Subsemnatul Loghin L. Alexandru declar pe propria raspundere ca acest cod nu a fost copiat din Internet sau din alte surse.

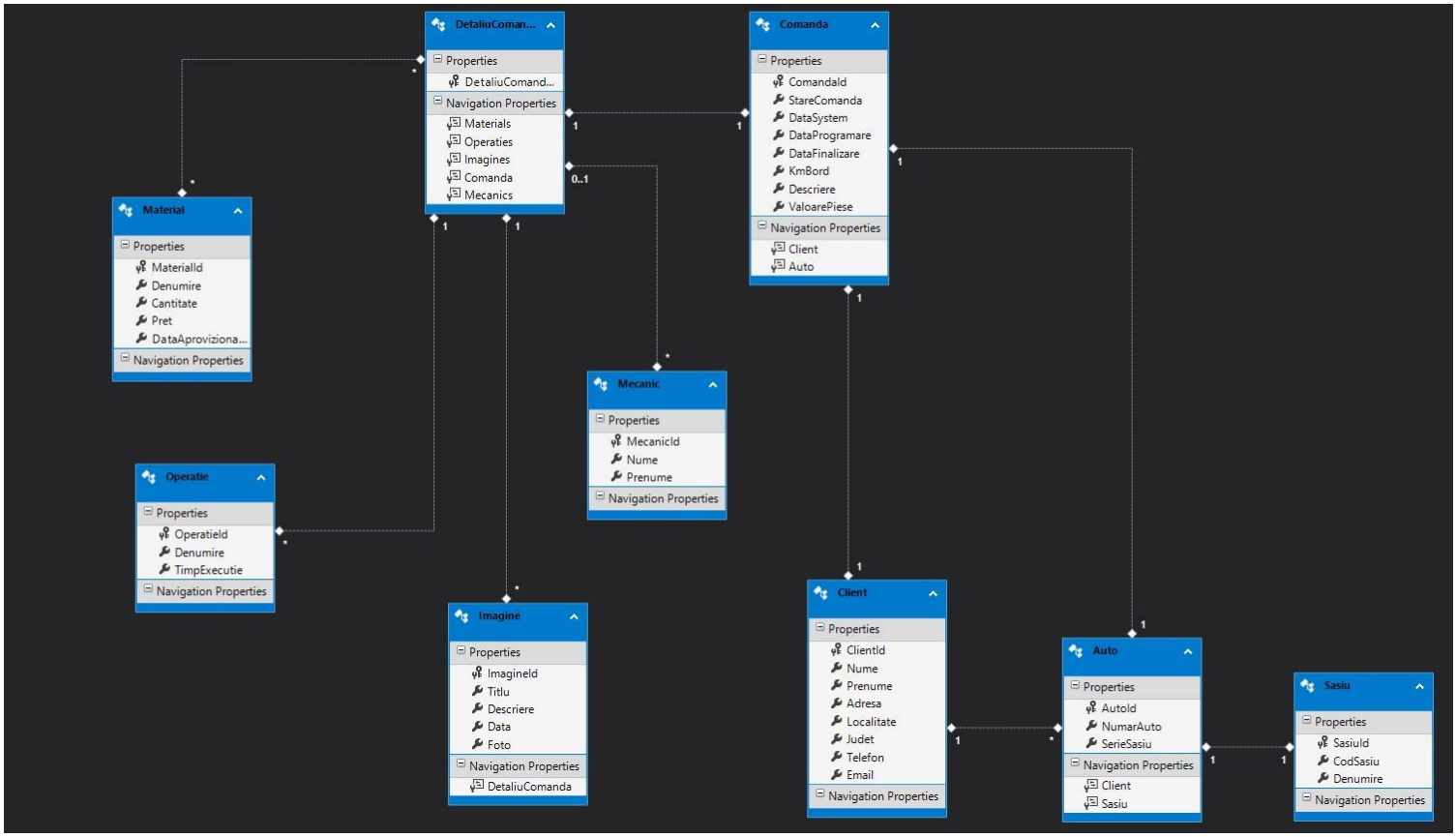
CarService - API

Proiectul nr 1

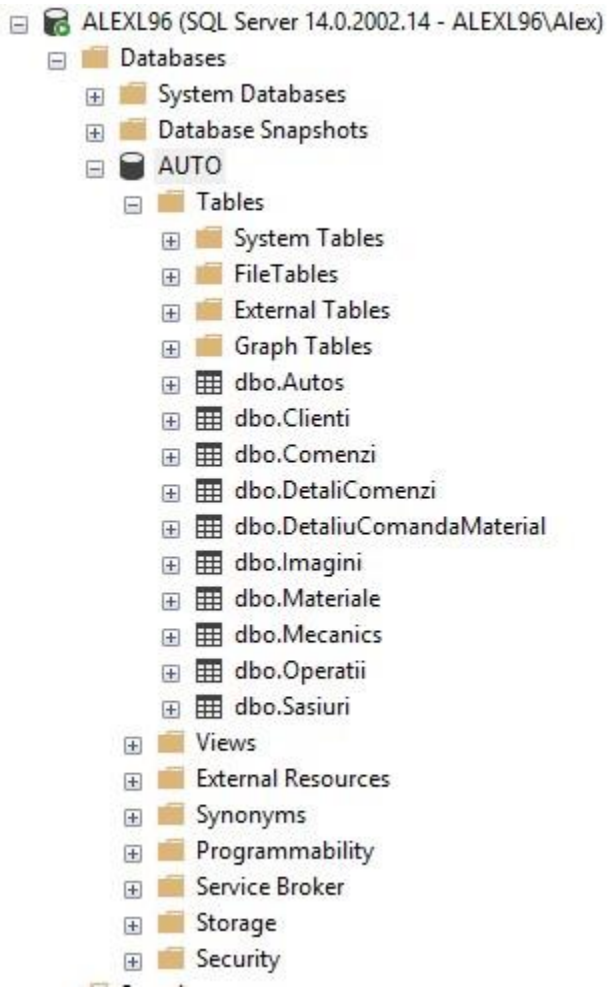
Application Properties	Assembly name:	CarService
	Default namespace:	CarService
	Target framework:	.NET Framework 4.6.1
	Output type:	Class Library

Am realizat modelul entitatilor urmand cu exactitate pasii din documentatia [EF Model Designer First]

La finalul etapei 1 modelul realizat arata ca mai jos sau [aici]



Utilizand scriptul general din modele, la final baza de date arata ca mai jos sau aici



Structura proiectului:

Pentru logica de design a proiectului am folosit Repository Pattern si pentru fiecare model am separat operatiile de citire si de scriere in doua mari categori: Read / Write.

