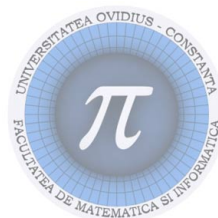


UNIVERSITATEA OVIDIUS



FACULTATEA DE MATEMATICĂ ȘI
INFORMATICĂ

LUCRARE DE LICENȚĂ

Dezvoltare aplicație web SPA bazată pe Python/Django și AngularJS

Student:
Ștefan Daniel MIHĂILĂ

Profesor îndrumător:
Lect. dr. Andrei RUSU

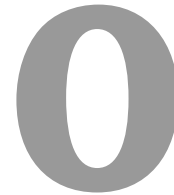
iulie 2015

Cuprins

0	Introducere	2
1	Aplicații SPA: moduri de dezvoltare	4
1.1	Framework-uri MVC JavaScript	4
1.2	AJAX	5
1.3	WebSocket	5
1.4	Plugin-uri pentru browser	6
2	Python și Django	7
2.1	Python	7
2.2	Django	10
3	JavaScript și AngularJS	11
3.1	JavaScript	11
3.2	AngularJS	12
4	Alte tehnologii folosite în aplicație	15
4.1	Bootstrap	15
4.2	Bower	15
4.3	Git și GitHub	16
4.4	API REST (Django REST Framework)	16
5	Descrierea aplicației	19
5.1	Scop și prezentare generală	19
5.2	Instalare aplicație și pornire server de dezvoltare	19
5.3	Folosirea aplicației	20
6	Detalii de implementare	24
6.1	Structura de directoare	24
6.2	Module AngularJS	25
6.3	API REST	26
6.4	Fluxul aplicației	26
6.5	Autentificarea	27
6.6	Operații CRUD pentru Expense	31
6.7	Modulele "layout" și "utils"	36
6.8	Posibilități de îmbunătățire ale aplicației	42
7	Concluzii	44

Internetul este un amalgam de tehnologii, legate împreună cu bandă adezivă, sfoară și gumă de mestecat. Nu este ceva proiectat într-un mod elegant, pentru că este un organism în creștere, nu o mașinărie construită cu intenție.

Mattias Petter Johansson (Programator la Spotify)



Introducere

Internetul a evoluat continuu și a ajuns în punctul în care poate face o mulțime de lucruri pentru care nici măcar nu a fost creat. Aproape toți programatorii din ziua de azi sunt programatori web, iar aplicațiile web seamănă tot mai mult cu aplicațiile desktop. În aceste condiții, a devenit foarte important pentru dezvoltatori să poată crea astfel de aplicații într-un mod rapid și eficient, iar uneltele pe care le au la dispoziție au fost reînnoite permanent cu altele mai bune.

Arhitectura web clasică este una client-server, în care clientul (browserul) cere o pagină folosind protocolul HTTP, serverul o crează dinamic folosind un limbaj de programare server-side (C#, Java, Python, PHP, Scala etc.) și o trimite browserului pentru afișare. Prin HTTP, conexiunile sunt întotdeauna inițiate de către client, care cere pagina web.

Această arhitectură este limitată. Să ne imaginăm de exemplu că avem o pagină web care afișează în timp real scorurile unor partide de fotbal. După încărcarea paginii, server-ul nu-i poate comunica browserului că un scor s-a schimbat. Browserul va afișa scorurile neactualizate până când utilizatorul reîmprospătează pagina.

Această problemă a fost rezolvată prin intermediul AJAX¹, o tehnică ce permite browserului să facă cereri asincrone către server după ce pagina a fost încărcată, prin intermediul JavaScript.

Următoarea etapă în acest proces incremental a fost crearea de *Single-Page Application*², denumite în continuare SPA. Într-un SPA, tot codul HTML, JavaScript și CSS este fie descărcat în momentul în care pagina este încărcată prima dată, fie în mod asincron, de obicei ca răspuns la acțiunile utilizatorului.

SPA oferă utilizatorului senzația unei aplicații fluide și poate uneori să ofere iluzia că aceasta răspunde la acțiuni imediat, fără să mai aștepte răspunsul serverului. Vom vedea în aplicația construită pentru această lucrare, de exemplu,

¹Asynchronous JavaScript and XML; în aplicațiile moderne se utilizează cu preferință JSON (JavaScript Object Notation) în loc de XML, dar denumirea a rămas.

²http://en.wikipedia.org/wiki/Single-page_application

că atunci când utilizatorul dorește ștergerea unei resurse, această resursă este întâi înlăturată din UI, apoi o cerere asincronă îi spune serverului să șteargă resursa din baza de date. Desigur, pentru că se comunică cu serverul prin TCP/IP, această comunicare poate eșua, caz în care un mesaj de eroare este afișat și resursa re apare în UI, dar în mai mult de 90% din cazuri, când totul merge bine, utilizatorul are senzația că resursa este ștearsă instant.

Două companii mari, Google și Facebook, au creat fiecare câte un framework pentru crearea de SPA: AngularJS și React, confirmând importanța acestui tip de aplicații. Experiență fluidă pentru utilizator, împreună cu alte avantaje pe care le vom discuta în capitolul următor au făcut ca SPA să crească foarte mult în popularitate în ultimii ani.

*Cândva oamenii credeau că internetul este o altă lume,
dar acum realizează că este o unealtă pe care o folosim în
lumea noastră.*

Tim Berners-Lee, inventatorul *www*-ului

1

Aplicații SPA: moduri de dezvoltare

În acest capitol vom enumera diferite moduri în care se pot dezvolta SPA și vom discuta avantajele și dezavantajele acestor moduri.

1.1 Framework-uri MVC JavaScript

Anumite framework-uri JavaScript pentru creare de aplicații web, cum ar fi Backbone.js¹, AngularJS², Ember.js³, React⁴ și Meteor⁵ și-au propus să ușureze dezvoltarea de aplicații web SPA.

Aceste framework-uri oferă de obicei și posibilitatea organizării codului folosind șablonul arhitectural *Model-view-controller*⁶ (MVC). MVC a fost folosit inițial în dezvoltarea aplicațiilor desktop, dar s-a dovedit mai târziu util și pentru dezvoltarea părții de back-end (server-side) a aplicațiilor web. Abia de curând el a fost adoptat și pe front-end (client-side).

MVC decuplează datele și logica aplicației de prezentare (interfața cu utilizatorul). Într-un framework JavaScript MVC, view-ul este reprezentat de șabloane HTML, controller-ul este un obiect JS care se ocupă de comunicarea dintre view și model, iar modelul este un obiect JS care, de obicei, mapează obiectele din baza de date de pe server la obiecte afișate de interfața cu utilizatorul. Desigur, o aplicație ce rulează în browser nu are acces în mod direct la baza de date, de aceea această mapare se face apelând un API REST⁷.

În continuare enumerăm câteva avantaje ale folosirii unui framework JS pentru dezvoltarea de aplicații SPA:

¹<http://backbonejs.org/>

²<https://angularjs.org/>

³<http://emberjs.com/>

⁴<http://facebook.github.io/react/>

⁵<https://www.meteor.com/>

⁶<http://en.wikipedia.org/wiki/Model-view-controller>

⁷http://en.wikipedia.org/wiki/Representational_state_transfer

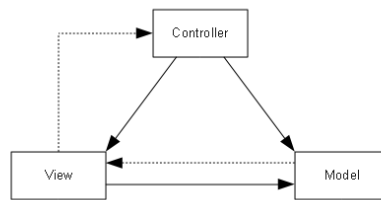


Figura 1.1: Interacțiunea dintre componentele MVC

- Folosirea MVC: un proiect MVC este mai ușor de navigat, este mai ușor de modificat și mai ușor de înțeles. De asemenea, colaborarea dintre designer și programator este ușurată de MVC.
- Viteza de dezvoltare (după depășirea curbei de învățare).
- Ușurința de dezvoltare.
- Diminuarea codului necesar a fi scris.

Bineînțeles, există și dezavantaje:

- Necesitatea învățării unei tehnologii noi. Ce este și mai trist, este că foarte posibil această tehnologie va fi depășită în doar câțiva ani.
- Unele framework-uri (AngularJS) au o curbă de învățare abruptă.
- Frameworkurile au tendința de a nu suporta browserele mai vechi.

1.2 AJAX

AJAX este modalitatea "clasică" prin care pot fi create SPA. Avantaje:

- Fiind o modalitate mai veche, este cunoscută de mai mulți programatori.
- Suport mai bun pentru browserele vechi.

Dezavantaje:

- Este necesar să se scrie mult cod.
- Folosirea MVC este mai dificilă.
- Viteză mică de dezvoltare.

1.3 WebSocket

WebSocket⁸ este un protocol ce permite comunicare bidirecțională cu serverul. Atunci când clientul inițiază comunicarea cu serverul prin HTTP, serverul îi poate cere clientului să treacă la WebSocket. Dacă trecerea are loc, atunci

⁸<http://en.wikipedia.org/wiki/WebSocket>

serverul îi poate trimite notificări clientului, lucru care nu este posibil pe HTTP. Performanța WebSocket este mai bună decât AJAX. În plus, soluția este mai elegantă. Cu AJAX, clientul face *polling*, adică întreabă serverul la un anumit interval de timp dacă informații noi sunt disponibile. Cu WebSocket, nevoia pentru polling este eliminată.

WebSocket este un protocol relativ nou și este implementat doar pe browserele moderne. Pentru browserele mai vechi există librării JS care simulează WebSocket folosind AJAX.

WebSocket și frameworkurile JS pentru SPA nu sunt mutual exclusive. De exemplu, Meteor folosește WebSocket atunci când clientul suportă acest protocol și *SockJS*⁹ atunci când protocolul nu este suportat.

1.4 Plugin-uri pentru browser

Java Applet-urile, o tehnologie creată de *Sun Microsystems* (cumpărat de Oracle) au promis prin anii '90 că vor revoluționa modul de dezvoltare al aplicațiilor web, dar nu au reușit să se țină de promisiune. Rata lor de adopție a rămas foarte mică.

