|  |  |  |
| --- | --- | --- |
|  | **Departamentul Automatică şi Informatică Industrială**  **Facultatea Automatică şi Calculatoare**  **Universitatea Națională de Știință și Tehnologie POLITEHNICA BUCUREȘTI** |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |

**LUCRARE DE DIPLOMĂ**

**BusHop – Sistem de gestiune a curselor de autocar**

Coordonator Absolvent

Conf. Dr. Ing. Adriana Olteanu Ion-Tudor Bondoc

**2024**

Cuprins

[1. Introducere 3](#_Toc169277702)

[2. Descrierea domeniului ales 3](#_Toc169277703)

[2.1 Aplicații similare 4](#_Toc169277704)

[3. Descrierea problemei abordate și a metodei de rezolvare propuse 4](#_Toc169277705)

[3.1 Tipuri de utilizatori 5](#_Toc169277706)

[3.1.1 Administratorii sistemului 5](#_Toc169277707)

[3.1.2 Clienți (pasageri) 6](#_Toc169277708)

[3.1.3 Șoferi 6](#_Toc169277709)

[3.2 Tipul aplicației 7](#_Toc169277710)

[3.3 Echipamente software utilizate 7](#_Toc169277711)

[3.3.1 MySQL Workbench 8](#_Toc169277712)

[3.3.2 Visual Studio Code 8](#_Toc169277713)

[3.3.3 Insomnia API Tester 8](#_Toc169277714)

[3.4 Tehnologii 9](#_Toc169277715)

[3.4.1 MySQL 9](#_Toc169277716)

[3.4.2 Express (framework al Node.js) 9](#_Toc169277717)

[3.4.3 React.js (bibliotecă JavaScript) 10](#_Toc169277718)

[3.5 Arhitectura sistemului 12](#_Toc169277719)

[4. Documentație tehică 13](#_Toc169277720)

[4.1 Proiectarea bazei de date 13](#_Toc169277721)

[4.2 Implementarea server-ului 17](#_Toc169277722)

[4.3 Implementarea aplicațiilor front-end 25](#_Toc169277723)

[4.3.1 Aplcația de administrare 27](#_Toc169277724)

[4.3.2 Aplicația destinată pasagerilor 32](#_Toc169277725)

[5. Bibliografie 33](#_Toc169277726)

# Introducere

Astăzi, mai mult ca oricând, societatea se bazează pe tehnologie și pe progresul tehnologic în aproape toate aspectele și sarcinile din viața cotidiană. Fie că avem nevoie de o cursă de tip „taxi” sau „ridesharing”, fie că vrem să vindem obiectele din casă de care nu mai avem nevoie sau să comandăm felul favorit de mâncare, avem la dispoziție o gamă variată de aplicații și platforme care să faciliteze aceste acțiuni, reducându-le la o chestiune de doar câteva click-uri.

Din această categorie de servicii fără de care mulți nu și-ar putea derula activitățile într-un mod rapid și eficient face parte și nevoia de găsi și rezerva un loc pentru o cursă de autocar. Chiar dacă unele persoane preferă încă să apeleze la sediul firmei de transport și să facă rezervări telefonic, din ce în ce mai mulți aleg să își managerieze toate aceste detalii prin confortul unui cont intr-o platformă online sau aplicație mobilă. Astfel, pasagerii pot avea parte de mult mai multe beneficii, printre care selectarea locului dorit (cu actualizarea locurilor ocupate în timp real), obținerea rezervării electronice și urmărirea cursei de autocar în timpul desfășurării acesteia.

Lucrarea de față își propune să detalieze toate beneficiile unui astfel de sistem de gestiune a curselor de autocar, punând accent pe toate etapele implementării și scoțând în evidență detaliile care aduc un plus în comparația cu soluțiile existente deja pe piață.

# Descrierea domeniului ales

Conform unui studiu realizat de FlixBus, prezentat de site-ul „bursa.ro”, autocarul este mijlocul de transport preferat de români când vine vorba de călătoriile lungi din țară. „În acest context, 80% dintre românii care obişnuiesc să meargă cu autocarul pe distanţe lungi în ţara noastră, preferă acest mijloc de transport în detrimentul altora. Autoturismul personal este preferat de 55% dintre respondenţi, în timp ce alternativa de transport de pasageri pe cale ferată este preferată de doar 43% dintre aceştia.” [2]

Prin intermediul unei aplicații de rezervare a locurilor pentru cursele de autocar, clienții își pot crea un cont, pot căuta curse după un anumit traseu și pot rezerva un loc.

## Aplicații similare

În acest moment sunt disponibile pe piață mai multe platforme destinate companiilor ce efectuează curse regulate de autocar, fiecare prezentând diferite funcționalități, precum și avantaje și dezavantajate. Iată o listă cu o parte din aceste platforme, folosite uzual în țara noastră:

* Dacos: Utilizatorii pot căuta curse introducând orașul de plecare și orașul destinație și data plecării. Rezervările sunt alcătuite pe baza completării unui formular. [3]
* Ionescu: Funcționalități similare cu Dacos. [5]
* Autogari.ro și bileteria.ro: platforme de tip intermediar între clienți și companiile de transport. [1]
* Flixbus: Platformă destinată călătoriilor internaționale. Pe lângă funcționalitățiile uzuale, clienții pot urmări status-ul unei curse cu ajutorul numărului unic de identificare a rezervării. [4]
* Busradar: Platformă de comparație a diverselor curse de autocar oferite de companii diferite. [Busradar]

Deși aceste platforme oferă majoritatea funcționalităților necesare clienților, am urmărit, prin dezvoltarea acestui sistem de aplicații, să introduc noi opțiuni care să contribuie la o experiență cu utilizatorul cât mai plăcută și mai eficientă, opțiuni ce vor fi discutate pe larg în capitolele următoare.

# Descrierea problemei abordate și a metodei de rezolvare propuse

Încă din etapa de proiectare am plecat de la ideea separării aplicațiilor pentru diferitele tipuri de utilizatori și sarcinile/cazurile de utilizare ale acestora. Astfel, am identificat trei tipuri de clase de utilizatori ce vor avea parte de interfețe diferite, dedicate nevoilor acestora.

## Tipuri de utilizatori

### Administratorii sistemului

Prima categorie de utilizatori este reprezentată de administratorii platformei: angajații responsabili de gestionarea (adăugarea, modificarea și ștergerea) tuturor elementelor care, puse cap la cap, formează ansamblul companiei de transport de persoane. Administratorii vor avea la dispoziție o aplicație dedicată, prin intermediul căreia vor putea să interacționeze cu traseele, autocarele, cursele și rezervările din sistem. Aceștia sunt o componentă esențială a sistemului, întrucât ei vor fi cei responsabili de coerența și consistența datelor din sistem (rezervări valide, curse corecte din punct de vedere al traseelor și al timpilor estimați, etc.). Diagrama cazurilor de utilizare pentru administratori este prezentată în Figura 1.

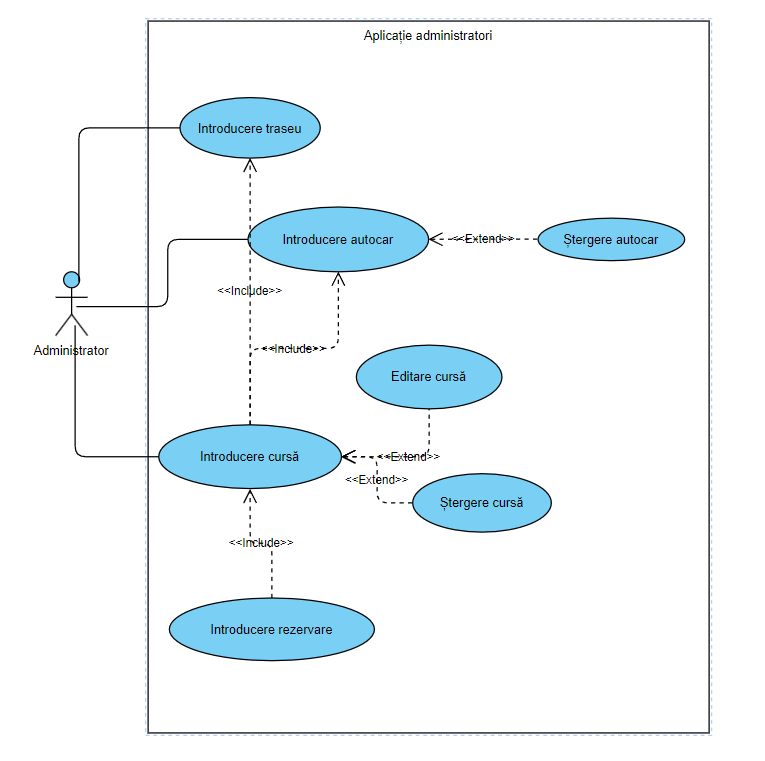


Fig. 1. Diagrama cazurilor de utilizare pentru administratori

### Clienți (pasageri)

Pasagerii sunt utilizatorii care vor beneficia de pe urma acestui sistem de aplicații. Aceștia își vor putea crea un cont și se vor putea autentifica, vor putea căuta cursa dorită pe baza unor filtre (oraș de plecare, oraș destinație și data plecării). De altfel, aceștia vor putea să facă rezervări, selectând locul dorit din autocar pe baza unei interfețe grafice și, ulterior, vor putea urmări locația autocarului în timp real, în timpul desfășurării cursei. Diagrama cazurilor de utilizare pentru pasageri este prezentată în Figura 2.

