# Система Домоуправител

## Описание

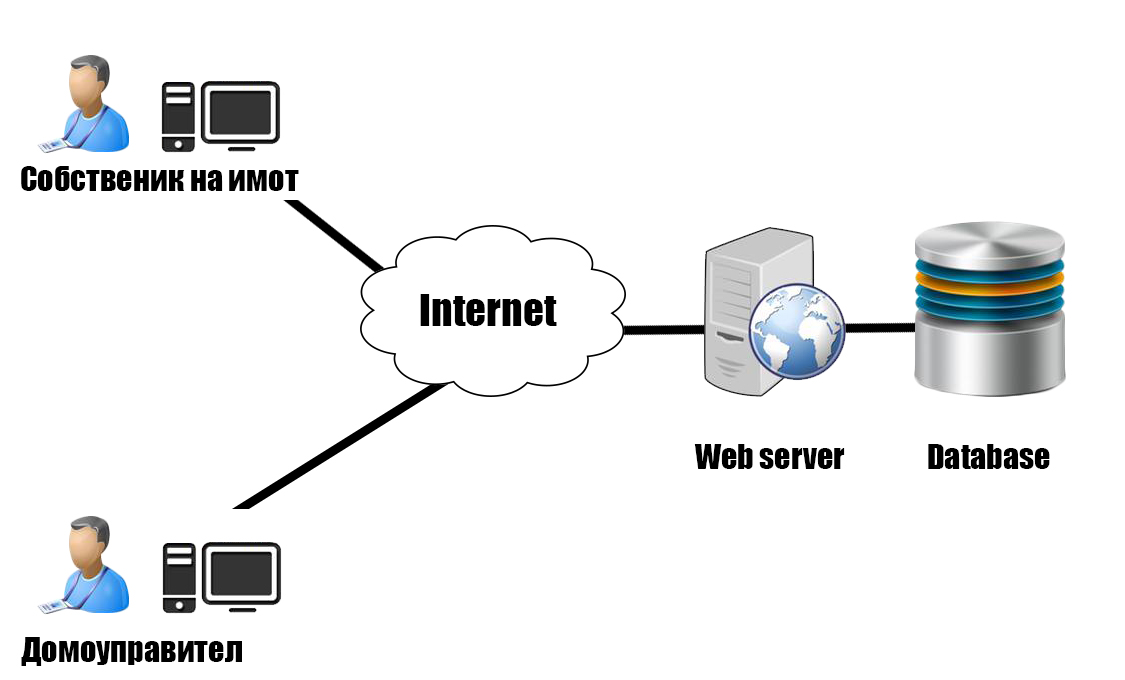
С нарастващия брой на нови сгради , а и вече съществуващите такива, се търси решение , което да улесни управлението им. Взимайки предвид броя на хората с които се налага да контактува един домоуправител а и желанието на отделните собственици да дават идеи, и участват в управлението на сградата, се налага създаването на софтуер,който да подпомага тази дейност.   
Обобщените функционалности на софтуера включват:

* В системата ще могат да бъдат регистрирани сгради.
* Към всяка сграда ще бъдат регистрирани имоти, които ще бъдат обвързани със собствениците им. Различните видове имоти ще могат да се дефинират динамично в системата, като пример това са апартамент, мазета, гаражи, паркоместа.
* Поради увеличения брой услуги, което ползва всяко едно домакинство ще бъде включена и функционалност за администриране на разходите на различните имоти. Собвствениците ще могат да проверяват своите разходи през съответното меню на софтуера.
* Анкетна функционалност. С цел подобряване на условията в жилищната сграда, а и удовлетворяване желанията на различните собственици ще се създават въпросници. Анкетите ще могат да бъдат инициирани от домоуправителите, а също така и от самите собственици. При решение взето от домоуправителя в звисимост от интереса към дадена анкета, тя ще бъде включвана към следващото общо събрание на сградата.
* Иницииране на общи събрания и информираност за тяхното провеждане.

С цел да се удовлетворят различните функционалности изполвани от различните потребители на програмата се налага интерфейс, който да предлага необходимите менюта на системата към дадения потребител. Системата се построява с използването на релационна база данни, стояща „зад“ уеб сървър , в който се съдържа самото приложение. Сървъра ще приема заявки от потребителските браузъри през световната мрежа - интернет, ще ги обработва и при необходимощ ще ги насочва към базата данни.

Директен достъп до базата от данни ще бъде възможен само за ограничен брой интернет адреси, в това число самия уеб сървър. Уеб сървърът от своя страна ще бъде отворен към интернет за да позволи използването на уеб сайта от всяка точка.

Конкретната реализация на архитектурата на даденото приложение може да се види на фигура 1.