Macromedia, ulterior achiziționat de *Adobe* a reușit să facă o treabă mult mai bună cu *Flash*, care a atins o rată de adopție mult mai mare, dar această rată este în prezent în continuă scădere. Unul din motivele începutului sfârșitului pentru Flash a fost decizia companiilor *Google* și *Apple* de a nu suporta plugin-ul pe platformele lor mobile (*Android* și *iOS*). În prezent cel mai cunoscut site care folosește Flash este YouTube, dar acesta oferă și o alternativă bazată pe HTML5.

Microsoft a creat un rival pentru Flash, și anume *Silverlight*, dar tehnologia a rămas aproape necunoscută, singurul site mare care a adoptat această tehnologie fiind *NetFlix*.

Aceste plugin-uri ușurau crearea de aplicații web complexe pentru dezvoltatori, dar desigur că era și un preț de plătit:

- Cerințe mari de resurse pentru calculatorul utilizatorului.
- Controlate de o companie.
- Closed-source.
- Interoperabilitate scăzută (suport scăzut pentru Linux, platforme mobile, browsere mai puțin cunoscute).

Din fericire, industria web "a decis" într-un mod organic să sprijine *Open Web*¹⁰, iar în prezent chiar și companiile din spatele acestor plugin-uri suportă, în ceea ce privește web-ul cel puțin, tehnologiile open-source și interoperabilitatea.

⁹<http://sockjs.org>

¹⁰http://en.wikipedia.org/wiki/Open_Web

În comunitatea Python, a spune despre ceva că este isteț,
nu este considerat un compliment.

Alex Martelli

2

Python și Django

În scurta mea carieră de inginer software, am lucrat deja cu trei limbaje de programare: *C#*, *Scala* și *Python*. În acest capitol voi explica de ce am ales Python pentru dezvoltarea back-end-ului acestui proiect.

2.1 Python

Python este un limbaj dinamic, interpretat, care pune accent pe lizibilitate.

Avantaje:

- Foarte ușor de învățat și folosit, poți deveni productiv în doar câteva zile.
- Comunitate puternică, deci se găsesc cu ușurință o mulțime de librării și framework-uri bine scrise și bine documentate și soluții la problemele comune.
- Comunitatea și filosofia¹ limbajului au pus mare accent pe lizibilitatea codului. Codul Python este foarte ușor de înțeles, chiar și de cineva fără experiență cu acest limbaj, iar limbajul permite scrierea codului într-un mod elegant, concis și expresiv.

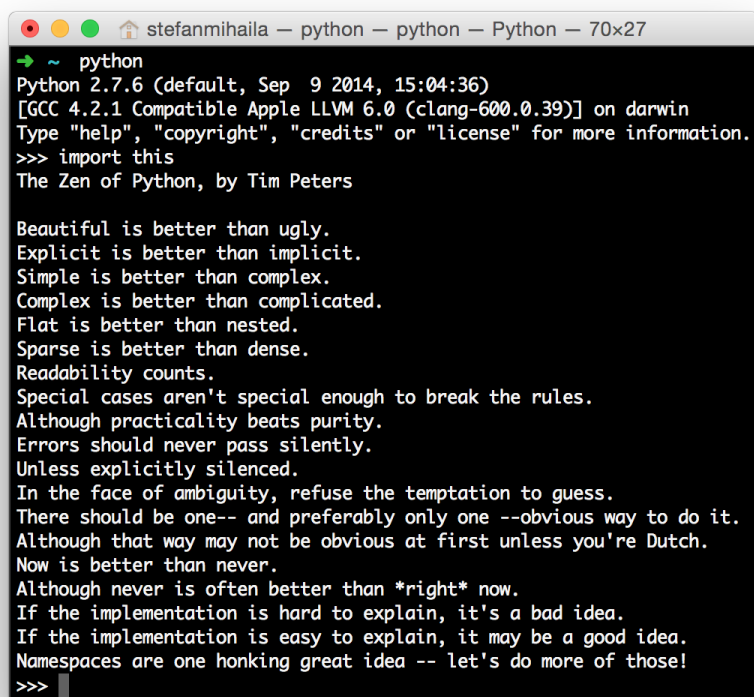
Dezavantaje:

- Fiind un limbaj dinamic, IDE-ul nu înțelege la fel de bine structura codului, iar refactorizările se fac mai greu decât în limbajele statice.
- Este mai încet decât limbajele compilate.

În continuare, pentru comparare, prezint codul sursă al unui *web crawler*² foarte simplu, scris mai întâi în Java, apoi în Python. Codul este inspirat din *Java vs Python Platforms Comparison*, ce poate fi văzut la <https://www.youtube.com/watch?v=ppspz2ZiBaY>.

¹Vezi Figura 2.1

²http://en.wikipedia.org/wiki/Web_crawler

A screenshot of a macOS terminal window. The title bar at the top shows three colored window control buttons (red, yellow, green) followed by the text 'stefanmihaila — python — python — Python — 70x27'. The terminal content shows a shell prompt '~' followed by the command 'python'. This launches the Python 2.7.6 interpreter, which displays its version, build date (Sep 9 2014, 15:04:36), and compiler information ([GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin). It also shows the help options: 'Type "help", "copyright", "credits" or "license" for more information.' The user then enters the command '>>> import this', which triggers the display of 'The Zen of Python, by Tim Peters'. The text of the Zen of Python is displayed line by line, including principles like 'Beautiful is better than ugly.', 'Explicit is better than implicit.', 'Simple is better than complex.', 'Complex is better than complicated.', 'Flat is better than nested.', 'Sparse is better than dense.', 'Readability counts.', 'Special cases aren't special enough to break the rules.', 'Although practicality beats purity.', 'Errors should never pass silently.', 'Unless explicitly silenced.', 'In the face of ambiguity, refuse the temptation to guess.', 'There should be one-- and preferably only one --obvious way to do it.', 'Although that way may not be obvious at first unless you're Dutch.', 'Now is better than never.', 'Although never is often better than *right* now.', 'If the implementation is hard to explain, it's a bad idea.', 'If the implementation is easy to explain, it may be a good idea.', and 'Namespaces are one honking great idea -- let's do more of those!'. The prompt '>>>' is followed by a cursor.

```

→ ~ python
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>> █

```

Figura 2.1: The Zen of Python

Main.java

```
import java.io.*;
import java.net.*;
import java.util.Scanner;
import java.util.regex.*;

public class Main {

    public static void main(String[] args) throws IOException {
        PrintWriter textFile = null;
        try {
            textFile = new PrintWriter("result.txt");
            System.out.println("Enter the URL you wish to crawl..");
            System.out.print("@>");
            String targetUrl = new Scanner(System.in).nextLine();

            String response = getContentByUrl(targetUrl);

            Matcher matcher = Pattern.compile(
                "href=[\"'](?:[^\\"'"]+|\"\"|'')\""
            ).matcher(response);
            while (matcher.find()) {
                String url = matcher.group(1);
                System.out.println(url);
                textFile.println(url);
            }
        } finally {
            if (textFile != null) {
                textFile.close();
            }
        }
    }

    private static String getContentByUrl(String urlString)
        throws IOException {
        URL url = new URL(urlString);
        URLConnection urlConnection = url.openConnection();
        BufferedReader in = null;
        StringBuilder response = new StringBuilder();
        try {
            in = new BufferedReader(new InputStreamReader(
                urlConnection.getInputStream()));
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
        } finally {
            if (in != null) {
                in.close();
            }
        }
    }
}
```

```

    }
    }
    return response.toString();
}
}

```

Iar acum, varianta Python:

```

crawler.py

import re
import urllib.request

def main():
    with open("result.txt", "wt") as text_file:
        print("Enter the URL you wish to crawl..")
        url = input("@> ")
        print(url)
        for i in re.findall(
            "href=[\" '\"](.\"^\\\" '\"]+)[\" '\"]",
            urllib.request.urlopen(url).read().decode(), re.I):
            print(i)
            text_file.write(i + '\n')

if __name__ == '__main__':
    main()

```

Ambele programe fac același lucru: primesc un URL de la utilizator, cer pagina aflată la acel URL și folosesc o expresie regulată pentru a găsi toate link-urile din pagina respectivă. Totuși, se poate observa că varianta scrisă în Python este mai scurtă și mai ușor de înțeles.

2.2 Django

Django este un framework web implementat în Python. Motto-ul lui este: "Un framework pentru perfecționiști cu termene limită".

Django poate fi considerat un framework MVC, dar folosește o terminologie proprie:

- *Modelul* este reprezentat de ORM-ul (Object-relational mapping)³ framework-ului care face legătura dintre obiecte Python și tabele dintr-o bază de date relațională.
- Clasele sau funcțiile care creează răspunsul la request sunt denumite *view*-uri. Acestea sunt denumite *controller*-ere în alte framework-uri MVC.
- Datele sunt prezentate folosind șabloane (templates). Acestea sunt ceea ce alte framework-uri numesc *view*-uri.
- *Controller*-ul este reprezentat de însuși framework, care rutează un request la view-ul corespunzător URL-ului cerut (prin URL dispatcher).

³http://en.wikipedia.org/wiki/Object-relational_mapping

Orice aplicație ce poate fi scrisă în JavaScript, va fi, la un moment dat, scrisă în JavaScript.

Jeff Atwood

3

JavaScript și AngularJS

JavaScript, un limbaj de programare controversat, a ajuns să fie cel mai important limbaj al web-ului. AngularJS este un framework JavaScript creat de Google pentru a depăși limitările HTML-ului și pentru a ușura crearea aplicațiilor SPA. În acest capitol voi descrie aceste două tehnologii.

3.1 JavaScript

JavaScript, cunoscut și ca ECMAScript a fost creat în 1995, de Brendan Eich, pentru Netscape. În ciuda numelui, JavaScript nu are nimic în comun cu Java în afară de sintaxa inspirată de C.

