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 Licenţă

Titlul, Numele şi prenumele _____

Data _____ Semnătura _____

DECLARAŢIE privind originalitatea conţinutului lucrării de licenţă

Subsemnatul(a) ……

domiciliul în ……..

născut(ă) la data de ……., identificat prin CNP ……., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de …….. specializarea ……., promoţia ……., declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul: _____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susţină în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării falsificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „*Titlul complet al lucrării*”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

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

Iaşi, *data*

Absolvent *Prenume Nume*

_____

(semnătura în original)

Cuprins

Introducere

Computerul (1) reprezintă una dintre cele mai strălucite invenții ale societății moderne, însă istoria acestuia se întinde până acum mai bine de 2500 de ani, când oamenii vremurilor respective au inventat un instrument pentru calcule aritmetice elementare.

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, astfel ele devenind din ce în ce mai mici, mai performante și accesibile publicului larg.

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

Nevoia achiziționării unui computer a devenit un lucru inevitabil în viața omului modern. În acest context, putem afirma că în general o persoană non-tehnică întâmpină dificultăți la alegerea componentelor uni computer și de aceea majoritatea sunt tentați să achiziționeze un PC gata configurat de către companiile ce se ocupă cu vânzarea acestor produse. Astfel aceștia se pot trezi cu computere care sunt mult prea performante pentru nevoile lor și care au un cost pe măsură, ori pot cădea în capcana unor magazine care comercializează produse din generații mai vechi prin diverse campanii de marketing, lichidări de stoc sau altele asemenea.

Un alt risc pe care și-l asumă o persoană care nu cunoaște foarte bine aspectele legate de structura și funcționalitatea unui computer este acela de a achiziționa diverse componente ale unor brand-uri pe care aceasta le consideră familiare, dar care să fie incompatibile unele cu altele.

De aceea, prin aplicația prezentată în această lucrarea îmi propun să rezolv o parte din problemele prezentate mai sus. Aplicația va ghida un utilizator în configurarea unui computer și se va asigura că componentele alese sunt compatibile. Totodată, aceasta îi va afișa în mod constant prețul și progresul configurării, astfel încât utilizatorul are posibilitatea să se reorienteze în privința alegerii componentelor.

Prezenta lucrare va fi structurată în trei capitole, în cadrul primelor două voi prezenta, pe rând, procesul de dezvoltare, principalele concepte folosite și detaliile de implementare prezente în cele două module (aplicații): server și client, iar în cel de-al treilea capitol voi prezenta funcționalitățile aplicației și cum interacționează un utilizator cu aceasta.

1.1 Motivație

Computerele din zilele noastre au la bază opt componente: procesorul ce trebuie neapărat să fie răcit de un cooler, placa de bază, memoria RAM, placa video, unitatea 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 opt 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, întrucât vorbim de un număr 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 configureze un PC.

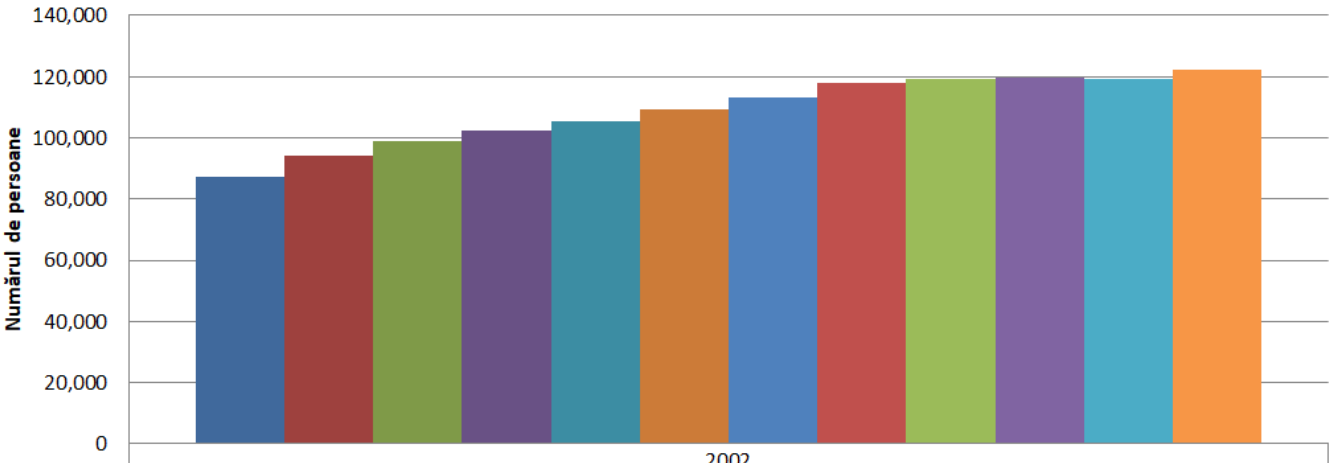
În zilele noastre cumpărarea unui computer fără a cere ajutorul nimănui poate fi o perspectivă descurajantă. Există literalmente sute de modele diferite din care poți alege, multe dintre ele arătând aproape identic, însă cu diferențe destul de mari la nivel de specificații. Nu este deloc surprinzător 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 cunoștințe 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.

1.2 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.

