

Universitatea „Politehnica” din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Aplicație web pentru studenți în vederea încărcării notițelor, salvarea în calendar a datelor pentru examene și organizarea în ansamblu a activității de student.

Proiect de diplomă

prezentat ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Electronică și Telecomunicații* programul de studii de licență *Rețele și Software de Telecomunicații*

Conducători științifici
Conf. dr. ing. Galațchi Dan

Absolvent
Bucnaru Cristian

2020

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul „Aplicație web pentru studenți în vederea încărcării notițelor, salvarea în calendar a datelor pentru examene și organizarea în ansamblu a activității de student”, prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității “Politehnica” din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *domeniul Telecomunicații*, programul de studii *Rețele și software de telecomunicații* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurărilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, 30.06.2020

Absolvent *Cristian BUCNARU*



Cuprins

Lista figurilor.....	iv
Lista tabelelor.....	v
Lista acronimelor.....	vi
Capitolul I. Introducere	1
Capitolul II. Introducere în Full Stack Development	2
1. Noțiuni generale	2
2. Mediu de dezvoltare	3
3. Back End Development	4
3. 1. Noțiuni generale	4
3. 2. Node.js. Express.js.....	5
3. 3. npm	6
3. 4. JSON Web Token.....	6
4. Baze de date – MongoDB.....	7
4.1. Noțiuni generale	7
4.2. Comparatie MongoDB – MySQL	8
4.3. MongoDB Atlas.....	9
5. Front End Development	10
5.1. Noțiuni generale	10
5.2. HyperText Markup Language – HTML	12
5.3. Cascading Style Sheets – CSS.....	13
5.4. JavaScript – JS.....	14
5.5. React.js	14
5.6. Asynchronous JavaScript and XML – AJAX.....	16
Capitolul III. Implementare	17
1. Implementare server	17
2. Implementare bază de date	19
3. Implementare login/register.....	21

Lista figurilor

Figură II. 1. Full Stack Development.	2
Figură II. 2. Exemplu de cerere GET folosind Postman.....	3
Figură II. 3. Diagramă Back End Development.	4
Figură II. 4. REST API.....	5
Figură II. 5. Exemplu comandă npm	6
Figură II. 6. Structură JWT.....	7
Figură II. 7. Exemplu de colecție MongoDB	8
Figură II. 8. Comparatie performanță MongoDB – MySQL.....	9
Figură II. 9. Date statistice MongoDB Atlas - trafic pe cluster	10
Figură II. 10. Dezvoltarea Front End.....	11
Figură II. 11. Diagramă MVC.	12
Figură II. 12. Arborele DOM	13
Figură II. 13. Comunicația părinte - copil.	15
Figură II. 14. Principiul AJAX.	16
Figură III. 1. Informațiile conținute de o cerere POST pentru înregistrare	23
Figură III. 2. Răspunsul serverului pentru o autentificare validată	23
Figură III. 3. Mesaj de eroare întors de server.....	23

Lista tabelelor

Tabel II. 1. Comparație termeni MongoDB - MySQL	8
Tabel II. 2. Comparație comenzi MongoDB - MySQL.....	9
Tabel II. 3. Stilizare CSS	13

SA NU COPIEZI BA PULA

Lista acronimelor

HTML = HyperText Markup Language

CSS = Cascading Style Sheets

JS = JavaScript

PHP = Hypertext Preprocessor

Npm = Node Packet Manager (manager de pachete Node)

MongoDB = Mongo Database

IDE = Integrated Development Environment (Mediu de dezvoltare)

VSCode = Visual Studio Code

REST = Representational State Transfer

API = Application Programming Interface

HMAC = cod de autentificare a mesajelor cu cheie

RSA = Rivest, Shamir, Adleman

ECDSA = semnătură digitală curbă eliptică

JSON = JavaScript Object Notation

Capitolul I. Introducere

Trăim într-o perioadă în care regăsim tehnologie în orice aspect al vieții cotidiene. Până și unele activități de bază, cum ar fi așteptatul la coadă sau poate chiar redactarea unui document implică folosirea tehnologiei (emiterea bonurilor de ordine folosind roboți, folosind suita Microsoft Office). Automatizarea activităților sau stocarea informațiilor online sunt două avantaje puternice ale acestei perioade. Așadar, este normal ca dezvoltatorii și companiile, prin aceștia, să găsească soluții sau aplicații pentru a putea ușura activitățile comune ale unui angajat, crescând productivitatea și, automat, profitul, atât material, cât și mental.

Tehnologia evoluează datorită scopurilor pe care oamenii îl au atunci când decid să investească timp și resurse financiare în cercetare/dezvoltare. Bazându-ne pe creșterea exponențială de care tehnologia dă dovadă, putem spune că există o infinitate de activități ce se vor a fi automatizate sau îmbunătățite.

Cu toate că instituțiile de învățământ au migrat și ele către tehnologizare și cercetare în acest domeniu, un obicei a rămas aproape neschimbat din această privință (dacă excludem, bineînțeles, căutarea informațiilor pe internet): activitatea individuală a studentului. Informația este stocată tot în caiete și cărți în format fizic, iar, cu toate că un cadru didactic se oferă să răspundă întrebărilor oricărui student, procesul de învățare rămâne, totuși, foarte puțin interactiv și aproape deloc provocator pentru un student.

Scopul lucrării de față este de a prezenta o posibilă soluție prin care studiul individual este regândit, cu accent pe competitivitate, productivitate și interactivitate sau, cu alte cuvinte, automatizarea procesului de învățare.

În acest proiect s-au folosit cunoștințe dobândite în timpul cursurilor de Tehnologii de Programare în Internet, Programare Obiect-Orientată, Baze de Date și Securitatea Rețelelor și Serviciilor. Tehnicile și informațiile adunate în urma promovării acestor subiecte au fost extrapolate și folosite pentru înțelegerea tehnologiilor cu aplicabilitate directă în câmpul muncii. Aplicația este realizată folosind aceste tehnologii cu motivul de a reprezenta o soluție reală și de actualitate pentru scopul în care a fost creată.

Capitolul următor, „Introducere în Full Stack Development”, oferă informații despre bazele acestui subiect pentru a facilita înțelegerea mediului în care a fost realizată aplicația.

Capitolul „Implementare” prezintă în amănunt modul în care a fost construit acest proiect, explicând cum și de ce s-a folosit fiecare tehnologie prezentată în capitolul anterior.

Partea de utilizare a aplicației este, practic, un tutorial, prin care se încearcă familiarizarea cu interfața grafică a proiectului și cu anumite funcții ce rulează în background.

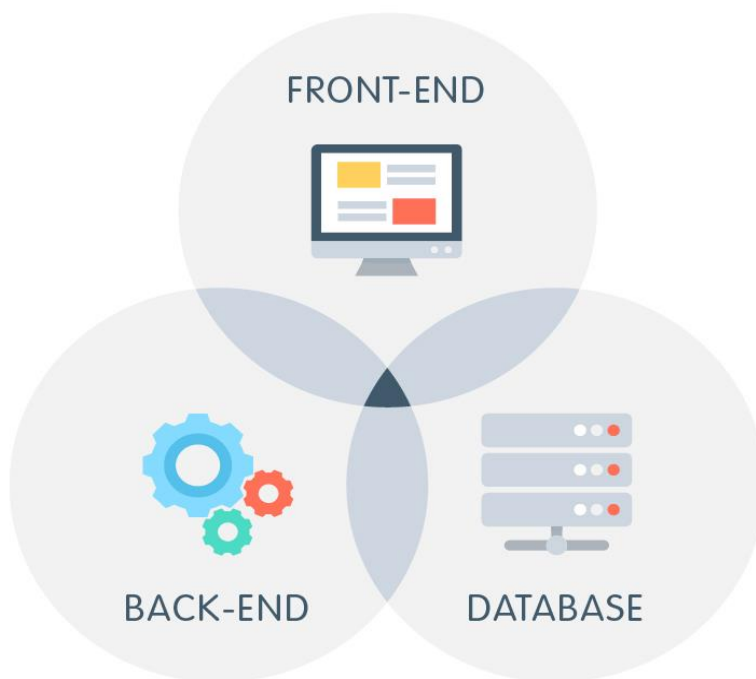
Tehnologiile utilizate în acest proiect sunt comune și aplicabile în viața reală, dar funcționalitatea proiectului și îmbinarea acestora pentru a obține produsul finit reprezintă contribuțiile practice personale ale autorului.

Capitolul II. Introducere în Full Stack Development

1. Noțiuni generale

În dezvoltarea unei aplicații web regăsim patru ramuri principale: design-ul și implementarea interfeței grafice, programarea unui server, programarea bazei de date și utilizarea unui serviciu prin care ne putem conecta la o bază de date din interiorul aplicației. Putem, așadar, considera că toate aceste ramuri constituie o stivă – de aici Full Stack (stivă întreagă). Dezvoltarea Full Stack se referă la programarea simultană a acestor tehnologii pentru a obține o aplicație.

FULL-STACK DEVELOPMENT



Figură II. 1. Full Stack Development.

Sursă: <https://skillvalue.com/jobs/wp-content/uploads/sites/7/2018/08/full-stack-development.jpg>

Dezvoltarea Full Stack este alcătuită din două părți generale: Front End (software-ul pentru utilizator, cu care acesta interacționează direct) și Back End (software-ul pentru server). Câteva exemple de tehnologii necesare pentru fiecare dintre aceste părți sunt: HyperText Markup Language (HTML), Cascading Style Sheets (CSS),

DB – Express – AngularJS – Node.js.