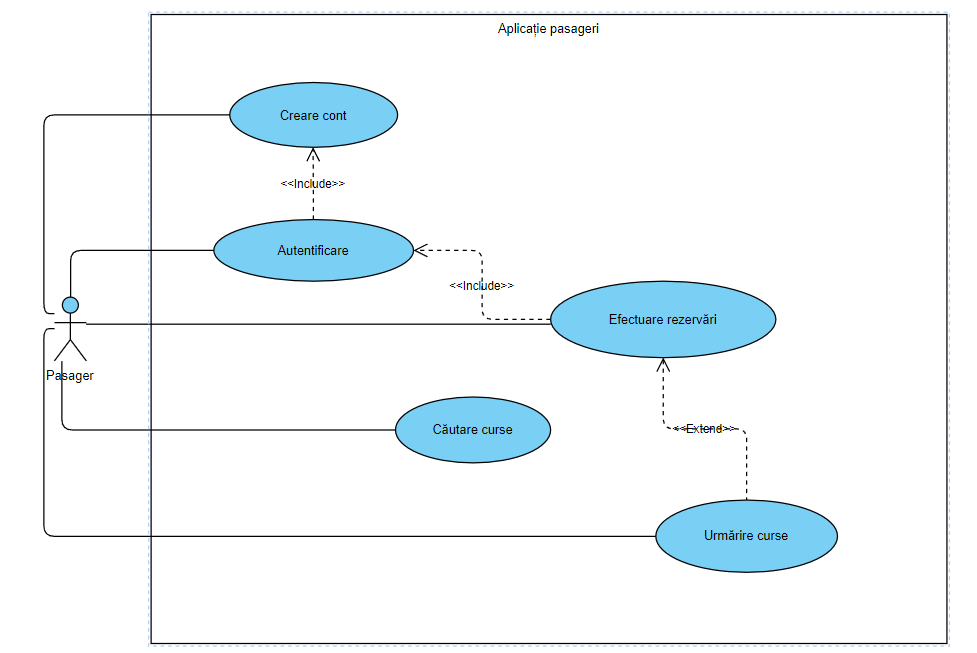


Fig. 2. Diagrama cazurilor de utilizare pentru pasageri

### Șoferi

Cea de-a treia categorie de utilizatori ai platformei este reprezentată de șoferii de autocar. Aceștia vor dispune de propria aplicație prin intermediul căreia vor putea începe o cursă, vor transmite locația curentă în mod continuu către aplicația clienților și, ulterior, vor finaliza cursa (Figura 3).

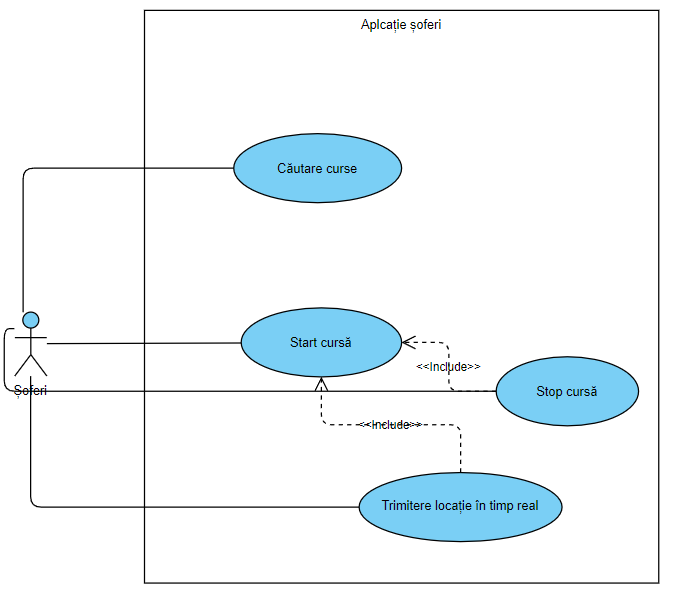


Fig. 3. Diagrama cazurilor de utilizare pentru șoferi

## Tipul aplicației

Dată fiind natura cerințelor pentru acest sistem de aplicații, am ales să implementez ansamblul urmărind arhitectura RESTful (Representational State Transfer) Web Services. Această arhitectură este tratată pe larg în [Masse, M] și are la bază separarea conceptelor de server și client într-o comunicație web. Prin această separare, atât aplicațiile de tip server, care deservesc cereri venite de la clienți, cât și clienții sunt construiți independent unii față de ceilalți, niciuna din părți neavând cunoștințe despre particularitățile celeilalte. Mai mult, un server nu știe nimic despre starea actuală în care se află un client și viceversa.

Comunicația între server și client se realizează prin protocolul HTTP (Hypertext Transfer Protocol), acesta fiind protocolul fundamental al World Wide Web. Elementele esențiale ale acestei comunicații sunt cererile și răspunsurile HTTP. Clientul emite cereri către server prin mai multe metode (CRUD – Create, Read, Update, Delete) prin care solicită fie adăugarea de date noi, citirea, modificarea sau ștergerea datelor existente. Ulterior, serverul trimite un răspuns HTTP, în care include un cod de stare (pentru a notifica clientul de succesul sau eșuarea cererii) și un câmp ce conține datele solicitate.

## Echipamente software utilizate

Pentru a implementa toate componentele necesare funcționării aplicațiilor, am ales să folosesc un mediu de dezvoltare și administrare pentru baza de date (MySQL Workbench) și un mediu de dezvoltare a codului pentru aplicațiile de tip backend (server) și frontend (clienți) – VS Code. De altfel, pentru a testa funcționalitățile serverului, am folosit aplicația Insomnia.

### MySQL Workbench

MySQL Workbench [MySQL Workbench] este o interfață vizuală complexă destinată proiectanților de baze de date, dezvoltatorilor și administratorilor. Aceasta oferă funcționalități pentru proiectarea vizuală, administrarea și dezvoltarea bazelor de date MySQL.

MySQL Workbench permite crearea vizuală a diagramelor de tip ER (Entity Relationship – entitate relație) complexe, facilitând proiectarea eficientă a bazelor de date. O diagramă ER conține toate categoriile de obiecte, caracteristicile acestora și relațiile dintre ele. Cu ajutorul editorului SQL integrat, utilizatorii pot crea interogări pentru a obține sau modifica datele existente (DML – Data Manipulation Language). În același timp, este posibilă definirea tuturor structurilor necesare funcționării bazei de date (DDL – Data Definition Language), cum ar fi scheme, tabele, etc.

Pentru conveniență, aplicația poate fi configurată să ruleze în permanență serverul MySQL pe mașina locală (localhost). În cazul meu, acesta rulează pe portul 3306.

### Visual Studio Code

Visual Studio Code (VS Code) este un editor de cod extrem de versatil și bine optimizat, cu ajutorul căruia am realizat aplicațiile de backend și frontend. Conform [Visual Studio Code], acesta oferă suport în mod implicit pentru aplicațiile scrise în JavaScript și aplicațiile Node.js. Node.js este un mediu de execuție JavaScript care ne permite să creăm aplicații de tip back-end care sunt independente de un browser, după cum reiese din [What is Node.js?]. Pe lângă dezvoltarea de aplicații JavaScript, VS Code poate fi modificat facil pentru a fi compatibil și a oferi suport pentru o gamă extinsă de limbaje de programare.

Un avantaj mare al utilizării VS Code în cadrul acestui proiect a fost integrarea acestuia cu utilitarul de versionare Git și platforma Github. Astfel, am avut posibilitatea să dezvolt proiectul pe mai multe ramuri (în terminologia Git – branches) dedicate noilor funcționalități sau depanării de erori. În același timp, platforma Github reprezintă un mediu bun de back-up, în caz de distrugere sau corupere a fișierelor aplicației pe mașina locală.

### Insomnia API Tester

După cum reiese din [Insomnia], Insomnia este o aplicație desktop destinată interacțiunii cu un API (Application Programming Interface). Un API este „o interfață software care permite mai multor programe să comunice între ele”, conform definiției din [API. Wikipedia]. Pe parcursul proiectului, această aplicație a avut un rol foarte important în testarea cererilor implementate în API-ul pe care l-am creat. Spre deosebire de un browser, prin intermediul căruia putem testa doar cererile de tip „GET”, Insomnia oferă o platformă capabilă să testeze și celelalte tipuri de cereri.

## Tehnologii

În implementarea proiectului am ales să folosesc următoarele tehnologii: pentru baza de date – MySQL; pentru server-ul API am folosit Express.js (cel mai popular framework al Node.js) iar pentru aplicațiile front-end am folosit React.js (bibliotecă JavaScript). Mai mult, am realizat versionarea proiectului folosind Git și am încărcat periodic actualizările pe platforma Github.

### MySQL

MySQL este un sistem de gestiune a bazelor de date relaționale de tip open-source (codul sursă al sistemului este disponibil tuturor utilizatorilor). Conform [MySQL], MySQL este dezvoltat în C și C++ și a fost creat de compania suedeză MySQLAB, care ulterior a fost achiziționată de către Sun Microsystems (actualmente corporația Oracle).

Serverul MySQL poate deservi mai multe baze de date (în terminologia MySQL – scheme; a nu se confunda cu schemele din alte sisteme de gestiune a bazelor de date relaționale, cum ar fi PostgreSQL sau Oracle). În [DuBois] se oferă o imagine de ansamblu asupra administrării bazei de date MySQL. Serverul este componenta principală, responsabilă cu gestionarea bazelor de date, tabelelor și celorlalte obiecte. Pe platformele de tip Unix, sunt prezente script-uri care să asiste buna funcționare a server-ului. Administratorii de sistem trebuie să fie familiarizați cu structura directoarelor de date, cu procesele ce rulează în fundal și cu uneltele de back-up, pentru a asigura consistența și disponibilitatea permanentă a datelor.

### Express (framework al Node.js)

În mod tradițional, limbajul JavaScript a fost folosit pentru a scrie aplicații web de tip client (front-end). De ce? Pentru că JavaScript are nevoie de un mediu de execuție pentru a putea rula. Browserele web au integrat acest mediu de execuție JavaScript, făcând posibilă rularea acestuia în aplicații. Odată cu introducerea Node.js, programatorii au putut folosi JavaScript și pentru a scrie aplicații web destinate logicii de server (back-end).

