

**Universitatea  
Transilvania  
din Brașov**

**FACULTATEA DE INGINERIE ELECTRICĂ  
ȘI ȘTIINȚA CALCULATOARELOR**

# PROIECT DE DIPLOMĂ

**Conducător științific:**

**Prof. KRISTÁLY Dominic Mircea**

**Absolvent:**

**Nițu Andreea**

**BRAȘOV, 2022**

Departamentul Tehnologia Informației  
Programul de studii: Licență

*NIȚU Andreea*

# EventiFind. Aplicație web pentru managementul evenimentelor

**Conducător științific:**  
Prof. *KRISTÁLY Dominic Mircea*

# Cuprins

## Cuprins

Lista de figuri, tabele și coduri sursă .....	5
Lista de acronime .....	7
1 Introducere .....	8
1.1 TEMA DE PROIECTARE .....	8
1.2 STADIUL ACTUAL AL PROBLEMEI ABORDATE .....	8
1.3 SCOPUL SI OBIECTIVELE PROIECTULUI .....	9
1.4 STRUCTURA PE CAPITOLE .....	10
2 Considerații Tehnice .....	10
2.1 TEHNOLOGII .....	10
2.1.1 Aplicații Web vs Aplicații native .....	10
2.1.2 Tehnologii pentru Aplicațiile Web .....	12
2.1.3 Ruby on Rails .....	14
2.2 ALGORITMI .....	16
3 Detalii de Realizare a Temei .....	17
3.1 PROIECTARE .....	17
3.1.1 Arhitectura .....	17
3.1.2 Structura Bazei de Date .....	20
3.2 DEZVOLTARE .....	22
3.2.1 Instrumente de dezvoltare folosite .....	22
3.2.2 Instalarea mediului de execuție .....	24
3.2.3 Metode dezvoltate .....	25
3.2.4 Moștenire .....	27
3.2.5 Devise și Action Mailer .....	30
3.2.6 Diagrame de secvență .....	32
3.2.7 Tipul de date Enum .....	36
3.2.8 Comunicarea dintre controller și view. Exemplu .....	38
3.2.9 Frontend .....	39
3.3 IMPLEMENTARE .....	40
3.3.1 Rolurile utilizatorului .....	40
3.3.2 Backoffice .....	41
3.3.3 Stocarea imaginilor .....	43
3.3.4 Aspectul de rețea de socializare .....	44
3.3.5 Public id .....	45
3.3.6 Validări .....	46
3.3.7 Testare .....	47

4	Concluzii.....	48
4.1	CONCLUZII GENERALE.....	48
4.2	CONTRIBUȚII PERSONALE.....	49
4.3	POSIBILITĂȚI DE DEZVOLTARE ULTERIOARĂ .....	50
	Bibliografie .....	51
	Rezumat .....	52
	Abstract.....	53

## LISTA DE FIGURI, TABELE ȘI CODURI SURSĂ

### FIGURI

Figură 1 - Sisteme de operare suportate de aplicația AnyDesk

Figură 2 - Interogare în pgweb

Figură 3 - Rezultatul interogării

Figură 4 - Schema bazei de date

Figură 5 - Exemplu de generare a unui model

Figură 6 - Relația many-to-many

Figură 7 - Logo Linux

Figură 8 - Logo Postgresql

Figură 9 - Logo Ruby on Rails

Figură 10 - Stivă de limbaj Ruby

Figură 11 - Prietenii care participă la un eveniment

Figură 12 - Email de confirmare a contului

Figură 13 - Procesul de înregistrare a unui utilizator

Figură 14 - Proces pentru adăugarea unui eveniment nou

Figură 15 - Proces pentru accesarea chat-ului

Figură 16 - Procesul de accesare a backoffice-ului

Figură 17 - Meniu de navigație pentru Backoffice

Figură 18- Exemplu de intrări din lista Utilizatorilor

Figură 19 - Relația dintre Users și Followers

Figură 20 - Url pentru evenimentul cu id 1

Figură 21 - Url pentru un eveniment cu public\_id

### TABELE

Tabelul 1 - Rute Event

### CODURI SURSĂ

Codul 1 - Exemplu de embedded ruby

Codul 2 - Metoda DESTROY din controller-ul de evenimente

Codul 3 - Afișarea mesajelor folosind Turbo Rails

Codul 4 - Implementarea metodei get\_all\_following\_participants

Codul 5 - Metoda initAutocomplete() ce inițializează autocomplete-ul din Google Maps

## API

Codul 6 - Metoda isFollowing(other\_user)

Codul 7 - Metoda create\_room din event.rb

Codul 8 - Exemplu de moștenire în JavaScript

Codul 9 - Clasa Message

Codul 10 - Definiția tabelului messages din schema.rb

Codul 11 - Cod ce stochează toate participările la un anumit eveniment

Codul 12 - Definirea tabelului Participations

Codul 13 - Definirea modulului Participatable

Codul 14 - Afișarea link-ului către profil

Codul 15 - Comenzi pentru pornirea serverului

Codul 16 - Exemplu de afișare pentru o iconiță

Codul 17 - Enum pentru categorii

Codul 18 - Variabila @following\_events stochează lista de evenimente

Codul 19 - Căutare folosind WHERE...LIKE

Codul 20 - Comandă pentru afișarea barei de navigație

Codul 21 - Adăugarea unei reguli într-un policy

Codul 22 - Mediul local din fișierul storage.yml

Codul 23 - Mediul amazon din fișierul storage.yml

Codul 24 - Cod pentru asocierea unei imagini cu un model

Codul 25 - Metoda add\_public\_id din modelul event

Codul 26 - Validarea ce asigură unicitatea username-ului

Codul 27 - Test pentru executarea metodei show din events\_controller

Codul 28 - Exemplu de înregistrare pentru un eveniment din fixtures

## LISTA DE ACRONIME

API – Application Programming Interface

DOM – Document Object Model

IESC – Inginerie Electrică și Știința Calculatoarelor;

MVC – Model View Controller

OOP – Object Oriented Programming

WSL – Windows Subsystem for Linux

# 1 INTRODUCERE

---

Tema de proiectare

Stadiul actual al problemei abordate

Scopul și Obiectivele proiectului

Domeniul de aplicabilitate

Structura pe capitole

---

## 1.1 TEMA DE PROIECTARE

Din cauza pandemiei de SARS-COV-2, din ce în ce mai mulți oameni au fost nevoiți să se izoleze în case. Această situație neașteptată și nefericită a afectat viețile majorității, în special felul în care oamenii interacționează și călătoresc.

Odată cu relaxarea restricțiilor, oamenii au început să revină treptat la vechile obiceiuri de socializare. Pentru a facilita această revenire și pentru a o face cât mai plăcută și ușoară, *EventiFind* pune la dispoziție o platformă web intuitivă și deschisă tuturor pentru organizarea și participarea la diverse evenimente.

Pentru că o să existe întodeauna o reticență și timiditate atunci când este vorba de a cunoaște oameni noi, platforma pune la dispoziție și un serviciu de mesagerie instantă cu ajutorul căruia utilizatorii se pot cunoaște mai bine online, înainte de a se întâlni.

## 1.2 STADIUL ACTUAL AL PROBLEMEI ABORDATE

Analizând soluțiile existente pe piața din România am ajuns la concluzia că cel mai apropiat serviciu de management al evenimentelor este *Facebook Events*. Diferența majoră dintre cele două servicii este că *EventiFind* a fost gândit pentru a promova apropierea dintre oameni într-un spațiu sigur, care oferă o mai mare intimitate.

Spre exemplu, dacă o persoană merge într-o plimbare prin Parcul Noua din Brașov și vrea un partener de conversație, aceasta poate adăuga un eveniment pe *EventiFind* urmând ca



persoanele interesate să i se alăture. De asemenea, aceasta poate adăuga un itinerariu care trece prin mai multe puncte de interes pe care intenționează să le viziteze, și, astfel, participanții se pot alătura pe tot parcursul drumului.

Prin comparație, evenimentele de pe *Facebook* sunt evenimente mari, unele dintre ele comerciale, și mai puțin personale. Utilizatorii nu se așteaptă să vadă adunări mici de oameni în lista lor de evenimente.

### 1.3 SCOPUL SI OBIECTIVELE PROIECTULUI

Primul obiectiv este acela de a sprijini utilizatorii să creeze evenimente prin dezvoltarea unei aplicații cât mai intuitive și ușor de folosit. Pentru ca cineva să creeze un eveniment sau un itinerariu trebuie doar să se înregistreze în platformă și să acceseze meniul de lângă poza sa din bara de navigație. Obiectul creat va apărea pe pagina principală și poate fi văzut de toată lumea imediat.

De asemenea, faptul că un utilizator este adăugat automat și instant în camera de chat asociată evenimentului respectiv, chat care poate fi accesat cu același cont ca și pe site-ul principal, face ca platforma să fie foarte intuitivă.

Pentru a vedea toate postările viitoare ale unui anumit utilizator, este nevoie doar să se acceseze butonul de *Follow* de pe profilul acestuia.

Al doilea obiectiv pe care l-am urmărit în dezvoltarea aplicației a fost crearea unui mediu sigur și filtrat de conținutul care ar putea afecta alți utilizatori. Libertatea de exprimare trebuie deci limitată pentru a favoriza siguranța fiecărui utilizator.

În acest fel, moderarea este făcută de administratori care au capacitatea de a înlătura de pe platformă postările și persoanele care nu corespund unor anumite politici.

Principalul scop al proiectului este acela de a îndemna oamenii să fie mai sociabili. Un studiu al Universității Harvard [1] a demonstrat că unul din cei mai importanți factori care influențează longevitatea și satisfacția unei persoane este numărul de relații apropiate și gradul de conexiune pe care acestea îl au. Acestea fiind spuse, proiectul *EventiFind* își dorește să aducă o schimbare pozitivă pe acest plan prin crearea unui context, un motiv pentru ca oamenii să poată interacționa și crea legături apropiate cu alte persoane din afara cercului lor de cunoștințe.

## 1.4 STRUCTURA PE CAPITOLE

Primul capitol este unul de introducere, în care am argumentat necesitatea și importanța temei de proiect în strânsă legătură cu unele comparații între modalitățile deja existente în rezolvarea temei. În acest capitol am prezentat și scopul și obiectivele lucrării precum și structurarea pe capitole.

Capitolul al doilea sintetizează aspectele teoretice legate de tematică, la care se va face referință în capitolul trei pentru rezolvarea temei de proiectare. Sinteza prezintă aspectele teoretice absolut necesare pentru fundamentarea capitolului următor. Informațiile prezentate au fost filtrate astfel încât să se păstreze esența și mesajul lor, dar să nu fie prezentate mult prea în profunzime pentru a deruta cititorii.

Următorul capitol conține părți originale, de proiectare și dezvoltare ale proiectului. Aici sunt prezentate toate aspectele implementate pentru rezolvarea temei de proiectare cum ar fi partea de realizare practică și alegerea soluțiilor cele mai convenabile ținând cont de aspectele teoretice prezentate în capitolul doi.

Ultimul capitol este de concluzii referitoare la modul în care am îndeplinit obiectivele, rezultatele obținute, utilitatea rezolvării temei, contribuțiile personale și direcții viitoare de cercetare/dezvoltare.

## 2 CONSIDERAȚII TEHNICE

---

**Tehnologii**

**Algoritmi**

---

### 2.1 TEHNOLOGII

#### 2.1.1 Aplicații Web vs Aplicații native

Din ce în ce mai multe aplicații noi create sunt destinate mediului online. Această platformă conferă un principal avantaj imens în comparație cu celelalte: **portabilitatea**. În cazul unei aplicații native, pentru ca segmentul de piață să se extindă la utilizatorii de pe toate platformele, trebuie dezvoltate aplicații native pentru toate sistemele de operare.

Problema apare în momentul în care începe analiza acestor sisteme de operare. Să luăm spre exemplu aplicația *AnyDesk*. Această aplicație facilitează controlul Desktop-ului de la distanță și, conform website-ului oficial, au fost dezvoltate nu mai puțin de opt versiuni diferite, una pentru fiecare sistem de operare suportat.



Figură 1 - Sisteme de operare suportate de aplicația AnyDesk [2]

Există o mulțime de diferențe între aceste sisteme de operare, lucru care face dezvoltarea aplicațiilor pe multiple platforme aproape imposibilă. Linux (Android, Raspberry Pi, Chrome OS) și BSD (macOS, FreeBSD, iOS) se aseamănă prin faptul că provin din sistemul de operare UNIX, pe când Windows are propriul *kernel*. În continuare, sistemele de operare pentru dispozitive mobile (Android și iOS – și în unele cazuri Chrome OS și macOS) se folosesc de arhitectura ARM64 pentru instrucțiunile procesorului pe când sistemele de Desktop se bazează pe arhitectura Intel x86.

În afara acestor diferențe de arhitecturi, se află diferențele legate de limbajele și framework-urile folosite pentru dezvoltarea aplicațiilor native. Spre exemplu, Android folosește Java pe când iOS folosește Swift, Linux folosește C/C++ pe când Windows folosește C# și .NET.

Desigur, există, cel puțin pentru dispozitive mobile, soluții de dezvoltare a aplicațiilor native folosind același *code base* precum Flutter sau React Native. Acestea însă lasă de dorit în privința performanțelor și în privința aspectului relativ la restul sistemului de operare.

În concluzie, cu cât se dorește dezvoltarea mai multor iterații ale aceleiași aplicații pe mai multe platforme, cu atât costul producției va crește. Din acest considerent multe companii preferă dezvoltarea aplicațiilor web. Portabilitatea este asigurată atât timp cât dispozitivul are un browser și o conexiune la Internet.

Un prim avantaj al aplicațiilor web este faptul că acestea pot fi accesate de oriunde. Un exemplu relevant este faptul că dacă editezi un document pe un calculator de la birou și ai vrea să îl recitești acasă pentru a face anumite modificări va trebui să mergi fizic pentru a îl copia și a-l deschide cu un procesor de text pe calculatorul personal. Folosind o aplicație precum Google Docs, un document poate fi creat și editat de pe orice dispozitiv, fie el mobil sau Desktop, și indiferent de locație. [3]

În al doilea rând, mentenanța pentru mai multe aplicații este complicată deoarece, dacă apare un bug în logica aplicației, trebuie modificate toate aplicațiile. Timpul pentru reparație și timpul alocat distribuirii actualizării pe magazinele de aplicații aferente sistemelor de operare este

exponențial mai mare în comparație cu aplicația web care rezolvă această problemă o singură dată iar actualizarea se face fără să depindă de alți terți.

De asemenea, costul realizării a unei aplicații web cu *frontend* și *backend* este mic relativ la cel pentru dezvoltarea aplicațiilor aferente sistemelor de operare. Codul este scris o singură dată de o singură echipă de programatori, nefiind nevoie de specialiști pentru fiecare limbaj de programare separat sau fiecare arhitectură.

