MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

MINI - PORTAL PENTRU AGREGARE DE ȘTIRI ȘI INTERACȚIUNI COLABORATIVE

LUCRARE DE LICENŢĂ

Absolvent: **Ovidiu BUDA**

Coordonator ştiinţific: **s.l. ing. Cosmina IVAN**

2015

MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

DECAN,
Prof. dr. ing. Liviu MICLEA

DIRECTOR DEPARTAMENT,
Prof. dr. ing. Rodica POTOLEA

Absolvent: **Ovidiu BUDA**

MINI - PORTAL PENTRU AGREGARE DE ȘTIRI ȘI INTERACȚIUNI COLABORATIVE

1. **Enunţul temei:** *Proiectul își propune realizarea unui sistem online, care să permită utilizatorilor cu acces la Internet să utilizeze un mini-portal.*

2. **Conţinutul lucrării:** *Cuprins, Introducere, Obiectivele proiectului, Studiu bibliografic, Analiză și fundamentare teoretică, Proiectare de detaliu și implementare, Testare și Validare, Manual de instalare și utilizare, Concluzii, Bibliografie.*

3. **Locul documentării**: *Biblioteca Universității Tehnice Cluj-Napoca*

4. **Consultanţi**: s.l. ing. Cosmina Ivan

5. **Data emiterii temei:** 1 noiembrie 2014

6. **Data predării:** 10 septembrie 2015

Absolvent: ___Ovidiu_Buda_____

Coordonator ştiinţific: ___s.l._ing._Cosmina_Ivan_____

Cuprins

Capitolul 1. Introducere

1.1. Contextul proiectului

Acest capitol prezintă contextul general al tehnologiilor web pentru realizarea unei aplicații web single page. Continuăm cu expunerea contextului specific, care constituie și motivarea dezvoltării unei aplicații web single page de tip portal..

1.1.1. Contextul general

În ziua de azi asistăm mai mult ca oricând la o evoluție impresionantă a tehnologiilor lansate pe piața IT și putem fi beneficiarii multor dintre inovațiile aduse de diferite companii de profil. Noutățile aduse în ultimii ani în domeniul IT au încurajat dezvoltatea unor noi tehnologii web open-source, care au revoluționat modul în care se dezvoltă aplicațiile web..

Pornind de la aceste premise, sistemul dezvoltat își propune să fie unul de actualitate, care să capteze tendințele tehnologice actuale și să constituie o aplicație utilă pentru utilizatorii țintă..

1.1.2. Contextul specific

În continuarea acestui capitol va fi expusă motivarea alegerii temei pentru proiectul actual, cât și modul prin care aceasta se integrează în necesitățile omului dinamic, modern, conectat la informație, pentru care resursa temporală este vitală.

Trăim într-o perioadă și o societate în care avem acces facil la informații, în care avem posibilitatea de a explora lumea într-un mod lipsit de bariere, în care ne putem exprima părerile cu privire la subiectele care ne interesează și în care vrem să fim la curent cu tot ceea ce se întâmplă în jurul nostru. Ritmul alert și dinamismul cu care se desfășoară lucrurile în societatea actuală ne-a impus să acordăm o atenție mult mai mare modului în care ne organizăm timpul dar și a relațiilor și interacțiunilor inter-umane. Astfel putem identifica un scenariu major în care tehnologia ne-ar îmbunătăți experiențele zilnice și interacțiunea inter-umană: dorința de a organiza timpul petrecut în mediul online cât mai eficient cu putință..

Având în vedere cele enunțate anterior, s-a decis dezvoltarea unei aplicații web sub forma unui portal. Aplicația oferă posibilitatea de a adaugă diferiți portleți și de a configura acești portleți în funcție de preferințele fiecărui utilizator. Sistemul este gândit similar unei rețele sociale deoarece satisface nevoile utilizatorilor de a fi informați , de a comunica și partaja informații. Aceste nevoi se realizează cu ajutorul unor entități care poartă numele de portleți, astfel interacțiunea dintre utilizatori se realizează printr-un portlet de tip chat. O altă funcționalitate oferită de aplicație constă în posibilitatea de a asigna task-uri între utilizatori cu ajutorul unui portlet de tip „Task Manager”. De asemenea utilizatorii pot utiliza portletul de tip „News” prin care au acces la știri în funcție de sursa de informații aleasă sau portletul de tip „Weather”.

1.2. Structura lucrarii pe capitole

Documentul curent este structurat în 8 capitole, după cum urmează:

- capitolul 1 - constituie introducerea şi detalierea contextului problemei
- capitolul 2 - prezintă obiectivele proiectului
- capitolul 3 - descrie studiul bibliografic necesar familiarizării cu domeniul temei având drept scop identificarea şi analiza de sisteme similare
- capitolul - 4 realizează o analiză şi fundamentare teoretică în ceea ce priveşte cerinţele sistemului, cât şi tehnologiile necesare implementării
- capitolul - 5 abordează etapele de proiectare ale proiectului
- capitolul - 6 prezintă aspectele referitoare la testarea şi validarea produsului software
- capitolul - 7 urmăreşte întocmirea unui manual de utilizare al sistemului
- capitolul - 8 prezintă concluziile deduse din urma realizării proiectului propus.

Capitolul 2. Obiectivele Proiectului

În acest capitol vom detalia aspecte legate de obiectivele sistemului.

Sistemul proiectat și implementat, este un produs care își propune să vină în ajutorul utilizatorului, al cărui profil îl putem asocia cu o persoana interesată de utilizarea cât mai eficienta a timpului prin utilzarea unui produs care să-i ofere acestuia cât mai multe informații fară a fi nevoit să utilizeze simultan mai mutle aplicații, proces care ar fi „time-consumig”.

În continuare sunt prezentate un set de obiective ale aplicației:

- Vizualizarea portalului

Sistemul pune la dispoziția utilizatorului vizualizarea portalului care este divizat în două zone principale. Prima zonă constă într-un meniu în care sunt definite diferite acțiuni legate de funcționalitatea portalului. A doua zonă constă într-un container pentru portleți.

- Vizualizarea portleților

Utillizatorul are posibilitatea de a vizualiza toate funcționalitățile puse la dispoziție de portal în momentul în care dorește să adauge un anumit portlet. Aceasta listă conține tipurile de portlet-uri disponibile alături de o descriere a fiecarui portlet care specifică funcționalitatea adusa de acesta. Tot în aceasta funcționalitate se încadreaza și filtrarea porteților în funcție de numele acestora.

- Adăugarea unui portlet

Utilizatorul interesat de funcționalitatea adusă de un anumit portlet are posibilitatea de a adăuga portlet-ul în cadrul portalului. Atunci când utlizatorul adaugă un portlet, portalul este actualizat împreuna cu lista de portlet-uri active. În meniul de portlet-uri active acestea sunt ordonate în funcție de modul în care acestea sunt dispuse în cadrul portalului.

- Ștergerea unui portlet

Sistemul pune la dispoziția utilizatorului două moduri de a șterge un portlet din cadrul portalului. Aceasta funcționalitate oferă posibilitatea de a șterge orice portlet împreuna cu datele asociate acestuia. Atunci când utlizatorul șterge un portlet, portalul este actualizat modificându-și structura. De asemenea și lista activa de portleți este actualizată în urma utilizării acțiunii de ștergere.

- Modificarea setărilor unui portlet și a portalului

Aceasta funcționalitate are o importanță majoră, deoarce îi permite utilizatorului să-și configureze portlet-ul în funcție de dorințele sale. Fiecare portlet în parte are diferite setări care pot fi configurate de la caz la caz, singura setare comună fiind numele portletului care va fi afișat în cadrul portalului. O altă funcționalitate a aplicației este aceea de a modifica setările portalului. Aceasta constă în schimbarea structurii și a modului în care sunt afișati portleții în cadrul aplicației. Tot în această funcționalitate se încardreaza și salvarea setărilor portalului în cadrul browserului fiind folosit modulul de local storage al browser-ului.

- Utilizarea chat-ului

Utilizatorul poate utiliza portlet-ul de tip chat prin care comunică cu alți utilizatori ai aplicației care și-au activat acest portlet. Comunicarea se face pe bază de mesaje text.

Conținutul chat-ului va afișa numele utilizatorului care a postat mesajul și ora la care mesajul a fost postat, de asemenea sistemul notifică utilizatorii în momentul în care un alt utilizator se conectează/deconectează la chat.

- Utilizarea portalului de știri

Sistemul pune la dispoziția utilizatorului un portlet în care se pot vizualiza știri. Știrile sunt preluate în funcție de url-ul setat în configurarea acestui portlet. Acest URL trebuie să fie un rss feed care reprezintă resursa de unde vor fi preluate știrile. Știrile sunt afișate în cadrul portlet-ului sub forma unei liste iar în momentul în care utilizatorul dă click pe una din știri, acesta va fi redirecționat spre site-ul de știri pentru a vizualiza subiectul respectiv.

- Actualizarea profilului utilizatorului

Cu ajutorul acestei funționalități, utilizatorul își poate actualiza profilul, își poate schimba parola și username-ul asociat contului aplicației.

- Partajarea de task-uri cu alți utilizatori

Prin această funcționalitate sistemul pune la dispoziția utilizatorului un portlet prin care acesta are posibilitatea de a partaja task-uri cu alți utilizatori. În momentul în care accesează un task, utilizatorului ii va aparea un pop-up în care poate selecta numele utilizatorului/lor cu care dorește să partajeze task-ul și alte detalii legate de taskul respectiv.

Sumarizând cele prezentate în cadrul acestui capitol, aplicația își dorește să fie una care să înglobeze aspecte similare unei rețele sociale capabile să asigure managementul informației afișate utilizatorului, oferind totodată o interacțiune ridicată cu utilizatorul.

Capitolul 3. Studiu Bibliografic

Acest capitol este dedicat descrierii studiului realizat înainte de a se contura produsul software. Studiul a fost realizat pentru a putea identifica care este potenţialul ideii care stă la baza sistemului nostru, și anume o aplicație single-page de tip portal utilizând un stack de tehnologii MEAN.

În referința [11] sunt prezentate conceptele de baza care stau la dezvoltarea unei aplicații web utilizând stack-ul de tehnologii MEAN și de asemena sunt expuse și avantajele pe care le aduce dezvoltarea utilizând pachetul de tehnologii MEAN fața de abordarea clasică pe 3 layere.

În referința [8] sunt prezentate regulile fundamentale care trebuie urmate în dezvoltarea componentelor front-end în cadrul aplicațiilor single-page. Această referință m-a ajutat în stabilirea priorităților ăn momentul dezvoltării unei aplicații single-page cât și de a înțelege modul în care interacționează componentele între ele.

În referința [9] sunt prezentați pașii necesari în dezvoltarea unei aplicații single-page utilizând limbajul de programare JavaScript atât pe partea de client cât și pe cea de server dar și principalele beneficii aduse de această tehnologie, față de alte tehnologii existente. În referințele [1] și [4] se prezintă modul de utilizare și integrare a framework-urilor Express și AngularJS în cadrul unei aplicații web împreună cu API-urile care țin de acestea și a sub componentelor.

3.1. Portaluri

Un portal web este o pagină web care aduce utilizatorului informații din diferite surse, sub o formă uniformă. în general, fiecare sursă de informație este reprezentată sub forma unei zone specifice pentru afișarea informațiilor către utilizator. Această zonă bine definită poartă numele de portlet. în general utilizatorul are posibilitatea de a configura ce anume vrea să îi fie afișat. Utilizatorul are un rol foarte important fiindcă acesta poate determina conținutul care va fi adăugat sau șters din cadrul configurației portalului. Majoritatea portalurilor utilizează un API al motoarelor de căutare, lucru care perimite utilizatorilor să caute conținut intranet spre deosebire de conținutul extranet prin restictionarea domeniilor în care se realizează căutarea. De asemenea portalurile web pot oferi servicii cum ar fi e-mail, știri, cotări la bursă, diferite informații din baza de date sau chiar conținut media.Scopul portalurilor este a o oferi o interfață consistență din punct de vedere vizual și al utilizării, un mod de a controla accesul pentru aplicații multiple care altfel ar fi fost enitati web diferite reprezentate prin URL-uri variate. Facilitățile oferite de un portal pot fi restricționate în funcție de tipul de utilizator care utilizează aplicația. Există și au existat multiple portaluri web până în ziua de azi: AOL, iGoogle, Excite, MSN sau MyYahoo.

În referința [13], autorii prezintă un studiu aprofundat asupra portalurilor web. Portalurile web sunt descrise ca fiind sisteme de management al cunoștințelor care oferă acces rapid la diferite servicii și informații personalizate.

Conform referinței mentionate mai sus, autorii prezintă tipurile de portaluri bazate pe conținut și pe publicul țintă:

- Portaluri Informaționale: aceste portaluri oferă informații utilizatorului.

- Portaluri Colaborative: aceste portaluri conectează utilizatorii și le oferă facilitați de a colabora între ei.
- Portaluri de Expertiză: aceste portaluri permit utilizatorilor să comunice între ei și să distribuie conținut.
- Portaluri de cunoștințe: categorie care oferă utilizatorului o combinație între cele menționate mai sus.

În referința [10], autorul împarte portalurile în șase categorii în funcție de conținutul oferit:

1. Portaluri Verticale: aceste portaluri acționeaza ca și o poartă cu scopul de a prezenta utilizatorului produse și servicii specifice unei anumite industrii. Totodata un portal vertical, denumit și „vportal" oferă toate informațiile, articolele și statisticile legate de un anumit domeniu. Drept exemplu avem „cnet.com" care prezită informații despre computere. La rândul lor portalurile verticale pot fi împărțite în două mari tiplogii: portaluri corporate care oferă acces personalizat unei resurse specifice a unei industrii, iar cea de-a doua categorie este cea a portalurilor comerciale care oferă informații business-to-consumer, business-to-business și e-commerce.
2. Portaluri Orizontale: aceste portaluri reprezintă punctul principal pentru utilizator obișnuit fiindcă oferă o varietate mare de resurse și informații legate de diferite subiecte. Un alt rol al acestui tip de portaluri este de a oferi posibilitatea de a personaliza pagina web prin diferite canale. Ca exemplu avem „yahoo.com".
3. Portaluri Intranet: acest tip de portal este utilizat în cadrul unei rețele intranet al unei companii, instituții sau organizații. Rolul lor este de a oferi angajaților informații cum ar fi : aplicații, documente, training online și de asemenea serviciu de e-mail sau chat.
4. Portaluri de cunoștințe: acestea au rolul de a oferi utilizatorului resurse și informații accesibile.
5. Portaluri enterprise: acestea oferă suport utilizatorilor cu statut de membru, prin oferirea accesului la resursele unei anumite companii sau organizații. Este folositor pentru angajații unei companii cât și pentru diferiții colaboratori ai acesteia.
6. Portaluri ca piață de desfacere: aceste portaluri se referă la partea de e-commerce; suporta business-to-business și business-to-customer.

Portalurile posedă anumite caracteristici iar cele mai importante dintre ele vor fi detaliate mai jos:

1. **O singură autentificare:** Un portal reprezintă o poartă pentru o gamă largă de aplicații. Mai degrabă decât să așteptăm un utilizator să-și mențină și să-și amintească o parolă pentru fiecare aplicație hostata de poertal, acesta oferă o schemă complexă de autentificare, unde utilizatorul trebuie să rețină o singură parolă. O dată ce este autentificat, utilizatorul are acces nerestricționat la toate aplicațiile la care are dreptul. în cazul aplicațiilor externe portalului, este necesar să existe un mecanism de mapping între parametrii de autentificare pentru aplicația externă și parametrii de autentificare pentru portal.

2. **Personalizare:** Utilizatorul poate schimba interfața și comportamentul portalului în funcție de modul în care funcționează sau în funcție de nevoile și preferințele acestuia. Acesta are posibilitatea de a se abona și dezabona la diferite canale și alerte dar are posibilitatea de a șterge și modifica anumiți parametrii al seturilor de aplicații oferită acestuia (schema de culori, font-urile sau coloanele în care sunt dispuse portlet-urile).
3. **Adaptarea:** Portalul este capabil să salveze anumite acțiuni pe care le efectuează utilizatorul precum programul și fluxul de lucru, astfel sistemul este capabil să schimbe serviciile oferite pentru a realiza recomandari noi, depinzând de informația stocată. Practic, schimbările portalului și comportamentul acestuia se schimbă în funcție de context.
4. **Integrarea:** Companiile utilizează portaluri cu scopul de a difuza informații angajaților într-o manieră eficientă din punct de vedere a timpului. Din această perspectivă, portalurile pot fi categorisite ca fiind o evoluție naturală a CMS-urlior(Content Management Systems), dar în momentul de față portalurile depun eforturi pentru a integra aplicații mai vechi. Această caracteristică este văzută că fiind una capitală. într-adevăr, anumiți autori definesc portalurile ca „un framework pentru integrarea aplicațiilor și proceselor dincolo de granițele organizationale”[6]. Sistemele de tip portal pot fi văzute ca „management de conținut”, dar ceea ce le diferenteaza de un CMS este faptul că facilitează accesul la informații variate din diferite aplicații, surse de date și structuri. Utilizatorii selectează dintr-o listă de de componente pre-definite (uneori denumite „portleți”) și gestionează aspectul și modul în care este prezentată această informație în cadrul locației unei pagini. Utilizatorul are posibilitatea de a adăuga aplicațiile selectate în cadrul portalului. O parte din aceste aplicații pot fi portal-uri cu date real-time sau funcții de raportare și poate personaliza modul în care arată pagina.

Pentru a putea fi reprezentate funcționalitățile necesare unui sistem de tip portal, au fost studiate două sisteme deja existente (iGoogle și MyYahoo). Sistemul iGoogle permite utilizatorului să adauge mai multe tipuri de portleți și de asemenea să modifice poziția acestora în cadrul containerului de portleți. Fiecare portlet poate fi personalizat și configurat în funcție de tipul acestuia, de asemenea și componenta portal dispune de mai multe proprietăți configurabile cum ar fi structura portalului. MyYahoo oferă în mare parte aceleași funcționalități ca iGoogle diferența majoră constând în look-and-feel-ul aplicației care este mult mai prietenos cu utilizatorul și care este optimizat pentru dispozitivele mobile. Sistemul propus în această lucrare prezintă câteva avantaje față de sistemele prezentate mai sus. Aceste avantaje constă în utilizarea portalului indiferent de starea conexiunii la Internet, salvarea preferințelor utilizatorului în localStorage-ul browser-ului astfel structura portalului se încărca din memoria locală nemaifiind necesară o redesenare a acestuia în cazul în care structura ar fi fost persistată într-o bază de date.

În cele din urmă, se poate deduce faptul că portalurie web oferă un mod de accesa informația cât și de a o căuta, reprezentând în același timp un centru de comunicare pentru utilizatori. Oricum trebuie luat în considerare faptul că un portal web nu va fi capabil să acopere și să ofere toate informațiile dorite de anumiți utilizatori.

3.2. Portlet

Portleții pot fi văzuți că o reprezentare vizuală a serviciilor web care sunt împachetate pentru a fi livrate prin aplicații Web third-party (de exemplu un portal). Portleții sunt orientați spre utilizatori (returnează fragmente de markup mai degrabă decât date în format JSON) și totodată acest lucru se realizează în mai multe etape (încapsulează un lanț de pași mai degrabă decât să realizeze o livrare a datelor într-un singur pas). În general portleții sunt utilizați că o tehnică de modularizare pentru a oferi o structură a conținutului din portal. Faptul că au abilitatea de a fi livrați prin diferite aplicații Web, facilitând arhitecturiile orientate spre servicii (SOA) dar acum la nivel de front-end.

Din această perspectivă, portleții au același rol la nivel de front-end ca serviciile web la nivel de back-end, și anume de a facilita ansamblarea aplicațiilor prin servicii web reutilizabile. În cazul portleților, diferența constă în ceea ce este reutilizat (de exemplu ceea ce este conținut în layer-ul de prezentare) și unde se obține integrarea (de exemplu la nivel de front-end).

Penru a evalua mai bine noțiunea de portlet, vom face comparații în urmatoarele secțiuni între portleți cu servicii Web și aplicații Web tradiționale.