Express.js este un framework al Node.js utilizat în special în dezvoltarea de API-uri și aplicații de tip server. Așa cum reiese din definiția din [Framework software], „un framework software este o abstractizare prin care niște programe, ce oferă o funcționalitate generică, pot fi modificate selectiv, prin cod scris de utilizator, creând astfel o aplicație specifică”. Express oferă posibilitatea de a crea un server HTTP și de a defini portul pe care acesta „ascultă” pentru a primi cereri. De altfel, programatorul poate crea așa numitele „endpoint-uri” – funcții ce spun serverului ce să facă în momentul în care primește o anumită cerere către o anumită cale.

Este de menționat faptul că în implementarea aplicației back-end am utilizat multe alte biblioteci și framework-uri ale Node.js, acestea având însă rol secundar, urmând a fi detaliate în capitolele următoare.

### React.js (bibliotecă JavaScript)

React este o bibliotecă JavaScript open-source, dezvoltată de Facebook, utilizată pentru dezvoltarea aplicațiilor de tip front-end, în special a aplicațiilor web cu interacțiuni complexe și dinamice. Lansată în 2013, React a devenit rapid una dintre cele mai populare tehnologii de dezvoltare web, datorită modului său eficient de manipulare a DOM-ului (Document Object Model) și a structurilor sale bazate pe componente. În general, prin DOM înțelegem o structură arborescentă ce conține noduri și atribute. Așa cum reiese din [DOM], în mod tradițional, o pagină web HTML (HyperText Markup Language – limbajul standard de marcare destinat dezvoltării de pagini web pentru a fi afișate în browser) este privită de către browser precum un DOM care este parsat și ulterior afișat.

Nu putem însă expune alte caracteristici ale React, fără a puncta mai întâi câteva aspecte cheie ale JavaScript. O imagine de ansamblu asupra limbajului este oferită în [Ullman]. JavaScript este un limbaj de programare orientat pe obiecte destinat dezvoltării de aplicații web. Spre deosebire de un limbaj procedural, majoritatea variabilelor din JavaScript sunt, în esență, obiecte, care au proprietăți și metode. Cu toate acestea, JavaScript diferă de un limbaj clasic OOP (Object Oriented Programming) în sensul că nu vom crea singuri clasele și obiectele (instanțe ale claselor), așa cum facem de exemplu în Java. Cu alte cuvinte, JavaScript este un limbaj „bazat pe prototipuri”. În același timp, JavaScript este un limbaj în care variabilele sunt converite dinamic, relativ ușor, de la un tip de date la altul (spre exemplu, un număr întreg poate fi tratat ca un string și invers). Nu în ultimul rând, JavaScript este un limbaj de programare orientat pe evenimente și este cunoscut pentru modul său asincron de execuție. Acest lucru permite gestionarea eficientă a operațiunilor de lungă durată, cum ar fi solicitările de rețea, fără a bloca firul principal de execuție.

Revenind la React, această bibliotecă este axată pe crearea de componente reutilizabile. O componentă este un bloc independent și reutilizabil de cod (poate fi o funcție sau o clasă) care definește un element UI (User Interface). Prin combinarea acestor componente, dezvoltatorii pot construi interfețe grafice cu utilizatorul complexe și coerente, adăugând dinamică aplicațiilor web. React utilizează JSX (JavaScript XML), o unealtă care permite scrierea de elemente HTML direct în JavaScript, oferind o sintaxă intuitivă și ușor de utilizat.

React introduce conceptul de Virtual DOM pentru a îmbunătăți performanța și eficiența manipulării DOM-ului tradițional. Virtual DOM este o reprezentare în memorie a DOM-ului real. Când starea unei componente React se schimbă, acesta creează un nou Virtual DOM și îl compară cu versiunea anterioară. Astfel, React identifică diferențele dintre cele două versiuni și calculează cele mai eficiente actualizări necesare pentru a aduce DOM-ul real în concordanță cu noul Virtual DOM, fapt ce conferă viteză și fluiditate în rularea aplicațiilor.

Un avantaj major introdus de React este reprezentat de reutilizarea codului (mai exact, reutilizarea componentelor). Putem observa în din ce în ce mai multe aplicații web tendința de a expune anumite liste de obiecte ce sunt construite asemănător (spre exemplu, filmele, în toate aplicațiile populare de streaming, sunt afișate sub forma unor „carduri” care sunt construite după același tipar: titlul, regizorul, vârsta minimă recomandată, etc, toate apar în același loc, dar diferă, bineînțeles, prin conținut). Să ne imaginăm acum cât de greu și repetitiv ar fi să codăm individual fiecare componentă, cu informațiile corespunzătoare. Din fericire, folosind React, putem crea o componentă reutilizabilă (o componentă ce conține așa zisele „props” – placeholderi – pe care o creăm o singură dată). Ulterior, folosind funcția „map()” din JavaScript (funcție ce prelucrează un vector, returnând un alt vector, pe baza celui de intrare) putem crea un vector de componente modificate cu datele specifice (datele pe care le avem în vectorul de intrare), după cum se poate observa în Figura 4.

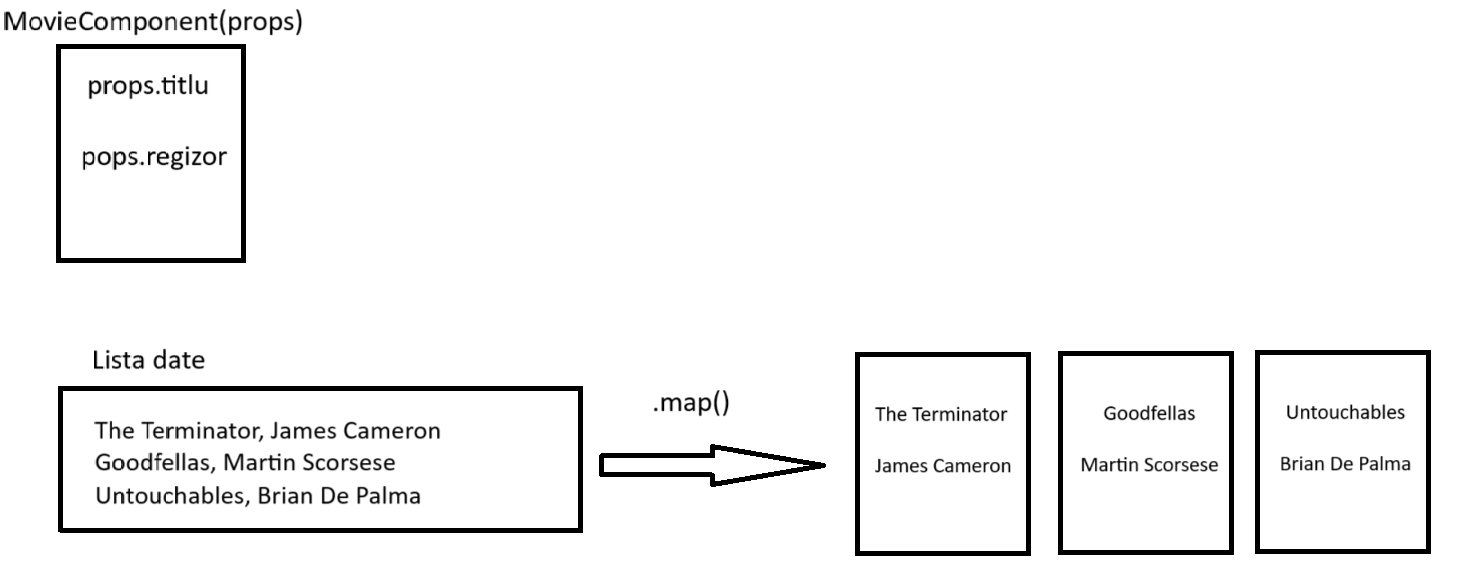


Fig. 4. Exemplu de reutilizare a codului folosind componente React

O altă caracteristică importantă a React este reprezentată de ancore (hooks). Acestea constituie funcții JavaScript care ne ajută să separăm logica funcțională a componentelor de logica ce gestionează starea (state-ul) sau alte efecte secundare. Cele mai importante ancore pe care le-am folosit în cadrul dezvoltării aplicațiilor front-end sunt „useState”, „useEffect” și „useNavigate”. Voi detalia fiecare din aceste concepte în capitolele ce urmează, evidențiind și exemple concrete de utilizare.

## Arhitectura sistemului

După cum am explicat la începutul acestui capitol, am gândit o arhitectură bazată pe separarea aplicațiilor de tip client (front-end) pentru cele trei categorii de utilizatori ai sistemului, și anume: administratori, pasageri și șoferi. Din punctul meu de vedere, această abordare aduce mai multe beneficii, printre care:

* Separarea clară a responsabilităților: fiecare aplicație are un scop bine definit, pretat pe nevoile categoriei de utilizatori pentru care este destinată;
* Simplificarea dezvoltării și a mententanței: pot aduce modificări punctuale unei aplicații, fără a pune în pericol integritatea și funcționalitatea celorlalte aplicații;
* Perofrmanță: având o încărcătură redusă, aplicațiile răspund mult mai bine interacțiunii cu utilizatorii, în comparație cu o singură aplicație care să gestioneze toate situațiile

O imagine de ansamblu asupra arhitecturii sistemului este prezentată în Figura 5.

