

**ТЕХНОЛОГИЧНО УЧИЛИЩЕ “ЕЛЕКТРОННИ СИСТЕМИ”  
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

## **ДИПЛОМНА РАБОТА**

Тема: ERP за сервизна дейност

Дипломант:

*Александър Маринов*

Научен ръководител:

*Мартин Тасков*

СОФИЯ  
2020



Дата на заданието: 15.11.2019 г.

Дата на предаване: 15.02.2020 г.

Утвърждавам: .....

/проф. д-р инж. Т. Василева/

## ЗАДАНИЕ за дипломна работа

на ученика Александър Иванов Маринов 12<sup>А</sup> клас

1.Тема: ERP за сервизна дейност

2.Изисквания:

- Responsive уеб сайт за управление на сервиз
- История на ремонтите, водене на worklog (с текст, снимки, поръчани части)
- Клиентска част за следене на ремонта и история на ремонти
- Онлайн плащане на ремонта
- Доставка с търсене на адрес и визуализация с google maps
- Генериране на pdf фактура или printer friendly страница и изпращане на email
- Интеграция със спедиторско API

3.Съдържание 3.1 Обзор

3.2 Същинска част

3.3 Приложение

Дипломант : .....

Ръководител: .....

/ Мартин Тасков /

Директор: .....

/ доц. д-р инж. Ст. Стефанова /

Александер Маринов изплати целия набор от  
необходими действия, свързани с дипломната работа  
общома известно и структурирано. Тези действия бяха  
процесуални, идентификация на малки и по-комплексни  
проблеми, изграждане на добър график на работата  
и темпо, и резултатът до момента е много добър.  
Немалък брой корекции и допълнения бяха направени  
по проекта като с това той бива приближен все  
повече до завършен вид. В хартиената част на  
проекта има доста подробно (об) описание на използва-  
ните технологии, както и графична част и примери.

Заключение: Изключително сериозно отношение към дипломната  
работа и проекта, както и мимиментацията.

За рецензент предлагам Феликс Илиев, част от  
лична по Бом. по

## УВОД

ERP представлява интегрирана софтуерна информационна система, която се грижи за планирането и управлението на всички ресурси в едно. Системата има за цел да обхване всички звена и бизнес процеси от живота на една компания.

Интегрирана система е тази, в която информацията се въвежда веднъж и без допълнителна обработка (в реално време) се отразява във всички модули и подсистеми, като става достъпна за всички потребители.

ERP замества традиционните информационни системи, като ги обединява в единна, унифицирана програма, разделена на взаимно свързани модули според отделните сектори в едно предприятие. Предназначена е за автоматизиране на управлението и отчетността.

Повечето системи от този тип разрешават имплементацията на отделни модули, покриващи един или повече бизнес процеси. В дипломната си работа съм създал ERP система, обслужваща дейностите свързани със сервизната дейност по ремонта на дефектирал уред, както и управлението на складовите наличности.

Реших да създам тази ERP система за гаранционно и извън гаранционно поддържане и ремонт на бяла и черна техника, защото след мое проучване на пазара на подобни продукти и разговори с хора от бранша се оказа, че се предлагат основно два вида софтуер – с много добри параметри и функционалност, но с висока цена или с по-ниска цена, но не достатъчно функционални или пък много тромави. Не съм разработил счетоводен модул, тъй като нямам необходимите знания по счетоводство и контрол, а и към момента се въвеждат изискванията по наредба Н-18, която урежда правилата за постоянна връзка на подобни системи с НАП.

Създаденият продукт е насочен основно към малкия и среден бизнес, тъй като считам, че той ще се търпи развитие в следващите няколко години/ особено в районите с по-малобройно население/. Модулите са изградени на основата на едни от най-новите софтуерни продукти.

Целта ми е да се автоматизира създаването на ремонтна карта, като се избегнат грешките при попълване на модел, сериен номер и продуктов код/ IMEI за дефектиралите уреди. Също така максимално опростения му, но с достатъчна функционалност за този вид дейност, дизайн ще доведе до повишаване ефективността на персонала, опростяване на административните дейности, извършвани от техниците, което от своя страна ще доведе до повишаване на ефективността им при ремонта на уредите.

Системата позволява към нея да бъдат прикачени и допълнителни модули, в зависимост от изискванията на съответния клиент/ напр. връзка със спедиторско API, визуализиране адреса на клиента, on-line плащане чрез PayPal и др./, но това ще доведе до оскъпяване на продукта като цяло.

# ПЪРВА ГЛАВА

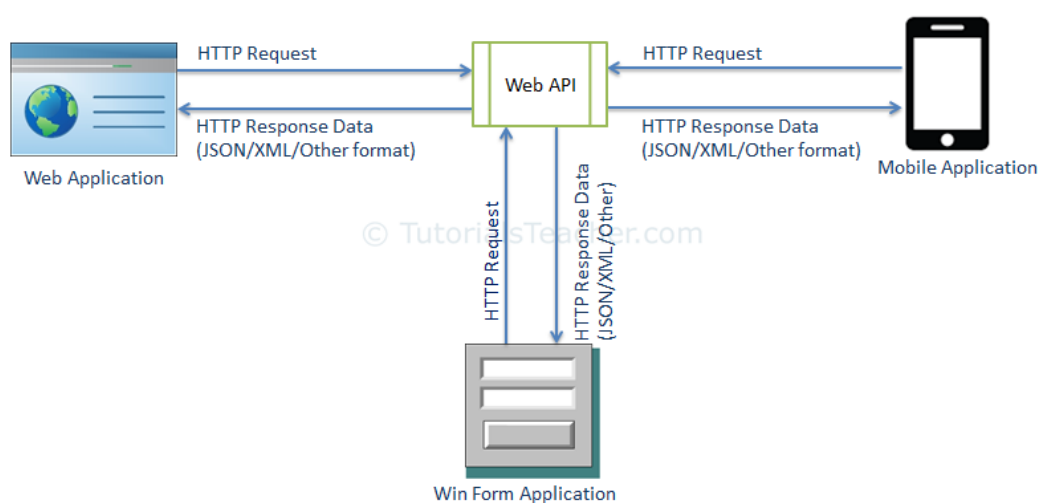
## Преглед на използваните технологии и съществуващи решения

### 1.1. Обзор на използваните технологии

#### 1.1.1. ASP.NET Web API

API (Application Programming Interface) е тип интерфейс, който има набор от функции, които позволяват достъп до специфични свойства или данни на приложение, операционна система или други сървиси. Web API както името показва, е API, което може да бъде достъпено чрез HTTP протокола.

ASP.NET Web API е extensible framework за изграждане на HTTP базирани сървиси, които могат да бъдат достъпени от различни приложения на различни платформи като уеб, уиндоус, мобилни и други. Принципа на действие е показан на фиг. 1.1.



Фиг. 1.1

#### 1.1.2. SignalR

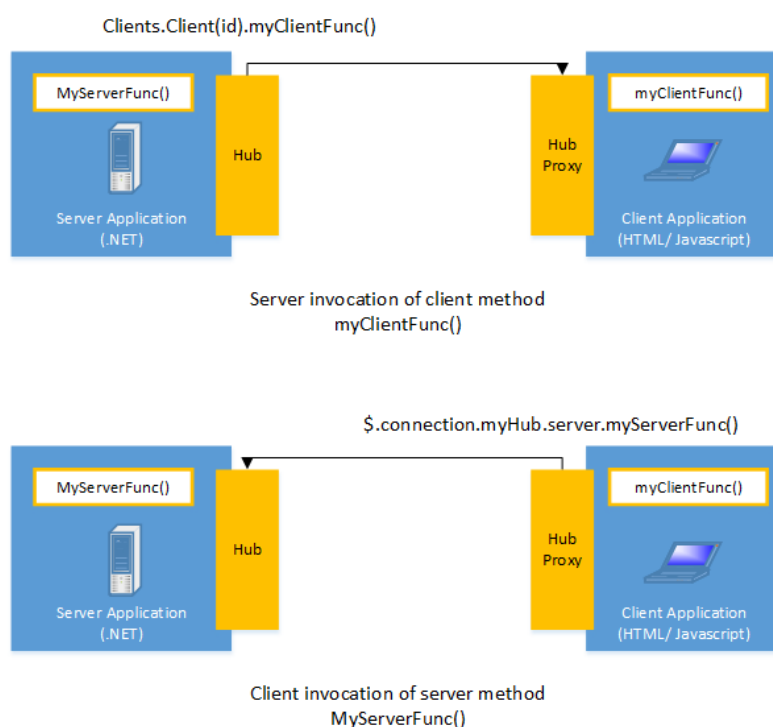
SignalR е библиотека с отворен код, която опростява процеса по добавяне на функционалности към дадено приложение в реално време (real – time web functionality (RTWF)) и е направена, за да бъде използвана от разработчици на ASP.NET. RTWF е способността на даден сървър да

качва ново съдържание в приложението и то да става достъпно веднага за всички свързани клиенти вместо клиента да прави нов request за данни.

SignalR може да бъде използвана за добавяне на всякакъв тип RTWF към вече съществуващо ASP.NET приложение. Най – често давания пример за това е чат приложение, но може да бъде направено много повече. Всеки път, когато потребител обнови уеб страница, за да види нови данни, или страницата имплементира long polling, за да вземе нови данни, това е кандидат за използването на SignalR.

SignalR също така предоставя възможността за напълно нов тип уеб приложения, при които се налага обновявания много често от сървъра като например real-time gaming.

SignalR предоставя просто API, което създава server-to-client remote procedure calls (RPC), което извиква JavaScript функции в клиентските браузъри от server-side .NET code. SignalR също така съдържа API за управление на връзката и връзка между групи. Принципът на действие е показан на фиг. 1.2.



Фиг. 1.2

SignalR handles connection management automatically, а това позволява на данните да бъдат разпрострени до всички клиенти едновременно, както и до специфичен клиент. Връзката между сървъра и клиента е постоянна, за разлика от класическата HTTP връзка, която трябва да се подновява за всяка комуникация.

SignalR поддържа “server push“ functionality, при която кодът в сървъра може да бъде извикан от кода при клиента в брауъра, използвайки RPC вместо request-response модела, който може да се види във всеки сайт днес. Тази библиотека е вградена по default в по – долу описаната технология.

### 1.1.3. Blazor

Blazor е фреймуърк за създаване на интерактивен client-side web UI, използвайки .NET: създаване на интерактивни UIs, използвайки C# вместо JavaScript (JS); server-side and client-side логика на приложението, написана на .NET; render UI като HTML и CSS и поддръжка за много брауъри, включително мобилни.

Използването на .NET за client-side web development предоставя следните предимства:

- Писане на C# вместо на JavaScript
- Използва съществуващата .NET екосистема от .NET библиотеки
- Share app logic across server and client
- Предимствата на производителността, надежността и сигурността на .NET.

Типовете проекти, които могат да се изграждат с този фреймуърк, са два: Blazor Server App и Blazor WebAssembly, но първо ще бъде обяснено какво е Razor Component.



### 1.1.3.1. Razor Components (RC)

Blazor приложенията са базирани на компоненти. Компонент в Blazor елемент от UI като страница, изскачащо прозорче или форма за попълване на данни.

Компонент са .NET класове, компилирани в .NET assemblies, които дефинират flexible UI rendering logic, обработват user events, могат да бъдат вложени и преизползвани.

Класът на компонентите обикновено се записва под формата на страница за маркиране на Razor с разширение на файла .razor. Компонентите в Blazor официално се наричат Razor Components. Razor е синтаксис за комбиниране на HTML маркиране с C # код, предназначен за производителност на разработчиците. Razor Pages и MVC също използват Razor. За разлика от Razor Pages и MVC, които са изградени около модела на заявка – отговор, компонентите се използват специално за логиката и състава от страна на клиента от потребителския интерфейс.

Фиг. 1.3 демонстрира компонент (Dialog.razor), който може да бъде вложен в друг компонент:

```
<div>
  <h1>@Title</h1>

  @ChildContent

  <button @onclick="OnYes">Yes!</button>
</div>

@code {
  [Parameter]
  public string Title { get; set; }

  [Parameter]
  public RenderFragment ChildContent { get; set; }

  private void OnYes()
  {
    Console.WriteLine("Write to the console in C#! 'Yes' button was selected.");
  }
}
```

Фиг. 1.3

Съдържанието на тялото на диалога (ChildContent) и заглавието (Title) се предоставят от компонента, който използва този компонент в своя потребителски интерфейс. OnYes е метод на C #, задействан от събитието onclick на бутона.

Blazor използва естествени HTML тагове за UI композиция. HTML елементите определят компоненти, а атрибутите предават стойности на свойствата на компонента.

В следващия пример компонентът Index използва компонента Dialog. ChildContent и Title се задават от атрибутите и съдържанието на елемента <Dialog> (фиг 1.4).

