UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Emergency Management System (EMS)

propusă de

Alexandru Dochițoiu

Sesiunea: *februarie, 2019*

Coordonator științific

Prof. Colab. Florin Olariu

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Emergency Management System (EMS)

Alexandru Dochițoiu

Sesiunea: *februarie, 2019*

Coordonator științific

Prof. Colab. Florin Olariu

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și prenumele _____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemnatul (a)……
domiciliul în ……..
născut(ă) la data de ....., identificat prin CNP …….……..……...,
absolvent(a) al(a) Universității „Alexandru Ioan Cuza" din Iași, Facultatea de ……. specializarea ……, promoția ……., declar pe propria răspundere, cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licență cu titlul:

_____

_____

_____

_____elaborată sub îndrumarea dl. / d-na_____,
pe care urmează să o susțină în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime. De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului său într-o bază de date în acest scop. Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice in vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diploma sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

DECLARAŢIE PRIVIND ORIGINALITATE ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul „Emergency Management System (EMS)” este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau din străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința precisă;
- codul sursă, imaginile etc. preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de autor și dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Iaşi, Absolvent *Alexandru Dochițoiu*

_____

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „Emergency Management System (EMS)”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iaşi, Absolvent *Alexandru Dochițoiu*

_____

Cuprins

1. Introducere 7

1.1 Ce este EMS? 7

1.2 Motivație 7

1.3 Aplicații asemănătoare 8

1.4 Funcționalități 9

1.5 Contribuții 10

2. Arhitectura proiectului 11

2.1 Domain-Driven-Design 11

2.2 Structura proiectului 12

2.3 Autentificare și autorizare 17

2.4 Design Pattern-uri 17

2.4.1 Repository 18

2.4.2 UnitOfWork 19

2.4.3 Factory 20

2.4.4 Dependency Injection 21

2.5 Principiile SOLID 22

3. Concluzii 23

4. Anexe 24

Tehnologii folosite 24

Anexa 1: Server 24

ASP.NET Core 2 25

Entity Framework Core 26

ASP.NET Identity 26

AutoMapper 27

LINQ 27

Anexa 2: Client 28

Angular, TypeScript și npm 28

HTML și CSS 29

5. Bibliografie 30

5.1 Resurse 31

1. Introducere

1.1 Ce este EMS?

Fig. 1
(Sigla aplicației Emergency Management System)

Emergency Management System (EMS) este o aplicație web care permite informarea și interacțiunea cu populația în cazul unei situații de urgență. Utilizatorii acestei platforme pot fi notificați prin intermediul mesajelor sms și a e-mail-ului. De asemenea, aceștia au posibilitatea de a accesa anumite ghiduri de supraviețuire, anunțuri publicate de autorități, de a raporta incidente la locația curentă sau selectând un punct anume folosind harta și multe alte funcții ajutătoare în cazul unui dezastru.

1.2 Motivație

În fiecare an, dezastrele naturale cum ar fi alunecările de teren, inundațiile, furtunile sau cutremurele aduc pagube economice semnificative și deseori pierderi umane. Pe lângă fenomenele naturale inevitabile se pot enumera și tragediile provocate de mâna omului: atacuri teroriste, accidente nucleare, accidente rutiere etc. Efectele dăunătoare pe care aceste fenomene le au asupra populației, mediului înconjurător și bunurilor materiale fac necesară cunoașterea acestor fenomene și a modului în care putem preveni, sau ne putem apăra în caz de urgență.

În tabelul de mai jos (*fig. 2)* sunt prezentate cele mai grave 10 tragedii din punct de vedere economic:

Top 10 Global Economic Loss Events

Date(s)	Event	Location	Deaths	Economic Loss (USD)	Insured Loss (USD)
Aug. 25 – Sept. 2	Hurricane Harvey	United States	90	~100 billion	~30 billion
September 18-22	Hurricane Maria	Caribbean Islands	Hundreds+	~65 billion	~27 billion
September 4-12	Hurricane Irma	U.S., Caribbean Islands	134	~55 billion	~23 billion
October	Wildfires	United States	43	13 billion	11 billion
Summer	Flooding	China	116	7.5 billion	300 million
Summer & Autumn	Drought	Southern Europe	N/A	6.6 billion	700 million
September 19	Earthquake	Mexico	370	4.5 billion	1 billion
July	Flooding	China	37	4.5 billion	125 million
August 23-25	Typhoon Hato	China	22	3.5 billion	250 million
May 8-11	Severe Weather	United States	0	3.4 billion	2.6 billion
			All Other Events	90 billion	38 billion
			Totals	**353 billion[1]**	**134 billion[1,2]**

Fig. 2[1]

Pe lângă pierderile economice, numărul de decese cauzate de aceste dezastre naturale este unul îngrijorător: Uraganul Maria care s-a dezlănțuit în perioada 18-22 septembrie 2017 a provocat sute de victime în Insulele Caraibe, cutremurul care a lovit Mexicul în aceeași perioadă a provocat 370 de decese și mii de răniți, iar exemplele pot continua.

Așadar, am decis să abordez tema – gestionarea situațiilor de urgență pentru lucrarea mea de licență deoarece poate veni în ajutorul utilizatorilor în cazul unui hazard. În repetate rânduri, natura s-a dezlănțuit și ne-a arătat forța ei de distrugere, echipajele de salvare fiind depășite de situație. Folosind această aplicație, cetățenii vor fi la curent cu toate informațiile legate de vreme, accidente, atacuri sau orice alertă transmisă de către autorități sau alți cetățeni.

1.3 Aplicații asemănătoare

În momentul actual, pe piață exista o serie de aplicații care deservesc gestionarii situațiilor de urgență, dar aceste aplicații vin cu anumite limitări, cum ar fi: sunt dependente de platformă (aplicații care rulează doar pe sistemele de operare Android și iOS), în cazul raportării unui incident este necesară inclusiv descrierea locului unde s-a provocat incidentul, fapt ce presupune acordarea unui timp mai îndelungat

[1] Dezastrele naturale din 2017 - http://www.1asig.ro/Dezastrele-naturale-din-2017-Pagube-record-pentru-asiguratori-articol-13,90-58289.htm

Una dintre cele mai cunoscute aplicații pentru situații de urgență este DSU[2] publicată chiar de Ministerul Afacerilor Interne. Este disponibilă gratuit în App Store și Google Play și are patru categorii – informare, alertare, raportare și învățare. Aceasta pune le dispoziție informații legate de comportamentul în cazul cutremurelor, inundațiilor, incendiilor dar, totodată, și tehnici care ar putea fi folositoare în acordarea primului ajutor.

1.4 Funcționalități

- **Înregistrare și autentificare**:

Utilizatorii au posibilitatea de a înregistra un cont folosind formularul oferit de aplicație, fiind nevoie să complete următoarele câmpuri: numele de cont, adresa de e-mail, numărul de telefon și parola. După ce utilizatorul și-a înregistrat datele, un mail este trimis automat în vederea activării contului. De asemenea, ei pot opta și pentru autentificarea folosind rețelele sociale: Facebook, Google și Twitter. În cazul în care un utilizator și-a uitat parola, aplicația oferă suport pentru recuperarea acesteia folosind mail-ul.