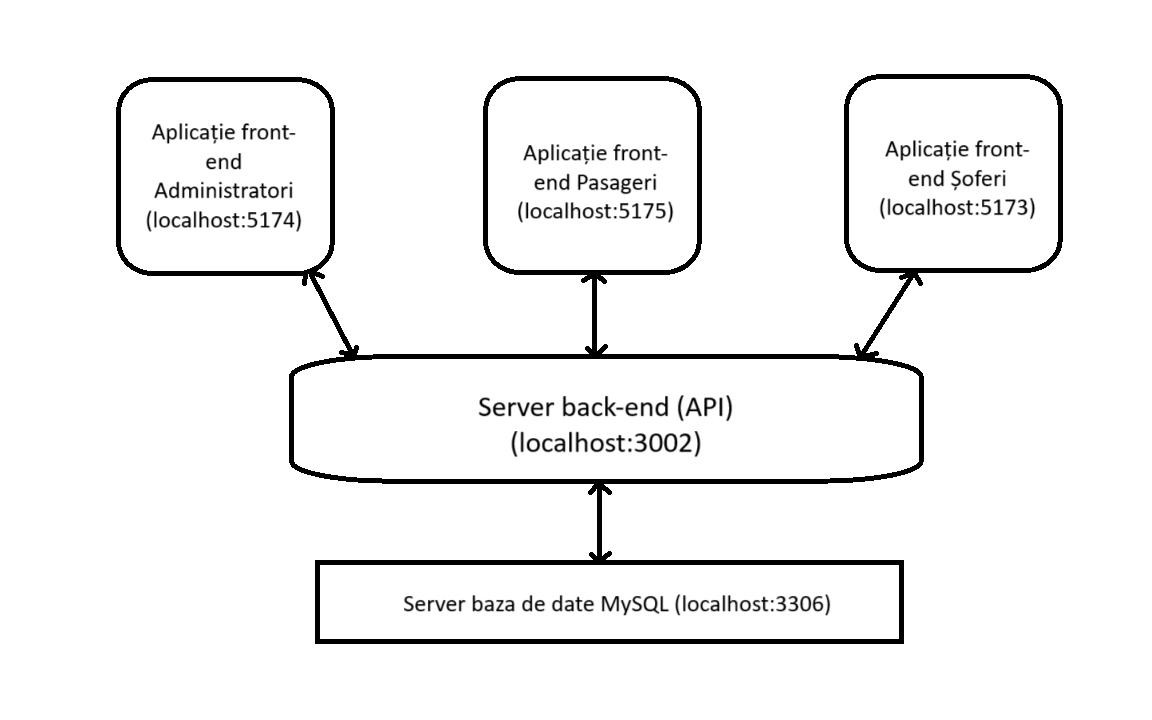


Fig. 5. Arhitectura sistemului

# Documentație tehică

## Proiectarea bazei de date

Primul pas în formarea bazei de date a fost crearea schemei „bushop” folosind comanda SQL „CREATE SCHEMA bushop”. Ulterior, am creat tabelele și constrângerile folosind *Sequelize.*

Sequelize [Sequelize] este o bibliotecă de tip ORM (Object-Relational Mapping) în JavaScript prin intermediul căreia se facilitează comunicația cu baza de date relațională, programatorul interacționând cu obiecte JavaScript în detrimentul cererilor SQL.

Pentru conectarea aplicației la baza de date, în fișierul „/config/config.json” am modificat obiectul JSON existent, specificând utilizatorul (root), parola, numele bazei de date, adresa IP la care se găsește serverul bazei de date (127.0.0.1 – localhost) și dialectul – mysql.

După instalarea dependenței sequelize în aplicația back-end (de acum o vom numi „server”), a fost creat directorul „/models” în care am definit tabelele bazei de date. Fișierele respectă convenția de nume „NumeTabela.js” și conțin următoarele porțiuni relevante de cod:

* O funcție anonimă (în terminologia JavaScript – arrow function) ce primește ca parametri obiectele sequelize și DataTypes (pentru tipurile de date ale câmpurilor). Această funcție este exportată pentru ca obiectul returnat să fie utilizat și în alte fișiere.
* În interiorul funcției, creez obiectul NumeTabelă (de exemplu, „const Trasee”) cu ajutorul metodei „define()” a obiectului sequelize, specificând câmpurile tabelei (nume, tip de date și proprietatea „allowNull”) și numele tabelei.
* O serie de asocieri sequelize.

În sequelize, asocierile au rolul de a defini constrângerile din baza de date (cum ar fi cheile primare și cheile străine). Asocierile pe care le-am folosit în proiectarea bazei de date sunt „hasMany” și „belongsTo”.

Conform [Sequelize], asocierea „A.hasMany(B)” implică o relație de tipul „one to many” între modelele (tabelele) A și B, cheia străină fiind creată în tabela B. Spre exemplu, în cadrul modelului „Curse” am definit asocierea



După cum putem observa, am specificat cheia străină „CursaID” (aceasta va fi creată în tabela „Rezervări”, fiecare rezervare având o cursă de care aparține). Mai mult, am specificat constrângerea „onDelete: CASCADE” astfel încât, atunci când este ștearsă o cursă, vor fi șterse și toate rezervările asociate cu acea cursă.

Pe de altă parte, asocierea „A.belongsTo(B)” implică o relație de tipul „one to one” între cele două tabele, cheia străină urmând a fi creată în tabela A. Un exemplu concludent pentru această asociere este prezent în modelul „Curse”:



Din perspectiva tabelei „Curse”, această asociere implică faptul că o cursă poate avea un singur traseu. Este de menționat că în multe cazuri am ales să implementez asocierile din „două sensuri”. Spre exemplu, chiar dacă am creat asocierea precedentă din perspectiva curselor, am definit și următoarea asociere, din perspectiva traseelor:



Această asociere indică faptul că un traseu poate figura în mai multe curse. Motivul pentru care, în unele cazuri, am ales să creez asocieri din ambele sensuri este dat de diferitele funcții de interogare sequelize care devin disponibile prin aceste asocieri. De altfel, aceasta este recomandarea oficială, oferită în documentația [Sequelize] pentru a crea relații de tipul „one to many” – o sursă (în cazul nostru – un traseu) este conectată cu mai multe ținte (curse), fiecare din această țintă având o singură asociere cu sursa.

Acum că am expus principalele caracteristici ale Sequelize și am analizat modalitatea de creare a tabelelor și a constrângerilor, iată o imagine de ansamblu a bazei de date. Aceasta conține următoarele tabele:

* Autocare
* id: int, autoincrementabil, cheie primară
* număr\_înmatriculare: varchar(255)
* număr\_locuri: tinyint
* marcă: varchar(255)
* createdAt: datetime
* updatedAt: datetime
* Trasee
* id: int, autoincrementabil, cheie primară
* oraș\_pornire: varchar(255)
* oraș\_1: varchar(255)
* oraș\_2: varchar(255)
* oraș\_3: varchar(255)
* oraș\_sosire: varchar(255)
* createdAt: datetime
* updatedAt: datetime
* Șoferi
* id: int, autoincrementabil, cheie primară
* nume: varchar(255)
* email: varchar(255)
* parola: varchar(255)
* createdAt: datetime
* updatedAt: datetime
* Curse
* id: int, autoincrementabil, cheie primară
* AutocarID: int, cheie străină
* TraseuID: int, cheie străină
* SoferID: int, cheie străină
* zi\_plecare: datetime
* oră\_plecare: time
* oră\_sosire: time
* status: varchar(255)
* createdAt: datetime
* updatedAt: datetime
* Rezervări
* id: int, autoincrementabil, cheie primară
* nume: varchar(255)
* CursaID: int, cheie străină
* PasagerID: int, cheie străină
* loc: tinyint
* createdAt: datetime
* updatedAt: datetime
* Pasageri
* id: int, autoincrementabil, cheie primară
* nume: varchar(255)
* email: varchar(255)
* parolă: varchar(255)
* telefon: varchar(255)
* createdAt: datetime
* updatedAt: datetime
* Coordonate
* id: int, autoincrementabil, cheie primară
* latitudine: decimal(10,8)
* longitudine: decimal(10,8)
* CursaID: int, cheie străină
* timp: datetime
* createdAt: datetime
* updatedAt: datetime

Pentru a înțelege și mai bine constrângerile și relațiile dintre tabele, diagrama ER este prezentă în figura 6. Diagrama a fost realizată în MySQL Workbench și conține tabelele și câmpurile acestora, precum și constrângerile relaționale dintre acestea.

A screenshot of a computer

Description automatically generated

Fig. 6. Diagrama entitate – relație

## Implementarea server-ului

Înainte de a începe prezentarea propriu-zisă a implementării server-ului, este util să înțelegem arhitectura server – client și protocolul HTTP (HyperText Transfer Protocol). În [HTTP Overview] sunt tratate pe larg aceste elemente cruciale ale comunicației între aplicațiile web. Prin arhitectura client – server înțelegem un ansamblu format din două sisteme, în care unul dintre ele inițiază cereri (clientul) și celălalt le rezolvă, trimițând înapoi răspunsuri (server-ul). În continuare, voi alterna modurile de referire la cereri și răspunsuri, cu terminologia specifică din domeniul dezvoltării web: „request-uri” și „response-uri”.

HTTP este protocolul fundamental prin care se realizează comunicația server-client. Poate cea mai importantă caracteristică a acestui protocol de comunicație este capacitatea de a transmite o varietate de date, pe lângă text și documente hypertext (text cu legături – links), cum ar fi imagini și videoclipuri.

