Aplicatie online pentru vanzarea biletelor la evenimente

Inginerie Software

Andrei-George Iclodean, Luca-Vasile Graur

Grupa: 30233

Facultatea de Automatica
si Calculatoare

An academic 2023-2024

Cuprins

1 Introducere

Odata cu rapida dezvoltare a internetului, majoritatea serviciilor s-au reolocat, fiind disponibile in mediul online oricarei persoane dispune de o conexiune la internet.

Printre acestea se numara si casele de bilete sau agentiile specializate in comercializarea si distribuirea biletelor pentru diverse evenimente.

Aplicatia pe care am dezvoltat-o in acest proiect propune o solutie pentru comercializarea biletelor la diverse evenimente in spatiul online, oferind utilizatorilor o interfata prietenoasa, dar si o varietate din moduri care usureaza procesul de cumparare a biletelor pentru un eveniment.

Pentru vizualizarea codului sursa, dar si a istoricului sau complet, va incurajum sa accesati **repo-ul proiectului de pe Github.**

1.1 Diagrama Use Case

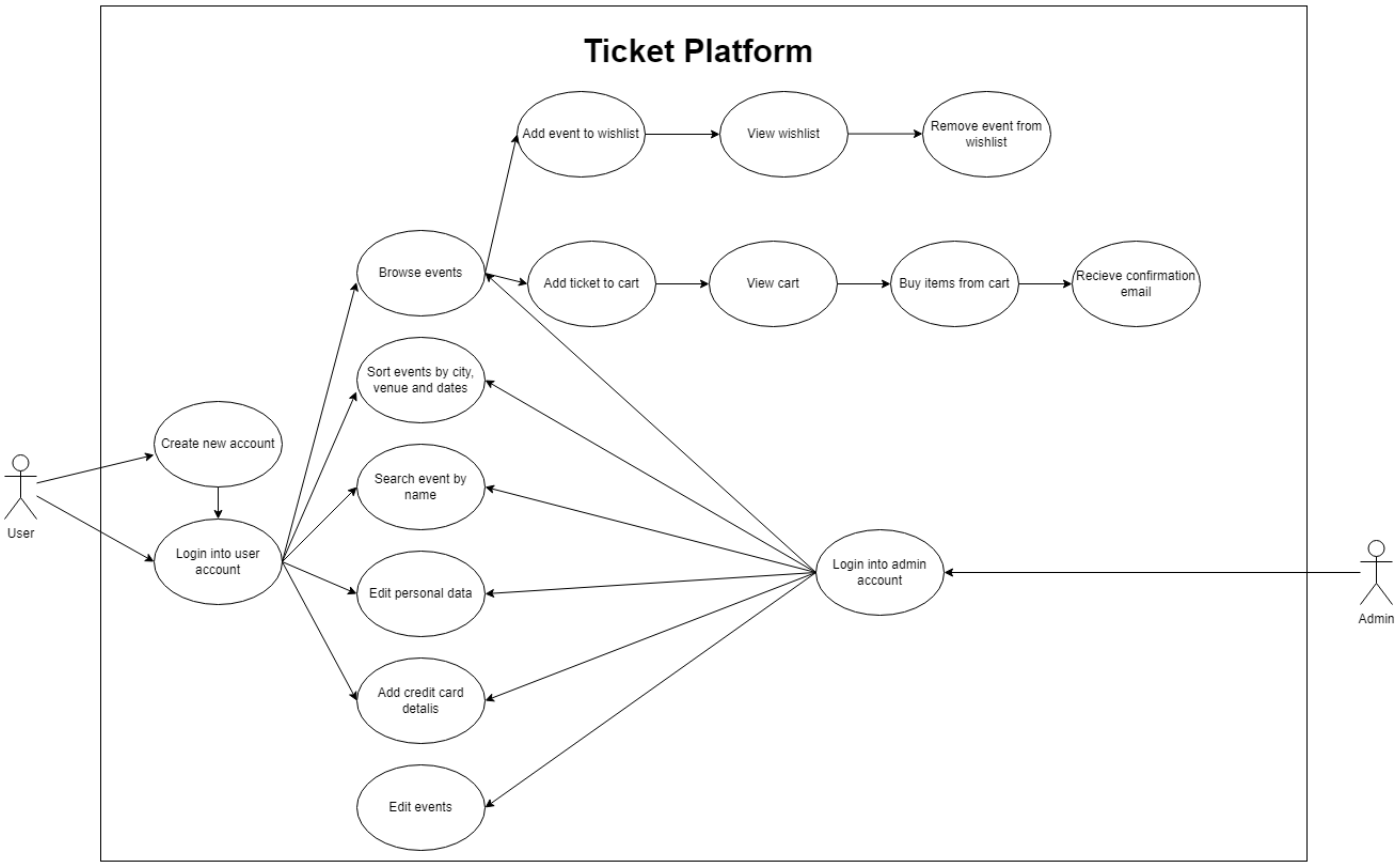

Figura 1: Use Case Diagram

2 Diagrame alese

In continuare vor fi prezentate cele doua diagrame alese, cate una de fiecare membru al echipei. Alegerile noastre sunt sequence diagram, respectiv state diagram.

State diagram este o diagrama ce descrie comportamentul aplicatiei, tinand cont de un numar finit de stari si tranzitiile dintre acestea.

Sequence diagram este o diagrama ce presupune dispunerea interactiunii intre diferitele etape ale aplicatiei, aranjate intr-o secventa cronologica.

