERE 4](#_Toc512012423)

[1. PREZENTAREA BIBLIOTECII REDUX 6](#_Toc512012424)

[1.2. Motivele utilizării Redux 6](#_Toc512012425)

[1.3. Bazele Redux-ului 7](#_Toc512012426)

[1.4. Argumentarea alegerii limbajului Ruby ca instrument pentru dezvoltarea aplicațiilor 10](#_Toc512012427)

[1.5. Framework Ruby on Rails 12](#_Toc512012428)

[2. TEHNOLOGII NECESARE LA DEZVOLTAREA APLICAȚIEI 14](#_Toc512012429)

[2.1. Standartele DRY și CoC 14](#_Toc512012430)

[2.2. Biblioteci și Framework-uri 15](#_Toc512012431)

[2.3. Landing-Page 16](#_Toc512012432)

[2.4. Lucrul cu baze de date 18](#_Toc512012433)

[3.1. Spațiul de lucru RubyMine 24](#_Toc512012434)

[3.2. Crearea aplicției Marius Review Site 26](#_Toc512012435)

[CONCLUZII 37](#_Toc512012436)

[BIBLIOGRAFIE 39](#_Toc512012437)

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.

În această lucrare se va pune accentul pe aplicațiile web care sunt populare la momentul de față. Trecerea de la aplicațiile desktop la cele web este o tendință actuală, iar popularitatea aplicațiilor web demonstrează posibilitatea transferului aplicațiilor desktop pe platformele web.

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 imparta 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 *Scroll The Universe.* 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 capitolul doi “Tehnologii necesare la dezvoltarea aplicației” sunt analizate tehnologiile, bibliotecile, standardele utilizate în dezvoltarea aplicațiilor pe platforma Ruby on Rails. Capitolul va cuprinde patru subcapitole:

* “Importanța conceptelor programării functionale in creeare aplicațiilor Redux”– subcapitolul dat va conține descrierea mai multor tehnologii aplicate în versiunea nouă a framework-ului.
* “Standardele DRY și COC” – standarde des întîlnite în lumea programării și desigur în RoR;
* “Biblioteci și Framework-uri” – se va discuta despre bibliotecile standarte ale RoR-ului și cele externe care pot ușura procesul de dezvoltare a aplicației;
* “Landing-Page” – se va vorbi despre pattern-uri ca MVC, unele standarde din subcapitolele precedente și de ce Rails pierde în aplicații de tip Landing-Page;
* “Lucrul cu baze de date” – se va discuta despre un design pattern al datelor ca ORM și cum RoR-ul îl implimentează.

În al treilea 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 42 pagini, conține 20 resurse bibliografice, 1 tabel și 17 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)*

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. Standartele DRY și CoC

*DRY* - acesta este un principiu de dezvoltare software care vizează reducerea recurenței diferitelor tipuri de informații, în special în sistemele cu mai multe straturi de abstractizare. Principiul DRY este formulat ca: "Fiecare bucată de cunoștințe trebuie să aibă o reprezentare unică, coerentă și cu autoritate într-un sistem". Acesta a fost formulat de Andy Hunt (ing.) Și Dave Thomas (Eng.) în cartea lor “Programatorul Pragmatic” (ing.). Ei au aplicat acest principiu la "diagrame de baze de date, planurile de testare, ansambluri, software, chiar documentație [4].

În cazul în care principiul DRY este aplicat cu succes, schimbarea unui singur element al sistemului nu necesită modificări în celelalte elemente, corelate logic. Acele elemente care sunt corelate logic, variază în mod predictibil și consistent. În plus față de utilizarea metodelor și a funcțiilor în cod, iar Thomas Hunt consideră că este necesar să se utilizeze generatoare de cod, sisteme automate de compilare [5].

Principiu DRY, de asemenea, cunoscut ca Single Source of Truth (Eng.), prevaleaza într-un sistem cu arhitectură Model, în care artefactele de program sunt extrase din modelul principal al obiectului și pot fi exprimate într-o astfel de formă ca UML. Codul scris pe principiul DRY, este creat prin conversia datelor și generatoare de cod care permit unui dezvoltator de software pentru a evita tăierea, copierea și lipirea. De obicei, codul scris pe acest principiu face mai ușoară gestionarea sistemelor informatice de dimensiuni mari. Instrumente, cum ar fi XDoclet (Eng.), și XSLT sunt exemple de tehnici de programare DRY. Exemple de sisteme care necesită dublarea informației sunt Enterprise Java Beans versiunea 2, care necesită nu numai duplicarea de cod Java, dar și fișierelor de configurare.

Exemple de sisteme care încearcă să elimine duplicarea informațiilor sunt cadre Symfony, web2py, Yii, Django, Ruby on Rails, Phalcon, precum și mediul de dezvoltare Visual Studio LightSwitch și Enterprise Java Beans versiunea 3.

*CoC* - "presupune că un programator are nevoie doar să specifice aspectele neconvenționale ale aplicației. De exemplu, în cazul în care există o clasă de vânzare în model, tabelul corespunzător din baza de date se numește vânzări în mod implicit. Dacă cineva se abate de la această convenție, cum ar fi apelarea tabelul "produse vândute", programatorul are nevoie să menționeze în cod ceea ce privește de aceste nume. În general, acest lucru duce la mai puțin cod și mai puține repetări [6].

* 1. Biblioteci și Framework-uri

Programatorii elaborînd o aplicație, se confrontă des cu situația, unde codul care a fost scris mai devreme de ei sau de alți programatori convine perfect pentru rezolvarea operației de acum. Rezolvarea problemei date în WeT este de a copia codul precedent. În realitate, în unele situații, metoda dată ar fi cea mai economică, dar în general, metoda dată poate aduce numai probleme. De exemplu: există dublarea codului în programă și este necesară optimizarea codului dat pentru lucrul cu memoria. În cazul cînd va fi nevoie ca optimizarile date să fie aplicate în toate secvențele de cod copiate, va fi necesar să se scrie optimizarea data pentru fiecare separat [7].

În așa caz, e mai corect și comod – să fie creată din secvența dată de cod un modul aparte și apoi modulul dat să fie folosit în diferite locuri necesare. În așa mod, dacă va fi nevoie de a schimba codul sau optimiza, va fi efectuată operația data într-un singur loc. O astfel de metodă de rezolvare în Ruby on Rails se numește DRY.