- **Căutare de incidente:**

Incidentele sunt plasate pe hartă sub forma de pin-uri colorate în funcție de gravitatea incidentului. Utilizatorul poate vedea detaliile despre incident apăsând pe pin, urmând să apară o fereastră cu informații. Fereastra conține informații legate de: titlu, descriere, severitate, de către cine a fost raportat incidentul, respectiv data și ora la care a fost raportat. Pin-ul de culoarea roșie reprezintă incidentele de severitate critică, iar cele de culoare portocalie și galben sunt pentru incidente majore, respectiv minore. (*Fig. 3*)

Fig. 3

Aplicația salvează poziția curentă a utilizatorului și îi va arăta pe hartă toate incidentele raportate în apropiere, pe o rază de 1.5 km. Ulterior, raza se poate mări sau micșora iar incidentele pot fi filtrate pe baza unui formular. Raza poate lua o valoare între 0.5 și 25 de km și incidentele se pot filtra după gradul de severitate, titlu, descriere sau după numele utilizatorului care a raportat incidentul. Utilizatorii pot accesa și incidentele raportate în altă locație, selectând țara și orașul.

[2] DSU este aplicația oficială a Departamentului pentru Situații de Urgență din cadrul Ministerului Afacerilor Interne. - http://www.dsu.mai.gov.ro/

- **Raportare incident:**

Un utilizator poate raporta un incident apăsând pe butonul “Report an incident” plasat deasupra hărții. Butonul îl va redirecționa către un formular, unde va trebui să completeze următoarele câmpuri: “Summary”, “Description” și “Severity”, să selecteze dacă incidentul s-a petrecut la poziția curentă (sau să aleagă locația manual, selectând o locație pe hartă) și, opțional, să încarce fotografii. După ce incidentul a fost raportat, toți utilizatorii pe o anumită rază sunt alertați prin mesaje SMS. În funcție de gravitatea incidentului raza ia următoarele valori: 3km pentru incidentele critice, 1.5km pentru cele majore și 500m pentru cele minore. De asemenea, ulterior această rază poate fi modificată de către utilizator.

- **Rute către spitale apropiate:**

Persoanele aflate în preajma unui incident, se pot folosi de o opțiune oferită de aplicație pentru a ajunge la cel mai apropiat spital sau refugiu. Apăsând butonul “Nearby hospitals”, pe hartă vor apărea toate spitalele din apropiere marcate cu pin-ul din *fig. 4* și o rută către un punct de siguranță. Utilizatorul este informat, de asemenea, și cu detalii legate de durata și lungimea drumului.

Fig. 4

- **Accesul la anunțuri și ghiduri publicate de autorități:**

Autoritățile pot publica anunțuri, ghiduri de supraviețuire, lecții de prim ajutor sau orice document ajutător pentru gestionarea unei situații de urgență. Utilizatorii pot accesa aceste informații, folosind meniul principal la secțiunile: “Announcements” și „Guides”.

1.5 Contribuții

Am dezvoltat o aplicație web cu design responsive independentă de platformă care permite utilizatorilor să fie la curent cu toate evenimentele petrecute în jurul lor. Cetățenii au posibilitatea de a raporta un incident la poziția curentă iar pe o anumită rază, utilizatorii sunt atenționați în timp real prin intermediul mesajelor sms. După o analiză amănunțită asupra aplicațiilor existente pentru gestionarea situațiilor de urgență am observat că în cazul raportării unui incident este necesară detalierea în scris a locului unde s-a petrecut incidentul, iar acest lucrul necesită timp prețios. Așadar, am decis să vin cu o metodă mai rapidă pentru raportarea unui incident folosindu-mă de serviciile oferite de Google Maps.

Prin urmare, cu ajutorul API-ului oferit de Google Maps, aplicația oferă utilizatorilor în stare de alertă rute către cele mai apropiate spitale sau refugii, indicând ruta optimă și durata până la destinație în funcție de mijlocul de transport.

Aplicația vine și la îndemâna autorităților, având o multitudine de funcționalități: pot publica anunțuri indicând gradul de severitate și alerta cetățenii dintr-un oraș sau chiar la nivel național, sunt puși la curent de evenimentele întâmplate, urmând ca ei să acționeze rapid, pot informa utilizatorii publicând ghiduri de supraviețuire, tehnici de prim ajutor și alte informații prețioase în cazul unei calamități.

2. Arhitectura proiectului

Aplicația pentru care am optat este fără îndoiala una complexă, prin urmare am decis să modelez arhitectura proiectului după un set de reguli și principii bine definite. Un design bun oferă posibilitatea de a duce proiectul mai departe cu ușurință indiferent de complexitatea cerințelor ulterioare.

2.1 Domain-Driven-Design

Ideea de domain-driven-design (DDD) a fost inițial introdusă de programatorul Eric Evans în cartea scrisă de el "*Domain-Driven Design: Tackling Complexity in the Heart of Software"* și are ca scop, în principal, plasarea atenției asupra centrului (domeniului) aplicației. Acest concept este constituit dintr-o serie de design pattern-uri și principii de programare. Eric Evans definește câțiva termeni care trebuie luați în vedere când aplicăm practicile DDD: *Context, Domain, Model, Ubiquitous Language.*

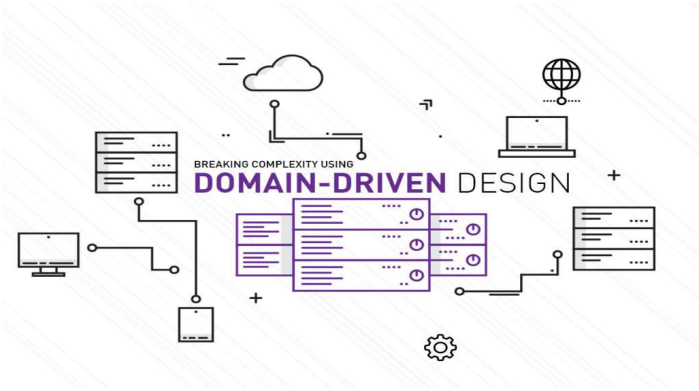

Fig. 5 (Domain-Driven Design)[4]

[4] DDD (Domain-Driven Design) - https://www.mitrais.com/news-updates/breaking-complexity-using-domain-driven-design/

2.2 Structura proiectului

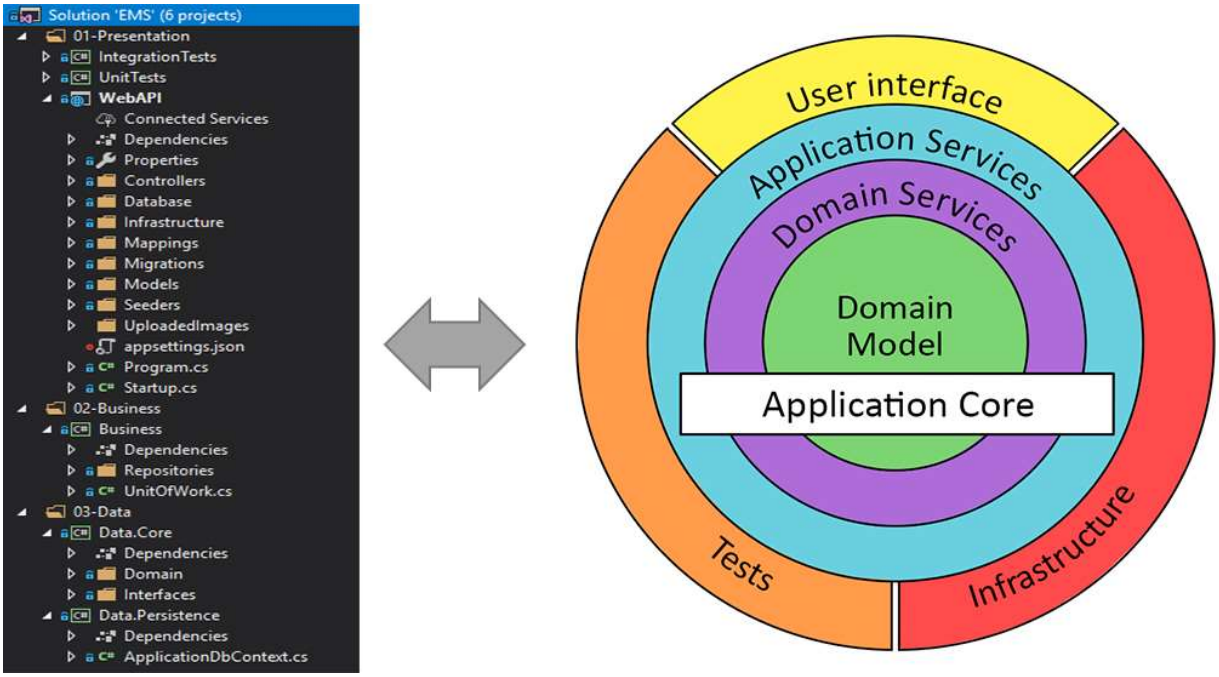

Fig. 6 (Arhitectura Onion)[5]