Unul dintre cele mai concludente și răspândite exemple de cereri este request-ul pe care browserul (clientul) îl trimite către server pentru a afișa o pagină web. Serverul răspunde cu fișierul HTML, pe care browserul îl interpretează ținând cont de anumite restricții (script-uri și stilizări pe care le voi expune în capitolele următoare). Prin natura sa, această cerere este o cerere de tip „GET” (clientul, în acest caz browserul, vrea să primească date). Protocolul HTTP dispune de mai multe tipuri de cereri (metode), însă cele mai importante și utilizate sunt următoarele:

* GET: clientul vrea să obțină (citească) date de la server
* POST: clientul vrea să adauge „corpul” cererii în evidența server-ului, cauzând o modificare în server și introducând o nouă resursă
* PUT: clientul vrea să actualizeze o resursă existenă în server
* DELETE: clientul vrea să șteargă o resursă existentă în server

Programatorii back-end trebuie să pregătească serverul astfel încât cererile venite de la aplicațiile de tip client să fie rezolvate corespunzător. Aceștia trebuie să implementeze endpoint-uri pentru diversele tipuri de entități existente în baza de date și cu care clienții ar trebui să interacționeze, așa cum vom vedea în cele ce urmează.

Pe lângă informația utilă trimisă de către server în răspunsuri (cum ar fi resursa cerută de client printr-o cerere GET) și alte elemente, acesta este responsabil și cu trimiterea unui cod de status (status code) care indică succesul sau diverse erori înâmpinate în încercarea de a deservi cererea. Cele mai cunoscute coduri de status sunt 200 – OK (cererea a fost rezolvată), 400 – Bad Request (eroare în cererea venită de la client), 403 – Forbidden (clientul nu are dreptul de a accesa resursa solicitată), 404 – Not Found (resursa nu există sau – în cazul unui browser – URL-ul introdus este invalid) și altele.

În cele ce urmează, voi prezenta aspectele tehnice principale care au stat la baza implementării aplicației back-end. Așa cum am prezentat în subcapitolul precedent (Proiectarea bazei de date), serverul a avut ca o primă sarcină crearea modelelor (tabelelor) utilizând ORM-ul sequelize.

Fișiserul principal al server-ului este „index.js”. Primul pas a fost să încarc modulul express prin metoda „require()” și să îl instanțiez în obiectul „app”. Apoi am încărcat modulul http, prin intermediul căruia am creat obiectul server folosind metoda „createServer” (server-ul este modificat să utilizeze aplicația express creată mai devreme: „app”).

Un aspect important în dezvoltarea server-ului a fost configurarea acestuia pentru a putea parsa cererile primite în format JSON (JavaScript Object Notation), acesta reprezentând o modalitate uzuală de interschimbare a perechilor de date de tip „atribut – valoare” între aplicațiile web. Astfel, am utilizat funcția de tip intermediar (middleware) „express.json()”. Conform [Express Middleware], o funcție de tip middleware are acces atât la obiectul req (request – cerere) și res (response – răspuns), preum și la funcția next() care apelează următorul middleware. În acest fel putem modifica obiectele cerere și răspuns înainte ca acestea să ajungă propriu-zis la endpoint-ul final al serverului (în cazul acesta, modificăm cererea pentru a putea fi citită în format JSON).

Pentru a permite comunicația între server și cele trei aplicații de tip client, a fost necesară utilizarea modulului „cors” (Cross-Origin Resource Sharing). Acest modul specifică server-ului ce fel de cereri poate să deservească și originile de unde pot veni aceste cereri. În cazul meu, am permis comunicația între aplicații prin următoarea configurație cors: originile permise sunt 'http://localhost:5173', 'http://localhost:5174' și 'http://localhost:5175' (cele trei aplicații front-end) iar metodele HTTP permise sunt GET, POST, PUT, DELETE și OPTIONS (o metodă esențială în utilizarea modulului cors care verifică daca originea este permisă înainte de a trimite cererea propriu-zisă).

Un ultim pas înainte de a permite server-ului să „asculte” pe un port (să fie pregătit să primească cereri) este acela de a sincroniza baza de date cu modelele sequelize existente în directorul „/models”. Abia după ce această sincronizare are loc (tabelele să existe, iar în caz contrar, să fie create) serverul poate primi cereri pe portul definit (în cazul meu, portul 3002).

Până în acest punct, serverul este pornit, însă nu are definite rute (endpoint-uri) pentru a putea primi cereri de la clienți. În acest scop, am creat directorul „/routes” în care am creat câte un fișier separat pentru fiecare categorie de resurse cu care aplicațiile front-end vor interacționa, și anume: trasee, autocare, curse, rezervări și pasageri. În fișierul principal al server-ului am creat rutele „/trasee”, „/autocare”, „/curse”, „/rezervari” și „/pasageri”, acestea folosind fiecare router-ul definit în fișierele din directorul „/routes”. Această abordare conferă modularitate sistemului de rutare, permițându-ne să separăm logica pentru fiecare tip de resursă (în fiecare fișier din directorul „/routes” se exportă obiectul router, instanță a clasei express.Router()).

Pentru a exemplifica crearea endpoint-urilor, să considerăm următorul caz în care o rută pentru a obține toate traseele este creată direct în fișierul principal al server-ului (Index.js):



După cum putem observa, deși funcționează conform așteptărilor (un client care face un request GET către “localhost:3002/trasee” va obține lista de trasee), această abordare nu este modulară, iar rezultatul final va fi un fișier principal „index.js” foarte mare și greu de întreținut. Iată acum un exemplu prin care aceeași logică de rutare este implementată într-un fișier dedicat traseelor:

În „/routes/Trasee.js”:



În „index.js”:

**

În fișierul dedicat rutelor pentru trasee, obiectul router este exportat pentru a fi disponibil și în alte fișiere. Mai exact, este folosit în fișierul principal al server-ului: toate cererile ale căror rute vor începe cu „/trasee” vor fi gestionate de acest router.

Acum că am detaliat modul de definire al endpoint-urilor, urmează să evidențiem logica de implementare pentru diverse tipuri de cereri. Să considerăm, pentru început, ruta GET pentru trasee. În acest moment API-ul (serverul) știe că există ruta, dar nu știe ce să facă atunci când o cerere apare pe acea rută. În esență, trebuie să trimită un răspuns care să conțină obiectul cerut (în acest caz, un obiect JSON conținând toate traseele din baza de date). Astfel, fiecare endpoint are atașată o funcție anonimă JavaScript ce primește doi parametri: „req” – request și „res” – response. În acest caz, logica este simplă: am apelat obiectul „Trasee” ce reprezintă modelul sequelize definit pentru tabela trasee. Prin sequelize avem acces la mai multe metode de a obține obiectele de interes din baza de date, astfel nu mai este nevoie să scriem interogări SQL. Una din aceste metode este „findAll()” – echivalentul a „SELECT \* FROM Trasee;”. Sequelize interacționează în mod asincron cu baza de date, motiv pentru care funcția anonimă asociată endpoint-ului trebuie să fie de tip async. După ce am obținut obiectul de interes din baza de date, îl trimitem prin răspuns în format JSON:



Această logică de implementare returnează toate toate traseele din baza de date. Dar dacă vrem să filtrăm anumite rezultate după un anumit câmp? Acest câmp este de obicei inclus în rută, ca și parametru. Spre exemplu, vrem să obținem datele unui pasager care are id-ul „id” (cheie primară). Endpoint-ul este definit astfel: router.get(“/:id”, …), unde id este parametru.

Definind astfel parametrul, putem obține acces la el în interiorul funcției prin obiectul „req.params” (parametrii cererii). Odată ce am identificat id-ul, apelăm metoda sequelize „Pasageri.findByPk(id)” – căutăm acel pasager care are cheia primară id și ulterior îl trimitem în răspuns, în format JSON.

Acum că avem o imagine de ansamblu asupra definirii și implementării rutelor de tip GET, trecem mai departe, la endpoint-urile de tip POST. Dacă printr-o cerere de tip GET, aplicația client solicită să primească date, printr-o cerere de tip POST aplicația solicită adăugarea unei noi resurse (în esență, adăugarea unei înregistrări în baza de date). De cele mai multe ori, în aplicația front-end se completează un formular ale cărui date (filtrate sau nu) vor fi trimise ca request către un endpoint de tip POST. Datele utile trimise de către client spre server sunt accesate prin câmpul „body” al obiectului „req” (req.body). Ulterior, folosim metoda sequelize „create” și trimitem un răspuns către client (fie un mesaj de tip „ok”, fie chiar obiectul primit în format JSON):



Pentru a șterge o înregistrare din baza de date prin intermediul API-ului, trebuie implementat un endpoint de tip DELETE. Să ne imaginăm următorul scenariu: un administrator dorește ștergerea unui traseu, întrucât acesta nu mai este de actualitate. Acesta va trimite ca și parametru id-ul traseului pe care dorește să îl șteargă. Pentru ca serverul să îndeplinească această solicitare, trebuie implementat un endpoint de tip DELETE (cu parametru, la fel cum am prezentat mai sus). Primul pas este să obținem id-ul prin câmpul „params” al obiectului „req”. Apoi trebuie sa găsim obiectul care are acel id și să apelăm metoda sequelize „destroy()”:



Ultimul tip de endpoint pe care l-am utilizat în implementarea server-ului este PUT. Printr-o cerere HTTP de tip PUT, clientul solicită modificarea unei resurse existente deja în baza de date. Logica de implementare este asemănătoare cu cele descrise mai sus (în sensul că mai întâi obținem id-ul ca și parametru și datele utile din req.body, urmând ca mai apoi să căutăm obiectul cerut și să îl salvăm într-o constantă), doar că în acest caz folosim metoda sequelize „update()”.

Un caz aparte în implementarea rutelor a fost crearea funcționalităților de înregistrare și logare a utilizatorilor. La prima vedere, adăugarea unui nou utilizator în baza de date pare banală: la fel ca în cazurile de mai sus, obținem câmpurile de interes (nume, email, parolă, număr de telefon) din corpul cererii și ulterior apelăm metoda sequelize create(). Cu toate acestea, această metodă prezintă riscuri majore de securitate, întrucât am stoca parolele utilizatorilor în format „plain text” (text care poate fi citit și înțeles de către oricine). În acest fel, parolele ar fi compromise imediat în cazul în care un atacator obține drepturi de citire asupra tabelei în cauză. Pentru a spori securitatea, am utilizat algoritmul de hash-ing „bcrypt”.

