**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI**

**FACULTATEA DE INFORMATICĂ**

****

LUCRARE DE LICENȚĂ

**PC Builder – configurator online**

**pentru computere**

**propusă de**

***Andrei – Răzvan Bordeianu – Cocea***

**Sesiunea:** *Iulie, 2018*

**Coordonator științific**

Drd. Colab. Florin Olariu

**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI**

**FACULTATEA DE INFORMATICĂ**

PC Builder – configurator online

pentru computere

*Andrei - Răzvan Bordeianu - Cocea*

**Sesiunea:** *Iulie, 2018*

**Coordonator științific**

Drd. Colab. Florin Olariu

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și prenumele \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Data \_\_\_\_\_\_\_\_\_\_\_\_ Semnătura \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**DECLARAȚIE privind originalitatea conținutului lucrării de licență**

Subsemntatul(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 fasificării 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 …………………………

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Titlul complet al lucrării*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de testetc.) 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.

Iași, *data*

Absolvent *Prenume Nume*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura în original)

ACORD PRIVIND PROPRIETATEA DREPTULUI DE AUTOR

Facultatea de Informatică este de acord ca drepturile de autor asupra programelor-calculator, în format executabil și sursă, să aparțină autorului prezentei lucrări, *Andrei-Răzvan Bordeianu-Cocea.*

Încheierea acestui acord este necesară din următoarele motive:

*[Se explică de ce este necesar un acord, se descriu originile resurselor utilizate în realizarea*

*produsului-program (personal, tehnologii, fonduri) și aportul adus de fiecare resursă.]*

Iași, *data*

Decan *Prenume Nume* Absolvent *Prenume Nume*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura în original) (semnătura în original)

Cuprins

[Introducere 6](#_Toc517552377)

[1.1 Motivație 7](#_Toc517552378)

[1.2 Context 8](#_Toc517552379)

[1.3 Specificații tehnice 11](#_Toc517552380)

[Contribuții 13](#_Toc517552381)

[Capitolul 1: Dezvoltarea aplicației server (API) 14](#_Toc517552382)

[1.1 Arhitectura aplicației 15](#_Toc517552383)

[1.1.1 Nivelul Data 16](#_Toc517552384)

[1.1.2 Nivelul Business 24](#_Toc517552385)

[1.1.3 Nivelul Web Api 28](#_Toc517552386)

[Bibliografie 29](#_Toc517552387)

# Introducere

Computerul reprezintă una dintre cele mai strălucite invenții ale societății moderne. Insă istoria acestuia se întinde până acum mai bine de 2500 de ani, când oamenii vremurilor respective au inventat un instrument pentru calcule aritmetice elementare, denumit “Abascus”(Fig.1). Acest instrument confecționat din sârmă și mărgele prinse într-un cadru dreptunghiular încă este utilizat în mai multe părți ale lumii. Diferența între un abac și un computer modern este destul de mare însă la baza acestor două invenții stă același principiu: “realizarea de calcule matematice într-un timp foarte scurt”.

Într-adevăr, istoria și evoluția computerelor este extraordinară și cu multe inovații tehnologice descoperite pe parcursul a foarte mulți ani de cercetare, însă progresul acestora a fost unul remarcabil.

Primele computere au fost construite la mijlocul anilor 1800 de către Charles Babbage și poartă denumirea de “motoarele Babbage”. Ideea acestuia a fost să creeze o mașină analitică, programabilă și complet automata care să calculeze probleme de matematică. Invenția lui Babbage a contribuit cu succes la dezvoltarea computerului modern prin introducerea mai multor concepte: idea de separare a stocării de procesare, structura logică și modul în care datele și instrucțiunile sunt introduse și afișate.

După succesul primelor computere, acestea au continuat sa evolueze într-un ritm alert, astfel că în la începutul secolului 20 apar primele calculatoare eletronice ce utilizau tuburi vidate și benzi de hârtie pentru a efectua o serie de operații logice. Tot în acea perioadă s-a descoperit și tubul catodic și precursorul memoriei RAM, denumit “Tubul Williams”.

În a doua jumătate a secolului 20, imediat dupa descoperirea tranzistorului și a microprocesorului, computerele au devenit din ce în ce mai mici, mai performante și accesibile publicului larg. Totodata au apărut și primele companii ce comercializau noile dispozitive revoluționare.

Astăzi, computerul personal, prescurtat PC, este o unealtă standard pe care o întalnim în aproape orice domeniu de activitate, în număr de peste 2 miliarde de unități la nivel global.

## Motivație

Computerele din zilele noastre au la bază 8 componente: procesorul ce trebuie neapărat să fie răcit de un cooler, placa de bază, memoria RAM, placa video, unitate de stocare ce poate fi de mai multe feluri: SSD, HDD sau SSHD, sursa de alimentare și carcasa, componentă ce le găzduiește pe toate anterior menționate. Pentru fiecare din cele 8 componente există zeci de modele disponibile ce diferă destul de mult la nivel de specificații tehnice.

An de an sunt lansate noi modele de componente din ce în ce mai performante, astfel utilizatorilor le este dificil să țină pasul cu avansul tehnologic, întrucăt vorbim de un numar foarte mare de produse. Acest lucru este resimțit și de magazinele de componente, întrucât un număr foarte mare de produse va conduce la necesitatea unui personal competent care să-i ajute pe clienți atunci când aceștia doresc să își configurere un PC.

În zilele noastre cumpărarea unui computer fără a cere ajutorul nimănui poate fi o perspectivă descurajantă. Există literalmente sute de modele diferite din care poți alege, multe dintre ele arătând aproape identic, însă cu diferențe destul de mari la nivel de specificații. Nu este deloc surprinzător faptul că majoritatea cumpărătorilor de computere fac minime comparații între componente și de multe ori ajung să cumpere componente la întâmplare, în funcție de ce brand le este cât de cât familiar, existând astfel un risc foarte mare ca acestea să fie incompatibile unele cu altele.

