**SkillPoint**

**- raport -**

**Ovidiu Tudosă**

**Ioana Sitaru**

**Bogdan Lupu**

**Valerio Mihailă**

**Adrian Perciun**

**Gabriel Tuduri**

**Cuprins:**

1. **Descriere**
2. **Workflow**
3. **API-uri Externe**
4. **Design Patterns**
   1. **MVW**
   2. **(MV)C**
   3. **DTO, IoC**
   4. **M(VC), Repository**
5. **Arhitectură**
6. **Object-Relation Mapping**
7. **Principii SOLID**
8. **Încapsulare**
9. **Metodologie de lucru**
10. **Linkuri utile**
11. **Autoevaluare**

**Descriere**

Aplicația vine în ajutorul utilizatorilor prin recomandarea de evenimente tehnice ce se pliază pe interesele personale ale fiecăruia.

**Workflow**

Primul contact al unui user cu aplicația va fi prin pagina principală. Aici va avea o scurta descriere a aplicației, urmată de o listă a unor evenimente viitoare, aleatorii. Următorul pas este crearea unui cont sau login-ul, dacă este deja înregistrat în sistem. La crearea contului, utilizatorul își va adăuga: username, password, email, locație și o lista de taguri, reprezentând interesele sale.

După etapa introductivă, de creare a profilului, utilizatorul are posibilitățile de:

* vizualizare pe pagina principala a recomandărilor personalizate de evenimente, filtrate pe baza intereselor
* editare a profilul
* adăugare de noi tag-uri în eventualitatea schimbării intereselor personale
* adăugare a unor evenimente în secțiunea de attending
* vizualizare a unui istoric al evenimentelor la care a dat attend, fie ele trecute sau viitoare prin intermediul paginii "My Events"
* vizualizare a evoluției pe pagina destinată achievementurilor

Din toate aceste funcționalități propuse inițial, singurul lucru pe care nu l-am terminat de implementat este componenta de gamification a aplicației: achievementurile. În schimb, am integrat cu 3 API-uri care ne oferă evenimente, pentru a avea o diversitate cât mai mare pe funcționalitățile actuale.