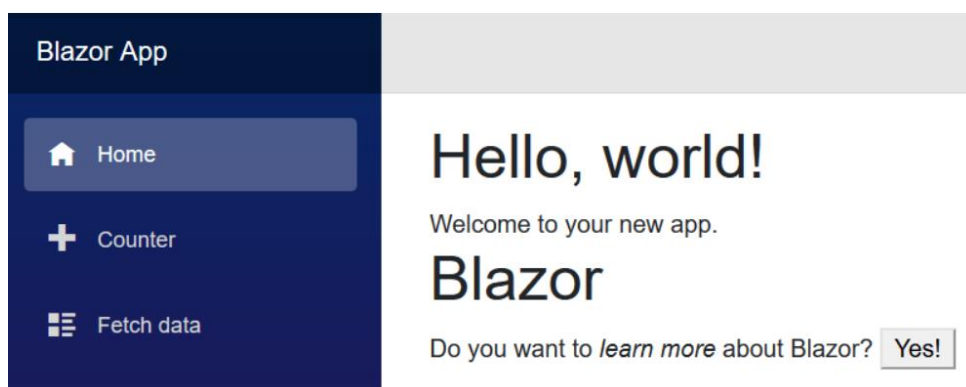
```
razor Copy
@page "/"
<h1>Hello, world!</h1>

Welcome to your new app.

<Dialog Title="Blazor">
    Do you want to <i>learn more</i> about Blazor?
</Dialog>
```

Фиг. 1.4

Диалогът се визуализира, когато родителят (Index.razor) бъде достъпен в браузъра (фиг. 1.5)



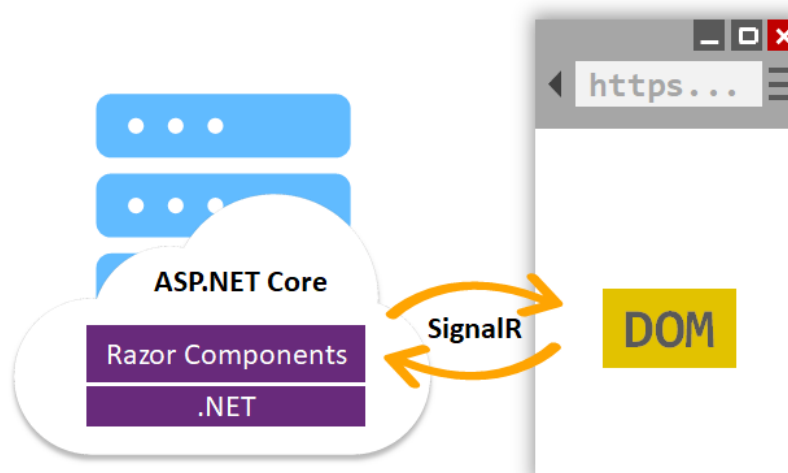
Фиг. 1.5

### 1.1.3.2. JavaScript interop

Приложения, при които се налага използването на third-party JavaScript libraries и достъп до APIs в браузъра, компонентите си взаимодействат с JS. Компонентите имат способността да използват всяка библиотека или API, което JS е способен да използва. C# код може да извиква JS код, както и JS код да извиква C# код.

Типовете проекти, които могат да се изграждат с този фреймуърк, са два: Blazor Server App и Blazor WebAssembly.

### 1.1.3.3. Blazor Server App (BSA)

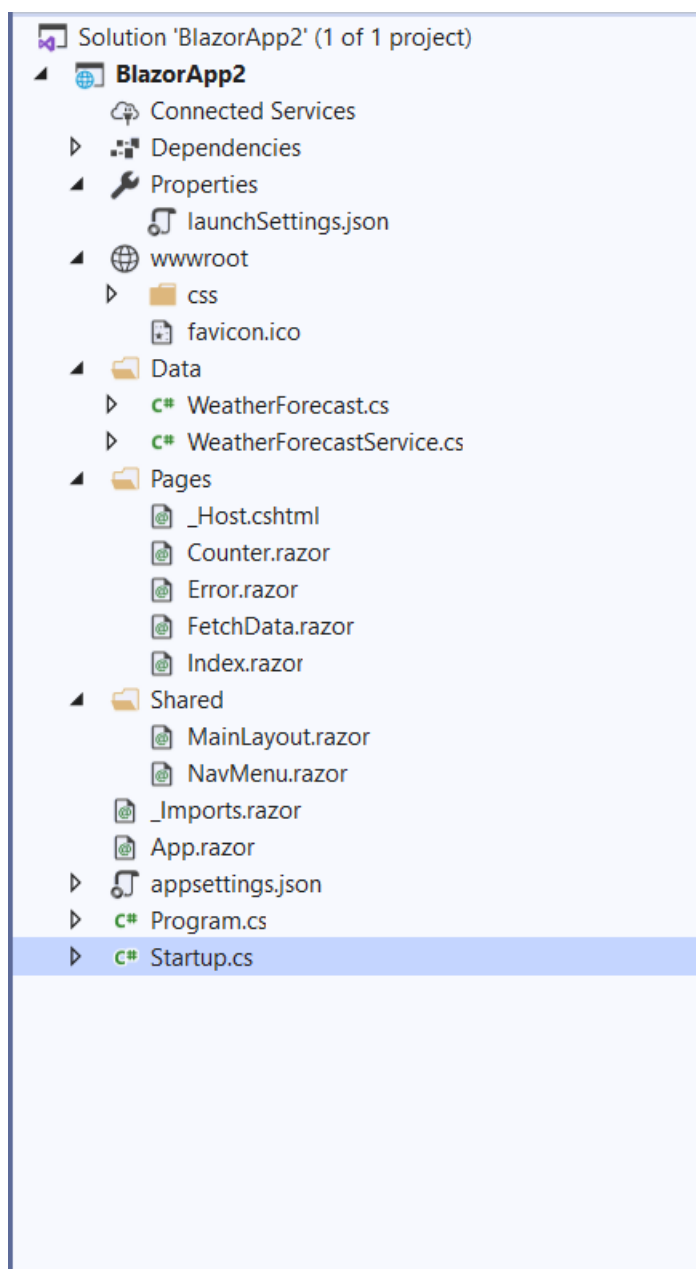


Фиг. 1.6

На фиг. 1.6 е показано на как работи едно BSA приложение. Blazor отделя component rendering logic от начина на прилагане на актуализациите на UI. Blazor Server предоставя поддръжка за хостинг на Razor компоненти на сървъра в ASP.NET Core приложението. Актуализациите на UI се обработват чрез SignalR connection.

The runtime обработва изпращането на UI събития от брауъра до сървъра и прилага актуализации на UI, изпратени от сървъра обратно към брауъра след стартиране на компонентите.

Връзката, използвана от BSA за комуникация с брауъра, се използва и за обработка на JS interop calls.

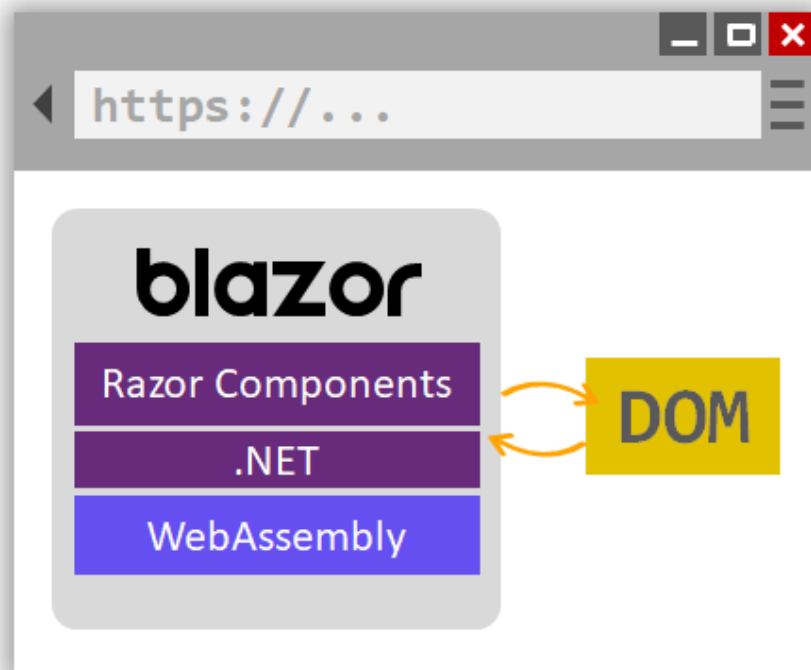


Фиг. 1.7

На фиг. 1.7 е показано каква е структурата на едно BSA приложение. То е разделено на следните части:

- **wwwroot** – в нея се съдържа всички css, js, icons и т.н., които са използвани в приложението
- **Data** – там се съдържат всички модели и сървиси, а сървисите трябва да се добавят в **Startup.cs**.
- **Pages** – в нея се съдържат всички страници, което е BSA приложение е възможно да има и те са .razor файлове, които са обяснени как работят по – нагоре. **\_Host.cshtml** е Razor Page, в която се добавят всички връзки към css и js файлове.
- **Shared** – в нея се съдържат файловете **NavMenu.razor** и **MainLayout.razor**. В **NavMenu.razor** се пишат всички елементи на менюто. В **MainLayout.razor** се пише как ще изглежда основната страница на приложението.
- **Startup.cs** и **Program.cs** – съответно съдържа конфигурацията на цялото приложение и компилира цялото приложение.

#### 1.1.3.4. Blazor WebAssembly (Wasm)



Фиг. 1.8

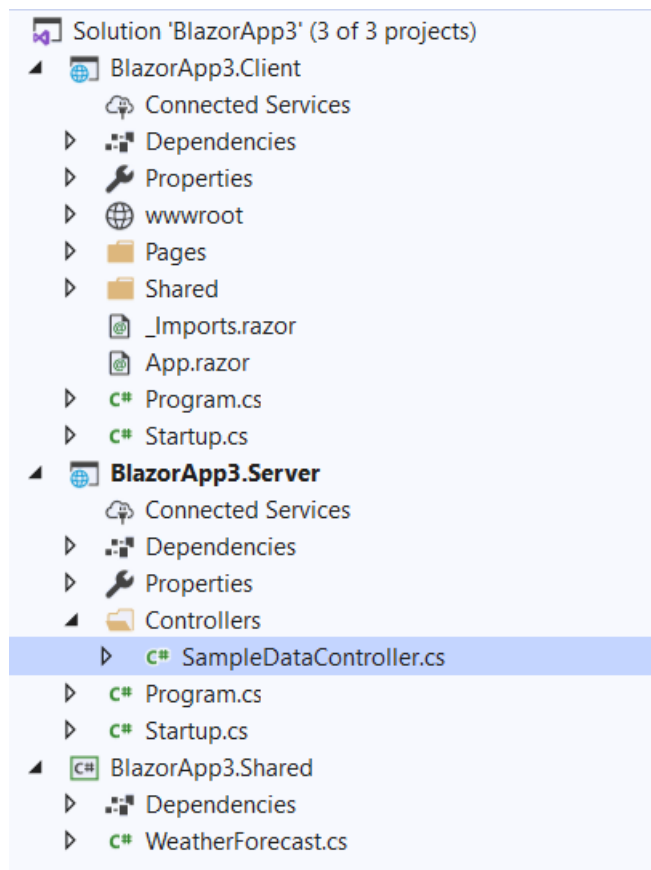
На фиг. 1.8 е показано как работи едно Wasm приложение. Blazor WebAssembly е SPA framework за изграждане на интерактивни client-side уеб приложения с .NET. Blazor WebAssembly използва отворени уеб стандарти без плъгини или преобразуване на код и работи във всички съвременни уеб браузъри, включително мобилни браузъри.

Изпълнението на .NET код в уеб браузърите става възможно от WebAssembly (съкратено Wasm). WebAssembly е компактен bytecode формат, оптимизиран за бързо изтегляне и максимална скорост на изпълнение. WebAssembly е отворен уеб стандарт и се поддържа в уеб браузъри без plugins.

Кодът на WebAssembly може да има достъп до пълната функционалност на браузъра чрез JS, наречен оперативна съвместимост на JavaScript (или JS interop). .NET код, изпълнен чрез WebAssembly в браузъра, работи в JavaScript sandbox на браузъра със защитите, които sandbox осигурява срещу злонамерени действия на клиентската машина.

Когато Wasm приложението е компилирано и работещо в браузъра:

- C# и Razor файловете се компилират в .NET assemblies.
- The assemblies and the .NET runtime се свалят в браузъра.
- Wasm стартира .NET runtime и конфигурира runtime, за да зареди монтажите за приложението. Wasm runtime използва JS interop за манипулация DOM дървото и извиквания на API в браузъра.



Фиг. 1.9

На 1.9 е показано каква е структурата на едно Wasm приложение. То се състои от три модула:

- Client – в него се съдържат целия front-end на приложението;
- Server – в него се съдържа цялата back-end логика;
- Shared – в него се съдържа всички модели на приложението.

В .NET Core 3.1 Wasm не е налично като тип проект, защото дава грешки, които са от .NET Core 3.

#### 1.1.4. Okta



Това е корпоративна услуга за identity management, създадена за използване в облака, но съвместима с on-premises приложения. С Okta IT може да се управлява достъпа на всеки потребител/служител до всяко приложение или устройство. Okta работи в облака, на сигурна, надеждна и обстойно одитирана платформа, която се интегрира дълбоко с on-premises, директории и identity management systems.