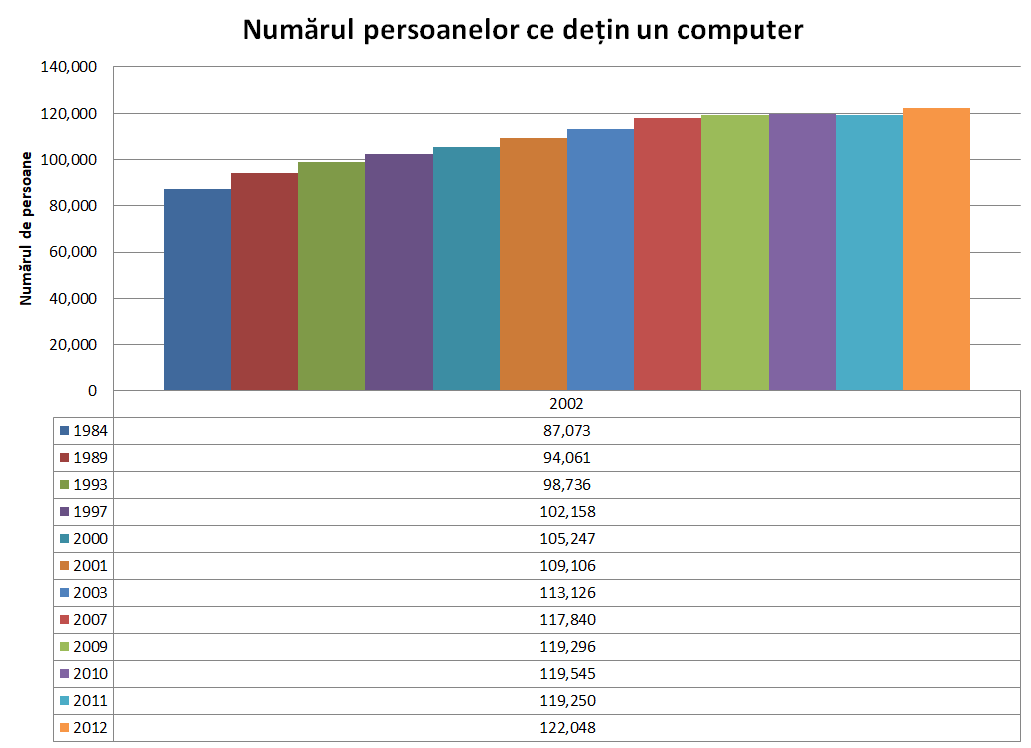
Aplicația prezentată în această lucrare vine atât în ajutorul magazinelor de componente, cât și persoanelor, fie ele cu cunoștiințe tehnice sau nu, ce doresc să își achiziționeze un PC. Magazinele de componente își pot încărca în aplicație produsele pe care le comercializează, iar persoanele ce folosesc aplicația au la dispoziție un configurator online prin intermediul căruia aceștia își pot alege după bunul plac ce componente doresc să conțină viitorul lor computer, fără a fi necesar să verifice ei înșiși dacă componentele alese sunt compatibile între ele. Astfel vor obține în final un computer cu componente în proporție de 100% compatibile, într-un timp foarte scurt. Totodată utilizatorii pot interacționa cu o zonă de tutorial în care pot observa cum arată și din ce este formată o configurație, cât și cu o zonă de tip galerie în care interacționează cu configurații create de alți utilizatori.

## Context

În sfera computerelor și a hardware-ului întâlnim o piață rapidă și în continuă evoluție. Cele mai recente tendințe lansate pot fi depășite după doar un an sau doi. Având la dispoziție o piață foarte diversificată, numărul clienților ai magazinelor de componente a crescut simțitor de la an la an.

Un studiu realizat de cei de la “United States Census Bureau” pe un eșantion de 150.000 de persoane confirmă faptul că interesul acestora pentru computere a crescut foarte mult in ultimii 30 de ani.

Fig. 2 - Numărul persoanelor ce dețin un computer



Totodată un alt studiu realizat de cei de la “United States Census Bureau” a scos în evidență faptul că tot mai multe persoane au început să aibă acces la internet.