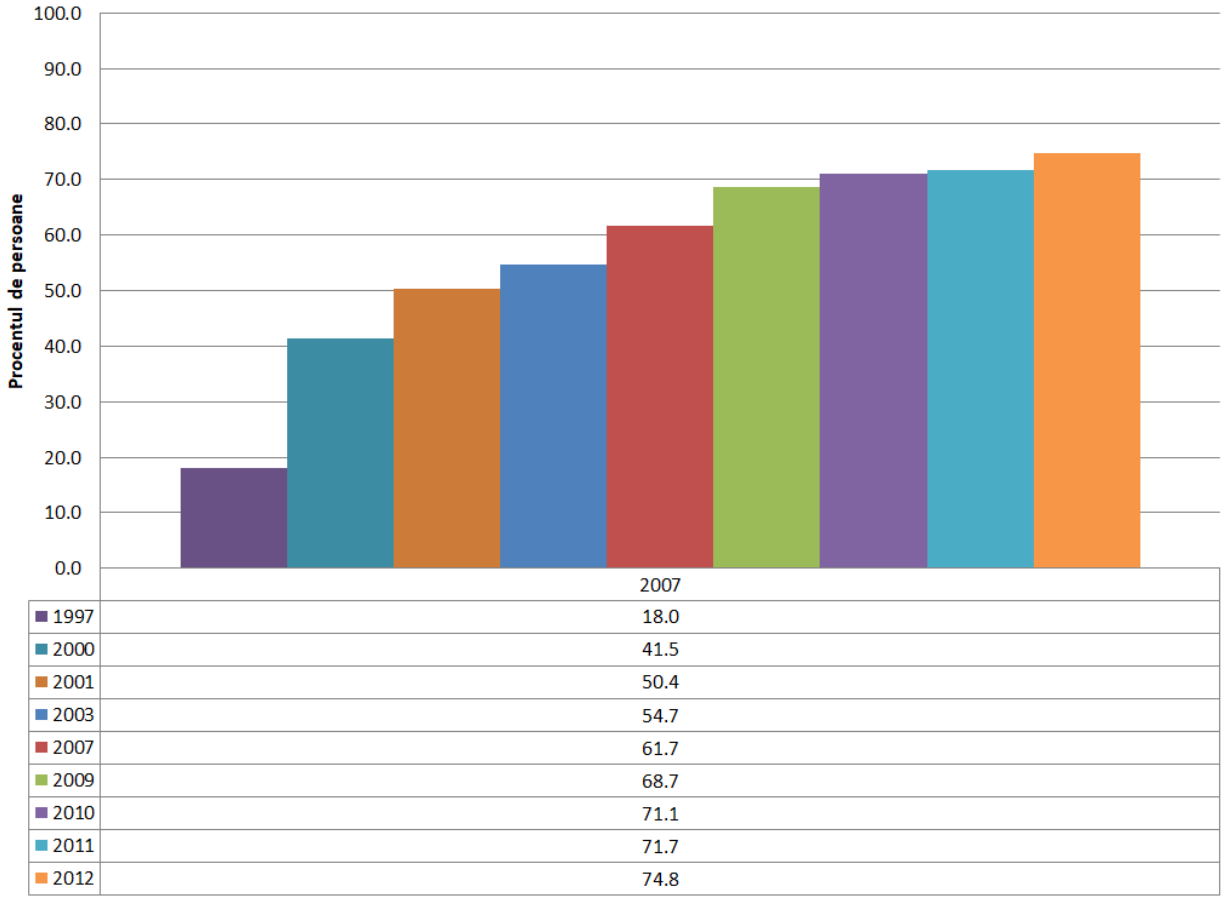
Un studiu realizat de cei de la "*United States Census Bureau*" pe un eșantion de 150.000 de persoane confirmă faptul că interesul acestora pentru computere a crescut foarte mult in ultimii 30 de ani. (2)



	2002
1984	87,073
1989	94,061
1993	98,736
1997	102,158
2000	105,247
2001	109,106
2003	113,126
2007	117,840
2009	119,296
2010	119,545
2011	119,250
2012	122,048

Fig. 1 - Evoluția numărului de persoane ce dețin un computer între anii 1984-2012

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. (2)



	2007
1997	18.0
2000	41.5
2001	50.4
2003	54.7
2007	61.7
2009	68.7
2010	71.1
2011	71.7
2012	74.8

Fig. 2 – Evoluția procentajului persoanelor cu acces la internet între anii 1997-2012

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. Acesta este un motor de căutare hardware prin care 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.

1.3 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 (3) 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 (4) 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 (5) 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* (6), 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 (7) 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 (8) 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 (9)
- Timpul de compilare scăzut

TypeScript (10) 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 codului. 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 hotărât 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 ofer sprijin și companiilor ce se ocupă cu vânzarea de componente pentru computere. Tot împreună cu acesta am decis atât tehnologiile folosite în dezvoltarea aplicației, cât și 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 ofere sprijin magazinelor 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 5. Pentru înțelegerea conceptelor aferente framework-ului mai sus menționat, am fost nevoit 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, scalabile, 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 observa caracterul evolutiv al codului și etapele intermediare prin care s-a obținut produsul finit.

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 modulului 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 dependency 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 dependențele 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 crearea și gestionarea bazei de date.

Dependency injection (10) este o tehnică pentru obținerea unui cuplaj redus între obiecte. 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.

1.1 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. (11)

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*.

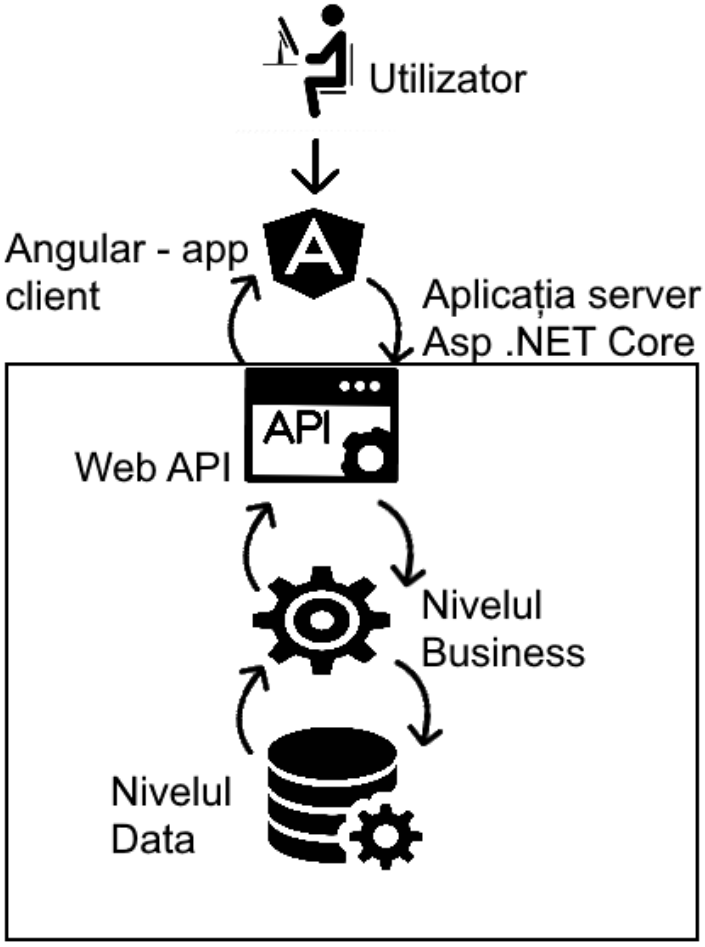


Fig. 3 - Arhitectura aplicației

1.1.1 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 crearea unei configurări de sistem:

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

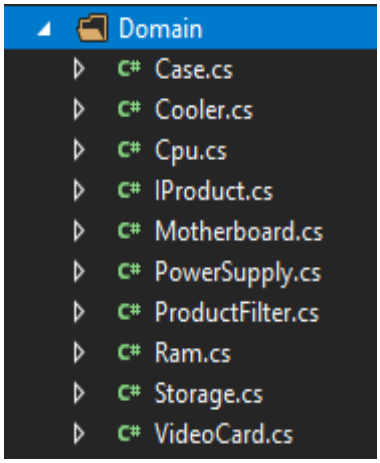


Fig. 4 - Lista claselor de domeniu

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

```
public interface IProduct
{
	Guid Id { get; }
	string Title { get; set; }
	double Price { get; set; }
	string ImageUrl { get; set; }
	string Url { get; set; }
}
```

Fig. 5 - Conținutul interfeței *IProduct*

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());
                }}}}
```

Fig. 6 - Conținutul clasei *ProductFilter*

Constructorul clasei ce definește obiectul de filtrare, are ca argument un șir de caractere datorită faptului că de pe aplicația client request-urile HTTP conțin la nivelul 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 într-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 rămânând independente unele de altele.

Pentru entitățile *case* și *cooler* am fost nevoit să salvez drept 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 crearea 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 convertită 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);
        }
```

Fig. 7 - Conținutul clasei *Case*

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 situate la nivelul *Business* se creează o abstracție între obiectele din domeniu și logica de business a aplicației.

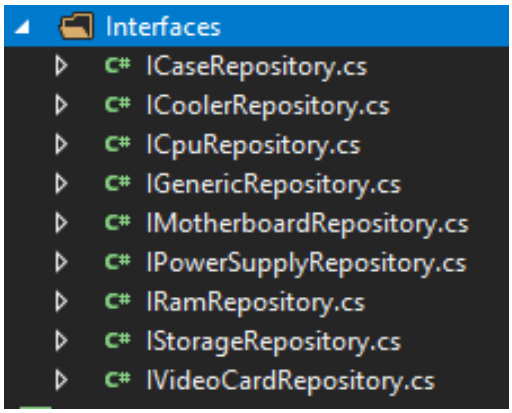


Fig. 8 - Listă cu interfețele pentru *repository*

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);
}
```

Fig. 9 - Interfața generică

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

```
public interface ICaseRepository: IGenericRepository<Case>
    {}
```

Fig. 10 - Interfața pentru repository

O altă parte componentă a acestui proiect o reprezintă clasele de configurare construite respectând șablonul *Fluent Api*. Acestea 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();
        }
    }
```

Fig. 11 - Fișierul generic de configurare

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();
        }
    }
```

Fig. 12 - Exemplu de fișier de configurare

Pentru buna funcționare a logicii aplicației, în cadrul acestor clase de configurare sunt setate principalele proprietăți mandatare ale fiecărui tip de entitate. Spre exemplu, nu este posibilă înregistrarea în baza de date a unei plăci de bază fără a se specifica factorul de formă, interfața grafică compatibilă, prezența soclului de tip 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 director 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 utilizarea corectă a acesteia a fost necesară instalarea pachetelor NuGet *Microsoft.EntityFrameworkCore* și *Microsoft.EntityFrameworkCore.SqlServer*.

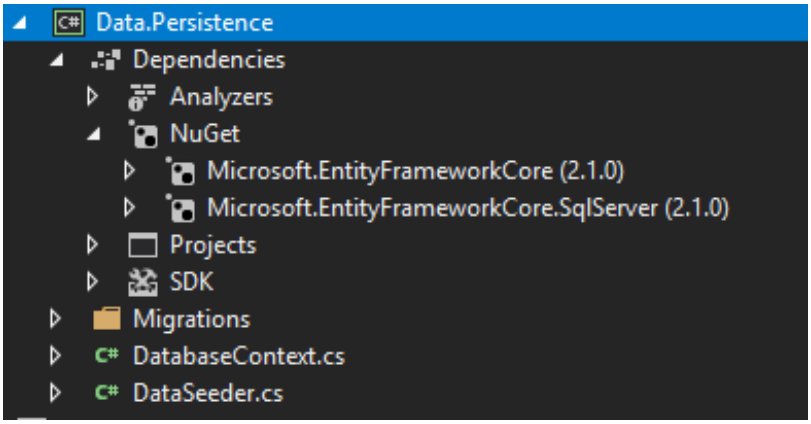


Fig. 13 - Conținutul proiectului Data.Persitence

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());
        }
    }
```

Fig. 14 - Conținutul clasei DatabaseContext

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.

```
public class DataSeeder
{
    public void Seed(DatabaseContext context)
    {
        if (!context.Cases.Any())...
        if (!context.Cpus.Any())
        {
            var cpu1 = new Cpu...;
            var cpu2 = new Cpu...;
            var cpu3 = new Cpu...;
            context.Cpus.AddRange(cpu1,cpu2,cpu3);
            context.SaveChanges();
        }
```

Fig. 15 - Adăugarea de procesoare folosind *DbSeeder*

1.1.2 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;
        }
}
```

Fig. 16 - Implementarea pentru un *repository*

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

Metoda *GetByIdAsync* primește drept 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 parametru un identificator unic și returnează *True* în cazul în care este șters înregistrarea corespunzătoare identificatorului sau *False* în caz contrar.

Metoda *UpdateAsync* primește drept parametru o entitate de tipul IProduct și modifică elementul respectiv în baza de date după care 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. Mai mult decât atât aceste metode sunt declarate virtual pentru a suporta suprascrierea lor, ceea ce s-a și întâmplat în cazul metodei *GetAllAsync*.

În principiu, întreaga logică de sugerare a componentelor pentru un anumit pas al unei configurări de sistem s-a făcut la acest nivel. De aceea voi oferi câteva 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();
      }
}
```

Fig. 17 - Logica de sugerare a coolerelor

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 înălțimea 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();

        }
    }
```

Fig. 18 - Logica de sugerare a plăcilor de bază

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 anterior 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();

        }
```

Fig. 19 - Logica de sugerare a memoriei RAM

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, 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();