Structura proiectului este bazată pe arhitectura Onion (Fig. 6). Soluția proiectului este formată din 6 proiecte structurate astfel:

1. **03-Data –** conține proiectele **Data.Core** și **Data.Persistence** ele reprezentând primul, respectiv al doilea nivel din arhitectura Onion. În **Data.Core** este definit domeniul aplicației (entitățile aplicației – pe lângă cele moștenite din ASP.NET Identity, au mai fost create următoarele: Address, Announcement, City, Country, Guide, Incident, BaseEntity, Gender și Severity). Tot aici se găsesc și interfețele pentru clasele definite în nivele care țin *core*-ul aplicației (în cazul meu, interfețele pentru *Repository* și *UnitOfWork – design pattern*-uri ce vor fi descrise mai jos). În **Data.Persistence** este definit contextul aplicației, mai exact, aici este definită structura bazei de date. Clasa *ApplicationDbContext* definește fiecare tabel care va fi creat de către Entity Framework sub forma de *DbSet*-uri. Exemplu de cod:

```
public DbSet<Incident> Incidents { get; set; }
```

2. **02-Business** – folder care este constituit din proiectul **Business** și reprezintă cel de-al treilea nivel din arhitectura Onion. Aici sunt implementate atât clasele *Repository*

[5] Onion Architecture - https://dzone.com/articles/onion-architecture-is-interesting

pentru fiecare entitate cât și un *Repository* generic, unde sunt definite operațiile reutilizabile pentru fiecare *Repository*, cum ar fi: *GetAll (...), Find (...), Add (...), Edit (...), Delete (...)*, etc. Mai jos (în *Fig. 7*) este prezentat codul interfeței pentru Repository-ul generic, pentru a vedea mai exact ce metode am definit:

```
public interface IGenericRepository<T> where T : class
{
    Task<IEnumerable<T>> GetAllAsync<TOrderKey>(Func<IQueryable<T>, IQueryable<T>> load,
        Expression<Func<T, TOrderKey>> orderBy = null);
    Task<IEnumerable<T>> GetAllAsync<TOrderKey>(Expression<Func<T, TOrderKey>> orderBy = null);
    Task<IEnumerable<T>> FindAsync(Expression<Func<T, bool>> predicate,
        Func<IQueryable<T>, IQueryable<T>> load);
    Task<IEnumerable<T>> FindAsync(Expression<Func<T, bool>> predicate);
    Task<T> AddAsync(T entity);
    Task AddRangeAsync(IEnumerable<T> entities);
    Task<T> EditAsync(T entity, object key);
    void Delete(T entity);
    void DeleteRange(IEnumerable<T> entities);
}
```

Fig. 7 (Interfața *IGenericRepository*)

3. **01-Presentation -** reprezentat de ultimul nivel din arhitectura Onion. Acest folder conține trei proiecte: **IntegrationTests**, **UnitTests** și **WebAPI**, primele două fiind destinate testării, iar cel de-al treilea este proiectul MVC (pattern arhitectural) translatat astfel: *Model* – conține clasele ce reprezintă domeniul aplicației (ele sunt definite în primul nivel din Onion dar ajung în nivelul de prezentare sub forma de clase DTO), *View* – fiind proiect de tip Web API, acesta este reprezentat prin returnarea obiectele JSON generate de controlere, *Controller* – este o clasă care implementează Controller din modulul Microsoft.AspNetCore.Mvc și gestionează relațiile dintre View și Model.

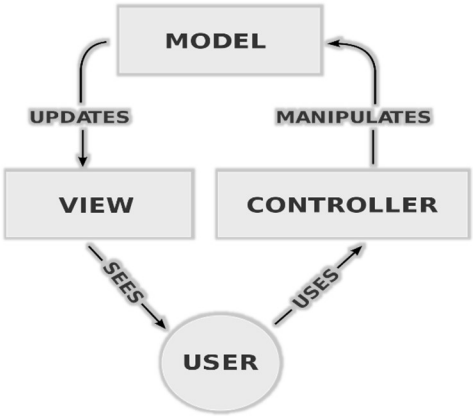

Fig. 8 (MVC) [6]

[6] MVC (Architectural Pattern) - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

În Web API au fost definite următoarele *Controllere* cu scopul de a expune servicii la diferite endpoint-uri:

- *AccountController.cs* - are scopul de a realiza înregistrarea și autentificarea utilizatorilor și expune următoarele endpoint-uri:
    - */api/account/register* prin metoda POST – verifică corectitudinea datelor introduce, salvează utilizatorul în baza de date și trimite un e-mail de confirmare.
    - */api/account/verify/email/{userId}/{emailToken}* prin metoda GET – are ca scop confirmarea e-mail-ului pentru utilizatorul cu ID-ul *userId* folosind token-ul *emailToken* generat la înregistrare.
    - */api/account/forgot-password* prin metoda POST – trimite un e-mail cu posibilitatea de a recupera parola.
    - */api/account/reset/password/{userId}/{resetPasswordToken}* prin metoda POST – accesând acest endpoint, utilizatorul își poate reseta parola în cazul în care token-ul *resetPasswordToken* este valid. Acest endpoint este trimis pe e-mail în urma folosiri endpoint-ului descris mai sus */api/account/forgot-password.*
    - */api/account/resend-verification-mail* prin metoda POST – endpoint folosit pentru retrimiterea e-mail-ului de confirmare.
    - */api/account/login* prin metoda POST – la acest endpoint este verificat dacă un utilizator există în baza de date și a confirmat e-mailul. În cazul în care utilizatorul este înregistrat și a introdus parola corectă el va fi autentificat și își va actualiza locația curentă.
    - */api/account/login/external* prin metoda POST – oferă suport pentru autentificarea și înregistrarea folosind rețelele sociale: Facebook, Google și Twitter.
- *AlertController.cs* – acest controller oferă un serviciu pentru alertarea cetățenilor. Expune un singur endpoint:
    - */api/alert/nearby* prin metoda POST – apelează la un API extern numit Twillio și are ca scop trimiterea de mesaje SMS tuturor utilizatorilor dintr-o anumită rază.
- *AnnouncementController.cs* – expune servicii care permit gestionarea anunțurilor. Au fost implementate două endpoint-uri pentru acest controller:

    - */api/announcements* prin metodele GET și POST – returnează toate anunțurile din baza de date, respectiv adaugă un anunț în baza de date.
    - */api/announcements/{id}* prin metodele GET, PUT și DELETE – returnează un anunț anume în funcție de id-ul acestuia, modifică un anunț existent, respectiv șterge un anunț.

- *CityController.cs* – controller folosit pentru accesarea orașelor din baza de date, având trei endpoint-uri:
    - */api/cities* prin metoda GET – returnează toate orașele din baza de date.
    - */api/cities/{id}* prin metoda GET – returnează un anumit oraș în funcție de id.
    - */api/cities/{name}* prin metoda GET – returnează un oraș în funcție de numele acestuia.