Фигура 1

## Разбивка функционалностите

### Дефиниция за администраторите на софтуера.

Особености

Администраторите на софтуера ще могат да дефинират сградите,след като получат искане за откриване на нова сграда.  
Администраторите на софтуера могат да създават администратори на сгради и да асоциират дадена сграда с даден администратор.

### Регистриране на жилищни сгради

Изискване за регистрация:

* Град, улица ,номер на блок, администратори на сградата.
* Блока се регистрира само от админ на софтуера и може да се управлява от задедени администратори на блока.
* Един блок може да има много администратори и те се създават от админа на софтуера
* Софтуера позволява редактирането на всеки един от параметрите на вече създадена сграда.

### Регистриране на имоти

Водеща единица при жилищните сгради са имотите. Различните видове имоти биват апартаменти, гаражи, паркоместа, мазета, магазини към сградата и т.н.

* Всеки един имот се регистрира като се изискват площ, вид на имота(мазе, апартамент, гараж и т.н), собственик на имота, сградата в която се помещава имота и коментар към имота.
* Един блок може да има много имоти
* Един имот може да има само един собственик
* Един собственик може да има много имоти, от една сграда или от много сгради

### Регистриране на потребители

Потребител е всяко едно лице , което притежава имот.

* Потребителите се регистрират като записват своите имена,телефон,електронна поща и парола. След регистрация , потребителя пуска искане да бъде свързан с даден имот. След проверка по документи, администратора на сградата може да свърже потребителя с имота.

### Задължения

* След регистрация, всеки имот може да получава известявания за дължими суми, начислени към него. Всяко едно задължение се регистрира в системата от администраторите на блока.
* Всеки потребител вижда своите задължения и погасените вече такива след като се впише в профила си.
* Погасяването на задълженията става от администратора на сградата след проверка за актуално плащане.

### Анкетна функционалност

* Всеки един потребител на системата, влючително и нейните администратори могат да повдигат важни за сградата въпроси/предложения. Самите въпроси ще излизат като вид анкета, на която потребителите ще гласуват. В зависимост от въпроса/предложението, потребителя може да хареса предложението, като така допринася за това, въпроса да бъде обсъден на следващото общо събрание.
* Освен въпроса , към анкетата се вписва и момента на създаване на въпроса. По въпроса ще може да се гласува според зададен в системата срок за активност на анкетите
* При определен от администраторите процент, дадения въпрос/предложение ще се приеме за гласуване на общото събрание на сградата.
* Администраторите могат да решат да изтрият или променят въпроса ако той не е подходящ.
* Всеки един въпрос/предложение ще се записва в лист,който ще може да бъде достъпен от менюто на сайта. Предложенията ще бъдат според датата им на създаване.
* Администраторите на сградата могат да решат дали да махнат правото на даден потребител да дава предложения.

### Общо събрание и приети точки

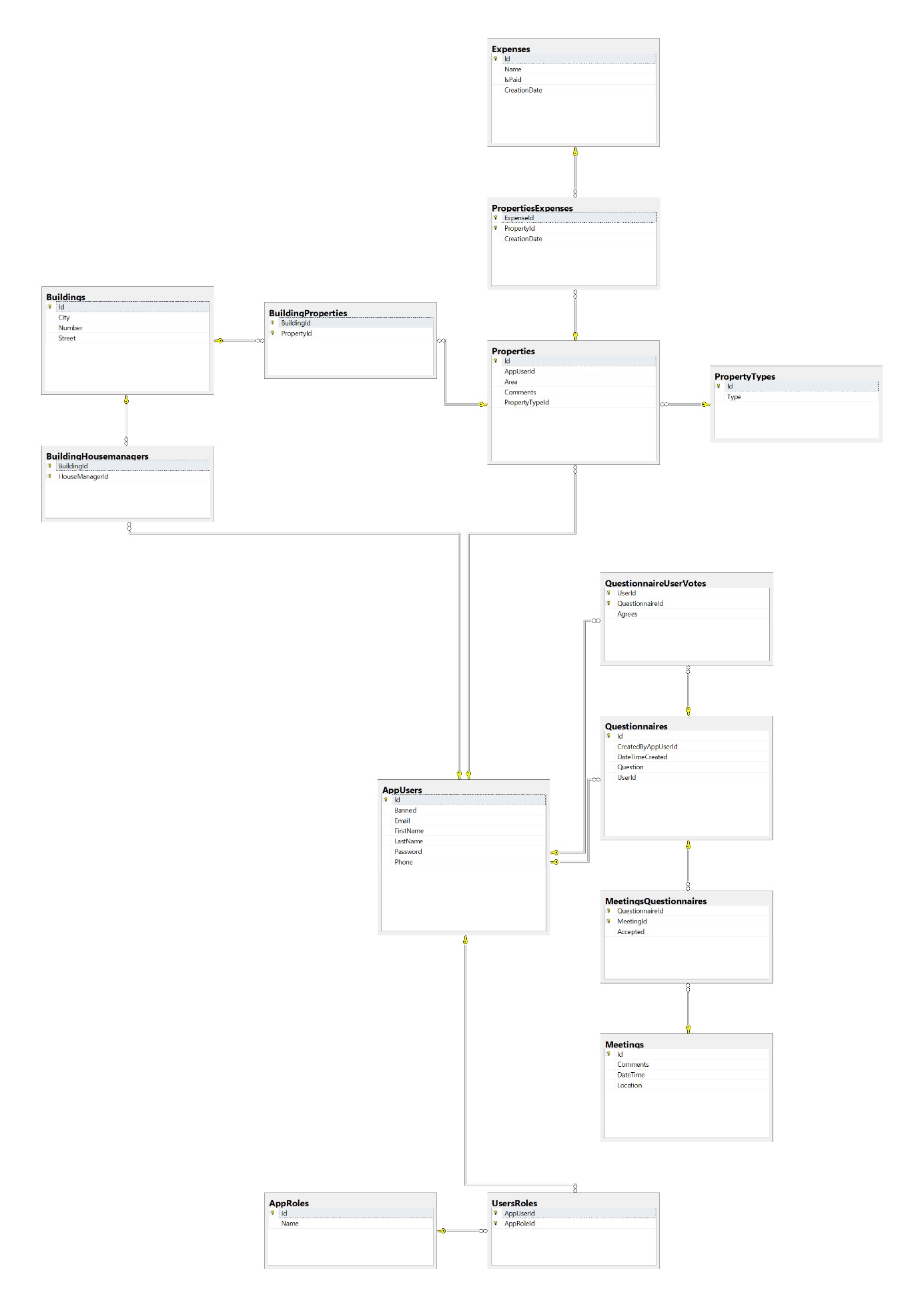
Част от менюто на софтуера. Дефинира общото събрание.