Pe de altă parte, principalul punct slab al aplicațiilor web îl reprezintă în sine faptul că sunt online și pentru a fi accesate este nevoie în permanență de acces la Internet. Cu toate acestea, există aplicații precum Google Docs care pun la dispoziție documentele offline pentru browser-ul Google Chrome, nefiind nevoie de acces la Internet pentru a le edita ci doar pentru a încărca modificările în cloud.

Nici din punctul de vedere al securității, aplicațiile web nu stau mai bine față de cele native deoarece există o multitudine de atacuri ce le vizează. De asemenea, în cazul atacurilor asupra aplicațiilor native, vulnerabilitatea lor depinde foarte mult de sistemul de operare pe care rulează. Spre exemplu, este cunoscut faptul că Windows este un sistem mai puțin sigur decât Linux sau macOS, la fel cum Android este mai puțin sigur decât iOS. Pentru un atac complet asupra unei aplicații, atacatorii ar trebui să dezvolte un plan de atac asupra tuturor iterațiilor acesteia spre deosebire de unul singur pentru web.

### 2.1.2 Tehnologii pentru Aplicațiile Web

Un avantaj pe care nu l-am menționat în subcapitolul anterior este faptul că, în comparație cu aplicațiile native, există foarte multe opțiuni de tehnologii folosite pentru a dezvolta o aplicație web.

Aplicațiile web moderne sunt împărțite în *backend* (partea care se ocupă cu manipularea datelor din baza de date) și *frontend* (partea care se ocupă cu afișarea datelor), iar dezvoltarea fiecăreia se poate face folosind diverse limbaje cu framework-urile asociate. Înainte de apariția limbajelor specifice folosirii pe web, paginile erau scrise exclusiv folosind HTML și stilizate folosind CSS. În prezent, până și site-urile statice necesită elemente dinamice pentru interacțiunea cu utilizatorul.

Limbajul de programare folosit pentru dezvoltarea frontend-ului este, fără îndoială, JavaScript. De la JavaScript simplu folosit împreună cu HTML și CSS până la framework-uri complexe precum Angular, JavaScript aproape are monopol pe ce înseamnă frontend. Sunt foarte puține website-uri care fie nu folosesc JavaScript deloc sau folosesc alternative precum Python.

Acest imens impact pe care JavaScript îl are asupra dezvoltării de aplicații web se datorează, în principal, faptului că poate manipula documente de HTML folosind DOM (Document Object Model) și asta face ca adăugarea de elemente dinamice cum ar fi încărcarea de date în pagină sau acțiuni declanșate de anumite evenimente să fie tratate foarte simplu.

Avantajele respective au fost duse și mai departe datorită framework-urilor dezvoltate pe baza limbajului JavaScript. Cel mai folosit dintre acestea, în prezent, este React JS, care, la bază, este de fapt doar o bibliotecă pentru manipularea DOM-ului, însă este la fel de puternic ca un framework precum Angular. De asemenea, au fost dezvoltate și tehnologii de backend ce au acest limbaj la bază cum ar fi Node JS și TypeScript.

De cealaltă parte, opțiunile pentru backend sunt mult mai diversificate și variate din punct de vedere al limbajului. În principiu, toate framework-urile de backend rezolvă aceeași problemă și fac același lucru, iar alegerea unuia nu reprezintă decât o simplă preferință. În trecut, singurul limbaj de backend folosit era PHP, care a introdus pentru prima dată date din backend într-o pagină de HTML.

Diferența dintre framework-urile de backend o reprezintă, în mare, vitezele de rulare și vitezele de dezvoltare. Nu există un framework care să ruleze perfect și foarte repede și în care se poate crea o aplicație în mai puțin de o zi, însă există framework-uri care să se axeze pe una din avantaje și să stea destul de bine la cealaltă. Un exemplu de framework care le face pe amândouă relativ bine este .NET cu limbajul C#. Limbajul se aseamănă foarte mult ca sintaxă și performanțe cu Java și o aplicație web relativ complexă se poate dezvolta destul de repede folosind opțiunile de *scaffoldings* din Visual Studio. Totodată, Visual Studio nu este disponibil pe toate sistemele de operare ceea ce poate reprezenta o problemă.

Pentru dezvoltarea proiectului, am ales framework-ul Ruby on Rails ce folosește limbajul Ruby. Ruby, fiind un limbaj de nivel înalt și fiind scris folosind C, seamănător Python, are rezultate mai slabe la capitolul performanță, însă la capitolele viteză de dezvoltare și securitate este printre cele mai bune limbaje. Ruby on Rails se aseamănă foarte mult framework-ului Django (ca și structură a fișierelor, comenzi, etc.), la fel cum Ruby împarte similarități cu limbajul Python.

Un alt aspect important legat de framework-ul ales este faptul că Ruby on Rails este *batteries included*, însemnând că atât frontend-ul cât și backend-ul pot face parte din aceeași aplicație. Acest lucru ajută foarte mult partea de securitate deoarece, cele două tehnologii diferite pentru backend și frontend vor trebui să comunice între ele prin *API* folosind informații sub formă de *json*. În acest fel se pot intercepta date mai ușor și, de asemenea, performanțele aplicației pot fi reduse semnificativ.

### 2.1.3 Ruby on Rails

Înainte de a intra în detalii legate de framework, voi prezenta, pe scurt, limbajul Ruby și de ce îl consider ca fiind un limbaj prietenos dar foarte puternic în același timp.

Lansat inițial în 1995, Ruby este un limbaj de programare interpretat, de nivel înalt și folosit pentru dezvoltarea de aplicații într-o varietate largă de tipuri de software [4]. Acesta a fost influențat de alte limbaje de programare precum Perl și Eiffel și se poate folosi procedural (ex. pentru script-uri), orientat obiect sau pentru programare funcțională [5].

Ruby tratează toate tipurile de date ca obiecte, inclusiv primitivele. Acest lucru este foarte rar în comparație cu restul limbajelor. Spre exemplu, numărul *8*, în Ruby, are proprietatea *times* care aplică o acțiune numărului respectiv, acțiunea repetându-se de 8 ori în cazul nostru.

Blocurile de cod sunt definite folosind structura *do...end* sau pur și simplu se încheie cu *end*, comparativ cu acoladele sau tab-urile din Python, aspect ce face codul mai ușor de urmărit.

Variabilele din Ruby nu trebuie declarate explicit iar acestea folosesc o anumită convenție a numelui pentru a defini scopul variabilei: *<nume\_variabilă>* (variabila locală), *@<nume\_variabilă>* (variabilă de instanță), *@@<nume\_variabilă>* (variabilă statică pentru clasă), *\$<nume\_variabilă>* (variabilă globală). [6]

Principalul motiv pentru care am ales framework-ul Ruby on Rails este faptul că dezvoltarea este foarte rapidă și foarte sigură în același timp. Deși nu pare la prima vedere datorită multiplelor cazuri în care totul se întâmplă "ca prin magie", Ruby on Rails este foarte ușor de modificat și de modelat în orice mod ai avea nevoie.

Pentru a combina informațiile din backend cu cele din frontend am folosit fișierele specifice framework-ului și anume *Embedded Ruby* (fișiere cu extensia *.html.erb*). Acestea permit introducerea elementelor de Ruby în HTML la fel de natural ca și folosind JavaScript.

Un alt motiv pentru care am ales acest framework este datorită faptului că este *open source* și portabil, însemnând că, indiferent de sistemul de operare pe care îl voi alege pentru dezvoltare, aplicația va funcționa fără programe ajutătoare sau configurări ulterioare. Spre exemplu, am dezvoltat aplicația în WSL (Windows Subsystem for Linux) însă am testat aplicația și pe Windows în cadrul aceluiasi dispozitiv.

Rails folosește tehnica de dezvoltare *code first* ce se referă la faptul că baza de date este generată în urma scrierii codului. Pentru aceasta sunt create migrări care, odată rulate, definesc tabelele cu toate proprietățile acestora.

Pentru a putea lega informațiile din baza de date de cod și frontend, Ruby on Rails folosește modelele din MVC cu ajutorul tehnologiei *Active Record*, specifică framework-ului.

Această tehnologie încapsulează proprietățile din tabele în clase specifice OOP. Aceleași clase rețin și legăturile dintre tabele (sau în acest caz obiecte) folosind funcțiile *callback* (ex. *after\_create :create\_room*) [7].

O altă tehnologie de afișare a datelor din backend în frontend pe care am folosit-o este *Turbo Rails* care oferă viteza unei pagini de tip *SPA* (Single Page Application) fără să fie nevoie de un framework de JavaScript menționat anterior în acest capitol [8].

*Turbo Streams* permite afișarea în timp real a modificărilor (creare, editare, ștergere) datelor din baza de date și face ca aplicația să fie foarte dinamică și intuitivă. Am folosit tehnologia respectivă în aplicația de mesagerie instantă deoarece aveam nevoie ca un mesaj să fie afișat de îndată ce o persoană îl trimite.

Sistemul de baze de date ales este *Postgresql*. Am ales acest sistem deoarece doream să folosesc o bază de date relațională pentru motive de scalabilitate și ușurință în navigare. Scalabilitatea este oferită de faptul că se pot defini propriile tipuri de date și propriile funcții înăuntrul unei baze de date de tip *Postgresql* [9]. De asemenea, *Postgresql* rulează pe majoritatea sistemelor de operare.

În proiectul meu am întâlnit toate tipurile de relații între tabele și, datorită comunicării fluide dintre Ruby on Rails și *Postgresql*, soluțiile au fost implementate rapid și eficient.

Relația *one-to-one* este folosită pentru a crea o legătură în care o singură înregistrare dintr-un tabel este asociată unei singure înregistrări din altul. De exemplu, un utilizator poate avea o singură poză de profil, iar o poză de profil poate aparține unui singur utilizator.

Relația *one-to-many* sau *many-to-one* este folosită pentru a exprima legătura de tip o înregistrare dintr-un tabel este asociată mai multor înregistrări din alt tabel. În proiectul meu, un utilizator poate avea asociate mai multe mesaje pe chat, dar un mesaj de pe chat nu poate aparține mai multor utilizatori.

Fiind o aplicație complexă cu multe asocieri de entități, am întâmpinat, în mai multe rânduri, relația de tip *many-to-many*. Abordarea mea pentru a implementa această relație a fost să creez o tabelă asociativă pentru a stoca toate relațiile dintre cele două tabele. Concret, un itinerariu poate avea mai multe evenimente în componența sa dar și un anumit eveniment poate face parte, la rândul său, din mai multe itinerarii [10]. Astfel, am creat tabela asociativă *event\_schedules* în care stochez asocierile dintre cele două tabele.

Deoarece am folosit framework-ul Ruby on Rails atât pentru partea de frontend cât și pentru partea de backend, am ales să mă ghidez după arhitectura *MVC*. Acest *design pattern* este alcătuit din 3 componente principale: *Model*, *View*, *Controller*.

**Modelul** este considerat cel mai de bază nivel și este responsabil pentru stocarea și menținerea datelor. Acesta reprezintă legătura directă dintre aplicație și baza de date și orice se întâmplă cu datele (adăugare, ștergere, modificare) se întâmplă în model.

Un **View** reprezintă vizual datele stocate de model. Acesta este, în fapt, cod de HTML ce afișează datele în pagină pentru utilizator. Componenta se folosește de Embedded Ruby pentru a interpola codul de Ruby în cel de HTML.

**Controller-ul** este partea care leagă cele două componente și acționează ca un intermediar. Acesta îi spune modelului ce să facă cu datele și îi spune view-ului ce să afișeze în pagină.

Avantajele acestei arhitecturi sunt faptul că este relativ ușor de menținut datorită structurii sale foarte riguroase, componentele sale sunt refolosibile pe parcursul aplicației, ajută la testarea mai ușoară a componentelor și ține la distanță logica de business de interfața cu utilizatorul [11].

## 2.2 ALGORITMI

În dezvoltarea unei aplicații de orice tip ce salvează date sensibile, cum ar fi parolele, este recomandat ca acestea să nu fie criptate și în niciun caz să nu fie stocate în clar. Pentru ca utilizatorul să își păstreze intimitatea asupra parolelor sale, și pentru ca acestea să se poată totuși verifica, cea mai bună metodă este aceea de a aplica o funcție de hash asupra parolei.

Algoritmul de hash pe care l-am ales și pe care îl folosește și gem-ul *devise* este *bcrypt* iar această metodă de securizare se obține în momentul în care modelul *User* are proprietatea *has\_secure\_password* [12]. În baza de date, parola este salvată sub numele de *password\_digest* și reprezintă hash-ul parolei unui utilizator.

Deoarece până și această metodă de securizare are o vulnerabilitate, și anume atacurile de tip *Rainbow Tables* ce trec o listă de posibile parole prin același algoritm și compară hash-urile rezultate cu cele din baza de date a aplicației atacate, *bcrypt* propune folosirea unui *salt*. Această proprietate reprezintă o valoare pseudo-aleatoare care este adăugată parolei utilizatorului înainte de a fi hash-uită [13].

Ca o metodă în plus de a securiza aplicația, pentru fiecare eveniment și itinerariu, am generat un ID unic numit *public\_id*. Acest id este folosit pentru a putea selecta un eveniment și pentru a nu oferi potențialilor atacatori vreo informație relevantă despre poziția evenimentului în baza de date (folosesc `/events/public_id` în loc de `/events/id`).



Imediat după ce se crează un nou eveniment sau itinerariu, folosind metoda *uuid* din cadrul interfeței *SecureRandom* [14], generez un Universally Unique Identifier versiunea a 4-a [17]. Acest tip de identificator nu folosește nicio informație despre sistem făcându-l complet aleator. Conform Wikipedia [15], pentru o șansă de 50% ca două *uuid-uri* să coincidă, trebuie generate 2.71 chintilioane de astfel de identificatoare.

## 3 DETALII DE REALIZARE A TEMEI

---

Proiectare

Dezvoltare

Implementare

---

### 3.1 PROIECTARE

#### 3.1.1 Arhitectura

Structura proiectului este alcătuită, în mare, din evenimente, itinerarii și utilizatori. Oricine poate să își creeze un cont de utilizator, orice utilizator poate să creeze evenimente sau itinerarii și să participe la evenimente create de alți utilizatori.

Totodată, aplicația de chat asociată acestei platforme include camere și mesaje, camerele fiind în relație *one-to-one* cu evenimentele/itinerariile și cu utilizatorii întrucât chat-ul privat cu fiecare utilizator reprezintă o cameră în sine. Mesajele sunt asociate fiecărei camere și sunt în relație *one-to-many* cu un utilizator.

Ruby on Rails este un framework *batteries included* însemnând că acesta, pe lângă controllere și clase, poate avea view-uri ce folosesc html și fragmente de cod de Ruby (și se numesc Embedded Ruby).

Arhitectura folosită se numește *MVC* (Model-View-Controller).