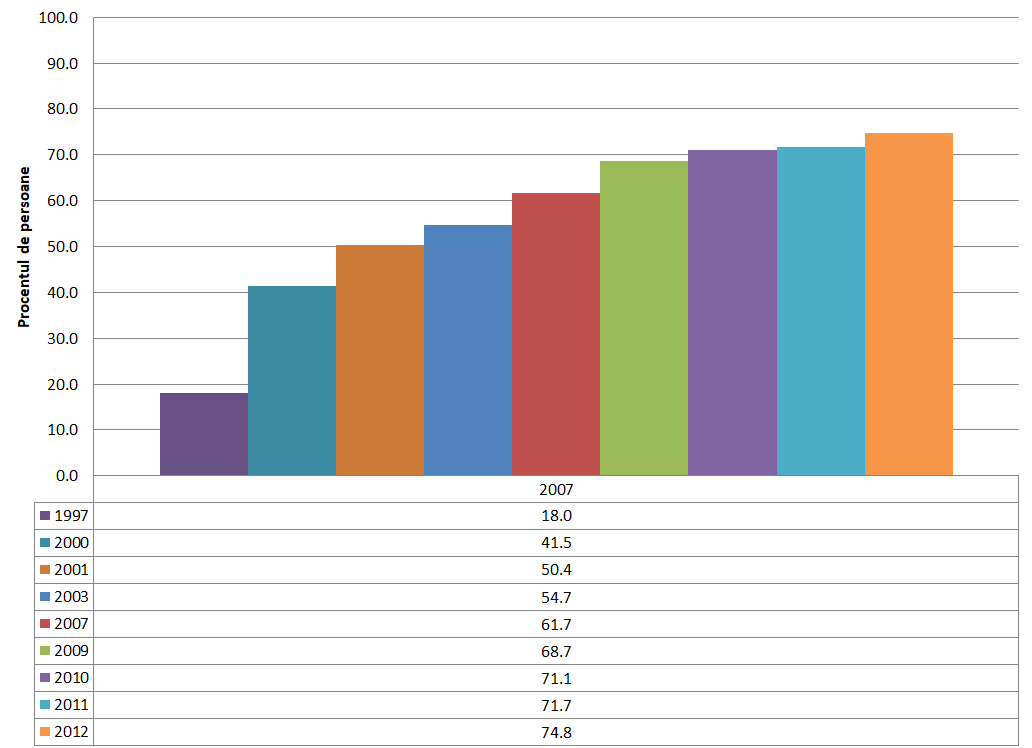
g

Fig. 3 - Procentul de persoane ce au acces la internet

Luând în considerare rezultatele studiilor anterior prezentate putem concluziona că numărul magazinelor de componente pentru computere ce și-au extins activitatea sau chiar au migrat cu totul în mediul online este una justificată. Acest lucru a permis clienților un acces mult mai facil la produse prin intermediul site-urilor web. Astfel că pentru a achiziționa cu succes produsele dorite, clienții au nevoie doar de o conexiune la internet și un card bancar. Un alt avantaj al mediului online îl reprezintă faptul că magazinele își prezintă întreaga ofertă de produse, fără a fi limitați de un spațiu fizic.

Chiar dacă interacțiunea dintre magazinele online și clienți a crescut foarte mult, cumpărarea unui computer pe piese are încă un caracter descurajant, alimentat de faptul că există un număr extrem de mare de componente asemănătoare, dar cu minime diferențe la nivelul specificațiilor tehnice. În încercarea de a diminua această problemă au apărut aplicații web cu ajutorul cărora clienții pot genera configurări de sistem, fără a mai fi nevoie ca aceștia să navigheze printr-o multitudine de pagini.

**Exemple de aplicații:**

* **PCPart Picker**: este o aplicație destul de populară pe teritoriul american. Aceasta permite realizarea de configurări pentru sisteme de tip desktop și oferă instrucțiuni de selectare a componentelor. Aplicația include un mecanism prin intermediul căruia utilizatorul este alertat dacă componentele selectate sunt sau nu compatibile.
* **IBuyPower:** este o aplicație lansată în anul 1999 ce permite personalizarea sistemelor de tip gaming.
* **Computer Sales:** este un motor de căutare vertical, dedicat pieței componentelor PC. Un motor de căutare hardware. Utilizatorii pot căuta componente folosind filtre comprehensive, specializate pe categorii.
* **DinoPC:** este o aplicație lansată în anul 2007 pentru clienții din UK. Aplicația include o serie de configurații din care utilizatorii pot alege să le modifice după necesitățile lor.

## Specificații tehnice

Aplicația PC Builder este compusă din două module (aplicații) independente: server și client. Pentru dezvoltarea celor două module m-am folosit de diverse tehnologii web pe care le-am considerat că ar fi de actualitate. Printre acestea se numără următoarele:

* Asp .Net Core, în versiunea 2.1
* SQL Server 2017
* Entity Framework Core
* HTML, CSS, Bootstrap
* Angular 5
* JavaScript/TypeScript
* Git

Modulul de server a fost dezvoltat în limbajul C# folosind suita de tehnologii specifice platformei de dezvoltare .NET. În linii mari, acesta reprezintă un API REST bazat pe ASP.NET Core în versiunea 2.1 ce procesează cereri HTTP. Pentru stocarea datelor am folosit SQL Server 2017, iar ca ORM am folosit Entity Framework Core. Mediul de dezvoltare pentru modulul server a fost Visual Studio 2017 în versiunea 15.7.

Limbajul C# a fost creat de Microsoft ca instrument de dezvoltare pentru arhitectura .NET. Acesta este un limbaj de programare de nivel înalt complet orientat pe obiecte, simplu și modern.

Asp .NET Core reprezintă un framework cu sursă deschisă, utilizabil pe diferite platforme, ce a fost construit peste .NET Core. Acesta este utilizat preponderent pentru dezvoltarea de aplicații web, dar și servicii web de tip “RESTful”. Web API-ul construit folosind framework anterior menționat se folosește de SQL Server alături de Entity Framework Core pentru lucrul cu datele din aplicație.

SQL Server este un sistem de gestiune a bazelor de date relaționale (RDBMS) dezvoltat de compania Microsoft. Acesta este construit în jurul unei structuri de tip tabel. Astfel, datele sunt stocate în tabele, fiecare tabel reprezentând un container pentru acestea. Într-un tabel de acest gen datele sunt organizate logic în format de rânduri și coloane. Fiecare rând este considerat o entitate, ce este descrisă de coloanele care marchează atributele entității.

Entity Framework Core este, după cum îi sugerează și numele, un framework ce servește ca ORM (Object Relational Mapper), acesta ajută lucrul cu baze de date, utilizând obiecte specifice domeniului și eliminând nevoia de a scrie codul de acces la respectiva baza de date. Interogările se fac utilizând LINQ (Language-integrated Query), un set de tehnologii bazate pe integrarea capacităților de interogate direct în limbajul C#. Cu ajutorul lui LINQ, o interogare se poate face mult mai simplu decât în mod tradițional, utilizând SQL, astfel că filtrarea, ordonarea sau gruparea datelor se face scriind un număr minim de linii de cod.

HTML reprezintă un limbaj de marcare utilizat pentru crearea paginilor web ce pot fi afișate într-un browser. Acesta împreună cu CSS-ul, prin intermediul căruia sunt adăugate stiluri paginilor web, sunt două componente esențiale în dezvoltarea oricărei aplicații web.