* Управлява се единствено от администраторите на сградата.
* Потребителите могат само да виждат датата на събранието и точките в него.
* Събранието може да има два статуса. Активно и Неактивно(променя се в зависимост дали датата е минала или не).   
  В неактивните събрания потребителите ще могат да виждат датата на събранието и списък с въпросите които са били гласувани и това дали те са били приети с Да или Не. В активните ще може да се вижда съответно само датата и точките, по които ще се гласува.
* В списъка със събранията те винаги ще бъдат подредени по дата на събранието,като бъдещото събрание ще бъде най-отгоре.
* За администраторите има допълнително меню , с което ще се дава възможност за регистриране на общо събрание. След това към него ще може да се „закачат“ въпроси, които са свързани с описаната по-горе функционалност за вдигане на въпроси/предложения. Въпросите ще бъдат подредени спрямо активността на потребителите.
* След завършване на дадено събрание в софтуера, администратора ще има възможност да прегледа и извади съответното в PDF файл.Така създаденото събрание ще позволява на потребителите да виждат точките,които са въвели администраторите и как са били гласувани те.

# Проектиране на базата данни и архитектурата на системата

1. Реализация на базата данни

За реализация на система за управление на релационни бази данни е избрана MSSQL версия 2012 година. На фигура 2 може да се види нейната структура.



Фигура 2

За реализация на базата и създаване на нейните компоненти е изпована технологията Entity Framework на Microsoft и нейния модел “Code First”. При този модел базата се гради въз основа на домейн модела на приложението, като първо се създават неговите класове и на база на тях чрез така наречените миграции се създават и обектите от базата. Механизма позволява отличен модел на абстракция, при който приложението не разчита на вече създадена схема на базата, а гради своя такава, базирана на домейн модела на приложението. Освновен плюс при използването на този метод а възможността активно да се променя схемата на базата, като това се базира на модела на приложението.

Примерния модел за изграждане на обектите от базата данни може да се види на фигура 3.

## CodeFirst

Фигура 3

Част от дефиницията за създаване на схемата от домейн обектите и техните връзки може да се види на последващите редове:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Property>().HasKey(p => p.Id);

modelBuilder.Entity<Property>().Property(p => p.Id).ValueGeneratedOnAdd();

modelBuilder.Entity<Property>()

.HasOne(p => p.AppUser)

.WithMany(a => a.Properties)

.HasForeignKey(p => p.AppUserId);

modelBuilder.Entity<Property>()

.HasOne(p => p.PropertyType)

.WithMany(pt => pt.Properties)

.HasForeignKey(p => p.PropertyTypeId);

modelBuilder.Entity<AppUser>(

au =>

{

au.Property(p => p.FirstName).IsRequired();

au.Property(p => p.LastName).IsRequired();

au.Property(p => p.Email).IsRequired();

au.Property(p => p.Password).IsRequired();