#### 1.1.5. Google Firebase

Google Firebase е Google – backed софтуер за разработка на приложения, който дава възможност на разработчиците да разработват приложения за iOS, Android и Web. Firebase предоставя инструменти за проследяване на аналитиката, отчитане и коригиране на сризове на приложения, създаване на маркетинг и експеримент с продукти.

Firebase предлага редица услуги, включително:

- Cloud Firestore – Cloud Firestore е flexible, scalable база данни за разработка на мобилни устройства, уеб и сървър от Firebase и Google Cloud Platform. Подобно на Firebase Realtime Database, тя поддържа данните синхронизирани между клиентските приложения чрез realtime listeners и предлага офлайн поддръжка за мобилни и уеб, за да може да се изграждат отзивчиви приложения, които работят независимо от латентността на мрежата или интернет



връзката. Cloud Firestore също предлага безпроблемна интеграция с други продукти на Firebase и Google Cloud Platform, включително Cloud Functions.

### **1.1.6. Google Cloud Platform (GCP)**

#### **1.1.6.1. GCP**

GCP е набор от Cloud Computing сървиси, предлагани от Google. Платформата предоставя различни услуги като изчисляване, съхранение, работа в мрежа, големи данни и много други, които работят на същата инфраструктура, която Google използва вътрешно за своите крайни потребители като Google Търсене и YouTube.

GCP предлага голям набор от функции, включително:

##### ➤ Compute Services

- Compute Engine – инфраструктура като услуга за стартиране на виртуални машини на Microsoft Windows и Linux. Това е компонент на платформата Google Cloud, която е изградена на същата инфраструктура, която управлява Google търсачката, YouTube и други услуги.

##### ➤ Storage Services

- Google Cloud Storage – интернет услуга за съхранение на файлове в Интернет за съхранение и достъп до данни в инфраструктура на GCP. Услугата съчетава ефективността и мащабируемостта на Google Cloud with advanced security and sharing capabilities.

## **ВТОРА ГЛАВА**

Аргументация за използваните технологии, описание на функционалните изисквания и структура на приложението

### **2.1. Функционални изисквания към приложението**

#### **2.1.1. Responsive уеб сайт за управление на сервиз**

Responsive уеб сайт означава да е с дизайн, който е подходящ както за компютри, така и за мобилни телефони.

#### **2.1.2. История на ремонтите, водене на worklog (с текст, снимки, поръчани части)**

Историята на дадена ремонтна карта означава да се записва всяка промяна в състоянието – промяна на статус, обновяване на данни по даден ремонт и други.

#### **2.1.3. Клиентска част за следене на ремонта и история на ремонти**

Клиентската част е с цел клиентът да провери на какъв етап от ремонта е неговия уред.

#### **2.1.4. Онлайн плащане на ремонта**

Това е свързано с извънгаранционните ремонти и е с цел да улесни клиента.

#### **2.1.5. Доставка с търсене на адрес и визуализация с google maps**

Това изискване е с цел улеснение на клиента, за да може ако няма възможност да отиде до сервиз да вземе уреда си, да бъде доставен до адрес.

#### **2.1.6. Генериране на pdf фактура или printer friendly страница и изпращане на email**

Това изискване е с цел да не се допускат грешки при попълването на данни в протоколите, които се използват в сервизната дейност. Пращането на email на клиента е с цел ако клиентът промени съдържанието на даден протокол, да се разбере, че клиентът е предал негово копие, а не оригинала, пратен от сервиза.

### **2.1.7. Интеграция със спедиторско API**

Това изискване е с цел изграждането на транспортен модул.

## **2.2. Аргументация за използваните технологии**

### **2.2.1. Blazor**

Blazor е SPA фреймуърк за разработване на уеб приложения. В настоящата дипломна работа е избран поради следните причини:

- Напълно съвместим е с .NET libraries и .NET tooling. За разлика от client-side Blazor (Wasm), който се предлага с непълни възможности за отстраняване на грешки, грешките на server-side Blazor могат да бъдат отстранявани по абсолютно същия начин като всяко друго ASP.NET приложение. По същия начин, всички други стандартни инструменти, които могат да бъдат използвани с всякакви други видове .NET приложения, могат да се използват с server-side Blazor.
- Използва точно същия синтаксис като client-side Blazor(Wasm). Въпреки, че Blazor от страна на сървъра може да не е най-добрият избор за определени сценарии, когато client-side Blazor е полезен, все още първоначално приложението да се напише на server-side Blazor, за да се преодолеят ограничените възможности за отстраняване на грешки на client-side Blazor. След като приложението е готово за release, кодът от приложението се

копира в проекта, който е clients-side Blazor. Синтаксисът позволява да бъдат написани class libraries, които ще работят с server-side Blazor или client-side Blazor.

- Малък размер на клиентски компоненти. Когато става дума за изтегляне на клиентски компоненти на server-side Blazor, те ще включват само сравнително малко количество HTML и JavaScript, нито един от които не е задължително да бъде написан.
- Works with thin clients. Няма значение през кой браузър се използва Server-side Blazor. Клиентът получава само стандартни HTML и JavaScript, които работят практически навсякъде. Следователно server-side Blazor е съвместим с почти всеки браузър.

### **2.2.2. Web API**

В настоящата дипломна работа ASP.NET Web API е избрано, защото Blazor WebAssembly(Wasm) все още е в beta версия и е все още в период на тестване. Затова чрез Blazor server app и Web API се симулира Wasm.

### **2.2.3. Google Firebase**

Firebase е добър избор, защото като продукт на Google често подлежи на обновяване и коригиране на проблеми, за някои услуги може да се ползва безплатно, като цяло работи доста прилично и просто, заради това много типове проекти са с Firebase. Firebase е избран, защото се намира в облачното пространство, а с навлизането на квантовите компютри няма да се налага миграция към облачните пространства.

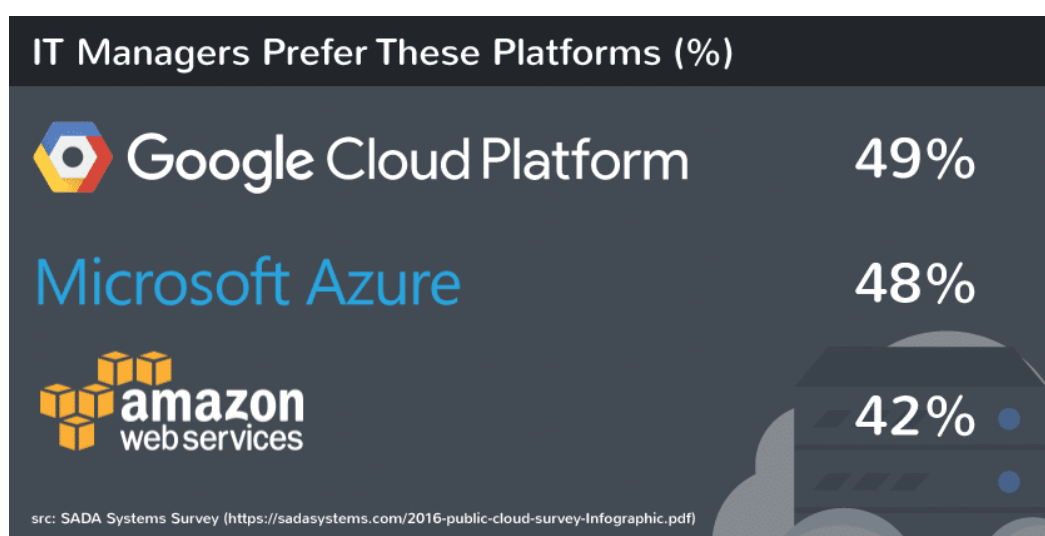
#### **2.2.3.1. Cloud Firestore**

Google Firebase Cloud Firestore е пуснат официално през 2018 година като част от Google Firebase Platform. Firestore е schema less база данни, а това означава, че няма строги правила за дефиниране на

съхраняваните данни. Всеки ред от базата данни може да има различни колони, а това е много голямо предимство за голяма част от разработчиците на приложения. Във Firestore данните се пазят в документи, които представляват двойки ключ-стойност, а от своя страна това предлага бързо търсене. Firestore е хоризонтално нарастваща база данни, а това означава нарастване чрез добавяне на повече машини в ресурса на приложението и всеки node (машина) съдържа част от данните, а това не налага закупуването на повече сървъри. Разделянето на node-ове на базата данни става автоматично и е скрито от разработчиците.

### 2.3. Google Cloud Platform(GCP)

GCP е избрана, защото според проучване на SADA Systems, 84% от IT мениджърите използват public cloud infrastructure (фиг. 2.1). Някои от най-големите компании като Cisco прогнозироват, че до 2020 година 92% от трафика ще представлява трафик в облачното пространство. Освен това, някои бележити компании вече използват GCP – Spotify, HSBC, Home Depot, Snapchat, HTC, Best Buy, Philips, Coca Cola, Domino's, Feedly, ShareThis, Sony Music, and Ubisoft.



Фиг. 2.1

Част от предимствата на GCP пред конкурентите са:

- Better Pricing Than Competitors
- Live Migration of Virtual Machines
- State of the Art Security
- Redundant Backups

### 2.3.1. Better Pricing Than Competitors

Google bills in minute-level increments (with a 10-minute minimum charge), а това означава, че се плаща само за използваното време и се предлагат намаления в цените ако приложението е използвано продължително и е натоварено без предварително ангажиране. След 1-месечно използване на VM, се получава отстъпка. Това го прави подходящ за startup компании и за IT enterprises, които искат да намалят разходите си. Сравнения между цените са показани на фиг. 2.2 и 2.3.

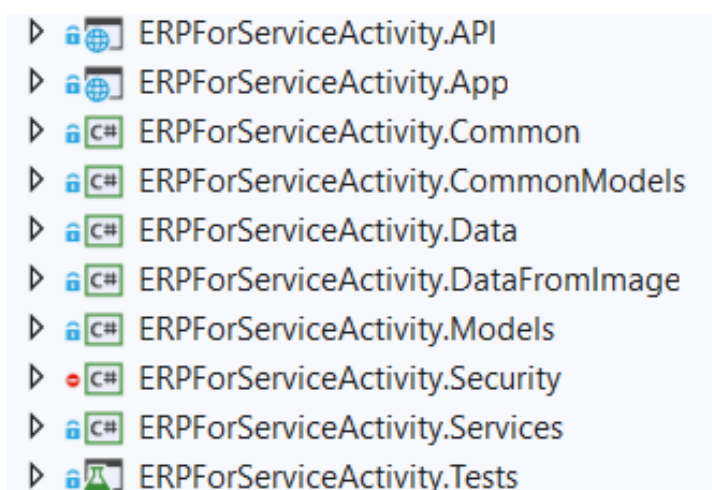
AWS – Virginia	GCP – South Carolina (East)
c4.4xlarge	Custom
– 16 CPUs	– 16 CPU
– 30 GB RAM	– 30 GB RAM
=====	=====
+ \$613.42 instance	+ \$357.25 instance (sustained-use)
500GB SSD EBS Volume	667GB SSD PD Volume
=====	=====
+ \$62.50 disk	+ \$113.39 disk
+ \$1040.00 IOPS (16K IOPS)	+ \$000.00 IOPS (20K IOPS)
=====	=====
= \$1715.92/month on-demand	= \$470.64/month
= \$1102.50/month 3yr (\$8,580.00)	

Фиг. 2.2

Azure – East	GCP – South Carolina (East)
D5 V2	Custom
– 16 CPUs	– 16 CPU
– 56 GB RAM	– 56 GB RAM
=====	=====
+ \$870.48 instance	+ \$419.43 instance (sustained-use)
512GB Premium Volume	667GB SSD PD Volume
=====	=====
+ \$73.22 disk (2.3K IOPS)	+ \$113.39 disk
X 8 RAID0 IOPS (18.4K IOPS)	+ \$000.00 IOPS (20K IOPS)
=====	=====
= \$1602.68/month	= \$532.82/month

Фиг. 2.3

## 2.4. Структура на приложението



Фиг. 2.5

На фиг. 2.5 е показано как изглежда структурата на приложението.

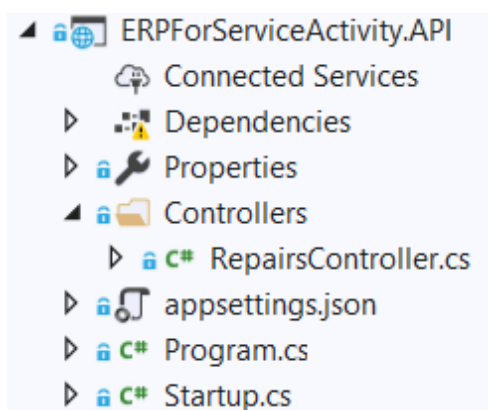
Състои се от десет модула:

- ERPForServiceActivity.API – използван е за обработка на заявки
- ERPForServiceActivity.App – главния модул на приложението
- ERPForServiceActivity.Common – съдържа всички използвани константи

- ERPForServiceActivity.CommonModels – съдържа всички binding и view модели
- ERPForServiceActivity.Data – конфигурация на връзката с базата данни
- ERPForServiceActivity.DataFromImage – взимане на информация от снимка
- ERPForServiceActivity.Models – съдържа моделите, които си комуникират с базата данни
- ERPForServiceActivity.Security – съдържа цялата защита на приложението, която няма да бъде описана в настоящата дипломна работа
- ERPForServiceActivity.Services – съдържа логиката за действията в приложението
- ERPForServiceActivity.Tests – съдържа всички тестове, написани за приложението

#### 2.4.1. ERPForServiceActivity.API

Структурата е показана на фиг. 2.6.