3.2.1. Portleți vs Sevicii Web

Serviciile web oferă tehnologii capabile de a livra conectivitate de tip business-to-business bazate pe Internet. Standardele serviciilor web facilitează distribuirea logicii de business, dar sugerează faptul că consumatorii de servicii Web trebuie să scrie un nou layer de prezentare pe baza logicii de business. Drept exemplu, să luăm în considerare un serviciu web care oferă două operații, denumite, findRoom și bookRoom. Acțiunea de a caută o cameră se bazează pe mai mulți parametrii de intrare (locație, dată ș.a.m.d) în timp ce rezervarea camerei preia informațiile legate de cameră (locație, dată) și rezervă o cameră în locația respectivă. La pasul următor, utilizatorul selectează una din camere și printr-un formular web aplicația preia informațiile legate de utilizator și metoda de plată. Această interacțiune va declanșa invocarea acțiunii de bookRoom. Acest exemplu ilustrează abordarea tradițională unde serviciile web oferă logică de business iar layer-ul de prezentare și cel de control sunt răspunzători de modul în care sunt apelate în aplicație. În orice caz , o asemenea abordare scoate în evidența importanța layer-ului de prezentare. Acest layer nu este responsabil doar de aspectele din punct de vedere estetic, ci și de de o serie întreabă de probleme cum ar fi problemele legate de reutilizare, managementul stări, manipularea erorilor, scriptarea client-side și logica de navigare. într-adevăr, majoritatea aspectelor care caracterizează un site Web ca fiind bun, sunt legate de probleme ce țin de interactivitate [2]. Re-crearea logicii de prezentarea în fiecare aplicație consumatoare are două mari limitări, creșterea timpului de lansare pe piață și periclitarea imaginii companiei[3].

Creșterea timpului de lansare pe piața. Logica de prezentare este este una dintre cele mai importante componente ale unui sistem dar procesul de creare al acesteia este greoi și necesită timp. Practicile presupun programarea personalizată pentru a crea câte un nivel de interfață utilizator pentru fiecare serviciu web nou. Acest lucru rezultă în creșterea timpului dedicat mentenanței lucru care împiedică inițiativele de dezvoltare a portalurilor în timp ce numărul de servicii web crește. Prin urmare, o abordare bazată pe

API că cea bazată pe servicii web tradiționale , se încadrează mai bine pentru aplicațiile complexe din punct de vedere al interactivității și al căror flux de utilizare implică mai multe pagini web.

Periclitarea imaginii companiei. Logica de prezentare realizează strategii bazate pe brand și experiența utilizatorului care devin factori importanți pentru o companie în continuă concurența cu competitorii ei. Lasând aplicația să decidă cum sunt ceruți parametrii și rezultatul trimis înapoi, poate periclita imaginea provider-ului de servicii. Compania poate să fie interesată în menținerea acestei experiențe în cazul în care serviciile oferite se realizează prin alte portaluri.

Necesitatea de a profita deplin de serviciile web din punct de vedere al tehnologic ca factor integrator în aplicații. Idealul conceptului de integrare în aplicație rezultă din punerea la dispoziție a unei aplicații în contextul unei a două aplicații, acest lucru incluzând interfața cu utilizatorul [12]. Obiectele Microsoft OLE reprezintă un exemplu perfect. De exemplu, aceasta tehnologie permite integrarea unui spreadsheet Excel direct în cadrul unui document Word prin deplasarea acestuia. În momentul în care a fost integrat, putem să lucram direct pe spreadsheet într-un document Word la fel ca și cum ar fi fost utilizat în Excel. Acesta este scenariul pe care susținatorii portleților îl au pentru aplicațiile web.

3.2.2. Portleți vs Aplicatii Web

Comparația precedenta sublinează notiune de portlet ca fiind o aplicație cu drepturi depline. În schimb, spre deosebire de aplicațiile Web, portleții au o cerință în plus: ei pot să fie subiectul compozitiei de parți terțe. Acest lucru are două implicații importante: interfețe clare și configurabile.

Interfețe clare. Acestea implică existența interfețelor bine definite și programabile pentru un portlet pentru a fi introduse în aplicație. În plus, inter operabilitatea recomandă ca această interfața să fie generica, astfel încat utilizatorul să poată interacționa cu portleții într-un mod standard. Acestea reprezintă eforturile standardului WSRP.

Configurabilitate. Este bine cunoscut în cadrul comunităților dezvoltatorilor de componente că dimensiunea componentei este invers proporționala cu numărul de reutilizari ale acesteia. Portleții sunt niște elemente dinamice fiindcă încapsuleaza atât layer-ul de prezentare dar și navigarea care vine cu acesta. Prin urmare, este necesar să existe mecanisme care să configureze portleții cu mediul în care portletul va fi „legat”. Acest context include starea ferestrei, profilurile utilizatorului, orientări din punct de vedere estetic și diferite preferințe legate de portlet. În urmatoarele paragrafe vom sublinia o parte din aceste proprietăți contextuale.

Starea ferestrei. Aceasta proprietate setează numărul de spații în cadrul paginii pe care îl va ocupa un fragment generat de portlet.

Profilul utilizatorului. Profilul utilizatorului este folosit pentru a personaliza conținutul în funcție de comportamentul acestuia. În momentul de față, acest conținut este oferit prin portleți.

Orientări din punct de vedere estetic. În momentul de față o pagină de portal este produsă ca „o combinatie de portleți”. Datorita acestui fapt este foarte important să se asigure un look-and-feel comun în cadrul mai multor portleți ca markup-ul să fie randat în aceeași pagină de portal (de exemplu: același fundal, fonturi, titluri s.a).

Markup-ul portletui trebuie să folosească CSS fiindcă acesta permite fragmentelor de HTML să parametrize o parte din aspectele estetice. Portletul returneaza fragmente parametrizate de CSS care sunt procesate de un consumator de portleți. Acest proces include oferirea de valori pentru parametrii CSS. Inter-operabilitatea necesită ca acești parametrii să fie standardizați astfel încat portalul să se astepte mereu la aceeași termeni indiferent de producatorul de portleți.

Preferințele portletului. Preferința unui portlet reprezintă un string de date cu o denumire care servește la personalizarea portletului. Aceste preferințe oferă un mecanism bazat pe parametrizare pentru a construi portletul. Ele pot fi schimbate în timpul configurării(de către administratorul portalului) sau la punerea în scena a acestuia. În ultimul caz, valorile pot fi setate chair de către portlet bazandu-se pe profilul utilizatorului sau solicitand utilizatorului acest lucru.

Portleții reprezintă componente integrabile care sunt manipulate și afișate în cadrul unui portal web. Portleții produc fragmente de markup (HTML,XHTML,WML) care sunt agregate în cadrul unui portal, astfel portalul poate fi considerat ca fiind o pagină web dedicată cu scopul de a agrega informații din diferite surse și de a le afișa într-o forma uniforma cu ajutorul componentelor de tip portlet.

În cele din urma, se poate deduce faptul ca portalurie web oferă un mod de accesa informatia cât și de a o cauta, reprezentand în același timp un centru de comunicare pentru utilizatori. Oricum trebuie luat în considerare faptul că un portal web nu va fi capabil să acopere și să ofere toate informațiile dorite de anumiți utlizatori.

Capitolul 4. Analiză și Fundamentare Teoretică

În acest capitol vor fi analizate cerințele funcţionale şi non-funcţionale ale sistemului, urmând să identificăm cazurile de utilizare pentru actorul sistemului. În ultimul subcapitol vor fi identificate tehnologiile folosite la crearea sistemului.

4.1. Cerințele aplicației

Cerințele sistemului reprezintă exprimarea serviciilor care sunt ncesare să fie puse la dispoziție de către sistem. Importanța acestora este majoră în construirea aplicației, deoarece reprezintă suportul de la care se pornește analiza și designul sistemului și oferă în același timp o perspectiva a obiectivelor impuse de părțile interesate .

Cerințele funcționale exprimă toate funcțiile pe care trebuie să le îndeplinească sistemul, cum trebuie să reacționeze în anumite situații, cum trebuie să răspundă anumitor intrări.

Cerințele non-funcționale reprezintă constrângerile serviciilor oferite de sistem și definesc capacitatea sistemului.

4.1.1. Cerințele funcționale

Cerinţele funcţionale urmăresc descrierea funcțiilor pe care trebuie să le implementeze sistemul şi a serviciilor oferite. Tabelul de mai jos expune cerinţele funcţionale ale aplicației, grupându-le în funcție de categoria logică în care se încadrează.

CF 1	**Adaugarea unui nou utilizator**
CF 2	**Actualizarea profilului utilizatorului**
CF 3	**Utilizarea portalului**
CF 3.1	Modificarea structurii întregului portal şi a modului în care sunt afişati portleții.
CF 3.1	Schimbarea temei portalului în fucntie de preferințe.
CF 4	**Functionaliţăti specifice unui portlet**
CF 4.1	Adaugarea unui portlet prin intermediul meniului.
CF 4.2	Eliminarea unui portlet
CF 4.3	Modificarea dimenisunii unui portlet
CF 4.4	Configurarea unui portlet
CF 4.5	Modificarea pozitiei unui portlet
CF 5	**Utilizarea portletului de tip news feed**
CF 6	**Utilizarea portletului de tip chat**
CF 7	**Utilizarea portletului care permite distributia de task-uri.**

4.1.2. Cerințele non-funcționale

Cerinţele non-funcţionale definesc proprietăţi şi constrângeri pe care produsul software trebuie să le îndeplinească alături de cele funcţionale. Aceste cerinţe au scopul de a defini performanţele sistemului, de a face o evaluare a sistemului din punct de vedere al calităţii, al securităţii, al rapidităţii, al comportamentului în situaţii extreme sau

a uşurinţei în întrebuinţare. Ele pot fi mai critice decât cerinţele funcţionale, deoarece dacă nu sunt îndeplinite, sistemul nu va fi util scopului în care a fost dezvoltat. În tabelul 4.2 sunt expuse principalele cerinţe non-funcţionale ale aplicației dezvoltate.

CNF 1 – Extensibilitate	Sistemul este ușor de extins, datorită faptului că s-a respectat o arhitectură stabilă, de tip MVC, cu elementele specifice stack-ului MEAN.
CNF 2 - Mentenabilitate	Aplicația este ușor de întreţinut.
CNF 3 - Performanța	Aplicația este rapidă, pentru că s-au folosit tehnici de programare avansate și concepte noi din tehnologiile web pentru a optimiza comportamentul aplicației.
CNF 4 - Reutilizare	Sistemul a fost conceput după o arhitectură MVC, cu caracteristicile tehnologiei JavaScript, astfel că este o aplicaţie modulară, în care fiecare componentă este ușor de refolosit.
CNF 5 - Disponibilitate	Aplicația funcţionează continuu. Nu sunt prevăzute cazuri speciale, în care sistemul să eşueze.
CNF 6 - Securitate	Datele despre utilizator sunt criptate, la fel şi username-ul şi parola.
CNF 7 - Încredere	Sistemul îşi patreaza performanţele de-a lungul timpului.
CNF 8 - Uzabilitate	Aplicația are o interfaţă prietenoasă, ușor de înţeles, folosindu-se de simboluri grafice ușor de recunoscut pentru orice tip de utilizator.

4.2. Cazuri de utilizare

Actorul principal al sistemului proiectat şi implementat este utilizatorul obişnuit. Idea definirii actorilor a fost de a oferi fiecaruia dintre ei drepturi egale pentru utilizarea portalului, astfel singura separare din punct de vedere a permisiunilor este între actorul înregistrat și logat în aplicație și utilizatorul neinregistrat. Pentru a beneficia de toate funcționalitățile puse la dispoziție de aplicație, actorul trebuie să fie logat, dar odată logat conexiunea la Internet nu reprezintă o necesitate, deoarece în cazul lipsei acesteia anumite operații nu vor putea fi realizate. Spre exemplu: utilizatorul care nu are o conexiune la Internet, poate să facă modificari asupra dashleților doar ca datele pe care operează acesta nu vor actualizate până când acesta va avea o conexiune la Internet. În principiu, utilizatorul poate să execute orice operațiuni asupra datelor cât timp este autentificat în aplicație, în caz contrar acesta este redirecționat către pagina de login / înregistrare.

Astfel, analiza cazurilor de utilizare se va realiza luând în considerare un utilizator autentificat.

4.2.1. Diagrama use-case pentru utilizatorul aplicației

În diagramele de use case, reprezentate mai jos, sunt ilustrate acțiunile pe care le poate realiza atât utilizatorul logat în aplicație, cât și cel neautentificat. Se observă că între cele două tipuri de utilizatori există o diferență majoră, deoarece acțiunile puse la

dispoziția utilizatorului neautentificat sunt limitate în comparație cu cele disponibile pentru un utilizator autentificat.

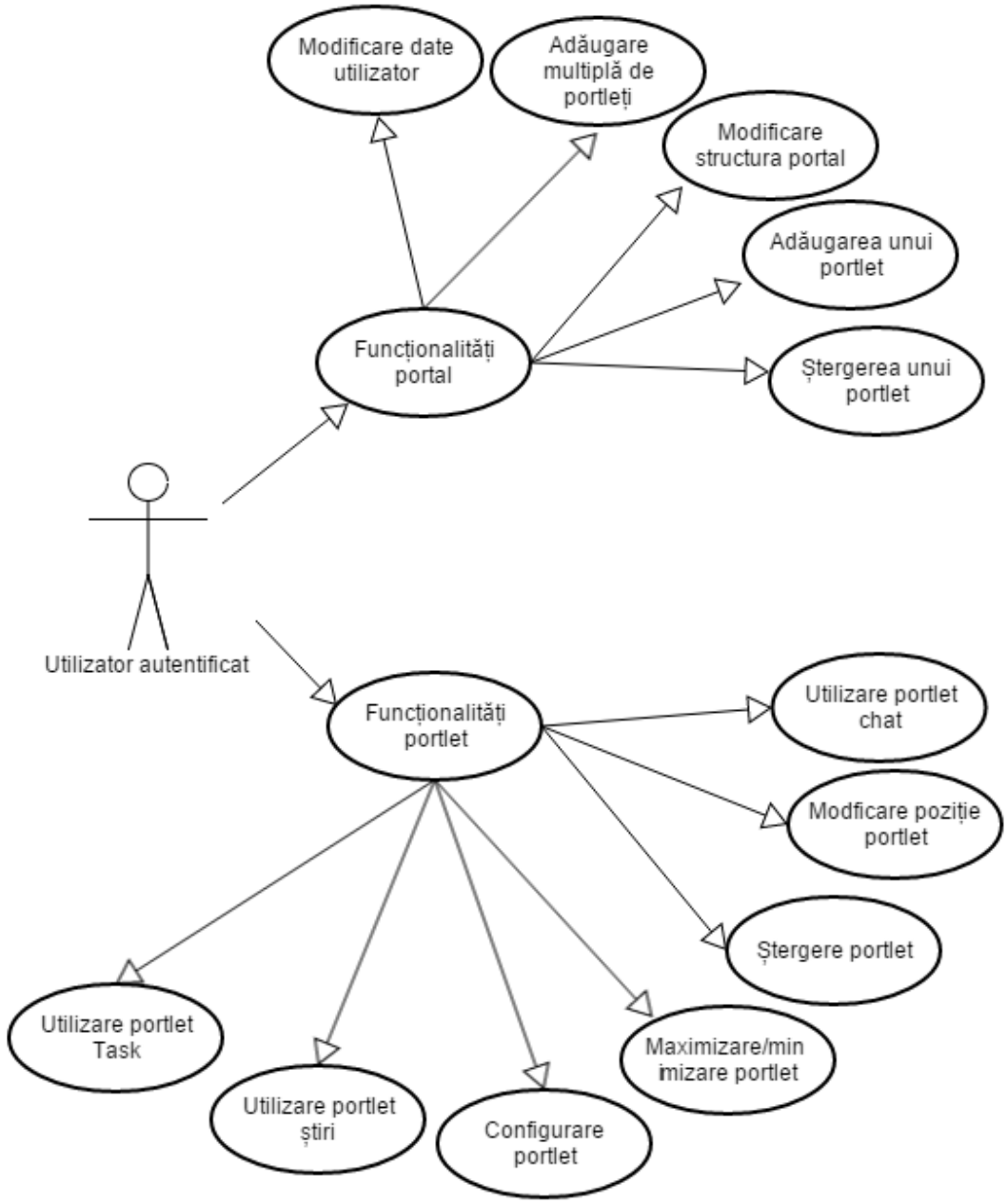

Figura 4.1 Diagrama Use-Case

În continuare, vor fi analizate câteva modele de use case mai interesante pentru utilizatorul obişnuit, pentru fiecare dintre acestea specificându-se precondiţiile, postcondiţiile, dar şi scenariul de succes, alături de scenariul de eșec. În mare parte a cazurilor nu vor există scenarii de eșec fiindcă sistemul pune la dispoziția utilizatorului un mecanism de toleranță al eșecului care constă în stocarea tuturor request-urilor executate în cazul în care nu există o conexiune la Internet într-o coada din care urmează să fie executate în momenul în care va există din nou o conexiune la Internet.

CU1. Titlu: Adaugarea unui portlet

Descriere: utilizatorul accesează meniul „Add new portlet”, dorind adăugarea unui portlet nou.

Actor primar: utilizatorul autentificat

Precondiții: utilizatorul dispune sau nu dispunde de conexiune la Internet

Postcondiții: nu există

Principiu: utilizatorul accesează aplicația, se loghează, și accesează meniul Active Portlets.

Scenariu de succes: utilizatorului i se afișează portalul în care se regăsește portletul adăugat de acesta.

Scenariu de eșec: nu există

CU2. Titlu: Modificarea structurii portalului

Descriere: utilizatorul accesează meniul „Settings”, dorind modificarea setărilor portalului.

Actor primar: utilizatorul autentificat

Precondiții: utilizatorul dispune sau nu dispunde de conexiune la Internet

Postcondiții: nu există

Principiu: utilizatorul accesează aplicația, se loghează, și accesează meniul Settings.

Scenariu de succes: utilizatorului i se afișează un pop-up de settings în care are posibilitatea de a modifica structura portalului. Acesta modifica structura după care apasa butonul Close urmând ca portalul să-și schimbe structura.

Scenariu de eșec: nu există

CU3. Titlu: Modificarea setărilor unui portlet

Descriere: utilizatorul accesează meniul „Configure”, dorind modificarea setărilor portletului, după care da click pe butonul „Edit”.

Actor primar: utilizatorul autentificat

Precondiții: utilizatorul dispune sau nu dispunde de conexiune la Internet

Postcondiții: nu există

Principiu: utilizatorul accesează aplicația, se loghează, și accesează meniul Settings al unui portlet.

Scenariu de succes: utilizatorului i se afișează un pop-up de settings în care are posibilitatea de a modifica diferiti parametrii ai unui portal în funcție de tipul acestuia. Acesta modifca portlet-ul în momentul în care dam click pe butonul „Close”.

Scenariu de eșec: în cazul în care utilizatorul nu dispune de conexiune la Internet în momentul configurării portlet-ului, caz în care sistemul va afișa în interiorul portletului în cauza un mesaj exprimat astfel: „The promises could not be resolved”.

CU4. Titlu: Vizualizarea detaliilor despre un portlet

Descriere: utilizatorul accesează meniul „Active portlets”, dorind să vada detalii legata de fiecare portlet în parte, după care da click pe butonul „Add multiple portlets”

Actor primar: utilizatorul autentificat

Precondiții: utilizatorul dispune sau nu dispunde de conexiune la Internet

Postcondiții: nu există

Principiu: utilizatorul accesează aplicația, se loghează, și accesează meniul „Active Portlets” al portalului.

Scenariu de succes: utilizatorului i se afișează un pop-up cu un tabel în care sunt afișate toate tipurile de portlet-uri disponibile pentru a fi adaugate. În dreptul fiecarui portlet se regăsește o descriere a acestuia.

Scenariu de eșec: nu există.

CU5. Titlu: Distribuirea de task-uri prin intermediul portletului de tip Tasks

Descriere: utilizatorul adaugă un portlet de tip Tasks, urmând să asigneze sau creeze anumite task-uri.

Actor primar: utilizatorul neautentificat sau autentificat

Precondiții: utilizatorul dispune sau nu dispune de conexiune la Internet

Postcondiții: nu există

Principiu: utilizatorul accesează aplicația și adaugă un portlet de tip „Tasks”. Urmeaza ca utilizatorul să selecteze un anumit task unde are posibilitatea de a atasa un fisier la task-ul respectiv, de a schimba starea taskului sau de a îl asigna la un alt utilizator.

Scenariu de succes: sistemul realizează acțiunea de distribuire a taskului, iar taskurile asignate unui anumit utilizator vor deveni vizibile pentru acesta.