În [Sobti, R] ese prezentată pe larg noțiunea de hash-ing în contextul criptografiei. Voi prezenta, în continuare, aspectele de interes pentru proiectul meu. Un algoritm de hash-ing este un algoritm ce primește la intrare un șir de caractere și returnează un alt șir, complet diferit, de lungime fixă, numit Hash sau Digest. Pe lângă această funcționalitate, un algoritm bun de hash-ing trebuie să respecte mai multe criterii, cum ar fi:

* fiind dat algoritmul, nu ar trebui să fim capabili să obținem un rezultat (hash) dorit;
* fiind dat hash-ul, să nu putem să găsim string-ul original din care a fost obținut hash-ul;
* o schimbare mică în string-ul de intrare ar trebui să producă o schimbare semnificativă în digest-ul rezultat.

Hash-ingul diferă de criptare prin faptul că, pornind de la un hash, nu ne putem întoarce la valoarea originală. În cazul criptării, fie ea simetrică (aceeași cheie folosită și pentru criptare, și pentru decriptare) sau asimetrică (pereche de chei folosite în criptare și decriptare) mereu există o cheie care poate decripta mesajul ascuns. Astfel, alegând să stocăm hash-urile parolelor în baza de date, protejăm utilizatorii, pentru că un atacator nu va putea afla parola, pe baza hash-ului. Totuși, există atacurile de tip „rainbow tables”, prin care atacatorii au la dispoziție un tabel ce conține parole uzuale, alături de hash-ul obținut printr-un algoritm anume. Astfel, pot parcurge hash-urile din baza de date la care au obținut acces și să caute fiecare rezultat în tabela lor. Pentru a contramanda acest tip de atac, pe lângă procesul de hash-ing, se adaugă un șir random de caractere la fiecare parolă, înainte de a se aplica algoritmul (acest șir de caractere se numește informație de tip „sare”).

Bcrypt este un algoritm de hash-ing care folosește informație de tip „sare”. În implementarea endpoint-ului POST pentru utilizatori (funcționalitatea de înregistrare) am folosit metoda „bcrypt.hash(parola, 10)”, unde „parola” este informația ce trebuie ascunsă, iar 10 reprezintă factorul de cost al algoritmului (cu cât este mai mare, cu atât va dura mai mult operația de hash-ing). Logica de aplicare a hash-ului și de salvare a informațiilor unui utilizator în baza de date este prezentată mai jos:



Întrucât hash-ingul este o operație care nu poate fi inversată (având digest-ul, ar trebui să fie imposibil să se obțină string-ul original), autentificarea se face în felul următor: se aplică același algoritm de hash-ing, cu aceeași informație de tip „sare” pe parola introdusă, acest hash urmând a fi comparat cu hash-ul existent în baza de date: dacă se potrivesc, utilizatorul este autentificat. Dar de unde știe bcrypt ce informație de tip „sare” a folosit pentru a realiza primul hash, în condițiile în care „sarea” este generată aleator și în mod unic pentru fiecare înregistrare? Această informație este stocată chiar în hash-ul final, după cum se poate observa în figura 7.

A diagram of a computer

Description automatically generated with medium confidence

Fig. 7. Formatul hash-ului în baza de date

Pentru funcționalitatea de autentificare (login), am creat un endpoint de tip POST pentru pasageri cu ruta „/login”. Pe lângă autentificare, în dezvoltarea aplicațiilor am avut nevoie și de operații de autorizare (spre exemplu, un pasager să poată face o rezervare doar dacă este autentificat), motiv pentru care am apelat la biblioteca Node.js „jsonwebtoken”, folosită pentru a genera, semna și verifica JSON Web Token-uri – metodă de transmitere a datelor în siguranță între două părți în formatul unui obiect JSON (conform [JWT]).

Un JSON Web Token este format din trei părți:

* un header, care conține informații despre algoritmul de hashing sau criptare folosit (de cele mai multe ori – SHA256) și despre token (jwt);
* o secțiune de tip „payload” care conține informațiile utile despre utilizator, necesare în procesul de autentificare și autorizare (cum ar fi id, nume, etc.);
* o semnătură care este formată din header, payload și un secret, folosită pentru a verifica faptul că token-ul nu a fost modificat din momentul în care a fost emis de către server.

În endpoint-ul de login, logica este următoarea: se verifică parola utilizatorului (comparând hash-urile prin metoda „bcrypt.compare()”) și dacă este corectă, se trece mai departe și se generează un token de acces, având ca payload numele, email-ul și id-ul utilizaotrului. Acest token este semnat cu un string secret – pentru ușurință, am folosit „importantsecret”, dar în practică trebuie folosit un secret mult mai complex, stocat în siguranță pe server. Ulterior, acest token (pe care l-am numit accessToken) este trimis în răspuns către client, alături de anumite date de interes (id, nume, email). Clientul (browser-ul) va salva acest accessToken în secțiunea „sessionStorage” și îl va trimite în header-urile cererilor pe care le va face și care vor necesita autorizare (mai multe detalii în capitolele destinate aplicațiilor front-end). Rolul acestui accessToken este de a a demnostra faptul că clientul care a făcut cererea este autentificat și că are dreptul să acceseze resursa respectivă.

În acest scop, a fost nevoie să implementez un fișier de tip middleware („/middlewares/AuthMiddleware.js”), din care am exportat o metodă numită „validateToken”. Acesta primește ca parametri obiectele req, res (la fel ca un endpoint normal) și metoda next() – metoda care semnifică trecerea la următorul middleware sau, dacă nu mai există altul, la endpoint-ul propriu-zis. Astfel, middleware-ul este o funcție intermediară, interpusă între client și server, care filtrează cererea înainte ca aceasta să ajungă la server. Părțile componente ale accessToken-ului sunt codate folosind Base64URL – deci pot fi ușor decriptate. Funcția „verify()” din biblioteca jsonwebtoken primește ca parametri obiectul accessToken (obținut din antetul cererii trimise de client) și string-ul secret, același care a fost folosit mai devreme pentru semnarea token-ului. Header-ul și payload-ul sunt decodate și semnate din nou cu ajutorul secretului, pentru a verifica autenticitatea token-ului. Dacă verificarea are loc cu succes, funcția verify returnează payload-ul decodificat (care, conform explicațiilor de mai sus, conține informațiile de interes despre utilizatorul care a făcut cererea, cum ar fi id-ul și numele), iar acest payload este trimis mai departe, fiind adăugat în corpul cererii care va ajunge la endpoint-ul final. Astfel, în etapa finală de procesare a cererii – endpoint-ul final - vom avea acces la datele utilizatorului. Toată această logică de implementare este rezumată mai jos:



Middleware-ul „validateToken” nu va fi folosit, în mod implicit, de toate endpoint-urile. Va fi folosit doar de acele endpoint-uri care îl specifică (deci, acele endpoint-uri care au nevoie să autorizeze utilizatorul). Un exemplu concludent îl constituie ruta de adăugare a unei rezervări la o cursă: nu vrem ca un pasager să poată adăuga o rezervare, dacă nu este autentificat. Astfel, am definit ruta în felul următor:



O altă funcționalitate importantă pe care am implementat-o în aplicația back-end este reprezentată de trimiterea e-mail-urilor de confirmare la efectuarea unei rezervări. Pentru această sarcină am utilizat modulul Node.js „Nodemailer” prin intermediul căruia aplicațiile web pot fi configurate să trimită email-uri în mod automat. Pentru trimiterea e-mail-urilor, am creat o adresă dedicată, folosind serviciul „Gmail”. De altfel, din motive de securitate impuse de Google, pentru a permite Nodemailer să folosească această adresă de e-mail, a fost necesar să creez o parolă dedicată pentru această aplicație, diferită de parola principală a contului de e-mail.

Primul pas a fost reprezentat de crearea unui obiect de tip „transport” în care am definit serviciul utilizat (gmail) și autentificarea folosită (adresa de e-mail și parola dedicată pentru această aplicație). Ulterior, am creat metoda „sendConfirmationEmail” cu parametrii „to” și „details” (destinatarul e-mail-ului, adică utilizatorul logat și detaliile rezervării: ambele vor fi disponibile în endpoint-ul de adăugare a rezervării). În cadrul acestei metode am creat apoi obiectul „mailOptions” în care am definit detaliile uzuale dintr-un e-mail: emițătorul, destinatarul, subiectul și text-ul e-mail-ului. În final, am apelat metoda „sendMail” a obiectului de tip „transport”, cu parametrul „mailOptions”.

În cadrul endpoint-ului pentru adăugarea unei rezervări, după crearea acesteia, am apelat metoda sendConfirmationEmail, definită anterior.

Un ultim aspect important în funcționarea server-ului este reprezentat de utilizarea „Websocket-urilor”, pentru a permite aplicațiilor front-end destinate pasagerilor și, respectiv, șoferilor, să comunice între ele în timp real, în scopul partajării locației autocarului. Websocket-urile reprezintă, la fel ca și HTTP, un protocol de comunicare între două elemente din rețea (putem păstra, pentru referință, același model client – server), însă prezintă câteva diferențe cheie, așa cum reiese din [Gupta]. Dacă HTTP este un protocol bazat pe conexiuni indedpendente (adică o nouă conexiune este deschisă între client și server la trimiterea unei cereri, iar această conexiune este închisă după primirea răspunsului), o conexiune realizată prin Websocket-uri va rămâne deschisă, clientul și serverul având la dispoziție un canal bidirecțional prin care pot să comunice, până când unul dintre ei solicită terminarea conexiunii.