Bootstrap este cel mai popular framework la nivel global, folosit în dezvoltarea aplicațiilor web la nivel de client pentru realizarea de pagini web adaptive. Acesta permite dispunerea conținutului unei pagini în funcție de rezoluția ecranului pe care este vizualizat.

Angular reprezintă o platformă ce permite dezvoltatorilor să realizeze aplicații web dinamice. Acesta vine la pachet cu numeroase beneficii la nivelul structurii, aspectului și funcționalității paginilor web. Printre aceste se numără:

* Structurarea aplicației pe module și componente, lucru ce conduce la scăderea dependenței între diferitele părți ale aplicației
* Multiplatformă, caracteristică ce permite aplicației să fie utilizabilă pe orice tip de dispozitiv ce poate folosi un browser web
* Mecanismul de dependency injection
* Timpul de compilare scăzut

TypeScript este un limbaj de programare cu sursă deschisă, dezvoltat de Microsoft ce stă la baza platformei Angular. Acesta păstrează avantajele limbajului JavaScript la care se adaugă posibilitatea de a scrie cod orientat obiect.

Git este la ora actuală unul dintre cele mai folosite sisteme de versionare a condului. Cu ajutorul acestuia se pot gestiona modificările efectuate asupra unui cod sursă a unui proiect.

# Contribuții

Din propria experiență pot spune că realizarea unui configurații de sistem este o acțiune ce necesită deseori multe cunoștințe de ordin tehnic și după caz implică și investirea unui timp îndelungat analizei componentelor dorite. Astfel, am venit cu propunerea unei aplicații care să ușureze realizarea acestei acțiuni. Ideea aplicației îmi aparține în totalitate și a fost fondată pe baza faptului că în mod repetat mi-a fost solicitat ajutorul atunci când cineva intenționa să își achiziționeze un nou computer. Imediat cum m-am hotarât asupra tematicii, am înaintat propunerea către domnul profesor coordonator, din partea căruia am primit un feedback pozitiv și mai mult decât atât acesta mi-a sugerat ca prin intermediul aplicației să sprijin și companiile care se ocupă cu vanzarea de componente pentru computere. Tot împreună cu acesta am decis atât tehnologiile folosite în dezvoltarea aplicației, scenariile de utilizare ale aplicației și metodologia de lucru.

Aplicația proiectată, denumită intuitiv PC Builder își propune să rezolve o problemă comună multor persoane, cea a alegerii de componente compatibile pentru configurarea unui sistem de tip desktop și totodată să sprijine magazinele ce le comercializează.

În cadrul aplicației am implementat un design unic, intuitiv și prietenos cu utilizatorul. Pentru dezvoltarea acestuia m-am folosit de framework-ul Angular în versiunea cu numărul 5 și de cunoștințele de bază dobândite la disciplinele “Tehnologii Web” și “Dezvoltarea aplicațiilor web la nivel de client”. Pentru înțelegerea conceptelor aferente framework-ului mai sus menționat, am fost nevoie să parcurg un mediu de învățare online, întrucât acesta mi-a fost în întregime străin înaintea începerii dezvoltării acestei lucrări.

Pentru dezvoltarea părții de server am avut în vedere obținerea unei aplicații eficiente, ușor de implementat, utilizat și întreținut, care să respecte cât mai multe best-practice-uri specifice platformei de dezvoltare .NET.

Pe toată durata dezvoltării proiectului am folosit Git ca sistem de versionare. Acest lucru demonstrează că lucrarea și munca depusă în cadrul acesteia îmi aparțin în totalitate. Totodată se poate se poate observa caracterul evolutiv al codului și etapele intermediare prin care s-a obținut produsul finit.

Prezenta lucrare va fi structurată în două capitole, în cadrul cărora voi prezenta, pe rând, procesul de dezvoltare, principalele concepte folosite și detaliile de implementare prezente în cele două părți ale aplicației: server și client.

# Capitolul 1: Dezvoltarea aplicației server (API)

Așa cum am menționat și în secțiunea “Specificații tehnice”, aplicația PC Builder este constituită din două aplicații (module) independente ce comunică prin request-uri HTTP.

În acest capitol voi prezenta procesul de dezvoltare, conceptele folosite și detaliile de implementare ale modului server. La dezvoltarea acestui modul am avut în avut în vedere obținerea unei aplicații performante, eficiente și care să fie în același timp și ușor de implementat. De aceea m-am hotărât să realizez un API REST bazat pe framework-ul ASP .NET Core, întrucât acesta din urmă vine la pachet cu numeroase beneficii:

* Sursă deschisă
* Multiplatformă
* Caracter modular
* Include suport pentru dependy injection
* Performanță îmbunătățită

Caracterul modular este subliniat de faptul că framework-ul este expediat sub formă de pachete NuGet, astfel aplicației îi sunt puse la dispoziție doar dependințele necesare. De obicei, aplicațiile construite cu ASP .NET Core, în versiunea 2.x, care vizează platformele .NET Core necesită doar un singur pachet NuGet. Astfel se obține o aplicație ce include securitate mai strictă, performanță îmbunătățită și întreținere redusă. În aplicația dezvoltată, pe lângă pachetul de bază am mai folosit alte două pachete NuGet: Microsoft.EntityFrameworkCore și Microsoft.EntityFrameworkCore.SqlServer. Acestea au fost folosite la nivelul “Data” al proiectului pentru a permite creearea și gestionarea bazei de date.

Dependency injection este o tehnică pentru obținerea unui cuplaj redus între obiecte și dependențele lor. Cu alte cuvinte, o clasă nu instanțiază în mod direct obiecte de care are nevoie, ci îi sunt “injectate” cel mai adesea prin intermediul listei de argumente a constructorului. În aplicația dezvoltată, m-am folosit de această tehnică la scrierea codului pentru controlere, dar și pentru implementarea fiecărui repository de la nivelul “Business”, injectând dependențele prin intermediul constructorului.