Scenariu de eșec: în cazul în care nu există conexiune la Internet fișierele nu vor fi uploadate până în momentul în care utilzatorul va avea conexiune la Internet iar upload-ul se va execută automat dintr-o coada de request-uri în care sunt pastrate toate request-urile care se fac dar nu se execută atunci când nu există conexiune la Internet.

CU6. Titlu: Configurarea portlet-ului de tip știri

Descriere: utilizatorul intră în aplicație și acceseaza meniul „Active Portlets”. Mai apoi da click pe butonul „Add portlet” urmând să dea click pe optiunea „News”. După adăugare, utilizatorul trebuie să configureze portletul nou adaugat prin selectia simbolului grafic „Configure” urmat de un click pe butonul „Edit”.

Actor primar: utilizatorul autentificat

Precondiții: utilizatorul dispune sau nu dispunde de conexiune la Internet

Postcondiții: evenimentul este adaugat în lista de „wishlist”

Principiu: utilizatorul intră în aplicație și acceseaza meniul „Active Portlets”. Mai apoi da click pe butonul „Add portlet” urmând să dea click pe optiunea „News”. După adăugare, utilizatorul trebuie să configureze portletul nou adaugat prin selectia simbolului grafic „Configure” urmat de un click pe butonul „Edit”. În cadrul pop-ului afișat acesta va modifica parametrii „Feed URL”.

Scenariu de succes: sistemul realizează acțiunea de modificare a parametriilor portlet-ului iar acesta va afișa o lista de știri bazata pe configurarea facuta anterior.

Scenariu de eșec: în cazul în care nu există conexiune la Internet modificarea nu se va realiza numia în momentul care avem conexiune la Internet din nou urmând ca stirile să fie preluate automat în momentul respectiv.

CU7. Titlu: Modificarea profilului utilizatorului

Descriere: utilizatorul accesează aplicația și dorește să-și modifice profilul.

Actor primar: utilizatorul autentificat

Precondiții: utilizatorul dispune sau nu dispune de conexiune la Internet

Postcondiții: nu există

Principiu: utilizatorul accesează aplicația și da click pe numele utilizatorului logat, urmând să acceseze meniul „My Profile". Urmează ca utilizatorul să modifice diferite date legate de profilul lui și să apese butonul „Save".

Scenariu de succes: sistemul realizează acțiunea de modificare a informațiilor ce țin de utilizator.

Scenariu de eșec: utilizatorul nu mai dispune de conexiune la Internet în momentul realizarii salvării datelor, urmând ca acestea să fie salvate în momentul în care va dispune de conexiune la Internet.

4.3. Tehnologii utilizate

4.3.1. Single Page Application

Aplicatiile single page reprezintă unul dintre subiectele principale de discutie ale dezvoltatorilor de aplicații web. Pana în momentul de față există o multitudine de framework-uri JavaScript disponibile care usureaza munca dezvoltatorilor web. Nu putem să negam faptul ca JavaScript reprezintă tehnologia care a evoluat cel mai mult de la aparitia Web-ului. Acum putem folosi limbajul JavaScript pentru dezvoltarea componentei de font-end cât și a celei de back-end (Node.JS)

Aplicatiile single-page sunt aplicații web care ocupa o singura pagină web cu scopul de a oferi utilizatorului o experiența fluida în utilizare și o interfața bogata. De asemenea, o singura pagină nu se rezuma la un singur fisier fiindcă există posibiliatatea de a avea mai multe template-uri în fișiere diferite. Scopul principal al aplicațiilor web este de a updata și modifica parți ale unei interfete cu utilizatorul fara a trimite și primi request-uri full-page. În ultimii ani popularitatea SPA-rilor a crescut, în mare parte datorită faptului ca sunt mai ușor de dezvoltat, și fiindcă de la inceput toata dezvoltarea se bazeaza pe componenta front-end a unei aplicații web folosind AJAX (pentru a interactiona cu serverul), template-uri HTML, un framework MVC/MVVM și JavaScript.

Există o multitudine de avantaje legate de implementarea unei aplicații web single-page iar o parte dintre acestea for fi detaliate mai jos:

- Din punct de vedere arhitectural avantajul unei aplicații single-page constă în reducerea „comunicarii" din cadrul aplicației. Daca majoritatea procesarilor se realizează în componenta de client, numărul request-urilor către server este dramatic redus, împreună cu erorile care au codul HTTP 503 și care tind să distrugă experiența de utilizare a aplicației. De fapt, aplicațiile single page oferă posibilitatea de a realiza procesarile offiline, ceea ce reprezintă un avantaj major în anumite cazuri.
- Performanta este îmbunătățită datorită faptului că randarea se face client-side , dar nu reprezintă motivul cel mai convingător pentru a dezvolta o asemena aplicație.

- Flexibilitatea din cadrul interfeței cu utilizatorul reprezintă cel mai important avantaj fiindcă componenta de front-end este bine separata de server astfel nu va influenta performanta acestuia.

4.3.1.1.Arhitectura unei aplicații single-page:

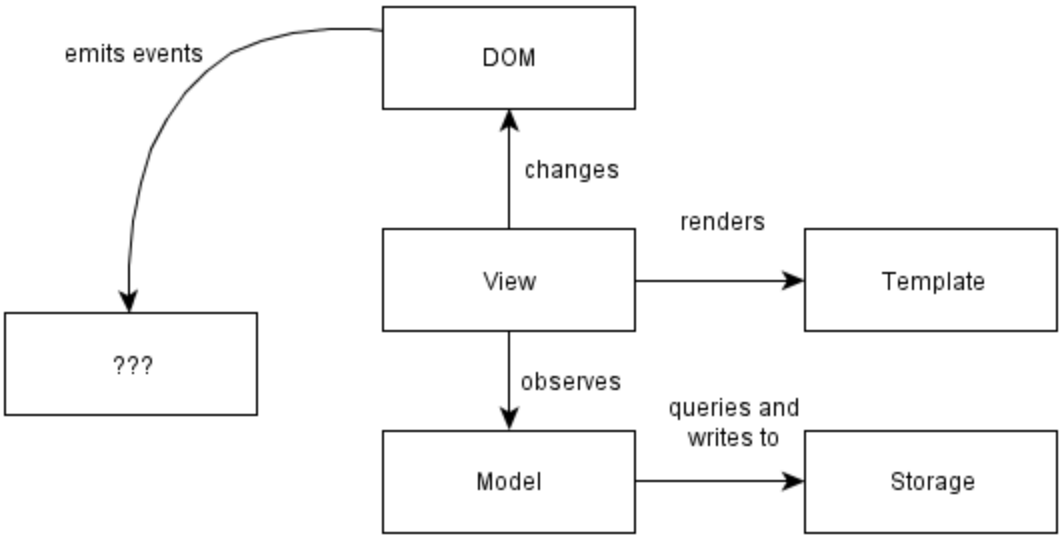

Figura 4.1 Arhitectura unei aplicații single page.

DOM(Document Object Model). Datele și starile nu sunt citite din DOM. Aplicația afiseaza HTML și se ocupa de operațiile asupra elementelor , dar niciodata nu se citeste nimic din DOM. Mentinerea stariilor în DOM este foarte greu de administrat deoarece este mult mai bine să avem un singur loc în care datele sunt servite și să randăm interfața utilizator cu ajutorul datelor, în special atunci când aceleasi date trebuie să fie afișate în locuri multiple din interfața cu utilizatorul.

Modelul are rolul de a depozita datele , de a emite evenimente atunci când datele se modifica, dar în același timp acesta poate fi serializat sau persistat. Modelul conține datele actuale(atributele) și poate fi transformat în format JSON pentru a fi servit spre layer-ul de prezentare sau pentru a fi salvat în backend. Un model poate avea mai multe asocieri , pot există reguli de validare și poate avea subscriberi(abonati) care urmaresc modificarea acestuia.

View are rolul de observa modificarile modelului și reflectă și conținutul acestuia. În cazul în care avem mai multe view-uri care depind de un singur model (ex: când un model isi schimba valoarea, redeseneaza view-urile), nu se dorește să urmarim fiecare modifare a fiecarui view în parte . Pentru a evită problema mentionata mai sus există un sistem care gestionează schimbarile prin care fiecare view primeste notificari de la model și se ocupa singur de redesenare.

4.3.1.2.Starea de run-time

Starea de run-time reprezintă modul în care arata aplicația în momentul în care aceasta ruleaza într-un browser și se refera la conținutul variabilelor și al pasilor care implică tranzitia de la o pagină la alta. Astfel există trei relatii :

1) URL < - > stare : Scopul aplicațiilor single page este cel de a oferi utlizatorului o interacțiune cât mai bogata cu aplicația . Activitatiile mai complexe implică faptul ca există mai multe stări ale view-ului care nu pot fi exprimate în URL. Pe de alta parte, se dorește ca să fie posibila salvarea paginii în bookmarks astfel oferind utilizatorului posibilitatea de a reveni în același punct în mod direct. Pentru a oferi suportul necesar salvarii în bookmarks, este necesar să reducem pe cât posibil nivelul de detalii care este suportat în URL. Daca fiecare pagină are un scop și o activitate primara(care este reprezentat până într-un anumit nivel al detaliilor în URL), atunci fiecare pagină poate fi reaccesata într-o anumită masura din bookmark. Activitatiile secundare (cum ar fi un chat în cadrul unei aplicații de e-mail) este resetat la starea initiala atunci când pagina este reincarcata, deoarece pastraerea datelor legata de acesta într-un URL care poate fi salvat nu are niciun sens.
2) Definitie < - > initializare : Componentele reutilizabile trebuie să fie definite fara a fi instanțiate sau activate, fiindcă acest lucru permite reutilizarea și testarea lor. Pentru realizarea acestui obiectiv avem la dispoziție trei abordări: prima este de a avea o funcție pentru fiecare modul care primeste ca parametru un ID și instantiaza view-urile și obiectele necesare. Cea de a doua varianta constă în pregatirea unui fisier global de „bootstraping” urmat de un sistem de rutare care încarcă starea corecta din lista de stări globale. A treia varianta este cea mai eleganta și presupune plasarea tuturor componentelor împreună, acest lucru facand imposibilă determinarea oridinii în care se realizează instanțierea.
3) Elemente HTML < - > obiecte view și Evenimente HTML < - > schimbari view: Aceasta relație ridică problema vizibilitatii care poate fi obtinuta în starea de run-time în funcție de framework-ul utilizat . Există anumite framework-uri care oferă posibilitatea de a vedea ce se va întâmpla în momentul în care utilizatorul va da click pe un anumit element HTML.

4.3.1.3.Mentenabilitatea depinde de modularitate

Modularitatea este unul din cele mai importante aspecte de care trebuie să se tina cont atunci când se dezvolta o aplicație. O buna modularizare usureaza procesul de testare și defineste cât de mentenabil este codul. Un cod mentenabil este ușor de inteles, ușor de testat și ușor de refactorizat. În schimb un cod „hard-to-maintain” are multe dependințe, ceea ce îl face greu de inteles și greu de testat independent și nu ca un tot unitar, acceseaza și scrie date la nivel global, acest lucru complicand procesul de testare, expune o mare parte din cod ceea ce îl face mai greu de refactorizat fara a afecta alte componente care depind de interfața publică. De asemenea are efecte secundare care conduc la imposibilitatea de a instantia ușor și în mod repetat al codului într-un test.

Browserele nu au un sistem de gestiune al modulelor ele fiind limitate la incararea de a încarcă fișiere care conțin cod JavaScript. Tot ce este în scopul radacină al fișierelor respective este injectat direct în scopul global prin variabila **window**, în aceeași ordine în

care sunt specificate tag-urile de script. Prin cod „modular JavaScript” se intelege utilizarea de namespaces(spatii de nume). Aceasta abordare constă în alegerea unui prefix (ex : window.myApp) căruia ii sunt asignate dependințe cu scopul de a avea fiecare obiect cu propriul sau nume astfel facilitand modularitea. Spatiile de nume creaza ierarhii dar sufera și de urmatoarele două probleme :

Alegerile legate de izolare trebuie să fie facute la nivel global: Intr-un sistem bazat doar pe spatii de nume există posibilitea de a avea funcții și variabile private, dar alegerile legate de izolarea trebuie facute la nivel global în cadrul unui singur fisier sursa. Fie expunem ceva în spatiul de nume global, fie nu facem acest lucru. Aceasta decizie nu oferă destul control deoarece în cadrul spatiilor de nume nu ne dorim să expundem doar anumite detalii pentru anumiți utilizatori fara a face codul accesibil prin spatiul de nume. Acest lucru conduce la cuplarea prin spatiile de nume accesibile la nivel global. Daca expunem un detaliu, nu avem control în cazul în care alte parți din cod pot accesa sau pot să depinda de ceva pe care am dorit să-l facem vizibil doar pentru un subsistem limitat.

Modulele devin dependente de starea globala: Problemele legate de spatiile de nume constă în faptul ca ele nu oferă nici o protectie când vine vorba de starea globala. Spatiile de nume globale au tendinta de a conduce la neglijenta : din moment ce avem control deplin asupra vizibilitatii, este foarte ușor să cadem în capcană de a adaugă sau a modifica lucruri în scopul global. Una din cauzele majore ale complexității de a scrie cod cu intrari remote (ex: lucruri referite în numele global care sunt definite și modificate în alte parți) sau cu efecte globale (ex: atunci când ordinea în care a fost inclus un modul afecteaza alt modul din cauza modificarii variabilelor globale). Codul la nivel global poate avea rezultate diferite în funcție de ce există în scopul global (ex: window.*). Modulele nu trebuie să adauge lucruri noi scopului global. Datele din scopul local sunt mai ușor de inteles, modificat și testat în comparatie cu datele din scopul global. Daca există cod care trebuie pus în scopul global, codul respectiv trebuie izolat și trebuie să devina partea a unui pas de initializare. Spatiile de nume nu ne oferă posibitatea de a face acest lucru, incurajand dezvoltatorul să schimba starea globala prin injectarea dependintelor în funcție de necesitati.

4.3.1.4. View Layer

Layer-ul de view este partea cea mai complexă din cadrul aplicațiilor single page. Acesta este și scopul principal din spatele aplicațiilor single page, cel de a view-uri bogate și interactive. Există două abordări de a implementa layer-ul de view: primul mod se bazeaza pe cod, și cel de al doilea care se bazeaza pe markup și are un sistem de templating destul de complicat.

View-urile trebuie să indeplineasca mai multe sarcini:

-**Randare** a unui template. Este necesar să avem un mod prin care preluam datele și le afisam ca HTML.

-Updatarea view-urilor ca răspuns la anumite evenimente. Atunci când datele modelului se schimba, este necesar să updatam view-urile cu care interactioneaza altfel în cât să reflecte schimbarile de date.

-Legarea comportamentului la HTML prin event handlers. Atunci când utilizatorul interactioneaza cu view-ul HTML, avem nevoie de un mod de declansare al comportamentului pe care îl are codul.

Prin implementarea layer-ului de view se asteapta să se ofere un mecanism standard sau o conventie pentru a realiza aceste sarcini. În figura 4.2 se observa modul în care un view interactioneaza cu modelele și HTML când realizează aceste sarcini:

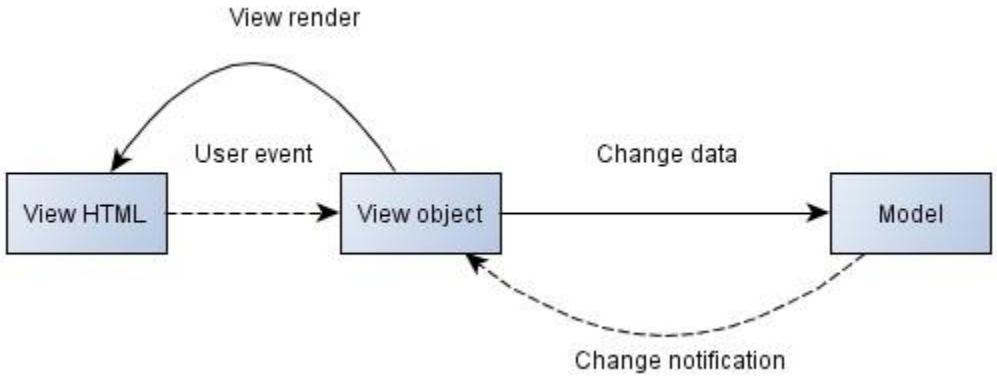

Figura 4.2 Interacțiunea view-model

Există mai multe aspecte pe care un dezvlotator le ia în considerarea atunci când scriem codul pentru layer-ul de view:

1) Interactivitate low-end vs high-end
   Pentru a exemplifica interactivitatea low-end vs high-end am ales să compar două aplicații web cunoscute: Github și Gmail.

Interactivitate Low-End(Github)	**Interactivitate high-end(Gmail)**
Paginile sunt statice	În general paginile sunt dinamice
Serveste documente(HTML) care conțin informații statice deja procesate carora li se adaugă interactivtate cu ajutorul JavaScript	Pune la dispoziție un set de date cu care utilizatorul poate interactiona în diferite moduri; schimbarea datelor se reflectă instant în pagina
Schimbarea datelor cauzează în general un refresh al paginii	Schimbarea datelor updateaza view-ul dar nu cauzează un refresh al paginii deoarece view-urile au stări intermediare care nu sunt salvate în baza de date.
Starea și datele pot fi stocate în HTML.Daca datele se schimba atunci pagina este reincarcata.	Pastrarea starii și a datelor în HTML nu reprezintă o idee buna deorece este greu să menti sincronizate mai multe view-uri care reprezintă aceleasi date.
Fiindca majoritatea stărilor se regăsesc în HTML, partii ale interfeței utilizator nu interactioneaza între ele.	Interacțiunile complexe sunt mai fezabile; datele sunt separate de layer-ul de prezentare.
Daca este nevoie de interacțiuni complexe ele se vor rezolva prin server.	Interacctiunile nu trebuie să se mapeze la actiunile din back-end.

Tabel 4.1 Comparatie Low-end interactivity vs High-end interactivity

Atat Gmail cât și Github sunt două aplicații web moderne dar au diferite nevoi și abordări. Aplicatiile web sunt o combinatie între mai multe tipuri de view-uri. O parte din aceste view-uri implică interacțiuni complexe și beneficiează de arhitectura creată pentru interativitatea high-end. Celălalte view-uri sunt componente simple care adaugă foarte puțina interactivitate. Ele pot fi mai ușor de implementat folosind metode simple.

Aproape de server vs. Aproape de client. Interfața utilizator generată de server este mai apropiata de baza de date, ceea ce oferă posibilitatea de a avea un timp de acces la baza de date mai mic și latență mai mica. În schimb interfețele utilizator care sunt randate pe partea de client oferă posibilitatea de a dezvolta o aplicație mai complexă, responsive mult mai apropiata de standardele actuale.

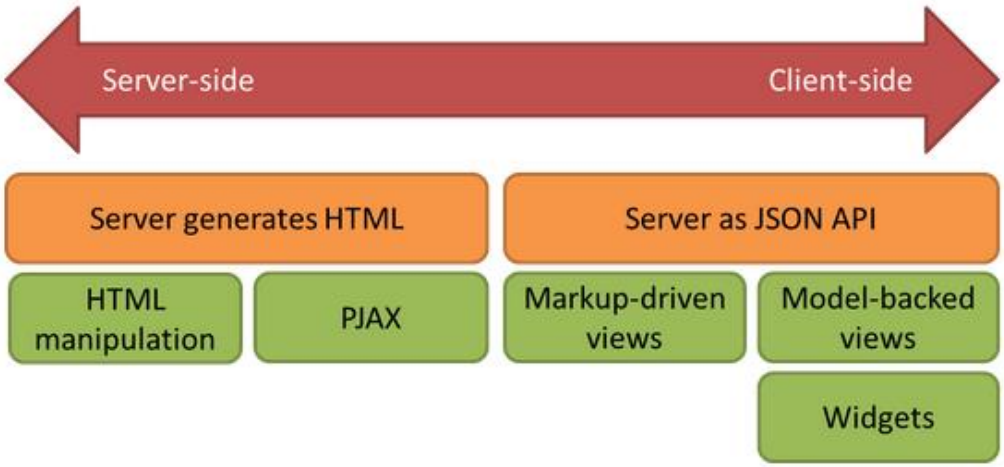

Figura 4.3 Divizarea sarcinilor între client și server

Datele în markup/manipularea HTML. În cazul în care datele sunt stocate în HTML, se servesc o gramadă de script-uri care folosesc DOM sau jQuery pentru a manipula HTML și a oferi o experiența mai placuta utilizatorului. Structurile specifice de markup HTML + CSS sunt folosite pentru mici parți ale documentului dinamic în cazul plugin-urilor de jQuery. Pentru aceste plugin-uri nu este necesar foarte mult cod JavaScript ci strictul necesar pentru configurarea lor. Aceasta abordare functioneaza în cazul implementarii interacțiunii low-end, când aceleasi date nu sunt afișate de mai multe ori și unde fiecare acțiune declanșează un refresh al paginii.

