Технически Университет – София

Факултет „Компютърни Системи и Технологии”

Дипломна работа

Образователна степен Магистър по Компютърно и Софтуерно Инженерство

Тема:

Електронен дневник за поддръжка на автомобили

Ръководител катедра “Компютърни системи” : .....

/ проф. Д-р инж. Милена Лазарова/

Научен ръководител :.....

/доц. Д-р инж. Георги Попов /

Дипломант: .....

/ Тодор Еников

Фак.№ 121316042 /

Съдържание

1.Увод

Настоящият проект – „Електронен дневник за поддръжка на автомобили“ (CarsSystem), представлява WEB приложение, което служи за запазване на информация за клиенти и техните автомобили, в даден сервиз.

- Информацията, която се пази за клиентите е:

  - Име
  - Презиме
  - Фамилия
  - Единен Граждански Номер (ЕГН)
  - Номер на Л.К.
  - Дата на издичане на Л.К.
  - Град
  - Тел. Номер
  - E-mail
  - Потребителско име (username)

- Информацията, която се пази за автомобилите на клиентите е :

  - Производител на превозното средство
  - Модел на превозното средство
  - Тип двигател на превозното средство
  - Регистрационен номер на превозното средство
  - VIN (идентификационен) номер на превозното средство
  - Брой гуми на превозното средство
  - Брой врати на превозното средство
  - Тип на превозното средство
  - Година на производство на превозното средство
  - Дата на изтичане на годишен технически преглед на превозното средство
  - Дата на изтичане на винетна такса на превозното средство
  - Дата на изтичане на застраховка на превозното средство

За разработката на проекта е избрана трислойна архитектура. Трите слоя са имплементирани така че да бъдат, независими 1 от друг, за да могат да бъдат лесно заменяеми. Те са:

- 1 слой - Слой за база данни - използва се MS SQL Server
- 2 слой - Сървърен слой - този слой се използва за достъп до базата данни
- 3 слой - Клиентско приложение - изградено на ASP .NET MVC, като се използва MVC шаблона

2. Теоретична част

2.1 Microsoft Visual Studio

Microsoft Visual Studio е мощна интегрирана среда за разработка на софтуерни прило-жения за Windows и за платформата .NET Framework. Използва се за разработка на конзолни и графични потребителски интерфейс приложения, както и Windows Forms или WPF приложения, уеб сайтове, уеб приложения и уеб услуги на всички поддържани платформи от Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework и Microsoft Silverlight.

2.1.1 Архитектура

Visual Studio не поддържа нито един език за програмиране, решение или вътрешен инструмент, вместо това функционалността от плъгини се кодира като VSPackage. Когато се инсталира, функционалността се предлага като услуга. Самото IDE предлага три услуги: SVsSolution, което дава възможност да се изброят проекти и решения; SVsUIShell, който предвижда прозоречна и UI функционалност (включително и табовете, ленти с инструменти и Windows-ските инструмент) и SVsShell, който се занимава с регистрация на VSPackages. В допълнение, IDE е отговорен за координирането и осигурява комуникация между услугите. Всички редактори, графиките, видовете проекти и други инструменти са внедрени като VSPackages.

2.1.2 Функции

- Текстов редактор - Текстовият редактор може да бъде използван за редактиране на текст или програмен код. Той поддържа функцията IntelliSense, която включва не само така нареченото оцветяване на синтаксиса, но и автоматично довършване на кода. Текстовият редактор може да се ползва за всички поддържани от Visual Studio програмни езици плюс XML, CSS и JavaScript, когато се създават уеб сайтове и уеб базирани приложения.
- Дебъгер - Microsoft Visual Studio предоставя дебъгер за проследяване поетапното изпълнението на разработвания софтуер, с цел откриване и отстраняване на грешки.
- Дизайнер - Microsoft Visual Studio включва графични дизайнери, подпомагащи разработката на апликации.
- Разширения - Visual Studio позволява на разработчиците да създават разширения за Visual Studio, с цел увеличаването на неговите възмжности. Тези разширения идват под формата на макроси, add-ins и пакети.

2.2 Microsoft SQL Server

Microsoft SQL Server е сървърна система за управление на бази от данни (и по-точно, на релационни бази от данни) на компанията Microsoft. Microsoft SQL Server е предназначена за управление на големи сървърно базирани БД за разлика от MS Access която е desktop базирана и не е предназначена за управление на големи корпоративни БД.

2.2.1 Система за управление на бази от данни (СУБД)

Система за управление на бази от данни или система за управление на бази данни, СУБД (Database management system, DBMS) е набор от компютърни програми, контролиращи изграждането, поддръжката и използването на бази от данни. Примери за такива системи са Microsoft SQL Server, MySQL, Access, Oracle, Paradox, dBase, FoxPro, Cliper, Sybase, Informix. Някои от тях са потребителски ориентирани програмни среди, а други са по-скоро езици за създаване и описване на бази от данни. Програмата Microsoft Access е типичен пример на потребителски ориентирана програма за управление на бази от данни. Макар че в Microsoft Access е възможно и програмиране с езика SQL, малки бази от данни могат да бъдат създавани и без програмиране.

2.2.2 SQL – език за структурирани запитвания

SQL (***S**tructured **Q**uery **L**anguage*) е популярен език за програмиране, предназначен за създаване, видоизменяне, извличане и обработване на данни от релационни системи за управление на бази данни. Стандартизиран е от ANSI / ISO.

SQL е създаден със специфична, ограничена цел - запитвания към данни в релационна база от данни. Като такъв той е множествено-базиран, декларативен език за програмиране, а не процедурен език като C или BASIC които, бидейки неспециализирани, са направени да разрешават по-голям обхват от проблеми. Разширения на езика като PL/SQL намаляват разликата добавяйки процедурни елементи, и контролни структури. Друг подход е да се позволи вграждане на кода. Например Oracle и други включват Java в базата данни, докато PostgreSQL позволява функциите да бъдат написани на различни езици, като Perl, Tcl, и C.

2.3 Език за програмиране C#

C# (C Sharp, произнася се Си Шарп) е обектно-ориентиран език за програмиране, разработен от Microsoft, като част от софтуерната платформа .NET. Стремежът още при създаването на C# езика е бил да се създаде един прост, модерен, обектно-ориентиран език с общо предназначение. Основа за C# са C++, Java и донякъде езици като Delphi, VB.NET и C. Той е проектиран да балансира мощност (C++) с възможност за бързо разработване (Visual Basic и Java). Те представляват съвкупност от дефиниции на класове, които съдържат в себе си методи, а в методите е разположена програмната логика – инструкциите, които компютърът изпълнява. Програмите на C# представляват един или няколко файла с разширение .cs., в които се съдържат дефиниции на класове и други типове. Тези файлове се компилират от компилатора на C# (csc) до изпълним код и в резултат се получават асемблита – файлове със същото име, но с различно разширение (.exe или .dll).

2.3.1 Език C# и платформата .NET

Първата версия на C# е разработена от Microsoft в периода 1999 – 2002 г. и е пусната официално в употреба през 2002 г., като част от .NET платформата, която има за цел да улесни съществено разработката на софтуер за Windows среда чрез качествено нов подход към програмирането, базиран на концепциите за „виртуална машина“ и „управляван код“. По това време езикът и платформата Java, изградени върху същите концепции, се радват на огромен успех във всички сфери на разработката на софтуер и разработката на C# и .NET е естественият отговор на Microsoft срещу успехите на Java технологията. Едно от най-големите предимства на .NET Framework е вграденото автоматично управление на паметта. Това сериозно повишава производителността на програмистите и увеличава качеството на програмите, писани на C#.

2.3.2 Управление на паметта

Едно от най-големите предимства на .NET Framework е вграденото автоматично управление на паметта. То освобождава програмистите от сложната задача сами да заделят памет за обектите и да търсят подходящия момент за нейното освобождаване. Това сериозно повишава производителността на програмистите и увеличава качеството на програмите, писани на C#.

За управлението на паметта в .NET Framework се грижи специален компонент от CLR, наречен „система за почистване на паметта“ (garbage collector). Основната задача на системата Garbage collector-а е да следи кога заделената памет за променливи и обекти вече не се използва, да я освобождава и да я прави достъпна за последващи заделяния на нови обекти.

2.3.3 Дизайнерски цели

- C# е създаден като прост, модерен с общо предназначение и обектно-ориентиран език за програмиране.
- Езикът е предназначен за използване в развиващите се софтуерни компоненти, той е подходящ и за разполагане в разпределена среда.
- На езика C# и върху .NET платформата може да бъде разработван разнообразен софтуер, като офис приложения, уеб приложения, уеб сайтове, настолни приложения, мултимедийни Интернет приложения, приложения за мобилни телефони, различни видове игри и много други.

2.3.4 Версии до сега на езика C#

Версия	Дата	.NET Framework	Visual Studio
C# 1.0	Януари 2002	.NET Framework 1.0	Visual Studio .NET 2002
C# 1.2	Април 2003	.NET Framework 1.1	Visual Studio .NET 2003
C# 2.0	Септември 2005	.NET Framework 2.0	Visual Studio 2005
C# 3.0	Август 2007	.NET Framework 3.5	Visual Studio 2008 Visual Studio 2010
C# 4.0	Април 2010	.NET Framework 4	Visual Studio 2010
C# 5.0	Август 2012	.NET Framework 4.5	Visual Studio 2012 Visual Studio 2013
C# 6.0	Юли 2015	.NET Framework 4.6	Visual Studio 2015
C# 7.0	Март 2017	.NET Framework 4.6.2	Visual Studio 2017
C# 7.1	Август 2017	.NET Framework 4.6.2	Visual Studio 2017

Фиг. 2: Версии на езика C#

3. Използвани технологии

3.1 Въведение в технологията ASP .NET MVC

ASP.NET е програмна платформа част от .NET Framework за разработка на уебприложения. Тя предлага съвкупност от класове, които работят съвместно, за да обслужват HTTP заявки. Също като класическото ASP (Active Server Pages), ASP.NET се изпълнява на уеб - сървър и предоставя възможност за разработка на интерактивни, динамични и персонализирани уеб-базирани приложения , както и за разработка и използване на уеб услуги.

Разликите между ASP и ASP.NET са значителни. Основният компонент на ASP.NET е уеб - формата – абстракция на HTML страницата, която Интернет потребителите виждат в браузъра си. Ключова характеристика на ASP.NET е възможността за разделяне на кода описващ дизайна от кода, реализиращ логиката(code-behind), но нивото на абстракция, включва и богат избор от уеб-контроли, подобни на тези в Windows Forms, и намалява нуждата програмиста да работи с чист HTML код.

ASP.NET уеб-формите се изпълняват от страна на сървъра, като в резултат се генерира HTML код, който да се върне като отговор на клиентската заявка. Този код е пригоден за типа на клиентския браузър, което позволява улеснена разработка на уеб - форми, тъй като те практически могат да работят върху всяко устройство, което разполага с Интернет свързаност и уеб-браузър. За генериране на страницата могат да се извършват обработки, изискващи достъп до бази от данни и до ресурсите на самия сървър, генериращи допълнителни уебформи и други.

Платформата използва C#, HTML, CSS, JavaScript и бази данни и е съвременно сред-ство за уеб приложения, което обаче не замества изцяло уеб формите. Платформата включва нови тенденции в разработката на уеб приложения, притежава много добър контрол върху HTML и дава възможност за създаване на всякакви приложения. ASP.NET MVC може да бъде много лесно тествана и допълвана, защото е изградена от отделни модули, изцяло независими един от друг. Чрез платформата се създават цялостни приложения, които се стартират, а не единични скриптове (като при PHP например).

