Технически университет - София Факултет Компютърни Системи и Технологии

**Дипломна Работа**

На тема:

**„Приложение за следене и управление на температура и влажност“**

Изготвил: Иван Бориславов Димитров

Специалност: Компютърно и софтуерно инженерство

Ф.№: 121320076

**Съдържание**

[**Увод** 4](#_Toc102930195)

[**1. Описание на проблемната област** 5](#_Toc102930196)

[**1.1. Най-известните уреди за измерване и промяна на температура и влажност** 5](#_Toc102930197)

[**А) Умен термостат** 5](#_Toc102930198)

[**Б) Овлажнител на въздух** 8](#_Toc102930199)

[**В) Климатик** 11](#_Toc102930200)

[**Г) Контролер на температура и влажност** 14](#_Toc102930201)

[**1.2. Каква е целта и мотивацията за създаването на нов продукт, който ще служи за регулиране на температура и влажност?** 15](#_Toc102930202)

[**1.3. Workflow диаграма на основното използване на продукта** 16](#_Toc102930203)

[**2. Проектиране на архитектура на системата** 19](#_Toc102930204)

[**2.1. Основните модели в системата (domain model)** 20](#_Toc102930205)

[**2.2. ER (entity relationship) диаграма** 24](#_Toc102930206)

[**2.3. Цялостна архитектура на приложението** 28](#_Toc102930207)

[**2.4. Анализ на входни данни на потребител, модератор и администратор.** 34](#_Toc102930208)

[**2.5. Използвани технологии** 39](#_Toc102930209)

[**3. Особености на програмната реализация** 47](#_Toc102930210)

[**3.1. Какво представлява клиентът?** 47](#_Toc102930211)

[**3.2. Какво представлява сървърът?** 50](#_Toc102930212)

[**3.3. Видове използвани релации използвани в главния сървър** 62](#_Toc102930213)

[**3.4. Какво представлява измервателната станция** 64](#_Toc102930214)

[**3.5. Какво представлява умният контакт** 64](#_Toc102930215)

[**3.6. Блокови схеми на основните операции** 64](#_Toc102930216)

[**А) Блокова схема на регистрация на потребител** 64](#_Toc102930217)

[**Б) Блокова схема на добавяне на курс** 65](#_Toc102930218)

[**В) Блокова схема на свързване на акаунт с GitHub** 65](#_Toc102930219)

[**Д) Блокова схема на регистрация на потребител** 65](#_Toc102930220)

[**4. Валидация на системата** 66](#_Toc102930221)

[**4.1. Unit тестове** 66](#_Toc102930222)

[**4.2. Ръчни тестове** 70](#_Toc102930223)

[**Заключение** 75](#_Toc102930224)

[**Код** 77](#_Toc102930225)

[**Източници** 91](#_Toc102930226)

# **Увод**

Интернет и технологиите са навлезли изключително много в днешно време и се използват вече почти навсякъде. Почти всеки дом има интернет и поне един компютър. Почти всеки човек притежава мобилно устройство, което също по някакъв начин използва интернет, за да приема и предава данни и комуникира с други свързани устройства. По този начин всички тези устройства предоставят на своите потребители да разбират за най-актуалната информация мигновено, както и да научават нови неща само с няколко кликвания с мишката. Но само информацията не винаги е достатъчна, за да получи човек всичко от, което има нужда. Доста често се налага софтуер да управлява хардуер или машина директно, които досега са били управлявани единствено от човек. Това води до постоянната автоматизация на все повече машини и техники.

С подобрението на технологиите се подобрява и човешкият живот. Все повече неща се автоматизират и улесняват, но с тях нараства и нуждата повече хора да се занимават и разбират и подобряват тази автоматизацията, както и да автоматизират все по нови ниши.

В днешно време развитието на технологиите става с изключително висока скорост и ежедневно нови и все по развити технологии излизат на пазара. Това развитие е напълно очаквано, защото компютрите и технологиите стават все по масово използвани от все повече хора и във все повече сфери и адаптацията на софтуера и хардуера трябва да бъде направена, за да се дигитализира, автоматизира и модернизира животът. Компютрите се използват във всички индустрии, от образование до производство на коли и машини, до все повече IoT инструменти, до пресмятане на сложни математически задачи. Популярността и необходимостта им нараства ежедневно и употребата им става все по-голяма като това се наблюдава вече повече от 30 години и няма никаква вероятност да се наблюдава обратният ефект. Поради тези причини се изисква обучението на повече хора в тази сфера, да се продължава с опитите все по малко сфери и дейности да изискват човешка намеса и да не спират да се развиват хората в сферата на компютърните и софтуерните технологии.

# **1. Описание на проблемната област**

Околната температура и влажност са от изключително значение както за хората, така и за животните, растенията дори и за изработването на продукти. Всички ние сме зависими на тях, но успяваме да ги контролираме на затворени места. Почти във всеки дом вече има уреди, които автоматично управляват температурата и нивото на влажността във въздуха, като ги държат в най-оптимални условия в зависимост от даденото място. Тези условия са нужни да се управляват както вкъщи, така и при отглеждане на животни, при отглеждане на растения, при създаване на определение продукти, химикали и т.н., почти навсякъде е от изключителна важност. Без да се поддържат тези определени граници на измерванията, би следвало създаването на лош продукт, лоша реколта и като цяло е ненадеждно. Затова обществото отдавна се опитва да ги контролира. Допреди век, повечето от измервания е трябвало да се следят ръчно от човек, който би следвало да направи нещо, за да може да не стане някакъв инцидент. През последните години и тази сфера се автоматизира, като има създадени най-различни уреди и IoT устройства, които помагат автоматичното регулиране на условията. Най-често срещаното и масово такова устройство, което всички знаем е климатикът. Той отговаря за поддържането на температурата и се включва и изключва, когато излезе от зададените граници. Друг такъв пример е овлажнителят на въздух, той също измерва количеството на водни изпарения във въздуха и ги контролира. Умният термостат, който е част от smart home технологиите е друга интересна версия на този тип устройства. Всички тези устройства четат температура, влажност и налягане чрез най-различни сензори обработват данните и управляват устройството. Но не винаги това е напълно достатъчно, тъй като устройствата управляват само себе, тоест не управляват други устройства, които също биха също променяли тези показатели.

## **1.1. Най-известните уреди за измерване и промяна на температура и влажност**

### **А) Умен термостат**

Интелигентните термостати са Wi-Fi термостати, които могат да се използват с цел домашна автоматизация и са отговорни за управлението на отоплението, вентилацията и климатизацията на дома. Те изпълняват подобни функции като програмируемемите термостати, тъй като позволяват на потребителя да контролира температурата в дома си през целия ден с помощта на график, но също така съдържат допълнителни функции, като сензори и Wi-Fi свързаност,[1] които подобряват разнородни проблеми с програмируемите термостати.

Подобно на други Wi-Fi термостати, те са свързани към интернет чрез Wi-Fi мрежа. Те позволяват на потребителите да регулират настройките за отопление от други устройства, свързани с интернет, като лаптоп или смартфони. Това позволява на потребителите да управляват термостата дистанционно. Тази лекота на използване е от съществено значение за осигуряване на икономии на енергия. Проучвания показват, че домакинствата с програмируеми термостати всъщност имат по-висока консумация на енергия от тези с обикновени термостати, тъй като жителите ги програмират неправилно или ги деактивират напълно.[2][3]

Програмируемите термостати, въведени за първи път преди повече от 100 години, са вид термостат, който позволява на потребителя да задава график за различни температури в различно време. Повечето програмируеми термостати също имат функция за задържане, която ефективно превръща термостата в ръчен термостат. [4][5]

Идеята на функцията за планиране е, че потребителите ще задават по-висока или по-ниска температура, когато домът е празен, за да спестят енергия и пари. Поради тази предполагаема икономия на енергия, някои строителни норми и правителствени програми са започнали да изискват използването на програмируеми термостати.[6] За съжаление, поради човешка грешка при използването на тези устройства, много програмируеми термостати водят до по-голяма консумация на енергия от основния ръчен термостат.[7]

Интелигентните термостати са подобни на програмируемите термостати в смисъл, че имат функция за планиране, която позволява на потребителите да задават различни температури за различни часове от деня. В допълнение към тази функция, интелигентните термостати прилагат други технологии, за да намалят количеството човешки грешки, свързани с използването на програмируеми термостати. Интелигентните термостати включват използването на сензори, които определят дали жилището е заето или не и могат да преустановят отоплението или охлаждането, докато обитателят се върне. Освен това интелигентните термостати използват Wi-Fi свързаност, за да предоставят на потребителя достъп до термостата по всяко време. Тези допълнителни технологии са доказали, че правят интелигентните термостати успешни в спестяването на енергия и пари на потребителите.[4]

Функцията за програмируем график на интелигентния термостат е подобна на тази на стандартните програмируеми термостати. На потребителите се дава възможност да програмират персонализиран график за намаляване на потреблението на енергия, когато са далеч от дома. Проучванията показват обаче, че ръчното създаване на график може да доведе до повече потребление на енергия, отколкото просто поддържане на термостата при зададена температура.[8] За да се избегне този проблем, интелигентните термостати също така предоставят функция за автоматичен график. Тази функция изисква използването на алгоритми и разпознаване на модели за създаване на график, който води до комфорт на обитателите и спестяване на енергия. След създаване на график, термостатът ще продължи да следи поведението на обитателите, за да направи промени в автоматичния график. Изваждайки човешката грешка от планирането, интелигентните термостати могат да създават интелигентни графици, които всъщност спестяват енергия.[8]

В опит да смекчи проблемите с човешка грешка, свързани с програмируемите термостати, интелигентният термостат използва сензор, който може да определи моделите на заетост, за да промени автоматично температурата въз основа на моделите и поведението на обитателите. По-специално термостатът „Nest Learning“ използва пасивни инфрачервени (PIR) сензори за движение вътре в модула, за да усети заетостта в близост до термостата. Този сензор информира термостата дали жилището е заето или не. В случай, че жилището не е заето, термостатът може да преустанови отоплението/охлаждането, докато сензорът не бъде активиран отново от обитател. Този сензор се използва и за определяне на моделите на заетост за създаване на автоматичен график. Пред сензора е поставен елемент на решетката, за да прикрие и защити PIR сензора за движение вътре в термостата. Решетката също помага визуално по-приятен и да се вписва в дома.[2] Въпреки че тази сензорна технология е важна за пестенето на енергия, тя не е лишена от недостатъци. Един от основните проблеми е, че сензорът трябва да се активира от някой, който върви пред или близо до термостата. Възможно е обитател да е вкъщи и да не минава пред сензора. В този случай термостатът ще изключи отоплението/охлаждането и ще намали комфорта на човека.[9]

Основна характеристика на Wi-Fi термостатите (като интелигентните термостати) е способността им да се свързват с интернет. Тези термостати са проектирани с Wi-Fi модул, който позволява на термостата да се свърже с домашната или офис мрежа на потребителя и да се свързва с уеб портал или приложение за смартфон, което позволява на потребителите да управляват термостата дистанционно.[10] Wi-Fi функцията също така има възможността да изпраща отчети за потреблението на енергия и ефективността на HVAC системата чрез уеб портала, като информира потребителя за тяхната енергийна ефективност и как тя се сравнява с други потребители на интелигентни термостати. Той също така може да предупреждава потребителите, когато възникне проблем с тяхната HVAC система или когато е време за поддръжка на оборудването. Термостатът може също да използва Wi-Fi връзката за показване на текущите метеорологични условия и прогнозата за времето.[1]

Друга функция, предлагана от някои интелигентни термостати чрез интернет връзката, е “geofencing”. Геооградата е периметърна граница, създадена около местоположението на смартфон или друго устройство, въз основа на GPS сигнали. Предимството от наличието на интелигентен термостат с възможности за геозониране е, че той използва местоположението на смартфона на потребителя, за да определи дали домът е зает. Вместо да използва график или сензор за определяне на заетостта, интелигентният термостат може да разчита на местоположението на геооградата, за да каже на HVAC системата дали трябва да бъде включена или изключена.[11] Тъй като повечето хора носят телефоните си със себе си, геозонирането може да бъде точен начин за определяне на моделите на заетост.[8]

За да покажат, че термостатите спестяват енергия и пари, много производители на интелигентни термостати са провели модели и проучвания, за да потвърдят своите твърдения за спестявания. Един популярен начин, по който производителите на интелигентни термостати изчисляват потреблението на енергия, е чрез енергийно моделиране. При тези модели интелигентният термостат се сравнява с термостат, настроен на постоянна температура, и се изчисляват спестяванията. Използвайки този метод, „Ecobee” е изчислила спестяванията на енергия, като е съпоставила колко дълго оборудването за отопление и охлаждане работи с местните метеорологични условия. Спестяванията на енергия са били изчислени спрямо постоянна температура от 22 °C (72 °F). При провеждането на този модел, „ Ecobee” е определила 23% спестяване на разходи за отопление и охлаждане за тези, които преминат към интелигентен термостат.[12] Използвайки подобен метод за моделиране, “Nest” е обявила 20% икономия на енергия за собствениците на жилища, които инсталират Nest Learning Thermostat.[13]

Тези проучвания доказват, че интелигентните и Wi-Fi термостати водят до спестяване на енергия за отопление и охлаждане, което ги прави изключително полезни при проектирането и създаването на отоплителна или охладителна система.

