UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA

DE

MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

APLICAȚIE WEB PENTRU PROMOVARE TURISTICĂ

Absolvent
Anton Dolete

Coordonator științific
Îndrumător Mihai Prunescu

București, septembrie 2022

Rezumat

Platforma creată pentru prezentarea lucrării „*PlacesYouShare*” permite atașarea elementelor grafice (imagine) și text (descriere și preț aproximativ pentru serviciile turistice), si un pin al locației specificate. Locația va fi reprezentată pe mapă, ajutând atât la orientarea geografică mai simplă, cât și la crearea unei imagini de ansamblu a zonei care este descrisă.

Feature-ul care doresc să-l evidențiez este gruparea de locații pe hartă, în momentul in care aplicația va avea mult mai multe locații atașate. Care poate contura utilizatorului un istoric al locațiilor vizitate, și o grupare de locații dintr-o anumită zonă. Spre exemplu pentru legarea unui traseu de obiective turistice, popasuri sau atracții cunoscute sau mai puțin cunoscute. Personal, istoricul locațiilor vizitate și adăugarea pozelor din acele locuri este unul din hobby-urile mele și acesta este motivația acestei aplicații.

Aplicația este bazată pe *Node.js, Express, MongoDB. Mongoose* pentru configurarea modelelor *MongoDB.* Baza de date a fost hostată în *Mongo Atlas, NodeJs* a fost folosit ca enviroment pentru server-side. Express pentru *HTTP* routes. *Method-Override* pentru a suprascrie funcțiile de *update* și *delete*. *EJS* pentru *templating, Passport*.js pentru funcția de *hash* a parolei și adăugarea de "*salt*". Express-Sessions pentru configurarea cookie-urilor, Geocoder cu MapBox pentru obținerea locațiilor/coordonatelor pentru ClusterMap. Bootstrap pentru *mobile responsive* și *CSS*. În cele din urmă, pot spune că am creat o aplicație de la început la final folosind *JavaScript*, în care mi-am întărit convingerile că este suficient pentru implementarea proiectului adus în discuție.

Summary

The platform created for the presentation of works "PlacesYouShare" allows the attachment of graphic elements (image) and text (description and approximate price for tourist activities), and a pin of the specified location. The location will be represented on the map, helping both to easier geographical orientation and to create an overview of the area being described.

The feature I want to highlight is the grouping of locations on the map, when the application will have several locations attached. Which can outline to the user a history of visited locations, and a grouping of locations in a certain area. For example, to link a route to a tourist objective, stops or well-known or lesser-known attractions. Personally, the history of visited locations and adding pictures from those places, one of my hobbies and this is the motivation of these apps.

The application is based on Node.js, Express, MongoDB. Mongoose for configuring MongoDB models. The database was hosted in Mongo Atlas, NodeJs was used as the server-side environment. Express for HTTP routes. Method-Override to override update and delete functions. EJS for templating, Passport.js for the password hash function and adding "salt". Express-Sessions for configuring cookies, Geocoder with MapBox for obtaining locations/coordinates for Cluster Map. Bootstrap for mobile responsive and CSS. Finally, I can say that i have created an application from start to finish using JavaScript, which reinforces my conviction that it is sufficient to implement the project discussed.

Cuprins

1. INTRODUCERE

1.1 Aplicabilitatea aplicației și motivația lucrării

1.1.1 Motivația lucrării

Motivația acestei lucrări este de a oferi o imagine de ansamblu de baza a tehnologiilor software și a principiilor de codare necesare de a crea o aplicație online receptivă care poate fi ușor accesibilă de către utilizatori care folosesc orice navigator web.

Vom aminti tehnologiile utilizate pentru implementarea aplicației în prima parte a proiectului înainte de a ne concentra pe descrierea procesului de dezvoltare și a produsului software rezultat.

Proiectul va fi prezentat și dezvoltat în articol din punct de vedere academic. Este nevoie de modificări pentru a facilita conformitatea cu normele legislației (inclusiv normele GDPR[1]) și standardele de securitate cibernetică înainte de deploy-ul site-ului in *WWW*[2].

1.1.2 Motivația PlacesYouShare

Scopul PlacesYouShare este de a facilita promovarea obiectivelor turistice mai puțin cunoscute oferind o platformă digitală atât persoanelor care doresc să-și împărtășească părerile despre o zonă turistică frecventată, dar și celor care caută noi locuri de vacanță sau week-end trips.

Platforma permite atașarea elementelor grafice (imagine) și text (descriere și preț aproximativ pentru serviciile turistice), si orientarea geografică a locației specificate. Acesta din urmă va fi reprezentat pe mapă, ajutând atât la localizarea simpla a locației, cât și la crearea unei imagini de ansamblu a zonei care este descrisă.

Cei interesați să facă cunoscută o anumită locație pot folosi platforma dezvoltată ca instrument de promovare. În acest fel, se pot aduce beneficii din punct de vedere turistic al locațiilor prin atragerea atenției către locațiile respective, facilitând lipsa campaniilor promoționale costisitoare.

[1] General Data Protection Regulation
[2] World Wide Web

1.2 Structura lucrării

Aceasta lucrare este secționată în 4 capitole:

Scopul lucrărilor și aplicației sunt descrise în introducere. Această parte oferă clarificări privind subiectul ce va fi discutat și, de asemenea, explică pe scurt selectarea acestei teme.

Explicarea teoretică a subiectelor și instrumentelor software este tratată în secțiunea următoare. Începem cu metodologiile, ideile abordate în dezvoltarea *Frontend*-ului , apoi trecem la metodologiile, ideile abordate în dezvoltarea *Backend*-ului . Subliniem instrumentele și ideile din spatele integrării continue, automatizării aplicațiilor și punerii în funcțiune pentru a încheia această secțiune.

Descrierea aplicației, proiectarea și implementarea acesteia sunt subiectele principale ale penultimei secțiuni.

În partea finală , intitulată „*Concluzii*”, sunt discutate aspecte ale folosirii și dezvoltării lucrării practice , împreună cu potențialele beneficii și eventuale viitoare îmbunătățiri ale aplicației.

2.PREZENTAREA TEORETICĂ A METODOLOGIILOR SI A COMPONENTELOR SOFTWARE

Scopul acestei secțiuni este de a introduce instrumentele software necesare pentru PIF-ul aplicației, precum și cele care servesc drept fundație pentru dezvoltarea sa.

Dualitatea care este fundamentală pentru arhitectura unei aplicații *full-stack* are o influență directă asupra organizării capitolului (aplicație client-server). Ca rezultat, vom discuta mai întâi despre tehnologiile utilizate pentru a crea partea client (*Frontend*), apoi ne vom îndrepta atenția asupra tehnologiilor *Backend*.

Utilizarea instrumentelor de integrare, automatizare, implementare și exploatarea aplicației va fi tratată în ultima parte a acestui capitol.

2.1 Dezvoltarea Front-end

2.1.1 Limbajul de programare JavaScript

Limbajul principal de programare a site-urilor web este J*avaScript*. Majoritatea paginilor web utilizează *JavaScript* și fiecare browser web actual — desktop, tabletă, telefon — include un interpretor *JavaScript*.

JavaScript este popular in rândul industriei software datorită lui *NodeJS*, care a făcut posibilă scrierea codului *JavaScript* în afara browserelor.[1]

Scopul JavaScript

JavaScript a fost introdus în `95, unul dintre obiectivele sale principale a fost să preia unele dintre sarcinile de validare a intrărilor care fuseseră anterior gestionate de limbaje de pe partea serverului. La acea vreme era nevoie de o „*călătorie dus-întors*” la back-end de a stabili dacă un anume „*field*”, are o valoare nul sau dacă valoarea este incorectă. JavaScript folosește o caracteristică nouă pe care *Netscape Navigator* a introdus-o pentru a încerca să remodeleze acest lucru. Când modem-urile *dial-up* erau încă utilizate pe scară largă, abilitatea de a gestiona o anumită validare simplă pentru client era o nouă capacitate interesantă. Fiecare solicitare adresată serverului de atunci necesita răbdare din cauza vitezei lente asociate.[2]

JavaScript s-a dezvoltat într-o componentă crucială a fiecărui navigator disponibil astăzi. *JavaScript* interacționează acum cu practic fiecare element al ferestre a navigatorului și conținutul acestuia, nefolosit doar pentru validarea simplă a datelor((închiderile (*closures*), funcțiile anonime (*lambda)* și chiar metaprogramarea)) sunt toate incluse în *JavaScript* ca sa arate capabilitatea acestuia să gestioneze calcule si decizii complexe. *JavaScript* s-a dezvoltat într-o componentă crucială pentru internet, acum este acceptat de alte navigatoare, inclusiv cele pentru dispozitive mobile și persoanele cu deficiențe.

Chiar și *Microsoft* a inclus în cele din urmă un *JavaScrip*t făcut în întregime de ei în *IE* începând cu prima ediție, în ciuda faptului că avea propriul limbaj de *scripting* numit *VBScript*. Evoluția *JavaScript* de la un limbaj bază la un limbaj mult mai evoluat nu a putut sa fie prevăzută. E necesar doar de câteva minute pentru a familiarizarea cu *JavaScript*, dar durează ani pentru a înțelege bine acest limbaj extrem de simplu, dar foarte complex. Este esențial să înțelegi natura și restricțiile *JavaScript* înainte de a începe călătoria spre utilizarea acestuia la maximul său potențial.[2]

Scurtă istorie

Pentru lansarea *Netscape Navigator 2*, Brendan Eich, un inginer *Netscape* la acea vreme, a demarat creerea un limbaj denumit *LiveScript, asta* în `95. Acesta a intenționat să fie folosit pe server, unde va fi cunoscut sub numele de *LiveWire*, precum și în *navigator*.

Pentru a termina *LiveScript* la timp, *Netscape* a format un parteneriat de colaborare cu Sun Microsystems. Valorificarea popularității limbajului de programare Java la acea vreme, *Netscape* a redenumit *LiveScript* în J*avaScript* apoi a fost lansat publicului.

Datorită popularității JS 1.0, a fost lansat JS 1.1 Într-o perioadă în care utilizarea online era la un nivel maxim, Netscape s-a poziționat drept lider în industrie. În acest moment, Microsoft a luat decizia de a investi mai multe resurse în Internet Explorer, un browser rival. Microsoft a dezvăluit IE3 cu o dezvoltare numita JScript la scurt timp.

Adoptarea Microsoft a JavaScript a decurs la crearea a unei noi versiuni JavaScript si JScript în IE.[2]

JS 1.1 a fost adus la cunoștința *Asociației Europene a Producătorilor de Calculatoare (ECMA)* în 1997. Pentru a „standardiza sintaxa și semantica unui limbaj de scripting neutru de tip

multiplatformă”, a fost înființat comitetul tehnic numărul 39 (TC39). Standardul ECMA-262, care specifică noul limbaj cunoscut sub numele de ECMAScript.

În anul următor, ECMAScript a fost acceptat și ca standard de către(ISO/IEC-16262). Din acel moment, navigatoarele web au încercat folosirea ECMAScript ca fundație de a dezvolta JS.

Implementarea JavaScript

JS este substanțial mai mare decât se descrie, în ciuda faptului că termenii sunt folosiți frecvent în mod interschimbabil. Următoarele trei componente separate alcătuiesc o implementare completă JavaScript Figura 2.1[2]:

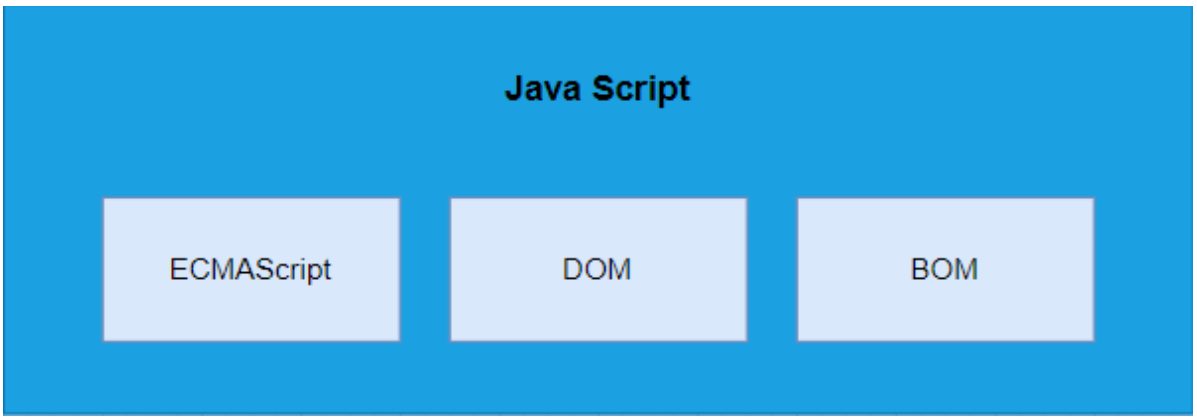

Figura 2.1

Standardul ECMAScript / (*DOM*) / (*BOM*)

Standardul definit, nu are conexiune directă cu navigatorele web. O bază pentru construirea de limbaje de scripting mai puternice este stabilită în *ECMA-262*. Singurul mediu gazdă posibil pentru o implementare *ECMAScript,* este un navigator web. Sintaxa ECMAScript sunt utilizate de extensii, cum ar fi *DOM* , pentru a extinde funcții specifice. *NodeJS* fiind folosit frecvent pentru dezvoltarea backend, este un mediu gazdă suplimentar.

Problema devine așadar, dacă *ECMA-262* nu se referă la navigatoarele web, ce specifică ? La cel mai fundamental nivel, descrie următoarele componente lingvistice: ECMAScript este doar o descriere a diverselor specificații, inclusiv sintaxă, variabile, instrucțiuni, cuvinte rezervate, operatori și obiecte globale cat și cuvinte cheie. JavaScript nu este singurul limbaj care acceptă ECMAScript. *Adobe ActionScript* este o ilustrare diferită.[2]

Structura JavaScript în plan lexical

Setul de bază de definiții care definesc felul în care aplicațiile trebuie să fie construite în acel limbaj . O sa o numim sintaxă la nivelul său cel mai fundamental.[1]

JavaScript face parte din categoria tehnologiilor *case sensitive*. Ca rezultat, este imperativ să introduceți toți identificatorii, variabilele , cuvintele rezervate în mod consecvent.