Моделът на изпълнение (execution model) на ASP.NET е следният (фиг.1):

- Клиентският браузър изпраща HTTP заявка на сървъра
- Сървърът идентифицира страницата, за която е предназначена заявката и започва да я изпълнява
- ASP.NET интерпретира кода на страницата
- Ако кодът не е бил компилиран до асембли, ASP.NET извиква компилатора.
- Средата за изпълнение (CLR) зарежда в паметта и изпълнява междинния код;
- Страницата се изпълнява, като в отговор на клиентската заявка се генерира
  HTML код. Сървърът връща този резултат на клиента като HTTP отговор.

Фиг. 1: Модел на изпълнение на ASP.NET

3.2 Модели, Контролери, Действия, Изгледи и Валидация в технологията ASP .NET MVC

3.2.1 MVC модели

Моделът представлява част от приложението, което реализира домейн логиката, също известна като бизнес логика. Домейн логиката обработва данните, които се предават между базата данни и потребителския интерфейс. Например, в една система за инвентаризация, моделът отговаря за това дали елемент от склада е наличен. Моделът може да бъде част от заявлението, което актуализира базата данни когато даден елемент е продаден или доставен в склада. Често моделът съхранява и извлича официална информация в базата данни.

Изглед модела позволява да се оформят няколко изгледа от един или повече модела от данни или източници в един обект. Този модел е оптимизиран за потребление и изпълнение.

MVC (model – view – control) предоставя различни видове и начини за това как да бъде изграден даден модел. Практиката е доказала, че поради липсата на много информация, практики и препоръки, е възможно да се получат разминавания за това как да бъде направен модел. Всъщност, истината е, че няма еднотипно решение, което да задоволява всички възникнали проблеми. Тук ще разгледаме някои от основните модели, които се използват, с описание на конкретен пример.

Модел 1 – Обектен домейн модел използван директно като модел-изглед(view model)

Модел който изглежда по следния начин:

```
public class ShowAllCarsViewModel
{
        public GUID Id { get; set; }
        public string Manufacturer { get; set; }
        public string Model { get; set; }
        public string RegistrationNumber { get; set; }
}
```

Когато този код премине в изглед, той ни позволява да пишем HTML помощници в стила на нашия избор:

```
<%=Html.TextBox("Manufacturer") %>
<%=Html.TextBoxFor(m => m.Manufacturer) %>
```

Разбира се с този модел по подразбиране може да се върнем обратно при кон-тролера:

```
public ActionResult Index(ShowAllCarsViewModel showAllCarsViewModel)
```

Въпреки, че този модел е доста опростен и работи изчистено и елегантно, той се „чупи“ много често дори и при по-опростени изгледи. Свързването към домейн модела в този случай не винаги е достатъчно за пълно представяна на изгледа.

Модел 2 – Собствен модел, съдържащ обектния домейн модел

Ако разгледаме пак ShowAllCarsViewModel от горния пример, в реалния живот е много по-реалистично изгледът за даден обект да се нуждае от повече от една характеристика, за да бъде представен правилно. Примерно, характеристиките от горния пример Manufacturer и Model ще бъдат извиквани от падащи менюта. Ето защо, този модел е предназначен да представи изгледа като съдържател на всички характеристики, необходими за правилното представяне:

```
public class ShowAllCarsViewModel
{
        public GUID { get; set; }
        public SelectList ManufacturerList { get; set; }
        public SelectList ModelList { get; set; }
        public string RegistrationNumber { get; set; }
}
```

В този случай, контролерът има отговорност за това моделът ShowAllCarsViewModel да е правилно попълнен с данни от хранилищата (Пример: получаване на „Кола“ (Car) от база данни, извличане на колекция

Manufacturer/Model от база данни). Нашите HTML помощници леко се променят, защото сега се отнасят за ManufacturerList, а не само за Manufacturer:

Когато формата се публикува, все още е необходимо да има Index() метод:

```
public ActionResult Index([Bind(Prefix = "ShowAllCarsViewModel")]
ShowAllCarsViewModel showAllCarsViewModel)
```

Модел 3 – Собствен модел, съдържащ обичайния модел – изглед

Когато изгледът стане по-сложен, обикновено е трудно да се запази обектният модел на домейна в синхрон с работата на изгледа. Ако разгледаме пак горепосочения пример, нека да предположим, че се изисква да се предостави на потребителя поле, в което той да може да добави кола, ако желае. Когато формата се публикува, контролерът трябва да прецени според стойностите, кой изглед да бъде показан следващ. Последното нещо, което трябва да се направи е да се добави това свойство на нашия домейн – модел, тъй като това е строго изискване.

Нека информацията за мотоциклета да е MotorcycleData:

```
public class CarData
{
        public string Manufacturer { get; set; }
        public string Model { get; set; }
        public int Year { get; set; }
        public string VIN { get; set; }
        public bool AddAdditionalCycle { get; set; }
}
```

Този модел е доста трудоемък и изисква слой за превеждане, за да се очертаят връзките назад и напред между ShowAllCarsViewModel и CarData, но в случая усилията си струват дори за по-сложни изгледи. Този модел със сериозно застъпен и препоръчан от авторите на книгата MVC in Action. Идеите са разширени и в публикациите на Jimmy Bogard (един от съавторите на книгата) – How we do MVC – View Models.

3.2.2 Контролери и действия

Контролерите са класове, които се създават в MVC приложението. Намират се в папка Controllers. Всеки един клас, който е от този тип, трябва да има име завършващо с наставка „Controller“. Контролерите обработват постъпващите заявки, въведени от потребителя и изпълняват подходящата логика за изпълнение на приложението.

Класът контролер е отговорен за следните етапи на обработка:

- Намиране и извикване на най-подходящия метод за действие (action method) и валидиране, че може да бъде извикан.
- Взимането на стойности, които да се използват като аргументи в метода за действие.
- Отстраняване на всички грешки, които могат да възникнат по време на изпълнението метода за действие.
- Осигуряване на клас WebFormViewEngine по подразбиране за отваряне на страници с изглет от тип ASP.NET.

Контролера е клас който се наследява от базовия клас System.Web.Mvc.Controller. Всеки публичен метод в контролера е показан като „controller action“. Ако искаме даден метод да не бъде извикан, трябва да сложим „NonAction“ атрибут върху неговото име. По подразбиране „Index()“ действието е извикано за контролера, когато друго такова не е изрично упоменато.

3.2.2.1 Методи за действие

В ASP.NET приложения, които не ползват MVC Framework, взаимодействията с потребителя са организирани и контролирани чрез страници. За разлика от това в приложения с MVC framework взаимодействията с потребителя са организирани чрез контролери и методи за действие. Контролерът определя метода за действие и може да включва толкова методи за действие, колкото са необходими.

Методи за действие обикновено имат функции, които са пряко свързани с взаимодействието с потребителя. Примери за взаимодействие с потребителите са въвеждане на URL адрес в браузъра, кликване върху линк, и подаването на формуляр. Всяко едно от тези потребителски взаимодействия изпраща заявка към сървъра. Във всеки един от тези случаи, URL адреса от заявката съдържа информация, която MVC Framework използва за да включи метод за действие.

```
using System;
using System.Collection.Generic;
using System.Linq;
using System.Web;
using System.Web.MVC;

namespace ControllerApplication.Controllers
{
        public class TestingController : Controller
        {
                //
                // GET: /Testing/

                public ActionResult Index()  //Default Action
                {
                        return View();
                }

                public ActionResult Testing1()
                {
                        return View();
                }

                public ActionResult Testing2(string x)
                {
                        return View();
                }
        }
}
```

В посочения горе пример имаме три действия (Index,Testing1,Testing2), които са дефинирани в контролер клас „TestingController“. За да извикаме различни дей-ствия, трябва да напишем следното в полето за адреси:

Начин на изписване: /{controller}/{action}/{id}

- /TestingController/Index :- Извиква Index() действие.
- /TestingController/Testing1 :- Извиква Testing1() действие.
- /TestingController/Testing2/pijush :- Извиква Testing2() действие с „pijush“ за Id параметър.
- /TestingController :- Извиква Index() действие.

3.2.2.2 Списък резултати от действия

Action Results

- View():- Представлява ASP.NET MVC изглед т.е., когато вашия браузър връща HTML. Това е най-рапространения „ActionResult“, който може да върне един контролер.
- PartialView():- Извиква част от ASP.NET MVC изгледа.
- RedirectToAction():- Пренасочва от едно контролер действие в друго. Параметри с които може да се изполва това действие са:

```
-actionName: Името на действието.
-controllerName: Името на контролера.
-routeValues: Стойностите които са предадени на действието.
```

Метода RedirectToAction() връща "302 found HTTP" статус код на браузъра, за да може да направи пренасочването към новото действие. Едно от предимствата на това действие е, че когато се прави пренасочване в браузъра — полето за адреси се обновява с новия URL линк. Недостатък е, че трябва да се направи повторна заявка от браузъра към сървъра.

- Redirect():- Пренасочва към друг контролер или URL линк.
- Content():- Данни които постъпват към браузъра. Параметрите, които могат да се използват като аргументи, са:

```
-string: Изобразява string на браузъра.
-contentype: Типът MIME на данните (по подразбиране за text/html).
-contentEncoding: Текстовото кодиране на данните (например:-Unicode или
ASCII)
```

- Json():- JavaScript object Notation (JSON) е изобретен от Douglas Crockford като по-лека алтернатива на XML за пращане на данни по интернет в Ajax приложения. Метода Json() използва клас в .NET framework наречен JavaScriptSerializer за да преобразува обект в JSON репрезентация.
- File():- Връщане на фаил от действие. Този метод приема следните параметри:

```
-filename, contentType, fileDownloadName, fileContents, fileStream.
```

- JavaScript():- Представлява JavaScript фаил.
- HandleUnknownAction():- Този метод се извиква автоматично когато контролера не може да намери ресурс. По подразбиране метода изкарва грешка "404 resource Not Found". Съобщението за грашка може да бъде сменено с друго, това обаче изисква да се пренапише контролер класа.

3.2.3 Изгледи

Изгледите са тези, които определят как ще бъде визуализиран потребителският интерфейс (UI) на приложението. В ASP.NET MVC се поддържат средства (engines) за генериране на изгледи.

Начин на работа на изгледите

Когато потребителт взаимодейства с изглед, данните се **рутират** от изгледа до метод за действие, който от своя страна може да създаде друг изглед. Едно MVC приложение може да има няколко контролери, всеки от които може да съдържа множество методи за действие, а всяко действие може да създаде различен изглед. Изгледите са организирани в папки, като името им се определя от това на свързания контролер (например изгледите от HomeController са в папката \Views\Home).

3.2.4 Валидация

Валидирането на входните данни, въведени от потребителя (user input), позволява да се уверим, че те съответстват на модела на данните в ASP.NET MVC. По този начин се защитава приложението от потребителски грешки или злонамерени потребители. Има много начини да се включи валидиране в едно MVC приложение. Например, може да бъде използвана публично достъпна библиотека за **валидиране**, като Microsoft Enterprise Library Validation Block.

3.2.4.1 Как работи валидирането?