        }
    }
```

Fig. 20 - Logica de sugerare a stocării

1.1.3 Nivelul Web API

Cel de al-treilea nivel al aplicației server, este reprezentat de un proiect de tip *Web API*. În aplicația mea, principalul rol al acestui proiect este de a expune logica din modulul server spre a fi utilizată de modulul client. Acest lucru se realizează pe baza claselor de tip *Controller*.

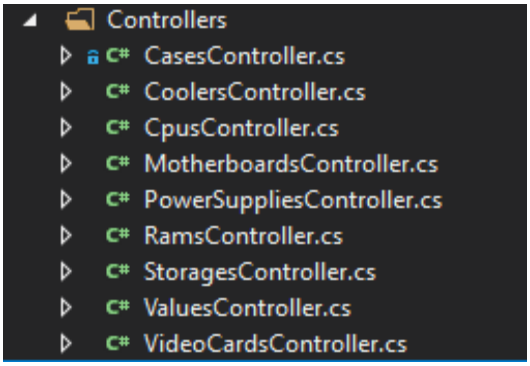


Fig. 21 - Clasele de tip controller din Web API

Acestea au implementate o serie de metode ce pot fi apelate prin intermediul request-urilor HTTP.

```
    public class MotherboardsController : Controller
    {
        private readonly IMotherboardRepository _repository;
        public MotherboardsController(IMotherboardRepository repository)
        {
            _repository = repository;
        }
        [HttpGet]
        public async Task<List<Motherboard>> Get(string filterObject)
        {
            var productFilter = new ProductFilter(filterObject);
            return await _repository.GetAllAsync(productFilter);
        }
        [HttpGet("{motherboardId}")]
        public async Task<Motherboard> GetById(Guid motherboardId)
        {
            return await _repository.GetByIdAsync(motherboardId);
        }
        [HttpPost]
        public async Task<Motherboard> Insert([FromBody] Motherboard
motherboard)
        {
            return await _repository.InsertAsync(motherboard);

        }
          }
```

Fig. 22 - Exemplu de controller

După cum se poate observa, într-un *controller* sunt injectate, prin intermediul constructorului, serviciile de care acesta are nevoie pentru a-și îndeplini scopul. *MotherboardsController* are rolul de a expune operațiile implementate în clasa *MotherboardRepository*, de la nivelul *Business*.

Metoda *Get este* accesibilă la ruta *api/motherboards* și prin intermediul acesteia se obțin toate plăcile de bază, din baza de date, ce respectă constrângerile impuse de obiectul de filtrare.

Metoda *GetById* este accesibilă la ruta *api/motherboards/Id* și întoarce din baza de date produsul a cărui *Id* corespunde identitificatorului unic primit drept parametru.

Controller-ul mai conține și o altă metodă accesibilă la ruta *api/motherboards* prin intermediul unui request de tip *HttpPost.* Această metodă, denumită în mod intuitiv *Insert*, preia datele din corpul request-ului după care apelează funcția ce inserează în baza de date noul produs.

Toate celelalte controllere au structură asemănătoare celei prezentate, singura diferență fiind tipul de produs.

Pe lângă clasele de tip *Controller,* proiectul *Web API* mai conține și alte două clase ce sunt generate automat la crearea acestuia: *Program* și *Startup.* Clasa *Program* conține punctul de intrare în execuția aplicației, aceasta fiind metoda statică *Main.* În cadrul acesteia am adăugat modalitatea de inițializare a bazei de date prin intermediul *DataSeeder-ului* definit la nivelul *Data*. Clasa *Startup* reprezintă nucleul de configurare al proiectului. Prin intermediul acesteia sunt adăugate serviciile ce pot fi injectate în clasele de tip *controller.*

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

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
            services.AddScoped(typeof(DataSeeder));

            services.AddScoped<ICaseRepository, CaseRepository>();
            services.AddScoped<ICpuRepository, CpuRepository>();
            services.AddScoped<ICoolerRepository, CoolerRepository>();
            services.AddScoped<IMotherboardRepository,
MotherboardRepository>();
            services.AddScoped<IRamRepository, RamRepository>();
            services.AddScoped<IVideoCardRepository, VideoCardRepository>();
            services.AddScoped<IStorageRepository, StorageRepository>();
            services.AddScoped<IPowerSupplyRepository,
PowerSupplyRepository>();
}
```

Fig. 23 - Metoda de configurare a serviciilor din clasa Startup

Capitolul 2: Dezvoltarea aplicației client

În acest capitol voi prezenta procesul de dezvoltare, conceptele folosite și detaliile de implementare ale aplicației client. Obiectivul principal al acestei aplicații a fost obținerea unui design unic, intuitiv și prietenos cu utilizatorul. De aceea, pentru dezvoltarea acesteia am folosit framework-urile *Angular* 5 și *Bootstrap,* în versiunea 4.1, prin intermediul limbajelor HTML, CSS și TypeScript. Mediul de dezvoltare utilizat în dezvoltarea aplicației client a fost *Visual Studio Code.*

1.2 Arhitectura aplicației

Angular reprezintă unul dintre cele mai moderne și folosite framework-uri de front-end la ora actuală datorită flexibilității și performanțelor ridicate. Acesta permite dezvoltatorilor să realizeze aplicații web dinamice de tip *Single Page Application*. Totodată, acesta vine la pachet cu numeroase beneficii la nivelul structurii, aspectului și funcționalității paginilor web.

Principalul avantaj al utilizării framework-ului *Angular* este acela că structurarea aplicației se realizează pe module și componente, lucru ce conduce la scăderea dependenței între diferitele părți ale aplicației.

Un modul reprezintă contextul de compilare pentru un set de componente dedicat unui anumit flux de lucru sau a unui set de capabilități strâns legate.

În general, o aplicație conține mai multe module, fiecare dintre acestea gestionând o anumită funcționalitate sau un flux de lucru. Organizarea aplicației în module funcționale distincte ajută la gestionarea mai ușoară a procesului de dezvoltare, încurajează scrierea unui cod reutilizabil și contribuie la dezvoltarea unei aplicații bine organizate și ușor de întreținut.