*Model* din numele arhitecturii reprezintă clasa obiectului cu toate proprietățile acestuia din baza de date. Altfel spus, modelul este componenta ce are legătură directă cu tabelul asociat din baza de date [11].

*View-urile* sunt componentele care reprezintă datele sub formă de html. Un *view* comunică doar cu *controller-ul* cu ajutorul căruia poate lua informații din model pentru a construi pagina. Folosind butoane sau alte metode de redirectionare a utilizatorului, *view-urile* pot trimite utilizatorul către o anumită metodă a *controller-ului* folosind o rută.

*Controller-ul* este puntea de legătură dintre *Model* și *View*. Prin rute, sunt apelate metodele *controller-ului* (fiecare rută este asociată unei metode dintr-un *controller*) care salvează informații din modele în variabile globale care sunt folosite în *view-uri*.

Fiecare model are asociat prin intermediul propriului controller următoarele operații de bază: adăugare, editare, ștergere, citire (CRUD). Acestea se folosesc pentru fiecare înregistrare de tipul Eveniment și pentru citire, adăugare, editare există view-uri proprii, la care sunt redirecționați utilizatorii când trimit request-ul respectiv.

Am utilizat această arhitectură în proiect pentru a lega backend-ul de frontend fără a folosi call-uri de API.

```
<div class="action d-flex justify-content-between mt-4 align-items-center">
  <small>
    <p>
      Categorie:
      <i class="<%= @event.icon %>"></i>
      <%= @event.category %>
    </p>
  </small>
</div>
```

*Codul 1 - Exemplu de embedded ruby*

Codul 1 conține o secvență de cod din view-ul de afișare a proprietăților asociat unui eveniment. Acest view afișează proprietățile modelului (clasei) *Event* și răspunde la ruta `/events/<public_id>`, unde `<public_id>` este un id public asociat unei înregistrări din tabelul de evenimente. Secvența de cod folosește Embedded Ruby pentru a afișa categoria evenimentului curent și iconița sa.

Abordarea *code-first* se referă la faptul că baza de date este generată în urma implementării claselor în cod. Deoarece framework-ul Ruby on Rails folosește în mod implicit abordarea *code-first*, modelul (clasa) *Event* cu toate proprietățile și relațiile acesteia descrie structura tabelului din baza de date.

Pentru a înțelege acest concept mai în profunzime propun următorul exemplu din proiect : în controller-ul de evenimente, printre altele, se află definită metoda *destroy* care șterge evenimentul înregistrarea curentă.

```
def destroy
  @event.destroy
  respond_to do |format|
    format.html { redirect_to events_url, notice: "Evenimentul a fost șters cu succes." }
    format.json { head :no_content }
  end
end
```

*Codul 2 - Metoda DESTROY din controller-ul de evenimente*

Pentru a vedea mai bine ce se întâmplă în baza de date, vom face o interogare în aplicația pgweb, aplicație pentru management al bazelor de date de tip Postgres.

Rezultatul interogării este format dintr-un tabel cu înregistrările de tip event (evenimentele).

```
1 select title, location, public_id from events;
```

Figură 2 - Interogare în pgweb

title	
Plimbare pe munte	Tâmpa, Braşov, România
Vizita Muzeul Taranului Roman	Muzeul Național al Țăranului Român

Figură 3 - Rezultatul interogării

Atunci când am creat entitatea *Event*, folosind comanda *rails scaffold* s-au generat automat modelul, controller-ul, view-urile și rutele pentru aceasta.

În tabelul următor am adăugat unele din rutele care au fost create pentru modelul *Event*, acestea fiind extrase folosind comanda *rails routes*.

Tabelul 1 - Rute Event

Prefix	Verb	URI Pattern	Controller#Action
events	GET	/events(.:format)	events#index
new_event	GET	/events/new(.:format)	events#new
edit_event	GET	/events/:id/edit(.:format)	events#edit
event	DELETE	/events/:id(.:format)	events#destroy

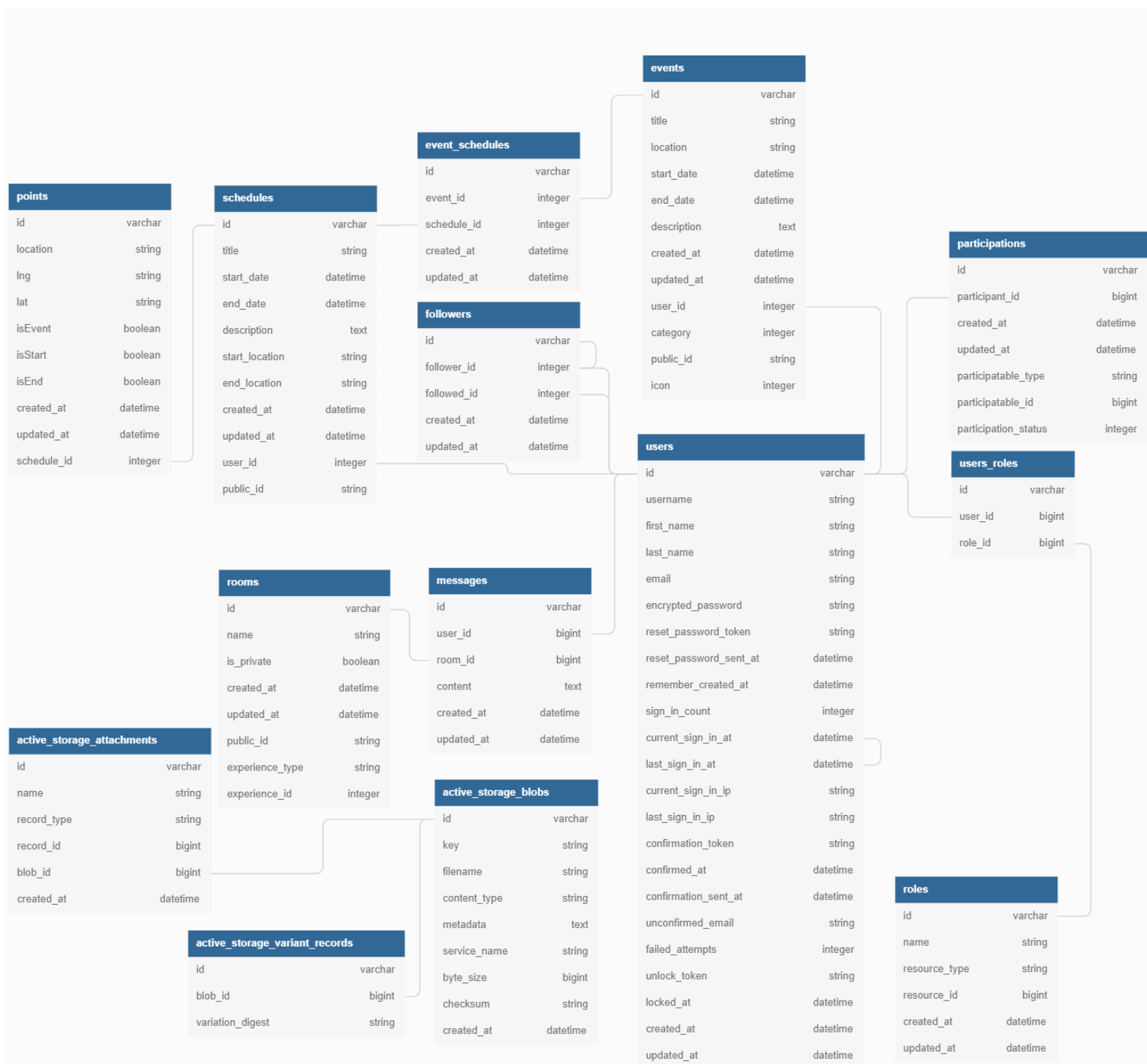
Atunci când utilizatorul apasă pe butonul de *Ștergere* de pe pagina unui eveniment creat de acesta, utilizatorul este redirecționat către o rută (un link) care este interpretată de controller. În cazul nostru, ruta este */events/<public\_id>* iar verbul este *DELETE*. Astfel, controller-ul alege metoda *destroy* (ultima din tabel).

Fiecare fișier din *views* are asociată o metodă din *controller*. Partea de final a unei metode din *controller* se ocupă cu randarea *view-ului* asociat acesteia. În același timp, metoda *destroy* nu are un *view* asociat, din motive evidente.

Deoarece Ruby are elemente de programare funcțională, metodele dintr-un *controller* pot fi clasificate ca funcții pure sau funcții cu *side effects*. Funcțiile pure nu modifică datele și, în cazul nostru, returnează un *view* iar funcțiile cu *side effects* au efect asupra datelor.

Metoda *destroy* este un exemplu de funcție cu *side effects* deoarece șterge o înregistrare și nu returnează niciun *view*.

### 3.1.2 Structura Bazei de Date



Figură 4 - Schema bazei de date

Baza de date a proiectului a fost realizată în PostgreSQL cu ajutorul comenzilor din Ruby on Rails pentru generarea modelelor.

```
andre@DESKTOP-TSFKVIR:~$ rails generate model Event title:string description:text start_date:datetime
```

Figură 5 - Exemplu de generare a unui model

În urma rulării acestor comenzi sunt create migrări (fișiere cu instrucțiuni pentru baza de date) și, odată rulate aceste migrări, este creată baza de date. Aceste fișiere sunt specifice abordării *code-first*, abordare care s-a dovedit utilă datorită faptului că toate legăturile complexe și necesare dintre tabele au fost realizate automat și acest lucru conferă o notă de securitate.

De asemenea, baza de date putând fi generată din cod, proiectul devine mai ușor de portat pe orice alt dispozitiv, eliminând probleme de depanare specifice fiecărui computer.

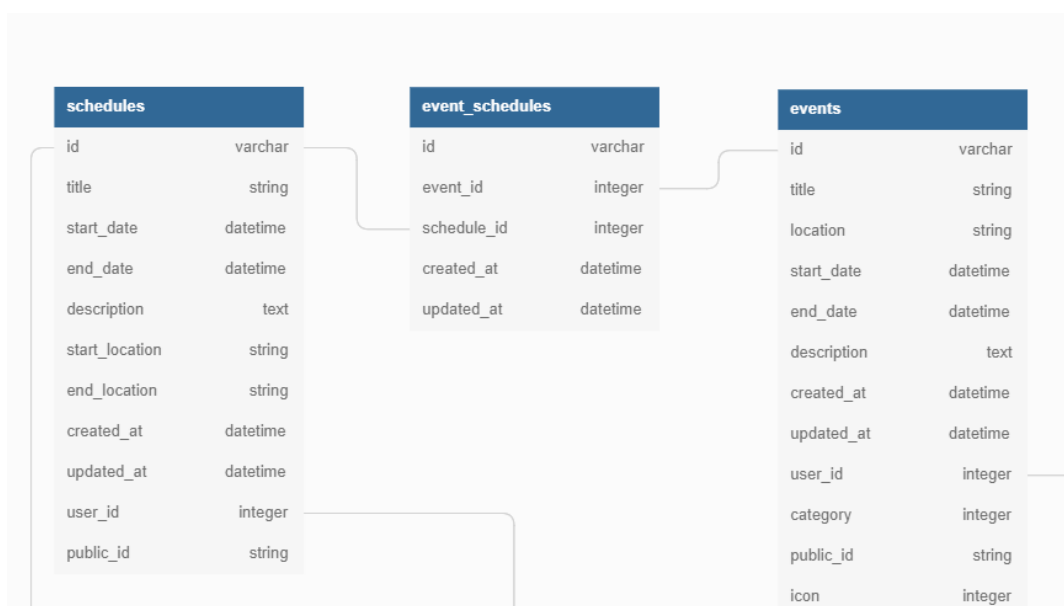
Pentru comparație, în cazul unei structuri ale unei baze de date MySQL folosită împreună cu PHP, pentru a fi portată pe alt calculator, este necesară scrierea manuală a codului de SQL pentru crearea tabelor și apoi rulate rând pe rând pe celălalt dispozitiv.

În cazul aplicațiilor dezvoltate în Ruby on Rails, tot ce trebuie făcut este rularea comenzii *rails db:migrate* care creează tabelele din baza de date conform migrărilor.

Conform descrierii din capitolul anterior, *Active Record* este tehnologia ce face legătura dintre clase și tabelele din baza de date.

În construcția bazei de date am întâmpinat situații din care au reieșit relații *many-to-many*. Spre exemplu, relația dintre itinerarii și evenimente. Un itinerariu este o colecție de evenimente care pot fi interpretate ca și puncte intermediare pe un traseu.

Un itinerariu poate avea mai multe evenimente și un eveniment poate face parte din mai multe itinerarii. Acest tip de relație a fost realizat folosind tabela asociativă *event\_schedules* ce conține id-ul evenimentului și id-ul itinerariului.



Figură 6 - Relația many-to-many

Presupunem că itinerariul cu id-ul 1 conține evenimentele cu id-urile 2, 3, 7. În același timp, itinerariul 2 conține evenimentele cu id-urile 3 și 7.

Se poate observa astfel că un itinerariu poate să conțină mai multe evenimente și un eveniment poate să aparțină mai multor itinerarii. Acest lucru este soluționat prin crearea unei înregistrări unice pentru fiecare legătură dintre evenimente și itinerarii.

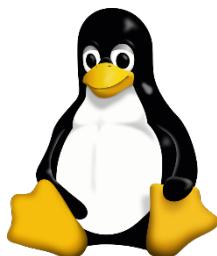
O situație reală în care poate fi întâlnit acest exemplu este în cazul în care lumea creează un itinerariu ce conține un eveniment popular, precum un concert, un joc sportiv, etc.

## 3.2 DEZVOLTARE

### 3.2.1 Instrumente de dezvoltare folosite

Pentru scrierea codului am folosit Visual Studio Code datorită multiplelor extensii pentru limbajul Ruby și pentru posibilitatea deschiderii terminalelor în interiorul programului.

Pentru a rula majoritatea comenzilor specifice framework-ului Ruby on Rails, se folosește o linie de comandă. Inițial, am început dezvoltarea folosind Command Prompt și Powershell (Windows), dar din lipsa documentației și a performanței scăzute, am ales să folosesc Linux. Din același considerent al performanței, nu am ales o mașină virtuală, ci am folosit Windows Subsystem for Linux (WSL) din interiorul sistemului de operare Windows.



Figură 7 - Logo  
Linux

Ca și tehnologie pentru bazele de date am folosit PostgreSQL datorită integrării foarte fluide cu Ruby on Rails și datorită faptului că preferam o bază de date relațională. Integrarea fluidă este realizată cu ajutorul *gem-ului pg* ce transpune clasele din aplicație în tabelele din baza de date folosind *Active Record*. Am ales bazele de date relaționale deoarece sunt mai rapide, pot stoca mai multe informații și sunt mai ușor de modificat.



Figură 8 - Logo PostgreSQL

Pentru vizualizarea bazei de date și executarea de interogări în limbajul SQL, am folosit aplicația *pgweb* și m-am conectat la baza de date creată folosind username-ul *postgres* și parola *root*.