Когато потребителят потвърди (submit) формуляр(form), данните от формуляра се предават на **метод за действие** (action method) с помощта на ViewDataDictionary колекция. ViewDataDictionary има ModelState свойство(property), което съдържа колекция от ModelState обекти. За всеки модел, който е дефиниран в MVC приложението, се създава съответния ModelState обект и добавя в колекцията. **Действието**, което получава данните от формуляра определя правилата за проверка, прилагани към формуляра с данни. Ако правилото е нарушено, действието използва ModelState свойство за да предаде информация за грешка обратно към **изгледа**.

3.2.4.2 Пример за валидация

Примерът показва клас Person, който дефинира свойства (properties) за съхраняване на лични данни:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zipcode { get; set; }
    public string Phone { get; set; }
    public string Email { get; set; }
}
```

Следното дефинира форма за въвеждане на стойности и създаване на инстанция на класа Person. Ако потребителят въведе невалидна стойност за инстанцията на Person, изгледът се генерира отново, този път със съобщение за грешка, което е подадено на изгледа от свойство на ModelState:

```
<h2>Create</h2>
<%= Html.ValidationSummary("Create was unsuccessful. Please correct the
errors and try again.") %>

<% using (Html.BeginForm()) {%>

    <fieldset>
        <legend>Fields</legend>
        <p>
            <label for="Name">Name:</label>
            <%= Html.TextBox("Name") %> Required
            <%= Html.ValidationMessage("Name", "*") %>
        </p>
        <p>
            <label for="Age">Age:</label>
            <%= Html.TextBox("Age") %> Required
            <%= Html.ValidationMessage("Age", "*") %>
        </p>
        <p>
            <label for="Street">Street:</label>
            <%= Html.TextBox("Street") %>
            <%= Html.ValidationMessage("Street", "*") %>
        </p>
        <p>
            <label for="City">City:</label>
            <%= Html.TextBox("City") %>
            <%= Html.ValidationMessage("City", "*") %>
        </p>
        <p>
            <label for="State">State:</label>
            <%= Html.TextBox("State") %>
            <%= Html.ValidationMessage("State", "*") %>
        </p>
        <p>
            <label for="Zipcode">Zipcode:</label>
            <%= Html.TextBox("Zipcode") %>
            <%= Html.ValidationMessage("Zipcode", "*") %>
        </p>
        <p>
            <label for="Phone">Phone:</label>
            <%= Html.TextBox("Phone") %> Required
            <%= Html.ValidationMessage("Phone", "*") %>
        </p>
```

```
        <p>
            <label for="Email">Email:</label>
            <%= Html.TextBox("Email") %> Required
            <%= Html.ValidationMessage("Email", "*") %>
        </p>
        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>

<% } %>
<div>
    <%=Html.ActionLink("Back to List", "Index") %>
</div>
```

Ако възникне грешка при валидирането, **методът за действие** извиква метода AddModelError за да добави грешката в свързания обект ModelState. След като се изпълнят правилата за валидиране, методът за действие използва свойството IsValid на колекциата ModelStateDictionary за да се определи дали получената информация е в съответствие с **модела**:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(Person person)
{
    if (person.Name.Trim().Length == 0)
    {
        ModelState.AddModelError("Name", "Name is required.");
    }
    if (person.Age < 1 || person.Age > 200)
    {
        ModelState.AddModelError("Age", "Age must be within range 1 to
200.");
    }
    if ((person.Zipcode.Trim().Length > 0) &&
(!Regex.IsMatch(person.Zipcode, @"^\d{5}$|^\d{5}-\d{4}$")))
    {
        ModelState.AddModelError("Zipcode", "Zipcode is invalid.");
    }
```

```
    if (!Regex.IsMatch(person.Phone, @"((\(\d{3}\) ?)|(\d{3}-))?\d{3}-
\d{4}"))
    {
        ModelState.AddModelError("Phone", "Phone number is invalid.");
    }
    if (!Regex.IsMatch(person.Email, @"^[\w-\.]+@([\w-]+\.)+[\w-
]{2,4}$"))
    {
        ModelState.AddModelError("Email", "Email format is invalid.");
    }
    if (!ModelState.IsValid)
    {
        return View("Create", person);
    }

    people.Add(person);

    return RedirectToAction("Index");
}
```

3.2.5 Рутиране

3.2.5.1 Какво е рутиране?

Рутирането е метод за валидация и обработка на заявката от потребителя. След като рутирането получи заявката, то проверява дали тази заявка отговаря на рутиращите шаблони и ако бъдат намерени съответствия, първият физически поставен в кода шаблон, отговарящ на дадената заявка, изпълнява функции за подаване на данните от заявката към съответните **контролери**, **действия** и **параметри**.

3.2.5.2 Как се регистрира и дефинира рутирането?

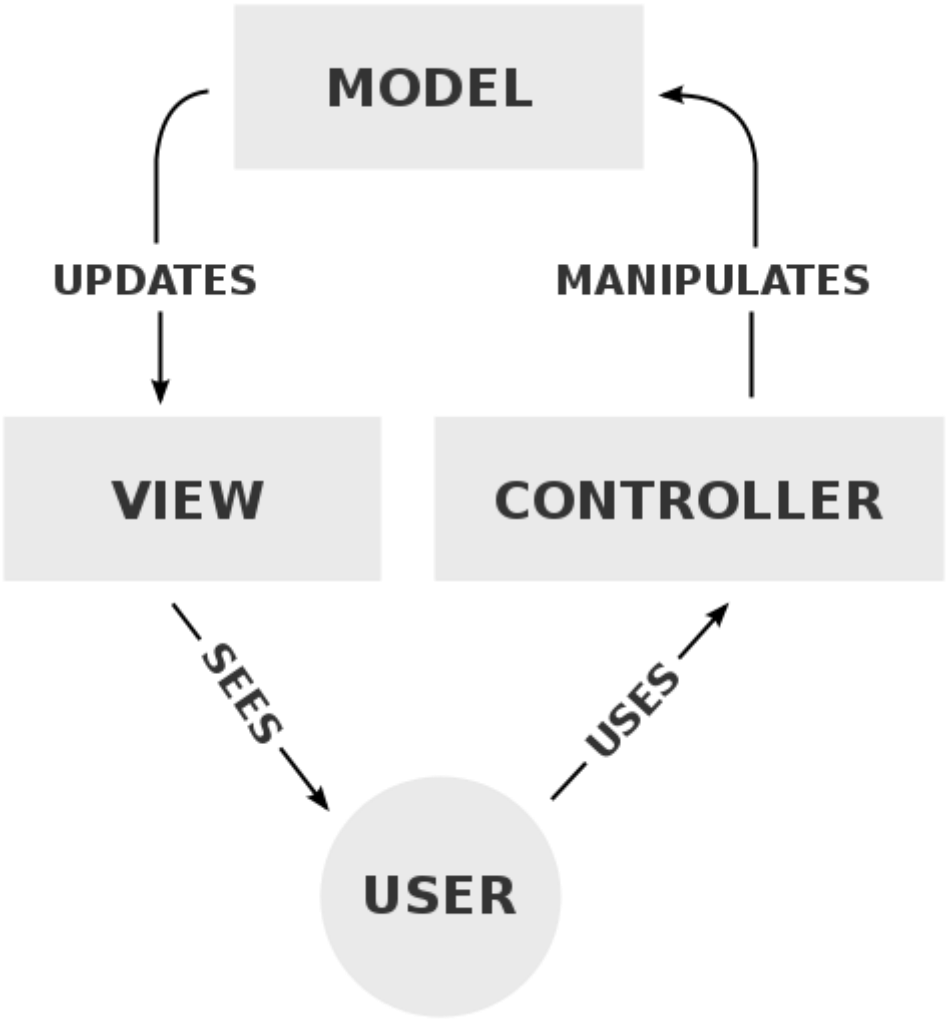
При стартиране на **ASP.NET MVC** приложението първият файл, който се стартира, се нарича "**Global.asax**" и съответно в него първият метод, който се извиква, е с име "Application_start()".Във този метод има фунции за регистрация на **Area**, **WebApi**, **Filters**, **Routing** и **Bundle**. Именно тук се регистрира и нашето рутиране (Routing) чрез извикване на класа **RoutingConfig**, през метода **RegisterRoutes** и през него се регистрира рутирането в глобалната таблица **RouteTable**.

```
public static void RegisterRoutes(RouteCollection routes)
{
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new {
                        controller = "Home",
                        action = "Index",
                        id = UrlParameter.Optional
                }
        };
}
```

Шаблоните в рутирането са **case-insensitive**, което означава, че шаблонът ще разпознае данните без значение дали са с малки или главни букви. Ако при заявката никой от шаблоните не бъде обхванат, то тогава сървърът връща **error-404**.

3.3 MVC (Model – View - Controller) шаблон

Модел-Изглед-Контролер (Model–View-Controller или MVC) е архитектурен шаблон за дизайн в програмирането, основан на разделянето на бизнес логиката от графичния интерфейс и данните в дадено приложение. За пръв път този шаблон за дизайн е използван в програмния език Smalltalk.



Фиг. 3 : Типично взаимоотношение между Model, View, и Controller

3.3.1 Описание на MVC шаблон

- **Модел (Model)** - ядрото на приложението, предопределено от областта, за която се разработва. Обикновено това са данните от реалния свят, които сме моделирали и над които искаме да работим - да въвеждаме, променяме, визуализираме и т.н.Трябва да се направи разлика между реалния обкръжаващ ни свят и въображаемия абстрактен моделен свят, който е продукт на нашият разум, който ние възприемаме във вид на твърдения, формули, математическа символика, схеми и други помощни средства. Например в банково приложение това биха били класовете, описващи клиентите, техните сметки, транзакциите, които са осъществили и т.н., както и класовете за извършване на операции над тези обекти (engines).

- **Изглед (View)** - тази част от изходния код на приложението, отговорна за показването на данните от модела. Например изгледът може да се състои от PHP шаблонни класове, JSP страници, ASP страници, JFrame наследници в Swing приложение. Зависи от това какъв графичен интерфейс се прави и каква платформа се използва.

- **Контролер (Controller)** - тази част от сорс кода (клас или библиотека), която взима данните от модела или извиква допълнителни методи върху модела, предварително обработва данните, и чак след това ги дава на изгледа. Например може да бъде създаден един малък обект, в който да бъдат сложени данните за транзакцията - като в контролера бъдат взети данните за транзакцията от модела, бъдат преведени датите от UNIX формат във четим от потребителя формат, бъде преобразувана валутата от долари в евро например, бъде закръглено до втория знак вместо да се виждат данните както са в модела (и в базата) до 10-тия. Също така когато се прави уеб графичен интерфейс това би довело до много лесна модификация на HTML кода дори от човек, който не е програмист - той ще гледа на шаблона просто като на обикновена HTML страница.

3.3.2 Предимства и недостатъци на MVC шаблона

- Предимства:

  - Моделът е независим от контролера и изгледа.
  - Моделът може да бъде планиран и осъществен независимо от другите части на системата.
  - За един и същи модел могат да бъдат осъществени различни изгледи (интерфейси) – например уеб интерфейс и нативен интерфейс към Facebook.
  - Контролерът и изгледът могат да бъдат променени, без да се налага промяна в модела.

- Недостатъци:

  - Софтуерната система достига ново ниво на сложност, което я прави по-трудно разбираема за софтуерните разработчици, работещи по нея.