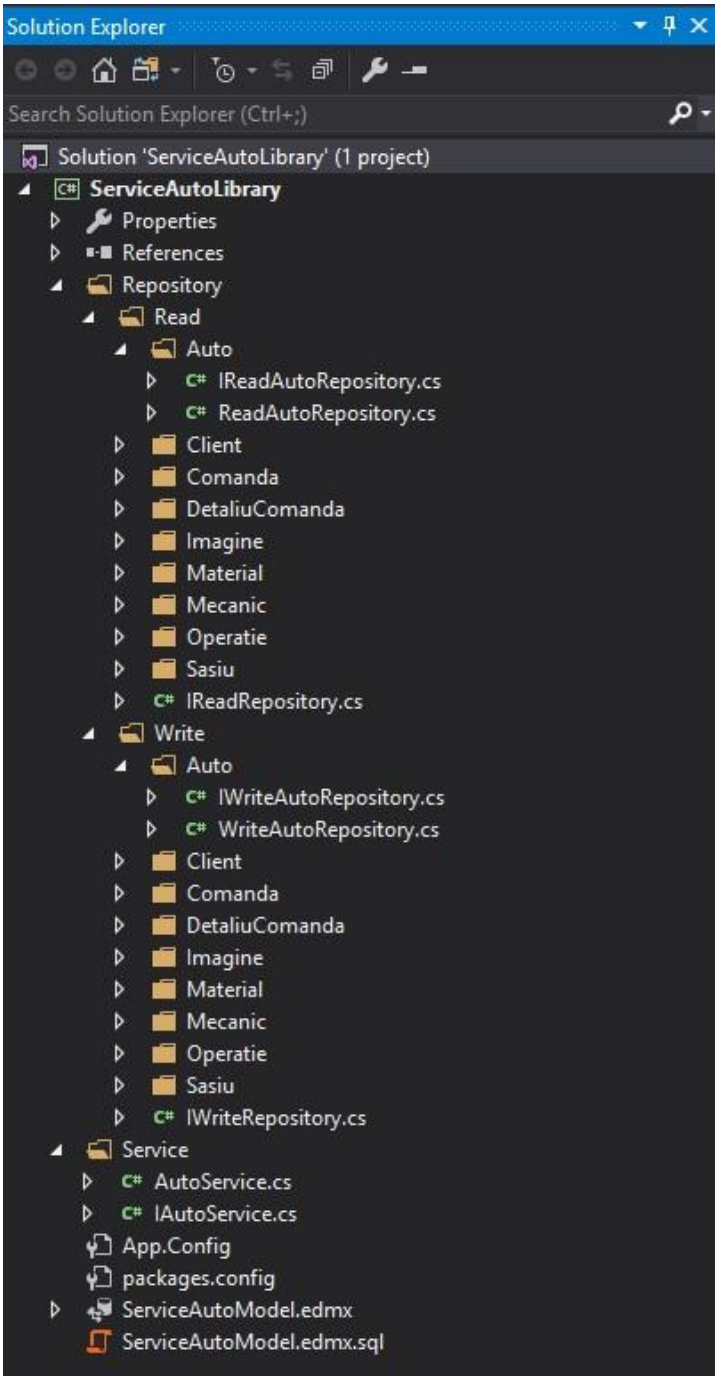
Pentru fiecare entitate avem doua interfete (IRead/IWrite) Repository si doua clase ce implementeaza interfetele respective si contin logica fiecarei metode.

La final, am setat modificatorul de acces pe internal si am adaugat un Service, cu modificator public, ce va contine cate o instanta din fiecare repo.

Iar mai departe, in etape urmatoare, accesul la api si baza de date se va face doar prin acel service.

Interfata IAutoService va cuprinde toate usecase-urile pe care aplicatia le va putea avea, adica pe care utilizatorul le va putea folosi.

La finalul etapei 1 solutia realizata arata ca mai jos sau aici



Modificari anterioare proiectul 1:

Pentru a usura munca de development pe partea de API a fost adaugata structuri un folder “Common” ce momentan contine o Exceptie custom denumita “DbEntityCustomException”. Aceasta preia orice mesaj de eroare aruncat de EF si il constrange intr-un mesaj tip String ce specifica in mod clar si usor erorile de tabele si key la nivel de DB.

Am mai implementat un mod de logare a activatilor pe intregul API, logari ce folosesc exceptia custom de mai sus realizat cu scopul de a usura munca de debug atunci cand apar erori intre api si baza de date. Toate aceste logari sunt salvate intr-un fisier text denumit “CarServiceLog.txt”. Aceste noi implementari se pot gasi in CarService folder-ul “Common”

Au fost adaugate pentru o mare parte din entitati metode de update, delete si metode pentru logica de bussiness (ca de exemplu, returnarea mecanicilor disponibili).

La nivel de baze de date, am optat ca un client sa aiba unique campurile de telefon si email (pentru o cautare mai eficienta in baza de date) si de asemenea cum fiecare autoturism are numarul auto specific, am declarat aceast field tot unique.

De asemenea am considerat ca un mecanic poate lucra la o singura comanda in acelasi timp.

