UNIVERSITATEA DIN BUCUREȘTI

FACULATEA

DE

MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

APLICAȚIE WEB PENTRU PROMOVARE TURISTICĂ

Absolvent

Ștefan Geamănă

Coordonator științific

Lect. dr. Mihail Cherciu

București 2020

CUPRINS

1. INTRODUCERE

1.1 Scopul lucrării și al aplicației

1.1.1 Scopul lucrării

Scopul lucrării de față constă în prezentarea teoretică a instrumentelor soft utilizate și a conceptelor de programare folosite pentru dezvoltarea unei aplicații web de tip *responsive,* care să poate fi facil accesată de către utilizatori prin intermediul unui *browser web,* de pe orice tip de dispozitiv cu acces la internet.

Într-o primă parte a lucrării, vom face o trecere în revistă a tehnologiilor folosite în implementarea aplicației, urmând ca apoi se ne axăm pe descrierea modului de dezvoltare a acesteia și a produsului software obținut.

Lucrarea are ca scop prezentarea și dezvoltarea proiectului din punct de vedere academic. Pentru lansarea aplicației în "producție" (adică publicarea ei pe un domeniu real în *word wide web,* spre a fi folosită de către publicul țintă) fiind necesare unele ajustări pentru a asigura respectarea normelor legislative (inclusiv norme GDPR[1]) și a standardelor în materia securității cibernetice.

1.1.2 Scopul aplicației

Scopul aplicației constă în facilitarea promovării locațiilor de interes turistic mai puțin cunoscute, prin oferirea unei platforme digitale, destinata atât celor care doresc să își expună punctul de vedere cu privire la o zonă de interes turistic vizitată, cât și celor care sunt în căutarea unor noi destinații de vacanță.

Prin intermediul platformei vor putea fi atașate elemente grafice (fotografii) și text (descrierea și prețul orientativ pentru serviciile turistice), precum și coordonatele geografice ale locului descris. Acestea din urmă vor fi reprezentate pe hartă, contribuind la identificarea mai ușoară din punct de vedere geografic, precum și la formarea unei imagini de ansamblu cu privire la zona descrisă.

Platforma dezvoltată va putea fi folosită și ca o unealtă de promovare de către cei ce au interes în a face cunoscută o anumită locație. În acest mod se pot aduce beneficii pecuniare pentru zonele cu potențial turistic, prin focalizarea atenției publicului larg către respectivele locații de interes, scutindu-i pe cei interesați de finanțarea unor campanii de promovare costisitoare.

[1] General Data Protection Regulation

1.2 Structura lucrării

Lucrarea este structurata în patru părți:

În partea introductivă este expus scopul lucrării și al aplicației. În această secțiune se oferă câteva lămuriri cu privire la ce urmează a fi ulterior prezentat și, de asemenea, sunt explicate succint motivele ce au stat la baza alegerii acestei teme.

În următoarea parte ne ocupăm cu prezentarea teoretică a conceptelor și instrumentelor software utilizate. Începem prin a expune tehnologiile și conceptele folosite pentru dezvoltarea părții de client (*Frontend*), urmate apoi de tehnologiile și concepte folosite pentru dezvoltarea părții de server (Backend). Încheiem această parte prin expunerea tehnologiilor și conceptelor folosite pentru integrarea continuă, automatizarea și punerea în funcțiune a aplicației.

În penultima parte ne axăm pe descrierea aplicației și ne concentrăm, printre altele, asupra modului de proiectare și implementare a aplicației.

Încheiem prin secțiunea de concluzii în care sunt descrise cele mai importante aspecte ale utilizării și implementării aplicației, precum și posibilitățile de îmbunătățire și dezvoltare ulterioară.

2. PREZENTAREA TEORETICĂ A CONCEPTELOR ȘI INSTRUMENTELOR SOFTWARE UTILIZATE

Prezentul capitol are ca scop prezentarea instrumentelor software utilizate, atât cele care stau la baza dezvoltării propriu-zise a aplicației cât și cele folosite pentru integrarea continuă, automatizare și punerea în funcțiune a acesteia (*continuous integration, automation and deployment*).

Structura acestui capitol este dictată însăși de dihotomia ce stă la baza proiectării unei aplicații de tip *full stack* (aplicație client-server). Așadar vom prezenta inițial tehnologiile ce stau la baza dezvoltării părții de client (*Frontend*), urmând ca apoi să ne concentram atenția asupra limbajelor și *framework*-urilor folosite pentru server și baza de date (*Backend*).

În secțiunea finală a acestui capitol vom detalia folosirea instrumentelor ajutătoare pentru integrarea continuă, automatizare, punerea în funcțiune și exploatarea aplicației.

2.1 Tehnologii și concepte folosite pentru dezvoltarea părții de client (*Frontend)*

2.1.1 JavaScript

JavaScript este limbajul de programare al web-ului. Marea majoritate a site-urilor web utilizează JavaScript, iar toate browserele web moderne - desktop-uri, tablete și telefoane - includ un interpretor JavaScript, făcând din JavaScript limbajul de programare cel mai răspândit din istorie.

De-a lungul ultimului deceniu, NodeJS a făcut posibilă programarea JavaScript în afara browser-elor, iar succesul spectaculos al Node a făcut ca JavaScript să fie cel mai folosit limbaj de programare în rândul dezvoltatorilor de software.[1]

Motivele din spatele JavaScript

Când JavaScript a apărut pentru prima dată în 1995, scopul său principal a fost să se ocupe de o parte din validarea de datelor de intrare care a fusese lăsată anterior în sarcina limbajelor de server, cum ar fi *Perl*. Înainte de acea perioadă, era necesară o "călătorie" dus-întors către server pentru a determina dacă un câmp necesar a fost lăsat necompletat sau dacă o valoare introdusă era nevalidă. *Netscape Navigator* a căutat să

schimbe acest lucru odată cu introducerea JavaScript. Capacitatea de a gestiona unele validări de bază pentru client a fost o caracteristică nouă, interesantă, într-un moment în care utilizarea modemurilor telefonice era însemnată. Vitezele lente asociate cu acestea transformau pe atunci fiecare cerere către server într-un exercițiu de răbdare.[2]

De atunci, JavaScript a devenit o caracteristică importantă a fiecărui browser web de pe piață. Nu mai este folosit pentru o simplă validare a datelor, JavaScript interacționează acum cu aproape toate aspectele ferestrei browserului și conținutul acesteia. JavaScript este recunoscut ca un limbaj de programare complet, capabil de calcule și interacțiuni complexe, inclusiv închideri (*closures*), funcții anonime (lambda) și chiar metaprogramare. JavaScript a devenit o parte atât de importantă a web-ului încât chiar și browserele alternative, inclusiv cele de pe telefoanele mobile și cele destinate utilizatorilor cu dizabilități, îl acceptă. Chiar și Microsoft, cu propriul său limbaj de script, numit *VBScript*, a ajuns să includă propria sa implementare JavaScript în Internet Explorer de la cea mai veche versiune. Creșterea JavaScript de la un simplu validator de intrare la un limbaj de programare puternic nu a putut fi prevăzută. JavaScript este simultan un limbaj foarte simplu și foarte complicat, care necesită doar câteva minute pentru familiarizare, dar ani pentru a fi pe deplin stăpânit. Pentru a începe calătoria către utilizarea întregului potențial al JavaScript, este important să înțelegem natura, istoria și limitările sale.[2]

Scurtă istorie

În 1995, Brendan Eich, pe vremea aceea dezvoltator Netscape, a început să dezvolte un limbaj de scripting numit *Mocha* (redenumit ulterior *LiveScript*) pentru lansarea Netscape Navigator 2. Intenția era să-l folosească atât în browser cât și pe server, unde urma să fie denumit LiveWire.

Netscape a intrat într-o alianță de dezvoltare cu Sun Microsystems pentru a finaliza implementarea LiveScript la timp pentru lansare. Chiar înainte ca Netscape Navigator 2 să fie oferit oficial publicului, Netscape a schimbat numele LiveScript în JavaScript pentru a se folosi de popularitatea pe care limbajul de programare Java o avea în acea perioadă.

Deoarece JavaScript 1.0 a avut un mare succes, Netscape a lansat versiunea 1.1 în Netscape Navigator 3. Popularitatea web-ului la acea vreme atingea noi culmi, iar Netscape s-a poziționat ca fiind compania lider pe piață. În acest moment, Microsoft a decis să introducă mai multe resurse într-un browser concurent numit *Internet Explorer*. La scurt timp după lansarea Netscape Navigator 3, Microsoft a introdus

Internet Explorer 3 cu o implementare JavaScript numită JScript (diferența de nume avea ca scop evitarea eventualelor probleme de licență cu Netscape).

Implementarea JavaScript de către Microsoft a dus la existența a două versiuni diferite de JavaScript: JavaScript în Netscape Navigator și JScript în Internet Explorer. Spre deosebire de limbajul C și de multe alte limbaje de programare, JavaScript nu avea standarde care să reglementeze sintaxa sau caracteristicile sale, iar cele două versiuni diferite au evidențiat această problemă. Odată cu creșterea temerilor din industrie, sa decis ca limbajul să fie standardizat[2].

În 1997 JavaScript 1.1 a intrat în atenția Asociației Europeane a Producătorilor de Calculatoare (ECMA). Comitetul tehnic nr. 39 (TC39) a fost desemnat să „standardizeze sintaxa și semantica unui limbaj de *scripting* neutru de tip multiplatformă. Programatori TC39 de la Netscape, Sun, Microsoft, Borland, NOMBAS și alte companii interesate de viitorul unui asemenea limbaj, s-a întâlnit luni de zile pentru a pune la punct ECMA-262, un standard care definește noul limbaj denumit ECMAScript.

Anul următor, Organizația Internațională de Standardizare și Comisia Electrotehnică Internațională (ISO / IEC) a adoptat, de asemenea, ECMAScript ca standard (ISO / IEC-16262). De atunci, browserele au încercat, cu diferite grade de succes, să folosească ECMAScript ca bază pentru implementările lor JavaScript.[2]

Implementarea JavaScript

Deși JavaScript și ECMAScript sunt adesea utilizate în mod sinonim, JavaScript este mult mai mult decât ceea ce este definit în ECMA-262. Mai exact, o implementare completă a JavaScript este alcătuită din următoarele trei părți distincte (a se vedea figura 2-1): *[2]*

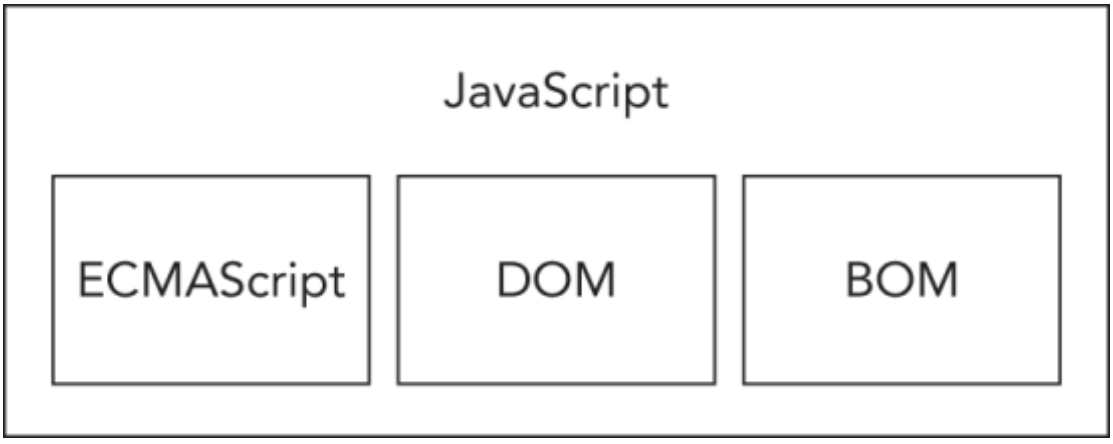


Figura 2-1

- Standardul *ECMAScript*
- *Modelul orientat-obiect pentru documente (DOM)*
- *Modelul orientat-obiect pentru browser (BOM)*

ECMAScript, limbajul definit în ECMA-262, nu este strict legat de browserele web. ECMA-262 definește o bază pe care pot fi construite limbaje de scripting mai puternice. Navigatoarele web sunt doar un mediu gazdă în care poate exista o implementare ECMAScript. Extensiile, cum ar fi *Modelul orientat-obiect pentru documente* (DOM), utilizează tipurile de bază și sintaxa ECMAScript pentru a oferi funcționalități suplimentare care sunt specifice mediului în care rulează. Un alt mediu gazdă este NodeJS, o platformă JavaScript folsită de obicei pentru dezvoltarea serverelor.

Se naște atunci întrebarea ce specifică mai exact ECMA-262 dacă nu face referire la browserele web? La un nivel foarte de bază, descrie următoarele părți ale limbajului: Sintaxa, tipuri (de variabile), expresii (*statament*-uri), cuvinte cheie, cuvinte rezervate, operatori, obiecte globale; ECMAScript este pur și simplu o descriere a unui limbaj care implementează toate fațetele descrise în specificații. JavaScript implementează ECMAScript, dar nu este singurul. Un alt exemplu poate fi *Adobe ActionScript*.[2]

Structura lexicală JavaScript

Structura lexicală a unui limbaj de programare este setul de reguli elementare care specifică modul în care sunt scrise programele în limbajul respectiv. Este sintaxa la cel mai de bază nivel.[1]

JavaScript este un limbaj *case sensitive*. Așadar cuvintele cheie, variabilele, numele funcțiilor și alți identificatori trebuie să fie întotdeauna tastate constant din punct de vedere al majusculelor.

JavaScript ignoră spațiile care apar între elementele din programe. Pe lângă caracterul de spațiu obișnuit (\u0020), JavaScript recunoaște de asemenea *tab*-ul, caracterele de control ASCII și diverse caractere spațiu Unicode, asimilându-le pe toate ca spațiu.

Comentarii

JavaScript acceptă două stiluri de comentarii. Orice text cuprins între „//” și sfârșitul unei linii este tratat ca un comentariu și este ignorat de JavaScript. Orice text între caractere „/ *” și „* /” este de asemenea notat ca un comentariu, acestea pot cuprinde mai multe linii, dar nu pot fi imbricate.

Identificatori și cuvinte rezervate

Un identificator este pur și simplu un nume. În JavaScript, identificatorii sunt folosiți pentru nume de constante, variabile, proprietăți, funcții și clase și pentru a furniza etichete pentru anumite bucle din codul JavaScript.

Un identificator JavaScript trebuie să înceapă cu o literă, un *underscore* (_) sau un semn dolar ($). Caracterele ulterioare pot fi litere, cifre, *underscore* sau semne dolar (cifrele nu sunt permise ca primul caracter, astfel încât JavaScript să poată distinge cu ușurință identificatorii de numere).

Cuvinte rezervate

Ca orice limbaj, JavaScript își rezervă anumite elemente de identificare pentru utilizarea limbii în sine. Aceste „cuvinte rezervate" nu pot fi utilizate ca identificatori obișnuiți.

Următoarele cuvinte fac parte din limbajul JavaScript. Multe dintre acestea (cum ar fi *if, while* și *for*) sunt cuvinte cheie rezervate care nu trebuie utilizate ca nume de constante, variabile, funcții sau clase (deși toate pot fi utilizate ca numele proprietăților din cadrul unui obiect). Altele (cum ar fi as *from, of, get*, și *set*) sunt utilizate în contexte limitate, fără ambiguitate sintactică și sunt perfect legale ca identificatori. Există și alte cuvinte cheie (cum ar fi *let*) care nu pot fi rezervate pe deplin pentru a menține compatibilitatea cu versiunile anterioare, astfel încât există reguli complexe care guvernează când pot fi utilizate ca identificatori și când nu pot. (*let* poate fi folosit ca nume de variabilă dacă este declarat cu *var* în afara unei clase, de exemplu, dar nu dacă este declarat în interiorul unei clase sau cu *const*.) Este de preferat să evitați folosirea oricăruia dintre aceste cuvinte ca identificatori, cu excepția *from, set*, și *target*, care se pot utiliza și sunt deja în uz comun.[1]

Unicode

Programele JavaScript sunt scrise folosind setul de caractere Unicode și putem utiliza orice caractere Unicode în șiruri și comentarii. Pentru portabilitate și ușurință de editare, este obișnuit să folosesim doar litere și cifre ASCII în identificatori. Dar aceasta este doar o convenție de programare, iar limbajul permite litere, cifre și caractere Unicode (dar nu emoji) în identificatori. Aceasta înseamnă că programatorii pot folosi simboluri matematice și cuvinte din limbi non-engleze pentru a declara constante și variabile. Spre exemplu `const π = 3.14;` Aici "π" este un caracter Unicode.