PJAX. reprezintă paginile generate ca HTML. Anumite actiuni ale utilizatorului declanșează cod care înlocuiește parți ale paginii existente cu conținut HTML nou, generat de server care este preluat printr-un apel AJAX. Se folosește PushState sau API-ul de history HTML5 pentru a oferi impresia de schimbare a pagini.

Model-backed views vs Markup-driven views. Framework-urile se pot diviza în două categorii bazate pe urmatoarea distinctie: acele în care majoritatea actiunilor se fac în markup și cele în care dacă majoritatea actiunilor au loc în cod.

[Data în JS models] [Data în JS models]
[Model-backed views] [Markup accesses models]

Model-backed: În cazul acestei abordări, modelele reprezintă punctul de pornire. Acestea trebuie instantiate, după care legate la view. Instantele de view-uri sunt atasate la DOM și randeaza conținutul prin pasarea datelor de model într-un template:

var model = new Article({ name: ‘Paper’, price: ‘13.2’}),

view = new **ArticleView**(model);

În mare ideea constă în faptul ca avem modelul care este legat de view în cod.

Markup-driven: De asemenea și aici avem view și model dar relația dintre ele se inverează.View-urile sunt declarate în general prin scrierea de markup de exepmplu:

{{view **ArticleView** }}
{{=window.model.name}}
{{/view}}

Idea din spatele acestei abordări constă în existent unui system de templating care genereaza view-uri și acestea acceseaza variabilele direct printr-un mecanism oferit de framework. În cazurile simple, probabil nici nu este nevoie de o instanța accesibila directă a unui view.

4.3.1.5.Model layer

Layer-ul de model este similar în cadrul mai multor aplicații single-page deoarce nu sunt foarte multe moduri de a privi aceasta problema. Rezolva problema reprezentarii datelor în seturi de date. Este nevoie de de un mecanism de a încarcă datele și preferabil de a salva(cache) o parte din date pentru a evită reincarcarea datelor pe care le avem deja. Daca ele există ca mecanisme separe sau ca parte a unui singur model mare este o chestie de detaliu. Diferența majora este reprezentată de modul în care sunt tratate colectile și acest lucru se datoreaza alegerilor care s-au facut în layer-ul de view(cu observables se dorește observarea șirurilor iar cu evenimente,cea a colecțiilor)

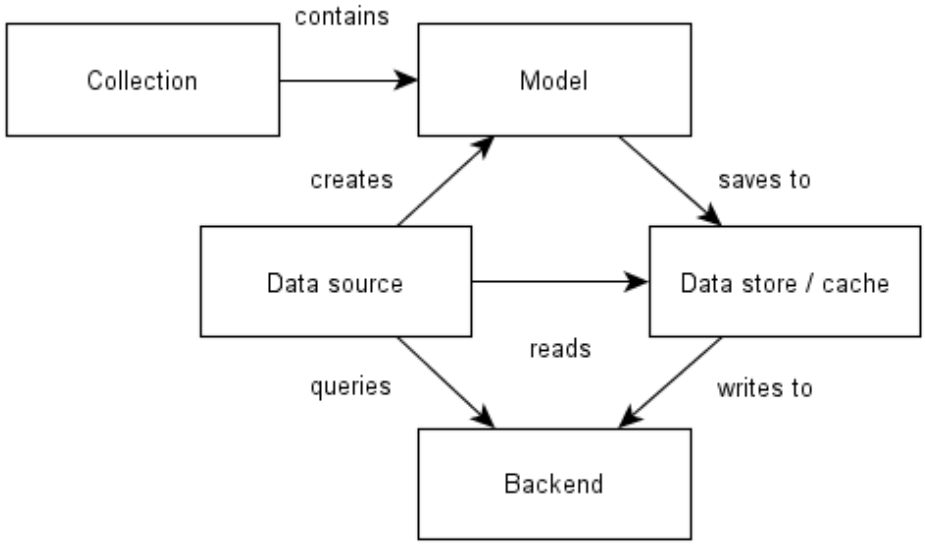

Figura 4.4 Traseul datelor în cadrul layer-ului de model

Sursa de date. O sursă de date (backend proxy sau API) este responsabil pentru citirea din backend folosind un API simplu și puternic. Accepta date JSON și returneaza obiecte JSON care sunt converite în modele. Cautările după ID pot fi aduse direct din cache dar sunt necesare interogari complicate către backend pentru a cauta seturi întregi de date.

Model. Are rolul de a depozita datele, de a emite evenimente în momentul în care datele se schimba, poate fi serializat și persistat. Modelul conține datele (atributele) și poate fi transformat într-un JSON pentru a cere sau trimite date la backend. Un model poate avea mai multe asocieri, poate avea mai multe reguli de validare și poate avea mai multi abonati.

Colectile conțin elemenete,emit evenimente în cazul în care un element este adaugat sau sters și au o ordine definita a elementelor. Existenta colectiilor usureaza manipularea seturilor de date. O colectie poate reprezenta un subset de modele, de exemplu, o lista de utilizatori. Colectiile sunt ordonate și reprezintă o selectie particulara de modele cu un anumit scop, în general de a desena view-uri. O colectie poate fi implementata ca un model de colectii care emite evenimente sau ca un sir observator de elemente. Implementarea aleasa este dependența de modul în care este structurat layer-ul de view. În cazul în care view-urile trebuie să contina propria lor logica atunci se dorește implementarea utilizand colectii care urmaresc modelul, asta fiindcă colectiile conțin modele cu scopul da a fi randate. Daca este nevoie ca view-urile să contina în general markup, cu alte cuvine să nu fie "component" dar să fie „legaturi subtiri" care se refera la alte elemente prin numele lor în scopul global, atunci trebuie utlizate observere de siruri. Din moment ce view-urile nu conțin comportament, vom avea controllere care vor reprezenta legatura dintre view-uri multiple.

Cache-ul de date stochează modelele după id, lcuru care permite accesarea ulterioară mai rapida. De asemenea are rolul de a intermedia salvarea de date pentru backend și de a preveni instanțierea multipla a aceluiasi model. Este folosit în managementul ciclului de viata a unui model, în salvarea,updatarea și ștergerea datelor reprezentate în modele . Există posibilitatea ca modelele să nu fie în concordanta cu ultima lor versiune, să nu fie foliste și astfel pot fi pre incarcate pentru a face accsul ulterior la date mult mai rapid. Diferența dintre o colectie și cache este faptul ca cache-ul nu este ordonat, datele din cache reporezentand toate modele care au fost incarcate și pastrate de către codul de pe client.

4.3.2. Node.js

Node.js este un mediu de rulare JavaScript open source, multi-platforma pentru aplicații server-side . Node.js este contruit pe engine-ul javascript Google V8, ceea ce inseamnă ca aplicațiile Node.js sunt scrise în javaScript și utilizeaza sitaxa similara cu aplicațiile JavaScript de front-end , incluzand obiecte , funcții și metode. Node.js include o librarie ce permite aplicațiile să se comporte asemeni unui web-server . Multumita arhitecturii bazate pe evenimente și a unui non-blocking API de intrare ieșire care optimizeaza transferul datelor , Node.js excelează când vine vorba de comunicarea în timp real.

Avantajele Node.js :

- Este foarte rapid și accesibil.
- Este ușor de configurat și de personalizat
- Vine cu manager de pachete (npm) care conține 140.000+ de pachete și managementul dependintelor se face foarte ușor.
- Permite construirea unei aplicații JavaScript „cap-coada"

- Este capabil să manipuleze un numar imens de conexiuni simultane cu un randament ridicat, de aceea exclează în consturirea aplicațiilor rapide și scalabile.
- Permite reutilizarea codului atât pe partea de client cât și pe cea de server.

Dezavantajul lui Node.js este faptul că nu este conceput pentru calcule dificile, deoarece orice operație care necesită un aport mare al procesorului anulează randamentul ridicat al acestuia iar toate requesturile primite sunt blocate deoarce thread-ul este ocupat cu rezolvarea calculelor .

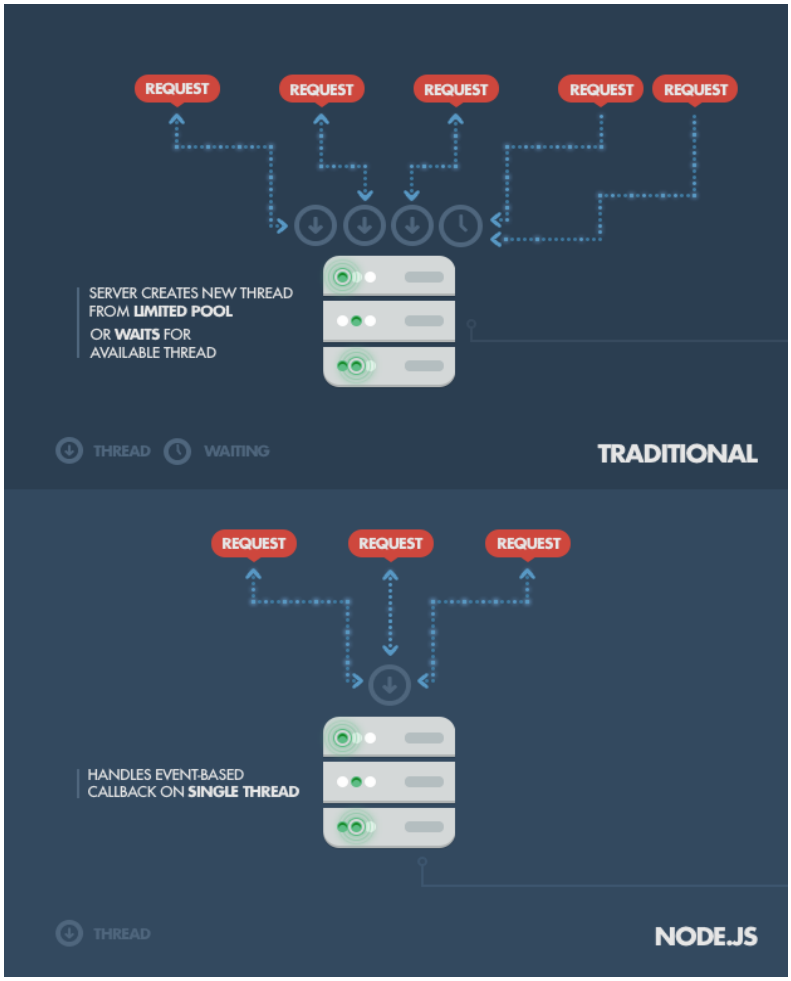

Figura 4.5 Node.JS vs Abordarile traditionale

Blocking vs Non-blocking I/O

Exemplu blocking: Un exemplu de blocking este cum serverele web ca cele în Java sau PHP gestionează requesturile . Daca codul face ceva care blocheaza execuția, cum ar fi citirea dintr-o baza de date , atunci codul „astepta” la linia respectiva până când operatia de citire este finalizata . În perioada respectiva se ocupa memoria masinii dar este ocupat și un thread care practic nu face nimic . Pentru a gestiona mai multe request-uri cât timp thread-ul respectiv sta se pot creea mai multe thread-uri sau dacă există un „load

balancer", requesturile pot fi trimise către instantele disponibile . Toate aceste operații conduc la un consum mai mare de memorie și de putere de procesare .
Exemplu non-blocking: În contrast , serverele non-blocking ca cele create în Node.js, folosesc un singur thread pentru toate requesturile. Cand requesturile ajung la server , fiecare request este servit unul după celalalt . În cazul în care codul serivit trebuie să interogheze baza de date , acesta trimite un request spre baza de date . Pentru a nu astepta după răspuns trimite un callback spre o a doua coada astfel codul continua să ruleze. În momentul în care baza de date returneaza datele , callback-ul este pus într-o a treia coada unde sunt executiile în asteptare. Cand sistemul nu face nimic (stiva este goala), preia un callback din a treia coada și îl execută.

4.3.3. AngularJS

AngularJS este un framework JavaScript open-source sponsorizat și mentinut de Google. Scopul acestui framework este de a pune la dispoziție tool-uri care au fost disponibile doar pentru development-ul server-side. AngularJS extinde HTML prin elemente customizabile, atribute, clase sau comentarii (denumite directive). Directivele sunt elemente din DOM care sunt randate de compilatorul de HTML al AngularJS și carora le este atasat un anumit comportament respectivului element al DOM-ului sau care transforma elementul DOM și copii acestuia.

Acestea sunt particularitățile puse la dispoziție de AngularJS:

- Date disponibile în ambele sensuri atât pentru layer-ul de view cât și pentru cel de model
- Engine de templating
- O versiune basic de jQuery denumita jqLite
- Pattern MVC care ajuta la divizarea aplicației în trei arii distincte: datele(model), logica care operează cu datele(controller), și logica care afiseaza datele (view).
- Management al dependintelor.
- Sistem de rutare deep-link care permite encodarea starii aplicației în URL astel oferind posibilitatea de a fi restaurat la aceeași stare.
- Servicii pentru comunicarea cu serverul REST

AngularJS este un framework utilizat pentru construirea de aplicații client-side web complexe și performante. Procesul de development pune accent pe dezvoltarea unei aplicații ușor de extins, de testat, și mentenabila. În figura 4.6 de mai jos se poate observa arhitectura unei aplicații dezvoltate utilizand stack-ul MEAN.js și rolul pe care îl al framework-ul AngularJS în cadrul sistemului:

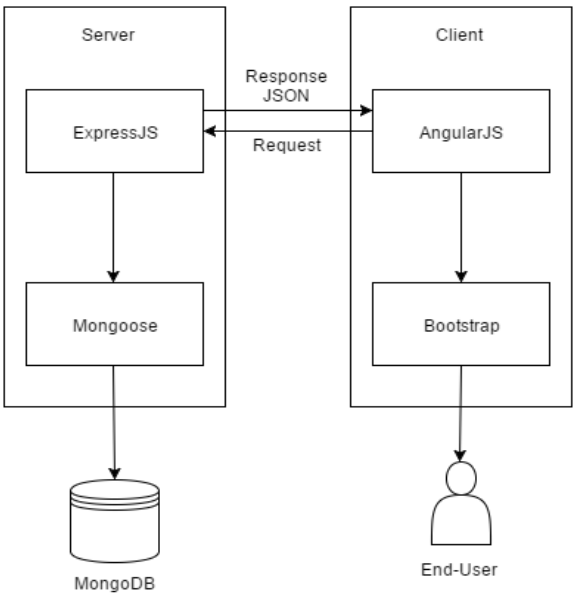

Figura 4.6 Arhitectura MEAN stack

La startul unei aplicații, browser-ul încarcă HTML-ul și îl parseaza în DOM împreună cu script-ul Angular.Js. După ce DOM-ul e incarcat, AngularJS cauta directiva ng-app care reprezintă locatia din DOM unde este initializata aplicația. Daca un modul este definit în directiva, acesta este folosit pentru a configura serviciul $injector care creaza seriviciul $compile și $rootScope. După aceea $compile complieaza DOM-ul și creeaza un link între acesta și $rootScope, urmând să se execute directivele ramase(dacă este cazul).

Argument: S-a folosit framework-ul AngularJS, deoarece este unul dintre cele mai populare framework-uri de JavaScript pentru partea de Front-End și are o abordare orientată obiect împărțind componentele de front-end în trei nivele bine definite(Model, View, Controller).

4.3.3.1. Two way data-binding

În momentul în care view-ul isi schimba valoarea și modelul se schimba iar când modelul isi schimba valoarea, la randul lui view-ul se modifica. Conceptul de two-way data-binding poate fi explicat mai ușor printr-un exemplu:

```
<input type=”text” ng-model=”example”>
<p> {{example}} </p>
```

În cazul în care utilizatorul schimba valoarea modelului prin input-ul din view, valorea modelului este stocata într-o variabila. Elementul de tip paragraf se va actualiza automat pentru a reflectă schimbarea valorii variabilei de model. De asemenea view-ul se va modifica dacă variabila va fi modificata manual din JavaScript.

4.3.3.2. Template-uri HTML

Multe alte framework-uri JavaScript MVC folosesc un sistem de templating care se bazeaza pe HTML cu markup special care este dificil de utilizat pentru dezvoltator. AngularJS nu se bazeaza pe motoare de templating externe ci foloseşte un sistem de templating care este construit pe baza HTML-ului şi care utilizeaza într-un mod foarte intuitiv ng-attributes . Browser-ul parseaza HTML-ul şi cauta o directiva care atunci când se execută creeaza o legatura între view şi model.

AngularJS template:

```
<table>
      <tr>
            <th>Titlu</th>
            <th>Creat</th>
            </tr>
            <tr ng-repeat=”item în items”>
                  <td>{{item.title}}</td>
                  <td>{{item.created}}</td>
            </tr>
      </tr>
</table>
```

Acest template va afişa un tabel cu articole care va avea 2 coloane : titlu şi data creeari. Se va execută o iterare prin toate articolele şi se va afişa cate un rand pentru fiecare articol în care vom putea vizualiza titlul şi data creeari.

4.3.3.3. Deep Linking

În cadrul unei aplicaţii single-page, este foarte importantă menţinerea stărilor aplicaţiei în URL astfel userii avand posibilitatea de a distribui sau salva linkurile în diferite stări ale aplicaţiei. AngularJS foloseşte API-ul de istoric pus la dispoziţie de HTML5. Aceasta functionalitate poate nu pare foarte importantă momentan, dar este extrem de utila în această era modernă în care distribuirea în mediul online este des intalnita.

4.3.3.4. Directive

Directivele oferă posibiliatea de a extinde funcţionalitatea HTML . În timp ce AngularJS are o colectie predefinită de directive, functionalitatiile acestora pot fi extinse prin directive custom oferind posibilitatea utilizatorului de a-şi creea propriul DSL (Domain Specific Language). Exemplu de directive:

HTML:

```
<div editableContent=”true” ng-model=”content”> Edit this </div>
<p> {{content}} </p>
```

JavaScript:

```
angular.module('directive',       []).directive('editablecontent',
function()
{
return {
      require: 'ngModel',
      link: function(scope, elm, attrs, ctrl) {
       // view -> model
```

```
            elm.bind('blur', function() {
                  scope.$apply(function() {
                        ctrl.$setViewValue(elm.html());
                  });
            });

            // model -> view
            ctrl.$render = function(value) {
           elm.html(value);
           };

           // load init value from DOM
           ctrl.$setViewValue(elm.html());
                  }
           };
});
```

4.3.3.5. *Dependency Injection*

În AngularJS este încurajată injectarea de dependințe și este una din componentele care îmbunătățește testabilitatea aplicației. Ideea acestui framework se bazeaza pe conceptul de modularitate și reutilizabilitate.

4.3.3.6. *Serviciul $http*

Este un serviciu core din AngularJS care faciliteaza comunicarea remote cu serverele HTTP printr-un obiect XMLHttpRequest sau prin JSONP.

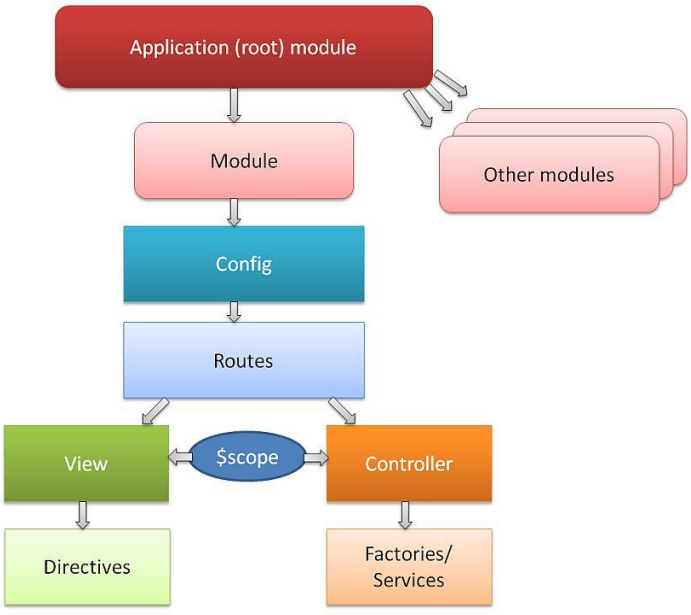

Figura 4.7 Arhitectura unei aplicații AngularJS.