CarService.WF - Windows Form App

Proiectul nr 2

Application Properties	Assembly name:	CarService.WF
	Default namespace:	CarService.WF
	Target framework:	.NET Framework 4.6.1
	Output type:	Windows Application

Am realizat modelul entitatilor urmand cu exactitate pasii din documentatia [EF Model Designer First]

Structura proiectului:

Am incercat sa construiesc pe cat posibil aplicatia dupa explicatiile din usecase-ul cerintei de proiect prin care un utilizator ar putea sa o foloseasca.

Obiectivul interfetei a fost ca accesul utilizatorului sa fie “dictat” de catre interfata, astfel multe dintre etapele de utilizare a aplicatie sunt facute dupa un anumit set de etape/reguli.

Modul de comunicare cu API-ul porneste dintr-un form parinte ce apeleaza serviciul, odata, in mod static si il trimite mai departe celorlalte interfete ca referinta, exemplu in imaginea de mai jos:

```
namespace CarService.WF.Forms
{
    3 references | Alexandru Loghin, 6 days ago | 1 author, 1 change
    public partial class MainForm : Form
    {
        private static bool _searchClientFieldEvent = true;
        private static IAutoService _serviceApi = new AutoService();

        1 reference | Alexandru Loghin, 6 days ago | 1 author, 1 change
        public MainForm()
        {
            InitializeComponent();
        }

        1 reference | Alexandru Loghin, 6 days ago | 1 author, 1 change
        private void AddClientButton_Click(object sender, EventArgs e)
        {
            var newClientForm = new NewClientForm(ref _serviceApi);
            Hide();
            newClientForm.ShowDialog();
            if (this.IsDisposed) return;
            Show();
        }
```

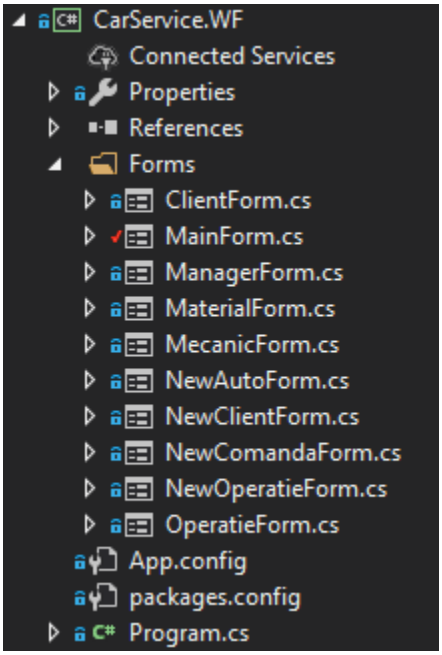
Aplicatia contine un Form principal denumit “MainForm” de unde exista 3 mari ramuri explicate pe rand mai jos:

- Cautare client existent
- Adaugare client
- Meniu Manager

1. Cautare client existent:
   (a) Un client poate fi identificat din interfata doar prin numarul de telefon, adresa de email sau de orice numaru auto al unei masini existente in baza de date
   (b) In cazul in care una din aceste 3 conditii au fost indeplinite, se poate selecta o masina ce a mai fost in service sau se poate adauga una nou
   (c) Dupa selectare sau adaugare masina, managerul poate adauga detaliile comenzii (imagini, descriere, KmBoard, etc) si poate salva comanda in modul de “asteptare” sau poate trece direct la adaugarea operatiilor si a mecanicilor.
   (d) Daca se opteaza pentru adaugarea imediata a operatiilor, managerul pe langa operatii poate atribuii direct niste mecanici disponibili (care nu mai au alta comanda atribuita in acel moment) pe comanda respectiva.

2. Adaugare client:
   Acest usecase este cel mai complet, parcurcand recursiv fiecare etapa prin care un client nou ajunge sa plaseze o comanda.
   (a) Managerul trebuie sa introduca datele clientului conform structurii stabilite
   (b) Apoi poate introduce una sau mai multe masini, in final selecand doar una
   (c) Pentru masina selectata se cere sa fie completate detaliile de comanda iar in continuare flow-ul este asemanator cu cel de la punctul 1

3. Meniu Manager:
   (a) Odata intrat in acest form putem spune ca suntem in usercase-ul de administrator principal al aplicatiei. De aici teoretic ar trebuii sa avem cate un form complex pentru fiecare entitate din aplicatia noastra. Momentan au fost introduse doar categoriile de Operatii, Mecanici si Materiale.
   (b) Fiecare view din cele 3 form-uri mentionate anterior aduc posibilitatea de adaugare, modificare, stergere ori din baza de date ori din comanda selectata.