2.1 Sequence Diagram (Iclodean)

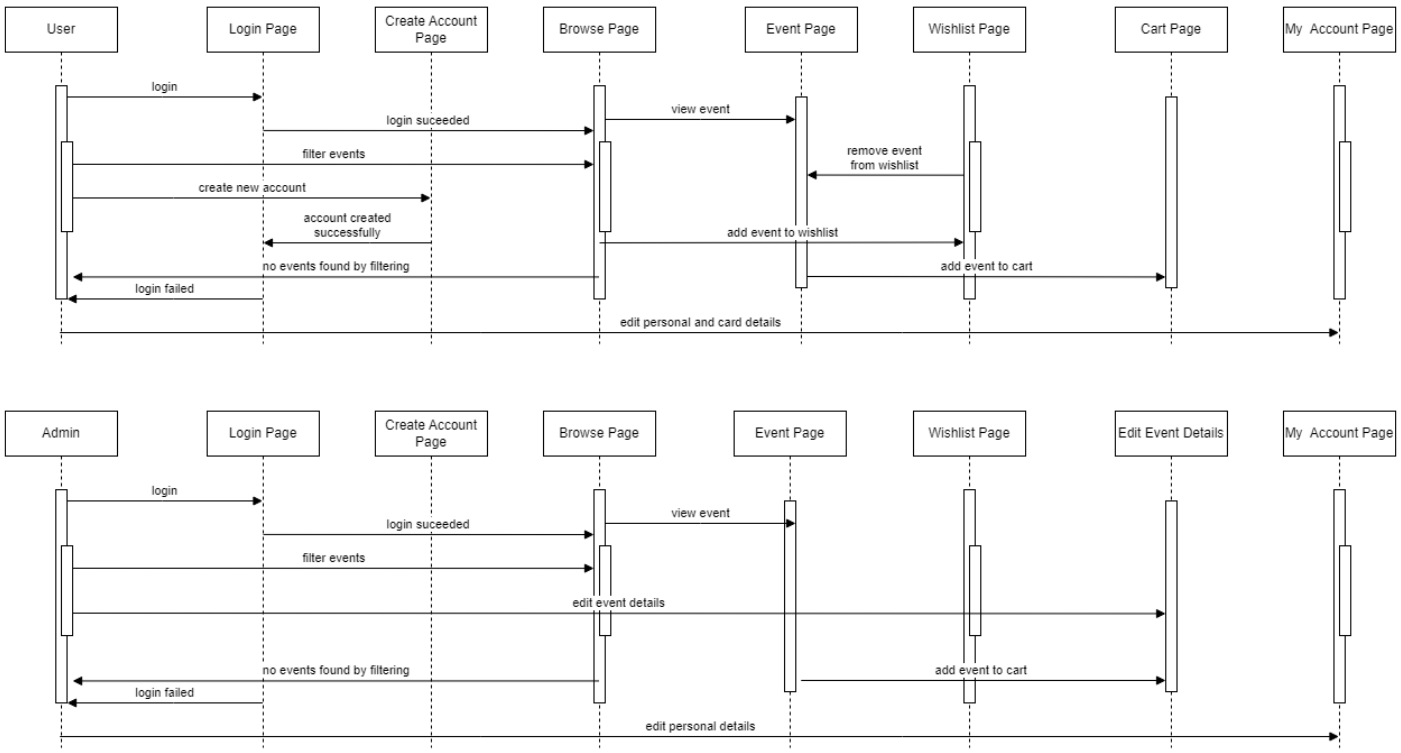

Figura 2: Sequence Diagram

2.2 State Diagram (Graur)

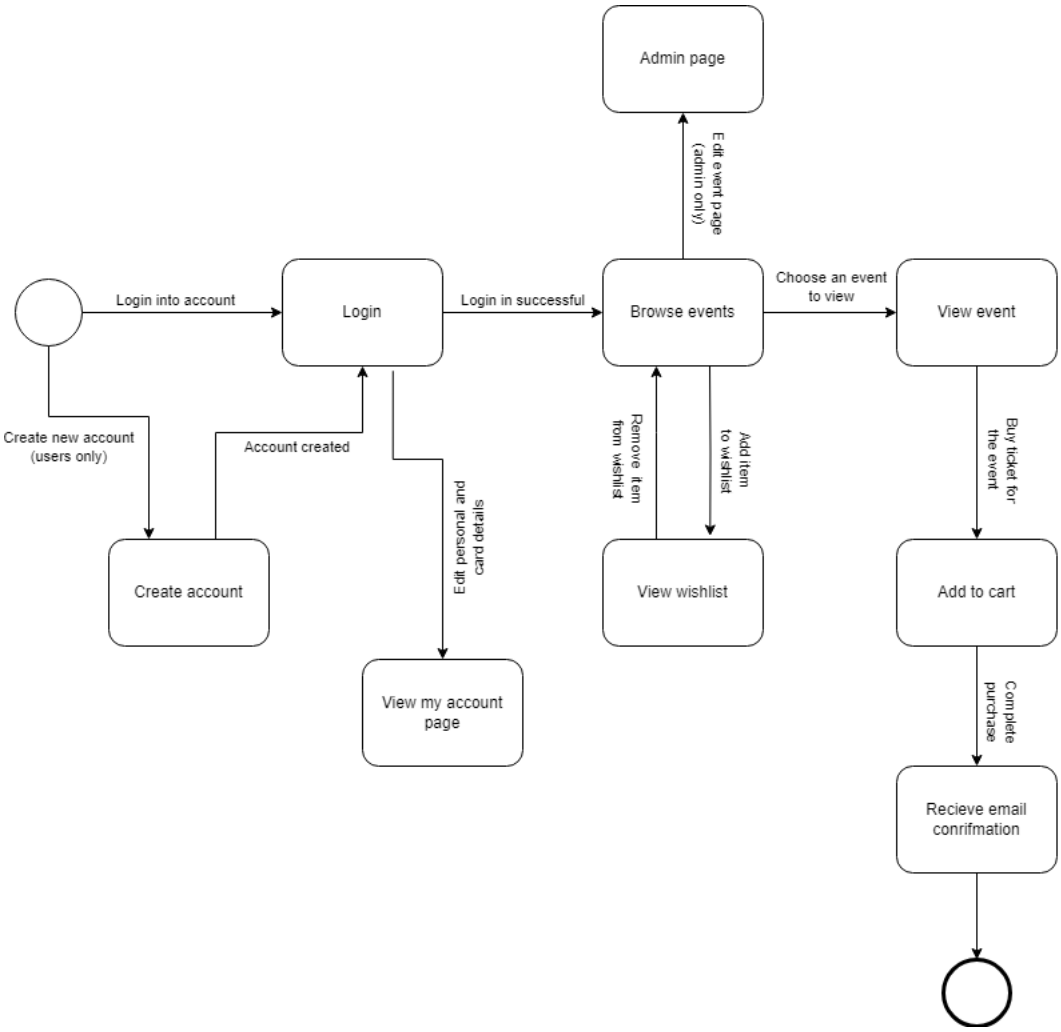

Figura 3: State Diagram

3 Design patterns

3.1 Controller-Service-Repository Pattern (Iclodean)

3.1.1 Descriere

Controller-Service-Repository Pattern este un model arhitectural des folosit in dezvoltarea aplicatiilor web, unde se doreste o separare puternica si clara intre toate nivelurile aplicatiei. Acest design pattern este asociat partea de backend sau server-side si are scopul de a oferi modularitate codului, dar, totodata, si de a fi usor de intretinut

Controller-ele sunt responsabile pentru gestionarea cererilor primite de la interfata cu utilizatorul (UI) sau sistemele externe. Actioneaza ca intermediari intre UI si business logic-ul aplicatiei. Acestea interpreteaza intrarea utilizatorului, invoca business logic-ul adecvat (Service) si pregatesc raspunsul pentru a fi trimis inapoi catre UI sau sisteme externe. Spre exemplu, intr-o aplicatie web, un controller poate gestiona solicitarile HTTP, poate extrage parametri si poate delega procesarea unui serviciu.