- *CountryController.cs* – utilizat pentru accesul la țările din baza de date. Acesta are următoarele endpoint-uri:
    - */api/countries* prin metoda GET – oferă toate țările din baza de date.
    - */api/countries/{id}* prin metoda GET – returnează o țară după id.
    - */api/countries/{name}* prin metoda GET – returnează o țară în funcție de numele acesteia.
    - */api/countries/{name}/cities* prin metoda GET – returnează orașele dintr-o anumită țară.

- *IncidentController.cs* – oferă servicii pentru manipularea incidentelor și expune următoarele endpoint-uri:
    - */api/incidents* prin metodele GET și POST – returnează toate incidentele din baza de date, respectiv oferă posibilitatea de a adăuga un nou incident.
    - */api/incidents/{id}* prin metoda GET – returnează un incident obținut în funcție de id.
    - */api/incidents/radius* prin metoda GET – returnează incidentele petrecute într-o anumită locație în funcție de o rază și un punct geografic.

- *UploadController.cs* – expune un serviciu pentru încărcarea de fișiere pe server având două endpoint-uri:
    - */api/upload* prin metoda POST – încarcă fotografii pe server prin corpul cererii sub formă de șir de caractere reprezentând imaginea în formatul base64. Acest

serviciu este folosit în special pentru raportarea de incidente, având posibilitatea de a încărca poze.

  - */api/upload/{incidentId}* prin metoda GET – returnează toate fotografiile unui incident.

- *UserController.cs* – controller destinat gestionării utilizatorilor din baza de date. Conține endpoint-urile următoare:

  - */api/users* prin metodele GET și POST – returnează toți utilizatorii din baza de date, respectiv adaugă un utilizator în baza de date.

  - */api/users/{id}* prin metodele GET, PUT și DELETE – returnează un utilizator anume, actualizează datele unui utilizator sau șterge un utilizator identificat prin id.

  - */api/users/{id}/announcements* prin metoda GET – returnează toate anunțurile publicate de un anumit utilizator.

  - */api/users/by-username/{username}* prin metoda GET – returnează un utilizator identificat prin numele de cont.

  - */api/users/{email}* prin metoda GET – returnează un utilizator în funcție de adresa de e-mail.

  - */api/users/radius* prin metoda GET – returnează toți utilizatorii dintr-o anumită locație. Acest serviciu este folosit pentru alertarea oamenilor din jurul unui incident tocmai raportat.

2.3 Autentificare și autorizare

Autentificarea și autorizarea securizată a aplicației a fost realizată folosind mecanismul oferit de *JSON Web Tokens* (JWT). Un JWT face posibilă verificarea deținătorului unor date în format JSON. Este un șir codificat, care poate conține o cantitate nelimitată de date și este semnat criptografic.

La autentificare, utilizatorului îi este generat un token pe baza datelor lui, acest token fiind salvat în *localStorage* (memoria browser-ului). Token-ul JWT este generat apelând metoda prezentată mai jos.

```
    public static string GenerateJwtToken (this ApplicationUser user)
    {
        var claims = new [] {
             new Claim(JwtRegisteredClaimNames.Jti,
    Guid.NewGuid().ToString()),
             new Claim(ClaimsIdentity.DefaultNameClaimType, user.UserName)
        };

        var credentials = new SigningCredentials(
             new SymmetricSecurityKey(Encoding.UTF8.GetBytes
  (IocContainer.Configuration ["JWTAuth:SecurityKey"])),
                 SecurityAlgorithms.HmacSha256);

        var token = new JwtSecurityToken (
                 issuer: IocContainer.Configuration["JWTAuth:Issuer"],
                 audience: IocContainer.Configuration["JWTAuth:Audience"],
                 claims: claims,
                 expires: DateTime.Now.AddMonths(1),
                 signingCredentials: credentials
         );
         return new JwtSecurityTokenHandler().WriteToken(token);
    }
```

2.4 Design Pattern-uri

În ingineria software, un șablon (*design pattern*) este o soluție care deseori se repetă în cazul problemelor frecvent întâlnite în proiectarea aplicațiilor software. Un *design pattern* este o descriere pentru rezolvarea unei probleme care poate fi utilizată în mai multe situații, ele nefiind structuri de date. Șabloanele au o multitudine de avantaje în ingineria software: reutilizarea codului sau arhitecturii, înțelegerea mai rapidă a soluției de către alți programatori, șabloanele definind un limbaj comun în domeniul software și sunt soluții bine documentate și testate.

Design pattern-urile se împart în trei categorii: șabloane creaționale (*creational design patterns*), șabloane structurale (*structural design patterns*) și șabloanele comportamentale (*behavioral design patterns*).

În realizarea arhitecturii proiectului meu am aplicat următoarele design pattern-uri: Generic Repository, Unit Of Work, Factory, Observable, Dependency Injection.

2.4.1 Repository

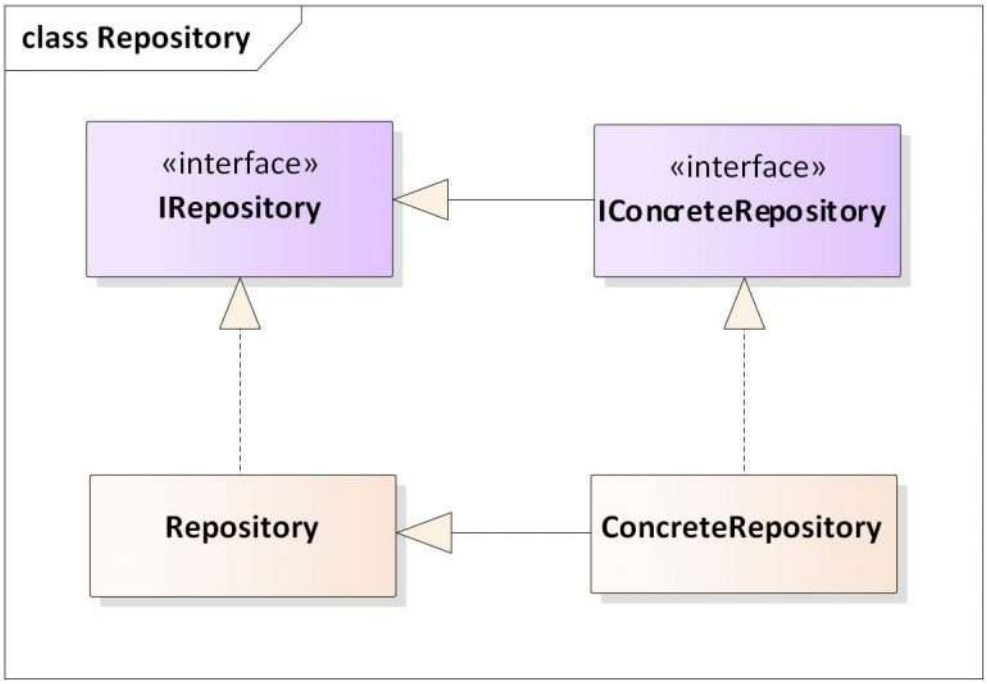

Fig. 9 (Repository Pattern)[7]

De-a lungul anilor *Repository Pattern* a devenit tot mai popular odată cu publicarea setului de reguli *Domain-Driven-Design* definite de Eric Evans în cartea sa în anul 2004. Șablonul *Repository* are în principal două roluri: oferă o abstractizare a nivelului de date și în același timp permite centralizarea și gestionarea obiectelor de domeniu. Adăugarea, eliminarea, actualizarea și selectarea elementelor dintr-o colecție de obiecte se realizează printr-o serie de metode simple, fără a fi necesară abordarea problemelor ce țin de baza de date cum ar fi conexiunile, comenzile, etc. Utilizând acest șablon codul scris respectă principiul cuplării reduse (*loose coupling*). Mai jos, puteți vedea un exemplu de Repository din codul aplicației mele, este vorba de Repository-ul care gestionează obiectele Incident:

[7] Repository Pattern Class Diagram - https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/

```
public class IncidentRepository : GenericRepository<Incident>,
IIncidentRepository
    {
        private readonly ApplicationDbContext _context;

        public IncidentRepository(ApplicationDbContext context) :
base(context)
        {
            _context = context;
        }

        public Task<Incident> GetByIdAsync(Guid id)
        {
            return _context.Incidents
                .Include(t => t.Reporter)
                .FirstAsync(a => a.Id == id);
        }

        public Task<List<Incident>> GetIncidentsWithinARadiusAsync(double
centerLat, double centerLng, double km)
        {
            return _context.Incidents
                .Include(t => t.Reporter)
                .Where(i => i.IsNear(centerLat, centerLng, km))
                .ToListAsync();
        }
    }
```

2.4.2 UnitOfWork

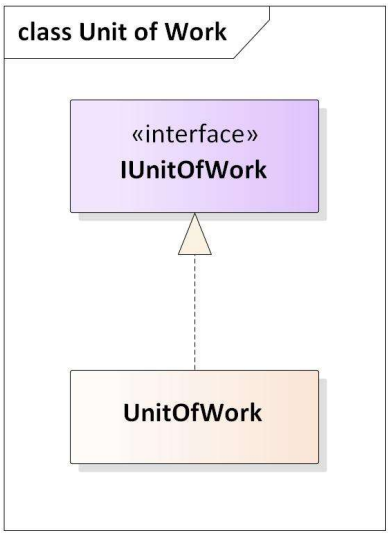

Fig. 10 (Unit Of Work Pattern)[8]

[8] Unit of Work Class Diagram - https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/

Șablonul *UnitOfWork* este utilizat în general împreună cu *Repository Pattern.* El are ca scop gruparea a una sau mai multe operațiuni (de obicei operațiuni pentru gestionarea bazei de date, cum ar fi: operații de inserare, actualizare sau ștergere) într-o singură tranzacție. Pentru înțelegerea acestui concept atașez mai jos corpul clasei UnitOfWork.cs din cadrul aplicației mele:

```
public class UnitOfWork : IUnitOfWork
{
    private readonly ApplicationDbContext _context;

    public IUserRepository Users { get; }
    public IAnnouncementRepository Announcements { get; }
    public IIncidentRepository Incidents { get; }
    public ICountryRepository Countries { get; }
    public ICityRepository Cities { get; }

    public UnitOfWork (ApplicationDbContext context)
    {
        _context = context;

        Users = new UserRepository(_context);
        Announcements = new AnnouncementRepository(_context);
        Incidents = new IncidentRepository(_context);
        Countries = new CountryRepository(_context);
        Cities = new CityRepository(_context);
    }

    public async Task<int> CompleteAsync()
       => await _context.SaveChangesAsync();

    public void Dispose() => _context.Dispose();
}
```

2.4.3 Factory

"Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses."[9]

Am folosit acest *design pattern* pentru implementarea entităților deoarece este specific conceptului de domain-driven-design (DDD). Instanțele obiectelor se realizează folosind metoda statică *Create()* în cadrul căreia se validează datele. Pentru validarea datelor am folosit librăria *EnsureThat*.

[9] Factory Pattern Definition (Gang of Four) - ***Design Patterns: Elements of Reusable Object-Oriented Software*** (1994)

2.4.4 Dependency Injection

Dependecy Injection este un design pattern realizat după principiul *Inversion Of Control*[10]. Acesta presupune faptul că ne oferă posibilitatea de a avea referințe la interfețe și nu la clasele concrete definite în interiorul aplicației. Clasele modelate folosind mecanismul de *Dependency Injection* sunt cuplate redus (respectă principiul *Loose Coupling*) deoarece nu comunică în mod direct cu alte clase. Există trei variante de a injecta dependențele: prin constructor, prin metode sau prin proprietăți. Metoda de injectare prin constructor este cea mai frecvent întâlnită, fiind de asemenea și metoda folosită de mine în realizarea aplicației mele.

Mecanismul aduce o serie de beneficii arhitecturii, cum ar fi: aplicația poate fi testată ușor, se respectă principiul *Depedency Inversion* din setul de principii *SOLID,* obiectele pot fi înlocuite cu ușurință.

În ASP.NET Core injectarea serviciilor de care aplicația are nevoie se poate realiza în trei modalități:

- *Transient* – o nouă instanță este creată când este nevoie de obiectul respectiv;
- *Scoped* – instanța obiectului este creată de fiecare dată când se realizează o cerere web;
- *Singleton* – obiectul este instanțiat doar o singură dată la pornirea aplicației;

Câteva exemple de servicii definite:

```
public static void AddUnitOfWork (this IServiceCollection services)
{
    services.AddScoped<IUnitOfWork, UnitOfWork>();
}

public static void AddTransientServices(this IServiceCollection services)
{
    services.AddTransient<CountriesSeeder>();
    services.AddTransient<CitiesSeeder>();
    services.AddTransient<AnnouncementsSeeder>();
    services.AddTransient<IDatabaseSeeder, DatabaseSeeder>();
    services.AddTransient<IEmailSender, SmtpEmailSender>();
    services.AddTransient<ISmsSender, EmsSmsSender>();
}
```

[10] Inversion Of Control Principle - https://en.wikipedia.org/wiki/Inversion_of_control

2.5 Principiile SOLID

SOLID este unul dintre cele mai cunoscute seturi de principii folosite în programarea orientată-obiect. SOLID este un acronim pentru cele cinci principii din OOD (*object-oriented design*) introduse de către Robert C. Martin. Atunci când aplicăm simultan aceste principii, dezvoltarea software devine mai ușor de realizat și de înțeles, fiind foarte flexibilă indiferent de numărul de programatori dintr-o echipă.

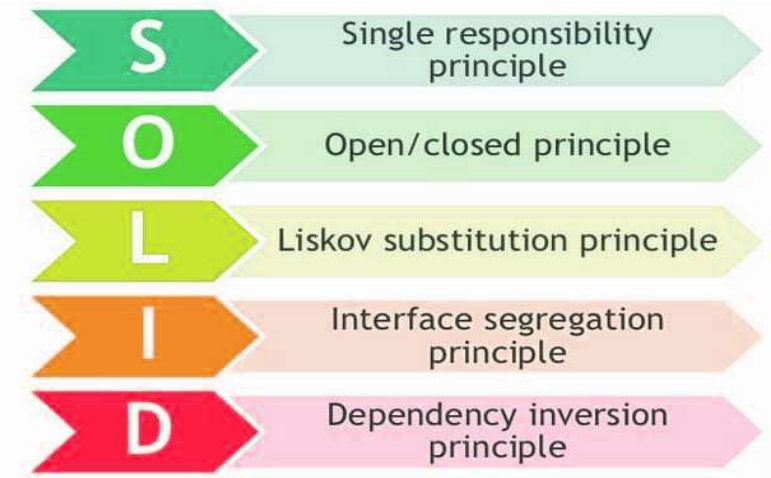

Fig. 11 (SOLID Principles)[11]

- ***Single Responsibility Principle*** (S.R.P) – acest principiu constă în faptul că o clasă trebuie sa aibă doar un singur motiv pentru a fi modificată;

- ***Open / Closed Principle*** – obiectele sau entitățile trebuie să fie deschise extensiei, dar închise pentru modificări;