4. Проектиране на системата

4.1 Използвани принципи за реализирането на системата – SOLID

В света на компютърното програмиране, SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) е мнемоничен акроним представен от Michael Feathers за „първите пет принципа“ основани и дефинирани от Роберт Мартин – Robert C. Martin в началото на 2000-ната година, който стои зад петте основни принципа на обектно-ориентираното програмиране. Когато принципите се прилагат заедно при разработването на една система, програмиста създава програма, която е лесна за поддръжка и разширение с течение на времето.[3] Принципите на SOLID са насоки, които могат да се прилагат по време на работа на софтуера за отстраняване на т.нар. „миризми по кода“ (код който не е написан качествено) от страна на програмиста при преработване на софтуерен код с цел той да е четим и разширяем.

4.1.1 Принцип за единствената отговорност - SRP

Всяка една единица от кода има една единствена отговорност, т.е. една променлива, метод, клас, библиотеки, фреймуърци правят едно единствено нещо.

По този начин кода става модуларен разпръснат на малки парчета и всяко парче върши строго специфицирана дейност. Изключение от този принцип прави embedded development, но там не се спазва този принцип поради ограничение на паметта.

SRP се базира на принципите:

- Strong Cohesion – силна сплотеност на функционалността на един клас, т.е. всички методи в него трябва да имат обща насока.
- Loose coupling – класа да е максимално „разкачен“ от всички останали (хипотетично, ако се изтрие един клас, проекта да може да се билдне отново). В реалността няма как да се случи, но идеята е да се стремим по възможност да избягваме зависимости между класовете.

4.1.2 Принцип отворен/затворен – OCP

В основата на принципа е идеята, че един код трябва да е лесно разширяем от други програмисти, но не трябва да може да се „бърка“, модифицира директно в самия него.

Ако е спазен този принцип и ако ни е нужно ново поведение, което трябва да надгради върху стара функционалност не трябва да става чрез директна модификация на стария код. Съгласно този принцип трябва да има начин да се надгражда старата функционалност и да се разширява понеже много често този код е компилиран.

Начини за постигане на OCP:

- Чрез наследяване. – Определен метод се прави виртуален, така че който иска да разширява този метод наследява класа и пренаписва новата желана функ-ционалност.
- Чрез параметризация. – В процедурните езици, където няма класове има опре-делени модули, методи приемащи определени параметри спрямо тях извършват определена функционалност. Т.е. ако променим параметрите, които подаваме можем да си разширим неговата функционалност. Пример в JavaScript, където можем да подаваме функция като параметър. В C# това става чрез делегати, но по правилния начин да се работи с абстракция или чрез наследяване да се пренапише поведение на родителския клас.

4.1.3 Принцип на заместване на Лисков – LSP

Възможността за заместване е принцип в обектно-ориентираното програмиране. То гласи, че в компютърната програма, ако S е подтип на T, то обектите от тип T могат да бъдат заменени от тези от тип S, без това да намалява функционалностите на програмата. По-формално, Принципът за заместване на Лисков, (Liskov substitution principle LSP) е конкретно определение на връзките между субтиповете в дадена програма, наречено строго поведение на субтиповете.

Принципът Лисков се дефинира чрез взаимозаменяемостта на обектите. Той е в действие, когато в дадена програма, обектите S от тип T, могат да заменят безпроблемно обектите T, без това да намалява функционалността на програмата. По просто казано, всеки наследник(подтип) трябва лесно да заменя всичките си базови типове. Подтипът не трябва да премахва нито една от функционалностите на базовия клас, а при нужда само да ги екстендва, ако по някаква причина даден клас не поддържа някоя от функционалностите на своя базов клас(родител), трябва да се помисли, дали се използва правилен вид наследяване.

4.1.4 Принцип за разделяне на интерфейсите – ISP

Този принцип гласи, че всеки интерфейс трябва да бъде разбит на много на брой малки интерфейси. Тези интерфейси трябва да отговарят за едно единствено нещо. Нито един клас не трябва да бъде принуждаван да имплементира методи, които няма да ползва никога.

„Дебел“, голям, пълен с различни методи интерфейс води до:

- Класовете да имплементират методи, които не са им нужни.
- Увеличена свързаност(coupling) между класовете.
- Намалена гъвкавост.
- Поддръжката става по-трудна.

4.1.5 Принцип на обръщане на зависимостите – DIP

Принципът на обръщане на зависимостите е специфичен начин за отвързване(отделяне; decoupling) на софтуерните модули. Когато следваме този принцип, модулите на по-високо ниво не зависят от тези на по-ниско ниво, като и двата трябва да зависят само и единствено от абстракции. В същото време абстракциите не трябва да зависят от детайлите, а детайлите трябва да зависят от абстракциите.

Правила за обръщане на зависимостите:

- Класовете трябва да декларират от какво имат нужда.
- Скритите зависимости трябва да бъдат показани.
- Зависимостите трябва да са абстракции.

Класически нарушения на принципа:

- Използване на ключовата дума „**new**“. Когато създавате нови инстанции на класовете, които са ви нужни, а не ги приемате както в посочените горе случаи, вие се обвързвате максимално с класовете, а не с техните абстракции(интерфейси, абстрактни класове).
- Използване на статични класове. Статичния клас не може да има абстракция над себе си(интерфейс), което означава, че не може да имате обръщане на зависимостите и статичен клас на едно място.

4.2 Dependency Injection (DI) – Какво представлява?

DI е част от общо приети добри практики в програмирането подобно на **S.O.L.I.D,** които не са абсолютни и задължителни, но съобразяването с тях, доколкото е възможно, води до по-добра архитектура, намалява „спагети код“, премахва скритите зависимости, позволява по-доброто преизползване на код.

D в **S.O.L.I.D** е за Dependency inversion, което е различно от **DI.** Много имплементации обединяват **Dependency Injection** и **Dependency Inversion** в едно.

Идеята е изключително проста, **DI** позволява класовете да не се интересуват от създаването на своите зависимости, а да ги получават на готово. Нека да видим следния код:

```
    public class CarsController : Controller
    {
        private readonly CarsService service;

        public CarsController()
        {

            this.service = new CarsService(new EfGenericRepository<Car>(new
CarsSystemDbContext()));

         }

     }
```

Примера е изкуствено опростен, но мисля че се вижда ясно какво се опитваме да направим – да извършим създаването на CarsService, в на който конструктор трябва да създадем нова инстанция на EfGenericRepository, от тип Car, на която инстанция трябва да в конструктора й CarsSystemDbContext. След което като създадем всичките инстанции ще можем да използваме CarsService за извличане, промяна, изтриване на информация от базата данни.

И точно тук имаме огромен проблем, самият клас CarsController създава всички свои зависимости в себе си, които от своя страна може да имат свой собствени зависимости. Това е изключително лоша практика поради много причини, но някой от основните са, че ние сме създали пряка зависимост – този клас за да работи трябва да има достъп до другите класове с техния **namespace** и тяхната конкретна имплементация. Този клас може да бъде използван само в тази „среда“, ако решим да го изнесем като библиотека ще трябва да вземем и останалите класове от които зависи и да задължим потребителя който ще го използва да използва и нашите класове за EfGenericRepository, Car, CarsSystemDbContext. Това е кошмарно, никой уважаващ себе си софтуерен архитект не би допуснал това да се случи.

4.2.1 DI в действие

DI е механизъм, който (най-често) използва кодиране спрямо интерфейси за да оправи цялата сложност по създаването на новите инстанции и тяхното инжектиране. Различните езици и библиотеки имат различни имплементации, но всички се свеждат до следните прости стъпки:

- Пишете си кода спрямо интерфейси
- Декларирайте зависимости на обект като опишете интерфейсите в конструктора на обекта
- Дефинирайте коя конкретна имплементация трябва да се използва е момента (най-често в конфигурационен файл) и решете какво да е поведението и, дали да е само една, или всеки път да се прави нова инстанция

В нашия случай имплементацията на кода който разгледахме преди малко би бил следния:

```
public class CarsController : Controller
{
    private readonly ICarsService service;

    public CarsController(ICarsService service)
    {
        this.service = service;
    }

}
```

По този начин CarsController класа се грижи само за създаването на CarsService класа, а не и на класовете EfGenericRepository и CarsSystemDbContext.

4.2.2 Недостатъци на DI

DI не е панацея, има своите проблеми и недостатъци, но ситуацията е, че тези недостатъци са по-малкото зло, а и някой от тях може да се премахнат или намалят използвайки добри практики. Ето някой от основните критики.

- Кода ни е силно зависим от самата **DI** библиотека Не може лесно да сменим **DI**контейнера. Иронично **DI,** който се бори със зависимостите, създава зависимост. Но според мен това не винаги е проблем, от гледна точка, че **DI** е част от framework-а който използваме, Ако имаме причина да подменяме целия framework явно имаме по-големи проблеми от **DI** контейнера.
- Често може да се загубим от интерфейси, особено при по-големи проекти. Решението на този проблем е добра софтуерна архитектура – нещо което така или иначе трябва да се прави независимо от езика и проекта.
- Честа критика е, че **DI** реално скрива сложността и създава нов слой между частите на приложението ни. Еми, то тази сложност трябва да отиде някъде, нека поне да е на място където може да се контролира.

- Забавя производителността – понеже **DI** ползва (най-често) рефлекция има известно забавяне понеже всеки обект трябва да се провери, да се види от какво зависи и да се вземе конкретната имплементация. В зависимост от конкретната реализация това време може да е доста сериозно. Също така има нужда от малко повече RAM. Но добрите **DI** контейнери са силно оптимизирани и отнемат само няколко милисекунди.

Dependency injection е механизъм който се е доказал като успешна практика за големи дълги проекти, които се развиват с години. Концепцията е тествана в битки на много езици и като изключим някои части на чистото функционално програмиране, се препоръчва да се използва винаги когато може.

Времето за научаване на конкретен **DI** контейнер не е голямо, по-големият проблем е програмистите да сменят мисленето си и да започнат да пишат кода в контекста на **DI**.

5. Разработка

5.1 Структура на системата

Системата трябва да може да се интегрира бързо, лесно и без много промени по клиентската част в други уеб-базирани приложения, като трябва да могат да се използват други уеб-услуги ако са предоставени. За тази цел най-удобно е да се използва трислойна архитектура, включваща следните слоеве:

- **Слой за данни** (Back end) - в случая се използва MSSQL сървър. При интеграция с друга система може да се използва всякаква система за управление на бази от данни (като MySQL), както и други форми на съхранение на данни (XML, CSV, дори и двоични файлове). Клиентското приложение по никакъв начин не зависи от слоя за данни.
- **Сървърен слой** - в случая е ASP.NET приложение, което е отговорно за извличане на данни от базата, обработката им за да бъдат подходящи за консумация от клиентски приложения и публикуването им в интернет. Този слой при нужда също може да бъде изграден с друга технология. Слоят в случая е разделен на подслоеве, което ще бъде разгледано по-късно.
- **Клиентско приложение** - изградено е на ASP .NET MVC. То е отговорно за извличане на данните от уеб-услугите и показването им в удобен за потребителя вид. Освен това предоставя възможност за интеракция с потребителя. Разделено е условно на подслоеве, които ще бъдат разгледани по-късно.