JavaScript este instrumentul folosit pentru a face paginile web interactive și este o parte esențială a aplicațiilor web. Alături de HTML și CSS, JavaScript este una dintre tehnologiile de bază ale World Wide Web.[1]

2.1.2 CSS (*Cascading Style Sheets*) și *Bootstrap*

Cascading Style Sheets (CSS) este un limbaj utilizat pentru descrierea prezentării (părții vizuale) a unui document scris într-un limbaj de marcare precum HTML. CSS este o tehnologie de temelie a World Wide Web, alături de HTML și JavaScript.

Scurta istorie a CSS

Povestea CSS începe în 1994 pe când Håkon Wium Lie lucra la CERN - leagănul Web - iar Web-ul începea să fie folosit ca platformă pentru publicarea electronică. O parte crucială a unei platforme de publicare lipsea totuși: nu exista nicio modalitate de a stiliza documente. De exemplu, nu exista o modalitate de a formata aspectul asemănător unui ziar într-o pagină Web. După ce a lucrat la prezentări personalizate de ziare la *MIT Media Laboratory*, Håkon a văzut necesitatea unui limbaj de stilizare pentru web.

Propunerea inițială CSS a fost prezentată la conferința Web din Chicago în noiembrie 1994, prezentarea inițială provocând multe discuții. Unii au considerat că CSS este prea simplu pentru sarcina pentru care a fost proiectat, susținând că pentru a stiliza documente, era nevoie de puterea unui limbaj de programare complet. CSS a mers exact în direcția opusă, scoțând la iveală un format simplu și declarativ.

Este dificil să precizăm cât de larg se folosește CSS astăzi la mai bine de 23 de ani de la lansarea oficială din 17 decembrie 1996, dar numărul de pagini HTML care nu utilizează CSS nu este probabil mai mare de câteva procente.

Avantaje si dezavantaje ale CSS

CSS este proiectat pentru a permite separarea părții grafice, incluzând aspectul paginii, culorile și fonturile, de conținutul unei aplicații web. Această separare poate îmbunătăți accesibilitatea conținutului, precum poate oferi mai multă flexibilitate și control în specificarea caracteristicilor de prezentare. De asemenea permite mai multor pagini web să partajeze formatarea prin specificarea codului CSS relevant într-un fișier *.css* separat, lucru care reduce complexitatea și duplicarea codului. Viteza de încărcare a paginilor este de asemenea îmbunătățită prin memorarea în *cache* a fișierului *.css* partajat de acestea.

În lumea dezvoltării de software, putem spune că CSS se diferențiază de alte tehnologii pe mai multe planuri. Nu este un limbaj de programare, strict vorbind, dar necesită gândire abstractă. Nu este doar un instrument de proiectare, ci necesită o anumită creativitate.

Numele "în cascadă" (*cascading*) provine din schema prioritară specificată pentru a determina ce regulă de stil se aplică dacă mai multe reguli se potrivesc cu un anumit element. Această schemă de prioritate în cascadă este previzibilă, însă pentru proiecte cu o complexitate însemnată, poate pune probleme, ajungându-se în unele cazuri (predominant atunci când partea de *frontend* este responsabilitatea unei întregi echipe) în special daca standardele nu sunt respectate cu strictețe, la introducerea în cod ale unor reguli CSS conflictuale.

De cele mai multe ori când trebuie să facem ceva în programarea convențională, ne putem da seama cu ușurință care anume este problema la a cărei rezolvare vrem să ajungem (de exemplu, „Cum găsesc elemente de tip x într-un tablou?"). Cu CSS, nu este întotdeauna ușor să cristalizăm situația ce trebuie rezolvată până ajungem la o singură întrebare. Chiar și când putem, răspunsul este adesea „depinde". Cel mai bun mod de a realiza ceva este deseori dat de constrângerile specifice ale aplicației noastre și de cât de precis dorim să gestionăm diferite cazuri marginale. Deși este util să cunoaștem câteva „trucuri" sau rețete utile pe care le putem urmări, stăpânirea CSS necesită o înțelegere în profunzime a principiilor care fac respectivele practici posibile[3]. CSS oferă o sintaxă declarativă înșelător de simplă, dar imediat ce începem să o aplicăm în proiecte a căror complexitate este cel puțin una medie, implementarea regulilor CSS intr-o manieră ordonata începe să devina din ce în ce mai dificilă.

Odată cu creșterea numărului dispozitivelor mobile începând cu anii 2000, cum ar fi telefoanele mobile și asistenții digitali personali (PDA), cererea de soluții specializate a luat amploare. Protocoalele nou definite pe atunci, cum ar fi WAP sau I-mode, puteau fi utilizate pentru a adapta paginile de internet la dispozitivele mobile însă nu fără o munca laborioasă întrucât pentru a fi realizate aceste adaptări, aproape toate paginile trebuiau rescrise.[4]

Mulțumită eforturilor depuse de către dezvoltatori (și chiar din partea unor companii) în încercarea de a rezolva (cel puțin parțial) neajunsurile descrise mai sus și pentru că website-urile de astăzi ar trebui să fie moderne, elegante și optimizate pentru dispozitive mobile, avem în prezent zeci de *framework*-uri CSS din care să alegem.

Fremework-uri CSS

Totul a început când *fremework*-urile CSS precum YUI (*Yahoo User Interface Library*) și *Blueprint* au devenit populare în jurul anilor 2006-07. Au adus cu ele multe resurse fundamentale precum resetarea CSS, fonturi, grile, efecte de animație, butoane etc. Dezvoltatorii au început să conștientizeze că aceste *fremework*-uri sunt utile pentru a face față multor sarcini repetitive și obositoare necesare dezvoltării unui site web iar că utilizarea lor ar putea îmbunătăți considerabil timpul de dezvoltare. Aceste *fremework*-uri CSS de bază au fost urmate de o generație de *fremework*-uri *frontend* „*full-edge*", cum ar fi Bootstrap, care a adăugat în implementarea sa și JavaScript. [5]

Framework-ul Bootstrap

Bootstrap este un produs *open source* conceput de Mark Otto și Jacob Thornto, în perioada în care aceștia erau angajați la Twitter. Necesitatea Bootstrap, așa cum a fost descrisa de Mark Otto în postarea de pe blogul de lansare a venit din nevoia de standardizare a numeroaselor instrumente *frontend* folosite de inginerii din întreaga companie.

Bootstrap 1.0.0 a fost lansat în 2011 doar cu componente CSS și HTML. Nu au fost incluse plugin-uri JavaScript până la Bootstrap 1.3.0, o versiune compatibilă cu IE7 și IE8. În 2012 a avut loc o altă actualizare importantă cu Bootstrap 2.0.0. A fost o rescriere completă a bibliotecii Bootstrap, devenind atunci un *framework responsive*. Componentele sale erau compatibile marea majoritate a dispozitivelor - telefoane mobile, tablete și desktop-uri, fiind atunci incluse și multe pachete noi CSS și JavaScript. După 15 actualizări majore, Bootstrap 3 în 2013 a fost o altă versiune semnificativă, devenind un cadru „Mobile First și mereu *responsive*". În versiunile anterioare ale *framework*-ului, crearea unui site web de tip *responsive* era opțională. În versiunea din 2013, au existat modificări în numele claselor și, de asemenea, în structura de directoare a proiectului. Bootstrap 3 nu este însă compatibil cu versiunile anterioare în sensul în care nu se poate migra direct la această versiune înlocuind fișierele principale CSS și JavaScript[3]. În prezent Bootstrap aflându-se la cea de-a patra versiune, este în continuare unul dintre cele mai folosite *framework*-uri de către dezvoltatori, în principal pentru că este compatibil cu toate browserele, scurtează timpul de dezvoltare, poate fi personalizat și este *open source.* Însă de departe cea mai folositoare caracteristică a *framework*-ului este sistemul sau de tip grila.

Bootstrap are unul dintre cele mai bune sisteme de tip grilă *responsive*, dezvoltat pentru dispozitivele mobile. Acesta ajută la scalarea unui singur site web de

la cel mai mic dispozitiv mobil până la afișaje de înaltă rezoluție, împărțind în mod logic ecranul în 12 coloane. Acest lucru permite programatorului să decidă cât de mult spațiu din suprafața display-ului ar trebui să ocupe fiecare element al designului.[6]

Sistemul implicit de grilă Bootstrap (a se vedea *figura 2-2*) folosește 12 coloane, creând un container cu o lățime de 940px fără ca funcțiile *responsive* să fie activate. Odată cu adăugarea fișierului CSS *responsive*, grila se adaptează pentru a fi de 724px sau 1170px lățime, în funcție de dimensiunile display-ului. Pentru display-uri de 767px, cum ar fi cele ale tabletelor și dispozitivelor mai mici, coloanele devin fluide și se stivuiesc vertical.

La lățimea implicită, fiecare coloană are 60 de pixeli lățime și compensează 20 de pixeli (*offset*) spre stânga. Un exemplu de aranjare al elementelor ținând cont de cele 12 coloane este prezentat în *figura 2-2*.[7]

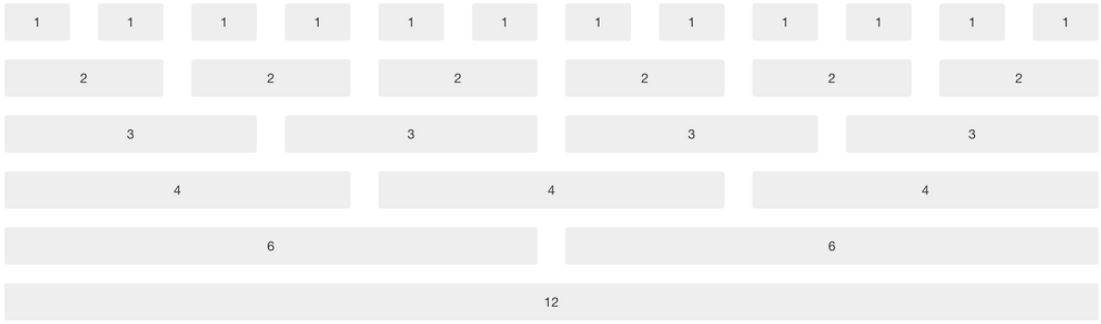


Figura 2-2.

Bootstrap a combinat componentele CSS și JavaScript utilizate în mod obișnuit, oferind împreună multe dintre cerințele de dezvoltare de bază, cum ar fi crearea de glisiere (*sliders*), crearea de efecte *pop-up* și meniurile derulante (*drop-down*). Bootstrap încapsulează multe componente utile care pot fi utilizate cu ușurință în proiectele site-ului și nu în ultimul rând, este folosit prin intermediul limbajului HTML standard. Cu Bootstrap, dezvoltatorii trebuie să se concentreze doar pe scrierea marcajelor HTML adecvate pe care *fremework*-ul le poate înțelege și reda în consecință. [3]

Bootstrap a devenit un instrument foarte popular în proiectele *frontend* de-a lungul anilor. Ușurința utilizării împreună cu compatibilitatea cu browserele, suportul pentru interfețele dispozitivelor mobile și capacitatea proiectării aplicațiilor web de tip *responsive*, îl fac un element esențial pentru orice proiect modern.

2.1.3 HTML5

HTML este acronimul pentru *HyperText Markup Language* și face două lucruri importante: descrie modul în care ar trebui să arate paginile web și definește semantica acestor pagini.[8]

Scurt istoric al HTML-ului și al World Wide Web

Până în 1990 accesarea informațiilor prin intermediul internetului era o problemă destul de tehnică. De fapt, era atât de dificil încât chiar și fizicienii care dețineau titlul de doctor au fost adesea frustrați când încercau să schimbe date. Un astfel de fizician, acum celebrul Tim Berners-Lee, a pregătit o modalitate de a accesa cu ușurință textul prin intermediul internetului, mai exact cu ajutorul link-urilor de tip hipertext. Aceasta nu a fost o idee nouă, însă simplitatea Hypertext Markup Language (HTML) a reușit să prospere, în timp ce proiectele de hipertext mai ambițioase au dispărut.

Hipertextul a însemnat inițial textul stocat sub formă electronică cu legături de referință încrucișată între pagini. Acum este un termen mai larg care se referă la aproape orice obiect (text, imagini, fișiere etc.) care poate fi legat de alte obiecte. Hypertext Markup Language este un limbaj pentru a descrie modul în care textul, grafica și fișierele care conțin alte informații sunt organizate și conectate.[9]

Până în 1993, doar aproximativ 100 de calculatoare din întreaga lume erau echipate pentru a servi pagini HTML. Aceste pagini interconectate au fost supranumite World Wide Web (WWW) și au dus la dezvoltarea mai multe programe de navigare web pentru a permite oamenilor să vizualizeze paginile web. Datorită popularității crescânde a Web-ului, câțiva programatori au dezvoltat browsere web care puteau vizualiza imagini grafice împreună cu textul. Din acel moment, dezvoltarea continuă a software-ului navigatoarelor web și standardizarea HTML-ului ne-au dus în lumea în care trăim astăzi, una în care peste un miliard de site-uri web servesc miliarde de fișiere text și multimedia.

2.2 Tehnologii și concepte folosite pentru dezvoltarea parții de server (*Backend)*

2.2.1 NodeJS

NodeJS este o platformă pentru scrierea aplicațiilor JavaScript în afara browserelor web. Mediul JavaScript cu care suntem familiarizați în browserele web nu se aplică în totalitate în cazul NodeJS deoarece în timp ce NodeJS execută același limbaj JavaScript pe care îl folosim în browsere, nu are unele caracteristici asociate acestora. De exemplu, nu există un DOM HTML încorporat în NodeJS.[10]

NodeJS este construit pe motorul JavaScript V8 al Google Chrome. Acesta este motorul JavaScript *open source* care rulează în Google Chrome și în alte browsere web bazate pe Chromium, inclusiv Brave, Opera și Vivaldi și este responsabil de compilarea JavaScript direct în codul mașină pe care computerul îl poate executa.

Popularitate

NodeJS devine rapid o platformă de dezvoltare populară și este adoptată de multe companii mari și mici. Unul dintre adoptatori este PayPal, care înlocuiește sistemul lor bazat pe Java cu unul scris în NodeJS. Pe lângă acesta mai putem menționa atât platforma de comerț online on-line Walmart, cât și LinkedIn sau eBay.

Conceptele ce stau la baza NodeJS

NodeJS este un mediu de dezvoltare JavaScript asincron, bazat pe evenimente, care oferă o bibliotecă standard puternică, dar concisă.[11]

În termeni simpli, când ne conectăm la un server tradițional, precum Apache, se va genera un nou *thread* (fir de execuție) pentru a gestiona cererea. Într-un limbaj precum PHP sau Ruby, orice operațiuni de intrare sau ieșire ulterioare (de exemplu, interacționarea cu o bază de date) blochează execuția codului până la finalizarea operațiunii. Adică, serverul trebuie să aștepte finalizarea căutării în baza de date înainte de a putea trece la procesarea rezultatului. Dacă apar noi solicitări în timp ce acest lucru se întâmplă, serverul va genera noi fire de execuție pentru a le trata. Acest lucru este potențial ineficient deoarece un număr mare de fire poate determina ca un sistem să devină mai lent și, în cel mai rău caz, ca site-ul să devina neoperațional. Cea mai comună modalitate de a sprijini mai multe conexiuni este să adăugarea mai multor servere.[12]

NodeJS, cu toate acestea, gestionează o singură sarcină simultan și folosește mai multe fire numai pentru sarcini care nu pot fi gestionate de firul principal.