Dacă s-a creat un modul dintr-o secvență de cod (practic, este de obicei unul sau mai multe fișiere cu codul sursă sau cu codul bibliotecii compilat) și s-a făcut ca modulul dat să fie destul de autonom (rezultă că poate lucra în orice altă programa cu cantități minimale de dependențe), atunci se poate menționa că s-a creat o bibliotecă. În Ruby on Rails bibliotecile se numesc gemuri (rubygems) și ele se instalează chiar din linia de comandă gem install. Autorii gemurilor în multe cazuri lasă codul sursă pe GitHub și de aceea nu va fi dificil de înțeles cum de utilizat bibliotecile.

Dar nu toate gemurile sunt biblioteci. Unele gemuri conțin în ele mai multe biblioteci, scrise de unul sau de mai mulți programatori (fig. 2.1). Și desigur sunt gemuri mari, care efectuează o mulțime de operații și conțin în ele multe biblioteci (fig. 2.2). Un astfel de tip de gem este Ruby on Rails. Gemul dat folosește alte biblioteci, care pot să comunice cu baze de date, să prelucreze http-cereri, să păstreze cookies și să verifice datele introduse de un user.

Structurile mari, care sunt create din mai multe biblioteci legate împreună pentru rezolvarea unei operații (în cazul nostru – aplicație web) se numește Framework.

Framework-ul în orice limbaj de programare oferă o structură oarecare, folosind-o se pot evita multe probleme legate de dezvoltarea și trecerea imediată la rezolvarea obiectivului propus [7].

Framework-ul va fi asociat cu un schelet, care creează bazele și o imagine vizuală. Orice medic este sigur de structura similară a scheletului uman, ce îi ușurează lucrul. Dacă structura scheletului la fiecare om ar fi diferită, medicina nu ar fi ajuns la așa succese mari. Așa și în programare, dacă se va elabora un proiect cu ajutorul unui framework, care este popular și știut de mai mulți programatori, atunci conlucrarea lor va fi mai simplă și eficientă.

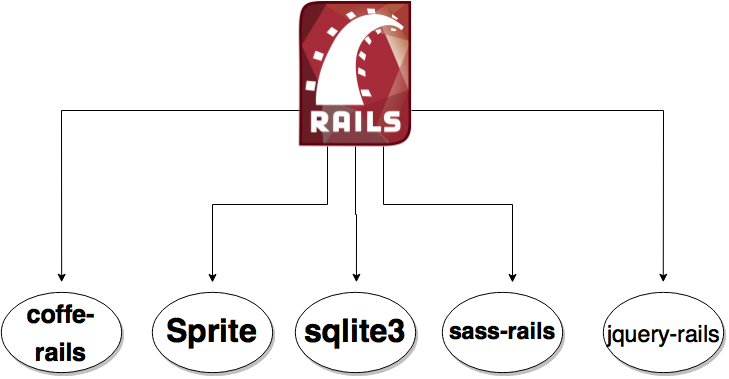


Fig. 2.1. Gem-uri standarte in RoR forma grafica.

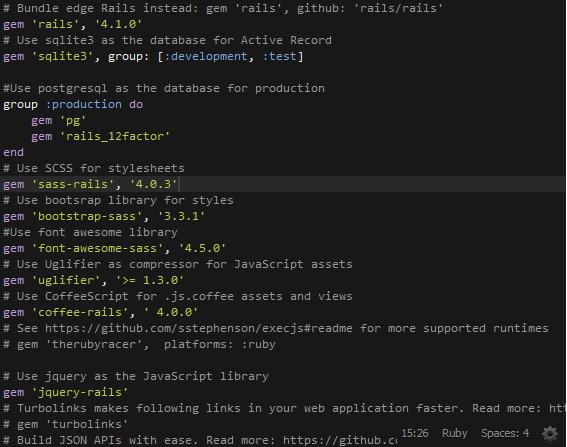


Fig. 2.2. Gem-uri standarte in RoR forma fișier.

* 1. Landing-Page

*MVC* – este un șablon de programare. El se folosește din anii 80-x ca mod de structurare la crearea programelor cu interfață grafică. MVC este prescurtarea a trei cuvinte cheie care explică esența șablonului, M- Model, V- View, C- Controller.

*Model* – sunt niște obiecte care conțin în ele toată informația despre un domeniu al aplicației. De exemplu într-un site cu articole vor fi prezente așa modele ca Item, User, Articol, Comment. În aplicația- Rails fiecare model prezintă în sine un class aparte, cu exemplare (obiecte) care vor fi aplicate în lucru mai departe.

*View* – este un template pentru modelul nostru, care ajută să arate utilizatorului date în forma plăcută și ușor de citit. Vederile în Rails- sunt de obicei cu extensia erb sau haml template-uri. Cu alte cuvinte, codul html cu inserări din limbajul ruby. Un exemplu cum poate arăta o vedere (fig. 2.3)

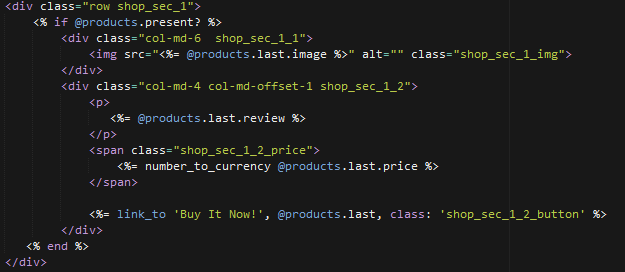


Fig. 2.3. Codul HTML cu inserări a ruby codului.

*Controller* - este o legătură între modelul și vedere. El răspunde la acțiunile utilizatorului, acesta decide ce să facă cu informațiile primite de la utilizator și, de asemenea, ce trebuie să-i arate utilizatorului ca răspuns. Așa cum este vorba despre Rails, aici controller-ul estre strîns legat de ceea ce vede utilizatorul în linia de adresare a browser. De exemplu, este prezentă cererea pe adresa /comments/new în aplicația dată se va ajunge în CommentsController în metoda #new, care, la rândul său, va fi scris în cod conceput pentru a aduce în browser utilizatorul la șablonul formular, pentru a crea un nou comentariu.

Se știe că *Landing-Page*sau aplicație web cu o singură pagină sunt mult bazate pe partea client. În așa tip de aplicații foarte multă logică a programei se conține în partea client și de obicei, e scrisă în limbajul Javascript (sau în limbajul care nu se compilează, d exemplu Coffescript sau Dart) [8].

O aplicație web de tip Landing-Page conține desigur și partea server, dar sunt cazuri datorită cărora Rails nu ar fi o idee bună pentru dezvoltarea aplicațiilor date.