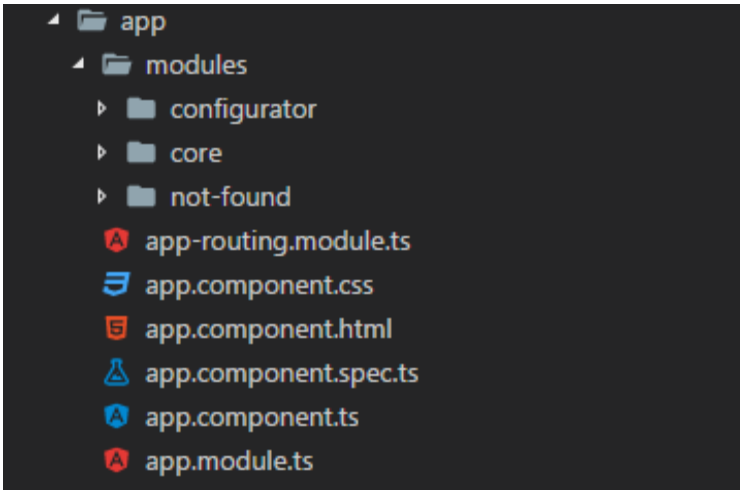


Fig. 24 - Organizarea aplicației client pe module

1.2.1 Modulul App

După cum se poate observa aplicația client este constituită din modulul rădăcină *(AppModule)* ce oferă mecanismul de pornire a aplicației. În cadrul acestuia sunt importate alte trei module, fiecare dintre acestea reprezentând un set important de funcționalități ale aplicației.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { NotFoundModule } from './modules/not-found/not-found.module';
import { AppRoutingModule } from './app-routing.module';
import { CoreModule } from './modules/core/core.module';
@NgModule({
  declarations: [
    AppComponent,
],
  imports: [
    BrowserModule,
    AppRoutingModule,
    CoreModule,
    NotFoundModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Fig. 25 - Conținutul modulului rădăcină

1.2.2 Modulul Core

Acest modul este constituit din trei componente, fiecare dintre acestea gestionând o pagină din aplicație.

O componentă este un tip special de directivă ce reprezintă elementul fundamental de construcție a unei pagini din aplicația client. Comportamentul acesteia este descris prin intermediul unei clase având un decorator special *(@Component)* cu ajutorul căruia sunt setate metadate specifice:

- *Selector*, prin intermediul acestuia este setat numele tag-ului ce trebuie folosit pentru a instanția o componentă
- *TemplateUrl*, ce face referire la calea către fișierul în care găsim șablonul componentei (codul html)
- *StyleUrls,* prin intermediul căruia putem adăuga o listă de fișiere ce conțin codul pentru stilizare

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
  title = 'app';

  constructor() { }

  ngOnInit() {
  }

}
```

Fig. 26 - Conținutul componentei *home*

Componenta *Home* gestionează pagina principală a aplicației. De aici, prin intermediul meniului, utilizatorul are posibilitate să navigheze către configurator, către pagina de tutorial sau către pagina *community*, fiecare dintre acestea fiind gestionată de către o componentă specifică. Detaliile legate de aceste pagini pot fi vizualizate în capitolul "Prezentarea aplicației".

1.2.3 Modulul Configurator

În cadrul acestui modul este modelată principala funcționalitate a aplicației și anume configuratorul pentru computere. În linii mari, această funcționalitate este oferită prin intermediul unui formular de tip *wizard*, alcătuit din nouă pași, fiecare dintre aceștia având asociați câte o componentă.

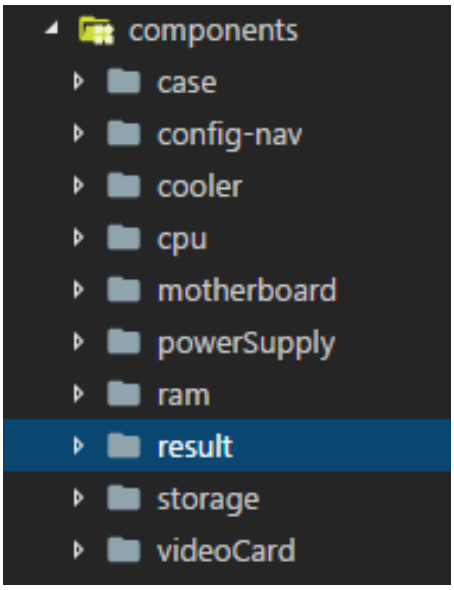


Fig. 27 - Lista componentelor din modulul Configurator

Componenta *config-nav* modelează meniul formularului. Prin intermediul acesteia utilizatorul este informat cu privire la pasul curent din formular și nivelul progresului său.

Componenta *result* modelează ultimul pas al formularului, respectiv pagina de sfărșit a acestuia. Pe această pagină îi este afișată utilizatorului configurația aleasă, pe care acesta o poate distribui pe pagina *community*. Mai mult decât atât, utilizatorul poate naviga către pagina fiecărui produs afișat, unde acesta are posibilitate achiziționa respectivul produs.

Restul componentelor modelează câte unul din pașii formularului. Acestea au un comportament destul de asemănător și se ocupă de afișarea unei liste de produse de un anumit tip.

```
<div class="container-fluid">
 <h3>Please select a product:</h3>
 <div class="col-sm-12">
   <ul class="cards" >
     <li class="cards__item" *ngFor="let case of cases">
         <app-case-card[case]="case"
         (caseEvent)="caseSelected($event)"></app-case-card>
     </li>
   </ul>
 </div>
 <div class="footer">
     <div class="col-sm-12 footer_content">
       <div class="col-sm-3">
           <p><span class="total_price">Total: ${{currentTotalPrice | number
            : '1.2-2'}}</span></p>
       </div>
       <div class="col-sm-6">
           <button class="btn btn-danger" routerLink="">Quit</button>
           <button class="btn btn-success" [disabled]="isDisabled"
           (click)="goToCPU()">Next ></button>
       </div>
     </div>
   </div>
</div>
```

Fig. 28 - Exemplu de șablon al unei componente ce modelază un pas al formularului

Fiecare element din listă este generat pe baza unei alte componente. Aceasta din urmă, comunică cu componenta părinte prin intermediul evenimentelor.