4.3.4. *ExpressJS*

ExpressJS are rolul de a procesa requesturile și în același timp are și cel de rutare, reprezentand practic layer-ul de middleware al aplicației. În momentul în care un request

este primit de ExpressJS, se declanșează diferite funcții care pot fi considerate parte a middleware-ului. Middleware-ul este reprezentat de orice funcție care este invocata de layer-ul de rutare al ExpressJS înainte de tratarea(handle) finala a request-ului. Aceste funcții poarta numele de „middleware stack" fiindcă sunt invocate în aceeași ordine în care sunt adaugate.

Exemplu:

```
app.get('/', function(req, res) {
 res.send('Hello World!');
});

app.get('/help', function(req, res) {
  res.send('Nothing to see here');
});
```

Aplicația de mai sus va răspunde cu „Hello world!" pentru un request spre ruta „/" și „Nothing to see here" în cazul rutei „/help". Daca dorm să inregistram și să urmarim fiecare request, trecem prin fiecare ruta și adaugam logica de afisare sau folosim middleware-ul.

Exemplu:

```
var app = express();

app.use(function(req, res, next) {
 console.log('%s %s', req.method, req.url);
next();
});

app.get('/', function(req, res, next) {
 res.send('Hello World!');
});

app.get('/help', function(req, res, next) {
res.send(Nothing to see here');
});
```

În exemplul de mai sus am adaugat o funcție noua care este invocata la fiecare request prin app.use(). Există anumite proprietati importante ce tin de funcția nou creată:

-Middleware este o funcție similara cu cea de manipulare a rutelor, și este invocata într-un mod similar.

-Semnatura functiei este identica cu cea folosita în rute.

-Am adugat aceasta funcție înainte de cele două funcții de manipulare a rutelor fiindcă vrem să se execute înainte de acestea.

-Avem acces la aceleasi obiecte de "request" și "response" care isi vor gasi drumul spre rute.

-Am folosit un al treilea parametru denumit "next" ca o funcție care are rolul de a indica faptul ca middleware-ul este finalizat.

-Avem posibilitatea de a adaugă mai multe la middleware folosind același API.

Middlware-ul este folosit pentru a execută diferite sarcini cum ar fi body parsing pentru request-uri JSON url-encoded, parsarea de cookies pentru manipulari de baza ale cookie-ului sau chiar pentru dezvoltarea de module JavaScript. Cand adaugam noi funționalități la middleware trebuie să fim atenti la modul în care acestea ar putea afecta obiectele de request și response. Pana în momentul de față am explicat modul în care este adaugat middleware-ul pentru fiecare request dar avem posibilitatea de a limita

middleware-ul la anumite căi(paths). În cazul în care rutele care derivă de la "/users" au nevoie de un middleware special, este necesar să folosim un parametru în cadrul functiei app.use().

Exemplu:
```
app.use('/users', function(req, res, next) {
    // invocat la fiecare request care începe cu /users
      next();
    });
app.get('/users/daily', function(req, res, next) {});
```

Există totusi o diferență între modul în care este interpretat acest parametru opțional în cadrul funcțiilor app.use() și app.get(). În cazul utilizarii parametrului în app.get() atunci funcția noastra va fi invocata atunci când cineva viziteaza exact ruta „/users", în schimb dacă este utilizat în cadrul app.use(), orice request care începe cu „/users" va invoca funcția noastra. Pentru a determina sub-paths ale „/users" se va utiliza proprietatea req.path care este setată de către layer-ul de rutare al ExpressJS. Cand este procesat un request pentru ruta „/users/daily", req.path va avea valoarea „/daily" în middleware. Există totusi câteva capcane ale middleware-ului:

- Ordinea conteaza: Urmatorul exemplu demonstrează faptul ca middleware-ul nu va execută reques-uri de GET, dar va gestiona request-urile de POST:
```
app.get('/', function(req, res) { res.send('handling get');
});
   app.use(function(req, res, next) {
     next();
    });
   app.post('/', function(req, res) { res.send('handling p
   ost');
});
```

Chiar dacă nu reprezintă mereu o capcană, este important să intelegem faptul ca app.use() este invocat pentru fiecare verb HTTP, middleware-ul este procesat în funcție de tratarea rutelor. Din moment ce handler-ul pentru GET răspunde request-ului, middleware-ul nostru nu va rula pentru request-urilde GET, dar handler-ul de POST este după middleware ceea ce ii permite middleware-ului să fie invocat înainte.

- Lipsa functiei next() : În cazul în care am adaugat middleware și testam pagina sau API-ul și avem impresia ca acesta sta sau nu răspunde, atunci cel mai probabil am uitat să apelam next(). Fiindca rutarea din ExpressJS nu stie dacă middleware-ul este gata, atunci acesta nu va aplea nicio ruta sau middleware.
- Suprascrierea proprietaților: Fiecare argument request sau response ale middleware-ului sau rutelor au aceeași instanța. Daca avem două middleware-uri care modifica proprietatile obiectului în moduri diferite, există posibilitatea ca acest lucru să genereze erori în aplicație.

Argument: Principalul motiv pentru care am folosit framework-ul ExpressJS constă în simplitatea oferita de acesta fiind cel mai popular framework de NodeJS. Un alt motiv este reprezentat de faptul ca face parte din stack-ul MEAN care permite dezvoltarea de aplicații web folosind un singur limbaj de programare (Javascript).

4.3.5. REST API

REST (Representational State Transfer) reprezintă un model arhitectural utilizat pentru crearea de servicii web. Sistemele REST comunica în mod obisnuit prin Hyper Transfer Protocol prin aceleasi verbe HTTP (GET, POST, PUT, DELETE) prin care bowserele web trimit și primesc date de la servere. Interfețele REST implică colectii de resurse cu identificatori, de exemplu: /userdetails/ovidiu, care pot fi accesate folosint verbe standard, cum ar fi : GET /userdetails/ovidiu.

Motivul pentru care a fost realizată o asemenea arhitectura a pornit de la faptul ca seviciile de tip RPC și SOAP sunt mai complexe. Practic simplitatea WEB a reprezentat sursa de insipiratie a serviciilor de tip REST . REST presupune construirea unui serviciu web utilizand HTTP, XML și URI. Intr-o arhitectura de tip REST se deosebesc două trăsături importante : datele asupra carora clientul ii spune serverului să opereze se gasesc în URI, operatia pe care serverul o execută asupra datelor este descrisa direct de metoda HTTP.

4.3.5.1. Proprietati arhitecturale

Proprietatile arhitecturale afectate de constrangeri ale stilului arhitectural REST sunt:

- Performanta: interacțiunea componentelor și eficienta retelei pot fi factori dominanti în performanta perceputa de utilizator.
- Scalablitatea: pentru a suporta un numar mare de componente și interacțiuni între componente
- Simplitatea interfețelor
- Portabilitatea componentelor

4.3.5.2. Constrangeri arhitecturale

Constrangerile REST sunt :

- **Stateless:** fiecare request de la client conține toate informațiile necesare pentru a servi request-ul, iar starea de sesiune este păstrată în client. Sesiunea poate fi transferata de server unui alt serviciu, de exemplu o baza de date, pentru a persista starea o anumită perioada de timp și de a permite autentificarea. Clientul începe să trimita request-uri când este gata să facă o tranziție la o noua stare. Cat timp unul sau mai mai multe request-uri nu sunt finalizate, clientul este considerat a fi într-o stare de tranziție. Reprezentarea fiecărei stări a aplicației conține link-uri care pot fi utilizate ulterior când clientul decide să initieze o noua tranziție a stărilor.
- **Client-Server:** există o interfața uniforma care separa clienții de servere. De exemplu clienții nu sunt interesati de modul în care sunt stocate datele, atributie care revine serverului, astfel imbunatatindu-se portabilitatea codului client. Totodata acest lucru duce la simplitatea și scalabilitatea serverelor.
- **Cachable:** În mediul web, clienții și intermediarii pot să cachuiasca raspunsurile. Un management bun al cachingului reduce sau elimina complet o parte din interacțiunile client-server, îmbunătățind scalabilitatea și performanța.

- **Sistem bazat pe layere**: Un client nu stie dacă este conectat direct la server sau există un intermediar până la server. Serverele intermediare pot imbunatatii scalabilitatea prin implementarea unui load balander și prin punerea la dispoziție a resurselor cached comune.
- **Interfața uniforma**: Reprezintă constrangerea fundamentala în cazul dezvoltarii oricarui serviciu REST. Interfața uniforma simplifica și decupleaza arhitectura, lucru care permite ca fiecare parte să evolueze separat.

4.3.6. Socket.IO

Un aspect important de care trebuie să tindem cont atunci când dezvoltam o aplicație web este de a asigura feedback în timp util utilizatorilor.Totul a inceput cu introducerea XMLHttlRequest de către Microsoft, care a devenit în timp AJAX. Long-polling a reprezentat pentru mult timp modul standard de a prelua date de la server în cadrul unei aplicații web, chiar dacă nu a reprezentat soluția ideala. Long-polling implică trimitrea periodica de request-uri HTTP pentru date, introducand latență și solicitand mai mult server-ul. În anul 2011, IETF a standardizat WebSockets, oferind posibilitatea dezvoltatorilor de a trimite și primi date printr-un socket TCP. Toate browserele importante au inceput să ofere suport pentur acest standar și dezvoltatorii au inceput să îl folosească în proiectele lor.

Socket.IO reprezintă un layer de comunicare bazat pe evenimente bi-directionale utilizat în cadrul aplicatilor web în timp real. Acesta librarie a fost construita pe baza Engine.IO și abstractizeaza multe din transporturile de date, inclusiv AJAX long-polling și WebSockets într-un singur API. Astfel dezvoltatorul are posibilitatea de a trimite și primi date fara a fi preocupat de compatibilitatea dintre browsere. Socket.IO a ajuns la versiunea 1.0 în 28 Mai, 2014. Inainte de versiunea 1.0 Socket.IO era impartit în două parți: o implementare care se ocupa cu manipulearea transporturilor și un API de nivel inalt. Implementarea care se ocupa cu manipularea transporturilor a fost mutata într-un framework separat : Engine.IO. Acest lucru permite dezvoltatorilor să creeze noi API-uri și proiecte pentru aplicațiile web în timp real. Socket.IO pune la dizpozitia dezvoltatorului componente clinet-side și server-side cu API-uri similare. Din punct de vedere **server-side**, Socket.IO functioneaza prin adaugarea de ascultatori de evenimente la o instanța a http.Server. Pentru a adaugă support Socket.IO unei instante http.Server avem urmatorul exemplu:

```
var server = require("net").createServer();
var io = require("socket.io")(server);
var handleClient = function (socket) {
    // avem o conexiune client
  socket.emit("item", {name: "nodesource", quantity: "1"});
};
io.on("connection", handleClient);
server.listen(8080);
```

Este de asemenea posibil să atasam un server Socket.IO unui alt framework HTTP, cum ar fi ExpressJS. Avem ca exemplu:

```
var app = require("express");
var server = require("http").Server(app);
var io = require("socket.io")(server);
io.on("connection", handleClient);
app.listen(8080);
```

Socket.IO este compatibil cu majoritatea framework-urilor care expun instanța de http.Server. Pe partea de client-side, serverul HTTP va expune libraria client la „/socket.io/socket.io.js”. Pentru a ne conecta la serverul nostru Socket.IO, trebuie să adaugam în tag-ul de body urmatorul cod:

```
<script src="/socket.io/socket.io.js"></script>
<script>
    var socket = io.connect("http://localhost");
</script>
```

Variabila globala socket este asemenătoare cu un obiect EventEmitter. Există posibilitea de a atasa un ascutlator care să se activeze în momentul în care există conexiune cu serverul. Avand în vedere ca obiectul de socket actioneaza ca un EventEmitter atât pentru server cât și pentru client, avem posibilitatea de a emite și asculta evenimente într-un mod bi-directional. Acem ca exemplu trimiterea unui eveniment „stire” de către server și ascultarea acestuia client-side:

Server-side:

```
io.on("connection", function (socket) {
var news= {publisher: "CNN", title: "Hello, world!"};

      var interval = setInterval(function () {
      socket.emit("news ", news);
}, 1000);

      socket.on("disconnect", function () {
            clearInterval(interval);
      });
});
```

Client-side:

```
socket.on("news", function(tweet) {
console.log("news from", news.publisher);
console.log("title:", news.title);
});
```

Se poate trimite și primi orice obiect serializabil JSON(include string-uri, numere, array-uri și boolean). Pe langa trimiterea de obiecte serializabile JSON, se poate crea un stream între browsere de la server. Mai jos avem un exemplu de trimitere a conținutului pe care îl are scriptul server-side :

```
var fs = require("fs");
var ss = require("socket.io-stream");

io.on("connection", function (socket) {
    ss(socket).emit("script",
fs.createReadStream(__filename));
});
```

Pentru a consuma fiecare bucata de date în browser, vom asculta după evenimentul „data”:

```
socket.on("script", function (stream) {
    var buffer = "";
    stream.on("data", function (data) {
        buffer += data.toString();
    });
    stream.on("end", function () {
        console.log(buffer);
    });
});
```

Argument: Am utilizat tehnologia Socket.IO fiindcă oferă suport pentru reconectare automata ceea ce indica faptul ca nu trebuie să ne ingrijoram în cazul unei caderi a conexiunii la Internet. Un alt atu constă în suportul de „chat rooms" ceea ce simplifica și accelereaza procesul de dezvoltare al aplicațiilor web în timp-real.

4.3.7. MongoDB

MongoDB NoSQL reprezintă un trend nou în dezvoltarea de baze de date și se refera în general la baze de date fara o schema fixa. Asemenea baze de date nu au o siguranta mare în cazul tranzactiilor, este o baza de date NoSQL open-source dezvoltata de 10gen în C++. Bazele de date NoSQL sunt mai rapide când vine vorba de accesarea datelor și se scalaeaza mai bine în comparatie cu bazele de date relationale. MongoDB apartine categoriei de baze de date bazate pe documente.

MongoDB este o baza de date care consistă într-un set de baze de date în care fiecare baza de date conține colectii multiple. Fiindca MongoDB lucreaza cu scheme dinamice, fiecare colectie poate conține diferite tipuri de obiecte . Fiecare obiect, denumit și document este reprezentat printr-o structura JSON : o lista de valori key-value. Valoarea poate fi de trei tipuri : valoare primitiva, un array de documente sau o lista de perechi key-value (document).

Reprezentarea UML a organizarii datelor într-o baza de date MongoDB pornind de la colectii ce conțin un set de documente al caror conținut sunt perechi cheie-valoare :

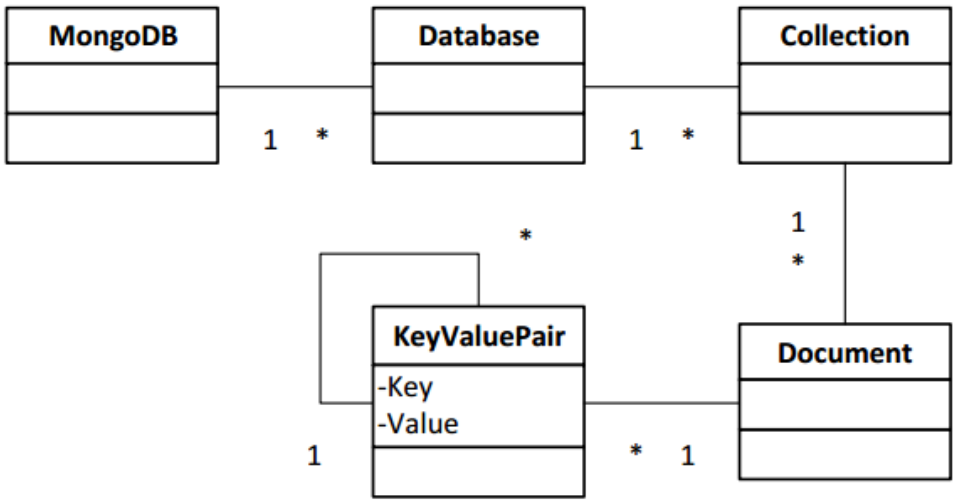

Figura 4.8 Reprezentarea UML a organizarii datelor în MonoDB

Pentru a interoga aceste obiecte, clientul poate seta un filtru pentru colectiile exprimate ca list de perechi key-value. Interograrile sunt structurate sub forma de JSON, astfel o interogare complexă poate ocupa mult mai mult spatiu în comparatie cu o interogare similara pentru o baza de date relationala. MongoDB oferă posibilitatea de a interoga campuri imbircate. De asemenea MongoDB necesită utilizarea tipului corect: dacă introducem o valoare de tip integer într-un document, trebuie să interogam valoarea de tip integer. Baza de date suporta indecsi : oferă posibilitatea de a crea indecsi descrescatori, crescatori și geo-spatiali. Acesti indecsi sunt implementati ca B-tree

indexuri. Coloana „_id” care poate fi gasita în radacina fiecarui document este mereu indexata.

MongoDB oferă suport pentru două tipuri de replicare : master-slave și replica sets. În cazul replicarii master-slave, master are acces deplin la date și scrie fiecare schimbare către slaves. Replica sets functioneaza similar cu master-slave, doar ca este posibil să alegem un mou master în cazul în care master-ul original nu functioneaza (a picat). Un alt atu al acestei baze de date este automatic sharding asfel oferind posibilitatea de a paritiona datele în noduri multiple. Singura persoana care poate să defineasca un key de sharding este administatorul. Trebuie definit un sharding key pentru fiecare colectie ceea ce defineste modul în care sunt paritionate documentele conținute.

Tranzactiile nu sunt suportate direct de către MongoDB dar există 2 variante ocolitoare : operațiile atomice și commiturile în două faze . Operațiile atomice permit execuția de operații multiple într-un singur apel. MongoDB nu oferă suport single server pentru persistenta datelor, ceea ce inseamnă ca este nevoie de replicari multiple pentru a evită pierderi de informații în cazul în care unul dintre servere inceteaza să funcționeze.

Argument: Motivul utilizarii bazei de date MongoDB pe langa avantajele mentionate mai sus, este reprezentat de faptul ca se preteaza pentru persistenta claselor, ele fiind serializate în format JSON și persistate în MongoDB.

Capitolul 5. Proiectare de Detaliu și Implementare

Capitolul de față reprezintă o detaliere a sistemului, alături de implelementare, arhitectura și a componentelor implicate.

5.1. Arhitectura sistemului

Sistemul dezvoltat este implementat utilizand o arhitectura client-server, asa cum se poate observa și în figura 5.1, care ilustreaza arhitectura conceptuală a sistemului. Scopul acesteia este de a oferi o viziune în ansamblu asupra modului în care se realizează comunicarea dintre client și server, practic între client și furnizorii de servicii, ilustrând formatul datelor care se transmit între cele două componente.

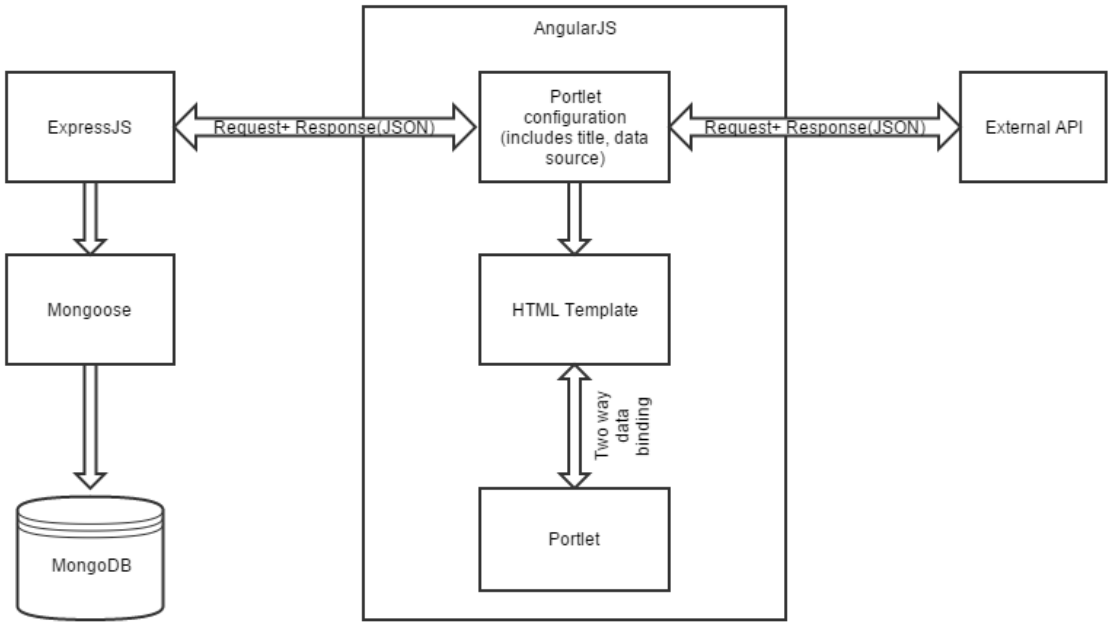

Figura 5.1 Arhitectura sistemului

Arhitectura client-server se bazează pe existența unei aplicații distribuite care partajează sarcinile între furnizorii de servicii numiți servere și cei care solicită servicii, numiți clienți. Această arhitectura este printre cele mai populare printre dezvoltatorii de aplicații web, astfel că s-a impus ca fiind unul dintre cele mai fiabile moduri de dezvoltare a aplicațiilor distribuite.