Pentru realizarea proiectului de față s-a folosit stiva MERN: MongoDB – Express.js – React.js – Node.js.

- Pentru realizarea proiectului de față s-a folosit stiva MERN: MongoDB – Express.js – React.js – Node.js.

Pentru realizarea proiectului de față s-a folosit stiva MERN: MongoDB – Express.js – React.js – Node.js.

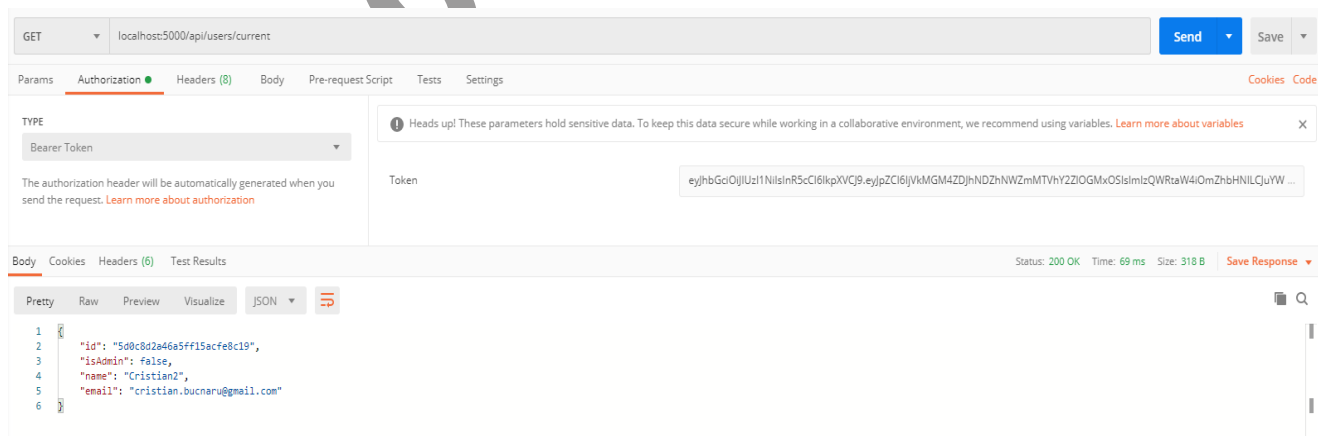
2. Mediu de dezvoltare

Orice aplicație, fie aceasta de tip web sau executabilă, este creată folosind un mediu de dezvoltare (IDE – *engl., integrated development environment*). Acesta este un instrument de editare de cod, compilare, depanare și testare.

Scopul unui mediu de dezvoltare este de a îmbina cât mai multe extensii necesare în facilitarea scrierii codului unui program. În trecut, mediile de dezvoltare erau specifice unui singur limbaj de programare și ofereau sugestii pentru completarea codului scris în limbajul respectiv, însă, odată cu evoluția tehnicilor de programare, s-au dezvoltat IDE-uri care oferă suport pentru o multitudine de limbaje de programare simultan.

Pentru aplicația prezentată s-a folosit Visual Studio Code. Acesta este un editor de cod dezvoltat de Microsoft, oferit gratuit, disponibil pe platformele Windows, MacOS sau Linux.

Desigur, pe lângă IDE, în dezvoltarea de aplicații intră, obligatoriu, managerul de pachete Node (npm), prin care se pot accesa extensii partajate de alți programatori, necesare pentru rezolvarea problemelor comune apărute în dezvoltare. Comenzile npm se execută într-un terminal integrat în IDE-ul VSCode. Comanda executată în Figura 2 arată versiunea de npm instalată pe mașina curentă.



Figură II. 2. Exemplu de cerere GET folosind Postman

A fost necesară utilizarea unui serviciu cloud pentru a putea stoca versiunile proiectului în timpul fazei de dezvoltare. A fost ales GitHub, pentru ușurința în utilizare și pentru suportul extins oferit în comunitatea de dezvoltatori. Cloud-ul este accesat prin comenzi de tipul „git” rulate în terminal.

O altă aplicație care a facilitat dezvoltarea proiectului este Postman, folosită pentru a manipula baza de date și pentru a trimite cereri de tip GET/POST/UPDATE/DELETE către aceasta.

După ce aplicația a ieșit din stadiul de dezvoltare, a fost necesar un serviciu de cloud hosting pentru a putea migra în producție. Alegerea a fost Heroku – o companie de cloud hosting ce oferă servicii gratuite pentru programatori și, totodată, suport pentru numeroase aplicații dezvoltate în diverse limbaje.

3. Back End Development

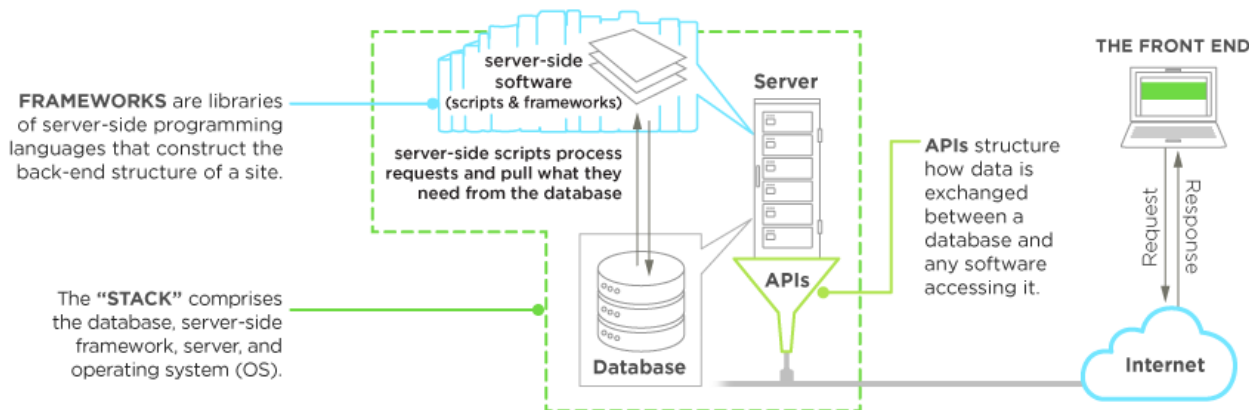
3.1. Noțiuni generale

Dezvoltarea Back End constă în implementarea unui server, o bază de date și o aplicație ce leagă serverul de baza de date. Practic, acest tip de dezvoltare este axat pe modul de funcționare al unei aplicații, nu pe interfața grafică sau interacțiunea dintre diferite elemente. Codul elaborat de un programator Back End este cel care comunică informații despre baza de date către browser. Limbajele de programare folosite sunt: Java, PHP, Python, .NET sau Node.js.

Dezvoltatorii sunt axați și pe viteza de răspuns a cererilor de tip GET/POST într-o bază de date. O aplicație care este dezvoltată fără Back End, adică fără server și fără bază de date, este doar o aplicație statică, în care nu putem interacționa cu o colecție.

BACK-END DEVELOPMENT & FRAMEWORKS IN SERVER SIDE SOFTWARE

upwork™



Figură II. 3. Diagramă Back End Development.

Sursă: <https://content-static.upwork.com/blog/uploads/sites/3/2015/05/05110024/Back-end-dev-logo.png>

În dezvoltarea acestui proiect s-au folosit: Node.js pentru crearea serverului, Express.js pentru a lega serverul de baza de date și MongoDB pentru baza de date.

3. 2. Node.js. Express.js

Node.js a apărut în 2009, fiind dezvoltat de Ryan Dahl. Inițial, acesta era disponibil doar pentru Linux și Mac OS X. În 2011, Microsoft s-a alăturat fondatorului Node.js pentru a dezvolta o versiune disponibilă și pentru Windows. Managerul de pachete Node (npm) a fost lansat în 2011 pentru a face posibilă o folosire la scară cât mai largă a lui Node.js.

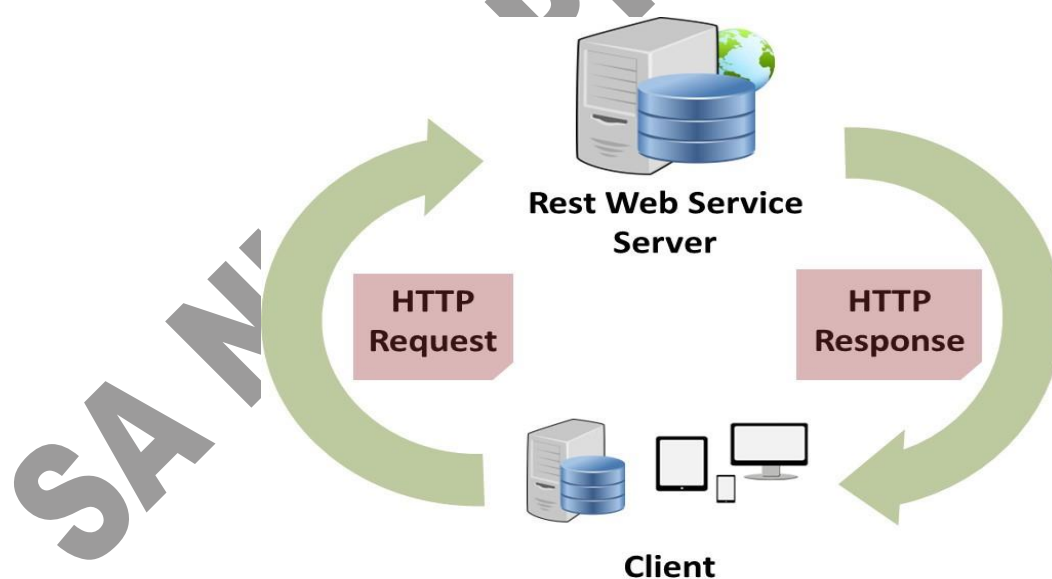
Prin Node.js se pot crea servere web folosind JavaScript¹ în tandem cu alte module ce asigură diferite funcționalități (de exemplu, Express.js pentru a conecta serverul la baza de date). Un avantaj pe care Node.js îl oferă față de alte module similare este faptul că funcțiile se execută fără a avea acțiuni I/O – procesorul pe care rulează nu va fi aproape niciodată blocat – și nu folosește fire de execuție.