Brendan Eich, un iubitor al limbajelor funcționale, își dorea să creeze un limbaj asemănător cu *Scheme*¹, care este un dialect a lui *Lisp*². Netscape avea însă alte scopuri. JavaScript a apărut în perioada în care se credea că applet-urile Java vor cuceri lumea, iar Netscape dorea un limbaj interpretat, dinamic, ca o alternativă mai puțin intimidantă a lui Java, pentru programatorii amatori.

Astfel, Eich a fost nevoit să creeze un limbaj cu o sintaxă bazată pe cea a lui C, dar iubirea lui a față de limbajele funcționale poate fi observată totuși în JS. JS este unul din primele limbaje folosite la scară mare care are funcții anonime (cunoscute și ca expresii lambda). Java a introdus funcțiile anonime abia în 2014, în versiunea 8.

Faptul că JS a încercat să fie un limbaj de programare adresat și amatorilor, a avut niște consecințe negative pentru limbaj. Limbajul încearcă să fie iertător atunci cu un programator ce nu știe ce face, lucru care care a dus la comportamentul neașteptat al limbajului pentru un programator care știe ce face. Mult timp, comunitatea programatorilor i-a privit pe programatorii JavaScript ca fiind amatori.

¹[http://en.wikipedia.org/wiki/Scheme_\(programming_language\)](http://en.wikipedia.org/wiki/Scheme_(programming_language))

²[http://en.wikipedia.org/wiki/Lisp_\(programming_language\)](http://en.wikipedia.org/wiki/Lisp_(programming_language))

Pe lângă toate astea, JS a fost mereu asociat cu API-ul de manipulare al DOM-ul (Document Object Model)³, care a fost implementat în mod diferit de fiecare browser, lucru care a făcut ca programarea în JS să fie grea și neelegantă.

Din aceste motive, nu mulți s-ar fi așteptat ca, dintre toate limbajele, JS să devină cel mai popular și mai căutat limbaj din lume. Dar s-a întâmplat. Mai întâi, Ajax l-a adus în prim-plan. jQuery⁴, o librărie scrisă în JavaScript, a venit să simplifice Ajax și să ascundă diferențele dintre browsere. Partea de front-end a aplicațiilor a devenit din ce în ce mai complexă, astfel tot mai mulți programatori au devenit interesați de ea, ducând la librării din ce în ce mai bune.

Pe partea de client, JS este câștigător incontestabil, fiind singurul limbaj cunoscut de toate browser-ele în mod nativ. Singura alternativă o reprezintă limbajele care sunt compilate în JS, cum ar fi CoffeeScript, ClojureScript etc, dar acestea sunt mult mai puțin folosite. Pentru ele, JS este noul limbaj de asamblare.

Pe lângă faptul că JS este de departe cel mai folosit limbaj pe browser, acesta nu s-a oprit aici. JavaScript a "scăpat" din browser. Prin intermediul lui Node.js⁵, o tehnologie ce capătă amploare din ce în ce mai mare, JavaScript poate fi folosit și pe server.

MEAN⁶ (MongoDB Express Angular.js Node.js) este o combinație de tehnologii care îi face pe dezvoltatori să se bucure de avantajele folosirii unui singur limbaj la toate nivelele aplicației, de la front-end, trecând prin back-end și până în baza de date. Acesta a fost numit de unii ca fiind noul LAMP⁷ (Linux Apache MySQL PHP).

În concluzie, indiferent de părerile personale și dacă ne place sau nu, un lucru este cert: în momentul de față, JavaScript este cel mai important jucător din lumea limbajelor de programare. JavaScript este pe locul 1 la numărul de repository-uri pe GitHub și cel mai căutat limbaj de programare în anunțurile de angajare.

3.2 AngularJS

În această secțiune voi trece foarte rapid peste componentele de bază ale lui AngularJS. Scopul secțiunii este sublinierea elementelor esențiale ale lui Angular: șabloane, directive, controllere, servicii.

AngularJS este un framework MVC ce folosește HTML-ul ca bază pentru limbajul său de șabloane. El extinde HTML-ul cu structuri de control, de exemplu pentru iterare și permite și definirea de noi *directive*, care pot fi elemente sau atribute HTML.

În continuare prezint o mică porțiune din codul tutorialului oficial Angular, disponibil la https://docs.angularjs.org/tutorial/step_11.

app/index.html

```
1 <body ng-controller="PhoneListCtrl">
```

³http://en.wikipedia.org/wiki/Document_Object_Model

⁴<https://jquery.com/>

⁵<https://nodejs.org>

⁶<http://mean.io>

⁷[http://en.wikipedia.org/wiki/LAMP_\(software_bundle\)](http://en.wikipedia.org/wiki/LAMP_(software_bundle))

```

2
3   <ul>
4     <li ng-repeat="phone_in_phones">
5       <span>{{phone.name}}</span>
6       <p>{{phone.snippet}}</p>
7     </li>
8   </ul>
9
10 </body>

```

```

                                app/js/services.js
1  var phonecatServices = angular.module(
2    'phonecatServices',
3    ['ngResource']);
4
5  phonecatServices.factory('Phone', ['$resource',
6    function($resource){
7    return $resource('phones/:phoneId.json', {}, {
8      query: {
9        method: 'GET',
10       params: {
11         phoneId: 'phones'
12       },
13       isArray: true
14     }
15   }));
16   }]);

```

```

                                app/js/controllers.js
1  var phonecatControllers = angular.module(
2    'phonecatControllers', []);
3
4  phonecatControllers.controller('PhoneListCtrl', [
5    '$scope', 'Phone', function($scope, Phone) {
6    $scope.phones = Phone.query();
7  }]);

```

În `index.html`, linia 1, prin intermediul atributului HTML `ng-controller` (ce corespunde directivei Angular `ngController`) Angular află că pentru toate elementele din interiorul elementului `body`, `scope`-ul este luat din controller-ul cu numele `PhoneListCtrl`, definit în fișierul `app/js/controllers.js` (linia 4).

Linia 4 din `index.html` folosește directiva `ngRepeat` pentru a itera prin array-ul cu numele `phones` de pe `scope`-ul curent. Acest array este declarat în `controllers.js`, linia 6.

În `controllers.js`, linia 5, `'Phone'` este pasat ca argument al funcției în interiorul unui array. Acest lucru îi spune framework-ului să injecteze serviciul `Phone`, definit în `app/js/services.js`, linia 5. Acest serviciu se folosește de `ngResource` pentru a crea mai multe metode ce apelează un API REST aflat la adresa `/phones`. Astfel, deși uitându-ne în controller putem avea senzația

că `$scope.phones` este populat sincron, în realitate, metoda `query` din serviciu face un HTTP GET la adresa `/phones` de unde un obiect JSON este returnat, iar `$scope.phones` este populat cu acest obiect.

4

Alte tehnologii folosite în aplicație

4.1 Bootstrap

*Bootstrap*¹, dezvoltat inițial de Mark Otto și Jacob Thornton la Twitter, este un framework ce simplifică stilizarea elementelor HTML cum ar fi form-uri și butoane și crearea de meniuri de navigație, casete modale etc. De asemenea, o parte foarte importantă a framework-ului este crearea aplicațiilor *responsive*, ceea ce înseamnă că aplicația este afișată corect pe dispozitive cu display-uri de dimensiuni diferite: calculatoare, tablete și telefoane.

Folosirea framework-ului este destul de simplă. Este suficient să fie adăugate două fișiere din Bootstrap, apoi clasele CSS pot fi folosite pentru stilizarea elementelor HTML.

4.2 Bower

*Bower*² este o unealtă folosită pentru managementul dependențelor de pe front-end. Bower folosește fișierul `bower.json` pentru a ști ce trebuie să instaleze.

app/bower.json

```
{
  "name": "money-keep",
  "version": "0.0.0",
  "homepage": "https://github.com/stefan-mihaila/money-keep",
  "authors": [
    "Stefan Daniel Mihaila <stefan.mihaila@gmail.com>"
  ],
  "private": true,
  "ignore": [
```

¹<http://getbootstrap.com>

²<http://bower.io>


```

    "**/*.*",
    "node_modules",
    "bower_components",
    "test",
    "tests"
  ],
  "dependencies": {
    "angular": "~1.3.8",
    "bootstrap": "~3.3.1",
    "lodash": "~2.4.1",
    "angular-route": "~1.3.8",
    "angular-cookies": "~1.3.9",
    "snackbarjs": "~1.0.0",
    "ngDialog": "~0.3.9"
  },
  "resolutions": {
    "angular": "1.3.9"
  }
}

```

Dependențele din `bower.json` sunt descărcate dintr-un repository central în directorul `static/bower_components` cu comanda `bower install`.

4.3 Git și GitHub

*Git*³ este un sistem distribuit pentru controlul sistemului de reviziuni al fișierelor, scris inițial de creatorul kernel-ului Linux, Linus Torvalds. *GitHub*⁴ este un serviciu online ce permite găzduirea repository-urilor Git.

Atât codul aplicației, cât și codul acestei lucrări, redactată folosind \LaTeX , sunt găzduite pe GitHub la adresele <https://github.com/stefan-mihaila/money-keep> și <https://github.com/stefan-mihaila/thesis>.

4.4 API REST (Django REST Framework)

REST⁵ (Representational State Transfer) este o modalitate de comunicare între servicii HTTP. REST a venit ca o alternativă la SOAP⁶ și WSDL⁷.

SOAP și WSDL au încercat să ușureze munca dezvoltatorilor construind abstractizări deasupra arhitecturii HTTP. REST a adoptat o altă strategie: a introdus conceptul de resurse și a folosit metodele HTTP existente pentru a procesa aceste resurse.

Metodele HTTP sunt următoarele:

OPTIONS este metoda HTTP ce oferă clientului posibilitate de a afla ce operații sunt disponibile pentru o resursă, sau ce capacități are serverul, fără însă a modifica resursa respectivă.