Acest proces poate suna contraintuitiv, dar în majoritatea aplicațiilor care nu necesită multă putere de procesare, acest singur fir de execuție poate gestiona și executa rapid toate sarcinile.

Deși la suprafață NodeJS folosește un singur fir de execuție, motivul fiind că limbajul JavaScript este proiectat pentru a rula astfel, în fundal folosește mai multe fire pentru a executa cod asincron (așa cum ne putem da seama din *figura 2-3*)

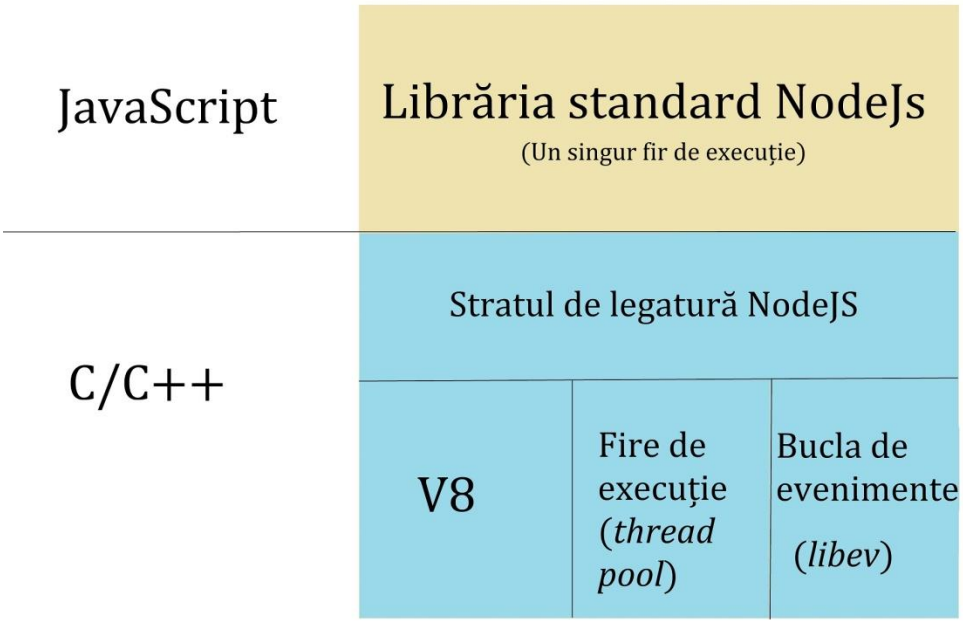


Figura 2-3

Este important să reținem că bucla de eveniment NodeJS se bazează pe un singur fir pentru a gestiona toate sarcinile sale, dar nu folosește în mod necesar acel *thread* doar pentru a rula fiecare sarcină până la finalizare. De fapt, NodeJS este proiectat pentru a transmite sarcini mai mari computerului gazdă, iar computerul poate crea noi *thread*-uri și procese pentru a opera aceste sarcini.

De exemplu, atunci când apare o nouă solicitare (un prim eveniment), serverul va începe procesarea acesteia. Dacă apoi întâlnește o operație de intrare / ieșire care în mod normal ar duce la blocarea firului de execuție, în loc să aștepte finalizarea acestui lucru, va înregistra funcția de *callback* înainte de a continua procesarea următorului eveniment. Când operațiunea de intrare / ieșire s-a încheiat (un alt tip de eveniment), serverul se va intoarce și va continua să lucreze la cererea inițială. Sub capotă, Node folosește biblioteca *libuv* (ce are la bază limbajul C) pentru a implementa acest comportament asincron (adică ne-blocant).[12]

O diagramă simplificată a buclei de evenimente este prezentată în *figura 2-4*. Pe măsură ce sarcinile sunt pregătite pentru a fi executate, acestea se introduc intr-o coadă pentru a fi procesate prin fazele specifice ale buclei de evenimente.

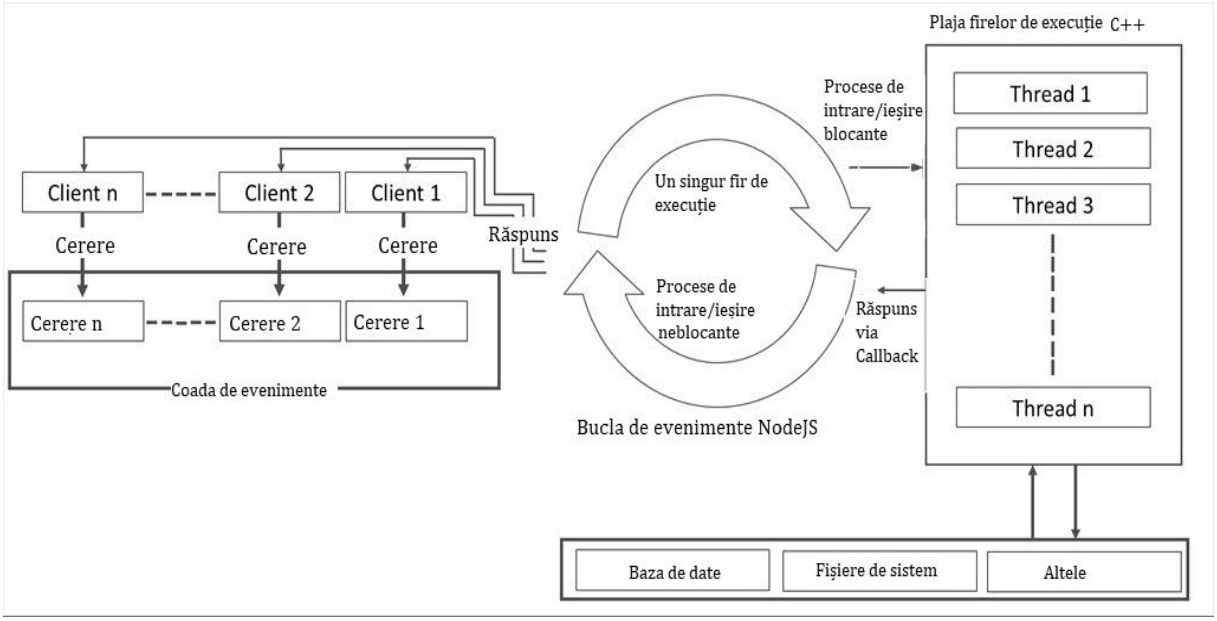


Figura 2-4

După cum îi spune și numele, bucla de evenimente NodeJS ciclează la infinit "ascultând" evenimentele JavaScript declanșate de server pentru a anunța o nouă sarcină sau finalizarea altei sarcini. Pe măsură ce numărul de sarcini crește, acestea se aliniază într-o coadă pentru a fi procesate incremental de către bucla de evenimente. Toate acestea, se întâmplă în fundal, programatorii trebuind doar să respecte convențiile asincrone, urmând ca NodeJS să planifice, să gestioneze și să rezolve sarcinile primite.[13]

Ryan Dahl, creatorul NodeJS, a oferit următorul exemplu. În prezentarea sa Cinco de NodeJS din mai 2010 Dahl ne-a întrebat ce se întâmplă atunci când executăm o linie de cod, cum ar fi aceasta:

```
result = query('SELECT * from db.table');
// operații asupra rezultatului
```

Desigur, programul se întrerupe în acest moment și așteaptă rezultatul sau eroarea în timp ce baza de date este interogată. Acesta este un exemplu de sarcină care blochează (pentru o perioada de timp) firul de execuție. În funcție de interogare, pauza poate fi destul de lungă. Această pauză inoportunează utilizatorul deoarece firul de execuție nu poate face nimic în timp ce așteaptă rezultatul. Deci putem concluziona că daca softul rulează pe o platformă cu un singur fir, întregul server nu ar putea răspunde altor cereri în această perioadă. Dacă în schimb aplicația rulează pe o platformă de server bazată pe mai multe *thread*-uri, este necesar un comutator context

pentru a satisface orice alte solicitări care sosesc (care va muta cererile respective către un alt *thread*). Cu cât este mai mare numărul de conexiuni la server, cu atât numărul de comutări între *thread*-uri este mai mare. Comutarea nu este gratuită, deoarece mai multe fire de execuție necesită mai multă memorie per thread și mai multă putere de procesare investită în administrarea firelor de execuție.[10]

În NodeJS, interogarea la care ne-am uitat anterior va arăta astfel:

```
query('SELECT * from db.table', function (err,
result) {
    if (err) throw err; // tratarea erorilor
    // operații asupra rezultatului
});
```

Programatorul furnizează o funcție care se va apela (de aici numele funcției de apelare - *callback*) atunci când rezultatul (sau eroarea) este disponibil. Interogarea bazei de date va dura aceeași perioadă de timp, dar în loc să blocheze firul de execuție, acesta revine la bucla de evenimente, fiind liber pentru a gestiona alte solicitări. NodeJS va declanșa în cele din urmă (după primirea răspunsului la interogarea bazei de date) un eveniment care face ca această funcție de *callback* să fie apelată cu indicarea rezultatului sau a erorii. O paradigmă similară este folosită în JavaScript pentru front-end, în funcțiile de gestionare a evenimentelor create de utilizator.

Performanță și utilizare

O parte din entuziasmul din jurul NodeJS se datorează debitului mare de cereri pe care le poate procesa (solicitările pe secundă pe care le poate servi). Comparația cu aplicații similare - de exemplu, Apache - arată că NodeJS câștigă la capitolul performanță. Un punct de referință este următorul server HTTP simplu care returnează un mesaj "Hello World":

```
var http = require('http');
http.createServer(function (req, res) {
res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('Hello World\n'); }).listen(8124,
"127.0.0.1");
console.log('Server running at
http://127.0.0.1:8124/');
```

Acesta este un model simplu de server web care poate fi construit cu NodeJS. Obiectul *http* încapsulează protocolul *HTTP*, iar metoda lui *http.createServer* creează un server web, ascultând pe portul specificat (8124). Fiecare solicitare (indiferent dacă este un GET sau POST, pe orice URL) către serverul web apelează funcția furnizată. În acest caz, indiferent de adresa URL, se returnează un text simplu ca răspuns: „*Hello World*"[10].

Inginerul Fabian Frank de la compania Yahoo! a publicat un studiu de caz privind performanta motorului de căutare al companiei folosind date de interogare din lumea reală. Implementarea motorului de căutare a fost făcută cu Apache / PHP intr-o primă variantă și alte două variante cu stive NodeJS. Aplicația este un panou de tip *pop-up* care prezintă sugestii de căutare în timp real atunci când utilizatorul introduce cuvinte în bara de căutare. În experiment s-a folosit o interogare HTTP bazată pe JSON. Versiunea NodeJS a putut gestiona de opt ori numărul de solicitări pe secundă cu aceeași latență de solicitare. Fabian Frank a declarat că ambele stive NodeJS au consumat resurse intr-un mod liniar până când utilizarea procesorului a atins 100%.

Concluzia este că NodeJS excelează la debitul de intrari / ieșiri bazat pe evenimente. Dacă un program NodeJS poate excela în cadrul aplicațiilor de calcul mai complexe depinde de ingeniozitatea programatorului de a evita anumite limitări în limbajul JavaScript.

2.2.2 ExpressJS

Express este un cadru (*framework*) de dezvoltare web minimalist, dar totodată flexibil și puternic pentru platforma NodeJS.

Express este minimalist deoarece nu este încărcat cu forte multe funcționalități. După instalare are doar caracteristicile de bază ale unui framework web și până și funcțiile acceptate nu sunt activate în mod implicit, așadar există opțiunea de a alege ce anume urmează să fie folosit în funcție de nevoile proiectului.[14]

Acesta este unul dintre cele mai atrăgătoare aspecte ale Express. De multe ori, dezvoltatorii de *framework-uri* uită că „mai puțin înseamnă mai mult”. Filozofia Express este de a oferi un strat minim între programator și server. Aceasta nu înseamnă că nu este robust sau că nu are suficiente funcții utile ci că se interpune în calea programatorului mai puțin, permițându-i exprimarea completă a ideilor sale. Este în același timp o unealtă adaptabilă prin care putem adăuga diferite părți ale funcționalității Express după cum ne este necesar, înlocuind tot ceea ce nu este necesar pentru proiect. Multe *framework-uri* încearcă să ofere totul, crescând dimensiunea proiectului, care ajunge de multe ori misterios și complex, înainte de a scrie chiar o singură linie de cod. Express ia abordarea opusă, permițându-ne să adăugăm ceea ce avem nevoie doar atunci când este necesar.[15]

Flexibilitatea în Express vine din utilizarea uneltelor intermediare și a modulelor Node. Uneltele Express intermediare și modulele Node sunt componente JavaScript conectabile, care fac aplicațiile modulare, flexibile și extensibile. Ceea ce

face Express este simplu în esență: acceptă cererile HTTP de la un client (care poate fi un browser, un dispozitiv mobil, un alt server, o aplicație de desktop, practic orice folosește protocolul HTTP) și returnează un răspuns HTTP. Acest model de bază descrie aproape tot ceea ce este conectat la internet, făcând Express extrem de flexibil în aplicațiile unde poate fi folosit.

Express este un *framework* puternic, deoarece ne oferă acces complet la API-ul NodeJS. Orice putem face cu Node, putem face și cu Express. Express poate fi folosit pentru a crea aplicații web de orice complexitate. Acesta ne oferă toate instrumentele necesare pentru a crea cele mai complexe aplicații, dar nu ne obligă să le utilizăm atunci când nu avem nevoie de ele.

Povestea din spatele Express.js

În cinci luni de la lansarea NodeJS, în iunie 2009, T.J. Holowaychuk, a lansat un proiect *open source* numit Express pentru a face dezvoltarea web mai facilă pentru proiectele ce foloseau NodeJS. Express a fost inspirat de Ruby Sinatra și construit peste API-ul NodeJS. Încă de la lansare a oferit unele funcționalități importante cum ar fi un sistem de rutare, asistență pentru sesiuni și *cookie*-uri, asistenți MIME, interfață RESTful, vizualizări bazate pe HAML ș.a. Cu toate acestea, Express v0.0.1 a fost foarte diferit de ceea ce este astăzi Express 3. Poate, singurul lucru comun între ele este numele „Express”. În iunie 2010 Sencha a început un proiect open source numit Connect, pentru a rezolva problema de modularitate și extensibilitate în API-ului NodeJS. Proiectul a fost inspirat de interfața serverului web Ruby Rack. Tim Caswell, angajat al lui Sencha, și T.J. Holowaychuk au fost încurajați să conducă proiectul. La fel ca Express, Connect a fost de asemenea construit peste API-ul NodeJS și a venit cu un sistem de *middleware,* care a permis conectarea unor programe mici reutilizabile pentru a gestiona funcționalități specifice HTTP. Conectarea acestor servicii a adăugat funcționalități necesare în aplicațiile web construite cu NodeJS. În plus, oricine își putea scrie propriul *middleware* pentru aplicațiile sale. Conectarea a îmbunătățit considerabil modularitatea și extensibilitatea API-ului NodeJS. Până acum, existau două framework-uri de dezvoltare web diferite pentru NodeJS: Express și Connect - unul era inspirat de Sinatra, iar celălalt de Rack. Acest lucru a provocat confuzie în comunitatea NodeJS, mai ales că Holowaychuk lucra la ambele proiecte. Mai târziu a devenit evident că Express și Connect sunt de fapt framework-uri complementare. Așadar, în iulie 2010, Holowaychuk a decis să rescrie arhitectura Express pentru a-l putea integra cu Connect, fuzionând efectiv

Connect cu Express pentru a crea o nouă încarnare a Express în v1.0.0. Odată cu Express v1.0.0, nu a mai existat nicio confuzie cu privire la ce framework de dezvoltare web să fie ales pentru proiectele NodeJS. Express a devenit Connect cu funcționalități suplimentare construite peste acesta din urmă. În prezent Express utilizează în continuare *middleware*-ul Connect și orice schimbare în Connect este reflectată invariabil și în Express. [14]

Aceasta este povestea modului în care Express a luat ființă și a modului în care Connect și Express au devenit un tot unitar.