});

modelBuilder.Entity<AppRole>(ar => ar.Property(p => p.Name).IsRequired());

Част от т.н. миграционен код, който служи за генериране DML SQL-a, може да се види на последвщите редове.

public partial class initial : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "Buildings",

columns: table => new

{

Id = table.Column<int>(nullable: false)

.Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),

City = table.Column<string>(nullable: false),

Number = table.Column<int>(nullable: false),

Street = table.Column<string>(nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_Buildings", x => x.Id);

});

migrationBuilder.CreateTable(

name: "Expenses",

columns: table => new

{

Id = table.Column<int>(nullable: false)

.Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),

Name = table.Column<string>(nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_Expenses", x => x.Id);

});

migrationBuilder.CreateTable(

name: "AppRoles",

columns: table => new

{

Id = table.Column<int>(nullable: false)

.Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),

Name = table.Column<string>(nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_AppRoles", x => x.Id);

});

Част от дефиниция на генерираните таблици и техните релации:

## Таблици:

### Потребители

* ID
* Парола
* Име
* Фамилия
* Телефон
* Email
* Баннат потребител
* Интереси

### Видове потребители

* ID
* Роля – Администратор на софтуер,администратор на сграда,собственик на имот

### Сграда

* ID
* Град
* Улица
* Номер

### Имоти

* ID
* Вид имот от таблица видове имоти
* Размер на имот
* Особености

### Видове имоти

* ID
* Вид – апартамент, мазе, гараж, паркомясто, магазин

Имоти към сгради – описва връзката между една сграда и нейните имоти

* SoftСграда ID
* Имот ID

Потребители към имоти -Many to many таблица описваща връзката между потребител и имот и неговата роля в имота

* ID
* Имот ID от таблица Имоти
* Потребител ID от таблица потребители
* Роля на потребителя

### Общо събрание

* ID
* Дата на събранието
* Местоположение
* Допълнителна информация

### Анкети

* ID
* Въпрос/предложение
* Създател – от таблица потребители
* Дата на създаване

### Точки към общо събрание

* Общо събрание ID
* ID на анкета от Анкетите
* Приета/отхвърлена

### Анкети\_Потребители\_Гласове

* Анкета ID
* Потребител ID
* Приема/Не приема

## Задължения

* ID
* Име на задължение
* Дали е платено
* Дата на създаване

## Задължения имоти

* Задължение ID
* Имот ID
* Дата на създаване задължението

## Реализация на архитектурата

Архитектурата на приложението е изградена в тип “N-Tier” application, при която различните функционалности на програмата са обединени в различни слоеве в зависимост от критерии които ги обядиняват. Многослойната архитектура е архитектура от тип клиент-сървър, в която интерфейсът, обработката на приложения и съхранението и обработката на данни са логически разделени на отделни модули. Като приет стандарта в изграждането на софтуерни приложения модулната архитектура, а изобщо и модулното създане на кода предоставя гъвкавост. Гъвкавостта се изразява в лесното подменяне на парчета код породено от липсата на силни връзки между отделните компоненти и силната кохезия в тях. Най-разпространената форма на многослойна архитектура е трислойната архитектура. Тя осигурява модел, по който разработчиците могат да създават гъвкави приложения, които могат да бъдат използвани многократно. При разделянето на едно приложение на слоеве, разработчиците имат възможността да добавят или променят отделен слой, вместо да преработват цялото приложение. В аналогия с написано, в много сфери от ежедневието ни може да се намери такъв тип архитектура. Котанктите в една къща са силен пример в това , как модулното разработване позволява лесното заменяне на компонентите, когато има нужда от това. В текущата реализация тази архитектура е изградена посредством използването на проекти във Microsoft Visual Studio. На по-ниско ниво е използвана абстракция в лицето на C# интерфейси. Това позволява да се дефинира единствено абстракцията на функционалностите от които има нужда даден модул. Това позволява гъвкавост в използването на различни решения на база на различните словете. Конкретния брой слоеве в приложението са три:

* DAL(Data access layer) - слой за съхранение на данните. Реализира връзката с базата данни реализиран чрез иползването на ОРМ(обектно-релационнен модел) – Entity Framework. Това ползволява връзката между обектите в базата и обектите в домейна на приложението. Получава се абстракция от механизма за съхранение на данните, което позволява при необходимост да се смени СУБД без да се налага променяне другите слоеве.
* BLL(Business logic layer) – слой съдържащ бизнес логиката която изгражда основните функционалности на приложението. Свърза се със слоя за за съхранение на данните и го използва пряко за нуждите си.
* UI(User interface) – слой който предоставя визуалната репрезентация на приложенито. За реализацията на слоя е използван Модел-Изглед-Контролер архитектурния шаблон за дизайн в програмирането. Модела дава добро разделение на различните компонентите на презентационния слой. Това спомага за по-лесно поддържане и разширяване на системата. Така слоя отговорен за генерирането на html кода остава разделен от слоя, кото се грижи за това информацията от слоя на бизнес логиката да бъде предоставен към потребителя.