Фиг. 2.6

Съдържа следните папки и файлове:

- Controllers – съдържа всички контролери, които са написани и които се използват.

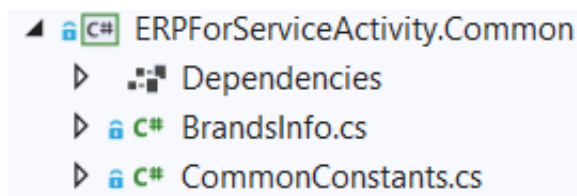


- appsettings.json – съдържа всички настройки, които изискват данни в този файл
- Program.cs и Startup.cs – файлове съответно, който пуска целия модул, и който съдържа настройки на приложението

**2.4.2. ERPForServiceActivity.App** – структурата е като на Blazor Server App от първа глава.

### 2.4.3. ERPForServiceActivity.Common

Структурата е показана на фиг. 2.7.

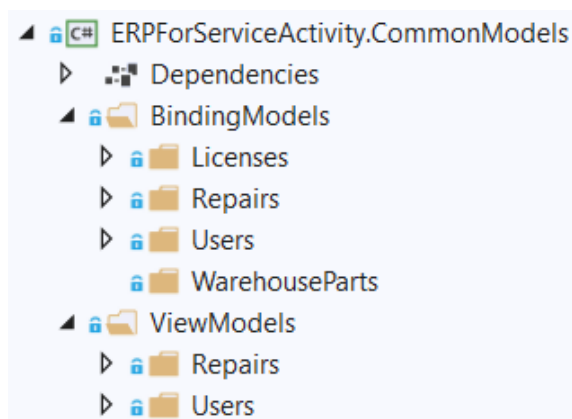


фиг. 2.7

В него се съдържат всички константи, които са използвани в приложението. BrandsInfo.cs съдържа поле за наличните марки като бива използван като лист от стрингове, който се визуализира като падащо меню при създаването на поръчка.

### 2.4.4. ERPForServiceActivity.CommonModels

Структурата е показана на фиг. 2.8.

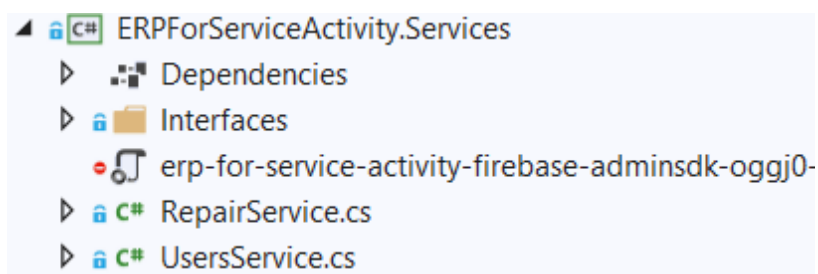


Фиг. 2.8

В този модул се съдържат всички модели, които са свързани с попълването на форми, промяна на данни и визуализиране на данни. Разделен е на две части – binding models и view models. В binding models се съдържат всички модели, които се добавят или променят данни. Във view models се съдържат всички модели, които се използват за визуализиране на данни.

#### 2.4.5. ERPForServiceActivity.Services

В този модул се съдържат всички класове съдържащи бизнес логиката на приложението. В папката Interfaces се съдържат всички интерфейси, които са имплементирани от съответните класове. За да може да се използват сървисите или класовете, всеки от тях имплементиращ съответен интерфейс, трябва да са добавени в главния модул на приложението. Структурата е показана на фиг. 2.9.



Фиг. 2.9

## **ТРЕТА ГЛАВА**

### **Същност и реализация на приложението**

#### **3.1. Какво е рекламация**

Всяко несъответствие на стоката в сравнение с договора за покупко-продажба, възникнало през периода на гаранцията (IW) или след изтичането му (OW).

IW несъответствията може да се покриват от гаранционните условия /ремонт по гаранция/, но е възможно и да не се покриват /негаранционен ремонт/.

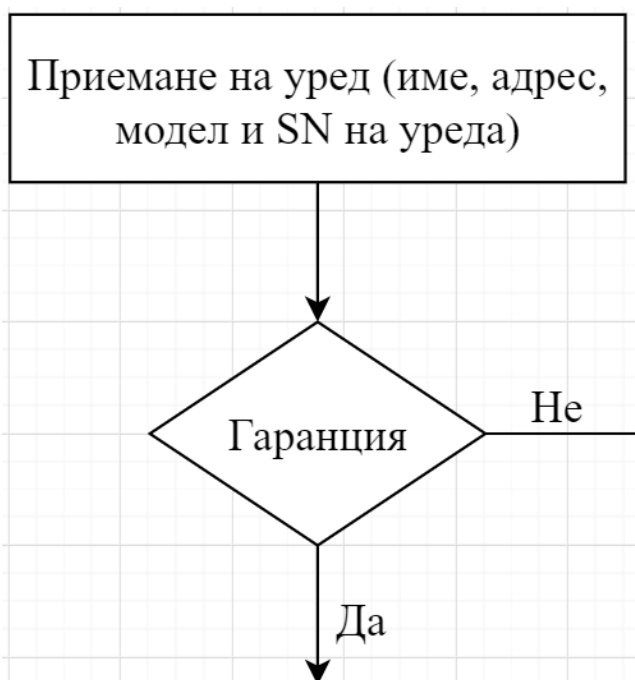
#### **3.2. Приемане на уред**

Първата стъпка е приемането на уред в оторизиран сервиз. За да се приеме един уред, са необходими следните данни:

- Име, телефон и адрес на клиента – предоставят се, защото сервиза трябва да осъществява връзка с клиента, ако се налага посещение в дома, доуточняване детайли по рекламацията или известяване на клиента за завършване на ремонта.
- Модел и сериен номер на уреда – тези данни се предоставят, защото само чрез тях е възможно поръчването на части за уреда, като и отчитането на ремонта към производител.

#### **3.3. Проверка дали уредът е в гаранция или не**

За да се провери дали един уред е в гаранция, трябва да се предоставят гаранционна карта и касов бон, които са доказателство, че уредът е купен на посочената от клиента дата. Това е показано на фиг. 3.1.



Фиг. 3.1

### 3.3.1. В гаранция

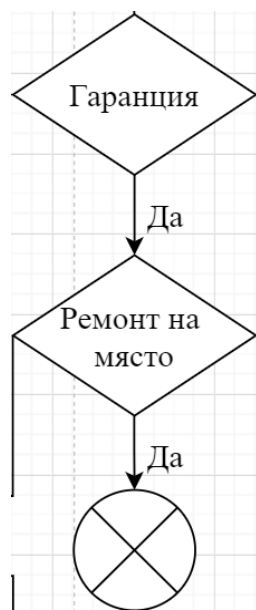
Ако се установи, че уредът е в гаранция, той трябва да бъде върнат на клиента в рамките на 30 дни от датата на приемането му. Процесът по обработка на този тип ремонти е посочен в по-долу описаните стъпки.

#### 3.3.1.1. Посещение на адрес

Ако ще има посещение на адрес, първо трябва да се уговори дата и час на посещението. Когато се отиде на адрес, най – важното е да се провери дали дефекта е гаранционен или не . Ако е възможно дефекта да се отстрани на място, ремонта се приключва. Ако не е, уреда се транспортира до сервизната база за допълнителна диагностика и поръчка на необходимите за ремонта части. Това е показано на фиг. 3.2 и фиг. 3.3.



фиг. 3.2



Фиг. 3.3

### 3.3.1.2. Диагностика на уреда

Диагностика на уреда се прави, когато няма посещение на адрес или когато няма да се прави ремонт на място. Отново се проверява дали уредът е в гаранция. Ако е, се прави допълнителна диагностика. Ако не е в гаранция, се издава протокол за отказ гаранция и уредът се ремонтира като извънгаранционен.

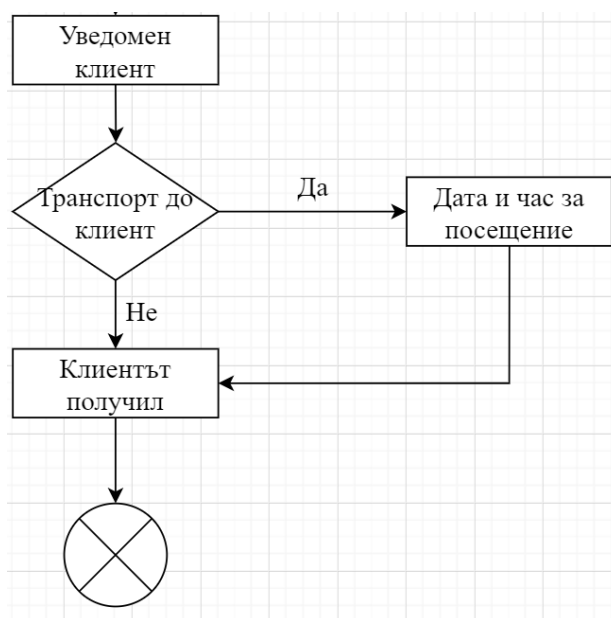
### 3.3.1.3. Части по ремонт

След диагностика се установява дали за ремонта на уреда са необходими части или не са. Ако не е нужно тоест нужно е запояване на кабелче или качване на фабричния софтуер за уреда, директно се уведомява клиентът, че ремонтът по уреда е приключил. Ако е необходима част, първо се проверява дали нужната част я има в наличността на склада, се изписва по този ремонт и след извършване на необходимите ремонтните дейности, се уведомява клиентът, че ремонтът е завършен. Ако необходимата част я няма в складова наличност, се поръчва към външен доставчик. След пристигането на частта, тя се завежда в склада на

сервиза, след което е възможно да бъде изписана по даден ремонт и след това се извършват необходимите ремонтни дейности. Ако уредът вече има три гаранционни ремонта и това е четвърта основателна рекламация уреда не се ремонтира, а се издава протокол за замяна и ремонтът приключва.

#### 3.3.1.4. Уведомяване на клиента за приключил ремонт

След приключването на всички ремонтни дейности по поръчката, се уведомява клиентът, че уредът е ремонтиран. След това се пита дали клиентът желае уредът му да бъде доставен до дома му. Ако желае доставка, трябва да уточни дата и час на посещение и се доставя до клиента. Ако не желае доставка, това означава, че клиентът ще дойде до сервиза. И в двата случая дали с или без доставка, клиентът получава уреда си и поръчката приключва. Това е показано на фиг. 3.3.



Фиг. 3.3

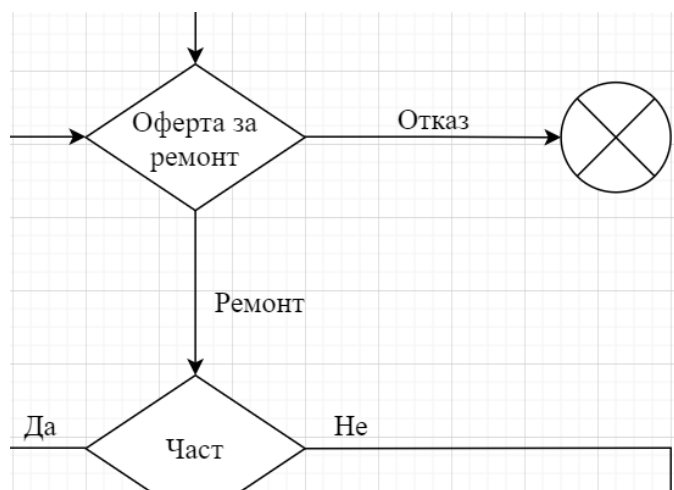
#### 3.3.2. Извън гаранция

Гаранцията на уред отпада, когато възникналия дефект не се покрива от гаранционните условия определени от производителя или е