Фиг. 4 : Схематично изобразяване на избраната архитектура

Предимства на избраната архитектура:

- Слаба обвързаност на информацията с нейното представяне на клиента. Връзката към база данните и уеб-услугите са разделени от клиентското приложение в отделни class library-та.
- Сигурност – сървъра, на който се хостват уеб услугите може да бъде скрит зад защитна стена (firewall), информацията се предава по HTTP протокол, и защитната стена не пречи, тъй като обикновено порт 80 остава “отворен”.

Недостатъци:

- Увеличен трафик между уеб услугите и клиентското приложение. Тъй като информацията се предава под формата на XML нейния обем значително се увеличава.

5.2 Описание на Системата

Системата се състои от 5 отделни class library-та, 2 тестови проекта, 1 конзолно приложение и 1 MVC клиент.

Петте class library-та са:

- CarsSystem.Data - в това class library е направена миграцията към базата данни, както и е имплементиран базовия EfGenericRepository<T>
- CarsSystem.Data.Models - в това class library се намират всички модели, използвани за създаването на базата данни
- CarsSystem.Services - в това class library са отделени видовете “services” (като UserService, CarsService, FilterService и MailService), чрез които е направена връзката между базата данни и клиента на приложението
- CarsSystem.Auth - в това class library е отделена логиката за автентикация на отделните потребители
- Common - в това class library са отделени всички константи използвани в проекта

Двата тестови проекта са:

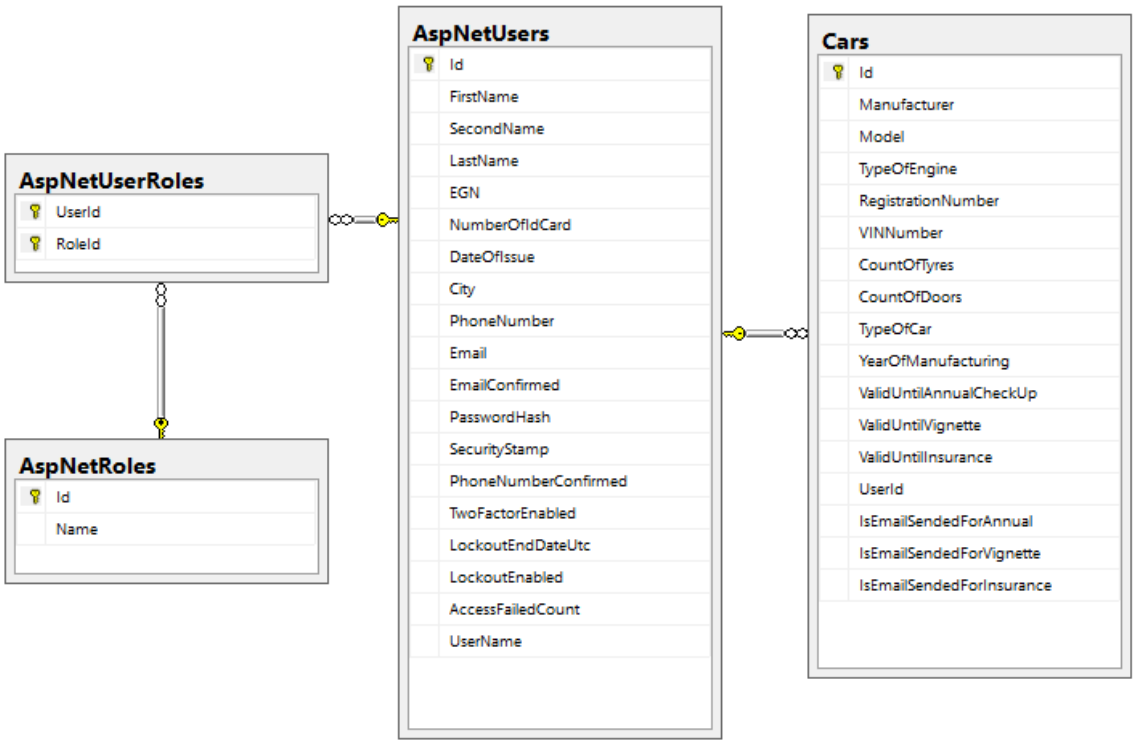
- CarsSystem.IntegrationTests – в този проект се намират Integration тестове на системата
- CarsSystem.Tests – в този проект се намират unit тестове на системата

Конзолното приложение е CarsSystemConsoleApp. То се използва за автоматизираното изпращане на e-mail-и (чрез Task Scheduler) до тези потребители, на които на днешния ден изтича – годишен технически преглед, застраховка или винетна такса.

MVC клиента е CarsSystem.WebClient.MVC. В него са имплементирани всички Web страници на приложението.

5.3 База от данни

Базата данни е написана чрез подхода Code First. Това означава, че първоначално са написани необходимите модели (таблици за нея), след което чрез ORM-а Entity Framework са генерирани към MSSQL база данни. Тук ще бъдат изброени таблиците, които се използват в проекта. За текущия проект се използва изградената база от данни ASP .NET Identity (от където са таблиците AspNetUsers, AspNetRoles и ASPNetUserRoles, необходими за запазване на информация за регистрираните потребители и техните роли) плюс таблица Cars, която се използва за запазване на информация за автомобилите на потребителите на дадения сервиз.



Фиг. 5 : Диаграма на базата данни

Създадената база от данни се състои от 4 таблици, от които основните са две: тази на клиентите (AspNetUsers) и тази на техните автомобили (Cars). Таблицата AspNetRoles се използва за запазване на „роля“ (Admin или User) на дадения клиент. AspNetUserRoles таблицата се използва за връзка на таблиците – AspNetUsers и AspNetRoles.

5.3.1 Основни полета на таблиците в базата данни CarsSystem

Основните полета на таблица AspNetUsers са:

- Id – цяло число – уникален идентификатор, първичен ключ (Primary key)
- FirstName – символен низ – име на потребителя
- SecondName – символен низ – презиме на потребителя
- LastName – символен низ – фамилия на потребителя
- EGN – цяло число – Единен Граждански Номер на потребителя
- NumberOfIdCard – цяло число – номер на Л.К.
- DateOfIssue – дата – дата на изтичане на Л.К.
- City – символен низ - град
- Email – символен низ – e-mail на потребителя
- Username – символен низ – потребителско име, с което да се логне, на потребителя

Основните полета на таблица Cars са:

- Id - GUID - уникално генериран идентификатор, първичен ключ (Primary key)
- UserId - цяло число - външен ключ (Foreign key) чрез който таблиците – AspNetUsers и Cars се свързват с връзка 1 към много
- Manufacturer - символен низ – Производител на автомобила
- Model - символен низ – модел на автомобила
- TypeOfEngine – енумерация – тип на двигателя на автомобила
- RegistrationNumber – символен низ – регистрационен номер на автомобила
- VINNumber - символен низ – идентификационен номер на автомобила
- CountOfTyres - цяло число – брой гуми на автомобила
- CountOfDoors – цяло число брой врати на автомобила
- TypeOfCar – енумерация – типа автомобил
- YearOfManufacturing – дата – дата на производство на автомобила
- ValidUntilAnnualCheckUp – дата – дата на валидност на годишен технически преглед
- ValidUntilVignette – дата – дата на валидност на винетна такса
- ValidUntilInsurance – дата – дата на валидност на застраховка
- IsEmailSendedForAnnual – булево поле – използва се за проверка дали е изпратен e-mail
- IsEmailSendedForVignette – булево поле – използва се за проверка дали е изпратен e-mail
- IsEmailSendedForInsurance – булево поле– използва се за проверка дали е изпратен e-mail

Основните полета на таблица AspNetRoles са:

- Id – цяло число – уникален идентификатор, първичен ключ (Primary key)
- Name – символен низ – име на ролята на потребителя

Основните полета на таблица AspNetUserRoles са:

- UserId - цяло число - външен ключ (Foreign key) чрез който таблиците – AspNetUsers и AspNetUserRoles се свързват с връзка 1 към много
- RoleId - цяло число - външен ключ (Foreign key) чрез който таблиците – AspNetRoles и AspNetUserRoles се свързват с връзка 1 към много

5.4 Сървърен слой

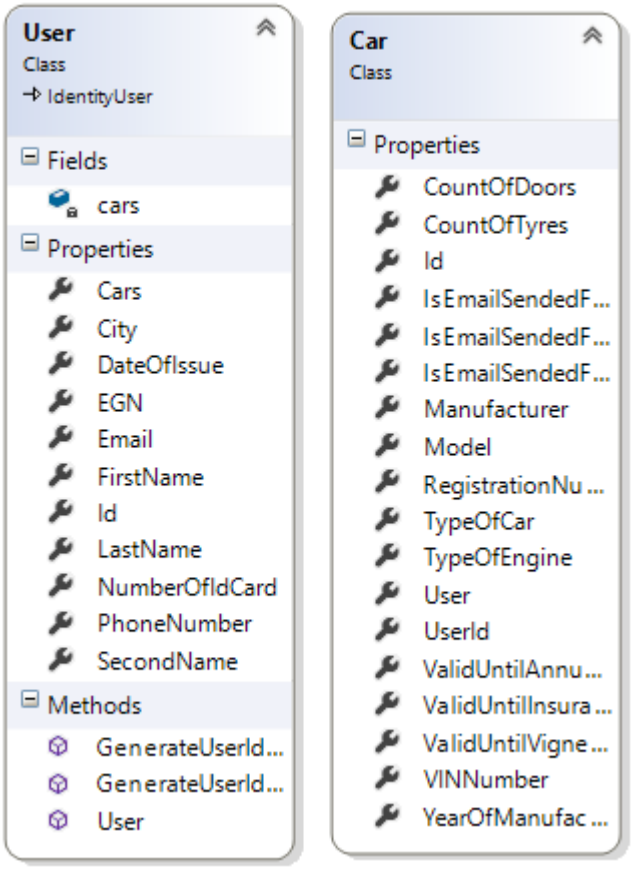
5.4.1 Структура

Съществуващата система CarsSystem има следните слоеве:

- Слой за достъп до базата данни – CarsSystem.Data и CarsSystem. Data.Models. В него се осъществява достъп до базата данни без да се обработват данните по какъвто и да било начин.
- Бизнес слой – CarsSystem.Services. В този слой се намира бизнес логиката и всякакви обработки на данните.
- Презентационен слой - CarsSystem.WebClient.MVC. Това е ASP.NET MVC уеб приложение, където се намира логиката за визуализацията на информацията.

5.4.2 Бизнес Обекти

Основните бизнес обекти в системата са Клиент/Потребител (User) и Автомобил (Car). Те съдържат информация за даден клиент и неговия автомобил. Полетата които имат тези бизнес обекти могат да бъдат видяни на фигура 6.



Фиг. 6 : Class диаграми на бизнес обектите User и Car

5.4.3 Слой за достъп до базата данни

В слоя за достъп до базите данни се включват двете class library-та : CarsSystem.Data и CarsSystem.Data.Models.

- CarsSystem.Data.Models съдържа моделите, които се използват в системата. Това са: User, Car, EngineType и CarType.
- CarsSystem.Data съдържа класовете за създаване на базата данни. Това са: CarsSystemDbContext и EfGenericRepository. В класа EfGeneric Repository са имплементирани базови методи като: GetAll, GetById, Add, Update, Delete, SaveChanges и други.