În cele mai multe cazuri, clienții și serverele comunică între ele prin intermediul Internetului, la fel cum se întâmpla și în cazul acestui sistem, fiindcă ne oferă beneficiul partajarii de date într-un mod sigur și rapid.

Componenta server isi imparte resursele cu clienții, în schimb un client nu isi partajeaza nici o resursa, ci doar solicita partajarea de resurse din partea serverului. Astfel, clienții initializeaza sesiuni de comunicare cu serverul și asteapta raspunsul acestuia. Partajarea resurselor reprezintă faptul ca serverele raspund cu infomtii utile clientului, iar clientul primeste informatia solicitata. Un alt avantaj este faptul ca serverul nu expune informații legate de platforma software sau hardware alea masinilor de pe care ruleaza, ci doar o interfață transparentă pentru clienți. În cazul de față acest lucru se

obține cu ajutorul unui API (Application Programming Interface), prin care clientul și serverul schimbă informații sub format JSON.

Datorita faptului că serverul pune la dispoziția clientului informații, deducem faptul că serverele au o putere mare de calcul, execută procesari intense și au resurse hardware puternice. Serverele trimit clientilor raspunsuri pe baza cererilor facute de acestia, lucru care menține o balanță a responsabilităților: procesarile complexe sunt executate de server iar cele mai puțin solicitante sunt executate de clienți care dispun de resurse hardware limitate.

5.1.1. Rolul componentelor sistemului

În cele ce urmează se vor prezenta rolurile componentelor server și client în cadrul sistemului actual, pentru a clarifica cum a fost conceput acesta.

Asa cum a fost precizat anterior, serverul este o componentă cu o putere de calcul mult mai mare și suportă mai multe operații pe secunda, acest lucru facilitând o procesare mai buna. Coomponenta client dispune de resurse hardware mult mai modeste și este indicat să execute doar operații simple.

5.1.1.1. Componenta server

Componenta server este responsabilă de realizarea unor operații complexe, precum:

- Gestionarea utilizatorilor
- Gestionarea mesajor din chat
- Realizarea de cautări rapide
- Trimiterea raspunsurilor la clienți pe baza cererilor înaintate de aceștia

Într-un final toate acestea se rezumă la gestionarea eficientă a resurselor hardware care sunt limitate și a bazelor de date.

5.1.1.2. Componenta client

Componenta client este implementată cu scopul de a fi la fel de eficientă ca și în cazul componentei server dar operațiile pe care le realizează nu sunt la fel de complexe. Fiindca experiența pe care o are utilizatorul în momentul utilizării aplicației este foarte importantă și datorită faptului că sistemul își păstrează o mare parte din funcționalități chiar și când nu există o conexiune la Internet implementarea acestei componente trebuie să fie eficientă.

Astfel, componenta client este responsabilă de urmatoarele operații:

- Parsarea informațiilor primite de la server
- Afișarea informațiilor în urma parsărilor
- Persistarea locală a anumitor informații
- Detecția stării de conexiune la Internet

Se poate observa faptul că operațiile realizate sunt repartizate uniform și în funcție de capabilitățile fiecărei componente. Astfel, verificarea rezultatelor se realizează mult mai ușor atât pe partea de server cât și pe cea de client, deoarece avem posibilitatea de a urmării fiecare operație.

5.2. Nivelul de comunicare și servicii

O mare parte din funcționalitățile aplicației se bazeaza pe existența unei conexiuni stabile la Internet. Comunicarea între client și server se realizează pe baza unor cereri exprimate de client, prin intermediul protocoloului HTTP, urmate de un răspuns de la server. Acest lucru se realizează prin obiecte XMLHttpRequest care este un API pentru extragerea de resurse. API-ul REST face posibil contorlul întregului sistem prin request-uri HTTP remote. Cele mai semnificative operații suportate de către sistem se regăsesc în tabelul de mai jos:

Nume	**Utilizator**	**Descriere**
Adaugă utilizator	Neautentificat	Crează un nou utilizator în cadrul sistemului.
Modifică utilizator	Autentificat	Modifica datele unui utilizator existent
Verificare utilizator existent	Neautentificat	Verifică dacă există un utilizator cu același username în baza de date a sistemului.
Autentificare	Neautentificat	Autentificare a utilizatorului
Sign out	Autentificat	Sign out al utilizatorului
Afișare task-uri	Autentificat	Afișează task-urile într-un portlet
Schimbare status task	Autentificat	Schimbare status task din „completed” în „uncompleted”
Ștergere task	Autentificat	Șterge un task din sistem
Adăugare fișiere	Autentificat	Adaugare de fișiere la un anumit task
Verifică Sesiunea	Autentificat	Verificarea stării sesiunii
Creează Sesiunea	Autentificat	Crearea unei sesiuni
Șterge Sesiunea	Autentificat	Ștergerea unei sesiuni.

Tabel 5.1 Operatii posibile

În tabelele de mai jos sunt detaliate o parte din request-urile facute către API:

Metoda	Adaugă utilizator
Utilizator	Neautentificat
Descriere	Creaza un nou utilizator în cadrul sistemului
HTTP request	POST /auth/users
URI params	-
Query params	-
Post params	Obiectul de tip utilizator
Response	201 Created
Error responses	500 Internal Server Error 400 bad request

Tabel 5.2 Descrierea metodei REST de adăugare a unui utilizator

Metoda	Modificare utilizator
Utilizator	Autentificat
Descriere	Modifică un nou utilizator în cadrul sistemului
HTTP request	PUT /auth/users/{userId}
URI params	id: id-ul utilizatorului
Query params	-
Post params	Obiectul de tip utilizator
Response	200 OK
Error responses	500 Internal Server Error 400 Bad Request 404 Not Found

Tabel 5.3 Descrierea metodei REST de modificare a unui utilizator

Metoda	Creare sesiune
Utilizator	Autentificat
Descriere	Creaza o sesiune noua pentru user-ul corespunzator
HTTP request	POST /auth/session
URI params	-
Query params	-
Post params	Obiectul de tip session login
Response	200 OK
Error responses	500 Internal Server Error 400 Bad Request

Tabel 5.4 Descrierea metodei REST de creare a unei sesiuni

Metoda	Verificare utilizator existent
Utilizator	Neautentificat
Descriere	Creaza o sesiune noua pentru user-ul corespunzator
HTTP request	GET /auth/check_username/{username}
URI params	username: numele utilizatorului
Query params	-
Post params	-
Response	Boolean care reprezintă existența utilizatorului (exists:boolean)
Error responses	500 Internal Server Error

Tabel 5.5 Descrierea metodei REST de verificare a unui utilizator existent

5.3. Nivelul de modele

La acest nivel a fost folosita tehnologia Express alături de Mongoose pentru a defini domeniul problemei. MongoDB are o schema flexibila care ne permite să avem o varietate între documente difereite ale aceleasi colectii, de exemplu în cazul în care baza de date evolueaza în timp. Dar aceasta flexibilitate nu inseamnă că trebuie să trecem cu vederea aspectre legate de tip-ul variabilelor.

În figura de mai jos este prezentat modelul de date pentru utilizatori și se realizează prin specificarea proprietaților într-o schema de tip mongoose:

```
var mongoose = require('mongoose'),
  Schema = mongoose.Schema,
  crypto = require('crypto');

var UserSchema = new Schema({
  email: {
    type: String,
    unique: true,
    required: true
  },
  username: {
    type: String,
    unique: true,
    required: true
  },
  hashedPassword: String,
  salt: String,
  name: String,
  admin: Boolean,
  guest: Boolean,
  provider: String
});
```

Figura 5.2 Modelul de date pentru utilizatori

Schema Mongoose de mai sus prezintă structura unui obiect de tip utilizator. Tot în aceasta schemă se poate observa faptul că e-mail-ul este de tip String dar Mongoose ne permite să realizăm funcții de validare indiferent de tipul atributului, după cum se observa în figura 5.3:

```
UserSchema.path('email').validate(function (email) {
  var emailRegex = /^([\w-\.]+@([\w-]+\.)+[\w-]{2,4})?$/;
  return emailRegex.test(email);
}, 'The specified email is invalid.');
```

Figura 5.3 Funcția de validare pentru e-mail

Avand în vedere faptul că utilizatorul este atenționat de existența unui alt utilizator cu același nume în momentul în care isi creaza un cont nou, am fost nevoit să fac aceasta verificare cu ajutorul mongoose după cum se observa în figura 5.4 :

```
UserSchema.path('username').validate(function(value, respond) {
  mongoose.models["User"].findOne({username: value}, function(err, user) {
    if(err) throw err;
    if(user) return respond(false);
    respond(true);
  });
}, 'The specified username is already in use.');
```

Figura 5.4 Verificarea existentei unui utilizator.

Tot în cadrul schemei de utilizator avem 3 metode corespunzatoare schemei pentru utilizator: **encryptPassword()** care encrypteaza parola într-un stream de biti base64, **makeSalt()** funcție care este responsabila de „salted password hashing” și funcția **authenticate()** care returneaza un boolean rezultat al verificarii de egalitate atât ca valoare cât și ca tip de date între parola encriptata și parola hashed.

Un alt model de date care mi s-a parut relevant este cel al task-urilor iar modelul de date Mongoose este prezentata în figura 5.5 de mai jos:

```
var TaskSchema = new Schema({
  name: {
    type: String,
    index: true,
    required: true
  },
  description: {
    type: String,
    default: '',
  },
  slug: {
    type: String,
    lowercase: true,
    trim: true
  },
  state: {
    type: String,
    default: 'in progress',
  }
  creationDate: Date,
  dueDate: [Date],
  creator: {
    type: Schema.ObjectId,
    ref: 'User'
  }
```

Figura 5.5 Modelul de date pentru un obiect de tip task

Mongoose este utilizat și pentru operațiile te tip CRUD iar un exemplul de utilizare pentru modelul de date de tip task este cautarea unui task în funcție de titlu și este prezentat în figura 5.6 :

```
TaskSchema.statics = {
  load: function(id, cb) {
    this.findOne({
      _id: id
    }).populate('creator', 'username').exec(cb);
  }
};

TaskSchema.statics.findByTitle = function (title, callback) {
  return this.find({ title: title }, callback);
}
```

Figura 5.6 Metoda de cautare a unui task în funcție de titlu

Fiecare model de date Mongoose urmează să fie mapat în baza de date.

5.4. Nivelul de controller

Nivelul controller-ului face legatura între inferfata și domeniul aplicației. Controlerul are responsabilitata de a pregăti datele necesare pentru a fi afișate în interfața și de a trimite datele spre domeniu cu scopul de a fi salvate în baza de date.

Luand în considerare faptul că aplicația a fost dezvoltata utilizand stack-ul MEAN am ales să fac o separare a controllerelor în funcție de modelul de date deservit : chat, users, session și task.

Controller-ul răspunzător de chat utilizează libraria Sockets.IO și se folosesteste de modelul de date al utilizatorilor. Acest controller expune patru funcții care pot fi

utilizate în layer-ul de view : **inituser(), joinRoom(), sendMessage** și **leaveRoom().** Mesajele recente sunt păstrate într-o coada sub forma unor array-uri împreună cu mesajele care urmează a fi trimise iar acest lucru se poate observa în figura de mai jos:

```
var messageQueue = (function(size){
  var queue = [];

  queue.push = function(a) {
    if(this.length >= size) this.shift();
    return Array.prototype.push.apply(this, arguments);
  };

  return queue;
})(15);
```

Figura 5.7 Funcție care gestionează coada de mesaje

Funcția **initUser(socket,user)** răspunde de asignarea utilizatorului unei conexiuni și emiterea de către socket al evenimentului de initializare împreună cu datele utilizatorului care s-a conectat. Funcția **joinRoom(socket,user)** are rolul de a reîmprospăta lista de utilizatori care utilizează chat-ul dar și de a afișa toate mesajele recente pentru utilizatorii care doar acum s-au logat în cadrul chat-ului. Socket-ul emite evenimentul de join și face în același timp broadcast cu datele ce conțin numele utilizatorilor și mesajele din coadă. Funcția **sendMessage(socket,user,data)** are rolul de a face broadcast la mesajele trimite de către un utilizator către toți utilizatorii aplicației. Structura unui mesaj poate fi observata în figura 5.8:

```
messageQueue.push({
  author: user,
  body:   data.body,
  date:   Date.now()
});
```

Figura 5.8 Stuctura unui mesaj din chat.

Funcția **leaveRoom(io,socket,user)** a fost creată cu scopul de a anunta restul utilizatorilor momentul în care un utilizator paraseste chat-ul și emite evenimentul de leave.

Controller-ul care răspunde de funcționalitățile ce tin de sesiune este session.js și expune două funcții către layer-ul de prezentare: **login()** și **logout().** Pentru a realiza componenta de login și a spori securitatea aplicației am ales să criptez parola și să folosesc un modul pus la dispoziție de Node.JS : PassportJS. Sesiunea este persistată în baza de date și este ștearsă în momentul în care utilizatorul se deloghează din aplicație. Acest modul de passport trebuie configurat în funcție de necesități și de tipul de autentificare care este utilizat, dacă este local sau sunt folosite servicii externe de autentificare (Google, Yahoo, Twitter). Pentru configurarea locală a fost necesar să stabilesc funcțiile de serializare și deserializare a datelor ce țin de utilizator și care practic setează utilizator la req.user și stabilește o sesiune printr-un cookie care este păstrat în browser. Structura funcțiilor de login și logout poate fi consultată în figura 5.9:

```
exports.logout = function (req, res) {
  if(req.user) {
    req.logout();
    res.send(200);
  } else {
    res.send(400, "Not logged in");
  }
};

/**
 *  Login
 *  requires: {email, password}
 */
exports.login = function (req, res, next) {
  passport.authenticate('local', function(err, user, info) {
    var error = err || info;
    if (error) { return res.json(400, error); }
    req.logIn(user, function(err) {
      if (err) { return res.send(err); }
      res.json(req.user.user_info);
    });
  })(req, res, next);
}
```

Figura 5.9 Functiile de login și logout.

Controllerele pot fi regăsite și nivel de front-end avand în vedere faptul că este utilizat un framework de tip MVC/MVVM. În aceste controllere se regăsește logica ce tine de funcționalitatea portalului și a sub-componentelor acestuia : crearea portalului pornind de la o un model de date JSON acest lucru realizându-se prin funcția **fillStructure()** precedată de **initModel()**. Orice schimbare a structurii portalului apelează funcția **changeStructure(model, structure, isInFullView)**, model-ul reprezentand valoarea sub forma căruia va fi afișat portalul, structure reprezintă structura noua a portalului care o înlocuiește pe cea veche iar parametrul isInFullView are rolul de a verifica dacă portletul este în fullView și va ocupa intreaga suprafață a portalului.

5.5. Nivelul de prezentare

La acest nivel se regăsesc toate componentele responsabile de interacțiunea utilizatorului cu aplicația.

În cadrul aplicației acest nivel a fost realizat folosind tehnologia AngularJS. Acest nivel conține mai multe pagini HTML, care vor interacționa cu clientul, pentru a realiza diferite operații. Fiecare pagină oferă posibilitatea utilizatorului de a realiza anumite operații cum ar fi : crearea unui cont, autentificarea sau utilizarea portalului.

În cadrul nivelului de prezentare pentru fiecare pagină am mapat un controller folosind fișierul „main.js” unde am specificat rutele posibile ale aplicației. Astfel se poate observa faptul că avem trei rute: login, signup si dashboard. Aceste rute reprezintă practic mapări pentru url-urile aplicației (Exemplu:localhost:9001/#/dashboard) iar configurarea este prezentată în figura 5.10.

```
.config(function($routeProvider, localStorageServiceProvider, $httpProvider){

    $httpProvider.interceptors.push('authHttpResponseInterceptor');
    localStorageServiceProvider.setPrefix('adf');
    $routeProvider.when('/dashboard', {
      templateUrl: 'src/partials/dashboard.html',
      controller: 'dashboardCtrl'
    })
  .when('/login', {
    templateUrl: 'src/partials/login.html',
    controller: 'loginCtrl'
  })
  .when('/signup', {
    templateUrl: 'src/partials/signup.html',
    controller: 'signUpCtrl'
  })
  .when('/popout', {
    templateUrl: 'src/adf/templates/pop-out-widget.html',
    controller: 'popOutCtrl'
  })
  .otherwise({
    redirectTo: '/login'
  });
```

Figura 5.10 Mapare URL la Controllere

Fiecare controller asignat unei pagini folosește un fisier de tip template HTML diferit care la randul lor conțin alte template-uri considerate sub-componente. În cadrul fiecarui controller răspunzător de comunicarea directă cu template-urile HTML este utilizat modulul de localStorage AngularJS. Astfel informațiile legate de structura portalului împreună cu datele despre fiecare portlet în parte vor fi salvate în local storage-ul browser-ului. sub format JSON. Un exemplu poate fi regasit în figura 5.11 de mai jos:

Key	Value
portal.test_19958494	{"type":"weather","uid":"19958494","config":{}}
portal.test_33135792	{"type":"task","uid":"33135792","config":{}}
portal.test_a81c5862	{"type":"news","uid":"a81c5862","config":{}}
portal.test_positions	{"rows":[{"columns":[{"widgets":["33135792","a81c5862","19958494"]}]}]}
portal.test_structure	{"name":"6-6","structure":{"rows":[{"columns":[{"styleClass":"col-md-6"},{"styleClass":"col-md-6"}]}]}}
cachedResource://listAllTasks	{"value":{"data":{"tasks":[{"id":"1","assignee":"me","creationDate":"2014-07-07T10:17:43.902+0000","state":"in progress","description":"Task descript...
portal.test_06232fae	{"type":"task","uid":"06232fae","config":{},"title":"Tasks"}
portal.test	{"type":"processModeler","uid":"b00964a5","config":{}}

Figura 5.11 Strucutra Local Storage

Fiecare key conține numele utilizatorului alături de id-ul generat al portletului. Am utilizat o funcție de generare a id-urilor pentru portleți fiindcă am considerat că aceasta ar fi soluția cea mai buna care ar permite utilizarea mai multor componente de tip portlet în cadrul aplicației. În cazul în care aplicația este utilizată de utilizatori multiplii pe același device, portalul are deja salvată structura pentru fiecare utilizator în localStorage, facilitand performante mai bune fiindcă se evită execuția unui request către server.