### **Б) Овлажнител на въздух**

Овлажнителят е устройство, предимно електрически уред, което повишава влажността (влагата) в една стая или в цяла сграда. В дома овлажнителите за обикновено се използват за овлажняване на една стая, докато овлажнителите за цялата къща, които се свързват към домашната ОВК система, осигуряват влажност на цялата къща. Медицинските вентилатори често включват овлажнители за повишен комфорт на пациента. Големите овлажнители се използват в търговски, институционални или промишлени контексти, често като част от по-голяма HVAC система.

Ниска влажност може да се появи в горещ, сух пустинен климат или на закрито в изкуствено отопляеми помещения. През зимата, особено когато студен външен въздух се нагрява на закрито, влажността може да падне до 10-20%. За повечето домове се препоръчва относителна влажност от 30% до 50%.[1]

Относителна влажност на закрито под 51% доведе до значително намаляване на нивата на акари и алергени.[9] Прекомерната употреба на овлажнител може да повиши относителната влажност до прекомерни нива, насърчавайки растежа на прахови акари и мухъл, а също така може да причини свръхчувствителен пневмонит (възпаление на тъканите на белия дроб).[10] Трябва да се използва правилно инсталиран и разположен хигростат за автоматично следене и контролиране на нивата на влажност или добре информиран и съвестен човешки оператор трябва постоянно да проверява за правилните нива на влажност.

Влажност под 50% може да предотврати кондензацията на вода върху строителните материали.[1]

Може да се използва изсушител за балансиране на влажността.

Промишлените овлажнители се използват, когато трябва да се поддържа определено ниво на влажност, за да се предотврати натрупването на статично електричество, да се запазят свойствата на материала и да се осигури комфортна и здравословна среда за работниците или жителите.

Статичните проблеми са преобладаващи в индустрии като опаковане, печат, хартия, пластмаса, текстил, електроника, автомобилно производство и фармацевтични продукти. Триенето може да доведе до статично натрупване и искри, когато влажността е под 45% относителна влажност (RH). Между 45% и 55% RH, статиката се натрупва при намалени нива, докато влажността над 55% RH гарантира, че статиката никога няма да се натрупа.[11] Американското дружество на инженерите по отопление, охлаждане и климатизация (ASHRAE) традиционно препоръчва диапазон от 45–55% относителна влажност в центровете за данни, за да се предотвратят искри, които могат да повредят ИТ оборудване.[12] Овлажнителите се използват и от производители на полупроводници и в болнични операционни зали.

Принтерите и производителите на хартия използват овлажнители за предотвратяване на свиване и извиване на хартията. Овлажнители са необходими в хладилните складови помещения, за да се запази свежестта на храната срещу сухота, причинена от високи температури. Музеите на изкуството използват овлажнители за защита на чувствителни произведения на изкуството, особено в изложбените галерии, където се борят със сухотата, причинена от отоплението, за комфорт на посетителите през зимата.[14]

Един вид изпарителен овлажнител използва само резервоар и фитил. Понякога наричани "естествен овлажнител", това обикновено са нетърговски устройства, които могат да бъдат сглобени на ниска цена или без разходи. Една версия на естествен овлажнител използва купа от неръждаема стомана, частично пълна с вода, покрита с кърпа. За потапяне на кърпата в центъра на купата се използва водоустойчива тежест. Няма нужда от вентилатор, тъй като водата се разпространява през кърпата чрез капилярно действие и повърхността на кърпата е достатъчно голяма, за да осигури бързо изпаряване. Купата от неръждаема стомана е много по-лесна за почистване от типичните резервоари за вода за овлажнители. Това, в комбинация с ежедневна или през ден подмяна на кърпата и периодично пране, може да овладее проблема с мухъл и бактерии.

Стайните растения могат да се използват и като естествени овлажнители на въздуха, особено ако са поставени в платнени саксии, тъй като те изпаряват водата във въздуха чрез транспирация. Все пак трябва да се внимава, за да се предотврати разрастването на бактерии или мухъл в почвата до прекомерни нива или да се разпръснат във въздуха.

Най-разпространеният преносим овлажнител, "изпарителен", "охладител на влага" или "фитил овлажнител", се състои само от няколко основни части: резервоар, фитил и вентилатор.

Фитилът е изработен от порест материал, който абсорбира вода от резервоара и осигурява по-голяма повърхност, от която да се изпарява. Вентилаторът е в непосредствена близост до фитила и издухва въздух върху фитила, за да подпомогне изпаряването на водата. Изпарението от фитила зависи от относителната влажност. Стая с ниска влажност ще има по-висока скорост на изпаряване в сравнение със стая с висока влажност. Следователно този тип овлажнители са частично саморегулиращи се; с увеличаване на влажността на помещението отделянето на водна пара естествено намалява.

Тези фитили могат да мухлясат, ако не се изсушат напълно между пълнежите и се насищат с минерални отлагания с течение на времето. Те се нуждаят редовно от изплакване или подмяна; ако това не се случи, въздухът не може да премине през тях и овлажнителят спира да овлажнява зоната, в която се намира и водата в резервоара остава на същото ниво.

Роторните овлажнители (охладител на мъгла) обикновено са по-шумни от другите поради въртящ се диск, който хвърля вода към дифузьор, който разбива водата на фини капчици, които плуват във въздуха. Водоснабдяването трябва да се поддържа стриктно чисто, в противен случай съществува риск от разпространение на бактерии или мухъл във въздуха.

Ултразвуковият овлажнител използва керамична диафрагма, вибрираща с ултразвукова честота, за да създаде водни капчици, които безшумно излизат от овлажнителя под формата на хладна мъгла. Обикновено мъглата се изтласква от малък вентилатор, докато някои ултра мини модели нямат вентилатори. Моделите без вентилатори са предназначени основно за лична употреба. Ултразвуковите овлажнители използват пиезоелектричен преобразувател за създаване на високочестотни (1-2 MHz[14]) механични трептения във филм от вода. Това образува изключително фина мъгла от капчици с диаметър около един микрон, която бързо се изпарява във въздушния поток.[15]

За разлика от овлажнителите, които кипят вода, тези водни капчици ще съдържат всякакви примеси, които са в резервоара, включително минерали от твърда вода (която след това образува труден за отстраняване лепкав бял прах върху близки предмети и мебели). Всички патогени, растящи в застоялия резервоар, също ще бъдат разпръснати във въздуха. Ултразвуковите овлажнители трябва да се почистват редовно, за да се предотврати разпространението на бактериално замърсяване във въздуха.

Количеството на минерали и други материали може да бъде значително намалено с помощта на дестилирана вода. Специалните патрони за деминерализация за еднократна употреба също могат да намалят количеството на материалите във въздуха, но EPA предупреждава, „способността на тези устройства да премахват минерали може да варира значително.“[16] Минералният прах може да има отрицателни последици за здравето. Овлажнителите на фитил улавят минералните отлагания във фитила. Типовете изпарители са склонни да събират минерали върху или около нагревателния елемент и изискват редовно почистване с оцет или лимонена киселина, за да контролират натрупването.

Както става ясно, овлажнителите на въздуха благоприятстват здравето на човека и са полезни, когато влагата се държи в определени норми, които ако бъдат преминати могат да доведат до нежелание последици. Това води и до нуждата от устройства за автоматизирано управление на влажността.

### **В) Климатик**

Климатизацията, често съкратено като A/C или AC, е процесът на отстраняване на топлината и контролиране на влажността на въздуха в затворено пространство за постигане на по-удобна вътрешна среда чрез използване на захранвани „климатици“ или различни други методи, включително пасивно охлаждане и вентилативно охлаждане. Климатизацията е член на семейство системи и техники, които осигуряват отопление, вентилация и климатизация (HVAC).

Климатиците, които обикновено използват парно компресионно охлаждане, варират по размер от малки единици, използвани в превозни средства или единични стаи, до масивни единици, които могат да охлаждат големи сгради. Термопомпите с въздушен източник, които могат да се използват както за отопление, така и за охлаждане, стават все по-разпространени в по-хладен климат. [17]

Според Международната агенция по енергетика (IEA) към 2018 г. са инсталирани 1,6 милиарда климатични единици, което представлява приблизително 20% от потреблението на енергия в сградите в световен мащаб, като броят им се очаква да нарасне до 5,6 милиарда до 2050 г.

Охлаждането в традиционните системи за променлив ток се постига с помощта на цикъла на компресия на пара, който използва принудителната циркулация и фазовата промяна на хладилен агент между газ и течност за пренос на топлина. Цикълът на компресия на пара може да се случи в рамките на единно или пакетирано оборудване или в охладител, който е свързан към терминално охладително оборудване (като вентилаторна конвектора във въздухообработвател) от страната на изпарителя и оборудване за отвеждане на топлината, като охладителна кула от страната на кондензатора. Термопомпата с въздушен източник споделя много компоненти с климатична система, но включва обратен клапан, който позволява на уреда да се използва както за отопление, така и за охлаждане на пространство.

Климатичното оборудване ще намали абсолютната влажност на въздуха, обработван от системата, ако повърхността на изпарителната намотка е значително по-хладна от точката на оросяване на околния въздух. Климатикът, проектиран за заето пространство, обикновено постига относителна влажност от 30% до 60% в заеманото пространство.

Повечето съвременни климатични системи разполагат с цикъл на изсушаване, по време на който компресорът работи, докато вентилаторът се забавя, за да се намали температурата на изпарителя и следователно да се кондензира повече вода. Изсушителят използва същия цикъл на охлаждане, но включва както изпарителя, така и кондензатора в един и същи въздушен път. Въздухът първо преминава през изпарителната намотка, където се охлажда[55] и изсушава, преди да премине през кондензаторната намотка, където отново се затопля, преди да бъде пуснат отново в стаята. [18]

Понякога може да се избере свободно охлаждане, когато външният въздух е по-хладен от вътрешния и следователно не е необходимо да се използва компресор, което води до висока ефективност на охлаждане за тези времена. Това може да се комбинира и със сезонно съхранение на топлинна енергия.

Някои климатични системи имат опцията да обърнат цикъла на охлаждане и да действат като термопомпа с въздушен източник, като по този начин произвеждат отопление вместо охлаждане във вътрешната среда. Те също така обикновено се наричат ​​"климатици с обратен цикъл". Термопомпата е значително по-енергийно ефективна от електрическото съпротивително отопление, тъй като премества енергията от въздуха и води към отопляемото пространство, както и топлината от закупената електрическа енергия. Когато термопомпата е в режим на отопление, вътрешната изпарителна намотка превключва ролите и се превръща в кондензатор, произвеждайки топлина. Външният кондензатор също превключва ролите, за да служи като изпарител и изпуска студен въздух (по-студен от външния въздух).