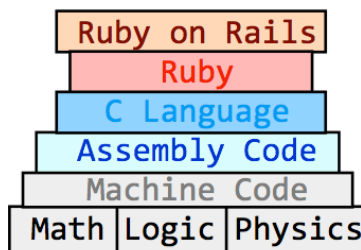
Am ales Ruby on Rails datorită faptului că este ușor de rulat pe orice sistem de operare, este ușor scalabil și are foarte multe instrumente incluse care facilitează dezvoltarea, cum ar fi generarea *scaffold-urilor* (pentru un anumit model se generează și un controller cu metode CRUD și view-urile asociate - html), generarea și rularea de teste, crearea automată a bazei de date, etc.



Figură 9 - Logo Ruby on Rails

Ca și limbaj de programare, Ruby este foarte asemănător limbajului Python, ambele fiind limbaje de nivel înalt ce facilitează dezvoltarea rapidă și sigură a aplicațiilor.

Limbajele de nivel înalt se caracterizează printr-un grad ridicat de abstractizare și elemente de limbaj natural (ex. *Date.today.beginning\_of\_week* -> returnează data de început a săptămânii curente) care simplifică procesul de dezvoltare și înțelegerea codului.



Figură 10 - Stivă de limbaj Ruby

Asemănător pachetelor ce se instalează folosind aplicația *pip* în Python, Ruby are *gem-uri*, librării sau extensii ajutătoare ce pot fi instalate folosind *gem install <nume\_gem>*. În proiect, am folosit numeroase *gem-uri* printre care:

- Devise: folosit pentru generarea tabelor, view-urilor și controller-ului pentru utilizatori. Este o metodă avantajoasă deoarece autentificarea creată este sigură, scalabilă și ușor de folosit.
- Turbo Rails: proiectul conține și o aplicație de mesagerie instantă ce necesită actualizare în timp real a mesajelor trimise și primite. Datorită turbo-rails, mesajele scrise sunt afișate instant în pagină.

```
<%= turbo_stream_from @single_room %>
<div id="messages">
  <%= render @messages %>
</div>
```

*Codul 3 - Afișarea mesajelor folosind Turbo Rails*

- Pundit și Rolify: sunt două gem-uri ce se ocupă cu autorizarea utilizatorilor și crearea rolurilor pentru aceștia. Aplicația dispune de 2 roluri (admin și utilizator) care au fost folosite pentru a filtra accesul în diferite zone cum ar fi pagina de management a utilizatorilor.
- Pg: este un gem ce leagă o bază de date de tip Postgres de Active Record din aplicația de Ruby on Rails. Modelele și tabelele sunt interconectate și ușurează utilizarea lor în restul proiectului.
- Aws-sdk-s3: este un gem ce asigură legătura cu mediul de stocare din cloud Amazon AWS S3. Acest mediu de stocare este folosit în aplicație pentru a încărca pozele de profil a utilizatorilor.

### 3.2.2 Instalarea mediului de execuție

Pentru ca aplicația să poată fi rulată local, ea are nevoie de câteva dependențe. În primul rând, este nevoie de *Ruby* versiunea 3.0.2 care poate fi instalată gratuit sau cu ajutorul comenzii *rbenv*. Apoi, folosind comanda *gem*, trebuie instalate gem-urile *rails* și *bundler*.

De asemenea, este nevoie să fie instalat sistemul de baze de date relaționale PostgreSQL care poate fi descărcat gratuit de pe Internet.

Pentru aplicația de chat este nevoie de serviciul Redis Server care, de asemenea este disponibil gratuit și care ajută la afișarea informațiilor și schimbărilor acestora în timp real.

Înainte de a porni serverul de *rails* trebuie rulate anumite comenzi de configurare a aplicației. Acestea sunt (pentru sistemul de operare Linux):



```

redis-server          # pentru pornirea serverului de redis pentru aplicația de chat
sudo service postgresql start # pentru pornirea serviciului de postgresql
bundle install        # pentru instalarea tuturor gem-urilor necesare proiectului
npm install           # sau yarn install pentru instalarea dependențelor de
                      # javascript
rails db:drop         # în cazul în care baza de date există deja și se dorește
                      # ștergerea ei
rails db:create       # pentru crearea bazei de date cu numele proiectului
                      # (se va crea o bază de date pentru development și una pentru
                      # test)
rails db:migrate      # pentru rularea migrărilor și astfel definirea tabelelor și
                      # legăturilor dintre tabele
rails db:seed         # pentru rularea fișierului db/seed.rb ce va conține date
                      # pentru popularea inițială a tabelor
rails s -p 3000       # pentru pornirea efectivă a serverului de rails (flag-ul -p
                      # reprezintă portul pe care va rula aplicația)

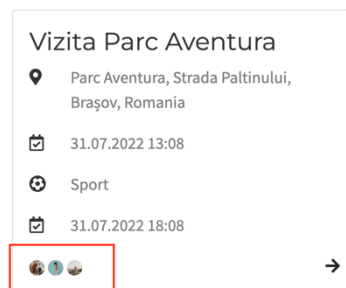
```

*Codul 4 - Comenzi pentru pornirea serverului*

### 3.2.3 Metode dezvoltate

În proiect am folosit principiile *OOP*(Object Oriented Programming) pentru a defini clasele ce iau informații stocate în baza de date și metode ce operează cu informațiile respective.

Deoarece utilizatorii ar dori să participe la evenimente la care se duc și prietenii lor, în lista de evenimente/itinerarii am adăugat o funcționalitate care îi arată utilizatorului iconițe cu pozele de profil a maxim trei dintre persoanele pe care le urmăresc și care participă la evenimentul/itinerariul respectiv.



*Figură 11 - Prietenii care participă la un eveniment*

Pentru a realiza această funcționalitate, am creat o funcție numită *get\_all\_following\_participants(<event/schedule>)* care intersectează o listă cu utilizatorii ce participă la evenimentul/itinerariul respectiv și lista cu prietenii (persoanele pe care le urmărește) utilizatorului. Această intersecție returnează primii trei prieteni (dacă există).

```

def get_all_following_participants(event)
  following = current_user.following

  following_users = []

  # stocam persoanele urmarite in following_users
  following.each do |person|
    following_users << person.followed
  end

  participations = event.participations

  users = []

  # stocam toate persoanele care participa
  # la evenimentul curent
  participations.each do |participation|

    user = User.find(participation.participant_id)
    users << user

  end

  # intersectam cele doua array-uri
  @intersection = following_users & users

  # retine doar primele 3 persoane
  @intersection = @intersection.first(3)
end

```

*Codul 5 - Implementarea metodei get\_all\_following\_participants*

O altă metodă dezvoltată este *initAutocomplete()* care sugerează locații utilizatorului pe măsură ce acesta completează câmpul de locație din formularele de adăugare a evenimentelor și itinerariilor. Aceasta folosește un serviciu oferit de către API-ul de la Google Maps. Funcția este de tip *async* deoarece răspunsul poate întârzia și astfel se evită o eroare.

```

async function initAutocomplete() {
  // autocompleteaza adresa pentru input-ul de locatie
  var input = document.getElementById('event_location');
  var autocomplete = new google.maps.places.Autocomplete(input);
  autocomplete.setComponentRestrictions({'country': ['ro']});
  autocomplete.setFields(['address_components', 'geometry', 'icon', 'name']);
}

```

*Codul 6 - Metoda initAutocomplete() ce inițializează autocomplete-ul din Google Maps API*

O funcționalitate a aplicației este aceea că utilizatorii se pot urmări între ei. Aceasta ajută la o mai bună conectare.

Această funcționalitate este posibilă datorită mai multor funcții printre care și *is\_following* aparținând clasei *User* care verifică dacă utilizatorul curent îl urmărește pe utilizatorul pe care îl primește ca parametru (*other\_user*).

```

def isFollowing(other_user)
  # stocam toti utilizatorii pe care ii urmareste
  # utilizatorul curent
  all_following = following

  # daca other_user se afla in lista de utilizatori
  # urmariti
  all_following.each do |follower|
    if follower.followed_id == other_user.id
      return true
    end
  end

  return false
end

```

*Codul 7 - Metoda isFollowing(other\_user)*

În momentul în care un utilizator creează un eveniment sau un itinerariu nou, metoda *create\_room()* definită atât în modelul *Event* cât și în modelul *Schedule* adaugă automat o cameră pentru chat în care utilizatorii care participă la evenimentul respectiv intră automat.

```

def create_room
  room = Room.new
  room.name = self.title
  room.is_private = false
  room.experience_id = self.id
  room.experience_type = "Event"
  room.save
end

```

*Codul 8 - Metoda create\_room din event.rb*

### 3.2.4 Moștenire

Clasele din framework-ul Ruby on Rails funcționează, în principiu, la fel ca și clasele din limbajul Ruby sau alte limbaje de programare.

Una din diferențele semnificative este moștenirea. Spre exemplu, în JavaScript clasa *Model* moștenește proprietatea *carname* de la *Car* folosind metoda *super* în constructor și adaugă proprietatea *model*.

```

class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

let myCar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML = myCar.show();

```

*Codul 9 - Exemplu de moștenire în JavaScript [16]*

Deoarece modelele din Rails sunt asociate unor tabele din baza de date (folosind Active Record), moștenirea nu se obține așa de ușor în cazul acestora. Pentru a înțelege mai bine diferențele vom exemplifica folosind o situație reală din cod. Clasa *Message* este folosită pentru a stoca mesajele din chat. Codul clasei *Message* arată în felul următor:

```

class Message < ApplicationRecord
  belongs_to :user
  belongs_to :room
end

```

*Codul 10 - Clasa Message*

Proprietățile clasei *Message* sunt stocate separat, în baza de date.

```

create_table "messages", force: :cascade do |t|
  t.bigint "user_id", null: false
  t.bigint "room_id", null: false
  t.text "content"
  t.datetime "created_at", precision: 6, null: false
  t.datetime "updated_at", precision: 6, null: false
  t.index ["room_id"], name: "index_messages_on_room_id"
  t.index ["user_id"], name: "index_messages_on_user_id"
end

```

*Codul 11 - Definiția tabelului messages din schema.rb*

Soluția simplă pentru a trece peste aceste diferențe o reprezintă implementarea celor două modele cu toate proprietățile lor, chiar dacă acestea se repetă. Moștenirea dorește descrierea unei

abstractizări din ce în ce mai complexe dar folosind această soluție, eliminăm conceptul de moștenire cu totul.

Pentru a implementa totuși o modalitate de a defini moșteniri în Ruby on Rails există *asocierea polimorfică*. Aceasta constă în crearea unei noi clase care poate să conțină unul sau mai multe atribute comune. Clasele cărora vrem să le oferim aceste atribute vor primi un atribut referință către clasa nou creată prin care vor putea accesa noile proprietăți.

Spre exemplu, să presupunem că avem următoarele tipuri de obiecte: *Sala\_Cinema* și *Pista\_Bowling*. Pentru a crea o abstractizare a acestora din care să putem extrage informații comune precum prețul de intrare, moștenirea ar putea propune o clasă părinte, *Spațiu\_Divertisment* ce conține proprietatea specificată. Pe de altă parte, asocierea polimorfică propune o nouă clasă de tipul `<poate_fi_făcut_de_ambele_clase_copii>`. În exemplul nostru am putea avea o clasă nouă numită *Închiriere*, iar cele două clase să aibă o proprietate numită *Se\_Poate\_Închiria* care să stocheze tipul de obiect (*Sala\_Cinema* sau *Pista\_Bowling*).

Clasa *Închiriere* va avea în componența sa, pe lângă noile atribute comune dorite, id-ul înregistrării din clasa *copii* (ex. *Sala\_Cinema* cu id-ul 1) și tipul clasei (*Sala\_Cinema* sau *Pista\_Bowling*). Mai departe, pentru a obține costul unei săli de cinema putem să îi interogăm toate atributele de tip *Închiriere*. În același timp putem vedea dacă o *Închiriere* este de tip *Pista\_Bowling* sau *Sala\_Cinema* interogând atributul *Se\_Poate\_Închiria*.

```
sala_cinema = Sala_Cinema.first
cost = sala_cinema.inchiriere.pret

inchiriere = Inchiriere.first
inchiriere.se_poate_inchiria # va returna un obiect de tip Sala_Cinema sau Pista_Bowling
```

*Codul 12 - Cod ce stochează toate participările la un anumit eveniment*

În proiectul meu, am folosit *asocierile polimorfice* pentru a crea funcționalitatea de participare la evenimente sau itinerarii. Utilizatorii pot, prin apăsarea unui buton, să participe sau să urmărească atât evenimentele cât și itinerariile. Prin urmare am avut nevoie de atributele comune *participant\_id* și de *partipation\_status* pentru ambele modele.

```

create_table "participations", force: :cascade do |t|
  t.bigint "participant_id", null: false
  t.datetime "created_at", precision: 6, null: false
  t.datetime "updated_at", precision: 6, null: false
  t.string "participatable_type", null: false
  t.bigint "participatable_id", null: false
  t.integer "participation_status", default: 0
  t.index ["participant_id"], name: "index_participations_on_participant_id"
  t.index ["participatable_type", "participatable_id"], name:
"index_participations_on_participatable"
end

```

*Codul 13 - Definirea tabelului Participations*

În clasele *Event* și *Schedule* am inclus modulul *Participatable* ce face legătura cu tabelul *Participations*.

```

module Participatable
  extend ActiveSupport::Concern

  included do
    has_many :participations, :as => :participatable
  end
end

```

*Codul 14 - Definirea modulului Participatable*

Deoarece între evenimente/itinerarii și utilizatori ar fi apărut o relație de tip *many-to-many* (un utilizator poate participa la mai multe evenimente/itinerarii iar la un eveniment/itinerariu pot participa mai mulți utilizatori), tabelul asocierii polimorfice *Participations* are rol și de tabelă asociativă.

### 3.2.5 Devise și Action Mailer

Desigur, o platformă de tip *social media* utilizatorul este punctul în jurul căruia gravitează toată aplicația. Pentru a realiza toate funcționalitățile asociate unui utilizator m-am ajutat de *devise* și *action mailer*.

*Devise* oferă o soluție sigură, scalabilă și completă de autentificare pentru aplicațiile web. Prin rularea câtorva comenzi și utilizarea unor proprietăți, aplicația dispune de autentificare. Pentru a nu încărca codul, *devise* generează doar modelul de *user*, migrarea pentru crearea tabelelor în baza de date și adaugă rutele necesare. Desigur, restul de obiecte (view-urile și controller-ele asociate) pot fi generate pentru a fi modificate după placul dezvoltatorului.

Pentru simplitate, modelul conține inițial doar proprietăți de bază precum: *username*, *first\_name*, *last\_name*, *email* și *encrypted\_password*, cea din urmă fiind obținută prin aplicarea

unei funcții de hash asupra parolei introduse de utilizator, funcție extrasă din modulul *bcrypt*. Desigur, se pot adăuga și alte proprietăți pentru a debloca funcționalități precum resetarea parolei, confirmarea emailului, blocarea contului sau urmărirea acțiunilor utilizatorului.