5.4.4 Web услуги

Web услугите на проекта CarsSytem се намират в class library-то CarsSystem. Services. То се състои от 4 класа (UsersService, CarsService, FilterService, MailService) и съответно 4 интерфейса (IUsersService, ICarsService, IFilterService, IMailService) за всеки един от класовете. Чрез методите имплементирани в тези класове се правят заявки към базата данни свързани с извличане, записване, филтриране, промяна и други действия на информация, посредством класа EfGenericRepository. По-надолу ще разгледаме всеки един от имплементираните методи в CarsSystem. Services проекта.

В UsersService класа са имплентирани методи, които извършват действия свързани с потребителите в системата. Методите са:

- IEnumerable<User> GetAllUsers() – извлича колекция от всички регистрирани потребители в системата
- User GetUserById(string id) – извлича информация, от базата данни, за специфичен потребител по зададено id
- IEnumerable<User> GetUserByEGN(long egn) – филтрира колекция от всички потребители, на които съвпада ЕГН-то, от базата данни, по зададено ЕГН
- string GetUserId(Car car) – връща id-то на потребител на регистриран автомобил в системата
- void Update(User user) – update-ва информация на потребител

В CarsService класа са имплентирани методи, които извършват действия свързани с автомобилите в системата. Методите са:

- IEnumerable<Car> GetAllCars() – извлича колекция от всички регистрирани автомобили в системата
- Car GetCarById(Guid id) – извлича информация, от базата данни, за специфичен автомобил по зададено id
- IEnumerable<Car> GetCarByVinNumber(string vinNumber) – филтрира колекция от всички потребители, на които съвпада ЕГН-то, от базата данни, по зададен VIN номер на автомобила
- Guid GetCarId(User user) – връща id-то на автомобил на регистриран потребител в системата
- void Update(Car car) – update-ва информация на автомобил
- void AddCar(Car carToAdd) – добавя нов автомобил в базата данни

В MailService класа е имплентиран само един метод, който извършва действия свързани с изпращането на e-mail до всички потребители в система, на които им изтича годишен технически преглед, застраховка или винетна такса на автомобила. Името на метода е void SendEmail(string subject, string content, IEnumerable<string> emails), като изисква входни параметри предмет (**subject**) на e-mail-а съдържание (**content**) на e-mail-а както и колекция (**emails**) от e-mail-и, на които да бъде изпратен e-mail.

В FilterService класа са имплентирани методи, които извършват действия свързани с изтичане на годишен технически преглед, застраховка или винетна такса на автомобила. Методите са:

- IEnumerable<Car> FilterExpiringVignetteCarsInTheNextSevenDays() – филтрира автомобилите на които винетната такса изтича в следващите седем дни
- IEnumerable<Car> FilterExpiringVignetteCarsToday() – филтрира автомобилите на които винетната такса изтича на днешния ден
- IEnumerable<Car> FilterExpiringInsuranceInTheNextSevenDays() – филтрира автомобилите на които застраховката изтича в следващите седем дни
- IEnumerable<Car> FilterExpiringInsuranceToday() – филтрира автомобилите на които застраховката изтича на днешния ден
- IEnumerable<Car> FilterExpiringAnnualCheckUpInTheNextSevenDays() – филтрира автомобилите на които годишния технически преглед изтича в следващите седем дни
- IEnumerable<Car> FilterExpiringAnnualCheckUpToday() – филтрира автомобилите на които годишния технически преглед изтича на днешния ден
- IEnumerable<string> GetMailsForCarsVignetteExpirationInTheNextSevenDays() – извлича колекция от всички e-mail-и, на потребителите, на които винетната такса изтича в следващите седем дни
- IEnumerable<string> GetMailsForCarsVignetteExpirationToday() – извлича колекция от всички e-mail-и, на потребителите, на които винетната такса изтича на днешния ден
- IEnumerable<string>GetMailsForCarsInsuranceExpirationInTheNextSevenDays() – извлича колек-ция от всички e-mail-и, на потребителите, на които застраховката изтича в следващите седем дни
- IEnumerable<string> GetMailsForCarsInsuranceExpirationToday() – извлича колекция от всички e-mail-и, на потребителите, на които застраховката изтича на днешния ден
- IEnumerable<string>GetMailsForCarsAnnualCheckUpExpirationInTheNextSevenDays() – извлича колекция от всички e-mail-и, на потребителите, на които годишния технически преглед изтича в следващите седем дни
- IEnumerable<string> GetMailsForCarsAnnualCheckUpToday() – извлича колекция от всички e-mail-и, на потребителите, на които годишния технически преглед изтича на днешния ден
- void SaveChanges() – използва се запазване на информация за това дали e-mail е изпратен до филтрирани потребители

5.5 Клиентско приложение

5.5.1 Структура

Клиентското приложение е изградено съгласно MVC шаблона (виж т. 3.3.). Това обуславя използването на следните 3 слоя:

- Model – този клас е отговорен за всякаква бизнес логика свързана с управлението на данните. Служи като DataContext на презентационния слой.
- View – презентационен слой, който съдържа потребителския интерфейс, както и логика за филтрация на потребители и автомобили. В тази логика не се обработват данните. Данните се обработва в Controller класа.
- Controller – този клас е отговорен за обръщенията към уеб услугата и извличането и запазването на данни от нея.

Ще разгледаме логиката за добавяне на потребител и неговия автомобил в системата.

5.5.2 Model

За добавянето на потребител и неговия автомобил към системата се използват два модела. Първият е User() модел, в който се съдържат полетата за информация на новия клиент/потребител. Вторият модел е Car. Този модел се съдържат полетата за информация на автомобила на новодобавения потребител.

User модела при добавяне на нов клиент/потребител изглежда по следния начин:

```
var userToAdd = new User()
{
    UserName = receivedModel.User.Username,
    FirstName = receivedModel.User.FirstName,
    SecondName = receivedModel.User.SecondName,
    LastName = receivedModel.User.LastName,
    EGN = receivedModel.User.Egn,
    NumberOfIdCard = receivedModel.User.NumberOfIdCard,
    DateOfIssue = receivedModel.User.DateOfIssue,
    City = receivedModel.User.City,
    PhoneNumber = receivedModel.User.PhoneNumber,
    Email = receivedModel.User.Email
};
```

Car модела при добавяне на нов автомобил, за текущия потребител, изглежда по следния начин:

```
var carToAdd = new Car()
{
    Manufacturer = receivedModel.Car.Manufacturer,
    Model = receivedModel.Car.Model,
    TypeOfEngine = receivedModel.Car.TypeOfEngine,
    RegistrationNumber = receivedModel.Car.RegistrationNumber,
    VINNumber = receivedModel.Car.VINNumber,
    CountOfTyres = receivedModel.Car.CountOfTyres,
    CountOfDoors = receivedModel.Car.CountOfDoors,
    TypeOfCar = receivedModel.Car.TypeOfCar,
    YearOfManufacturing = receivedModel.Car.YearOfManufactoring,
    ValidUntilAnnualCheckUp = receivedModel.Car.ValidUntilAnnualCheckUp,
    ValidUntilVignette = receivedModel.Car.ValidUntilVignette,
    ValidUntilInsurance = receivedModel.Car.ValidUntilInsurance,
    IsEmailSendedForAnnual = false,
    IsEmailSendedForInsurance = false,
    IsEmailSendedForVignette = false,
    UserId = userToAdd.Id,
    Id = Guid.NewGuid()
};
```

5.5.3 View

View – то, което се използва за добавяне на нов клиент/потребител и неговия автомобил се казва AddUser.cshtml. В него се съдържат всички Input и други полета, които се използват за въвеждане на новата информация, за клиента/потребителя и неговия автомобил, от системния администратор на системата. То е пряката връзка между системния администратор и Backend-а на системата. Всички полета които се използват за въвеждане на информация са валидирани и по този начин, ако администратора на системата, обърка някое от тях той бива известен със съобщение, намиращо се под сбърканото поле.

5.5.4 Controller

Контролера за добавяне на нов клиент/потребител и неговия автомобил е AddUserController, а метода в него, който се грижи за това е public ActionResult AddUser(AddUserViewModel receivedModel). При натискане на бутона „Add new customer“ в view-то AddUser.cshtml, се прави проверка за това дали някое от полетата са с невалидни данни, след което метода ActionResult AddUser се извиква. Създават се нови обекти от тип User и Car. Новосъздадения потребител се създава, чрез ASP .Net Identity в базата данни. Ако регистрацията на новия потребител е успешна се изпълнява web услугата service от тип CarsService, която се използва за добавяне на автомобила на новорегистрирания клиент/потребител. Ако и това е успешно новият клиент/потребител и неговия автомобил се регистрират в системата.

Кодът на метода AddUser излежда по следния начин:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult AddUser(AddUserViewModel receivedModel)
{

    var userToAdd = new User()
    {
        UserName = receivedModel.User.Username,
        FirstName = receivedModel.User.FirstName,
        SecondName = receivedModel.User.SecondName,
        LastName = receivedModel.User.LastName,
        EGN = receivedModel.User.Egn,
        NumberOfIdCard = receivedModel.User.NumberOfIdCard,
        DateOfIssue = receivedModel.User.DateOfIssue,
        City = receivedModel.User.City,
        PhoneNumber = receivedModel.User.PhoneNumber,
        Email = receivedModel.User.Email
    };

    IdentityResult result = UserManager.Create(userToAdd, "123456");

    var carToAdd = new Car()
    {
        Manufacturer = receivedModel.Car.Manufacturer,
        Model = receivedModel.Car.Model,
        TypeOfEngine = receivedModel.Car.TypeOfEngine,
        RegistrationNumber = receivedModel.Car.RegistrationNumber,
        VINNumber = receivedModel.Car.VINNumber,
        CountOfTyres = receivedModel.Car.CountOfTyres,
        CountOfDoors = receivedModel.Car.CountOfDoors,
        TypeOfCar = receivedModel.Car.TypeOfCar,
        YearOfManufacturing = receivedModel.Car.YearOfManufactoring,
        ValidUntilAnnualCheckUp = receivedModel.Car.ValidUntilAnnualCheckUp,
        ValidUntilVignette = receivedModel.Car.ValidUntilVignette,
        ValidUntilInsurance = receivedModel.Car.ValidUntilInsurance,
        IsEmailSendedForAnnual = false,
        IsEmailSendedForInsurance = false,
        IsEmailSendedForVignette = false,
        UserId = userToAdd.Id,
        Id = Guid.NewGuid()
    };


    if (result.Succeeded)
    {
        UserManager.AddToRole(userToAdd.Id, ApplicationConstants.UserRole);
        this.service.AddCar(carToAdd);
        return RedirectToAction("Index", "Success", new { area =
"Administration" });
    }