La finalul etapei 2 structura proiectului de windows form arata ca in imaginea din stanga.

CarService.WCF & CarService.WPF

Proiectul nr 3

Application Properties	Assembly name:	CarService.WCF
	Default namespace:	CarService.WCF
	Target framework:	.NET Framework 4.6.1
	Output type:	Console Application

Application Properties	Assembly name:	CarService.WPF
	Default namespace:	CarService.WPF
	Target framework:	.NET Framework 4.7.2
	Output type:	Windows Application

Am realizat modelul WCF urmand cu exactitate pasii din exemplu cu PostComment realizat de dumneavoastra: Laborator_WCF_2019.pdf

Structura proiectului - WCF:

Claselor generate de EF din proiectul 1 le-am adaugat adnotarea de (`DataContract` si `DataMember`) iar serviciului deja existent i-am implementat namespace-ul ”`ServiceModel`” cu tag-urile (`ServiceContract` si `OperationContract`).

Claselor Hostul WCF a fost construit in oglinda, urmand exemplele de laborator si adaugand modificarile necesare pentru a putea realiza comunicarea cu serviciul API. Adresa pentru care am optat ca host-ul asa ruleze este “`http://localhost:8080/service`“.

De asemenea am fost nevoit sa maresc si capacitatea pe care host-ul o accepta la un mesaj trimit de catre client (`maxReceivedMessageSize="2147483647"`).

Am mai observant ca utilizand tool-ul “svcutil” pentru generarea fisierului proxy.cs, proprietarile ICollection erau convertite in vectori, dupa cateva cercetari comanda de generare a ajuns sa arate asa:

```
svcutil http://localhost:8080/service -config:app.config -out:"C:\proxy.cs"
/language:C# /ct:System.Collections.Generic.ICollection`1
```