În primul rînd, framework-ul Ruby On Rails este adaptat pentru randarea a multor http-păgini mici ca răspuns la http-cereri din browser. Aplicațiilor web cu o pagină, de obicei, nu le este comod un așa mod de lucru. Ea singură știe cum trebuie să prezinte datele unui utilizator, ca urmare ea este interesată de datele propriu-zise. Simpla transmitere a datelor (de exemplu în formatul json) din Rails este posibilă, dar aceasta este mai degrabă o excepție, ci nu o regulă pentru aplicații Rails.

În al doilea rînd, în Ruby On Rails se folosește concepția REST. Ea presupune că datele transmise în momentul cererii se referă la aceeași resursă (de exemplu date despre un produs).Aplicația web cu o pagină ca regulă trebuie să poate sincroniza dintr-o cerere o cantitate mare de date, care nu sunt legate între ele obligatoriu. De exemplu, un utilizator se află în avion fără acces la internet, folosește o aplicație web cu o pagină (de exemplu redactor text) și introduce schimbări în document, în același rînd schimbînd setările account-ului său. Aici sunt două resurse (document și account), care trebuie să fie sincronizate la accesarea rețelei. În Rails este posibilă manipularea dată, dar nu este comodă [8].

Aplicația web cu o pagină este supusă problemei dublării informației. În cazul necesității validării datelor introduse de un utilizator într-o formă, aplicaţia web cu o singură pagină, care ar trebui să funcţioneze în mod normal, fără o conexiune la reţea, va trebui să realizeze funcționalitatea validării formei și de parte server şi cea client. Şi acest lucru funcţional - nu este copy-paste, pentru că se folosește un limbaj de partea server Rails, iar în cazul clientului - Javascript.

Rails, cu toate acestea, este folosit cu succes astăzi, în colaborare cu clienții grași.  
În mod specific, există destul de multe materiale cu privire la modul de utilizare a Rails împreună cuframework-ul Backbone.js care lucrează cu partea client. Acest lucru este privit destul de sceptic, iar opțiunea ideală pentru o singură aplicație ar fi să se utilizeze același framework scris în Javascript pe partea client și partea de server. Astfel, de exemplu, partea client și server folosesc același model – manipulările date ar reduce semnificativ numărul de funcționalitate duplicată și ar simplifica foarte mult dezvoltarea. Așa tip de framework este NodeJS.

* 1. Lucrul cu baze de date

ORM este o tehnică de programare ce face posibilă accesarea și manipularea obiectelor fără ca programatorii să fie interesați de sursa de date de unde provin aceste obiecte. Această tehnică a apărut din nevoia de a depăși diferențele de paradigmă dintre modelul orientat pe obiecte (susținut de limbajele de programare de nivel înalt actuale) și modelul relațional (utilizat de cele mai populare sisteme de gestiune a bazelor de date).

Limbajele de programare orientate pe obiecte reprezintă datele într-un graf interconectat de obiecte, pe când bazele de date relaționale folosesc un mod tabelar de reprezentare. Efortul de a conecta atributele claselor definite prin intermediul unui limbaj orientat pe obiecte cu câmpurile tabelelor din baza de date nu poate fi ignorat, iar scopul unui ORM este acela de a crea o relație naturală, transparentă, fiabilă și de durată între cele două modele (fig. 2.4). Această nepotrivire de paradigmă pare să nu își fi găsit încă o soluționare definitivă care să fie aprobată de toți actorii din industria IT, însă opinia generală este aceea că framework-urile ORM reprezintă un important pas înainte (tab. 2.1)[9].

Procesul automat de stocare a obiectelor într-o bază de date relațională folosind un framework ORM, constă în maparea obiectelor la tabelele corespunzătoare, asocierea dintre ele fiind descrisă folosind metadata. Un framework ORM complet include următoarele funcționalități:

* un API pentru operațiile CRUD aferente claselor persistente;
* un limbaj pentru specificarea interogărilor, adresând clasele persistente și atributele acestora;
* un mod care să faciliteze definirea metadată pentru mapările dintre obiect și tabelă;
* abordarea consistentă a tranzacțiilor, a metodelor de stocare a datelor ("caching") și a asocierilor dintre clase;
* tehnici de optimizare în funcție de natura aplicației [10].

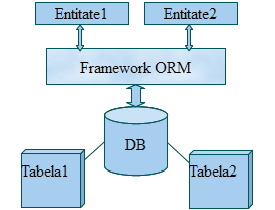


Fig. 2.4. ORM [9].

Tabelul 2.1. Avantajele și dezavantajele ORM

|  |  |
| --- | --- |
| Avantaje | Dezavantaje |
| Modelarea domeniului real | Timp investit în învățarea unui framework ORM |
| Reducerea cantității de cod | Dificultăți în definirea mapărilor |
| Mentenanță simplificată | Performanța interogărilor generate |
| Abordări eficiente a problemelor de concurență | Pe termen scurt nu aduce un salt important din punct de vedere al productivității |
| Abordări eficiente pentru stocarea datelor în memorie |  |
| Productivitatea pe termen mediu și lung |  |

AR – este un model de proiectare a aplicțiilor, reprezentat de Martin Fowler în cartea “Patterns of Enterprise Aplication Architecture”. AR este o modalitate populară de acces la datele din baze de date relaționle în programarea orientată pe obiect.

De exemplu există tabele într-o bază de date. Pentru o tabelă se creează o clasă specială AR, clasa dată trebuie să fie reflecția tabelului în așa mod ca:

* fiecare exemplar al clasei date corespunde unei înregistrări din tabel;
* cînd se adăuga un examplar nou în tabel, se adăuga o nouă înregistrare;
* cînd se schimbă sau se șterge un obiect, se schimbă sau se șterge date care îi corespund.

*Realizarea a ActiveRecords*

ActiveRecords este folosit în multe medii de dezvoltare pentru diferite tipuri de programare. De exemplu în cazul cînd avem tabelul “Plans” cu cîmpurile name (string type) și price (number type), modelul Active Record este realizat în clasa Plan și putem observa urmatorul pseudocod:

*plan = new Plan()*

*plan.name = "Basic"*

*plan.price = 0*

*plan.save()*

Această parte de cod o să adauge o înregistrare în tabelul plans cu datele definite mai sus și apoi o să se aplice comanda SQL:

*INSERT INTO plans (name, price) VALUES ('Basic', 0);*

Din altă parte, clasa poate fi folosită pentru apelarea la baza de date:

*b = Plan.find\_first("name", "Basic")*

*SELECT \* FROM plans WHERE name = 'basic' LIMIT 1; -- MySQL or PostgreSQL*

Sunt niște metode de lucru în modulul Active Record [10]:

* find
* create\_with
* distinct
* eager\_load
* extending
* from
* group
* having
* includes
* joins
* lef\_outer\_j
* limit
* lock
* none
* offset
* order
* preload
* readonly
* references
* reorder
* reverse\_ord
* select
* distinct
* where

Toate metodele date întorc exemplare ActiveRecord::Relation. În general, operațiile de bază Model.find(options) sunt:

* De transformat opțiunile oferite în sql cerere echivalentă;
* De executat cererea SQL și de primit rezultatele corespunzătoare din baza de date;
* De creat un exemplar echivalent obiectului din Ruby pentru fiecare linie din rezultatul cererei din modelul corespunzător.