    return View(receivedModel);
}
```

Изпращането на e-mail с известяване на клиента, че му изтича годишен технически преглед, застраховка или винетна такса се осъществява по два начина:

- Чрез Partial view-то _SendingEmailForm.cshtml, което се визуализира на всички web страници, които се използват за филтрация на клиенти/потребители
- Чрез конзолното приложение CarsSystemConsoleApp, след като бъде настроено чрез Task scheduler

Логиката за изпращане на e-mail, през web приложението, се намира в контролера FilterController и взависимост от това по какъв начин са филтрирани клиентите са имплементирани ActionResult – а:

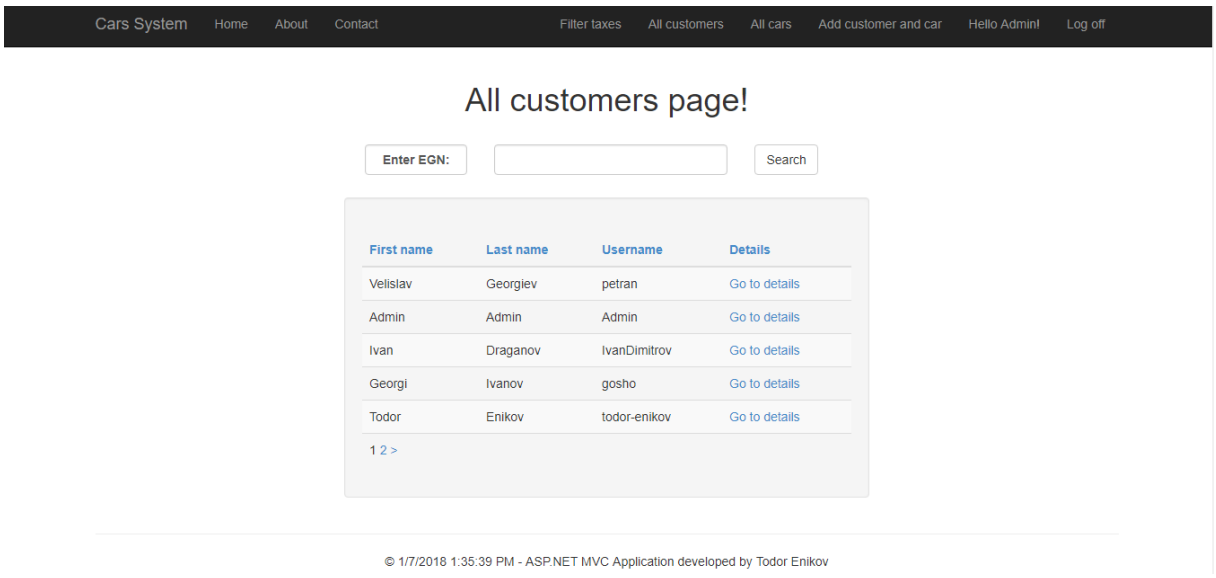
- ActionResult FilterExpiringByAnnualCheckUpInTheNextSevenDays(string emailSubjectTextBox, string emailContentBox)
- ActionResult FilterExpiringByAnnualCheckUpToday(string emailSubjectTextBox, string emailContentBox)
- ActionResult FilterExpiringByVignetteInTheNextSevenDays(string emailSubjectTextBox, string emailContentBox)
- ActionResult FilterExpiringByVignetteToday(string emailSubjectTextBox, string emailContentBox)
- ActionResult FilterExpiringByInsuranceInTheNextSevenDays(string emailSubjectTextBox, string emailContentBox)
- ActionResult FilterExpiringByInsuranceToday(string emailSubjectTextBox, string emailContentBox)

Ще разгледаме един от тях. Например ActionResult–а FilterExpiringByVignetteToday. При натискане на бутона „Send e-mails to the filtered cars!“ в view-то FilterExpiringByVignetteToday.cshtml, ActionResult FilterExpiringByVignetteToday се извиква. Като параметри на този ActionResult се подават emailSubjectTextBox (заглавие на e-mail) и emailContentBox (съдържание на e-mail). Прави се филтрация на всички клиенти/потребители, на които винетната такса изтича днес, чрез FilterService filterService. След което се изпраща e-mail, до всички филтрирани клиенти/потребители, чрез MailService mailService и полетата, които се грижат за това да се провери дали e-mail е изпратен се update-ват на булевата стойност true, след което ActionResult – а се redirect - ва към Web страницата Success (Index.cshtml в Success папката на проекта).

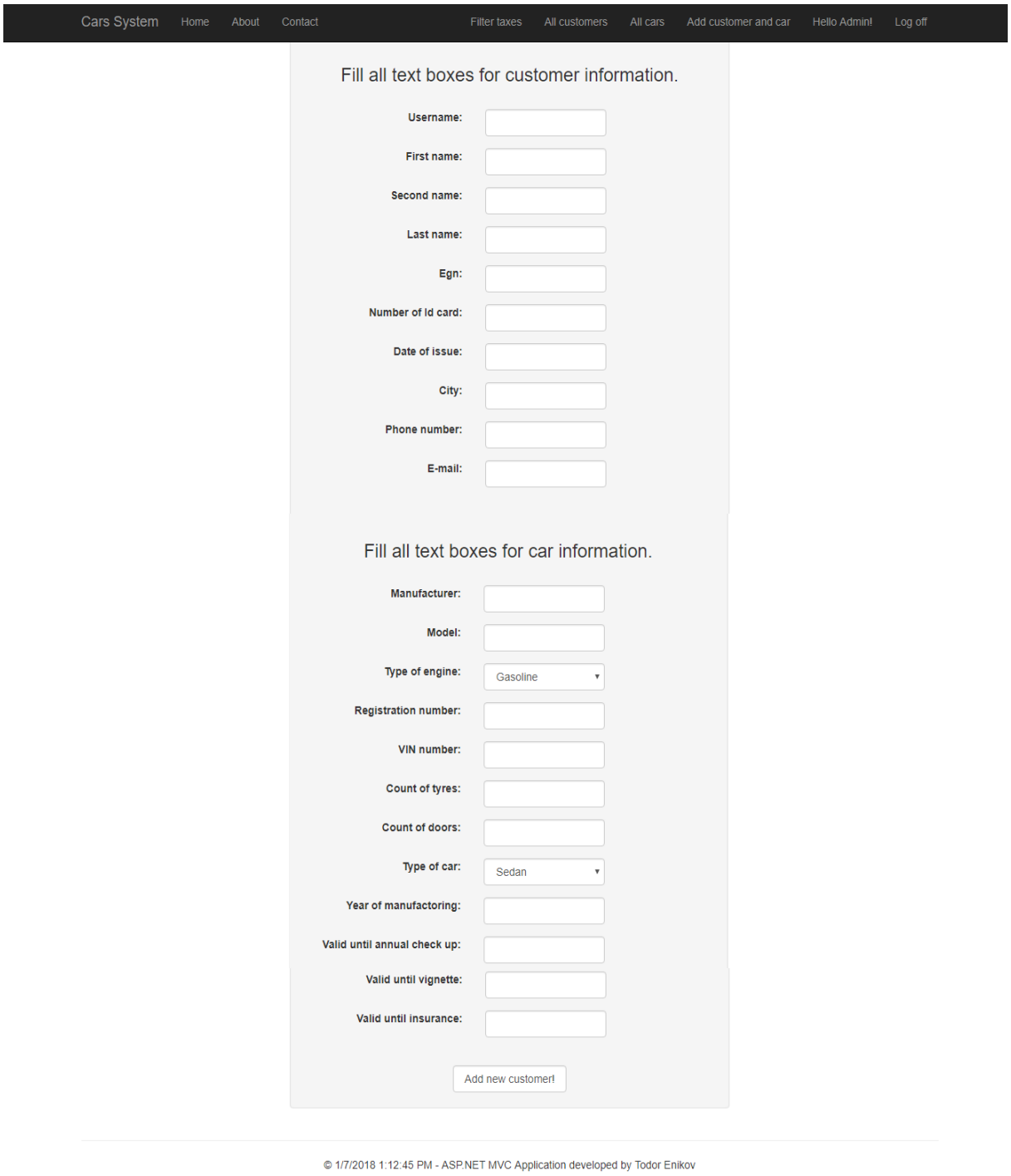
Кодът на метода FilterExpiringByVignetteToday излежда по следния начин:

```
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult FilterExpiringByVignetteToday(string emailSubjectTextBox,
string emailContentBox)
        {
            var emails = this.filterService
                              .GetMailsForCarsVignetteExpirationToday()
                              .ToList();

            this.mailService.SendEmail(emailSubjectTextBox, emailContentBox, emails);

            var cars = this.filterService
                           .FilterExpiringVignetteCarsToday()
                           .ToList();
            this.UpdateIsEmailSendedForVignettePropertyAfterSendingEmails(cars);