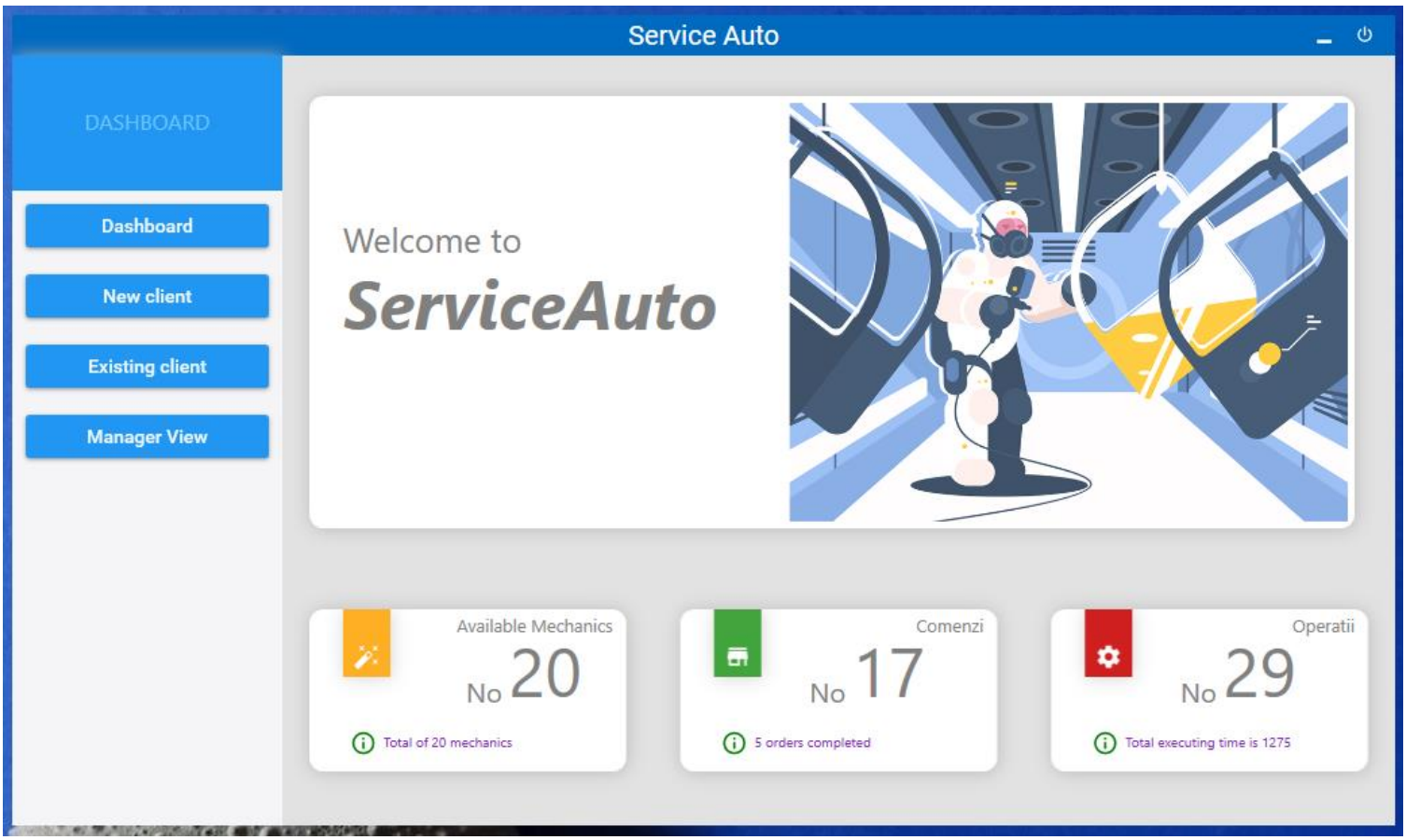
Structura proiectului - WPF:

Aplicatia a imprumutat structura proiectului 2 de windows form, cu mici ajustari si imbunatatiri.

Core-ul aplicatie este MainWindow, aici porneste si ramane aplicatia. MainWindow este un Panel, iar restul view-urilor sunt de tipul “Page” si sunt incarcare in Panel printr-un <Frame/> cu navigatia ascunsa.

Pentru un aspect cat se poate de modern, am optat sa folosesc un open-source nugget denumit MaterialDesign si am customizat pana si TopBar-ul aplicatie.