- ***Liskov Substitution Principle –*** odată stabilit comportamentul clasei modificările care se fac asupra claselor din afara domeniul desemnat de obiect nu ar trebui sa impună modificarea comportamentului intern al clasei;

- ***Interface Segregation Principle –*** un client nu ar trebui niciodată să fie obligat să implementeze o interfață de care nu are nevoie sau să depindă de niște metode pe care nu le folosește;

- ***Dependency Inversion Principle –*** entitățile trebuie să depindă de abstracții, iar abstracțiile să nu depindă de detalii. Presupune ca modulele aflate la nivelele superioare într-o arhitectură să nu depindă de cele aflate la nivele inferioare;

[11] SOLID Principles - https://codeburst.io/solid-design-principle-using-swift-fa67443672b8

3. Concluzii

În concluzie, scopul acestei lucrări a fost atât de a realiza o aplicație web care vine în ajutorul cetățenilor și al autorităților, cât și realizarea unei cercetări științifice asupra seturilor de reguli și principii "Domain-Driven Design". În dezvoltarea proiectelor, atât cel de server, cât și cel de client am întâlnit tehnologii noi pe care le-am aplicat și le-am înțeles, astfel dezvoltându-mi cunoștințele din domeniul dezvoltării web.

Aplicația EMS (Emergency Management System) se pune în valoare prin ușurința raportării de incidente, locația utilizatorului fiind automat detectată folosind serviciile de la Google Maps iar prin intermediul mesajelor sms, utilizatorii sunt alertați în timp real. În același timp, aplicația oferă alte funcționalități, precum: accesul la anunțuri postate de autorități, accesul la rute către cele mai apropiate spitale, căutarea de incidente din alte orașe etc.

Pe partea tehnică am folosit următoarele tehnologii: ASP.NET Core 2 pentru realizarea aplicației server, SQL Server și EntityFramework Core pentru gestionarea datelor, iar pentru dezvoltarea aplicației client am optat pentru Angular 6.

4. Anexe

Tehnologii folosite

În acest capitol sunt prezentate tehnologiile și principalele biblioteci folosite în dezvoltarea aplicației web *Emergency Management System*. Așadar, aplicația este constituită din două părți: partea de server, aici fiind implementată logica aplicației și partea de client, ce constă în interfața aplicației.

Anexa 1: Server

Server-ul este un Web API și a fost implementat în limbajul de programare C# utilizând următoarele framework-uri: ASP.NET Core 2, SQL Server, EntityFramework, AutoMapper, LINQ și Swagger. Mediul de lucru folosit în dezvoltarea server-ului a fost Microsoft Visual Studio Preview[12].

Am preferat ca aplicația de server să fie un Web API (*fig. 5*), serviciile web având avantaje majore:

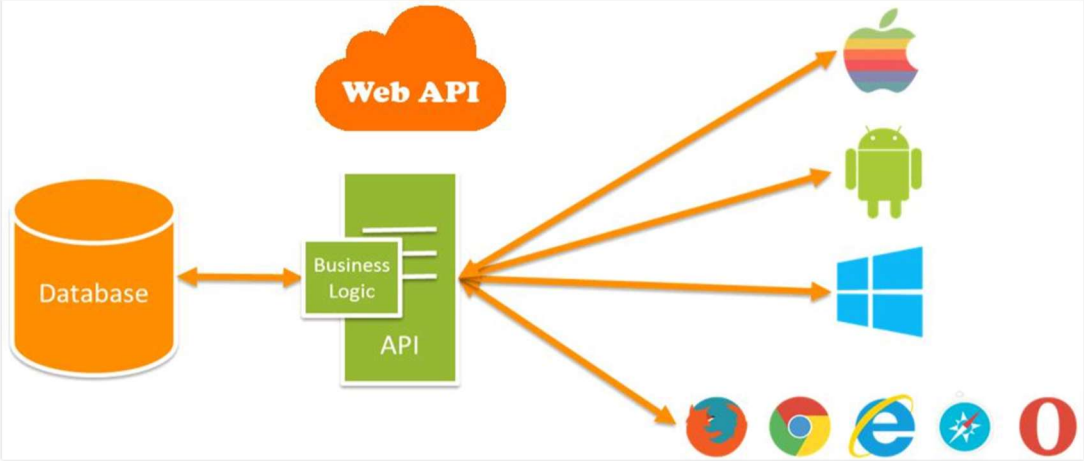

Fig. 5[13]

- interoperabilitatea între aplicații;
- serviciile pot fi reutilizate;
- expunerea ușoară a informației către consumatori;
- independență de platformă sau limbaj de programare;
- respectarea principiului cuplaj slab (eng. *Loose coupling*);

În ceea ce privește limbajul de programare, C# este un limbaj creat de compania Microsoft și oferit publicului în anul 2000, lansarea oficială având loc în primăvara anului 2002. Este un limbaj

[12] Microsoft Visual Studio Preview IDE (Mediu de dezvoltare) - https://visualstudio.microsoft.com/vs/preview/
[13] http://www.mukeshkumar.net/articles/web-api

de programare orientat obiect care permite crearea de aplicații industriale robuste. Limbajul C# nu este dependent de sistemul de operare Windows, el se poate compila și pe alte sisteme de operare, cum ar fi Linux. Conform companiei pentru calitatea software-ului TIOBE[14], limbajul C# se află pe locul 7 în topul celor mai folosite limbaje din ianuarie 2019.

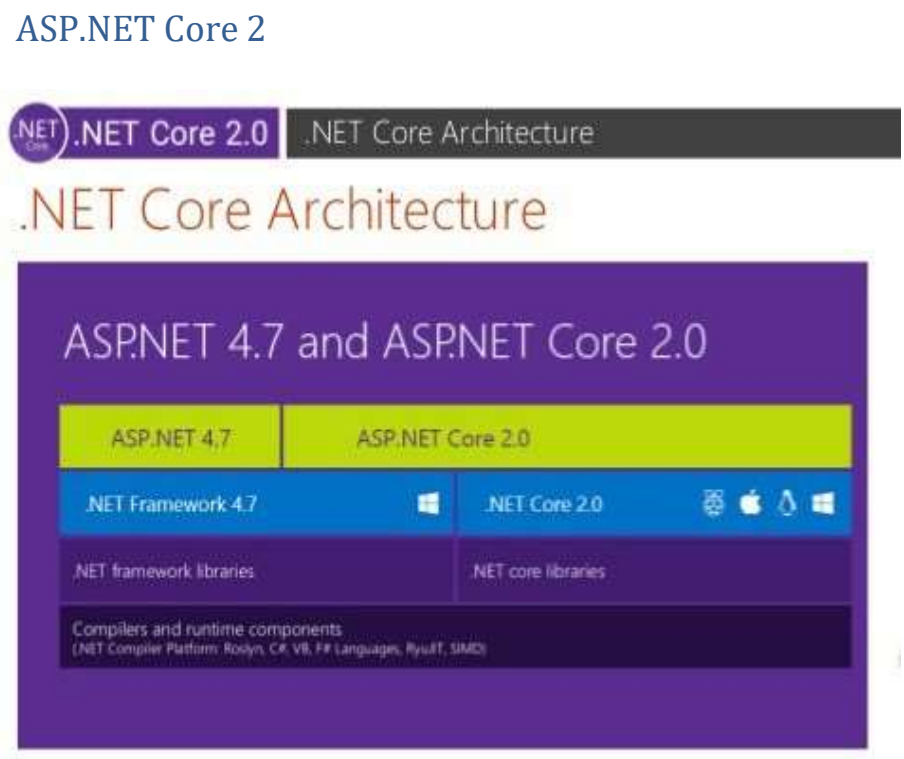

Fig. 6[15]