Posibilitățile modelului dat sunt cu mult mai vaste și în mare parte pun accent pe baze de date relaționale, crearea relațiilor, ștergerea relațiilor sau modificarea lor. În RoR ele se numesc asocieri.

*RailsAssociations*

În general, o asociație este o conexiune între două modele Active Record. Asocierile dintre modele fac operațiunile și interacțiunile comune mai simple. De exemplu, se ia în considerație o aplicație simplă în Rails care include un model pentru autori și un model pentru cărți. Fiecare autor poate avea multe cărți. Fără ajutorul asociațiilor, declarațiile de model ar arăta astfel [10]:

*class Author < ApplicationRecord*

*end*

*class Book < ApplicationRecord*

*end*

Acum de exemplu este necesară adăugarea unei cărți noi pentru un autor existent.

*@book = Book.create(published\_at: Time.now, author\_id: @author.id)*

Sau de exemplu este necesară ștergerea unui autor:

*@books = Book.where(author\_id: @author.id)*

*@books.each do |book|*

*book.destroy*

*end*

*@author.destroy*

Cu Active Record associations, este posibilă eficientizarea acelora – sau altor – operații care să-i spună lui Rails că modelele sunt legate între ele. Mai jos se va examina o metodă cum de creat asocierea dată:

*class Author < ApplicationRecord*

*has\_many :books, dependent: :destroy*

*end*

*class Book < ApplicationRecord*

*belongs\_to :author*

*end*

În codul de mai sus se observă legături dintre modelele Active Record:

* “has\_many”
* “belongs\_to”

Cuvintele legate între ele cu “\_” sunt asocierile. Rails-ul suportă șase tipuri de asocieri:

* *belongs\_to*

Relația belongs\_to (fig. 2.5) arată apartinența cărții la un autor. Este o asociere de tip unul la unu, dar diferența este în denumire și înțelegere.

*class Book < ApplicationRecord  # ActiveRecord::Base до Rails 5.0*

*belongs\_to :author*

*end*

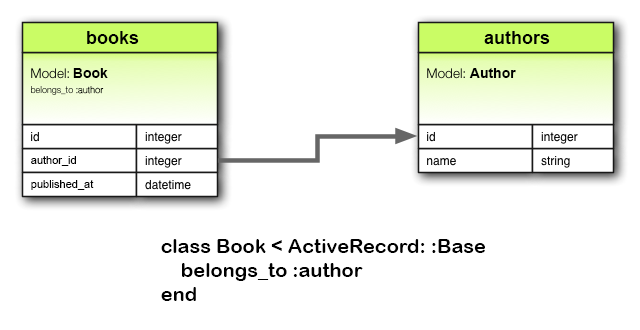


Fig. 2.5. Asocierea belongs\_to [10].

* *has\_one*

Relația has\_one (fig. 2.6) este tot relație de tip unul la unu dar cu alt sens (alte rezultate). Relația data ne arată că fiecare exepmplar al modelului conține sau îi aparține unui exemplar din alt model.

*class Supplier < ApplicationRecord  # ActiveRecord::Base до Rails 5.0*

*has\_one :account*

*end*

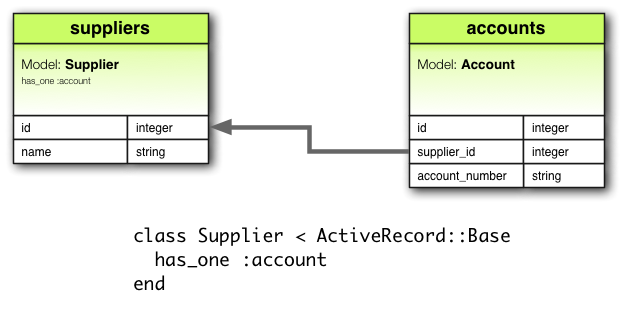


Fig. 2.6. Asocierea has\_one [10].

* *has\_many*

Relația has\_many (fig. 2.7) este o relație de tip unul la mulți. Relația data ne arată că fiecare exemplar al modelului conține sau îi aparține zero sau mai multe exemplare din alt model.

*class Author < ApplicationRecord  # ActiveRecord::Base до Rails 5.0*

*has\_many :books*

*end*

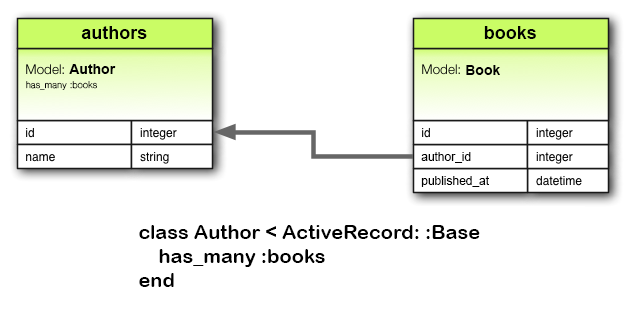


Fig. 2.7. Asocierea has\_many [10].

1. ELABORAREA APLICAȚIEI ”MARIUS REVIEW SITE”
   1. Spațiul de lucru Visual Studio Code

Pentru a elabora o aplicație web este necesar de diferite instrumente, cum ar fi:

* Serverul, mașină locală sau IDE;
* Un redactor textual;
* 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 cel mai popular editor 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 RubyMine și se creează un proiect nou de tip Rails cu configurări pentru baza de date PostgreSQL și se alege Ruby SDK (fig.3.1).

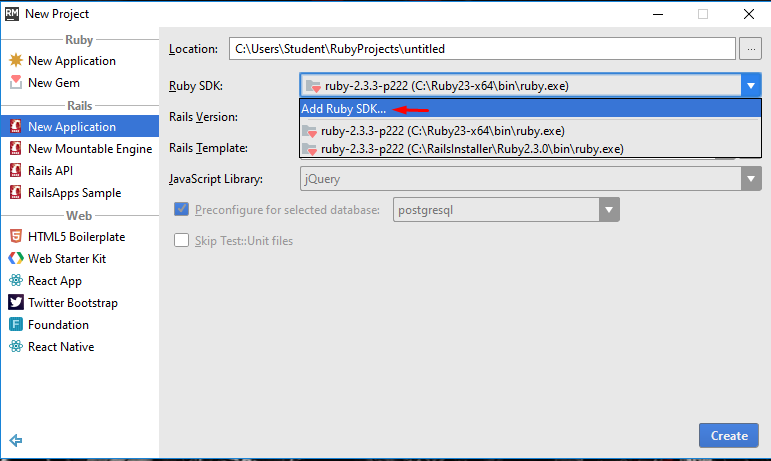


Fig. 3.1. Crearea unui proiect nou cu toate setările.

După apăsarea butonului Create în cîteva clipe se vor crea toate dosarele și fișierele necesare.

Pentru a conecta baza de date se accesează fișierul care se află pe calea dată config/database.yml, în el vor fi setate toate configurările necesare (fig.3.2). După setarea configurărilor, spațiul de lucru este total apt pentru a începe dezvoltarea aplicației [20].

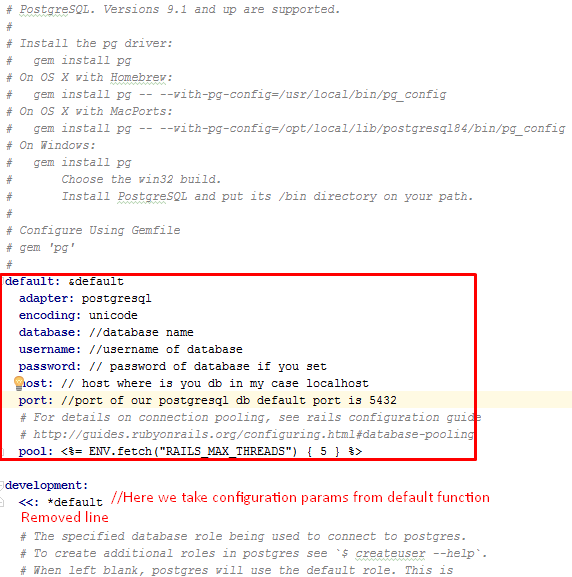


Fig. 3.2. Configurările necesare pentru conectare la BD.

* 1. Crearea aplicției Marius Review Site
     1. Ierarhia aplicației

Datorită tehnologiilor menționate mai sus, se observă că structura este specifică multor framework-uri pe care le utilizează alte limbaje de programare. ORM și MVC sunt criterii de bază pentru RoR. O pagină simplă din proiect, care nu conține logică, ci doar pur și simplu un template html și css nu are nevoie de tehnica de programare ORM sau toată structura MVC. Se accentuează faptul necesității structurelor date la pagini care folosesc tabelele din baza de date sau/și logică de accesare, etc. În mare parte fiecare pagină a aplicației este un obiect al bazei de date. De exemplu există o pagină cu afișarea articolelor, reiese că pagina singură se va numi articole, dar fiecare articol în parte va fi o înregistrare (cîmp) din tabelul *articles*. Fiecare cîmp este cîte un obiect care conține un număr de ordine și informație referitor la articolul salvat.

După pașii de configurare a spațiului de lucru va fi în file manager structura generată de Rails. Ea conține în ea toate dosarele necesare pentru lucru. În ele deja se vor efectua următoarele operații [15]:

Datorită interfeții IDE RubyMine se poate accesa, crea și șterge fișiere, dosare fără a utiliza linia de comandă (fig. 3.3.).

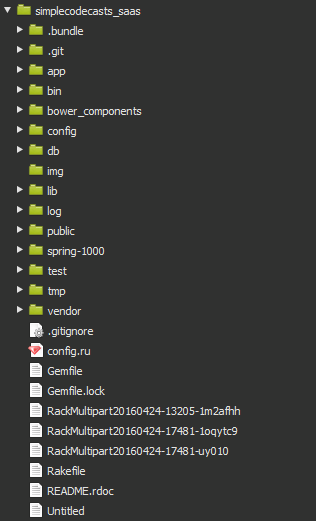


Fig. 3.3. Ierarhia dosarelor în aplicația Rails.

Ierarhia dată este puțin modificată de la acea, care este la începutul proiectului nou. Sunt adăugate dosare și fișiere noi care au scopurile lor în aplicație. De exemplu dosarul .git el a fost adăugat odată cu instalarea conexiunii cu repozitoriile git-ului și conține în el configurări și restricții.

În general cea mai mare parte a aplicației se află în dosarul *app.* Poate fi găsit în dosarul dat, dosarul *assets,* care la rîndul său conține trei dosare *javascript, stylesheets, images.* Desigur nu uităm de standartul MVC. În dosarul *app* se află dosarele *models, views, controllers* (fig. 3.4.).

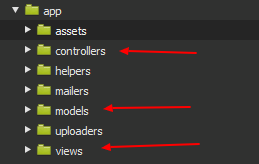


Fig. 3.4. Ierarhia dosarului app în aplicația Rails.

Fiecare dosar și fișier dintr-o aplicație RoR are o însemnatate enormă la nivel de utilitate cît și configurare (fig. 3.5).



Fig. 3.5. Esența fiecărui dosar și fișier din ierarhie.

* + 1. Baza de date (PostgreSQL, Active Record Migration)

Începînd crearea unei aplicații în rails în primul rînd este nevoie modelarea bazei de date, deoarece ea este un punct foarte important. Desigur nu toate paginile vor avea nevoie în parte de ierarhia lor proprie MVC și de bază de date. Baza de date a aplicației este de tip relațional și conține opt tabele (fig. 3.6) :

TezaDeAnDB

Fig. 3.6. Baza de date relațională.

Folosirea bazelor de date relaționale oferă posibilități înalte și desigur micșorează tabelele inițiale. Baza de date care va fi utilizată este PostgreSQL [11]. PG este un gem care reprezintă în el toate tehnologiile și setările necesare pentru a efectua lucrul cu baza de date. Toate gem-urile se află într-un fișier special dedicat, Gemfile Fig.1.2. Se înscrie în fișier numele gem-ului [14]:

*gem 'pg', group: [:development, :test]*

După salvarea documentului se scrie în console *bundle install.* Comanda va instala gem-ul dat sau dacă au fost adăugate mai multe gemuri. Gem-urile se află în mare parte pe GitHub și se instalează direct de acolo.