Фигура 4

## Особености на програмната реализация

1. Използвани технологии

При разработката на приложението Домоуправител, като основа е използвана платформата на Microsoft - .Net Core ASP.NET, конкретно MVC.

Предимствата в използването на ASP.Net Core се изразяват във възможноста софтуера да се инсталира на различни платформи като Windows, Linux, MacOS, висока производителност и това че е отворен код. Факта че платформата използва отворен код позволява на разработчика да разглежда точна имплементация на библиотеките които иползва, като това от своя страна дава възможност за събиране и разучаване на добри практики от общността.

За реализацията на приложението са приложени добрите практики в програмирането, което се обуславя с използването и прилагането на принципа на капсулацията, наследяването, полиморфизма, абстракцията и петте принципа на обектно-ориентираното програмиране дефинирани за пръв път през 2000г. – SOLID. Те предоставят възможността софтуера да се поддържа и разширява лесно с течение на времето. 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. Всичко това е част от стратегията за това, че изходен софтуерен код е гъвкав (англ. agile).

Петте принципа са следните :

* Принцип за единствена отговорност. Класът трябва да има една единствена отговорност (т.е. само промени в софтуерните спецификации могат да предизвикат промяна в спецификациите на класа). По този начин кода става модуларен разпръснат на малки парчета и всяко парче върши строго специфицирана дейност. Като пример в разработеното приложение може да се даде използването на т.н. сървиси, които със силна кохезия. В дадения сървиз функционалността му се обуславя от общ признак – Общо събрание или Домоуправители.
* Прицип отворен/затворен. Софтуерните единици трябва да са отворени за разширение, но затворени за промяна. 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, т.е. е писан от други програмисти (пример една библиотека като log4net, Entity Framework и други). Пример: в .Net на Microsoft посредством extension методите ни се дава възможност да разширяваме написаните вече и работещи функционалности въпреки, че кода е компилиран.
* Принцип на заместване на Лисков. Всеки наследник (подтип) трябва лесно да заменя всичките си базови типове. Принципът Лисков се дефинира чрез взаимозаменяемостта на обектите. Той е в действие, когато в дадена програма, обектите от тип S, могат да заменят безпроблемно обектите от тип Т, без това да намалява функционалността на програмата.
* Принцип за разделяне на интерфейсите. Много на брой малки интерфейси е по-добре от един голям общ интерфейс. 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. В разработеното приложение дадения принцип се обуславя от използваните сагрегирани интерфейси на сървисите от слоя за бизнес логика.
* Принцип на обръщане на зависимостите. Всички класове трябва да зависят от абстракции и нито един не трябва да зависи от конкретен клас. 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. Един от най-важните принцип заложен още в самото начало на разработката. Обуславя се от използването на инжктор на зависимостите предоставен от платформата - .Net Core . Реализацията става благодарение на извеждането зависимостите във всеки един клас в неговия конструктор. Дадените зависимости се задават като абстракция на външните функционалности от които има нужда дадения клас. Благодарения на това, класовете остават „развързани“ от конкретната имплементация на функционалността от която имат потребност. Иползването на инжектори предоставя късна реализация и свързване с конкретната имплементация на вече споменатата абстракция. Така кода остава модулен и независим. В зависимост от нуждите на използващия зависимостта клас инжектора може да предостави всеки път една и съща инициализация имплементация или да я инициализа при всяко извикване. Това позволява използването на единични имплементации които могат да поддържат състояние през цялото време на съществуване на програмата. Конкретен пример на принципа може да се даде от слоя за визуална презентация. Във всеки един контролер са зададени само абстракциите от които има нужда за да може да работи. По този начин в бъдеще може да се „инжектира“ различна имплементация на абстракцията, която да бъде съвсем различен сървиз със съвсем разлиен начин на функциониране. Така презентационни слой остава напълно независим от имплементацията за съхраняване на данните. При необходимост в бъдеще, имплементацията на съхранението може да бъде заменена с такава тип NOSQL или да бъде заменена с извикване на вънеш ресурс за съхранение през WCF. Поради разлика в моделите на различните слоеве а и с цел допълнителното им разделяне е използван инструмент за навръзване – Automapper.

С цел подобряване на производителността на софтуера са използвани асинхронни методи.  
Те позволяват използването на текущата нишка от други части на програмата докато изпълнението на текущото извикане не се върне в нея. Модела на ASP.NET обуславя използването на нишка от общия кюп при всяка една заявка от клиентски браузър. Взимайки предвид ,че създаването на нишки е тежка операция, която заема процесорно време и оперативна памет използването на aсинхронни операции силно оптимизира системата.