            return RedirectToAction("Index", "Success", new { area = "Administration"
            });
        }
```

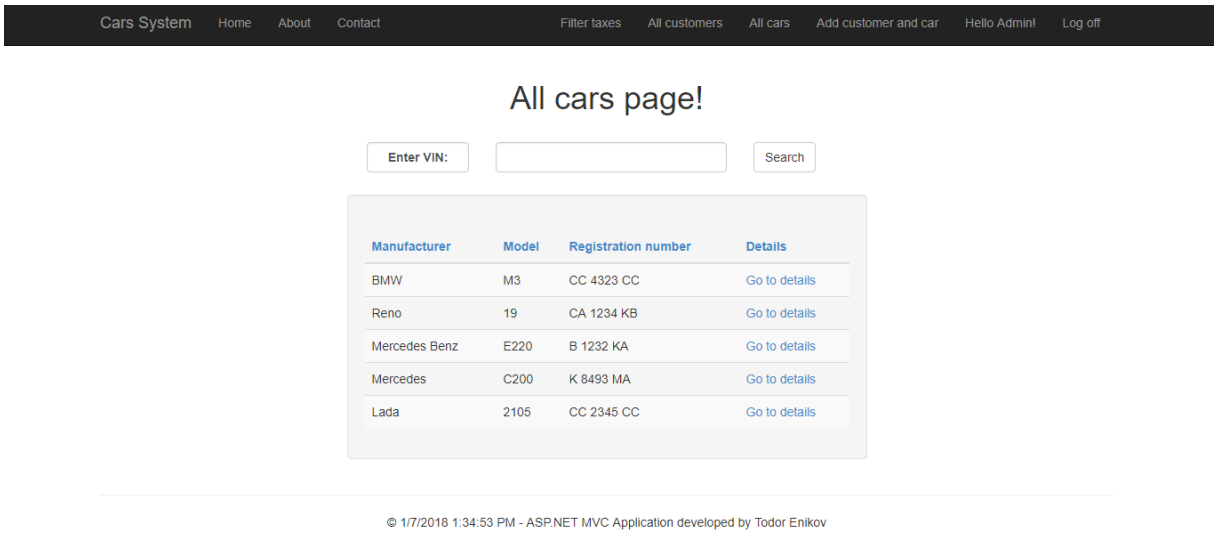
5.6 Снимки от клиентското приложение



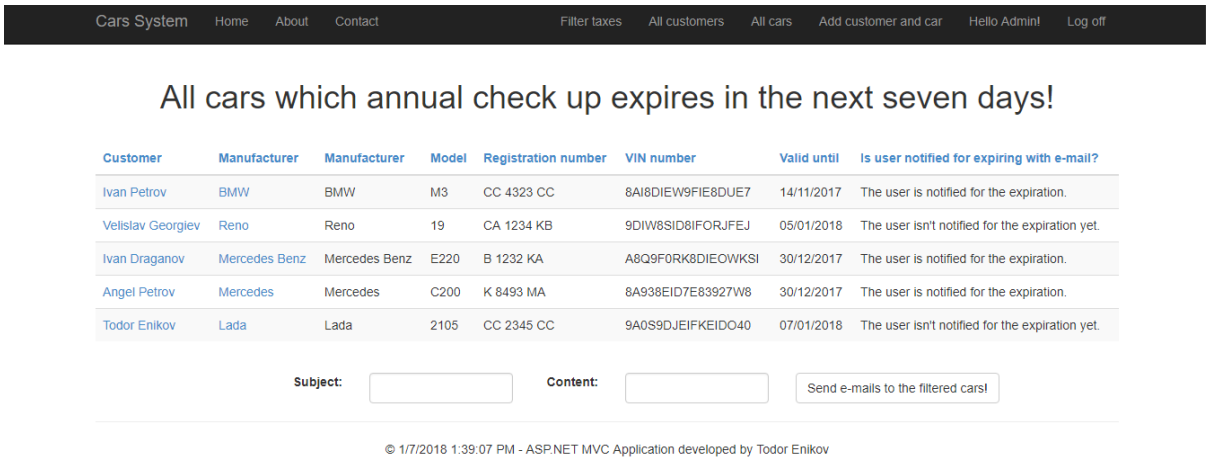
Фиг. 7: Всички регистрирани клиенти



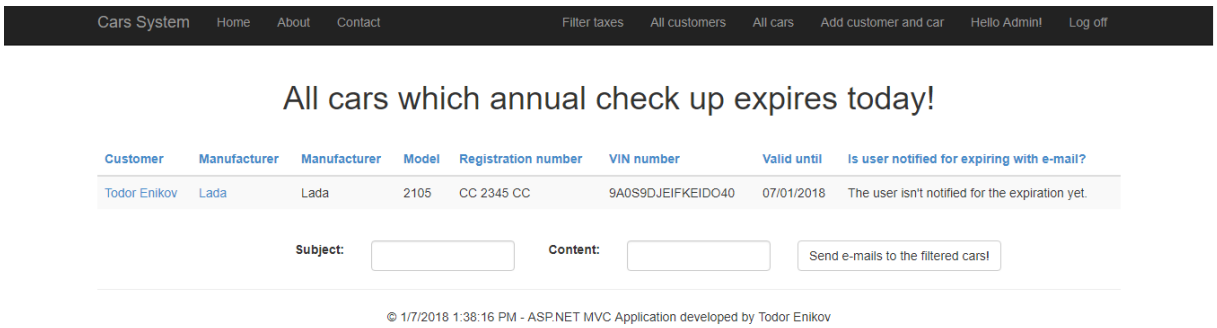
Фиг. 8: Добавяне на нов клиент/потребител и неговия автомобил



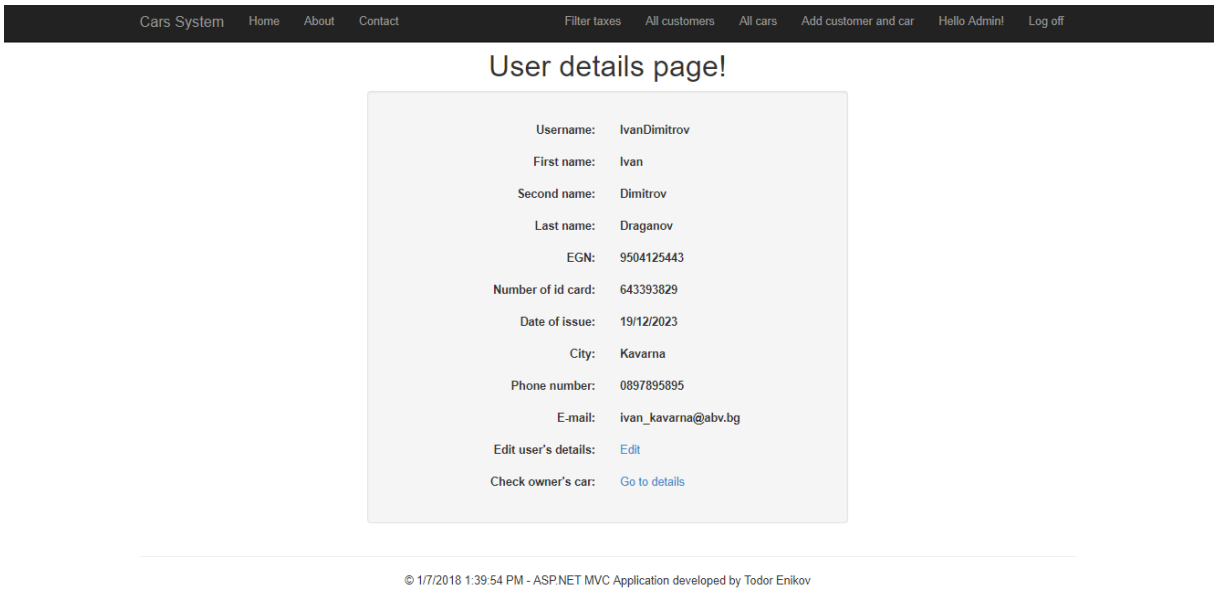
Фиг. 9: Всички регистрирани автомобили



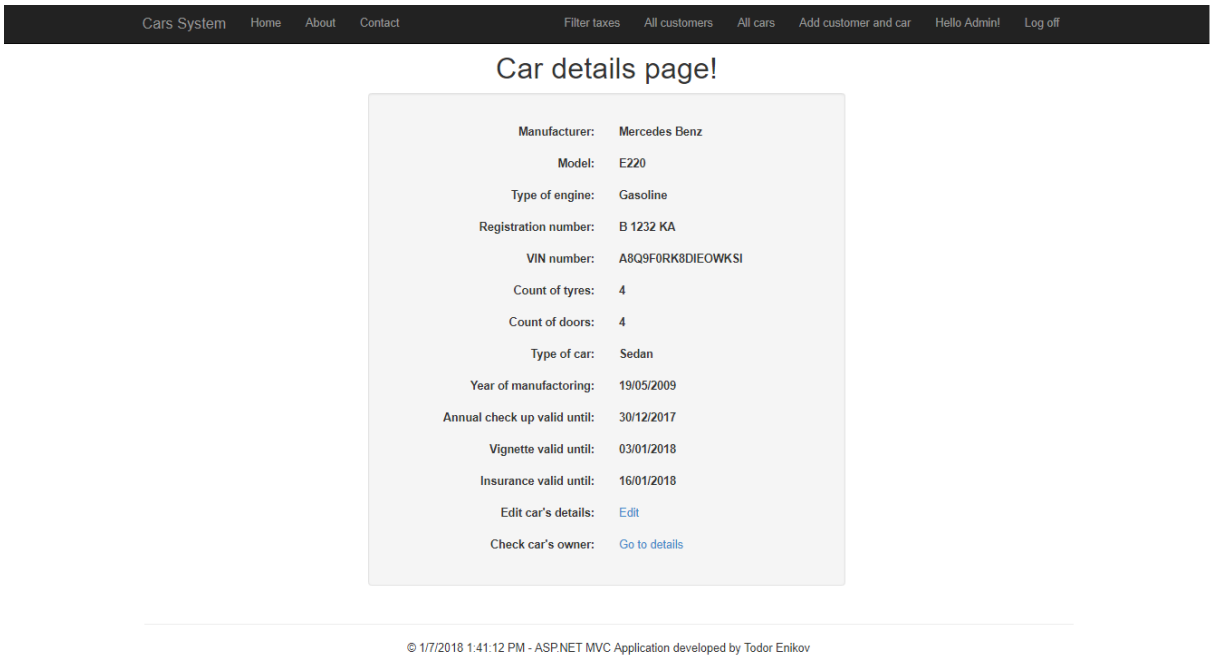
Фиг. 10: Всички автомобили, на които ГТП изтича в следващите 7 дн



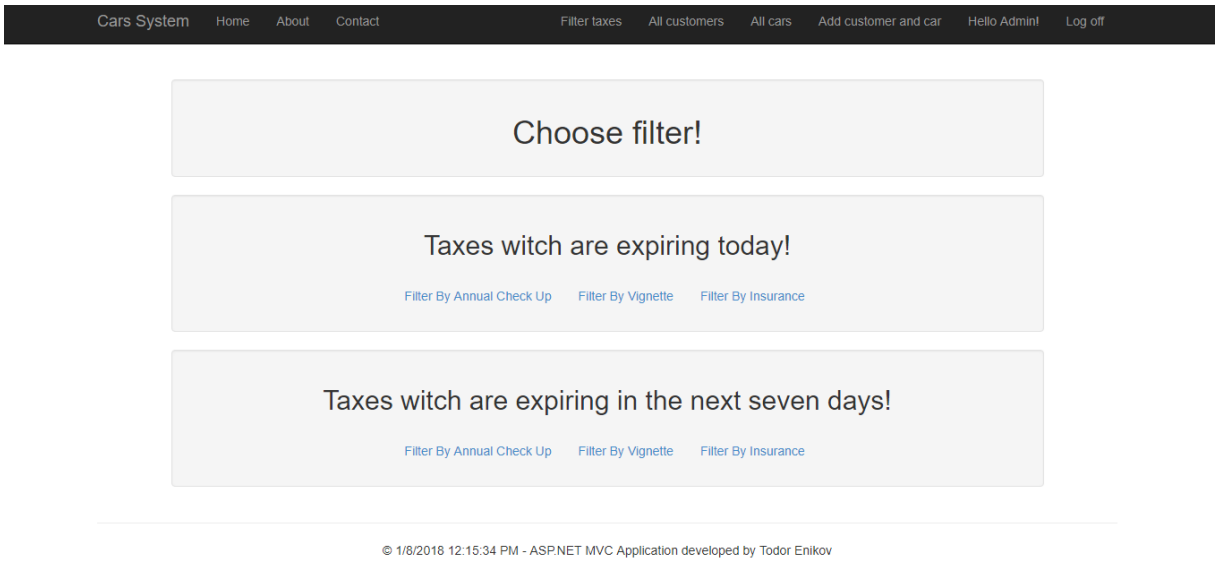
Фиг. 11: Всички автомобили, на които годишния технически преглед изтича днес



Фиг. 12: Детайлна информация за клиент/потребител



Фиг. 13: Детайлна информация за автомобил



Фиг. 14: Web страница с видовете филтрация на автомобили

6. Заключение

Разработеният програмен продукт се характеризира със следните особености:

Предимства:

- Лесен за използване потребителски интерфейс. Само с няколко движения и клика с мишката потребителят може да разгледа елементите, които го интересуват.
- Възможност за бързо внедряване на клиентското приложение в други системи.
- Добра скалируемост – разделяне на продукта на отделни слоеве, като промяна в единия не налага задължително промяна в другите.

Недостатъци:

- Липса на поддръжка на клиентското приложение за Unix базирани системи.
- Сравнително бавно първоначално зареждане на Web приложението

Насоки за бъдещо развитие:

- Сортиране на таблиците за визуализране на всики клиенти/ потребители и автомобили по азбучен ред.
- Web приложението да бъде разширено, така че да работи с всякакъв вид автомобили, като камиони, мотоциклети, джипове и други, а не само с обикновени автомобили както е имплементирано в момента.
- Поддръжка на стрелките от клавиатурата.
- Автоматизираното изпращане на e-mail известяване да бъде имплементирано чрез Windows Service или друг вид service, вместо с Task Scheduler
- Имплементиране на приложението на други езици различни от английски (Multiple Language), като – Български (BG), Немски (De), Френски (Fr), Испански (Es) и други

7. Използвана литература

1. http://gatakka.eu

2. http://www.vasil.me

3. https://bg.wikipedia.org/

4. https://en.wikipedia.org/

5. https://www.google.bg/

6. Professional ASP .NET MVC 5 – автори Брад Уилсън, Дейвид Мацън, Джон Гжалоуай и К. Скот Алън

7. Pro ASP .NET MVC 5 – автор Адам Фриман