Cum îmbunatățește Express funcționalitatea NodeJS

Când creăm o aplicație web (mai precis, un server web) în NodeJS, scriem o singură funcție JavaScript pentru întreaga aplicație. Această funcție ascultă cererile unui browser web ori cererile de la o aplicație mobilă sau orice alt client care comunică cu serverul. Când va primi o solicitare, această funcție va analiza cererea și va stabili cum să răspundă. De exemplu dacă vizităm pagina principală într-un browser web, această funcție va determina că dorim pagina de pornire și va trimite înapoi răspunsul HTML.[16]

Spre exemplu o simplă aplicație web care le indică utilizatorilor ora de pe server, va funcționa astfel: Dacă clientul solicită pagina principală, aplicația va returna o pagină HTML care arată ora. Dacă clientul solicită altceva, aplicația va returna o eroare HTTP 404 „Pagina nu există”. Dacă am construi aplicația folosind doar NodeJS fără Express, parcursul unei cereri către server ar arăta ca în *figura 2-5*.

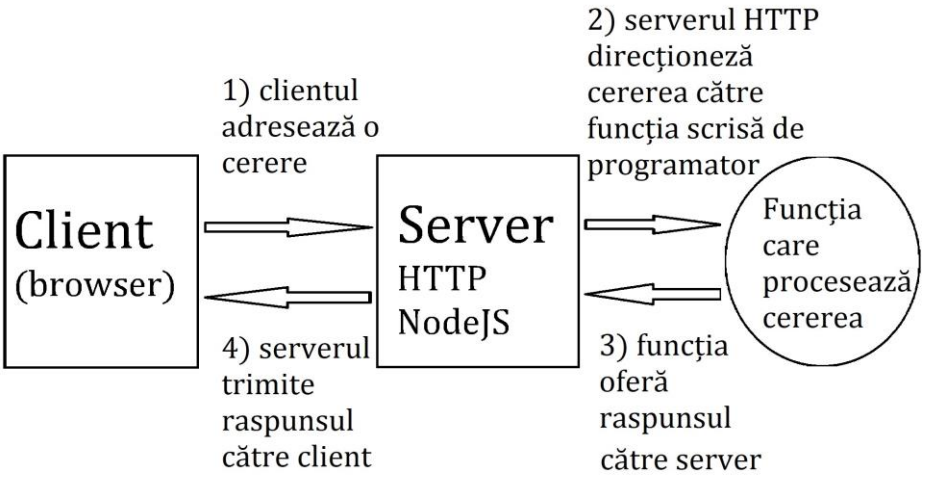


Figura 2-5

Funcția JavaScript care procesează solicitările browser-ului se numește *request handler* (funcție de gestionare a cererilor). Este o funcție JavaScript care preia cererea, o procesează și răspunde. Serverul HTTP al NodeJS gestionează conexiunea dintre

client și funcția JavaScript respectivă, astfel încât să nu fie nevoie ca programatorul să se ocupe de protocoale de rețea complicate. În cod, funcția de gestionare a cererilor este o funcție care are două argumente: un obiect care reprezintă cererea și un obiect care reprezintă răspunsul. În aplicația prezentată anterior, funcția de gestionare a solicitării poate verifica adresa URL solicitată de client. Dacă se solicită pagina principală, funcția de gestionare a cererilor ar trebui să răspundă cu ora curentă într-o pagină HTML. În caz contrar, ar trebui să răspundă cu o eroare de tip 404.

Fiecare aplicație NodeJS funcționează în acest fel: există o singură funcție de gestionare a cererilor care răspunde solicitărilor. Conceptual, este destul de simplu. Problema este că API-urile NodeJS pot deveni complexe. Spre exemplu dacă dorim să trimitem un singur fișier JPEG, va trebui să scriem aproximativ 45 de linii de cod. Serverul HTTP al NodeJS este puternic, dar lipsesc o mulțime de funcționalități pe care le-am putea folosi în programarea unei aplicații reale.

Express a luat naștere tocmai pentru a ușura scrierea aplicațiilor web cu NodeJS.

În linii mari, Express adaugă două funcționalități importante serverului HTTP dezvoltat în NodeJS:

Un prim beneficiu este reprezentat de faptul că aduce o serie de facilități utile pe serverul HTTP al NodeJS, eliminând o mare parte din complexitate. De exemplu, trimiterea unui singur fișier JPEG, așa cum am amintit în paragraful anterior este destul de complexă în NodeJS (mai ales dacă încercăm să facem transferul într-un mod performant); Express reduce această sarcină la o linie de cod.

Pe de altă parte Express ne permite să divizăm funcția monolitică de gestionare a cererii, prezentă intr-o aplicație NodeJS standard, în mai multe funcții mai mici care se ocupă fiecare de o anumită parte specifică. Acest lucru este mai ușor de întreținut în timp și mai modular.[16]

Spre deosebire de *figura 2-5 (*care ne arăta parcursul unei cereri către server dacă am construi aplicația folosind doar NodeJS fără Express*)*, *figura 2-6* ne arată parcursul unei cereri intr-o aplicație Express. Deși schița ar putea să pară mai complicata, în realitate, din perspectiva programatorului, este o abordare cu mult simplificata.

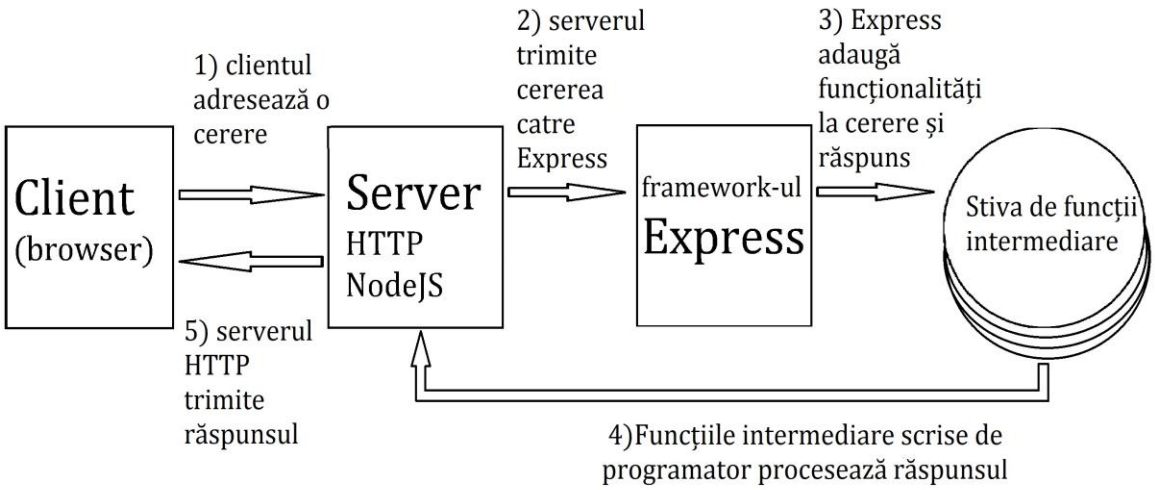


Figura 2-6

În Express, același cod crearea serverul web pe care l-am prezentat în secțiunea referitoare la NodeJS ar arăta în felul următor:

```
var express = require('express');
var app = express();
app.get('/', function(req, res) {res.send(200,
'Hello World');});
app.listen(8124, "127.0.0.1", function ()
{console.log('Server running at
http://127.0.0.1:8124/');
});
```

După cum am văzut, cu Express nu facem mult mai mult decât putem face în NodeJS-ul standard. Cu toate acestea, Express oferă o structură mai curată pentru cod și ajută programatorul prin modularizarea unor funcții monolitice.

2.2.3 Baza de date (Mongo DB)

Persistența datelor într-o aplicație web implică, de obicei, folosirea unei baze de date relațională, bazată pe un limbaj de interogare structurat (SQL). Această abordare oferă pentru multe aplicații un răspuns la modelarea datelor aplicației și a relațiilor dintre ele. Cu toate acestea, natura datelor din bazele de date SQL este reprezentată de scheme structurate cu tipuri clar definite, fiind în contrast cu obiectele flexibile, semi-structurate ale JavaScript. Această abordare poate face uneori dificil lucrul cu structuri de date între serverul web și baza de date. Relația datelor dintre aplicație și bazele de date SQL este gestionată prin serializare și deserializare între structurile bazei de date și modelul serverului web. Decalajul dintre ceea ce sunt datele atunci când sunt în baza de date și interpretarea lor prin transformare în serverul web, poate provoca schimbări nedorite ale contextului între diferitele limbaje specifice ale

domeniului (*domain specific languages - DSL*) pentru interogarea acestor date. Acest lucru poate duce uneori la greșeli și frustrări neașteptate atunci când menținem o aplicație web mare sau complexă. Cea mai comună soluție la aceste probleme este să ne bazăm pe o bibliotecă de mapare relațională a obiectelor (*object-relational mapping - ORM*) pentru a gestiona interacțiunile dintre aplicația web și baza de date. Cu toate acestea, niciun ORM nu poate eroda complet necesitatea de a înțelege intrinsec ceea ce este serializat și deserializat între baza de date și serverul web. Acesta este motivul pentru care mulți dezvoltatori au apelat la soluții de baze de date fără SQL (*NO-SQL*), precum MongoDB, pentru a lucra cu datele din aplicațiile web.

Capacitățile MongoDB de a lucra nativ cu structurile de obiecte JSON pentru datele aplicației permit dezvoltatorilor o relație naturală între obiectele de pe serverul web sau chiar din aplicația front-end și cele persistente în baza de date. Acest lucru, combinat cu multe alte caracteristici legate de arhitecturi alternative pentru scalarea performanței bazei de date și prevalența structurilor de date ale documentelor față de cele relaționale în multe aplicații, a făcut din MongoDB o alternativă foarte populară pentru echipele care doresc să dezvolte o aplicație web bazată pe JavaScript.[17]

MongoDB este puternic, dar ușor de folosit. În continuare, vom introduce câteva dintre conceptele de bază ale MongoDB:

1. Un document este unitatea de bază a datelor pentru MongoDB și este aproximativ echivalent cu un rând într-un sistem relațional de gestionare a bazelor de date (dar mult mai expresiv).
2. În mod similar, o colecție poate fi gândită ca un tabel cu o schemă dinamică.
3. O singură instanță a MongoDB poate găzdui mai multe baze de date independente, fiecare conținând propriile colecții.
4. Fiecare document are o cheie specială, "_id", care este unică într-o colecție.

Documente

În centrul MongoDB se află documentul: un set ordonat de chei cu valori asociate. Reprezentarea unui document variază în funcție de limbajul de programare, dar majoritatea limbajelor au o structură de date care este potrivită natural, cum ar fi o hartă, hash sau dicționar. În JavaScript, de exemplu, documentele sunt reprezentate ca obiecte: `{"întâmpinare" : "Bună ziua!"}`

Acest document simplu conține o singură cheie, „întâmpinare”, cu o valoare „Bună ziua!”. Majoritatea documentelor vor fi mai complexe decât acesta cuprinzând mai multe perechi cheie / valoare: `{"întâmpinare" : "Bună ziua!", "vizualizări" : 3}`

După cum putem vedea, valorile din documente pot fi de tipuri de date diferite (sau chiar un întreg document încorporat - *Embedded Documents*). În acest exemplu, valoarea pentru „întâmpinare” este un șir de caractere, în timp ce valoarea pentru „vizualizări” este un număr întreg.

Cheile dintr-un document sunt șiruri de caractere. Orice caracter UTF-8 este permis într-o cheie, cu câteva excepții notabile: Cheile nu trebuie să conțină caracterul “\0” (caracterul nul). Acest caracter este folosit pentru a semnifica sfârșitul unei chei. Caracterele „$” și „.” au unele proprietăți speciale și trebuie utilizate numai în anumite circumstanțe. În general, acestea ar trebui considerate rezervate. MongoDB este sensibil la tip și la diferența dintre literele mari și literele mici.

Un ultim lucru important de reținut este că documentele din MongoDB nu pot conține chei duplicate. De exemplu, în următorul exemplu nu avem un document legal definit: `{"întâmpinare" : " Bună ziua!", " întâmpinare " : " Bună seara!"}`[18]

Colecții și scheme dinamice

O colecție este un grup de documente. Dacă un document este analogul MongoDB al unui rând dintr-o bază de date relațională, atunci o colecție poate fi considerată analogul unui tabel.

Colecțiile au scheme dinamice. Aceasta înseamnă că documentele dintr-o singură colecție pot avea mai multe “forme”. De exemplu, ambele documente următoare ar putea fi stocate într-o singură colecție:

```
{"întâmpinare" : "Bună ziua!", "vizualizări" : 3}
{"ieșire": "noapte bună"}
```

Documentele de mai sus au chei diferite, numere diferite de chei și valori de diferite tipuri. Deoarece orice document poate fi pus în orice colecție, poate apărea întrebarea: „De ce avem nevoie de colecții separate?” Fără a fi nevoie de scheme separate pentru diferite tipuri de documente, de ce ar trebui să folosim mai multe colecții? Motivele sunt următoarele:

Păstrarea diferitelor tipuri de documente în aceeași colecție poate fi un coșmar pentru dezvoltatori și administratori. Dezvoltatorii trebuie să se asigure că fiecare interogare returnează doar documentele care aderă la o schemă particulară sau că

atunci când codul aplicației efectuează o interogare, poate gestiona documente de diferite forme.

Este mult mai rapid să obținem o listă de colecții decât să extragem o listă cu tipurile de documente dintr-o colecție. De exemplu, dacă am avea un câmp „tip” în fiecare document care a specificat dacă documentul este „mic”, „mediu” sau „mare”, ar fi mult mai lent să găsim acele trei valori într-o singură colecție decât să avem trei colecții separate și să interogăm colecția corectă.

Gruparea documentelor de același fel în aceeași colecție permite localizarea datelor. Obținerea mai multor postări de blog dintr-o colecție care conține numai postări va necesita mai puține căutări pe disc decât obținerea acelorași postări dintr-o colecție care conține postări și date despre autor.

Începem să impunem o anumită structură documentelor noastre atunci când creăm indexuri. (Acest lucru este valabil mai ales în cazul indexurilor unice.) Acești indici sunt definiți pe colecție. Punând doar documente de un singur tip în aceeași colecție, ne putem indexa colecțiile mai eficient.[18]

O colecție este identificată prin numele său. Nume de colecție poate fi orice șir UTF-8, cu câteva restricții: Șirul gol ("") nu este un nume de colecție valid. Numele de colecții nu pot conține caracterul “\0” (caracterul nul), deoarece aceasta delimitează sfârșitul un nume de colecție. Nu ar trebui să creăm colecții cu nume care încep cu *system.*, un prefix rezervat colecțiilor interne. De exemplu, colecția *system.users* conține utilizatorii bazei de date, iar colecția *system.namespaces* conține informații despre toate colecțiile bazei de date. De asemenea, colecțiile create de utilizatori nu ar trebui să conțină caracterul rezervat „$” în numele lor.

Baze de date

Pe lângă gruparea documentelor după colecție, MongoDB grupează colecțiile în baze de date.

O singură instanță a MongoDB poate găzdui mai multe baze de date, fiecare grupând zero sau mai multe colecții. O regulă bună este stocarea tuturor datelor pentru o singură aplicație în aceeași bază de date. Bazele de date separate sunt utile la stocarea datelor pentru mai multe aplicații sau utilizatori pe același server MongoDB.

Concatenând un nume de bază de date cu o colecție din baza de date respectivă, putem obține un nume de colecție complet, la care ne referim ca *namespace*. De exemplu, dacă utilizăm colecția *blog.posts* din baza de date *cms*,

namespace-ul acelei colecții ar fi *cms.blog.posts*. *Namespace*-urile sunt limitate la 120 de octeți deși, în practică, au o lungime mai mică de 100 de octeți.

Cheia specială, "_id"

Fiecare document păstrat într-o bază de date MongoDB trebuie să aibă o cheie "_id". Valoarea cheii "_id" poate fi de orice tip, dar implicit este de tipul *ObjectId*. Într-o singură colecție, fiecare document trebuie să aibă o valoare unică pentru „_id”, ceea ce asigură faptul că fiecare document dintr-o colecție poate fi identificat în mod unic. Adică, dacă am avea două colecții, fiecare ar putea avea un document în care valoarea „_id” era 123. Cu toate acestea, nicio colecție nu poate conține mai mult de un document cu un „_id” de 123.