Service-urile incapsuleaza business logic-ul aplicatiei sau functionalitatea specifica aplicatiei. Ele efectueaza prelucrarea propriu-zisa necesara pentru a indeplini solicitarile initiate de operatori. Astfel, in aceasta etapa regasim maparea de la modelele de intrare la instantele propriu-zise ale acestora. Service-urile abstrag si incapsuleaza operatiuni de afaceri complexe, facandu-le reutilizabile in diferite parti ale aplicatiei. Acestia pot interactiona cu unul sau mai multe depozite pentru a accesa si manipula datele. Spre exemplu, un serviciu poate gestiona autentificarea utilizatorilor, gestiona tranzactii sau poate efectua calcule complexe.

Repository-urile sunt responsabile pentru accesul si stocarea datelor. Ele retrag detaliile despre modul in care datele sunt preluate, stocate si gestionate. Repository-urile ofera o interfata curata pentru accesul la date, ascund detaliile de stocare a datelor subiacente (de exemplu, baze de date). Acestea includ adesea metode de interogare, salvare, actualizare si stergere a entitatilor. Spre exemplu, un depozit poate oferi metode de preluare si stocare a datelor legate de o anumita entitate, cum ar fi preluarea informatiilor despre utilizator dintr-o baza de date.

3.1.2 Implementare

Intrucat implementarea celor celor 3 layere este mai mult sau mai putin identica pentru toate modelele, voi prezenta in continuare implementarea pentru modelul Admin.