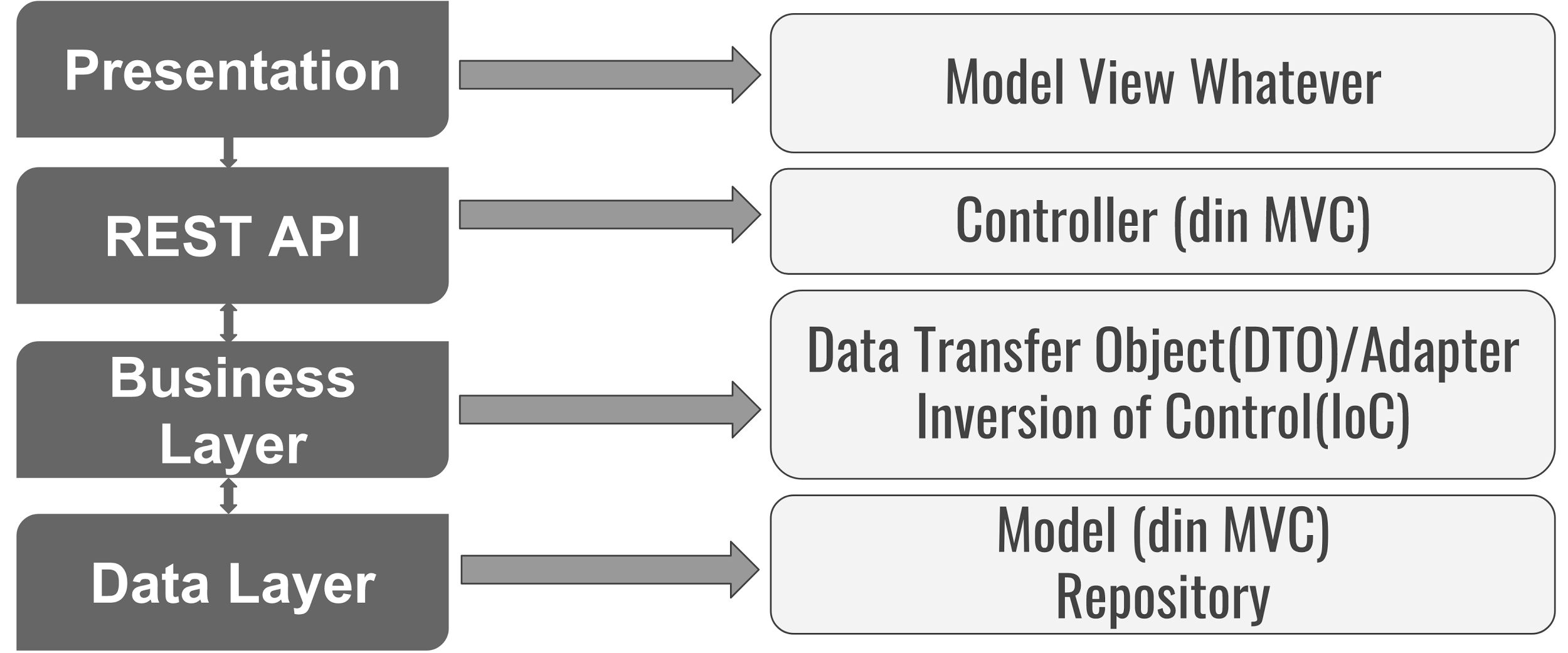
**API-uri Externe**

Pentru obținerea datelor necesare atingerii scopului propus de aplicația noastră web, vom recurge la folosirea API-urilor oferite public de siteuri specializate pe domenii tehnice.

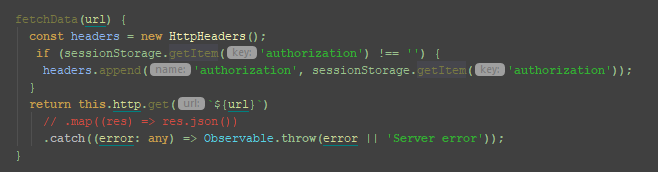
În urma configurării profilului personal, pe pagina principală vor exista sugestii de evenimente la care utilizatorul poate participa în vederea îmbunătățirii abilităților tehnice. Sugestiile cuprind atât conferințe, traininguri, workshopuri, cât și concursuri și stagii de practică din ariile sale de interes. Evenimentele sunt preluate de pe site-uri precum: Eventbrite, Meetup, Hackathon. Am construit un wrapper pentru a mapa response-ul venit de la API cu modelul nostru, ce cuprinde: titlul, poza evenimentului, locația, data, și o descriere, pentru a oferi, în mod structurat, utilizatorilor toate informațiile necesare pentru a participa la eveniment.

În afara de aceste API-uri, pe partea de front-end, pentru realizarea unei interfețe cât mai user-friendly, am recurs la folosirea Google Maps API pe pagina About, cu ajutorul căruia am afișat pe hartă locația noastră.