ObjectId este tipul implicit pentru „_id”. Clasa ObjectId este concepută pentru a genera o valoare unică la nivel global pe diferite mașini. Natura distribuită a MongoDB este principalul motiv pentru care se folosește ObjectId spre deosebire de ceva mai tradițional, cum ar fi o cheie primară autoincrementantă: ar fi dificil să sincronizăm cheile primare autoincrementare pe mai multe servere. Deoarece MongoDB a fost conceput pentru a fi o bază de date distribuită, este important să putem genera identificatori unici într-un mediu “fragmentat”.

Id-urile utilizează 12 octeți de memorie, fiind reprezentate printr-un șir de 24 de cifre hexadecimale: 2 cifre pentru fiecare octet. Este important să reținem că, deși un ObjectId este adesea reprezentat ca un șir hexadecimal uriaș, șirul este de fapt de două ori mai mare decât datele stocate. Dacă creăm mai multe ObjectId-uri noi în succesiune rapidă, putem vedea că doar ultimele câteva cifre se schimbă de fiecare dată. În plus, câteva cifre din mijlocul ObjectId-lui se vor schimba doar dacă distanțăm crearea lor la câteva secunde. Acest lucru se datorează modului în care sunt create ObjectId-urile. Cei 12 octeți ai unui ObjectId sunt generați după cum urmează:

Primii patru octeți ai unui ObjectId reprezintă timpul în secunde de la 1970 până în prezent.

Următorii cinci octeți ai unui ObjectId sunt o valoare aleatorie. Ultimii trei octeți sunt un contor care începe cu o valoare aleatorie pentru a evita posibilele coliziuni pe diferite mașini.

Aceasta permite generarea de până la 256^3 (16.777.216) obiectId-uri unice pe proces într-o singură secundă. Dacă nu există nicio cheie „_id” la introducerea unui document, una va fi adăugată automat la documentul inserat.[18]

Pentru a rezuma, MongoDB este un sistem de gestionare a bazelor de date bazat pe documente, *open source*, proiectat pentru bazele de date și dezvoltarea aplicațiilor web moderne.

2.3 Tehnologii și concepte folosite pentru integrarea continuă, automatizarea și punerea în funcțiune a aplicației

Pentru dezvoltarea aplicației (scrierea codului) am folosit ca editor de text VisualStudio Code. Acesta a fost integrat cu contul de Github pentru a facilita controlul versiunilor. Fiecare modificare a codului, dupa ce este ajunge în *repository-ul Github*, declanșează, prin intermediul uni webhook, un *pipeline* de Jenkins care se ocupa cu oprirea instanței anterioare a aplicației și pornirea uneia noi, care să conțină ultimele modificări. Întregul ciclu, de la momentul în care modificările ajung în *repository* până când aplicația este din nou funcțională durează aproximativ un minut.

Pentru siguranță, baza de data a fost mutată în *cloud (MongoDB Atlas*), și atât serverul NodeJS al aplicației cât și serverul de Jenkins folosit au fost găzduite pe un computer Raspberry Pi care poate fi accesat din exterior prin intermediul unui DNS dinamic.

În continuare vom prezenta pe scurt tehnologiile amintite mai sus.

2.3.1 Sistemul de control al versiunilor (Git și Github)

Sistemul de control al versiunilor (VCS), cunoscut și sub denumirea de control de sursă sau SCM, este un software care ține evidența fiecărei versiuni a fiecărui fișier într-un proiect software împreună cu o marcă de timp pentru momentul creării respectivei versiuni și autorul modificărilor.

GitHub, așa cum îi sugerează numele, este construit având la bază Git. Git este un tip de sistem de control al versiunilor (exista și altele precum Subversion) gratuit și *open source*, ceea ce înseamnă că oricine îl poate folosi, și chiar îmbunătăți.

GitHub este o gazdă pentru *repository*-urile (depozitele) Git. Este util a fi folosit atât ca o copie de rezervă a codului, cât și ca o modalitate prin care alți programatori pot colabora la codul respectiv. Ca gazdă Git, GitHub oferă toate caracteristicile Git și în plus câteva servicii suplimentare utile.

Spre deosebire de multe sisteme de control al versiunilor, Git funcționează cu instantanee, nu cu diferențe. Aceasta înseamnă că nu urmărește diferența dintre două versiuni ale unui fișier, ci face o imagine a stării curente a proiectului. Din acest motiv Git este foarte rapid în comparație cu alte VCS distribuite. Spre deosebire de un sistem centralizat de control al versiunilor, Git este complet opusul. Nu este necesar să comunice cu un server central deoarece Git este un VCS distribuit, fiecare utilizator

având un depozit complet, cu propriul istoric și seturi de modificări. Astfel, totul se face local, cu excepția partajării patch-urilor sau seturilor de modificări. Când Git realizează un instantaneu, acesta efectuează o sumă de control (*hash*) asupra acestuia; deci, știe ce fișiere au fost schimbate prin compararea sumelor de verificare. Acesta este motivul pentru care Git poate urmări cu ușurință modificările între fișiere și directoare și verifică dacă fișierele au fost corupte. Caracteristica principală a sistemului Git este sistemul său cu „trei stări". Stările sunt directorul de lucru, zona de înregistrare și directorul git:

Directorul de lucru este doar instantaneul curent pe care lucrăm.

Zona de înregistrare este locul în care fișierele modificate sunt marcate în versiunea lor actuală, gata de a fi stocate în baza de date.

Directorul git. este baza de date în care este stocat istoricul.

Practic Git funcționează după cum urmează: modificăm fișierele, adăugăm fiecare fișier pe care dorim să-l includem în instantaneu în zona intermediară (*git add*), apoi facem instantaneul și îl adăugăm la baza de date (*git commit*). Pentru terminologie, numim „fișier" un fișier modificat adăugat în zona de intermediară și „angajat" (*commited*) un fișier adăugat în baza de date. Deci, un fișier trece de la „modificat" (*modified*) la „etapizat" (*staged*) la „angajat" (*commited*).[19]

Odată îndepliniți pașii descriși mai sus prin intermediul editorului de text (în cazul de față am folosit VisualStudio Code), modificările ajung în repository-ul privat găzduit de platforma Github, de unde, prin intermediul unui *webhook* (care funcționează similar cu un *trigger* intr-o bază de date relațională), serverul de Jenkins este informat că există noi modificări, și astfel declanșează un nou "*job*" pentru a îngloba modificările într-o noua instanță a aplicației.

2.3.2 Integrare și dezvoltare continuă (Jenkins)

Jenkins este un server de automatizare open source care este utilizat pe scară largă de multe organizații pentru a implementa practici DevOps populare, cum ar fi integrarea continuă (*Continuous Integration* - CI) și livrarea continuă (*Continuous Delivery* - CD). Jenkins este bogat în caracteristici și poate fi extins prin intermediul plugin-urilor.[20]

Prezentare generală a parcursului (*pipeline*) CI / CD

Ciclul de viață al dezvoltării aplicației este în mod tradițional un proces manual îndelungat. În plus, necesită o colaborare eficientă între echipele de dezvoltare și cele de operațiuni. *Pipeline*-ul CI / CD este o demonstrație a automatizării implicate în ciclul de viață al dezvoltării aplicațiilor care conține executarea automată a construcției, executarea automată a testelor, notificări către părțile interesate și implementarea în diferite medii de execuție în mod eficient. Un *pipeline* de implementare este o combinație de integrare continuă și livrare continuă, prin urmare înglobează o parte din practicile *DevOps*. [21]

Figura următoare (*Figura 2-7*) prezintă procesul *pipeline*-ului. În funcție de cultura organizației și de instrumentele disponibile, fluxul și instrumentele pot diferi:

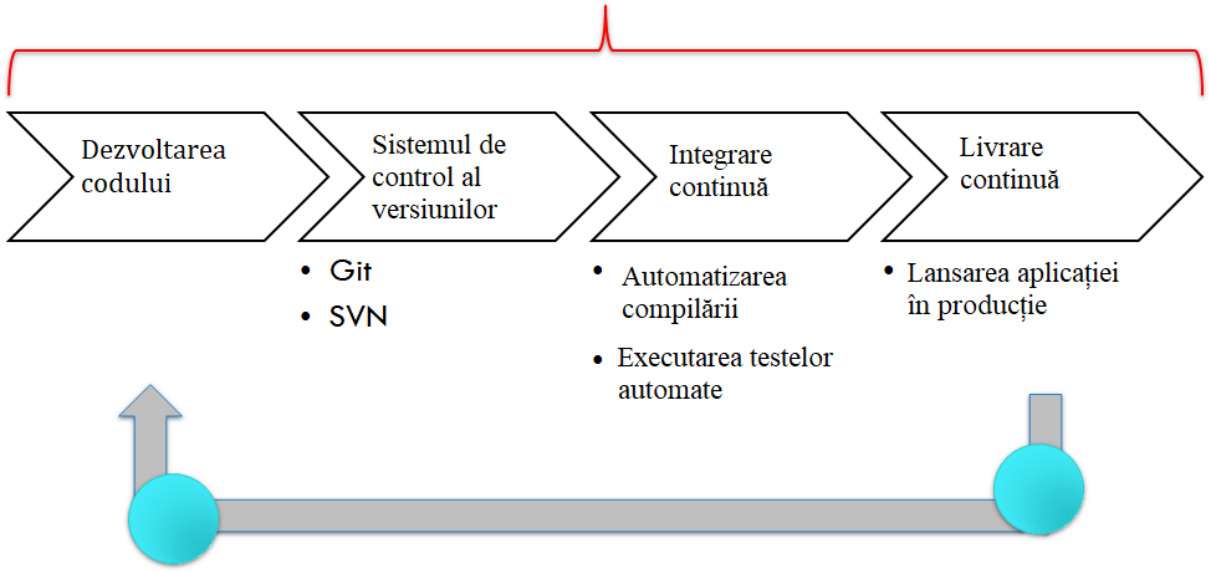


Figura 2-7

Membrii echipei de dezvoltare adaugă codul într-un sistem de control al versiunilor. Produsele de integrare continuă, cum ar fi Jenkins, sunt configurate pentru a interoga modificările din sistemul respectiv. Atunci când apar modificări, acestea sunt descărcate în spațiul de lucru local iar Jenkins declanșează un proces de compilare care este asistat de Ant sau Maven sau Gradle sau de orice script de compilare. Executarea automată a testelor, testarea unităților, analiza statică a codului, raportarea și notificarea proceselor de construcție reușite sau eșuate fac parte, de asemenea, din procesul de integrare continuă.

Odată ce versiunea are succes, poate fi implementată în diferite medii de execuție, cum ar fi testarea, pre-producția, producția și așa mai departe.[20]

2.2.3 Găzduirea bazei de date în *cloud* (MongoDB Atlas)

MongoDB Atlas, care a fost lansat în iunie 2016, este o platformă de baze de date ca serviciu (*Data Base as a Service* - DBaaS) creată, testată și susținută de aceeași echipă care a creat și continuă să dezvolte MongoDB.

După crearea bazei de date de care avem nevoie; Atlas va furniza instanța și ne va oferi următoarele:

Funcții de securitate automate - Instanțele de bază de date sunt implementate într-un cloud virtual privat unic (VPC) pentru a asigura izolarea rețelei. Alte caracteristici de securitate includ listarea IP-urilor care pot avea acces la bază, autentificare permanentă, criptare în repaus și criptare în tranzit, gestionare sofisticată a accesului bazată pe roluri și multe altele.

Replicare încorporată - Platforma ne oferă mai multe servere pentru disponibilitate permanentă, pentru a ne asigura că aplicația funcționează, chiar și atunci când serverul principal nu este activ.

Backup-uri și recuperare punctuală - sistemul MongoDB Atlas face eforturi puternice pentru a proteja împotriva corupției datelor, fie dacă aceasta se produce din cauza unor erori sau a unui atac cibernetic.

Monitorizare - Avem o mulțime de informații, organizate în mai multe moduri, pentru a ne ajuta să recunoaștem când este timpul să ducem lucrurile la nivelul următor (instanțe suplimentare pot fi furnizate atunci când numărul de utilizatori crește).

Atlas oferă extindere pe orizontală pentru baze de date utilizând o tehnică numită fragmentare (*sharding).* MongoDB distribuie date pe mai multe seturi de replici numite fragmente. Cu echilibrarea automată, MongoDB se asigură că datele sunt distribuite în mod egal între fragmente pe măsură ce volumele de date cresc sau dimensiunea clusterului se modifică. *Sharding*-ul permite implementărilor MongoDB să se extindă peste limitările unui singur server, cum ar fi blocaje în RAM sau operațiuni de citire/scriere pe disc, fără a adăuga complexitate aplicației.

Serviciul MongoDB Atlas poate avea la bază AWS, Azure sau Google Cloud. Prin folosirea serviciului putem implementa, opera și scala o bază de date MongoDB. Pentru proiectul de față vom folosi AWS (Amazon Web Services), urmând să găzduim baza de date cât mai aproape posibil de serverul NodeJS din București. Cea mai bună opțiune dintre cele disponibile fiind în Europa centrala – Frankfurt

Baza de date va rula într-o mașină virtuală care în configurația gratuită poate suporta până la 100 de baze de date, 500 de colecții și 500 de conexiuni simultane.

2.3.4 Găzduirea serverului NodeJS (Raspberry Pi 3B)

Raspberry Pi este un mic computer de dimensiunea unui card de credit. Ideea pentru Raspberry Pi a apărut după ce informaticianul Eben Upton a devenit îngrijorat de faptul că studenților le lipseau abilitățile necesare pentru a lucra cu hardware-ul computerului, concentrându-se mai mult pe partea de software. În timp ce lucra pentru Universitatea din Cambridge, Marea Britanie, în 2006, Eben împreună cu colegii Rob Mullins, Jack Lang și Alan Mycroft au dezvoltat un prototip construit manual, care era departe de computerele de dimensiunile cardului de credit din zilele noastre. Abia în februarie 2012 Eben și colab. au reușit să lanseze computerul accesibil, dar puternic, pe care îl cunoaștem astăzi.

Versiunea folosită în proiectul de față (Raspberry Pi 3B) a fost lansată în februarie 2016. Aceasta a oferit arhitectura ARMv8-A 64/32-bit ce rulează la o viteză de 1,2 GHz. Împreuna cu cei 1 GB de SDRAM avem suficientă putere pentru a rula un sistem de operare Linux, bazat pe Debian. Consumul de energie este unul redus de doar 300 mA (1,5 W) când este în repus, și 1,34 A (6,7 W) în sarcină maximă.

Întrucât conectarea la internet se face din spatele unui router, acesta redirecționează cererile HTTP primite către adresa IP din rețeaua interna dedicata computerului Raspberry PI. Pentru a facilita modul de accesare al aplicației, ținând cont că furnizorul de internet alocă un IP dinamic către router, am folosit un serviciu de DNS (Domain Name System) dinamic.

Am folosit această platformă pentru găzduirea serverelor de NodeJS și Jenkins pentru a putea face proiectul vizibil în afara mediului local de dezvoltare spre a putea fi accesat și evaluat din punct de vedere academic. Pentru lansarea aplicației în "producție", adică publicarea ei pe un domeniu real în *Word Wide Web*, spre a fi folosită de către publicul țintă, fiind necesare unele ajustări pentru a asigura respectarea normelor legislative (inclusiv norme GDPR) și a standardelor în materia securității cibernetice.

Proiectul poate fi accesat la adresa http://licenta-fmi.go.ro/

3. DESCRIEREA APLICAȚIEI

3.1 Analiza problemei abordate și utilitatea practică

În prezent, între promovarea turistică a locațiilor consacrate și promovarea locațiilor mai puțin cunoscute, dar care au potențial turistic însemnat, observăm o discrepanță însemnată. Ne aflăm, așadar, într-un ciclu vicios, situație ce la prima vedere pare fără rezolvare: turiștii suprapopulează zonele promovate, unde de cele mai multe ori cererea depășește oferta serviciilor HoReCa[2], iar locații cu cel puțin același potențial rămân necunoscute turiștilor, lucru ce duce și la lipsa de interes a învestitorilor pentru regiunile respective.