## Arhitectura aplicației

În cadrul aplicației am folosit o arhitectură dispusă pe mai multe straturi (nivele). Mai exact, aplicația server a fost dezvoltată pe trei nivele: Data (stratul de date), Business (stratul de logică) și Web Api (stratul de serviciu). Acest lucru a condus la obținerea unei aplicații flexibile, reutilizabile și deschisă spre extensie. Totodată, împărțirea pe nivele a ajutat la creșterea scalabilității și mentenabilității, astfel aplicația permite modificarea sau adăugarea de noi nivele fără a fi nevoie rescrierea ei.

Odată cu dezvoltarea aplicației am inclus și șablonul de proiectare *Repository.* Acesta propune un nivel de abstractizare între obiectele din domeniu și logica de business a aplicației. Printre atuurile acestui șablon se faptul că simplifică scrierea codului și promovează mecanismul de *Dependency injection*.

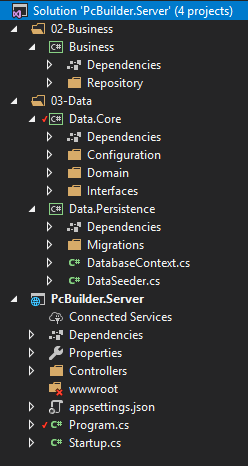
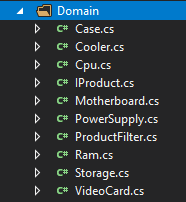


Fig. 3 Arhitectura aplicației

### Nivelul Data

Nivelul Data este o componentă foarte importantă a aplicației deoarece aici se realizează crearea bazei de date și managementul acesteia. De aceea, a fost nevoie de adăugarea pachetelor NuGet pentru Entity Framework Core și SQL Server. În linii mari, acest nivel cuprinde două proiecte de tip “class library” denumite intuitiv: Data.Core și Data.Persistance.

Proiectul Data.Core stă la baza aplicației, întrucât acesta cuprinde clasele de domeniu, clasele de configurare pentru fiecare entitate și interfețele pentru fiecare repository. Clasele de domeniu sunt în număr de opt și reprezintă întocmai componentele necesare pentru creearea unei configurări de sistem:

* Carcasă
* Procesor
* Cooler
* Placă de bază
* Memorie RAM
* Placă video
* Stocare
* Sursă

Fig. 4 Lista claselor de domeniu

Datorită faptului că entitățile menționate mai sus au proprietăți comune m-am decis să adaug o interfată denumită *IProduct,* care să le cuprindă. Astfel fiecare din cele opt entități va deține proprietățile interfeței prin simpla implementare a acesteia.

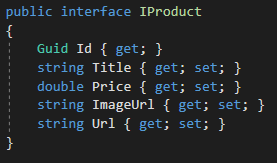


Fig. 5 Conținutul interfeței IProduct

public class Cpu: IProduct

{

public Guid Id { get; private set; }

public string Title { get; set; }

public double Price { get; set; }

public string ImageUrl { get; set; }

public string Url { get; set; }

public string Socket { get; set; }

public string Series { get; set; }

public string Core { get; set; }

public int Cores { get; set; }

public int Threads { get; set; }

public double BaseFrequency { get; set; }

public double TurboFrequency { get; set; }

public double Cache { get; set; }

public bool HasStockCooler { get; set; }

public string TypeOfRam { get; set; }

public int MaximumRamMemory { get; set; }

public double RamFrequency { get; set; }

public static Cpu Create(string title, double price, string imageUrl, string socket, string series, string core,

int cores, int threads, double baseFrequency, double turboFrequency, double cache, bool hasStockCooler,

string typeOfRam, int maximumRamMemory, double ramFrequency, string url)

{

var instance = new Cpu

{

Id = new Guid()

};

instance.Update(title, price, imageUrl, socket, series, core, cores, threads, baseFrequency, turboFrequency,

cache, hasStockCooler, typeOfRam, maximumRamMemory, ramFrequency, url);

return instance;

}

private void Update(string title, double price, string imageUrl, string socket, string series, string core, int cores,

int threads, double baseFrequency, double turboFrequency, double cache, bool hasStockCooler, string typeOfRam, int maximumRamMemory, double ramFrequency,string url)

{

Title = title;

Price = price;

ImageUrl = imageUrl;

Socket = socket;

Series = series;

Core = core;

Cores = cores;

Threads = threads;

BaseFrequency = baseFrequency;

TurboFrequency = turboFrequency;

Cache = cache;

HasStockCooler = hasStockCooler;

TypeOfRam = typeOfRam;

MaximumRamMemory = maximumRamMemory;

RamFrequency = ramFrequency;

Url = url;

}

}

Datorită faptului că componentele unui computer depinde foarte mult unele de altele, am implementat un obiect de filtrare cu ajutorul căruia reușesc să selectez din baza de date doar acele produse ce sunt compatibile cu componentele deja alese până la un anumit pas. Obiectul de filtrare are ca și proprietăți id-urile componentelor dintr-o configurație.

public class ProductFilter

{

public Guid CaseId { get; set; }

public Guid CpuId { get; set; }

public Guid CoolerId { get; set; }

public Guid MotherboardId { get; set; }

public Guid RamId { get; set; }

public Guid VideoCardId { get; set; }

public Guid StorageId { get; set; }

public Guid PowerSupplyId { get; set; }

public ProductFilter(string filter)

{

var filterObject = new JObject();

if (filter != null)

{

filterObject = JObject.Parse(filter);

if (!string.IsNullOrEmpty(filterObject["caseId"].ToString()))

{

CaseId = Guid.Parse(filterObject["caseId"].ToString());

}

if (!string.IsNullOrEmpty(filterObject["cpuId"].ToString()))

{

CpuId = Guid.Parse(filterObject["cpuId"].ToString());