По-старите поколения термопомпи с въздушен източник стават по-малко ефективни при външни температури, по-ниски от 4°C или 40°F, това отчасти се дължи на образуването на лед върху топлообменната намотка на външното тяло, което блокира въздушния поток над бобината. За да компенсира това, термопомпената система трябва временно да се върне обратно в режим на обикновен климатик, за да превключи външната изпарителна намотка обратно в кондензаторна намотка, така че да може да се нагрява и размразява. Поради това някои термопомпени системи ще имат форма на отопление с електрическо съпротивление във вътрешния въздушен път, който се активира само в този режим, за да компенсира временното вътрешно охлаждане на въздуха, което иначе би било неудобно през зимата.

По-новите модели имат подобрена производителност при студено време, с ефективен капацитет на отопление до −14 °F (−26 °C). Въпреки това винаги има вероятност влагата, която се кондензира върху топлообменника на външното тяло, да замръзне, дори при модели, които имат подобрена производителност при студено време, което изисква извършване на цикъл на размразяване.

Проблемът с обледеняването става много по-сериозен при по-ниски външни температури, така че термопомпите понякога се инсталират в тандем с по-конвенционална форма на отопление, като електрически нагревател, природен газ, нафта или камина на дърва или централно отопление, който се използва вместо или в допълнение към термопомпата при по-сурови зимни температури. В този случай термопомпата се използва ефективно при по-ниски температури и системата се превключва на конвенционален източник на топлина, когато външната температура е по-ниска. [19]

Климатиците също са устройства, които могат да поддържат определена температура в затворено помещение, също като термостатите, но обикновено имат малък температурен диапазон - около 20 градуса.

### **Г) Контролер на температура и влажност**

Контролерите на температура и влажност наблюдават и поддържат правилни им нива в приложения за тестове за околната среда, зони за съхранение на храни и помещения за електронно оборудване. Те често включват функции за индикация на скорост и възможности за сумиране, регистратор на данни и запис на графики. Предлагат се и контролери за влажност, които осигуряват контрол на отоплението и охлаждането. Многофункционалните продукти могат да се използват за управление на термоелектрически, термисторни или термодвойки нагреватели или за управление на съпротивителни температурни детектори (RTD), резистивни нагреватели или нагревателни елементи. Обикновено доставчиците посочват дали контролерите за влажност са предназначени за използване с течности, твърди вещества, прах, газове, въздух или пара. Някои контролери за влажност имат форм-фактор на печатна платка (PCB). Други са предназначени за монтаж в стелаж, на DIN шина или на стена, шаси, корпус или шкаф. Самостоятелните технологични контролери са настолни или подови модули с пълен корпус или шкаф и интегриран интерфейс. [20]

Контролерите на температура влажност използват няколко различни техники за контрол. Контролът на ограниченията (контрол при изключване) установява зададени точки или граници, които, когато бъдат достигнати, изпращат сигнал за спиране или стартиране на устройство. Линейното управление съпоставя променлив входен сигнал със съответно променлив контролен сигнал. След това кондиционирането, филтрирането и усилването на сигнала се използват за получаване на правилния изходен контролен сигнал. Пропорционалното, интегрално и производно управление (PID) изисква обратна връзка от системата в реално време. Управлението с пренасочване осигурява компенсация с директно управление от референтния сигнал. Този тип техника за управление може да бъде отворен контур или да се използва във връзка с по-усъвършенствано PID управление. Размитата логика е вид контрол, при който променливите могат да имат неточни стойности (като частична истина), а не бинарен статус (напълно вярно или напълно невярно). Разширените или нелинейни контроли включват алгоритми като адаптивно усилване и невронни мрежи. [20]

Броят на входовете е общият брой сигнали, изпратени към контролера на влажността. Броят на изходите е сумата от всички изходи, използвани за контролиране, компенсиране или коригиране на процеса. Типовете входове за контролери за влажност включват постоянно напрежение (DC), токови контури, аналогови сигнали от резистори или потенциометри, честотни входове и входове за превключватели или релеи. Типовете изходи включват аналогово напрежение, токови контури, превключвателни или релейни изходи и импулси или честоти. Някои контролери на влажността могат също да изпращат входове или да получават изходи в сериен, паралелен, Ethernet или други цифрови формати, които показват променлива на процеса. Други могат да изпращат входове и да получават изходи от информация, преобразувана в индустриален протокол fieldbus, като CANbus, PROFIBUS® или SERCOS. PROFIBUS е регистрирана търговска марка на PROFIBUS International. [20]

Контролерите за влажност се различават по отношение на характеристиките на потребителския интерфейс и съответствието с нормативните изисквания. Много продукти разполагат с цифров преден панел или аналогови компоненти като копчета, превключватели и измервателни уреди. Предлагат се също компютърно програмируеми, уеб-съвместими и Ethernet или готови за мрежа контролери за влажност. По отношение на съответствието, устройствата, предназначени за продажба на европейския пазар, трябва да отговарят на изискванията на директивите на Европейския съюз (ЕС) за ограничаване на опасните вещества (RoHS) и за отпадъци от електрическо и електронно оборудване (WEEE). [20]

## **1.2. Каква е целта и мотивацията за създаването на нов продукт, който ще служи за регулиране на температура и влажност?**

Първата и основна цел на проекта е да обединява контролирането и автоматизирането на множество устройства, през една входна точка за потребителя. Всеки потребител може да добавя свои т.н. станции, които мерят температура или влажност, към тях се закачат умни контакти, които биват контролирани от станциите на база на въведени критерии от потребителите.

Всички от изброените устройства работят, посочени горе, работят сами за себе си, измерват една величина, например температура, и отговарят само за нея. Например термостатите, ще пускат или спират определено устройства в зависимост от дадена температура. Изключение тук е контролерът на температура и влажност, но неговият недостатък е, че той контролира управлението на няколко устройство, което е зададено от производителя. Той не позволява добавянето на нови устройства.

Тези устройства не предоставят на потребителя обобщен начин да бъдат контролирани заедно и да променят множество величини успоредно.

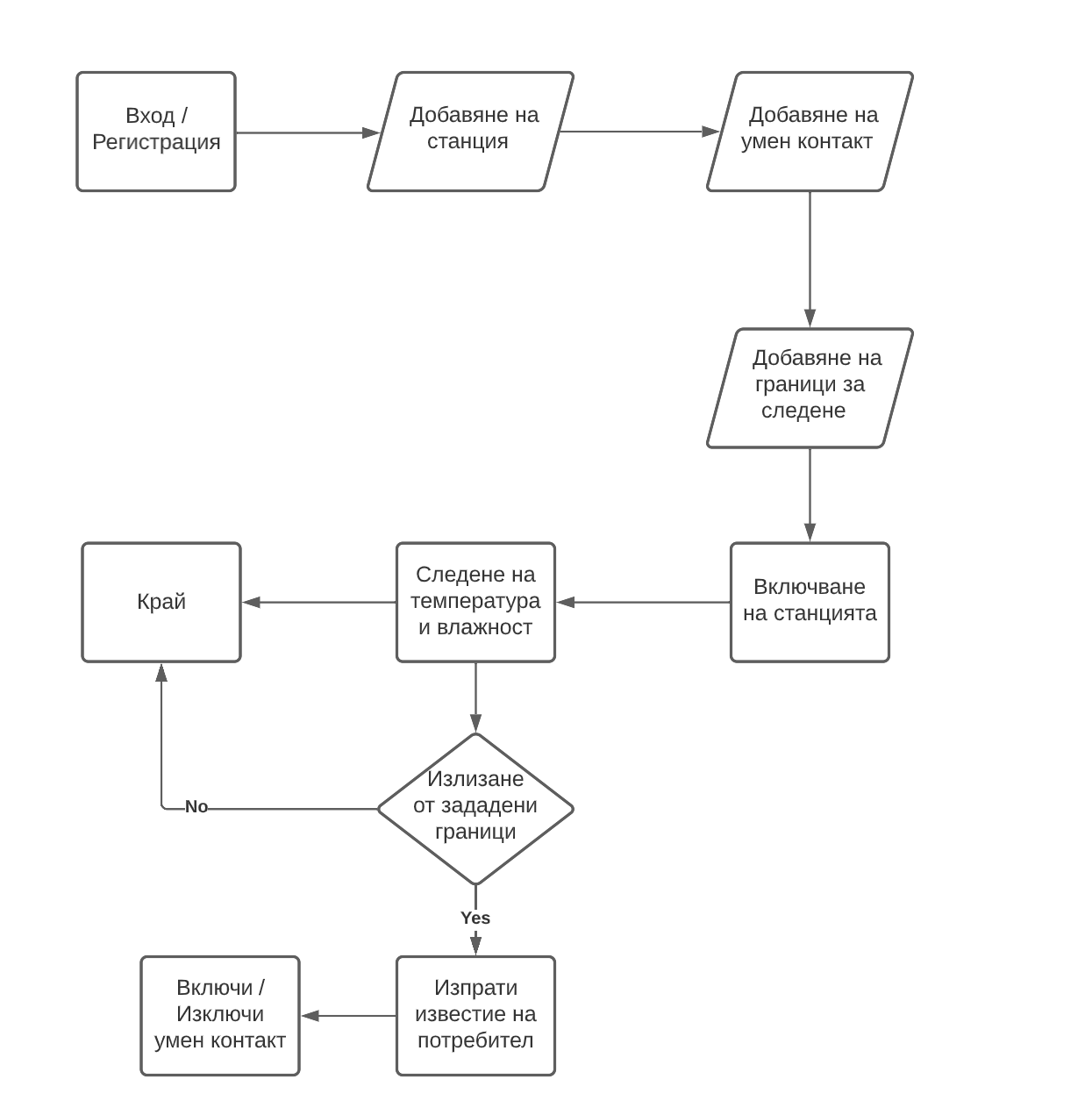
Друга цел на проекта да се предостави възможност на потребителите да следят графиката на температурата за всеки час, като средната температура във всеки един момент от денонощието е запазена.

Проектът също предоставя възможност на потребителите да бъдат известявани, когато някои от зададените критерии биват прехвърлени. Например, ако потребителят е въвел определена величина, за температура, ако тя бъде прехвърлена, освен че станцията ще спре/включи умния контакт, следва да получи известие за отклонението от нормалната температура.

Тъй като станциите могат да контролират множество продукти, следва да бъдат поддържани както температура, така и влажност в определени граници постоянно, дори могат да пускани и спирани множество устройства, които да отговарят за температура и влажност при различни критерии.s

Друга цел на проекта е отдалеченото контролиране на тези показатели. Потребителят не трябва да е на същото място където се намират станциите, тъй като те се контролират от сървър, който не е задължително да работи в същата мрежа. По този начин потребителят е улеснен, тъй като може да управлява и следи показатели от всяка точка на света. Това също е полезно, ако станциите биват използвани на места опасни за човека, например, при твърде високи или ниски температури или на места които съдържат отровни вещества за хората, стига да бъдат поддържани от сензорите.

## **1.3. Workflow диаграма на основното използване на продукта**



Фиг 1. Схема за използване на продукта

Първоначално трябва да се изясни най-важният въпрос, кой и къде ще използва приложението. Апликацията има специфично място за употреба. Клиентите, които ще го ползват трябва да имат нужда от постоянно и точно следене на температура и/или влажност на определено място. Добър такъв пример и поддържането на постоянни показатели, например, в музей. В музеите се намират експонати, които за да бъдат запазени трябва да са изложени на консистентна температура и влажност, които са предварително зададени.

Както става ясно от workflow диаграмата на проекта, първото нещо, което потребител би следвало да направи е да е регистрира свой профил в приложението и след като се е регистрирал, следва да влезе в своя профил.

След като потребителят е влязъл в профила си, той добавя станция за измерване на температурата. Накратко станцията представлява raspberry компютър със свързан сензор за мерене на температура и влажност.