За имплементиране на всеобщите отговорности за прихващане на грешки, тяхното регистриране а и функционалноста за одитиране на операциите е използвано апектно ориентирано програмиране. Имплементацията е реализаране чрез използване на Принципа на декориране. Посредством тази методология се позволява изчистването на бизнес логката от чество повтаряния код за прихващане на грешките и одитинг на операциите. Така кода става по-четим, по-лесно се разбира основното му предназначение. Поради същността на принципа на декориране, зависимостите в компонентите остават абстрактни и не разбират за съществуването на декораторите. Самото декориране се осъществява в инжектора. Поради липсата на механизъм да дефиниране на декоратори е използвана библиотеката Scrutor на Kristian Hellang, която в същността си е разширение за инжектора на зависимости на .Net Core и конректно манипулиране на ServiceDescriptor-ите му. Изисква се първо да бъде записана конкретна имплементация на дадената абстракция, след това тя се замества при регистрация на декоратора. Предоставя се възможност за многократно декориране, като последната имплементация на дадена абстракция остава последния декоратор. Последователност на инжектиране на декораторите обуславя последователност на декориране на методите от дадения клас. Примерно използване може да се види в поседващите редове.

var collection = new ServiceCollection();

// First, add our service to the collection.

collection.AddSingleton<IDecoratedService, Decorated>();

// Then, decorate Decorated with the Decorator type.

collection.Decorate<IDecoratedService, Decorator>();

// Finally, decorate Decorator with the OtherDecorator type.

// As you can see, OtherDecorator requires a separate service, IService. We can get that from the provider argument.

collection.Decorate<IDecoratedService>((inner, provider) => new OtherDecorator(inner, provider.GetRequiredService<IService>()));

var serviceProvider = collection.BuildServiceProvider();

// When we resolve the IDecoratedService service, we'll get the following structure:

// OtherDecorator -> Decorator -> Decorated

var instance = serviceProvider.GetRequiredService<IDecoratedService>();

Пример от приложението :

services.AddSingleton<IMeetingService, MeetingService>();

services.Decorate<IMeetingService, MeetingServiceLoggerDecorator>();

services.Decorate<IMeetingService, MeetingServiceRetryDecorator>();

services.Decorate<IMeetingService, MeetingServiceExceptionHandlingDecorator>();

В приложението е изполвана библиотеката Polly, отворен код .NET, който предоставя модели и градивни елементи за отказоустойчивост и устойчивост в приложенията. Грешките при обработката и възобновяването на надеждността в случай на грешка са ахилесовата пета на много софтуерни проекти. Polly предоставя различни механизми за справяне с посочените грешки, като в имплементацията на Домоуправител, библиотеката е иползвана за заявките към базата.

Следва примерен код за имплементацията с използването на библиотеката

public class MeetingServiceRetryDecorator : IMeetingService

{

private readonly IMeetingService meetingService;

private readonly AsyncRetryPolicy policy;

public MeetingServiceRetryDecorator(IMeetingService meetingService)

{

var logger = LogManager.GetLogger(GetType());

policy = Policy.Handle<Exception>().WaitAndRetryAsync(

3,

i => TimeSpan.FromSeconds(2),

((exception, span, retry, context) =>

{

logger.Warn($"Execution failed with exception {exception.Message}. Waiting {span} before next retry. Retry attempt {retry}");

}));

this.meetingService = meetingService;

}

public async Task<List<Meeting>> GetAllMeetings()

{

return await policy.ExecuteAsync(async () => await meetingService.GetAllMeetings());

}

За одитиране на операциите и записване на грешките се използва библиотеката Log4Net.

Абстракцията на операциите предоставена от библиотеката ползволява лесното ѝ заменяне при необходимост, защото те не правят зависимост на кода който я използва към дадена конкретна имплементация на записването. Така лесно може записването във файл да се замени с записване в база данни или друг начин на запазване.



За стилизиране на интефейса е иползвана библиотеката Bootstrap. Тя предоставя инструменти за по-лесно изграждане дизайна на мрежовите страници на програмата. В допълнение към обичайните HTML компоненти, Bootstrap има много други компоненти, написани на CSS и JavaScript, проектирани така, че да улеснят разработчиците да създадат висококачествен потребителски интерфейс.

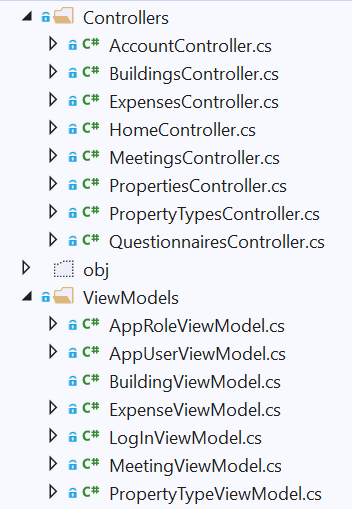
Компонентите включват: бутони, изображения, прогрес бар и съобщения. За опростяване достъпа до различните елементи на документния обектен модел, асинхронното извикване на контролерите и манупулиране на събития е иползвана библиотеката jQuery.