O zona cu potențial turistic se poate dezvolta în două moduri:

Un prim mod, mai rar întâlnit în țara noastră, constă într-o investiție însemnată intr-o zona cu potențial turistic pentru a pune la dispoziția viitorilor turiști toate facilitățile necesare, urmată apoi de campanii de promovare a zonei respective. Toate acestea care presupun de asemenea cheltuieli majore, posibil neamortizabile în viitorul apropiat.

Cel de-al doilea mod, întâlnit mai des în practică, constă într-o dezvoltare incrementală, ciclică, în care turiștii joacă rolul principal. Într-o primă fază, persoanele sunt atrase intr-o anumită zonă datorită amplasării geografice, a peisajului, a unor elemente benefice pentru sănătate (calitatea aerului, izvoare termale etc). Inițial sunt atrași amatorii de camping, urmând ca apoi localnicii să ofere servicii de cazare, iar într-un final investitorii încep să își facă apariția în regiunile respective. Putem vedea acest șablon de dezvoltare în localități precum Gura Portiței, Vama Veche, Sângeorz-Băi, Colibița s.a.

Motivația din spatele aplicației dezvoltate este ca aceasta să funcționeze precum un catalizator pentru ambele scenarii expuse mai sus.

Scopul aplicației constă în facilitarea promovării locațiilor de interes turistic mai puțin cunoscute, prin oferirea unei platforme digitale, atât pentru cei ce doresc să își expună punctul de vedere cu privire la o zonă de interes turistic vizitată, cât și pentru cei ce sunt în căutarea unor noi destinații de vacanță.

Prin intermediul platformei vor putea fi atașate elemente grafice (fotografii) și text, un preț orientativ pentru servicii, precum și coordonatele geografice ale locului descris. Acestea din urmă vor fi reprezentate pe hartă, contribuind la identificarea mai

[2] Acronimul pentru industria ospitalității: Hoteluri, Restaurante, Cafenele

ușoară din punct de vedere geografic, precum și la formarea unei imagini de ansamblu cu privire la zona descrisă.

Platforma dezvoltată va putea fi folosită și ca o unealtă de promovare de către cei ce au interes în a face cunoscută o anumită locație. În acest mod se pot aduce beneficii pecuniare pentru zonele cu potențial turistic, prin focalizarea atenției publicului larg sau prin atragerea investitorilor către respectivele locații de interes, scutindu-i pe aceștia din urmă de finanțarea unor campanii de promovare costisitoare.

3.2 Modul de proiectare și implementare a aplicației

3.2.1 Baza de date

Reamintim câteva dintre conceptele de bază ale MongoDB pe care le-am detaliat atunci când am discutat despre bazele de date NoSql si MongoDB:

1. Un document este unitatea de bază a datelor pentru MongoDB și este aproximativ echivalent cu un rând într-un sistem relațional de gestionare a bazelor de date (dar mult mai expresiv).
2. În mod similar, ne putem referi la o colecție ca fiind echivalentul unui tabel dintr-o baza de date relațională. Documentele sunt grupate în colecții.
3. O singură instanță a MongoDB poate găzdui mai multe baze de date independente, fiecare conținând propriile colecții.
4. Fiecare document are o cheie specială, "_id", care este unică într-o colecție.

Baza de date creată pentru aplicația de față este structurată în trei colecții:

O primă colecție conține toate documentele create atunci când o locație nouă este adăugată. Fiecărei locații îi corespunde un document.

În continuare oferim ca exemplu un document din colecția respectivă pentru a-l putea analiza:

```
_id: ObjectId("5f4bf8aa2cad60518cbae077")
name: "Marea de la munte!"
image: "https://images.unsplash.com/photo-1550561372-913b058c5f69?ixlib=rb-1.2..."
description: "lacul de la Colibita"
cost: 50
location: "Colibița, Romania"
lat: 47.1734134
lng: 24.9275811
comments: Array
   0: ObjectId("5f4bf9d52cad60518cbae078")
author: Object
   id: ObjectId("5de2df27dd358c2b5c282292")
   username: "Stefan"
createdAt: 2020-08-30T19:06:18.613+00:00
__v: 1
```

Documentul are ca prim câmp un `_id` de tipul `ObjectId`. Acesta este un câmp unic la nivel de colecție care ne ajută să identificăm respectivul document în interiorul colecției. Următoarele cinci câmpuri (*name, image, description, cost, location*) corespund cu câmpurile formularului de înregistrare a locului.

Câmpurile `lat` și `lng` sunt generate automat. Odată ce utilizatorul introduce un șir de caractere care reprezintă locul vizitat (în cazul de față – *Colibița, Romania)*, acest șir de caractere este procesat prin intermediul API-ului *Google Maps* și coordonatele sunt returnate pentru a fi înregistrate în câmpurile `lat` și `lng`. Ulterior vom folosi aceste coordonate pentru a reprezenta locația respectivă pe hartă

Următorul câmp, *comments*, este compus dintr-un *array* de `ObjectId`-uri, aici fiind stocate id-urile comentariilor pentru locația înregistrată, care fac parte dintr-o altă colecție ce cuprinde doar comentarii. Comentariu ce este stocat pe poziția 0 a *array*-ului este stocat în colecția sa așa cum se poate observa în captura de mai jos:


```
_id: ObjectId("5f4bf9d52cad60518cbae078")
text: "Campingul de la Colibița este înca deschis. Din păcate festivalul Coli..."
createdAt: 2020-08-30T19:11:17.866+00:00
__v: 0
author: Object
   id: ObjectId("5de2df27dd358c2b5c282292")
   username: "Stefan"
```


Putem observa că id-ul comentariului (*5f4bf9d52cad60518cbae078*) corespunde cu id-ul de pe poziția 0 a *array*-ului din cadrul documentului anterior, în acest fel făcându-se legătura dintre cele două documente.

Un alt aspect de interes este reprezentat de câmpul *author* care este prezent atât în documentul din colecția în care sunt stocate locații cât și în documentul din colecția în care sunt stocate comentariile. Acesta este un obiect, conținând atât id-ul documentului din cea de-a treia colecție (cea a utilizatorilor) cat si *username*-ul utilizatorului. În cazul de față id-urile coincid însă din pură întâmplare, mai exact pentru ca utilizatorul care a adăugat locația este același cu cel care a adăugat comentariul.

În ceea ce privește ultima colecție amintită, cea a utilizatorilor, un document este stocat în aceasta atunci când se creează un cont nou.

Documentul pentru un utilizator înregistrat are următoarele câmpuri:


```
_id: ObjectId("5de2df27dd358c2b5c282292")
salt: "fcf7df6fbe11d6d53ab4b723e47452acc6c4ef25a899b6884230e2da26b4cfbe"
hash: "6f5aa9349b6baa716ec10323f02250cde4c02d9ab44b87da1739ae378500a5cb41761d..."
username: "Stefan"
isAdmin: true
__v: 0
```

Primul câmp este id-ul unic al utilizatorului pe care l-am întâlnit deja ca element de legătură între documentele din colecțiile anterioare.

Urmează un câmp numit *salt,* acesta este o valoare aleatoare folosita pentru a cripta parola înainte ca ea să fie stocată în baza de date. Vom detalia mai mult acest aspect în secțiunea privind aspectele de securitate.

Următorul câmp este *hash*-ul (amprenta digitală) a parolei, generată cu ajutorul unei funcții unidirecționale.

Ultimul câmp are o valoare booleană și influențează drepturile pe care utilizatorul le are în aplicație.

Modul în care baza de date interacționează cu restul componentelor aplicației este reprezentat prin *figura 3-1:*

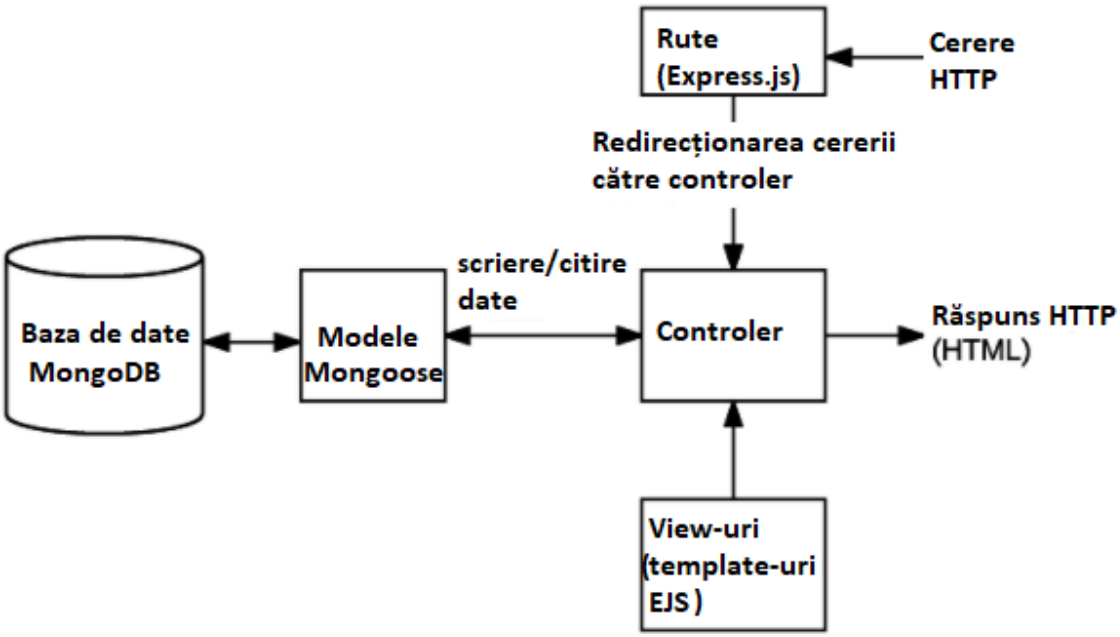


Figura 3-1(Arhitectura aplicației)

Modele Mongoose

Mongoose este o bibliotecă de modelare a datelor (*Object Data Modeling* - ODM) pentru MongoDB și NodeJS. Gestionează relațiile dintre date, oferă validarea schemelor și este utilizat pentru a face tranziția între obiecte în cod și reprezentarea acelor obiecte în baza de date MongoDB.

Modelele pot fi descrise ca și constructori compilați din definițiile schemei. O instanță a unui model se numește document. Modelele sunt responsabile pentru crearea și citirea documentelor din baza de date MongoDB.

Atunci când ne proiectăm modelele, este logic ca acestea să fie separate pentru fiecare tip de „obiect” (putem descrie un obiect ca fiind un grup de informații conexe).

În cazul nostru candidații evidenți pentru aceste modele sunt locațiile, comentariile și utilizatorii.

Adăugăm mai jos un exemplu de model care ne ajută la introducerea locațiilor în baza de date:

```
var campgroundSchema = new mongoose.Schema({
    name: String,
    image: String,
    description: String,
    cost: Number,
    location: String,
    lat: Number,
    lng: Number,
    createdAt: { type: Date, default: Date.now },
    author: {
        id: {
            type: mongoose.Schema.Types.ObjectId,
            ref: "User"
        },
        username: String
    },
    comments: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: "Comment"
        }
    ]
}, {
    usePushEach: true
});

module.exports = mongoose.model("Campground", campgroundSchema);
```

3.2.2 Arhitectura aplicației

Arhitectura unei aplicații descrie tiparele și tehnicile utilizate pentru proiectarea și construirea unei aplicații. Arhitectura ne oferă o privire de ansamblu și cele mai bune practici de urmat atunci când dezvoltăm o aplicație, astfel încât să ajungem la o un produs finit bine structurat și ușor de menținut.

Arhitectura aplicației, așa cum este detaliată în *figura 3-1*, presupune folosirea mai multor tehnologii și concepte a căror utilitate practică o vom prezenta succint în paragrafele ce urmează.

Întrucât am prezentat deja conceptele referitoare la baza de date (MongoDB și Mongoose), continuam cu detalii referitoare la controler, *view*-uri și rute.

Controller

În linii mari, ne putem referii la un *controller* ca la un set de funcții care separă codul menit în a direcționa cererile, de codul care procesează respectivele cereri.

Setul de funcții ce reprezintă *controllerul* preiau cererile de la modele, creează o pagina cu datele cerute și o redirecționează către client pentru a fi vizualizată.

View-uri (EJS)

EJS sau *Embedded Javascript Templating* este un motor de *templating* folosit de Node.js. "Motorul de șabloane" ajută la crearea unui șablon HTML cu minimum de cod. De asemenea, poate injecta date în șablonul HTML din partea clientului și poate produce HTML-ul final. EJS este un limbaj simplu de șablonare care este utilizat pentru a genera markup HTML folosind JavaScript. Și reversul este valabil întrucât poate ajuta și la încorporarea JavaScript în paginile HTML. Un model de *view* folosit la adăugarea locațiilor în baza de date este redat mai jos:

```
<% include ../partials/header %>
    <div class="row">
        <h1 style="text-align: center">Create a New Campground</h1>
        <div style="width: 30%; margin: 25px auto;">
            <form action="/campgrounds" method="POST">
                <div class="form-group">
                    <label for="name">Name</label>
                    <input class="form-control" type="text" name="name" id="name"
                    placeholder="Yosemite">
                </div>
                <div class="form-group">
                    <label for="image">Image Url</label>
                    <input class="form-control" type="text" name="image" id="image"
                    pattern=".*.(gif|jpe?g|tiff?|png|webp|bmp)" title="Only images  allowed!"
                    placeholder="link.gif, jpeg, png, webp, bmp" required>
                </div>
                <div class="form-group">
                    <label for="cost">Cost</label>
                    <input class="form-control" type="number" name="cost" id="cost"
                    placeholder="9.99" step="0.01" min="0" required>
                </div>
                <div class="form-group">
                    <label for="location">Location</label>
                    <input class="form-control" type="text" name="location" id="location"
                    placeholder="Yosemite National Park, CA">
                </div>
                <div class="form-group">
                    <label for="description">Description</label>
                    <input class="form-control" type="text" name="description" id="description"
                    placeholder="California's Sierra Nevada mountains.">
                </div>
                <div class="form-group">
                    <button class="btn btn-lg btn-primary btn-block">Submit!</button>
                </div>
            </form>
            <a href="/campgrounds">Go Back</a>
        </div>
    </div>
<% include ../partials/footer %>
```

Șablonul prezentat mai sus conține simple elemente HTLM precum *div, label, input, button.* De asemenea, atributele HTML folosite nu necesită detalieri suplimentare cu excepția atributului `pattern` pe care îl vom explica pe scurt în paragrafele următoare.

Atributul `pattern` al elementului `input` ne permite să adăugăm validarea datelor de bază fără a recurge la JavaScript. Funcționează prin verificarea valorii de intrare cu o expresie regulată. O expresie regulată este un șir formalizat de caractere care definesc un model. De exemplu `[a-zA-Z0-9] +` este un model care se potrivește cu un șir de orice lungime, atâta timp cât șirul conține doar litere mici (az), litere mari (AZ) si cifre (0-9).

Dacă valoarea de intrare nu este șirul gol și respectiva valoarea nu se potrivește cu întreaga expresie regulată, există un *modelMismatch* (o nepotrivire de model).

În exemplul prezentat se verifică dacă linkul introdus are ca terminație una dintre cele mai des utilizate extensii ale unui fișier imagine cum ar fi `gif, jpeg` etc.

Motivele din spatele acestei validări le vom prezenta în secțiunea privind aspectele de securitate.

Rutare

Rutarea se referă la modul în care punctele finale ale unei aplicații (URI-uri) răspund la solicitările clientului.

Rutarea se face folosind metode ale obiectului aplicației Express care corespund metodelor HTTP; de exemplu, `app.get()` pentru a gestiona solicitările GET și `app.post()` pentru a gestiona solicitările POST.

Aceste metode de rutare specifică o funcție de apelare inversă – *callback,* (uneori numită „*handler*”) care este apelată atunci când aplicația primește o solicitare către ruta specificată (punct final) cu metoda HTTP respectivă. Cu alte cuvinte, aplicația „ascultă” cererile care se potrivesc cu traseul (rutele) și metoda (metodele) specificate, iar atunci când detectează o potrivire, apelează funcția specificată.