}

if (!string.IsNullOrEmpty(filterObject["coolerId"].ToString()))

{

CoolerId = Guid.Parse(filterObject["coolerId"].ToString());

}

if (!string.IsNullOrEmpty(filterObject["motherboardId"].ToString()))

{

MotherboardId = Guid.Parse(filterObject["motherboardId"].ToString());

}

if (!string.IsNullOrEmpty(filterObject["ramId"].ToString()))

{

RamId = Guid.Parse(filterObject["ramId"].ToString());

}

if (!string.IsNullOrEmpty(filterObject["videocardId"].ToString()))

{

VideoCardId = Guid.Parse(filterObject["videocardId"].ToString());

}

if (!string.IsNullOrEmpty(filterObject["storageId"].ToString()))

{

StorageId = Guid.Parse(filterObject["storageId"].ToString());

}

if (!string.IsNullOrEmpty(filterObject["powersupplyId"].ToString()))

{

PowerSupplyId = Guid.Parse(filterObject["powersupplyId"].ToString());

}

}

}

}

Constructorul clasei ce definește obiectul de filtare, are ca argument un șir de caractere datorită faptului că de pe aplicația client request-urile HTTP conțin la nivelul de parametrilor un obiect de filtrare, cu proprietăți asemănătoare celui descris mai sus, ce este convertit în șir de caractere înainte de a fi trimis către aplicația server, iar pentru a putea fi folosit în continuare, acesta trebuie sa fie transformat îmtr-un obiect de tip *ProductFilter*.

Datorită acestui obiect de filtrare nu a mai fost nevoie să trasez relațiile dintre entitățile bazei de date, acestea din urmă ramânând independente unele de altele.

Pentru entitățile *case* și *cooler* am fost nevoit să salvez ca și proprietate o listă de șiruri de caractere reprezentând lista de tipuri de plăci de bază în cazul carcaselor sau lista de tipuri de sockete ale procesoarelor în cazul coolerelor. Acest lucru a fost posibil prin creearea a unei proprietăți de tip șir de caractere și a unei proprietăți, nemapate, de tip listă de șiruri de caractere. Prin intermediul metodei get, aplicată proprietății nemapate, se obține lista de șiruri de caractere, obținută prin deserializarea șirului de caractere, iar folosind metoda set lista este covertită în șir de caractere, prin serializare.

public class Case: IProduct

{

internal string MotherBoardFormFactor { get; set; }

public Guid Id { get; private set; }

public string Title { get; set; }

public double Price { get; set; }

public string ImageUrl { get; set; }

public string Url { get; set; }

public string Type { get; set; }

public int NumberOfSlots { get; set; }

public int CoolerHeight { get; set; }

public int VideoCardWidth { get; set; }

public int Fans { get; set; }

public int TotalFans { get; set; }

[NotMapped]

public List<string> \_motherboardFormFactor

{

get => MotherBoardFormFactor == null ? null : JsonConvert.DeserializeObject<List<string>>(MotherBoardFormFactor);

set => MotherBoardFormFactor = JsonConvert.SerializeObject(value);

}

Așa cum am menționat și în descrierea succintă a arhitecturii aplicației, am folosit șablonul de proiectare *Repository*. Astfel proiectul Data.Core mai conține pe lângă clasele de domeniu și interfețele pentru repository. Acestea joacă un rol foarte important în aplicație deoarece prin intermediul lor și al implementărilor din nivelul *Business* se propune un nivel de abstractizare între obiectele din domeniu și logica de business a aplicației.

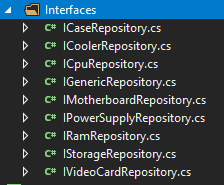


Fig. 6 Listă cu interfețele pentru *repository*

Pentru implementarea acestui șablon fiecărei clase de domeniu îi corespunde o interfață care expune un set de operații CRUD (create, read, update, delete). Pentru a reduce generarea de cod duplicat am definit o interfață generică care poate acceptă obiecte derivate din entitatea de bază *IProduct.*

public interface IGenericRepository<T> where T: IProduct

{

Task<bool> DeleteAsync(Guid id);

Task<List<T>> GetAllAsync(ProductFilter filter);

Task<T> GetByIdAsync(Guid id);

Task<T> InsertAsync(T entity);

Task<bool> UpdateAsync(T entity);

}

Astfel, fiecare interfață de repository ce va extinde interfața generică îi va moșteni toate semnăturile metodelor ei.

public interface ICaseRepository: IGenericRepository<Case>

{}

O altă parte componentă a acestui proiect, o reprezintă clasele de configurare. Acestea alcătuiesc implementarea *Fluent Api-ului* și sunt folosite ca alternativă la *DataAnnotations*, ce se realizează prin atribute adăugate la nivelul proprietăților din clasele de domeniu. Întrucât clasele de domeniu sunt extinse dintr-o clasă generică am aplicat acest model și în cazul claselor de configurare. Astfel, am implementat o clasă generică *ConfigureUtils* ce acceptă clase ce implementează interfața generică *IProduct*.

public sealed class ConfigureUtils

{

public static void ProductConfigure<T>(EntityTypeBuilder<T> builder) where T: class, IProduct

{

builder.HasKey(t => t.Id);

builder.Property(t => t.Title).HasMaxLength(255).IsRequired();

builder.Property(t => t.Price).IsRequired();

builder.Property(t => t.ImageUrl).IsRequired();

}

}

Celelalte clase de configurare se vor folosi de configurările realizate în această clasă generică, prin intermediul apelului metodei *ProductConfigure*.

public class MotherboardConfiguration: IEntityTypeConfiguration<Motherboard>

{

public void Configure(EntityTypeBuilder<Motherboard> builder)