1. Програмна реализация

На фигура 5 е показана обобщен модел на архитектурата на програмата.



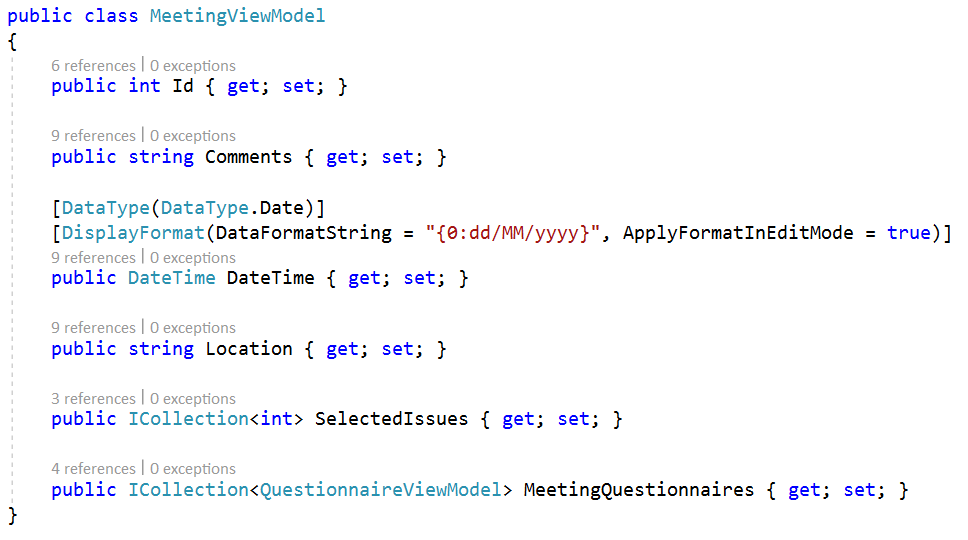
Фигура 5

В проекта на потребителския интерфейс са дефинирани различните конролери отговарящи на съответните функционалности на програмата. Account, Buildings, Expenses, Home, Meetings, Properties, PropertyTypes, Questionnaires.

Всеки един контролер се инициализира с прилежащите към него зависимости.

Всяка една единица за която се грижи съвответния контролер се дефинира с два типа модели. Единия е този използван за потребителския интерфейс т.н. ViewModel, а другата е тази която се записва в базата данни EntityModel.

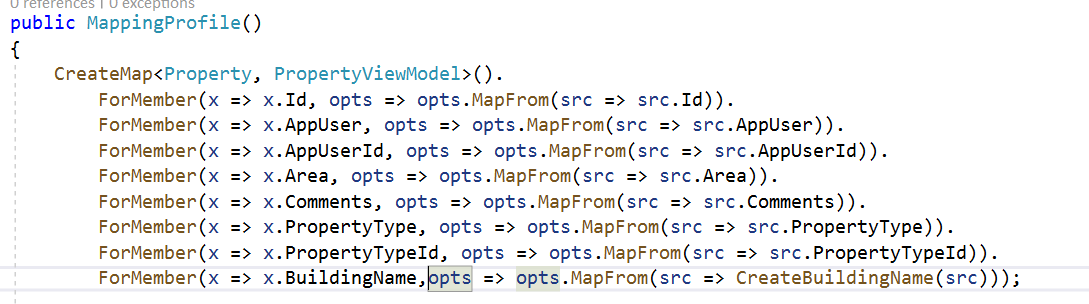
Примерен модел, който захранва потребителксия интерфейс може да се види на последващата фигура.



Активното преобразуване между двата типа модели става чрез използването на библиотеката Automapper.