След като е добавена измервателна станция, потребителят добавя умен контакт към станцията. Накратко това е IoT (internet of things) устройство, което може да бъде пускано или спирано през Wi-Fi мрежа.

След като вече има добавен и контакт, потребителят следва да добави граници, при които станциите ще пускат или спират умните контакти. Например, потребителят може да добави горна граница на температура за определена станция и когато тази температура се достигне станцията ще включи или изключи всички контакти, които отговарят за температурата. Същото може да се направи и за влажността. Поддържаните граници са долна и горна, тоест какво да се случи при най-ниска зададена температура и какво да се случи при най-висока зададена температура.

Вече имаме и добавени граници за станцията и следва я включим. С включването станцията започва да праща измерените показатели към сървъра. Данните се запазват за период от 7 дни назад и потребителят може да достъпи графиката през web приложението и наблюдава средната температура за всеки час. Ако се стигне до излизане от зададените граници, сървърът ще включи или изключи един или повече умни контакта и ще изпрати известие на потребителя под формата на имейл с съдържание, коя станция е засякла аномалия в показателите.

Потребителите нямат ограничения колко станции, умни контакти или граници за температура и влажност да добавят.

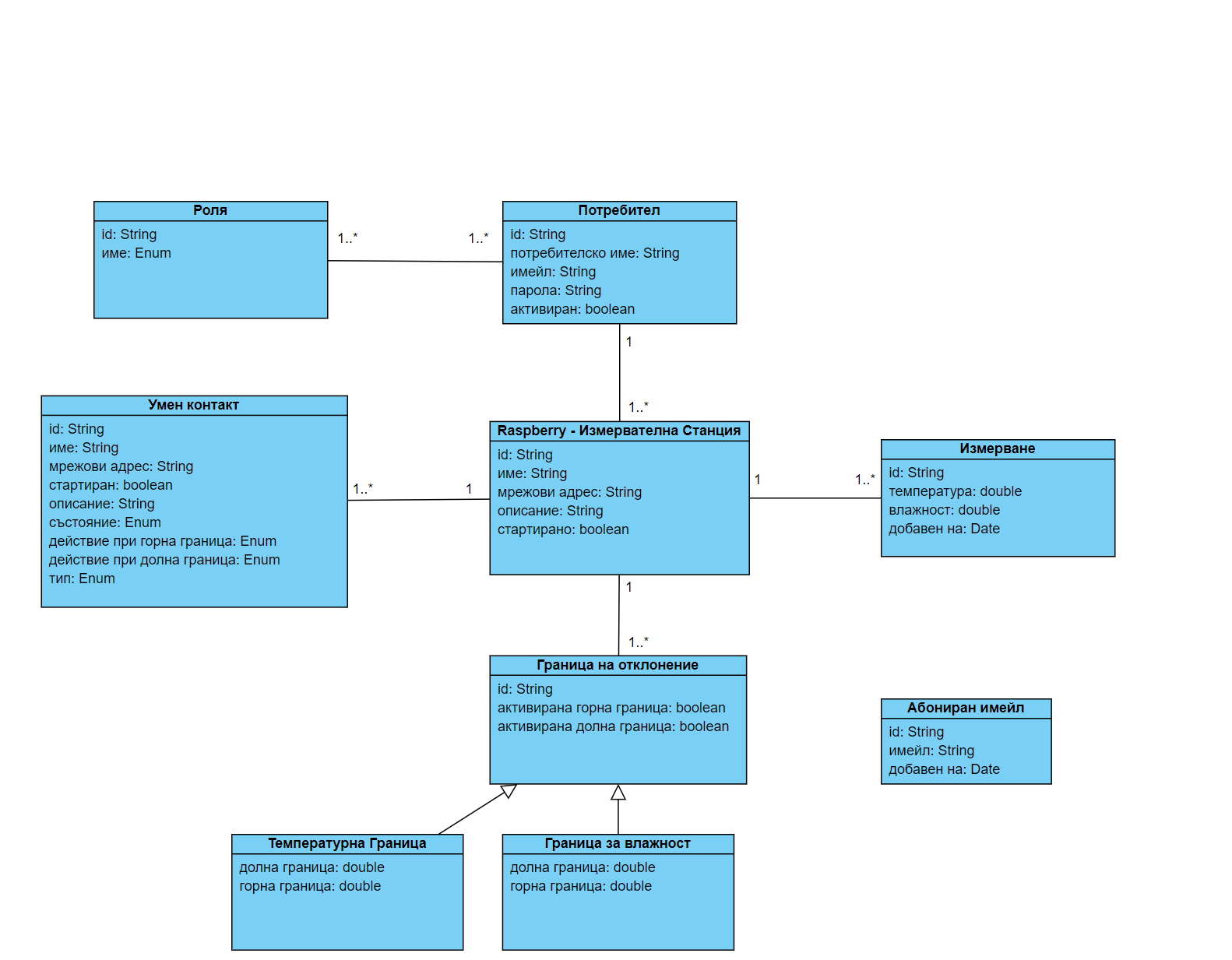
# **2. Проектиране на архитектура на системата**

Както става ясно от описанието на workflow диаграмата на приложението, идеята на проекта е да се автоматизира следенето и контролирането на устройства, които отговарят за изменението на температурата и влажността в затворени помещения.

Системата се състои от няколко компонента:

1. Сървърно приложение – представлява основната изчислителна единица в приложението, както и мястото за съхраняването на всички данни от потребителя и от изпращаните данни от измервателните станции. Сървърът представлява web приложение, което предоставя REST API, през който другите компоненти в системата могат да комуникират с него през HTTP протокола. Сървърът също така ползва релационна база данни, за да пази информацията. През него се добавят измервателните станции и също той ги контролира. Също така станциите пращат информацията за измерените данни към него. Също той контролира и изчислява кога да включи или изключи умен контакт.
2. Клиентското приложение – представлява визуалната част от приложението, с което потребителите на приложението ще работят. То предоставя приятен и интуитивен потребителски интерфейс, през който потребителите се регистрират, добавят, редактират измервателните станции, умните контакти, както и добавят границите, които станциите да използват, за да контролират контактите. Клиентското приложение комуникира само със сървърът през REST API-я му, за да получава и изпраща данни към него.
3. Измервателна станция – тя представлява Raspberry PI компютър, към който има свързан сензор BME 280, чрез който се четат температурата и влажността на въздуха. Върху raspberry PI е инсталиран друг сървър, който също предоставя REST API, чрез който комуникацията с главния сървър се осъществява.
4. Умен контакт (Shelly Plug) – представлява IoT устройство (контакт), което може да се свързва с Wi-Fi мрежа. То също предоставя REST API, чрез което главният сървър изпраща заявки кога да се включи или изключи контактът. Към него следва да бъде включен друг уред, за печка, овлажнител и т.н., който да отговаря на това как е настроен контактът, дали е за температура или влажност.

## **2.1. Основните модели в системата (domain model)**



Фиг. 2 Визуално представяне на домейн модела

По-горе посочената диаграма представлява основните модели в системата.

Най-големите от тях са потребителите, измервателните станции и умните контакти и около тях се базира главната идея на приложението.

Модели:

1. **Потребител** – Представлява лицето, което работи със системата. Зад този клас стои реален човек, който взаимодейства със системата, изпраща или консумира информация. Описание на основните полета:

* ID (идентификационен номер) – уникален номер, който всеки потребител получава когато се регистрира в системата. Той е автоматично генериран и служи за уникалността при търсене по този критерий.
* Парола – това е паролата на потребителя. Ограниченията, които налага е да не е празна и да не надвишава 256 символа.
* Имейл – това е имейлът на потребителя. Той трябва да бъде съобразен със стандарта за валиден имейл, да бъде уникален и също така да не надвишава 256 символа.
* Потребителско име – това е името, с което потребителят работи най-много. С това име потребителят се автентикира в системата и също така се използва, като името, с което потребители могат да се различават един от друг.
* Активиран – това поле се ползва от системата, за да валидира дали потребителят си е активирал акаунта. Това става през отварянето на линк изпратен на имейла на потребителя след неговата регистрация. Ако потребителят не активира своя профил, той не може да влезе в него и да ползва функционалностите на приложението.

1. **Роля** – този клас представлява роли, с които потребителите разполагат. Един потребител може да има повече от една роля. Всяка роля дава допълнителни права, с които съответният потребител разполага. Съществуват три типа роли. Първият тип роля е “User”. Тази роля я получава всеки потребител и тя има най-ограничени права. Правата, които тя предоставя са следните. Потребителят може да навигира свободно в приложението, да добавя измервателни станции, да добавя умни контакти, да добавя граници за отклонение, както и да стартира и спри измервателни станции. Следващата роля е „Moderator“, като се нея потребител може да вижда изпратените имейли от „абонирай се за приложението“, което се показва на началния екран на приложението, също така може да променя роли на други потребители. Ролята с най-големи правомощия е „Admin“ (администратор), като тя включва всички правя предоставени от „Moderator“ ролята, както и изтриването на потребители. Ролите „Moderator“ и „Admin“ се дават автоматично на първия регистрирал се потребител, това е направено така с цел да може да име поне един потребител с пълни правомощия в системата, като се трябва да се има предвид, че щом се регистрира първия потребител системата все още е в процес на първоначално настройване. Полетата, които съдържа ролята са:

* ID (идентификационен номер) – идентификационният номер се генерира при създаването на ролята. Трите горепосочени роли се генерират, ако не съществуват, при първото пускане на приложението и се запазват в базата с данни.
* Име – това е полето, в което се запазват имената на ролите. В момента ролите, с които разполага приложението са “User”, “Moderator” и “Admin”.

1. **Raspberry Измервателна станция** – това е класът, който отговаря за измервателните станции в системата. Зад него стои, т.н. изчислително устройство (компютър) Raspberry PI, който служи за събирането на текущите показатели и изпращането им. Потребителят отговаря за добавянето на измервателните станции, както за настройката на техните адреси и имена. Полета за този клас са следните:

* ID (идентификационен номер) – автоматично генериран при добавянето на нова измервателна станция.
* Име – име на измервателната станция. Служи за идентифицирането на станциите от потребителя. Ограничението, което има е да не е празно.
* Мрежов адрес – това е мрежовият адрес, който измервателната станция има. Когато е включена в домашна мрежа обикновено трябва да е статичен. Мрежовият адрес трябва да е уникален в системата. Също ограничение е адресът да не е празен.
* Описание – това поле е опционално, тоест потребителят не е длъжен да добавя нищо в него. Служи за подсещане какво точно прави тази измервателна станция. Ограничението е да е до 8196 символа. Може да е празно.
* Стартирано – това е флаг, който се ползва от приложението, за да се знае дали е пусната или спряна измервателната станция.

1. **Измерване** – това е класът, който отговаря за измерванията, които се пращат от измервателните станции към главния сървър. Полетата, които съдържа са:

* ID (идентификационен номер) – автоматично генериран при добавянето на ново измервате.
* Температура – температурата измерена от сензора на измервателната станция. Тя е число и не трябва да бъде празна.
* Влажност – влажността измерена от сензора на измервателната станция. Тя е число и не трябва да бъде празна.
* Добавено на – автоматично генерираната дата и час, в който измерването е записано в базата данни. Ограничението е да не е празно.