Implementarea controller-ului foloseste o instanta a unui service corespunzator, iar mai apoi implementeaza metode precum GET, PUT, POST sau DELETE.

```
1  [ApiController]
2  [Route("admins")]
3  public class AdminController : ControllerBase
4  {
5      private readonly AdminService adminService;
6      private readonly ILogger<AdminController> _logger;
7
8      public AdminController(ILogger<AdminController> logger, AdminService service, IMapper mapper)
9      {
10         _logger = logger;
11         adminService = service;
12     }
13
14     [HttpGet]
15     public IEnumerable<Admin> Get([FromQuery] QueryParameters parameters)
16     {
17         return adminService.GetAllAdmins(parameters);
18     }
19
```

```
    // resutul metodelor...
}
```

Implementarea service-ului foloseste o instanta a unui repository corespunzator, iar mai apoi implementeaza metode unde are loc si maparea folosind AutoMapper de la obiectele In la obiectele propriu-zise.

```
public class AdminService
{
    private IAdminRepository _repository;
    private readonly IMapper _mapper;

    public AdminService(IAdminRepository repository, IMapper mapper)
    {
        _repository = repository;
        _mapper = mapper;
    }

    public List<Admin> GetAllAdmins(QueryParameters parameters)
    {
        return _repository.GetAllAdmins(parameters);
    }

    // // resutul metodelor...
}
```