*Devise* dispune, de asemenea, de multiple metode ajutătoare pentru integrarea în paginile aplicației a diverselor acțiuni ce necesită autentificare. Două dintre cele mai folosite astfel de metode sunt: *user\_signed\_in?*, *current\_user*. Prima dintre acestea returnează un boolean ce este *true* în cazul în care utilizatorul este conectat și *false* altfel. *Current\_user* este o variabilă de tip sesiune (se menține de-a lungul sesiunii în care utilizatorul este conectat) ce stochează informațiile din baza de date aferente utilizatorului conectat.

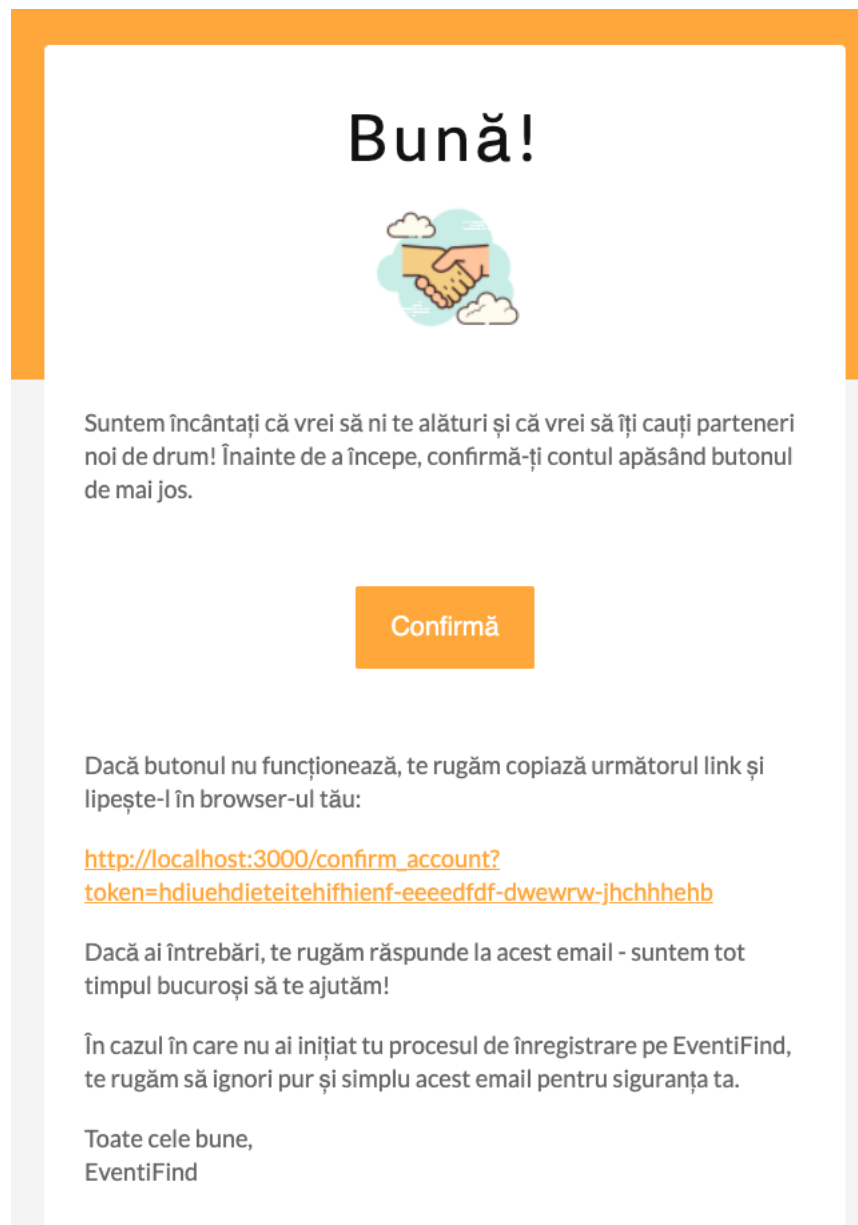
```
<% if user_signed_in? %>
  <%= link_to profile_path(current_user.username), class: "text-dark text-decoration-none" do %>
    Profil
    <i class="fa-solid fa-arrow-right"></i>
  <% end %>
<% end %>
```

*Codul 15 - Afișarea link-ului către profil*

În momentul în care un utilizator completează formularul pentru înregistrarea unui cont nou și trece mai departe, acestuia, folosind email-ul completat în formular și *Action Mailer*, îi este trimis un email de confirmare a contului. În email sunt prezente câteva informații referitoare la procesul de activare a contului, cum ar fi faptul că trebuie apăsat butonul de confirmare și, în cazul în care acesta nu funcționează, trebuie copiat link-ul de sub el și inserat într-un browser.

De asemenea, în email este prezent un mesaj de prevenție și evitare a atacului de tip *impersonare* în care atacatorul se folosește de informațiile unei alte persoane pentru a obține diferite avantaje. Tot procesul de confirmare a contului a fost gândit tocmai pentru acest considerent.

Email-ul este configurat și trimis folosind *action mailer* din Ruby on Rails și un cont de google mail. Configurarea este specifică serviciului google smtp folosind credențialele contului de google. Stilizarea email-ului pe care îl primește utilizatorul este realizată folosind html, css și framework-ul *Bootstrap*.



Figură 12 - Email de confirmare a contului

Link-urile din acest email duc înapoi către *EventiFind* pentru confirmarea contului. După cum se poate observa în imagine, link-ul conține un parametru numit *token*, o variabilă cu valoare unică folosită pentru confirmare. Acest *token* pe lângă faptul că poate fi folosit o singură dată pentru activarea contului, are un timp finit de valabilitate după care va trebui generat un alt link și trimis un alt email.

### 3.2.6 Diagrame de secvență

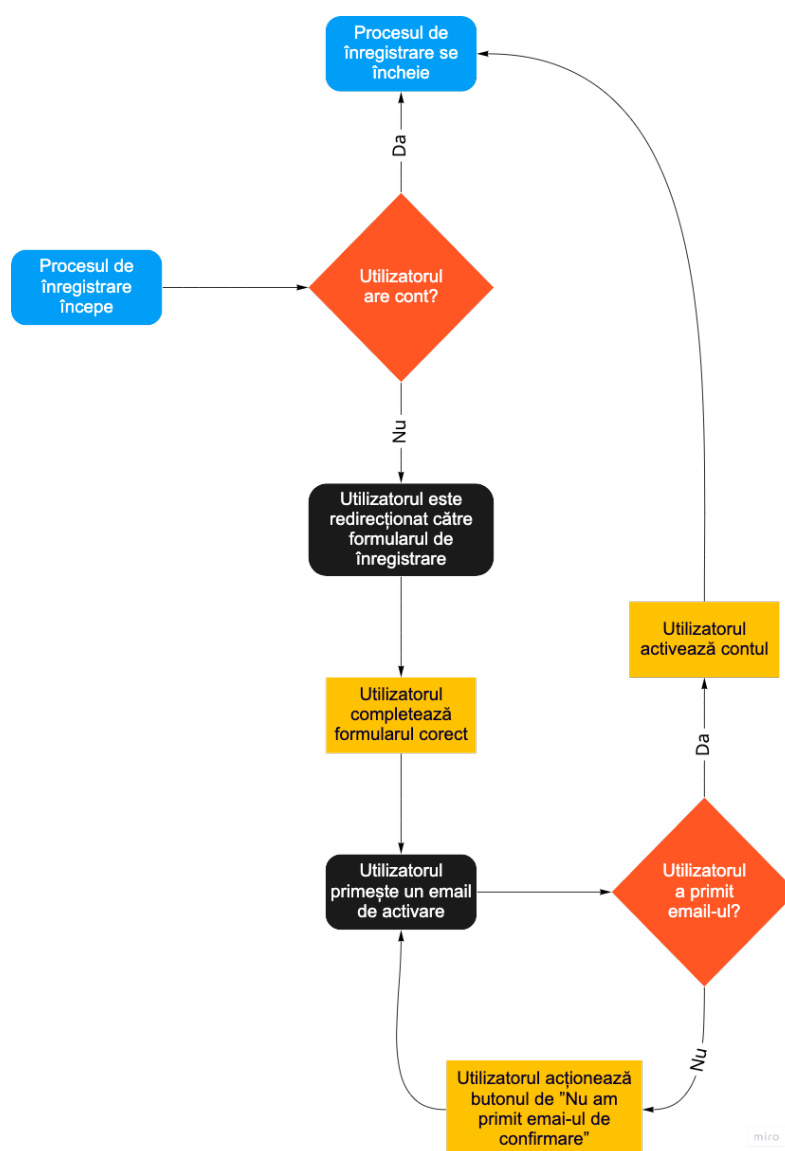
Aplicația dezvoltată de mine conține diverse fluxuri pentru îndeplinirea anumitor obiective și acestea se împart, de asemenea, în mai multe fluxuri în funcție de tipul de utilizator



(administrator sau utilizator normal). Pentru a prezenta câteva dintre aceste fluxuri, am ales să le reprezint folosind *flowchart-uri*.

Am ales această reprezentare pentru a le face cât mai puțin tehnice și a evita, astfel, confuzia. Flowchart-ul păstrează atenția asupra logicii din spatele fiecărui flux iar în cadrul acestuia primează expunerea vizuală a ideilor.

Fiecare bloc din diagramă reprezintă un tip de interacțiune cu utilizatorul. Blocurile negre reprezintă acțiunile automate la care este supus utilizatorul (redirecționări, trimis de email-uri, etc.), blocurile galbene reprezintă acțiuni pe care le face utilizatorul (completarea de formulare, acționarea de butoane, etc.), blocurile portocalii sub formă de romb reprezintă structuri de decizie asupra anumitor stări ale fluxului, iar blocurile colorate în alte culori reprezintă blocuri ce delimitează începutul și sfârșitul fluxului.

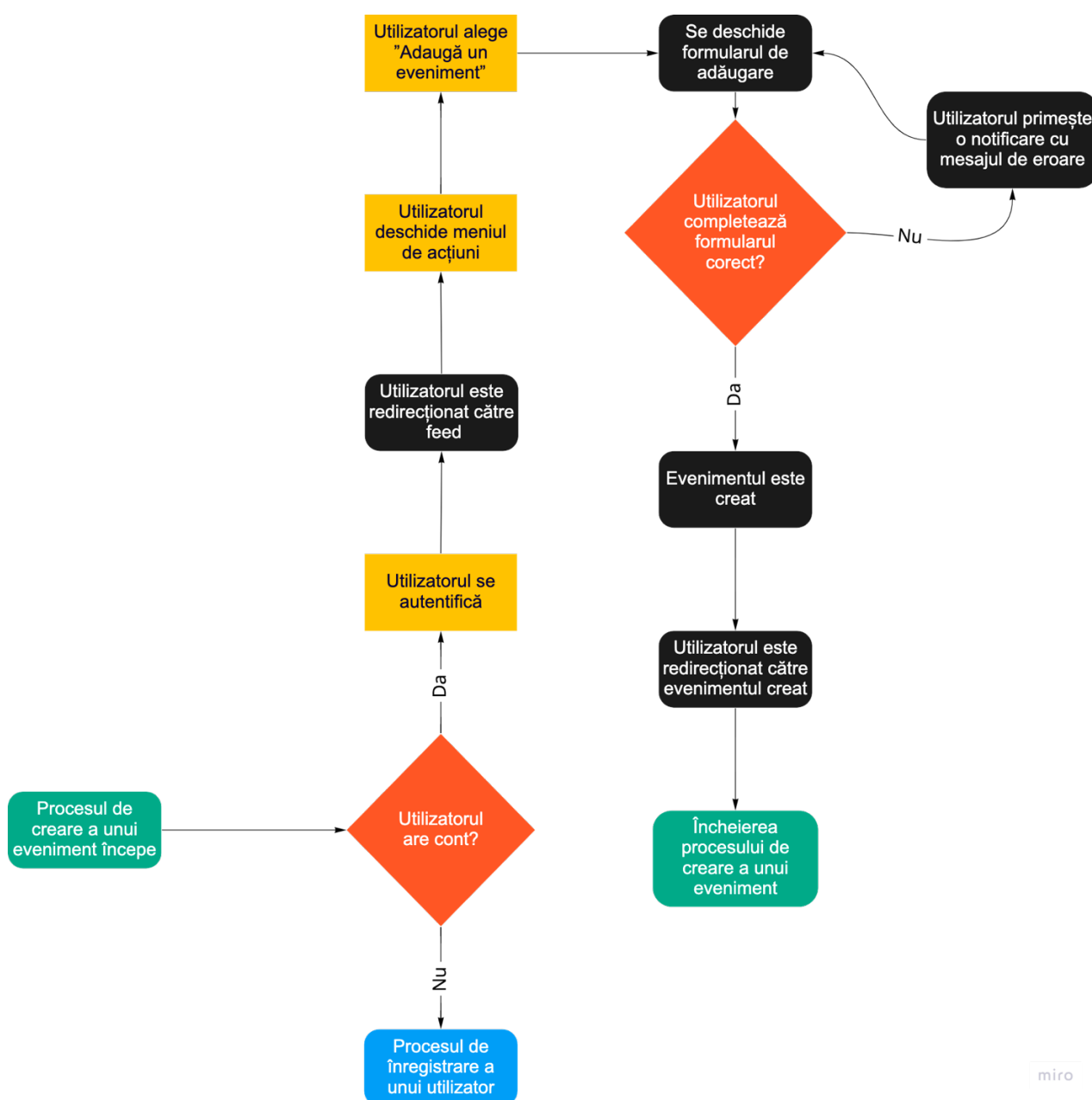


Figură 13 - Procesul de înregistrare a unui utilizator

Aceste fluxuri puse cap la cap formează imaginea de ansamblu a website-ului, ea conținând totalitatea acțiunilor pe care un utilizator le poate executa. Din cauza complexității ridicate a diagramei, am ales să prezint doar câteva din aceste fluxuri, toate separate unul de celălalt.

Unele dintre cele mai de bază fluxuri dintr-o aplicație web sunt cele de înregistrare și autentificare a utilizatorilor. Înainte de a putea realiza orice acțiune pe website, utilizatorii trebuie atât să aibă un cont activ, cât și să fie conectați cu acesta. Pentru a nu repeta descrierea unui proces relativ simplu, următoarea diagramă reprezintă fluxul de înregistrare a unui utilizator.