³<https://git-scm.com>

⁴<http://github.com>

⁵https://en.wikipedia.org/wiki/Representational_state_transfer

⁶<https://en.wikipedia.org/wiki/SOAP>

⁷https://en.wikipedia.org/wiki/Web_Services_Description_Language

GET este metoda HTTP ce oferă detaliile unei resurse. Această metodă este sigură și idempotentă.

HEAD este o metodă care se comportă ca și GET, cu excepția că serverul nu răspunde cu un BODY la acest request, ci întoarce doar HEADER-ele HTTP.

POST este metoda HTTP ce creează o resursă nouă. Serverul trebuie să răspundă cu 201 (Created) atunci când resursa a fost creată cu succes și răspunsul trebuie să conțină headerul LOCATION cu URL-ul resursei nou creată.

PUT este metoda HTTP folosită pentru a edita o resursă. Toate proprietățile resursei trebuie trimise de client, iar serverul editează suprascrie proprietățile resursei. În cazul în care totul a decurs bine, răspunsul serverului trebuie să fie 204 (No Content).

DELETE este metoda HTTP ce șterge resursa de pe server.

Proprietăți metodelor HTTP sunt următoarele:

siguranța Metodele GET și HEAD sunt considerate sigure, deoarece ele nu au efecte secundare, cum este cazul metodelor POST, PUT și DELETE.

idempotența Metodele GET, HEAD, PUT și DELETE sunt idempotente, adică rezultatul este același indiferent dacă ele sunt chemate o dată sau de mai multe ori.

REST se pretează foarte bine arhitecturilor client server. În REST, se consideră că există o mapare între URL-uri și resurse. Avantajul REST este legat de faptul că acesta folosește arhitectura existentă HTTP. De exemplu, pentru că se folosesc metodele HTTP standard, resursele accesate prin metoda GET pot fi stocate pe CDN-uri pentru acces rapid, pentru că răspunsul acestei metode poate fi cacheable.

Django REST Framework este un plugin de Django ce permite crearea rapidă de resurse HTTP folosindu-se modelele din Django. În plus, endpoint-urile acestui framework oferă o interfață web ce ușurează debugging-ul și dezvoltarea rapidă.

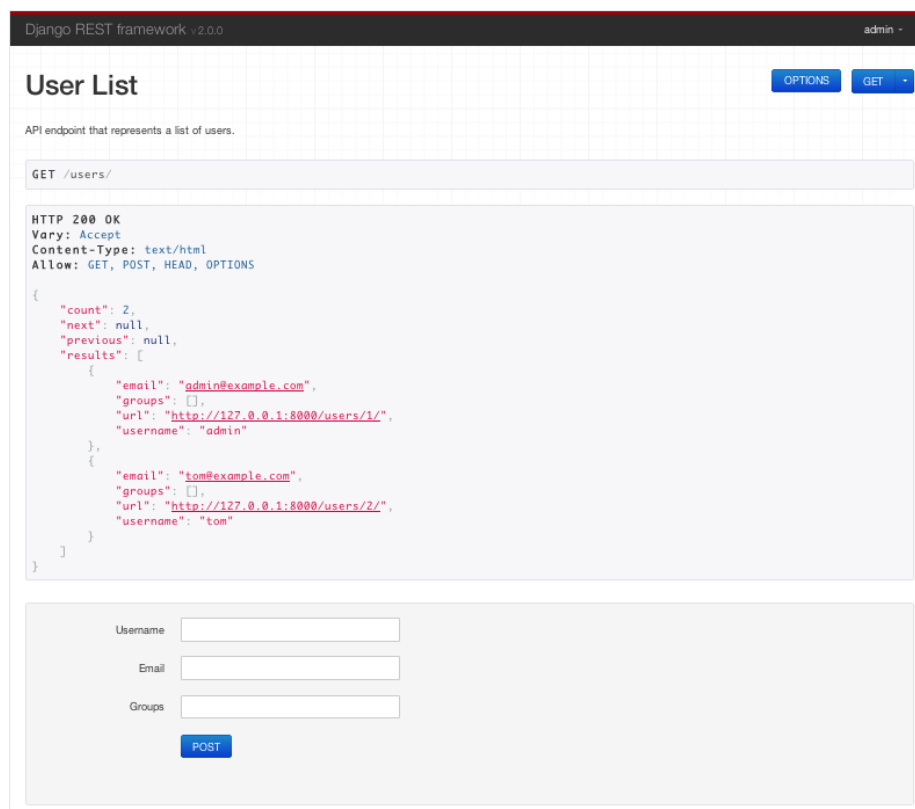


Figura 4.1: Interfață web REST Framework

5

Descrierea aplicației

În acest capitol voi descrie pe scurt aplicația SPA creată pentru lucrarea de licență.

5.1 Scop și prezentare generală

Aplicația este un *expense tracker*, adică permite unui utilizator să țină evidența cheltuielilor. Ea permite unui utilizator să se înregistreze sau să se logheze, apoi acesta are acces la operațiile CRUD (Create Read Update Delete) asupra cheltuielilor sale.

Aplicația permite și căutarea cheltuielilor după cuvinte ce se găsesc în descrierea acestora, sau filtrarea după dată. De asemenea se pot printa toate cheltuielile dintr-o săptămână.

5.2 Instalare aplicație și pornire server de dezvoltare

Următorii pași trebuie urmați pentru pornirea server-ului de dezvoltare pe un sistem Linux / Mac OS X:

1. Se clonează proiectul folosind comanda "git clone <https://github.com/stefan-mihaila/money-keep>".
2. Se schimbă directorul curent în cel al aplicației: "cd money-keep".
3. Se recomandă crearea unui mediu izolat pentru instalarea pachetelor Python necesare aplicației. Pentru asta, se folosește comanda "pip install virtualenv" pentru instalarea utilitarului *virtualenv* și comanda "virtualenv create env" pentru crearea mediului virtual. Apoi se activează acest

mediu folosind "source env/bin/activate". De acum încolo, comenzile "python" și "pip" nu mai sunt cele instalate global în sistem, ci cele din "env/bin/". Pentru dezactivarea mediului virtual se rulează "deactivate".

4. Se instalează pachetele adiționale Python folosite (incluzând Django). Aceste pachete sunt enumerate în fișierul 'requirements.txt'. Toate pachetele din acest fișier pot fi instalate rulând comanda "pip install -r requirements.txt".
5. Se rulează migrațiile cu comanda "./manage.py migrate".
6. Se instalează Node.js și npm. Instrucțiuni de instalare se găsesc aici: <https://nodejs.org>. npm este necesar pentru a instala Bower.
7. Se instalează Bower folosind comanda "npm install -g bower".
8. Se instalează dependențele de front-end folosind comanda "bower install". Această comandă instalează dependențele listate în "bower.json".
9. Se poate porni serverul cu comanda "./manage.py runserver". Dacă totul a mers bine, serverul acceptă request-uri la adresa <http://127.0.0.1:8000>.

Comanda `./manage.py migrate` de mai sus a creat baza de date a aplicației. Baza de date este configurată în fișierul `money_keep/settings.py`:

Configurare implicită a bazei de date

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

După cum se observă, în mod implicit se folosește *SQLite*. SQLite este o bază de date ce își ține toate tabelele într-un singur fișier (`db.sqlite3` în cazul nostru). Am ales să folosesc această bază de date pentru dezvoltare deoarece este foarte ușor de folosit / administrat. Pentru a se folosi o bază de date "mai serioasă", este necesar doar să fie instalată baza de date și să se modifice dicționarul `DATABASES` din `settings.py`.

5.3 Folosirea aplicației

Se face click pe "Register", apoi se completează formularul cu email-ul, numele utilizatorului și parola (vezi Figura 5.1). După click pe "Submit", utilizatorul este creat și logat în aplicație.

După ce s-a logat, utilizatorul poate adăuga o nouă cheltuială în aplicație dând click pe "New expense". Acesta este rugat să introducă detaliile cheltuielii: data, ora, descrierea, suma și un comentariu opțional (vezi Figura 5.2).

După ce au fost adăugate cheltuieli, acestea pot fi editate sau șterse. De asemenea, cheltuielile pot fi filtrate după dată sau se pot efectua căutări după cuvinte ce se găsesc în titlul sau comentariul cheltuielilor. Există și opțiunea de a printa cheltuielile dintr-o anumită săptămână.