Implementarea repository-ului consta in implementarea unei interfere specifica repository-ului, unde se afla toate metodele pe care acesta le implementeaza.

```
public interface IAdminRepository
{
    public List<Admin> GetAllAdmins(QueryParameters parameters);
    public ErrorOr<Admin> GetAdminById(int id);
    public int InsertAdmin(Admin admin);
    public bool UpsertAdmin(int id, Admin admin);
    public bool DeleteAdmin(int id);
    public bool DeleteAdmins(List<int> ids);
}
```

```
public class AdminRepository : IAdminRepository
{
    private readonly SqlConnection _sqlConnection;
    private readonly Configurations _configuration;

    public AdminRepository(IOptions<Configurations> configurations)
    {
        _configuration = configurations.Value;
        _sqlConnection = new SqlConnection(_configuration.ConnectionString);
    }

    public List<Admin> GetAllAdmins(QueryParameters parameters)
    {
        return _sqlConnection.GetAll<Admin>().ToList();
    }
```

```
16
17      // // resutul metodelor...
18  }
```

3.2 Decorator (Graur)

3.2.1 Descriere

Decorator Pattern este un model structural care permite adaugarea de comportament unui obiect individual, fara a afecta restul obiectelor din clasa respectiva.

De fapt, scopul acestui design pattern este de a permite atasarea de responsabilitati adidionale unui obiect in mod dinamic.

Principalele componente ale unui Decorator Pattern sunt urmatoarele:

Component defineste interfata pentru obiectele ce au responsabilitati adaugate lor in mod dinamic.

Concrete Component implementeaza interfata Component definita anterior.

Decorator are rolul de a mentine o referinta la un obiect de tipul Component, putand fi astfel o clasa abstracta sau o interfata.

Concrete Decorator sunt clasele care adauga functionalitate componentei.

3.2.2 Implementare

In continuare sunt prezentate cateva exemple de implementare unde au fost adaugate functionalitati aditionale in mod dinamic, respectandu-se astfel design pattern-ul Decorator.

```
1  locationFilterSelect.addEventListener('change', function() {
2    // Handle the selected city
3    selectedCity = this.value;
4    filterEvents();
5
6  });
7
8  venueFilterSelect.addEventListener('change', function() {
9    // Handle the selected venue
10   selectedVenue = this.value;
11   filterEvents();
12
13 });
14
15 searchFilter.addEventListener('input', function() {
16   searchText = searchFilter.value;
17   filterEvents();
18 });
19
20 const startDate = document.getElementById('start-date-filter');
21 const endDate = document.getElementById('end-date-filter');
22
23 startDate.addEventListener('change', function() {
24   filterEvents();
```

```
25  });
26
27  endDate.addEventListener('change', function() {
28    filterEvents();
29  });
```

4 Limbaje de programare, framework-uri si baze de date

4.1 Limbaje de programare

Alegerea limbajelor de programare utilizate in implementarea acestui proiect se datoreaza exclusiv experientei pe care o avem cu acestea de la locurile de munca.

Astfel, pentru implementarea partii de Backend am ales sa utilizam exclusiv limbajul C#, iar pentru partea de Frontend am ales HTML, CSS si JavaScript.

4.1.1 Backend

Implementrea partii de Backend a aplicatiei a fost realizata folosind limbajul C#, impreuna cu librariile prezentate in sectiunile urmatoare. Codul sursa pentru aceasta poate fi regasit in folderul TicketPlatformBackend.API din folderul TicketPlatform.

4.1.2 Frontend

Implementrea partii de Frontend a aplicatiei a fost realizata folosind limbajele HTML, CSS si JavaScript. Codul sursa pentru aceasta poate fi regasit in folderul TicketPlatformFrontend din folderul TicketPlatform.

4.2 Framework-uri si librarii

Odata cu noile versiuni de .NET Core, au inceput sa apara o multitudine de framework-uri si librarii cu o utilitate semnificativa in ceea ce priveste scrierea de clean code si portabilitate. Astfel, am ales sa utilizam o serie de astfel de pachete, care ni s-au parut relevante temei, dar si care au rezolvat unele inconveniente ce ar fi ingreunat procesul de relaizare al aplicatiei.

4.2.1 .NET Core 8

Principalul framework utilizat in implementarea acestui proiect este .NET Core 8. Acesta este un framework folosit pentru construirea de aplicatii compatibile cu sisteme precum Mac, Windows si Linux. Principalul avantaj al sau este portabilitatea pe care o ofera, aplicatiile implementate fiind compatibile cu toate sistemele de operare mentionate anterior.

4.2.2 Dapper si Dapper.Contrib

Dapper este o bibliotecă NuGet care imbunatateste conexiunile ADO.NET prin metode de extensie pe instanța DbConnection utilizata. Acesta oferă un API simplă și eficient pentru invocarea de statement-uri SQL, cu suport atât pentru accesul la date sincron, cât și asincron.

Dapper Contrib inglobeaza cateva dintre cele mai des utilizate helper methods pentru manipularea unei bazei de date, precum insert, get, update si delete.

4.2.3 AutoMapper

AutoMapper este o librarie ce ajuta la evitarea maparii manuale a obiectelor dintre modelele existente si baza de date. Astfel, nu mai este nevoie de scrierea codului prin care atribuim fiecarei proprietati dintr-un model coloana corespunzatoare dintr-o tabela a bazei de date.

4.2.4 ErrorOr

ErrorOr este un package ce simplifica modul in care apar diverse erori, realizand o reuniune intre o posibila eroare si un rezultat.

In contextul actual, ErrorOr este folosit pentru a evita anumite errori in momentul realizarii unui request HTTP, fiind capabil ca in cazul unei erori sa furnizeze prin intermediul respone body-ului erorea aparuta si posibile detalii ale acesteia in format JSON. Astfel, eroarea poate fi mult mai usor manipulata in partea de Frontend a aplicatiei.

4.2.5 FluentValidation

FluentValidation este o librarie .NET ce permite crearea de reglui de validare pentru datele primite prin intermediul unui request body. Libraria permite crearea unor clase specifice fiecarui model in care sa se defineasca unele reguli pentru datele introduse, spre exemplu anumite lungimi de string-uri, string-uri nenule si altele.

4.2.6 Swashbuckle.AspNetCore

Swagger este o unealta ce faciliteaza construirea de API-uri folosind .NET Core, oferind o interfata ce permite vizualizarea si testarea cu usurinta a tuturor operatiilor, direct din controllere si modele.

4.3 Baze de date

Alegerea bazei de date folosite in contextul proiectului nostru a fost bazata pe integrarea cu limbajul de programare folosit pentru partea de Backend. Astfel, am considerat ca cea mai buna alegere pentru acest lucru este sa utilizam MSSQL datorita integrarii pe care o are cu .NET prin librari precum ADO.NET sau Dapper (inclusiv Dapper Contrib).

4.3.1 Microsoft SQL Server

Microsoft SQL Server este o baza de date relationala dezvoltata de Microsoft. Sintaxa folosita de aceasta baza de date este extrem de similara cu sintaxa folosita de MySQL, diferentele fiind extrem de mici.

Aditional, pe langa aplicatia SQL Server Management Studio folosita pentru a administra baza de date, am utilizat si SQL Server Configuration Manager pentru a administra serverele (server-ul local in cazul de fata).

Astfel, a fost posibil ca realizarea conexiunii cu baza de date sa fie realizata exclusiv printr-un Connection String, localizat in configuriarile aplicatiei. Datorita acestei abordari, facotrul portabilitatea aplicatiei este mai crescut, iar modificarile ce tin de baza de date utilizata sunt relativ usor de realizat.

Totodata, aboradarea folosita pentru conexiunea cu baza de date faciliteaza si o eventuala folosire a unei baze de date in cloud, pe o platforma precum Microsoft Azure.

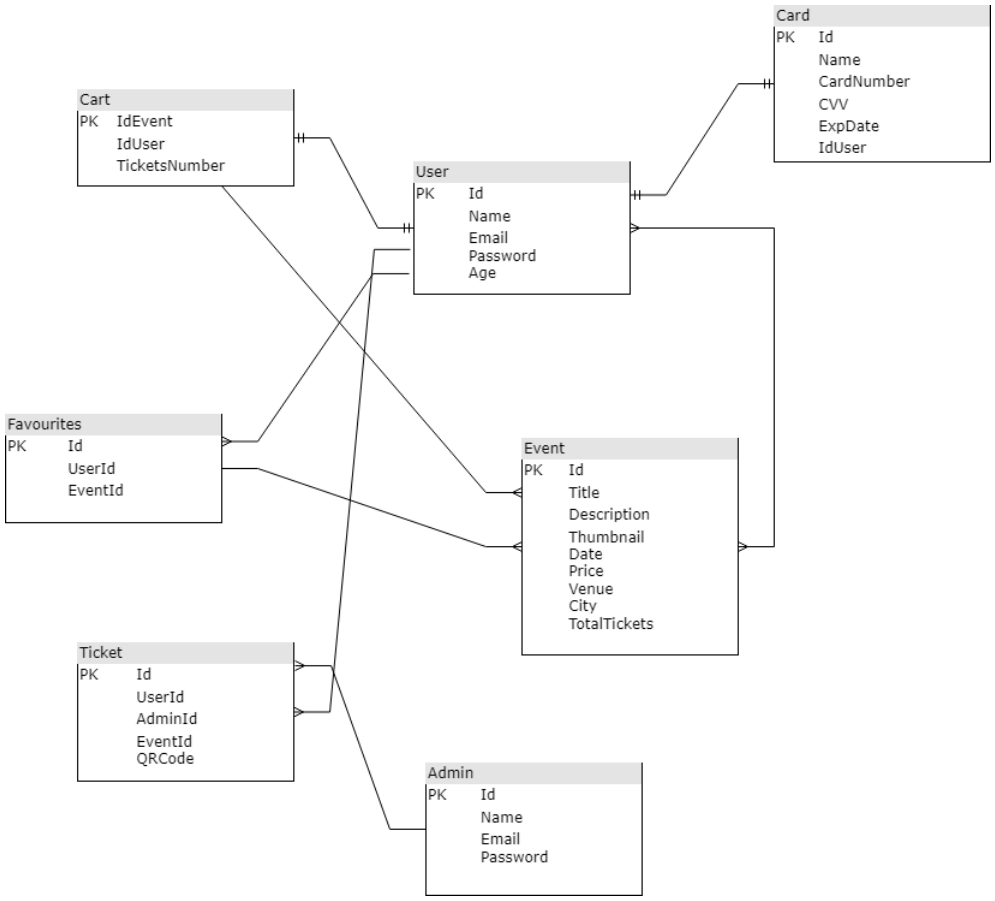

Figura 4: Diagrama bazei de date

5 Readme: Utilizarea aplicatiei

Utilizarea aplicatiei se realizeaza intr-un mod simplu si intuitiv, interfata nefiind incarcta cu prea multe butoane si fiind dispusa intr-un mod user-friendly.

La accesarea site-ului, utilizatorul va trebui sa se logheze pentru a putea continua. La logare acesta este nevoit sa introduca adresa de email a contului impreuna cu parola asociata. In cazul in care contul nu exista in baza de date, acesta are posibilitatea de a-si creea un cont pe loc, dupa care poate reveni la logare.

La logarea unui admin pe site, acesta va trebui sa introduca aceleasi credentiale, insa sa bifeze si casuta "Admin". Acest proces functioneaza daca utilizatorul are deja un cont de admin. Conturile de admin nu se pot crea din interfata, ci doar din baza de date.

Odata logat pe site, utilizatorul poate sa navigheze prin evenimentele disponibile, sa le sorteze dupa orar, locatie sau data evenimentului, dar si sa le caute dupa nume. Acesta poate vedea detalii despre un eveniment, aduga sau sterge evenimentul din wishlist si cumpara bilete pentru acel evenmient. Odata cu cumpararea biletului, utilizatorul va primi un email cu confirmarea achizitiei.

Pe langa vizualizarea evenientelor, utilizatorul poate sa-si editeze datele personale si sa adauge un card de credit.

Un admin are in plus optiunea de a adauga si edita evenimente.

6 Bibliografie

[1] The Catalog of Design Patterns
[2] Swagger Documentation
[3] Object-Oriented Software Engineering, Chapter 6: Using Design Patterns