1. **Умен контакт**  – това е класът, който отговаря за умните контакти в приложението. Полетата, които съдържа са:

* ID (идентификационен номер) – автоматично генериран при добавянето на новия умен контакт.
* Име – име на умния контакт. Служи за идентифицирането на контактите от потребителя. Ограничението, което има е да не е празно.
* Мрежов адрес – това е мрежовият адрес, който умният контакт има. Когато е включен в домашна мрежа обикновено трябва да е статичен. Мрежовият адрес трябва да е уникален в системата. Също ограничение е адресът да не е празен.
* Стартирано – това е флаг, който се ползва от приложението, за да се знае дали е пуснат или спрян умният контакт.
* Описание – това поле е опционално, тоест потребителят не е длъжен да добавя нищо в него. Служи за подсещане какво точно този умен контакт. Ограничението е да е до 8196 символа. Може да е празно.
* Състояние – това е състоянието, в което умният контакт се намира в момента. Състоянието на контакта в момента може да бъде включено или изключено, но това поле дава възможността за добавянето на други състояния на контакта. Стойността по подразбиране не изключена. Също не се позволява това поле да е празно.
* Действие при горна граница на отклонение – това поле представлява действието, при което ако се достигне горната граница на отклонение ще направи умният контакт. Действията, които може да извърши контакта в случая са включване и изключване. Това се задава от потребителя в зависимост от устройството захранвано през контакта. Не може да бъде празно.
* Действие при долна граница на отклонение – това поле представлява действието, при което ако се достигне долната граница на отклонение ще направи умният контакт. Действията, които може да извърши контакта в случая са включване и изключване. Това се задава от потребителя в зависимост от устройството захранвано през контакта. Не може да бъде празно.
* Тип – това поле представлява какъв тип устройство ще отговаря на контакта. Дали ще бъде от тип, който променя температурата или от тип, който променя влажността на въздуха. Това поле също се задава от потребителя и не може да бъде празно.

1. **Граница на отклонение**  – това е базов клас, на двете граници на отклонение, за температура и влажност. Класът, съдържа общите полета за двата наследника. Полетата, които съдържа са:

* ID (идентификационен номер) – автоматично генериран при добавянето на границата за отклонение.
* Активирана горна граница – това поле означава дали горната граница ще се вземе предвид в случай, че бъде достигната. Ако не е активирана, дори и станция да отчете горна граница на измерване, сървърът няма да изпрати заявка за включване или изключване към умните контакти.
* Активирана долна граница – това поле означава дали долната граница ще се вземе предвид в случай, че бъде достигната. Ако не е активирана, дори и станция да отчете долна граница на измерване, сървърът няма да изпрати заявка за включване или изключване към умните контакти.

1. **Температура граница на отклонение**  – Този клас отговаря за температурните граници на отклонение. Тоест каква ще е максималната или минималната температура, при която сървърът ще включи или изключи умен контакт, ако е границата е активирана.

* Долна граница – това е долната границата, която ако бъде достигната ще, сървърът ще включи или изключи умните контакти.
* Горна граница – това е горната границата, която ако бъде достигната ще, сървърът ще включи или изключи умните контакти.

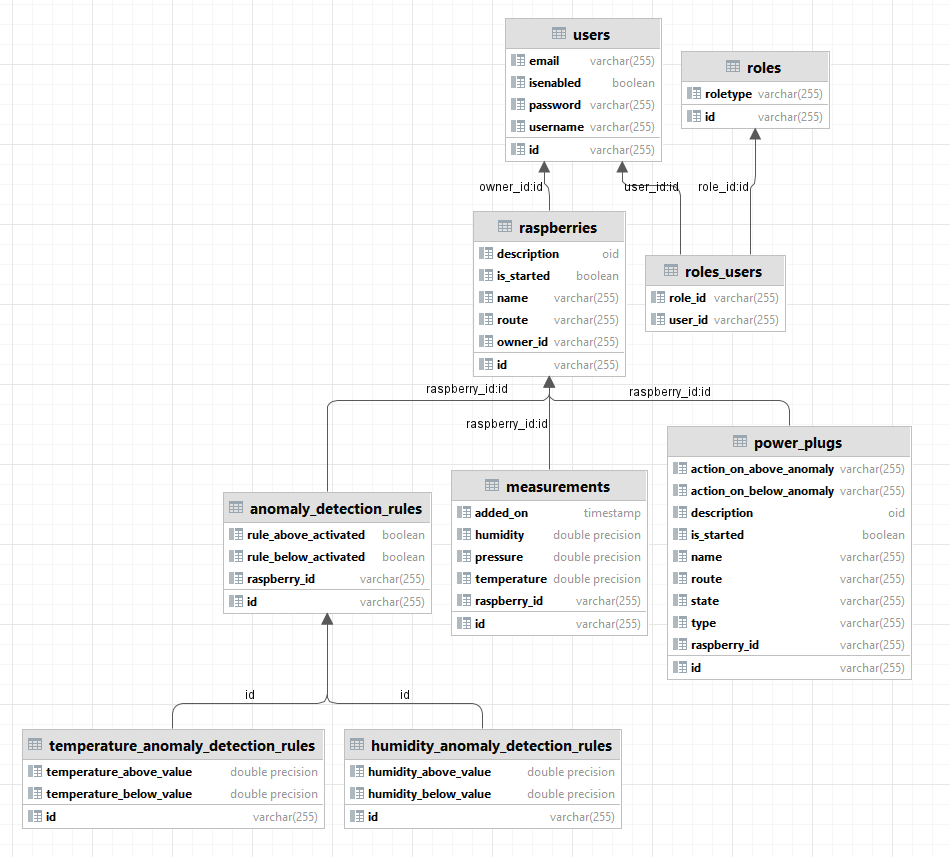
1. **Абониране имейли**  – Този клас отговаря за имейлите изпратени от нерегистрираните потребителите. Няма релации и отговаря сам за себе си. Съдържа следните полета:

* ID (идентификационен номер) – автоматично генериран при добавянето на абониран имейл.
* Имейл – това е имейлът на потребителя. Той трябва да бъде съобразен със стандарта за валиден имейл, да бъде уникален и също така да не надвишава 256 символа.
* Добавено на – автоматично генерираната дата и час, в който абонираният имейл е записан в базата данни. Ограничението е да не е празен.

## **2.2. ER (entity relationship) диаграма**

Подходът, който е използван за създаването на базата е “Entity first approach”. Този подход представлява начин, чрез който първо в кода на приложението се дефинират класовете, които ще бъдат основните таблици в базата, след това с добавят и връзките между класовете, което резултира в чужди ключове и нови спомагателни таблици при някои от връзки. Връзките между таблиците бива три вида: едно-към-едно, едно-към-много и много-към-много. В тази база има и от трите вида връзки, като най-много са едно-към-много. При формирането на връзка едно-към-едно във една от двете таблици се добавя чужд ключ, който е свързан с първичния ключ в другата връзка. Връзка едно-към-едно има само при наследените таблици за граничните отклонения, тази релация идва от използваната библиотека Hibernate за връзката с таблиците наследници. Вторият вид връзка е едно-към-много. При него в таблицата, която има много записи, или с други думи, таблицата към, която е N връзката, съдържа чужд ключ, който е първичния ключ от другата таблици. Така повече от един ред от едната таблица може да сочи към един единствен ред в другата. Пример за такава връзка в базата е: “user” -> “raspberries ”. Третата връзка, която съществува между таблиците е много-към-много. При нея всеки един от ред във всяка една от таблиците е възможно да сочи в повече от един от ред в другата. Тази връзка се реализира чрез трета помощна таблицата, която има две колони, които съответно са чужди ключове, които сочат към първичните ключове в другите две таблици. В най-честия случай първичният ключ на спомагателната таблица е съвкупността от двата ключа, които тя съдържа. Пример за такава връзка в базата е “users” -> “roles”. Помощната таблицата, чрез която се реализира връзката е “users\_roles“.

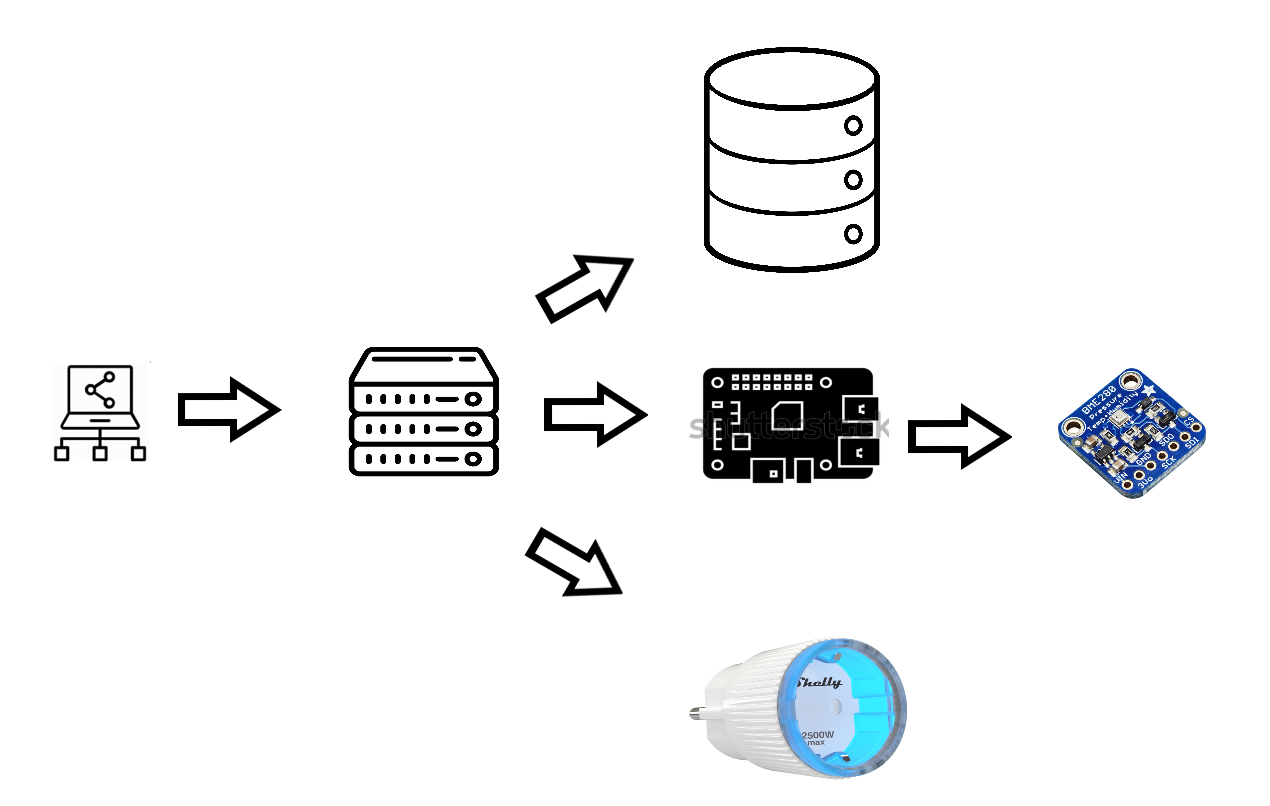
Таблиците, които съдържа базата се генерират чрез domain моделите. За тази цел се използва допълнителна библиотека Hibernate, която прави създаването на таблиците и връзките между тях на базата на анотации. Таблици, които са реализирани на тяхната база са следните:



Фиг. 3 ER диаграма на продукта

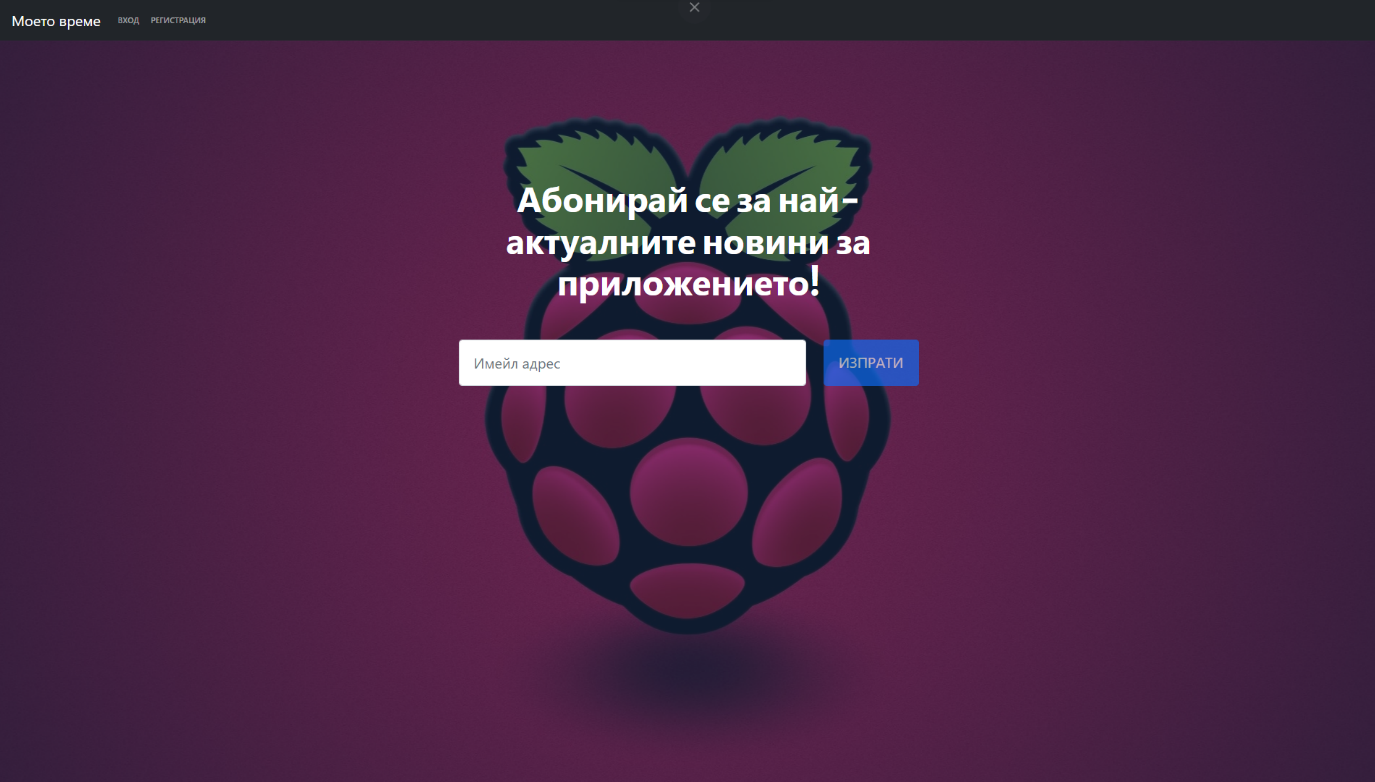
1. “users” – Таблицата пази информация за потребителите на приложението. Целта е да се съхранява информация за потребителя. Чувствителната информация, която в случая е паролата, не се пази в чист вид. Използва се хеширащ алгоритъм, който също така добавя и сол към паролата и следователно чак тогава се запазва в базата. Това се прави с цел допълнителна сигурност и дори данните от базата да бъдат взети по някакъв начин, автентикацията с даден потребител няма да е възможна, защото паролата не се пази в чист вид.
2. “roles” – В тази таблица се пази информация за ролите, които потребителите притежават. Таблицата е свързана с таблицата “users” чрез допълнителна таблица, за да може да осъществи много-към-много връзката.
3. “roles\_users” – Това е допълнителна таблица, в която се държат ключовете на таблиците “roles” и “users”, като се ползва за осъществяването на много-към-много връзката.
4. “raspberries” – В тази таблица се пази информация за измервателните станции. Съдържа се достатъчна информация, за да могат лесно намирани и да бъдат достатъчно полезни за крайните потребители. Таблицата съдържа чуждия ключ към базата с потребители, която е от тип едно-към-много.
5. “power\_plugs” – В тази таблица се държи се държи информацията за умните контакти. Запазват се всички важни данни за контактите. Таблицата има връзка към “raspberries” таблицата посредством чуждия ключ на измервателната станция. Връзката също е едно-към-много.
6. “measurements” – В тази таблица се държи информацията измерванията изпращани от измервателните станции. Таблицата има връзка към “raspberries” таблицата посредством чуждия ключ на измервателната станция. Връзката също е едно-към-много.
7. “anomaly\_detection\_rules” – В таблицата се държи информация за границите на отклонения. Това е базова таблица за другите две таблици “temperature\_anomaly\_detection\_rules” и “humidity\_anomaly\_detection\_rules”. Основната таблица е свързана с наследниците посредством първичните и чуждите ключове. Използван е този подход, за да се нормализира базата и да се избегнат излишни повторения на данни.
8. “temperature\_anomaly\_detection\_rules” – Таблица наследник. Използва се, за да се пази допълнителна информация за границите за отклонения на температура.
9. “humidity\_anomaly\_detection\_rules” – Таблица наследник. Използва се, за да се пази допълнителна информация за границите за отклонения на влажност.

## **2.3. Цялостна архитектура на приложението**

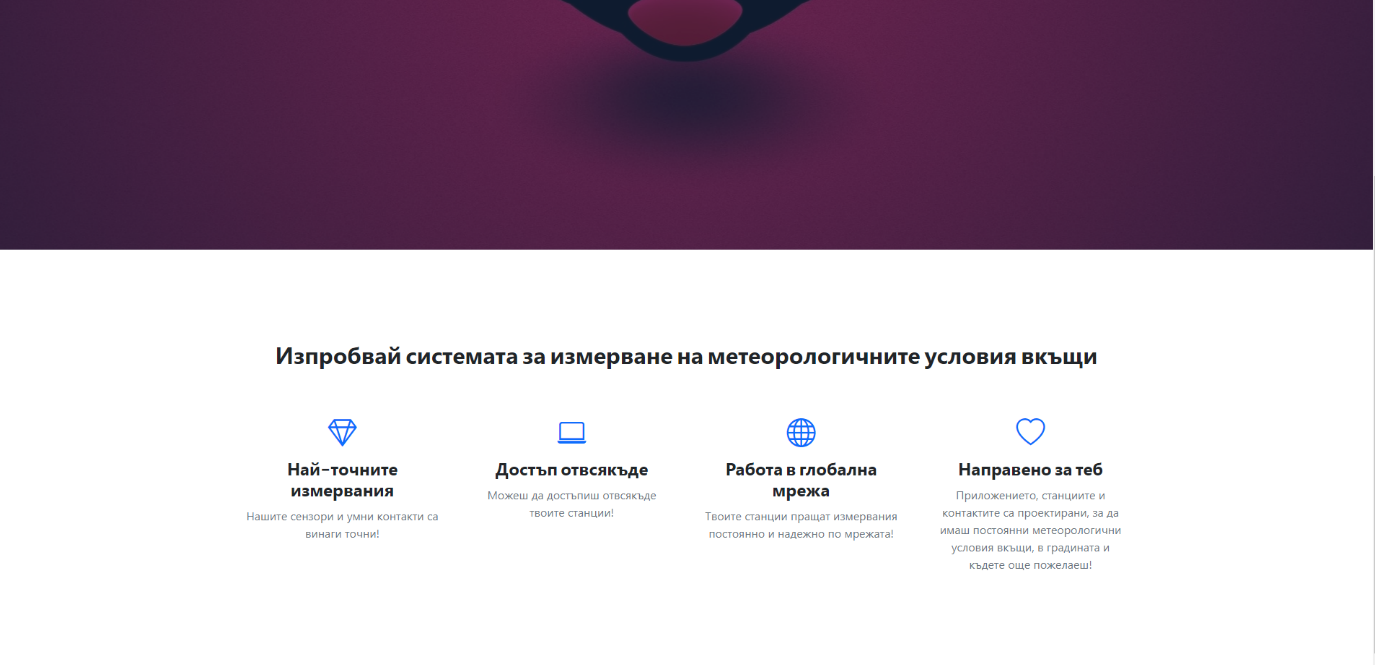


Фиг. 4 Архитектура на приложението

Основният екран на нерегистриран потребител има няколко линка само. Съответно те са за влизане в профила, за регистрация в системата, както и за изпращането на имейл към системата.

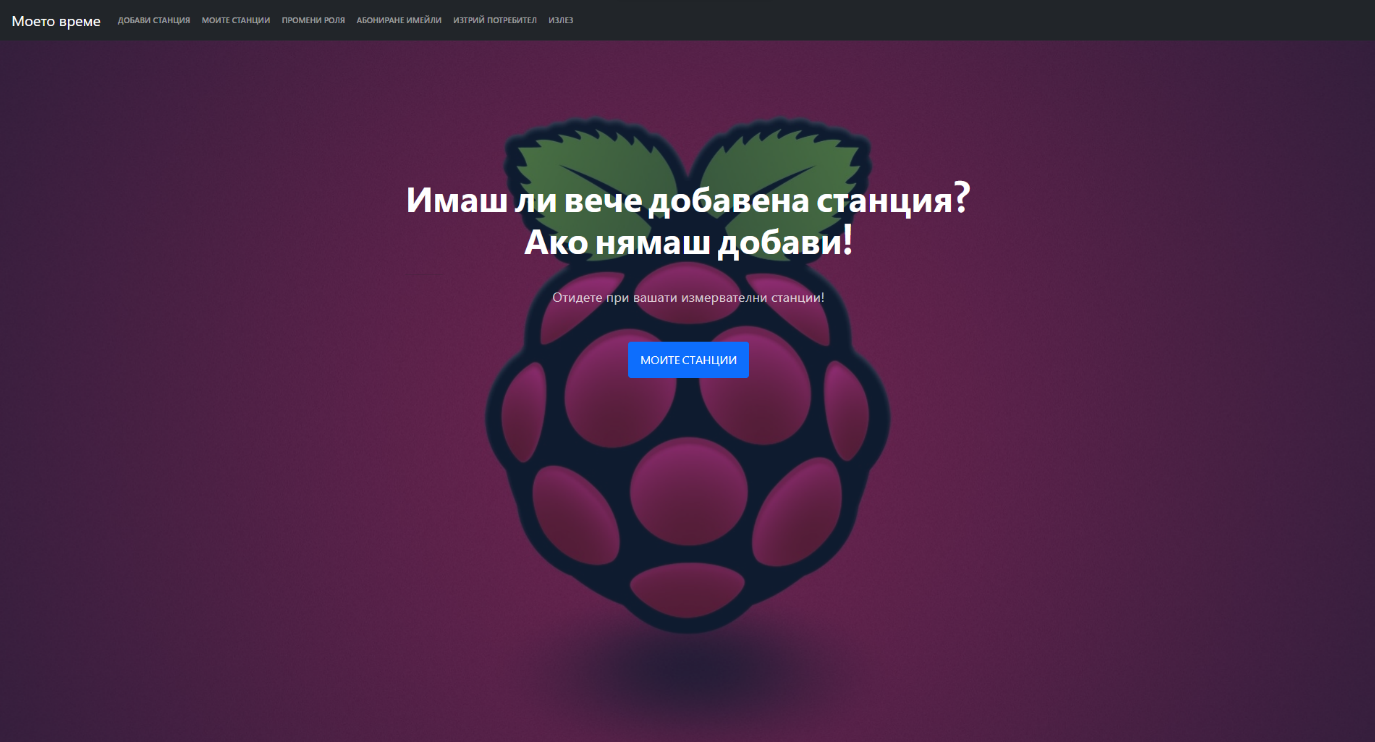


Фиг. 5 Начална страница на приложението на потребител, който не е влязъл в профила си – част 1