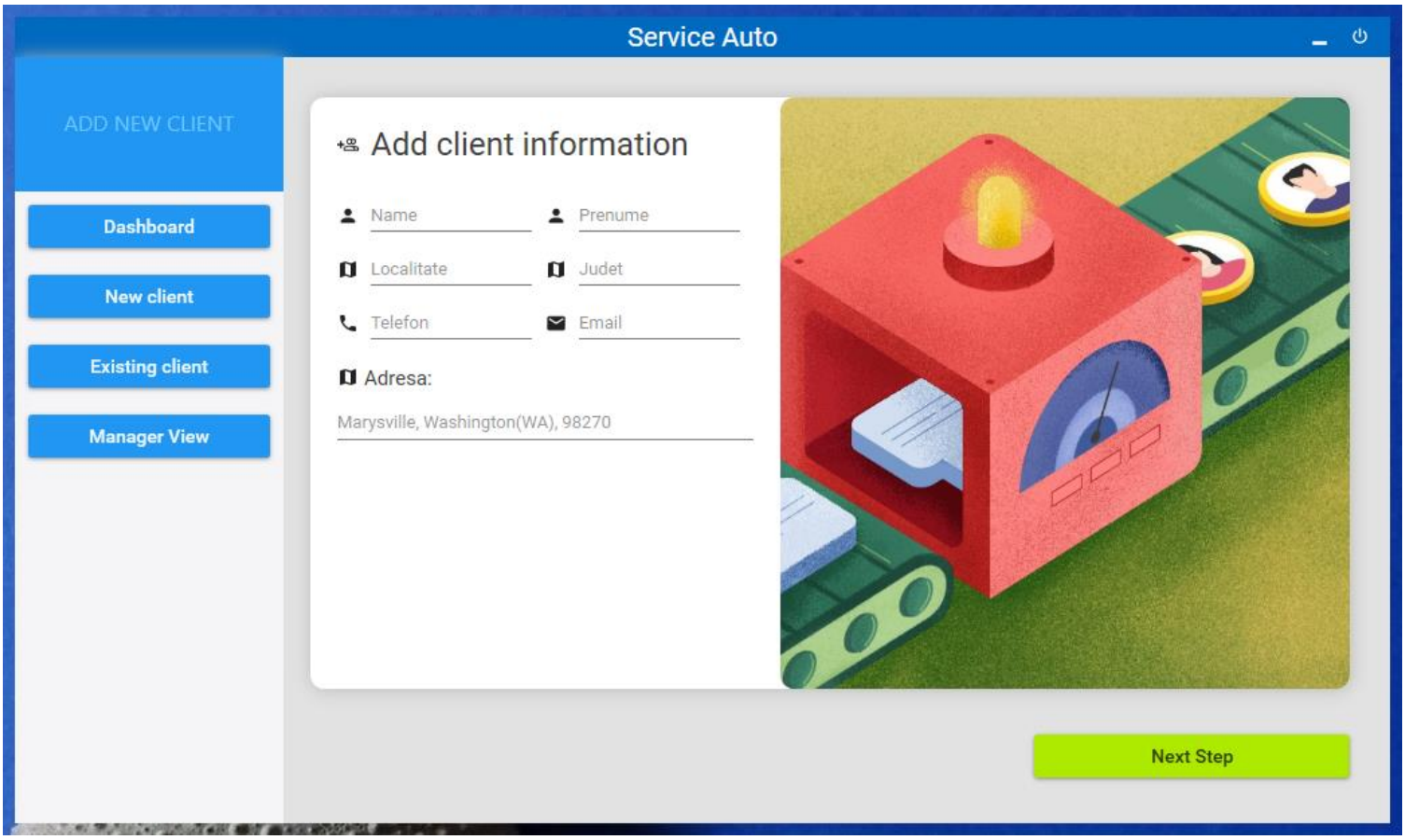
Aplicatia cand porneste arata asa:



O problema ce am intampinat-o incercand sa aduc acest aspect modern in aplicatie, a fost consumul de memorie, ce ajungea la apropape 4GB RAM doar dintr-o singura pagina ce rula un mic gif intr-un loop. Dupa o lunga perioada de frustrari am conceput o clasa statica denumita `CommonItem`.

Aceasta clasa pe langa alte proprietati, contine o lista statica cu fiecare pagina deschisa din aplicatia noastra. Si desi tag-ul de <Frame/> are un raport bun de eliminare a paginilor ce nu mai sunt

focusate, am fost nevoit sa specific cand doresc si cand nu doresc ca masina virtuala sa-mi stearga pagina din memorie.
Un exemplu ar fi pe flow-ul de “New client”, aici trecem prin mai multe pagini si vrem sa putem mentine fiecare pagina activa, in cazul in care utilizatorul doreste sa modifice niste date anterioare. Astfel putem controla cat sa mentinem fiecare pagina in viata sis a ne asiguram ca avem doar o singura instant deschisa.



Pe langa aspectul modern si putin schimbat fata de windows form-ul din proiectul 2, logica si pasii de executie vor fi aceeasi, cu mici adjustari de compatibilitate.

Din pacate, din lipsa de timp, aplicatia nu are validari decat un mic exemplu la field-ul “Telefon” (imaginea de mai sus) si singurul flow functional este cel de New client si pagina de home/dashboard.

La finalul etapei 3 solutia WPF realizata arata ca mai jos: