UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și tehnologia informației
SPECIALIZAREA: Tehnologia informației

Aplicație pentru colectarea și analiza personalizată a datelor

LUCRARE DE DIPLOMĂ

Coordonator științific
Ș.l.dr.ing. Cristian Nicolae BUȚINCU

Absolvent

Fechet Ionela-Paula

Iași, 2021

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII
LUCRĂRII DE DIPLOMĂ

Subsemnatul(a) FECHET IONELA - PAULA,
legitimat(ă) cu CI seria MZ nr. 512615, CNP 2980624225900
autorul lucrării Aplicație pentru colectarea și analiza personalizată a datelor

elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi" din Iași, sesiunea iulie a anului universitar 2020 - 2021, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi" din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data
6.07.2021

Semnătura
Fechet

Table of Contents

Aplicație pentru colectarea și analiza personalizată datelor

Ionela-Paula Fechet

Rezumat

Există o diversitate de aplicații pentru a monitoriza ceea ce își dorește utilizatorul în mod particular, dar aplicațiile care pot fi personalizate și care nu se specializează doar pe monitorizarea unui singur lucru sunt puține. Această aplicație pentru colectarea și analiza datelor vin în ajutorul celor care nu doresc să aibă atât de multe aplicații de monitorizare, ci doresc ca toate să fie unite într-o singură aplicație. Acesta nu este singurul scop al aplicației, alt scop este acela de a oferi utilizatorului ocazia de a observa cât de corelate sunt datele. Aplicația conține servicii de autentificare (login, signup, forget password) împreună cu posibilitatea creărilor de categorii în care se pot înregistra datele zilnice, sau cât se poate de des pentru a putea avea rezultate corecte. Categoriile create pot fi, spre exemplu, numărul de cești de cafea băute pe zi, orele de somn pe noapte, nivelul intensității unor dureri, nivelul de fericire, etc. Pe graficul generat în urma adăugărilor de înregistrări, utilizatorul poate da click și să adauge un nou punct/înregistrare în grafic. În caz că are unele date salvate în format CSV și le dorește în aplicație, are opțiunea de a face import, dar și export. În cazul în care a greșit adăugarea unui punct, el poate face "undo".

Flexibilitatea este oferită de customizarea unei categorii. Utilizatorul poate alege unitatea de măsură a diferitelor seturi de date, culoarea graficului, limita inferioară, limita superioară și poate introduce înregistrări pentru categorii pe o anumită dată. Vizualizarea datelor se poate face prin intermediul unui grafic, iar corelarea datelor poate fi calculată folosind algoritmul de corelație Pearson. Acest coeficient care va rezultă vă spune multe despre datele celor două grafice care se vor a fi comparate. Utilizatorul poate introduce diverse categorii de date pe care dorește să le monitorizeze și poate cu ușurință să vadă care este relația/legătură dintre oricare 2 seturi de date. Să luăm de exemplu cantitatea de apă băută în fiecare zi și durerea de cap monitorizată în aceleași zile în care au fost înregistrate datele despre cantitatea de apă băută.

Coeficientul de corelație va avea valori între [-1, 1]. Cu cât rezultatul este mai apropiat de –1, cu atât datele din cele două categorii alese au o legătură puternic negativă, în sensul că dacă un setul de date A crește, setul de date B descrește. Dacă rezultatul este 0, nu există nicio relație între cele două seturi de date. Dacă rezulatul este 1, există o relație puternic pozitivă, în sensul că dacă setul de date A crește, crește și setul de date B. Corelația pune în evidență relațiile ce există între două serii de observații considerate simultan. Dacă ne interesează doar existența unei legături între cele două variabile, se calculează coeficientul de corelație. Un coeficient de corelație mare indică o legătură puternică.

Introducere

Motivația mea pentru crearea acestei aplicații are la bază creșterea conștiinței populare asupra importanței și valorii datelor personale și a analizei lor. Odată cu creșterea în popularitate a dispozitivelor inteligente cum ar fi smart watch care iți oferă informații despre datele tale biometrice și creșterea în popularitatea a conceptelor din aria well being, oamenii au devenit mai interesați de propriile date personale și de îmbunătățirea propriei lor vieți.

Căutând aplicații care să satisfacă această nevoie pe deplin, am fost dezamăgită să aflu că marea majoritate sunt fie de o calitate inferioară fie oferă foarte puțină flexibilitate. O astfel de aplicație populară poate să înregistreze date doar despre anumite lucruri, spre exemplu monitorizarea apei.

Am dorit o aplicație care să îmi ofere flexibilitate pentru a înregistra și analiza orice câmp de date folositoare utilizatorului, pornind de la premisa că utilizatorul este o persoană care știe ce date are nevoie și știe să analizeze cu asistență minimă. Înainte de a începe dezvoltarea aplicației, am stabilit o listă de cerințe pe care ea ar trebui să le îndeplinească.

Lista cu cerințe este următoarea:

- utilizatorul să aibă posibilitatea de a înregistra date în orice unitate de măsură dorită de el. Utilizatorul ar trebui, de asemenea, să aibă posibilitatea de a defini un interval pentru valorile permise pentru toate înregistrările. utilizatorul poate vedea datele intr-un format prietenos, de tipul graficelor liniare.

- a oferi posibilitatea utilizatorului de a suprapune graficele și de a facilita vizualizarea lor. Utilizatorul să poată corela datele a două grafice și să afle coeficientul de corelație Pearson.

- utilizatorul poate să exporte datele unui grafic, dar și să facă import. Importul este folositor pentru utilizatori care vor să înregistreze date biometrice. Funcționalitatea de export este foarte folositoare pentru a oferi utilizatorului posibilitatea de a vizualiza datele în alte aplicații specializate, ca exemplu, una dintre ele fiind excel.

Capitolul 1. Fundamentarea teoretică a temei

1.1. Baza de date - Microsoft MySQL Server Management Studio

SQL Server Management Studio(SSMS) este o aplicație software lansată pentru prima dată cu Microsoft SQL Server 2005, care este utilizată pentru configurarea, gestionarea și administrarea tuturor componentelor din Microsoft SQL Server.

Modelul Entitate-Relație folosit în aplicație este următorul:

Tabelele folosite sunt:

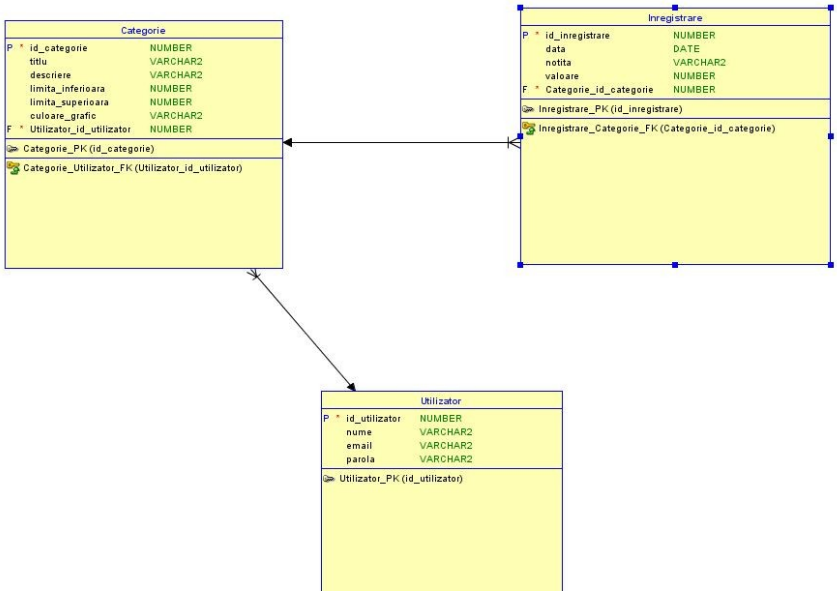

Figure 1. Modelul Entitate-Relatie

- Tabela ***Categorie*** conține informații despre id categorie, titltu, descriere, limita inferioară, limita superioară, culoarea graficului, id-ul utilizatorului care a creat categoria.
- Tabela ***Înregistrare*** conține informații despre id înregistrare, data, valoarea și id-ul categoriei de care informația aparține.
- Tabela ***Utilizator*** conține informații despre id utilizator, nume, email, parola care este de fapt incriptată.

1.2. Front-end – Angular

Angular (numit și "**Angular 2+**" sau "**Angular v2 și mai nou**") este o platformă de dezvoltare web open source bazată pe limbajul TypeScript. Proiectul este dezvoltat de Echipa Angular de la Google și de o comunitate de utilizatori individuali și companii. Angular este o

rescriere completă, de către aceeași echipă, a frameworkului AngularJS.

Inițial, versiunea rescrisă a AngularJS a fost numită "Angular 2" de echipă, însă acest lucru a provocat confuzie printre dezvoltatori. De aceea, echipa a anunțat că "AngularJS" se va referi la versiunile 1.X și "Angular" (fără "JS") la versiunile 2 și ulterioare. [1]

1.2.1. Typescript

TypeScript este un limbaj de programare dezvoltat și întreținut de Microsoft. Este un superset strict sintactic de JavaScript. TypeScript este conceput pentru dezvoltarea de aplicații mari și transcompilează în JavaScript.

TypeScript poate fi utilizat pentru a dezvolta aplicații JavaScript atât pentru executarea pe partea de client, cât și pe cea a serverului (că în cazul Node.js sau Deno).

Compilatorul TypeScript este el însuși scris în TypeScript și compilat în JavaScript. Este licențiat sub licența Apache 2.0. TypeScript este inclus că limbaj de programare de primă clasă în Microsoft Visual Studio 2013 Update 2 și ulterior, alături de C # și alte limbaje Microsoft. O extensie oficială permite Visual Studio 2012 să accepte și TypeScript. Anders Hejlsberg , arhitect principal al C # și creator Delphi și Turbo Pascal, a lucrat la dezvoltarea TypeScript.

TypeScript a fost făcut public pentru prima dată în octombrie 2012 (la versiunea 0.8), după doi ani de dezvoltare internă la Microsoft. La scurt timp după anunț, Miguel de Icaza a lăudat limbajul în sine, dar a criticat lipsa suportului IDE matur în afară de Microsoft Visual Studio, care nu era disponibil pe Linux și OS X în acel moment. Începând cu aprilie 2021 există suport în alte IDE și editoare de text, inclusiv Emacs, Vim, Webstorm, Atom și propriul cod Visual Studio al Microsoft.

În iulie 2014, echipa de dezvoltare a anunțat un nou compilator TypeScript, pretinzând câștiguri de performanță de 5 ori mai mare. În același timp, codul sursă, care a fost găzduit inițial pe CodePlex, a fost mutat în GitHub.

La 22 septembrie 2016, a fost lansat TypeScript 2.0; a introdus mai multe caracteristici, inclusiv posibilitatea programatorilor de a împiedica opțional variabilelor să li se atribuie valori null, uneori denumită greșeala de miliarde de dolari.

Caracteristici

TypeScript este o extensie de limbaj care adaugă caracteristici la ECMAScript 6. Funcțiile suplimentare includ:

- Adnotarea și verificarea tipului în timpul compilării
- Inferența de tip (se referă la detectarea automată a tipului unei expresii într-un limbaj formal. Acestea includ limbaje de programare și sisteme de tip matematic, dar și limbaje naturale în unele ramuri ale informaticii și lingvisticii)
- Ștergerea tipurilor
- Interfețe
- Tipuri enumerate
- Generice

- Namespaces (este un set de nume care sunt utilizate pentru a identifica și a se referi la obiecte de diferite tipuri. Un namespace asigură faptul că un set dat de obiecte au nume unice, astfel încât să poată fi identificate cu ușurință)
- Tuples

Din punct de vedere sintactic, TypeScript este foarte asemănător cu JScript. .NET , o altă implementare Microsoft a standardului de limbaj ECMA-262 a adăugat suport pentru tastarea statică și caracteristicile limbajului orientat către obiecte clasice, cum ar fi clasele, moștenirea, interfețele și spațiile de nume.

1.3. Back-end - .NET

.NET este un framework pentru Windows, casa mai multor tehnologii cum sunt: ASP.NET WebForms, ASP.NET MVC, WebAPI, Azure, Xamarin și multe altele. NET este sinonim cu dezvoltarea de aplicații pe toate platformele Windows și nu doar atât.

În ultimul an, Microsoft a reușit să depășească realmente granițele platformei sale prin .NET Core – o versiune a framework-ului .NET care oferă posibilitatea scrierii de aplicații pe mai multe platforme. Acum putem scrie aplicații cu excelentul Visual Studio 2017 (care este de fapt versiunea 15) pentru platformele: Windows, Linux (diferitele distribuții: Red Hat, Fedora, Debian, Ubuntu, Android etc.), Mac OS X.[2]

1.3.1. Avantaje și dezavantaje .NET

Avantaje:

- Bazat pe OOP
- System caching bun
- Visual studio
- Cross-platform
- Ușor de menținut

Dezavantaje:

- Probleme legate de object-relațional
- Costul
- Probleme de stabilitate pentru noi lansări

1.3.2. Entity Framework Core

Entity Framework (EF) Core este o versiune ușoară, extensibilă, open source și multi-platformă a populare tehnologii de acces la date Entity Framework. [3]

EF Core poate servi ca un object-relational mapper (O / RM), care:

- Permite dezvoltatorilor .NET să lucreze cu o bază de date folosind obiecte .NET.
- Elimină necesitatea majorității codului de acces la date care de obicei trebuie scris.

Cu EF Core, accesul la date se realizează utilizând un model. Un model este alcătuit din clase de entități și un obiect context care reprezintă o sesiune cu baza de date. Obiectul context permite interogarea și salvarea datelor.

EF susține următoarele abordări de dezvoltare a modelului:

- Generarea unui model dintr-o bază de date existentă.
- Scrierea codului manual a unui model care să se potrivească cu baza de date.
- Odată creat un model, utilizați EF Migrations pentru a crea o bază de date din model. Migrațiile permit dezvoltarea bazei de date pe măsură ce modelul se schimbă.

1.3.3. Crearea proiectului în .NET

La crearea aplicației, am început cu configurarea backendului și crearea serviciilor, după care am creat proiectul în Angular, pentru a lega interfața de backend.

Pentru început, am adăugat swaggerul în configurarea aplicației. Swagger (OpenAPI) este o specificație lingvistică-agnostică pentru descrierea API-urilor REST. Permite atât computerelor, cât și oamenilor să înțeleagă capacitățile unei API-uri REST fără acces direct la codul sursă.

Principalele sale obiective sunt:

- Minimizarea cantității de muncă necesară pentru conectarea serviciilor decuplate.
- Reducerea timpului necesar pentru a documenta cu exactitate un serviciu.

1.3.4. Swagger UI

Swagger UI oferă o interfață unde se află informații despre serviciu, utilizând specificația OpenAPI. Atât Swashbuckle cât și NSwag includ o versiune încorporată a Swagger UI, astfel încât să poată fi găzduită în aplicația ASP.NET Core folosind un apel de înregistrare middleware. [4]

UI-ul web arată astfel:

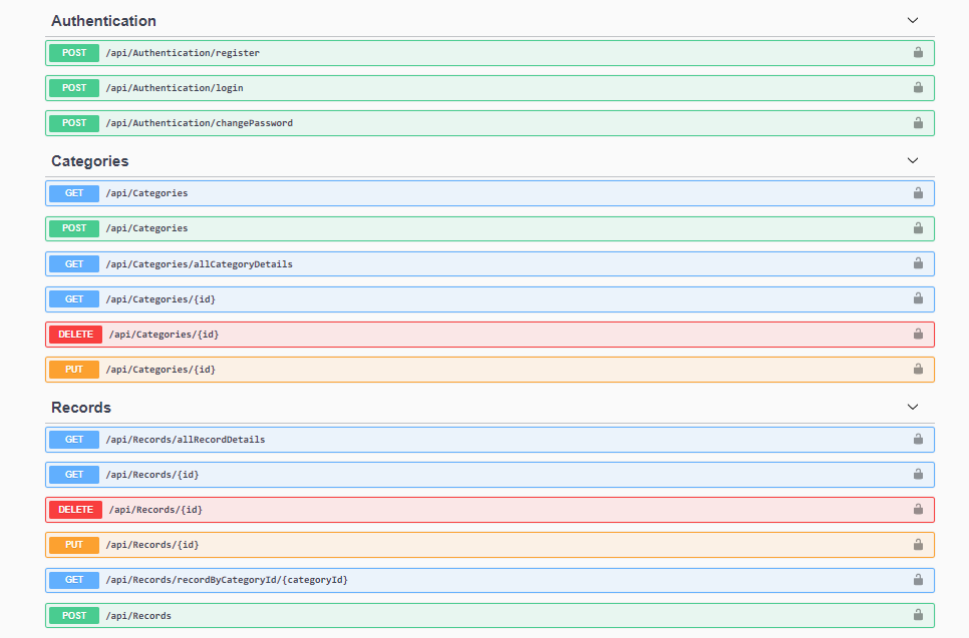

Figure 1. Swagger UI al aplicației

OpenApi vs Swagger

Proiectul Swagger a fost donat Inițiativei OpenAPI în 2015 și de atunci a fost denumit OpenAPI. Ambele nume sunt utilizate în mod interschimbabil. Cu toate acestea, „OpenAPI" se referă la specificație. „Swagger" se referă la familia de produse open-source și comerciale de la SmartBear care funcționează cu specificația OpenAPI. Produsele open-source ulterioare, precum OpenAPIGenerator, intră și sub numele de familie Swagger, în ciuda faptului că nu au fost lansate de SmartBear.

Pe scurt:

- OpenAPI este o specificație.
- Swagger este unealtă care folosește specificația OpenAPI

Adăugarea si configurarea unui Swagger middleware

Adăugam swagger în metoda Startup.ConfigureServices ca în figura de mai jos.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddControllers();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen();
}
```

Figure 2. Adăugare Swagger în configurare

În metoda Startup.Configure, activăm middleware pentru activarea swaggerului:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseSwagger();
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });
```

Figure 3. Activare swagger

1.3.5. Crearea contextului bazei de date

Clasa principală care coordonează funcționalitatea pentru un model de date este clasa contextului bazei de date DbContext. Această clasă este derivată din Microsoft.EntityFramewrokCore.DbContext. Clasele DbContext derivate care precizează entitățile sunt incluse în modelul de date. În acest proiect, clasa este numită LookAppContext. [5]

```
namespace LookApp.Database.Models
{
    17 references
    public sealed class LookAppContext : DbContext
    {
        0 references
        public LookAppContext(DbContextOptions<LookAppContext> options) : base(options)
        {
            Database.Migrate();
        }

        5 references
        public DbSet<Category> Categories { get; set; }
        6 references
        public DbSet<Record> Records { get; set; }
        5 references
        public DbSet<User> Users { get; set; }
    }
}
```

Figure 4. Contextul bazei de date

Codul precedent creează un DbSet pentru fiecare set de entități. În terminologia EF:

- Un set de entități corespunde de obicei unui tabel de baze de date.
- O entitate corespunde unui rând din tabel.

Când baza de date este creată, EF creează tabele care au nume identice cu numele DbSet. Numele proprietăților pentru colecții sunt de obicei la plural. De exemplu, mai degrabă Categories decât Category.

1.3.6. Înregistrarea LookAppContext

ASP.NET Core include injectarea dependenței. Serviciile, precum contextul bazei de date EF, sunt înregistrate cu dependency injection în timpul pornirii aplicației.

Acelor componente care necesită aceste servicii, cum ar fi controlerele MVC, li se furnizează aceste servicii prin intermediul parametrilor constructorului. Pentru a înregistra LookAppContext ca serviciu, am deschis Startup.cs și am adăugat liniile evidențiate la ConfigureServices.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors();
    services.AddHttpContextAccessor();
    services.AddDbContext<LookAppContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddScoped<ICategoryService, CategoryService>();
```

Figure 5. Adăugarea contextului în configurări

Informațiile despre acel "DefaultConnection" găsim în appsettings.json și acolo punem string-ul de conexiune către baza de date din Microsoft Sql Server Management Studio.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Server=.\\SQLEXPRESS;Database=LookApp;Trusted_Connection=True"
  },
  "JwtOptions": {
    "Issuer": "https://localhost:44387",
    "Audience": "LookApp-api",
    "Key": "DB1E6CB9-9C4A-4C34-898A-5C0FB8299B91",
    "TokenExpirationInHours": 24
  }
}
```

Figure 6. Adăugarea string de conectare cu baza de date

1.3.7. Crearea modelelor

În folderul **Modele**, am creat clasele **Category**, **Record** si **User**.

```
using System.Collections.Generic;
namespace LookApp.Database.Models
{
   public class Category
```

```
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string UnitOfMeasure { get; set; }
        public double? LowerLimit { get; set; }
        public double? UpperLimit { get; set; }
        public string? GraphColor { get; set; }
        public ICollection<Record> Records { get; set; }

        public int CreatorId { get; set; }
        public User Creator { get; set; }
    }
}

namespace LookApp.Database.Models
{
    public class Record
    {
        public int Id { get; set; }
        public DateTime Date { get; set; }
        public string Note { get; set; }
        public double Value { get; set; }
        public int CategoryId { get; set; }
        public Category Category { get; set; }
    }
}

namespace LookApp.Database.Models
{
    public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Email { get; set; }
        public string Password { get; set; }
    }
}
```

1.3.8. Dependency injection

ASP.NET Core acceptă modelul de proiectare a software-ului de injectare a dependenței (DI), care este o tehnică pentru realizarea Inversion Of Control (IoC) între clase și dependențele acestora. [6]

O dependență este un obiect de care depinde un alt obiect.

Dependecy injection se descrie prin:

1. Utilizarea unei interfețe sau a unei clase de bază pentru a abstractiza implementarea dependenței.
2. Înregistrarea dependenței într-un container de servicii. ASP.NET Core oferă un container de servicii încorporat, IServiceProvider. Serviciile sunt de obicei înregistrate în metoda Startup.ConfigureServices.
3. ***Injectarea*** serviciului în constructorul clasei în care este utilizat. Cadrul își asumă responsabilitatea de a crea o instanță a dependenței și de a o elimina atunci când nu mai este necesară.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors();
    services.AddHttpContextAccessor();
    services.AddDbContext<LookAppContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddScoped<ICategoryService, CategoryService>();
    services.AddScoped<IRecordService, RecordService>();
    services.AddScoped<IPasswordHasher, PasswordHasher>();
    services.AddScoped<IAuthenticationService, AuthenticationService>();

    services.AddTransient<ICategoryMapper, CategoryMapper>();
    services.AddTransient<IRecordMapper, RecordMapper>();

    services.AddControllers().AddNewtonsoftJson(options =>
        options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore
    );

    services.AddSwagger();

    AddAuthentication(services);
}
```

Figure 7. Înregistrarea serviciilor și a tipului lor concret

Aplicația înregistrează serviciul ICategoryService cu tipul concret CategoryService. Metoda AddScoped înregistrează serviciul cu o durată de viață egală cu cea a procesării cererii curente.

Prin utilizarea DI, controlerul:

- Nu folosește tipul concret al serviciului, ci interfața pe care o implementează. Acest lucru face ușoară schimbarea implementării pe care controller folosește fără a modifica controllerul.
- Nu creează o instanță, ci este creată de containerul DI.

```
public class CategoriesController : ControllerBase
{
    private readonly ICategoryService _categoryService;
    private readonly ICategoryMapper _categoryMapper;

    private readonly int _currentUserId;

    0 references
    public CategoriesController(
        ICategoryService categoryService,
        ICategoryMapper categoryMapper,
        IHttpContextAccessor httpContextAccessor)
    {
        this._categoryService = categoryService;
        this._categoryMapper = categoryMapper;

        this._currentUserId = int.Parse(httpContextAccessor.HttpContext.User.FindFirst("Id").Value);
    }
```

Figure 8. Exemplu de utilizare a dependency injection

1.3.9. Autentificare și autorizare

Configurez folosirea serviciului de autentificare și autorizare în Startup.cs în metoda Configure.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseSwagger();
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthentication();
    app.UseCors(options => options.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Figure 9. Adăugarea serviciului de autentificare și autorizare in Configure

```
1 reference
private void AddAuthentication(IServiceCollection services)
{
    var jwtOptions = Configuration.GetSection("JwtOptions").Get<JwtOptions>();
    services.Configure<JwtOptions>(Configuration.GetSection("JwtOptions"));

    services
        .AddAuthentication(options =>
        {
            options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
            options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
        })
        .AddJwtBearer(options =>
        {
            options.RequireHttpsMetadata = true;
            options.SaveToken = true;
            options.TokenValidationParameters = new TokenValidationParameters
            {
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtOptions.Key)),
                ValidateIssuer = true,
                ValidateAudience = true,
                ValidIssuer = jwtOptions.Issuer,
                ValidAudience = jwtOptions.Audience
            };
        });
}
```

Figure 10. Serviciul de autentificare

În codul de mai sus am adăugat două handlere, unul pentru autentificare și unul pentru bearer. JWT este setat sa fie generat folosind un algoritm pentru generare de chei simetrice.

- RequiredHttpsMetadata setează dacă HTTPS este necesar pentru adresa sau autoritatea metadatelor. Valoarea implicită este adevărată.

- SaveToken definește dacă bearer token trebuie stocat în AuthenticationProperties după o autorizare reușită.

- ValidateIssuerSigningKey setează sau primește un boolean care controlează daca validarea cheii de securitate care a semnat securityToken este apelata. IssuerSigningKey primește sau setează un boolean pentru a controla daca emitentul o să fie validat pe periodă validării tokenului.

JSON Web Token (JWT) este un standard (RFC 7519) care definește un mod compact și autonom pentru transmiterea în siguranță a informațiilor între părți ca obiect JSON. Aceste informații pot fi verificate și de încredere deoarece sunt semnate digital. JWT-urile pot fi semnate folosind un secret (cu algoritmul HMAC) sau o pereche de chei publice / private folosind RSA sau ECDSA. [6]

Deși JWT-urile pot fi criptate pentru a fi, de asemenea, secret între părți, ne vom concentra pe token-urile semnate . Token-urile semnate pot verifica integritatea revendicărilor conținute în acesta, în timp ce jwt-urile criptate ascund aceste revendicări de la alte părți. Atunci când token-urile sunt semnate folosind perechi de chei publice/private, semnătura certifică, de asemenea, că numai partea care deține cheia privată este cea care a semnat-o.

Iată câteva scenarii în care JWT sunt utile:

- **Autorizare** : Acesta este cel mai frecvent scenariu pentru utilizarea JWT. Odată ce utilizatorul este conectat, fiecare cerere ulterioară va include JWT, permițându-i utilizatorului să acceseze rute, servicii și resurse care sunt permise cu acel token. Single Sign On este o caracteristică care utilizează pe scară largă JWT în zilele noastre, datorită cheltuielilor sale reduse și capacității sale de a fi ușor de utilizat în diferite domenii.

- **Schimb de informații** : Json web tokens reprezintă o modalitate bună de a transmite în siguranță informații între părți. Deoarece JWT-urile pot fi semnate - de exemplu, folosind perechi de chei publice / private - puteți fi siguri că expeditorii sunt cei care spun că sunt. În plus, deoarece semnătura este calculată folosind antetul, puteți verifica, de asemenea, dacă conținutul nu a fost modificat.

În forma sa compactă, JSON Web Tokens constă din trei părți separate prin puncte (.), care sunt:

- Antet
- Payload
- Semnătură

Prin urmare, un JWT arată de obicei ca următorul:xxxxx.yyyyy.zzzzz.

1.3.10. Migrări

Modelele de date se schimbă pe măsură ce caracteristicile sunt implementate: se adaugă și se elimină entități sau proprietăți noi, iar schemele bazei de date trebuie schimbate.

La un nivel ridicat,migrările funcționează în felul următor:

- Când se introduce o modificare a modelului de date, dezvoltatorul folosește instrumentele EF Core pentru a adăuga o migrare corespunzătoare care descrie actualizările necesare pentru a menține sincronizată schema bazei de date. EF Core compară modelul actual cu vechiul model pentru a determina diferențele și generează fișiere sursă de migrare; [7]

- Odată ce a fost generată o nouă migrare, aceasta poate fi aplicată unei baze de date în diferite moduri.EF Core înregistrează toate migrațiile aplicate într-un tabel special de istorie, permițându-i să știe ce migrații au fost aplicate și care nu.

Crearea primei migrări

EF Core va crea un director numit **Migrations** în proiect și va genera unele fișiere.

Add-Migration InitialCreate

Crearea bazei de date

Aplicația este gata să ruleze pe noua bază de date și nu a trebuit să scriu o singură linie SQL. Reținem că acest mod de aplicare a migrărilor este ideal pentru dezvoltarea locală, dar este mai puțin potrivit pentru mediile de producție.

Update-Database

1.4. Coeficientul de corelație *Pearson*

În statistici, Coeficientul de corelație Pearson denumit și A lui Pearson r, Coeficientul de corelație produs-moment Pearson (PPMCC) sau corelație bivariată este o statistică care măsoară liniar corelație între două variabile X și Y. [8]

Se folosesc formule de coeficient de corelație pentru a afla cât de puternică este o relație între date. Formulele returnează o valoare între -1 și 1, unde:

- 1 indică o relație puternică pozitivă.
- -1 indică o relație negativă puternică.
- Un rezultat zero nu indică nicio relație.
-

A fost dezvoltat de Karl Pearson dintr-o idee conexă introdusă de Francis Galton în anii 1880 și pentru care formula matematică a fost derivată și publicată de Auguste Bravais în 1844. Denumirea coeficientului este astfel un exemplu de Legea lui Stigler.

Definiție

Coeficienții de corelație sunt utilizați pentru a măsura cât de puternică este o relație între două variabile. Există mai multe tipuri de coeficient de corelație, dar cel mai popular este Pearson. Corelația lui Pearson este un coeficient de corelație utilizat în mod obișnuit în regresia liniară. Corelația evidențiază existența unei relații între două lucruri; totuși nu prezice cauzalitatea.

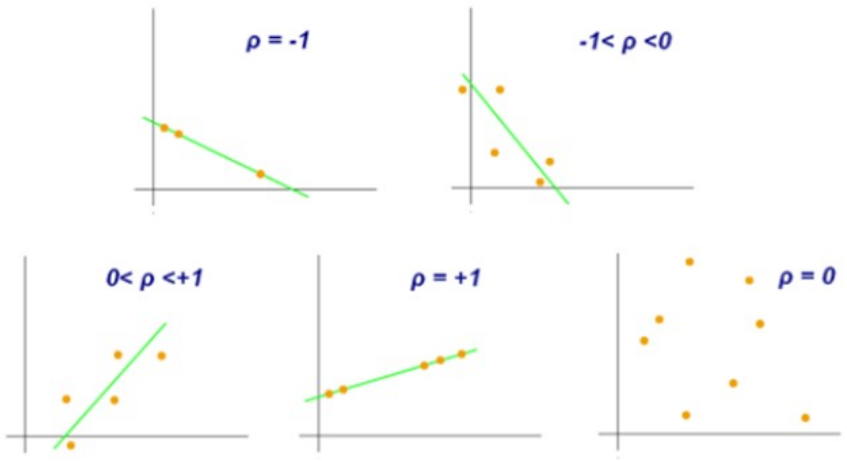

Figure 11. Exemple de diagrame scatter cu valori diferite ale coeficientului de corelație (ρ)

- Un coeficient de corelație 1 înseamnă că pentru fiecare creștere pozitivă a unei variabile, există o creștere pozitivă a unei proporții fixe în cealaltă. De exemplu, dimensiunile pantofilor cresc în corelație (aproape) perfectă cu lungimea piciorului.

- Un coeficient de corelație de -1 înseamnă că, pentru fiecare creștere pozitivă a unei variabile, există o scădere negativă a unei proporții fixe în cealaltă. De exemplu, cantitatea de gaz dintr-un rezervor scade în corelație (aproape) perfectă cu viteza.

- Zero înseamnă că, pentru fiecare creștere, nu există o creștere pozitivă sau negativă. Cele două nu sunt înrudite.

Formula de calcul a coeficientului de corelație Pierson este:

$$r_{xy} = \frac{\frac{1}{n} \cdot \sum_i (x_i - \bar{x}) \cdot (y_i - \bar{y})}{s_x \cdot s_y} = \frac{s_{xy}}{s_x \cdot s_y} \quad (1)$$

$$s_x = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}} \quad (2)$$

$$s_y = \sqrt{\frac{\sum_{i=1}^{n} (y_i - \bar{y})^2}{n}} \quad (3)$$

$$r = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{\sqrt{[\,n\Sigma x^2 - (\Sigma x)^2\,]\,[\,n\Sigma y^2 - (\Sigma y)^2\,]}} \quad (4)$$

În ecuațiile (2) și (3), sunt reprezentate formulele de calul pentru deviația standard a tuturor variabilelor x, respective y.

Ecuația (4) reprezintă o altă formulă de calcul mai simplă a coeficientului de corelație Pearson.

n = mărimea eşantionului
x = valorile individuale ale variabilei x
y = valorile individuale ale variabilei y
$\bar{x}$ = media aritmetică a tuturor valorilor x
$\bar{y}$ = media aritmetică a tuturor valorilor y
s_x = deviația standard a tuturor valorilor x
s_y = deviația standard a tuturor valorilor y

Așadar, calculul coeficientului se realizează prin împărțirea covarianței la produsul dintre deviația standard a tuturor valorilor x și deviația standard a tuturor valorilor y.

În teoria probabilității și statistică, covarianța este măsura de variație comună a două variabile aleatorii. Dacă valorile mari ale unei variabile corespund, în general, valorilor mari ale celeilalte variabile, și dacă același lucru este valabil în cazul valorilor mici (i.e. cele două variabile au comportamente similare), covarianța este pozitivă. Pe de altă parte, dacă valorile mari ale unei variabile corespund, în general, valorilor mici ale celeilalte variabile (i.e. cele două variabile au comportamente opuse), covarianța este negativă. Prin urmare, semnul covarianței arată direcția relației liniare existente între cele două variabile. Magnitudinea covarianței nu este ușor de interpretat, deoarece nu este normalizată și, prin urmare, depinde de magnitudinea variabilelor. [9]

Capitolul 2. Proiectarea aplicației

2.1. Arhitectura aplicației

Arhitectura web tratează atât structura individuală a componentelor unei aplicații, cât și interacțiunea dintre acestea. Proiectul dezvoltat este format din front-end și back-end.

În ingineria software, termenii front-end și back-end se referă la separarea preocupărilor dintre nivelul de prezentare (front-end) și nivelul de acces la date (back-end) sau infrastructura fizică sau hardware. În modelul client-server, clientul este de obicei considerat front-end-ul și serverul este de obicei considerat back-end-ul, chiar și atunci când unele lucrări de prezentare se efectuează de fapt pe serverul însuși.

2.1.1. Arhitectura back-end-ului

N-tier

Arhitectura N-tier, denumită și multitier sau multilayered este un model arhitectural foarte des întâlnit în cadrul aplicațiilor web. Aceasta se bazează pe separarea aplicației în părți sau domenii logice, denumite niveluri. Fiecare nivel rezolvă cate o problemă, nivelurile comunicând între ele.

O reprezentare simpla a arhitecturii N-tier este modelul 3-tier, care este folosit si de mine:

- Presentation Layer, UI, cu rol estetic, de prezentare a datelor
- Business Logic Layer, logica din spatele UI, back-end-ul și comunicarea dintre acestea
- Data Layer, accesul la date, modul in care sunt stocate datele

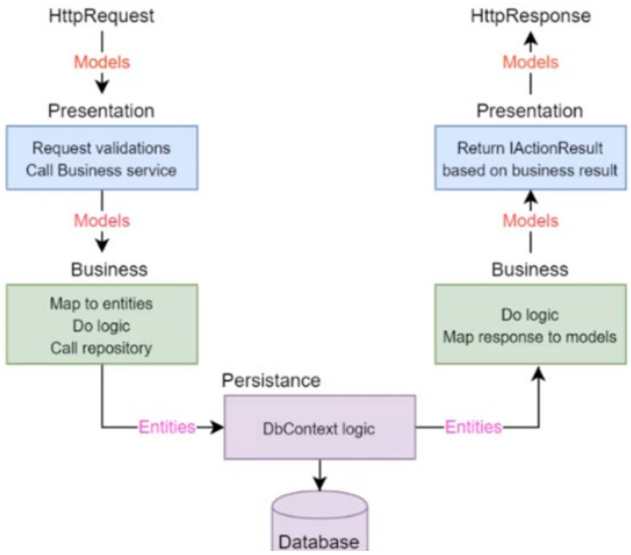

Figure 12. Arhitectura back-end

2.1.1.1. HTTP Rest API

Representational State Transfer (REST) este un stil arhitectural modern care definește modul in care interacționează serviciile web. Acesta se bazează pe cereri HTTP (e.g.: GET, POST, PUT, DELETE) pentru a manipula informația si pe media-types (e.g.: JSON) pentru a stabili cum arată informația transferată.

Un API de tip REST este un serviciu Web care permite interacțiunea prin mecanisme REST.

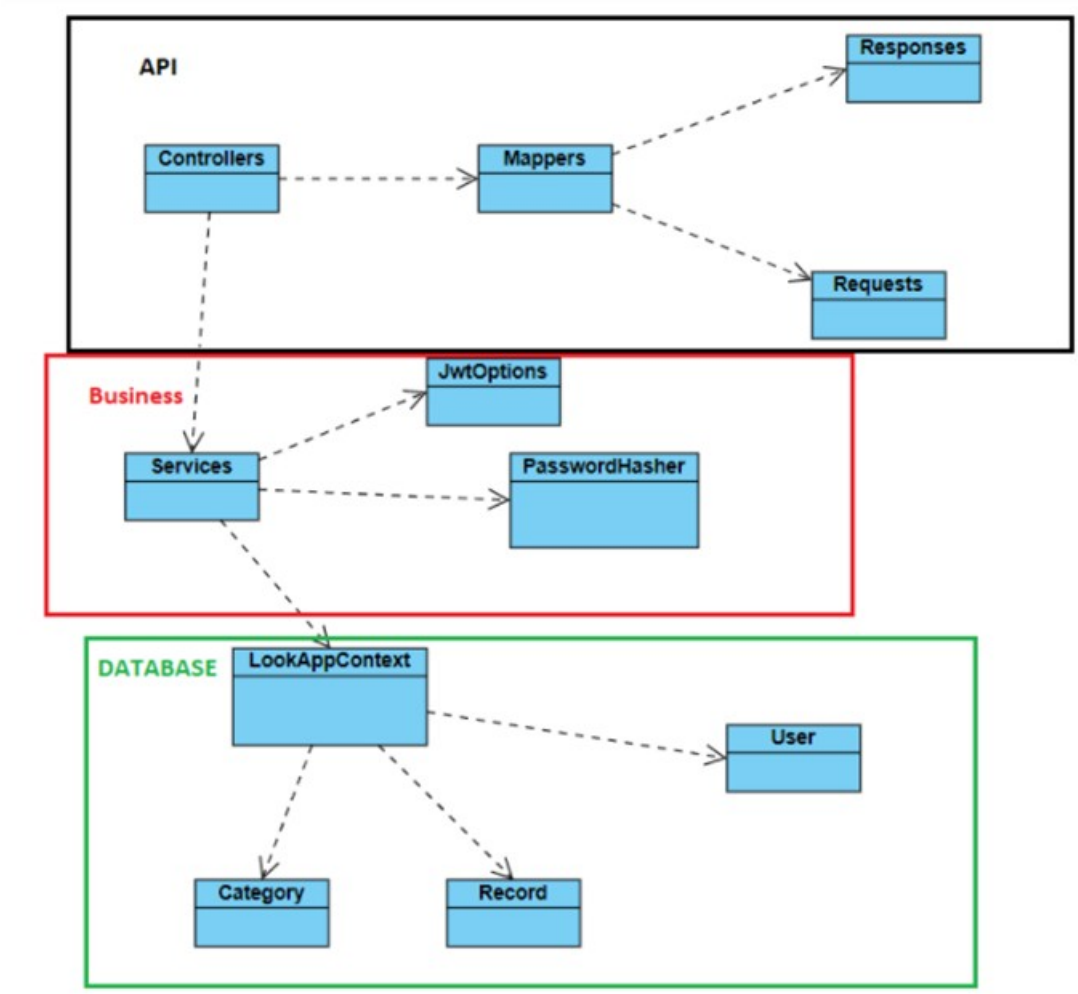

Figure 13. Diagrama UML a serverului

Aceasta este diagrama UML a claselor folosite de aplicatie, creată pe baza celor 3 niveluri: API Layer, Bussines Logic Layer si Persistance Layer.

Clasele folosite de aplicație sunt într-un număr mai mare, dar diagrama UML urmărește să prezinte mai sintetizat logica aplicației.

La nivelul API, există trei controlere: AuthenticationController, CategoriesController și RecordsController. Toate aceste controlere folosesc servicii din Business Layer. De exemplu, AuthenticationController folosește AuthenticationService care implementează interfața IAuthenticationService. De asemenea, CategoriesController folosește serviciul din CategoryService, care implementează interfața ICategoryService, dar în plus, folosește și CategoryMapper. Acest CaetgoryMapper ne ajută să avem flexibilitate în ceea ce privește proprietățile pe care dorim să le folosim când creem un request sau un response. RecordsController folosește RecordService, dar și RecordMapper.

2.1.1.2. API Layer

API este acronimul pentru Application Programming Interface, care este un intermediar software care permite celor două aplicații să vorbească între ele.

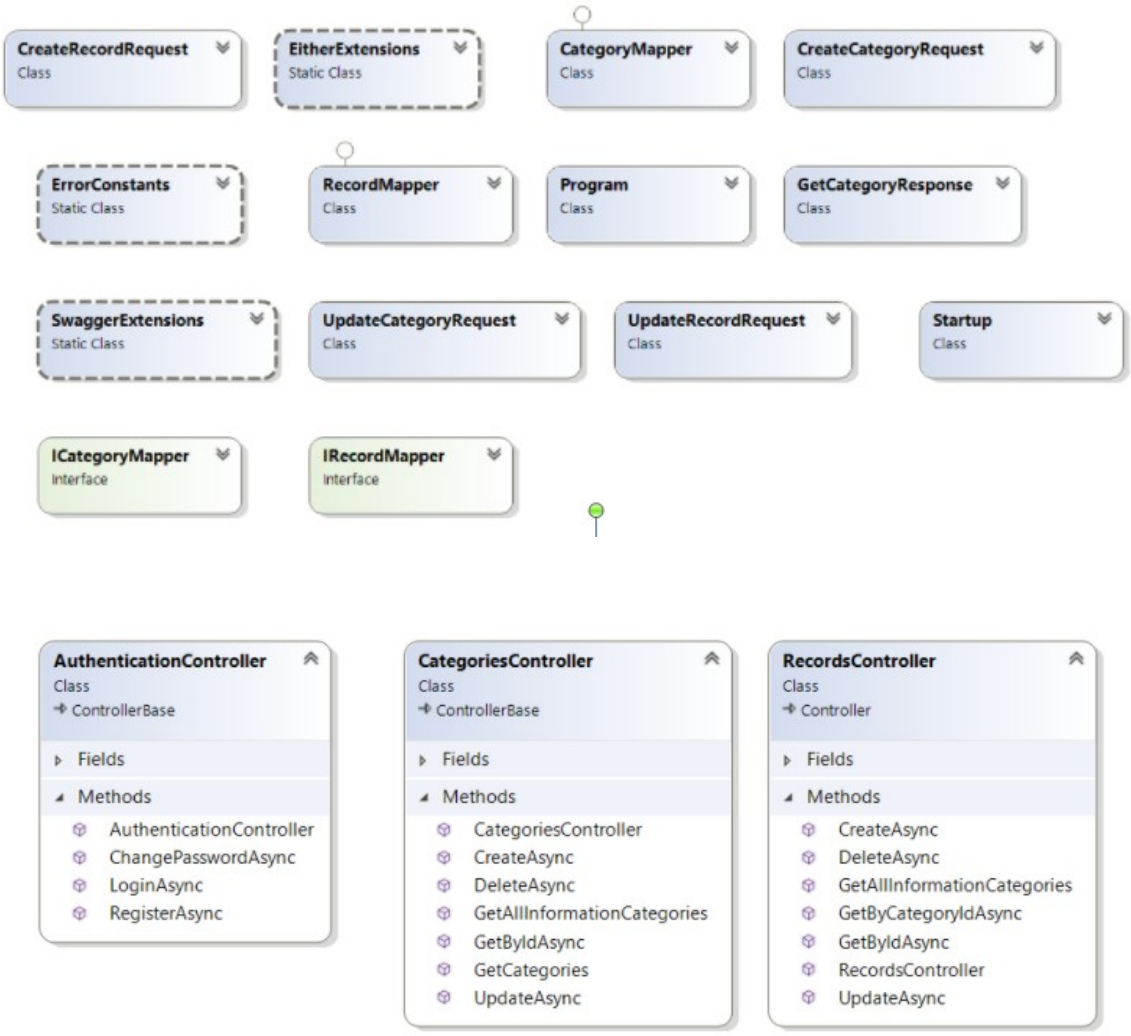

Figure 14. Clase folosite in API Layer

ASP.NET Core acceptă crearea de servicii RESTful, cunoscute și sub numele de API-uri web, utilizând C#. Pentru a gestiona cererile, un API web folosește controlere. Controlerele dintr-un API web sunt clase care derivă din ControllerBase. [10]

Controller derivă din ControllerBase și adaugă suport pentru vizualizări, deci este pentru gestionarea paginilor web, nu a cererilor API web. Există o excepție de la această regulă: dacă intenționam să utilizam același controlor atât pentru vizualizări, cât și pentru API-urile web.

```
namespace LookApp.API.Controllers
{
   [Route("api/[controller]")]
   [ApiController]
   public class AuthenticationController : ControllerBase
   {...
```

Clasa oferă multe proprietăți și metode care sunt utile pentru tratarea cererilor HTTP. De exemplu, ControllerBase.

Atribute

Namespace-ul Microsoft.AspNetCore.Mvc oferă atribute care pot fi utilizate pentru a configura comportamentul controlerelor API web și metodele de acțiune. Următorul exemplu folosește atribute pentru a specifica verbul de acțiune HTTP acceptat și orice coduri de stare HTTP cunoscute care ar putea fi returnate.

Exemplu de atribute sunt următoarele:

Attribute	Notes
[Route]	Specifies URL pattern for a controller or action.
[Bind]	Specifies prefix and properties to include for model binding.
[HttpGet]	Identifies an action that supports the HTTP GET action verb.
[Consumes]	Specifies data types that an action accepts.
[Produces]	Specifies data types that an action returns.

Figure 15. Lista cu atribute

AuthenticationController injectează serviciul de autentificare în constructor. Acest controller se ocupă de funcționalitatea de schimbare parolă, înregistrare și logare.

CategoriesController se ocupă de funcționalitatea legată de categorii: creare categorii, ștergeri categorii, obținerea informațiilor despre o categorie, obținerea tuturor categoriilor, obținerea categoriilor după id, dar și actualizarea unei categorii. Ca și AuthenticationController, CategoriesController folosește dependency injection pentru a injecta serviciul CategoryService,

serviciul care se află în Business Layer.

RecordsController se ocupă de funcționalitatea legată de înregistrări: creare înregistrări, ștergere înregistrări, returnarea înregistrării cu un anumit id, returnarea tuturor înregistrărilor, dar și actualizarea unei înregistrări.

Clasa CreateCategoryRequest este un DTO și este folosit de CategoryRequest pentru mapare. Sunt situații în care în unele câmpuri din modelul Category/Records să nu fie necesare când trebuie trimise către front-end și invers. În acest caz, aceste clase CreateCategoryRequest, CreateCategoryResponse, GetCategoryResponse, UdateCategoryRequest, UpdateCategoryResponse intră în ajutor.

În domeniul programării, un **obiect de transfer de date** (**DTO)** este un obiect care transportă date între procese. Motivația utilizării sale este că comunicarea între procese se face de obicei apelând la interfețe la distanță (de exemplu, servicii web), unde fiecare apel este o operațiune costisitoare. Deoarece majoritatea costului fiecărui apel este legat de timpul dus-întors între client și server, o modalitate de reducere a numărului de apeluri este utilizarea unui obiect (DTO) care să agregheze datele care ar au fost transferate de mai multe apeluri, dar acesta este deservit de un singur apel.

Diferența dintre obiectele de transfer de date și obiectele de business sau obiectele de acces la date este că un DTO nu are niciun comportament, cu excepția stocării, regăsirii, serializării și deserializării propriilor date (mutatori, accesori, parseri și serializatori). Cu alte cuvinte, DTO-urile sunt obiecte simple care nu ar trebui să conțină nicio logică de afaceri, dar pot conține mecanisme de serializare și deserializare pentru transferul de date prin cablu.

Acest model este adesea folosit incorect în afara interfețelor la distanță. Acest lucru a declanșat un răspuns din partea autorului său unde reiterează că întregul scop al DTO-urilor este de a transfera datele în apeluri la distanță costisitoare.

2.1.1.3. Business Layer

În software-ul computerului, business logic este partea programului care codifică regulile de afaceri din lumea reală care determină modul în care datele pot fi create, stocate și modificate. Este în contrast cu restul software-ului care ar putea fi preocupat de detalii de nivel inferior despre gestionarea unei baze de date sau afișarea interfeței cu utilizatorul, infrastructura de sistem sau, în general, conectarea diferitelor părți ale programului.

O arhitectură pe mai multe niveluri formalizează această decuplare prin crearea unui **business layer** separat de alte niveluri, cum ar fi database layer. Fiecare nivel „știe” doar o cantitate minimă despre codul din celelalte niveluri- suficient pentru a îndeplini sarcinile necesare.

Business logic:

- Prescrie modul în care obiectele interacționează între ele
- Aplică rutele și metodele prin care obiectele comerciale sunt accesate și actualizate
- Modelează obiecte din viața reală (cum ar fi conturi, împrumuturi și inventare)

Business logic cuprinde:

- Fluxuri de lucru care sunt sarcinile ordonate de a transmite documente sau date de la un participant (o persoană sau un sistem software) la altul.

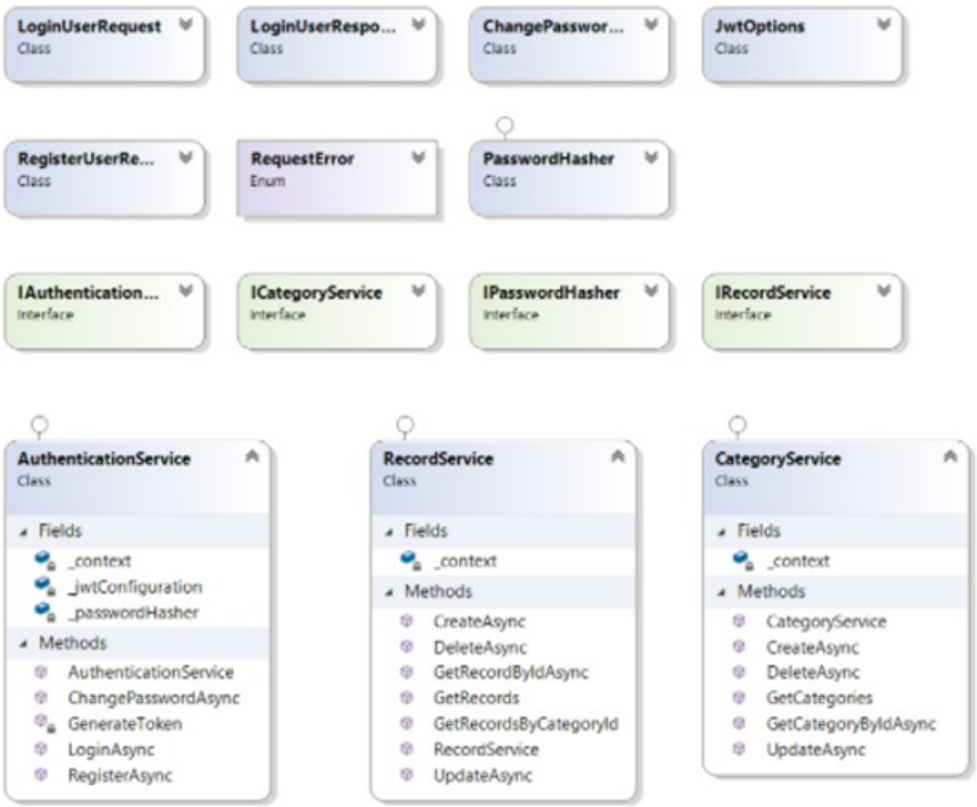

Figure 16. Clase folosite în Business Layer

În clasele din Business Layer, avem în principal serviciile folosite: AuthenticationService, RecordService, CategoryService.

Așa cum sugerează și poza de mai sus, AuthenticationService are metodele LoginAsync, RegisterAsync, ChangePasswordAsync. În cazul apariției câtorva erori, avem enum RequestError. Acest enum are următoarea structura:

```
public enum RequestError
  {
    UserAlreadyExists,
    PasswordFormatError,
    InvalidCredentials
  }
```

În cazul metodei LoginAsync, aceasta folosește clasa LoginUserRequest și returnează Task<LoggedinUser>, adică o eroare de autentificare ca una din cele enumarate în RequestError, sau returnează LoginUserResponse, care fata de LoginUserRequest are în plus un token. Aceasta

metoda folosește însă și clasa PasswordHasher care folosește un algoritm de criptare cu cheie simetrica (SHA256). [11]

SHA-2 (Secure Hash Algoritm 2) este un set de funcții hash criptografice conceput de Agenția de Securitate Națională (NSA) a Statelor Unite ale Americii. Aceasta are la bază structurile Merkle–Damgård, care la rândul ei este o funcție de compresie unidirecțională făcută cu structuri Davies–Meyer cu un cifru pe blocuri specializat.

Funcțiile criptografice hash sunt operații matematice care se aplică pe date digitale; prin compararea unui „hash" (valoarea finală a funcției) cu un alt hash cunoscut, se poate determina integritatea. De exemplu, calcularea hash-ului unui fișier descărcat și compararea acestuia cu un hash publicat anterior se poate determina dacă în procesul de descărcare, fișierul a fost modificat sau alterat.

SHA-2 include schimbări semnificative de la predecesorul său, SHA-1. SHA-2 este formată din șase funcții hash cu valori care pot fi de 224, 256, 384 sau 512 biți: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256.

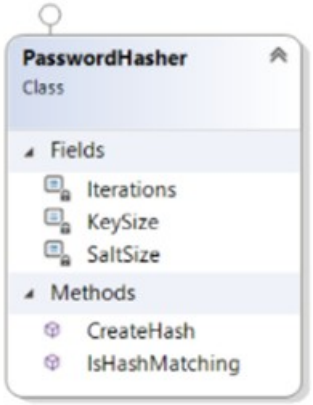

Figure 17. Clasa PasswordHasher folosită în AuthenticationService

Aceasta clasă conține funcția CreateHash și IsHashMatching. Funcția CreateHash întoarce un rezultat de tipul salt.key, care rezultă din aplicarea algoritmului HSA256. Funcția IsHashMatching verifică dacă parola introdusă de utilizator este corectă, prin compararea valorilor cheilor.

LookAppContext ne abstractizează baza de date prin DbSet și astfel, ne ajută să facem operații pe datele din model.

RecordService folosește clasa LookAppContext, care este injectată prin dependency injection în constructorul serviciului. Acest serviciu se ocupă de get, create, delete și update.

CategoryService folosește clasa LookAppContext, care este injectat prin dependency injection în constructorul serviciului. Acest serviciu se ocupă, ca și RecordService de get, create, delete și update.

2.1.1.4. Database Layer

Un layer de abstractizare a bazei de date (DBAL sau DAL) este o interfață a aplicației care unifică comunicarea dintre o aplicație și baze de date. [12]

Layerele de abstractizare a bazei de date reduc cantitatea de lucru oferind dezvoltatorului un API consistent și ascund cât mai mult posibil detaliile bazei de date în spatele acestei interfețe.

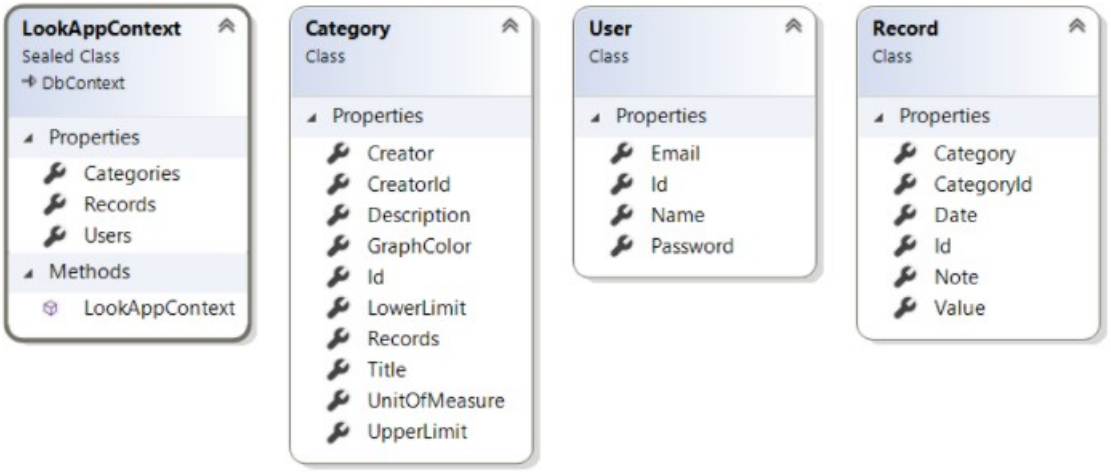

Figura 18. Clasele din Models

În Database Layer, în folderul “Models” se află cele 3 clase care o să fie folosite de întreagă aplicație.

Clasa principală care este responsabilă de interacțiunea cu datele ca obiecte este DbContext. Modul recomandat de a lucra cu contextul este de a defini o clasă care derivă din DbContext și expune proprietățile DbSet care reprezintă colecțiile entităților specificate în context.

```
namespace LookApp.Database.Models
{
    public sealed class LookAppContext : DbContext
    {
        public LookAppContext(DbContextOptions<LookAppContext> options) :
base(options)
        {
            Database.Migrate();
        }

        public DbSet<Category> Categories { get; set; }
        public DbSet<Record> Records { get; set; }
        public DbSet<User> Users { get; set; }
    }
}
```

2.1.2. Arhitectura front-end

Tehnologia folosită pentru implementarea interfeței aplicației este Angular. Motivul pentru care am ales această tehnologie este pentru că Angular ajută la crearea de aplicații interactive și dinamice, inclusiv șablonare, legare bidirecțională, modularizare, gestionare API RESTful, injecție de dependență și gestionare AJAX.

Pe lângă acest lucru dacă aplicația devine foarte mare, folosind JavaScript poate deveni dificilă mentenanța, spre deosebire de Angular care are un cod curat, ușor de întreținut și de testat.

Frameworkul Angular se bazează pe componente, care încep în același stil. De exemplu, fiecare componentă plasează codul într-o clasă de componente sau definește un decorator @Component (metadate). Aceste componente sunt elemente de interfață mici, independente una de cealaltă și, prin urmare, vă oferă mai multe avantaje, inclusiv:

Reutilizarea
Structura bazată pe componente a Angular face componentele extrem de reutilizabile în aplicație.

Testarea unitară simplificată
Fiind independente una de cealaltă, componentele facilitează testarea unității.

Mai citeț
Coerența în codificare face ca citirea codului să fie mai plăcută.

Ușurința de întreținere
Componentele decuplate pot fi înlocuite cu implementări mai bune. Pur și simplu, permite întreținerea și actualizarea eficientă a codului.

Paginile existente în aplicație sunt: pagina de login, signup, change password, dashboard, category list, category-details, pagina de help și pagina pentru corelație.

Pentru ca front-end-ul să comunice cu back-end-ul, am definit 2 repositoryService (categoryRepoitoryService, recordRepositoryService) care au același endpoint cu cel din back-end. Pentru eficientizare și performanță, am folosit aceste metode în servicii și am reținut rezultatul în memorie pentru a nu apela de prea multe ori la backend și pentru a mai scurta timpul de răspuns atunci când datele au fost deja accesate o dată.

```
@Injectable({
  providedIn: 'root'
})
export class CategoryRepositoryService {

  private readonly endpoint: string = 'https://localhost:44387/api/categories';

  constructor(private readonly http: HttpClient) { }

  getAllCategories(): Observable<CategoryModel[]> {
    return this.http.get<CategoryModel[]>(`${this.endpoint}/allCategoryDetails`);
  }

  getById(id: number): Observable<CategoryModel> {
    return this.http.get<CategoryModel>(`${this.endpoint}/${id}`);
  }
```

```
addCategory(categoryModel: CategoryModel): Observable<CategoryModel> {
  return this.http.post<CategoryModel>(`${this.endpoint}`, categoryModel);
}

deleteCategory(id: number): Observable<void> {
  return this.http.delete<void>(`${this.endpoint}/${id}`);
}

updateCategory(id: number, categoryModel: CategoryModel): Observable<CategoryModel> {
  return this.http.put<CategoryModel>(`${this.endpoint}/${id}`, categoryModel)
}
```

Majoritatea aplicațiilor front-end trebuie să comunice cu un server prin protocolul HTTP, pentru a descărca sau încărca date și a accesa alte servicii back-end. Angular oferă un API HTTP client pentru aplicațiile Angular, HttpClient. Clasa CategoryRepositoryService va comunica cu back-end-ul și extrage și încarca date către acesta. Metodele implementate de această clasă sunt: getAllCategories,getById, addCategory, deleteCategory, updateCategory.

Utilizăm metoda HttpClient.get() pentru a prelua date de pe un server. Metoda asincronă trimite o cerere HTTP și returnează un Observable care emite datele solicitate la primirea răspunsului.

Metoda HttpClient.put() ajută la a face update unei categorii. Metoda asteaptă ca parametru id-ul categoriei și modelul categoriei și va returna un observabil de tip CategoryModel.

Metoda HttpClient.delete() ajută la a șterge o anumită categorie. Metoda așteaptă ca parametru id-ul categoriei care urmează a fi ștearsă și returnează un observabil de tip void.

Metoda HttpClient.post() este similară cu get() prin faptul că are un parametru de tip, pe care îl putem utiliza pentru a specifica că ne așteptăm ca serverul să returneze date de un anumit tip. Metoda ia o adresă URL și doi parametri **suplimentari**:

Body - Datele către POST în corpul cererii.
Options - Un obiect care conține opțiuni care specifică antetele necesare.

```
export class CategoryService {

  private readonly categories: BehaviorSubject<CategoryModel[]>;

  constructor(private categoryRepositoryService: CategoryRepositoryService) {
    this.categories = new BehaviorSubject<CategoryModel[]>([]);
  }

  populateCategories(): Observable<Observable<CategoryModel[]>> {
    return this.categoryRepositoryService.getAllCategories()
      .pipe(
        map(categories => {
          this.categories.next(categories);
          return this.categories.asObservable();
        })
      );
  }

  getById(categoryId: number): Observable<CategoryModel> {
    let category = this.categories.value.find(category => category.id === categoryId);
    if (!category) {
      return this.categoryRepositoryService.getById(categoryId);
    }
    return of(category);
  }
```

Clasa CategoryService folosește dependency injection și astfel folosește categoryRepositoryService. Scopul acestei clase este de a eficientiza timpul de răspuns al paginilor. Spre exemplu, la prima accesare a unei pagini care dorește să preia o categorie după id-ul dat, serviciul va apela la back-end și va primi rezultatul, însă la o nouă cerere, datele pot fi returnate din memorie. Cine face acest lucru să fie posibil? În acest caz, este vorba de variabila categories de tip BehaviorSubject.

BehaviorSubject este un tip de observabil (adică un flux de date la care ne putem abona la fel ca observabilul returnat din cererile HTTP). Spun un *tip* de observabil pentru că este puțin diferit de un standard observabil. Ne abonăm la un BehaviorSubject așa cum am face la un observabil normal, dar beneficiul unui BehaviorSubject pentru scopurile noastre este că:

- Va returna întotdeauna o valoare, chiar dacă încă nu au fost emise date din fluxul său
- Când vă abonați la acesta, acesta va returna imediat ultima valoare care a fost emisă imediat (sau valoarea inițială dacă nu au fost încă date emise)

Vom folosi BehaviorSubject pentru a păstra datele pe care dorim să le accesăm în întreaga aplicație. Acest lucru ne va permite să:

- Încărcam datele o singură dată sau numai când este nevoie
- Asigurăm că o anumită valoare validă este întotdeauna furnizată oricărui utilizator care o folosește (chiar dacă o încărcare nu a terminat încă)

- Notificăm momentul în care datele se schimbă pe oricine care este abonat la

BehaviorSubject.

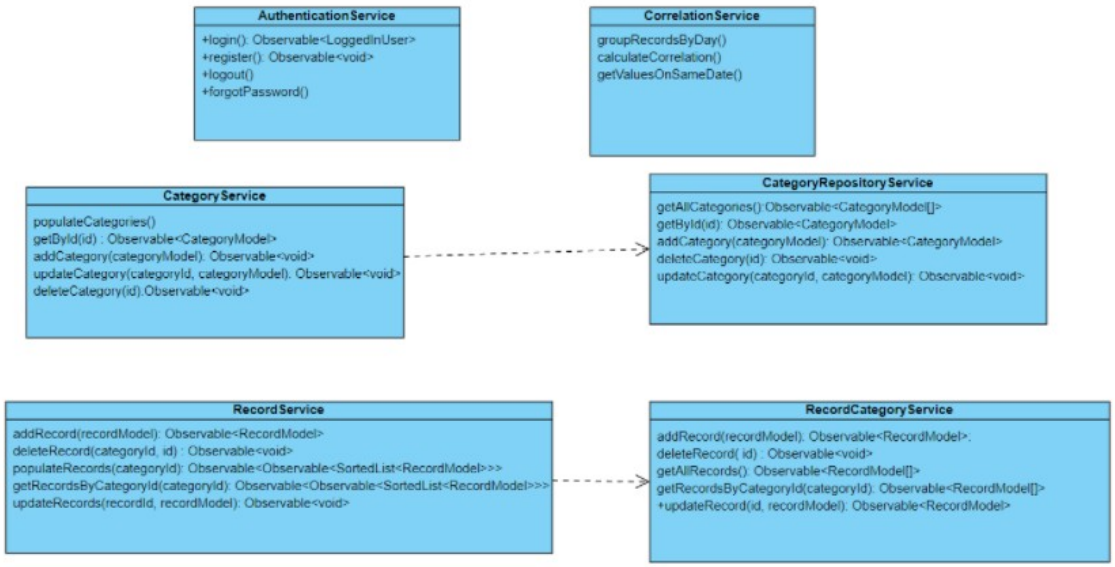

Figure 19. Diagrama serviciilor

În momentul în care utilizatorul face import de fișier CSV și vrea să vadă noile date importate în grafic, el poate face undo la acțiunea. De asemenea, poate face undo și la operația de adăugare a unui punct.

CommandService are ca atribut o listă de comenzi de tipul Command. În momentul în care utilizatorul adaugă o înregistrare (o comandă de tipul AddRecordComponent), în lista commandsHistory o să fie adăugată această comandă, precum și următoarele care o să mai vină.

În momentul în care utilizatorul apasă pe undo, din acea listă o să fie scoasă ultima comandă și se vor efectua operațiile specifice de ștergere corespunzătoare comenzii.

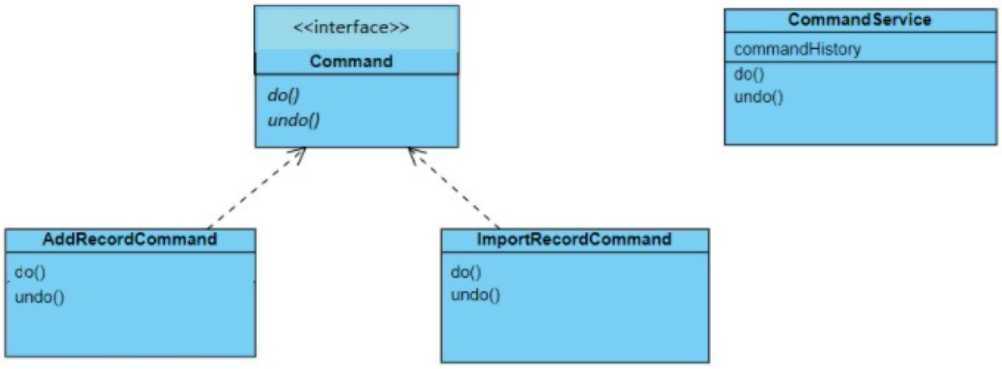

Figure 20. Diagrama command pattern

Capitolul 3. Implementarea aplicației

3.1. Implementare back-end

Când am început implementarea aplicației, primul pas a fost să creez API layer-ul aplicației. Al doilea pas a fost proiectarea bazei de date în Microsoft SQL Server Management Studio. Pentru aplicația aceasta, a fost nevoie de 3 tabelele: Records, Categories și Users. Lucrul acesta putea fi făcut ușor manual în aplicația Microsoft SQL Server Management Studio, dar totuși, am ales să creez aceste tabele din C# (code first), în proiectul ce aparține layer-ului database.

```
namespace LookApp.Database.Models
{
    25 references
    public class Category
    {
        4 references
        public int Id { get; set; }
        3 references
        public string Title { get; set; }
        3 references
        public string Description { get; set; }
        3 references
        public string UnitOfMeasure { get; set; }
        2 references
        public double? LowerLimit { get; set; }
        2 references
        public double? UpperLimit { get; set; }

        2 references
        public string? GraphColor { get; set; }
        3 references
        public ICollection<Record> Records { get; set; }

        4 references
        public int CreatorId { get; set; }
        0 references
        public User Creator { get; set; }
    }
```

```
namespace LookApp.Database.Models
{
    25 references
    public class Record
    {
        3 references
        public int Id { get; set; }
        3 references
        public DateTime Date { get; set; }
        2 references
        public string Note { get; set; }
        2 references
        public double Value { get; set; }

        3 references
        public int CategoryId { get; set; }
        0 references
        public Category Category { get; set; }
    }
}
```

```
7 references
public class User
{
    3 references
    public int Id { get; set; }
    2 references
    public string Name { get; set; }
    5 references
    public string Email { get; set; }
    3 references
    public string Password { get; set; }
}
```

Al doilea pas a fost să creez contextul bazei de date și astfel se creează și tabelele din baza de date. Când baza de date este creată, EF creează tabele care au nume identice cu numele DbSet. Numele proprietăților pentru colecții sunt de obicei la plural. De exemplu, mai degrabă Categories decât Category.

```
namespace LookApp.Database.Models
{
    17 references
    public sealed class LookAppContext : DbContext
    {
        0 references
        public LookAppContext(DbContextOptions<LookAppContext> options) : base(options)
        {
            Database.Migrate();
        }

        5 references
        public DbSet<Category> Categories { get; set; }
        6 references
        public DbSet<Record> Records { get; set; }
        5 references
        public DbSet<User> Users { get; set; }
    }
}
```

Următorul pas a fost crearea business layer-ului și crearea serviciilor fără de care aplicația nu poate funcționa. Serviciile folosite sunt: CategoryService, RecordService și AuthenticatinService. În Capitolul 2 (Arhitectura aplicației) am explicat care sunt serviciile aplicației în ansamblu, iar acum o să explic mai în detaliu functionalitatile oferite de aceste servicii. CategoryService, așa cum specifică și numele se ocupă de funcționalitățile legate de o categorie. Acest serviciu conține metode precum GetCategories(),GetCategoryByIdAsync, CreateAsync, DeleteAsync, UpdateAsync. Toate aceste metode se termina în Async, deoarece metodele folosesc await async.

Modelul de programare asincronă Task (TAP) oferă abstractizare asupra codului asincron. Putem citi codul ca și cum fiecare instrucțiune se completează înainte ca următoarea să înceapă. Compilatorul efectuează multe transformări, deoarece unele dintre aceste instrucțiuni pot începe să lucreze și să returneze un Task care este încă în desfășurare. Cuvântul await oferă o modalitate non-blocantă de a porni un task, apoi de a continua execuția la finalizarea acelei sarcini. Acesta este obiectivul acestei sintaxe: activăm codul care citește ca o succesiune de instrucțiuni, dar se execută într-o ordine mult mai complicată bazată pe alocarea resurselor externe și când sarcinile

se finalizează. Este similar cu modul în care oamenii dau instrucțiuni pentru procesele care includ sarcini asincrone. Ați scrie instrucțiunile de genul următoarei liste pentru a explica cum să preparați un mic dejun:

1. Se toarnă o ceașcă de cafea.
2. Încălziți o tigaie, apoi prăjiți două ouă.
3. Se prăjesc trei felii de bacon.
4. Prăjește două bucăți de paine.
5. Adăugați unt și gem la pâine prăjită.
6. Se toarnă un pahar de suc de portocale.

Dacă aveți experiență în gătit, veți executa aceste instrucțiuni în **mod asincron**. Ați începe să încălziți tigaia pentru ouă, apoi ați găti baconul. Ați pune pâinea în prăjitorul de pâine, apoi ați prăji ouăle. La fiecare pas al procesului, veți începe o sarcină, apoi vă veți îndrepta atenția asupra sarcinilor care sunt gata.

Metoda getCategories filtrează categoriile în funcție de utilizator. Fiecare utilizator poate să vadă și să modifice doar propriile sale categorii. Metoda createAsync, cum zice și numele ajută la crearea unei categorii. DeleteAsync ajută la ștergerea unei categorii, pe când updateAsync ajută la actualizarea unei categorii. Funcționalitățile din RecordService sunt asemănătoare. RecordService se ocupă de obținerea înregistrărilor, crearea lor, obținerea unei înregistrări după un id, ștergerea unei înregistrări și actualizarea unei înregistrări. [13]

```
[Route("api/[controller]")]
    [ApiController]
    [Authorize]
    public class CategoriesController : ControllerBase
    {
        private readonly ICategoryService _categoryService;
        private readonly ICategoryMapper _categoryMapper;

        private readonly int _currentUserId;

        public CategoriesController(
            ICategoryService categoryService,
            ICategoryMapper categoryMapper,
            IHttpContextAccessor httpContextAccessor)
        {
            this._categoryService = categoryService;
            this._categoryMapper = categoryMapper;

            this._currentUserId =
int.Parse(httpContextAccessor.HttpContext.User.FindFirst("Id").Value);
        }

        [HttpGet]
        public ActionResult<List<GetCategoryResponse>> GetCategories()
        {
            var categories = _categoryService.GetCategories(this._currentUserId);
            var categoriesResponseList = categories.Select(c =>
_categoryMapper.MapToGetCategoryResponse(c));

            return Ok(categoriesResponseList);
        }
```

```
        [HttpGet("allCategoryDetails")]
        public ActionResult<List<Category>> GetAllInformationCategories()
        {
            var categories = _categoryService.GetCategories(this._currentUserId);
            return Ok(categories);
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<Category>> GetByIdAsync([FromRoute] int id)
        {
            var result = await _categoryService.GetCategoryByIdAsync(id,
this._currentUserId);
            if (result == null)
            {
                return NotFound("There's no category with the provided id.");
            }

            return Ok(result);
        }

        [HttpPost]
        public async Task<ActionResult<Category>> CreateAsync(CreateCategoryRequest
createCategoryRequest)
        {
            var categoryToAdd = _categoryMapper.MapToCategory(createCategoryRequest,
this._currentUserId);
            var addedCategory = await _categoryService.CreateAsync(categoryToAdd);

            return Created(addedCategory.Id.ToString(), addedCategory);
        }

        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteAsync([FromRoute] int id)
        {
            var result = await _categoryService.GetCategoryByIdAsync(id,
this._currentUserId);
            if (result == null)
            {
                return NotFound("There's no category with the provided id.");
            }

            await _categoryService.DeleteAsync(result);
            return NoContent();
        }

        [HttpPut("{id}")]
        public async Task<IActionResult> UpdateAsync([FromRoute] int id,
UpdateCategoryRequest updatedCategoryRequest)
        {
            var categoryToUpdate = await _categoryService.GetCategoryByIdAsync(id,
this._currentUserId);
            if (categoryToUpdate == null)
            {
                return NotFound("There is no category with the provided id.");
            }

            var updatedCategory = _categoryMapper.MapToUpdatedCategory(categoryToUpdate,
updatedCategoryRequest, this._currentUserId, id);
            await _categoryService.UpdateAsync(categoryToUpdate);
```

```
            return Ok(categoryToUpdate);
        }
```

3.2. Implementare front-end

La implementarea interfeței a fost mult de lucru și a fost nevoie de atenție la detalii. Biblioteca folosită în crearea graficelor se numește Chart.js, am folosit și RxJS (observabile, behaviorSubject) și Angular Material. Am folosit o gardă de activare, implementată cu ajutorul interfeței canActivate, interfață pe care o clasă o poate implementa pentru a decide dacă se poate activa o rută. Dacă se returnează true, navigarea continuă. Dacă se întoarce un guard false, navigarea este anulată. În aplicația aceasta, utilizatorul nu poate intra pe nicio pagină fară să fie autentificat. Singurele pagini la care are acces sunt paginile de login, sign up și change password.

Tot pe această parte de autentificare, am folosit un JWT interceptor care ajută la setarea unui header de autentificare cu JWT dacă utilizatorul este autentificat cu succes.

Serviciile folosite sunt: AuthenticationService, CategoryService, RecordService, ChartService, CommandService (folosește CommandPattern), CorrelationService (grupează datele, le filtrează, calculează coeficientul de corelație Pearson).

Una din dificultățile întâmpinate în implementarea interfetei a fost actualizarea graficului în timp real de fiecare dată când acționam asupra datelor sale. De exemplu, când adaugam un punct nou pe grafic dând click pe chart, trebuia să mai încarc o dată pagina (refresh) ca să pot vedea rezultatul. În rezolvarea acestei probleme am venit cu o soluție, și anume stocarea datelor graficelor într-o variabilă de tip BehaviorSubject. De ce? Vom folosi BehaviorSubject pentru a păstra datele pe care dorim să le accesăm în întreaga aplicație. Acest lucru ne va permite să:

- Încărcam datele o singură dată sau numai când este nevoie
- Asigurăm că o anumită valoare validă este întotdeauna furnizată oricărui utilizator care o folosește (chiar dacă o încărcare nu a terminat încă)
- Notificăm momentul în care datele se schimbă pe oricine care este abonat la BehaviorSubject

```
export class CategoryService {

 private readonly categories: BehaviorSubject<CategoryModel[]>;

 constructor(private categoryRepositoryService: CategoryRepositoryService) {
  this.categories = new BehaviorSubject<CategoryModel[]>([]);
 }

 populateCategories(): Observable<Observable<CategoryModel[]>> {
  return this.categoryRepositoryService.getAllCategories()
   .pipe(
    map(categories => {
     this.categories.next(categories);
     return this.categories.asObservable();
```

```
      })
    );
  }

  getById(categoryId: number): Observable<CategoryModel> {
   let category = this.categories.value.find(category => category.id === categoryId);
   if (!category) {
    return this.categoryRepositoryService.getById(categoryId);
   }
   return of(category);
  }

  addCategory(categoryModel: CategoryModel): Observable<void> {
   return this.categoryRepositoryService.addCategory(categoryModel)
    .pipe(
     map(addedCategory => {
      this.categories.next([...this.categories.value, addedCategory]);
     })
    );
  }

  updateCategory(categoryId: number, categoryModel: CategoryModel): Observable<void> {
   let updatedCategoryList = [];
   return this.categoryRepositoryService.updateCategory(categoryId, categoryModel)
    .pipe(
     map(updatedCategory => {
      this.categories.value.forEach(element => {
       if (element.id == updatedCategory.id) {
        updatedCategoryList.push(updatedCategory)
       } else {
        updatedCategoryList.push(element)
       }
      });
      this.categories.next(updatedCategoryList);
     }
     )
    )
  }

  deleteCategory(id: number): Observable<void> {
   return this.categoryRepositoryService.deleteCategory(id)
    .pipe(
     map(() => {
      this.categories.next(this.categories.value.filter(c => c.id !== id));
     })
    );
  }
}
```

În codul de mai jos este implementarea lui CorrelationService. Prima metodă este groupRecordsByDay care primește ca parametru un set de date, care este de tipul ChartPointModel, id-ul categoriei și titlul categoriei.

```
export class ChartPointModel {

  public x: string;
  public y: number;

  public constructor(x: string, y: number) {

    this.x = x;
    this.y = y;
  }
}
```

Acest model conține date despre data calendaristică (x) și valoarea înregistrată (y). Metoda din CorrelationService folosește o procedură Reduce() care face operația de adunare a valorilor grupate după aceeași data calendaristică.

```
export class CorrelationService {

  public correlationCoeff : number = undefined;

  public groupRecordsByDay(chartDataSetToCorrelate: ChartPointModel[], categoryId: number, catego
ryTitle: string): RecordsByDay {

    let groupedDataSet: RecordsByDay = chartDataSetToCorrelate.reduce(
      (acc: RecordsByDay, record: ChartPointModel) => {

        let day: string = record.x.toString().split("T")[0];
        let val: number = record.y;

        if (acc.recordsByDay[day]) {
          acc.recordsByDay[day] += val;
        } else {
          acc.recordsByDay[day] = val;
        }

        return acc;

      }, new RecordsByDay(categoryId, categoryTitle));

    return groupedDataSet;
  }
```

Următoarea metodă returnează valorile înregistrate în aceeași zi din cele 2 seturi de date.

```
getValuesOnSameDate(firstDataSet: RecordsByDay, secondDataSet: RecordsByDay): CategoriesToCorrela
te[] {

    let firstDataSET = JSON.parse(JSON.stringify(firstDataSet));
    let secondDataSET = JSON.parse(JSON.stringify(secondDataSet));

    let categoriesToCorrelate: CategoriesToCorrelate[] = [];

    for (const [key, value] of Object.entries(secondDataSET.recordsByDay)) {
      if (firstDataSET.recordsByDay[key] == undefined) {
        delete secondDataSET.recordsByDay[key];
      }
    }

    for (const [key, value] of Object.entries(firstDataSET.recordsByDay)) {
      if (secondDataSET.recordsByDay[key] == undefined) {
        delete firstDataSET.recordsByDay[key];
      }
    }

    categoriesToCorrelate.push({ "firstCategory": firstDataSET, "secondCategory": secondDataSET }
);

    return categoriesToCorrelate;
  }
```

Metoda de mai jos se ocupă de calcularea coeficientului de corelație. Formula de calcul folosită este Ecuația (4) din Capitolul 1.

```
calculateCorrelationCoef(categoriesToCorrelate: CategoriesToCorrelate[]): number {
  let sumOfX: number = 0;
  let sumOfY: number = 0;

  let x: number = 0;
  let y: number = 0;

  let sumOfXMultipliedWithY: number = 0;

  let sumOfSquareX = 0;
  let sumOfSquareY = 0;

  let n = 0;

  for (const [key, value] of Object.entries(categoriesToCorrelate[0].firstCategory.recordsByDay
) {
    x = value;
    sumOfX += x;

    y = categoriesToCorrelate[0].secondCategory.recordsByDay[key];
    sumOfY += y;

    sumOfXMultipliedWithY += x * y;

    sumOfSquareX += x * x;
    sumOfSquareY += y * y;
  }
  n = Object.getOwnPropertyNames(categoriesToCorrelate[0].firstCategory.recordsByDay).length;

  let numitor: number = Math.sqrt((n * sumOfSquareX - sumOfX * sumOfX) * (n * sumOfSquareY -
sumOfY * sumOfY));
  let numarator: number = (n * sumOfXMultipliedWithY - sumOfX * sumOfY);
  this.correlationCoeff = numarator / numitor;