Aplicația pune la dispoziția utilizatorului o parte din funționalități chiar dacă conexiunea la Internet este întreruptă, iar acest lucru este posibil datorită modulului de cachedFactory pus la dispoziție de AngularJS alături de modulul de localStorage. În figura 5.11 se poate observa resursa cu cheia „cachedResource://listAllTasks” și valoarea aferentă acesteia. Numele cheii reprezintă apelul API care este cach-uit în localStorage împreună cu datele corespunzatoare. În cazul în care datele sunt modificate în momentul în care dorim să adaugăm un task nou și nu există o conexiune la Internet, informatia ce ține de acest task este salvată în localStorage sub numele „cachedResource://newTask”. Pentru detecția stării conexiunii la Internet este folosit API-ul browser-ului și mai precis a proprietății: „navigator.onLine” care are valori de tip boolean. Această proprietate este atașată la $rootScope fiindcă detecția trebuie să aiba loc la nivel global. Putem observa mai jos serviciul de AngularJS creat cu scopul de a urmării starea conexiunii la Internet:

```
7   .factory('onlineStatus', ["$window", "$rootScope", function ($window, $rootScope) {
8     var onlineStatus = {};
9
10    onlineStatus.onLine = $window.navigator.onLine;
11
12    onlineStatus.isOnline = function () {
13      return onlineStatus.onLine;
14    };
15
16    $window.addEventListener("online", function () {
17      onlineStatus.onLine = true;
18      $rootScope.$digest();
19    }, true);
20
21    $window.addEventListener("offline", function () {
22      onlineStatus.onLine = false;
23      $rootScope.$digest();
24    }, true);
25
26    return onlineStatus;
27  }]);
```

Figura 5.12 Detecția stării conexiunii

În momentul crearii unui nou cont, utilizatorul este nevoit să-și intrdoduca adresa de e-mail, un username și o parola. În cazul în care adresa de e-mail nu are un format valid sau parola nu are minim 5 caractere, utilizatorul aplicației va fi atenționat în legătura cu problemele aparute. Daca există deja un utilizator cu același nume sau e-mail în baza de date, va fi afișat un mesaj de atenționare, acest lucru demostrând faptul că a fost implementat un mecanism de validare la nivel de front-end care verifică în timp real veridicitatea datelor introduse.

Pentru a restricționa accesul utilizatorilor la alte rute, se crează o sesiune în momentul logării , iar în momentul în care aceștia încearcă să acceseze o rută din aplicație se verifica mereu dacă sunt logați, iar în caz contrar aceștia vor fi redirecționați către pagina de login.

Pentru a realiza header-ul și meniul aplicației am decis să folosesc template-uri separate HTML lucru care ne permite să avem o organizare mai buna a fișierelor și a codului.

Portalul reprezintă o componentă de tip AngularJS directive care este responsabilă de gestionarea unităților de tip portlet și este reprezentată în mark-up astfel : „`<adf-dashboard name="{{name}}" structure="4-8/4-4-4/12" adf-model="model" />`”. Atributele HTML „name” , „structure” și „adf-model” reprezintă configurările de baza ale portalului. Numele componentei portal se schimbă în funcție de utilizatorul autentificat, același lucru se întâmpla și în cazul structurii și a modelului de date. Reprezentarea structurii se face în funcție de numărul de rânduri și de coloane fiind utilizat un sistem de tip „grid” conținut în framework-ul de css Bootstrap.

Fiecare unitate de tip portlet are un template HTML diferit cât și un controller diferit. În cazul în care avem mai multi portleți de același tip vom avea instante diferite ale controllerelor. Există un template general pentru anumite funcții ale portlet-ului cum ar fi mesajul de „Remove” sau toate mesajele modale de confirmare. Pozitia fiecarui portlet este salvată în localStorage sub forma unui JSON. Din punct de vedere al implementarii al unui portlet am încercat să dezvolt o componenta care să fie personalizabilă și care să poată fi reutilizata în dezvoltarea specifică a unui tip de portlet. Portletul reprezintă la randul lui o componentă de tip directiva și are anumite atribute care sunt comune tuturor unităților de tip portlet:

- **templateUrl** reprezintă locatia din care portlet-ul isi va prelua template-ul HTML.
- **title** reprezintă titlul portlet-ului care va fi afișat în portal
- **controller** reprezintă controller-ul răspunzător de actiunile fiecarui portlet.
- **edit** reprezintă templateUrl-ul folosit în momentul în care se editează un portlet și acesta variaza de la un portlet la altul**.**

Starea și dimensiunea unui portlet sunt pastrate la nivelul componentei de portlet astfel avem de a face cu patru noi atribute: definition, col, collapsible și popped. Fiecarui portlet ii corespunde un serviciu AngularJS de tip config, un exemplu fiind prezentat în figura 5.13 de mai jos:

```
28  .value('weatherServiceUrl', 'http://api.openweathermap.org/data/2.5/weather?units=metric&q=')
29  .config(function(dashboardProvider){
30    dashboardProvider
31      .widget('weather', {
32        title: 'Weather',
33        description: 'Display the current temperature of a city',
34        templateUrl: 'src/scripts/widgets/weather/weather.html',
35        controller: 'weatherCtrl',
36        reload: true,
37        resolve: {
38          data: function(weatherService, config){
39            if (config.location && navigator.onLine){
40              return weatherService.get(config.location);
41            }
42          }
43        },
44        edit: {
45          templateUrl: 'src/scripts/widgets/weather/edit.html'
46        }
47      });
48  })
```

Figura 5.13 Serviciu de configurare al portletului

Pagina de autentificare poate fi accesată atât de utilizatorii logati cât și de cei nelogati, dar în momentul în care acestia vor dori să între în aplicație, vor fi rugati să se autentifice sau în cazul în care nu dețin un cont necesar utilizarii portalului vor fi nevoiți să-și creeze un cont fiind redirectionați către pagina de **Sign-Up.**

Pagina portalului va putea fi accesată doar de utilizatorii logați, pagina fiind generată dinamic în funcție de utilizatorul care s-a logat iar generarea se face pe baza numelui utilizatorului și a informațiilor prezente în localStorage. Informațiile afișate de aceasta pagina sunt generate de către controllere și serviciile aferente fiecarui controller. În header-ul portalului se găsește meniul de unde utilizatorul poate să adauge sau să șteargă portleți dar în același timp poate să-și configureze portalul.

5.6. Interacțiunea dintre componente

În acest subcapitol voi descrie interacțiunile dintre componente ce au loc pentru a satisface două cazuri de utilizare: preluarea de taskuri și preluarea de informații de la un API extern .

Pentru a prelua lista de taskuri, utilizatorul trebuie să se logheze. În cazul în care logarea are log acesta va fi redirecționat spre pagina portalului. Urmeaza ca utilizatorul să-și adauge un portlet de tip tasks. În momentul în care portletul este adăugat se apelează funcția **getAllTasks()** din serviciul de Angular **taskService**. Acest serviciu apeleaza un enpoint RESTful în cazul de fața „/api/task/getAllTasks”. La urmatorul pas intervine controller-ul din NodeJS și se apelează funcția **allTasks()** care face request-ul la MongoDB prin modelul de date Taks definit în Mongoose. După ce datele au fost găsite se returnează datele la controller-ul din NodeJS care la randul lui returnează datele împreuna cu status code-ul sub forma unui JSON serviciului din Angular **taskService**. Acesta va apela au loc urmatoarele interacțiuni între componente:

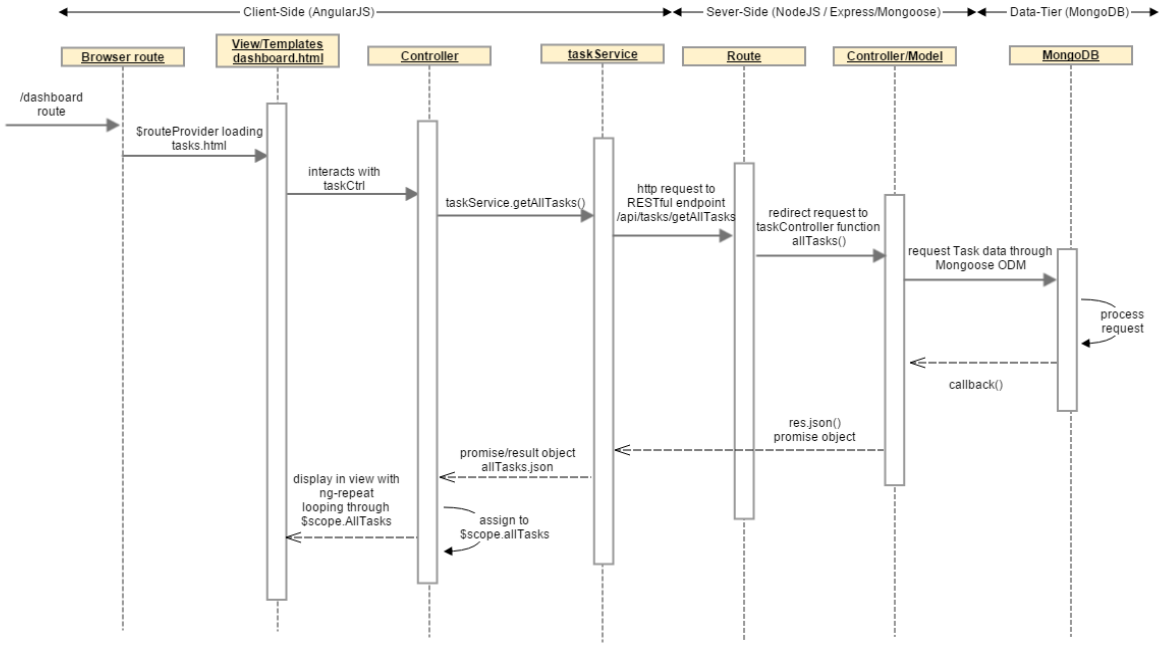

Figura 5.14 Diagrama de secvențe pentru preluarea tuturor task-urilor

Cel de-al doilea caz constă în apelarea unui API third-party și prezintă interacțiunea componentelor pe partea de client în figura 5.15.

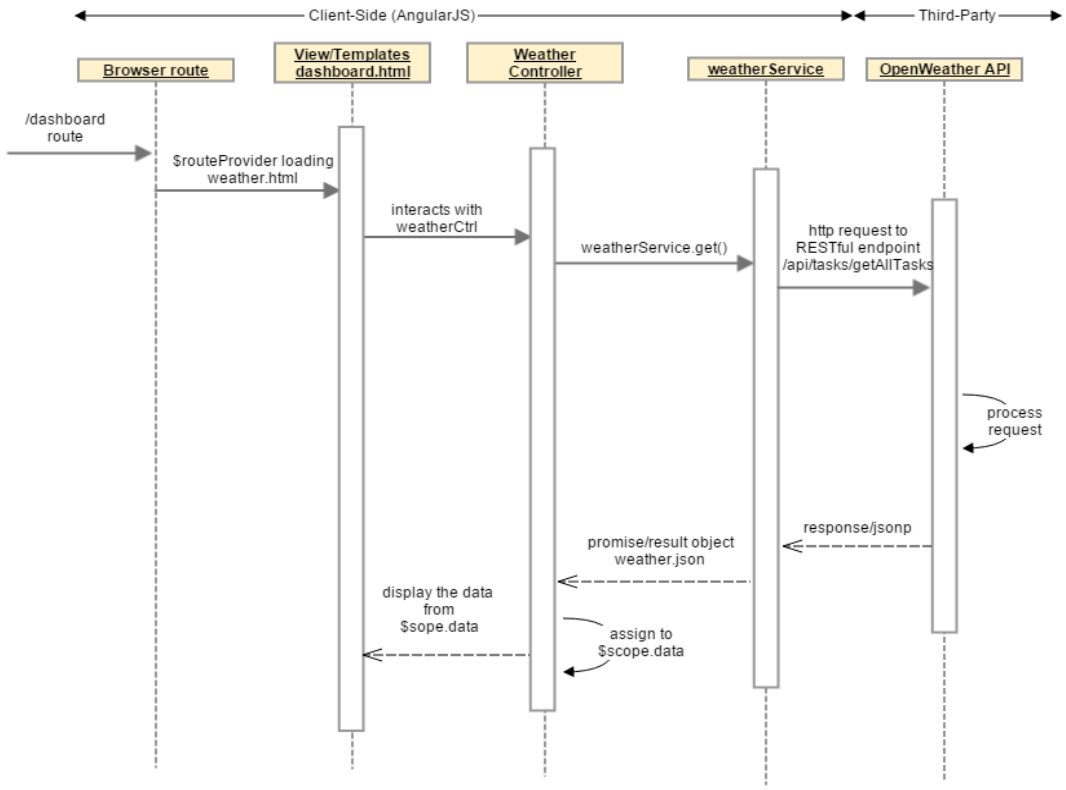

Figura 5.15 Diagrama de secvențe pentru preluarea de date de la un API third-party

Capitolul 6. Testare și Validare

Testarea software reprezintă procesul folosit pentru identificarea corectitudinii și a calitatii produsului software dezvoltat. Testarea este un proces care are scopul de a descoperi informații despre produsul realizat în cadrul contextului în care se dorește a fi utilizat. Acest proces presupune executarea unor pași cu ajutorul unor unelte software pentru depistarea erorilor existente. Prin intermediul procesului de testare se realizează o verificare a comportamentului și a starii aplicației în funcție de specificatiile acestuia.

Indiferent de metodele de testare folosite, rezultatele testarii reprezintă un nivel de incredere în produsul dezvoltat, astfel încat dezvoltatorul are un nivel de siguranta și incredere cu privire la produsul dezvoltat dar în același timp rezultatele ii oferă o idee privind rata de defectare a produsului.

Una din problemele majore ale testarii produselor software este dată de faptul că defectele care apar pot să fie foarte mari din în același timp numărul configurarilor pe care le suporta sistemul pot să fie și mai mari. Există anumite probleme care apar foarte rar iar acestea sunt greu de descoperit prin intermediul testarii, de aceea este necesar ca efortul depus în testarea unui produs să fie cel puțin egal cu jumatate din timpul petrecut pentru dezvoltarea produsului.

Testarea sistemului s-a realizat la finalul fiecărei etape de proiectare. Respectându-se maniera iterativă de dezvoltare software, fiecare componentă nou introdusă a fost testată înainte de a fi aprobată și adăugată în etapa iterativă curentă a proiectului. Acest lucru îmbunătățește calitatea produsului software și reduce viitoarele posibile erori apărute în cadrul unui modul.

6.1. Testare manuală

Aplicația a fost testată în mod manual, urmărind scenariile de test intocmite pe baza funcționalităților sistemului. În cadrul aplicației au fost integrate și două utilitare responsabile de testarea automata a aplicației denumite KarmaJS și Protractor.

În cele ce urmează vor fi descrise sapte scenarii de test relevante din punct de vedere al complexității elementelor testate.

Scenariul 1 (Scenariul de creare cont)

Pasul 1: Utilizatorul accesează aplicația și este întâmpinat de ecranul introductiv de logare

Pasul 2: Utilizatorul apasa butonul „Sign up”

Pasul 3: Utilizatorul apasa butonul „Sign up with email” din cadrul ecranului introductiv de inregistrare

Pasul 4: Utilizatorul completează campurile pentru nume, prenume, adresa de email, parola, selecteaza sexul și data nasterii. Utilizatorul introduce o parola care are lungimea de minim 5 caractere și o adresa de email valida, adică corespunde formatului xyz@domain.com sau alte top level domain

Pasul 5: Utilizatorul apasa butonul „Sign up” și un nou cont a fost creat.

Pasul 6: Utilizatorul verifica corectitudinea emailului pentru care a fost generat contul, și confirma „username”-ul generat de către sistem.

Rezultatul asteptat: Utilizatorul apasa butonul „Register” și contul este salvat.

Scenariul 2 (Scenariu de eșec în cazul autentificarii)

Pasul 1: Utilizatorul acceseaza aplicația și este întâmpinat de ecranul introductiv de logare

Pasul 2: Utilizatorul apasa butonul „Sign În”

Pasul 3: Utilizatorul apasa butonul „Sign În” din cadrul ecranului introductiv de inregistrare

Pasul 4: Utilizatorul selecteaza un camp, insa nu îl completează și trece la completarea unui altul camp.

Rezultatul asteptat: Afișarea unui mesaj de tipul „Value required” în dreptul campului care a fost selectat și nu a fost completat.

Scenariul 3 (Scenariu de eșec în cazul creării unui cont)

Pasul 1: Utilizatorul acceseaza aplicația și este întâmpinat de ecranul introductiv de logare

Pasul 2: Utilizatorul apasa butonul „Sign up”

Pasul 3: Utilizatorul introduce o parola care nu are lungimea de minim 5 caractere

Rezultatul asteptat: Afișarea unui mesaj de tipul „Password must be at least 5 characters” în dreptul campului destinat introducerii parolei.

Scenariul 4 (Scenariu eșec creare cont din cauza lipsei de informații)

Pasul 1: Utilizatorul acceseaza aplicația și este întâmpinat de ecranul introductiv de logare

Pasul 2: Utilizatorul apasa butonul „Sign up”

Pasul 3: Utilizatorul nu introduce textul în nici unul dintre campurile destinate introducerii de informații necesare crearii unui cont

Pasul 5: Utilizatorul apasa butonul „Sign up”

Rezultatul asteptat: Afișarea unui mesaj de tipul „Required information” în dreptul tuturor campurilor.

Titlu: Ștergerea unui portlet

Descriere: Utilizatorul trebuie să își poată crea un cont cu ajutorul unei adrese de email valide însoțită de o parolă sigură

Scenariul 5 (Scenariu de succes ștergere portlet)

Conditie de test: Utilizatorul care realizează scenariul de test este utilizatorul autentificat și iar acesta are posibilitatea de a șterge unul sau mai multe portlet-uri.

Pasul 1: Utilizatorul acceseaza aplicația

Pasul 2: Utilizatorul acceseaza meniul “Active portlets”

Pasul 3: Utilizatorul face hover peste unul din portlet-urile active.

Pasul 4: Utilizatorul da click pe butonul de close din partea dreaptă a portletului activ

Pasul 5: Utilizatorul apasa butonul „OK”

Rezultatul asteptat: Afișarea unui mesaj de tipul „Do you really want to remove this portlet?” și ștergerea portletului din portal.

Scenariul 6 (Scenariu de eșec în cazul editării unui portlet)

Conditie de test: Utilizatorul care realizează scenariul de test este utilizatorul autentificat iar dispozitivul nu mai este conectat la Internet în momentul în care utilizatorul încearcă modifice setarile portletului de stiri.

Pasul 1: Utilizatorul acceseaza aplicația

Pasul 2: Utilizatorul adaugă un portlet de tip „News”

Pasul 3: Utilizatorul apasa butonul „Configure” al portlet-ului din coltul dreapta sus

Pasul 4: Utlizatorul apasa butonul „Edit”

Pasul 5: Utilizatorul modifica setarile din campul „Feed URL”

Pasul 5: Utilizatorul apasa butonul „Close”

Rezultatul asteptat: Afișarea unui mesaj de tipul „Could not resolve all promises”.

Scenariul 7 (Scenariu de succes în cazul modificării unui task)

Conditie de test: Utilizatorul care realizează scenariul de test este utilizatorul autentificat iar dispozitivul este conectat la Internet. Utilizatorul are deja adaugat un portlet de tip tasks și dorește să schimbe statusul unui task din „în progress” în „completed”.

Pasul 1: Utilizatorul acceseaza aplicația

Pasul 2: Utilizatorul adaugă un portlet de tip „Tasks”

Pasul 3: Utilizatorul da click în checkbox-ul unuia din task-urile care nu sunt marcate ca fiind „completed”.

Rezultatul asteptat: Schimbarea statusului din „în progress” în „completed”.

6.2. Testarea interfeței grafice

Pentru testarea interfeței grafice am folosit utilitarele KarmaJS[5] și Protractor [7] pentru a testa dacă elementele interfeței functioneaza corect. KarmaJS este un utilitar JavaScript care este utilizt din linia de comandă și care este folosit pentru a crea un server web care încarcă codul sursa al aplicației și care execută testele. KarmaJS poate fi configurat să ruleze în mai multe browsere, ceea ce ajuta foarte mult în cazul în care produsul trebuie utilizat în mai multe browsere. Karma este executat în linia de comandă și va afișa rezultatele testelor în linia de comandă după ce acestea au fost rulate în browser. Protractor este un utilitar dezvoltat în Node.Js care folosește frameworkul de teste Jasmine pentru interfața de testare.

Am realizat un test pentru a verifica pașii necesari autentificarii în cadrul portalului. Acest scenariu de test acoperă și componenta de login a aplicației, iar în momentul în care utilizatorul nelogat încearcă să se autentifice cu date incorecte vom verifica dacă apar mesajele corespunzatoare în cadrul paginii.

```
describe('LoginController', function () {

        it('initially has login inputs', function () {

            browser.get('http://localhost:9000/#/login);
```

```
            expect(element(by.id('username')).getText()).toEqual('usern
      ame');
            expect(element(by.id('username')).getText()).toEqual('passw
      ord');
        });
       it('clicking the sign în button changes the route of the app',
function () {
            browser.get('http://localhost:9000/#/login);
            element(by.css('[ng-model="username"]')).sendKeys('test');
            element(by.css('[ng-
            model="password"]')).sendKeys('testpassword');
            element(by.css('.btn-default')).click();
            expect(element(by.id('menu')).getText()).toEqual('Active
            widgets');
        });
       it('clicking the button does not login into app', function () {
            browser.get('http://localhost:9000/#/login);
            element(by.css('.btn-default')).click();
            expect(element(by.id('error')).getText()).toEqual('Wrong
credentials!');
       });
});
```

6.3. Testarea performantei

Pentru a determina performanta aplicației am utilizat componenta de Profiling care face face parte din utilitarul de dezvoltare al aplicațiilor web, pus la dispoziție de browser-ul Google Chrome. O parte din problemele greu de determinat în urma unei isntalari a aplicației web pe server sunt cele legate de performanta. Acestea pot aparea datorită unor probleme a produsului software, sistemului de back-end care nu functioneaza cum ar trebui, o problema de retea sau alti factori interni sau externi.