Pentru implementarea comunicației în timp real am utilizat biblioteca „socket.io”, aceasta permițându-ne să realizăm canale bidirecționale de comunicare (conexiuni) între aplicații, bazate pe evenimente. Un client poate semnala o acțiune sau trimie un mesaj prin „socket.emit(‘nume\_eveniment’)”, iar server-ul trebuie să trateze aceste evenimente prin „socket.on(‘nume\_eveniment’)”. Evenimentele pe care le-am tratat, în cadrul server-ului, sunt următoarele:

* ‘join-cursa’: când un client emite acest eveniment, server-ul îl adaugă într-o cameră destinată acestei curse („socket.join(...)”);
* ‘start-cursa’: acest eveniment este emis de către șofer. Șoferul este adăugat la camera destinată cursei și toate părțile conectate primesc confirmarea inițierii cursei;
* ‘update-coordonate’: eveniment emis de șofer, clienții conectați primesc noile coordonate ale autocarului;
* ‘stop-cursa’: eveniment emis de șofer. Toți clienții vor primi finalizarea cursei, iar șoferul se va deconecta de la camera dedicată cursei.

Voi prezenta alte detalii despre comunicarea prin Websocket-uri și în capitolele dedicate aplicațiilor front-end.

## Implementarea aplicațiilor front-end

Pentru crearea aplicațiilor React am folosit utilitarul „Vite”. Acesta permite crearea de proiecte tip „șablon” pentru mai multe framework-uri populare cum ar fi Vue și React, prin comanda „npm create vite”. [Vite]

După specificarea numelui proiectului și a tehnologiei utilizate (în cazul meu, React), utilitarul creează o structură de fișiere și directoare. Dezvoltarea aplicației se realizează în directorul „src” (source). Fișierele sursă JavaScript pot avea extensiile „.js” sau „.jsx” (acestea sunt interschimbabile, însă un fișier cu extensia „.jsx” este mai potrivit pentru fișierele React care definesc componente și care definesc structuri de tip HTML în interiorul acestora). Fișierele de interes sunt:

* index.html - documentul principal care este interpretat de către browser (aici este creat un div cu id-ul „root”;
* main.jsx – încarcă componenta „App.jsx” (componenta principală React) în div-ul „root” definit în index.html;
* App.jsx – componenta principală în care am definit rutele și paginile specifice pentru fiecare rută

De altfel, pe lângă structura șablon creată de utilitarul „vite”, am ales să creez, în directorul sursă al fiecărei aplicații, directoarele „components” (pentru componente, cum ar fi header, cursă, etc.), „pages” (în esență, paginile sunt tot componente, dar le-am creat în așa fel încât să fie formate din mai multe componente și să aibă rol de pagini web) și „styles” (director în care am inclus fișierele de stilizare CSS – Cascading Style Sheets – pe care le-am folosit pentru a stiliza aplicațiile). În figura 8 am prezentat structura de directoare și fișiere pentru una din cele trei aplicații (aplicația de administrare).

A screenshot of a computer

Description automatically generated

Fig. 8. Structura de fișiere și directoare pentru o aplicație React

Pe parcursul implementării celor trei aplicații am folosit biblioteca „axios” pentru a face cereri HTTP către server în mod asincron. Dacă o pagină sau o componentă au nevoie de anumite date de la server încă de la încărcare, aceste cereri trebuiesc definite în interiorul unui bloc de tip „useEffect” – un mecanism React în cadrul căruia definim metodele ce trebuiesc executate imediat după ce o componentă a fost montată. Ulterior, avem nevoie de niște obiecte în care să salvăm datele primite în răspuns de la server. Deși, teoretic, putem folosi constante sau variabile, în practică trebuie utiliza un alt tip de obiect, și anume o „stare” React definită prin mecanismul „useState” ce conține două elemente: obiectul propriu-zis pentru care vrem să salvăm starea, precum și o metodă ce setează o nouă stare pentru acest obiect. Prin utilizarea „useState”, componenta va fi reîncărcată de fiecare dată când o stare este schimbată, deci schimbările vor apărea pe ecran în timp real, fără a fi nevoie ca utilizatorul să reîncarce pagina.

Un exemplu de utilizare a acestor mecanisme este următorul:



După cum putem observa, „useEffect” primește o funcție anonimă „() => { …}” precum și un vector (în cazul de mai sus, un vector nepopulat). Acest vector se numește vector de dependețe și are rolul de a specifica ce va acțiuni vor determina o rulare a mecanismului useEffect. În acest caz, vectorul „[ ]” semnalează faptul că metodele trebuiesc executate o singură dată, la montarea componentei. Lipsa acestui vector ar determina React să ruleze aceste metode într-o buclă infinită.

Alte două „hook-uri” React pe care le-am utilizat sunt „useParams” și „useNavigate”. Prin useParams, obținem acces la parametrii rutei la care se găsește pagina curentă. Spre exemplu, să considerăm ruta:



În componenta „AddRezervare” am folosit apoi useParams pentru a obține parametrul id din rută. În acest caz, acest lucru este util, pentru că știm id-ul cursei pentru care dorim să facem rezervare (id pe care îl putem trimite către server, împreună cu celelalte date ale rezervării).

Pentru a comuta între pagini, am folosit mecanismul „useNavigate” (acesta nu știe componenta către care se va comuta, dar știe ruta).

### Aplcația de administrare

Încă de la început am proiectat aplicația de administrare ca o interfață grafică pentru comunicarea cu serverul și cu baza de date. Aici vor fi efectuate operațiile de bază de tip CRUD (Create, Read, Update, Delete – Creare, Citire, Actualizare și Ștergere) ale componentelor aplicației, și anume: trasee, autocare, curse și rezervări (pasagerii și șoferii vor fi creați prin formulare de înregistrare, în aplicațiile dedicate).

Astfel, am creat pagini separate (fiecare pagină având o rută dedicată) pentru următoarele funcționalități: creare și afișare pentru trasee, autocare, curse, rezervări și o pagină de editare a unei curse deja create. Administratorul poate selecta pagina dorită prin intermediul unei bare de navigare (navbar). De altfel, în fiecare pagină am introdus elemente UI (User Interface) care să sugereze cât mai bine acțiuni cum ar fi adăugarea, modificarea sau ștergerea unei resurse.

Așa cum am prezentat în capitolul 3.4.3, React se bazează pe lucrul cu componente de tip șablon (funcții ce acceptă ca parametru un obiect „props”). Un exemplu concludent îl constituie autocarele, pentru care am creat o componentă implementată astfel:



Folosirea acoladelor ne permite să includem variabile și cod JavaScript în interiorul elementelor HTML ale componentelor. Declararea în acest fel a componentelor ne permite să le reutilizăm. De exemplu, pentru a genera o listă cu toate autocarele existente, trebuie să parcurgem următoarea secvență logică: 1) Apel API de tip GET pentru a obține lista de autocare => 2) Maparea listei obținute către o listă de componente (fiecare obiect autocar din lista primită de la server va fi trimis către parametrul „props” al unei componente) cu ajutorul metodei JavaScript „map()”.

Am aplicat această logică (crearea unei componente șablon și apoi maparea unei liste de obiecte către o listă de componente) pentru toate celelalte elemente de interes ale aplicației. Dacă în cele mai multe cazuri nu a fost nevoie să filtrez rezultatele căutării de elemente, un caz aparte l-a constituit căutarea unor curse după un anumit traseu și o dată calendaristică. Motivul acestei implementări este dat de faptul că nu ar fi practic pentru un administrator să vadă toate cursele existente în baza de date, ci doar pe cele de interes. Întrucât în cadrul unei curse avem acces doar la „TraseuID” și nu avem acces la alte date despre traseu, cum ar fi orașul de plecare sau cel destinație, a fost nevoie să implementez un algoritm mai elaborat:

1. Plecând de la lista completă de curse primită de la server (listaCurse), am creat o nouă listă (listaFiltrata) cu ajutorul metodei JavaScript „filter()”. Astfel, pentru fiecare cursă, am obținut obiectul traseu (din lista completă de trasee) pe baza egalității între id-ul traseului și câmpul „TraseuID” al cursei curente.
2. În același timp, pentru fiecare cursă, am obținut data plecării, pe care am formatat-o astfel încat să aibă același format cu data introdusă de administrator în formularul de căutare.
3. Am păstrat, în obiectul listaFiltrata, doar acele curse care al căror traseu și dată de plecare (dacă s-a introdus o dată) corespund cu datele introduse de utilizator.
4. În final, am mapat lista filtrată de curse către o listă de componente de tip cursă.

Un alt aspect important în dezvoltarea aplicației a fost manipularea datelor introduse de utilizatori prin intermediul unor formulare. Am utilizat biblioteca „Formik” care este concepută pentru lucrul cu formulare în React și care simplifică în mod semnificativ gestionarea acestora. Pentru fiecare formular, am creat un obiect numit „initialValues” în care am stocat valorile inițiale pentru fiecare câmp, și un obiect „validationSchema” (prin intermediul bibliotecii „yup”, cu rol în validarea câmpurilor). De exemplu, pentru formularul de adăugare a unui traseu, inițial, toate câmpurile sunt șiruri de caractere „empty”, iar obiectul „validationSchema” impune ca toate câmpurile introduse să fie șiruri de caractere, cu mențiunea că orașul de pornire și cel destinație reprezintă câmpuri obligatorii.

În plus, un formular creat cu Formik necesită și o metodă numită „onSubmit()” ce primește ca parametru obiectul „values”, acesta conținând datele câmpurilor din formular, introduse de utilizator. Această metodă definește ce trebuie să se întâmple mai departe cu acele date. De cele mai multe ori, acestea sunt trimise în corpul unei cereri, către server.