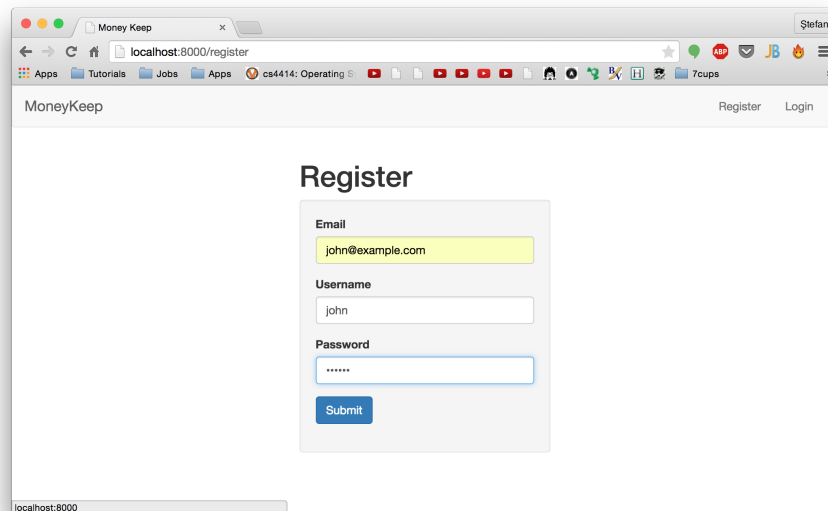


Figura 5.1: Crearea unui utilizator nou

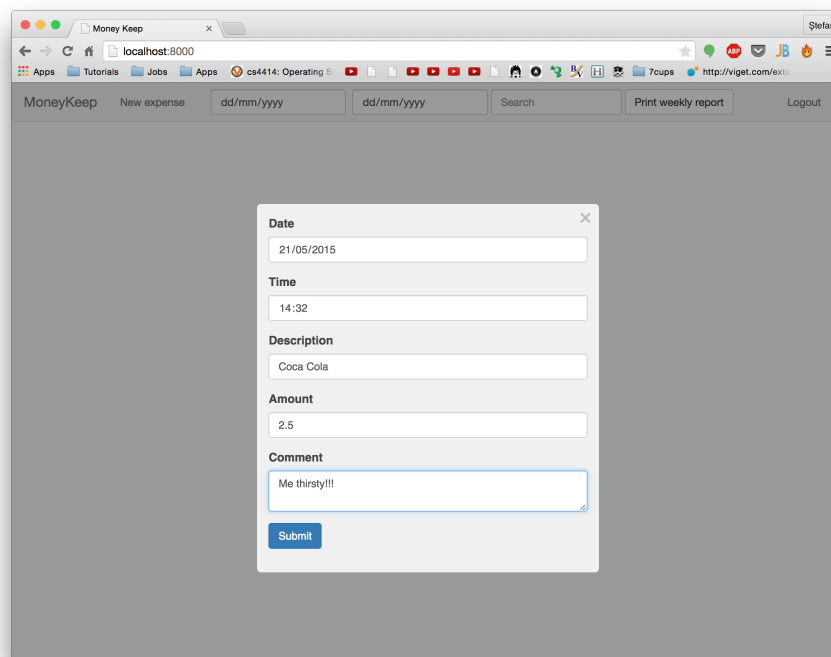


Figura 5.2: Adăugarea unei cheltuieli noi

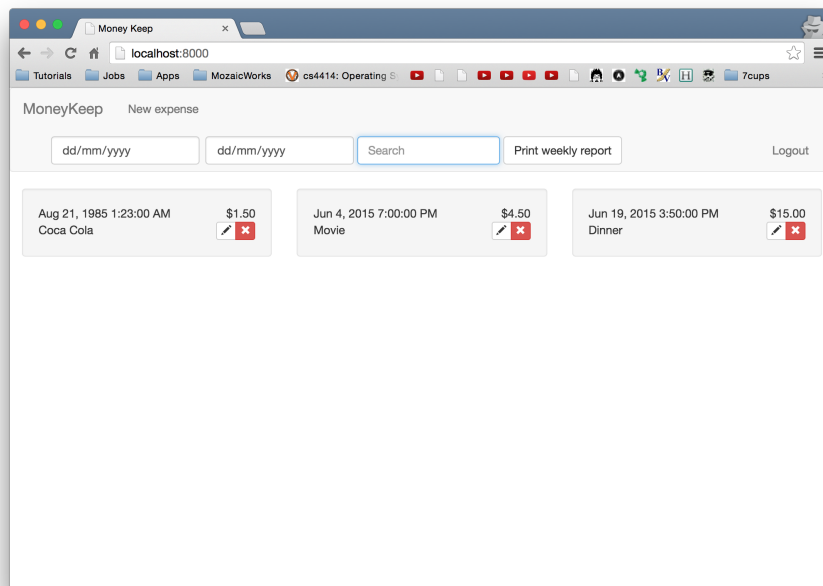


Figura 5.3: Afișarea cheltuielilor

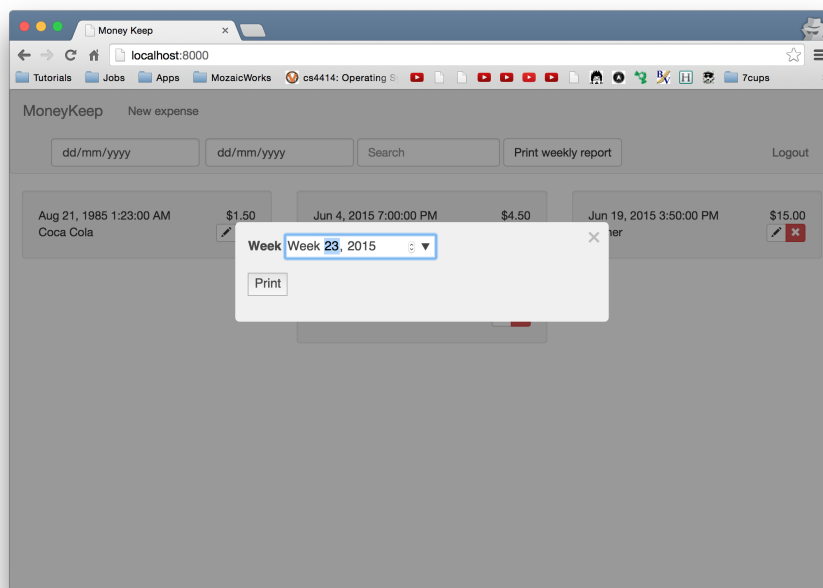


Figura 5.4: Selectare săptămână printare

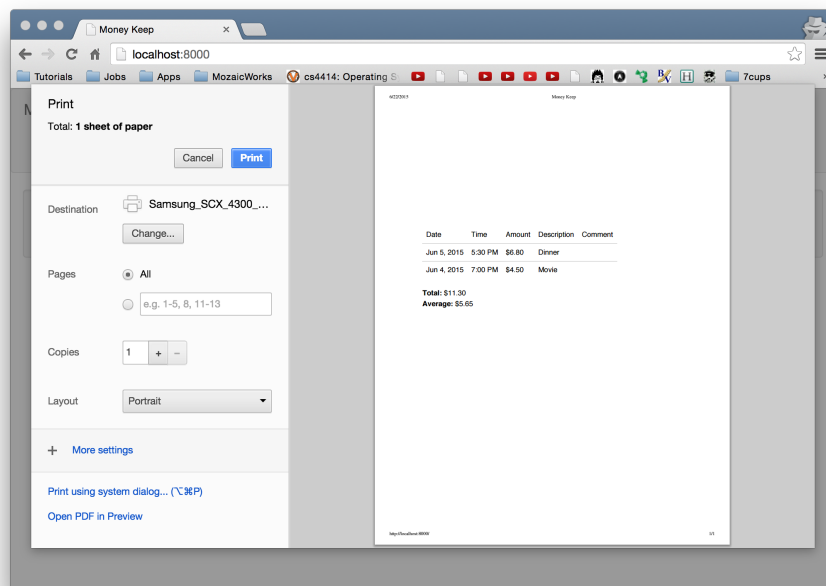


Figura 5.5: Printare

6

Detalii de implementare

6.1 Structura de directoare

Aplicația, la nivelul cel mai înalt, are 3 module Python (interpretorul Python directoarele ce conțin fișierul gol `__init__.py` ca fiind module):

authentication este modulul responsabil înregistrare și autentificarea utilizatorilor. Acest modul conține modelul ORM al utilizatorului și view-urile REST corespunzătoare creării de utilizatori noi și autentificării unui utilizator.

expenses este modulul ce conține modelul ORM al unei cheltuieli și view-urile REST corespunzătoare operațiilor CRUD asupra cheltuielilor.

money_keep este modulul responsabil de configurarea aplicației. Conține, printre altele, fișierul de configurare `settings.py` și fișierul ce conține toate rutele aplicației (URL-urile la care aplicația răspunde), `urls.py`.

Pe lângă aceste module, la nivelul cel mai înalt al aplicației mai sunt directoarele:

static conține toate resursele statice:

- componentele instalate cu *Bower*
- CSS-urile customizate ale aplicației
- întreaga aplicație AngularJS, cu toate șabloanele, controllerele, directivele și serviciile ei.

templates conține șabloanele Django. Ele sunt scrise folosind sintaxa *Django Template Language* (DTL) și nu trebuie confundate cu șabloanele din directorul "static/templates". Șabloanele din acest director sunt folosite de Django, iar cele din "static" sunt folosite de AngularJS.

Structura de directoare a aplicației

```
-- authentication
|   |-- migrations
-- expenses
|   |-- migrations
-- money_keep
-- static
|   |-- bower_components
|   |-- css
|   |-- js
|       |-- authentication
|       |   |-- controllers
|       |   |-- services
|       |-- expenses
|       |   |-- controllers
|       |   |-- directives
|       |   |-- services
|       |-- layout
|       |   |-- controllers
|       |-- utils
|       |   |-- controllers
|       |   |-- services
|       |-- templates
|       |   |-- authentication
|       |   |-- expenses
|       |   |-- layout
|       |   |-- utils
-- templates
```

6.2 Module AngularJS

Pe partea de front-end, aplicația este structurată în următoarele module:

authentication este modulul responsabil de autentificare. Acest modul conține serviciul *Authentication* care are metodele pentru autentificare, logout și verificarea dacă utilizatorul este autentificat.

expenses este modulul responsabil de operațiile CRUD pentru o cheltuială. Pe lângă serviciul care are apelează metodele REST ale back-end-ului, în acest modul sunt definite și directivele AngularJS *expense* și *expenses*. Aceste directive fac posibilă folosirea în șabloane AngularJS a elementelor HTML `<expense>` și `<expenses>`.

layout este modulul care conține logica de filtrare / căutare a cheltuielilor și leagă butoanele din bara de navigare de funcționalitatea acestora.

utils este un modul care conține logica de printare precum și un serviciu utilitar ce se ocupă cu prelucrarea obiectelor date / time.

6.3 API REST

Deoarece front-end-ul aplicației este scris cu AngularJS, comunicarea dintre front-end și back-end se face prin intermediul unui API REST. Acest API REST a fost creat cu ajutorul unui plugin Django: *Django REST Framework*¹.

Django REST Framework permite crearea rapidă de API-uri REST pentru aplicațiile Django. Pentru a îl folosi, în aplicație am creat câte o clasă de serializare pentru fiecare model ce trebuie serializat (`Account` și `Expense`). Aceste clase moștenesc din `rest_framework.serializers.ModelSerializer`. În clasele de serializare specificăm ce câmpuri vrem să fie serializate. Clasele de serializare pot fi văzute în "authentication/serializers.py" și "expenses/serializers.py".