Funcțiile se execută în mod asincron, nefiind necesară finalizarea execuției unui cod rulat anterior pentru a putea trece la interpretarea altei linii. Pentru transmisia datelor, în dezvoltare, se folosesc funcții de tip „call-back” care semnalează terminarea execuției unei linii de cod și care trimit informația dintr-o parte în alta a aplicației.

Acest modul folosește funcții de tip RESTful API (*Representational State Transfer Application Programming Interface*) pentru a servi diferitele cerințe ale dezvoltatorului sau utilizatorului.

REST este o arhitectură de sistem care definește un set de reguli utilizate în dezvoltarea Web. Acestea asigură interacțiunea dintre sistemele de calcul conectate la internet. Cele mai comune metode folosite sunt cele HTTP (GET, POST, PUT, DELETE)

API-urile de tip web sunt funcțiile prin care un utilizator interacționează cu serverul pe care sunt stocate datele unei aplicații și sunt definite ca un set de specificații (exemplu fiind cererile de tip HTTP).



Figură II. 4. REST API.

Sursă: https://miro.medium.com/max/782/1*EbBD6IXvf3o-YegUvRB_IA.jpeg

¹ <http://blog.training.com/2016/09/about-nodejs-and-why-you-should-add.html>, accesat la data de 27.05.2020

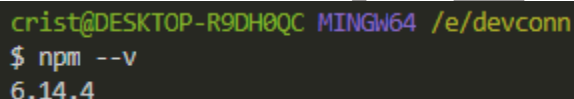
Express.js, principalul modul Node.js utilizat pentru a dezvolta aplicații web, oferă interfața minimală și conține unelte pentru dezvoltare. Popularitatea acestui modul provine din lipsa constrângerii programatorului la un set de reguli și posibilitatea oferită acestuia de a-și implementa ideile în diferite moduri printr-un suport dinamic și flexibil.

3.3. npm

Managerul de pachete Node (npm) este o resursă utilizată pentru mediul de lucru Node.js. A fost dezvoltat în limbajul JavaScript, în 2011, de către Isaac Schlueter.

npm este alcătuit dintr-o linie de comandă, numită tot npm, și o bază de date prin intermediul căreia pachetele sunt stocate. Managerul poate administra pachete locale care intră în alcătuirea unui proiect sau, în egală măsură, poate administra pachetele instalate global.

Principalul beneficiu adus de npm în alcătuirea unui proiect este acela că, printr-o singură comandă, se pot instala succesiv toate dependențele unui proiect pentru a putea începe dezvoltarea, fără a fi nevoie de rularea repetată a aceleiași comenzi pentru fiecare pachet.



```
crist@DESKTOP-R9DH0QC MINGW64 /e/devconn  
$ npm --v  
6.14.4
```

Figură II. 5. Exemplu comandă npm

3.4. JSON Web Token

JSON (*JavaScript Object Notation*) este un standard utilizat pentru a transmite date de la un browser către server, făcând totodată și conversia acestora în format de tip text – formatul obligatoriu pentru a putea stoca informația. JSON folosește sintaxă de tip JavaScript, făcând astfel manipularea obiectelor JSON facilă în dezvoltarea unei aplicații web.

Pentru a simplifica procesul de manipulare a datelor returnate de server, JSON are o funcție predefinită care realizează conversia din text în obiect JavaScript.

JSON Web Token (JWT) este un standard care definește un mod de a transmite în siguranță informații între sisteme ca obiect JSON. Aceste informații pot fi verificate și de încredere, deoarece sunt semnate digital. JWT-urile sunt securizate cu o cheie secretă (folosind algoritmul HMAC) sau o pereche de chei publice/private folosind RSA sau ECDSA².

Cea mai comună utilizare a acestui standard este în cazul autentificării folosind nume de utilizator și parolă. Un utilizator se conectează la aplicație folosind credențialele înregistrate în baza de date, iar serverul începe verificarea acestor date. Dacă verificarea returnează un rezultat pozitiv (credențialele sunt corecte), serverul transmite un subiect sub formă JSON, conținând un jeton (Bearer Token) care i se atribuie utilizatorului pentru a

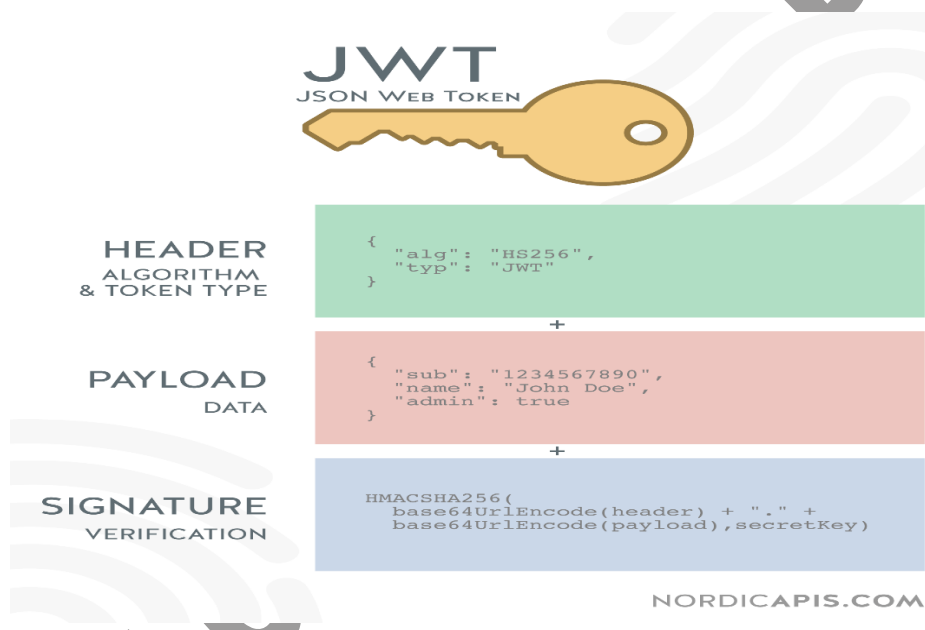
² <https://jwt.io/introduction/> , accesat la data de 27.05.2020

putea accesa aplicația în întregime. În esență, acest sistem se traduce prin „oferă acces către deținătorul acestui jeton” – de aici „*bearer*” (engl., deținător) + „*token*” (engl., jeton). Toate cererile pe care utilizatorul le va efectua către server vor conține acest jeton pentru a i se putea returna datele cerute.

A doua utilizare importantă a JSON Web Token este transferul de informații. Prin securizarea jetoanelor folosind chei publice sau private, se garantează transferul nealterat – identitățile celor doi utilizatori implicați în comunicație sunt confirmate, iar datele transmise sunt nemodificate pe parcursul transferului. Totuși, ca orice altă tehnologie de securizare, JWT nu este total invulnerabil în fața potențialilor atacatori.

Elementele componente JSON Web Token sunt:

- Antet (*header*) – conține tipul jetonului (JWT) și algoritmul de semnare (HMAC SHA256, RSA);
- Încărcătura (*payload*) – conține datele cerute;
- Semnătura (*signature*) – garantează faptul că transferul nu a fost alterat.



Figură II. 6. Structură JWT.

Sursă: https://miro.medium.com/max/3894/1*4W8JT5yod3lBvYSL6uN1DQ.png

4. Baze de date – MongoDB

4.1. Noțiuni generale

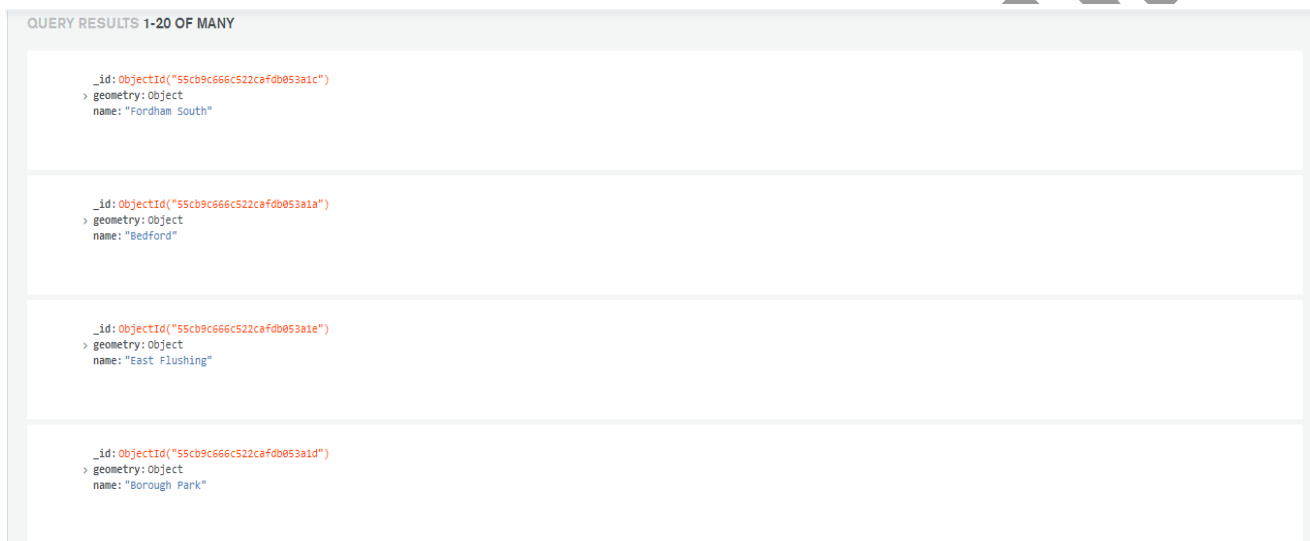
O bază de date reprezintă un set de informații stocate într-un mod organizat, de regulă, sub formă de tabele ce conțin înregistrări – rânduri. Fiecare înregistrare respectă structura impusă a tabelului din baza de date. Această modalitate de stocare definește modelul relațional, fiind cea mai răspândită din punct de vedere tehnic.