{

ConfigureUtils.ProductConfigure(builder);

builder.Property(m => m.FormFactor).IsRequired();

builder.Property(m => m.GraphicInterface).IsRequired();

builder.Property(m => m.M2).IsRequired();

builder.Property(m => m.MaximumRamMemory).IsRequired();

builder.Property(m => m.RamFrequency).IsRequired();

builder.Property(m => m.RamSlots).IsRequired();

builder.Property(m => m.TypeOfRam).IsRequired();

builder.Property(m => m.Socket).IsRequired();

builder.Property(m => m.Sata).IsRequired();

}

}

Pentru buna funcționare a logicii aplicației, în cadrul acestor clase de configurare sunt setate principalele proprietăți mandare ale fiecărui tip de entitate. Spre exemplu, nu este posibilă întregistrarea în baza de date a unei plăci de bază fără a se specifica factorul de forma, interfața grafică compatibilă, slotul M2, memoria RAM suportată, soclul pentru procesor și porturile de tip Sata deoarece aceste proprietăți sunt folosite atunci când se realizează filtrarea unui produs de un anumit tip. Astfel dacă utilizatorul alege la un moment dat un procesor cu soclul 1151, atunci când acesta va dori să aleagă o placă de bază, aplicația îi va sugera doar plăcile de bază a căror soclu pentru procesor este întocmai 1151.

Cel de al doilea proiect de la nivelul de date, denumit *Data.Persitance*, conține un folder cu migrări generat automat la crearea migrării inițiale, clasa *DatabaseContext* și clasa *DataSeeder*. Clasa *DatabaseContext* reprezintă principala legătura a aplicației cu baza de date. Pentru generarea acesteia a fost necesară instalarea pachetelor NuGet *Microsoft.EntityFrameworkCore* și *Microsoft.EntityFrameworkCore.SqlServer.*

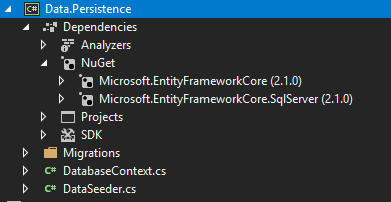


Fig. 6 Conținutul proiectului Data.Persitence

Clasa *DatabaseContext* extinde clasa *DbContext* din *Entity Framework Core* și conține mulțimea tuturor entităților. Tot în cadrul acesteia sunt aplicate și configurările pentru entități.

public sealed class DatabaseContext: DbContext

{

public DatabaseContext(DbContextOptions<DatabaseContext> options) : base(options)

{}

public DbSet<Case> Cases { get; set; }

public DbSet<Cpu> Cpus { get; set; }

public DbSet<Cooler> Coolers { get; set; }

public DbSet<Motherboard> Motherboards { get; set; }

public DbSet<Ram> Rams { get; set; }

public DbSet<VideoCard> VideoCards { get; set; }

public DbSet<Storage> Storages { get; set; }

public DbSet<PowerSupply> PowerSupplies { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.ApplyConfiguration(new CaseConfiguration());

modelBuilder.ApplyConfiguration(new CpuConfiguration());

modelBuilder.ApplyConfiguration(new CoolerConfiguration());

modelBuilder.ApplyConfiguration(new MotherboardConfiguration());

modelBuilder.ApplyConfiguration(new RamConfiguration());

modelBuilder.ApplyConfiguration(new VideoCardConfiguration());

modelBuilder.ApplyConfiguration(new StorageConfiguration());

modelBuilder.ApplyConfiguration(new PowerSupplyConfiguration());

}

}

Clasa DataSeeder expune o metodă folosită la inițializarea bazei de date. În cadrul acesteia sunt create mai multe instanțe ale fiecărui tip de produs și populate cu date veritabile, după care aceste instanțe sunt adăugate în baza de baza prin intermediul contextului.

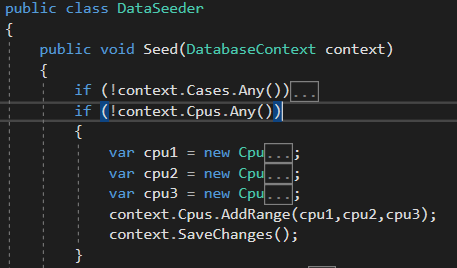


Fig. 7 Adăugarea de procesoare folosind *DbSeeder*

### Nivelul Business

La nivelul Business al aplicației server avem doar un proiect de tip class library în cadrul căruia regăsim clasele ce implementează concret interfețele pentru repository de la nivelul data. Ca și în cazul interfețelor, există o clasă generică *GenericRepository* ce implementează interfața *IGenericRepository* și expune metodele de bază necesare pentru lucrul cu entități.

public class GenericRepository<T>: IGenericRepository<T> where T: class, IProduct

{

protected readonly DatabaseContext \_context;

protected readonly DbSet<T> \_entities;

public GenericRepository(DatabaseContext context)

{

\_context = context;

\_entities = context.Set<T>();

}

public virtual async Task<bool> DeleteAsync(Guid id)

{

var entity = \_entities.FirstOrDefault(e => e.Id == id);

if (entity == null)

return await \_context.SaveChangesAsync() > 0;

\_entities.Remove(entity);

return await \_context.SaveChangesAsync() > 0;

}

public virtual async Task<List<T>> GetAllAsync(ProductFilter filter)

=> await \_entities.ToListAsync();

public virtual async Task<T> GetByIdAsync(Guid id)

=> await \_entities.FirstOrDefaultAsync(e => e.Id == id);

public virtual async Task<T> InsertAsync(T entity)

{

\_entities.Add(entity);

await \_context.SaveChangesAsync();

return entity;

}

public virtual async Task<bool> UpdateAsync(T entity)

{

\_entities.Update(entity);

return await \_context.SaveChangesAsync() > 0;

}

}

Metoda *GetAllAsync* primește ca și argument un obiect de filtrare și întoarce toate entitățile ce respectă anumite constrângeri impuse de acesta.