În afară de clasele de serializare, am creat și permisiuni pentru fiecare model. Folosindu-ne de aceste permisiuni, ne asigurăm că un utilizator nu poate citi datele altui utilizator. Permisunile se găsesc în fișierele "permissions.py" din modulele "authentication" și "expenses".

Având permisiunile și clasele de serializare, folosim clasele din `rest_framework` pentru a customiza mai departe operațiile pe care le dorim: de exemplu pentru filtrarea cheltuielilor după dată, sau pentru a crea un utilizator nou stocându-i parola în formă hash.

API-ul creat este următorul:

POST `/api/v1/auth/login/` permite logarea unui utilizator în aplicație. Parametrii primiți sunt adresa de email a utilizatorului și parola.

POST `/api/v1/auth/logout/` deautentică utilizatorul curent.

POST `/api/v1/accounts/` crează un utilizator nou. Primește ca parametri numele utilizatorului, parola și adresa de email.

GET `api/v1/expenses/` întoarce lista cu toate cheltuielile pentru utilizatorul logat în aplicație.

POST `/api/v1/expenses/` adaugă o cheltuială nouă.

DELETE `api/v1/expenses/{:id}/` șterge cheltuiala cu id-ul specificat.

6.4 Fluxul aplicației

Fiind o aplicație SPA ce folosește un framework MVC pe back-end și un alt framework MVC pe front-end, rutarea URL-urilor are loc la două nivele.

Orice request către aplicație este mai întâi redirectionat de URL handler-ul Django. În fișierul `urls.py` sunt configurate toate rutele aplicației. Indiferent ce pagină cere utilizatorul, acestuia îi este servit șablonul Django `index.html` ce conține întreaga aplicație AngularJS.

A doua rutare are loc la nivelul AngularJS. Serviciul de autentificare din AngularJS verifică dacă utilizatorul este autentificat sau nu. Dacă nu este, utilizatorul este redirectat către pagina de autentificare. Este important să înțelegem că acest redirect se întâmplă doar prin Ajax. AngularJS este cel care cere șablonul HTML "login.html" de pe server și îl afișează pe client, fără ca pagina să fie reîmprospătată.

¹<http://www.django-rest-framework.org>

6.5 Autentificarea

Django, prin structura sa modulară, include un modul de autentificare, numit `auth`. Acest modul conține un model, numit `User` ce poate fi folosit de aplicațiile Django pentru autentificare. În cazult acestei aplicații, am decis să nu folosesc acest model, ci să-mi creez propriul model, ce moștenește din `AbstractBaseUser`, model din care moștenește și clasa `User`. În acest mod, am putut modifica logarea standard din Django, bazată pe `username`, cu logarea bazată pe adresă de email.

```
authentication/models.py

import uuid
from django.contrib.auth.models import (AbstractBaseUser,
                                         BaseUserManager)
from django.core.mail import send_mail
from django.db import models

class AccountManager(BaseUserManager):
    def create_user(self, email, username, password,
                    first_name='', last_name=''):
        if not email:
            raise ValueError(
                "Users must have a valid email address.")

        if not username:
            raise ValueError(
                "Users must have a valid username.")

        account = self.model(email=self.normalize_email(email),
                              username=username,
                              first_name=first_name,
                              last_name=last_name)
        # TODO: create account as inactive and
        # send activation link
        account.set_password(password)
        account.activation_code = uuid.uuid4().hex
        account.save()

        return account

    def create_superuser(self, email, username, password,
                        first_name='', last_name=''):
        account = self.create_user(email, username, password,
                                    first_name=first_name,
                                    last_name=last_name)

        account.is_admin = True
        account.save()
```

```

        return account

def activate_user(self, activation_code):
    try:
        account = self.get(activation_code=activation_code)
    except Account.DoesNotExist:
        raise ValueError('Invalid_activation_code.')

    if account.is_active:
        return False

    account.is_active = True
    account.save()
    return True


class Account(AbstractBaseUser):
    email = models.EmailField(unique=True)
    username = models.CharField(max_length=30, unique=True)

    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=30, blank=True)

    is_admin = models.BooleanField(default=False)

    activation_code = models.CharField(max_length=32,
                                       unique=True)

    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    objects = AccountManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username']

    def __unicode__(self):
        return self.email

    def get_full_name(self):
        return '_'.join([self.first_name, self.last_name])

    def get_short_name(self):
        return self.first_name

    def email_user(self, subject, message,
                  from_email=None, **kwargs):
        send_mail(subject, message, from_email,
                  [self.email], **kwargs)

```

Autentificarea este pe bază de cookie. Pe pagina de autentificare, AngularJS face POST la endpoint-ul "api/v1/auth/login". Dacă user-ul și parola se potrivesc, server-ul răspunde cu codul 200 (OK). În acest caz AngularJS folosește serviciul \$cookies pentru a reține datele de sesiune primite înapoi de la server. Acestea sunt trimise până la logout cu fiecare request, astfel serverul știe că utilizatorul s-a autentificat în aplicație.

Codul JavaScript ce se ocupă de autentificare este în directorul "static/js/authentication". Aici se găsesc controllerele pentru login și register și serviciul de autentificare. Șabloanele corespunzătoare loginului și funcției de înregistrare utilizator nou sunt în directorul "static/templates".

În fișierul *authentication.service.js* este definit serviciul AngularJS *Authentication*. Acest serviciu are funcțiile:

register primește ca parametri email-ul, username-ul și parola. Cu acestea, face POST la endpoint-ul *api/v1/accounts*. Dacă răspunsul este 200 (OK), ceea ce înseamnă că backendul a creat un utilizator nou cu credențialele trimise, serviciul cheamă funcția "login".

login ia ca parametrii emailul și parola, cu care face post la endpoint-ul 'api/v1/auth/login'. Dacă răspunsul este 200 (OK) atunci este setat un cookie cu utilizatorul autentificat, iar utilizatorului i se afișează pagina cu cheltuielile sale.

isAuthenticated verifică dacă a fost setat un cookie cu utilizatorul curent. Returnează **true** dacă utilizatorul s-a autentificat sau **false** dacă utilizatorul nu este autentificat.

logout face POST la endpoint-ul 'api/v1/auth/logout' care îi spune server-ului să deautentifice utilizatorul curent. După POST funcția **unauthenticate()** este chemată.

unauthenticate șterge utilizatorul autentificat din cookie.

```
static/js/authentication/services/authentication.service.js

(function () {
    'use_strict';

    angular
        .module('moneyKeep.authentication.services')
        .factory('Authentication', Authentication);

    Authentication.$inject = ['$http', '$cookies'];

    function Authentication($http, $cookies) {
        var Authentication = {
            register: register,
            login: login,
            isAuthenticated: isAuthenticated,
            logout: logout,
            getAuthenticatedAccount: getAuthenticatedAccount
        };
    }
});
```

```

return Authentication;

function register(email, username, password) {
    return $http.post('api/v1/accounts/', {
        username: username,
        password: password,
        email: email
    }).then(registerSuccessFn, registerErrorFn);

    function registerSuccessFn(response) {
        Authentication.login(email, password);
    }

    function registerErrorFn(response) {
        console.log("Registration_error");
    }
}

function login(email, password) {
    return $http.post('api/v1/auth/login/', {
        password: password,
        email: email
    }).then(loginSuccessFn, loginErrorFn);

    function loginSuccessFn(response) {
        setAuthenticatedAccount(response.data);

        window.location = '/';
        return response;
    }

    function loginErrorFn(response) {
        console.log("Login_error");
        return response;
    }
}

function getAuthenticatedAccount() {
    if (!$cookies.authenticatedAccount) {
        return;
    }

    return JSON.parse($cookies.authenticatedAccount);
}

function setAuthenticatedAccount(account) {
    $cookies.authenticatedAccount = JSON.stringify(account);
}

function isAuthenticated() {

```

```

        return !!$cookies.authenticatedAccount;
    }

    function unauthenticate() {
        delete $cookies.authenticatedAccount;
    }

    function logout() {
        return $http.post('api/v1/auth/logout/')
            .success(function (data, status, headers, config) {
                unauthenticate();
                window.location = '/';
            });
    }
}
})();

```

În șabloanele HTML pentru înregistrarea utilizatorului nou și pentru login se găsește câte un form. În momentul în care utilizatorul face submit acestui form, este apelată o metodă din controller-ul corespunzător (login sau register) care, la rândul său apelează metoda serviciului.

```

static/templates/authentication/register.html
<form role="form" ng-submit="vm.register()">
  <div class="form-group">
    <label for="email">Email</label>
    <input type="email" class="form-control"
      id="email" ng-model="vm.email"
      placeholder="email" required="" />
  </div>
  <!-- ..... -->

```

Prin folosirea directivei "ng-submit", aplicația Angular știe că în momentul în care se face submit form-ului, trebuie chemată funcția "register()" din controller. Prin folosirea directivelor "ng-model", frameworkul crează o legătură bidirecțională între textul din elementele "input" HTML și proprietățile controllerului. De exemplu, de fiecare dată când valoarea inputului "email" se modifică, proprietatea "email" este setată pe controller astfel încât valoarea ei să reflecte valoarea inputului.