MongoDB este o bază de date NoSQL, a companiei 10gen. Dezvoltarea MongoDB a început în 2007, prima versiune fiind lansată în 2010, iar ultima versiune publicată datează din 2014. Limbajul de programare folosit pentru dezvoltarea acestei baze de date este C++. Aceasta oferă suport pentru o multitudine de sisteme de operare, printre care se numără Windows, Linux, Ubuntu, Debian și OS X.

O bază de date NoSQL oferă un mecanism de stocare și accesare a datelor, modelat prin alte mijloace decât relațiile tabulare utilizate în bazele de date relaționale.

Principala diferență dintre MongoDB și bazele de date relaționale este aceea că stocarea datelor se face folosind documente BSON (Binary JSON) și nu tabele, oferind flexibilitate și dinamism în procesul de stocare.

O colecție MongoDB este echivalentă unui tabel dintr-o bază de date MySQL. În loc de rânduri, într-o colecție se regăsesc obiecte de tip BSON, care nu limitează programatorul la un anumit tip de structură, cum este cazul MySQL. Așadar, MongoDB oferă flexibilitate în stocarea datelor. Totuși, un dezvoltator poate seta o structură, numită *Schema*, acest aspect fiind chiar recomandat pentru a putea face administrarea unei colecții mai simplă și mai intuitivă.



Figură II. 7. Exemplu de colecție MongoDB

4.2. Comparație MongoDB – MySQL

Cu toate că MongoDB este o bază de date de tip NoSQL, aceasta păstrează principii din structura MySQL.

MongoDB	MySQL
Bază de date	Bază de date
Colecție	Tabelă
Document BSON	Rând
Câmp BSON	Coloană
_id	Cheie primară

Tabel II. 1. Comparație termeni MongoDB - MySQL

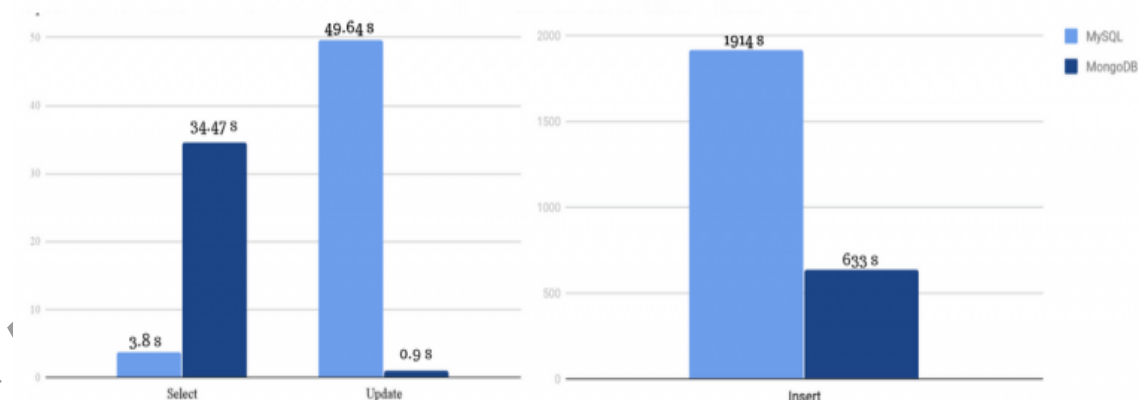
MongoDB	MySQL
SELECT * from tabel	db.tabel.find()
INSERT INTO tabel (camp1, camp2) VALUES (valoare1, valoare2)	db.tabel.insert({ camp1: valoare1, camp2: valoare2 })
DESCRIBE tabel	db.tabel.explain()

Tabel II. 2. Comparație comenzi MongoDB - MySQL

Din punct de vedere al evoluției, MongoDB utilizează o tehnologie superioară comparativ cu MySQL, deoarece datele salvate în bază pot evolua și își pot modifica structura. MySQL necesită modificarea întregii structuri a unui tabel pentru această operație, pe când MongoDB nu are nevoie de intervenții ulterioare pentru a fi capabil de a stoca date cu structuri diferite.

Viteza de procesare a MongoDB se datorează faptului că datele nu sunt modificate de îndată ce o cerere a fost trimisă, comenzile fiind executate asincron. Totuși, nu se poate spune că una dintre aceste două tehnologii este mai rapidă decât cealaltă, deoarece există variații ale MySQL care pot fi mai performante decât MongoDB și viceversa.

Nu există niciun standard ce poate oferi informații exacte despre superioritatea unei baze de date față de alta. Performanța unei baze de date este strâns legată de tipul de date cu care se lucrează și de complexitatea acestor date.



Figură II. 8. Comparație performanță MongoDB – MySQL.

Sursă: <https://www.simform.com/wp-content/uploads/2017/11/Add-subheading-1-1-768x402.png>.

4.3. MongoDB Atlas

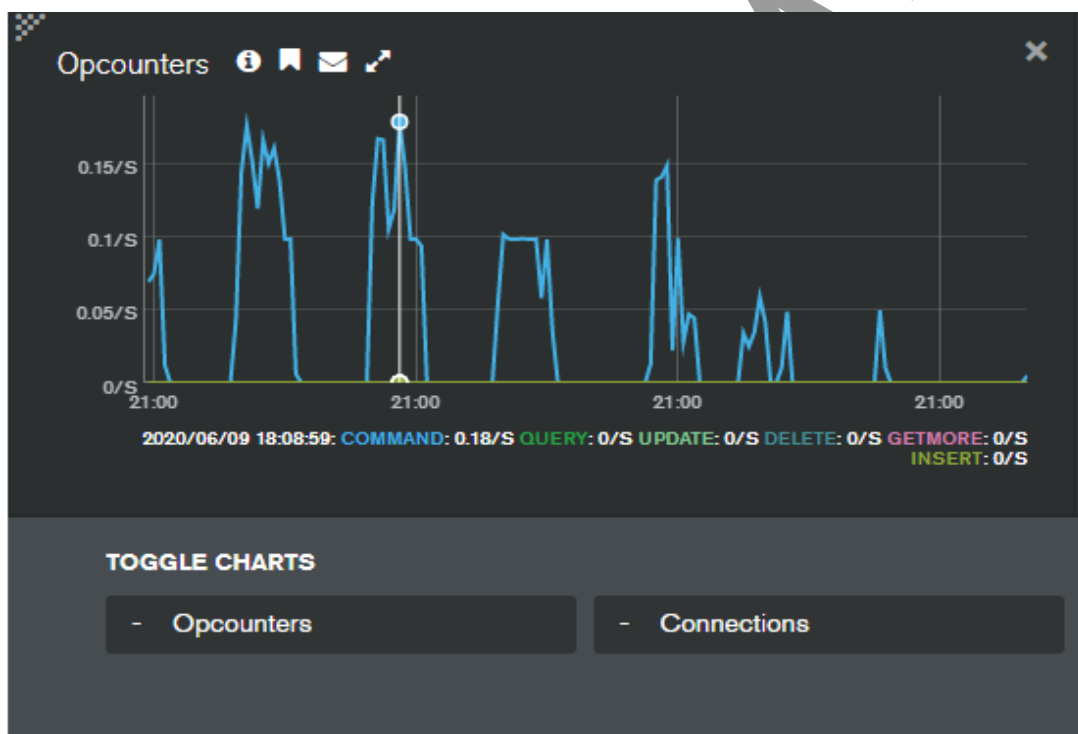
MongoDB Atlas este un serviciu de baze de date bazat pe cloud, care oferă găzduire bazelor de date MongoDB și care este disponibilă pe mai multe platforme. Acesta oferă posibilitatea de a crea gratuit un cluster –

un spațiu de stocare – pe unul din serverele MongoDB, prin intermediul căruia o aplicație poate accesa bazele de date.

Conectarea la un cluster se face utilizând un link generat de serviciul Atlas. Proprietarul cluster-ului trebuie să adauge utilizatori care pot accesa informațiile, iar link-ul de conectare trebuie să conțină credențialele unui utilizator pentru autentificare. Link-ul este interpretat de serverul aplicației – în cazul de față, serverul Node.js. După conectare, aplicația are acces la datele stocate pe cluster și le poate accesa folosind funcțiile de tip REST API.