În continuare voi prezenta procesul prin care un utilizator poate crea un eveniment folosind meniul de acțiuni. Acest meniu se poate deschide apăsând pe poza de profil a utilizatorului

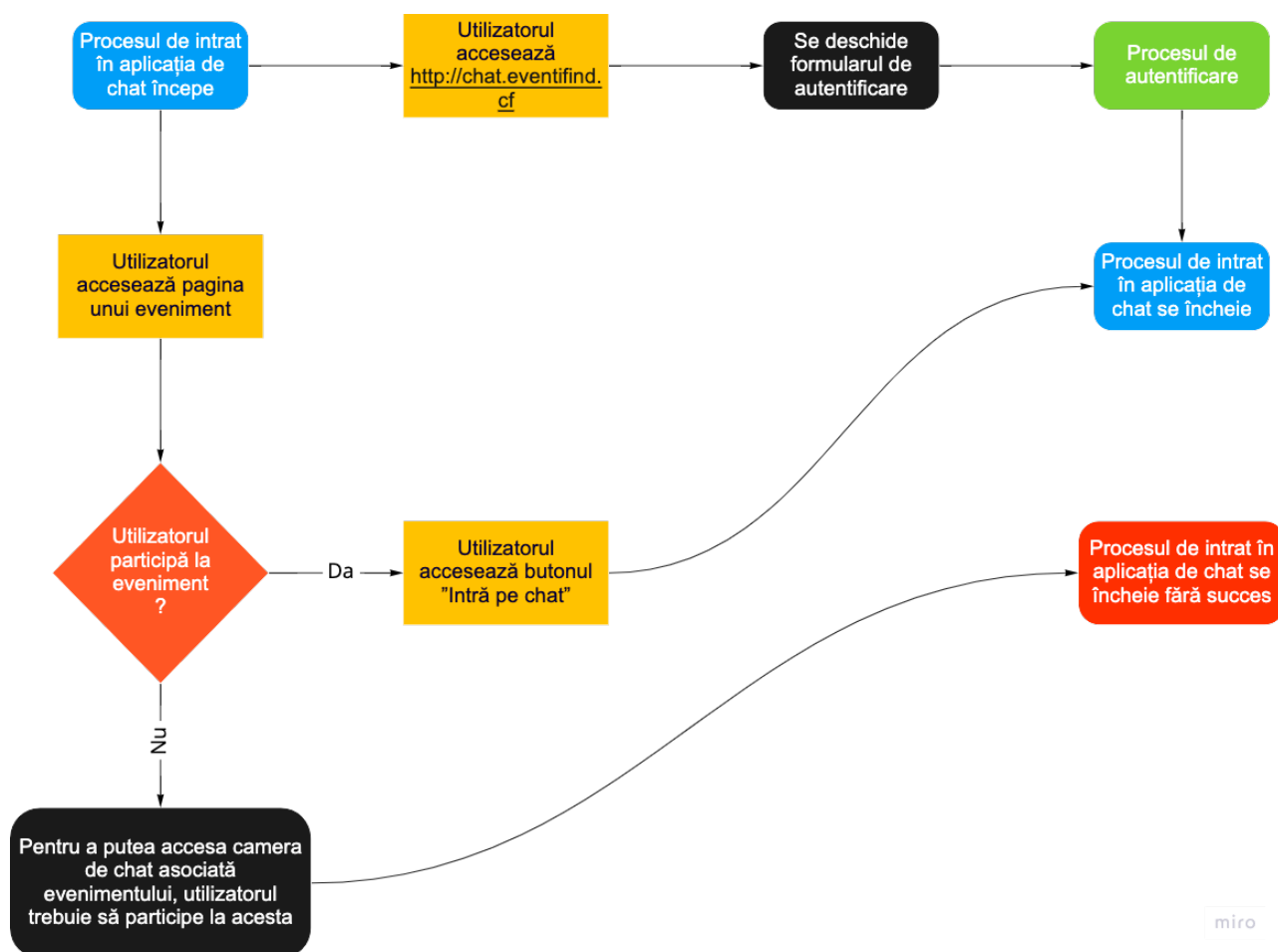


Figură 14 - Proces pentru adăugarea unui eveniment nou

conectat ce se află în partea din dreapta sus a paginii și în partea dreaptă a bării de navigație. În momentul în care se deschide un modal cu multiple opțiuni, utilizatorul va acționa butonul "Adaugă un eveniment" și apoi va fi redirectionat către formularul de adăugare.

Evident că, pentru adăugarea unui itinerariu trebuie urmați aproximativ aceiași pași. Asemănător, editarea funcționează pe același principiu cu mențiunea că butonul de modificare a unui eveniment se află în pagina evenimentului/itinerariului, unde se poate găsi și butonul de ștergere.

O altă funcționalitate a cărei flux îl voi reprezenta folosind o diagramă este cea de accesat chat-ul. O primă modalitate este accesarea directă folosind url-ul <http://chat.eventifind.cf> urmată de conectarea cu contul utilizatorului. Un alt mod de a deschide chat-ul este prin apăsarea butonului "Intră pe chat" din pagina unui eveniment. Butonul va apărea doar în momentul în care utilizatorul participă la, nu doar urmărește, evenimentul respectiv.

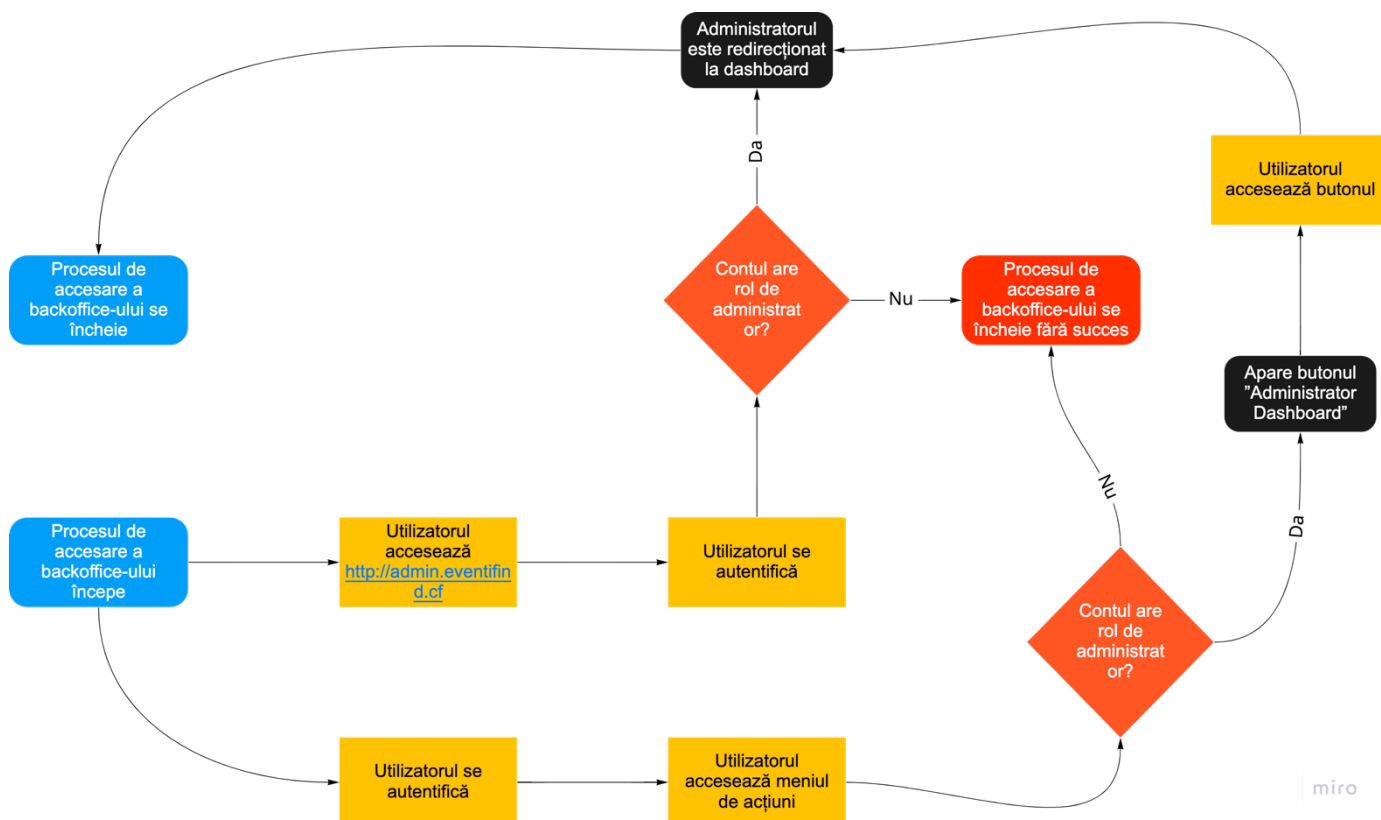


Figură 15 - Proces pentru accesarea chat-ului

Un alt flux foarte important îl reprezintă modul în care un administrator intră în cont și accesează *dashboard-ul* ce controlează atât utilizatorii cât și ce au postat aceștia. Pentru a intra

în *dashboard*, administratorul trebuie, în primul rând, să se conecteze cu un cont ce are rolul de *admin*. Apoi, asemănător adăugării unui eveniment nou, din meniul de acțiuni va accesa butonul "Administrator Dashboard" și va fi redirecționat către dashboard.

De aici, el poate, spre exemplu, să modifice rolurile unui utilizator accesând secțiunea *Utilizatori* și făcând click pe unul dintre aceștia.



Figură 16 - Procesul de accesare a backoffice-ului

### 3.2.7 Tipul de date Enum

Pe lângă celelalte proprietăți, clasa evenimente prezintă două ce se diferențiază față de celelalte: *category* și *icon*. Atributul *category* reprezintă numele categoriei din care face parte evenimentul respectiv iar *icon* este un atribut ce stochează numele iconiței specifice categoriei. Acesta ia valoarea unei clase specifice din librăria de iconițe *font-awesome* care va fi folosită la afișare.

```
<i class="fa-brands fa-apple"></i>
```

Codul 16 - Exemplu de afișare pentru o iconiță

Deoarece doresc să se păstreze un stil uniform și ordonat al website-ului, am decis că nu voi permite utilizatorilor să adauge propriile categorii și propriile iconițe. De asemenea, deoarece consider că lista de categorii nu va mai avea nevoie de alte intrări, nu am creat un tabel propriu în baza de date pe care un administrator să îl poată modifica ulterior. Acest lucru ar fi complicat baza de date și ar fi îngreunat înțelegerea și mentenanța acesteia. În schimb, am folosit tipul de date *enum*.

Un *enum* stochează toate categoriile respective sub forma *cheie* -> *valoare* iar atunci când un utilizator adaugă o categorie, numele acesteia va fi asociat cu cheia din *enum*. La fel ca și în cazul categoriei, *enum-ul* ce stochează clasele iconițelor va fi de aceeași formă, important fiind faptul că proprietatea *cheie* a categoriei cu proprietatea *cheie* a iconiței vor trebui să fie aceleași. Asocierea între cele două se face în controller-ul *events\_controller*, în metoda *create* și constă în preluarea cheii de la categorie, extragerea valorii atribuite cheii respective din *enum-ul* de iconițe și atribuirea acestei valori câmpului *icon* din noul eveniment creat.

```
# in event.rb
enum category: {
  Altele: 0,
  Muzica: 1,
  Sport: 2,
  Cultura: 3,
  Educatie: 4,
  Business: 5,
  Artă: 6,
  Social: 7,
  Stiinta: 8,
  Politica: 9,
  Concurs: 10
}

enum icon: {
  "fa-solid fa-question": 0,
  "fa-solid fa-music": 1,
  "fa-solid fa-futbol": 2,
  "fa-solid fa-book": 3,
  "fa-solid fa-graduation-cap": 4,
  "fa-solid fa-briefcase": 5,
  "fa-solid fa-palette": 6,
  "fa-solid fa-users": 7,
  "fa-solid fa-flask": 8,
  "fa-solid fa-flag": 9,
  "fa-solid fa-gavel": 10
}

# in events_controller metoda create
category_number = Event.categories[params[:event][:category]]
@event.icon = Event.icons.key(category_number)
```

*Codul 17 - Enum pentru categorii și iconițe*

### 3.2.8 Comunicarea dintre controller și view. Exemplu

În arhitectura MVC, la sfârșitul execuției unei metode a controller-ului, aceasta randează view-ul asociat (view-ul cu același nume al metodei dacă nu este specificat altfel). În view pot fi folosite informații prelucrate în controller și declarate global folosind caracterul *@* la începutul numelui variabilei. O astfel de cooperare poate fi observată în cazul *feed-ului*.

Feed-ul este o pagină pe care utilizatorul o poate accesa doar dacă intră în cont și poate vedea trei categorii de informații: la ce participă persoanele pe care acesta le urmărește, ce organizează persoanele pe care acesta le urmărește și ultimele schimbări ce au avut loc la evenimentele/itinerariile la care participă. Cele trei tipuri de informații sunt stocate în liste (variabile globale din controller) și folosite în view.

În Ruby on Rails, variabilele globale pot fi folosite în alte metode din controller și în view-urile asociate, dar nu și în alte clase. Pentru ca listele să rămână relevante și informațiile să nu se amestece unele cu altele, am ales să le împart în cele trei categorii discutate și să afișez doar informații referitoare la evenimentele/itinerariile din viitor. Odată ce evenimentul s-a încheiat, intrarea din feed a acestuia va dispărea păstrând feed-ul organizat și ușor de urmărit.

Spre exemplu, în controller-ul asociat feed-ului, în metoda *index* am calculat variabila *following\_events* ce conține evenimentele/itinerariile la care vor participa persoanele urmărite de utilizatorul curent, sortate după data de început a evenimentului/itinerariului.

```
@following_events = []  
# variabila following_events stocheaza evenimente/itinerarii  
# la care va participa utilizatorul curent  
@following.each do |user|  
  @following_events << user.find_future_participations  
end  
  
@following_events.flatten!  
  
# sorteaza dupa data de inceput  
@following_events.sort_by! { |event| event.participatable.start_date }
```

*Codul 18 - Variabila @following\_events stochează lista de evenimente*

Asemănător *feed-ului* avem funcția de căutare în site. Aceasta poate fi accesată din bara de navigație și poate fi folosită de utilizatori pentru a căuta atât evenimente/itinerarii cât și alți utilizatori.

De obicei atunci când căutăm ceva pe o pagină web, vrem să avem o plajă cât mai largă de rezultate. Tocmai de aceea, am ales să implementez o căutare care să returneze cât mai multe

rezultate relevante iar aceste rezultate să fie împărțite pe categorii, asemenea feed-ului. Această metodă oferă utilizatorului resursele necesare fără să facă o filtrare ulterioară a rezultatelor.

Pentru a transforma această idee în realitate, în metoda *index* din controller-ul *search\_controller* am implementat un algoritm care, după ce sanitizează șirul de caractere primit de la utilizator, folosind secvența *WHERE...LIKE* selectează din baza de date toate înregistrările de tip eveniment/itinerariu ce conțin secvența căutată în titlu, în descriere sau la locație (în cazul evenimentelor). În cazul utilizatorilor, căutarea se face pentru *username*, nume și prenume. Rezultatele sunt afișate sub formă de liste ce conțin elemente de tip link-uri ce duc la pagina aferentă fiecăruia.