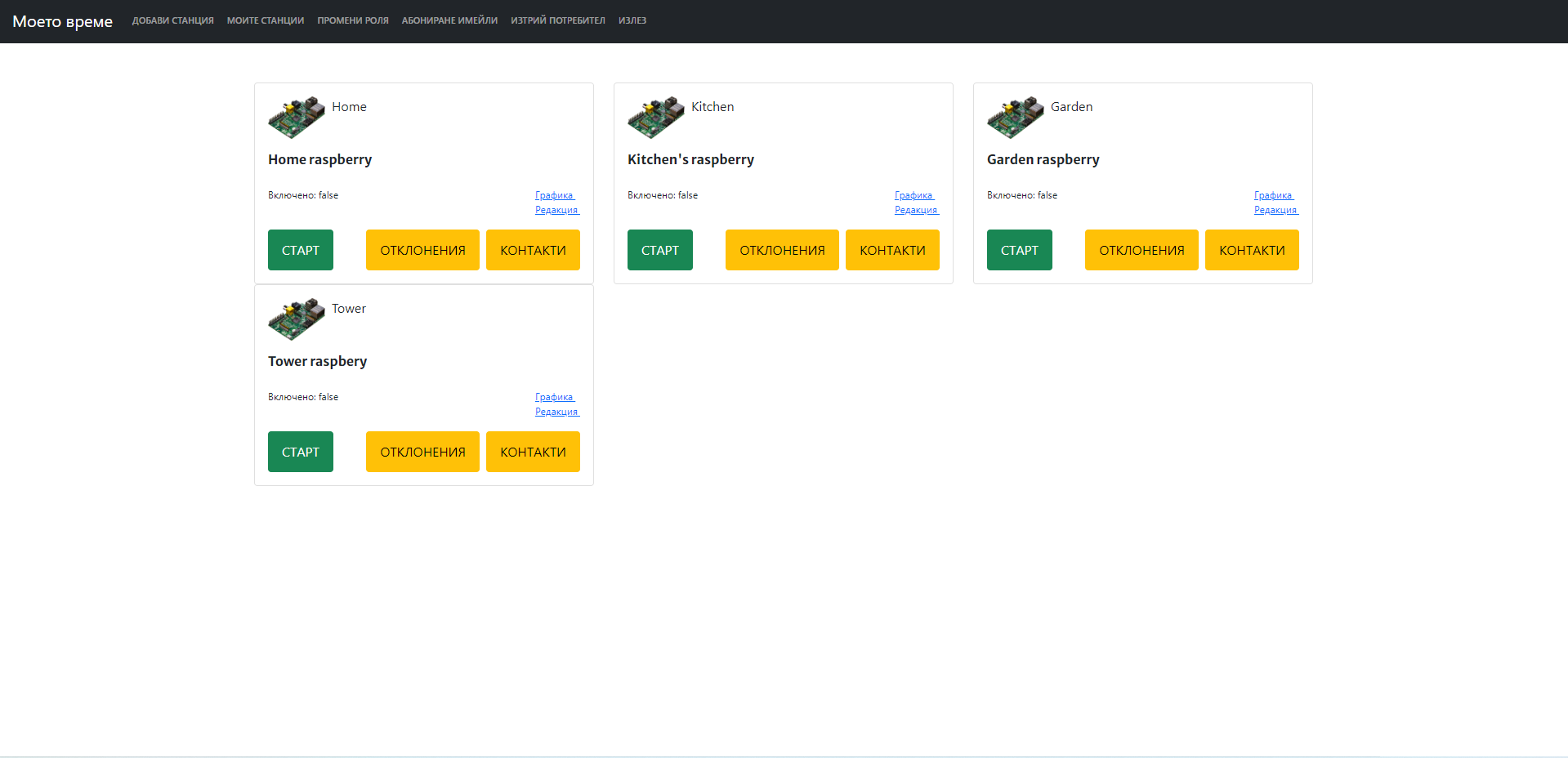
Фиг. 6 Начална страница на приложението, който не е влязъл в профила си – част 2

Основния екран на приложението на влязъл потребител е относително прост. Потребителят може да избере да отвори „Добави станция“, „Моите станции“ или „Излез“, да излезе от профила или да избере „Моето време“, като по този начин се връща в началния екран на приложението. В случая е показана началната страница на потребител, който притежава роля администратор, която съдържа два допълнителни линка за променяне на роля на потребител и за премахване на потребител, както и да вижда абонираните потребители. Също така в средата на страницата има бутон, който също води към измервателните станции. Направен е цел улеснение.



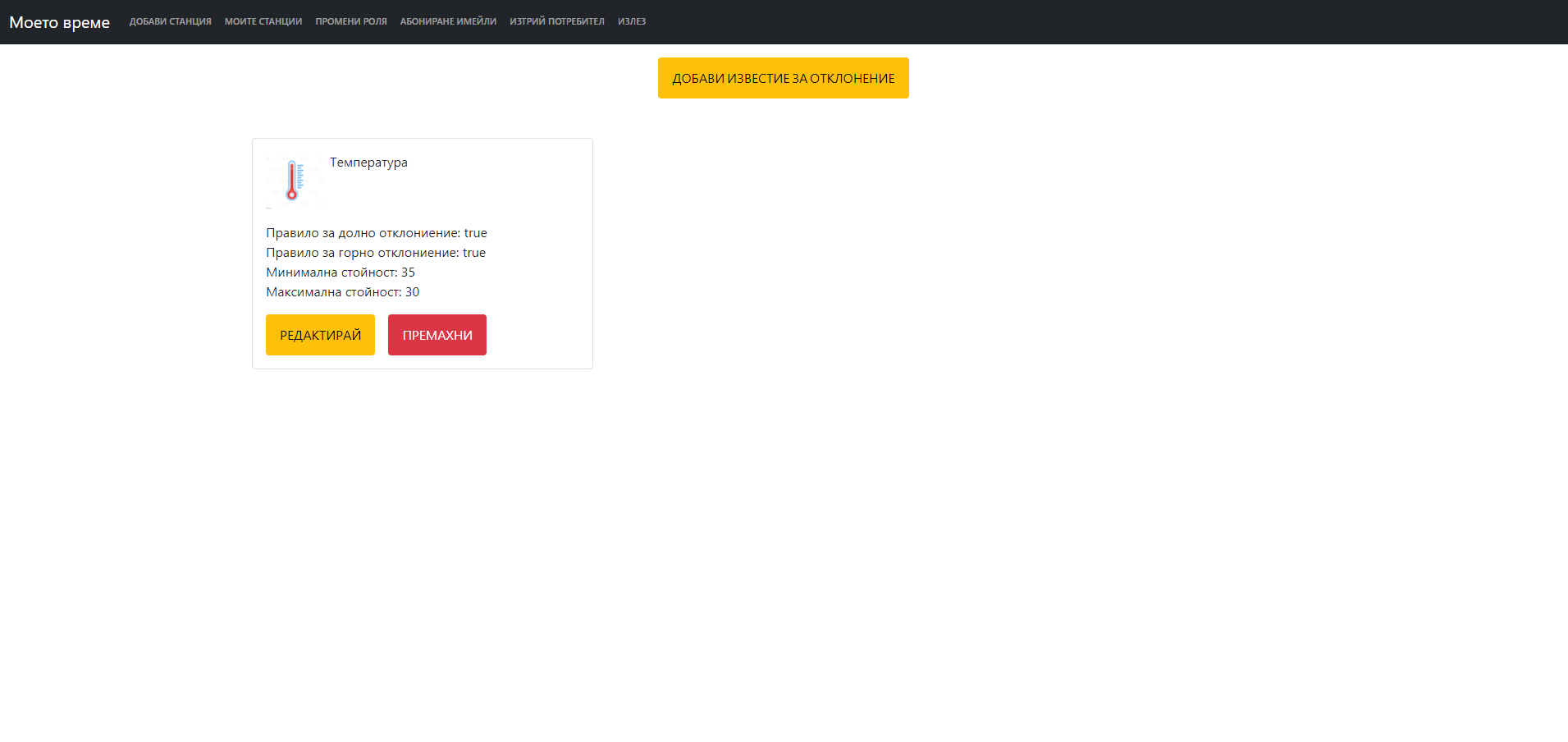
Фиг. 7 Начална страница на приложението на влязъл потребител

Това е началната страница на приложението на влязъл в профила си потребител. Регистрираният потребител има възможността да добавя станции в приложението, като отвори линка от горната лента на страницата, който сочи към страницата за добавяне на станции. След като потребителят е добавил измервателна станция, той може да достъпи страницата за „Моите станции“.



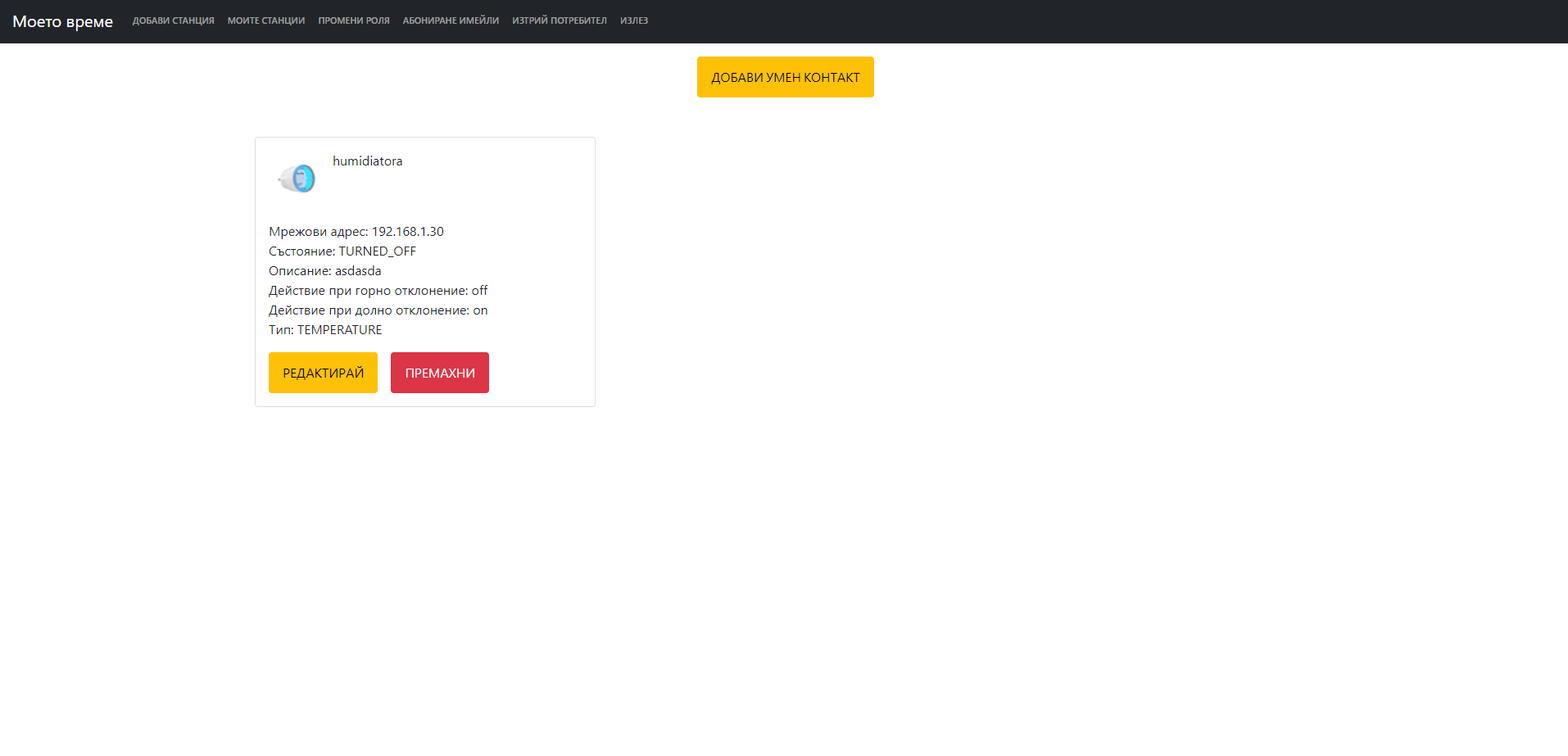
Фиг. 8 Измервателните станции на потребител

Измервателните станции се добавят от потребителя. Това е обобщена страница, където потребителят може да вижда своите добавени измервателни станции. За всяка една станция се вижда името, описание и дали е включено. Също то може да бъде стартирано, тоест измервателната станция да започне да праща измерванията към главния сървър. Също след като е стартирано бутонът ще се промени на “Спри”, който отговаря за спирането на изпращане на данни от измервателната станция. Може да се видят границите на отклонение чрез натискането на бутона „отклонения“. Също така може да се видят умните контакти чрез натискането на „контакти“. Измервателните станции могат да бъдат променяни чрез „редактиране“. Може да се отвори и графика за деня, която визуализира графиката на температурата и влажността като показва средната стойност за всеки час. Графики могат да се виждат за предните 7 дни, по-стари от 7 дни се изтриват.