Folosind acest serviciu, administratorul bazei poate să blocheze diferite IP-uri și să restricționeze accesul pe baza IP-ului. Atlas este monitorizat constant, generând alerte în cazul în care se găsește o breșă de securitate.

Pe lângă toate acestea, Atlas oferă și instrumente de editare/adăugare/ștergere de înregistrări într-o colecție printr-o interfață grafică sau printr-o linie de comandă, făcând manipularea datelor accesibilă pentru orice dezvoltator. Aceasta este utilitatea principală a serviciului oferit de MongoDB. Totodată, se pot vedea date statistice despre traficul pe un cluster (diagrame, înregistrări de IP).



Figură II. 9. Date statistice MongoDB Atlas - trafic pe cluster

5. Front End Development

5.1. Noțiuni generale

Front End Development reprezintă crearea interfeței unei aplicații web folosind HTML, CSS și JavaScript pentru a putea face posibilă interacțiunea dintre un utilizator și aplicație. Altfel spus, un dezvoltator Front End

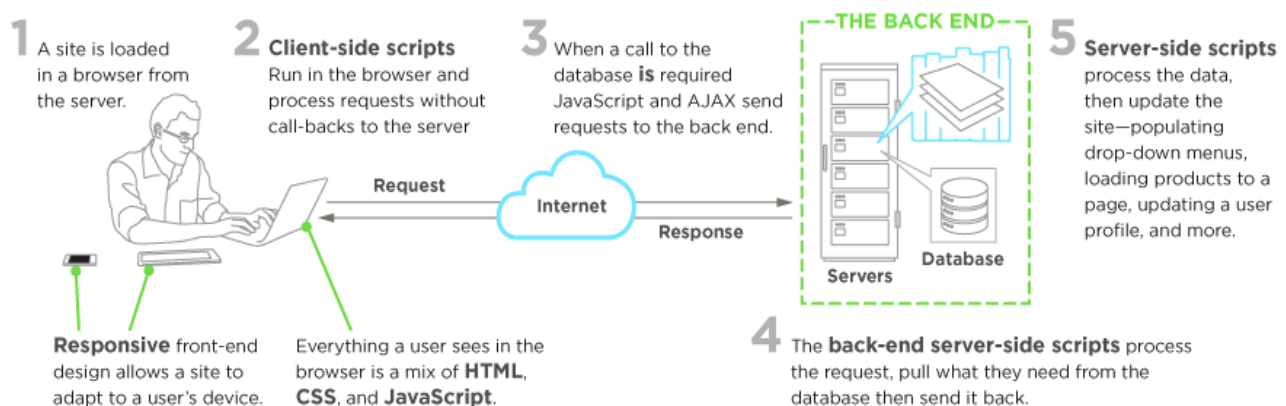
transformă date în elemente HTML, CSS și JavaScript, iar acestea sunt afișate într-un browser sub formă de pagină web.

Deseori, dezvoltatorul Front End este perceput ca fiind cel care se ocupă de aspectul unei aplicații. Această teorie este parțial adevărată, deoarece design-ul intră în sfera de dezvoltare Front End, dar nu este elementul exclusiv. Design-ul unei aplicații are două ramuri: interfața pe care o vede utilizatorul (UI – engl., *User Interface*) și experiența acestuia în interacțiunea cu elementele aplicației (UX – engl., *User Experience*).

Un dezvoltator Front End nu este obligat să gândească și partea de UI a unui proiect, deseori aceasta fiind stabilită de alte echipe din cadrul unei companii. Responsabilitatea programatorului în acest caz este implementarea interfeței și adaptarea acesteia.

Poate mai important decât aspectul este funcționalitatea elementelor ce alcătuiesc interfața. Intuitivitatea este cel mai important aspect pe care dezvoltatorul trebuie să îl aibă în vedere. Toate elementele componente trebuie să funcționeze așa cum un utilizator se așteaptă pentru ca aplicația să aibă succes.

Interacțiunile dintre utilizator și elemente pot avea legături în partea de server a aplicației, în partea de interfață sau chiar în amândouă. Cele care au legături cu partea de server implică Back End-ul, dar, până la a accesa baza de date, trebuie implementate funcții ce permit legarea Front End de Back End. Aceasta este partea de programare propriu-zisă ce intră în dezvoltarea Front End.



Figură II. 10. Dezvoltarea Front End.

Sursă: <https://www.upwork.com/hiring/development/front-end-developer/>

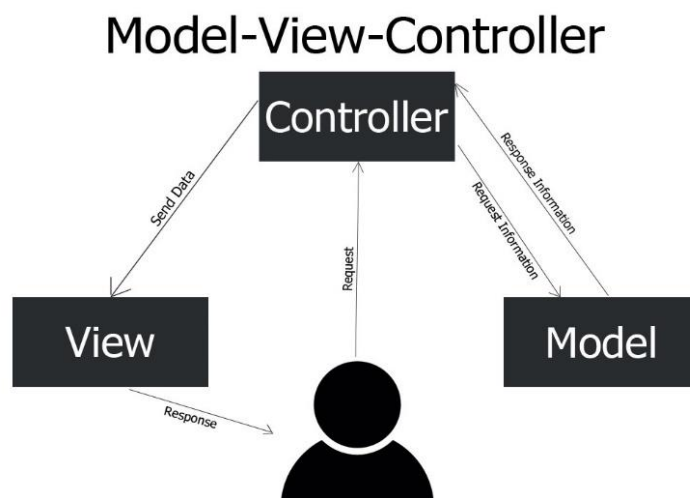
Baza unei aplicații web stă în trei tehnologii ce se completează reciproc:

- HTML – permite transpunerea informației în elemente care pot fi citite de un browser;
- CSS – se ocupă cu stilizarea elementelor HTML și aspectul general al acestora;
- JavaScript – oferă dinamism elementelor și face posibilă afișarea condiționată a informațiilor.

HTML și CSS sunt tehnologii care au stagnat din punct de vedere evolutiv. Pentru CSS au apărut biblioteci care au fost adoptate de majoritatea dezvoltatorilor pentru o uniformizare a aplicațiilor (exemplu: Bootstrap). Totuși, JavaScript este într-o continuă dezvoltare, iar un programator Front End este obligat să țină pasul cu noile

tehnologii pentru a se putea adapta. Acest aspect oferă dezvoltării Front End provocarea pe care un dezvoltator o caută atunci când își alege drumul în carieră.

O tehnică preluată din dezvoltarea software, regăsită și în cadrul Front End, este MVC (engl., *Model – View – Controller*), care impune o organizare a codului sursă al unei aplicații pentru a putea aduce modificări într-o singură secțiune care să nu afecteze întreg ecosistemul. Pentru a aplica această tehnică, s-au dezvoltat framework-uri pentru JavaScript care fac posibilă integrarea facilă, cum ar fi React.JS, AngularJS, Vue.js etc.



Figură II. 11. Diagramă MVC.

Sursă: <https://i2.wp.com/thereviewstories.com/wp-content/uploads/2018/05/MVC.jpg?fit=750%2C500&ssl=1>

5.2. HyperText Markup Language – HTML

Un limbaj de marcare este un set de reguli de formatare a unui text pentru o pagină web, care îmbină textul propriu-zis cu informații despre acesta exprimate prin marcatori. Diferențele dintre un limbaj de marcare și unul de programare sunt evidente, cel de marcare fiind un cod care este interpretat atât de un sistem, cât și de factorul uman, pe când limbajul de programare reprezintă un set de comenzi transmise către un sistem pentru a fi compilate și rulate.

HyperText Markup Language (HTML) este un limbaj de marcare utilizat pentru a crea pagini web care vor fi afișate într-un browser. HTML face posibilă structurarea informației pe o pagină web folosind marcaje (ex. <div> – diviziune, <p> – paragraf) care au la bază un set de instrucțiuni de formatare. Deseori, HTML este perceput ca un limbaj de programare, dar această afirmație nu este adevărată, el fiind doar o interpretare a scheletului unei pagini web.

O pagină HTML poate fi creată utilizând orice editor de text, fiind necesară specificarea faptului că ceea ce urmează a fi interpretat de sistem este o pagină web, folosind marcajul <!DOCTYPE html> la începutul documentului și </html> la sfârșit. Un fișier text se salvează cu extensia .html sau .htm pentru a fi interpretat de un browser.

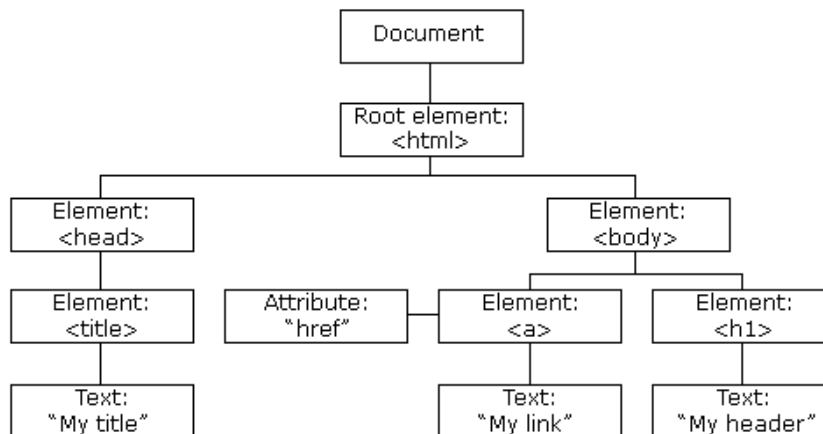
Un document HTML este structurat în trei secțiuni diferite:

- Antet – *header* – se specifică titlul documentului, logo-ul și importuri de script-uri JS sau CSS;
- Corp – *body* – reprezintă conținutul efectiv al paginii;

- Subsol – *footer* – de regulă, informații adiționale despre pagină sau dezvoltatorul acesteia.