Spațiile care există între elementele programului sunt ignorate de *JavaScript*. Alături de caracterul spațiu standard (u\0020), *JavaScript* înțelege pana si tab ca si spațiu.

JavaScript Comentarii(Notițele pentru mai târziu)

JavaScript suportă două feluri de *comments*. JavaScript consideră totul în „//” și capătul unui rând ca fiind un comentariu și îl ignoră. Un comentariu este definit ca orice text între literele „/ *” și „* /”, care se poate extinde pe o porțiune mai mare, dar nu și imbricat.

JavaScript Identificatorii și setul de cuvinte rezervate

Simplu spus, *identificatorii* sunt *nume*. *Identificatorii* sunt folosiți în JS pentru a da nume *constantelor*, *variabilelor*, *proprietăților*, *funcțiilor* și *claselor*, precum și pentru a oferi etichete codului în interiorul buclelor. Un identificator JS începe cu o literă și acest lucru este strict, caracterul(_), sau($). Literele, cifrele, caracterul *underscore*(_), caracterul *dolar*($) sunt acceptate ca caractere ulterioare.

JavaScript cuvintele rezervate

Ca și orice alt limbaj, *JS* folosește doar un anumit set de identificatori pentru propriile scopuri. Acești „termeni rezervați” nu pot fi utilizați ca identificatori obișnuiți(de exemplu identificarea variabilelor sau a constantelor).

Limbajul *JavaScript* include termenii de mai jos. Mulți dintre aceștia sunt termeni rezervați și nu ar trebui folosiți ca *nume* pentru *constante, variabile, funcții* sau *clase* (precum *if, while* și *for*) deși toți pot fi folosiți ca nume de proprietăți într-un obiect. Altele, inclusiv *from*, *of*, *get* și *set*, sunt folosite cu moderație și fără ambiguitate gramaticală ca identificatori și sunt în întregime valide. Alte cuvinte cheie, ca si *let*, nu se pot rezerva în totalitate pentru păstrarea compatibilității anterioară, prin urmare există restricții complicate care dictează când pot și nu pot fi utilizate ca identificatori. (De exemplu, dacă *let* este definit *var* în *exteriorul* clase, atunci se poate folosii ca

identificator; dar, în cazul in care *let* este definit ca *const* în interiorul clasei, nu se poate.) Toți acești termeni, exceptând *of* ,set, target si *from*, care se pot utiliza și sunt utilizate în prezent pe scară largă, nu ar trebui să fie utilizate ca identificatori.[1]

Unicode

Orice caracter *Unicode* poate fi folosit în șiruri și comentarii, deoarece aplicațiile *JavaScript* sunt create folosind setul de caractere *Unicode*. Este o practică obișnuită să se limiteze ID-urile la caractere și cifre ASCII pentru portabilitate și simplitate a editării. Limbajul permite doar litere, cifre și caractere *Unicode* ca identificatori; aceasta este pur și simplu *o convenție*. Aceasta implică faptul că, în timp ce declară *constante* și *variabile*, programatorii pot folosi caractere și expresii matematice provenind din alte limbaje decât engleza.

JS este limbajul folosit la crearea unei pagini interactive, fiind o componentă crucială a unei aplicații. *JavaScript* este una dintre tehnologiile web fundamentale, împreună cu *HTML* și *CSS*.[1]

2.1.2 Cascading Style Sheets impreuna cu Bootstrap

Limbaje de marcare precum *HTML*, cum ar fi *CSS*, este utilizat la specificarea părții vizuale a unui document. Alături de *HTML* și *JavaScript*, *CSS* este unul din *trio-ul* WWW.

Scurta istorie a CSS

Când Hkon Wium Lie a început să lucreze la CERN, locul de naștere al rețelei, în `94, abia atunci începe utilizarea ca spațiu de publicare digitală. Acesta este momentul în care începe povestea CSS. Cu toate acestea, lipsea o componentă fundamentală a unei spațiu digital de publicare, nu era o metodă pentru *stilare* lucrărilor. De exemplu, nu a fost posibilă formatarea unei pagini Web cu un stil de ziar. Hkon a văzut cererea pentru un limbaj de *design* ca si o necesitate cruciala.

Propunerea creării unui limbaj de design , a fost făcută la una din conferințele web ale acelei vremi, la Chicago in anul `94, iar prezentarea a făcut furori sunt multe dezbateri pe aceasta tema. Mulți au susținut că *CSS* este ușor pentru task-urile care a fost creat, cu argumente ca si cum necesitatea cunoștințelor de programare cuprinzătoare pentru a stila corect documentele. *CSS* are o săgeată precisă îndreptată în direcția corectă, derulând o sintaxă clară și declarativă.

Este greu de spus cât de larg este folosit *CSS* acum, de la debutul său oficial din `96, deși procentul de pagini care nu se folosesc *CSS* probabil nu poate fi mai mare de un număr mic de puncte procentuale.

CSS argumente PRO si CONTRA

Prin utilizarea *CascadingStyleSheets*, posibil să se separe estetica unei aplicații web, cum ar fi aspectul paginii, culorile și font-urile, de textul acesteia. Această împărțire poate crește accesibilitatea conținutului, oferind, de asemenea, specificatorului elementelor de prezentare mai multă libertate și control. În plus, elimină complexitatea și duplicarea codului, permițând furnizarea codului *CascadingStyleSheets*, pertinent in fișiere diferite. Reținerea în cache a *fișierelor* pe care l-au partajat, timpul de încărcare a paginii este, de asemenea, accelerat.

Putem susține că *CascadingStyleSheets*, se deosebește in domeniul dezvoltării software pe o serie de aspecte. Deși nu este din punct de vedere tehnic un limbaj de programare, necesită gândire abstractă. Necesită puțină ingeniozitate și este mai mult decât un instrument de proiectare.

Cuvântul „în cascadă” se referă la structura de prioritate declarată care determină ce regulă de stil este implementată atunci când multiple seturi de instrucțiuni se refera la un element. Acest set de priorități poate fi prezis, dar poate cauza probleme pentru proiecte de o complexitate considerabilă, ajungând în unele situații (când *front-end* se dezvolta de mai mulți dezvoltatori), mai ales dacă regulile sunt omise si încălcate, când se introduc seturi de reguli greșite în cod.

Atunci când este necesar să efectuăm o acțiune folosind tehnici standard de programare, putem identifica rapid problema care trebuie rezolvată (de exemplu, „Cum descopăr componente de tip x într-un tablou?”). Tehnica ideală pentru a realiza orice este adesea determinată de limitările specifice ale aplicației noastre și de modul exact în care dorim să abordăm anumite situații. Deși sunt necesare „tricks” practice pe care le putem folosi, înțelegerea *CascadingStyleSheets* necesită o definire aprofundată a regulilor care permit astfel de actiuni.[3] *CascadingStyleSheets* adaugă un vocabular declarativ ușor de greșit, de îndată ce îl folosim pentru aplicații care sunt cel puțin moderat complicate, devine din ce în ce mai dificil să implementăm regulile *CSS* într-o manieră ordonată.

Începând cu anii 2000, odată cu creșterea numărului de dispozitive mobile, a crescut si cerea pentru dispozitive mult mai speciale si specifice. La acea vreme, *WAP* sau *I-mode*, se puteau

folosii pentru a converti documentele web pentru dispozitive mobile, dar acest lucru a necesitat un efort considerabil, deoarece aproape toate site-urile trebuiau reconstruite complet.

În prezent avem sute de resurse *CSS* din care să alegem datorită muncii depuse de diverse organizații în efortul de a remedia limitările prezentate , pentru ca in prezent site-urile web sa aibă o structura ușor inteligibila, un design plăcut și optimizat pentru toate dispozitivele.

Framework-uri CSS

Între anii 2006–2007, resursele *CSS* precum *Blueprint* și *Yahoo* s-au utilizat pe scară largă. Au adus cu ei mai multe instrumente esențiale, inclusiv butoane, grile, efecte de animație, fonturi și resetarea *CSS*. Adoptarea acestor resurse de către dezvoltatori are potențialul de a reduce semnificativ timpul necesar dezvoltării site-urilor web prin gestionarea multor activități repetitive și laborioase. O generație de resurse frontale *„full-edge*", inclusiv *Bootstrap,* care a inclus *JavaScript* în implementarea sa, a venit după aceste resurse *CSS* fundamentale.[5]

Framework-ul Bootstrap

În timp ce lucrau la Twitter, Mark Otto și Jacob Thornto au creat aplicația *open source* cunoscută sub numele de Bootstrap. Nevoia de *Bootstrap,* subliniază Mark Otto în post-ul pe blog care s-a anunțat introducerea sa, a izvorât din necesitatea de a standardiza numeroasele tehnologii front-end utilizate de dezvoltatorii din întreaga organizație.

În 2011 a fost lansat *Bootstrap* v1.0, care a inclus doar elemente *CSS* și *HTML*. Până in versiunea v1.3 a *Bootstrap,* aceasta fiind compatibila cu InternetExplorer7 și InternetExplorer8, nu erau prezente plugin-uri *JavaScript*. O altă schimbare semnificativă a venit în 2012 cu Bootstrap v2.0. Biblioteca *Bootstrap* a fost rescrisă în întregime, iar ulterior a evoluat într-un cadru receptiv. Telefoanele mobile, tabletele și desktop-urile puteau folosi toate componentele sale și au fost incluse și numeroase pachete *CSS* și *JavaScript* noi. *Bootstrap 3,* o altă lansare mare, devenind un cadru „*Mobile First* and *Always Responsive*" după 15 modificări substanțiale. Crearea unui site web receptiv a fost opțională în iterațiile anterioare ale cadrului. Structura de directoare a proiectului, precum și numele claselor au fost modificate în versiunea 2013. Cu toate acestea, *Bootstrap v3.o* nefiind compatibil cu versiunea precedenta, în sensul că fișierele *CSS* și *JavaScript* primare nu pot fi actualizate direct la această versiune[3]. *Bootstrap*, ajuns la a patra ediție, continuă să fie unul dintre cele mai populare resurse în rândul dezvoltatorilor, mai ales pentru că este *open*

source, funcționează cu o multitudine de navigatoare web, reduce timpul de dezvoltare și se poate modifica. Dar arhitectura grilă a resursei este, fără îndoială, componenta sa cea mai avantajoasă.

Cele mai receptive sisteme „grilă" create pentru dispozitive cu o rezoluție mai mica, se găsește în *Bootstrap*. Prin împărțirea logică a ecranului în 12 coloane, este mai ușor să scalați un document web pentru toate tipurile de rezoluție. Ca rezultat, programatorul poate alege cât de mult din suprafața afișajului ar trebui să ocupe fiecare element de design. [6]

Cele 12 coloane din sistemul standard de grilă *Bootstrap* (vezi *Figura 2-2*) oferă un container lățime de 940 px fără utilizarea caracteristicilor receptive. Grila se adaptează la 724/ 1170px cu includerea fișierului *CSS responsive*. Coloanele devin fluide și se stivuiesc vertical pentru afișaje mai mici de 767 px, cum ar fi cele de pe tablete și alte dispozitive portabile.

Figura 2-2 ilustrează modul în care piesele pot fi aranjate pe un *layer* de 12. [7]

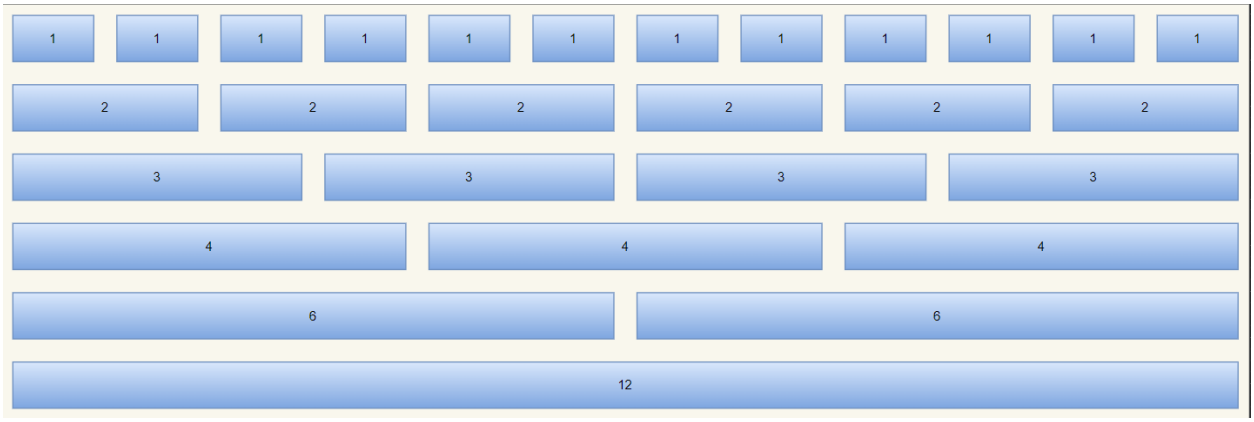

Figura 2-2

Crearea meniurilor si a efectelor sunt doar câteva dintre nevoile fundamentale de dezvoltare care sunt îndeplinite de *Bootstrap*, care combină componente *CSS* și *JavaScript* utilizate pe scară largă. Pe lângă faptul că include o varietate de elemente utile care sunt ușor de utilizat în proiectarea de site-uri web, *Bootstrap* folosește și limbajul *HTML* standard. Dezvoltatorii trebuie pur și simplu să se concentreze pe generarea de marcaje *HTML* adecvate utilizând *Bootstrap*, astfel încât cadrul să îl poată înțelege și să genereze site-ul web așa cum este prevăzut.[3]

De-a lungul anilor, *Bootstrap* a devenit un instrument foarte popular pentru aplicațiile *front-end*. Este o componentă crucială pentru fiecare proiect modern datorită simplității utilizării, interoperabilității între navigatoare, suportului pentru interfețele dispozitivelor mobile și capacității de a crea aplicații web receptive.