  if(numitor == 0){
    this.correlationCoeff = 0;
  }

  return this.correlationCoeff;
```

3.3. Funcționalitățile aplicației

Flow-ul aplicației este următorul: utilizatorul intră pe aplicație, pe pagina login și poate intra pe contul său după ce trece email-ul și parola in câmpul email, respectiv password. În cazul în care nu are cont creat, poate crea un cont pe pagina signup. Condițiile de creare cont sunt ca emailul să nu mai fie folosit până acum și parola să aibă lungimea de minim 8 caractere, dintre care o literă mare și o cifră.

După ce autentificarea este realizată cu succes, utilizatorul va fi redirecționat către pagina de categorii, unde are opțiunea de a crea o categorie customizată și de a căuta printre categoriile create.

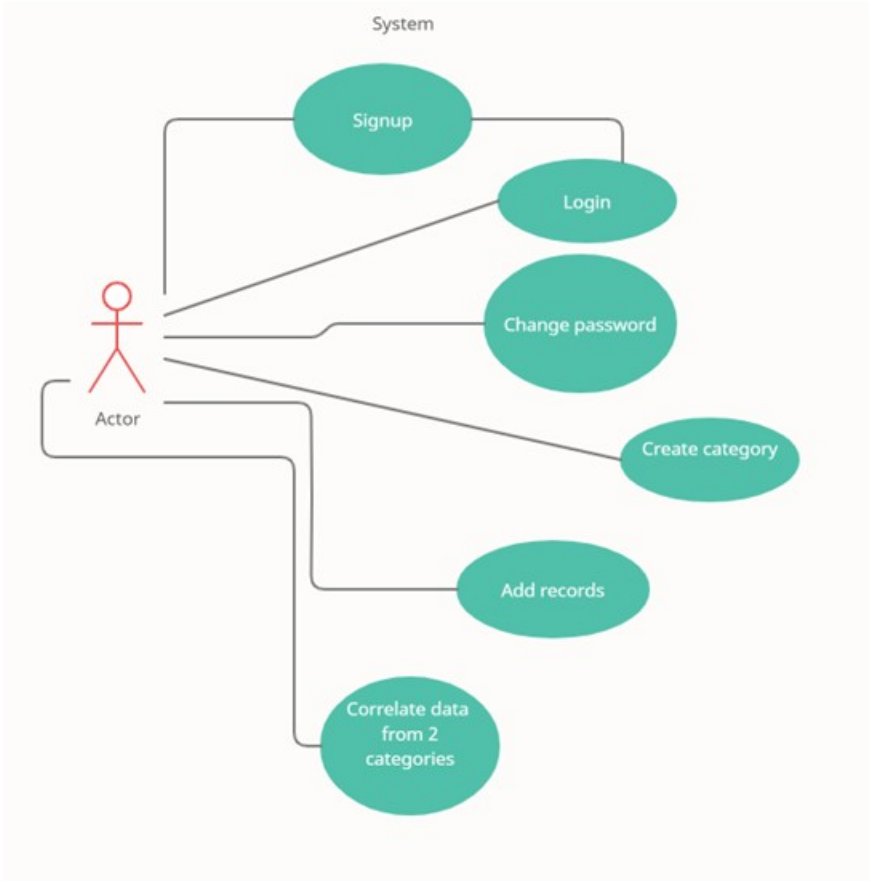

Figure 21. Flow al aplicației

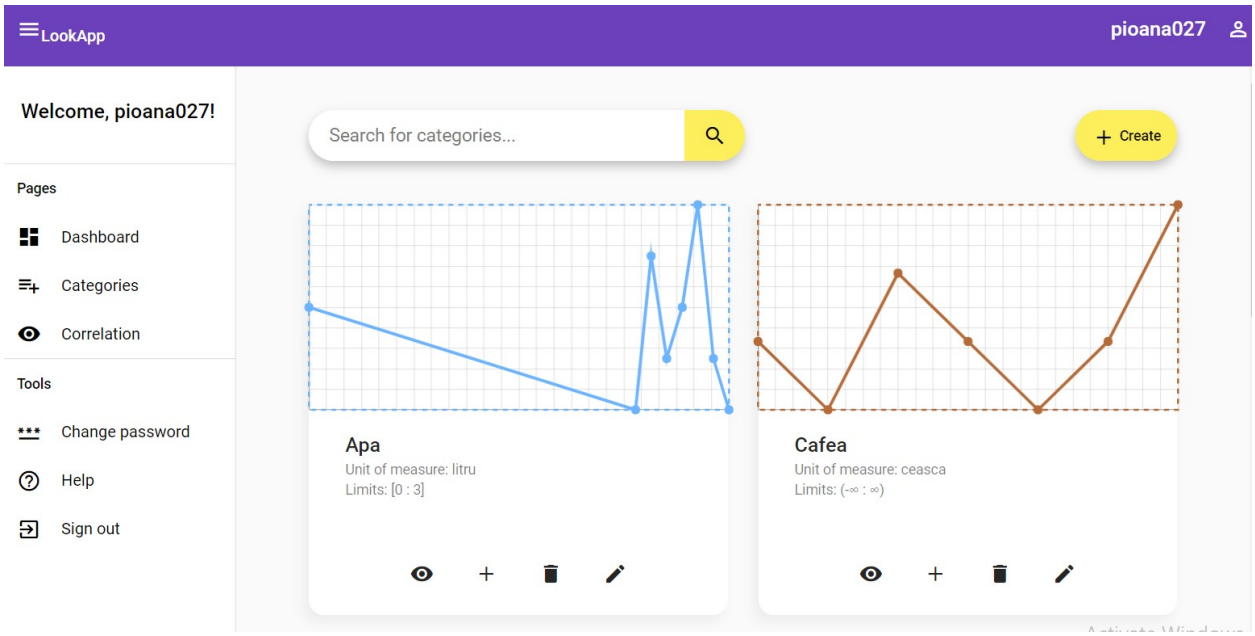

Figure 22. Pagina de categorii

La crearea unei categorii se va deschide un modal, în care utilizatorul este rugat să scrie date despre titlul categoriei care se dorește a fi adăugat, unitatea de măsură, descrierea, culoarea graficului, limita inferioară și limita superioară.

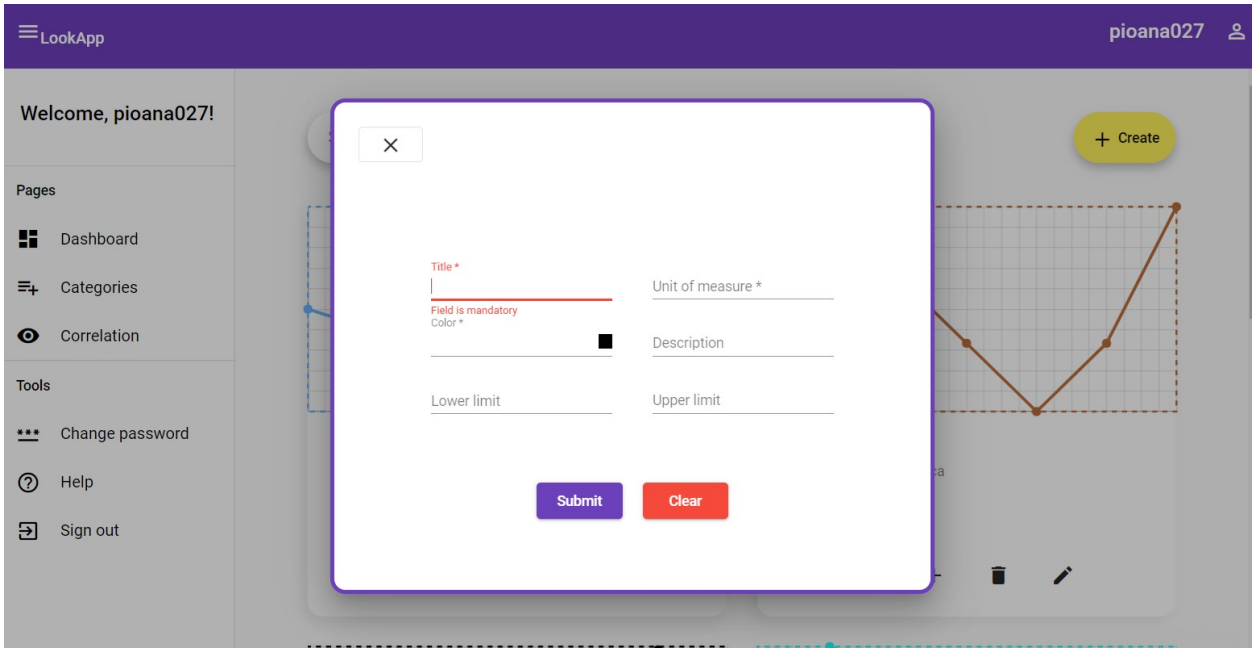

Figure 23. Modal de editare categorie

Fiecare categorie va fi prezentată într-un mat-card format din imaginea de previzualizare a graficului, datele despre categorie (unitatea de măsură, limitele), descrierea și un mic meniu cu butoane pentru a efectua prelucrări asupra categoriei. Primul buton ajută la redirectarea către pagina de detalii a categoriei, al doilea buton ajută la deschiderea unui modal pentru adăugarea unor înregistrări pentru categoria respectivă, al treilea buton este pentru ștergerea categoriei, iar al patrulea este pentru editarea categoriei, buton care va deschide un modal asemănător cu modalul deschis pentru crearea unei categorii, dar câmpurile sunt autocompletate cu date.

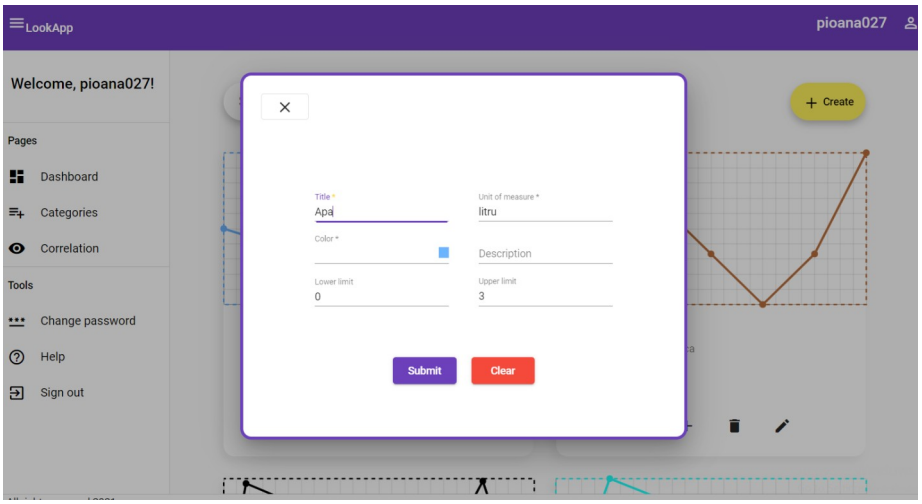

Figure 24. Modal de editare categorie

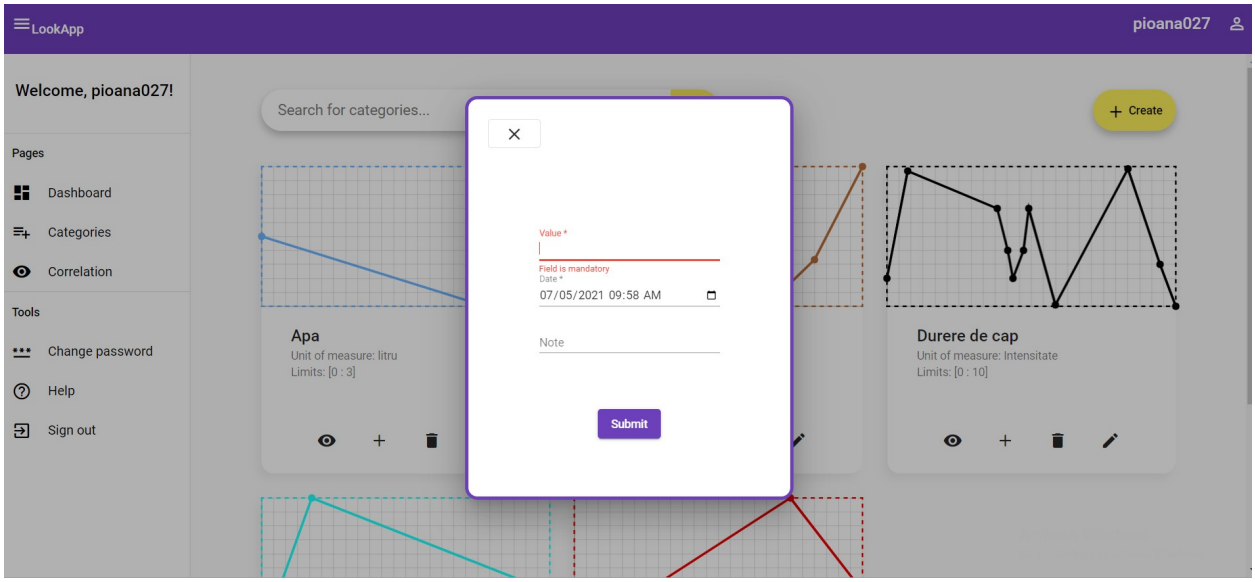

Figure 25. Modal de adăugare inregistrare

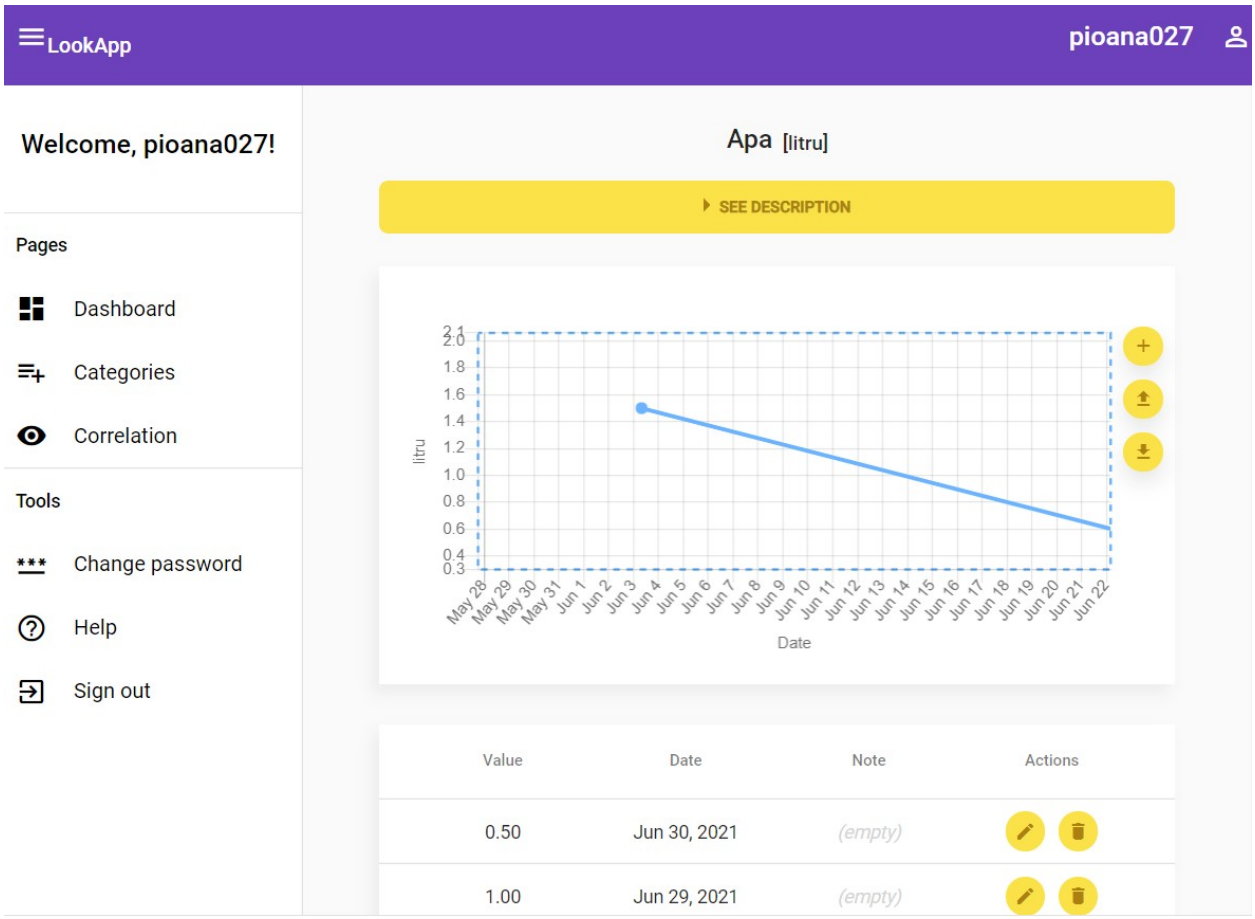

Figure 26. Pagina detalii categorie

Pe pagina dashboard, este prezentat un tabel cu valorile coeficienților de corelație Pierson a combinațiilor de categorii, ordonate descrescător după valoarea coeficientului. O să fie combinații de categorii cu valorile coeficienților, cu valori intre [-1, 1].

Dacă valoarea va fi -1, asta indică prezența unei relații puternic negative, în sensul că dacă datele primei categorii cresc, datele celei de-a doua categorii descresc. Aceste valori negative sunt colorate cu roșu.

Dacă valoarea este 0, asta indică faptul că nu există o relație între date. Aceste valori de 0 sunt colorate cu gri.

Dacă valoarea este pozitivă, asta indică faptul că există o relație puternic pozitivă, în sensul că dacă datele primei categorii cresc, atunci și datele celei de-a doua categorii sunt în creștere. Aceste valori sunt colorate cu verde.

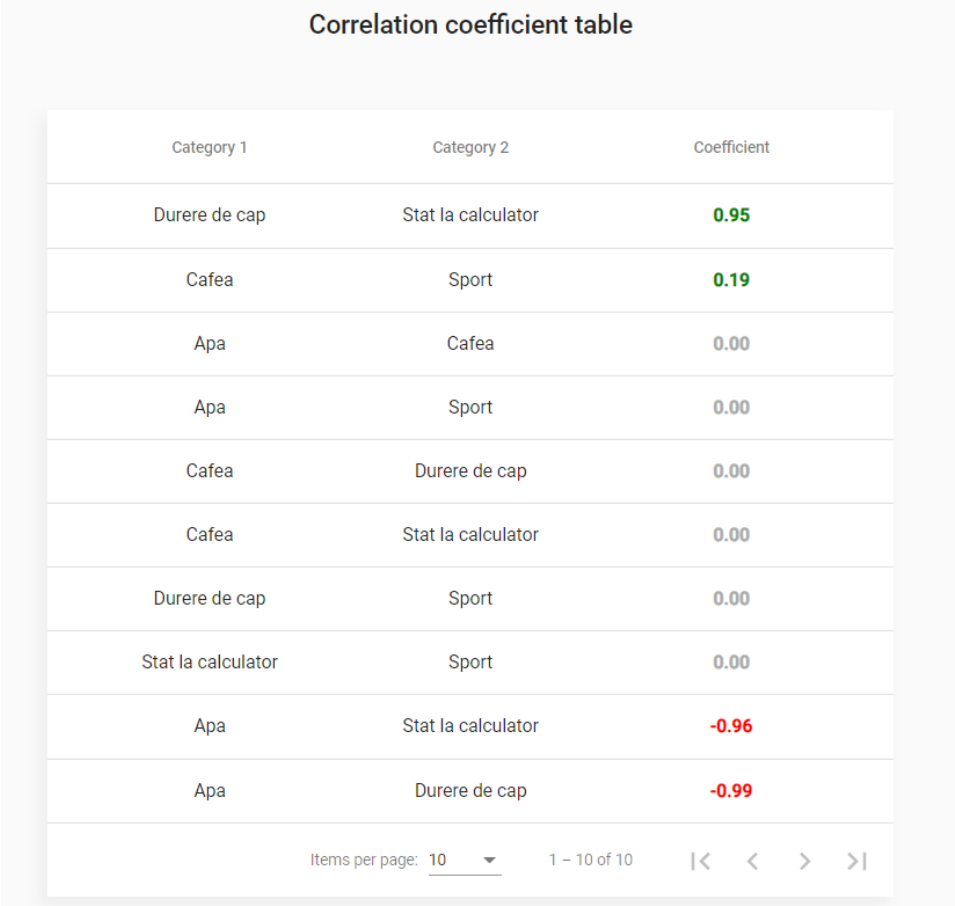

Figure 27. Pagina dashboard

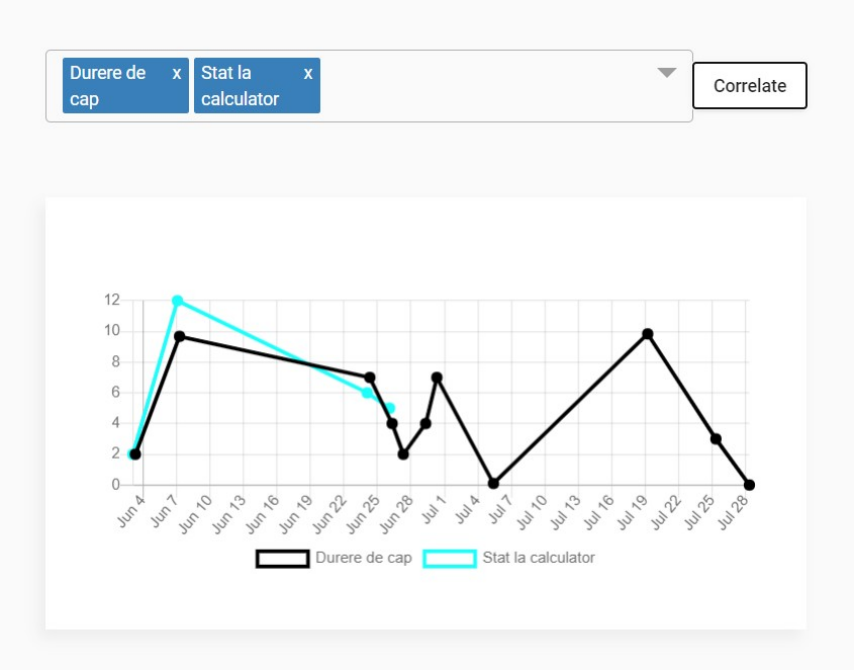

Figure 28. Pagina corelatie

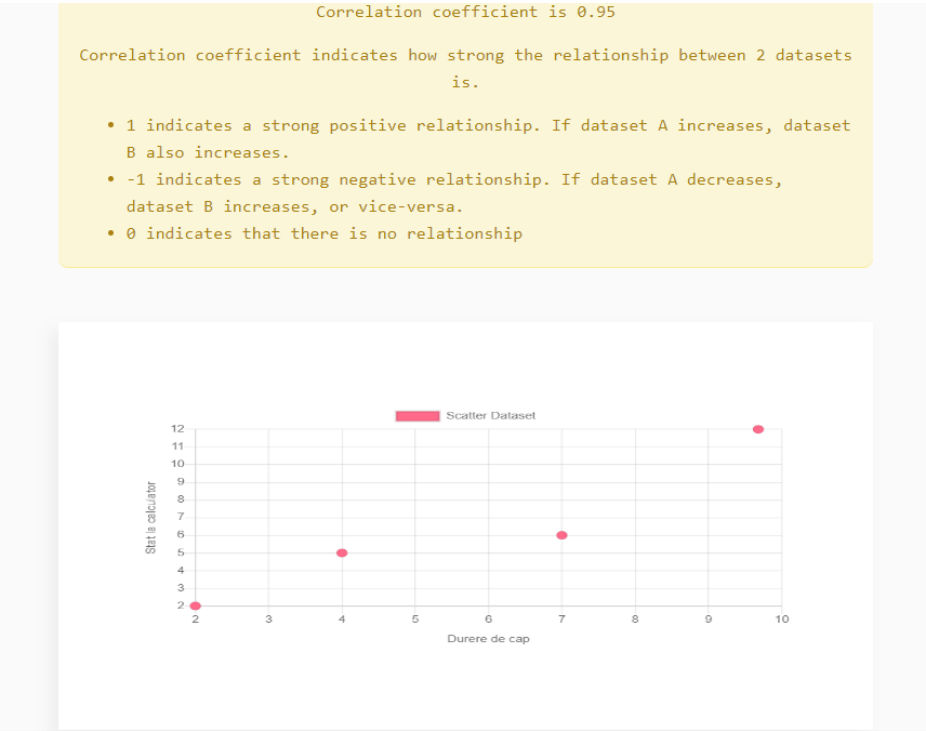

Figure 29. Pagina corelatie

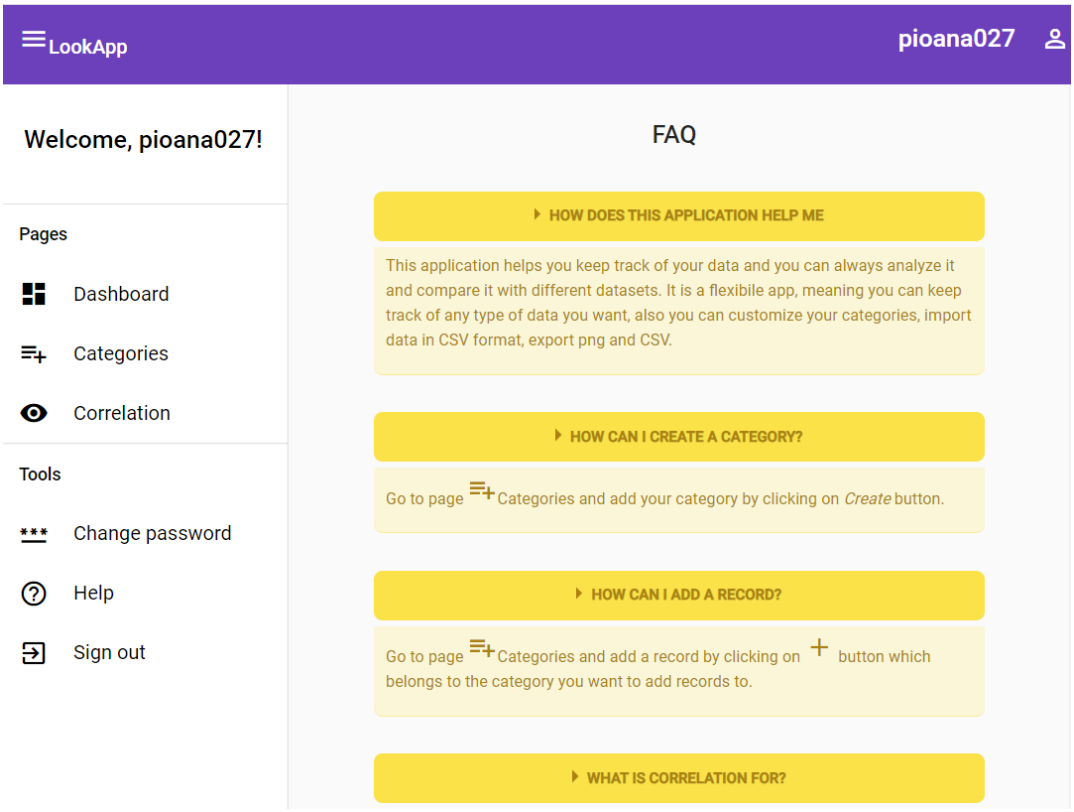

Figure 30. Pagina help

Capitolul 4. Testarea aplicației

Aplicația lucrează cu multe date și în spatele algoritmului folosit sunt multe calcule. În acest capitol urmăresc să prezint corectitudinea rezultatelor și să evidențiez folosirea mesajelor de eroare sau de atentioare în interfață, pentru că utilizatorul să aibă o experiență plăcută folosind această aplicație.

Primele pagini accesate de utilizator sunt cele de login, signup. Interfața abordează și situațiile în care informațiile introduse nu sunt valide, iar utilizatorul este înștiințat că ceva nu a funcționat corect.

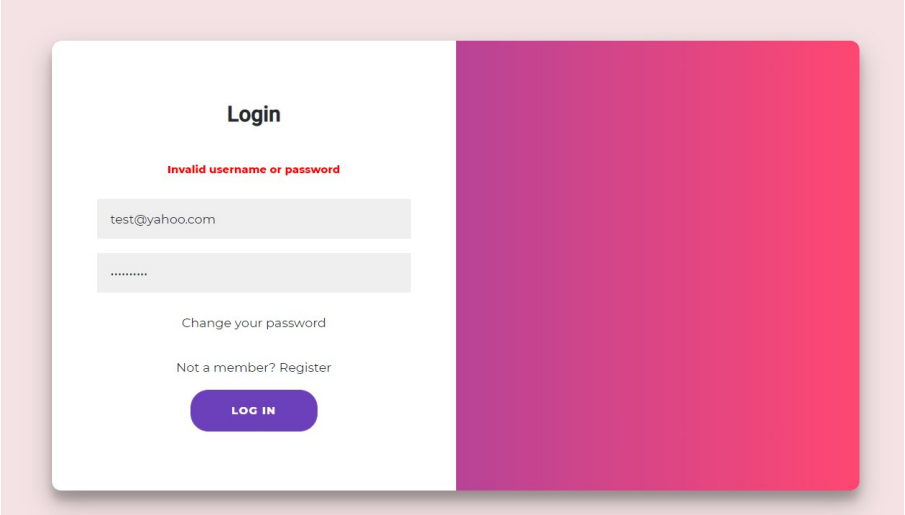

Figure 31. Pagina de login si mesaj de eroare

Pe interfața de login va fi afișat un mesaj de eroare în cazul în care credențialele nu sunt corecte. În cazul în care utilizatorul dorește să creeze un cont și nu introduce parola în formatul dorit (O litera mare, cifre, iar dimensiunea minimă să fie 8), o să apară un mesaj de eroare și va fi înștiințat.

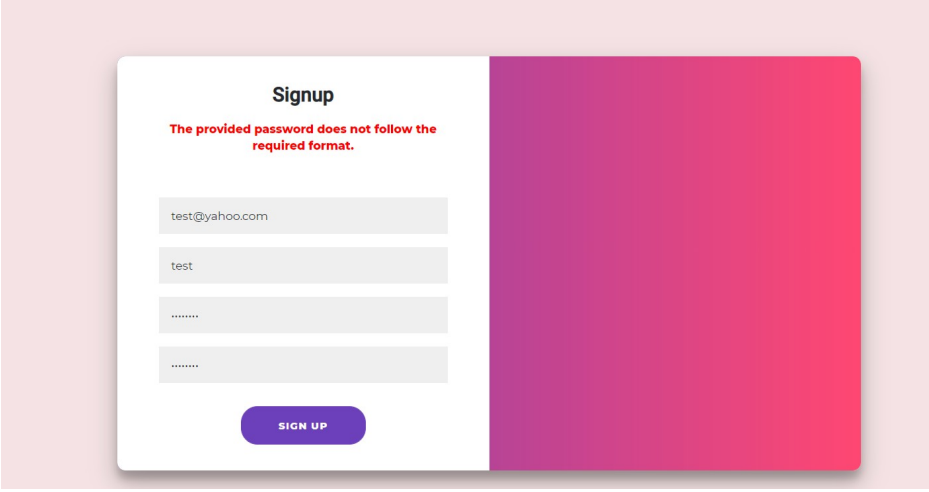

Figure 32. Pagina de signup si mesaj de eroare

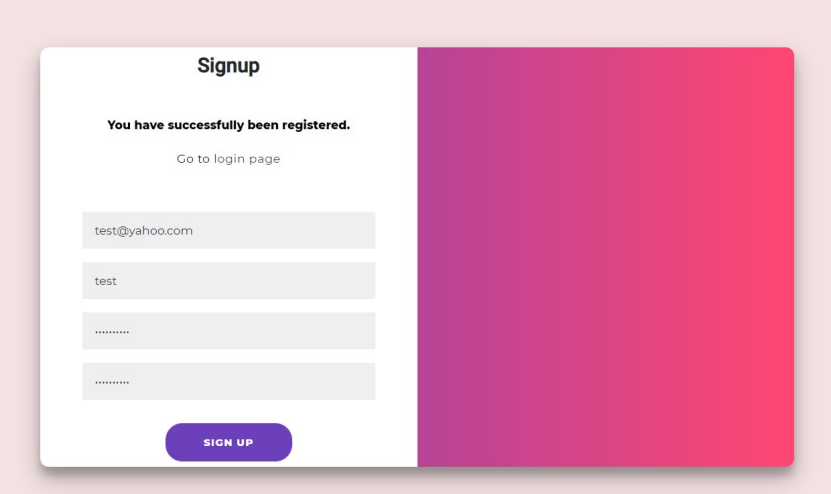

Figure 33. Pagina de signup si mesaj cu succes

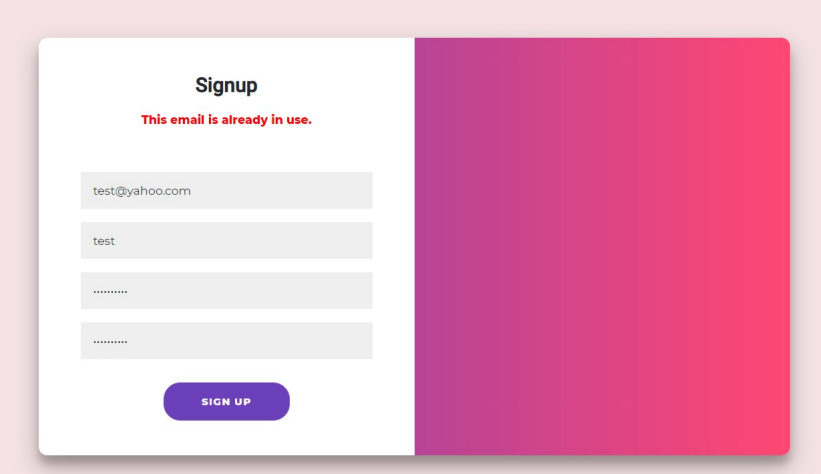

Figure 34. Pagina de signup si mesaj de eroare "email already in use"

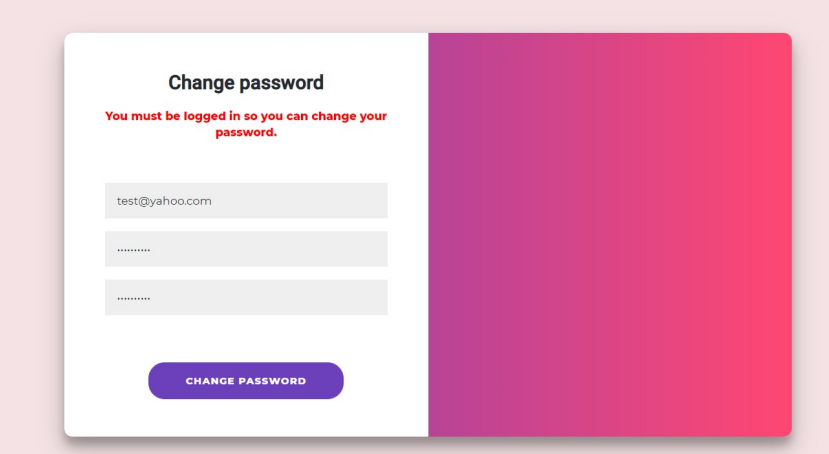

Figure 35. Pagina de change password si mesaj de eroare

Pentru verificarea acurateții oferite de algoritm, am ales să introduc niște seturi de date cu

titlurile: durere de cap, cafea, apă, stat la calculator și sport.

Am introdus datele astfel încât, cu cât am introdus că am băut mai puțină apă, cu atât în acea zi am introdus că durerea de cap a apărut și într-o intensitate mai mare. Coeficientul de corelație in acest caz ar trebui să aibă valoarea aproximativ egală cu -1 deoarece relația dintre date este negativă, când un set de date este crescător, celălalt set de date este descrescător.

Figure 36. Graficul rezultat in urma suprapunerii a două seturi de date

Ceea ce putem observa pe grafic este că în zilele în care utilizatorul a băut mai puțină apă, durerea de cap a fost mai mare, iar în zilele în care a notat că a băut mai mult, intensitatea durerii de cap a fost mai scăzută. Acest tip de relație este negativă, iar rezultatul este -0.99, aproximând, putem spune că este -1.

În figura de mai jos, observăm că au fost extrase 3 valori care au fost înregistrate de utilizator în aceeași zi. Perechile de valori sunt reprezentate într-un scatter dataset. Rezulatul este o linie descrescătoare, indicând că există o relație negativă între date. Explicația poate fi găsită în capitolul 1, paragraful 1.4.

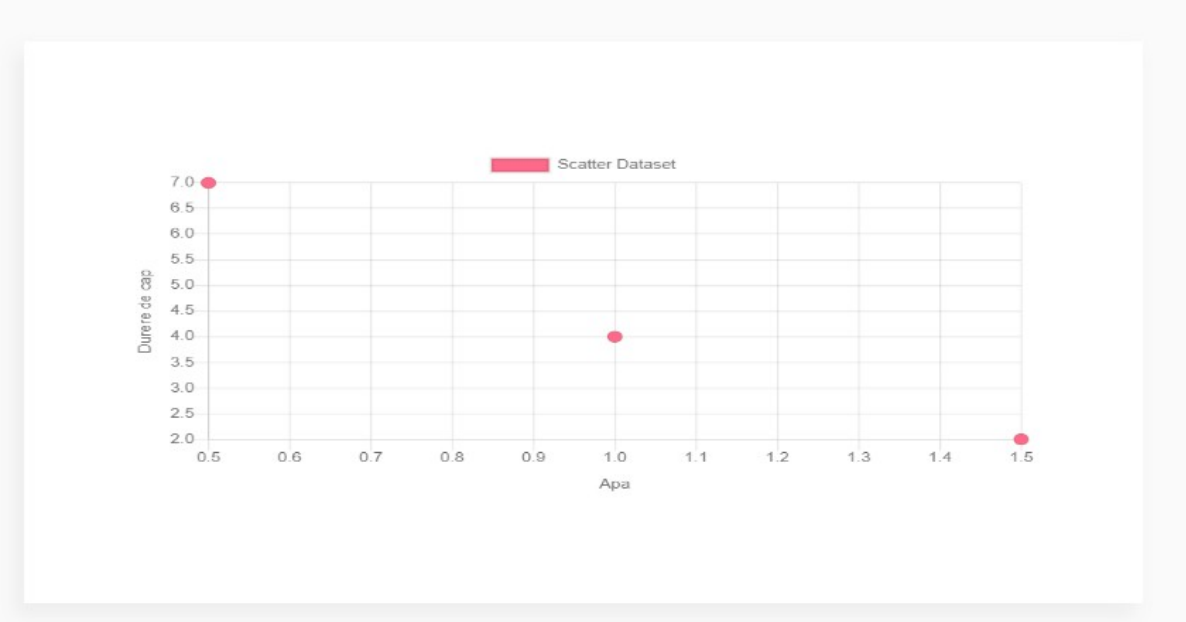

Figure 37. Scatter dataset

În imaginea de mai jos am folosit site-ul https://www.statskingdom.com/correlation-calculator.html pentru a verifica corectitudinea rezultatului. În concluzie, rezultatul este același, atât valoarea coeficientului este același, cât și formă scatter chart-ului. [14]

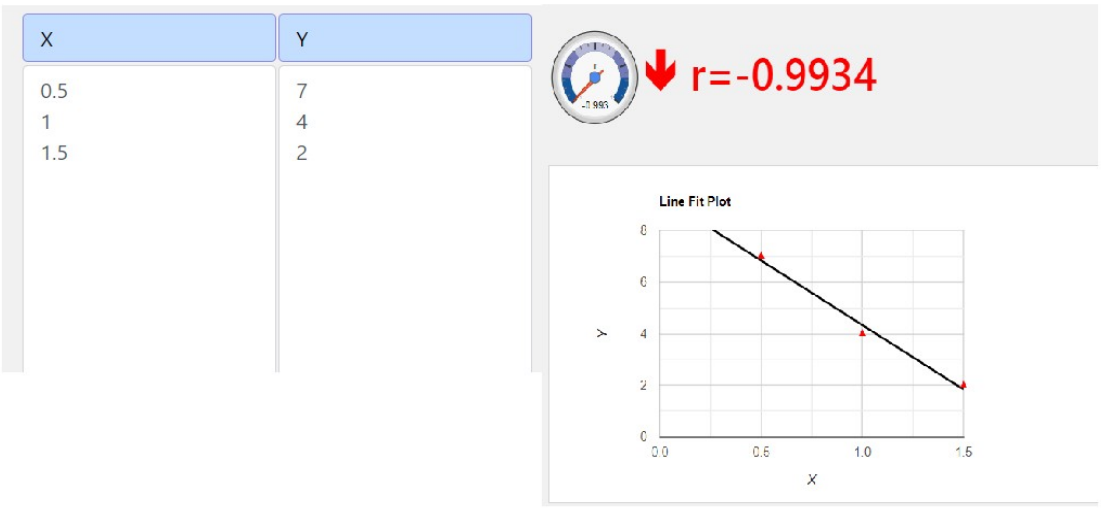

Figure 38. Calcul coefficient de corelatie pe site-ul https://www.statskingdom.com/correlation-calculator.html

Concluzii

Odată cu trecerea timpului, conștiința oamenilor asupra importanței și valorii datelor personale și a analizei lor a crescut semnificativ. Odată cu creșterea în popularitate a dispozitivelor inteligente cum ar fi smart watch care îți oferă informații despre datele tale biometrice și creșterea în popularitatea a conceptelor din aria well being, oamenii au devenit mai interesați de propriile date personale și de îmbunătățirea propriei lor vieți.

Aplicația această combină mai multe tipuri de "tracking app" într-una singură și oamenilor le este mult mai la îndemână să organizeze și să adune toate datele în același loc. Plusurile aplicației față de o aplicație simplă de monitorizare și înregistrare de date sunt, cum am zis și mai sus, flexibilitatea, personalizarea tipurilor de date și posibilitatea de a suprapune și de a vedea relațiile dintre date.

La ce sunt bune datele?

- Îmbunătățesc viețile oamenilor
- Data = cunoștințe. Datele oferă dovezi clare, pe când lipsa datelor conduce la presupuneri, posibil concluzii false
- Ajută la monitorizarea sănătății
- Ajută la măsurarea eficacității unei strategii aplicate. Colectarea datelor vor ajută în concluzionarea a cât de bine funcționează soluția ta
- Ajută la găsirea soluțiilor pentru probleme
- Ajută la a cunoaște ce faci bine și ce faci rău
- Ajută la a ține evidență

Prin crearea acestei aplicații, am învățat multe lucruri pe care nu le știam așa bine. Nu am mai creat până acum server în .NET, iar cunoștințele aplicate au fost noi pentru mine. Am rezolvat problemele pe care le-am întâmpinat, spre exemplu interfața nu se actualiza în timp real și am folosit noțiunea de BehaviorSubject, am întâlnit cazul în care era nevoie de implementarea unui design pattern, anume command pattern.

Dacă ar fi să mai continui aplicația, aș mai adăugă funcționalitatea de a calcula câte înregistrări a adăugat utilizatorul pe zi și aș crea un grafic în care utilizatorul poate observa evoluția folosirii aplicației. Aș mai fi adăugat funcționalitatea de redo pe pagina de detalii a unei categorii.

Bibliografie

[1] Wikipedia, [Online], Disponibil la adresa: https://ro.wikipedia.org/wiki/Angular, Accesat: 2021.

[2] [Online], Disponibil la adresa: https://scoalainformala.ro/ce-este-net-si-de-ce-sa-cunosti-aceasta-tehnologie/, Accesat: 2021.

[3] Microsoft, [Online], Disponibil la adresa: https://docs.microsoft.com/en-us/ef/core/, Accesat: 2021.

[4] Microsoft,[Online], Disponibil la adresa: https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-5.0&tabs=visual-studio, Accesat: 2021.

[5] Microsoft docs, [Online], Disponibil la adresa: https://docs.microsoft.com/en-us/dotnet/api/system.data.entity.dbcontext?view=entity-framework-6.2.0, Accesat: 2021.

[6] JWT, [Online], Disponibil la adresa: https://jwt.io/ , Accesat: 2021.

[7] Microsoft, [Online], Disponibil la adresa: https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli, Accesat: 2021.

[8] Wikipedia, „Pearson”, , , pp. , 2021.

[9] Wikipedia, Covarianta [Online], Disponibil la adresa: https://ro.wikipedia.org/wiki/Covarian%C8%9B%C4%83, Accesat: 2021.

[10] Microsoft, [Online], Disponibil la adresa: https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-5.0 , Accesat: .

[11] Wikipedia, [Online], Disponibil la adresa: https://ro.wikipedia.org/wiki/SHA-2 , Accesat: 2021.

[12] Wikipedia, Database abstraction layer [Online], Disponibil la adresa: https://en.wikipedia.org/wiki/Database_abstraction_layer, Accesat: 2021.

[13] Microsoft , Async [Online], Disponibil la adresa: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/, Accesat: 2021.

[14] Statskingdom, Correlation Calculation [Online], Disponibil la adresa: https://www.statskingdom.com/correlation-calculator.html, Accesat: .

Anexe.

Anexa 1. Back-end

//API LAYER