Spre exemplu o rută simplă care răspunde cu mesajul “salut” atunci când o cerere “GET” este făcută către pagina principală arată așa:

```
app.get('/', function (req, res) {
  res.send('salut')
})
```

În aplicația dezvoltată am împărțit rutele în trei fișiere pentru a avea o vedere de ansamblu și a facilita mentenanță.

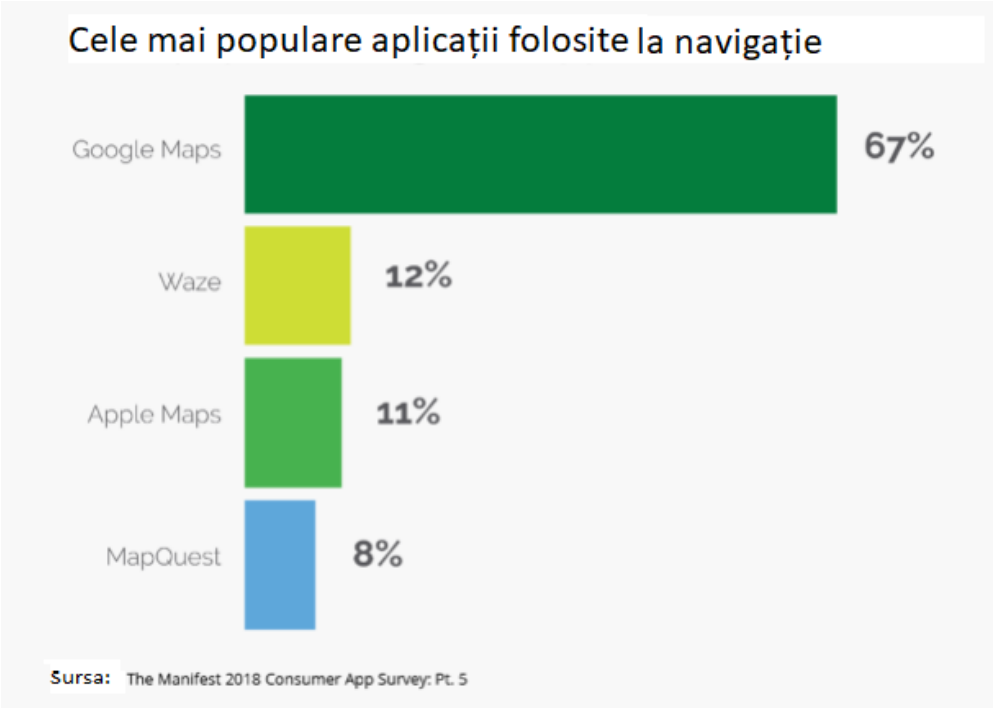
Într-un prim fișier avem rutele folosite pentru operațiile cu locații (afișare, creare, editare si ștergere). Am decuplat astfel rutele pentru comentarii, care se afla într-un fișier separat, precum și cele care se ocupă de crearea contului de utilizator și autentificare, care sunt de asemenea grupate într-un alt fișier.

3.2.3 Harta Google

Întrucât scopul aplicației noastre este de a promova zonele turistice, este necesar să alegem o soluție optimă prin care să putem facilita localizarea acestora. Inițial am luat în calcul posibilitatea adăugării unui câmp obligatoriu cu coordonatele geografice atunci când o noua locație este introdusă în baza de date de către utilizator. Această abordare, deși mai simplu de implementat, poate descuraja persoanele mai puțin tehnice și nu oferă o experiență fluidă a utilizării.

Pentru aceste motive, ținând cont și de faptul că 90% din informațiile transmise creierului uman sunt vizuale, și că imaginile sunt procesate de 60.000 de ori mai rapid decât textul, am ales să implementăm o soluție prin care locațiile să fie afișate pe o hartă, folosindu-ne doar de ceea ce utilizatorul introduce în câmpul dedicat denumirii locației.

Analizând serviciile care oferă posibilitatea implementării unei hărți precum și preferința utilizatorilor pentru astfel de servicii, am decis să folosim platforma Google pentru această sarcina. Așa cum reiese si din *figura 3-2* Google este lider incontestabil în topul preferințelor utilizatorilor:



Cât de familiari sunt utilizatorii cu diferite elemente prezente în interfață, este un punct definitoriu și direct proporțional cu ușurința în utilizare a aplicației. De asemenea, familiaritatea joacă un rol major în experiența utilizatorului. De fapt, majoritatea interfețelor pe care le folosim zilnic ne sunt atât de familiare încât le folosim fără să depunem efort cognitiv.

Cu peste un miliard de utilizatori Google Maps în întreaga lume, putem fi sigur că majoritatea utilizatorilor sunt deja cunoscători ai interfeței Maps. Chiar și utilizatorii care preferă un serviciu de hartă diferit de cel al Google vor recunoaște instantaneu aspectul și vor ști instinctiv să îl folosească.

Deci, indiferent dacă utilizatorii preferă serviciul de hartă Google sau nu, construirea soluției noastre pe tehnologia Google va oferi aplicației cel mai înalt grad de familiaritate posibil.

Pentru implementare am folosit pachetul *Geocoder*. Prin intermediul acestuia accesăm API-ul *Google Maps* pentru a primi coordonatele geografice ale locației introduse.

Mai jos putem vedea un exemplu de implementare folosit pentru adăugarea unei noi locații pe hartă.

```
//Geocoder
geocoder.geocode(req.body.location, function (err, data) {
  var lat = data.results[0].geometry.location.lat;
  var lng = data.results[0].geometry.location.lng;
  var location = data.results[0].formatted_address;
  var newCampground = {name: name, image: image, description: desc,
    cost: cost, author:author, location: location, lat: lat, lng: lng};
  // crearea unei locații
  Campground.create(newCampground, function(err, newlyCreated){
      if(err){
          console.log(err);
      } else {
          //redirectarea către pagina principală
          console.log(newlyCreated);
          res.redirect("/campgrounds");
      }
  });
},{ key: 'AIzaSyAArnpVtHbTADJZiS5oPdXdEhpF[redacted]8'});//cheia google API
});
```

Putem observa că funcția *geocode* primește trei parametrii:

Primul parametru *req.body.location* reprezintă locația pe care utilizatorul o introduce în formular.

Cel de-al doilea parametru este o funcție de *callback*. Odată ce primim informația despre locație prin intermediul parametrului anterior, ea este transmită către această funcție în parametrul *data*, unde urmează să fie procesată. Mai exact se va accesa api-ul Google și vor fi returnate coordonatele geografice ale locației introduse, pe care le vom salva în variabilele *lat* și *lng*. În continuare acestea vor fi folosite pentru a afișa harta, care va fi centrată pe zona respectivă. Utilizatorul poate introduce în câmpul de locație orice șir de caractere pentru a descrie cât mai bine zona (nume de țări, orașe, străzi, etc.)

Cel de-al 3-lea parametru este reprezentat de cheia api-ului Google care este obligatorie pentru a putea accesa serviciile Google Maps. Din motive de securitate cheia nu este afișată în totalitatea ei în captura prezentată mai sus.

3.2.4 Aspecte de securitate

Salt și *hash* pentru salvarea parolelor

Așa cum am observat atunci când am exemplificat modul în care datele sunt stocate în bază, documentul ce reprezintă un utilizator conține două câmpuri folosite la stocarea și verificarea parolei. Mai exact câmpurile *salt* și *hash*:

```
_id: ObjectId("5de2df27dd358c2b5c282292")
salt: "fcf7df6fbe11d6d53ab4b723e47452acc6c4ef25a899b6884230e2da26b4cfbe"
hash: "6f5aa9349b6baa716ec10323f02250cde4c02d9ab44b87da1739ae378500a5cb41761d..."
username: "Stefan"
isAdmin: true
__v: 0
```

În criptografie, un *salt* (sare) reprezintă date aleatorii care sunt folosite ca intrare suplimentară la o funcție unidirecțională care calculează *hash*-ul unor date sau a unei parole.

Salt-urile sunt folosite pentru a proteja parolele în mediul de stocare. Din punct de vedere istoric, parolele au fost stocate în clar pentru o bună perioadă de timp dar de-a lungul vremii au fost dezvoltate măsuri de protecție suplimentare pentru a proteja parola unui utilizator împotriva citirii din sistem. O „sare” este una dintre aceste metode.

O nouă “sare” este generată aleatoriu pentru fiecare parolă. Într-o setare tipică, sarea și parola sunt concatenate și procesate cu o funcție *hash* criptografică, iar valoarea hash-ului de ieșire (dar nu parola originală) este stocată împreună cu sarea într-o bază de date. Hashing-ul permite autentificarea ulterioară, fără a păstra și, prin

urmare, fără a risca expunerea parolei în clar în cazul în care stocarea datelor de autentificare este compromisă.

"Sărurile" apără și împotriva unui atac de hash pre-calculat. Deoarece sărurile nu trebuie să fie memorate de oameni, acestea pot face ca dimensiunea parolei ce urmează a fi hash-uita să fie prohibitiv de mare, fără a pune o povară asupra utilizatorilor. Deoarece sărurile sunt diferite în fiecare caz, ele protejează, de asemenea și parolele "standard" adică cele utilizate foarte frecvent, sau acei utilizatori care folosesc aceeași parolă pe mai multe site-uri.

Sărurile criptografice sunt utilizate pe scară largă în multe sisteme informatice moderne, de la acreditările de sistem Unix la securitatea Internetului.

Validarea datelor

Validarea datelor este procesul prin care ne asigurăm că datele au fost supuse curățării pentru a fi atât corecte, cât și utile. Validarea folosește rutine, deseori numite „reguli de validare", „constrângeri de validare" sau „rutine de verificare", prin care se asigură corectitudinea, semnificația și securitatea datelor care sunt introduse în sistem. Regulile pot fi puse în aplicare prin facilitățile automate ale unui dicționar de date sau prin includerea logicii explicite de validare.

Validarea în partea clientului este o verificare inițială și o caracteristică importantă pentru o bună experiență a utilizatorului; prin captarea datelor nevalide în client, utilizatorul le poate remedia imediat. Dacă așteptăm ca datele să ajungă la server și apoi se dovedește că respectivul câmp este respins, apare o întârziere vizibilă până utilizatorul primește informația că este necesar să modifice datele.

În exemplul dat atunci când am vorbit despre atributul pattern am făcut practic o validare a linkului de imagine introdus de utilizator.

```
<div class="form-group">
    <label for="image">Image Url</label>
    <input class="form-control" type="text" name="image" id="image"
    pattern=".*.(gif|jpe?g|tiff?|png|webp|bmp)" title="Only images  allowed!"
    placeholder="link.gif, jpeg, png, webp, bmp" value="<%= campground.image %>" required>
```

Cu toate acestea, validarea în partea clientului nu trebuie considerată o măsură de securitate exhaustivă! O aplicație ar trebui să efectueze întotdeauna verificări de securitate asupra oricăror date trimise de formular atât pe partea serverului, cât și pe partea clientului, deoarece validarea din partea clientului este prea ușor de ocolit, astfel încât utilizatorii rău intenționați ar putea trimite cu ușurință date invalide către server.

Pentru a respecta această regulă am repetat validarea de mai sus si în server prin implementarea funcției `isSafe()` pe care o dăm exemplu mai jos:

```
isSafe: function(req, res, next) {
  if(req.body.image.match(/\.(gif|jpe?g|tiff?|png|webp|bmp)$/i)) {

    next();
  }else {
    req.flash('error', 'Only images allowed.\n (.gif, jpeg, png, webp, bmp)');
    res.redirect('back');
  }
 }
}
```

Funcția `isSafe` este apoi folosită în interiorul funcțiilor de rutare atunci când se adaugă o nouă locație în baza de date (`router.post`),

```
router.post("/", isLoggedIn, isSafe, function(req, res){
```

sau când se editează o locație existentă:

```
router.put("/:id", isSafe, function(req, res){
```

Motivul pentru care este necesară validarea și la nivel de server este de a preveni abuzul din partea utilizatorilor rău intenționați. Validarea necorespunzătoare a datelor este una dintre principalele cauze ale vulnerabilităților de securitate. Acesta ne expune site-ul la atacuri precum *header injections* (injectări de antet), *cross-site scripting* și injecțări de NoSQL.

Atacurile de injectare a antetului pot fi folosite pentru a trimite e-mailuri *spam* de pe respectivul server web.

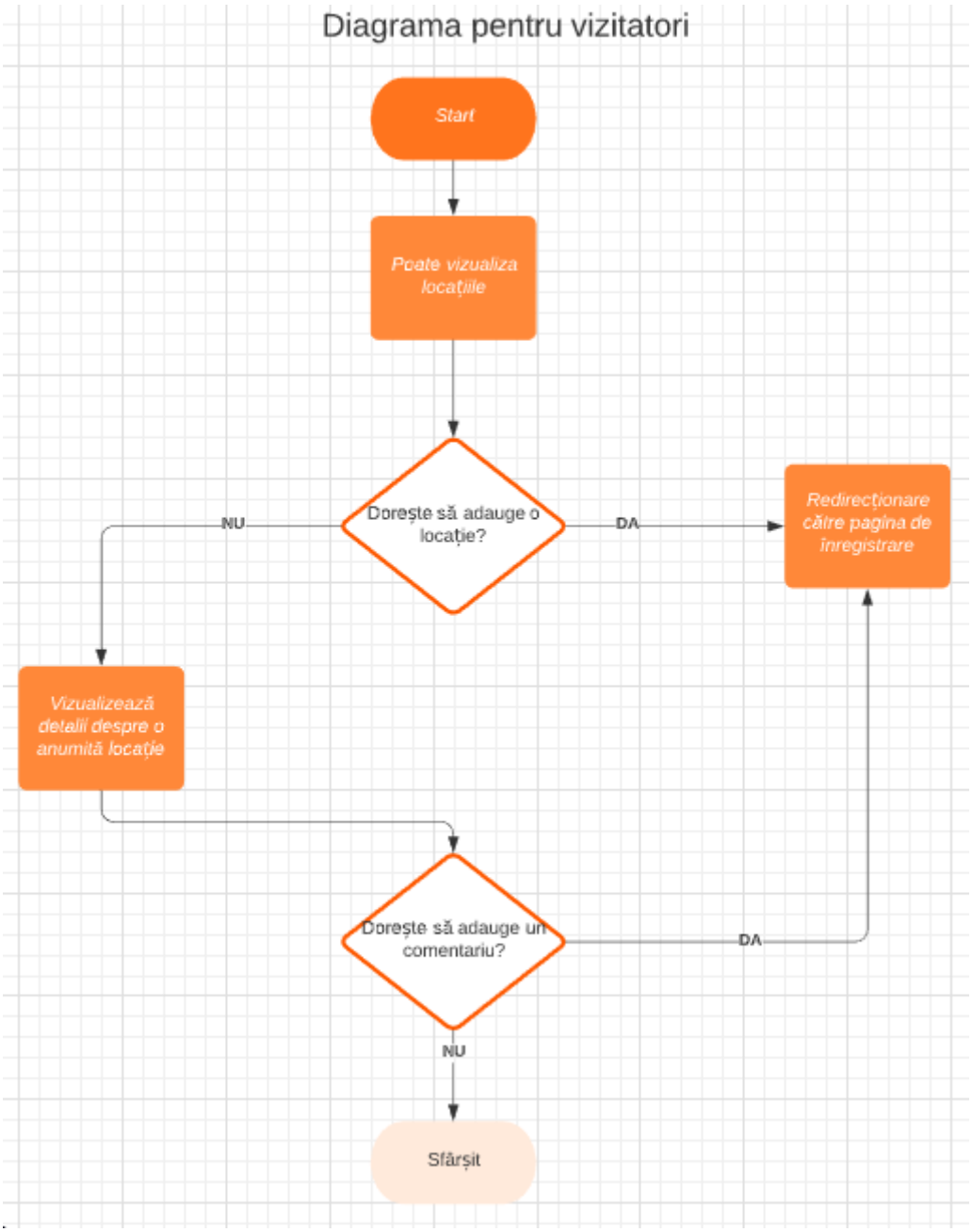
Cross-site scripting-ul poate permite unui atacator să afișeze orice date pe site-ul victimă.

