



ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

---

ФАКУЛТЕТ КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ

# ДИПЛОМНА РАБОТА

**ТЕМА:** Проектиране и реализация на приложение  
за управление и преглед на сензори

**Ръководител:** Гл. Ас. Д-р Невен Николов

София, 2020

## Съдържание

1.	I. Въведение	3
2.	II. Анализ на темата. Цели и задачи на приложението	4
1.	Цели	4
2.	Задачи	4
3.	Анализ	4-5
3.	III. Обзор на използваните технологии	6
1.	ASP.NET Core	6
2.	MVC	7
3.	Entity Framework Core	8
4.	Microsoft SQL Server	8-9
5.	Flutter	9-10
6.	JavaScript	10
7.	jQuery	11
8.	HTML	11
9.	CSS	12
4.	IV. Проектиране и архитектура на приложението	13
1.	Проектиране на приложението	13-11
2.	Логическа архитектура на уеб приложението	15-21
3.	Логическа архитектура на мобилното приложение	22-24
4.	Архитектура на базата данни	24-26
5.	V. Програмна реализация	26
1.	Регистрация и вход. Валидация	26-28
2.	Търсене и филтриране на потребителски сензори	29-31
3.	Запис в базата данни	31-32
4.	Създаване на сензор от мобилното приложение	33-35
6.	VI. Ръководство на потребителя	36
1.	Начална страница на мобилното приложение	36-37
2.	Добавяне и редактиране на настройка на сензор. Валидация	38-39
3.	Добавяне и редактиране на сензор. Валидация	40-41
4.	Начална страница на уеб приложението	42-43
5.	Регистрация и вход	43-44
6.	Магазин за сензори	45
7.	Добавяне и редактиране на потребителски сензор. Валидация	46-47
8.	Преглед на потребителските сензори	47-49
9.	Известия	49
7.	VII. Експериментални резултати	50
1.	Достъп до чужди записи	50
2.	Прихващане на изключения	50
8.	VIII. Използвана литература	51
9.	IX. Приложение	51

## I. Въведение

В днешно време тенденциите в разработването на програмните системи са насочени все повече към към уеб технологиите. Една от основните причини за това е, че уеб приложенията са достъпни от всяка точка на света. При тях липсва необходимостта от инсталациране на локалната машина на потребителя, а е достатъчен уеб браузър. Това ги прави много удобни и лесно приложими. С помощта на уеб базираните информационни системи могат лесно да се организират и поддържат различни управлениски, комуникационни и др. дейности.

В следващите страници ще бъде представено изграждането на онлайн система за управление и преглед на сензори. В това забързано ежедневие, което водим в наши дни хората имат нужда от система, която могат да достъпват от всяко едно устройство било то настолен компютър, лаптоп или смартфон и да контролират своите технологии на едно централизирано място. В случая онлайн системата за управление и преглед на сензори предлага точно това – едно централизирано място, на което потребителите могат да управляват, преглеждат и бъдат известявани относно закупените от тях сензори.

Разработката на онлайн системата за управление и преглед на сензори е представена в следните глави:

- Във втора глава са разгледани целите и ползите от приложението. Описва се какво има направено до този момент като приложения и програми.
- В трета глава са описани използваните технологии – езици за програмиране, бази данни и др.
- В четвърта глава са разгледани архитектурите на уеб приложението, мобилното приложение и базата данни.
- В пета глава се разглеждат някои по-важни аспекти от кода, за какво служи той и начините по които се реализира дадената функционалност.
- В шеста глава е показано подробно ръководство на потребителя.
- В седма глава са разгледани експериментални резултати от срещани проблеми и прихващане на грешки.
- В осма глава е представена използваната литература.
- В девета глава е описано къде може да бъде намерен source кода на системата.

## **II. Анализ на темата. Цели и задачи на приложението**

### **Цели**

В това забързано ежедневие, в което живеем днес е от значение всяка секунда време, което отделяме. Затова системата е предназначена да улесни информирането и известяването на потребителя за закупените от него сензори като предоставя данни за тях в реално време на едно централизирано място.

### **Основни задачи**

1. Приложението трябва да е лесно и интуитивно за употреба.
2. Да осигурява възможност за бърза регистрация и автентикация.
3. Да предоставя лесен начин за добавяне/премахване на сензори от магазина.
4. Да предоставя лесен начин за „закупуване“/премахване на сензори от всеки потребител.
5. Да предоставя лесен начин за преглед на закупените сензори от потребителя на карта.
6. Да предоставя лесно видими данни за всеки един закупен сензор от потребителя в реално време.
7. Да има възможност за допълнително лесно разширяване и обновяване на съществуващите функции, както и добавяне на нови функционалности.

### **Анализ**

#### **Datum Platform Interactive<sup>[1]</sup>**

Системата Platform Interactive на Datum е софтуер, чрез който клиентите могат да получават информация за закупените си устройства.

Част от предлаганите функционалности:

- Системата им е изцяло онлайн, което означава, че не са необходими никакви хардуерни действия от хора, които да инсталират или монтират системата.
- Системата им позволява скалиране, чрез повишаване/намаляване на плана, за да се предостави пълноценно потребителско изживяване.
- Системата им разполага с различни видове известяване – СМС, имейл, телефонно обажддане.



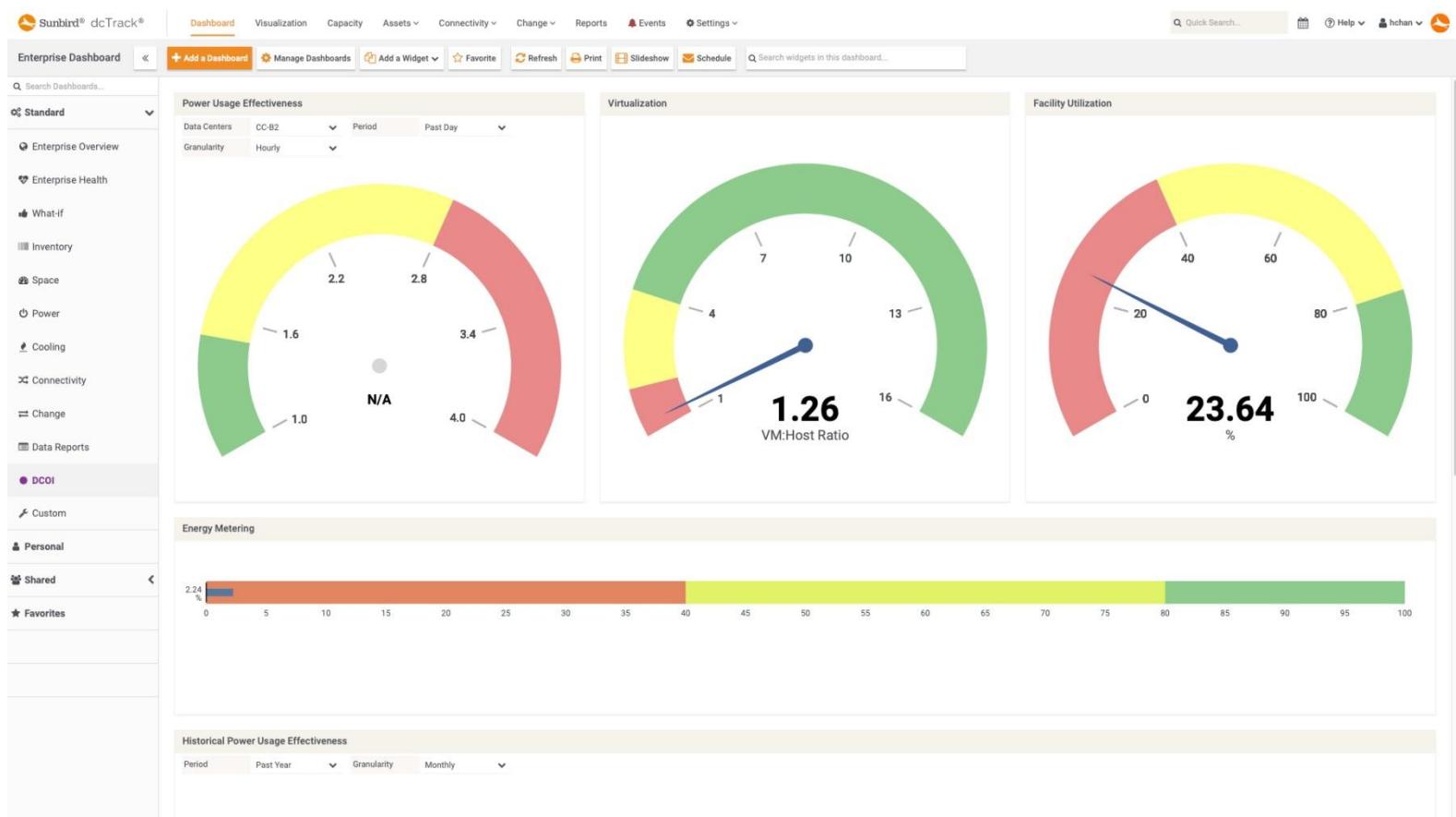
Фиг. 1. Datum Platform interactive UI

## Sunbird Energy [2]

Системата Energy на Sunbird предоставя софтуер за преглед на данни от сензори в реално време с цел измерване на ефективност, пестене и редуциране на използваната енергия.

Част от предлаганите функционалности:

- Възможност за измерване и мониторинг на данните от сензорите
- Информация за сензорите в реално време
- Добре разработен и лесен за употреба визуален интерфейс



Фиг. 2. Sunbird Energy UI

След анализ на двете представени системи можем да направим следните изводи:

- Подобни приложения вече съществуват, което не е учудващо предвид масовостта на идеологията за контролиране на техниката от едно централизирано място. Всяко от тях е уникално и предоставя различни допълнителни функционалности.
- Sunbird Energy има разработено десктоп приложение, докато Datum Platform Interactive е онлайн тоест уеб базирано решение.
- Datum Platform Interactive предлага различни видове известявания, които липсват при Sunbird Energy.

### III. Обзор на използваните технологии

#### ASP.NET Core [3]

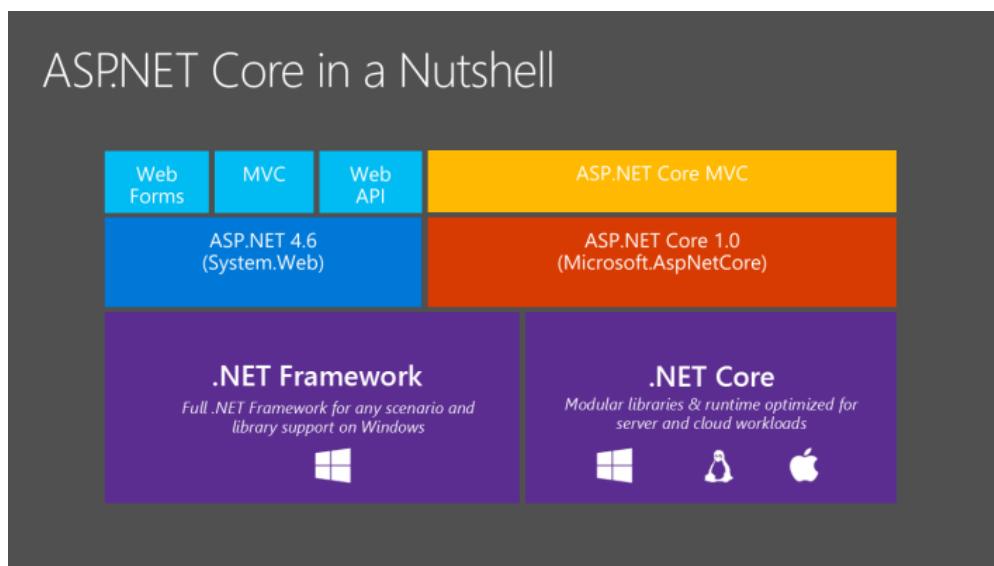
ASP.NET Core е бесплатна софтуерна рамка за уеб разработка с отворен код. Също така тя се явява и следващата стъпка в еволюцията на ASP.NET. Тя е разработена съвместно от Microsoft и общността, която е събрала през годините на своето развитие. ASP.NET Core е модуларна софтуерна рамка, която върви на крос-платформената .NET Core. Въпреки, че е нова софтуерна рамка изградена върху нов уеб стек, тя има висока степен на съвместимост и прилики с ASP.NET.

Спримо своите предшественици ASP.NET Core поддържа нова функция – т.нр. “side by side versioning”. При нея различни приложения, които използват една и съща машина могат да таргетират различни версии на ASP.NET Core, в зависимост от версията (и нуждите) си. Това не е възможно с по-стари издания на ASP.NET.

Съвместим е с Windows, MacOS и Linux, което го прави гъвкав и удобен за използване. Подходящ е за употреба, тъй като има интегрирани множество от библиотеки и компоненти, които се изпълняват по време на работа и компилация.

В случая не само платформите са от голяма значимост, но и това, че разработваната система има нужда от висока производителност и надграждане. .NET Core и ASP.NET Core са най-добрата опция в случая, тъй като предоставят нужната производителност за стартиране и функциониране на сървърната част. Освен това, управляемото време за изпълнение улеснява разработката, garbage-collection-а и гарантира безопасно изпълнение.

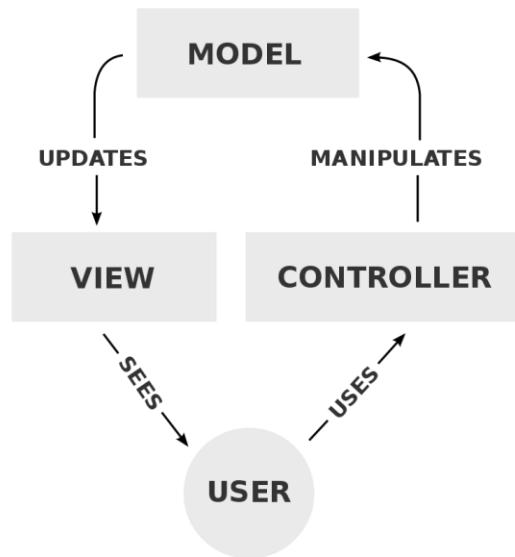
.NET Core е набор от изпълними, библиотечни и компилационни компоненти, които могат да се използват в различни конфигурации за работа на устройства и в облака. Междуплатформено и с отворен код .NET Core предоставя олекотен модел за разработване и гъвкавостта да се работи с различни инструменти за разработване и платформени ОС. .NET Core е достъпно в GitHub под лиценза MIT като включва няколко технологии - .NET Core, ASP.NET Core, Entity Framework Core и др.



Фиг. 3. Шаблон с възможности на ASP.NET Core

## MVC<sup>[4]</sup>

Архитектурата на приложението е изградена върху архитектурния шаблон MVC (Model-View-Controller), който помага да се постигне разпределение на идеите. При използването на този шаблон заявките на потребителите се насочват към контролера, който отговаря за работата с модела, за да се извършат очакваните от потребителя действия или да се изведат резултати от заявката.



Фиг. 4. Взаимодействие между Model – View – Controller и User

### Модел:

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

### Изглед:

Изгледите са тези, които определят как ще бъде визуализиран потребителският интерфейс (UI) на приложението. Характерното при тях е, че може да се включи C# код във файловете. Те са със специално разширение .cshtml и използват Razor Engine за компилация.

### Контролер:

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

## **Entity Framework Core** [5]

Entity Framework Core е лека, разширяема и многоплатформена версия на популярната технология за достъп и поддръжка на данни Entity Framework. Тя може да ни служи за създаване на връзка между обекти, позволявайки на разработчиците да работят с база от данни с .NET обекти и елиминирали нуждата от голяма част от кода за достъп до данните, която би се наложило да напишат.

Има много начини, чрез които може да се проектира такава библиотека, EF Core е създадена като object-relational mapper (ORM). Този тип дизайн работи като създава връзката между две страни базата данни с API-то, което използва и обектно ориентирана софтуерна страна. Това е един от най-бързите начини, чрез които разработчиците на софтуер достъпват базата от данни бързо и лесно. Именно затова, EF Core се използва за разработката на текущото приложение – за достъп и контрол на база от данни, която съхранява данните на приложението.



Фиг. 5. Лого на Entity Framework Core

## **Microsoft SQL Server** [6]

В същността си MS SQL Server е система за управление на релационни бази от данни. Като съвър, основната му функция е съхраняване и предоставяне на данни така, както са заявени от други приложения, които могат да са пуснати както на същата машина, така и през мрежа.

Основната причина за това MS SQL Server да е предпочитана платформа за разработка е това, че когато потребителите искат данни, те ги искат колкото може по-бързо. Табличите са оптимизирани в използването на памет, а за натоварване при анализ на данните, потребителите могат да се възползват от обновяеми кълстеририани и индексирани колони за тези таблици за около 100 пъти по-бързи заявки.

Друг голям плюс на MS SQL Server е, че включва няколко инструмента в помощ за планиране на миграции на данните в съществуващите таблици. Сигурността също е добре планирана – на ниво ред, а данните са винаги криптирани. Това означава, че данните, пазни в облака не могат да бъдат разчетени от никого.

И за да затворим цикъла, MS SQL Server, в пълната си версия, може да бъде съхранен в облачното пространство без да трябва да се управлява какъвто и да е хардуер. Това става възможно благодарение на виртуалните машини на Azure, които са пуснати на много и различни географски региони по света. Освен това те имат разнообразие от размери (в памет) на машините. Така потребителите могат да създават виртуална машина със SQL Server, която да бъде с правилната версия и операционна система. Обновяването и поддръжката се случват автоматично, както и създаването на архив на състоянието на базата, така че да не се губи информация.



Фиг. 6. Лого на Microsoft SQL Server

## Flutter [7]

Flutter е софтуерна рамка с отворен код създаден от Google през 2017, чрез който могат да се създават приложения за Android, iOS, Windows, Linux, MacOS и уеб като споделят една и съща версия на кода.

Архитектурата на Flutter се разделя на 4 категории:

1. **Dart платформата** – Flutter приложенията използват език за програмиране наречен Dart. Изпълняват се в Dart виртуална машина, която поддържа just-in-time компилация, което означава, че програмата може да се компилира в реално време докато работи, което позволява така наречената функция – “hot reload”, чрез която модификации на кода могат да бъдат заредени върху вече работещо приложение.
2. **Flutter Engine** – Flutter използва engine, който е написан основно на C++, като позволява достъп до ниско ниво на Google Skia графичната библиотека. Също така използва специфични интерфейси на Android и iOS SDK-та. Flutter Engine-а имплементира основните библиотеки на рамката, които включват анимации, графики, работа с файлове, достъп до мрежата и др.

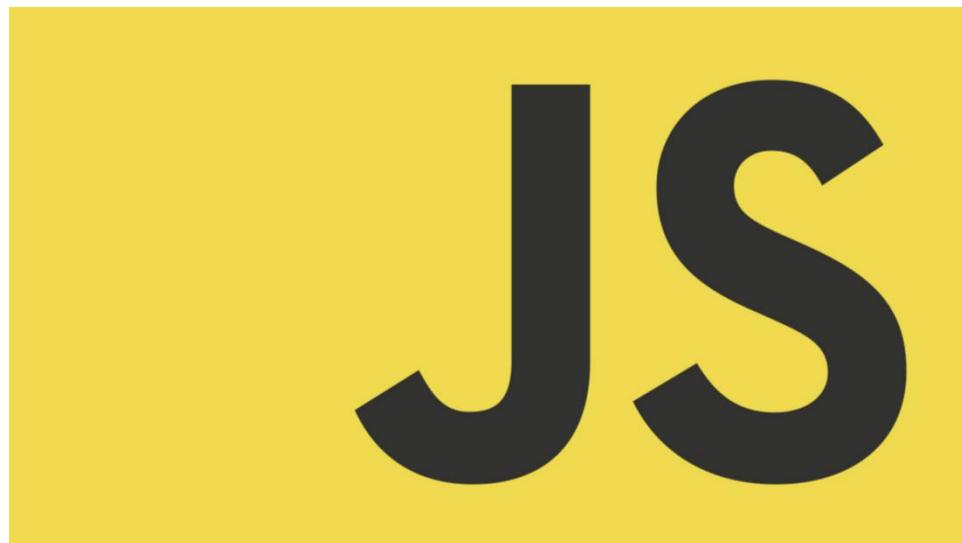
3. ***Foundation library*** – библиотека наисана на Dart, която предоставя достъп до базови класове и функции, които се използват при създаването на Flutter приложения както и API-та, които комуникират с Engine-на.
4. ***Design-specific widgets*** – Flutter приложенията използват така наречените уиджети при разработка. Основните типове са два – Material Design уиджети, които имплементират и следват насоките на Google за дизайн и разработка и Cupertino уиджети, които имплементират и следват насоките на Apple за дизайн и разработка.



Фиг. 7. Лого на Flutter

#### JavaScript [8]

JavaScript е интерпретиран език за програмиране, разпространяван с повечето уеб браузъри. Поддържа обектно-ориентиран и функционален стил на програмиране. Създаден е в Netscape през 1995 г. Най-често се прилага към HTML-а на Интернет страница с цел добавяне на функционалност и зареждане на данни. Може да се ползва също за писане на сървърни скриптове JSON, както и за много други приложения. JavaScript не трябва да се бърка с Java, съвпадението на имената е резултат от маркетингово решение на Netscape. JavaScript е стандартизиран под името EcmaScript.



Фиг. 8. Лого на JavaScript

## jQuery [9]

jQuery е разпространена библиотека на JavaScript, публикувана в началото на 2006 от Джон Резиг. В основата си jQuery опростява достъпа до всеки елемент на дадена уеб-страница, като по този начин позволява лесно изграждане на динамична функционалност в страниците.

jQuery е бесплатен и open source софтуер, лицензиран под MIT лиценз. jQuery се използва в 72% от 10000-те най-посещавани сайтове, което я прави най-популярната JavaScript алтернатива днес



Фиг.9. Лого на jQuery

## HTML [10]

HTML (съкращение от термина на английски: HyperText Markup Language, произнасяно най-често като „ейч-ти-ем-ел“, в превод „език за маркиране на хипертекст“) е основният маркиращ език за описание и дизайн на уеб страници. HTML е стандарт в Интернет. Последната версия е 5 от 2014 г.

Описанието на документа става чрез специални елементи, наречени HTML елементи или маркери, които се състоят от етикети или тагове (HTML tags) и ъглови скоби (като например елемента <html>). HTML елементите са основната градивна единица на уеб страниците. Чрез тях се оформят отделните части от текста на една уеб страница, като заглавия, цитати, раздели, хипертекстови препратки и т.н. Най-често HTML елементите са групирани по двойки <h1> и </h1>.

В повечето случаи HTML кодът е написан в текстови файлове и се хоства на сървъри, свързани към Интернет. Тези файлове съдържат текстово съдържание с маркери – инструкции за браузъра за това как да се показва текстът. Например <маркер> Някакъв текст. </край на маркера>.

Предназначенето на уеб браузърите е да могат да прочетат HTML документите и да ги превърнат в уеб страници. Браузърите не показват HTML таговете, а ги използват, за да интерпретират съдържанието на страницата.

Основното предимство на HTML е, че документите, оформени по този начин, могат да се разглеждат на различни устройства, а не само на екрана. Документът може да бъде правилно оформлен и върху монитора на персонален компютър, и върху миниатюрния дисплей на пейджър или мобилен телефон.

HTML може да прикрепя скриптове писани на езици като JavaScript, което променя поведението на уеб страницата. Може да се използва Cascading Style Sheets (CSS), който определя изгледа и оформлението на текста и други материали. World Wide Web Consortium (W3C) поддържа и двете CSS и HTML и насърчава използването на CSS в HTML страниците от 1997. Това допринася за разделяне съдържанието и структурата на уеб страниците от тяхното визуално представяне

## CSS (Cascading Style Sheets) [11]

CSS (Cascading Style Sheets) е език за описание на стилове (език за стилови листове, style sheet language) – използва се основно за описване на представянето на документ, написан на език за маркиране. Най-често се използва заедно с HTML, но може да се приложи върху произволен XML документ. Официално спецификацията на CSS се поддържа от W3C.

CSS е създаден с цел да бъдат разделени съдържанието и структурата на уеб страниците отделно от тяхното визуално представяне. Преди стандартите за CSS, установени от W3C през 1995 г., съдържанието на сайтовете и стила на техния дизайн са писани в една и съща HTML страницата. В резултат на това HTML кода се превръща в сложен и нечетлив, а всяка промяна в проекта на даден сайт изисквала корекцията да бъде нанасяна в целия сайт страница по страница. Използвайки CSS, настройките за форматиране могат да бъдат поставени в един-единствен файл, и тогава промяната ще бъде отразена едновременно на всички страници, които използват този CSS файл.



Фиг. 10. Лого на CSS 3 и HTML 5

## **IV. Проектиране и архитектура на приложението**

### **Проектиране на приложението**

Системата за управление и преглед на сензори се състои от 3 основни компонента - уеб приложение, мобилно приложение и база от данни.

#### **Основен принцип на работа:**

Първата ключова част е мобилното приложение. То служи за управление на наличните сензори в магазина. Чрез него могат да бъдат добавяни/редактирани/премахвани типовете сензори и самите сензори. То комуникира чрез HTTP заявки към Web API, което пък от своя страна комуникира с базата данни и съответно изпълнява CRUD операции върху нея в зависимост от заявката. Мобилното приложение предоставя клиентска валидация преди изпращането на формите към Web API, която е налична и на самия сървър.

След като вече имаме добавени сензори в магазина идва частта на уеб приложението. Чрез него потребителите могат да виждат наличните сензори в магазина и закупените публични сензори от ползвателите на услугата на Google Maps прозорец, който е видим на началната страница, но за да могат да използват тези сензори трябва да се регистрират в системата. При регистрацията е необходимо да бъде въведен реален имейл адрес, на който ще бъде изпратен линк за потвърждение на новорегистрирания акаунт и парола. След като потребителят потвърди акаунта си, то той вече може да се автентицира в системата със зададените имейл адрес и парола.

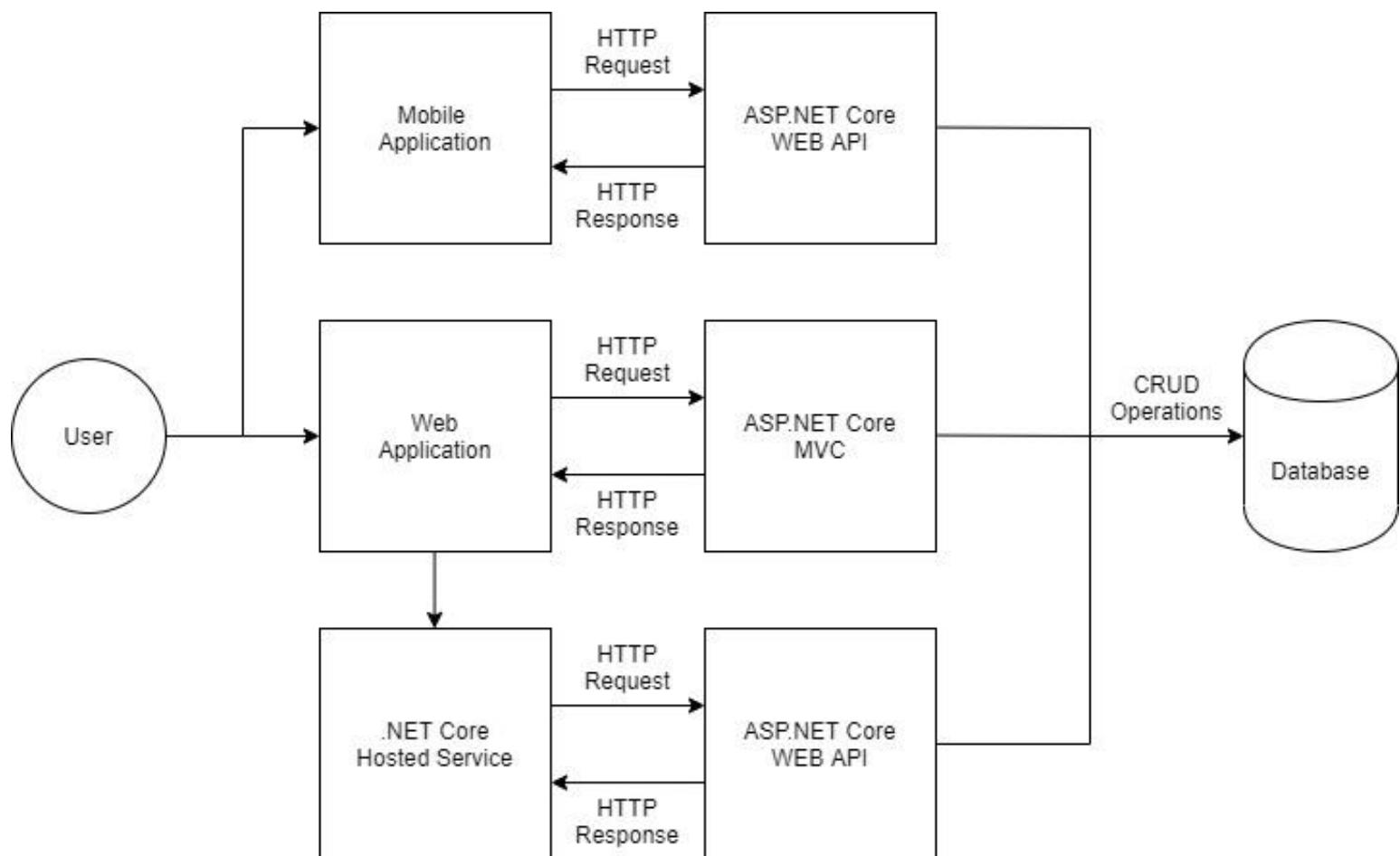
При закупуване/редактиране на сензор от магазина потребителят трябва да попълни следните данни след което се изпълнява HTTP Post заявка към съответния MVC контролер грижещ се за добавянето/редактирането на потребителски сензор:

- Име на сензор – името, което ще се показва при преглед на потребителските сензори.  
*Валидация: името на сензора е задължително!*
- Координати на картата – Служат за показване на местоположението на сензори на картата.  
*Валидация: Координатите на сензора са задължителни!*
- Настройка дали сензора е публичен – Ако сензора е настроен да бъде публичен, то той ще бъде видим на картата от началната страница за всички посетители на сайта.
- Описание на сензора – описанието, което ще се показва при преглед на потребителските сензори.  
*Валидация: Описанието на сензора е задължително!*
- Интервал на опресняване – Интервалът на опресняване се използва за показване на данните за сензори в реално време като през определения интервал те ще бъдат обновени и показани на потребителя.
- Аларма – Настройката е налична само при сензори, които са различни от тип „превключвател“. Ако опцията е налична се появяват двете допълнителни настройки:
  - Минимален обхват на сензора – Ако сензора покаже данни по-малки от минималния обхват – то алармата ще бъде задействана и потребителя ще бъде известен.  
*Валидация: Минималния обхват не може да бъде по-голям от максималния!*
  - Максимален обхват на сензора – Ако сензора покаже данни по-големи от максималния обхват – то алармата ще бъде задействана и потребителя ще бъдзе известен.  
*Валидация: Максималния обхват не може да бъде по-малък от минималния!*

След като потребителя е закупил сензор, той може да бъде видян на страницата за преглед като там ще бъдат показани 3 броя сензори с налична опция за pagination. Презареждането на страницата се случва, чрез изпълняване на GET AJAX заявка към MVC контролер, който от своя страна връща частичен изглед (partial view).

Всеки сензор показва и обновява данните си в реално време според неговия интервал на опресняване. Потребителят също има възможност да премахне закупен от него сензор, което от своя страна изпълнява DELETE HTTP заявка към съответния MVC контролер.

Данните на сензорите биват генериирани от .NET Core Hosted Service, който работи на заден фон при стартиране на уеб приложението. Service-а прави GET HTTP заявка, която се изпълнява от съответния Web API контролер, който от своя страна взима всички сензори от базата данни и генерира произволни стойности за всеки един от тях на база на неговия тип и обхват. Web API контролера връща списък от сензорите с попълнени стойности като резултат, след което service-а взима всички потребителски сензори, като ги филтрира според типа сензор и прави актуализация на данните за всеки един потребителски сензор.



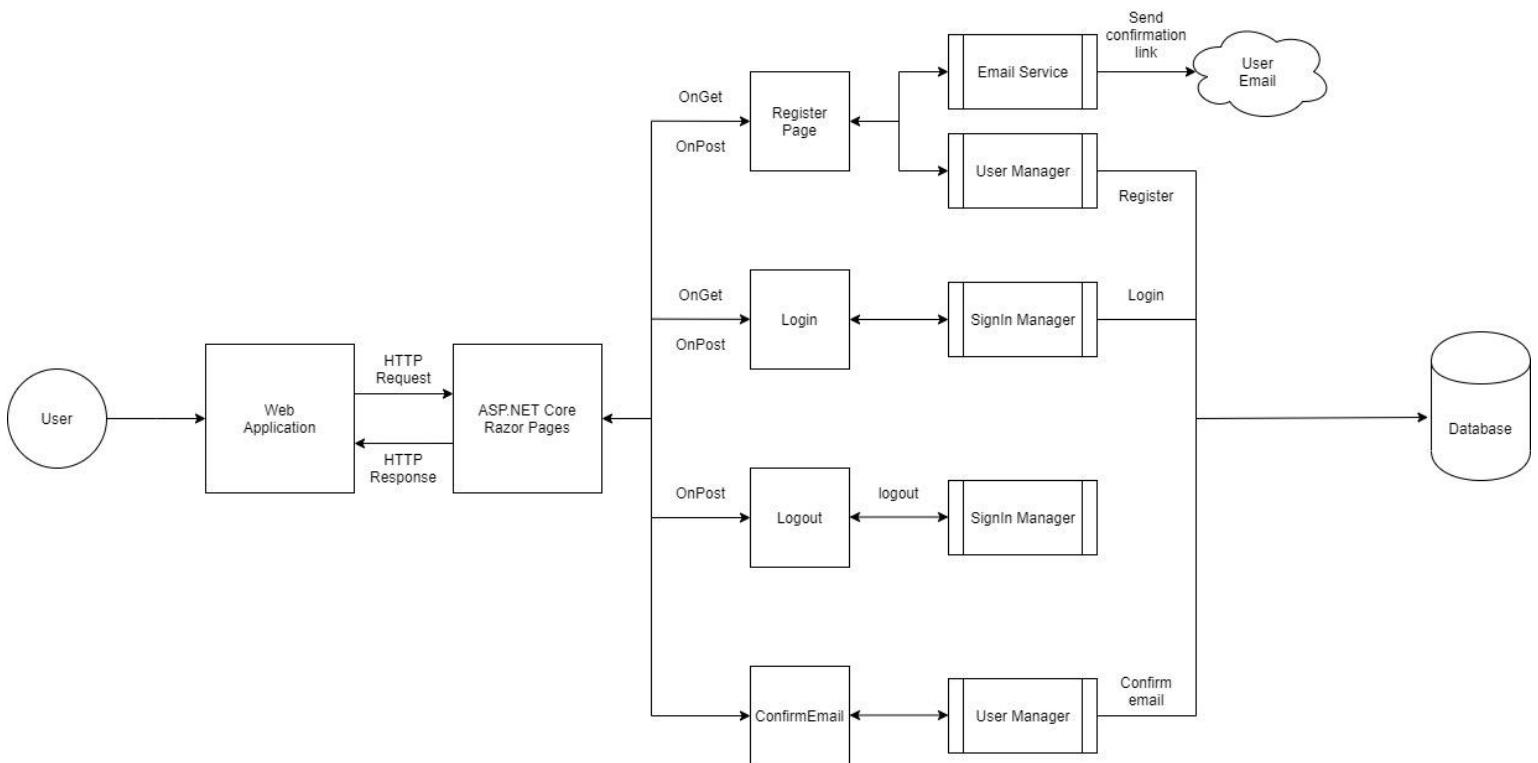
Фиг. 11. Диаграма показваща основния принцип на работа

## Логическа архитектура на уеб приложението

Уеб приложението на системата за управление и преглед на сензори се състои от 4 основни части разпределени в различни проекти – **App, Models, Services, Common**

**App:** ASP.NET Core проект използващ MVC шаблон за архитектура. Съдържа следните компоненти:

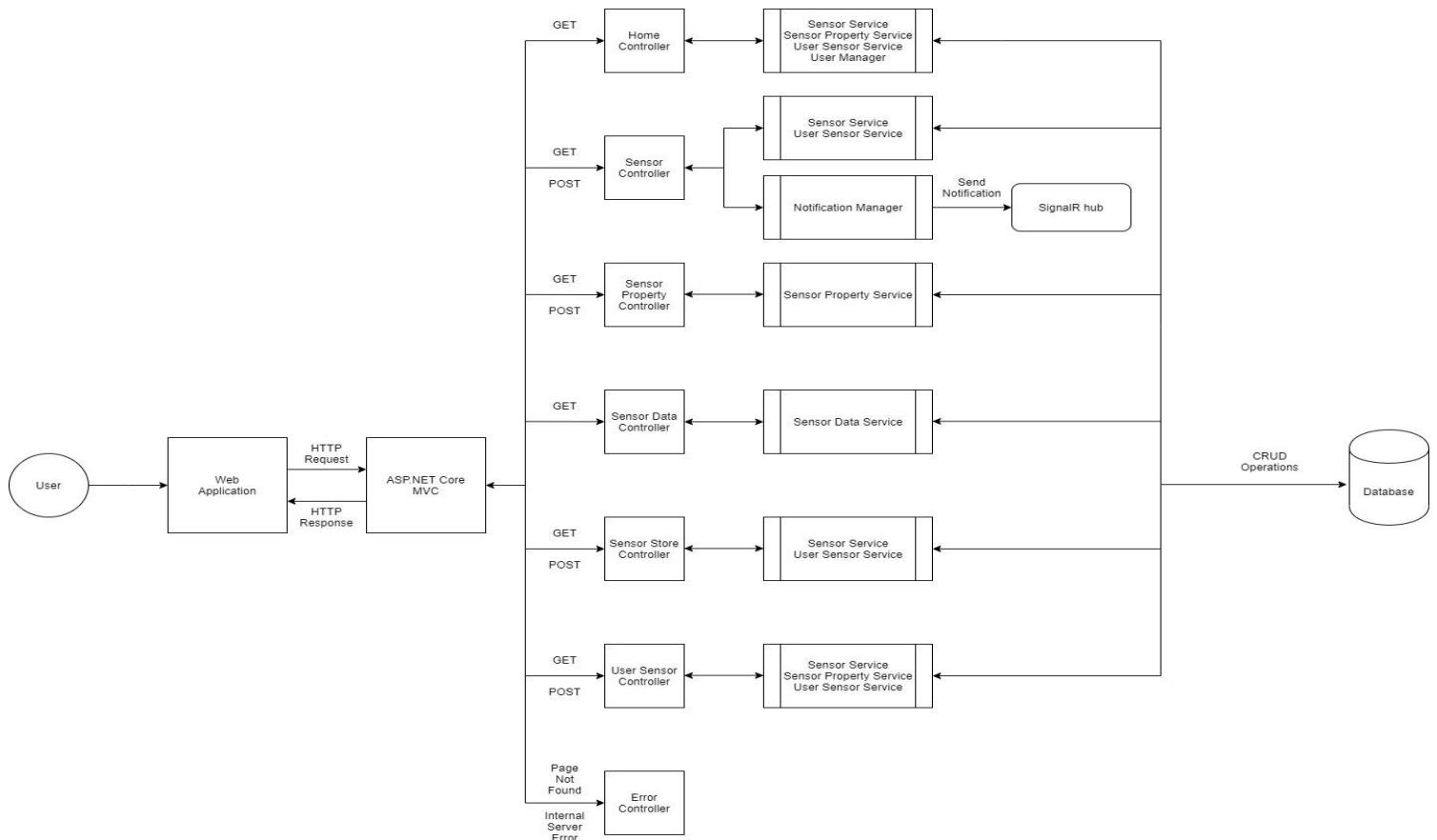
1. **Identity Area** – Служби за автентикация на потребителите като използва Razor Pages архитектура.
  - a. **Register Page** – Служби за регистрация на нови потребители в системата. Поддържа следните Action методи:
    - i. **OnGetAsync**
    - ii. **OnPostAsync**
  - b. **Login Page** – Служби за вход в системата на регистрираните потребители. Поддържа следните Action методи:
    - i. **OnGetAsync**
    - ii. **OnPostAsync**
  - c. **Logout Page** – Служби за изход/отписване на потребителя от системата. Поддържа следните Action методи:
    - i. **OnPostAsync**
  - d. **ConfirmEmail Page** – Служби за потвърждение на академа при отваряне на линка изпратен на имейл адреса на потребителя при регистрация. Поддържа следните методи:
    - i. **OnGetAsync**



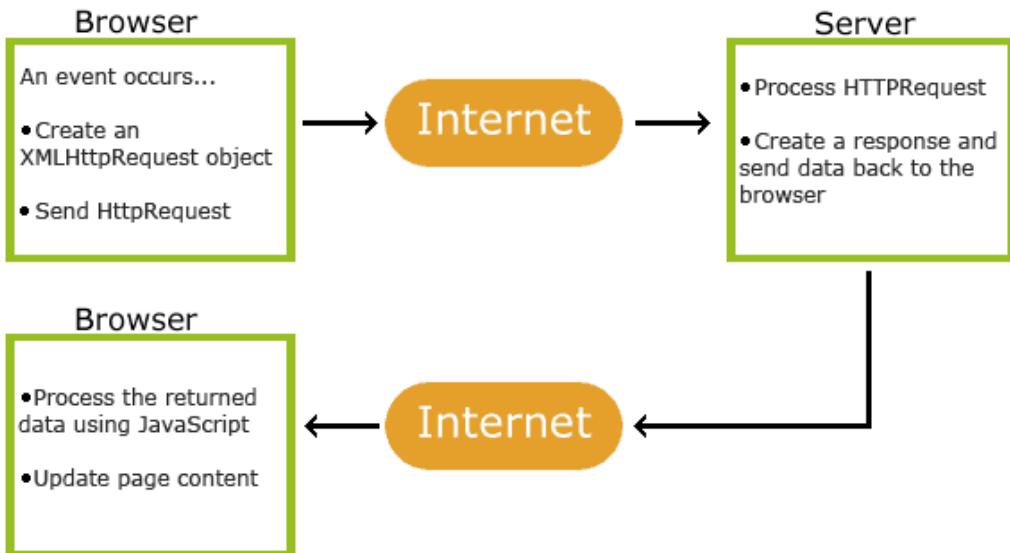
Фиг. 12. Диаграма показваща процеса свързан с автентикация на уеб приложението

2. **Controllers** – Служат за управлението и връщането на данните.
  - a. **HomeController** – Обслужва заявките на началната страница на приложението. Поддържа следните Action методи:
    - i. **Index** - Връща HomeIndexViewModel модел с попълнени данни за визуализиране на изгледа.
    - ii.  **GetUserSensorCoordinates** - Връща JSON с данните за закупените потребителски сензори, които ще бъдат визуализирани с помощта на Google карти.
  - b. **SensorController** – Обслужва заявките за CRUD операции на сензорите в магазина. Поддържа следните Action методи:
    - i. **GET/all** – Връща всички налични сензори в базата данни.
    - ii. **GET/{id}** – Връща един сензор филтриран по id.
    - iii. **GET/userdCound/{id}** – Връща броя на потребители закупили сензора филтриран по id.
    - iv. **POST** – Служи за създаване на нов сензор. Изпраща SignalR известие за създаден сензор.
    - v. **PUT** – Служи за обновяване на вече съществуващ сензор.
    - vi. **DELETE/{id}** – Служи за изтриване на сензор филтриран по id. Изпраща SignalR известие за премахнат сензор.
  - c. **SensorPropertyController** – Обслужва заявките за CRUD операции на настройките на сензорите в магазина. Поддържа следните Action методи:
    - i. **GET/all** – Връща всички налични настройки на сензорите в базата данни.
    - ii. **GET/{id}** – Връща една настройка за сензор филтрирана по id.
    - iii. **GET/sensormeasuretypes** – Връща всички налични видове сензори от базата данни.
    - iv. **POST** – Служи за създаване на нова настройка за сензор
    - v. **PUT** – Служи за обновяване на вече съществуваща настройка за сензор
    - vi. **DELETE/{id}** – Служи за изтриване на настройка засензор филтрирана по id
  - d. **SensorDataController** – Връща стойностите на всеки сензор от базата данни. Поддържа следните Action методи:
    - i. **GET/all**
  - e. **SensorStoreController** - Обслужва заявките за управление на сензорите в магазина. Поддържа следните Action методи:
    - i. **Index** – Връща SensorStoreViewModel модел използван за визуализиране на сензорите в изгледа.
    - ii. **ReloadSensorsTable** – Служи за обновяване на страницата със сензори при прилагане на филтър.
  - f. **UserSensorController** – Обслужва заявките за управление на закупените сензори от потребители. Поддържа следните Action методи:
    - i. **Index** – Връща UserSensorIndexViewModel модел използван за визуализиране на потребителските сензори в изгледа.
    - ii. **ReloadUserSensorsTable** – Връща PartialView съдържащо HTML част показваща таблицата със закупените сензори. Поддържа филтриране и pagination. *Фиг. 14*
    - iii. **Create** – Служи за създаване на потребителски сензор.
    - iv. **Edit** – Служи за редактиране на вече съществуващ потребителски сензор.
    - v. **GetGaugeData** – Връща стойността на даден потребителски сензор филтриран по id от базата данни.

- vi. **Delete** – Изтрива потребителски сензор филтриран по id.
  - vii.  **GetUserSensorCoordinates** – Връща координадите на потребителските сензори, които ще бъдат визуализирани с помощта на Google карти.
  - g. **ErrorController** - Служи за показване на страници за грешка при изключения или несъществуващи страници. Поддържа следните Action методи:
    - i. **Index** – Използва се при грешки със статус код различен от 404.
    - ii. **PageNotFound** – Използва се при грешки със статус код 404.
3. **Hubs** – Служи за изпращане на съобщения до потребителския интерфейс използвайки SignalR сокет. Съдържа следните класове:
- a. **SensorStoreHub** – Използва се за изпращане на съобщения до всички автентифицирани потребители при добавяне/изтриване на сензор от магазина.
  - b. **NotificationManager** – Служи за контейнер, който съдържа наличните Hub-ове и изпраща съобщения през тях.
4. **Views** – Изгледи, които се използват за визуализация на потребителския интерфейс. Разпределени са по папки като за всеки контролер има отделна папка, която съдържа .cshtml файлове с имената на всеки Action
5. **Startup.cs** – Служи за настройване на ASP.NET Core сървъра. Тук са регистрирани ендпоинтите, класовете за Dependency Injection, SignalR, Logging и др.
6. **Program.cs** – Стартовата точка на уеб приложението. Съдържа и стартиране на Logging provider-a.



Фиг. 13. Диаграма показваща принципа на работа на MVC частта със Service Layer-a.

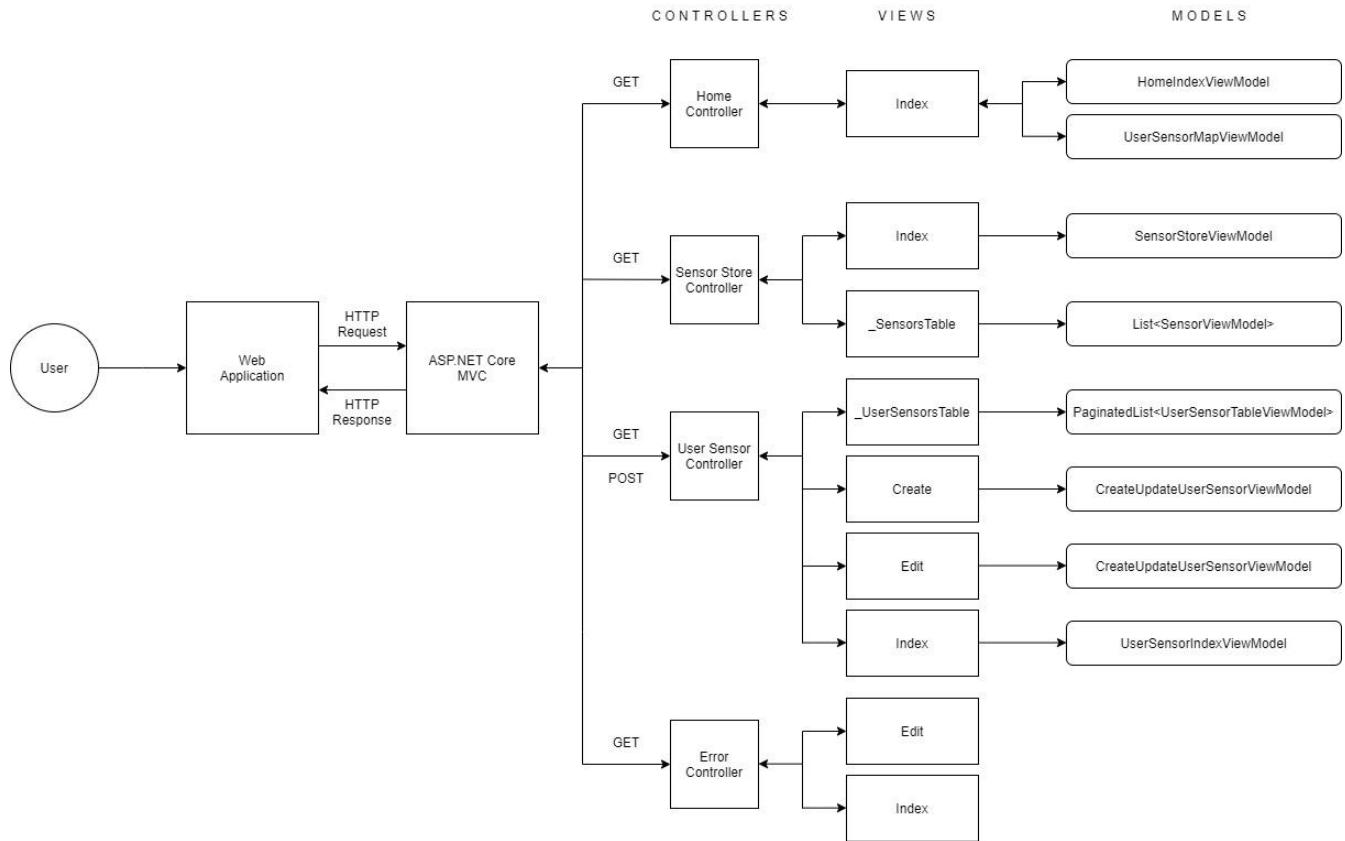


Фиг. 14. Процес на AJAX заявка

**Models** - .NET Core class library съдържаща всички модели използвани от приложението. Разпределение по папки:

1. **AutoMapper** – Съдържа конфигурационен профил и разширения за Automapper библиотеката.
2. **DTOs** – Съдържа Data Transfer Objects използвани от Sensor, SensorProperty и SensorData API. Съдържа следните класове:
  - a. **SensorDataDTO** – Използван от SensorDataController за трансфер на данни.
  - b. **SensorDTO** – Използван от SensorController за трансфер на данни.
  - c. **SensorPropertyDTO** – Използван от SensorPropertyController за трансфер на данни.
  - d. **UserSensorGaugeData** – Използван от UserSensorController за трансфер на данни
3. **Entities** – Съдържа класовете използвани за писане/четене на данни от базата данни. Съдържа следните класове:
  - a.  **BaseEntity** – Базов клас съдържащ основни пропъртита.
  - b.  **RoleEntity** – Използва се за управление на потребителските роли.
  - c.  **SensorEntity** – Използва се за записване/четене на сензор обекти.
  - d.  **SensorPropertyEntity** – Използва се за записване/четене на настройки за сензор обекти.
  - e.  **UserEntity** – Използва се за достъп до потребителите.
  - f.  **UserSensorEntity** – Използва се за записване/четене на потребителски сензори.
4. **ViewModels** – Служат за показване на каква точно информация да бъде показана в изгледите. Съдържа следните класове:
  - a. **CreateUpdateUserSensorViewModel** – Използва се при създаване/редактиране на потребителски сензор
  - b. **HomeIndexViewModel** – Използва се при показване на данните от началната страница.
  - c. **PagedList** – Използва се като помощен клас за pagination.
  - d. **SensorPropertyViewModel** – Използва се за показване на настройка на сензор
  - e. **SensorStoreViewModel** – Използва се за показване на сензори в магазина

- f. **SensorViewModel** - Използва се при презареждане на сензорите в магазина.
- g. **UserSensorIndexViewModel** – Използва се за показване на закупените потребителски сензори.
- h. **UserSensorMapViewModel** – Използва се за показване на закупените потребителски сензори в Google карти.
- i. **UserSensorTableViewModel** – Използва се при таблицата с потребителски закупените сензори.
- j. **UserSensorViewModel** – Използва се при презареждане потребителски закупените сензори.

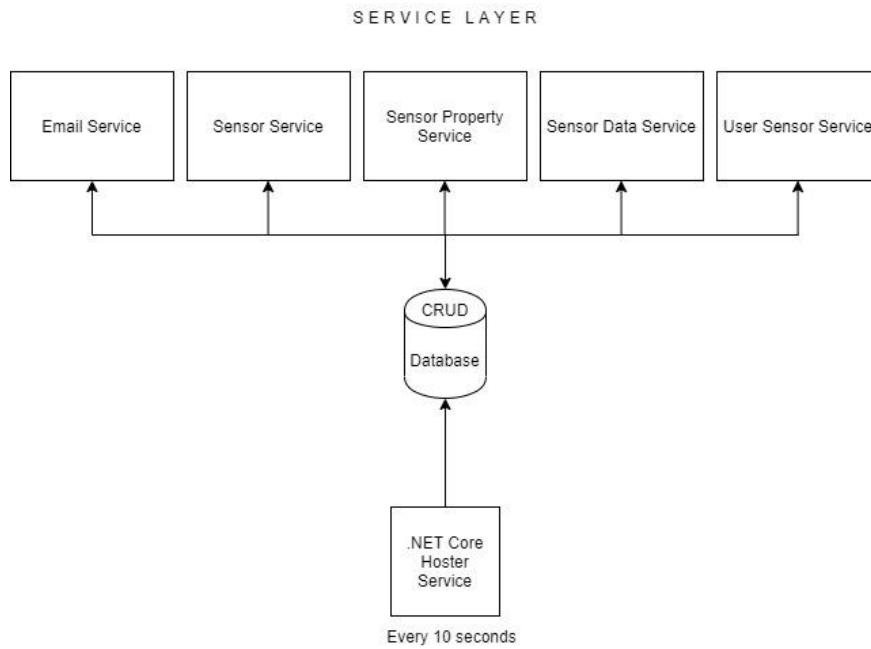


Фиг. 15. MVC структурирана на моделите и изгледите на приложението

**Services** - .NET Core class library съдържащата бизнес логика относно операциите за четене/запис с базата данни. Съдържа следните класове и интерфейси:

1. **IEmailService** – **EmailService** – Използва се за генериране и изпращане на съобщения по даден имейл адрес.
2. **SensorDataFetchHostedService** - .NET Core Hosted Service, който работи на заден фон, докато уеб приложението е стартирано. Служи за актуализиране на стойностите на сензорите като прави API заявка към SensorDataController на всеки 10 секунди. След като получи данните за сензорите актуализира тези стойности на съответните потребителски сензори в базата данни.
3. **ISensorDataService** - **SensorDataService** – Връща всички сензори като генерира произволна стойност за всеки един от тях в рамките на неговия обхват и тип.

4. ***ISensorPropertyService*** – ***SensorPropertyService*** – Използва се за осъществяване на CRUD операциите за настройка на сензорите предоставени от SensorPropertyController.
5. ***ISensorService*** – ***SensorService*** - Използва се за осъществяване на CRUD операциите NA сензорите предоставени от SensorController.
6. ***IUserSensorService*** – ***UserSensorService*** - Използва се за осъществяване на CRUD операциите на потребителските сензори предоставени от UserSensorController.

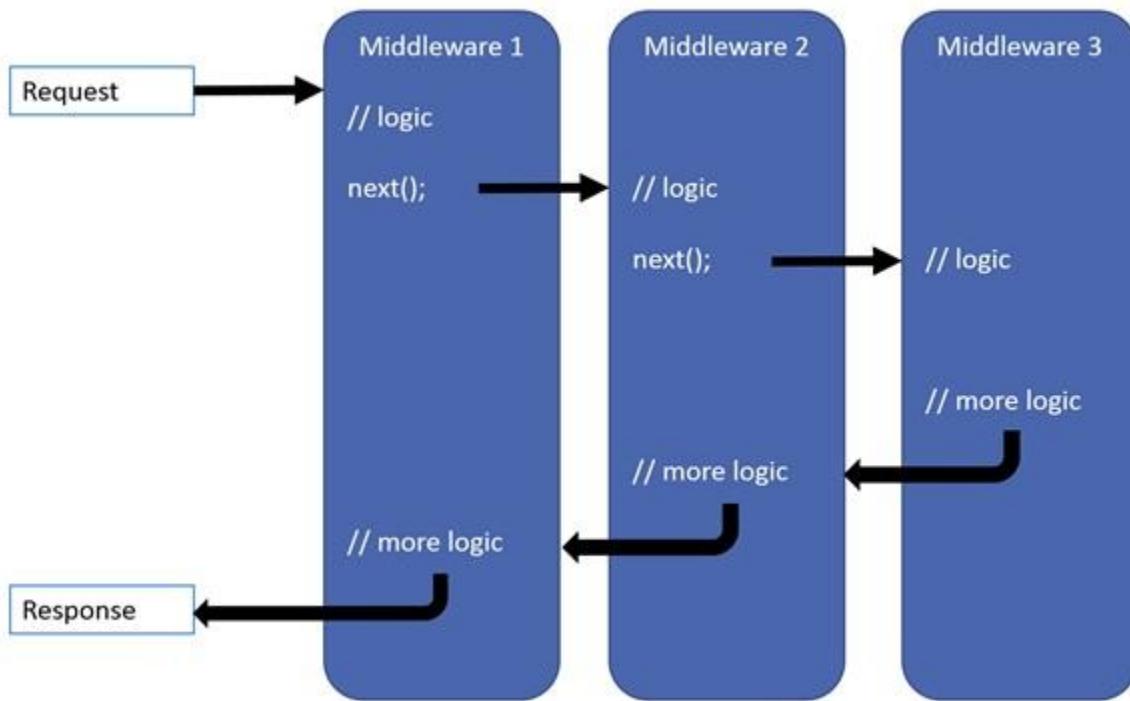


Фиг. 16. Service Layer на уеб приложението

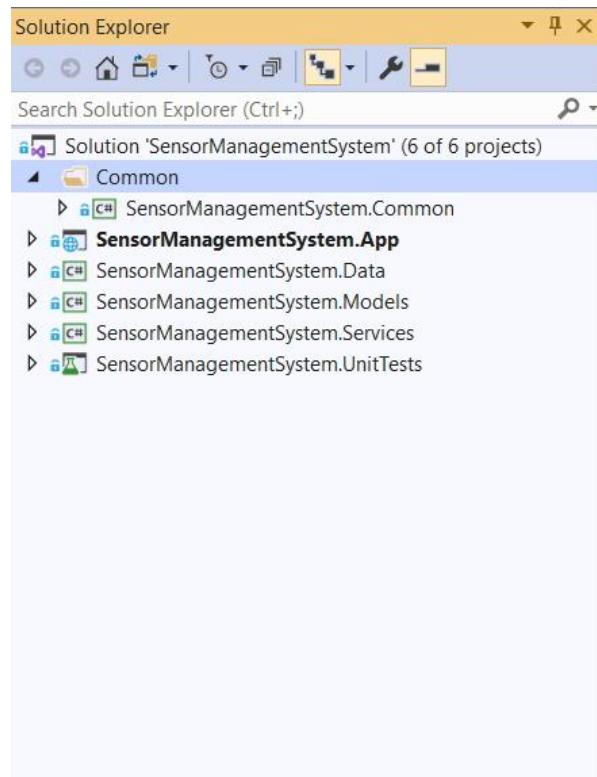
**Common** - .NET Core class library съдържаща логика, която може да бъде използвана от няколко проекта. Съдържа следното разпределение

1. ***Extensions*** – Съдържа класове разширяващи функционално други класове.
  - a. ***ClaimsPrincipalExtensions*** – Съдържа метод `GetId()`, който връща Id стойността на автентфикирания потребител.
  - b. ***IApplicationBuilderExtensions*** – Съдържа `UseErrorLogging()` и `UseErrorHandler()` методи, които свързват Middleware-ите към основните настройки на приложението.
  - c. ***JsonHelper*** – Съдържа помощни методи, които сериализират/десериализират Stream обекти в JSON и обратно.
2. ***Middlewares*** – Служат за допълнителна логика преди/след достигане на заявката до контролерите.
  - a. ***ErrorHandlingMiddleware*** – Ако възникне изключение пренасочва към `ErrorController` със съответния Action спрямо статус кода на отговора от заявката за „страницата не е намерена“ или за „грешка в сървъра“.
  - b. ***ErrorLoggingMiddleware*** – Служи за логване на възникнали изключения при HTTP заявките.
3. ***WebClients*** – Съдържа клиент, който служи за извършване на различни видове HTTP заявки.
  - a. ***HttpClient*** – Клас който съдържа различни методи извършващи HTTP заявки.

4. **Constants** – Клас с константи достъпни за целия Solution.



Фиг. 17. Последователност и начин на изпълнение на Middlewares в ASP.NET Core



Фиг. 18. Файлова структура на проектите представена във Visual Studio

## Логическа архитектура на мобилното приложението

Мобилното приложение на системата за управление и преглед на сензори се състои от 4 основни части разпределени в различни проекти – **Models, Services, Widgets, Main**

**Models** – Съдържа модели, които биват използвани за различни операции из приложението.

1. **Sensor.dart** – Модел, който дефинира полетата на един SensorDTO обект. Съдържа методи като:
  - a. **fromJson()** – Връща инстанция на обекта от JSON.
  - b. **toJson()** – Връща JSON от попълнен обект.
  - c. **All()** – Създава Resource обект, който ще върне лист от обекти.
  - d. **InitResourceByIdWithoutResponse(id)** – създава Resource обект, който не очаква обратно инстанция на обекта.
  - e. **initResourceByIdWithResponse(id)** – Създава Resource обект, който очаква обратно инстанция на обекта.
  - f. **initResourceByIdWithIntResponse(id)** – Създава Resource обект, който очаква обратно число като резултат.
  - g. **initWithJsonBody(payload)** – Създава Resource обект, който праща обект в HTTP заявката като Body параметър.
2. **SensorProperty.dart** Модел, който дефинира полетата на един SensorPropertyDTO обект. Съдържа методи като:
  - a. **fromJson()** – Връща инстанция на обекта от JSON.
  - b. **toJson()** – Връща JSON от попълнен обект.
  - c. **All()** – Създава Resource обект, който ще върне лист от обекти.
  - d. **InitResourceByIdWithoutResponse(id)** – създава Resource обект, който не очаква обратно инстанция на обекта.
  - e. **initResourceByIdWithResponse(id)** – Създава Resource обект, който очаква обратно инстанция на обекта.
  - f. **initWithJsonBody(payload)** – Създава Resource обект, който праща обект в HTTP заявката като Body параметър.

**Services** – Съдържа класове, които служат за извършване на HTTP заявки.

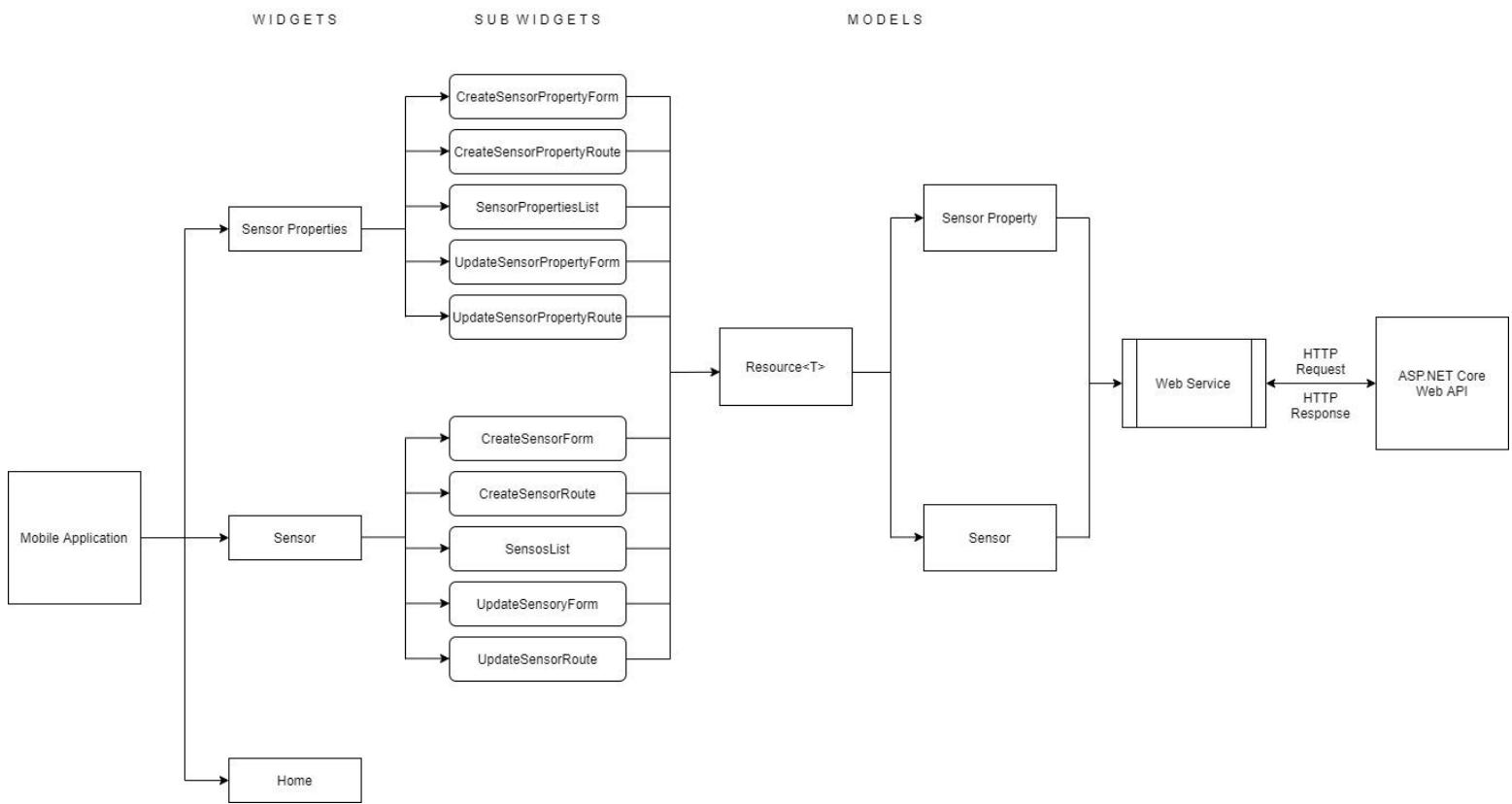
1. **Resource<T>** - Generic клас, който съдържа данни като URL, функция за parse-ване на обект/JSON-JSON/обект и body.
2. **WebService** – Връща различни видове Future като load, fetch, delete, send, update.

**Widgets** – Уиджети служещи за визуализация и бизнес логика. Разпределение на файловете:

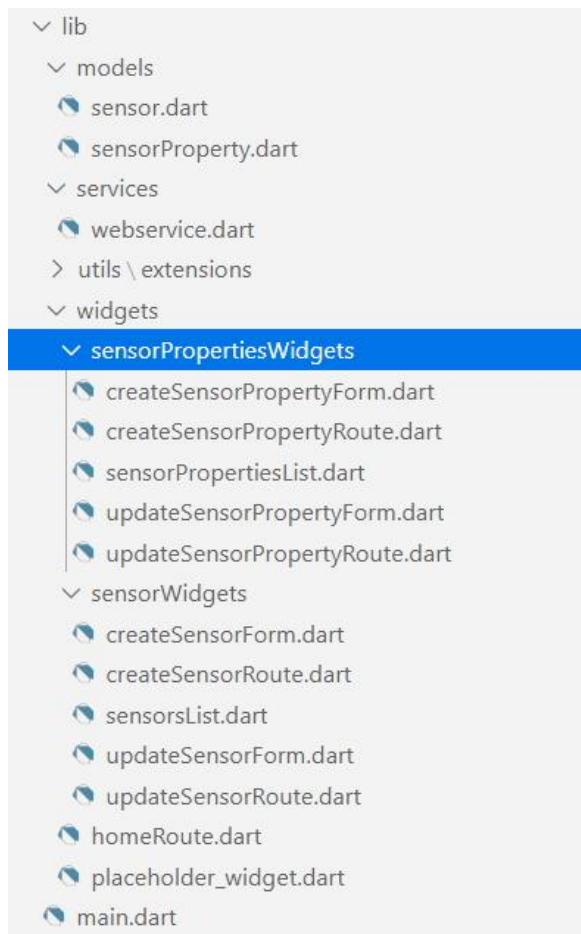
1. **SensorPropertiesWidgets** – Служат за логиката свързана с настройките на сензорите
  - a. **CreateSensorPropertyForm** – Уиджет, който служи за създаване на настройка за сензор и валидиране на формата.
  - b. **CreateSensorPropertyRoute** – Показва екрана за създаване на настройка за сензор.
  - c. **SensorPropertiesList** – Показва лист от наличните настройки за сензори като ги взима, чрез API заявка към SensorPropertyController.
  - d. **UpdateSensorPropertyForm** – Уиджет, който служи за редактиране на настройка за сензор и валидиране на формата.

- e. ***UpdateSensorPropertyRoute*** – Показва екрана за редактиране на настройка за сензор.
- 2. ***SensorWidgets*** – Служат за логиката свързана с сензорите
  - a. ***CreateSensorForm*** – Уиджет, който служи за създаване на настройка за сензор и валидиране на формата.
  - b. ***CreateSensorRoute*** – Показва екрана за създаване на сензор.
  - c. ***SensorsList*** – Показва лист от наличните сензори като ги взима, чрез API заявка към SensorController.
  - d. ***UpdateSensoryForm*** – Уиджет, който служи за редактиране на сензор и валидиране на формата.
  - e. ***UpdateSensorRoute*** – Показва екрана за редактиране на сензор.
- 3. ***HomeRoute*** – Служи за показване на екрана за navigation bar с бутони, които превключват между различните екрани.

**Main** – Стартовата точка на мобилното приложение.



Фиг. 19. Диаграма показваща принципа на работа на мобилното приложение



Фиг. 20. Файлова структура на приложението представена във Visual Studio Code

### Логическа архитектура на базата данни на приложението

Базата данни съдържа 4 таблици грижещи се за сензорите, настройките на сензорите и потребителските сензори. Данните за потребители се намират в таблици, които са автоматично генериране от Identity provider-а идващ от ASP.NET Core.

#### Sensors

В тази таблица се съхраняват наличните сензори от магазина за сензори. Има външен ключ към SensorProperties таблицата. Съдържа следните колони:

- Id (PK, int, not null) – Уникален идентификатор на сензора
- SensorPropertyId (FK, int, not null) – Идентификатор за настройката на сензора
- Description (nvarchar(255), not null) – Описание на сензора
- PollingInterval (int, not null) – Период на обновяване на сензора
- MinRangeValue (float, null) – Стойност на минималния обхват на сензора
- MaxRangeValue (float, null) – Стойност на максималния обхват на сензора
- CreatedOn (datetime2(7), null) – Дата на създаване на сензора
- ModifiedOn (datetime2(7), null) – Дата на последното обновяване на сензора

## SensorProperties

В тази таблица се съхраняват наличните настройки на сензорите от магазина за сензори. Съдържа следните колони:

- Id (PK, int, not null) – Уникален идентификатор за настройката на сензора
- MeasureType (nvarchar(450), not null) – Тип на сензора
- MeasureUnit (nvarchar(450), not null) – Мерна единица за типа на сензора
- IsSwitch (bit, not null) – Флаг показващ дали сензора е „превключвател“
- CreatedOn (datetime2(7), null) – Дата на създаване на сензора
- ModifiedOn (datetime2(7), null) – Дата на последното обновяване на сензора

## UserSensors

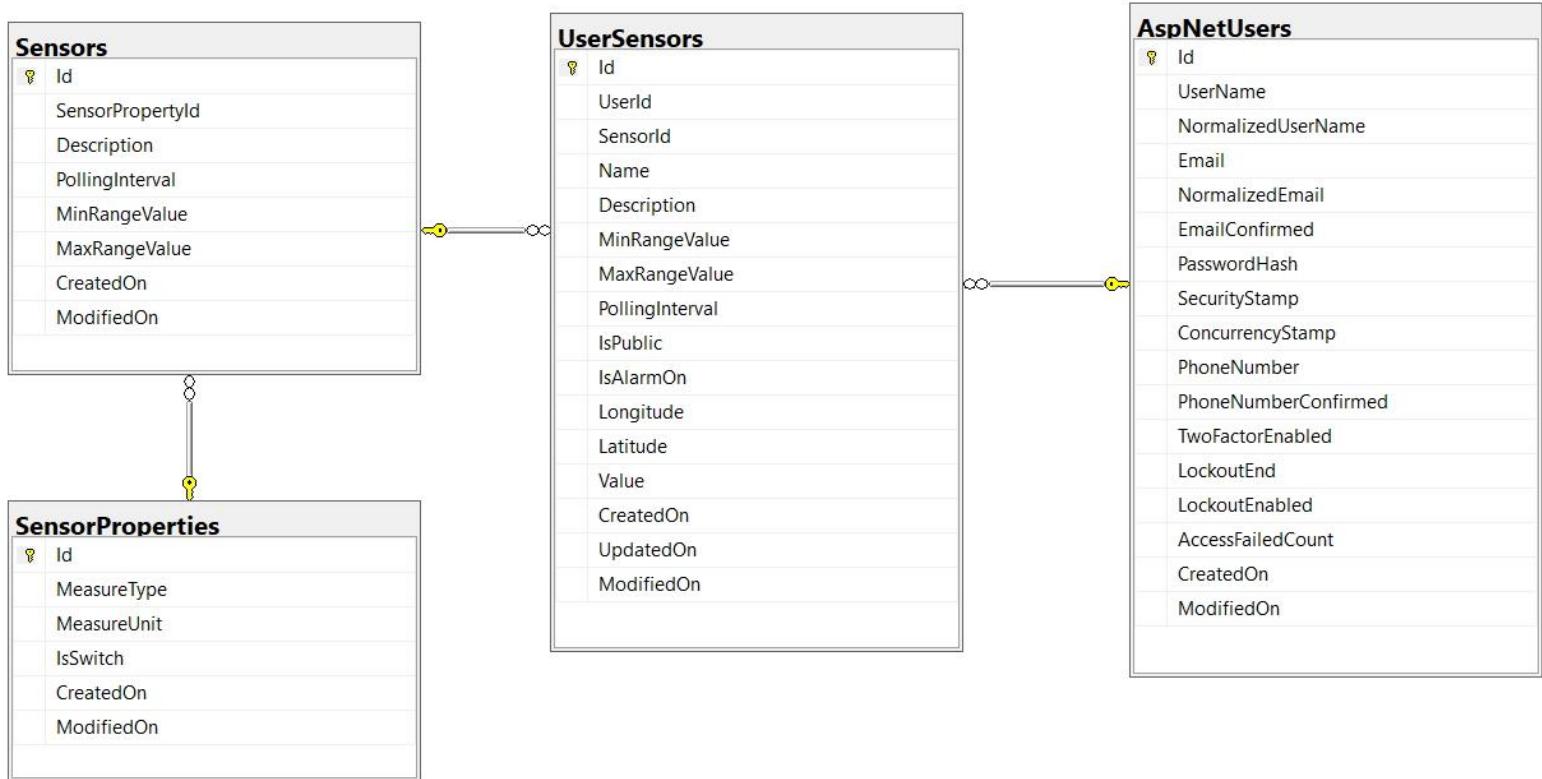
В тази таблица се съхраняват закупените потребителски сензори. Има външни ключове към Sensors и AspNetUsers таблиците. Съдържа следните колони:

- Id (PK, int, not null) – Уникален идентификатор на потребителския сензор
- UserId (FK, int, not null) – Идентификатор на потребителя притежаващ сензора
- SensorId (FK, int, not null) – Идентификатор на използвания сензор от магазина за сензори
- Name (nvarchar(255), not null) – Име на сензора
- Description (nvarchar(255), not null) – Описание на сензора
- MinRangeValue (float, null) – Минимален обхват на сензора зададен от потребителя
- MaxRangeValue (float, null) – Максимален обхват на сензора зададен от потребителя
- PollingInterval (int, not null) – Период за обновяване на сензора зададен от потребителя
- IsPublic (bit, not null) – Флаг показващ дали сензора е публичен или личен
- IsAlarmOn (bit, not null) – Флаг показващ дали сензора е с включена/изключена аларма
- Longitude (float, not null) – Географска дължина на сензора
- Latitude (float, not null) – Географска широта на сензора
- Value (nvarchar(255), null) – Последна стойност на сензора
- CreatedOn (datetime2(7), null) – Дата на създаване на сензора
- UpdatedOn(datetime2(7), null) – Дата на последното обновяване на стойността на сензора
- ModifiedOn (datetime2(7), null) – Дата на последното обновяване на сензора

## AspNetUsers

В тази таблица се съхраняват потребителите на приложението. Съдържа следните колони (тези, които реално се използват като останалите просто идват от ASP.NET Core Identity):

- Id (PK, int, not null) – Уникален идентификатор на потребителя
- Email (nvarchar(256, null)) – Имейл адрес на потребителя
- EmailConfirmed (bit, null) – Флаг показващ дали потребителът е потвърдил акаунта си, чрез линка изпратен на имейл адреса използван при регистрация
- CreatedOn (datetime2(7), null) – Дата на създаване на потребителя



Фиг. 21. Цялостна структура на базата данни

## V. Програмна реализация

В тази глава ще бъдат разгледани някои от по-важните функционалности на системата. Ще бъде показан част от кода, който извършва определени действия заедно с кратко описание какъв е процеса и последователността на изпълнение.

### Регистрация и вход

За да може да бъде използвана системата за управление и преглед на сензори, трябва да имаме регистриран потребител. За целта, когато потребителят отвори системата през ще вижда бутон за регистрация („Registration“). При натискането му ще се появи HTML страница с полета, които при попълване публикуват форма, водеща към OnPost екшън метод на съответната Register Razor Page. За всяко поле се използват ASP.NET Core tag-helpers, които улесняват model binding-а и валидацията на полетата.

Част от кода показващ формата:

```
<form action="" method="post">
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="input-group mb-3">
        <input asp-for="Input.Email" type="email" class="form-control" placeholder="Email">
        <div class="input-group-append">
            <div class="input-group-text">
                <span class="fas fa-envelope"></span>
            </div>
        </div>
    </div>
    <div class="input-group mb-3">
        <input asp-for="Input.Password" type="password" class="form-control" placeholder="Password">
        <div class="input-group-append">
            <div class="input-group-text">
                <span class="fas fa-lock"></span>
            </div>
        </div>
    </div>
    <div class="row">
        <div class="col-8">
        </div>
        <!-- /.col -->
        <div class="col-4">
            <button type="submit" class="btn btn-primary btn-block">Register</button>
        </div>
        <!-- /.col -->
    </div>
</form>
```

При публикуване на формата влизаме в OnPost екшъна на Register Razor Page-а. Там първо има проверка дали изпратения модел от формата е валиден. Ако проверката е успешна проверяваме дали има вече съществуващ потребител регистриран с този имейл адрес. Ако има правим проверка дали времето за активация от имейл адреса (2 мин.) на този потребител е изтекло и ако е – този потребител бива изтрит, така проблема с използване на чужд имейл бива решен. Ако не е изтекло връщаме грешка със съобщение, че потребителя трябва да активира акаунта си от имейл адреса използван при регистрация.

```
if (ModelState.IsValid)
{
    UserEntity existingUser = await _userManager.FindByEmailAsync(Input.Email);
    if (existingUser != null && !existingUser.EmailConfirmed)
    {
        if (existingUser.CreatedOn.Value.AddMinutes(EmailActivationExpireTime) < DateTime.UtcNow)
        {
            // Continue registration
            await _userManager.DeleteAsync(existingUser);
        }
        else
        {
            ModelState.AddModelError("EmailToBeConfirmed", "This email is already taken. If it is yours, please go and activate your account from there!");
        }
    }
}
```

След като потребителя не съществува то следва да бъде създаден. За целта създаваме нов обект от тип UserEntity и генерираме линк с токен, чрез който потребителя може да потвърди акаунта си. Линка бива изпратен на използвания имейл адрес въведен от потребителя.

```
var user = new UserEntity { UserName = Input.Email, Email = Input.Email };

var result = await _userManager.CreateAsync(user, Input.Password);

if (result.Succeeded)
{
    _logger.LogInformation("User created a new account with password.");
    var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
    code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));

    var callbackUrl = Url.Page(
        "/Account/ConfirmEmail",
        pageHandler: null,
        values: new { userId = user.Id, code },
        protocol: Request.Scheme);

    await _emailService.SendAsync(user.Email, "Successfull registration", $"Confirm your account by clicking on this link:\n\n{HtmlEncoder.Default.Encode(callbackUrl)}\n\nLink will be active only for {EmailActivationExpireTime} minutes.");
}
```

При натискане на линка изпратен на имейл адреса на потребителя, той бива пренасочен към OnGet екшън метод на ConfirmEmail Razor Page. След това има проверка дали потребителя съществува и дали кода изпратен от линка е валиден. Ако е валиден – флаг-ът показващ дали потребителя е с потвърдил имейл адреса си става “true”.

---

```
if (userId == null || code == null)
{
    throw new InvalidOperationException($"Error confirming email for user with ID '{userId}'");
}

var user = await _userManager.FindByIdAsync(userId);
if (user == null)
{
    return NotFound($"Unable to load user with ID '{userId}'.");
}

code = Encoding.UTF8.GetString(WebEncoders.Base64UrlDecode(code));
var result = await _userManager.ConfirmEmailAsync(user, code);

if (!result.Succeeded)
{
    throw new InvalidOperationException($"Error confirming email for user with ID '{userId}':");
}
```

След това потребителя може да се автентикира в системата успешно.

## Търсене и филтриране на потребителски сензори

След като потребителя вече се е вписал успешно в системата и е закупил сензори, то той може да ги преглежда като има възможност да приложи филтриране по различни критерии и да търси по име на сензор. За целта в кода използваме 2 изгледа – един, на който имаме опциите за филтриране, търсене и превключване на страниците и втори частичен изглед, където поставяме резултата от филтрацията и търсениято. Част от HTML кода за филтрите, pagination-а и търсенето.

```
<div class="form-group">
    <label>Filter by Sensor Measure Type:</label>
    <select asp-for="MeasureType" asp-items="Model.MeasureTypes"
        class="form-control select2bs4" style="width: 100%;">
        <option value="null">-- All --</option>
    </select>
</div>

<div class="col-md-3">
    <div class="form-group">
        <label>Filter by Alarm status:</label>
        <select asp-for="IsAlarmOn"
            class="form-control select2bs4" style="width: 100%;">
            <option value="null">-- All --</option>
            <option value="true">Alarm On</option>
            <option value="false">Alarm Off</option>
        </select>
    </div>
</div>

<div class="col-md-3">
    <label>Search by Sensor Name:</label>
    <div class="input-group mb-3 form-group">
        <input type="text" id="search-filter-input" class="form-control rounded-0">
        <span class="input-group-append">
            <button type="button" id="search-filter-button" class="btn btn-info btn-flat">Go!</button>
        </span>
    </div>
</div>

<div class="row">
    <button id="previous-page-button" type="button" class="btn btn-flat btn-outline-primary"
        @(Model.UserSensors.HasPreviousPage ? string.Empty : "disabled")>
        Previous
    </button>
    &nbsp;&nbsp;
    <button id="next-page-button" type="button" class="btn btn-flat btn-outline-primary"
        @(Model.UserSensors.HasNextPage ? string.Empty : "disabled")>
        Next
    </button>
</div>
```

При избор на филтър или търсене през jQuery се „слуша“ за промяна на някое от полетата за и се изпълнява AJAX заявка. Ако заявката е успешна се връща частичен изглед от UserSensor контролера от екшън метода „ReloadUserSensorsTable“ и се извиква reloadTable функцията, която взима необходимите стойности, за да бъде подгответен при следващия приложен филтър или търсене от потребителя.

Частта от кода извършващ AJAX заявката:

```
$(document).ready(function () {
    const reloadTable = function () {
        let measureType = $("#MeasureType").val();
        let isPublic = $("#IsPublic").val();
        let isAlarmOn = $("#IsAlarmOn").val();
        let searchTerm = $("#search-filter-input").val();
        var pageIndex = $("#PageIndex").val();
        var requestUrl = '/UserSensor/ReloadUserSensorsTable';

        requestUrl += '?pageIndex=' + pageIndex;

        if (measureType != 'null' && measureType != undefined) {
            requestUrl += '&measureType=' + measureType;
        }
        if (isPublic != 'null' && isPublic != undefined) {
            requestUrl += '&isPublic=' + isPublic;
        }
        if (isAlarmOn != 'null' && isAlarmOn != undefined) {
            requestUrl += '&isAlarmOn=' + isAlarmOn;
        }
        if (searchTerm != '' && searchTerm != undefined) {
            requestUrl += '&searchTerm=' + searchTerm;
        }
    }

    $.ajax({
        type: "GET",
        url: requestUrl,
        dataType: "html",
        success: function (response) {
            $("#user-sensor-cards").html(response);
            var hasPreviousPage = $("#HasPreviousPage").val().toLowerCase() === 'true';
            var hasNextPage = $("#HasNextPage").val().toLowerCase() === 'true';
            if (hasPreviousPage) {
                $("#previous-page-button").removeAttr("disabled");
            } else {
                $("#previous-page-button").attr("disabled", true);
            }
            if (hasNextPage) {
                $("#next-page-button").removeAttr("disabled");
            } else {
                $("#next-page-button").attr("disabled", true);
            }
            handleGaugeUpdate();
        }
    });
});
```

```
$("#MeasureType").change(reloadTable);
$("#IsPublic").change(reloadTable);
$("#IsAlarmOn").change(reloadTable);
$("#search-filter-input").change(reloadTable);

$("#search-filter-input").keyup(function (event) {
    if (event.keyCode === 13) {
        reloadTable();
    }
});

$("#search-filter-button").click(reloadTable);
$("#previous-page-button").click(function () {
    let pageIndex = $("#PageIndex").val();
    $("#PageIndex").val(parseInt(pageIndex) - 1);
    reloadTable();
});
$("#next-page-button").click(function () {
    let pageIndex = $("#PageIndex").val();
    $("#PageIndex").val(parseInt(pageIndex) + 1);
    reloadTable();
});
```

Частта от кода, представляваща екшън метода в UserSensor контролера

```
0 references | Krasimir Etov, 9 days ago | 1 author, 3 changes
public async Task<IActionResult> ReloadUserSensorsTable([FromQuery] int pageIndex, string measureType = null, bool?
    isPublic = null, bool? isAlarmOn = null, string searchTerm = null)
{
    var userId = User.GetId();
    var model = await _userSensorService
        .GetAllFilteredAsync<UserSensorTableViewModel>(userId, pageIndex, PageSize, measureType, isPublic, isAlarmOn,
        searchTerm);

    return PartialView("_UserSensorsTable", model);
}
```

Част от кода представляващ филтрирането на данните чрез използване на LINQ и Entity Framework Core

```
private IQueryable<UserSensorEntity> GetFilteredQuery(IQueryable<UserSensorEntity> userSensorEntitiesQuery, string measureTypeFilter = null, bool? isPublic = null, bool? isAlarmOn = null, string searchTerm = null)
{
    if (!string.IsNullOrEmpty(searchTerm) || !string.IsNullOrWhiteSpace(searchTerm))
    {
        userSensorEntitiesQuery = userSensorEntitiesQuery
            .Where(x => x.Name.Contains(searchTerm) || x.Description.Contains(searchTerm));
    }

    if (!string.IsNullOrEmpty(measureTypeFilter) || !string.IsNullOrWhiteSpace(measureTypeFilter))
    {
        userSensorEntitiesQuery = userSensorEntitiesQuery
            .Where(x => x.Sensor.SensorProperty.MeasureType == measureTypeFilter);
    }

    if (isPublic.HasValue)
    {
        userSensorEntitiesQuery = userSensorEntitiesQuery.Where(x => x.IsPublic == isPublic.Value);
    }

    if (isAlarmOn.HasValue)
    {
        userSensorEntitiesQuery = userSensorEntitiesQuery.Where(x => x.IsAlarmOn == isAlarmOn.Value);
    }

    return userSensorEntitiesQuery;
}
```

---

## Запис в базата данни

Записът на информация в базата данни става през Service слоя на приложението. Той бива извикан през контролерите. Част от кода на SensorController, който използва CreateAsync метод на SensorService:

```
[HttpPost]
0 references | Krasimir Etov, 4 days ago | 1 author, 3 changes
public async Task<IActionResult> Post([FromBody] SensorDTO payload)
{
    var result = ValidateDTO(payload);

    if (!string.IsNullOrEmpty(result))
    {
        return BadRequest(result);
    }

    await this._sensorService.CreateAsync(payload);
    await _notificationManager.SendToAuthenticatedUsersAsync("New sensor has been added to the store!");

    return Ok();
}
```

---

SensorService класа от своя страна прави мапинг от DTO обекта към Database обекта използвайки Automapper:

```
2 references | Krasimir Etov, 23 days ago | 1 author, 1 change
private IEnumerable<T> MapToDTO<T>(IEnumerable<SensorEntity> sensorEntities)
{
    List<T> sensorDTOS = new List<T>();

    foreach (var sensorEntity in sensorEntities)
    {
        sensorDTOS.Add(_mapper.Map<T>(sensorEntity));
    }

    return sensorDTOS;
}

1 reference | Krasimir Etov, 23 days ago | 1 author, 1 change
private T MapToDTO<T>(SensorEntity sensorEntity)
{
    T sensorDTO = _mapper.Map<T>(sensorEntity);

    return sensorDTO;
}

1 reference | Krasimir Etov, 23 days ago | 1 author, 1 change
private SensorEntity MapToEntity<T>(T sensorDTO)
{
    return _mapper.Map<SensorEntity>(sensorDTO);
}
```

След което записва модела в базата използвайки Entity Framework Core DbContext, ако валидацията е минала успешно.

```
2 references | Krasimir Etov, 9 days ago | 1 author, 5 changes
public async Task CreateAsync(SensorDTO sensorDTO)
{
    var sensorEntity = MapToEntity(sensorDTO);

    var existingEntity = await _dbContext.Sensors
        .Include(x => x.SensorProperty)
        .FirstOrDefaultAsync(x => x.SensorPropertyId == sensorDTO.SensorPropertyId && x.MinRangeValue ==
            sensorDTO.MinRangeValue && x.MaxRangeValue == sensorDTO.MaxRangeValue && x.PollingInterval ==
            sensorDTO.PollingInterval);

    if (existingEntity != null)
    {
        throw new Exception($"Sensor with Measure Type: {existingEntity.SensorProperty.MeasureType}, Polling
            Interval: {existingEntity.PollingInterval}, Min Range: {existingEntity.MinRangeValue} and Max Range:
            {existingEntity.MaxRangeValue} already exists!");
    }

    await this._dbContext.Sensors
        .AddAsync(sensorEntity);

    await this._dbContext.SaveChangesAsync();
}
```

## Създаване на сензор от мобилното приложение

При натискане на бутона за създаване на сензор отиваме към CreateSensorRoute, което е StatelessWidget. Там се зарежда CreateSensorForm класа като body на страницата. Кода на CreateSensorRoute:

```
class CreateSensorRoute extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(title: Text('Create Sensor')),
            body: CreateSensorForm()); // Scaffold
    }
}
```

При инициализирането си CreateSensorForm извършва Web API заявка към уеб приложението, за да вземе данни за сензорите и техните настройки. Част от кода извършващ тези действия:

```
class _CreateSensorFormState extends State<CreateSensorForm> {
    final GlobalKey<FormState> _formKey = GlobalKey<FormState>();
    Sensor _model = Sensor();
    List<Sensor> _sensors = List<Sensor>();
    List<SensorProperty> _sensorProperties = List<SensorProperty>();
    SensorProperty _selectedSensorPropertyFromDropdown;
    String _tempMinRangeValue = "0";
    String _tempMaxRangeValue = "0";
    String _tempPollingInterval = "0";
    Future _loadSensorProperty;

    @override
    void initState() {
        super.initState();
        _loadSensorProperty = _populateSensorProperties();
        _populateSensors();
    }

    Future _createSensor() {
        return WebService().send(SensorinitWithJsonBody(_model));
    }

    Future _populateSensorProperties() {
        return WebService().load(SensorProperty.all).then((sensorProperties) => {
            setState(() {
                _sensorProperties = sensorProperties;
                _selectedSensorPropertyFromDropdown = _sensorProperties.first;
                _model.sensorPropertyId = _selectedSensorPropertyFromDropdown.id;
            });
        });
    }

    Future _populateSensors() {
        return WebService().load(Sensor.all).then((sensors) => {
            setState(() => {_sensors = sensors});
        });
    }
}
```

След като данните са взети е време да се извика Build метода, който ще създаде уиджет обекта. В тази част на кода фигурират създаването на полетата и тяхната валидация:

```
@override
Widget build(BuildContext context) {
    return FutureBuilder(
        future: _loadSensorProperty,
        builder: (context, snapshot) {
            if (snapshot.hasData) {
                return Form(
                    key: _formKey,
                    child: SingleChildScrollView(
                        child: Column(
                            mainAxisAlignment: MainAxisAlignment.start,
                            children: <Widget>[
                                TextFormField(
                                    decoration: InputDecoration(
                                        labelText: 'Description',
                                        icon: Icon(Icons.description)), // InputDecoration
                                    validator: (value) {
                                        if (value.isEmpty) {
                                            return 'Description is required';
                                        }
                                        if (value.length < 3) {
                                            return 'Description must have at least 3 characters!';
                                        }
                                        return null;
                                    },
                                    onChanged: (newValue) {
                                        _model.description = newValue;
                                    },
                                ), // TextFormField
                                TextFormField(
                                    keyboardType: TextInputType.numberWithOptions(),
                                    decoration: InputDecoration(
                                        labelText: 'Polling Interval',
                                        icon: Icon(Icons.timer)), // InputDecoration
                                    validator: (value) {
                                        if (value.isEmpty) {
                                            return 'Polling Interval is required';
                                        }
                                    },
                                ),
                            ],
                        ),
                    ),
                );
            }
        },
    );
}
```

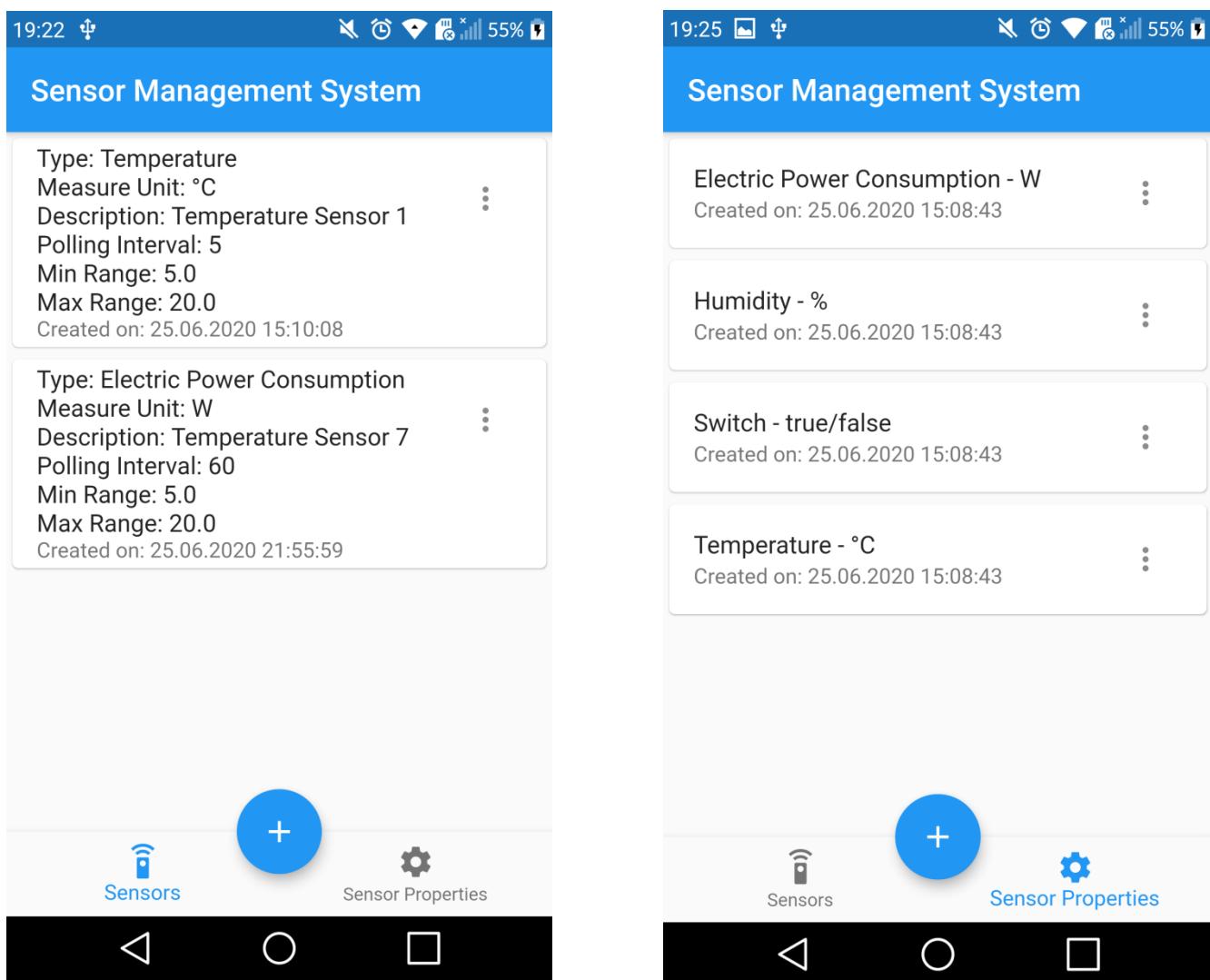
```
        onchanged: (newValue) {
            setState(() {
                _tempMaxRangeValue = newValue;
            });
            _model.maxRangeValue = _tempMaxRangeValue;
        },
    ), // TextFormField
Padding(
    padding: const EdgeInsets.symmetric(vertical: 16.0),
    child: RaisedButton(
        onPressed: () async {
            // Validate returns true if the form is valid, or false
            // otherwise.
            if (_formKey.currentState.validate()) {
                // If the form is valid, display a Snackbar.
                await _createSensor().then((value) {
                    Scaffold.of(context).showSnackBar(
                        SnackBar(content: Text('Created sensor')));
                }).catchError(() {
                    Scaffold.of(context).showSnackBar(SnackBar(
                        content: Text(
                            'Error when trying to create sensor'))); // Text // SnackBar
                });
            },
            child: Text('Create'),
        ), // RaisedButton
    ), // Padding
], // <Widget>[]
)); // Column // SingleChildScrollView // Form
} else {
    return CircularProgressIndicator();
}
}); // FutureBuilder
}
}
```

## VI. Ръководство на потребителя

В тази глава ще бъде представена функционалността на системата с прикачени снимки от нея.

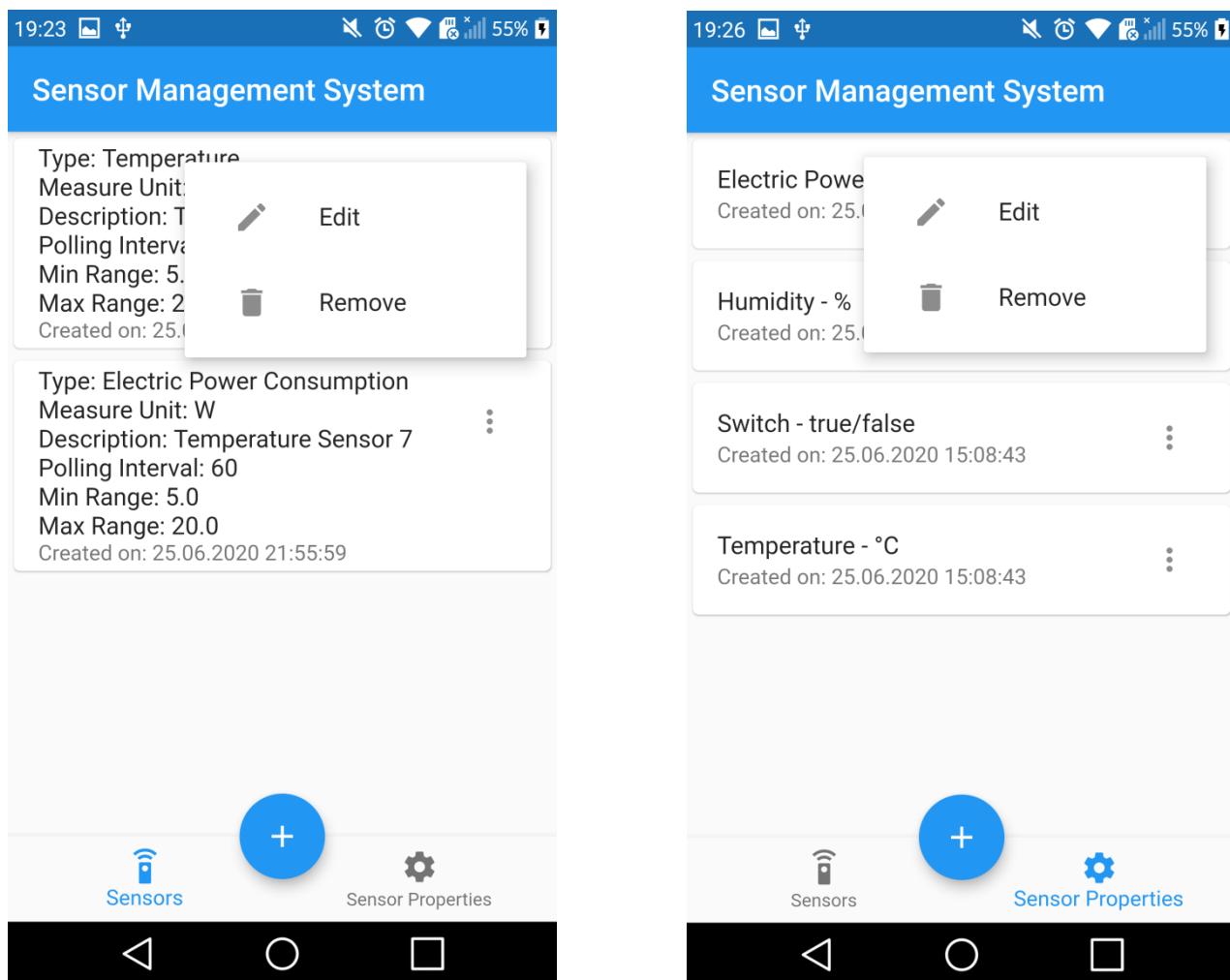
### Начална страница на мобилното приложение

При отваряне на мобилното приложение се отваря и началната страница. Тя сърдържа два таба за навигация, които превключват между два екрана – Sensors и Sensor Properties. При избран таб потребителя вижда списък от вече добавените сензори/настройки на сензорите. По средата на экрана има и бутон за бърз достъп, който при натискане отваря еcran за създаване на нов сензор/настройка за сензор в зависимост от отворения таб.



Фиг. 22. Начален еcran на мобилното приложение

На всеки елемент от листа има прикачен бутон за меню, което открива 2 опции – за редактиране и за премахване на сензор/настройка на сензор. При избиране на бутона „Remove“ елемента бива премахван от списъка.

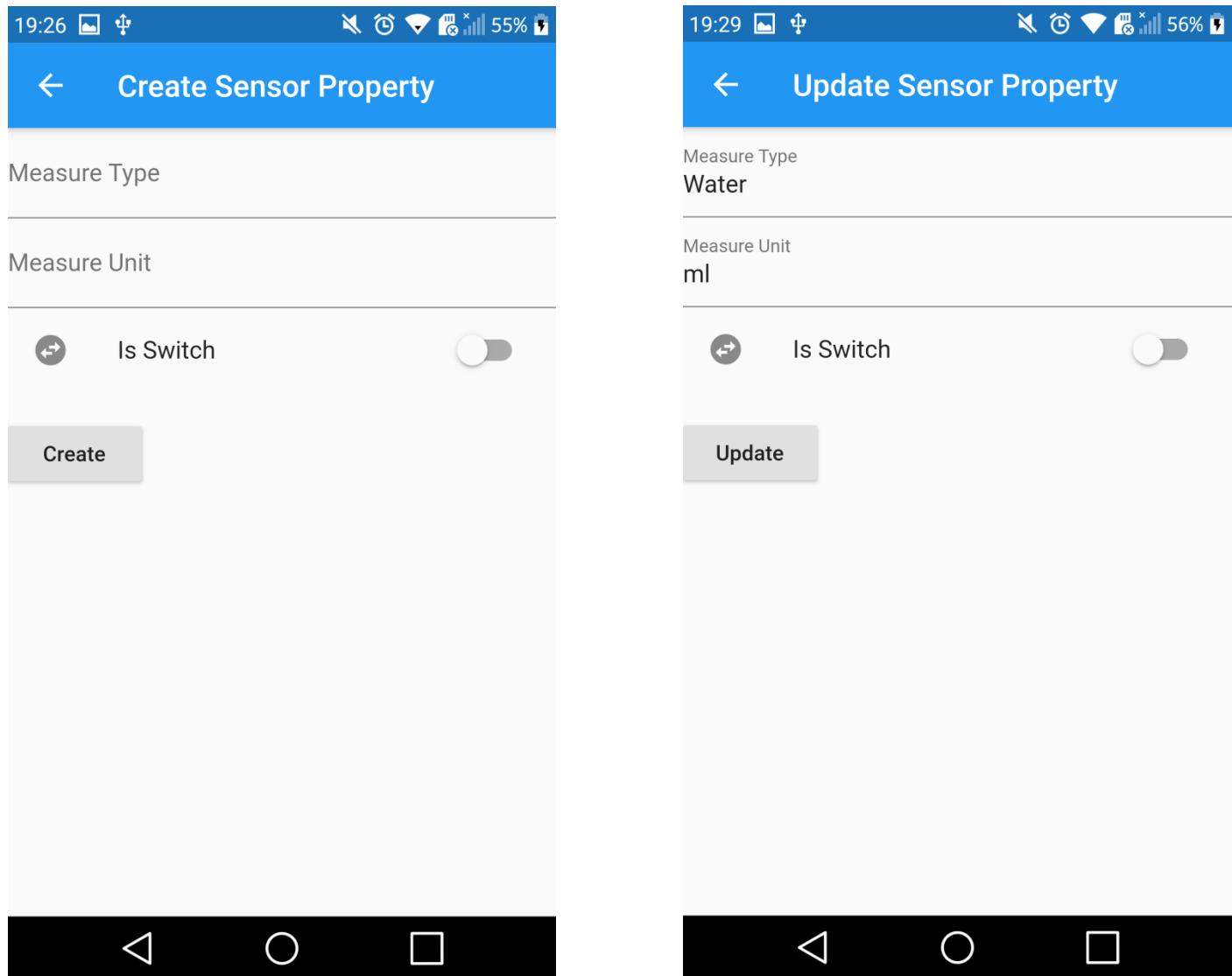


Фиг. 23. Начален екран с отворен меню бутон показващ допълнителни настройки

## Добавяне и редактиране на настройка на сензор. Валидация

### Добавяне/редактиране на настройка на сензор

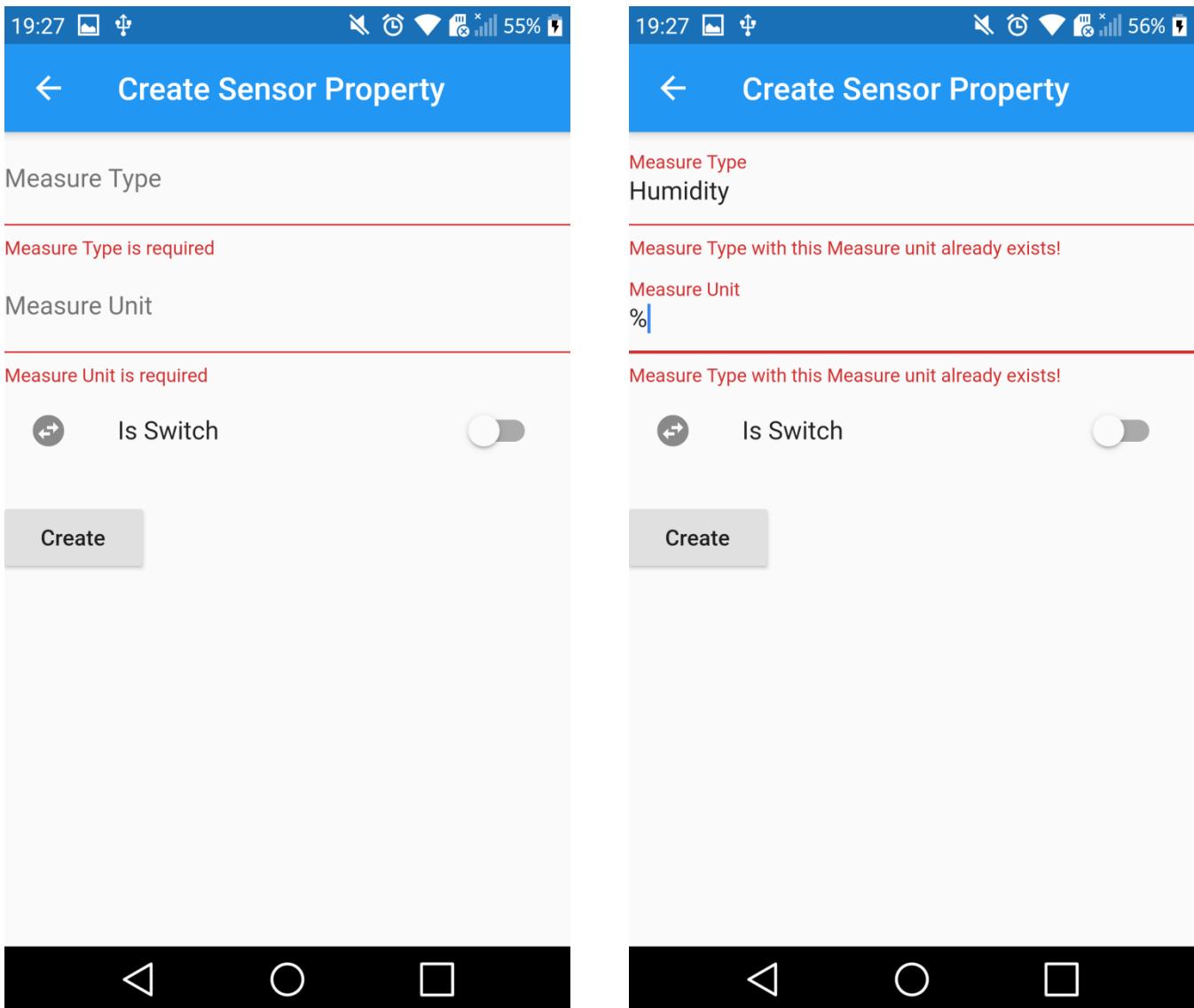
При натискане на кръглия бутон за бърз достъп докато е избран таб „Sensor Properties” се отваря екран с полета за създаване на настройка на сензор. При редактиране се използва същия екран, но с разликата, че полетата са попълнени с вече съществуващи данни.



Фиг. 24. Екран при създаване/редактиране на настройка на сензор

## Валидация

При създаване/редактиране на настройка на сензор има валидация на полетата, която при натискане на бутона ("Submit", "Update") не публикува формата, а показва грешките в червен текст видим за потребителя.

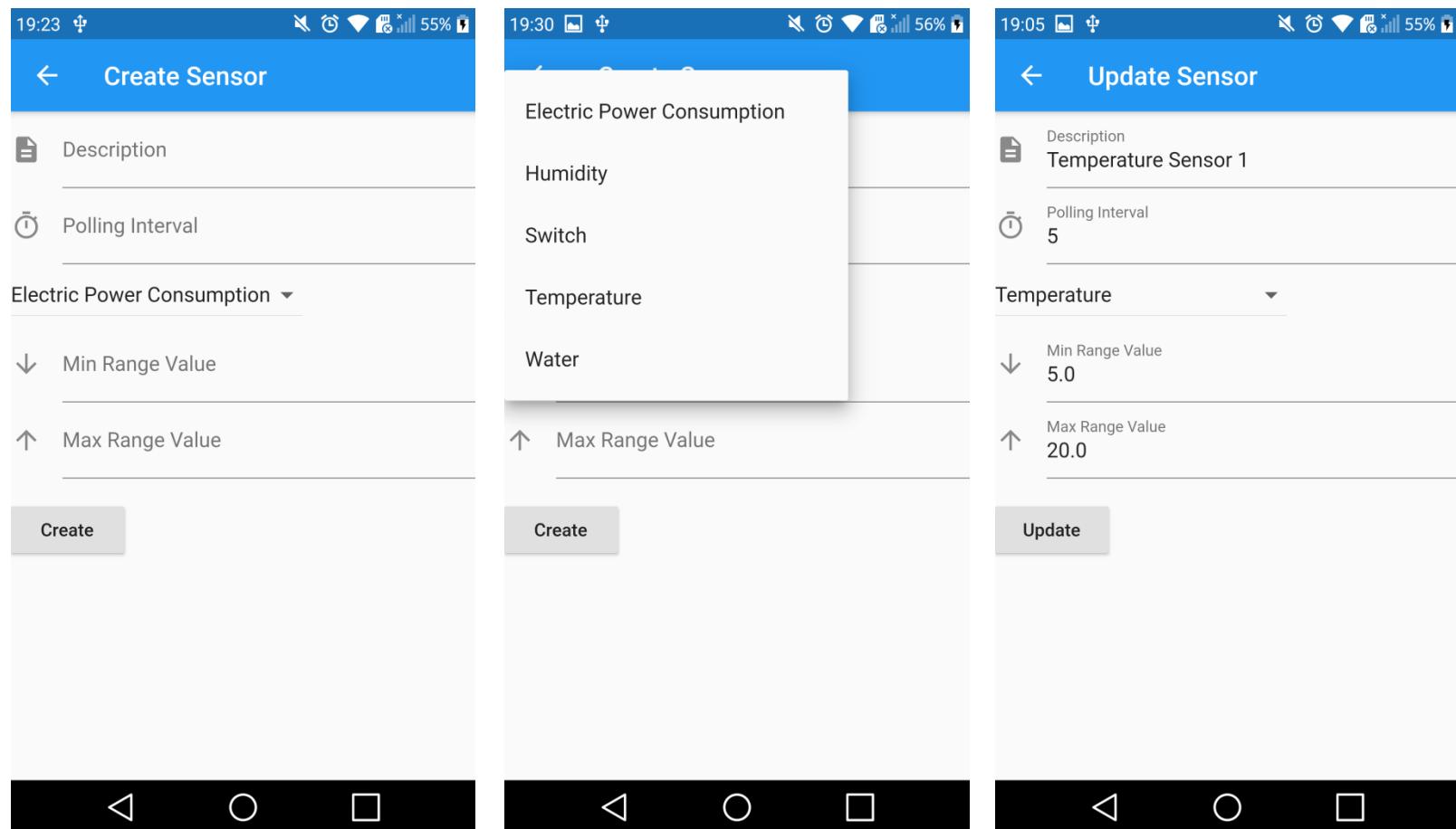


Фиг. 25. Валидация при създаване/редактиране на настройка на сензор.

## Добавяне и редактиране на сензор. Валидация

### Добавяне/редактиране на сензор

При натискане на кръглия бутон за бърз достъп докато е избран таб „Sensors” се отваря еcran с полета за създаване на сензор. При редактиране се използва същия еcran, но с разликата, че полетата са попълнени с вече съществуващи данни.



Фиг. 26. Еcran при създаване/редактиране на сензор

## Валидация

При създаване/редактиране на сензор има валидация на полетата, която при натискане на бутона ("Submit", "Update") не публикува формата, а показва грешките в червен текст видим за потребителя.

The figure consists of two side-by-side screenshots of a mobile application interface titled "Create Sensor".

**Left Screenshot:**

- Description:** The field contains "a". Below it, a red error message says "Description must have at least 3 characters!".
- Polling Interval:** The field contains "5". Below it, a red error message says "Polling Interval is required".
- Electric Power Consumption:** A dropdown menu is open, showing:
  - Min Range Value:** Set to "30". Below it, a red message says "Min Range shold be less than Max Range!".
  - Max Range Value:** Set to "20". Below it, a red message says "Max Range should be more than Min Range!".
- Create:** A button at the bottom left.

**Right Screenshot:**

- Description:** The field contains "Already exists".
- Polling Interval:** The field contains "5". Below it, a red message says "Sensor with this Polling Interval, Min and Max Range alrea...".
- Temperature:** A dropdown menu is open, showing:
  - Min Range Value:** Set to "5". Below it, a red message says "Sensor with this Polling Interval, Min and Max Range alrea...".
  - Max Range Value:** Set to "20". Below it, a red message says "Sensor with this Polling Interval, Min and Max Range alrea...".
- Create:** A button at the bottom left.

Фиг. 27. Валидация при създаване/редактиране на сензор

## Начална страница на уеб приложението

При отваряне на уеб приложението на потребителя се показва началната страница. Тя съдържа статистика показваща броя на сензорите в магазина, публични се потребителски сензори и броя на регистрираните потребители. Ако потребителът е вписан в системата се показва и допълнителна статистика показваща броя на притежаваните от него сензори. Под статистика има Google карта, която показва местоположението на публичните сензори с кратка информация за тях.

Sensor Management System

[Register](#) [Login](#)

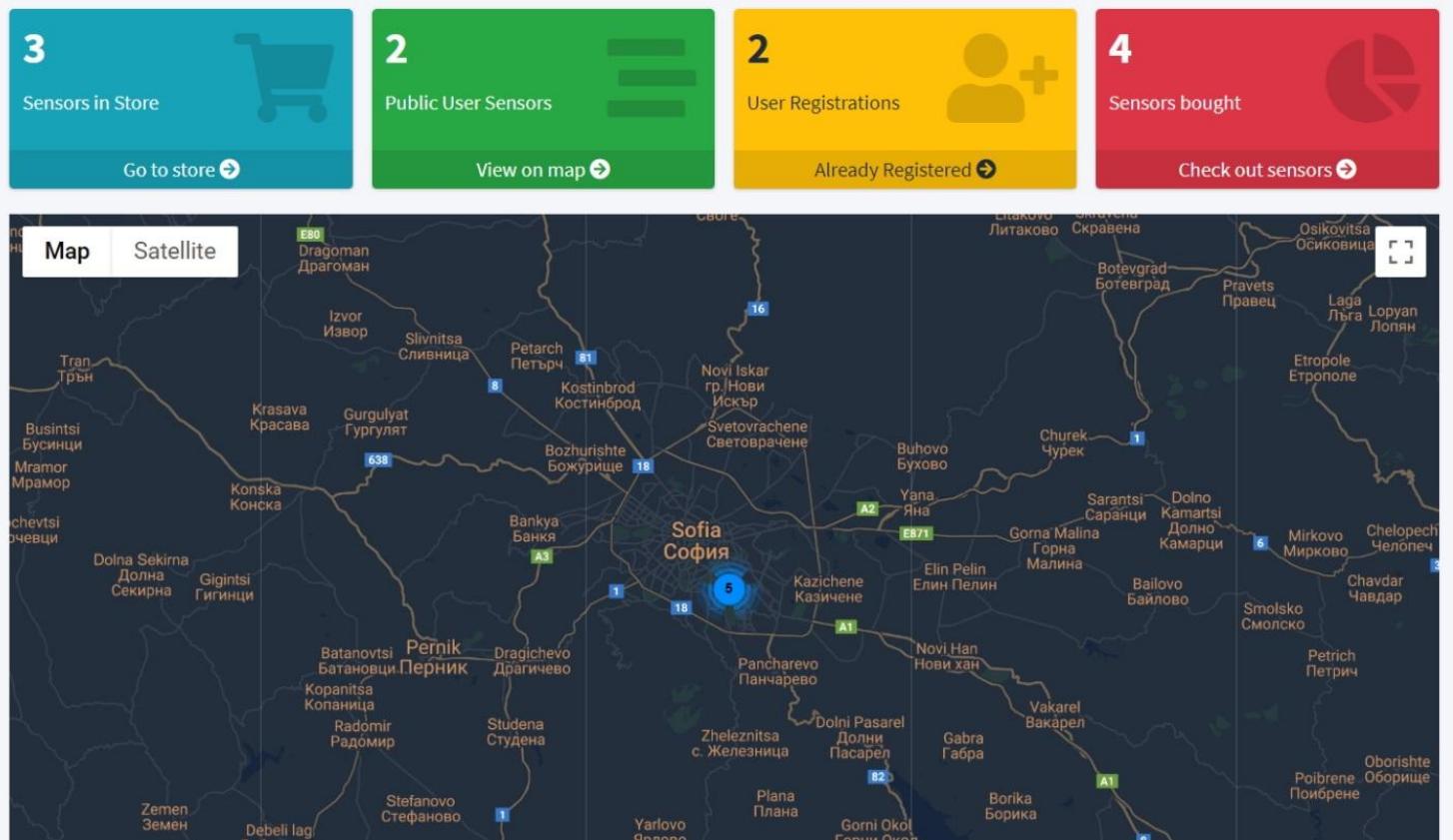
The screenshot shows the homepage of the Sensor Management System. At the top, there are three summary cards:

- Sensors in Store**: Shows 3 sensors in a store, represented by a shopping cart icon. Below it is a button: [Go to store](#).
- Public User Sensors**: Shows 2 public user sensors, represented by a double bar chart icon. Below it is a button: [View on map](#).
- User Registrations**: Shows 2 user registrations, represented by a person icon with a plus sign. Below it is a button: [Register now](#).

Below the cards is a map of a city area. The map includes several location markers and labels:

- Kindergarten 34 Brezichka (ул. „Крум Кючук“)
- MINNINO-GEOLISKI UNIVERSITET "SV. IVAN RILSKI" (Минно-геологически университет "Св. Иван Рилски")
- Muzeiko (Музеико)
- Oncology Hospital (УСБАЛ по онкология ЕАД гр. София)
- High School 8 Vasili Levski (8-мо СУ „Васил Левски“)
- Baby store (Бебешка градина)
- Sports complex "Peace and Friendship" (Спортен комплекс „Мир и дружба“)
- High College of Telecommunications (Висше училище по телекомуникации и...)
- Technical University (Технически Университет - София)
- High School 55 Petko Karavelov (55 СУ „Петко Каравелов“)

Фиг. 28. Начална страница на приложението



Фиг. 29. Начална страница при потребител вписал се в системата

## Регистрация и вход. Валидация

### Регистрация и вход

При натискане на “Register” – “Login” бутон от навигационния таб се показва екран с полета за попълване на имейл адрес и парола.

Register a new membership

Check email for activation link after registration

✉

🔒

Register

[I already have a membership](#)

Sign in to start your session

✉

🔒

Remember me?

Sign In

[Register a new membership](#)

Фиг. 30. Екрани за регистрация и вход в системата

## Валидация

The figure consists of two side-by-side screenshots of a web application's registration page.

**Left Screenshot:**

- Header: "Register a new membership".
- Text: "Check email for activation link after registration".
- List: "• The Email field is required.  
• The Password field is required."
- Inputs:
  - Email input field with placeholder "Email" and a mail icon.
  - Password input field with placeholder "Password" and a lock icon.
- Buttons:
  - A blue "Register" button.
  - A blue "I already have a membership" link.

**Right Screenshot:**

- Header: "Register a new membership".
- Text: "Check email for activation link after registration".
- List: "• This email is already taken. If it is yours, please go and activate your account from there!"
- Inputs:
  - Email input field containing "svalqmmmp34@mail.bg" with a mail icon.
  - Password input field with placeholder "Password" and a lock icon.
- Buttons:
  - A blue "Register" button.
  - A blue "I already have a membership" link.

Фиг. 31. Валидация при регистриране на потребител

The figure consists of two side-by-side screenshots of a web application's login page.

**Left Screenshot:**

- Header: "Sign in to start your session".
- List: "• The Email field is required.  
• The Password field is required."
- Inputs:
  - Email input field with placeholder "Email" and a mail icon.
  - Password input field with placeholder "Password" and a lock icon.
- Checkboxes:
  - A checkbox labeled "Remember me?".
- Buttons:
  - A blue "Sign In" button.
  - A blue "Register a new membership" link.

**Right Screenshot:**

- Header: "Sign in to start your session".
- List: "• This email is not confirmed. If it is yours, please go and activate your account from there!"
- Inputs:
  - Email input field containing "svalqmmmp34@mail.bg" with a mail icon.
  - Password input field with placeholder "Password" and a lock icon.
- Checkboxes:
  - A checkbox labeled "Remember me?".
- Buttons:
  - A blue "Sign In" button.
  - A blue "Register a new membership" link.

Фиг. 32. Валидация при вход на потребител

## Магазин за сензори

При натискане на бутона „Sensor Store“ от навигационния таб потребителя ще бъде пренасочен към страницата на магазина за сензори. Там ще бъдат видими всички сензори като могат да бъдат филтрирани според типа на сензора. При натискане на „+“ изображението на всеки сензор потребителя ще може да „закупи“ този сензор и да го ползва като личен.

Choose Sensor Measure Type:

-- All --

Temperature  
Measure Unit: °C

Description

Temperature Sensor 1

5 MIN RANGE    5 sec POLLING    20 MAX RANGE

Electric Power Consumption  
Measure Unit: W

Description

Temperature Sensor 7

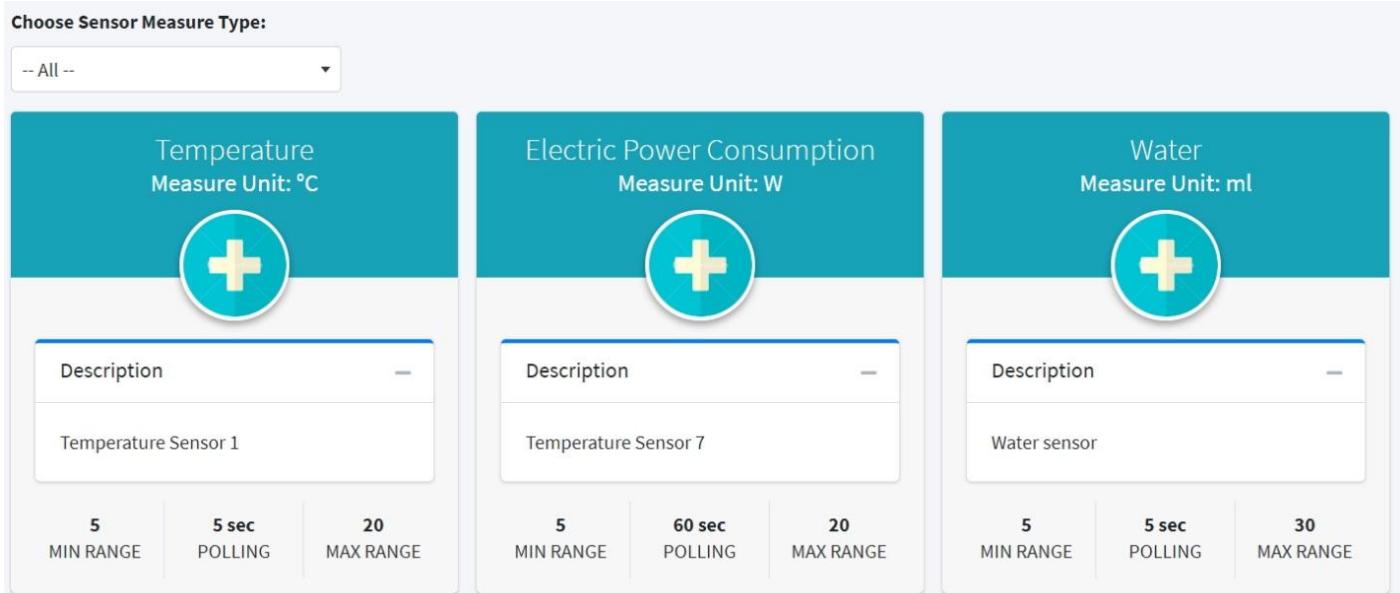
5 MIN RANGE    60 sec POLLING    20 MAX RANGE

Water  
Measure Unit: ml

Description

Water sensor

5 MIN RANGE    5 sec POLLING    30 MAX RANGE



Фиг. 33. Магазин за сензори. Без приложен филтър

Choose Sensor Measure Type:

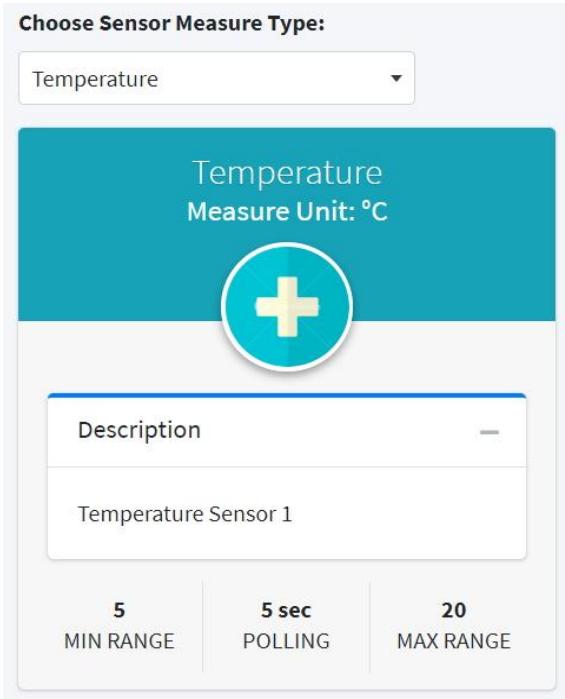
Temperature

Temperature  
Measure Unit: °C

Description

Temperature Sensor 1

5 MIN RANGE    5 sec POLLING    20 MAX RANGE



⚠ Oops! Sensor store is empty.

No sensors found in the store!

Consider adding some sensors from the mobile application first.

Фиг. 34. Съобщение при празен магазин за сензори

Фиг. 35. Магазин за сензори. С приложен филтър

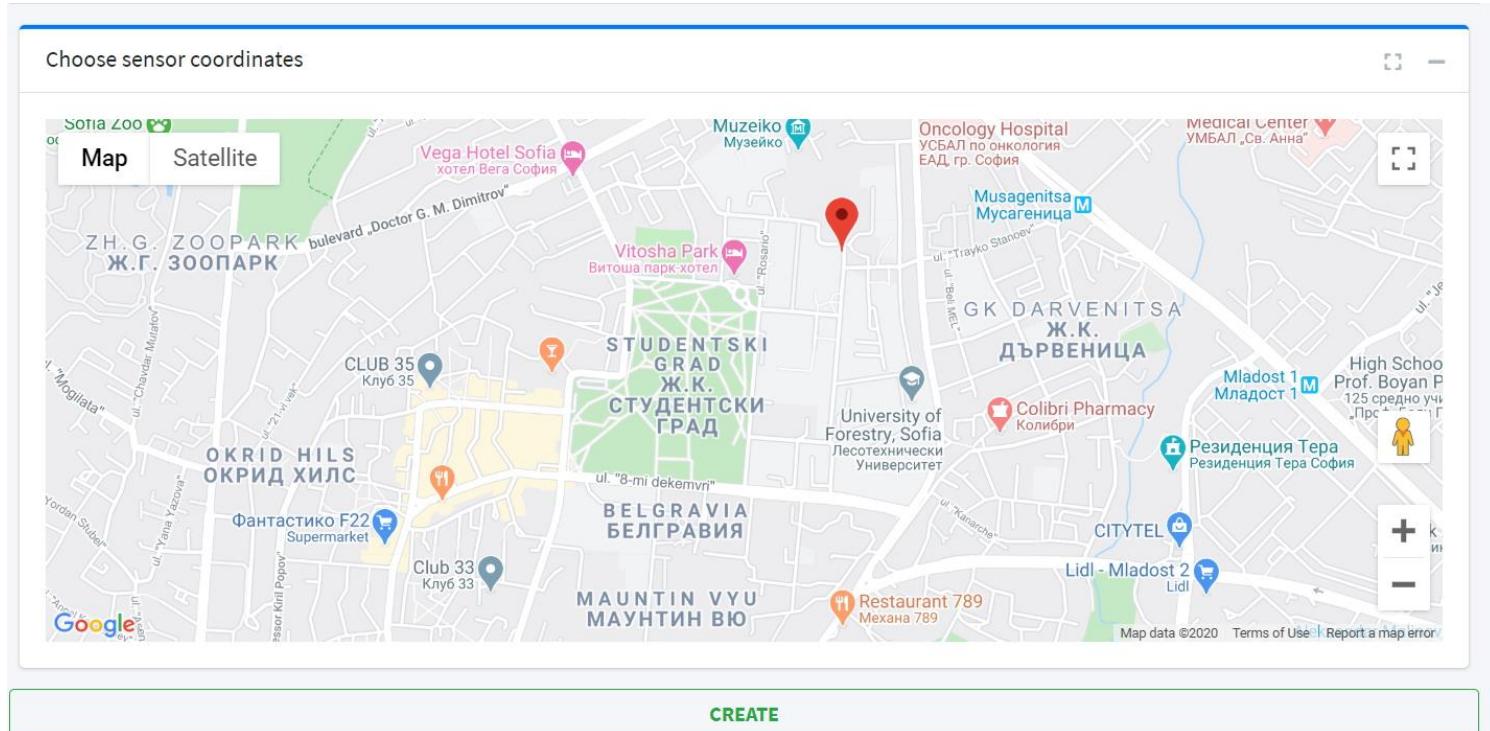
## Добавяне и редактиране на потребителски сензор. Валидация

### Добавяне/редактиране на потребителски сензор

При натискане на бутона за добавяне/редактиране на потребителски сензор потребителят бива пренасочен към страница, където трябва да попълни данни за сензора като: Име, описание, публичен ли е или не, включване на аларма и избиране на граници, при които алармата да бъде задействана и координати на сензора.

Sensor Name	Sensor Measure Type - Measure Unit	Longitude	Latitude	Public	Alarm
My Sensor	Temperature - °C	23.35656781669°	42.65720310301°	ON	ON
Sensor Description	Sensor Polling Interval	Sensor Min Range	Sensor Max Range		
My Sensor Description	5 sec 109 sec	300 sec 5 °C 8.5 °C	20 °C 5 °C 14 °C 20 °C		
Choose sensor coordinates					
<button>CREATE</button>					

Фиг. 36. Екран при създаване/редактиране на потребителски сензор



Фиг. 37. Избиране на координати от Google карта при създаване/редактиране на сензор

## Валидация

Полета като име, описание и координати на сензора са задължителни. При включена аларма минималната граница не може да бъде по-голяма от максималната и обратно. При грешка бутона “Create – “Update” става изключен и валидационна грешка бива показвана на екрана.

The screenshot shows a user interface for creating a new sensor. At the top, there is a red error box containing the message "Error!" and a list of validation errors:

- The Name field is required.
- Latitude is required!
- Longitude is required!
- The Description field is required.

The main form fields include:

- Sensor Name:** A text input field with the placeholder "Enter name...".
- Sensor Measure Type - Measure Unit:** A dropdown menu set to "Temperature - °C".
- Longitude:** A coordinate input field with a location pin icon.
- Latitude:** A coordinate input field with a location pin icon.
- Public:** A green "ON" switch.
- Alarm:** A green "ON" switch.
- Sensor Description:** A text input field with the placeholder "Enter description...".
- Sensor Polling Interval:** A dropdown menu set to "5 sec".
- Sensor Min Range:** A slider set to 12.5 °C. Below it is a red validation message: "Min Range should be less than 9".
- Sensor Max Range:** A slider set to 9 °C. Below it is a red validation message: "Max Range should be more than 12.5".
- Coordinates:** A section labeled "Choose sensor coordinates" with a map icon and a plus sign.
- CREATE:** A green button at the bottom right.

Фиг. 38. Валидация при създаване/редактиране на потребителски сензор

## Преглед на потребителските сензори

При натискане на навигационни бутона „My Sensors“ потребителя бива пренасочен към страница показваща всички негови сензори. Страницата използва pagination и се показват по 3 сензора на страница. Има опции като редактиране и изтриване на сензор.

Налични са и следните опции за филтриране и търсене:

- Филтриране по тип на сензора
- Филтриране по статус на поверителност – публичен или личен
- Филтриране по татус на аларма – включена или изключена
- Търсене по име

Допълнителни опции:

- Преглед на стойността на сензора в реално време
- Редактиране на сензор
- Изтриване на сензор

**Filter by Sensor Measure Type:** -- All --

**Filter by Privacy status:** -- All --

**Filter by Alarm status:** -- All --

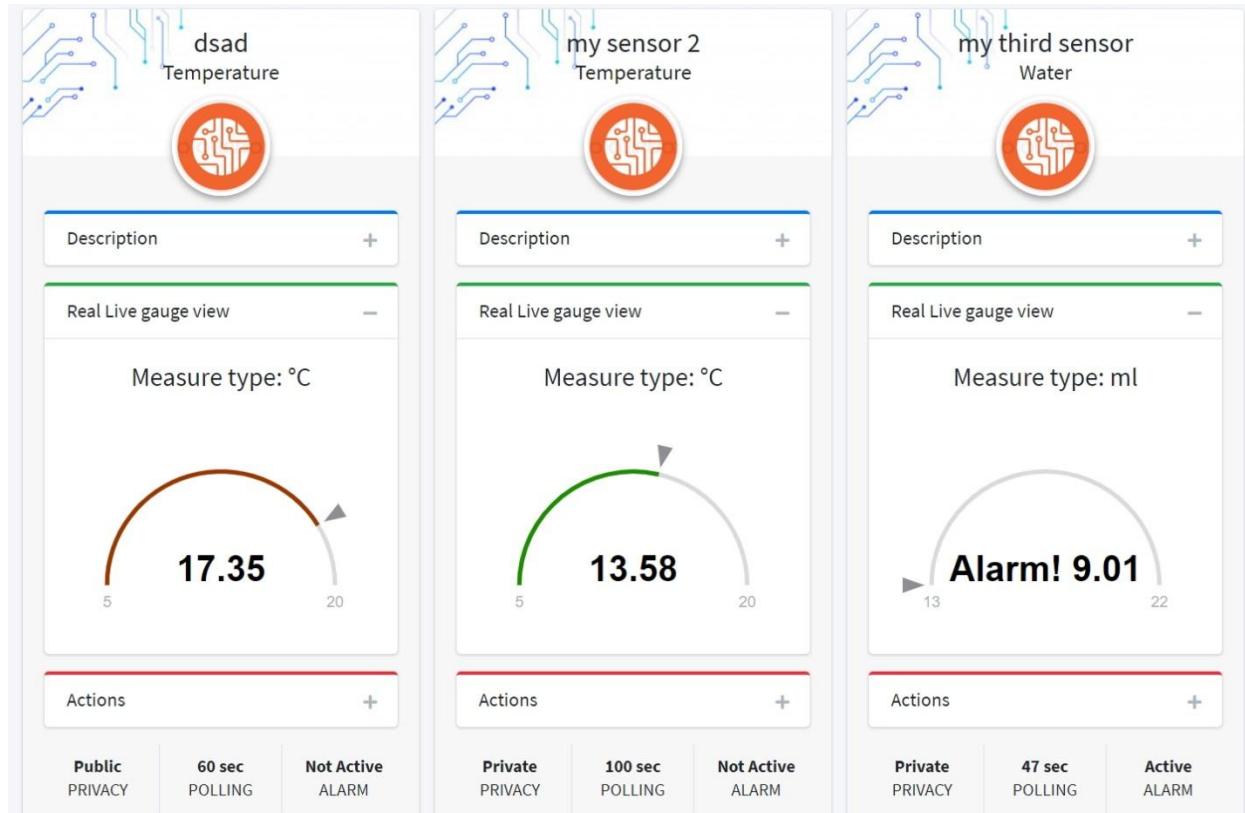
**Search by Sensor Name:**  Go!

Sensor Name	Description	Measure Type	Privacy	Polling	Alarm Status
dsad Temperature	Real Live gauge view	Temperature	Public PRIVACY	60 sec POLLING	Not Active ALARM
my sensor 2 Temperature	Real Live gauge view	Temperature	Private PRIVACY	100 sec POLLING	Not Active ALARM
my third sensor Water	Real Live gauge view	Water	Private PRIVACY	47 sec POLLING	Active ALARM

[View all sensors on map](#)

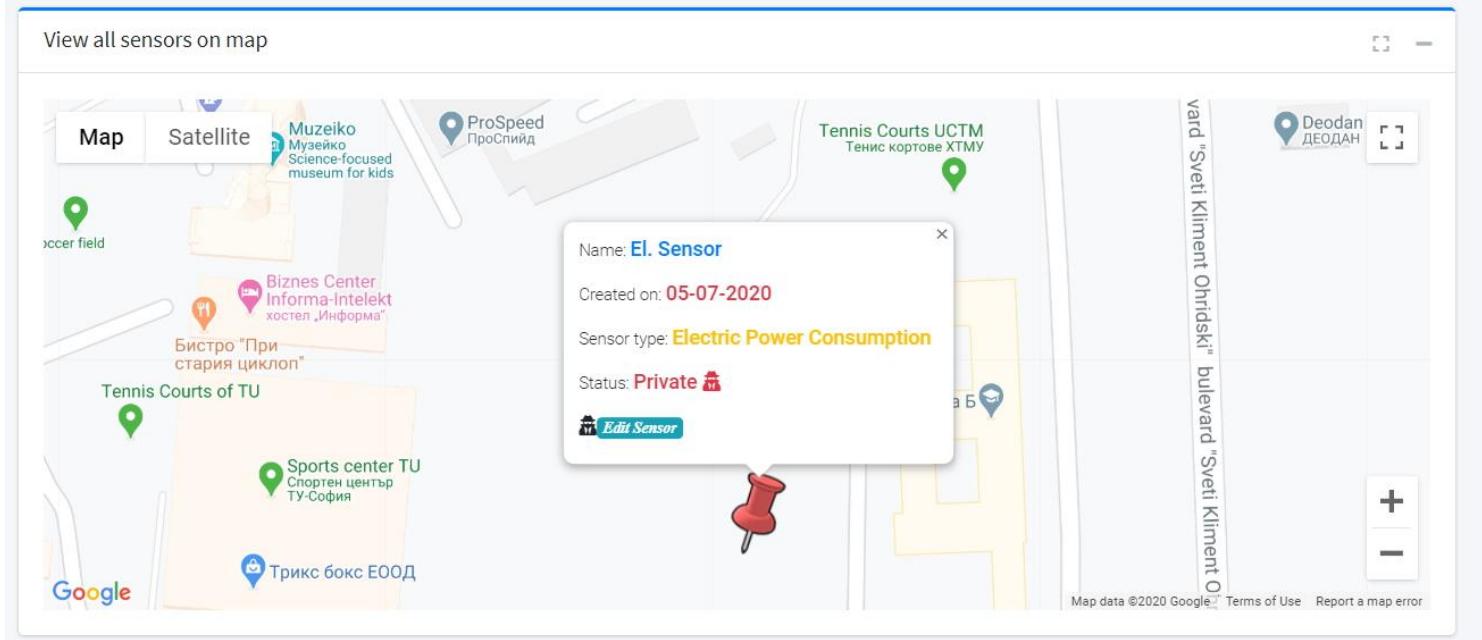
[Previous](#) [Next](#)

Фиг. 39. Преглед на потребителските сензори



Фиг. 40. Преглед на потребителски сензори и стойностите им в реално време

Actions			Actions			Actions		
<a href="#">Edit</a> <a href="#">Delete</a>			<a href="#">Edit</a> <a href="#">Delete</a>			<a href="#">Edit</a> <a href="#">Delete</a>		
Public PRIVACY	60 sec POLLING	Not Active ALARM	Private PRIVACY	100 sec POLLING	Not Active ALARM	Private PRIVACY	47 sec POLLING	Active ALARM

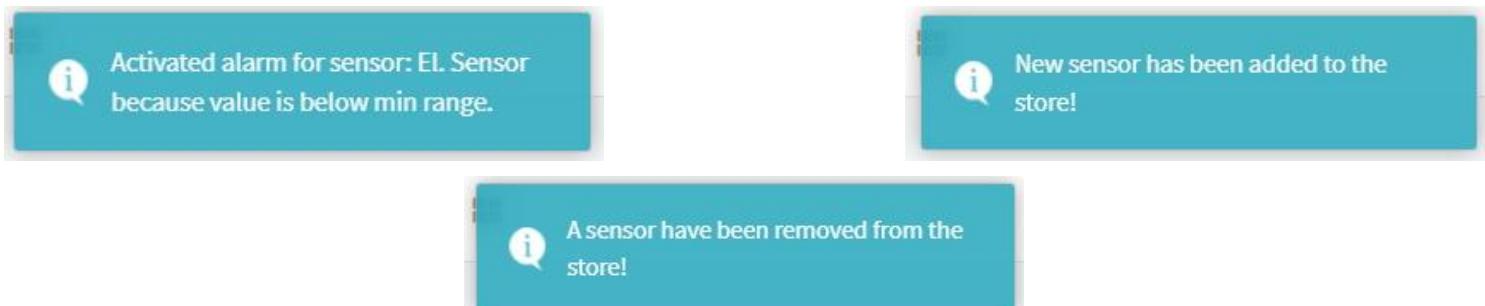


Фиг. 41. Преглед на допълнителните опции и показване на сензорите на Google карта

## Известия

Налични са 3 вида известия в системата. Показват се за определен брой секунди и след това изчезват.

1. Известие при добавяне на нов сензор в магазина от мобилното приложение
2. Известие при премахване на сензор в магазина от мобилното приложение
3. Известие при активирана аларма на потребителски сензор.



Фиг. 42. Видове потребителски известия

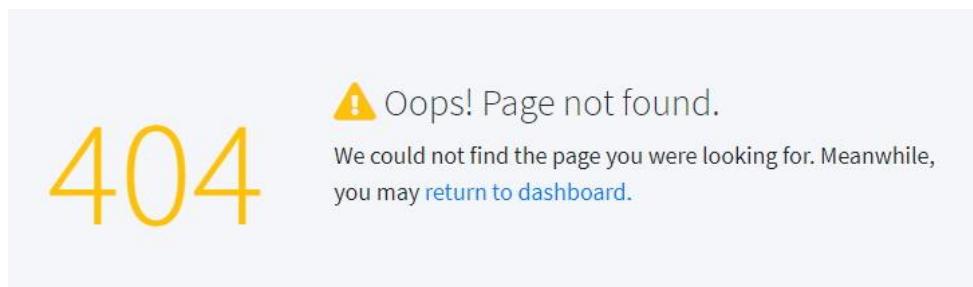
## VII. Експериментални резултати

В тази глава ще бъдат тествани някои често срещани уязвимости при онлайн системите.

### Достъп до чужди записи

Първият тест, който ще бъде разгледан е неправомерен достъп до информация за сензор чрез попълване на произволно ID в URL-а на страницата:

- Когато потребител, който въдете URL в браузъра като : .../UserSensor/5, но сензор с Id 5 не принадлежи на този потребител, то ще бъде показана страница “Page not found”.
- Когато потребителят въведе несъществуващ адрес в браузъра то ще бъде показана страница “Page not found”



Фиг. 43. Показване на страница “Page not found”

### Прихващане на изключения

Още един важен компонент от защитата на системата е да не разкриваме информация за кода (stack trace), когато възникне непредвидена грешка (exception). Затова след всяка една заявка ErrorHandler middleware се грижи при грешка потребителя да бъде пренасочен към страница показваща, че е възникнала грешка. В същото време ErrorLogging middleware записва логва всяка една грешка във файл.

```
0 references | Krasimir Etov, 9 days ago | 1 author, 1 change
public async Task Invoke(HttpContext context)
{
    try
    {
        await this._next(context);

        if (context.Response.StatusCode == 404)
        {
            context.Response.Redirect("/pagenotfound");
        }

        if (context.Response.StatusCode == 500)
        {
            context.Response.Redirect("/error");
        }
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Exception was thrown.");
        throw;
    }
}
```

```
0 references | Krasimir Etov, 9 days ago | 1 author, 2 changes
public async Task Invoke(HttpContext context)
{
    try
    {
        await this._next(context);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Exception was thrown.");
        throw;
    }
}
```

Фиг. 44. ErrorHandler и ErrorLogging middlewares

## **VIII. Използвана литература**

- [1] Datum Platform Interactive - <https://datum-group.com/products/platform-interactive/>
- [2] Sunbird Energy - <https://www.sunbirddcim.com/product/data-center-energy-management>
- [3] ASP.NET Core - <https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-3.1>
- [4] MVC Architecture - [https://bg.wikipedia.org/wiki/ASP.NET\\_MVC](https://bg.wikipedia.org/wiki/ASP.NET_MVC)
- [5] Entity Framework Core - <https://docs.microsoft.com/en-us/ef/>
- [6] Microsoft SQL Server - [https://en.wikipedia.org/wiki/Microsoft\\_SQL\\_Server](https://en.wikipedia.org/wiki/Microsoft_SQL_Server)
- [7] Flutter - [https://en.wikipedia.org/wiki/Flutter\\_\(software\)](https://en.wikipedia.org/wiki/Flutter_(software))
- [8] JavaScript - <https://bg.wikipedia.org/wiki/JavaScript>
- [9] jQuery - <https://bg.wikipedia.org/wiki/JQuery>
- [10] HTML 5 - <https://bg.wikipedia.org/wiki/HTML5>
- [11] CSS 3 - <https://bg.wikipedia.org/wiki/CSS>

## **IX. Приложение**

Source кодът на приложението може да бъде намерен на:

- Следният GitHub адрес - <https://github.com/KrasimirEtov/SensorManagementSystem>
- Приложения диск към документацията.