```
namespace LookApp.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthenticationController : ControllerBase
    {
         private readonly IAuthenticationService _authenticationService;

        public AuthenticationController(IAuthenticationService authenticationService)
        {
            _authenticationService = authenticationService;
        }

        [HttpPost("register")]
        public async Task<IActionResult> RegisterAsync([FromBody] RegisterUserRequest
registerUserRequest)
        {
            var result = await _authenticationService.RegisterAsync(registerUserRequest);
            if (result.IsLeft)
            {
                var requestError = result.GetLeft();
                return BadRequest(ErrorConstants.Messages[requestError]);
            }

            var createdUser = result.GetRight();

            return Created(createdUser.Id.ToString(), null);
        }

        [HttpPost("login")]
        public async Task<IActionResult> LoginAsync([FromBody] LoginUserRequest
loginUserRequest)
        {
            var result = await _authenticationService.LoginAsync(loginUserRequest);
            if (result.IsLeft)
            {
                var requestError = result.GetLeft();
                return BadRequest(ErrorConstants.Messages[requestError]);
            }

            var loginResponse = result.GetRight();

            return Ok(loginResponse);
        }

        [HttpPost("changePassword")]
        [Authorize]
        public async Task<IActionResult> ChangePasswordAsync([FromBody] ChangePasswordRequest
changePasswordRequest)
        {
            var result = await
_authenticationService.ChangePasswordAsync(changePasswordRequest);
            if (result.IsLeft)
            {
                var requestError = result.GetLeft();
                return BadRequest(ErrorConstants.Messages[requestError]);
            }
```

```
            var changePasswordResponse = result.GetRight();

            return Ok(changePasswordResponse);
        }
    }
}

namespace LookApp.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    [Authorize]
    public class CategoriesController : ControllerBase
    {
        private readonly ICategoryService _categoryService;
        private readonly ICategoryMapper _categoryMapper;

        private readonly int _currentUserId;

        public CategoriesController(
            ICategoryService categoryService,
            ICategoryMapper categoryMapper,
            IHttpContextAccessor httpContextAccessor)
        {
            this._categoryService = categoryService;
            this._categoryMapper = categoryMapper;

            this._currentUserId =
int.Parse(httpContextAccessor.HttpContext.User.FindFirst("Id").Value);
        }