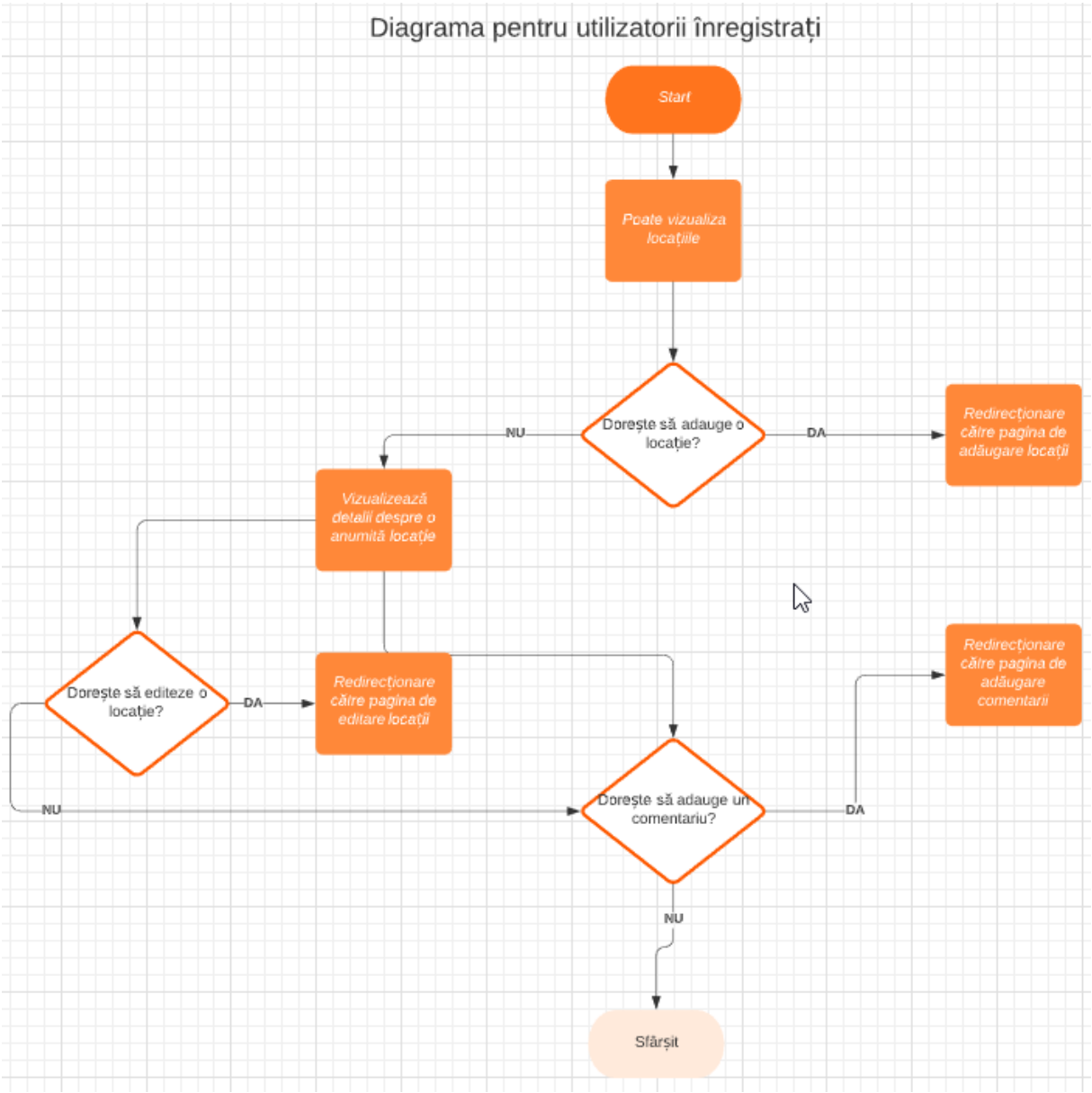
Injecția NoSQL poate deteriora baza de date. O pagină web poate solicita acreditările contului de utilizator într-un formular web, poate stoca informații de autentificare într-o bază de date MongoDB și poate efectua verificări de acreditare utilizând JavaScript. Integrarea strânsă a MongoDB în JavaScript și alte limbaje de programare înseamnă că atacurile de injecție NoSQL pot fi utilizate pentru a efectua atacuri la alte niveluri. De exemplu, Mongo va sprijini evaluarea codului JavaScript dacă este plasat într-o clauză *where* sau trecut într-o funcție de grup sau *mapReduce*. Un atac de injecție NoSQL care include cod JavaScript (formatat corect) va permite executarea codului respectiv în cadrul aplicației web.

3.3 Modul de utilizare

3.3.1 Prezentarea modului de utilizare

Aplicația poate fi accesată și utilizată atât de către vizitatori (utilizatorii care nu sunt înregistrați) cât și de utilizatorii cu drepturi depline (cei care și-au creat un cont și s-au înregistrat). Între aceste două categorii de utilizatori există însă unele diferențe în modul de utilizare al aplicației. În continuare vom reda schematic modul de utilizare pentru ambele cazuri:





3.3.2 Aspecte privind interfața de utilizare

Proiectarea interfeței cu utilizatorul (UI – *user interface*) se concentrează pe anticiparea a ceea ce ar trebui să facă utilizatorii și asigurarea faptului că interfața are elemente ușor de accesat, de înțeles și de utilizat pentru a facilita acțiunile respective. Interfața reunește concepte de la interacțiune, design vizual și arhitectură informațională.

Alegerea elementelor de interfață

Utilizatorii s-au familiarizat cu elementele interfeței care acționează într-un anumit mod, din acest motiv trebuie sa fim consecvenți și previzibili în alegerile pe care le facem.

Elementele interfeței includ, dar nu se limitează la:

Controale de intrare: butoane, câmpuri de text, casete de selectare, butoane radio, liste derulante, câmpuri de dată etc.

Componente de navigație: câmp de căutare, glisor, etichete, pictograme etc.

Componente informaționale: sfaturi, pictograme, bara de progres, notificări, casete de mesaje, ferestre modale etc.

Există momente în care mai multe elemente ar putea fi adecvate pentru afișarea conținutului. Când se întâmplă acest lucru, este important să luăm în considerare compromisurile. Uneori elementele care ne pot ajuta să economisim spațiu, pun o sarcină mai mare din punct de vedere mental asupra utilizatorilor, de exemplu forțându-i să ghicească ce se află în meniul derulant.

Cele mai bune practici pentru proiectarea unei interfețe

Pentru proiectarea interfeței va trebui să cunoaștem utilizatorii, inclusiv înțelegerea obiectivelor, abilităților, preferințelor și tendințelor acestora. Apoi vom luam în considerare următoarele la proiectarea interfeței:

Păstrăm interfața simplă. Cele mai bune interfețe sunt aproape invizibile pentru utilizator și evită elementele inutile.

Cream consistență și utilizam elemente comune de interfață. Utilizând elemente comune în interfața, utilizatorii se simt mai confortabil și pot face lucrurile mai repede. De asemenea, este important să cream modele (secvențe similare) în limbă, aspect și design pe tot site-ul pentru a facilita lucrul cu interfața. Odată ce un utilizator învață cum să facă ceva, ar trebui să poată transfera această abilitate în alte părți ale site-ului.

Folosim strategic culoarea și textura. Putem direcționa atenția spre sau redirecționa atenția către elementele folosind culoarea, lumina, contrastul și textura în avantajul nostru.

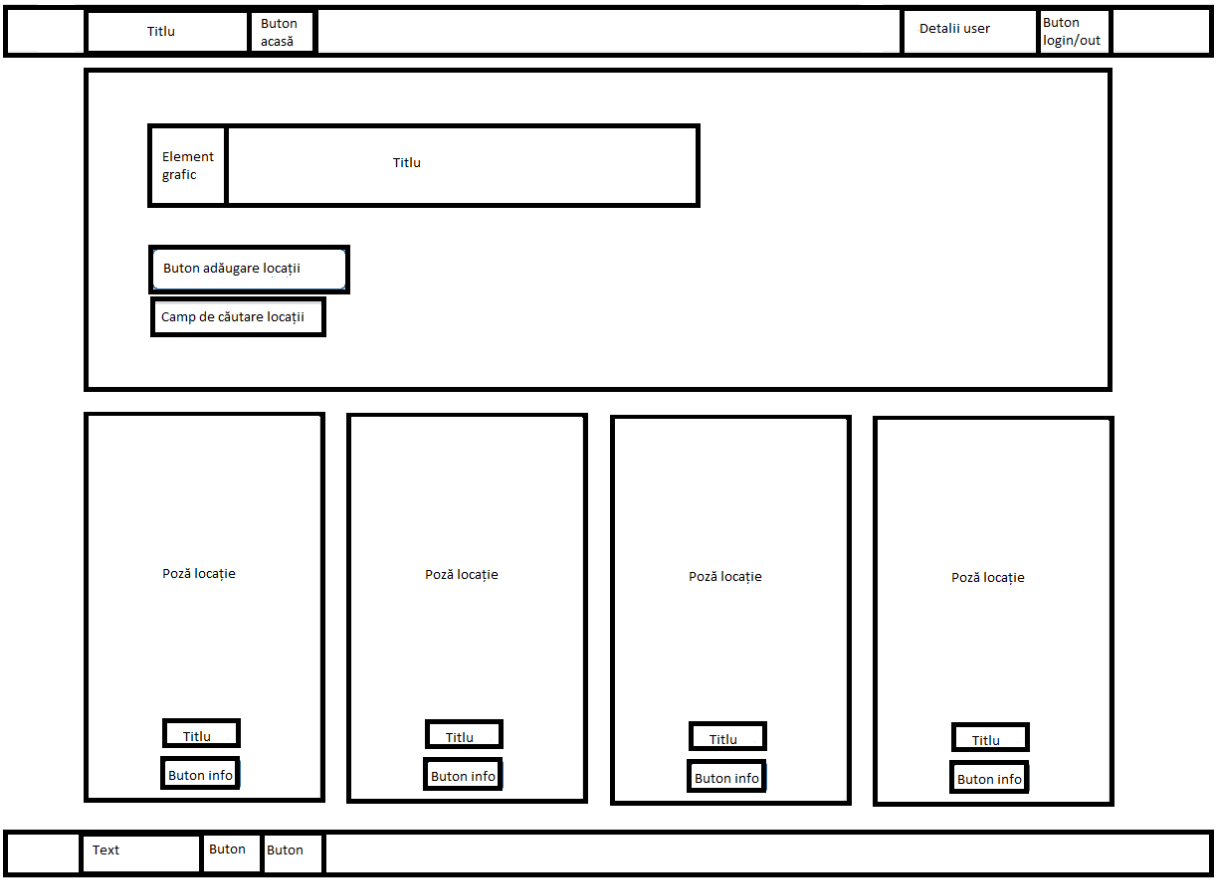
Folosim tipografia pentru a crea ierarhie și claritate. Luăm în considerare cu atenție modul în care utilizăm fontul. Diferite dimensiuni, fonturi și aranjarea textului ne ajută la creșterea vizibilității și lizibilității.

Ne asigurăm că sistemul comunică ce se întâmplă. Ne informăm întotdeauna utilizatorii despre acțiuni, modificări de stare sau erori. Utilizarea diferitelor elemente de interfață pentru a comunica starea și, dacă este necesar, pașii următori pot reduce frustrarea utilizatorului.

Descrierea interfeței de utilizare

Interfața de utilizare a paginilor principale este similară. Întrucât ne referim la o interfață de tip *responsive*, ce se adaptează fiecărui tip de dispozitiv în funcție de mărimea ecranului, nu putem face o descriere exhaustivă a acesteia.

Vom prezenta schematic pagina principală așa cum este afișată pe un laptop cu un ecran de 17 inch:



4. CONCLUZII

Ca o variantă de îmbunătățire viitoare, am putea include *framework*-ul Angular pentru dezvoltarea părții de *frontend*.

AngularJS este un *framework* structural pentru aplicații dinamice. Ne ofera posibilitatea de a folosi sintaxa HTML dar si de a o extinde pentru a dezvolta componentele aplicației noastre. Conceptele de *Data binding* si *dependency injection* folosite în Angular, elimina mult din codul ce ar trebui scris în mod normal.

Prin această îmbunătățire am ajunge la folosirea stivei de tehnologii MEAN (MongoDB, Express.Js, AngularJs si Node.Js). Deținători ai unor platforme importante au ales suita MEAN din punct de vedere arhitectural. Aici putem aminti Walmart, PayPal, Yahoo, Netflix, Uber și LinkedIn.

Pentru o eventuală îmbunătățire a părții de *backend* putem lua în calcul varianta folosirii limbajului Golang, dezvoltat de Google.

Deși anterior am dat exemplu compania Uber ca folosind suita MEAN, este necesar să menționăm ca o parte din microserviciile acestei companii au fost migrate în anul 2016 de la NodeJS la Golang. Principalul motiv care a stat în spatele acestei schimbări este că Uber folosește pentru respectivele microservicii algoritmi de tipul punct în poligon (PIP – *Point In Poligon*) care solicită foarte mult procesorul. Acest lucru combinat cu faptul ca existau peste 170 000 de cereri pe secundă, au făcut necesara trecerea la Golang, care spere deosebire de NodeJS rulează pe mai multe fire de execuție. Cu toate acestea, pentru aplicația prezentată în lucrarea de față, trecerea la Golang ar fi momentan redundantă întrucât cererile făcute către server nu sunt de o asemenea complexitate încât să putem beneficia de avantajele acestui limbaj de programare.

În încheiere, putem spune că am reușit să dezvoltăm o aplicație folosind în întregime JavaScript, un limbaj suficient de rapid, scalabil, asincron și ușor de utilizat, toate acestea făcându-l potrivit pentru implementarea proiectului prezentat.

5. BIBLIOGRAFIE

[1] DAVID FLANAGAN, JavaScript: The Definitive Guide - 7th Edition, O'Reilly Media, Inc., 2020

[2] MATT FRISBIE, Professional JavaScript for Web Developers - 4th Edition, Wrox, 2019

[3] KEITH J. GRANT, CSS in Depth, Manning Publications, 2018;

[4] HENRIK STORMER, Personalized Websites for Mobile Devices using dynamic Cascading Style Sheets, University of Fribourg, 2004;

[5] SYED FAZLE RAHMAN, Jump Start Bootstrap, SitePoint Pty. Ltd, 2014;

[6] SYED FAZLE RAHMAN, Your First Week With Bootstrap, SitePoint Pty. Ltd, 2018;

[7] JAKE SPURLOCK, Bootstrap: Responsive Web Development, O'Reilly Media, Inc., 2013;

[8] JOE CASABONA, HTML and CSS: Visual QuickStart Guide, Peachpit Press, 2020;

[9] JULIE C. MELONI, Sams Teach Yourself HTML, CSS, and JavaScript All in One - Third Edition, Sams, 2019;

[10] DAVID HERRON, NodeJS Web Development - Fifth Edition, Packt Publishing, 2020

[11] BRADLEY MECK, NodeJS in Action, Manning Publications, 2017

[12] DAVID GREEN, Your First Week With NodeJS, SitePoint, 2020

[13] JON WEXLER, Get Programming with NodeJS, Manning Publications, 2019

[14] HAGE YAAPA, Express Web Application Development, Packt Publishing, 2013

[15] ETHAN BROWN, Web Development with Node and Express, O'Reilly Media, Inc., 2019

[16] EVAN M. HAHN, Express in Action: Writing, building, and testing NodeJS applications, Manning Publications, 2016

[17] NICHOLAS MCCLAY, MEAN Cookbook, Packt Publishing, 2017

[18] SHANNON BRADSHAW, MongoDB: The Definitive Guide, O'Reilly Media, Inc., 2019

[19] MARIOT TSITOARA, Beginning Git and GitHub, Apress, 2019

[20] MITESH SONI, Jenkins 2.x Continuous Integration Cookbook, Packt Publishing, 2017

[21] MITESH SONI, Jenkins Essentials, Packt Publishing, 2017