Pe lângă marcasele de structură, se pot introduce elemente de conținut, cum ar fi imagini, folosind ``, videoclipuri, folosind `<video>` sau link-uri către alte pagini web, folosind `<a>`.

Atunci când o pagină HTML este încărcată de un browser, acesta generează modelul de obiect al documentului – *Document Object Model* (DOM). DOM-ul este construit ca un arbore de obiecte HTML, care vor fi controlate prin intermediul JavaScript.



Figură II. 12. Arborele DOM
Sursă: https://www.w3schools.com/whatis/img_htmltree.gif

Deoarece prin HTML se poate dezvolta doar structura unui website, trebuie incorporate și limbaje prin care să poată fi posibilă stilizarea și adăugarea funcționalității elementelor – CSS și JavaScript. Limbajul de marcare permite integrarea script-urilor JS și regulilor de formatare expuse într-un fișier CSS.

5.3. Cascading Style Sheets – CSS

Cascading Style Sheets (CSS) este un set de reguli de formatare pentru marcasele dintr-un document HTML. Regulile sunt atribuite elementelor HTML prin specificarea unui fișier extern cu extensia .css sau prin adăugarea marcajului `<style>` în antetul unui document HTML.

Fiecărui element HTML i se poate atribui o clasă sau un id, iar stilizarea se face, de regulă, pe baza acestor atribute. Se poate aplica un stil global pentru un marcaj prestabilit HTML.

Un exemplu pentru aplicarea unui anumit stil pe un element este următorul:

```
<p class="exemplu-clasa" id="exemplu-id">Acesta este un text</p>
```

Stilizare pe clasă	Stilizare pe id	Stilizare globală pe marcaj
.exemplu-clasa { color: red; }	#exemplu-id { color: red; }	p{ color: red; }

Tabel II. 3. Stilizare CSS

5.4. JavaScript – JS

JavaScript este un limbaj de programare orientat pe obiecte al cărui rol principal este adăugarea de funcționalități unui document HTML. Chiar dacă există o coincidență de nume, limbajul JavaScript nu are nicio legătură cu limbajul Java. Amândouă au o sintaxă asemănătoare cu cea scrisă folosind C, dar fiecare are particularități și execută sarcini diferite.

Codul JavaScript este integrat într-un document HTML prin specificarea sursei unui fișier .js extern, folosind marcajul `<script src="exemplu-script.js"></script>` în antetul documentului, sau prin marcajul `<script>//cod javascript</script>` adăugat oriunde este nevoie de acesta.

Prin JavaScript se atribuie funcții elementelor HTML și se pot manipula alte elemente dintr-o pagină web prin apelul clasei sau id-urilor acestora, folosind funcțiile `getElementByClassName('exemplu-clasa')` sau `getElementById('exemplu-id')`.

Odată cu evoluția tehnologiilor de Front End, programatorii au migrat către utilizarea funcțiilor JavaScript specifice elementelor HTML cu ajutorul jQuery, o bibliotecă JS care se ocupă strict cu administrarea elementelor din document, iar restul funcționalităților (stări intermediare ale aplicației, algoritmi de afișare condiționată etc.) sunt dezvoltate prin intermediul framework-urilor specifice JavaScript – React.js, Angular.js, Vue.js etc.

Framework-urile oferă posibilitatea structurării codului sursă și respectării principiului MVC, prin dezvoltarea organică a interfeței unei aplicații. În general, extensiile React.js sau Angular.js sunt dezvoltate de marile companii care caută să sporească producția proprie, dar oferă gratuit actualizările și îmbunătățirile aduse unui framework.

5.5. React.js

React.js este o bibliotecă JavaScript utilizată în dezvoltarea de interfețe pentru aplicații web. A fost dezvoltată de inginerii Facebook ca fiind un serviciu *open-source*, însemnând că orice dezvoltator poate folosi, modifica sau îmbunătăți codul sursă.

Funcția principală realizată de React.js este afișarea sau manipularea elementelor HTML dintr-o pagină web. Fiecărui element HTML i se poate atribui o componentă JavaScript, care poate fi construită în diverse moduri pentru a simplifica editarea acesteia și implementarea principiului MVC. Componentele modifică DOM-ul paginii web, declanșând reîncărcarea și actualizarea informațiilor afișate de browser³:

```
const element = <h1>Hello World</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

Afișarea elementelor HTML se face prin cod JSX⁴, o extensie de sintaxă JavaScript prin care realizează dezvoltarea simultană de cod JavaScript și cod HTML. Codul JSX scris în React.js definește un element de tip

³ <https://reactjs.org/docs/rendering-elements.html>, accesat la data de 02.06.2020

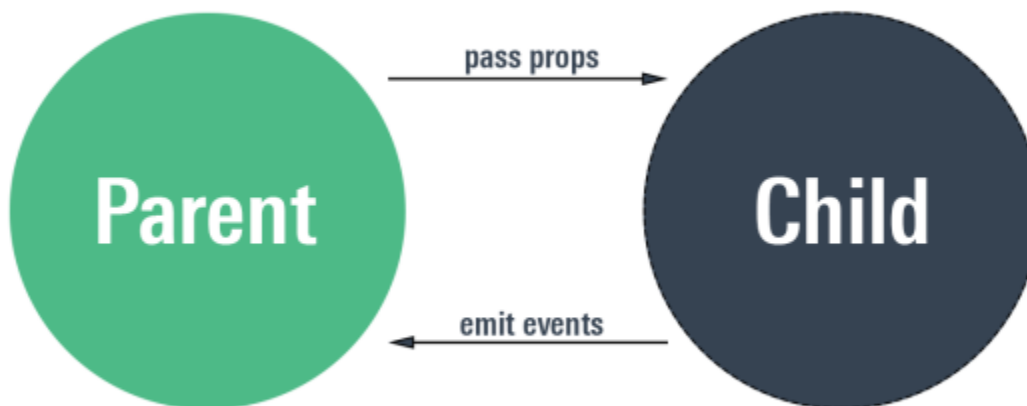
⁴ <https://reactjs.org/docs/introducing-jsx.html>, accesat la data de 02.06.2020

React. Folosind acest principiu, se facilitează afișarea condiționată de alte evenimente ce au loc în cadrul unei aplicații web (de exemplu, afișează un element numai după apăsarea unui buton).

Modelul recomandat după care se ghidează un dezvoltator React.js este cel relațional: fiecare element trebuie să reprezinte o componentă, având propria logică și propriile funcționalități. În general, există o componentă denumită „părinte” – elementul principal al unei pagini web care dictează logica oricărui alt element subordonat – denumit „copil”.

Componentele React.js au un constructor în care se inițializează obiectul de stare, *state*, ce menține starea în care se află componenta la un moment dat. Pe lângă obiectul de stare, se definește și obiectul de proprietăți, care este controlat de componenta părinte. Modificarea oricărui obiect dintre cele două declanșează reîncărcarea componentei, modificând DOM-ul și actualizând informația vizibilă în browser. Avantajul acestui principiu este faptul că se actualizează doar componenta modificată și nu tot documentul, așadar sunt rare cazurile în care reîncărcarea are o durată sesizabilă.

Legarea componentei părinte de subordonați se face prin transmiterea de proprietăți către aceștia și returnarea informațiilor, ce sunt interpretate prin funcții realizate de către dezvoltator. În procedura de afișare din părinte se face apelul către o componentă copil, declanșând metoda de afișare din aceasta.



Figură II. 13. Comunicația părinte - copil.

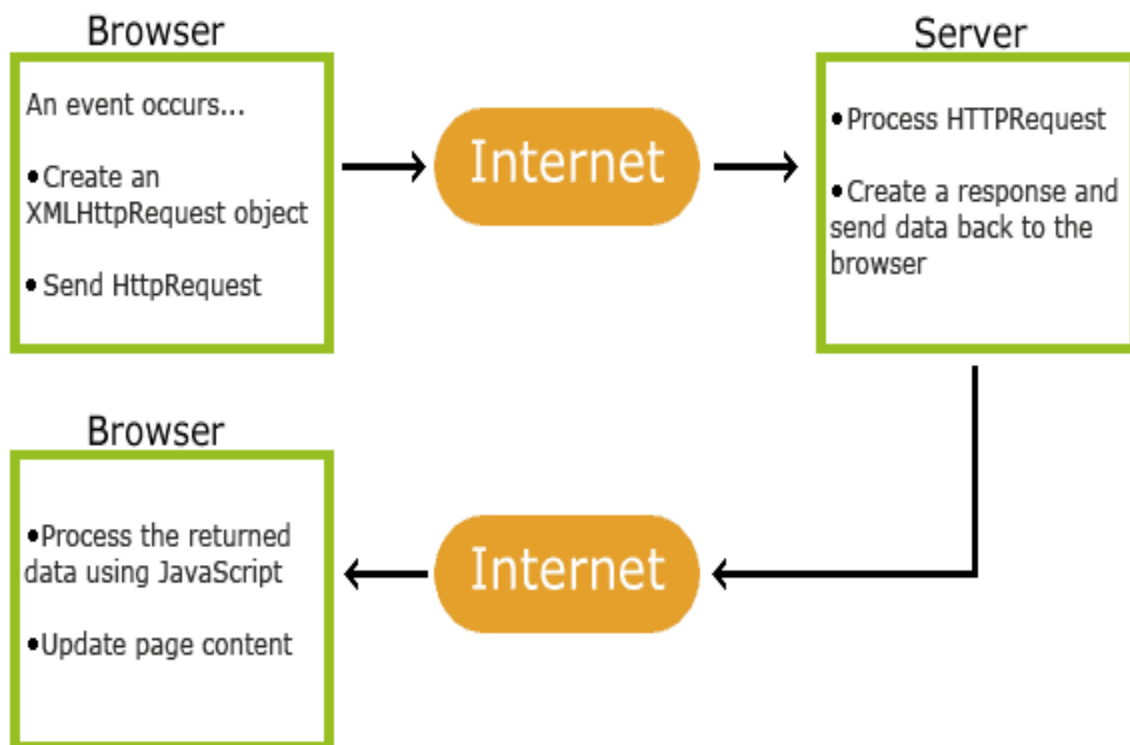
Sursă: https://miro.medium.com/max/640/1*HzAian51hZJBH3nHzKrpua.png

Avantajul major pe care React.js îl are față de competitori este acela că, pe lângă dezvoltarea de aplicații web, permite și programarea aplicațiilor pentru dispozitivele mobile, prin extensia React.Native – un framework specializat în crearea aplicațiilor iOS și Android. Diferența față de React.js o reprezintă modul în care cele două framework-uri interacționează cu afișajul: React.js influențează DOM-ul, iar React.Native influențează componentele mobile. Deoarece codul elaborat folosind React.Native rămâne același pentru orice platformă, dezvoltarea unei singure aplicații este suficientă pentru a o putea accesa de pe sisteme de operare diferite.

5.6. Asynchronous JavaScript and XML – AJAX

AJAX este o metodă prin care se pot face operații care accesează serverul unei aplicații web, în background, fără a influența interfața documentului în timpul efectuării operației. Scopul acestei tehnici este de a elimina nevoia de reîncărcare a unei pagină web după fiecare apel în baza de date, pentru a spori viteza și experiența pe care o are utilizatorul atunci când interacționează cu aplicația.

Pe baza principiilor AJAX au fost dezvoltate mai multe extensii similare, dar cu aplicabilitate sporită pentru anumite framework-uri. Una dintre aceste extensii este *axios*, un client HTTP special conceput pentru a fi integrat împreună cu Node.js⁵. Chiar dacă aceste principii se aplică pe partea de Back End, dezvoltarea algoritmilor AJAX este realizată de programatorul Front End pentru a putea influența în mod direct componentele create.



Figură II. 14. Principiul AJAX.
Sursă: <https://www.w3schools.com/xml/ajax.gif>

Cu toate că numele AJAX a rămas neschimbat, X-ul reprezentând XML (*Extensible Markup Language*), în practică, s-a renunțat la folosirea acestui format în detrimentul JSON, pentru a putea lucra direct cu obiectele pe care serverul le returnează în urma apelurilor făcute dintr-o aplicație, fără a mai fi nevoie de efectuarea conversiilor.

Deoarece partea de Back End din aplicația prezentată este construită pe baza Node.js, a fost preferată extensia *axios* în detrimentul tehnicilor AJAX de bază.

⁵ <https://www.npmjs.com/package/axios>, accesat la 03.06.2020

Capitolul III. Implementare

Primul pas pentru a începe dezvoltarea unei aplicații folosind stiva MERN este inițializarea pachetului npm pentru a putea importa toate framework-urile necesare programării prin rularea comenzii `npm init --y`. Această comandă creează fișierul `package.json`, în care sunt specificate toate framework-urile instalate și toate dependențele pe care aplicația le are.

După ce inițializarea npm este finalizată, se instalează pachetele care vor fi folosite în dezvoltarea serverului: *express*, *mongoose*, *body-parser*, *passport* și *nodemon*. Ultimul dintre acestea declanșează reîncărcarea serverului când se detectează modificări.

1. Implementare server

Înainte de a configura serverul propriu-zis, s-au importat bibliotecile necesare funcționării corecte și implementării cât mai apropiate de realitate:

```
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const passport = require("passport");
```

S-au importat, în ordinea liniilor de cod, *Express.js* – extensia Node.js pentru crearea serverului, *mongoose* pentru a ne conecta la baza de date MongoDB, *body-parser* pentru a formata răspunsurile primite din partea serverului sau informațiile transmise către acesta, iar *passport* pentru modulul de autentificare.

Următoarea etapă este stabilirea rutelor către colecțiile din baza de date, unde se vor trimite apeluri HTTP (GET/POST). În această etapă s-au atribuit unor constante URL-urile locale necesare.

```
const users = require("../routes/api/users");
const profile = require("../routes/api/profile");
const posts = require("../routes/api/posts");
const projects = require("../routes/api/projects");
const quizzes = require("../routes/api/quizzes");
const questions = require("../routes/api/questions");
const categories = require("../routes/api/categories");
const subjects = require("../routes/api/subjects");
const tasks = require("../routes/api/tasks");
const app = express(); - definește aplicația curentă ca o aplicație de tip Express.js
```

Prin extensia *body-parser*, se selectează tipul de format dorit pentru prelucrarea datelor. S-a ales formatul JSON pentru motive ce au fost enunțate în capitolul anterior.

```
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
```


Link-ul necesar conectării la baza de date este stocat într-un fișier extern și trebuie salvat într-o constantă accesibilă direct din server. Următoarea linie de cod realizează acest aspect.

```
const db = require("./config/keys").mongoURI;
```

mongoURI este link-ul propriu-zis ce face conexiunea la baza de date și este salvat în format *string*:
'mongodb://admin:admin123@ds117010.mlab.com:17010/devconn'

Urmează conectarea efectivă la baza de date prin folosirea funcției `connect()` din cadrul extensiei *mongoose*:

```
mongoose
  .connect(db, { useNewUrlParser: true })
  .then(() => console.log("MongoDB Connected"))
  .catch((err) => console.log(err));
```

`useNewUrlParser: true` are rolul de a intercepta eventualele modificări ale standardului de formatare pentru link-ul de conectare la baza de date. În cazul în care sunt detectate aceste modificări, link-ul este actualizat automat la cea mai nouă versiune.

Pentru modulul de login, se folosește *passport*, care trebuie inițializat și configurat în prealabil.

```
app.use(passport.initialize());
require("./config/passport")(passport);
```

Inițializarea se face prin apelul unei funcții predefinite din modul – `initialize()`, iar configurarea se alege pentru standardul JSON.

```
const options = {};
options.jwtFromRequest = ExtractJwt.fromAuthHeaderAsBearerToken();
options.secretOrKey = keys.appSecret;
module.exports = passport => {
  passport.use(new JwtStrategy(options, (jwt_payload, done) => {
    User.findById(jwt_payload.id).then(user => {
      if (user) {
        return done(null, user);
      }
      return done(null, false);
    }).catch(err => console.log(err));
  }));
};
```

Următoarea etapă este utilizarea efectivă a rutelor pentru baza de date, folosind funcția predefinită `use()`:

```
app.use("/api/users", users);
app.use("/api/profile", profile);
app.use("/api/posts", posts);
app.use("/api/projects", projects);
app.use("/api/quizes", quizes);
app.use("/api/questions", questions);
app.use("/api/categories", categories);
```


Ultimul aspect ce a fost implementat este portul pe care aplicația va rula în diferitele stadii (dezvoltare sau producție). Folosind modulul preinstalat, *port*, interceptăm stadiul în care aplicația se află. În cazul în care proiectul este în dezvoltare, se alege portul 5000, stabilit de programator, iar dacă este în producție, se va alege portul pe care îl asignează serviciul de găzduire prin folosirea variabilei *process.env.PORT*.

```
const port = process.env.PORT || 5000;
console.log("PORT", process.env.PORT);
app.listen(port, () => console.log("Server running on port " + port));
```

2. Implementare bază de date

Primul pas pentru configurarea bazei de date este crearea unui cluster pe pagina web MongoDB. După generarea acestuia, se creează utilizatori care pot accesa baza de date și se generează link-urile de conectare.

Următorul pas este stabilirea unui model pentru fiecare colecție pe care o vom avea. După ce am importat modulul *mongoose* pentru a putea lucra cu proprietatea *Schema* a bazei de date MongoDB, am construit modelul propriu-zis. O colecție MongoDB este, practic, o listă de obiecte JavaScript, așadar, se construiește obiectul și se specifică proprietățile acestuia, incluzând tipul de date al fiecărei proprietăți.

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;
```

```
const UserSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
  avatar: {
    type: String,
  },
  date: {
    type: Date,
    default: Date.now,
  },
  completedQuizes: [
    {
      quizId: {
        type: Schema.Types.ObjectId,
        ref: "quizes",
      },
    },
  ],
});
```

```

        grade: {
            type: Number,
        },
    },
],
subjects: [
    {
        subjectId: {
            type: Schema.Types.ObjectId,
            ref: "subjects",
        },
    },
],
tasks: [
    {
        name: {
            type: String,
            required: true,
        },
        deadline: {
            type: Date,
            required: true,
        },
    },
],
isAdmin: {
    type: Boolean,
    default: false,
},
});

```

Atunci când se folosește atributul *ref*, se face legătura între obiect și colecția specificată, folosind atributul unic *_id*, definit prin tipul *Schema.Types.ObjectId*.