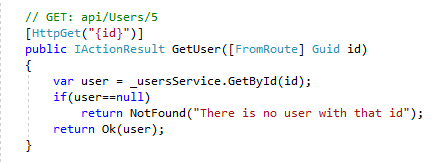
**Design Patterns**

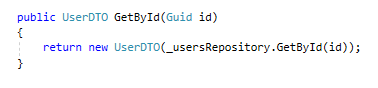


**Presentation**

****

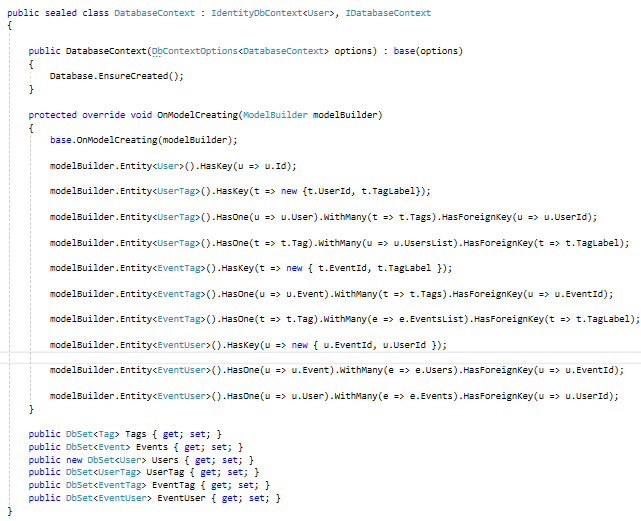
**REST API**

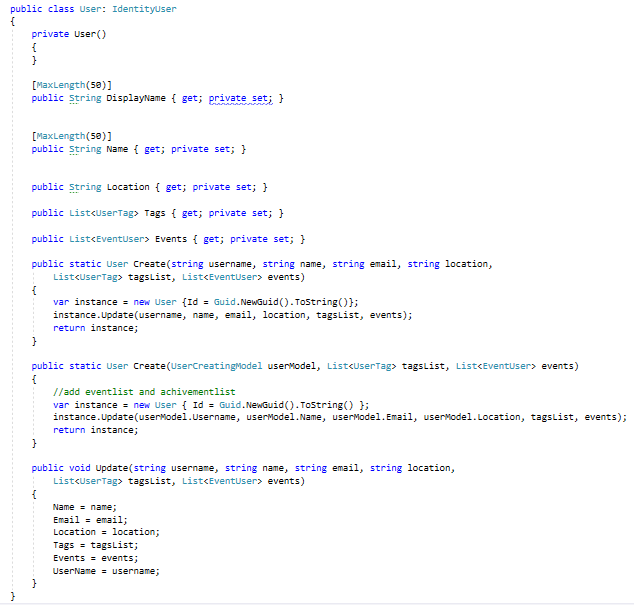


**Business Layer**

****

**Data Layer**

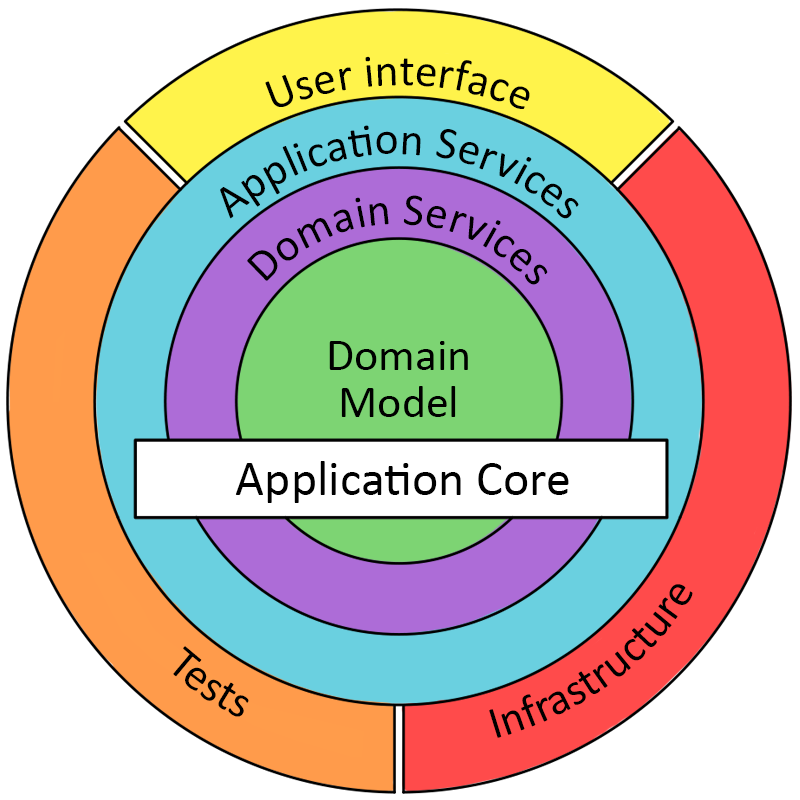
****

****

**Arhitectură**

Am recurs la o arhitecturăbazată pe layere (Onion Arhitecture) care are următoarea structură:

* *Domain*: în care avem toate entitățile și definițiile relațiilor dintre ele (Database Context)
* *Business*: compus din repositories și servicii fiecare având câte o interfață și o clasă ce o implementează. Toată logica aplicației este definită și implementată la acest nivel
* *Application*: clase controller care expun end-point-urile API-ului



Folosind această arhitectură clasele de la un anumit nivel depind numai de cele de la un nivel inferior, cuplarea fiind întotdeauna către centru.

**Object-Relation Mapping**

ORM-ul pe care l-am folosit este Entity Framework Core și stategia pentru construirea bazei de date este de tipul “code-first”, în spate fiind SQL Server 2016.

**Principii SOLID**

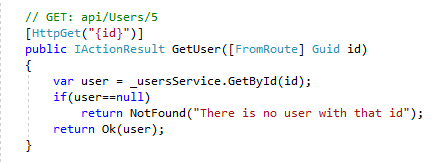
Vom prezenta pe scurt principiile respectate în cadrul dezvoltării aplicației, cu exemple concrete din codul sursă.

* **Single responsability**

Fiecare clasă din fiecare layer al aplicației are un rol unic, bine definit.

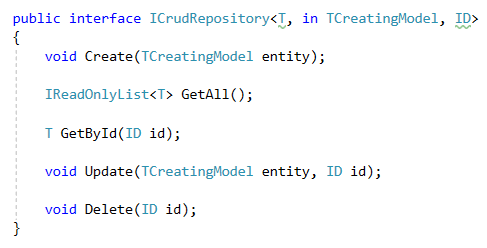
De exemplu în clasele de tip “controller” se face doar handling la tipul de cerere, pasând datele către layer-ele inferioare, în cele din urmă primind un rezultat și returnându-l mai departe către nivelul client. Astfel, controllerele doar deleagă responsabilități.

Exemplu de endpoint:



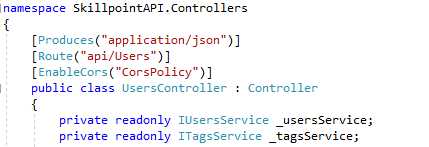
* **Open-closed principle**

Fiecare interfață de tip service/repository implementează o interfață minimală numită ICrudService respectiv ICrudRepository care oferă operațiile de baza pentru fiecare entitate. De asemenea, această abordare aduce și avantajul arhitectural “Microservices Ready“, ceea ce face ca aplicația noastră să fie ușor scalabilă, oricând putând fi făcută în mod facil trecerea la o arhitectură bazată pe microservicii.