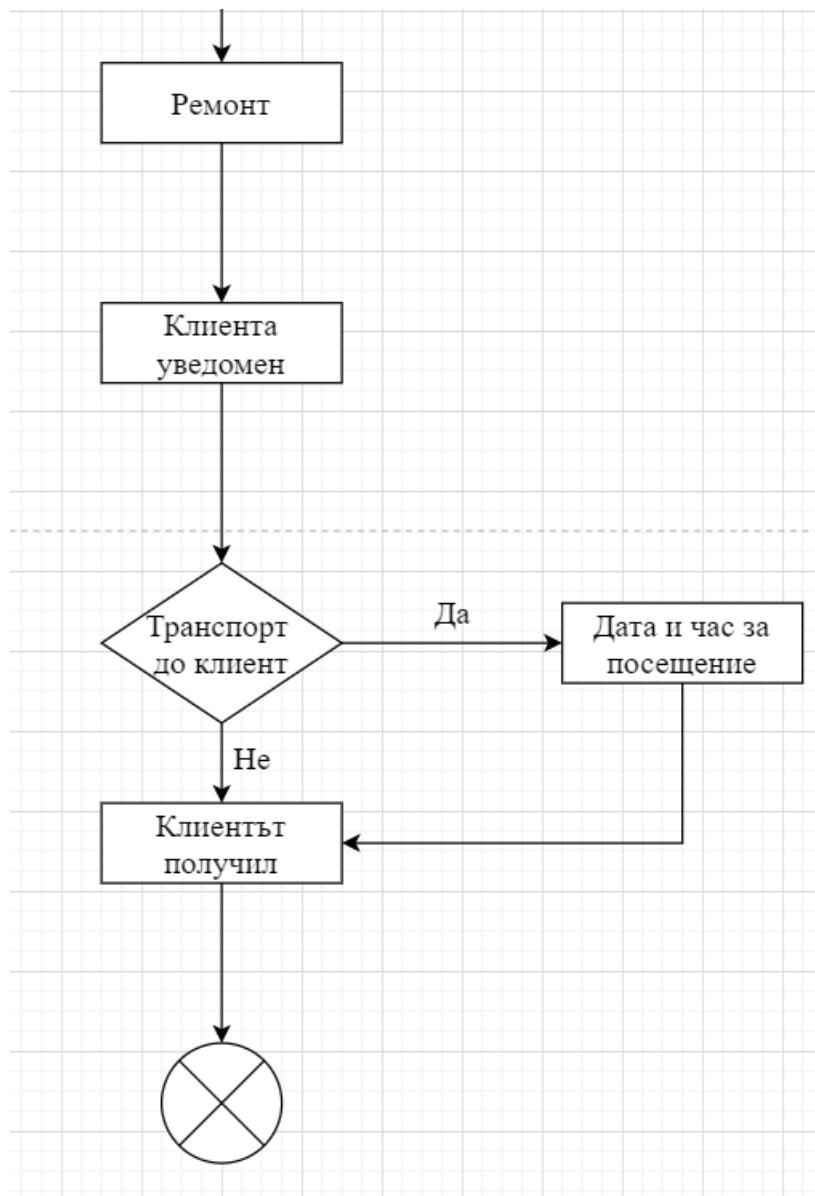
изтекъл гаранционния срок. Гаранцията може да се отхвърли на следните етапи от процеса на ремонт:

- При приемането на уреда в сервиза;
- При посещение на място след проверка на документите;
- След направена диагностика и установени причини за повредата.

Във всеки от горепосочените случаи процедурата по провеждане на извънгаранционен ремонт е една и съща. Започва се с предлагане на оферта за извънгаранционен ремонт. Ако клиентът откаже, уредът се връща на клиента и поръчката приключва. Ако се съгласи, започват ремонтните дейности по уреда. Проверява се дали е необходима част за поръчката и ако не е, се извършват ремонтните дейности, след което клиентът се уведомява, че ремонтът е приключил. След това се уточнява с клиента дали иска доставка на уреда и ако желае, се определя дата и час на доставка, а ако не желае доставка, клиентът трябва да дойде до сервиза и да си вземе уреда. След това поръчката приключва. Ако обаче е необходима част по поръчката, се изпълнява процедурата по поръчване на част от гаранционния ремонт. Процедурата е показана на фиг. 3.4 и фиг. 3.5.



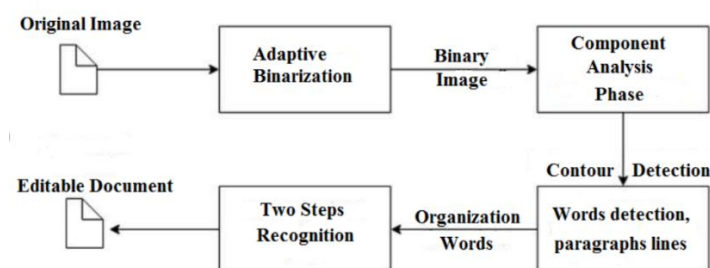
Фиг. 3.4



Фиг. 3.5

### 3.4. Принципът на работа на Google Cloud Vision API

Принципът на работа на Google Cloud Vision API е сходен с този на Tesseract Optical Character Recognition (TOCR). Той е показан фиг. 3.6.



Фиг. 3.6



Намирането на думи е направено чрез организиране на текстови линии в петна, а линиите и регионите се анализират за фиксиран висок или пропорционален текст. Текстовите линии се разбиват на думи по различен начин според вида на разстоянието между знаците. След това разпознаването протича като процес с два прохода. В първия пропуск се прави опит за разпознаване на всяка дума от своя страна. Всяка дума, която е задоволителна, се предава на адаптивен класификатор като данни за обучение. След това адаптивният класификатор получава шанс за по-точно разпознаване на текст по-надолу в страницата.

### **3.5. Обяснение принципа на работа на PayPal API**

PayPal предлага REST API за нови интеграции. Тези API използват HTTP методи, структура на RESTful крайни точки, протокол OAuth 2.0 и полезни натоварвания във формат JSON.

PayPal REST API използва протокола OAuth 2.0 за to authorize calls. OAuth е отворен стандарт, който много компании използват, за да осигурят сигурен достъп до защитени ресурси.

Когато е създадена sandbox или на live REST API приложение, PayPal генерира набор от клиентски идентификатор на OAuth 2.0 и секретни идентификационни данни за sandbox or live environment. Когато бъде осъществено извикване с маркер за достъп, set the Authorization header to these credentials for the environment in which you're making the call.

В замяна на тези идентификационни данни, сървърът за оторизация на PayPal връща вашия маркер за достъп в полето access\_token:

```
{  
  "scope": "scope",  
  "access_token": "Access-Token",  
  "token_type": "Bearer",  
  "app_id": "APP-80W284485P519543T",
```

```
"expires_in": 31349,  
"nonce": "nonce"  
}
```

Този маркер се включва като на преносител в заглавието за упълномощаване със схемата за удостоверяване на Bearer в обажданията на REST API, за да докажете вашата самоличност и да защитите ресурсите си. Тази заявка включва:

```
curl -v -X GET  
https://api.sandbox.paypal.com/v1/invoicing/invoices?page=3&page_size=4  
&total_count_required=true \  
-H "Content-Type: application/json" \  
-H "Authorization: Bearer Access-Token"
```

Access tokens имат ограничен живот. Полето `expires_in` съдържа броя секунди, след което маркерът изтича. Например маркер за достъп със стойност на изтичане на 3600 изтича за един час от момента на генериране на отговора.

За да се открие, access token изтича, написаният код трябва да обработва следните две неща:

- Да се следи стойността на `expires_in` в token response.
- Трябва да обработва HTTP 401 unauthorized. The API endpoint issues this status code when it detects an expired token.

Access token-а може да се използва, докато не изтече. След това трябва да се генерира нов token.

## 3.6. Реализация на приложението

### 3.6.1. Дейности, свързани с поръчки или ремонтни карти

#### 3.6.1.1. Създаване на поръчка

Формата за създаване на поръчка е показана на фиг. 3.7.

The form is divided into two main sections: 'Customer data' and 'Unit data'. The 'Customer data' section includes input fields for Name, Address, and Phone Number, a checkbox for 'Going to address', a text area for 'Defect by customer', a button for 'Add equipment', and another text area for 'Other information'. Below these is a large area for 'Other images' with an upload icon. The 'Unit data' section includes two dropdown menus for 'Select Brand', a 'Model' input field, 'S/N' and 'Product code/IMEI' input fields, a 'Bought at' date picker and a 'WC number' input field, a 'Warranty period' dropdown set to '0', and a 'Dealer/Shop' input field. Below these is a large area for 'S/N image' with an upload icon. At the bottom center are 'SAVE' and 'CANCEL' buttons.

Фиг. 3.7

В секцията Customer data се съдържат всички полета, които са свързани с данни за клиента. Описанието на всяко поле е представено в табл 3.2.

Текстово поле	Обяснение
Name	Двете имена на клиента
Address	Адреса на клиента
Phone number	Телефонът на клиента
Going to address	Дали ще има посещение на адрес
Defect by customer	Описание на проблема според клиента
Add equipment	Каква е комплектацията на уреда
Other information	Забележки по ремонта

Other images	Допълнителни снимки (гаранционна карта, касов бон, снимки на дефекта)
--------------	---

Табл 3.2.

В секцията Unit data се съдържат всички данни за уреда. Обяснение за всяко поле е представено в табл. 3.3.

Текстово поле	Обяснение
Brand	Марката на уреда
Type	Типът на уреда
Model	Моделът на уреда
S/N	Серийният номер на уреда
Product code/IMEI	Продуктовият код на уреда (IMEI за мобилни телефони)
Bought at	Датата на покупка на уреда
WC number	Номер на гаранционната карта на уреда
Warranty period	Продължителност на гаранцията
Dealer/Shop	Откъде е закупен уредът
S/N image	Снимката на стикера на уреда

табл. 3.3

При отварянето на страницата се взима най-новодобавеното Id на ремонтната карта и се увеличава с 1, за да няма дублиращи се идентификационни номера на ремонтните карти. След това се инициализират следните променливи:

- `bool showResultFromOcr` – използва се за отваряне на прозореца с резултатите от четенето на информация от снимка (първоначална стойност = false);
- `string imagePath` – пътеката до снимката на стикера на уреда (първоначална стойност = `string.Empty ("")`);
- `AddRepairBindingModel repair` – моделът, чрез който ще бъдат предадени данните на модела, който ще комуникира с базата данни. Съдържа следните полета:

- CustomerName – съдържа информация за името на клиента.
- CustomerAddress – съдържа информация за адреса на клиента.
- CustomerPhoneNumber – съдържа телефонния номер на клиента
- DefectByCustomer – какво е проблемът според клиента
- GoingToAddress – дали ще има посещение на адрес
- InWarranty – дали уредът е в гаранция
- ApplianceBrand – марка на уреда
- ApplianceType – тип на уреда (печка, пералня, хладилник, телевизор и т.н.)
- ApplianceModel – съдържа данни за модела на уреда.

Например модел на LG TV съдържа следните данни:

- Първите две цифри – размера на екрана в inches
- Трети знак (буква) – резалюция на дисплея
- Четвърти знак (буква) – година на производство
- Пети знак (цифра) – серия
- Шести знак (цифра) – модел в серията
- Седми знак (цифра) – информация за дизайн
- Осми знак (цифра или буква) – екстри или цвят

Пример: 55UK750V е 55 inches TV, с UHD резолюция, произведен през 2018, горна серия, 5 модел в серията, черен цвят, с тунер DVB - T/T2/C/S/S2

OLED TV имат различна абривиатура.

- ApplianceSerialNumber – серийния номер на уреда. Например серийния номер на уред LG съдържа следните данни:
  - Първи знак (цифра) – година на производство
  - Втори и трети знак (цифри) – месец на производство
  - Четвърти и пети знак (букви) – завод производител
  - Шести и седми знак (букви) – регионално предназначение
  - Последни пет (букви или цифри) – пореден производствен номер

Пример: 712WRVA2X615 е произведен през 2017 декември, в Полша, кодовете на заводите се знаят само от производителя, пореден производствен номер(2X615).

- ApplianceProductCodeOrImei:
  - Product code – съдържа данни за модела, региона, за който е предназначен уред (Европа, Азия и т.н.), цветови и други характеристики.
  - IMEI (International Mobile Equipment Identity) – уникален номер за идентифициране на мобилни телефони, който се изписва на екрана като се въведе \* # 06 # на клавиатурата или заедно с друга системна информация в менюто за настройки на операционните системи. В него се съдържа информация за модел, модел код и

сериен номер. Тази информация може да бъде достъпена само през официалния сайт на производител само ако има оторизиран достъп.

- ApplianceEquipment – каква е окомплектацията на уреда. Например за телевизор – дистанционно, стойка, самият телевизор, user manual.
  - BoughtFrom – магазин или дилър, от който е купен уредът.
  - WarrantyCardNumber – номер на гаранционна карта на уреда.
  - WarrantyPeriod – продължителност на гаранцията на уреда, даваща се в месеци.
  - BoughtAt – дата на покупка на уреда.
  - AdditionalInformation – забележки към ремонта.
  - ModelSNImage – полето, което съдържа информация за снимката на стикера на уреда.
  - OtherImages – масив, който съдържа информация за всички останали качени снимки.
- ResultFromOCRBindingModel result – моделът, чрез който ще бъдат предавани данните от взимането на информация от снимка. Съдържа следните полета:
- ApplianceBrand – марката на уреда.
  - ApplianceType – типът на уреда.
  - И други полета.

Когато бъде засечено, че е качена снимката на стикера на уреда, се прави заявка към API контролера, за да бъдат взети необходимите данни

като се извиква метода GetData(Фиг. 3.8 и Фиг. 3.9), като това се прави като се пуска нова нишка и се взима резултата от изпълнението ѝ.

```
public async Task<ResultFromOCRBindingModel> GetData() {  
    Environment.SetEnvironmentVariable(  
        "GOOGLE_APPLICATION_CREDENTIALS",  
        CommonSecurityConstants.PathToGoogleCloudJson);  
  
    ResultFromOCRBindingModel result =  
        new ResultFromOCRBindingModel();  
}
```

Фиг. 3.8

```
ImageAnnotatorClient client =  
    await ImageAnnotatorClient.CreateAsync();  
  
Image image = await Image  
    .FromFileAsync(  
        @"E:\ALEKS\Images\pictures-diploma-project\1.jpg");
```

Фиг. 3.9

Това се прави така с цел бързодействие. Методът Task.Run създава нишка, която се добавя в опашката на чакащите задачи, минава през thread pool и се записва в опашката на изпълнените задачи. В метода, който се извиква при прашане на заявка към контролера, в който се случва следното:

- Създава се обект от типа ResultFromOCRBindingModel, който ще бъде върнат след изпълнението на целия метод.
- Взимат се регех-ите, което се прави като се предостави марката на уреда и типа на уреда.
- Създава се обект от типа ImageAnnotatorClient чрез ImageAnnotatorClient.Create(), защото класът е абстрактен,



който е предоставен от библиотеката `Google.Cloud.Vision.V1`, която е използвана за взимане на информация от снимка.

- Взима се пътеката на снимката на стикера чрез метода `GetImagePath(string, int)`, който от посочената директория взима пътеката на файла на посочения индекс.
- Създава се обект от типа `Google.Cloud.Vision.V1.Image` чрез `Google.Cloud.Vision.V1.Image.FromFile`, който очаква като аргумент пътеката до файла.
- Създава се обект от типа `ReadOnlyList<EntityAnnotation>` чрез `client.DetectText`, очакващ снимката, от която ще взима резултатите, а те се записват в новосъздадения обект.
- Итерираща се през колекцията от предната точка и се гледа дали има съвпадение с по-горе посочените `regex`-и.
- Резултатният обект се връща.

След като приключи изпълнението на нишката, се променя стойността на `showResultFromOcr` на `true`. Показва се диалог, на който се проверява за грешки като например заместването на 0 с O. След като се провери от човешка гледна точка дали има грешки, се натиска бутона `Ok`, който затваря диалога. След това се извиква метода `StateHasChanged`, който е наследен от `ComponentBase` класът, който е наследен от всеки новосъздаден `Razor Component`. Това се прави с цел да бъде `real-time`.

Ако бъде засечено, че се на полето `Other images` има промяна, всички файлове се прехвърлят на полето `OtherImages` в класа `AddRepairBindingModel`.

Когато се натисне бутона `Save`, се прави `Http Post` заявка към контролера, при което се отива на метода `UploadRepair` в контролера. В този метод се извиква метода от сървиса за ремонтите, на който се подава два аргумента – първият е името на сервиза, а вторият – обект от типа

AddRepairBindingModel, който съдържа цялата информация за поръчката. Методът в сървиса изпълнява следното:

- Създава се обект от типа FirestoreDb, който ще бъде използван за връзка и комуникация с базата данни.
- Взима се Id на документа, на който който ще името на сервиза, съвпада с името на името от подадения аргумент.
- Създава се обект от типа CollectionReference, който ще посочи къде ще се качат данните.
- Създава се обект от типа Repair, който е модела, който ще комуникира с базата данни, като в конструктора се подава втория аргумент на метода.
- Стойността на Id на ремонта се променя на Id на ремонта на подадения ремонт.
- След това се започва асинхронна транзакция, в която се обновява най-новото Id на ремонта за кой сервиз и на съответната колекция се качва ремонта.

### 3.6.1.2. Показване на всички ремонти

```
List<RepairViewModel> repairs = new List<RepairViewModel>();

QuerySnapshot snapshot = await db
    .Collection("service-repairs")
    .GetSnapshotAsync();

Parallel.ForEach(snapshot.Documents, ds => {
    if (ds.Exists) {
        RepairViewModel model = NewModel(ds);

        repairs.Add(model);
    }
});
```

Фиг. 3.10

Извиква се метода `GetAllRepairs` (Фиг. 3.10), който визуализира всички ремонти, се взимат всички, чието име е равно на подадения `path variable serviceName`. Методът прави следните действия:

- Създава се обект от типа `FirestoreDb` чрез `config.GetFirestoreDb`, в който се настройва и се взима инстанция на `FirestoreDb`. Взима се `Id` на документа чрез метода `GetDocumentId.Result`, защото методът е асинхронен и връща `Task<T>`, където `T` е типът, който връща.
- Създава се `List<RepairViewModel>`, който ще бъде върнат и ще съдържа всички резултати.
- След това се създава `Query`, което взима всички документи в колекцията за ремонтите от базата данни и след това се изпълнява.
- Извиква се метода `Parallel.ForEach` (от `System.Threading.Tasks`), който приема за аргументи колекция и ламбда функция.
- Списъкът се връща подреден по `Id` на ремонта във възходящ ред.

След като се вземат, се визуализират на екрана. При натискане на бутона `Edit` се отваря цялата ремонтна карта. Функционалностите, които могат да се използват са следните: промяна на всяко едно от полетата и запазване на промените (бутон `Save`) и генериране на PDF документ (бутон `Generate PDF`). При натискане на бутона `Generate PDF` се извиква метода `CreatePdf` (Фиг. 3.11), който генерира PDF файл в посочена от клиента папка.

```

public async void CreatePdf(int id, string type, string notes) {
    Encoding.RegisterProvider(
        CodePagesEncodingProvider.Instance);

    if(type == "Received") {
        await CreateAcceptPdf(id, notes);
    }
    else if(type == "Repaired") {
        await CreateResignPdf(id, notes);
    }
}

```

Фиг. 3.11

При необходимост от използване на резервна част, се натиска бутона Add part, който извиква метода AddRequestedPartToRepair (Фиг 3.12).

```

public async void AddRequestedPartToRepair(string partNumber, int id) {
    FirestoreDb db = connection.GetFirestoreDb();
    QuerySnapshot snapshot = db
        .Collection("repair-parts")
        .WhereEqualTo("RepairId", id)
        .GetSnapshotAsync()
        .Result;

    if(snapshot.Count == 0) ...
    else ...
}

```


Фиг 3.12

### 3.6.2. Дейности, свързани с склада на сервиз

#### 3.6.2.1. Добавяне на част в склада

Формата за добавяне е показана на фиг. 3.13.

### Add part to warehouse

Part number	Substitute Part
Model	Invoice number
Description	Date of receiving invoice 
Brand	Price 0

**ADD PART TO WAREHOUSE**

Фиг. 3.13

Обяснението на всяко поле е представено в табл. 3.4.

Поле	Обяснение
Part number	Идентификационния номер на частта
Model	За кой модел става частта
Description	Описание на частта
Brand	Марка на частта
Substitute part	Част заместител
Invoice number	Номер на фактурата на частта
Date of receiving invoice	Дата на получаване на фактурата
Price	Цена на частта

При отваряне на страницата се създават следните променливи:

- Обектът part от типа AddWarehousePartBindingModel – използва се за пренасяне на данни от формата към back-end. Обяснение за всяко поле е представено в табл.3.4.

Табл. 3.4

Поле	Обяснение
PartNumber	Уникален идентификационен номер за части. Всеки

	производител си има негова идентификация.
Availability	Наличност в склада
Model	Списък, съдържащ за кои модели става частта.
Description	Описание на частта
Brand	Марка на частта
SubstituteParts	Списък, съдържащ всички нейни заместители
Price	Цена на частта
Invoice	Номер на фактура, с която е пристигнала
InvoiceDate	Дата на издаване на фактура
ReceivedDate	Дата на получаване на частта

- substitutePart от типа string – използва се за взимане на част заместител

При натискане на бутона Add part to warehouse се изпълнява метода AddPartToWarehouse (Фиг 3.14), който наличността на частта се променя на 1, датата на завеждане на частта в склада се променя на DateTime.UtcNow, защото например ако частта е поръчана на 20.12.2019г., има време на транспорт, което отнема определено време. part.InvoiceDate се променя на същата дата, но в UTC формат. Списъкът на частите заместители се инициализира на new List<string>, както и part.Model и в part.Model се добавя въведения модел. Проверява се дали е въведена част заместител и ако да, се добавя в списъка с частите заместители. Последно се прави Http Post заявка към контролера за частите в склада.

```

public async void AddWarehousePart(
    AddWarehousePartBindingModel model,
    string serviceName) {

    FirestoreDb db = connection.GetFirestoreDb();
    CollectionReference colRef = db
        .Collection("warehouse-parts")
        .Document("bcyvKBFBWE6DxnviQ1Kn")
        .Collection("parts");

    WarehousePart newPart = new WarehousePart(model);
    QuerySnapshot partsWithSamePN = await colRef
        .WhereEqualTo("PartNumber", newPart.PartNumber)
        .GetSnapshotAsync();

    if(partsWithSamePN.Count == 0) ...
    else ...
}

```

Фиг 3.14

След като се засече, че е направена заявка, се извиква метода AddPart (Фиг. 3.15), който приема два параметъра - path variable serviceName, която определя за кой сервиз се отнася и AddWarehousePartBindingModel, който съдържа цялата информация за частта. В него се извиква метода от сървиса за добавяне на част, който приема двата по-горе посочените параметъра. В метода в сървиса се изпълняват следните действия:

- Създава се инстанция на FirestoreDb, която ще служи за връзка с базата данни
- Създава се инстанция на CollectionReference, която ще се качват в базата данни
- Създава се инстанция на WarehousePart, която ще бъде качена в базата данни
- Създава се инстанция на QuerySnapshot, където се пише.

```

[HttpPost("add-part/{serviceName}")]
public void AddPart(
    [FromBody] AddWarehousePartBindingModel newPart,
    string serviceName) {
    service.AddWarehousePart(newPart, serviceName);
}

```

Фиг. 3.15

### 3.6.2.2. Вземане на всички складови части

При инициализация на страницата, която ще визуализира всички части, налични в склада, се прави HTTP GET заявка (Фиг 3.16), която извиква от сървиса метода, който ще вземе всички складови наличности, който прави следните действия:

- Създава се инстанция на FirestoreDb.
- Създава се списък от WarehousePartViewModel, в който ще бъдат добавени всички резултати.
- Създава се query, което ще взима всички части, чиято наличност е по-голяма или равна на 1.
- Извиква се метода Parallel.ForEach, който стартира необходимия брой нишки като във всяка от тях се прави следното:
  - Конвертира резултата към WarehousePartViewModel чрез метода ConvertDsToViewModel.
  - Добавя частта в резултатния списък.
- Списъкът се връща.



```
[HttpGet("get-parts/{serviceName}")]
public async Task<List<WarehousePartViewModel>>
    GetWarehouseParts(string serviceName) {
    return await service.GetAllParts(serviceName);
}
```

Фиг 3.16

### 3.6.3. Процеси, свързани със справки в сервиза

Основната функция на справките е осигуряването на частична информация, свързана с конкретен процес от ремонта и отчета му към производителя. За целта се създава клас BaseCheckup, който ще съдържа следните полета:

- ServiceName – име на сервиза
- BeginDate – начало на периода на справката
- EndDate – края на периода на справката

#### 3.6.3.1. Справка за труда на техниците

За да бъде реализирана тази справка, се създава клас TechnicianCheckupViewModel, който наследява от BaseCheckup, и който съдържа следните полета:

- Name – име на техника
- Labor – цена на труда на техник

Методът, който ще взима резултата от тази справка, ще прави следното:

- Създава се списък от TechnicianCheckupViewModel, в който ще бъдат добавени всички резултати.
- Взимат се всички ремонти от базата данни.

- Групира се по име на техник и се смята за всеки техник неговия труд.
- Списъкът се връща.

### **3.6.3.2. Справка по статус**

Целта на тази справка е да се следят дейностите, предстоящи в рамките на конкретен ден. Създават се класовете RepairStatusCheckupViewModel, който ще съдържа полетата RepairId (Id на ремонта) RepairStatus (статус на ремонта), GoingToAddress(дали ще има посещение на адрес), UnitModel(модел на уреда), CustomerAddress(адрес на клиента), CustomerPhoneNumber(телефон за връзка с клиента). Методът ще работи по посочения начин:

- Създава се списък от RepairStatusCheckupViewModel, в който ще бъдат добавени всички резултати.
- Взимат се всички ремонти, чийто статус е равен на подадения статус.
- Визмат се по-горе посочените полета.
- Списъкът се връща.

### **3.6.4. Логин система**

В приложението ще има два типа вход:

- Вход за клиенти – въвеждат се името на клиента и номер сервизна поръчка. След като бъдат въведени тези данни приложението ще връща текущия статус на ремонта.
- Вход за оторизиран персонал – въвеждат се username, password. Пренасочва се към login страницата на Okta, където се въвеждат email и password и след като те се правилно въведени, се пренасочва към началната страница на приложението.

### 3.6.5. Онлайн плащания чрез PayPal

В клиентската част след като се напише името на клиента и номера на рекламация, се проверява дали статуса е Awaiting payment и ако е се визуализира бутона Pay и колко струва частите и труда по ремонта. След натискане на бутона Pay, се извиква метода GetResponse(Фиг. 3.17), който прави заявка към PayPal за създаване на плащане. След като бъде успешно създадено, клиентът се пренасочва към страницата на PayPal за завършване на плащането и след това се извиква метода CompletePayment(Фиг 3.18), който извършва плащането.

```
OrdersCreateRequest request = new OrdersCreateRequest();
request.Prefer("return=representation");
request.ContentType = "application/json";
request.RequestBody(order);
```

```
HttpResponse response = Client().Execute(request).Result;
```

Фиг. 3.17

```
OrdersCaptureRequest request =
    new OrdersCaptureRequest(
        response.Result<Order>().Id);

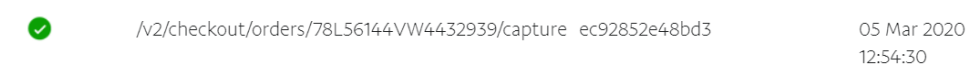
request.RequestUri = new Uri(response
    .Result<Order>().Links[3].Href);

request.RequestBody(new OrderActionRequest());
request.ContentType = "application/json";
request.Prefer("return=representation");

HttpResponse captureResponse = await Client()
    .Execute(request);
```

Фиг 3.18

Клиентът може да провери в акаунта си в PayPal (Фиг 3.19) дали плащането е извършено.



Фиг 3.19

### 3.6.6. Визуализация адрес клиент чрез Google Maps

След натискане на бутона Show in Maps в таблицата с всички ремонти, се отваря диалог, на който се визуализира google maps, показващ местоположението на адреса на клиента. Извиква се метод, който взема координатите на местоположението (Фиг. 3.20), който прави заявка към google maps и взема на първото съвпадение координатите.

```
GoogleSigned.AssignAllServices(  
    new GoogleSigned(CommonSecurityConstants  
        .GoogleMapsApiKey));  
  
GeocodingRequest request = new GeocodingRequest();  
request.Address = new Location(address);  
  
GeocodeResponse response = await Task.Run(() => {  
    return new GeocodingService()  
        .GetResponseAsync(request).Result;  
});
```

Фиг. 3.20

### 3.6.7. Интеграция със спедиторското API на Speedy

След натискане на бутона за генериране на товарителница, се отваря диалог, на който се попълват необходимите данни, и след това се извиква метода GenerateConsignment (Фиг. 3.21), който прави заявка към Speedy.

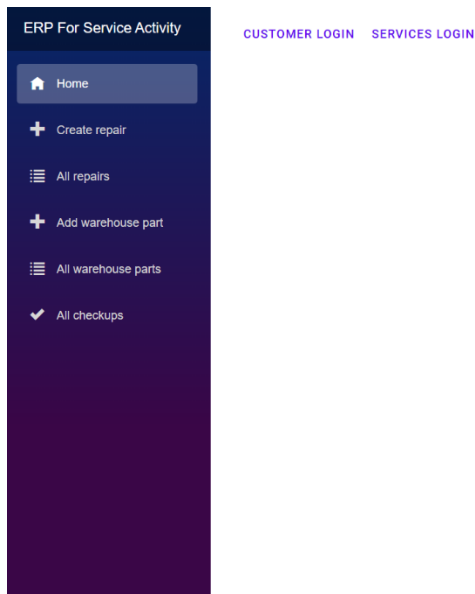
```
HttpRequestMessage message = new HttpRequestMessage(  
    HttpMethod.Post,  
    "https://api.speedy.bg/v1/shipment");  
  
message.Content = new StringContent(  
    System.Text.Json.JsonSerializer.Serialize(shipment, options),  
    System.Text.Encoding.UTF8,  
    "application/json");  
  
HttpResponseMessage response = await Task.Run(() => {  
    return client.SendAsync(message);  
});
```

Фиг. 3.21

## ЧЕТВЪРТА ГЛАВА

### Ръководство на потребителя

От произволен компютър в работната мрежа на сервиза да се достъпи следния уеб адрес <https://xxxxxxxxxxxxx.bg/>. Отваря се следната страница, показана на фиг. 4.1.

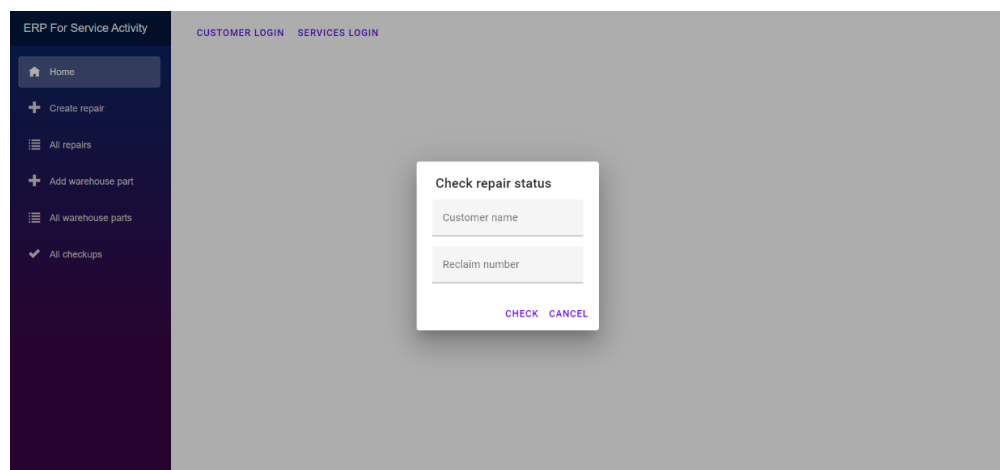


Фиг. 4.1

### 4.1. Login в системата

#### 4.1.1. Login за клиенти

След като бъде натиснат бутона Customer Login, ще се отвори следният прозорец, който е показан на фиг. 4.2.

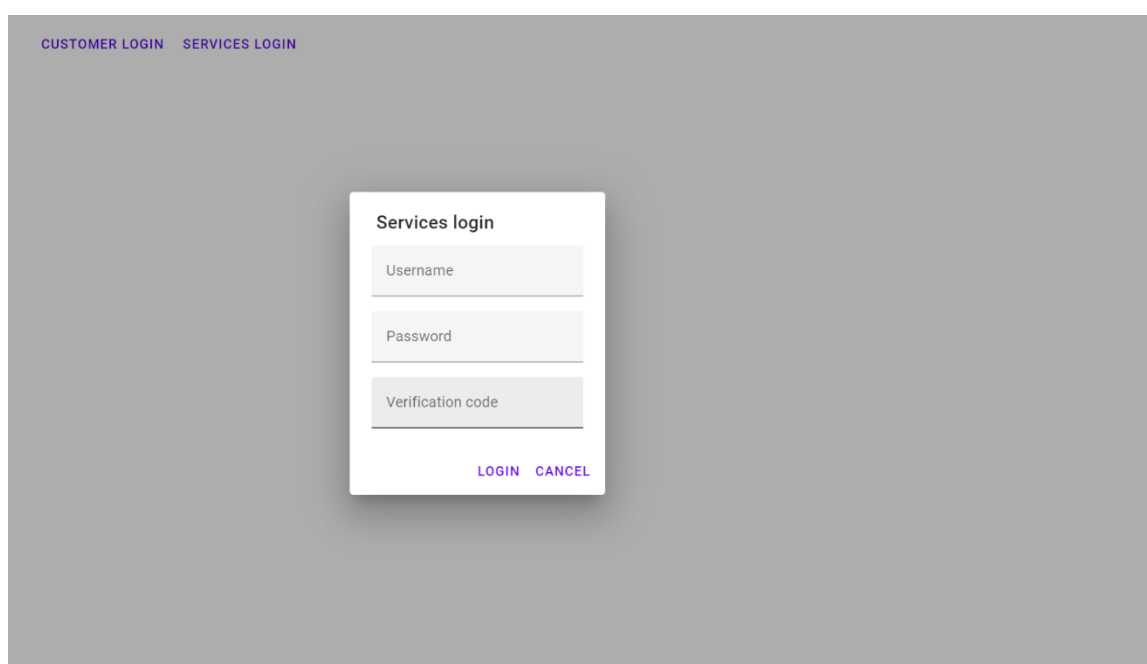


Фиг. 4.2

След като се въведат данни в полетата Customer name и Reclaim number, които съответно означават двете имена на клиента и номер на рекламацията, се натиска бутона Check, който показва какъв е статусът на поръката в настоящия момент.

#### 4.1.2. Login за оторизиран персонал

След като бъде натиснат бутона Services login, ще се покаже прозорец, показан на фиг. 4.3.

The image shows a web application interface with a dark gray background. At the top left, there are two links: 'CUSTOMER LOGIN' and 'SERVICES LOGIN', both in purple. In the center, a white modal dialog box titled 'Services login' is displayed. Inside the dialog, there are three input fields: 'Username', 'Password', and 'Verification code'. At the bottom right of the dialog, there are two buttons: 'LOGIN' and 'CANCEL', both in purple.

Фиг. 4.3

Когато бъдат въведени Username, Password и Verification code, които съответно означават името на съответния служител на даден сервиз, неговата парола и кода за потвърждение, който се изпраща на имейла на съответния user, се натиска бутона Login, който ще отведе оторизирания персонал в системата.

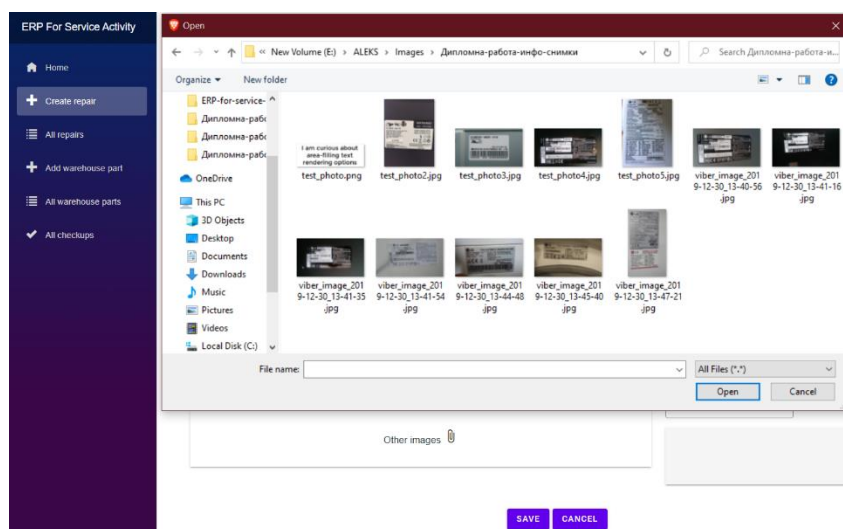
## 4.2. Създаване на поръчка

След като бъде натиснат бутона Create Repair от страничното меню за навигация, ще се отвори формата, показана на фиг. 4.4.

The screenshot shows the 'Create a new repair' form. On the left is a dark sidebar with the title 'ERP For Service Activity' and a menu with options: Home, Create repair (highlighted), All repairs, Add warehouse part, All warehouse parts, and All checkups. The main form is divided into two sections: 'Customer data' and 'Unit data'. The 'Customer data' section includes input fields for Name, Address, and Phone Number, a checkbox for 'Going to address', and buttons for 'Defect by customer', 'Add equipment', and 'Other information'. Below these is an 'Other images' upload area. The 'Unit data' section includes dropdowns for 'Select Brand' and 'Select Type', input fields for 'Model', 'S/N', and 'Product code/IMEI', a date picker for 'Bought at' and a field for 'WC number', a 'Warranty period' dropdown set to 0, and a 'Dealer/Shop' field. Below these is an 'S/N image' upload area. At the bottom right are 'SAVE' and 'CANCEL' buttons.

Фиг. 4.4

При натискането на S/N Image се отваря прозорец, на който трябва да бъде избрана снимката на стикера на уреда, което е показано на фиг. 4.5.



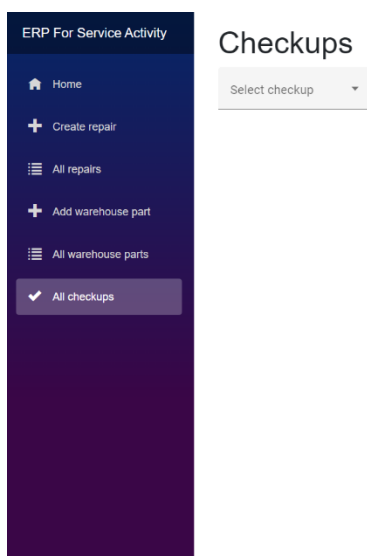
Фиг. 4.5



След като бъде избрана снимката, след 6-8 секунди ще се появи прозорец, който ще показва снимката и генерираните данни, за да може, ако има грешки, да се поправят ръчно.

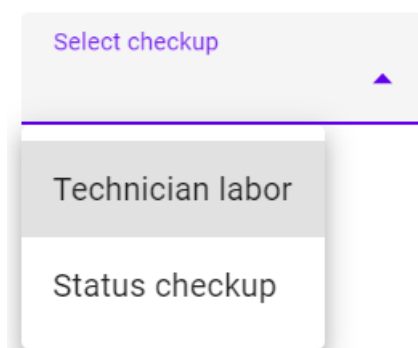
### 4.3. Справки

След като бъде натиснат бутона All checkups от страничното меню, се показва страница, която е показана на фиг. 4.6.



Фиг. 4.6

В зависимост какъв тип справка е необходима, се избира от падащото меню, което е показано на фиг. 4.7.



Фиг. 4.7

Това са примерни справки, но са най-често използваните в сервизната дейност.

## 4.4. Управление на склад

### 4.4.1. Добавяне на част

След като се натисне бутона от страничното навигиращо меню Add warehouse part, се отваря формата за добавяне на част в склада, която е показана на фиг. 4.8.

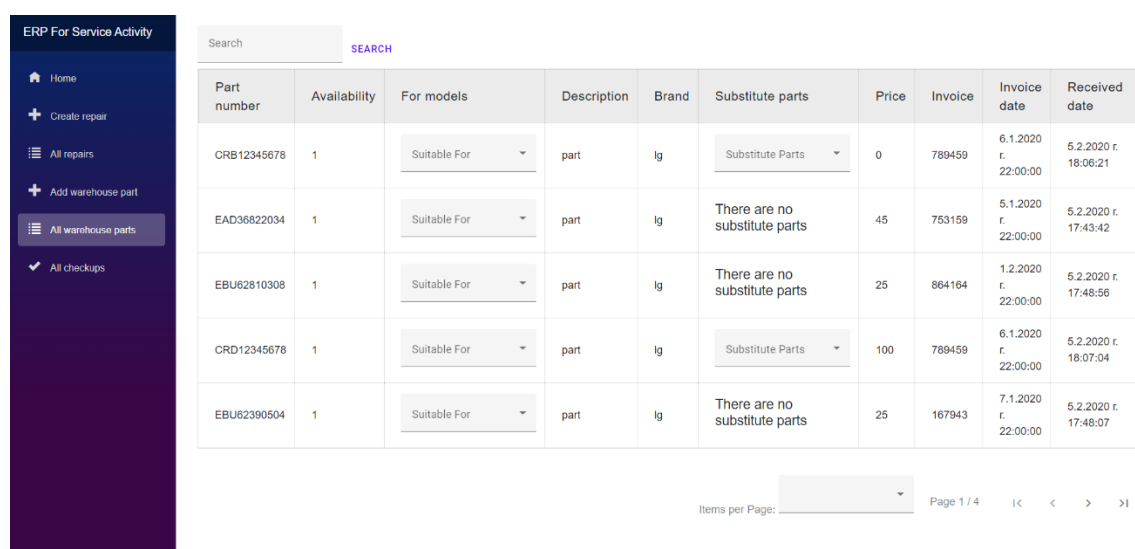
The screenshot displays the 'ERP For Service Activity' mobile application interface. On the left is a dark blue sidebar menu with the following options: 'Home' (house icon), 'Create repair' (plus icon), 'All repairs' (list icon), 'Add warehouse part' (plus icon, highlighted with a light blue background), 'All warehouse parts' (list icon), and 'All checkups' (checkmark icon). The main content area on the right is white and contains a form for adding a part to the warehouse. The form fields are: 'Part number', 'Model', 'Description', 'Brand', 'Substitute Part', 'Invoice number', 'Date of receiving invoice' (with a calendar icon), and 'Price' (with the value '0' displayed). At the bottom of the form is a purple button labeled 'ADD PART TO WAREHOUSE'.

Фиг. 4.8

След като бъдат въведени всички данни във формата правилно, при натискане на бутона Add part to warehouse, частта се добавя в склада.

#### 4.4.2. Управление на складова наличност

Управление на складова наличност е нужно за сервизите, за да се знае ако липсва дадена част, тя да бъде поръчана от външен доставчик/производител. След натискането на бутона All warehouse parts от страничното навигационно меню се отваря страница, която е показана на фиг. 4.9.



Part number	Availability	For models	Description	Brand	Substitute parts	Price	Invoice	Invoice date	Received date
CRB12345678	1	Suitable For	part	Ig	Substitute Parts	0	789459	6.1.2020 г. 22:00:00	5.2.2020 г. 18:06:21
EAD36822034	1	Suitable For	part	Ig	There are no substitute parts	45	753159	5.1.2020 г. 22:00:00	5.2.2020 г. 17:43:42
EBU62810308	1	Suitable For	part	Ig	There are no substitute parts	25	864164	1.2.2020 г. 22:00:00	5.2.2020 г. 17:48:56
CRD12345678	1	Suitable For	part	Ig	Substitute Parts	100	789459	6.1.2020 г. 22:00:00	5.2.2020 г. 18:07:04
EBU62390504	1	Suitable For	part	Ig	There are no substitute parts	25	167943	7.1.2020 г. 22:00:00	5.2.2020 г. 17:48:07

Фиг. 4.9

Отваря се таблица, в която са показани всички части в склада, които са налични в склада. Чрез падащото меню, разположено до Items per Page, user-ът може да си наглася по колко реда да виждат на една страница от таблицата. С търсачката може да се търси по part number на частта или по част от part number-a. Например може да се напише EBU\*, което ще изкара всички части, чийто part number започва с EBU както и всички части, които съдържат части заместители, започващи с EBU.

## ЗАКЛЮЧЕНИЕ

Създаден е интуитивен интерфейс, който може да се ползва от всички потребители, които са свързани със сервизната дейност на черна, бяла и климатична техника.

Като перспектива за развитие на настоящата дипломна работа бих добавил транспортен модул – собствен или чрез интеграция на спедиторско API, счетоводен модул, директно свързан с НАП в съответствие с наредба Н-18. Също така бих доразвил тази ERP система по такъв начин, че да удовлетворява в максимална степен изискванията на други типове сервизна дейност (например сервиз на автомобили, индустриална техника и т.н.).

В бъдещ план, след като се доразвият необходимите функционалности на Firebase библиотеките за .NET, бих доработил създадената от мен ERP система, така че да е възможно да се инсталира от .exe файл на Windows или Linux по желание на потребителя.

## ИЗПОЛЗВАНА ЛИТЕРАТУРА

- Описание на ASP.NET Web API – <https://www.tutorialsteacher.com/webapi/what-is-web-api>
- Описание на SignalR – <https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-3.1>
- Описание на Blazor – <https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-3.1>
- Описание на Firebase – <https://searchmobilecomputing.techtarget.com/definition/Google-Firebase>
- Описание на Cloud Firestore – <https://firebase.google.com/docs/firestore>
- Описание на Google Cloud Platform – <https://intellipaat.com/blog/what-is-google-cloud/>
- Закон за защита на потребителя – <https://www.lex.bg/laws/ldoc/2135513678>
- Причини избиране Firebase
  - <https://www.altexsoft.com/blog/firebase-review-pros-cons-alternatives/>
  - <https://www.cleveroad.com/blog/a-story-of-firebase-or-your-next-favourite-cloud-based-service>
- Причини избиране Firestore
  - <https://medium.com/@abhinavkorpai/scaling-horizontally-and-vertically-for-databases-a2aef778610c>
  - <https://github.com/vaquarkhan/vaquarkhan/wiki/Difference-between-scaling-horizontally-and-vertically>
- Причини избиране GCP - <https://kinsta.com/blog/google-cloud-hosting/>
- Описание принципа на работа на Paypal API - <https://developer.paypal.com/docs/api-basics/>

- Описание принципа на работа на Econt API -  
<https://www.econt.com/developers/31-sazdavane-na-dokument-za-izprashtane-tovaritelnitsa.html>
- Описание принципа на работа на Google Cloud Vision API -  
<https://becominghuman.ai/how-to-use-google-cloud-vision-api-4fbd800641f9>
- Описание принципа на работа на OCR или Tesseract OCR
  - <https://www.explainthatstuff.com/how-ocr-works.html>
  - <https://www.efilecabinet.com/how-does-ocr-work/>
  - <https://www.youtube.com/watch?v=izWqK53gheM>
  - [https://en.wikipedia.org/wiki/Optical\\_character\\_recognition](https://en.wikipedia.org/wiki/Optical_character_recognition)
  - <https://www.codeproject.com/Articles/15304/Unicode-Optical-Character-Recognition>
  - <https://nanonets.com/blog/ocr-with-tesseract/>
  - <https://www.geeksforgeeks.org/tesseract-ocr-with-java-with-examples/>
- Конвертиране на клас в Dictionary -  
<https://stackoverflow.com/questions/9210428/how-to-convert-class-into-dictionarystring-string/9210493>
- Взимане на резултат от Task.Run -  
<https://stackoverflow.com/questions/18050836/getting-return-value-from-task-run>
- Разчитане на модел, сериен номер и продуктов код на LG -  
<https://forum.setcombg.com/lg-%D1%82%D0%B5%D0%BB%D0%B5%D0%B2%D0%B8%D0%B7%D0%BE%D1%80%D0%B8/58650-lg-%D1%82%D0%B5%D0%BB%D0%B5%D0%B2%D0%B8%D0%B7%D0%BE%D1%80%D0%B8-2017-a.html>

- Документацията на Google Cloud за .NET -  
<https://cloud.google.com/dotnet/docs/>

## Съдържание

УВОД.....	1
ПЪРВА ГЛАВА.....	3
1.1.Обзор на използваните технологии .....	3
1.1.1. ASP.NET Web API.....	3
1.1.2. SignalR .....	5
1.1.3. Blazor.....	6
1.1.3.1. Razor Components.....	6
1.1.3.2. JavaScript interop.....	8
1.1.3.3. Blazor Server App.....	8
1.1.3.4. Blazor WebAssembly .....	10
1.1.4. Okta .....	13
1.1.5. Google Firebase.....	13
1.1.6. Google Cloud Platform (GCP) .....	14
1.1.6.1. Cloud Computing.....	14
1.1.6.2. GCP.....	14
ВТОРА ГЛАВА .....	15
2.1. Функционални изисквания към приложението.....	15
2.1.1. Responsive уеб сайт за управление на сервиз .....	15
2.1.2. История на ремонтите, водене на worklog (с текст, снимки, поръчани части).....	15
2.1.3. Клиентска част за следене на ремонта и история на ремонти .....	15
2.1.4. Онлайн плащане на ремонта .....	15
2.1.5. Доставка с търсене на адрес и визуализация с google maps.....	15
2.1.6. Генериране на pdf фактура или printer friendly страница и изпращане на email.....	16
2.1.7. Интеграция със спедиторско API.....	16
2.2.Аргументация за използваните технологии.....	16
2.2.1. Blazor.....	16
2.2.2. Web API .....	17
2.2.3. Google Firebase.....	17
2.2.3.1. Cloud Firestore.....	18
2.3.Google Cloud Platform(GCP) .....	18
2.3.1.1. Better Pricing Than Competitors .....	19
2.4. Структура на приложението.....	20
2.4.1. ERPForServiceActivity.API.....	22
2.4.2. ERPForServiceActivity.App .....	22
2.4.3. ERPForServiceActivity.Common .....	22
2.4.4. ERPForServiceActivity.CommonModels .....	23
2.4.5. ERPForServiceActivity.Services .....	23



ТРЕТА ГЛАВА.....	25
3.1.Какво е рекламация .....	25
3.2.Приемане на уред.....	25
3.3.Проверка дали уредът е в гаранция или не .....	25
3.3.1. В гаранция.....	26
3.3.1.1.  Посещение на адрес .....	26
3.3.1.2.  Диагностика на уреда .....	27
3.3.1.3.  Части по ремонт .....	27
3.3.1.4.  Уведомяване на клиента за приключил ремонт .....	28
3.3.2. Извън гаранция.....	28
3.4.Принципът на работа на Google Cloud Vision API.....	30
3.5.Обяснение припципа на работа на Paypal API .....	31
3.6.Реализация на приложението .....	33
3.6.1. Дейности, свързани с поръчки или ремонтни карти .....	33
3.6.1.1.  Създаване на поръчка .....	33
3.6.1.2.  Взимане на всички ремонти.....	40
3.6.2. Дейности, свързани с склада на сервиз.....	42
3.6.2.1.  Добавяне на част в склада .....	42
3.6.2.2.  Взимане на всички части.....	46
3.6.3. Процеси, свързани със справки в сервиза.....	47
3.6.3.1.  Справка за труда на техниците.....	47
3.6.3.2.  Справка по статус .....	48
3.6.4. Логин система.....	48
3.6.5. Онлайн плащания през PayPal .....	49
3.6.6. Визуализация на адрес на клиент чрез Google Maps .....	50
3.6.7. Интеграция със спедиторското API на Spedy.....	51
ЧЕТВЪРТА ГЛАВА.....	52
4.1.Login в системата.....	52
4.1.1. Login за клиенти .....	52
4.1.2. Login за оторизиран персонал.....	53
4.2.Създаване на поръчка.....	54
4.3.Справки.....	55
4.4.Управление на склад .....	56
4.4.1. Добавяне на част.....	56
4.4.2. Управление на складова наличност .....	57
ЗАКЛЮЧЕНИЕ .....	58
ИЗПОЛЗВАНА ЛИТЕРАТУРА.....	59
СЪДЪРЖАНИЕ.....	62