Au fost create și scheme pentru colecțiile specifice materiilor și activităților pe care utilizatorul (studentul) le dorește salvate.

```

const SubjectSchema = new Schema({
    title: {
        type: String,
        required: true,
    },
});

```

```

const TaskSchema = new Schema({
    title: {
        type: String,
        required: true,
    },
    deadline: {
        Type: Date,
        Required: true,
    }
});

```

Pentru a conecta colecția MongoDB cu modelul creat se folosește funcția `model()` predefinită din modulul *mongoose*.

```
module.exports = User = mongoose.model("users", UserSchema);
```

După ce au fost create toate modelele necesare fiecărei colecții pe care o va conține baza de date, trebuie implementate rutele către colecții pentru a putea efectua operații din interiorul aplicației. Se folosește sintaxă Node.js, combinată cu operațiile specifice MongoDB.

```
const express = require("express");
const router = express.Router();
const gravatar = require("gravatar");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const keys = require("../config/keys");
const passport = require("passport");
const User = require("../models/User");
```

Așa cum s-a procedat și în cazurile precedente, se începe cu importurile necesare. Pentru a construi rutele se folosește funcția `Router()` din modulul *Express.js*. *bcryptjs* este folosit pentru a cripta parolele salvate în baza de date.

S-a dezvoltat o metodă de test prin care se poate vedea dacă accesul la colecția MongoDB este posibilă:

```
router.get("/test", (req, res) => res.json({ msg: "Users Works" }));
```

3. Implementare login/register

Metodele care îndeplinesc aceste două funcții trebuie dezvoltate având în vedere un standard foarte strict, deoarece pot avea loc atacuri în momentul în care serverul face verificarea datelor unui utilizator pentru login, dar și în timp ce datele sunt trimise pentru a fi stocate în bază.

Prima funcție returnează lista de obiecte conținând detaliile tuturor utilizatorilor înregistrați, după ce verifică dacă inițiatorul acestui apel este administratorul aplicației. Se verifică și dacă autentificarea utilizatorului s-a făcut prin *jwt*; dacă ambele verificări întorc un rezultat pozitiv, se aduc informațiile din baza de date, dar fără câmpul de parole, respectând standardul. Pentru a adăuga un strat adițional de securitate aplicației, parolele se regăsesc criptate în baza de date.

```
/** @route GET api/users
 * @desc User list
 * @access Private
 */

router.get(
  "/",
  passport.authenticate("jwt", { session: false }),
  (req, res) => {
    if (req.user.isAdmin) {
      User.find()
```

```

        .select("-password")
        .sort({ date: -1 })
        .then((users) => res.json(users))
        .catch((err) =>
            res.status(404).json({ noProjectFound: "No users found" })
        );
    } else {
        res.status(403).json({ forbidden: "You have no access to this area."
    });
    }
}
);

```

Următoarea metodă dezvoltată este cea de înregistrare. După trimiterea datelor de înregistrare de către utilizator către server, acestea trec printr-o funcție de validare (adresa de e-mail și parola trebuie să aibă structura corectă). Dacă validarea întoarce rezultat negativ, acesta este indicat prin mesaje de eroare pentru ca utilizatorul să le poată corecta. În caz contrar, datele trec de validare și se face o verificare adițională, de data aceasta pentru a vedea dacă utilizatorul este deja înregistrat. Dacă acesta nu se află în baza de date, un nou obiect JavaScript este creat, conținând datele utilizatorului. Obiectul este convertit în format JSON și trimis în baza de date pentru a fi salvat.

```

/** @route GET api/users/register
 * @desc User registration
 * @access Public
 */
router.post("/register", (req, res) => {
    const { errors, isValid } = validateRegisterInput(req.body);

    if (!isValid) {
        return res.status(400).json(errors);
    }

    User.findOne({ email: req.body.email }).then((user) => {
        if (user) {
            errors.email = "Email already exists";
            return res.status(400).json(errors);
        } else {
            const avatar = gravatar.url(req.body.email, {
                s: "200", // Size
                r: "pg", // Rating
                d: "mm", // Default
            });

            const newUser = new User({
                name: req.body.name,
                email: req.body.email,
                avatar: avatar,
                password: req.body.password,
                completedQuizes: [],
            });

```

```

    tasks: [],
    subjects: [],
  });

  bcrypt.genSalt(10, (err, salt) => {
    bcrypt.hash(newUser.password, salt, (err, hash) => {
      if (err) throw err;
      newUser.password = hash;
      newUser
        .save()
        .then((user) => res.json(user))
        .catch((err) => console.log(err));
    });
  });
}
});
});

```

```

▼ {isAdmin: false, _id: "5ee8d1e5771ef03b346026fe", name: "utilizator test", email: "test@test.com",...}
  avatar: "https://www.gravatar.com/avatar/b642b4217b34b1e8d3bd915fc65c4452?s=200&r=pg&d=mm"
  completedQuizes: []
  date: "2020-06-16T14:06:29.369Z"
  email: "test@test.com"
  isAdmin: false
  name: "utilizator test"
  password: "$2a$10$Avt910mmFRIGrtTZVsw3cOS382NxNKrKXP/54b4ISp1YdmeUsZVP6"
  subjects: []
  tasks: []
  __v: 0
  _id: "5ee8d1e5771ef03b346026fe"

```

Figură III. 1. Informațiile conținute de o cerere POST pentru înregistrare

Metoda de login este similară cu cea de înregistrare: datele utilizatorului trec prin validarea input-ului, după care se caută adresa de e-mail cu care utilizatorul încearcă să se conecteze. Dacă adresa este găsită, se criptează parola și se compară cu cea salvată în baza de date. În cazul parolei pe care factorul uman o introduce pentru conectare este aceeași cu cea din baza de date, acestuia i se atribuie un jeton pentru a putea accesa aplicația, folosind metoda JWT.

```

▼ {success: true,...}
  success: true
  token: "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjVlZThkMUU1NzcwZWYwM2IzNDYwMjZmZSI6Im1zQWRtaW4iOmZhbHNlLCJyYWI1IjoiaidXRpbG16YXRv

```

Figură III. 2. Răspunsul serverului pentru o autentificare validată

Dacă datele de conectare nu sunt validate, prin interfața grafică a aplicației sunt afișate mesajele de eroare întoarse de server.

```

▼ {password: "Password is incorrect"}
  password: "Password is incorrect"

```

Figură III. 3. Mesaj de eroare întors de server

```

/** @route POST api/users/login
 * @desc Login User / Returning JWT Token
 * @access Public
 */

router.post("/login", (req, res) => {
  const { errors, isValid } = validateLoginInput(req.body);

  if (!isValid) {
    return res.status(400).json(errors);
  }

  const email = req.body.email;
  const password = req.body.password;

  User.findOne({ email }).then((user) => {
    /** Check for user */
    if (!user) {
      errors.email = "User not found";
      return res.status(404).json(errors);
    }

    bcrypt.compare(password, user.password).then((isMatch) => {
      if (isMatch) {
        /** User match & Create JWT Payload*/
        const payload = {
          id: user.id,
          isAdmin: user.isAdmin,
          name: user.name,
          avatar: user.avatar,
        };

        jwt.sign(
          payload,
          keys.appSecret,
          { expiresIn: keys.expireIn },
          (err, token) => {
            res.json({ success: true, token: "Bearer " + token });
          }
        );
      } else {
        errors.password = "Password is incorrect";
        return res.status(400).json(errors);
      }
    });
  });
});
});

```

După ce serverul primește o cerere de înregistrare pe care o execută până la final, fără erori, aplicația redirecțiază utilizatorul către pagina de login, folosind funcția `push()` din cadrul modulului *history*. Prin *history* se salvează toate URL-urile din cadrul aplicației care au fost vizitate în timpul utilizării.

```
export const registerUser = (userData, history) => dispatch => {
  axios
    .post('/api/users/register', userData)
    .then(res => history.push('/login'))
    .catch(err => dispatch({
      type: GET_ERRORS,
      payload: err.response.data
    }));
};
```

Pentru a nu fi nevoie de apeluri în baza de date de fiecare dată când utilizatorul dorește să facă operații în interiorul aplicației, imediat după autentificare, se salvează datele și jetonul atribuite acestuia în starea globală a aplicației. Efectuarea operației necesită utilizarea unui modul a cărui funcționalitate principală este exact acest aspect; în cazul de față, s-a folosit Redux.

```
export const setCurrentUser = decoded => {
  return {
    type: SET_CURRENT_USER,
    payload: decoded
  }
}
```

Deoarece apelurile efectuate în bază sunt efectuate asincron, pentru a intercepta răspunsul serverului se folosește metoda `then()` din cadrul *axios*.

```
export const loginUser = userData => dispatch => {
  axios
    .post('api/users/login', userData)
    .then(res => {
      const { token } = res.data;
      localStorage.setItem('jwtToken', token);
      setAuthToken(token);
      const decoded = jwt_decode(token);
      dispatch(setCurrentUser(decoded));
    })
    .catch(err => dispatch({
      type: GET_ERRORS,
      payload: err.response.data
    }));
};
```

Prin apelarea funcției de `logout`, aplicația trimite către Redux un obiect gol. Detectarea unui astfel de obiect reprezintă faptul că aplicația nu mai este folosită de jetonul asignat până în acel punct.

```
dispatch(setCurrentUser({}));
```