6.6 Operații CRUD pentru Expense

Pe partea de client, serviciul care se ocupă de operațiile CRUD corespunzătoare unui cheltuieli se numește *Expenses* și se găsește în fișierul "static/js/expenses/-services/expenses.service.js".

```

static/js/expenses/services/expenses.service.js
(function () {
    'use strict';

```



```

angular
    .module('moneyKeep.expenses.services')
    .factory('Expenses', Expenses);

Expenses.$inject = ['$http'];

function Expenses($http) {
    var Expenses = {
        all: all,
        create: create,
        update: update,
        destroy: destroy
    };

    return Expenses;

    function all(week_contains) {
        var params = {};
        if (week_contains) {
            params.week_contains = week_contains;
        }
        return $http.get('/api/v1/expenses/',
            {params: params})
            .then(function (response) {
                angular.forEach(response.data,
                    function(expense) {
                        expense.timestamp = new Date(
                            expense.timestamp);
                        expense.amount = Number(
                            expense.amount);
                    });
                return response;
            });
    }

    function create(expense) {
        return $http.post('/api/v1/expenses/',
            expense);
    }

    function update(expense) {
        return $http.put('/api/v1/expenses/' +
            expense.id + '/', expense);
    }

    function destroy(expense) {
        return $http.delete('/api/v1/expenses/' +
            expense.id + '/');
    }
}

```

```
    }
  })();
```

Acest serviciu are metodele:

all este funcția care face un HTTP GET la enpoint-ul `/api/v1/expenses/`.

Această funcție primește un parametru opțional: *week_contains*, care este un obiect reprezentând o dată. Dacă parametrul este specificat, acesta este trimis ca parametru GET serverului. Serverul întoarce lista cu toate cheltuielile utilizatorului, dacă parametrul `"week_contains"` nu este specificat, sau o listă cu aferente săptămânii din care data din `"week_contains"` face parte, dacă parametrul este specificat. Funcția întoarce un *Promise*² care este îndeplinit în cazul în care serverul răspunde cu 200 (OK) cu lista de cheltuieli returnate de server.

create este o funcție care primește ca parametru un obiect ce conține detaliile unei cheltuieli (dată, titlu, descriere, comentarii) și face un HTTP POST la endpoint-ul `/api/v1/expenses` cu aceste detalii. Serverul crează o cheltuială nouă cu detaliile primite și o asociază utilizatorului curent.

update este o funcție care primește, ca și create, un obiect cu detaliile unei cheltuieli. Spre deosebire de obiectul pe care-l primește create, în update obiectul trebuie să aibe setat și atributul `"id"`. Funcția face HTTP PUT la endpoint-ul `/api/v1/expenses/id/`, unde `'id'` este înlocuit cu `"expense.id"`. Ca urmare a acestui apel, server-ul ia din baza de date cheltuiala cu ID-ul specificat în adresă și updatează acea cheltuială.

destroy este metoda responsabilă cu ștergerea unei cheltuieli. Această metodă funcționează ca și update, cu diferența că metoda HTTP folosită este DELETE în loc de PUT.

```

                                expenses/views.py
from datetime import datetime, timedelta
import dateutil.parser

from rest_framework import viewsets, permissions
from expenses.models import Expense
from expenses.permissions import IsExpenseOwner
from expenses.serializers import ExpenseSerializer

class ExpensesViewSet(viewsets.ModelViewSet):
    serializer_class = ExpenseSerializer
    queryset = Expense.objects.all()

    def get_permissions(self):
        return (permissions.IsAuthenticated(),
                IsExpenseOwner())

    def get_queryset(self):
```

²<http://www.html5rocks.com/en/tutorials/es6/promises/>

```

queryset = self.queryset.filter(
    account=self.request.user)

week_contains = self.request.QUERY_PARAMS.get(
    'week_contains')
if week_contains:
    dt = dateutil.parser.parse(week_contains)
    start_date = dt - timedelta(days=dt.weekday())
    end_date = start_date + timedelta(days=7)

    queryset = queryset.filter(
        timestamp__gte=start_date,
        timestamp__lt=end_date)

return queryset.order_by('-timestamp')

def perform_create(self, serializer):
    serializer.save(account=self.request.user)

```

Toate aceste metode sunt folosite de controller-ele din "static/js/expenses/controllers", și anume: "EditExpenseController", "ExpenseController" și "NewExpenseController".

De asemenea, în "static/js/expenses/directives" avem definite două directive, și anume: "expenses" și "expense". Aceste directive permit folosirea elementelor HTML <expense> și <expenses> și fac astfel codul mai modular și concis.

```

static/js/expenses/expenses.directive.js

(function () {
    'use strict';

    angular
        .module('moneyKeep.expenses.directives')
        .directive('expenses', expenses);

    function expenses() {
        var directive = {
            controller: 'ExpensesController',
            controllerAs: 'vm',
            restrict: 'E',
            scope: {
                expenses: '='
            },
            templateUrl: '/static/templates/expenses/expenses.html'
        };

        return directive;
    }
})();

```

```

static/js/expenses/expense.directive.js

(function () {
    'use strict';

    angular
        .module('moneyKeep.expenses.directives')
        .directive('expense', expense);

    function expense() {
        var directive = {
            controller: 'ExpenseController',
            controllerAs: 'vm',
            restrict: 'E',
            scope: {
                expense: '='
            },
            templateUrl: '/static/templates/expenses/expense.html'
        };

        return directive;
    }
})();

```

```

static/templates/expenses/expenses.html

<div class="well">
  <div class="row">
    <div class="col-sm-8">
      <span class="pull-left">
        {{ expense.timestamp | date:'medium' }}
      </span>
    </div>
    <div class="col-sm-4">
      <span class="pull-right">
        {{ expense.amount | currency }}
      </span>
    </div>
  </div>
  <div class="row">
    <div class="col-sm-8">
      <div class="expense-description">
        {{ expense.description }}
      </div>
    </div>
    <div class="col-sm-4">
      <div class="btn-group btn-group-xs pull-right">
        <a class="btn btn-default"
          href="javascript:void(0)" ng-click="vm.edit()">
          <span class="glyphicon glyphicon-pencil"
            aria-hidden="true">

```

```

        </span>
      </a>
      <a class="btn btn-danger"
        href="javascript:void(0)" ng-click="vm.remove()">
        <span class="glyphicon glyphicon-remove"
          aria-hidden="true">
        </span>
      </a>
    </div>
  </div>
</div>

```

```

static/templates/expenses/expense.html
<div class="row" ng-cloak>
  <div ng-repeat="column in vm.columns">
    <div class="col-xs-12 col-sm-6 col-md-4 col-lg-3">
      <div ng-repeat="expense in column">
        <expense expense="expense"></expense>
      </div>
    </div>
  </div>

  <div ng-hide="vm.columns_&&vm.columns.length">
    <div class="col-sm-12 no-posts-here">
      <em>You haven't added any expenses.</em>
    </div>
  </div>
</div>

```

6.7 Modulele "layout" și "utils"

În "static/js/layout/controllers" avem două controllere: *IndexController* și *NavbarController*.

IndexController "ascultă" pe "\$scope" propagarea evenimentelor "expense.created", "expense.deleted" și "expenses.filter", iar când unul din aceste evenimente este declanșat, se face filtrarea cheltuielilor și variabila "vm.expenses" este actualizată.

```

static/js/layout/navbar.controller.js

(function () {
  'use strict';

  angular
    .module('moneyKeep.layout.controllers')
    .controller('NavbarController',
      NavbarController);

```

```

NavbarController.$inject = ['$rootScope',
    '$scope', 'Authentication', 'ngDialog'];

function NavbarController($rootScope, $scope,
    Authentication, ngDialog) {
    var vm = this;

    vm.logout = logout;
    vm.createExpense = createExpense;
    vm.print = print;
    vm.isAuthenticated = isAuthenticated;

    $scope.$watchGroup(['vm.startDate',
        'vm.endDate', 'vm.searchString'], function () {
        $rootScope.$broadcast('expenses.filter', {
            startDate: vm.startDate,
            endDate: vm.endDate,
            searchString: vm.searchString
        });
    });

    function logout() {
        Authentication.logout();
    }

    function createExpense() {
        ngDialog.open({
            template:
                '/static/templates/expenses/edit-expense.html',
            controller: 'NewExpenseController as vm'
        });
    }

    function print() {
        ngDialog.open({
            template: '/static/templates/Utils/print.html',
            controller: 'PrintController as vm'
        });
    }

    function isAuthenticated() {
        return Authentication.isAuthenticated();
    }
}
})();

```

NavbarController implementează funcțiile accesibile din bara de navigație: logout, adăugare cheltuială nouă și deschiderea dialogului cu selectarea săptămânii pentru printare.

În modulul "utils" se află controllerul PrintController și serviciile DateUtils

și Snackbar.

PrintController aduce de pe server cheltuielile dintr-o anumită săptămână, calculează suma și media acelor cheltuieli și le printează.

Șablonul corespunzător acestui controller definește un element "div" cu id-ul "print-area", ale cărui stiluri sunt definite în styles.css. Din CSS, div-ul este ascuns atunci când mediul este "screen" și afișat atunci când mediul este "print".

```
static/js/utils/controllers/print.controller.js

(function () {
    'use strict';

    angular
        .module('moneyKeep.utils.controllers')
        .controller('PrintController',
            PrintController);

    PrintController.$inject = ['$timeout', 'Expenses',
        'Snackbar'];

    function PrintController($timeout, Expenses, Snackbar) {
        var vm = this;

        vm.print = print;
        vm.expenses = [];
        vm.total = 0;
        vm.average = 0;

        function print() {
            Expenses.all(vm.week).then(function (response) {
                vm.expenses = response.data;
                if (vm.expenses.length > 0) {
                    var amounts = vm.expenses.map(
                        function (expense) {
                            return expense.amount;
                        });
                    vm.total = amounts.reduce(function (a, b) {
                        return a + b;
                    }, 0);
                    vm.average = vm.total / vm.expenses.length;
                    $timeout(function () {
                        window.print();
                    });
                } else {
                    Snackbar.show(
                        "No expenses for the selected week.")
                }
            })
        }
    }
})
```

```

    })();

    static/templates/utils/print.html
<form role="form" ng-submit="vm.print()">
  <div class="form-group">
    <label for="week">Week</label>
    <input type="week" id="week"
      ng-model="vm.week" required />
  </div>
  <div class="form-group">
    <label for="print" class="sr-only"></label>
    <button type="submit" id="print">Print</button>
  </div>
</form>

<div id="print-area">
  <table class="table">
    <thead>
      <tr>
        <td>Date</td>
        <td>Time</td>
        <td>Amount</td>
        <td>Description</td>
        <td>Comment</td>
      </tr>
    </thead>
    <tbody>
      <tr ng-repeat="expense_in_vm.expenses">
        <td>
          {{ expense.timestamp | date:'mediumDate' }}
        </td>
        <td>
          {{ expense.timestamp | date:'shortTime' }}
        </td>
        <td>
          {{ expense.amount | currency }}
        </td>
        <td>
          {{ expense.description }}
        </td>
        <td>
          {{ expense.comment }}
        </td>
      </tr>
    </tbody>
  </table>
  <strong>Total: </strong>{{ vm.total | currency }}<br>
  <strong>Average: </strong>{{ vm.average | currency }}
</div>