Folosind profiler-ul oferit de browser-ul Google Chrome am analizat resursele folosite de aplicația web. Prima analiza a fost facuta cu scopul de a determina scurgerile de memorie din aplicație:

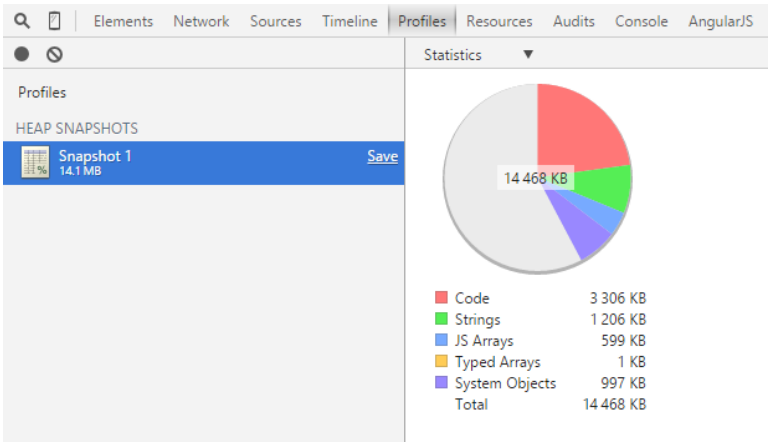

Figura 6.1 Heap profiling

Cea de a doua analiza a fost realizată cu scopul de a determina timpul de executie a funcțiilor de JavaScript în cadrul aplicației web. Pentru a realiza acest lucru am folosit unealta de CPU Profiling iar rezultatele pot fi observate în figura de mai jos:

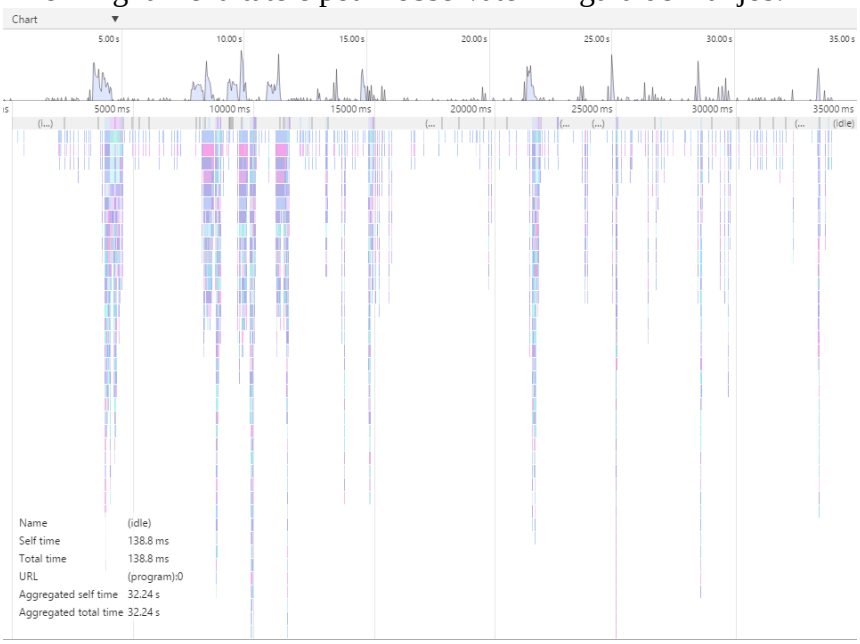

Figura 6.2 CPU profiling

Capitolul 7. Manual de Instalare și Utilizare

În cele ce urmează voi prezenta resursele necesare pentru functionarea aplicației, precum și pașii necesari pentru a instala aplicațiile și dependințele necesare utilizarii sistemului de tip portal.

7.1. Manual de instalare

Cerinte Sistem

Petru ca aplicația web să funcționeze în conditii optime avem nevoie de o masina cu urmatoarele cerinte minime:

- 512MB de RAM
- Spatiu pe disc de 300 MB

Aplicatiile care trebuie instalate sunt:

NodeJS

Instalati NodeJs, urmând pașii de mai jos:

1. Mergeti la CD-ul furnizat o dată cu aplicația și localizati fișierul "node-v0.12.7-x64.msi" sau la adresa: https://nodejs.org/en/download/
2. Urmati instructiunile de pe ecran

Visual Code

Instalati VisualStudioCode, urmând pașii de mai jos:

1. Mergeti la CD-ul furnizat o dată cu aplicația și localizati fișierul "VSCodeSetup.exe" sau la adresa: https://code.visualstudio.com/download
2. Urmati instructiunile de pe ecran

Ruby v2.2.3

Instalati Ruby, urmând pașii de mai jos:

1. Mergeti la CD-ul furnizat o dată cu aplicația și localizati fișierul "rubyinstaller-2.2.3-x64.exe" sau la adresa https://www.ruby-lang.org/en/downloads/.
2. Urmati instructiunile de pe ecran
3. Deschideti un command line și executați urmatoarele comenzi : "gem update –system" și "gem install compass".

MongoDB v3.0.6

Instalati MongoDB, urmând pașii de mai jos:

1. Mergeti la CD-ul furnizat o dată cu aplicația și localizati fișierul „mongodb-win32-x86_64-3.0.6-signed.msi" sau la adresa https://www.mongodb.org/downloads.
2. Urmati instructiunile de pe ecran

Instalare dependințe proiect

Instalarea dependintelor ce tin de proiect se face urmând pașii de mai jos:

1. Mergeti la CD-ul furnizat o dată cu aplicația și localizati folder-ul „dash-portal".

2. Accesati folder-ul mentionat mai sus și deschideti un command line
3. Executati urmatoarele comenzi : „npm install” urmat de „bower install”

După instalarea tuturor aplicațiilor și a dependintelor, deschideti un command line în folder-ul aplicației și executați comanda „grunt server”.

7.2. Manual de utilizare

Pentru utilizarea aplicației, voi relua unele scenarii ale cazurilor de utilizare descrise în capitolele precedente.

Pagina de „Home” a site-ului permite utilizatorilor să vizualizeze portalul în sine și să acceseze mai multe tipuri de portleți: chat, vreme , task-uri , stiri.

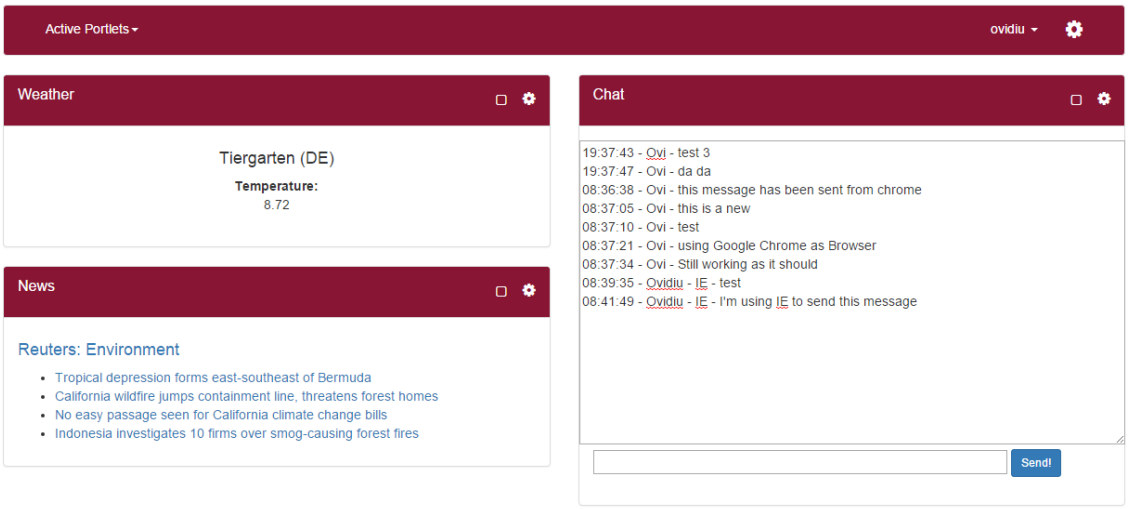

Figura 7.1 Pagina portalului

Pentru a se loga, utilizatorul trebuie să acceseze pagina de „login” unde va trebui să introducă adresa de email și parola. În cazul în care datele vor fi corecte, utilizatorul va fi redirectionat spre pagina de portal iar în caz contrar va fi afișat un mesaj de eroare.

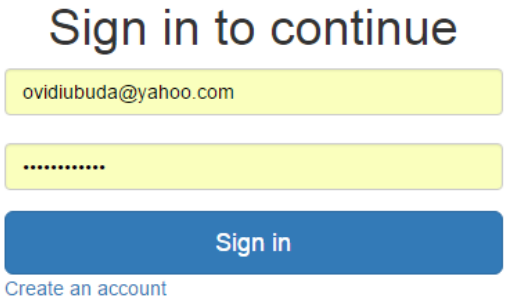

Figura 7.2 Pagina de login a aplicației

Daca utilizatorul nu are un cont pentru accesarea aplicației, acesta poate accesa meniul Create an account din pagina de login urmând să fie redirectionat către pagina de Sign Up a aplicației.

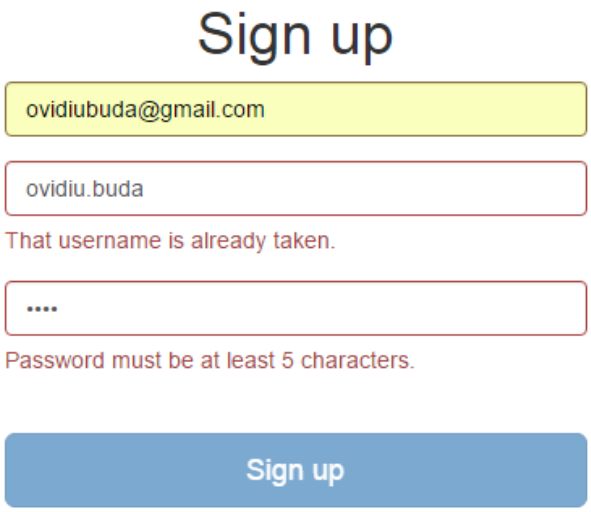

Figura 7.3 Pagina de Sign Up a aplicației

Pentru a vizualiza lista de portleți activa, utilizatorii vor accesa meniul „Active Portlets" care se gaseste în header-ul portalului. Tot aici au posibilitatea de a șterge un portlet în funcție de dorintele sale.

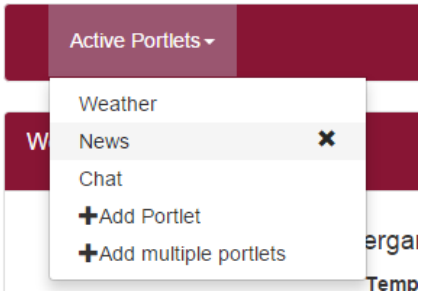

Figura 7.4 Meniul Active Portlets

Pentru a adaugă un portlet utilizatorul are la dispoziție două variante, cea de a adaugă un singur portlet („Add portlet" sau de a adaugă mai multi portleți simultan „Add multiple portlets"). În momentul în care va da click pe Add portlet se va deschide un modal în care poate să selecteze portletul dorit printr-un click.

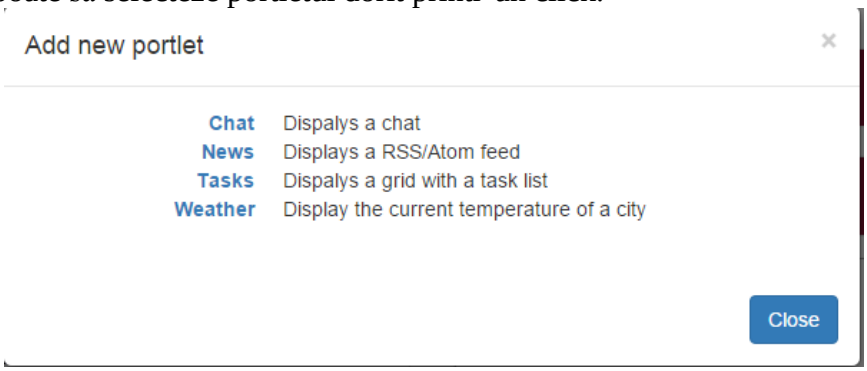

Figura 7.5 Adaugarea unui singur portlet

După ce a adaugat un portlet, utilizatorii pot să configureze fiecare portlet în parte accesand meniul de „Configure” al portletului respectiv.Tot aici au posibilitatea de a maximiza portletul astfel încat să ocupe spatiul pus la dispoziție de portal, de a șterge portletul sau de a îl minimiza.

Figura 7.6 Meniu Portlet

Pentru a schimba setarile portalului se va acesa meniul de configurari generale al portalului din partea dreapta a header-ului , iar utilizatorului ii va fi afișat un modal în care poate să selecteze structura portalului.

Figura 7.7 Setari Portal

Utilizatorul are la dispoziție un meniu prin care poate să se delogheze din aplicație sau să-și modifice anumite setari legate de cont.

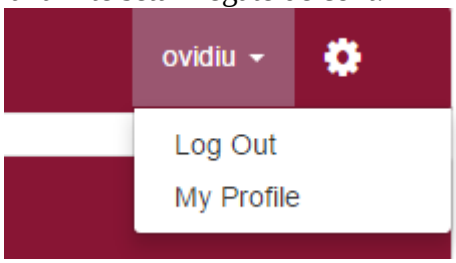

Figura 7.8 Meniu Utilizator.

Capitolul 8. Concluzii

8.1. Realizări

Proiectul a constituit o oportunitate de dezvoltare tehnica pentru dezvoltatorul software. Astfel, cele mai improtante realizări au fost:

- Dobândirea de cunoștinte solide despre aplicațiile web single page, despre conceptele cu care se opereaza în cadrul acesteia, cât și despre „good practice”-urile aferente tehnologiei mobile
- Participarea activă în etapa de analiza și proiectare a proiectului, fapt care a format o viziune de ansamblu asupra proiectului, a conturat o perspectiva asupra posibilelor implementari
- Implementarea sistemul proiectat într-o maniera iterativa, care a permis parcurgerea tuturor etapelor dezvoltarii software pentru fiecare iteratie în parte
- Idenficarea solutiilor pentru diversele provocari tehnice
- Alegerea solutiilor optime pentru a rezolva „task”-urile fiecărei iterații.

Proiectul a dus la îndeplinirea tuturor obiectivelor impuse și enunțate în cadrul capitolului 2, astfel că proiectul final permite utilizatorului să:

- Își creeze un cont corelat cu o adresă validă de email
- Se logheze în cadrul aplicației
- Comunice cu alti utilziatori în timp real prin intermediul portletului de tip chat
- Utilizeze portletul de știri
- Distribuie task-uri altor utilizatori
- Configureze portalul după bunul plac
- Configureze fiecare portlet în parte
- Schimbe structura portalului
- Își modifice profilul.

Mai mult, proiectul a încurajat dobandirea de cunoștințe noi în ceea ce priveste diferitele „framework”-uri disponibile la momentul actual.

8.2. Dezvoltari ulterioare

Orice sistem poate fi imbunatatit și actualizat pentru a se mapa pe cerințele și necesitățile utilizatorilor țintă, pentru a fi pe masura așteptării acestora. Printre îmbunatatirile care ar putea fi aduse sistemului actual pentru o versiune viitoare, se enumeră:

- **Adaugarea mai multor tipuri de portleți**. Utilizatorul ar avea posiblitatea de a adaugă mai multe tipuri de portleți care deservesc difereite necesități : e-mail, harti, video streaming.
- **Integrarea Google OAuth**. Sistemul ar trebui să permita utilizatorului să își integreze propriul cont de Google în momentul crearii unui cont. În acest

mod, utilizatorul are posibilitatea de a folosi diferite aplicații oferite de Google(Gmail, Google Maps, Google+ sau chiar YouTube).

- **Integrare social media.** În momentul crearii unui cont, sistemul ar trebui să permita utilizatorului să-și introducă datele de autentificare de la retelele de utilizre Facbook sau Twitter astfel avand posibilitatea de a utiliza aplicații și informații legate de utilizatori. Din punct de vedere tehnic acest lucru presupune sa definim o nouă strategie de login în Passport.
- **Autentificare utilizând SMS sau Google Authenticator.** Sistemul ar trebui să permita trimiterea de SMS-uri în cazul în care același utilizator s-a logat în sistem folosind două IP-uri diferite într-un interval de timp scurt. Astfel procesul de login ar necesită o verificare suplimentara, care ar solicita utilizatorului să introducă codul primit prin SMS în momentul în care dorește să se autentifice. Totodată această opțiune de securitate ar putea fi pusa la dispoziție utilizatorului la fiecare autentificare. O altă abordare presupune dezvoltarea unui modul care sa utilizeze Google Authenticator, mecanism care presupune instalarea aplicației Google Authenticator pe mobil unde utilizatorul primește un cod format din 6 cifre care se schimbă la un interval de 30 de secunde.
- **Implementarea unui portlet de tip hashtag**. Implementarea unui portlet care să permita cautarea de informații prin hashtag în cadrul mai multor retele sociale utilizand API-ul pus la dispoziție de acestea. Acesta poate fi folosit cu mai multe scopuri printre care se numără crearea de conținut customizabil pentru reteaua respectiva de socializare, crearea de feed-uri live și de asemenea poate fi utilizat cu scopul de a dezvolta mai bine o afacere.
- **Internaționalizare.** Sistemul ar trebui să permita utilizatorului să schimbe limba în care este furnziat conținutul. Astfel fiecare portlet împreună cu conținutul acestuia să fie tradus în limba selectata de utilizator. Din punct de vedere tehnic acest lucru se poate realiza prin integrarea modului „angular-translate” și traducerea conținutului existent în mai multe limbi urmând ca utilizatorul să selecteze limba dorită folosind un dropdown.

Bibliografie

[1] AngularJS Documentation, [Online] Disponibil: https://docs.angularjs.org/guide

[2] G.P. Marquis. *Application of traditional system design techniques to web site design. Information and Software Technology*, Apress 2002

[3] E. Reshef. *Building Interactive Web Services with WSIA & WSRP*. Web Services Journal, pages 2–6, December 2002. [Online] Disponibil: http://webservices.sys-con.com/read/39627.htm Consultat la data de: 27.06.2015

[4] ExpressJS API, [Online] Disponibil: http://expressjs.com/4x/api.html Consultat la data de: 23.06.2015

[5] KarmaJS API, [Online] Disponibil: http://karma-runner.github.io/0.13/dev/public-api.html Consultat la data de: 23.06.2015

[6] J.Boye. *Portals: from idea to reality-the dangers of the current state of portals în the marketplace*. Apress 2005

[7] Protractor API [Online] Disponibil: http://www.protractortest.org/#/api Consultat la data de: 24.06.2015

[8] Mikito Takada, *Single page apps în depth [Online].Disponibil:* http://singlepageappbook.com/ Consultat la data de: 27.06.2015

[9] Michael S. Mikowski and Josh C. Powell Foreword by Gregory D. Benson, *Single Page Web Applications Javascript end-to-end*, Manning, September 2013

[10] Murray ,G. *The Portal is the Desktop*. California: Intraspect, Inc , 1999 Consultat la data: 27.06.2015

[11] Jeff Dick, *Write Modern Web Aps with the MEAN Stack: Mongo, Express, AngularJS, and Node.js*, Peachpit Press, 2014

[12] S.Wong, *WebServices:The Next Evolution of Application Integration*, 2001. [Online] Disponibil: http://www.ebizq.net/topics/web_services/features/1526.html Consultat la data de: 28.06.2015

[13] Ajan Masourvar, Norizan Mohd Yasin, „*World Academy of Science,Engineering and Technology*", vol:4, 2010 [Online]. Disponibil: http://waset.org/publications/5677/web-portal-as-a-knowledge-management-system-în-the-universities Consultat la data de: 23.06.2015