Este interesant de notat faptul că, în cazul formularului pentru adăugarea unei curse, de exemplu, am oferit utilizatorului posibilitatea să selecteze traseul, autocarul și șoferul pentru o cursă din liste deja populate cu resursele existente (liste obținute prin cereri de tip GET către server, imediat ce pagina este încărcată). Flexibilitatea oferită de Formik mi-a permis să includ, ca și text, informația utilă (cum ar fi numele șoferului), dar valoarea câmpului să fie de fapt id-ul resursei. Astfel, atunci când datele sunt trimise către server, nu mai sunt necesare prelucrări pentru a găsi id-ul.



Pentru adăugarea rezervărilor, am considerat utilă implementarea unei interfețe vizuale intuitive și ușor de folosit. În acest scop, am creat o componentă numită „Scaun” care, în esență, este un pătrat de dimensiune 50px, acesta putând avea trei clase CSS, și anume „occupied”, „selected” sau „free”, fiecare clasă setând o culoare diferită de fundal pentru scaunul respectiv. Această componentă primește parametri („props”), printre care și numărul asociat scaunului și variabile boolene care arată dacă scaunul este ocupat sau selectat (în funcție de acestea se decide și clasa CSS a componentei).

Ulterior, în pagina destinată adăugării unei rezervări, am efectuat o cerere de tip GET către server, pentru a primi lista de rezervări (numerele scaunelor) deja efectuate pentru cursa selectată. În același timp, am creat un vector ce conține numerele de la 1 la 40 (numărul de locuri al autocarelor) pe care l-am „mapat” către un vector de componente de tip „Scaun”, trimițând ca parametri numărul scaunului și starea sa (ocupat, dacă numărul scaunului curent se găsește în vectorul de locuri ocupate primit de la server sau selectat, dacă utilizatorul selectează scaunul).

În prealabil, am stilizat această listă de componente, pentru a fi afișată conform cu realitatea, pe patru coloane, cu un spațiu mai mare între coloanele din mijloc. Lângă panoul de selectare a scaunului, am adăugat un câmp text în care administratorul poate introduce numele persoanei care dorește rezervarea (acesta este motivul pentru care tabela Rezervări conține atât câmpul nume, cât și cheia străină PasagerID. Când un administrator efectuează o rezervare în această aplicație, se va popula doar câmpul nume, iar când un pasager autentificat realizează o rezervare din aplicația pentru pasageri, se vor popula atât câmpurile nume, cât și cheia străină PasagerID).

### Aplicația destinată pasagerilor

Dacă în aplicația de administrare m-am focusat pe operațiile de bază de tip creare, citire, actualizare și ștergere a resurselor din baza de date (prin intermediul server-ului), în aplicația destinată pasagerilor am încercat să creez o interfață cât mai modernă și ușor de folosit, prin care aceștia să poată efectua următoarele acțiuni:

* Creare cont și autentificare
* Căutarea curselor după traseu și data plecării
* Efectuarea rezervărilor pentru curse (cu confirmare pe e-mail)
* Vizualizarea rezervărilor active în pagina de profil
* Urmărirea autocarului pe hartă, în timp real
* Trimiterea de sugestii prin intermediul unui formular de contact

Pentru căutarea curselor și efectuarea rezervărilor, am utilizat aceiași algoritmi ca și în cazul aplicației de administrare (cu mențiunea că, de această dată, la efectuarea unei rezervări, se va popula și câmpul PasagerID din tabela rezervări).

Pentru a ține evidența utilizatorului care este logat în toate componentele și paginile aplicației, am utilizat mecanismul react „createContext”. Astfel, în componenta „App.jsx” (componenta principală a aplicației), am impus ca toate rutele aplicației să aibă acces la un obiect de tip „useState”: authState și setAuthState. Obiectul authState conține informații despre pasagerul care este autentificat (nume, id, email și status). În momentul în care un utilizator se autentifică, se face o cerere POST către endpoint-ul din server responsabil cu logarea, iar în răspuns se primește de la server un „accessToken” (conform explicațiilor din capitolul 4.2), precum și celelalte date de interes despre utilizatorul care s-a logat. Cu aceste date se populează obiectul authState, iar accessToken-ul este salvat în secțiunea „sessionStorage” a browser-ului. Mai departe, de fiecare dată când utilizatorul va face o cerere către server care necesită autorizare (cum ar fi efectuarea unei rezervări), va include acest accessToken în antetul cererii HTTP:



Cererea va fi interceptată de către middleware-ul „AuthMiddleware” pe care l-am prezentat în implementarea server-ului. Acesta va verifica accessToken-ul primit de la client și, dacă este valid, va trimite cererea mai departe către ruta finală de adăugare a rezervării. Dacă utilizatorul nu ar fi autentificat, middleware-ul din server nu va găsi niciun accessToken în antetul cererii și va respinge cererea.

Pentru delogare, am implementat ștergerea accessToken-ului din sessionStorage-ul browser-ului și setarea variabilei authState la valorile implicite.

În pagina de profil a utilizatorilor am creat două secțiuni: o secțiune în care afișez date despre de interes despre utilizatorul autentificat, prin intermediul obiectului authState (nume, email și număr de telefon) și o secțiune în care am introdus rezervările active ale utilizatorului (rezervările pentru cursele care au status-ul „neinițiată” sau „în desfășurare). Lista de rezervări am obținut-o printr-o cerere de tip GET către server, unde am implementat un endpoint care returnează rezervările unui utilizator după id-ul acestuia.

Pentru a economisi spațiu în interfața vizuală, am ales să afișez rezervările folosind biblioteca „React Slick”, prin intermediul căreia un „slider”, afișând, astfel, câte o rezervare pe rând.

Pentru urmărirea curselor, am creat o pagină dedicată, care poate fi accesată prin click pe o cursă al cărei status este „în desfășurare”. Am utilizat biblioteca „socket.io-client” pentru a gestiona comunicația prin Websocket-uri din perspectiva clientului. De altfel, am creat un obiect useState în care salvez statusul cursei, iar în funcțite de acest obiect – stare afișez mesajele corespunzătoare în interfață.

Când clientul accesează pagina de urmărire a cursei, trimite mesajul „join-cursa”, împreună cu id-ul cursei, către server, pentru a se alătura camerei dedicate cursei respective. Ulterior, clientul așteaptă actualizări de la aplicația șoferului care a inițiat cursa. Spre exemplu, când clientul primește evenimentul „new-coordonate”, setează noile coordonate primite într-o variabilă de tip useState, ceea ce provoacă imediat o actualizare a hărții:



Pentru afișarea hărții am utilizat biblioteca „React-Leaflet” prin intermediul căreia am creat un obiect MapContainer ce afișează o hartă centrată în punctul primit în timp real de la șofer:



După ce componenta este demontată (pagina de urmărire a cursei este închisă), clientul se dezabonează de la evenimentele pe care le-a primit de la șofer prin Websocket-uri.

O altă funcționalitate pe care am inclus-o în pagina principală a aplicației este aceea de a afișa cele mai populare trei curse la momentul accesării platformei. Astfel, am efectuat o cerere GET către server, pentru a obține o listă ce conține toate rezervările iar apoi, pentru fiecare cursă, o cerere pentru a obține numărul de rezervări aferente acelei curse. Apoi am ordonat vectorul descrescător, în funcție de numărul de rezervări și am păstrat primele 3 obiecte, pe care le-am mapat către niște componente în interfața vizuală.

# Bibliografie

1. //API. Wikipedia <https://en.wikipedia.org/wiki/API> Accesat: 2024
2. Autogari.ro <https://www.autogari.ro/> Accesat: 2024
3. Bursa.ro. <https://www.bursa.ro/studiu-flixbus-pasagerii-romani-isi-doresc-autocare-moderne-25141731> Accesat: 2024
4. Busradar <https://www.busradar.ro/autocar/tara/romania/> Accesat: 2024
5. Aplicația Dacos <https://www.dacos.com.ro/> Accesat: iunie 2024
6. DuBois, P. (2013). *MySQL*. Addison-Wesley. Part III-10. Introduction to MySQL administration
7. Flixbus <https://www.flixbus.ro/> Accesat: 2024
8. Framework software <https://en.wikipedia.org/wiki/Software_framework> Accesat: 2024
9. Insomnia. Documentatie <https://docs.insomnia.rest/insomnia/get-started> Accesat: 2024
10. Ionescu <https://ionescu.autogari.ro/> Accesat: 2024
11. Masse, M. (2011). *REST API design rulebook: designing consistent RESTful web service interfaces*. " O'Reilly Media, Inc.".
12. MySQL Workbench <https://www.mysql.com/products/workbench/> Accesat: 2024
13. //MySQL. Wikipedia <https://en.wikipedia.org/wiki/MySQL> Accesat: 2024
14. Visual Studio Code <https://code.visualstudio.com/docs> Accesat: 2024
15. What is Node.js? <https://www.freecodecamp.org/news/what-is-node-js/> Accesat: 2024
16. //DOM. Wikipedia <https://en.wikipedia.org/wiki/Document_Object_Model> Accesat: 2024
17. Ullman, L. (2012). *Modern JavaScript: Develop and Design*. Peachpit Press. p 4-8
18. Sequelize. <https://sequelize.org/> Accesat: 2024
19. Express Middleware. <https://expressjs.com/en/guide/writing-middleware.html> Accesat: 2024
20. HTTP Overview. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview> Accesat: 2024
21. Sobti, R., & Geetha, G. (2012). Cryptographic hash functions: a review. *International Journal of Computer Science Issues (IJCSI)*, *9*(2), p 461-462.
22. JWT. Introducere. <https://jwt.io/introduction> Accesat: iunie 2024
23. Vite. <https://vitejs.dev/guide/> Accesat: iunie 2024
24. Gupta, B., & Vani, M. P. (2018). An overview of web sockets: The future of real-time communication. *Int. Res. J. Eng. Technol. IRJET*, *5*(12), p 434-435.