```


static/css/styles.css

```
@media screen {
  #print-area {
    display: none;
  }
}

@media print {
  body * {
    visibility: hidden;
  }
  #print-area, #print-area * {
    visibility: visible;
  }
}
```

Pe server, obiectele de tipul datetime sunt stocate într-o singură coloană, dar pe client input-urile de tipul date sunt separate de cele pentru timp. Din acest motiv se folosește serviciul DateUtils care face conversia între cele două metode de a stoca obiectele pentru datetime.

static/js/utils/services/dateutils.service.js

```
(function () {
  'use strict';

  angular
    .module('moneyKeep.utils.services')
    .factory('DateUtils', DateUtils);

  DateUtils.$inject = [];

  function DateUtils() {
    var DateUtils = {
      dateTimeFromDateAndTime:
        dateTimeFromDateAndTime,
      dateFromDateTime: dateFromDateTime,
      timeFromDateTime: timeFromDateTime
    };

    return DateUtils;

    function dateTimeFromDateAndTime(date, time) {
      return new Date(
        date.getFullYear(),
        date.getMonth(),
        date.getDate(),
        time.getHours(),
        time.getMinutes(),
        time.getSeconds()
      );
    }
  }
})
```

```

        time.getMilliseconds());
    }

    function dateFromDateTime(datetime) {
        return new Date(
            datetime.getFullYear(),
            datetime.getMonth(),
            datetime.getDate(),
            0, 0, 0, 0);
    }
    function timeFromDateTime(datetime) {
        return new Date(1970, 1, 1,
            datetime.getHours(),
            datetime.getMinutes(),
            datetime.getSeconds(),
            datetime.getMilliseconds());
    }
}

})();

```

Pentru afișarea de mesaje către utilizator cu privire la operațiile ce se efectuează asincron se folosește librăria *SnackbarJS*³. Această librărie oferă posibilitatea de a afișa mesaje ce arată bine din punct de vedere estetic în partea din stânga jos a aplicației. Aceste mesaje pot fi ascunse cu un simplu click (Figura 6.1). Pentru a fi integrată librăria într-o aplicație AngularJS, am scris serviciul *Snackbar*.

```

static/js/utils/services/snackbar.service.js

(function ($, _) {
    'use strict';

    angular
        .module('moneyKeep.utils.services')
        .factory('Snackbar', Snackbar);

    function Snackbar() {
        var Snackbar = {
            error: error,
            show: show
        };

        return Snackbar;

        function _snackbar(content, options) {
            options = _.extend({ timeout: 3000 }, options);
            options.content = content;

```

³<http://fezvrasta.github.io/snackbarjs/>

```

        $.snackbar(options);
    }

    function error(content, options) {
        _snackbar('Error: ' + content, options);
    }

    function show(content, options) {
        _snackbar(content, options);
    }
})($, _);

```

6.8 Posibilități de îmbunătățire ale aplicației

Datorită limitărilor de timp, există numeroare posibilități de îmbunătățire și extindere a aplicației. De exemplu:

- Adăugare login cu Facebook. Acesta este așteptat de la majoritatea aplicațiilor din ziua de azi, eliminând necesitatea utilizatorului de a-și crea un cont pentru fiecare aplicație folosită.
- Înlocuirea login-ului pe bază de cookie cu un alt mecanism, de exemplu OAuth. Acest lucru decuplează back-end-ul de front-end, astfel API-ul REST ar putea fi folosit și de aplicații mobile.
- Îmbunătățirea stilurilor CSS.
- Filtrarea pe server în loc de client.

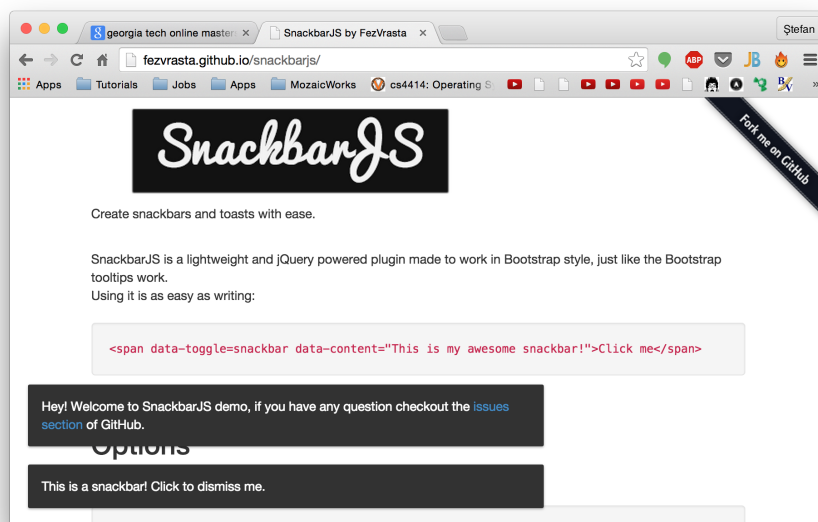


Figura 6.1: Exemple de mesaje SnackBar de pe pagina librăriei javascript

7

Concluzii

Aplicațiile web SPA sunt pe un trend ascendent și vor deveni tot mai comune în anii ce urmează. Necesitatea creării rapide de aplicații SPA a împins companii mari să pună la dispoziția dezvoltatorilor tehnologii care să le ușureze munca.

Scopul proiectului dezvoltat ca parte din această lucrare, nu este acela de a crea o aplicație plină de funcționalități și cu o utilitate în lumea reală. Scopul aplicației este de a evalua și a demonstra anumite tehnologii pentru crearea de aplicații web. Accentul nu este pus pe numărul de trăsături ale aplicației, ci pe implementarea corectă a acestora, dar și pe implementarea ușoară, într-un timp cât mai scurt. Implementarea rapidă de trăsături pentru o aplicație este un criteriu foarte important pe baza căruia companiile angajează programatori.

Denumirea SPA, duce oarecum în eroare, datorită faptului că implică faptul că aplicația are o singură pagină. O denumire mult mai descriptivă este cea de *Rich Web Application*. Această denumire sugerează, în mod corect, că scopul unei astfel de aplicații este defapt de a micșora diferențele existente între aplicațiile web și aplicațiile desktop.

Pentru crearea de SPA, pe partea de back-end, alternativele sunt multe, mature și stabile. Django este una dintre aceste alternative, fiind o tehnologie matură ce pune accent pe lizibilitate și dezvoltarea rapidă și cu ușurință de aplicații web moderne.

Un lucru foarte important pentru construirea back-end-ului este folosirea unui API REST. Un astfel de back-end este decuplat de front-end, permițând refolosirea aceluiași back-end, de exemplu, pentru aplicații mobile. Django, împreună cu Django REST Framework este o soluție matură pentru crearea unui back-end într-un timp scurt.

Pe partea de front-end, situația nu este la fel de clară. AngularJS este în momentul de față cea mai matură tehnologie, dar, după părerea multora, această tehnologie nu va mai fi aici peste câțiva ani. AngularJS 2.0 a fost anunțat deja, iar acesta nu va fi compatibil cu versiunea actuală, adică migrarea unei aplicații AngularJS scrisă cu versiunea 1.3 la versiunea 2.0 va fi foarte dificilă. Există și alte alternative promițătoare, cum ar fi ReactJS sau Meteor, dar aceste

tehnologii nu sunt la fel de mature ca AngularJS.

Este foarte posibil ca toate tehnologiile folosite în momentul de față pe front-end pentru aplicații SPA să nu mai fie folosite în 5 ani. Acest lucru nu face decât să sublinieze necesitatea oricărui dezvoltator de aplicații web de a fi la curent în permanență cu tehnologiile folosite și de a învăța permanent pe parcursul întregii cariere. De asemenea, este important ca un dezvoltator să nu cadă în capcana de a se concentra prea mult pe tehnologii, scăpând astfel din vedere faptul că ele sunt doar unelte pentru rezolvarea unor probleme.

În ciuda problemelor sale, AngularJS este folosit în multe aplicații puse în producție și reprezintă o alegere relativ sigură pentru dezvoltarea unei aplicații SPA.

Deși accentul acestei lucrări a fost pus destul de mult pe tehnologii, nu vreau să las impresia că cea mai importantă trăsătură a unui dezvoltator de aplicații este faptul că a lucrat cu anumite tehnologii. Tehnologiile folosite sunt cele care ne fac mai productivi ca programatori și uneori ele pot fi cauza unui proiect reușit sau eșuat. În același timp, este important pentru un dezvoltator să aibă înțeleagă principiile din spatele unei tehnologii mai mult decât tehnologia în sine.

De exemplu, consider că un programator care știe să folosească Django REST Framework, dar nu știe proprietățile unui API REST, diferențele dintre metodele HTTP și când trebuie folosită fiecare metodă HTTP este mai puțin valoros decât un dezvoltator care nu a folosit niciodată Django REST Framework, dar știe foarte bine toate proprietățile unui API REST. Un dezvoltator care are cunoștințele fundamentale bine puse la punct se poate ține la curent cu ultimele framework-uri mult mai ușor decât unul ce are lagune în acest domeniu.