```
@Component({
  selector: 'app-case-card',
  templateUrl: './case-card.component.html',
  styleUrls: ['./case-card.component.css']
})
export class CaseCardComponent implements OnInit {

  @Input() case;
  @Output() caseEvent = new EventEmitter<number>();

  onSelect() {
    this.case.isSelected = !this.case.isSelected;
    this.caseEvent.emit(this.case.id);
  }
  constructor() {}

  ngOnInit() {}
}
```

Fig. 29 - Componentă ce generează un element al liste de produse

În componentele ce modelează pașii formularului, este injectat prin intermediul constructorului un serviciu pe baza căruia se păstrează o evidență a componentelor alese până la pasul curent. Totodată, acesta memorează prețul total al produselor alese și progresul utilizatorului în formular.

```
@Injectable()
export class ConfigComputerService {
  public computer: Computer;
  public price: number;
  public progress: number;
  public baseUrl: string;

  constructor(private http: HttpClient) {
    this.computer = new Computer();
    this.price = 0;
    this.progress = 1;
    this.baseUrl = environment.apiUrl;
  }
```

Fig. 30 - Declararea serviciului utilizat în componentele formularului

Mai mult decât atât serviciul implementează și o serie de metode ce se folosesc de serviciul HttpClient pentru a trimite cereri către aplicația server pentru procurarea datelor necesare.