Ca o măsură de securitate, pentru a se evita eventualele atacuri de tip *SQL Injection*, am folosit un *placeholder* marcat cu caracterul *?* după care am specificat valoarea căutată. Ruby on Rails sanitizează șirul de caractere căutat și elimină posibilele caractere periculoase (*"/"; "--"*). De asemenea, șirul de caractere căutat este transformat în litere mici folosind metoda *downcase*, îi sunt șterse caractere non-alfanumerice, îi sunt înlăturate spațiile în plus și îi sunt adăugate caractere *%* între cuvinte pentru a putea fi căutate în interogare.

```
@found_schedules = Schedule.where("lower(title) LIKE ? OR lower(description) LIKE ?",
@search_string, @search_string)
```

*Codul 19 - Căutare folosind WHERE...LIKE*

### 3.2.9 Frontend

O aplicație web este alcătuită din *frontend* și *backend*.

Pentru a înțelege mai bine cele două componente voi folosi următoarea analogie: presupunem că utilizatorii sunt clienții unui restaurant. Frontendul reprezintă totalitatea lucrurilor cu care clientul interacționează atunci când intră în restaurant (mesele, luminile, tacâmurile, etc.). În general, se dorește ca frontendul să fie intuitiv (clienților să nu le fie greu să își dea seama cum să navigheze pe aplicație) și plăcut din punct de vedere vizual. Continuând analogia, backendul este bucătăria, locul în care se preiau informațiile de la clienți și sunt prelucrate astfel încât aceștia să primească ce își doresc.

Spre deosebire de alte stack-uri de tehnologii (React/NodeJs), în care API-ul lucrează cu răspunsuri de tip json, între frontendul și backendul de Ruby on Rails se transmite html. În analogia făcută, API-ul reprezintă ospătarul care aduce răspunsul din partea backend-ului după ce a fost făcută o cerere de către clienți.

Datorită faptului că Ruby on Rails este un framework *batteries included*, atât frontendul cât și backendul au fost realizate în cadrul aceleiași aplicații.

Frontendul în aplicație este scris în html, în fișiere de tip embedded ruby (.html.erb), cu elemente de JavaScript și jQuery și reprezintă view-urile din cadrul arhitecturii MVC.

Stilizarea a fost realizată folosind framework-ul de CSS, Bootstrap 5 deoarece permite accesul la elemente de frontend plăcute vizual și ajută la crearea paginilor responsive (elementele de frontend își modifică dimensiunile dinamic în funcție de mărimea dispozitivului de pe care clientul accesează aplicația). De asemenea, am folosit iconițele FontAwesome pentru a face aplicația mai intuitivă (pentru a face design-ul asemănător cu cel al site-urilor populare cu care utilizatorii să fie deja familiari).

Elementele de JavaScript au fost folosite, în principal, pentru comunicarea cu API-ul de la Google Maps. Am ales să folosesc serviciul Google Maps pentru multitudinea de micro-servicii pe care le oferă (hărți, completarea automată a locațiilor, etc.) și pentru acuratețea. Am vrut să afișez hărțile pentru ca utilizatorul să aibă o experiență mai plăcută vizual și pentru a elimina anumite neclarități care ar fi putut apărea în momentul în care un utilizator ar fi completat o adresă greșită. În acest caz, el poate vedea imediat pe hartă locația exactă a adresei introduse.

De asemenea, în cazul itinerariilor, am vrut să afișez pe hartă sub forma unui traseu care unește anumite puncte pentru a avea o poveste (image de ansamblu) ușor de conceptualizat.

Pentru componentele de html care trebuie încărcate în mai multe pagini am folosit parțiale. Spre exemplu, pentru o navigare mai facilă, vrem ca bara de navigație să apară pe majoritatea paginilor dar ar fi redundant să repetăm codul în toate fișierele. Astfel, folosim un *partial* pentru a accesa codul dintr-un fișier în celelalte și îl afișăm în pagină folosind helper-ul render.

```
<%= render partial: "shared/navbar" %>
```

*Codul 20 - Comandă pentru afișarea barei de navigație*

În cazul mesageriei, utilizatorii sunt obișnuiți să primească mesaje instant și pentru a satisface această nevoie, frontendul a fost adaptat folosind unele instrumente puse la dispoziție de Ruby on Rails.

## 3.3 IMPLEMENTARE

### 3.3.1 Rolurile utilizatorului

Deoarece aplicația va afișa în mod direct conținut generat de către utilizatori, este nevoie de moderatori care să poată menține conținutul potrivit în platformă și să înlăture utilizatorii ce nu



se conformează cu regulile impuse. Deoarece moderatorii sunt persoane cu drepturi mai elevate față de utilizatorii normali, este nevoie de o modalitate de a le atribui roluri diferite. Metoda aleasă de mine a fost folosirea gem-ului *pundit* în combinație cu gem-ul *rolify*. Din perspectiva securității, utilizarea unor librării cunoscute și actualizate constant este un avantaj comparativ cu dezvoltarea acestui sistem manual.

*Rolify* oferă posibilitatea de a crea rolurile și a le stoca în baza de date pentru ca acestea să poată fi folosite ulterior de *pundit*. Aplicația mea dispune de două roluri: *user* și *admin*. *User-ul* este rolul primit în mod implicit de oricine își creează un cont nou. Drepturile *user-ului* sunt standard, el putând să creeze evenimente și itinerarii, să le șteargă și să le editeze doar pe cele create de el. Pe de altă parte, *admin-ul*, pe lângă operațiile de bază, poate gestiona atât utilizatorii cât și ce postează aceștia. Toate acestea sunt făcute din partea din aplicație destinată lui și anume *backoffice-ul*.

Pentru setarea regulilor ce țin de roluri, *pundit* folosește clase speciale numite *policies*. Aceste clase se definesc pentru fiecare model din proiect căruia se dorește a i se aplica reguli de autorizare. Spre exemplu, pentru a defini o regulă ce spune că doar utilizatorii cu rolul *admin* pot accesa metoda *index* din controller-ul de *users*, adăugăm în fișierul *user\_policy.rb* următoarele:

```
def index?  
  @user.has_role? :admin  
end
```

*Codul 21 - Adăugarea unei reguli într-un policy*

De asemenea, în cadrul controller-ului trebuie adăugată linia de cod *authorize @users* pentru a îi spune policy-ului să verifice dacă utilizatorul curent are destule drepturi pentru a accesa toți utilizatorii. Politicile se pot seta atât în funcție de rolul utilizatorului, cât și de identitatea utilizatorului. Spre exemplu, din cadrul policy-urilor se poate seta ca un anumit utilizator să aibă acces de editare sau ștergere exclusiv la evenimentele create de el și niciun altul.

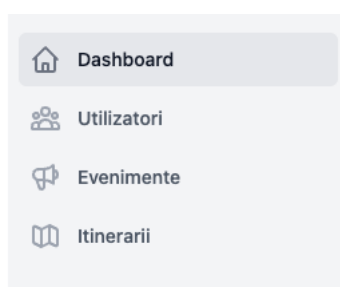
### 3.3.2 Backoffice

Administratorul/administratorii au un rol foarte important în buna funcționare a aplicației. Pe lângă rolul de utilizator normal cu ajutorul căruia pot crea și participa la evenimente, aceștia pot modera tot ce ține de aplicație. Un administrator va putea șterge sau modifica un eveniment dacă acesta consideră că nu corespunde cu politicile site-ului și poate bloca utilizatori din același

considerent. Administratorul poate crea alți administratori prin modificarea rolurilor unui utilizator sau poate înlătura acest drept de la administratori deja existenți.

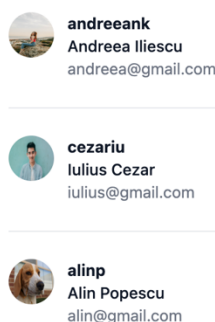
Toate aceste operațiuni pot fi făcute din partea de aplicație numită *backoffice*. Aceasta este construită sub forma unui sistem de management intuitiv la care au acces doar administratorii. În acest segment, prin simpla apăsare a unor butoane, un administrator poate modifica sau șterge un anumit utilizator sau eveniment/itinerariu.

În cadrul aceleiași aplicații este integrat un meniu principal ce conține următoarele secțiuni: *Dashboard, Utilizatori, Evenimente, Itinerarii*.



*Figură 17 - Meniu de navigație pentru Backoffice*

Dacă administratorul accesează unul dintre obiectele meniului, acesta va fi redirecționat către o listă cu toate acestea. Fiecare intrare în listă va avea afișate informații despre entitate. Spre exemplu, un utilizator va avea numele, prenumele, username-ul și emailul afișate pe când un eveniment va avea titlul, username-ul utilizatorului care a creat evenimentul și categoria.



*Figură 18- Exemplu de intrări din lista Utilizatorilor*

Fiecare obiect din listă va reprezenta un link către pagina asociată acestuia unde administratorul poate acționa butoanele: *Editează, Șterge și Blochează* (în cazul utilizatorilor).

### 3.3.3 Stocarea imaginilor

Pentru o interfață mai prietenoasă, fiecare utilizator are opțiunea de a își adăuga o imagine de profil (avatar). Pentru a face posibilă încărcarea unei poze, am folosit Active Storage. Acest serviciu specific framework-ului Ruby on Rails facilitează stocarea de documente local sau în cloud. Inițial, mediul de stocare activ era cel local, fișierele încărcate fiind salvate în folder-ul *storage* din directorul aplicației.

```
local:
  service: Disk
  root: <%= Rails.root.join("storage") %>
```

*Codul 22 - Mediul local din fișierul storage.yml*

Deși această abordare este simplă și rapidă, ea nu reprezintă o soluție scalabilă (reprezintă o posibilă vulnerabilitate pentru aplicație). Din acest considerent, am apelat la serviciul de stocare în cloud, și anume Amazon AWS S3 și am adăugat configurația de mai jos în fișierul *storage.yml*.

```
amazon:
  service: S3
  access_key_id: <%= Rails.application.credentials.dig(:aws, :access_key_id) %>
  secret_access_key: <%= Rails.application.credentials.dig(:aws, :secret_access_key) %>
  region: <%= Rails.application.credentials.dig(:aws, :region) %>
  bucket: <%= Rails.application.credentials.dig(:aws, :bucket) %>
```

*Codul 23 - Mediul amazon din fișierul storage.yml*

Informațiile legate de conexiune sunt preluate din fișierul *credentials.yml* folosind o cheie pentru decriptarea informațiilor. Deoarece nu pot fi adăugate imagini într-un tabel din baza de date, folosim un atribut în modelul *user* numit *avatar*. De asemenea, în baza de date au fost create trei tabele în urma rulării comenzii *rails active\_storage:install* (*active\_storage\_attachments*, *active\_storage\_blobs*, *active\_storage\_variant\_records*). Aceste tabele stochează informația despre fișierele încărcate și legătura cu Amazon AWS. Dacă vrem să adăugăm imagini altor modele din aplicație, nu va mai fi nevoie să creăm alte tabele ci vor fi folosite aceleași. Asocierea unei imagini cu un model și încărcarea acesteia de către utilizator se fac prin simpla adăugare a următoarelor linii de cod:

```
has_one_attached :avatar # în modelul user.rb
```

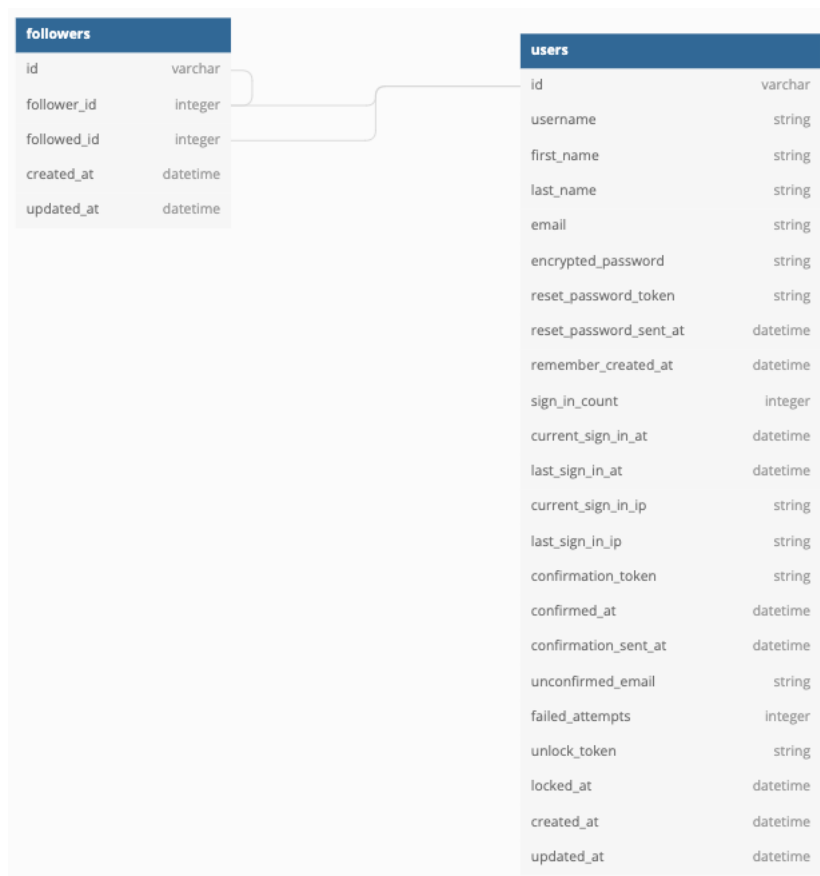
```
<%= f.file_field :avatar %> # în formularul de adăugare/editare a unui utilizator
```

*Codul 24 - Cod pentru asocierea unei imagini cu un model*

### 3.3.4 Aspectul de rețea de socializare

Datorită faptului că platforma se dorește a fi una socială iar legăturile formate între utilizatori să rămână, am adăugat funcționalitatea de *urmărire* a altui utilizator. Folosind un buton aflat pe profilul unei persoane, orice utilizator poate urmări pe oricine. Legăturile create pot fi observate sub forma unei liste pe pagina de profil a utilizatorului curent. Din această secțiune, utilizatorul poate să șteargă persoane pe care le urmărește.


Relația din spatele acestei funcționalități este una de many-to-many și este implementată cu ajutorul unei tabele asociative ce leagă un *user* de un alt *user*. Tabela respectivă are două proprietăți: *follower\_id* ce reprezintă id-ul utilizatorului care a acționat butonul de *follow* și *followed\_id* ce reprezintă id-ul utilizatorului pe care acesta îl va urmări.