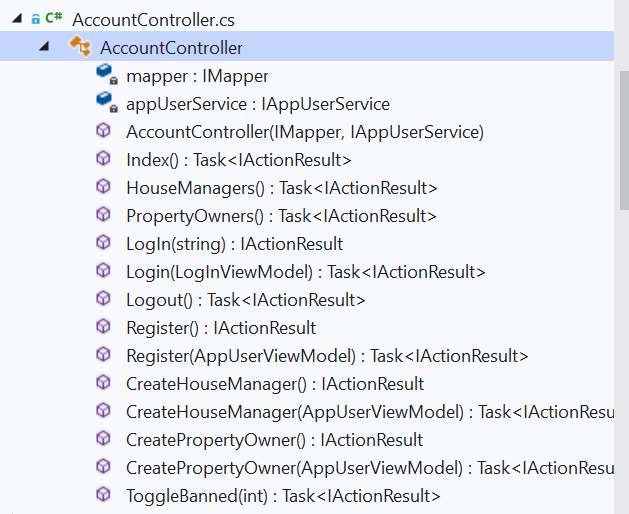
Чрез дефиниране на профил на преобразуването се изнася логиката, която превръща единия тип модел в другия без да се замърсява пространството в контролера. По този начин в контролера се изписва само визуализационната логика и кода става по-четим. Гореспоменатата функционалност се въвежда във всички контролери чрез внасянето на зависимост през абстракция, в случая чрез използването на IMapper интерфейс предоставен от библиотеката. Отново, по този начин конкретната имплементация на функционалността за преобразуване остава скрита. В последствие остава възможността да въведе нова библиотека, която имплементира интерфейса IMapper или да се използва новата през клас посредник (Proxy).

Примерна имплементация на прфила за конвертиране на моделите може да се види на следващата фигура.



Account controller е свързан с AppUserService през IAppUserService интерфейс. Потребителския контролер служи за управление услигите за интеракция с потебителите на системата. От една страна той служи за логване на потребителите на системата.

При зареждане на приложението се използва метода LogIn за зареждане на полетата за въвеждане на данните за вписване в системата – потребителско име и парола.

След вписване с системата се създава сесия на база на бисквитки на сървъра, която способства за поддържане на потребителя вписан, така се избягва постоянна необходимост от въвеждане на удостоверяващите данни от потребителя при всяка извършена опрация.

Чрез използване на усглугата за потребители се проверява дали данните въведени от потребителя при използване на формата за вписване отговарят на записаните в базата данни. Ако потребителските данни съвпадат потребителя се вписва в системата и може да продължи да използва нейните функционалности, които отговарят на потребителския профил който той има. Ако данните са неверни се изписва съобщение информиращо потребителя,че това което е въвел е грешно и ако желае да въведе отново необходимите данни.

Метода Logout служи за отписване на потребителя от системата. Така създадената сървърна сесия се разваля и за последващо използване на функционалностите на система се налага потребителя да се впише отново.

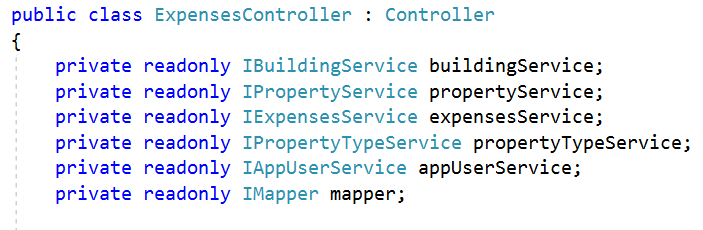
Метода Register се използва за регистриране на администратори на софтуера. Функционалността е без меню, остава скрита и се използва само при първоначална инициализация на софтуера.

Метода CreateHouseManager служи за регистриране на домоуправители, а CreatePropertyOwner за създаване на потребители, които са собственици на имоти. При изпълнението на ToggleBanned се забранява на даден потребител да създава анкети от анкетната функционалност.

Всеки един от останалите контролери управлява по сходен начин единицата с която е дефиниран.

Building контролера служи за управление на сградите в приложението. Дефинира методи за преглеждане на текущи сгради, тяхното създаване, промяна по техните данни и изтриването им.

Expenses controller служи за управление на разходите по имотите. Това включва преглеждането на същестуващите разходи, създаването на нови, изтриването им и маркирането на даден разход като платен. Контролера използва множество зависимости, инжектирани през конструктора му за да изпълни изискванията на потребителския интерфейс.



Meeting Controller служи за управление на общите събрания в сградата. Подобно на другите контролери той предоставя стандартните функции за преглеждане, създаване, промяна и изтриване. Контролера включва допълнителен метод с име GetMeetingQuests. Метода се извиква асинхронно от клиентската част с помощта на библиотеката jQuery. Метода се използва за изкарване на вписаните въпроси в нов прозорец на браузъра чрез използването на html.

Оставащите контролери – Properties, PropertyTypes, Questionnaires дефинират същите сходни методи за преглеждане, създаване, промяна и изтриране.

За конкретната визуализация на на страниците от интерфейса е иползвана технологията Razor на Microsoft, която е част от MVC.

Razor е методология създена да връща страници в RESTful , уеб базиран модел, който разделя логиката за създаването на информацията и самата ѝ репрезентация.

Към всяка едно от дефинираните операции СЧПИ(създавам, чета, променям, изтривам) в контролерите се предоставя като отделен файл с разширение .cshtml. В него чрез използване на т.н. tag helpers или html helpers се създава магията.

В сл