```
      public get cases() {
          const baseUrl = this.baseUrl;
          return {
            getById: (id: string) => {
              const url = `${baseUrl}/api/cases/${id}`;
              return this.http.get<Case>(url);
            },
            getAll: (filter: string) => {
              const url = `${baseUrl}/api/cases`;
              let params = new HttpParams();
              params = params.append('filterObject', filter);
              return this.http.get<Case[]>(url, { params: params });
            }
          };
        }
```

Fig. 31 - Exemplu de metode din serviciul *ConfigComputer*

Capitolul 3: Prezentarea aplicației

În momentul deschiderii aplicației utilizatorul intră în contact cu pagina inițială *(home)*. De aici, acesta are acces rapid către configurator și *tutorial* prin intermediul celor două butoane așezate central. Totodată acesta poate naviga prin meniul așezat în dreapta-sus a paginii către configurator, *tutorial* sau zona de *community*.

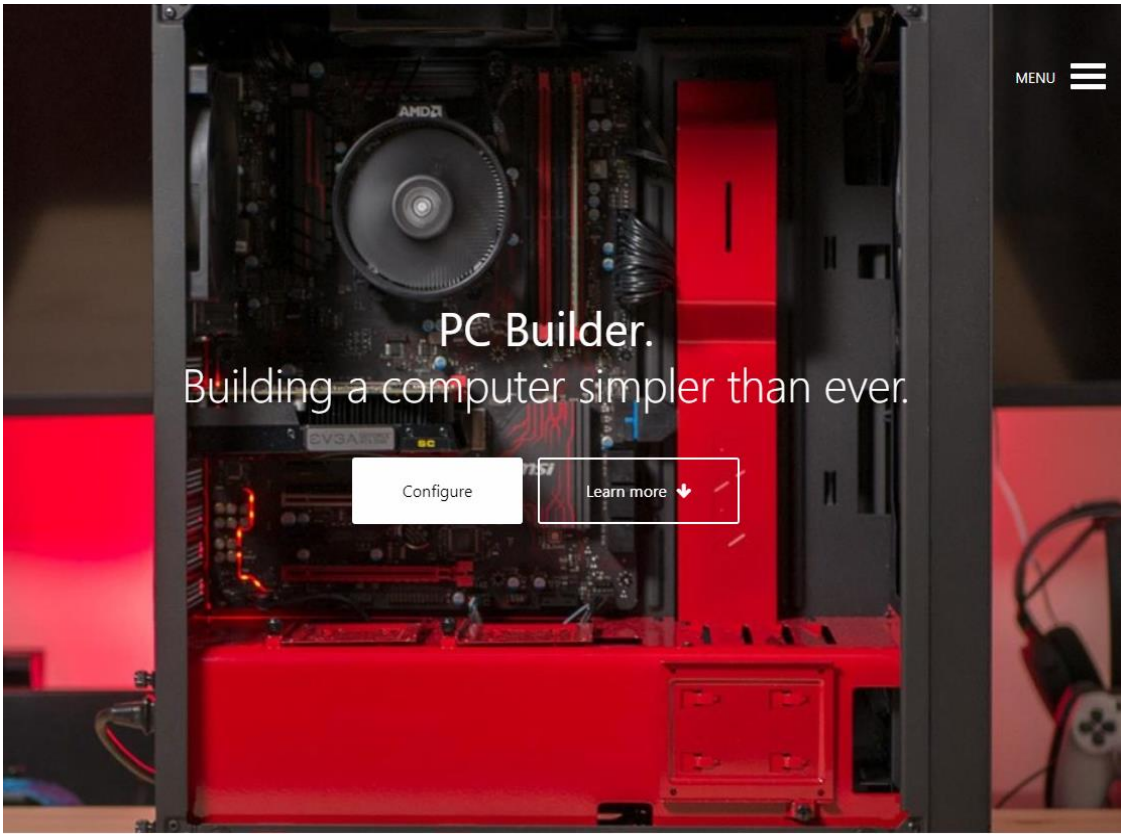


Fig. 32 - Pagina *Home*

Nucleul aplicației este reprezentat de zona configuratorului. O dată accesat utilizatorul interacționează cu un formular de tip *wizard*, alcătuit din nouă pași. Primii opt dintre aceștia sunt contorizați de bara de navigare așezată în partea superioară, astfel că utilizatorul este informat cu privire la pasul curent și progresul realizat.

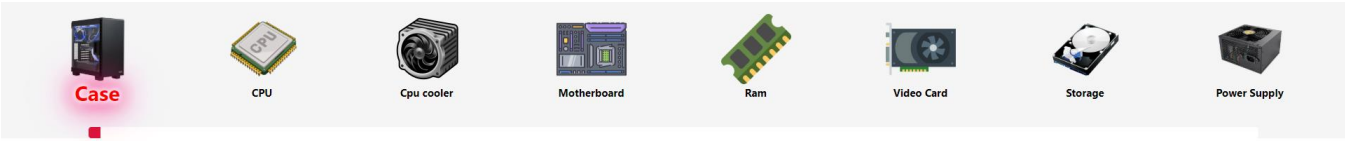


Fig. 33 - Bara de navigare a formularului de tip *wizard*

În fiecare pas al formularului, utilizatorul poate alege un singur produs din lista de produse afișată, selecția realizându-se prin apăsarea butonului “select” a fiecărui card generat. În momentul selecției, utilizatorul este informat de alegerea făcută prin conturarea card-ului cu culoarea roșie și poate trece la următorul pas al formularului.

Un card modelează un produs de un anumit tip. Acesta este format dintr-o imagine de dimensiuni mari, ce îl ajută pe utilizatorul în identificarea mai ușoară a produsului, titlul produsului, prețul acestuia și principalul set de specificații, ce se poate vizualiza în momentul în care utilizatorul trece cu mouse-ul peste imaginea produsului.

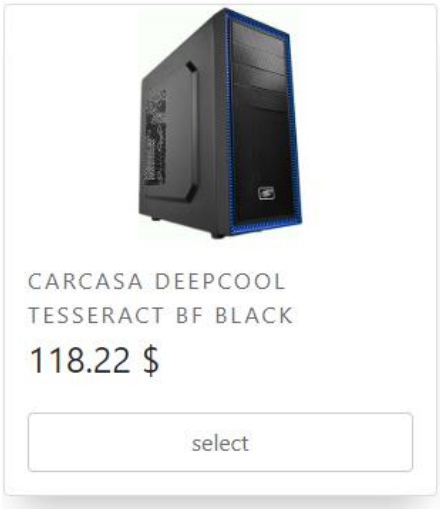


Fig. 34 - Exemplu de card al unui produs

Totodată, pe ecran este afișat tot timpul prețul total al configurării la momentul curent și butoanele de navigare ale formularului.

Fig. 35 - Butoanele de navigare și afișarea prețului total

Dacă utilizatorul nu face o alegere la un anumit pas acesta nu poate înainta către un nou pas al formularului, excepție făcând doar pasul de alegere al coolerului deoarece anumite procesoare vin la pachet cu unul de fabrică, iar achiziționarea unui cooler nu mai este necesară.

În orice pas al formularului utilizatorul are posibilitatea să se răzgândească cu privire la produsele alese pentru configurare, astfel că aplicația îi permite să se întoarcă oricât de mulți pași înapoi și să aleagă alte produse. Chiar dacă utilizatorul navighează prin formular, produsele selectate sunt reținute în memorie, astfel că acesta nu este nevoie să reselecteze un produs la un anumit pas.

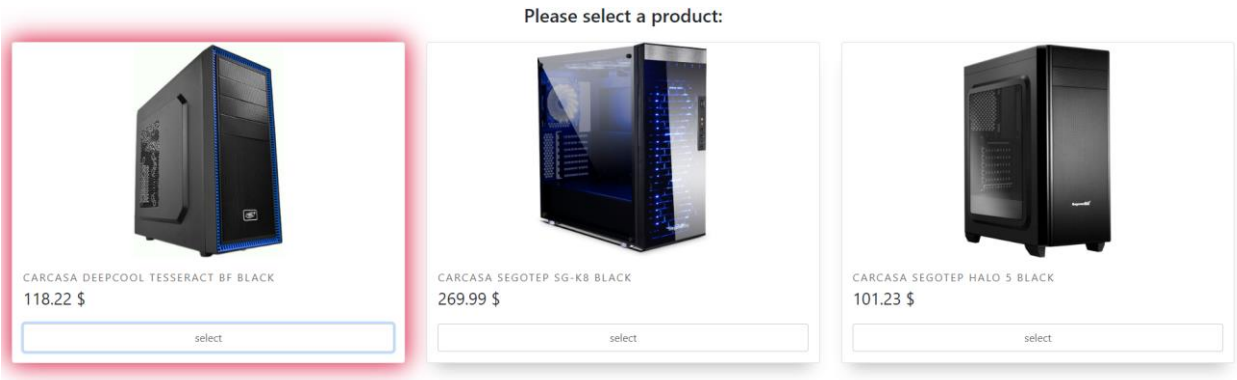


Fig. 36 - Exemplu de listă de carduri cu produse

În ultimul pas al formularului utilizatorul observă lista componentelor alese pentru configurare. Acesta are posibilitatea să navigheze către pagina produsului de pe site-ul magazinului ce îl comercializează și totodată acesta poate să distribuie lista componentelor în pagina de *community*, pentru a putea fi vizualizată și de alți utilizatori ai aplicației.