        [HttpGet]
        public ActionResult<List<GetCategoryResponse>> GetCategories()
        {
            var categories = _categoryService.GetCategories(this._currentUserId);
            var categoriesResponseList = categories.Select(c =>
_categoryMapper.MapToGetCategoryResponse(c));

            return Ok(categoriesResponseList);
        }

        [HttpGet("allCategoryDetails")]
        public ActionResult<List<Category>> GetAllInformationCategories()
        {
            var categories = _categoryService.GetCategories(this._currentUserId);
            return Ok(categories);
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<Category>> GetByIdAsync([FromRoute] int id)
        {
            var result = await _categoryService.GetCategoryByIdAsync(id, this._currentUserId);
            if (result == null)
            {
                return NotFound("There's no category with the provided id.");
            }

            return Ok(result);
        }

        [HttpPost]
        public async Task<ActionResult<Category>> CreateAsync(CreateCategoryRequest
```

```
createCategoryRequest)
        {
            var categoryToAdd = _categoryMapper.MapToCategory(createCategoryRequest,
this._currentUserId);
            var addedCategory = await _categoryService.CreateAsync(categoryToAdd);

            return Created(addedCategory.Id.ToString(), addedCategory);
        }

        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteAsync([FromRoute] int id)
        {
            var result = await _categoryService.GetCategoryByIdAsync(id, this._currentUserId);
            if (result == null)
            {
                return NotFound("There's no category with the provided id.");
            }

            await _categoryService.DeleteAsync(result);
            return NoContent();
        }

        [HttpPut("{id}")]
        public async Task<IActionResult> UpdateAsync([FromRoute] int id, UpdateCategoryRequest
updatedCategoryRequest)
        {
            var categoryToUpdate = await _categoryService.GetCategoryByIdAsync(id,
this._currentUserId);
            if (categoryToUpdate == null)
            {
                return NotFound("There is no category with the provided id.");
            }

            var updatedCategory = _categoryMapper.MapToUpdatedCategory(categoryToUpdate,
updatedCategoryRequest, this._currentUserId, id);
            await _categoryService.UpdateAsync(categoryToUpdate);
            return Ok(categoryToUpdate);
        }
    }
}


namespace LookApp.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class RecordsController : Controller
    {
        private readonly IRecordService _recordService;
        private readonly IRecordMapper _recordMapper;

        public RecordsController(
                 IRecordService recordService,
                 IRecordMapper recordMapper)
        {
            this._recordService = recordService;
            this._recordMapper = recordMapper;
        }

        [HttpGet("allRecordDetails")]
        public ActionResult<List<Record>> GetAllInformationCategories()
        {
            var record = _recordService.GetRecords();
```

```
            return Ok(record);
        }

        [HttpGet("{id}")]
        public async Task<ActionResult> GetByIdAsync([FromRoute] int id)
        {
            var records = await _recordService.GetRecordByIdAsync(id);
            if (records == null)
            {
                return NotFound("There's no record with the provided id.");
            }

            return Ok(records);
        }

        [HttpGet("recordByCategoryId/{categoryId}")]
        public ActionResult<List<Record>> GetByCategoryIdAsync([FromRoute] int categoryId)
        {
            var records =  _recordService.GetRecordsByCategoryId(categoryId);
            if (records == null)
            {
                return NotFound("There's no category with the provided category id.");
            }

            return Ok(records);
        }

        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteAsync([FromRoute] int id)
        {
            var result = await _recordService.GetRecordByIdAsync(id);
            await _recordService.DeleteAsync(result);
            return NoContent();
        }

        [HttpPost]
        public async Task<ActionResult<Record>> CreateAsync(CreateRecordRequest
createRecordRequest)
        {
            var newRecord = _recordMapper.mapToRecord(createRecordRequest);
            var addedRecord = await _recordService.CreateAsync(newRecord);

            return Created(newRecord.Id.ToString(), addedRecord);
        }

        [HttpPut("{id}")]
        public async Task<IActionResult> UpdateAsync([FromRoute] int id, UpdateRecordRequest
updatedRecordRequest)
        {
            var recordToUpdate = await _recordService.GetRecordByIdAsync(id);
            if (recordToUpdate == null)
            {
                return NotFound("There is no record with the provided id.");
            }

            var updatedRecord = _recordMapper.MapToUpdatedRecord(recordToUpdate,
updatedRecordRequest, id);
            await _recordService.UpdateAsync(updatedRecord);
            return Ok(updatedRecord);
        }
    }
}
```

//BUSINESS LAYER

```
namespace LookApp.Business
{
    public class RecordService : IRecordService
    {
        private readonly LookAppContext _context;

        public RecordService(LookAppContext dbContext)
        {
            this._context = dbContext;
        }

        public List<Record> GetRecords()
        {
            return _context.Records.ToList();
        }

        public List<Record> GetRecordsByCategoryId(int id)
        {
            return _context.Records
                .Where(r => r.CategoryId == id)
                .OrderByDescending(r => r.Date)
                .ToList();
        }

        public async Task<Record> GetRecordByIdAsync(int id)
        {
            return await this._context.Records.FindAsync(id);
        }

        public async Task<Record> CreateAsync(Record newRecord)
        {
            await this._context.Records.AddAsync(newRecord);
            await this._context.SaveChangesAsync();

            return newRecord;
        }

        public async Task DeleteAsync(Record recordToDelete)
        {
            if (recordToDelete != null)
            {
                this._context.Records.Remove(recordToDelete);
            }

            await this._context.SaveChangesAsync();
        }

        public async Task UpdateAsync(Record updatedRecord)
        {
            if (updatedRecord == null)
            {
                return;
            }

            this._context.Records.Update(updatedRecord);
            await this._context.SaveChangesAsync();
        }
```

```
}
}


namespace LookApp.Business
{
    public class CategoryService : ICategoryService
    {
        private readonly LookAppContext _context;

        public CategoryService(LookAppContext dbContext)
        {
            this._context = dbContext;
        }

        public List<Category> GetCategories(int userId)
        {

            return _context.Categories
                .Where(c => c.CreatorId == userId)
                .Include(c => c.Records)
                .ToList();
        }

        public async Task<Category> GetCategoryByIdAsync(int id, int userId)
        {
            var startDate = DateTime.UtcNow.Subtract(TimeSpan.FromDays(7));
            return await this._context.Categories.Include(c =>
c.Records).FirstOrDefaultAsync(c => c.Id == id && c.CreatorId == userId);
        }

        public async Task<Category> CreateAsync(Category newCategory)
        {
            await this._context.Categories.AddAsync(newCategory);
            await this._context.SaveChangesAsync();

            return newCategory;
        }

        public async Task DeleteAsync(Category categoryToDelete)
        {
            if (categoryToDelete == null)
            {
                return;
            }

            this._context.Categories.Remove(categoryToDelete);
            await this._context.SaveChangesAsync();
        }


        public async Task UpdateAsync(Category updatedCategory)
        {
            if (updatedCategory == null)
            {
                return;
            }

            this._context.Categories.Update(updatedCategory);
            await this._context.SaveChangesAsync();
        }
    }
```

```
}
```

//DATABASE LAYER

```
namespace LookApp.Database.Models
{
    public class Category
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string UnitOfMeasure { get; set; }
        public double? LowerLimit { get; set; }
        public double? UpperLimit { get; set; }

        public string? GraphColor { get; set; }
        public ICollection<Record> Records { get; set; }

        public int CreatorId { get; set; }
        public User Creator { get; set; }
    }
}

namespace LookApp.Database.Models
{
    public class Record
    {
        public int Id { get; set; }
        public DateTime Date { get; set; }
        public string Note { get; set; }
        public double Value { get; set; }

        public int CategoryId { get; set; }
        public Category Category { get; set; }
    }
}


namespace LookApp.Database.Models
{
    public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Email { get; set; }
        public string Password { get; set; }
    }
}

namespace LookApp.Database.Models
{
    public sealed class LookAppContext : DbContext
    {
        public LookAppContext(DbContextOptions<LookAppContext> options) : base(options)
        {
            Database.Migrate();
        }
public DbSet<Category> Categories { get; set; }
        public DbSet<Record> Records { get; set; }
        public DbSet<User> Users { get; set; }
```

```
    }
}
```

Anexa 2. Front-end

```
export class AuthenticationService {

 public loggedInUser: LoggedInUser;

 private readonly endpoint: string = 'https://localhost:44387/api/authentication';

 constructor(private readonly http: HttpClient) {
  this.loggedInUser = JSON.parse(sessionStorage.getItem('loggedInUser'));
 }

 login(email: string, password: string): Observable<LoggedInUser> {
  return this.http.post<LoggedInUser>(`${this.endpoint}/login`, { email, password })
   .pipe(
    map(loggedInUser => {
     sessionStorage.setItem('loggedInUser', JSON.stringify(loggedInUser));
     this.loggedInUser = loggedInUser;
     return loggedInUser;
    })
   );
 }

 register(username: string, email: string, password: string): Observable<void> {
  return this.http.post<void>(`${this.endpoint}/register`, { username, email, password });
 }

 logout(): void {
  sessionStorage.removeItem('loggedInUser');
  this.loggedInUser = null;
 }

 forgotPassword(email: string, newPassword: string){
  return this.http.post(`${this.endpoint}/changePassword`, { email, newPassword });
 }
}

export class RecordService {

 private readonly recordsPerCategory: Map<number, BehaviorSubject<SortedList<RecordModel>>>;

 constructor(private readonly recordRepositoryService: RecordRepositoryService) {
  this.recordsPerCategory = new Map<number, BehaviorSubject<SortedList<RecordModel>>>();
 }

 addRecord(recordModel: RecordModel): Observable<RecordModel> {
```

```
  return this.recordRepositoryService.addRecord(recordModel)
    .pipe(
     map(addedRecord => {
      let records = this.recordsPerCategory.get(recordModel.categoryId)
      if (records) {
       let updatedRecords = SortedList.copy(records.value);
       updatedRecords.add(addedRecord);
       records.next(updatedRecords);
      }

      return addedRecord;
     })
    );
}

deleteRecord(categoryId: number, id: number): Observable<void> {
  return this.recordRepositoryService.deleteRecord(id)
    .pipe(
     map(() => {
      let records = this.recordsPerCategory.get(categoryId);
      if (records) {
       let updatedRecords = SortedList.copy(records.value);
       updatedRecords.delete(r => r.id !== id);
       records.next(updatedRecords);
      }
     })
    );
}


populateRecords(categoryId: number): Observable<Observable<SortedList<RecordModel>>> {
  return this.recordRepositoryService.getRecordsByCategoryId(categoryId)
    .pipe(
     map(records => {
      let newRecords = new SortedList(records, this.orderByDateDescending);
      let categoryRecords = new BehaviorSubject<SortedList<RecordModel>>(newRecords);
      this.recordsPerCategory.set(categoryId, categoryRecords);
      return categoryRecords.asObservable();
     })
    );
}

getRecordsByCategoryId(categoryId: number): Observable<Observable<SortedList<RecordModel>>> {
  let records = this.recordsPerCategory.get(categoryId);
  if (!records) {
   return this.populateRecords(categoryId);
  }
  return of(records);
```

```
  }

  orderByDateDescending(a: RecordModel, b: RecordModel) {
    // convert date object into number to resolve issue in typescript
    return +new Date(b.date) - +new Date(a.date);
  }

  updateRecord(recordId: number, recordModel: RecordModel): Observable<void> {
    return this.recordRepositoryService.updateRecord(recordId, recordModel)
      .pipe(
        map(updatedCategory => {
          let records = this.recordsPerCategory.get(recordModel.categoryId);
          if (records) {
            let updatedRecords = SortedList.copy(records.value);
            updatedRecords.delete(r => r.id !== recordId);
            updatedRecords.add(updatedCategory)
            records.next(updatedRecords);
          }
        })
      );
  }
}


export class CategoryService {

  private readonly categories: BehaviorSubject<CategoryModel[]>;

  constructor(private categoryRepositoryService: CategoryRepositoryService) {
    this.categories = new BehaviorSubject<CategoryModel[]>([]);
  }

  populateCategories(): Observable<Observable<CategoryModel[]>> {
    return this.categoryRepositoryService.getAllCategories()
      .pipe(
        map(categories => {
          this.categories.next(categories);
          return this.categories.asObservable();
        })
      );
  }

  getById(categoryId: number): Observable<CategoryModel> {
    let category = this.categories.value.find(category => category.id === categoryId);
    if (!category) {
      return this.categoryRepositoryService.getById(categoryId);
    }
    return of(category);
  }
```

```
  addCategory(categoryModel: CategoryModel): Observable<void> {
   return this.categoryRepositoryService.addCategory(categoryModel)
    .pipe(
     map(addedCategory => {
      this.categories.next([...this.categories.value, addedCategory]);
     })
    );
  }

  updateCategory(categoryId: number, categoryModel: CategoryModel): Observable<void> {
   let updatedCategoryList = [];
   return this.categoryRepositoryService.updateCategory(categoryId, categoryModel)
    .pipe(
     map(updatedCategory => {
      this.categories.value.forEach(element => {
       if (element.id == updatedCategory.id) {
        updatedCategoryList.push(updatedCategory)
       } else {
        updatedCategoryList.push(element)
       }
      });
      this.categories.next(updatedCategoryList);
     }
     )
    )
  }

  deleteCategory(id: number): Observable<void> {
   return this.categoryRepositoryService.deleteCategory(id)
    .pipe(
     map(() => {
      this.categories.next(this.categories.value.filter(c => c.id !== id));
     })
    );
  }
}

export class ChartService {

  constructor() { }

  public isInBounds(chart: Chart, coordinates: { x: number, y: number }): boolean {

   const left = chart.chartArea.left;
   const top = chart.chartArea.top;
   const right = chart.chartArea.right;
   const bottom = chart.chartArea.bottom;
```

```
   if (coordinates.x < left || coordinates.x > right) return false;
   if (coordinates.y < top || coordinates.y > bottom) return false;

   return true;
  }
}


export class CorrelationService {

  public correlationCoeff : number = undefined;

  public groupRecordsByDay(chartDataSetToCorrelate: ChartPointModel[], categoryId: number, categoryTit
le: string): RecordsByDay {

   let groupedDataSet: RecordsByDay = chartDataSetToCorrelate.reduce(
    (acc: RecordsByDay, record: ChartPointModel) => {

     let day: string = record.x.toString().split("T")[0];
     let val: number = record.y;

     if (acc.recordsByDay[day]) {
      acc.recordsByDay[day] += val;
     } else {
      acc.recordsByDay[day] = val;
     }

     return acc;

    }, new RecordsByDay(categoryId, categoryTitle));

   return groupedDataSet;
  }


  getValuesOnSameDate(firstDataSet: RecordsByDay, secondDataSet: RecordsByDay): CategoriesToCorrelate[
] {

   let firstDataSET = JSON.parse(JSON.stringify(firstDataSet));
   let secondDataSET = JSON.parse(JSON.stringify(secondDataSet));

   let categoriesToCorrelate: CategoriesToCorrelate[] = [];

   for (const [key, value] of Object.entries(secondDataSET.recordsByDay)) {
    if (firstDataSET.recordsByDay[key] == undefined) {
     delete secondDataSET.recordsByDay[key];
    }
   }
```

```
  for (const [key, value] of Object.entries(firstDataSET.recordsByDay)) {
    if (secondDataSET.recordsByDay[key] == undefined) {
      delete firstDataSET.recordsByDay[key];
    }
  }

  categoriesToCorrelate.push({ "firstCategory": firstDataSET, "secondCategory": secondDataSET });

  return categoriesToCorrelate;
}

calculateCorrelationCoef(categoriesToCorrelate: CategoriesToCorrelate[]): number {
  let sumOfX: number = 0;
  let sumOfY: number = 0;

  let x: number = 0;
  let y: number = 0;

  let sumOfXMultipliedWithY: number = 0;

  let sumOfSquareX = 0;
  let sumOfSquareY = 0;

  let n = 0;

  for (const [key, value] of Object.entries(categoriesToCorrelate[0].firstCategory.recordsByDay)) {
    x = value;
    sumOfX += x;

    y = categoriesToCorrelate[0].secondCategory.recordsByDay[key];
    sumOfY += y;

    sumOfXMultipliedWithY += x * y;

    sumOfSquareX += x * x;
    sumOfSquareY += y * y;
  }
  n = Object.getOwnPropertyNames(categoriesToCorrelate[0].firstCategory.recordsByDay).length;

  let numitor: number = Math.sqrt((n * sumOfSquareX - sumOfX * sumOfX) * (n * sumOfSquareY - sumOfY
* sumOfY));
  let numarator: number = (n * sumOfXMultipliedWithY - sumOfX * sumOfY);
  this.correlationCoeff = numarator / numitor;

  if(numitor == 0){
    this.correlationCoeff = 0;
  }
```

```
    return this.correlationCoeff;
  }

  toChartPoints(records: RecordModel[]): ChartPointModel[] {

    return records.map((record: RecordModel) => {
      return new ChartPointModel(record.date.toString(), record.value);
    });
  }

}
export class CommandService {

  private readonly commandHistory: Command[] = [];
  private readonly commandCount: BehaviorSubject<number> = new BehaviorSubject<number>(0);

  public readonly commandCount$: Observable<number> = this.commandCount.asObservable();

  public do(command: Command): void {

    command.do();
    this.commandHistory.push(command);
    this.commandCount.next(this.commandHistory.length);
  }

  public undo(): void {

    let lastCommand = this.commandHistory.pop();
    if (lastCommand != undefined) {
      lastCommand.undo();
      this.commandCount.next(this.commandHistory.length);
    }
  }
}
export interface Command {
  do(): void;
  undo(): void;
}

export class AddRecordCommand implements Command {
  private recordService: RecordService;
  private recordModel: RecordModel;

  constructor(recordService: RecordService, recordModel: RecordModel) {
    this.recordService = recordService;
    this.recordModel = recordModel;
  }
```

```
  public do(): void {

    this.recordService.addRecord(this.recordModel).subscribe(addedRecord => {
      this.recordModel.id = addedRecord.id;
    });
  }

  public undo(): void {

    this.recordService.deleteRecord(
      this.recordModel.categoryId,
      this.recordModel.id).subscribe();
  }
}

export class ImportRecordCommand implements Command {

  private csvRecords: any[];
  private categoryId: number;
  private recordService: RecordService;
  private recordModel: RecordModel[];


  constructor(recordService: RecordService, recordModel: RecordModel[], categoryId: number) {
    this.categoryId = categoryId;
    this.recordService = recordService;
    this.recordModel = recordModel;
  }

  public do(): void {

    for (let i = 0; i < this.recordModel.length; i++) {
      let index = i;
      this.recordService.addRecord(this.recordModel[index]).subscribe(addedRecord => {
        this.recordModel[index].id = addedRecord.id;
      });
    }
  }

  public undo(): void {

    for (let i = 0; i < this.recordModel.length; i++) {
    this.recordService.deleteRecord(
      this.categoryId,
      this.recordModel[i].id).subscribe();
    }
  }
}
```