ASP.NET Core este lansat ca o reimplementare a framework-ului clasic ASP.NET. Versiunea folosită în dezvoltarea aplicației este ASP.NET Core 2 și apare în anul 2016 aducând o multitudine de beneficii programatorilor:

- Portabilitate;
- *Open-Source* – codul sursă al framework-ului este făcut public*;*
- *Cross-Plarform* - aplicația creată poate fi compilată și rulată pe diferite platforme: Windows, Mac si Linux;
- Un nou model de configurare (înlocuiește fișierul *web.config* din ASP.NET Framework);
- Suport pentru rularea aplicației fie în varianta clasica în IIS sau *self-hosting* folosind Kestrel;
- Instalarea de pachete sau librării folosind NuGet;

[14] TIOBE (the software quality company) - https://www.tiobe.com/tiobe-index/
[15] .NET Core Architecture - https://medium.com/@dens.scollo/introduction-to-asp-net-core-mvc-and-docker-application-1cbd4c0475d1

- *Dependecy Injection* integrat în framework – înlocuirea metodei folosind *Autofac*;

Entity Framework Core

Entity Framework permite crearea de aplicații software care lucrează cu baze de date. Este un ORM (*Object-Relational Mapping*), ceea ce înseamnă că accesarea și manipularea obiectelor în aplicație se face într-un mod simplu. În aplicația mea am folosit Entity Framework Core, tehnologie lansată odată cu ASP.NET Core.

Pentru arhitectura bazei de date există mai multe modele de programare folosind Entity Framework, și anume: *Database First*, *Model Design First* și *Code First*.

Modelul *Database First* presupune întâi crearea bazei de date și apoi sunt generate entitățile ce formează logica aplicației (acest model este folosit, în general pentru procesul de *reverse engineering*[16])

Model Design First se realizează construind întâi diagrama modelului aplicației folosind mediu de lucru Visual Studio. Pe baza acestei diagrame sunt generate tabele fiecărei entități create.

Modelul abordat de mine în definirea entităților aplicației a fost Code First. Acest model constă în scrierea codului fiecărei clase în primă fază, ca apoi baza de date fiind generată rulând câteva comenzi în consola *Package Manager.*

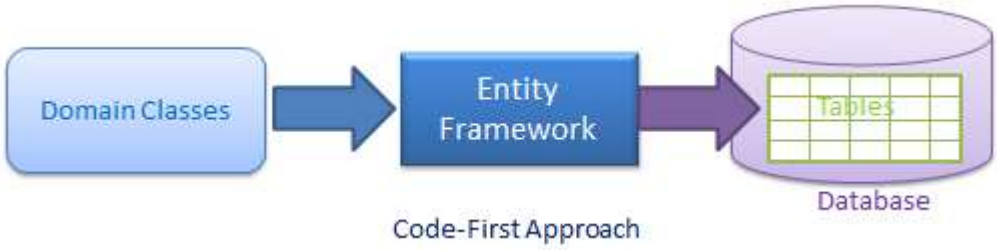

Fig. 7[17]

ASP.NET Identity

Pentru modelarea entităților necesare gestionării utilizatorilor am folosit ASP.NET Identity, împreună cu Entity Framework Core – Code First menționat anterior.

[16] Reverse Engineering - https://en.wikipedia.org/wiki/Reverse_engineering#Reverse_engineering_of_software
[17] Code-First Approach - http://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx

ASP.NET Identity este o bibliotecă care permite crearea și manipularea entităților de bază dintr-o aplicație ce conține funcționalități de înregistrare, autentificare etc. Acest pachet, creează automat următoarele tabele cu atributele necesare:

- **Users** (*Id*, *UserName, NormalizedUserName, Email, NormalizedEmail, EmailConfirmed, PasswordHash, SecurityStamp, ConcurrencyStamp, PhoneNumber, PhoneNumberConfirmed, TwoFactorEnabled, LockoutEnd, LockoutEnabled, AccessFailedCount*)
- **Roles** (*Id, Name, NormalizedName, ConcurrencyStamp*)
- **RoleClaims** (*Id, RoleId, ClaimType, ClaimValue)*
- **UserClaims** (*Id, UserId, ClaimType, ClaimValue)*
- **UserLogins** (*LoginProvider, ProviderKey, ProviderDisplayName, UserId*)
- **UserTokens** (*UserId, LoginProvider, Name, Value)*

AutoMapper

AutoMapper este o bibliotecă simplă care face posibilă transformarea unui obiect care ține de domeniul aplicației într-un obiect DTO (*Data Transfer Object*). Ea vine cu un set de funcționalități: convertori – poți scrie cod care să convertească un anumit tip de date sau chiar o valoarea, formatare – datele din obiectul DTO pot fi formatate, aplicarea de funcții – ca și în cazul formatării, datele pot fi apelate printr-o funcție înainte de a fi transferate, validarea datelor – la run-time datele sunt validate, așadar pot fi gestionate eventualele erori provocate de datele invalide, crearea de profiluri – clase cu scopul de a organiza codul.

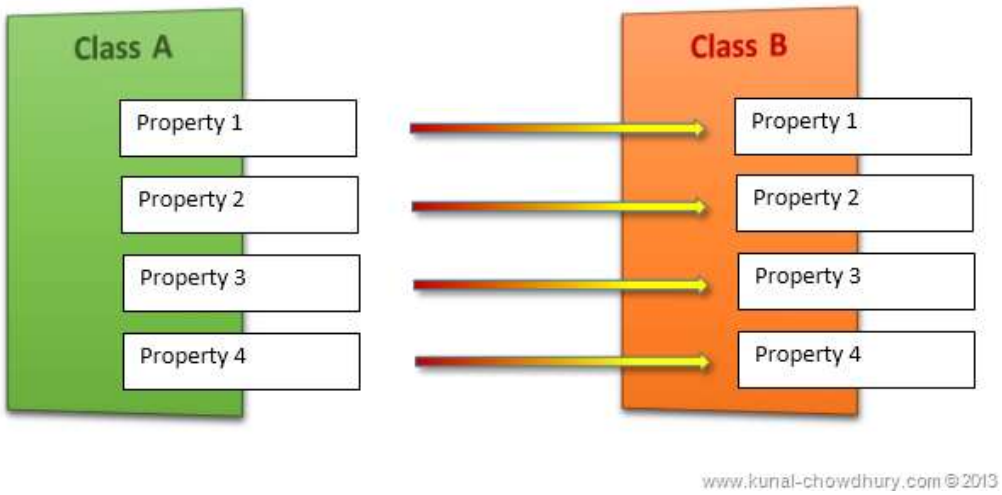

Fig. 8[18]

LINQ

[18] AutoMapper (maparea obiectelor) - https://sites.google.com/site/wcfpandu/automapper

LINQ (*Language-Integrated Query*) este o componentă specifică platformei .NET care permite manipularea colecțiilor de obiecte. Este folosit pentru crearea de interogări compatibile cu Entity Framework. Există 2 metode pentru folosirea mecanismului LINQ:

- *Query Syntax:* se apropie foarte mult de limbajul SQL

```
// string collection
IList<string> stringList = new List<string> () {
     "C# Tutorials",
     "VB.NET Tutorials",
     "Learn C++",
     "MVC Tutorials",
     "Java"
};

// LINQ Query Syntax
var result = from s in stringList
             where s.Contains("Tutorials")
             select s;
```

Exemplu de *Query Syntax* in LINQ C#[19]

- *Method Syntax*: folosește metode din interfața *IEnumerable* pentru a selecta datele necesare. Pentru manipularea datelor, am folosit această metodă. Mai jos este prezentată o secțiune din codul aplicației de server unde am aplicat *Method Syntax*:

```
public async Task<City> GetByNameAsync (string name) =>
      await _context.Cities
                  .FirstOrDefaultAsync(t => t.Name == name);
```

Anexa 2: Client

Angular, TypeScript și npm

Client-ul este o aplicație web dezvoltată folosind framework-ul Angular. Angular este o platformă creată de cei de la Google bazată pe limbajul de programare TypeScript și permite crearea de aplicații web de tip SPA (*Single Page Application*). Prima versiune de Angular a fost AngularJS (sau Angular 1.x) care folosea limbajul JavaScript, urmând ca versiunile ulterioare să aducă în discuție limbajul TypeScript.