Metoda *GetByIdAsync* primește ca și parametru un identificator unic și returnează obiectul a cărui *Id* este egal cu cel primit ca parametru sau null în caz contrar.

Metoda *DeleteAsync* primește ca și argument un identificator unic și returnează True în cazul în care este șters înregistrarea corespunzătoare indentificatorului sau False în caz contrar.

Metoda *UpdateAsync* primește drept parametru o entitate de tipul IProduct și modifică elementul respectiv în baza de date și returnează rezultatul modificării.

Metoda *InsertAsync* are structură asemănătoare celei de update doar că entitatea primită drept parametru este adăugată în baza de date.

După cum se observă, în codul de mai sus, metodele definite au un comportament asincron, acestea fiind constrânse prin utilizarea cuvintelor cheie *async* și *await*. Mai mult decât atât aceste metode sunt declarate virtual pentru a suporta suprascrierea lor, ceea ce s-a și intamplat în cazul metodei *GetAllAsync.*

În principiu, întreaga logică de sugerare a componentelor pentru un anumit pas al unei configurări de sistem s-a facut la acest nivel. De aceea voi oferi cateva detalii de implementare referitoare la acest lucru.

public override async Task<List<Cooler>> GetAllAsync(ProductFilter filter)

{

if (filter != null && filter.CpuId != Guid.Empty && filter.CaseId != Guid.Empty)

{

var computerCase = await \_context.Cases.FirstOrDefaultAsync(cc => cc.Id == filter.CaseId);

var cpu = await \_context.Cpus.FirstOrDefaultAsync(c => c.Id == filter.CpuId);

return await \_entities.Where(cool => cool.\_compatibleSockets.Contains(cpu.Socket).Equals(true) && cool.Height < computerCase.CoolerHeight).ToListAsync();

}

return await \_entities.ToListAsync();

}

}

În cazul componentei cooler, sugerarea produselor compatibile se face prin preluarea din obiectul de filtrare a identificatorilor unici pentru carcasă și procesor și crearea de instanțe pentru fiecare dintre acestea, după care se returnează acele coolere care au înălțimea mai mică decât înalțimea maximă a carcasei și permit instalarea pe soclul procesorului ales.

public class MotherboardRepository: GenericRepository<Motherboard>, IMotherboardRepository

{

public MotherboardRepository(DatabaseContext context) : base(context)

{

}

public override async Task<List<Motherboard>> GetAllAsync(ProductFilter filter)

{

if (filter != null && filter.CaseId != Guid.Empty && filter.CpuId != Guid.Empty)

{

var computerCase = await \_context.Cases.FirstOrDefaultAsync(cc => cc.Id == filter.CaseId);

var cpu = await \_context.Cpus.FirstOrDefaultAsync(c => c.Id == filter.CpuId);

return await \_entities.Where(m => computerCase.\_motherboardFormFactor.Contains(m.FormFactor) && m.Socket.Equals(cpu.Socket) &&

m.MaximumRamMemory <= cpu.MaximumRamMemory && m.RamFrequency >= cpu.RamFrequency && m.TypeOfRam.Equals(cpu.TypeOfRam)).ToListAsync();

}

return await \_entities.ToListAsync();

}

}

În cazul plăcilor de bază, acestea sunt sugerate pe baza listei de factori de formă a carcasei, a soclului procesorului, și a informațiilor referitoare la memoria RAM suportată de procesor: tipul memoriei, dimensiunea maximă a memoriei și frecvența ei.

Lista plăcilor video returnată de metoda *GetAllAsync* conține acele produse care încap în carcasa aleasă întru-un pas interior și sunt compatibile cu interfața grafică suportată de placa de bază.

În cazul memoriei RAM, sugerarea de produse compatibile se face în funcție de informațiile plăcii de bază. Mai exact se verifică tipul memoriei suportate, frecvența maximă cu care aceasta poate funcționa și capacitatea pe care aceasta o poate gestiona.

public override async Task<List<Ram>> GetAllAsync(ProductFilter filter)

{

if (filter != null && filter.MotherboardId != Guid.Empty)

{

var motherboard = await \_context.Motherboards.FirstOrDefaultAsync(m => m.Id == filter.MotherboardId);

return await \_entities.Where(r => r.Type.Equals(motherboard.TypeOfRam) && r.Capacity <= motherboard.MaximumRamMemory &&

r.Frequency <= motherboard.RamFrequency).ToListAsync();

}

return await \_entities.ToListAsync();

}

Întrucât produsele din categoria stocare sunt de mai multe tipuri: SSD cu interfață sata, SDD cu interfață de tip M.2 sau HDD, filtrarea lor se face pe baza factorului de formă și a interfeței pe care acestea funcționează, dar și în funcție de placa de bază aleasă.

public override async Task<List<Storage>> GetAllAsync(ProductFilter filter)

{

if (filter != null && filter.MotherboardId != Guid.Empty)

{

var storages = new List<Storage>();

var motherboard = await \_context.Motherboards.FirstOrDefaultAsync(m => m.Id == filter.MotherboardId);

if (motherboard.M2 != 0)

storages.AddRange(\_entities.Where(s => s.FormFactor.Equals("M.2")));

storages.AddRange(\_entities.Where(s => s.Interface.Equals("SATA-III") && !s.FormFactor.Equals("M.2")));

return storages.ToList();

}

return await \_entities.ToListAsync();

}

}

### Nivelul Web Api

# Bibliografie

*SQL Server*. (,). Retrieved from SqlShack: https://www.sqlshack.com/sql-server-table-structure-overview/

*ASP .NET Core*. (n.d.). Retrieved from https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.1

*Dependency injection*. (n.d.). Retrieved from https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.1

*Entity Framework Core*. (n.d.). Retrieved from EF Core Overview: https://docs.microsoft.com/en-us/ef/core/

*LINQ*. (n.d.). Retrieved from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/