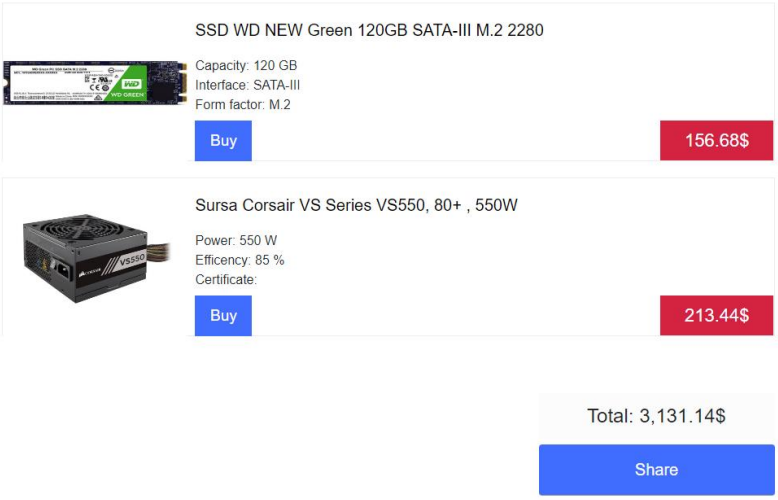


Fig. 37 – Secțiune din pagina *result*

O altă parte importantă a aplicației o reprezintă și zona de *tutorial*. Pe această pagină, utilizatorul observă o imagine de profil a unui computer, cu care acesta poate interacționa în mod dinamic, prin intermediul unor puncte așezate în dreptul fiecărei componente.

Fig. 37 - Pagina de *tutorial*

În momentul în care utilizatorul dă click pe unul din puncte, i se va deschide o fereastră modală în cadrul căreia îi sunt prezentate detalii cu privire la o componentă de un anumit tip.

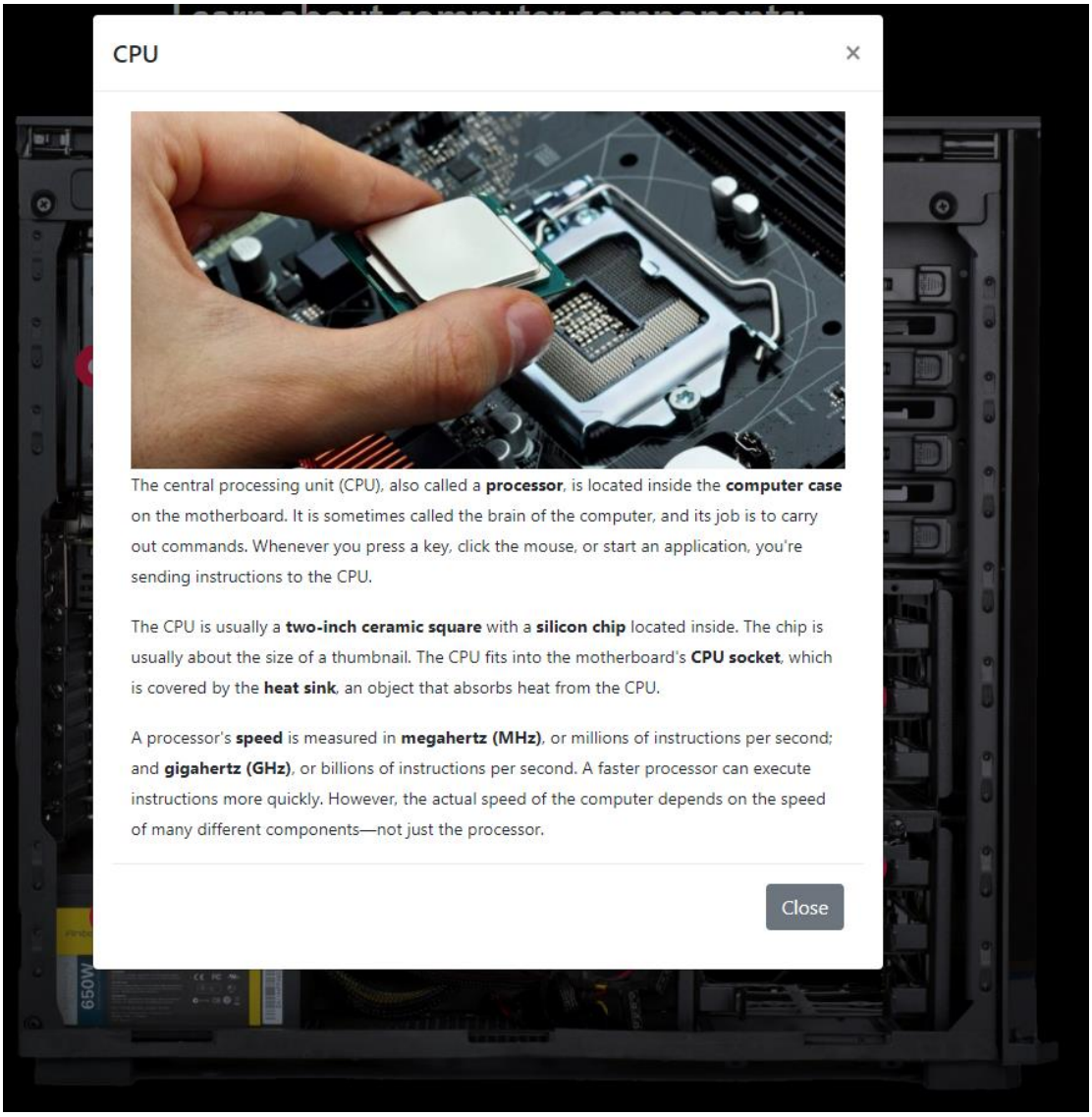


Fig. 38 - Exemplu de fereastră modală din pagina de *tutorial*

Ultima parte a aplicației constă în pagina *community* în cadrul căreia utilizatorul observă diferite configurații realizate de alți utilizatori pe care le poate folosi drept puncte de referință pentru propriile decizii.

Concluzii

Aplicația PC Builder, prezentată în această lucrare a fost gândită cu scopul de a diminua caracterul descurajant pe care îl are configurarea unui computer în mediul online. Aceasta vine în ajutorul utilizatorilor cu mai puține cunoștințe tehnice, întrucât în momentul configurării aplicația le sugerează acestora doar produsele compatibile între ele. Mai mult decât atât, aplicația permite și companiilor ce se ocupă cu vânzarea online de componente pentru computere să își dezvolte activitatea prin punerea în valoare a produselor comercializate, pe baza configuratorului.

Prin implementarea aplicației mi-am propus și atingerea unor scopuri de ordin tehnic:

- Fixarea cunoștințelor dobândite în facultate
- Utilizarea de tehnologii actuale ce sunt la mare căutare pe piața IT
- Dezvoltarea pe plan tehnic, profesional

Proiectul ar putea fi îmbunătățit prin extinderea posibilităților de configurare. Aceasta ar putea permite generarea de configurări mai puțin întâlnite (exemple: cu procesoare multiple, plăci video multiple, sisteme complexe de răcire, etc.). De asemenea aplicația ar putea dezvolta și interacțiunea cu magazinele online prin realizarea unui modul mai complex de funcționalități ce ar permite buna gestiunea a produselor încărcate în aplicație.

În concluzie, consider că aplicația și-a atins pentru moment scopul pentru care a fost dezvoltată. Mai mult decât atât, consider că modalitatea de dezvoltare a acesteia permite adăugarea de noi funcționalități într-o manieră destul de simplă.

Bibliografie

1. Computer. - https://www.explainthatstuff.com/historyofcomputers.html.

2. Households With a Computer and Internet Use: 1984 to 2012. - https://www.census.gov/data/tables/2012/demo/computer-internet/computer-use-2012.html.

3. Asp .NET Core. - https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.1.

4. SQL Server. - https://www.sqlshack.com/sql-server-table-structure-overview/.

5. Entity Framework Core. - https://docs.microsoft.com/en-us/ef/core/.

6. LINQ. - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/.

7. Bootstrap. - https://getbootstrap.com/.

8. Angular 5. - https://angular.io/docs.

9. Dependency Injection. - https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.1&viewFallbackFrom=aspnetcore-2.1..

10. TypeScript/JavaScript. - https://medium.com/codingthesmartway-com-blog/the-2018-roadmap-to-fullstack-web-development-8884ff02557a.

11. What is N-Tier architecture? - https://stackoverflow.com/questions/312187/what-is-n-tier-architecture.

12. Styles - http://freefrontend.com/.