Acum poate fi începută crearea tabelelor în baza de date. Sunt două posibilități de a crea un tabel :

* Generarea unei migrări de date care se introduce manual:

*rails generate migration CreatePlans*

Va fi generat un fișier de migrare care va conține

*class CreatePlans < ActiveRecord::Migration*

*def change*

*//introducem codul singuri*

*end*

*end*

*end*

* Generarea unei migrări de date care va seta coloanele automat:

*rails generate migration CreatePlans name:string,price:float*

Va fi generat un fișier de migrare care va conține

*class CreatePlans < ActiveRecord::Migration*

*def change*

*create\_table :plans do |t|*

*t.string :name*

*t.decimal :price*

*t.timestamps*

*end*

*end*

*end*

Pentru fiecare tabel din baza de date se pot utiliza aceste două modalități. Reieșind din codul de mai sus, se observă că acesta este un pseudo limbaj. Active Record lucrează la un nivel mai înalt, ceea ce presupune utilizarea oricărei baze de date posibile. El singur va tranforma pseudo codul în cod SQL.

* + 1. Modele în Rails (Model, Active Record Relation)

Pentru a lămuri toată structura unei aplicații Rails se va folosi entitatea *Profile*. Se observă mai sus că *Profile* este legat cu tabelul *Users*. Tabelul *Users* este creat de un gem foarte popular și util Device. Acest gem este folosit pentru politica utilizatorilor pe site. La rîndul său Device creează sesiunile, posibilitățile de a restabili parola, de a o modifica și multe altele.

Se începe cu configurarea modelelor, apoi deja vor fi configurate vederile.

În dosarul models se creează fișierul *profile.rb* el va conține urmatorul cod:

*class Profile < ActiveRecord::Base*

*mount\_uploader :image, ImageUploader*

*belongs\_to :user*

*end*

Deja aici se poate observa exemplele de mai sus cu relațiile între tabele. *Belongs\_to :user* zice că un profil aparține unui utilizator. În fișierul *user.rb* se adaugă o linie de cod :

*class User < ActiveRecord::Base*

*devise :database\_authenticatable, :registerable,*

*:recoverable, :rememberable, :trackable, :validatable*

*belongs\_to :plan*

*has\_one :profile*

*has\_many :comments*

*attr\_accessor :stripe\_card\_token*

*end*

El spune că un utilizator are numai un profil ce presupune legătura unul la unu [12].

* + 1. Rute în Rails(Routes)

Legătura este stabilită și pasul întîi este finisat. Acum este nevoie de crearea unei rute pentru pagina Profile. Acțiunea data presupune (fig. 3.7):

RailsMVC(3)

Fig. 3.7. Exemplu de rutări pentru pagină.

Odată cu crearea unui tabel nou, mai este nevoie de a arăta aplicației regulile routes. Se accesează fișierul *config/routes.rb.* Așa cum profilul este legat de utilizator, reiese în regulile routes trebuie să se indice faptul dat :

*resources :users do*

*resource :profile*

*end*

Pentru a controla dacă regulile s-au instalat, trebuie scrisă comanda *rake routes.*

*user\_profile POST /users/:user\_id/profile(.:format) profiles#create*

*new\_user\_profile GET /users/:user\_id/profile/new(.:format) profiles#new*

*edit\_user\_profile GET /users/:user\_id/profile/edit(.:format) profiles#edit*

*GET /users/:user\_id/profile(.:format) profiles#show*

*PATCH /users/:user\_id/profile(.:format) profiles#update*

*PUT /users/:user\_id/profile(.:format) profiles#update*

*DELETE /users/:user\_id/profile(.:format) profiles#destroy*

Fiecare linie presupune trei lucruri. Link-ul /users/user\_id/profile(format) arată că regulile routes sunt setate corect, link-ul dat este și URN format în browser și ce funcții vor răspunde pentru paginile date[16].

* + 1. Controlere în Rails(Controller, CRUD)

Sistema MVC din Rails este foarte strictă și fiecare greșeală mică va erona enorm. Cu funcțiile de bază CRUD lucrează controller-ul. La rîndul său vederea lucrează cu partea vizuală. Lucru cu aceste entități deseori este paralelă, deoarece fiecare vedere are funcția sa în controller sau fiecare funcție din controller are pagina sa în vedere. Se creează în dosarul *views* dosarul *users* și în el se creează fișierul *show.html.erb*. El va răspunde pentru interfața profilului fiecărui utilizator. De asemenea, se creează dosarul *profiles* și în el fișierul *new.html.erb*. Pînă cînd fișierele se lasă necompletate, apoi va avea loc întoarcerea la ele [17].

Se elaborează în dosarul *controllers* fișierul *profiles\_controller.rb* și se completează cu următorul cod:

*class ProfilesController < ApplicationController*

*def new*

*# Form where user can fill out their own profile.*

*@user = User.find( params[:user\_id] )*

*@profile = Profile.new*

*end*

*def create*

*@user = User.find( params[:user\_id] )*

*@profile = @user.build\_profile(profile\_params)*

*if @profile.save*

*flash[:success] = "Profile Updated"*

*redirect\_to user\_path ( params[:user\_id])*

*else*

*render action: :new*

*end*

*end*

*private*

*def profile\_params*

*params.require(:profile).permit(:first\_name, :last\_name, :job\_title, :phone\_number, :contact\_email, :description, :image)*

*end*

*end*

Se observă trei funcții cu care se va lucra:

* *def new* răspunde pentru fișierul new.html.erb și conține în sine căutarea utilizatorului cu parametrul *user\_id*. Lucrul dat se face pentru ca vederea să înțeleagă pentru ce utilizator se va introduce informația. Profile.new spune despre crearea obiectului. Parametrii dați se păstrează în variabile petru a lucra apoi cu ele.
* *def create* răspunde pentru metoda POST din fișierul new.html.erb. Cînd se va realiza întoarcerea la vederea dată se va arăta pentru ce.
* *private def profile\_params* este o funcție secundară care răspunde pentru permiterea metodei POST de a transmite pe server parametrii care sunt indicați mai sus.

Se creează un controller pentru user*. users\_controller.rb*

*class UsersController < ApplicationController*

*def show*

*@user = User.find(params[:id])*

*end*

*end*

Funcția date răspunde pentru fișierul show.html.erb și conține în sine căutarea utilizatorului cu parametrul *user\_id*. Se face lucrul dat pentru că vederea să înțeleagă pentru ce utilizator se va afișa informația.