Figură 19 - Relația dintre Users și Followers

### 3.3.5 Public id

Ruby on Rails este un framework relativ sigur când vine vorba de diverse tipuri de atacuri, însă un mod de a adăuga la această siguranță este modificarea id-ului din url. În momentul în care se generează pentru prima dată controller-ele, modelele și view-urile iar utilizatorul accesează pagina de afișare (metoda show din controller ce afișează view-ul show) url-ul este de forma:

 localhost:3000/events/1

Figură 20 - Url pentru evenimentul cu id

1

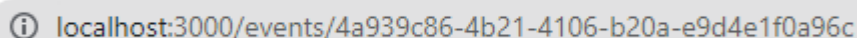
Această informație este foarte importantă pentru un potențial atacator deoarece conține numărul de ordine din baza de date a entității respective. Similar evenimentelor, profilul utilizatorilor conținea inițial id-ul în url.

Pentru a evita această situație cu potențial distructiv, în cazul evenimentelor și a itinerariilor, după adăugare, am generat într-o proprietate numită *public\_id* un id care poate fi folosit atât pentru identificarea înregistrării cât și pentru securizare.

Pentru a genera acest *public\_id* am folosit modulul *SecureRandom* și metoda *uuid* (universal unique identifier). Id-ul este unic pentru fiecare înregistrare iar acesta nu dezvăluie nicio informație referitoare la poziția înregistrării în tabelul din baza de date. O altă schimbare determinată de această proprietate este căutarea după *public\_id* în locul *id-ului* inițial.

```
after_create :add_public_id
def add_public_id
  self.public_id = SecureRandom.uuid
  self.save
end
```

*Codul 25 - Metoda add\_public\_id din modelul event*



*Figură 21 - Url pentru un eveniment cu public\_id*

Similar cu această problemă a evenimentelor și itinerariilor o are și modelul de utilizator (*user*), mai exact pagina de profil a acestuia. În acest caz, nu am generat un nou id public pentru url ci am folosit *username-ul* utilizatorului întrucât acestuia îi este interzis prin intermediul validărilor să conțină spații și, de asemenea, este unic. Această îmbunătățire adusă aplicației îi conferă și un aspect mai prietenos și profesional în momentul în care un utilizator distribuie link-ul către un profil, eveniment sau itinerariu.

### 3.3.6 Validări

Validările sunt făcute în fișierul unde este definit modelul pentru toate entitățile. Validările reprezintă interzicerea sau impunerea unor reguli pentru operațiunile efectuate asupra proprietăților obiectului respectiv. Spre exemplu, luând modelul *user*, am stabilit că *username-ul* acestuia este unic și nu poate fi luat de către alt utilizator. Pentru acest lucru, se definește în cadrul clasei *User* următoarea regulă:

```
validates_uniqueness_of :username
```

*Codul 26 - Validarea ce asigură unicitatea username-ului*

Un alt tip de validare folosit în multe instanțe a fost validarea existenței unei proprietăți. Acest lucru interzice crearea unei înregistrări de tip obiectul respectiv care să nu conțină nicio informație în câmpul destinat proprietății.

Spre exemplu, am folosit această validare pentru a mă asigura că fiecare eveniment are un titlu, o dată și oră de început și una de final. Acest aspect ajută la securizarea în profunzime a aplicației limitând aportul pe care îl are un utilizator asupra conținutului creat întrucât generarea de conținut din partea utilizatorilor poate duce la breșe de securitate.

### 3.3.7 Testare

Pentru a putea vedea dacă metodele scrise corespund cu rezultatele așteptate, am integrat testarea automată în cadrul proiectului. Testarea metodelor am făcut-o folosind gem-ul *capybara*.

Testarea controllere-lor se face în cadrul folder-ului */test/controllers* iar o metodă este denumită sub forma *test "should <acțiune a metodei controllerului>" do*. Spre exemplu, pentru a testa dacă o cerere de tip *get* la url-ul *event\_url(/events)* cu parametrul *public\_id* va executa cu succes metoda *show* din cadrul controller-ului *events\_controller* vom rula următorul test:

```
test "should show event" do
  get event_url(@event.public_id)
  assert_response :success
end
```

*Codul 27 - Test pentru executarea metodei show din events\_controller*

În același timp, în cadrul folderului *test/fixtures*, există și o bază de date statică sub forma de fișiere cu extensia *.yaml* ce reprezintă exemple de entități care vor fi folosite în cadrul celorlalte teste. În cazul evenimentelor, acest fișier conține exemple de valori pe care le pot avea atributele unui eveniment. Cu cât sunt mai variate aceste exemple de valori, cu atât testele acoperă mai multe cazuri. De exemplu, se pot completa doar anumite atribute pentru unul dintre aceste înregistrări pentru a testa metodele de validare cu privire la existența valorilor pentru un anumit câmp.

```
one:
  title: "Event 1"
  description: "This is the first event"
  start_time: "2020-01-01T00:00:00Z"
  end_time: "2020-01-01T00:00:00Z"
  location: "Braşov"
  user_id: 1
  category_id: 1
  icon: "fa-solid fa-question"
  public_id: "f4c46b17-3fd7-43c1-ab1e-7321da317a02"
  created_at: "2020-01-01T00:00:00Z"
  updated_at: "2020-01-01T00:00:00Z"
```

*Codul 28 - Exemplu de înregistrare pentru un eveniment din fixtures*

Pentru a rula toate testele se foloseşte comanda *rails test*. În cadrul testelor de metode în care este nevoie de date pentru crearea de înregistrări, datele sunt preluate din fişierele menţionate anterior, şi anume cele din directorul *fixtures*. Pe lângă rularea tuturor testelor, se poate rula un singur fişier de teste cu comanda *rails test &ltcale\_ fişier>* dar şi rularea unei anumite metode din cadrul unui fişier de teste adăugând la comanda anterioară flag-ul *-n* urmat de numele metodei.

## 4 CONCLUZII

---

**Concluzii generale**

**Contribuţii personale**

**Posibilităţi de dezvoltare ulterioară**

---

### 4.1 CONCLUZII GENERALE

Una din concluziile generale pe care o pot formula după ce am lucrat la dezvoltarea proiectului de licenţă este aceea că Ruby on Rails este printre cele mai avantajoase instrumente pentru dezvoltarea unui proiect care să nu necesite o echipă mare de persoane.

Conform articolului [16], o mulţime de start-up-uri folosesc cu succes Ruby on Rails pentru a îşi construi website-urile proprii. Un start-up este o companie mică şi nou-creată pe piaţă. În mod clar, nevoile acestor companii sunt complet diferite fată de nevoile corporaţiilor, întrucât acestea nu au avut destul timp pentru a se expune pe piaţă şi a îşi crea o reputaţie.



În primul rând, o astfel de companie are nevoie de un produs bun într-un timp cât mai scurt, lucru care este asigurat de dezvoltarea rapidă în Ruby on Rails care folosește multiple module și librării pentru a integra funcționalități comune fără ca programatorii să trebuiască să le scrie manual.

În al doilea rând, un start-up nu are resursele financiare ale unei corporații, deci nu își permite să investească mulți bani în anumite tehnologii. Ruby on Rails este un framework *open source* distribuit cu licență MIT, asemenea *gem-urilor* care pot fi folosite gratuit în dezvoltare.

Cea de-a doua concluzie a venit în mod natural atunci când am realizat cât de importantă este gestionarea datelor în toată industria informatică. Deși ca studentă, foarte mare parte din curiculă a fost construită în jurul datelor, abia după ce am realizat o aplicație care să aibă un potențial real de a fi utilizată la scară largă am conștientizat cu adevărat cât de mare parte din viața noastră o reprezintă datele și ce avantaj enorm ne conferă sistemele de calcul prin faptul că ne permit să manipulăm și să stocăm o cantitate inimaginabilă de date.

Datele necesare pentru ca Eventifind să funcționeze nu sunt deloc puține. Dacă ar fi să rezum în câteva cuvinte ce face aplicația pe care am construit-o, aceste cuvinte ar fi: manipularea corectă a datelor. Securitatea este desigur și ea un mare jucător pe scena aceasta a importanței dar și ea, în definitiv, este doar răspunsul la întrebarea: cum ne asigurăm că datele noastre ajung acolo unde trebuie?

Eventifind este o aplicație care ușurează accesul la anumite date, vrem să fim conectați cu lumea din jurul nostru, vrem să primim și să transmitem informații. Nu trebuie să căutăm mult în natura noastră umană pentru a găsi această curiozitate, această dorință de a înțelege și a cunoaște. Tot ce face aplicația mea este să aducă mai aproape informația de persoana care are nevoie de ea, manipularea corectă a datelor.

## 4.2 CONTRIBUȚII PERSONALE

Domeniul de *social media*, felul în care evoluează socializarea și formarea de legături dintre oameni au fost din totdeauna subiecte care m-au pasionat. Facebook, Instagram și alte platforme de socializare vor rămâne cu siguranță în istorie pentru felul în care au revoluționat comunicarea.

Cu toate acestea, suntem foarte la început. Prietenii virtuali, mesageria instantă, fluxul interminabil de informații sunt toate invenții fără precedent în istoria umanității și este complet normal să nu ne pricepem foarte bine la gestionarea lor dar acest aspect nu reprezintă o scuză ca să nu încercăm.

Evenimentele de pe Facebook fac o treabă extraordinară la a aduce oamenii împreună dar sunt deficitare pe partea de creare a relațiilor profunde în care participanții să se simtă în siguranță și în largul lor în același timp. Aportul meu a fost acela de identifica problema și a veni cu o soluția pe care am considerat-o cea mai eficientă.

Primul pas spre a reuși să ating acest obiectiv a fost să îmi sintetizez ideile cu privire la platformă și să reușesc să schițez un plan pentru dezvoltarea aplicației. Pentru aceasta m-am folosit de *flowchart-uri* pentru a alcătui logica de business și apoi am construit schema pentru baza de date.

Deși, de multe ori, în procesul de dezvoltare depanarea erorilor este trecută cu vederea foarte ușor, acest proces a constituit o parte semnificativă din timpul pe care l-am alocat dezvoltării proiectului.

Depanarea erorilor este momentul în care un dezvoltator este obligat să analizeze și să înțeleagă proiectul în profunzime. Deși pot afecta moralul, erorile sunt o parte esențială a procesului de învățare.

#### **4.3 POSIBILITĂȚI DE DEZVOLTARE ULTERIOARĂ**

În viitor am în plan să dezvolt pe fundația creată de această aplicație și să adaug o aplicație de mobil.

Pentru acest lucru voi dezvolta un API ce va folosi aceeași bază de date și două aplicații native de mobil, una pentru Android și una pentru iOS ce vor putea să comunice cu baza de date folosind API-ul. Asemenea aplicației de web, aplicația de mobil va avea un chat și o secțiune unde utilizatorul va putea să caute evenimente și itinerarii.

Pe lângă aceasta, voi oferi posibilitatea utilizatorilor ca, după ce au participat la un anumit eveniment sau itinerariu, să fie rugați să ofere un feedback atât pentru experiență în sine cât și pentru organizator. Astfel, fiecare utilizator ce organizează evenimente va avea un scor pe profilul său în funcție de notele pe care le primește de la participanți.

## BIBLIOGRAFIE

- [1] Studiul Harvard asupra fericirii, <https://www.ambasador.ro/studiul-harvard-asupra-fericirii/>
- [2] AnyDesk Downloads, <https://anydesk.com/en/downloads/windows>
- [3] Why you should choose a web application over a desktop one, <https://w3-lab.com/choose-web-application-over-desktop/>
- [4] General-purpose programming language, [https://en.wikipedia.org/wiki/General-purpose\\_programming\\_language](https://en.wikipedia.org/wiki/General-purpose_programming_language)
- [5] Ruby (programming language), [https://en.wikipedia.org/wiki/Ruby\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Ruby_(programming_language))
- [6] About Ruby, <https://www.ruby-lang.org/en/about/>
- [7] Ruby on Rails, <https://rubyonrails.org/>
- [8] Turbo-rails, <https://github.com/hotwired/turbo-rails>
- [9] PostgreSQL, <https://www.postgresql.org/>
- [10] Types of Relationship in Database Table, <https://www.javatpoint.com/types-of-relationship-in-database-table>
- [11] Zanfina Svirca (29.05.2020), Everything you need to know about MVC architecture, <https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1>
- [12] Amit Choudhary, Rails 6 allows configurable attribute on #has\_secure\_password, [https://www.bigbinary.com/blog/rails-6-allows-configurable-attribute-name-on-has\\_secure\\_password](https://www.bigbinary.com/blog/rails-6-allows-configurable-attribute-name-on-has_secure_password)
- [13] Bcrypt-ruby, <https://github.com/bcrypt-ruby/bcrypt-ruby>
- [14] Securerandom, <https://github.com/ruby/securerandom>
- [15] Universally Unique Identifier, [https://en.wikipedia.org/wiki/Universally\\_unique\\_identifier#Version\\_4\\_\(random\)](https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_4_(random))
- [16] Class Inheritance, [https://www.w3schools.com/js/js\\_class\\_inheritance.asp](https://www.w3schools.com/js/js_class_inheritance.asp)
- [17] RubyGarage, What are the Benefits of Using Ruby on Rails for Your Startup, <https://medium.com/@RubyGarage/what-are-the-benefits-of-using-ruby-on-rails-for-your-startup-f3d6f6b9938d>

## REZUMAT

EventiFind este o aplicație web de tip rețea de socializare care se ocupă cu managementul evenimentelor, al itinerariilor și al utilizatorilor.

Prin evenimente se înțelege o colecție de întâlniri bine specificate în timp și spațiu între mai mulți utilizatori care să aibă o anumită tematică. Un itinerariu este o colecție de evenimente și locații aflate într-o succesiune logică atât temporală cât și spațială.

Utilizatorii pot interacționa între ei în diferite moduri: funcția de urmărire se referă la faptul că un utilizator poate urmări activitățile oricărui alt utilizator (participarea respectivului la anumite evenimente/itinerarii dar și crearea de evenimente/itinerarii de către acesta).

Utilizatorii mai pot comunica între ei prin aplicația de chat care le permite să schimbe mesaje cu toate persoanele care participă la aceleași evenimente/itinerarii ca și ei.

De asemenea utilizatorii își pot modifica profilele (informațiile pe care aleg să le facă publice pentru alți utilizatori).

Aplicația folosește Ruby on Rails ca și framework de dezvoltare și Postgresql pentru lucrul cu baza de date, alături de alte tehnologii integrate în cele două.

## ABSTRACT

EventiFind is a social networking web application that handles event, itinerary and user management.

By events we mean a collection of meetings well-specified in time and space between multiple users with a specific theme. An itinerary is a collection of events and locations in a logical sequence both temporal and spatial.

Users can interact with each other in different ways: the following function refers to the fact that a user can follow the activities of any other user (their participation in certain events / itineraries but also the creation of events / itineraries by them).

Users can also communicate with each other through the chat application that allows them to exchange messages with all the people who participate in the same events / itineraries as them.

Users can also change their profiles (the information they choose to share with other users).

The application uses Ruby on Rails as a development framework and Postgresql for working with the database, along with other technologies integrated in these two.