****

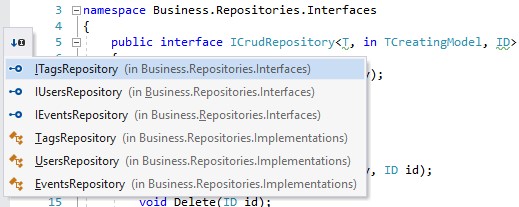
* **Liskov Substitution principle**

Datorită mecanismului de dependency injection, fiecare instanță a unei alte clase din alt layer este declarată ca o interfață. Astfel, interacțiunea dintre clase se realizează mereu prin intermediul interfețelor, și clasele ce le folosesc nu depind de implementările acestora



* **Interface segregation principle**

Fiecare interfață de la fiecare nivel este specializată pe un singur model (Event, User, sau Tag). Astfel, spre exemplu, implementarea service-ul pentru Useri nu are nevoie să implementeze metode specifice celui pentru Event-uri; prin urmare, fiecare clasă implementează doar metode din interfață de care are nevoie și pe care le utilizează.



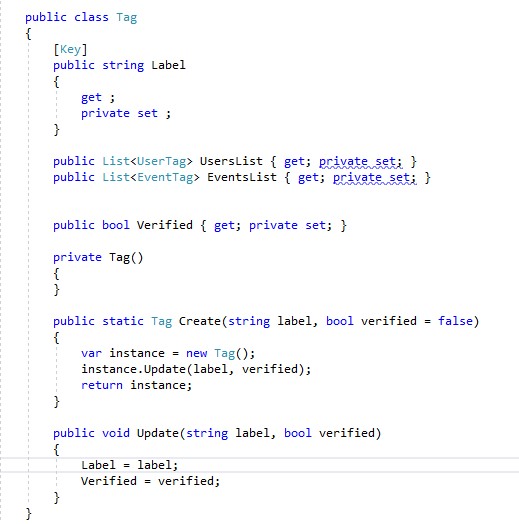
* **Dependency Inversion**

Deoarece fiecare clasă depinde de o interfață și nu de o implementare (dependency inversion principle), este nevoie ca la runtime să existe un mecanism care “injectează” o implementare în locurile care depind de o interfață aferentă. Pentru asta am folosit mecanismul implicit de dependency injection oferit de .NET Core 2.0.

Exemplu din codul sursă:



**Încapsulare**

Am folosit încapsularea pe clasele ce definesc entitățile din aplicație, atributele lor fiind private, având acces la acestea prin intermediul metodelor declarate publice.****

**Metodologie de lucru**

Proiectul nostru a fost împărțit pe sprint-uri de o săptămână, iar la începutul fiecăruia am avut câte un meeting în care am făcut task breakdown, am stabilit scoruri pentru fiecare story și am încercat să abordăm un commitment mediu spre light, target-ul nostru fiind cât de mult cod rulabil la fiecare sprint. Pentru a avea o vedere de ansamblu a proiectului am folosit Trello.

Un alt aspect important al lucrului în echipă a fost cel de pair-programming, prin intermediul căruia am îmbunătățit atât skillurile de comunicare ale fiecăruia, cât și cele tehnice, prin knowledge sharing. Totodată, ne-am împărțit taskurile în funcție de timp și având în vedere acumularea de cunoștințe a fiecărui membru, încurajând pe fiecare în parte, când deadline-urile nu erau prea presante, să ia taskuri în afara zonei de confort (pentru unii frontend, pentru alții backend). Astfel, scopul a fost aducerea cât mai aproape de skillset-ul unui fullstack developer pentru fiecare dintre noi.

**Linkuri utile**

* **Github:** [**https://github.com/ioanasitaru/dotNet**](https://github.com/ioanasitaru/dotNet)
* **Video prezentare:** [**https://youtu.be/KDVcn4FSyU0**](https://youtu.be/KDVcn4FSyU0)

**Autoevaluare**\*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Nume** | Bogdan | Ioana | Eduard | Adrian | Valerio | Ovidiu |
| **Bogdan** | 10 | 10 | 10 | 9 | 9 | 10 |
| **Ioana** | 10 | 10 | 10 | 10 | 10 | 10 |
| **Eduard** | 10 | 10 | 9 | 9 | 9 | 10 |
| **Adrian** | 9.5 | 9 | 9 | 9 | 9 | 9.5 |
| **Valerio** | 10 | 9 | 9.25 | 9 | 8 | 10 |
| **Ovidiu** | 10 | 10 | 9 | 9 | 9 | 10 |

\*(s-a completat pe linie)