* + 1. Vederi în Rails(view, html.erb)

Acum e timpul de revenire la fișierul *new.html.erb ,* înserînd în el următorul context:

*<div class="row">*

*<div class="profile\_new\_form\_decript">*

*<h1 class="text-center">Create your profile</h1>*

*<p class="text-center">Be a part of Maik Review community, fill your profile!</p>*

*</div>*

*<div class="well profile\_new\_form">*

*<%= form\_for @profile, url: user\_profile\_path do |f| %>*

*<div class="form-group">*

*<%= f.label :first\_name %>*

*<%= f.text\_field :first\_name, class: 'form-control' %>*

*</div>*

*<div class="form-group">*

*<%= f.label :last\_name %>*

*<%= f.text\_field :last\_name, class: 'form-control' %>*

*</div>*

*<div class="form-group">*

*<%= f.label :job\_title %>*

*<%= f.select :job\_title,['Developer','Enterpreneur','Investor'], {},class: 'form-control' %>*

*</div>*

*<div class="form-group">*

*<%= f.label :phone\_number %>*

*<%= f.text\_field :phone\_number, class: 'form-control' %>*

*</div>*

*<div class="form-group">*

*<%= f.label :contact\_email %>*

*<%= f.text\_field :contact\_email, class: 'form-control' %>*

*</div>*

*<div class="form-group">*

*<%= f.label :image %>*

*<%= f.file\_field :image %>*

*</div>*

*<div class="form-group">*

*<%= f.label :description %>*

*<%= f.text\_area :description, class: 'form-control' %>*

*</div>*

*<div class="form-group">*

*<%= f.submit "Create", class: 'submit\_product\_form' %>*

*</div>*

*<% end %>*

*</div>*

*</div>*

Pagina dată prezintă o formă de împlinire cu ajutorul căreia se pot introduce datele personale în profil. Se poate observa sintaxa creării formularului în RoR cu metoda *user\_profile\_path* care va răspunde pentru metoda POST [18].

Ultimul pas va fi adăugarea codului în fișierul paginei, unde se va vizualiza profilul dat s*how.html.erb*

*<div class="row profile\_section">*

*<figure class="fig\_profile\_img col-md-4">*

*<img src="<%= @user.profile.image %>" class="profile\_img"></img><br/>*

*</figure>*

*<div class="col-md-8 profile\_info">*

*<span><%= @user.profile.first\_name %> <%= @user.profile.last\_name %> </span><br/>*

*<span><%= @user.profile.job\_title %></span><br/>*

*<% if current\_user.plan\_id == 2 or current\_user.try(:admin)%>*

*<span><%= @user.profile.phone\_number %></span><br/>*

*<span><%= @user.profile.contact\_email %></span><br/>*

*<% end %>*

*<p><%= @user.profile.description %></p>*

*</div>*

*</div>*

* + 1. Structura ierarhică finală a aplicației

Acest exemplu este foarte bun pentru a înțelege logica utilizări framework-ului RoR. Fiecare pagină cu acces la baza de date are o structură asemănătoare în mare parte. Sunt cazuri cînd pașii de creare sunt mai grei, iar uneori mai ușori. Articolele se află în tabelul cu numele Products.

Va fi prezentată structura dosarelor MVC pentru cazul dat, iar parcurgînd pașii de sus, realizarea acestui punct important nu va fi o problemă (fig. 3.8):

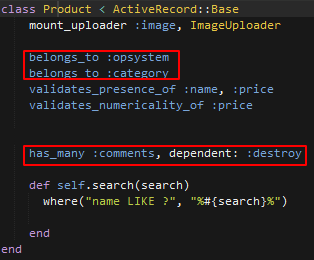
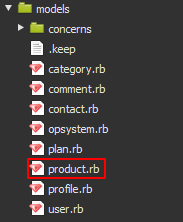


Fig. 3.8. Modelul paginii Articole.

Modelul product prezintă în site un modul mai complex. Sunt prezente relații cu mai multe tabele, dar care folosesc aceleași structuri, cum ar fi *belongs\_to* sau *has\_many.* Realizarea unui motor simplu de căutare este datorită funcției *search,* care efectuază căutarea după numele dispozitivului.

Ierarhia dosarului c*ontrollers* arată cîte module sunt implicate în aplicație. Controller-ul paginii *Articole* se numește *products\_controller.rb* și conține toate funcțiile necesare pentru afișare, ștergerea și adăugarea unui articol nou (fig.3.9).

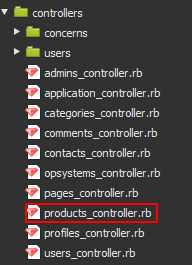


Fig. 3.9. Controller-ul paginii Articole.

Fiecare funcție din fișierul *products\_controller.rb* deține o vedere din dosarul views (fig. 3.10). Analizînd fișiriele din ierarhia dosarului *products* din *views,* se reiau cinci funcții de bază din denumirile fișierelor:

1. \_form.html.erb reprezintă funcția *create,* scopul fiind plasarea infrormației pe server (POST);
2. edit.html.erb reprezintă funcția *edit,* scopul fiind editarea unui articol deja existent;
3. index.html.erb sau punctul de intrare, reprezintă funcția *show,* scopul fiind afișarea tuturor articolelor;
4. new.html.erb reprezintă funcția *new,* scopul fiind culegerea datelor din formă (GET);
5. show.html.erb reprezintă funcția *show,* scopul fiind afișarea articolului pentru fiecare product în parte.

Fișierul sub numele *\_product.html.erb* este o parte a fișierului *index.html.erb.* Design-ul aplicat la pagina dată mărește relativ numărul de linii de cod și pentru a simplifica codul este posibilitatea de a împărți fișierul în mai multe părți mici. Chemarea în fișierul de bază se face în modul următor:

<% if @products.present? %>

<%= render "product" %>

<% else %>

<p>There are no posts containing the term(s) <%= params[:search] %>.</p>

<% end %>

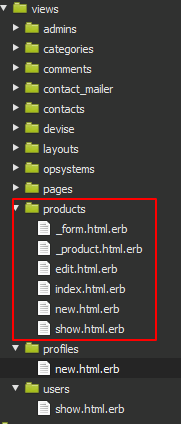


Fig. 3.10. Vederile paginii Articole.

CONCLUZII

Lucrarea dată reprezintă o cercetare a limbajului de programare Ruby și a framework-ului Ruby on Rails. Astfel au fost punctate particularitățile limbajului, referințe la apariția sa istorică și prezentarea sintaxei acestuia. Framework-ul RoR reprezintă continuarea limbajului de programare Ruby, din domeniul desktop în web. Framework-ul dat schimbă structura totală a limbajului Ruby, adăugînd în sine multiple module și tehnologii.

RoR-ul reprezintă un complex masiv datorită tehnologiilor, paradigmelor și tehnicilor de programare încapsulate:

* MVC;
* ORM;
* DRY & CoC.