2.1.3 **HTML5**

HyperText Markup Language, sau *HTML*, îndeplinește două funcții cruciale: definește semantica paginilor web și specifică cum ar trebui să apară.[8]

WWW si HTML

Accesul la informații prin Internet a fost o provocare tehnică semnificativă înainte de `90. De fapt, in acest tot timp, pana si cele mai clipitoare minți si utilizatori din diferite domenii de activitate din cadrul academic aveau dificultăți, în timp ce încercau să partajeze date. Tim Berners-Lee, a conceput o metodă de accesare rapidă a textului prin Internet, și anume prin utilizarea legăturilor hipertext. Deși acesta nu a fost un concept nou, ușurința de utilizare a *HTML* i-a permis să supraviețuiască atunci când eforturile de *hipertext* mai sofisticate au eșuat.

Hypertext se referea inițial la text stocat electronic cu legături interne între pagini. Aproape orice articol *(fișiere , text, fotografii, etc*.) care poate face legătura cu alte lucruri este acum menționat cu acest nume mai mare. Organizarea și legarea de text, imagini și alte fișiere care conțin informații sunt descrise folosind limbajul de marcare *hipertext*.[9]

Doar aproximativ 100 de calculatoare erau capabile să deservească site-uri web *HTML* până în anul `93. *World Wide Web (WWW)*, care constă din aceste pagini conectate, a inspirat crearea unui număr de navigatoare web care permit utilizatorilor să privească un document. Din cauza popularității în creștere a *WWW*, mai mulți dezvoltatori au creat navigatoare care ar putea afișa atât *text, cât și imagini*.

2.2 Modulele utilizate pentru Backend

2.2.1 NodeJS

În afara navigatoarelor web, aplicațiile *JavaScript* pot fi create folosind cadrul *NodeJS*. *NodeJS* rulează aceeași versiune JS integrat in navigatoare, dar îi lipsesc unele dintre capabilitățile însoțitoare, astfel încât mediul *JS* cu care suntem obișnuiți în navigatoare nu i se aplică exact. De exemplu, *NodeJS* nu vine cu un *DOM HTML* încorporat. [10]

NodeJS se bazează pe motorul *JS versiunea a8-a* găsit în *Google Chrome*. Acesta este motorul *JS* gratuit și open-source care este folosit de Google *Chrome* și de alte navigatoare ca

fundație pe *Chromium* pentru a traduce *JavaScript* în cod de mașină executabil(putem aminti navigatoare ca Opera).

Popularitate

Numeroase companii, atât mari cât și mici, folosesc *NodeJS*, care câștigă rapid popularitate ca platformă de dezvoltare. *PayPal* este unul dintre primii adoptatori, trecând la *NodeJS* de pe platforma lor bazată pe *Java*. Împreună cu aceasta, este posibil să aducem și piața online *Walmart*.

Motivele fundamentale ale NodeJS

Un mediu de programare *JS async,* având la bază „events”, numit *NodeJS,* asigura o librărie robustă, totodată succintă.[11]

Într-un limbaj simplu, un nou *thread* va fi creat pentru a gestiona cererea atunci când realizam conectarea la un server convențional ca si *Apache*. Orice acțiune ulterioară de intrare sau de ieșire (cum ar fi interfața cu o bază de date, de exemplu) oprește rularea codului în limbaje precum *PHP* sau *Ruby* până când operația este terminată. Cu alte cuvinte, serverul trebuie să aștepte ca căutarea în baza de date să fie terminată înainte de a procesa rezultatele. Serverul va crea thread-uri de execuție noi pentru a procesa orice solicitări primite dacă ajung în timp ce acest lucru are loc. Atât de multe fire pot încetini un sistem și, în cea mai proastă situație, pot face site-ul inaccesibil. Adăugarea de servere suplimentare este cea mai tipică tehnică pentru a găzdui mai multe conexiuni.[12]

Cu toate acestea, *NodeJS* gestionează doar o singură sarcină la un moment dat și folosește numai *thread*-uri suplimentare pentru activități pe care *thread*-ul principal nu le poate gestiona.

Deși această metodă poate părea paradoxală, de obicei funcționează bine în aplicațiile care nu au nevoie de multă putere de procesare, deoarece un singur *thread* poate efectua eficient toate activitățile.

NodeJS execută cod *asincron* folosind mai multe thread-uri în fundal, în ciuda faptului că *JavaScript* ar trebui să funcționeze într-un singur thread de execuție la suprafață (după cum putem vedea din Figura 2.3).

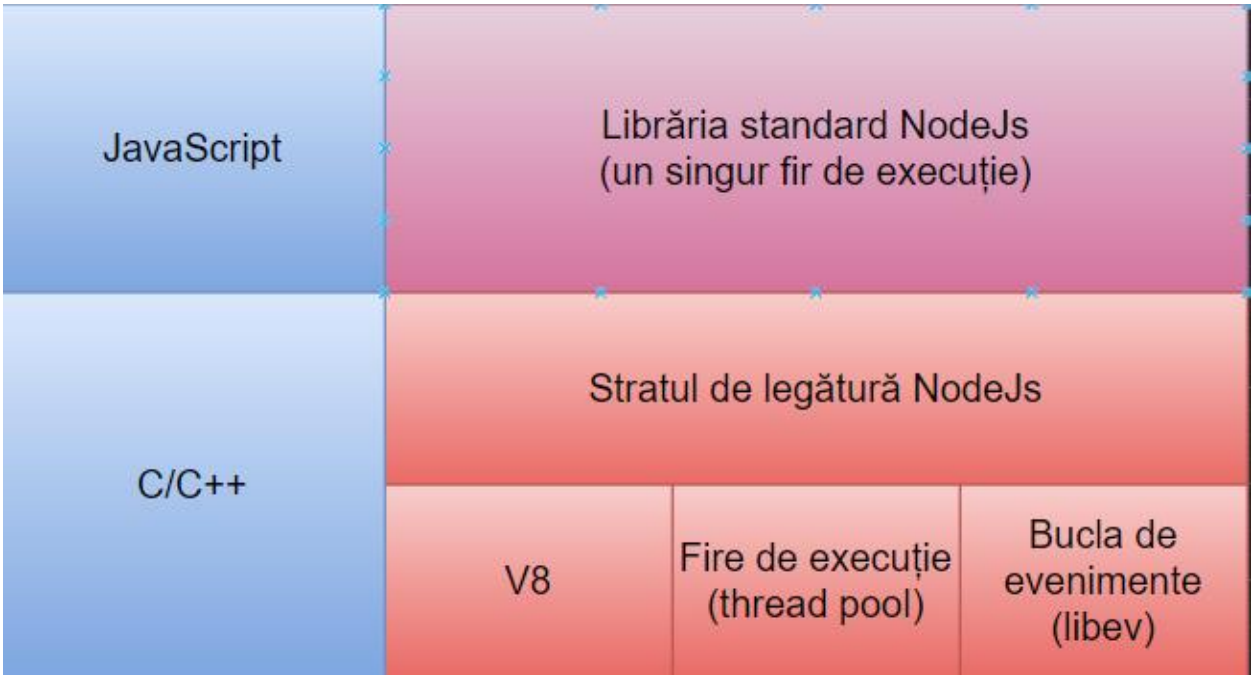

Figura 2.3

Loop-ul de evenimente *NodeJS* este bazat pe doar un *thread* pentru a-și gestiona toate activitățile, dar nu folosește întotdeauna acel *thread* pentru a efectua toate acțiunile complet. Calculatorul gazdă poate stabili noi fire și procese pentru a realiza aceste activități, deoarece *NodeJS* este construit pentru a-i trimite activități mai complexe și mai multe.

De exemplu, *server-ul* va începe să proceseze o nouă solicitare atunci când are loc (*primul eveniment*). Apoi, în loc să aștepte o activitate *de intrare/ieșire* care ar determina în mod obișnuit blocarea *thread*-ului de execuție, va înregistra funcția *call-back* și va continua să gestioneze următorul eveniment. *Server*-ul va reveni la lucru la cererea inițială după ce procesul de *intrare/ieșire* a luat sfârșit (un alt tip de eveniment). Biblioteca *libuv*, care se bazează pe limbajul de programare C, este folosită de *Node* pentru a realiza această funcționalitate asincronă (non-blocant).[12]

Figura 2.4 prezintă o diagramă de *loop* de evenimente condensată. Când un *thread* este pregătit pentru execuție, acesta este adăugat la o coadă și procesat prin anumite etape ale *loop*-ului de evenimente.

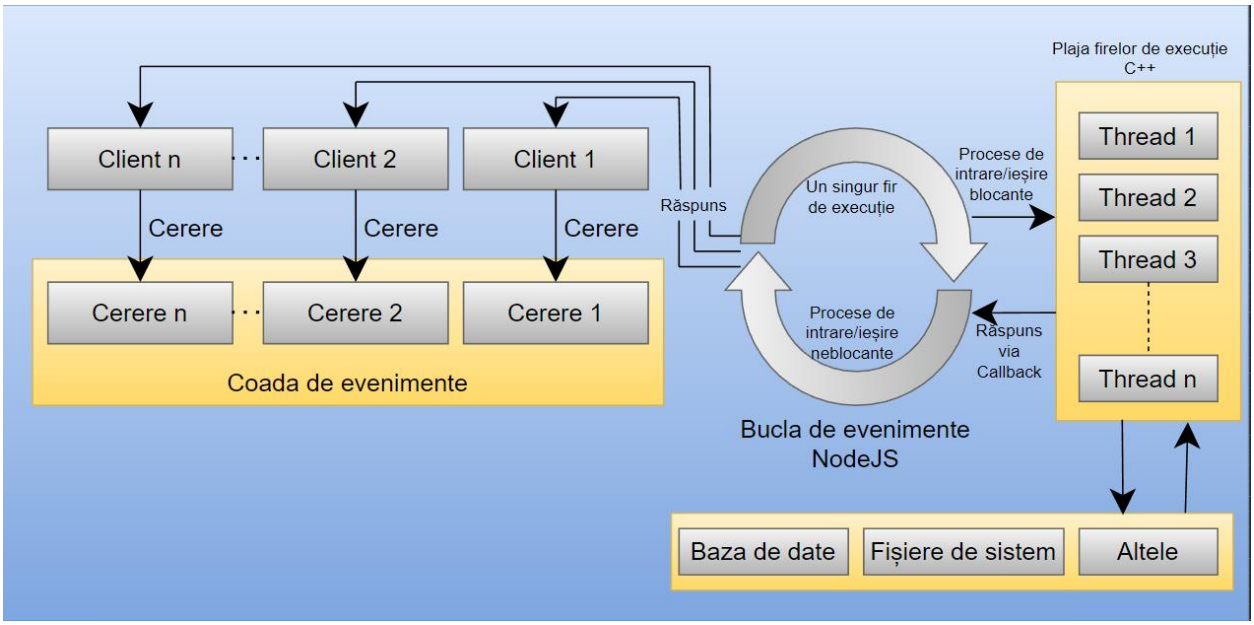

Figura 2.4

Loop-ul de evenimente *NodeJS*, așa cum indică numele, se repetă continuu în timp ce „ascultă” evenimente *JavaScript* emise de server pentru a semnala începutul unei noi sarcini sau terminarea unei activități anterioare. În funcție de activitățile mărite, acestea stau la coadă pentru a fi gestionate pe rând de loop-ul de evenimente. Programatorii trebuie doar să respecte regulile asincrone, în timp ce *NodeJS* se ocupă de programarea, gestionarea și rezolvarea sarcinilor primite.[13]

Inventatorul lui *NodeJS*, Ryan Dahl, a oferit următoarea ilustrație. Dahl ne-a întrebat la discursul său Cinco de NodeJS din mai 2010 despre ce se întâmplă atunci când rulăm o bucată de cod precum:

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

Desigur, în acest moment software-ul se întrerupe și așteaptă rezultatul sau o eroare la interogarea bazei de date. Aceasta este o ilustrare a unui proces care întrerupe temporar *thread*-ul. Durata așteptării variază în funcție de cerere. Utilizatorul va găsi acest decalaj iritant, deoarece *thread*-ul este inactiv în timp ce așteaptă rezultatul. Astfel, rezultă că serverul în ansamblu nu ar putea răspunde la solicitări suplimentare în acest timp dacă programul rulează pe o platformă cu un singur thread. Dacă, totuși, platforma serverului are mai multe thread-uri, este necesară o comutare de context pentru a gestiona orice solicitări suplimentare primite (care vor muta acele

solicitări într-un alt thread). Numărul de schimburi de *thread*-uri crește pe măsură ce crește numărul de conexiuni la server. Comutarea nu este atât de ușoara, deoarece gestionarea mai multor thread-uri de execuție necesită mai multă putere de calcul și necesită mai multă memorie pentru fiecare *thread de execuție*.[10]

Interogarea pe care am examinat-o anterior va apărea după cum urmează în *NodeJS*:

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

Când rezultatul (sau eroarea) este disponibil, programatorul oferă o funcție care va fi apelată; prin urmare, numele funcției de *call-back*. Va dura același timp pentru a interoga baza de date, dar în schimb se întrerupă *thread*-ul de execuție, acesta se întoarce la *loop*-ul de evenimente, eliberându-l pentru a face față solicitărilor ulterioare. După obținerea răspunsului de la interogarea bazei de date, *NodeJS* va declanșa în cele din urmă un eveniment care face ca această funcție *call-back* să fie declanșată cu *succes* sau *eșec*. În *handlerele* de evenimente create de utilizator, *JavaScript front-end* utilizează un concept similar.

Performanță și utilizare

Debitul mare de interogări pe care *NodeJS* le poate procesa este unul dintre motivele pentru o parte din furorile din jurul acestuia (cereri pe secundă pe care le poate servi). *NodeJS* depășește aplicațiile asemănătoare, cum ar fi *Apache*, din punct de vedere al performanței. Următoarele instrucțiuni vor returna mesajul *„Hello World”*[10]

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

Acest șablon de server web simplu se poate crea folosind *NodeJS*. Funcția *http.createServer* a obiectului *http* produce un server web care ascultă pe portul furnizat în timp ce încapsulează protocolul *HTTP* (8124). Funcția furnizată este apelată de fiecare cerere către

serverul web, fie că este un *GET* sau *POST* al oricarui *URL*. În acest caz, un răspuns text simplu cu cuvintele „Hello World” este livrat indiferent de adresa *URL*.