[19] LINQ Query Syntax in C# - https://www.tutorialsteacher.com/linq/linq-query-syntax

TypeScript este un limbaj de programare realizat de către cei de la GitHub, care permite scrierea unui cod bazat pe programare orientată-obiect (crearea clase, interfețe etc.), ca apoi, la compilare fiind transformat în cod JavaScript. Limbajul vine și cu alte beneficii: tipurile de date adăugate variabilelor, funcțiilor sau proprietăților dintr-o clasă ajută compilatorul cu eventualele erori în cod, IntelliSense – folosirea unui IDE de calitate, scrierea de cod TypeScript este foarte simplă, mediul de dezvoltare indicând sugestii de cod ce pot fi adăugate, completări, etc. În dezvoltarea aplicației Client am folosit mediul de dezvoltare Visual Studio Code.

Pentru instalarea modulelor necesare dezvoltării aplicației am folosit *npm* (*Node Package Manager*) – similar cu NuGet (*package manager* pentru partea de server).

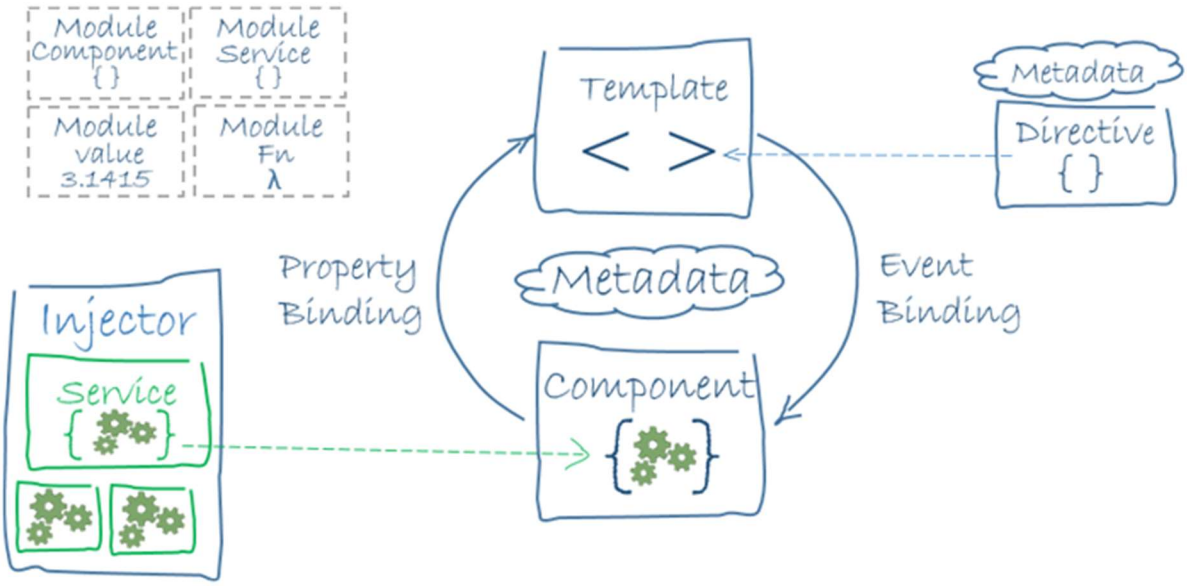

Fig. 8[20]

HTML și CSS

HTML (*HyperText Markup Language*) este un limbaj de marcare folosit pentru crearea unei pagini web care poate fi accesată dintr-un browser. Permite adăugarea de etichete și tag-uri specifice, ulterior ele fiind stilizate și animate folosind CSS și JavaScript.

CSS (*Cascading Style Sheets*) este un limbaj pentru formatarea și stilizarea unei pagini web. Aplică culori, margini, font-uri, așezare în pagină etc. folosind fișiere externe cu extensia *.css* unui document HTML. Există o serie de librării faimoase construite în CSS, cum ar fi: Bootstrap, Semantic UI, Materialize, Pure, Font Awesome și multe altele. În principal, pentru realizarea design-ului și al așezării componentelor în pagină am folosit librăriile Bootstrap și Font Awesome, plus alte componente specifice framework-ului Angular.

[20] Arhitectura unei aplicații Angular - https://angular.io/guide/architecture

5. Bibliografie

[1] Dezastrele naturale din 2017 - http://www.1asig.ro/Dezastrele-naturale-din-2017-Pagube-record-pentru-asiguratori-articol-13,90-58289.htm

[2] DSU este aplicația oficială a Departamentului pentru Situații de Urgență din cadrul Ministerului Afacerilor Interne. - http://www.dsu.mai.gov.ro/

[3] DDD (Domain-Driven Design) - https://www.mitrais.com/news-updates/breaking-complexity-using-domain-driven-design/

[4] Onion Architecture - https://dzone.com/articles/onion-architecture-is-interesting

[5] MVC (Architectural Pattern) - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[6] Repository Pattern Class Diagram - https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/

[7] Unit of Work Class Diagram - https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/

[8] Factory Pattern Definition (Gang of Four) - Design Patterns: Elements of Reusable Object-Oriented Software (1994)

[9] Inversion Of Control Principle - https://en.wikipedia.org/wiki/Inversion_of_control

[10] SOLID Principles - https://codeburst.io/solid-design-principle-using-swift-fa67443672b8

[11] Microsoft Visual Studio Preview IDE (Mediu de dezvoltare) - https://visualstudio.microsoft.com/vs/preview/

[12] http://www.mukeshkumar.net/articles/web-api

[13] TIOBE (the software quality company) - https://www.tiobe.com/tiobe-index/

[14] .NET Core Architecture - https://medium.com/@dens.scollo/introduction-to-asp-net-core-mvc-and-docker-application-1cbd4c0475d1

[15] Reverse Engineering - https://en.wikipedia.org/wiki/Reverse_engineering#Reverse_engineering_of_software

[16] Code-First Approach - http://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx

[17] AutoMapper (maparea obiectelor) - https://sites.google.com/site/wcfpandu/automapper

[18] LINQ Query Syntax in C# - https://www.tutorialsteacher.com/linq/linq-query-syntax

[19] Arhitectura unei aplicații Angular - https://angular.io/guide/architecture

5.1 Resurse

[20] https://medium.com/@levifuller/building-an-angular-application-with-asp-net-core-in-visual-studio-2017-visualized-f4b163830eaa

[21] https://www.dotnetcurry.com/entityframework/1348/ef-core-web-api-crud-operations

[22] https://chsakell.com/2016/06/23/rest-apis-using-asp-net-core-and-entity-framework-core/

[23] https://github.com/zkavtaskin/Domain-Driven-Design-Example

[24] https://anthonychu.ca/post/aspnet-identity-20-logging-in-with-email-or-username/

[25] https://medium.com/engineering-on-the-incline/reloading-current-route-on-click-angular-5-1a1bfc740ab2

[26] https://medium.com/volosoft/asp-net-core-dependency-injection-best-practices-tips-tricks-c6e9c67f9d96

[27] https://stackblitz.com/edit/angular-google-maps-demo

[28] https://robby570.tw/Agm-Direction-Docs/

[29] https://alligator.io/rxjs/hot-cold-observables/