Limbajul Ruby și framework-ul RoR sunt, în mare parte, rezultatul aplicării paradigmelor și tehnicilor de programare menționate recent, exemplu fiind ActiveRecords. ORM presupune crearea unei relații naturale, transparente, fiabile și de durată între cele două modele. Relațiile dintre tabelul din baza de date cu obiectul din RoR îl efectuează AR și în același timp se ține de ideea conectării atributelor claselor definite cu câmpurile tabelelor din baza de date. Dependența structurii aplicației este în mare parte prestabilită de principiile sau/și standartele de programare. Standartele date servesc la o gîndire structurată și concretă la etapa de programare. Rails MVC structurează strict ierarhia datelor pentru fiecare tip de configurare sau vizualizare în parte.

Partea teoretică conține în sine o cantitate vastă de tehnologii utilizate des în programare. Datorită informației despre fiecare tehnologie în parte și intercomunicarea lor în spațiul de dezvoltare Ruby, a fost elaborată o aplicație web cu următoarele posibilități:

* Înregistrarea utilizatorilor;
* Crearea profiluilui (Nume, Prenume, Poza, Nr. Telefon, etc.);
* CRUD a dispozitivelor noi, care la rîndul său conțin informații standare (pret, nume) și opinia autorului;
* Fiecare utilizator logat are posibilitatea anexării comentariilor la articol și ștergerea celor proprii;
* Sortarea dispozitivelor după categorii;
* Căutarea articolului după nume.

Deși limbajul Ruby a apărut în anul 1994, o dezvoltare mai pronunțată a avut-o în ultimii ani, datorită apariției framework-ului Ruby on Rails, care la moment nu este cercetat suficient, iar în limba română documentație practic nu există. Datorită insuficienței materialului teoretic în limba română, suportul teoretic al acestei lucrări poate fi utilizat de programatorii care doresc să studieze limbajul Ruby.

Yukihiro "Matz" Matsumoto nu o singură dată a menționat că Ruby fiind un limbaj vizual simplu, conține în sine un motor complex de programare, principiul care a fost preluat și de framework-ul Ruby on Rails.

BIBLIOGRAFIE

1. David Thomas, Chad Fowler, Andrew Hunt*. Programing Ruby: The Pragmatic Programmer’s Guid, Second Edition.* Boston : Addison-Wesley, United States of America. 2004. pag. 5-50.
2. Mircea Sîrbu. *Ruby: Un limbaj cu ochii oblici* [on-line].Publicat în NET Report 110 - noiembrie 2001. [citat 30.10.2016]. Disponibil : <http://www.intraweb.ro/txt/Articole/Limbaje/Ruby/show>.
3. Bogdan Gaza. *De ce sa nu folosim Ruby on Rails* *?* [on-line] Publicat în InfoEducație, 2010. [citat 5.11.2016]. Disponibil : <http://www.slideshare.net/hurrycane/de-ce-sa-nu-folosim-ruby-on-rails>.
4. Dharmdip Pathod. *What is CRUD, COC and DRY in RoR* *?* [on-line] [citat 6.11.2016]. Disponibil: <https://dharmdip.wordpress.com/2008/11/15/what-is-crud-coc-and-dry-in-ror/>.
5. Sam Ruby, Dave Thomas. *Agile Web Development with Rails. – Pragmatic Programmers, LLC.*, United States of America. 2011. pag.17.
6. Bachle, M., & Kirchberg. *Ruby on Rails. Software, IEEE*, United States of America. 2007. DOI 10.1109/BCI.2009.31., pag.105-108.
7. Роман Снитко. *Правильное обучение разработке веб-приложений на Ruby On Rails,* Rusia. 2013. pag. 14-17.
8. Роман Снитко. *Правильное обучение разработке веб-приложений на Ruby On Rails, Одностраничные веб-приложения ипочему Rails не подходит для ихсоздания,* Rusia. 2013. pag. 18-19.
9. Anghel Contiu*.* *Analiza mecanismului Object– Relational Mapping (ORM) cu exemplificări Hibernate* [on-line]. [citat 10.11.2016]. Disponibil : <http://www.todaysoftmag.ro/article/73/analiza-mecanismului-object-relational-mapping-orm-cu-exemplificari-hibernate>.
10. ORM*.* *Modelarea bazelor de date folosind ORM. Ce este ORM?* [on-line], [citat 17.11.2016]. Disponibil : <http://www.scrigroup.com/calculatoare/baze-de-date/Modelarea-bazelor-de-date-util41169.php>.
11. The PostgreSQL Global Development Group*.**Документация к PostgreSQL 9.6.2* [on-line], 2017. [citat 7.03.2017]. Disponibil : <http://repo.postgrespro.ru/doc/pgsql/9.6.2/ru/postgres-A4-fop.pdf>.
12. David Heinemeier*.* *Active Record Query Interface* [on-line]. [citat 20.11.2016]. Disponibil : <http://guides.rubyonrails.org/v3.2.8/active_record_querying.html>.
13. Роман Снитко*.* *Правильное обучение разработке веб-приложений на Ruby On Rails, Преимущества и недостатки Rails*. 2013. pag. 19-20.
14. Cloud9 IDE. *Cloud9 IDE* [on-line]. [citat 04.01.2016]. Disponibil : <https://twitter.com/cloud9ide?lang=ru>.
15. Robben Dey. *The Complete Web Developer Bootcamp- Bginer to Expert* [on-line], 2015. [citat 05.01.2016]. Disponibil : <https://www.udemy.com/job-ready-web-developer/learn/v4/content>.
16. David Heinemeier Hansson*.**Rails Routing from the Outside In* [on-line]. 2016. [citat 15.01.2017]. Disponibil : <http://guides.rubyonrails.org/routing.html>.
17. David Heinemeier Hansson.*Action Controller Overview* [on-line]. 2016. [citat 25.02.2017]. Disponibil : <http://guides.rubyonrails.org/action_controller_overview.html>.
18. David Heinemeier Hansson*.**Action View Overview* [on-line]. 2016. [citat 06.03.2017]. Disponibil : <http://guides.rubyonrails.org/action_view_overview.html>.
19. Jet Braincs Inc.*Ruby Development, Motion, Testing, Debugging and Tools.* [on-line]. 2017. [citat 12.03.2017]. Disponibil : <https://www.jetbrains.com/ruby/features/ruby_ide.html>.
20. Țurcanu Marius-Dan. *Spring vs Ruby on Rails*. [on-line]. 2017. [citat 20.03.2017]. Disponibil :<https://drive.google.com/open?id=18fZ3_02wToho6yESxolRdX9i9HFcR7zv3152z1_817o>.