Folosind datele reale de interogare, inginerul Fabian Frank de la Yahoo! a lansat un studiu privind eficacitatea motorului de căutare al companiei. Într-o primă formă, motorul de căutare a fost implementat folosind *Apache* și *PHP*, iar în alte două variante au fost folosite stivele *NodeJS*. Când un utilizator introduce termeni în câmpul de căutare, aplicația apare și oferă recomandări de căutare în timp real. În experiment, a fost folosită o interogare *HTTP* bazată pe *JSON*. Cu aceeași întârziere de solicitare, versiunea *NodeJS* ar putea procesa de opt ori mai multe cereri pe secundă. Potrivit lui Fabian Frank, amândouă stivele *NodeJS* au folosit resurse liniar până când utilizarea procesorului si-a atins capacitățile.

În general, *NodeJS* funcționează excepțional de bine în ceea ce privește debitul I/O bazat pe evenimente. Dacă un software *NodeJS* reușește în aplicații de calcul mai complicate, depinde de capacitatea programatorului de a ocoli mai multe restricții de limbaj *JS*.

2.2.2 ExpressJS

Pentru platforma *NodeJS*, *Express* este un sistem de dezvoltare simplu, eficient cat și adaptabil.

Express este simplu, deoarece nu are o mulțime de funcționalități. Nici măcar funcțiile acceptate nu sunt activate în mod implicit după instalare, lăsând utilizatorului posibilitatea de a selecta ce capabilități să utilizeze în funcție de cerințele proiectului.[14]

Una dintre cele mai importante caracteristici sunt acestea. Dezvoltatorii nu țin cont de „mai puțin este mai mult”. Principiul călăuzitor *Express* este de a pune cât mai puțin posibil între dezvoltator și server. Acest lucru nu insinuează că nu ar fi fiabil sau că îi lipsesc suficiente caracteristici valoroase; mai degrabă, înseamnă doar că obstrucționează capacitatea programatorului de a-și exprima complet gândurile mai puțin. Este, de asemenea, un instrument flexibil care ne permite să înlocuim tot ceea ce nu este necesar pentru proiect și să adăugăm alte capabilități *Express* după cum este necesar. Înainte de a crea o simpla instrucțiune, multe resurse încearcă să ofere totul, proiectul crescând considerabil, ceea ce ajunge adesea să fie criptic și dificil. *Express* adoptă o strategie diferită, permițându-ne să adăugăm conținut așa cum este necesar.[15]

Modulele intermediare și *Node* sunt modul în care *Express* devine flexibil. Componentele *JavaScript* conectabile, cum ar fi *middleware-ul Express* și modulele *Node*, permit aplicațiilor să fie modulare, adaptabile și extensibile. În esență, *Express* este un program simplu care preia solicitări *HTTP* de la clienți (care pot fi navigatoarele, dispozitive mobile, alte servere, aplicații desktop sau aproape orice altceva care utilizează protocolul *HTTP*) și răspunde la aceste solicitări cu *HTTP*. Deoarece practic orice este legat de Internet se încadrează în acest concept fundamental, *Express* este incredibil de versatil în aplicațiile în care poate fi utilizat.

Express este un framework puternic, asigurând ca putem accesa in totalitate *API*-ul *NodeJS*. *Express* poate fi folosit pentru orice pentru care poate folosi *Node*. Orice nivel a aplicațiilor web poate fi creată cu *Express*. Ni se asigura toate resursele de a construi chiar și cele mai complicate aplicații, dar nu suntem obligați să folosim tot timpul acestea.

Scurta istorie a ExpressJs

După ce *NodeJS* a fost lsansat, T.J. Holowaychuk a început să lucreze la un proiect *open source* acum supranumit *Express* de a eficientiza dezvoltarea web pentru aplicațiile bazate pe *NodeJS*. *Express* a fost creat pe deasupra *API-ului NodeJS*. De la introducerea produsului au fost adăugate un sistem de rutare, sesiune și suport pentru cookie-uri, ajutoare MIME, o interfață *RESTful*, vizualizări bazate pe HAML și alte caracteristici semnificative. Dar *Express v0.0.1* nu era deloc ca *Express 3*, așa cum este acum. Cuvântul *„Express”* poate fi singurul lucru pe care îl au în comun.

Sencha a lansat proiectul open source *Connect* în iunie 2010 pentru a aborda lipsa de flexibilitate și extensie a *API-ului NodeJS*. Interfața serverului web *Ruby* Rack a servit drept inspirație pentru proiect. Tim Caswell, un angajat Sencha, și T.J. Holowaychuk a fost îndemnat să preia conducerea proiectului. Similar cu *Express, Connect* se baza *pe API-ul NodeJS* și avea o arhitectură *middleware* care permitea conectarea unor aplicații minuscule, reutilizabile, pentru a efectua operațiuni specifice *HTTP*. Conectarea acestor servicii a oferit capacitatea suplimentară cerută de aplicațiile web construite de *NodeJS*.

Capacitatea de a crea propriul middleware pentru aplicații a fost, de asemenea, disponibilă pentru toată lumea. Flexibilitatea și extensibilitatea *API-ului NodeJS* au fost îmbunătățite considerabil de *plugin*. Până în acest moment, *NodeJS* avea două framework-uri distincte de dezvoltare web. Ca urmare, comunitatea *NodeJS* a devenit confuză, mai ales că Holowaychuk a

fost implicat în ambele proiecte. Mai târziu a devenit clar că *Connect* și *Express* sunt de fapt cadre complementare. Pentru a fuziona *Express* cu *Connect*, Holowaychuk a luat decizia de a reproiecta arhitectura *Express* în iulie 2010. Ca urmare, Express v1.0.0 este o nouă iterație a *Express* care combină *Connect* cu *Express*. Acum este clar ce cadru de dezvoltare web ar trebui utilizat pentru aplicațiile *NodeJS* datorită *Express v1.0.0*. Cu capabilități suplimentare adăugate pe lângă *Express*, a devenit *Connect*. *Express* continuă să folosească *middleware*-ul *Connect* în acest moment, iar orice modificări aduse *Connect* sunt întotdeauna încorporate în *Express*.[14]

Aceasta este povestea creării *Express* și a uniunii dintre *Connect* și *Express*.

Îmbunătățiri dintre Node si Express

În *NodeJS*, o aplicație este construită în întregime din o singura metoda JS. Metoda ține cont de solicitările venite de la navigatoarele web, aplicații mobile și alți clienți care interacționează cu *server*-ul. Această funcție va examina o solicitare după ce o primește pentru a decide cum să reacționeze. De exemplu, dacă folosim un navigator web pentru a vizita pagina de pornire, această metodă va recunoaște ca avem nevoie de una dintre pagini si ne-o va oferi ca răspuns.[16]

De exemplu, o aplicație web simplă care afișează ora server-ului utilizatorilor va funcționa după cum urmează: Programul va întoarce un document *HTML* cu ora dacă clientul solicită pagina de pornire. O eroare 404 „Pagina nu există” va fi returnată de program dacă clientul face o cerere diferită. Fluxul de cereri către server ar apărea ca în *figura următoare* dacă ar fi să creăm aplicația folosind pur și simplu *NodeJS* și nu *Express*.

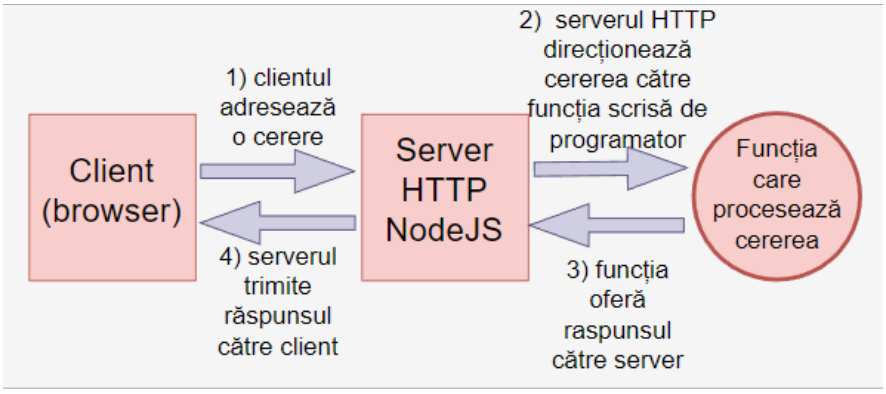

Figura 2-5

Un *handler* de solicitare este funcția *JavaScript* care răspunde la solicitările navigatorului. O metoda JS primește *request*-ul, o gestionează iar apoi dă un răspuns. *Metoda* corespunzătoare

este gestionată de serverul *HTTP NodeJS*, salvând programatorul de a se ocupa de protocoale de rețea dificile. Cei doi parametri pe care îi ia funcția de gestionare a cererii în cod este obiectul cererii si cel al răspunsului. Caracteristica de administrare a programului care a fost afișată anterior poate verifica adresa URL pe care a solicitat-o clientul.

Există un singur *handler* de cereri care reacționează la solicitări în fiecare aplicație *NodeJS*. Este destul de simplu din punct de vedere conceptual. Node poate fi provocator chiar si pentru cei mai experimentați. Va fi nevoie de circa 50 de instrucțiuni pentru a transfera o singură imagine *JPEG*. Deși serverul este robust, este deficitar în multe domenii pe care le putem folosi pentru a dezvolta aplicații practice.

Express a fost dezvoltat special pentru a simplifica crearea de aplicații *web* cu *NodeJS*.

În general, *Express* îmbunătățește serverul *HTTP NodeJS* cu două caracteristici cruciale:

Primul avantaj este că simplifică serverul *HTTP NodeJS* prin adăugarea unui număr de capabilități practice. De exemplu, transmiterea unui singur fișier *JPEG* e suficient de complicată cu *Node*. *Poate fi folosit Express si doar o singura instrucțiune.*

Dimpotrivă, *Express* permite să împărțim funcția cuprinzătoare de procesare a cererilor văzută într-un proiect tipic NodeJS în mai multe funcții mai mici, fiecare dintre ele gestionând un aspect diferit.[16]

Figura 2.6 ilustrează fluxul de solicitări într-o aplicație *Express*, spre deosebire *de figura 2.5*, care a demonstrat cum o solicitare ar merge către server dacă aplicația ar fi creată doar folosind *NodeJS* și fără *Express*. Deși schița poate părea prea complexă, este de fapt o metodă mult mai simplă din punctul de vedere al programatorului.

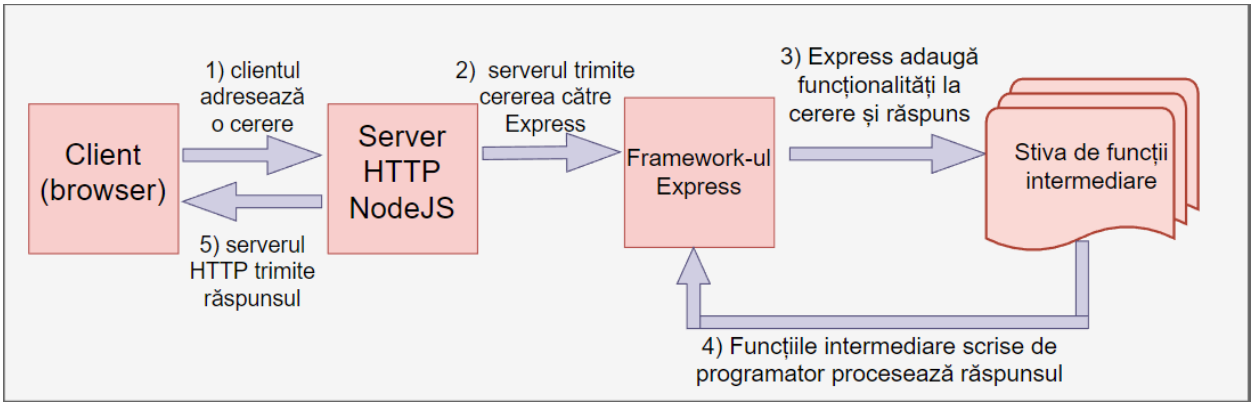

Figura 2-6

Codul identic furnizat în topicul despre *Node* pentru a stabili serverul web în *Express* ar arăta după cum urmează:

```
var express = require('express');
var app = express();
app.get('/', function(req : … , res : Response<ResBody, Locals> ) {res.send( body: 200,'Hello World');});
app.listen( port: 8124,  hostname: "127.0.0.1",  backlog: function ()
{console.log('Server running athttp://127.0.0.1:8124/');
});
```

Express, pe de altă parte, oferă o structură de cod mai bună și sprijină programatorul prin împărțirea funcțiilor pe module. Fata de Node nu are mai multe funcții.

2.2.3 MongoDB

Relaționarea construită pe *Structured Query Language* este utilizată de obicei pentru durabilitatea datelor în aplicațiile online (*SQL*). Această metodă oferă multor aplicații o soluție pentru conexiunilor dintre aplicații si diferite colecții. Spre deosebire de obiectele adaptabile, semi-structurate ale *JavaScript*, SQL este organizat pe scheme, mod explicit tipul acestora. Lucrul la structurile datelor între serverul web și baza de date poate fi uneori o provocare folosind această strategie. Serializarea și deserializarea dintre modele si structuri controlează interacțiunea datelor intre aplicație și bazele de date *SQL*.

Diferența de natură a datelor când datele sunt stocate in grupările bazei și modul în care sunt interpretate de serverul web poate duce la schimbări neintenționate de context între numeroasele limbi specifice domeniului (*DSL - domain specific languages*) utilizate pentru a interoga aceste date. Când utilizați o aplicație online mare sau complicată, acest lucru poate duce ocazional la erori și iritații neprevăzute. Cea mai populară metodă de a menține legătura dintre o interfața web si baza acestuia fără Express este să se bazeze pe o bibliotecă de cartografiere relațională a unui obiect(*ORM*). Deși schița poate părea prea complicată, din perspectiva codificatorului, este o modalitate mult mai ușoară. Pentru a lucra cu date din aplicații online, mulți dezvoltatori au optat pentru sisteme *NoSQL*.

Fiind simplu de utilizat. O sa trecem prin câteva dintre metodologiile ale *MongoDB*:

1. Documentul e tipul fundamental de date pentru *MongoDB*
2. O grupare se poate comparată cu un tablou care are componenta schemei.
3. Mai multe instanțe MongoDB pot fi găzduite pe server.
4. Orice cheie distinctă numită „_id” e exclusivă pentru doar o singură grupare.

Înregistrare/Document MongoDB

Documentul, care constă dintr-o listă ordonată de chei și valori aferente, este fundamentul *MongoDB*. Limbajele de programare diferă în ceea ce privește modul în care sunt reprezentate *documentele*, dar cele mai multe dintre ele au o structură care este o potrivire naturală. De exemplu, *documentele* sunt inserate ca *objects* în *JavaScript*: {"*întâmpinare*" : "Bună ziua!"}

Singura cheie din acest text simplu, „întâmpinare”, are valoarea „Bună ziua!”. Majoritatea documentelor o sa fie mai sofisticate ca aceasta, cu multiple combinații cheie/valoare, cum ar fi {"*întâmpinare*" : "Bună ziua!", "vizualizări" : 3}.

Valorile documentului pot include o multitudine de tipuri pana la un întreg fișier. În această ilustrație, „*întâmpinare*” are valoarea unui șir de caractere, dar „vizualizări” are o valoare întreagă.

Șirurile servesc drept chei pentru document. O cheie poate include orice caracter *UTF*-8, cu următoarele excepții semnificative: Cheile nu pot avea caracterul „\0” în ele (caracterul nul). Sfârșitul unei *chei* este reprezentat de acest caracter. Folosim *simbolurile* „$” și „.” doar în situații specifice deoarece au proprietăți unice. Ele ar trebui să fie considerate în general rezervate. *MongoDB* ține cont atât de tip cât și de diferența dintre *case sensitive*.

Un alt punct crucial de reținut este că *documentele* nu au voie să aibă *chei cu duplicat*. Putem vedea un prim exemplu: {"*întâmpinare*" : " Bună ziua!", " *întâmpinare* " : " Bună seara!"} [18]

Collections / Modele MongoDB

O grupare poate fi asemănata cu un tablou, dacă un document este echivalentul bazei de date relaționale în *MongoDB* al unui rând.

Schemele dinamice sunt văzute în colecții. Aceasta implică faptul că ar putea exista mai multe „forme” pentru o singură colecție de documente. De exemplu, ambele lucrări enumerate mai jos sunt păstrate astfel in aceeași grupare: {"întâmpinare" : "Bună ziua!", "vizualizări" : 5}
{"ieșire": "noapte bună"}

Lucrările menționate, includ diverse ”number keys”, „keys”, valori complet diferite. De ce avem nevoie de colecții diferite când orice document poate fi adăugat la orice colecție este o întrebare validă. Este necesar sa avem modele diferite pentru tipuri diferite de colecții? Următoarele sunt cauzele:

Obținerea unei liste de *colecții* este substanțial mai rapidă decât obținerea unei liste de tipuri de documente dintr-o colecție. Dacă fiecare document ar conține un câmp „tip” care indica dacă e „mic”, „mediu” „mare”, de exemplu, ar putea dura foarte mult pentru a localiza acea valoare pentru a crea colecții total nerelatate.

Localizarea datelor este posibilă prin plasarea in același loc a documentelor de același tip. Va fi nevoie de mai puține căutări pe disc pentru a obține un număr de postări de blog dintr-o colecție care pur și simplu este compusa din postări, decât de a face acest lucru de la una care include și date despre autor.

Când stabilim index-urile , de abia din acel moment avem începutul unei structuri pentru documentele noastre, (Acest lucru este deosebit de precis pentru index-uri diferite.) Indicii colecției sunt specificați acolo. Ne putem indexa mai eficient colecțiile dacă includem doar documente de același fel în fiecare colecție. [18]

Numele unei colecții este folosit pentru a o identifica. Cu următoarele excepții, numele colecției poate fi orice șir *UTF*-8: Numele colecției "" nu poate fi un șir gol (""). Caracterul nul, „0”, care marchează sfârșitul unui nume de colecție, nu poate fi folosit în numele colecțiilor. System., un prefix desemnat pentru colecțiile interne, nu trebuie utilizat pentru denumirea colecțiilor. De exemplu, *system* fiind una din colecții care oferă detalii despre toate colecțiile de baze de date, în timp ce colecția *system.users* conține utilizatori de baze de date. În plus, litera *rezervată* „$” nu ar trebui să fie folosită în numele colecțiilor create de utilizator.

Baze de date

Colecțiile sunt, de asemenea, grupate în baze de date de către *MongoDB*, pe lângă *documente*.

Un singur server Mongo, poate stoca si tine mai multe colecții. Pe un singur server *Mongo* ,mai multe baze de date pot fi benefice pentru stocarea datelor pentru diverși utilizatori sau aplicații.

Un nume complet de colecție, sau *namespace*, poate fi obținut prin concatenarea numelui unei baze de date cu numele unei *colecții*. *Namespace-ul* unei *colecții*, de exemplu *com.reviews.post* dacă am folosi colecția review.post din com, com fiind baza de date. Deși au adesea mai puțin de 100 de octeți lungime.

Cheița "_id"

O cheița „_id” este mandatorie pentru fiecare document. Valoarea cheii „_id” este de regula *ObjectId* in schimb se poate utiliza orice tip. Fiecare document dintr-o colecție este necesar ca sa fie cu totul alta si fără duplicat pentru „_id” pentru a vă asigura că fiecare poate fi recunoscut separat. O singura colecție poate include un document cu valoarea „_id” 33. În el poate fi prezent, totuși, un singur document cu „_id” de 33.

Tipul implicit al funcției “_id” este *ObjectId*. Clasa *ObjectId* este destinată să producă o singură valoare la nivel mondial pe mai multe computere. Ar fi o provocare să sincronizați cheile primare cu autoincrementare pe diferite servere datorită arhitecturii distribuite a *Mongo,* care este justificarea fundamentală pentru utilizarea *ObjectId*. Mongo a fost proiectat pentru un mediu distribuit, si a avea identități diferite și unice este crucial.

Un șir de 24 de cifre hexazecimale sau două cifre hexazecimale pentru fiecare octet este utilizat pentru a reprezenta un *id*, care ocupă 12 octeți de memorie. Șirul pentru un *ObjectId* este într-adevăr de două ori mai lung decât datele pe care le reprezintă, în ciuda faptului că este adesea afișat ca un șir hexazecimal masiv. Putem observa că fiecare dată de timp afectează doar ultimele câteva numere dacă creăm rapid un număr de *ObjectId*-uri noi.

Centrul cifrelor *ObjectId-ului* se va schimba, de asemenea, doar dacă micșorăm, ceea ce va duce la crearea acestor cifre în câteva secunde. *ObjectId*-urile sunt formate în așa fel încât acesta să fie cazul.

Din acest motiv, până la 2563 (16.777.216) obiecte distincte pot fi create într-o singură secundă. O cheie „_id” va fi adăugată documentului inserat dacă nu există una dacă introduceți un document fără unul. [18]

Mongo este un DBMS open-source, care are la baza colecții , creat pentru baze de date si aplicații robuste si online.

2.3 Componente pentru dezvoltare si PIF

Am folosit *WebStorm IDE* ca mediu de dezvoltare pentru a crea aplicația. Gestionarea versiunilor este simplificată datorită integrării cu contul *Github*.

După intrarea sa în *repository-ul* Github, fiecare actualizare a codului se face printr-un webhook. Atlas a fost folosit pentru a încarcă in cloud colecția, si pentru un plus de securitate.

Aici, vom oferi o privire de ansamblu rapidă a tehnologiilor menționate anterior.

2.3.1 Modulul de control

Software-ul care vă arată fiecare versiune pentru orice document dintr-o aplicație, cu un marcaj al timpului în care fiecare versiune a fost generată, precum și autorul modificărilor, cunoscut ca si *VCS* sau ca fișier sursă.

După cum sugerează numele, *GitHub* se bazează pe Git. *Git* este unul dintre numeroasele sisteme de control al versiunilor gratuite și open source (există altele, cum ar fi *Subversion*), așa că oricine îl poate folosi.

Repository-urile Git pot fi găsite pe *GitHub*. Poate fi folosit atât pentru a face backup pentru cod, cât și pentru a permite altor programatori să lucreze împreună la el, ceea ce este atât de benefic. *GitHub* este bazat pe *Git* oferind aceleași funcții.

Git operează mai degrabă pe instantanee decât pe diferențe. Prin urmare, nu ține evidența diferențelor intre 2 instanțe ale unui document, ci face un instantaneu al proiectului așa cum este acum. Prin urmare, *Git* este mult mai rapid decât alte *VCS-uri* distribuite din această cauză. *Git* este destul de diferit de un control al versiunilor centralizat la nivel de sistem. *Git* este un *VCS* distribuit, ceea ce înseamnă că fiecare utilizator are un *repository* cu istoricul acestuia. Acest lucru elimină nevoia de comunicare cu un server central. Ca rezultat, făcându-se schimbări local ,schimbului path-urilor si *seturi* în funcție de diferențe. Se face un *hash*(suma) pentru fiecare instantaneu pe care îl creează, permițându-i să determine ce fișiere s-au modificat prin compararea sumelor de control. Ca urmare, *Git* poate monitoriza rapid modificările din interiorul fișierelor și directoarelor și, de asemenea, poate verifica dacă fișierele au fost deteriorate. Mecanismul „cu trei stări” este componenta principală a sistemului *Git*.

Singura activitate la care lucrăm acum este instantaneul din directorul de lucru.

Fișierele modificate sunt notate în cea mai recentă versiune în zona de înregistrare, făcându-le gata pentru a fi înregistrate în baza de date.

Baza de date în care se păstrează istoricul este directorul git. *Git* funcționează în principal după cum urmează: după ce facem modificări la fișiere, adăugăm imediat orice document pe care îl vrem sa-l includem în baza de date in locul de „staging” (*git add*) (*git commit*). În scopuri terminologice ne referim la „*commit*”. Un fișier se duce astfel „*modificat*” , „în *etapizat*” , „*committed*”. [19]

2.3.2 Mutarea bazei Mongo in Cloud

Platforma (*DBaaS*) *MongoDB Atlas* a fost a fost dezvoltată ce aceeași dezvoltatori care au proiectat *MongoDB.*

Atlas va oferi instanței tot ce avem nevoie după construirea bazei de date și vom oferi instanței următoarele:

Un cloud privat virtual (*VPC*) este utilizat pentru a izola rețeaua prin implementarea instanțelor de baze de date cu măsuri de securitate automate. Printre măsurile de securitate suplimentare se numără liste de adrese *IP* care au acces la bază, autentificare în curs de desfășurare, criptare în tranzit și în repaus, acces complex de management bazat pe roluri și alte caracteristici de securitate.

Servere suplimentare sunt oferite de către platforma , pentru a fi mereu la îndemâna continuă, asigurând că aplicația continuă să funcționeze pana când chiar si cel principal e offline.

Sistemul *MongoDB* Atlas utilizează copii de rezervă și recuperare la un moment dat pentru a se proteja cu fermitate împotriva coruperii datelor, care se poate întâmpla ca urmare a unei greșeli umane sau virtual. Aceasta se numește replicare pre-făcută.

Date care sunt aranjate într-o varietate de moduri pe care le putem folosi pentru a determina când este timpul să trecem lucrurile la următorul nivel (pot fi furnizate mai multe cazuri pe măsură ce numărul utilizatorilor crește), aceasta se numește monitorizare.

Atlas folosește o metodă cunoscută sub numele de sharding pentru a permite scalarea orizontală pentru bazele de date. Datele sunt dispersate pe fragmente, care sunt colecții de replici. In funcție de volumul datelor care crește, echilibrarea automată în *MongoDB* garantează că simt distribuite uniform datele. Fără a crește complexitatea aplicației, shardingul permite instalațiilor *Mongo sa crească* dincolo de constrângerile de a fi doar un singur server.

AWS, Azure sau Google Cloud pot fi baza pentru serviciul *MongoDB Atlas.*

Serviciul va fi implementat, rulat folosind Mongo. Vom folosi *Amazon Web Services (AWS)* pentru proiectul curent pentru a găzdui baza de date cât mai aproape posibil fata de locația noastră curenta.

Baza de date o sa ruleze pe o *VM(Virtual Machine)* care in versiunea de baza poate stoca pana la 500 de colecții si conexiuni in același timp.

3. Modul in care funcționează aplicația(Descriere)

3.1 Descriere a utilității practice

Acum există o mare diferență în modul în care destinațiile bine-cunoscute sunt promovate pentru turism în comparație cu destinațiile mai puțin populare, cu potențial ridicat de vizitatori. Așadar, ne aflăm într-o situație care, din exterior privind înăuntru, nu pare să aibă o ieșire: vizitatorii sunt extrem de mulți in locațiile deja cunoscute, unde cererea de servicii HoReCa(Hoteluri, Restaurante, Cafenele) depășește cel mai adesea oferta, iar locațiile cu cel puțin același potențial rămâne necunoscut vizitatorilor, si descurajează chiar pana investitorii să investească în regiunile corespunzătoare.

Două lucruri se pot întâmpla unui loc cu potențial turistic:

Prima metodă, care este mai puțin răspândită în țara noastră, este realizarea unei investiții considerabile într-o regiune cu potențial turistic puternic pentru a oferi facilități esențiale pentru potențialii vizitatori, urmată de campanii de marketing pentru regiunea în cauză. Toate sugerează, de asemenea, costuri semnificative care ar putea deveni neamortizabile în curând.

A doua metodă, care este folosită mai frecvent în practică, presupune o dezvoltare treptat, ciclic, cu vizitatorii care preiau majoritatea rolurilor. În stadiul inițial, oamenii sunt atrași de o anumită regiune din cauza poziției sale fizice, a peisajului său, unele aspecte care sunt bune pentru sănătate (calitatea aerului, izvoarele termale etc.). La început, sunt dacă poți atrage în rulote, localnicii vor începe să ofere cazare și in cele din urmă, investitorii încep să apară în zonele corespunzătoare.

Un preț aproximativ pentru servicii, conținut, componente vizuale (imagini) și locația locului menționate pot fi adăugate folosind site-ul. Acestea vor fi afișate pe mapă, făcându-le mai ușor de identificat din perspectivă geografică și ajutând la crearea unei imagini cuprinzătoare a regiunii discutate.

Platforma creată poate fi utilizată și ca instrument de marketing de către persoanele care doresc să facă publicitate unei anumite zone. Atrăgând atenția publicului larg sau atrăgând investitorii către regiunile de interes relevante, aceștia pot oferi beneficii financiare zonelor cu potențial turistic, scutindu-le, în același timp, de a plăti pentru eforturile de publicitate.

3.2 Proiectarea si Implementarea

3.2.1 Proiectarea si Implementarea(MongoDB)

Sa readucem in discuție ideile fundamentale *ale Mongo atunci când am prezentat Mongo si NoSQL*:

1. Tipul fundamental de date al *MongoDB* este documentul, care corespunde aproximativ unui rând dintr-un sistem de management al bazelor de date relaționale (dar mult mai expresiv).
2. O colecție poate fi considerată ca omologul bazei de date relaționale a unui tabel. Colecțiile sunt folosite pentru organizarea documentelor.
3. O instanță *MongoDB* poate suporta numeroase baze de date separate, fiecare cu propriile sale colecții.
4. Cheia specială „*_id*”, care este exclusivă într-o colecție, este prezentă în fiecare document.

Trei colecții alcătuiesc baza de date concepută pentru aplicația curentă.

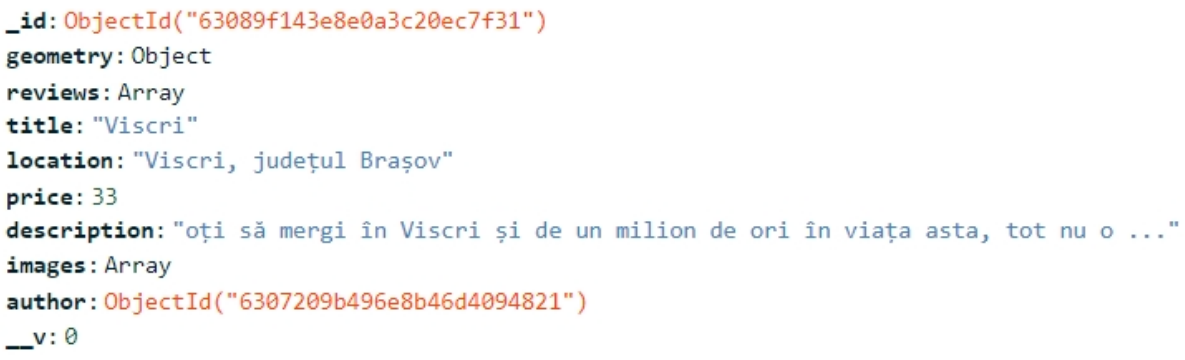

Figura 3-1(Colectia „Locului” in baza de date)

Documentul are un câmp *_id* de tip *ObjectId* ca prim *câmp*. Acest *câmp* special la nivel de colecție ajută la localizarea acelui document în interiorul colecției. Următoarele cinci *câmpuri*, care sunt *titlu, fotografii, descrieri, prețuri, locații și autori*, sunt aceleași cu cele din formularul de înregistrare a locului.

Generarea automată a câmpului *geometry*. Atunci când un utilizator furnizează un șir de caractere pentru a reprezenta o locație pe care a vizitat-o (în acest exemplu, „Viscri, județul Brașov”), *API*-ul *MapBox* prelucrează șirul de caractere și returnează coordonatele care trebuie introduse în câmpul de geometrie. Aceste coordonate vor fi folosite pentru a afișa acel loc pe mapă în viitor.

Câmpul Reviews, fiind un *vector de ObjectID* în care sunt stocate *id*-urile comentariilor locației, aparțin unei colecții diferite care conține exclusiv comentarii. După cum se vede în următorul instantaneu, comentariul care este păstrat in vector pe primul sau loc:


```
_id: ObjectId("6308a1c73e8e0a3c20ec7f33")
rating: 4
body: "Ne-a placut enorm de tare acest loc!"
author: ObjectId("6307209b496e8b46d4094821")
__v: 0
```


Figura 3-2(Colectia „Review-ului” in baza de date)

După cum putem vedea, id-ul *comentariului*, „6308a1c73e8e0a3c20ec7f33” este *același* cu documentul precedent, creând o conexiune intre fișiere.

Câmpul „author” care este inclus atât in stocarea locațiilor si in cel al „*review*-urilor”. Acesta este perceput ca si un obiect, acesta conține atât și id-ul colecției de users. In prezentul caz *id*-urile corespund.

La crearea unui cont nou, un document este pus în ultima colecție, colecția utilizatorilor. Următoarele câmpuri sunt prezente în document pentru un utilizator înregistrat:


```
_id: ObjectId("6307209b496e8b46d4094821")
isAdmin: true
email: "kh3ops97@geeemail.com"
username: "secret"
salt: "c287e9b7c7d304eb4f87628bad355edd9f575368bba6dfcac0f2c9ac724f40cb"
hash: "e5f74c39dfd9ae0e4ecddb788ffdde862921bc815cd79feb5608dbadf6c93efe5e1e79..."
__v: 0
```

Figura 3-3(Colectia User-ului in baza de date)

„*ID-ul*” fiind primul parametru, este distinct pentru fiecare utilizator, cum am văzut deja legătura între lucrările din colecțiile anterioare.

Al doilea parametru „*salt*”; conține aleatoriu o valoare care este folosit pentru a cripta parola înainte de a fi introdusa. În secțiunea privind problemele legate de securitate, vom aprofunda acest lucru.

Câmpul „*hash*” (*amprenta digitală*) a parolei, care a fost produsă printr-o funcție unidirecțională, se află în câmpul următor.

Figura 3.4 ilustrează cum Mongo comunică cu celelalte componente ale programului.

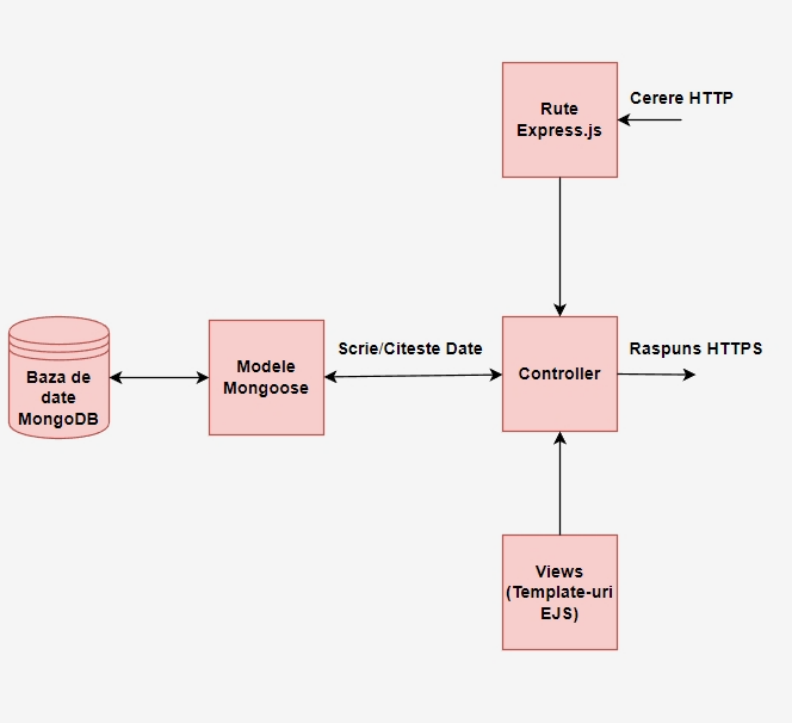

Figura 3.4(Proiectarea arhitecturii)

Mongoose(Modelele concepute)

Mongoose este o bibliotecă de modelare a datelor ODM amintita mai devreme in *Figura 2.1* atât în Node cat și Mongo. Oferă scheme de validare, gestionează asocierile de date și tranzițiile între obiectele si scheme lor.

Modelele pot fi considerate constructori creați din definiții ale schemelor. Un document este o anumită instanță a unui model. Responsabilitatea modelelor este citirea/scrierea datelor in baza de date Mongo.

Ar trebui să le separăm pentru fiecare tip de „*obiect*” în timp ce ne creăm modelele, este doar logic (informații legate).

Locațiile, comentariile și utilizatorii sunt posibilitățile evidente în situația noastră pentru aceste modele.

Pentru a ne ajuta să introducem locații în baza de date, am inclus următorul model:

```
const Schema = mongoose.Schema;
const PlaceSchema = new Schema({
    title: String,
    images: [ImageSchema],
    geometry: {
        type: {
            type: String, // don't do location {type : String}
            enum: ['Point'], // location.type must be 'Point'
            required: true
        },
        coordinates: {
            type: [Number],
            required: true
        }
    },
    price: Number,
    description: String,
    location: String,
    author: {
        type: Schema.Types.ObjectId,
        ref: 'User'
    },
    reviews: [
        {
            type: Schema.Types.ObjectId,
            ref: 'Review'
        }
    ]
},opts);
module.exports = mongoose.model( name: 'Place', PlaceSchema);
```

Figura 3-5(Modelul Mongoose al Locului)

3.2.2 Arhitectura aplicației

Modelele și metodele utilizate în proiectarea și construcția unei aplicații sunt descrise în arhitectura aplicației. Pentru a crea o aplicație care este bine structurată și simplu de întreținut, este necesara o arhitectura minimalista.

Figura 3.4 arata arhitectura aplicației, aceasta folosește o serie de concepte și idei, in care implementarea va fi discutata pe scurt în paragrafele care urmează.

Am acoperit anterior elementele fundamentale ale bazei de date (*MongoDB* și *Mongoose*), așa că vom trece la *controller*, *views* și *rute*.

Controller

Un *controler*, în general, este un grup de funcții care separă *codul* destinat direcționării cererilor de codul destinat executării respectivelor cereri.

Colecția de funcții care compun controlerul primește solicitări de la modele, produce o pagină cu informațiile necesare si apoi trimite pagina si conținutul acesteia către client.

Figura 3-6(Organizarea Controllere-lor)

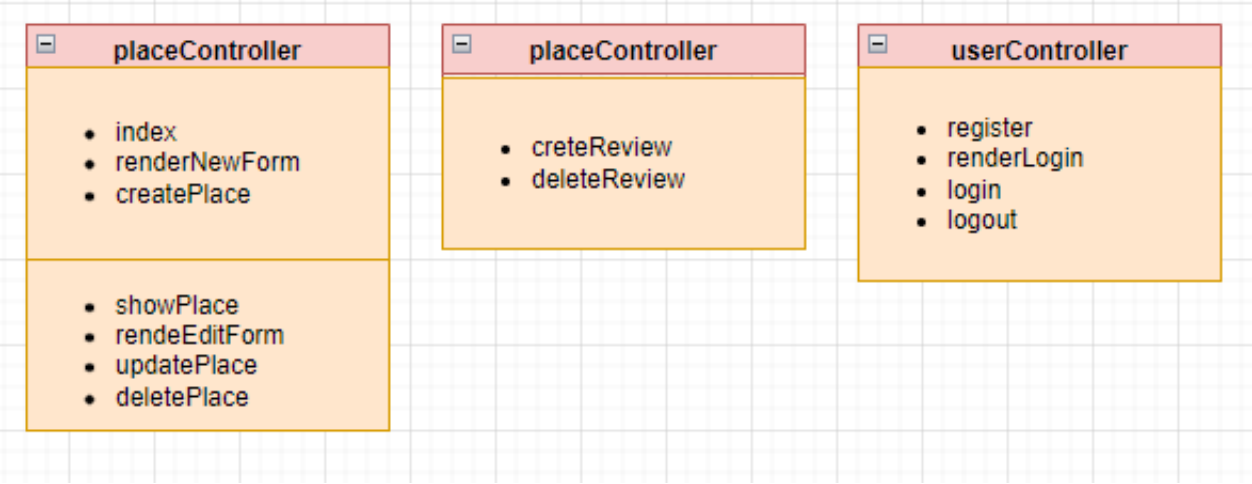

Figura 3-7(Functiile organizate per modul)

EJS Vizualizările(Componentele Template)

Este prescurtare pentru *Embedded JS Template* folosit in *Node*. Folosind cel mai minimal cod posibil, „*Template Engine*” ajută la crearea de șabloane *HTML*. Pe lângă producerea *HTML*-

ului final, poate adaugă date în template *HTML* de pe partea de frontend. Folosind *JavaScript*, marcajul este produs folosind un limbaj de șabloane simplu ca si EJS.

Opusul este și adevărat, deoarece paginile *HTML* pot beneficia și de capacitatea de a încorpora *JavaScript*. Următoarea imagine reprezintă un model de view care este aplicat la vizualizarea locului:

```
<% layout('layouts/boilerplate')%>
<link rel="stylesheet" href="/stylesheets/stars.css">

<div class="row">
    <div class="col-6">
        <div id='map' style='...'></div>
        <div id="placeCarousel" class="carousel slide" data-ride="carousel">
            <div class="carousel-inner">
                <% place.images.forEach((img, i) => {  %>
                    <div class="carousel-item <%= i === 0 ? 'active' : ''%>">
                        <img src="<%= img.url%>" class="d-block w-100" alt="">
                    </div>
                <% }) %>

            </div>
            <% if(place.images.length > 1) {%>
                <a class="carousel-control-prev" href="#placeCarousel" role="button" data-slide="prev">
                    <span class="carousel-control-prev-icon" aria-hidden="true"></span>
                    <span class="sr-only">Previous</span>
                </a>
                <a class="carousel-control-next" href="#placeCarousel" role="button" data-slide="next">
                    <span class="carousel-control-next-icon" aria-hidden="true"></span>
                    <span class="sr-only">Next</span>
                </a>
            <% } %>
        </div>

        <div class="card mb-3">
            <div class="card-body">
                <h5 class="card-title"><%= place.title%></h5>
                <p class="card-text"><%= place.description%></p>
            </div>
            <ul class="list-group list-group-flush">
                <li class="list-group-item text-muted"><%= place.location%></li>
                <li class="list-group-item">Submitted by <%= place.author.username%></li>
                <li class="list-group-item">$<%= place.price%>/visit</li>
            </ul>
            <%  if( currentUser && place.author.equals(currentUser._id) || currentUser && currentUser.isAdmin)  {%>
            <div class="card-body">
                <a class="card-link btn btn-info" href="/places/<%=place._id%>/edit">Edit</a>
                <form class="d-inline" action="/places/<%=place._id%>?_method=DELETE" method="POST">
```

Figura 3-8(Exemplu EJS view prezent in pagina de visualizare a locului)

Rutare

URI-urile unui site care reacționează pentru interogările utilizatorului este denumit *rutare*. *Rutarea* se realizează folosind metode obiectului aplicației *Express* care sunt echivalente cu metodele *HTTP*, cum ar fi app.get() pentru cererile „*GET*” și *app.post*() pentru cererile „*POST*”.

Când o aplicație primește o solicitare specifică metodei *HTTP* către ruta dată (punctul final), funcția call-back definită de aceste metode de rutare - denumită adesea „*handler*” - este invocată. Cu alte cuvinte, software-ul „ascultă” în esență request-urile care se mulează pe anumite rute, iar in momentul in care detectează o potrivire si rulează funcția necesara.

De exemplu, ruta unei solicitări „*GET*” pentru pagina principală cu mesajul „bună ziua” arată astfel:

```
app.get('/', function (req :… , res :Response<ResBody, Locals> ) {
    res.send('salut')
})
```

Rutele sunt fracționate in 3 documente pentru aplicația construită pentru a permite întreținerea.

Un prim fișier conține rutele utilizate pentru acțiunile de locație (*vederea*, *adăugarea*, remodelarea, *ștergerea*). Acest lucru a fost realizat dezactivând rutele pentru *review*-uri, care sunt stocate în alt document, dar si cele care se ocupa de *autentificarea user*-ului, care sunt, de asemenea, adunate într-un alt fișier.

3.2.3 MapBox(Map Cluster)

Motivația folosirii unui MapCluster

Trebuie să selectăm cea mai bună opțiune pentru a facilita localizarea destinațiilor turistice, deoarece scopul aplicației noastre este de a promova acele locații. Când un utilizator introduce un loc nou în baza de date, inițial m-am gândit la potențialul de a adăuga un câmp obligatoriu cu coordonatele geografice. Deși această metodă este mai ușor de dezvoltat, poate opri mai mulți oameni, deoarece este puțin tehnică și nu oferă o experiență plăcută utilizatorului.

Datorită acestor factori și a faptului că informațiile pe care le trimite creierul uman sunt imagini și se executa mult mai repede decât scrisul, am decis sa cream o soluție în care locurile

sunt prezente pe o mapă folosind doar informațiile introduse de utilizatorul în câmpul desemnat numelui locației.

Un factor important care afectează direct cât de ușor de utilizat este un program este cât de familiari sunt utilizatorii cu diferitele componente ale UI. Experiența utilizatorului este foarte influențată de familiaritate. Interfețele zilnice într-adevăr sunt obișnuite pentru noi încât le utilizam fără a ne gândi prea mult la asta.

Capacitățile de „*Map Cluster*” încorporate ale *Mapbox GL JS* sunt utilizate în acest exemplu pentru a descrie punctele dintr-un „circle layer” ca si grupuri.

Figura 3-9(Exemplu MapBox Cluster)

API key Mapbox

In blocurile de cod ce urmeaza vom ilustra modul in care functioneaza gruparea locatiilor. Pentru a putea folosii *API*-ul *Mapbox*, este necesara integrarea in aplicatie a unei „*cheite publice*” asociate contului creat.

```
// Intregrarea cheitei publice MapBox
mapboxgl.accessToken = 'pk.eyJ1IjoidG9ueWRuOTgiLCJhIjoiY2w3NHc1MngxMGVkcTNudGhjYm4zZDc2biJ9.Bn5sLOhN4xk-jRJfhYdQVA';

const map = new mapboxgl.Map({
    container: 'map', // container-ul in care se va afisa;
    style: 'mapbox://styles/mapbox/light-v10',
    center: [-103.59179687498357, 40.66995747013945], // Coordonatele de start ale mapei;
    zoom: 3 // Zoom-ul initial
});

map.addControl(new mapboxgl.NavigationControl()); // Control-ul navigarii ;
map.on('load', function () {
    // Add a new source from our GeoJSON data and
    map.addSource('places', {
        type: 'geojson',
        // Point to GeoJSON data;
        data: places,
        cluster: true,
        clusterMaxZoom: 14, // Max zoom to cluster points on
        clusterRadius: 50 // Radius of each cluster when clustering points (defaults to 50)
    });
```

Figura 3-10(Integrarea cheitei publice si crearea unei instante)

Pentru început avem nevoie sa asociem constantei „*accessToken*”, cheița publica pe care o găsim in contul de utilizator. Se va crea o noua instanța de tip „*mapboxgl.Map*” in care se va furniza „container-ul *HTML*” care va afișa harta, coordonatele „*center*” sunt coordonatele de start a mapei la prima încărcare. Coordonatele locațiilor se parseaza prin intermediul câmpului „*geometry*” amintit in modelele noastre. Totodată asocierea locațiilor mapei prin intermediul GeoJSON.

Layer-ul care definește mărimea și culoarea grupării

În layer-ul din *Figura 3.11*, este definită mărimea și culoarea „circle layer-ului” în funcție de numărul de locații din acea zonă. Grupul va avea culoarea albastră și mărimea de 20px când sunt mai puțin de 100 locații, culoarea galbena și mărimea de 30px atunci când sunt intre 100 și 750 de locații, și culoarea roz și mărimea de 40px când sunt mai mult de 750 locații. Bineînțeles toate aceste valori precum culoarea și mărimea se pot modifica și edita după bunul plac.

```
map.addLayer({
    id: 'clusters',
    type: 'circle',
    source: 'places',
    filter: ['has', 'point_count'],
    paint: {
        /*
        * Schimbarea culorii grupurilor de locatii
        * Albastru ,20px, cand punctele sunt mai putin de 100
        * Galben ,30px, cand punctele sunt intre 100 si 750
        * Roz ,40px, cand punctele sunt mai mult de 750
        * */
        'circle-color': [
            'step', ['get', 'point_count'], '#00BCD4',
            10,  '#2196F3', 30, '#3F51B5'],
        'circle-radius': [
            'step',
            ['get', 'point_count'],
            15, 10, 20, 30, 25]}});
```

Figura 3.11(Layer-ul mărimii și culorii al grupării)

Layerele locațiilor grupate și negrupate

Layerele ce urmează în *Figura 3-11*, definesc locațiile grupate și negrupate, evident cele grupate vor avea ca instanța numărul acestora, iar cele „*stand alone*” un singur „*point-location*”.

```
// Layer-ul numarului locatiilor grupate;
map.addLayer({
    id: 'cluster-count',
    type: 'symbol',
    source: 'places',
    filter: ['has', 'point_count'],
    layout: {
        'text-field': '{point_count_abbreviated}',
        'text-font': ['DIN Offc Pro Medium', 'Arial Unicode MS Bold'],
        'text-size': 12
    }
});

// Layer-ul locatiilor negrupate;
map.addLayer({
    id: 'unclustered-point',
    type: 'circle',
    source: 'places',
    filter: ['!', ['has', 'point_count']],
    paint: {
        'circle-color': '#11b4da',
        'circle-radius': 4,
        'circle-stroke-width': 1,
        'circle-stroke-color': '#fff'
    }
});
```

Figura 3.12(Layer-ul locațiilor grupate si negrupate)

Point Popup HTML al locației

```
// Inspectarea unui punct "clusted" cu un click
map.on('click', 'clusters', function (e) {
    const features = map.queryRenderedFeatures(e.point, {
        layers: ['clusters']
    });
    const clusterId = features[0].properties.cluster_id;
    map.getSource('campgrounds').getClusterExpansionZoom(
        clusterId,
        function (err, zoom) {
            if (err) return;

            map.easeTo({
                center: features[0].geometry.coordinates,
                zoom: zoom
            });
        }
    );
});

// Deschide un "pop-up" al unui punct negrupat cu informatiile acestuia;
map.on('click', 'unclustered-point', function (e) {
    const { popUpMarkup } = e.features[0].properties;
    const coordinates = e.features[0].geometry.coordinates.slice();

    // Ensure that if the map is zoomed out such that
    // multiple copies of the feature are visible, the
    // popup appears over the copy being pointed to.
    while (Math.abs(e.lngLat.lng - coordinates[0]) > 180) {
        coordinates[0] += e.lngLat.lng > coordinates[0] ? 360 : -360;
    }

    new mapboxgl.Popup()
        .setLngLat(coordinates)
        .setHTML(popUpMarkup)
        .addTo(map);
});
```

Evenimentul „*on-click*” a unei grupări, va face zoom pentru coordonatele respective și va separa punctele de referința pentru acea zona. Cu alte cuvinte, ca de exemplu, un grup de 10 locații dintr-o anume zona, va face zoom, iar grupările se vor dezbina intre(1,2,3 locații).

Evenimentul „*on-click*” al unui punct „*stand-alone*” va facilita un *Popup HTML* cu datele locației iar aceasta ne poate duce pe pagina locației. Un exemplu de *popup HTML* al unei locații va fi prezentat in următoarea figura.

Figura 3.13(Popup HTML al unei locații pe mapă)

3.2.4 Securitatea Parolei

Parole(adăugarea de „sare” și funcțiile hash)

Documentul utilizatorului are doi parametrii utilizați pentru a verifica corectitudinea parolei, așa cum am observat în timp ce demonstrăm cum sunt salvate datele în bază. mai precis, câmpurile *salt* și hash.

Figura 3-14(User salt si hash)

În domeniul criptografiei, termenul „*sare*” se referă la informații aleatorii care sunt adăugate la funcția unidirecțională utilizată pentru a calcula hash-ul unor date sau a unei parole.

Parolele de pe mediile de stocare pot fi securizate folosind „săruri”. Dintr-o perspectivă istorică, parolele au fost mult timp stocate la vedere, dar pe măsură ce timpul a trecut, au fost create mai multe măsuri de securitate pentru a preveni citirea de sistem a parolelor utilizatorilor. O astfel de abordare este adăugarea de „*sare*”.

Pentru fiecare parolă, o „*sare*” diferită este produsă la întâmplare. Cele doua componente sunt adesea combinate, folosind o metoda hash bazată pe criptografie, parola rezultată (!originală) mai apoi este salvată adăugând „sare” într-o DB. Hashing-ul se ocupa cu logarea viitoare stocarea nefiind necesară și, în consecință, fără a rula pericolul ca stocarea autentificării să fie piratată și să expună parola la vedere.

Un atac care utilizează un hash pre-generat este, de asemenea, prevenit folosind „sărurile”. Ele pot face ca dimensiunea parolei care a folosit metoda hash să fie excesiv de mare, fără a împovăra utilizatorii, utilizatorii nu sunt nevoiți să rețină aceste valori de „sare”. De asemenea, asigură parolele „standard”, sau cele care sunt folosite în mod regulat sau de către utilizatorii mai neexperimentați și cu (o singură parolă) pe diverse site-uri web, deoarece valorile „salt” sunt unice în fiecare situație.

De la autentificarea sistemului Unix la securitatea pe Internet, sărurile criptografice sunt adesea folosite într-o gamă largă de sisteme informatice.

Pentru aplicația noastră a fost folosit *API*-ul *Passport*. Acesta este responsabil de toate operațiunile de hash și adăugare de salt pe care le-am detaliat mai sus. În cele ce urmează va fi introdus un instant ce reprezintă modul de configurare al acestuia(*Figura 3.15*).

```
const passport = require('passport'); //Require Passport
const LocalStrategy = require('passport-local'); // Require Local-Passport
//TODO: Config Express Session
const sessionConfig = {
    store,
    name: 'session',
    secret,
    resave: false,
    saveUninitialized: true,
    cookie: {
        httpOnly: true,
        // secure:true,
        expires: Date.now() + 1000 * 60 * 60 * 24 * 7,
        maxAge: 1000 * 60 * 60 * 24 * 7
    }
}
app.use(session(sessionConfig))


// TODO: passport Setup
app.use(passport.initialize());
app.use(passport.session()); //passport session with express session;
passport.use(new LocalStrategy(User.authenticate())); // User auth

passport.serializeUser(User.serializeUser()); // store a user in a session;
passport.deserializeUser(User.deserializeUser()); //  get a user out of the session;
```

Figura 3.15(Configurarea Passport API)

3.3 Utilizarea Aplicației

3.3.1 Utilizarea(prezentare)

Vizitatorii (utilizatorii care nu și-au făcut cont și nu s-au înregistrat) și utilizatorii compleți (cei care sau înregistrat)pot accesa și utiliza programul. Aceste două grupuri de utilizatori folosesc programul în moduri ușor diferite, totuși. Următoarele vor oferi o demonstrație schematică a modului de utilizare în ambele scenarii:

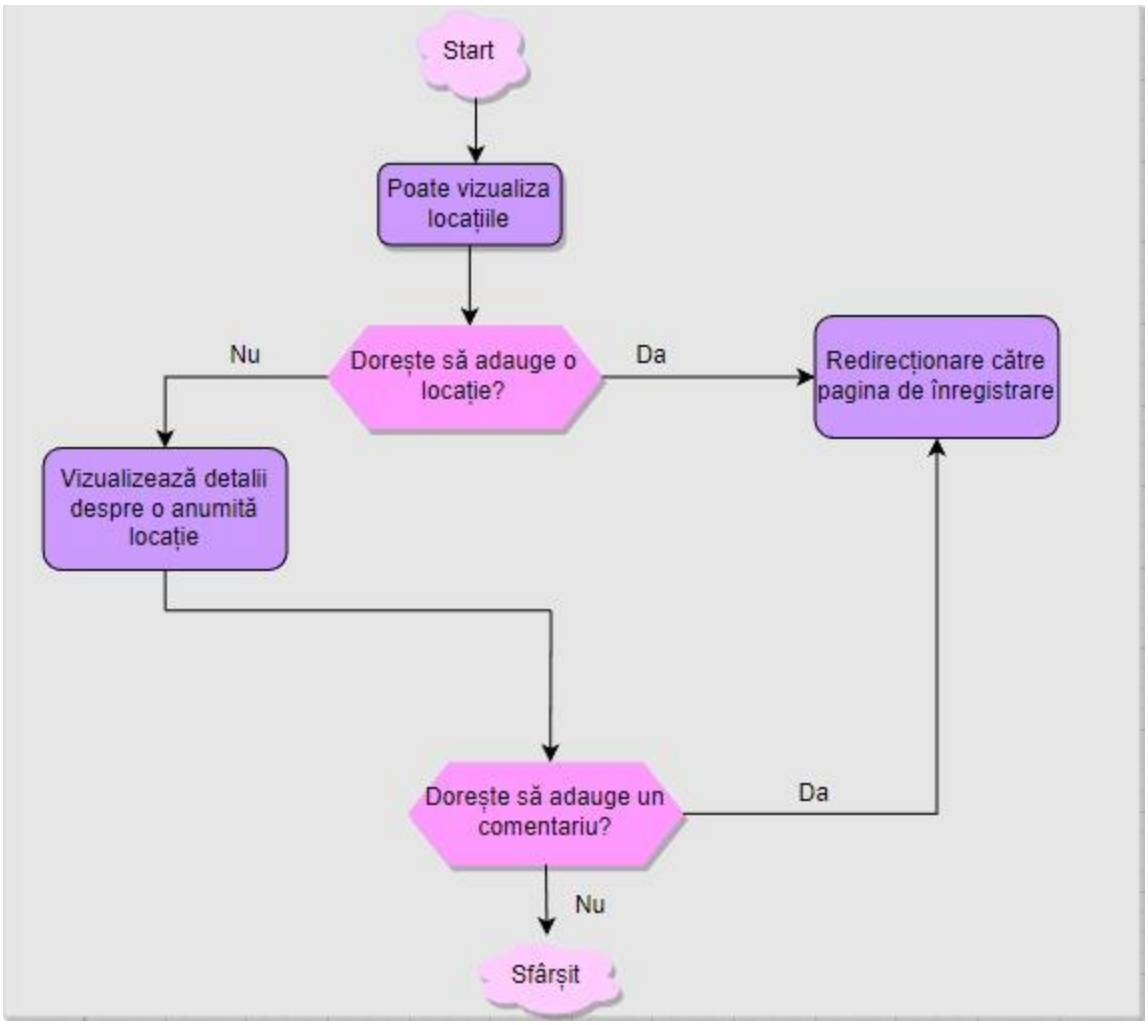

Figura 3.16(Diagrama pentru vizitatori)

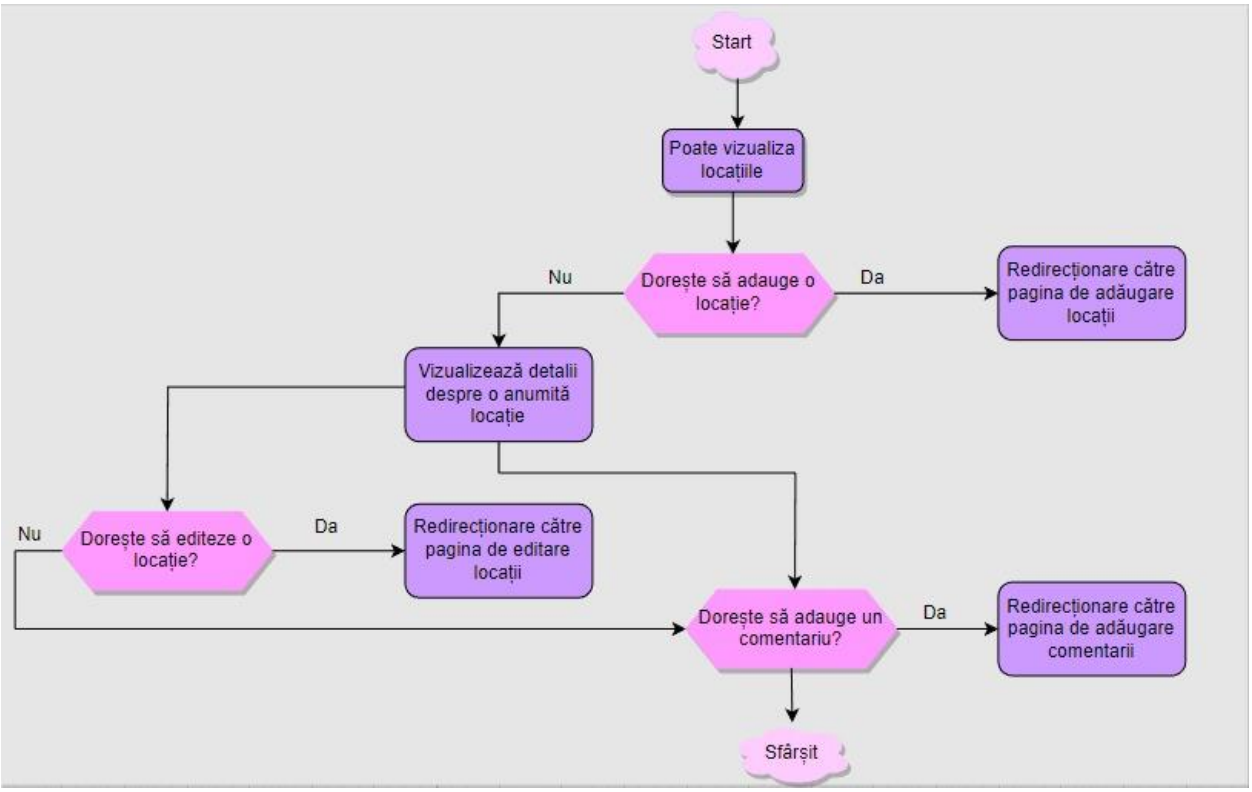

Figura 3.17(Diagrama pentru utilizatorii înregistrați)

3.3.2 Utilizarea(Interfața)

Design-ul UI se focusează pe prezicerea a ce funcții vor avea userii și pe bazarea că interfața conține părți care sunt ușor de utilizat, de înțeles și de accesat.

Termenii de (design, arhitectură, si conexiuni) sunt toți combinați în interfață.

Utilizare (Alegerea Interfeței)

Userii se obișnuiesc cu comportamentele specifice ale componentelor UI, prin urmare trebuie să fim singuri în deciziile noastre.

Următoarele sunt doar câteva exemple de componente de interfață:

Input-uri: Buton, câmp de text, casetă de selectare, buton radio, buton radio, listă derulantă, câmp pentru dată etc.

Navigație: câmpul de căutare, glisorul, etichetele, pictogramele etc.

Componentele care oferă informații includ indicii, pictograme, bare de progres, alerte, casete de mesaje, ferestre modale etc.

Mai multe componente pot fi acceptabile pentru afișarea materialului uneori. Este esențial să luăm în considerare compromisurile atunci când se întâmplă acest lucru. Este posibil ca utilizatorii să fie nevoiți să muncească mai mult pentru a deduce ceea ce este într-un meniu derulant.

Practici bune pentru proiectarea interfețelor

Înțelegerea obiectivelor, capacităților, preferințelor și înclinațiilor utilizatorilor va fi necesară pentru proiectarea interfeței. La crearea interfeței de utilizare, vom ține cont de acești factori:

Interfața noastră este păstrată simplă. Majoritatea oamenilor pot vedea cu greu cele mai bune interfețe de utilizator.

Evitam utilizarea componentelor străine în timpul utilizării. Folosim componente standard de interfață și menținem coerența. Utilizatorii sunt mai în largul lor și pot finaliza sarcinile mai rapid atunci când interfața are aspecte familiare. Pentru a face interfața mai ușor de utilizat, este de asemenea esențial să stabiliți modele (secvențe similare) în limbajul, stilul și designul site-ului. Odată ce un utilizator stăpânește o sarcină, el sau ea ar trebui să o poată aplica în alte zone ale site-ului web.

Folosim în mod deliberat textura și culoarea. Putem folosi totul in a avea un avantaj in a atrage atenția asupra sau a devia atenția de la anumite componente.

Pentru a oferi ierarhie și claritate, se folosește tipografia. Folosirea fontului este ceva pe care îl examinăm cu atenție. Putem îmbunătăți lizibilitatea și vizibilitatea prin diferite dimensiuni ale textului, fonturi și aranjamente. Ne asigurăm că sistemul transmite informații. Când există acțiuni, modificări de stare sau greșeli, îi anunțăm întotdeauna utilizatorilor noștri. Pentru a indica starea și, dacă este necesar, următoarele acțiuni, interfața ar trebui să utilizeze o varietate de caracteristici. Acest lucru va reduce supărarea utilizatorului.

Descrierea interfeței de utilizare

Paginile principale au o interfață de utilizator similară. Nu putem oferi o explicație detaliată a acesteia, deoarece vorbim despre o interfață receptivă care se modifică în funcție de dimensiunea ecranului fiecărui tip de dispozitiv. Folosind un laptop cu un ecran de 17 inch ca exemplu, vom prezenta schematic pagina de pornire după cum urmează:

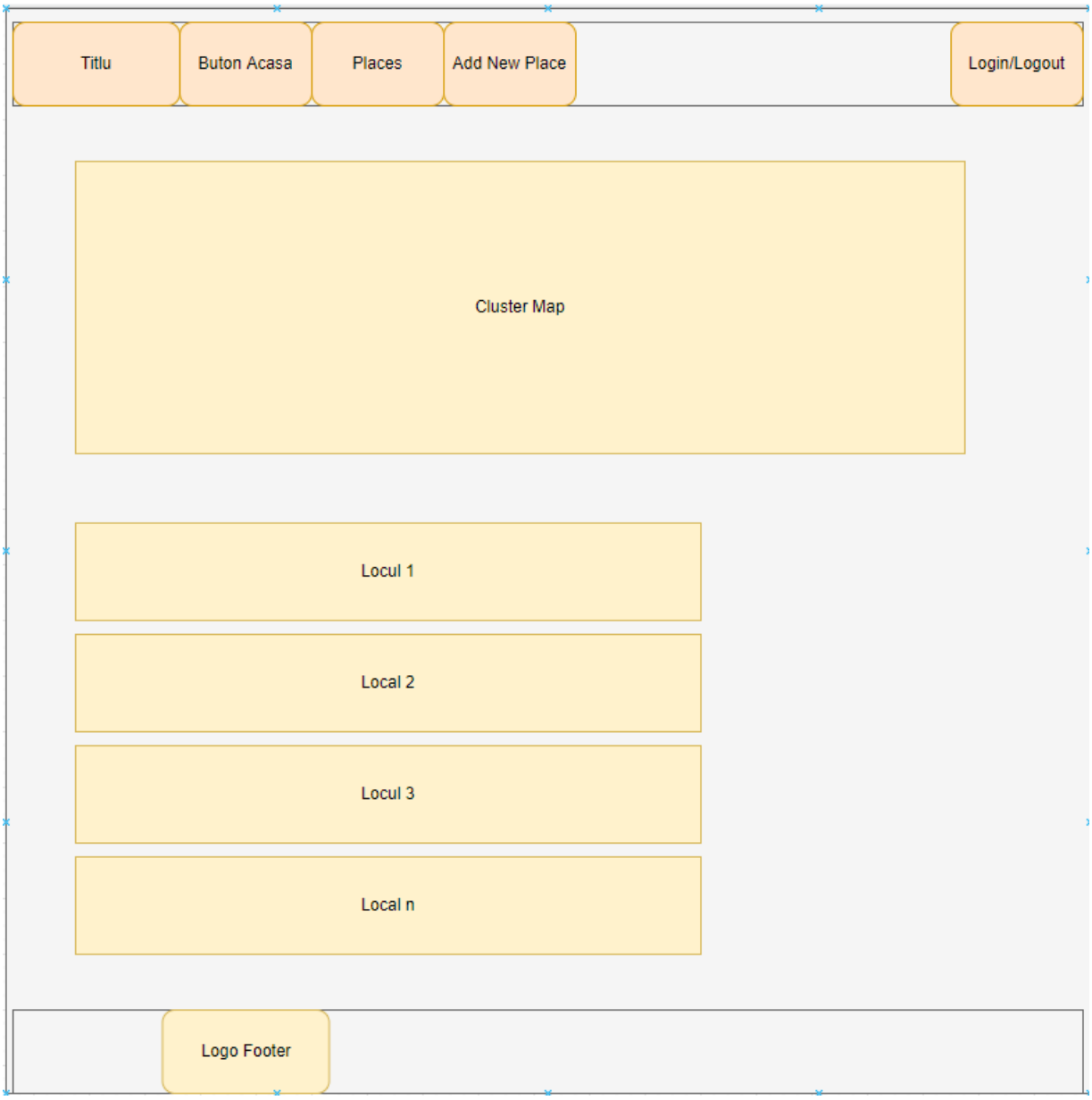

Figura 3.18(Template Pagină de pornire)

4. CONCLUZII

Putem adăuga *librăria* Angular capabilă pentru dezvoltarea „frontend” ca o potențială opțiune viitoare.

O librărie pentru sit-uri dinamice, putem opta pentru *AngularJS* fiind structural. Permite să creăm părți ale programului nostru folosind sintaxa *HTML*, permițându-ne totodată să o extindem. În *Angular*, o mare parte din codul care ar trebui scris în mod obișnuit este eliminat datorită principiilor de *binding data* și *injecție de dependență.*

Stiva tehnologică *MEAN* ar fi utilizată ca urmare a acestei îmbunătățiri (*MongoDB, Express.Js, AngularJs și Node.Js*). În ceea ce privește arhitectura, proprietarii anumitor platforme semnificative au optat pentru suita *MEAN*. Aici îmi vin în minte companii precum LinkedIn, Netflix și Yahoo.

Am putea lua în considerare o variantă a utilizării limbajului *Golang* dezvoltat de *Google* pentru o eventuală actualizare a porțiunii de *backend*.

Deși anterior am folosit *Uber* ca exemplu de firmă care folosește suita *MEAN*, este important de reținut că o parte din microserviciile acestei organizații au fost mutate de la *NodeJS* la *Golang*. Uber utilizează algoritmi de tipul acelor microservicii punctate în poligon (*PIP - Point In Polygon*), care consumă mult din procesor, care este motorul cheie din spatele acestor ajustări. Ei au făcut conversia necesară în *Golang*, deoarece funcționează pe multe *thread*-uri de execuție, spre deosebire de *NodeJS*, și pentru că au fost peste *200k* de solicitări pe secundă.

Trecerea la *Golang*, însă, ar fi inutilă în acest moment pentru aplicația descrisă în articol, deoarece solicitările serverului nu sunt suficient de complicate pentru ca noi să folosim capacitățile acestui limbaj de programare.

În cele din urmă, putem afirma că am structurat o aplicație de la cap la coada în *JS*, care ne-am întărit convingerile, făcându-l adecvat pentru implementarea proiectului în discuție.

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