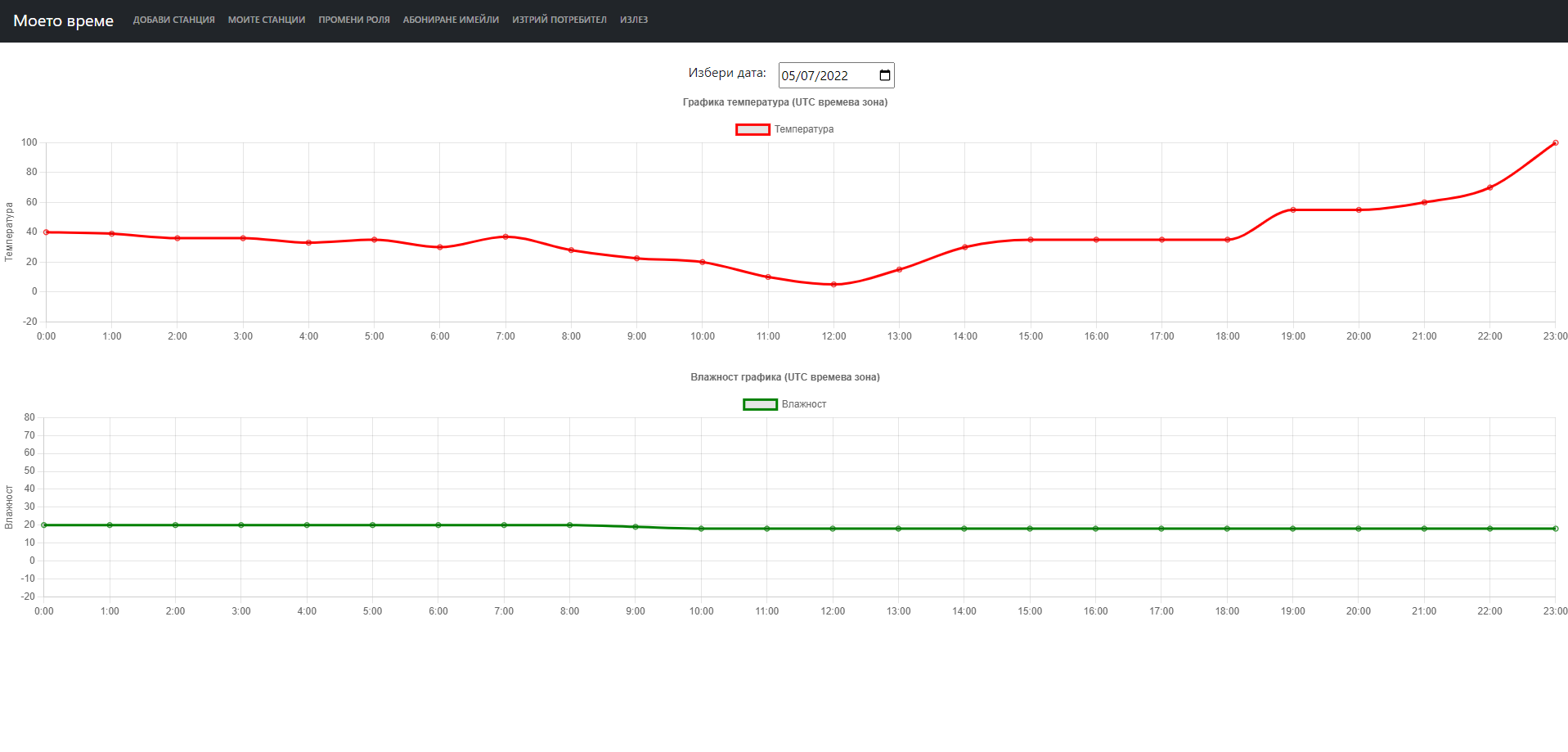
Фиг. 9 Страница с граници на отклонение

Страницата за граници за отклонения дава възможност на потребителите да виждат добавените граници. Когато някоя от тези граници бъде достигната и правилата за отклонения са включени, сървърът ще изключи или включи един или повече умни контакти. Потребителят може да изтрива или редактира отклоненията. Също така най-горе има бутон, който води до друга страница, през която могат да се добавят граници на отклонения.

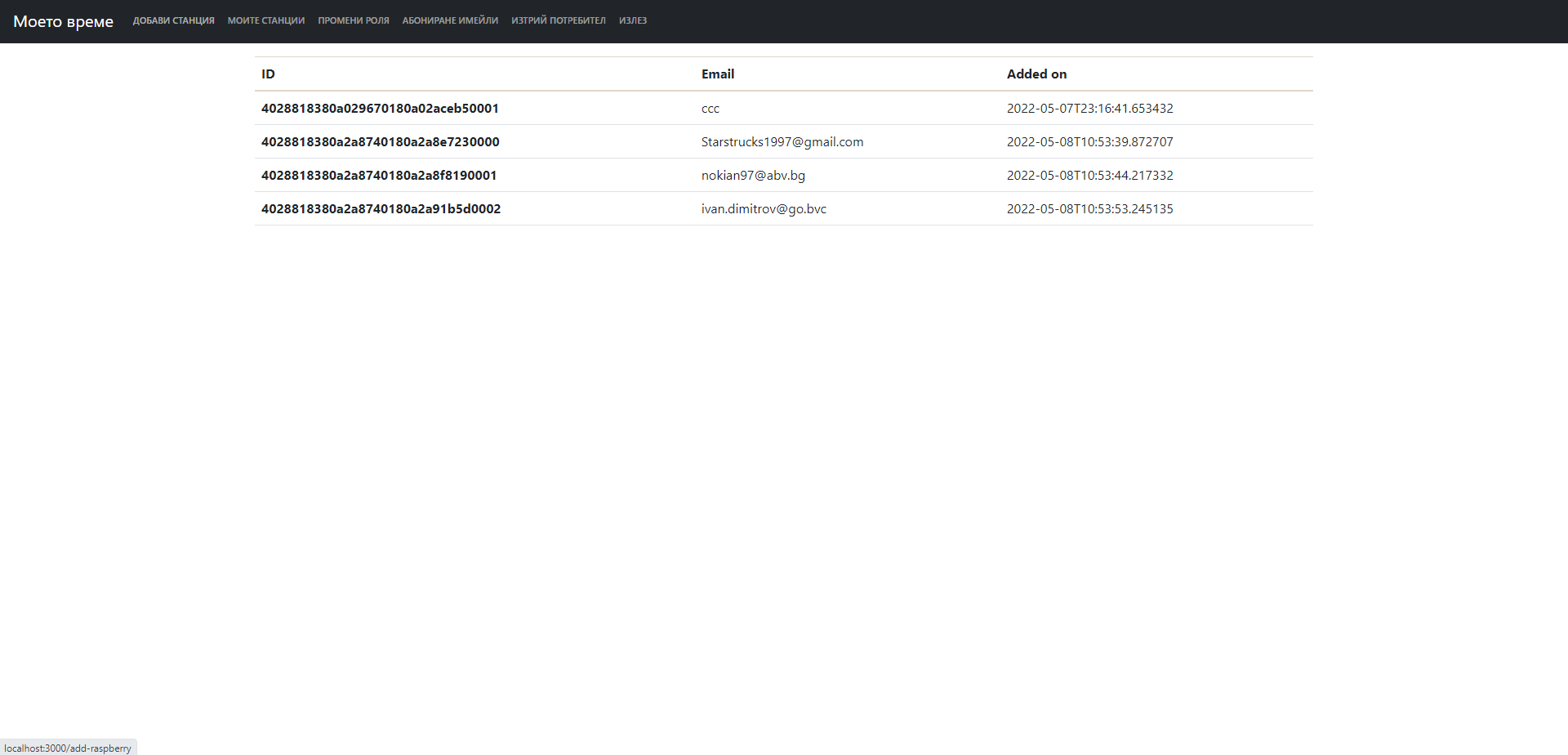


Фиг. 10 Страница за умни контакти

Страницата съдържа умните контактите, които са свързани към тяхната измервателна станция. Потребителят има правото да ги редактира и премахва. Също горе в страницата, има бутон, който води към друга страница за добавянето на нов умен контакт към тази измервателна станция.

 Фиг. 10 Страница за визуализиране на графиката на осреднена температура за час

Графиките представлява измерваната температура и влажност за всеки час от денонощието. Точките за всеки час се изчисляват като се взима средноаритметичната температура за всеки час.

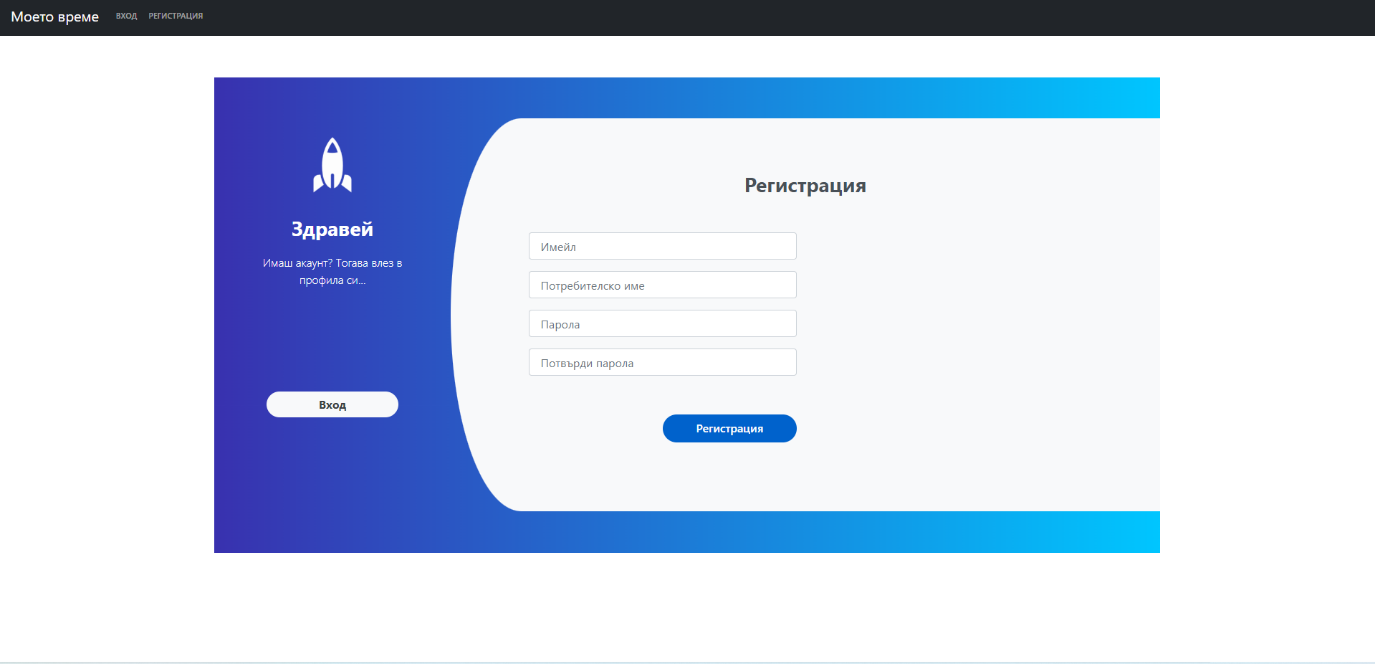


Фиг. 11 Страницата за изпратените имейли от началната форма

Страницата съдържа информация за имейлите на потребители, които са изпратили своите имейли през началната страница. Тези имейли могат да бъдат достъпвани само от потребителите, които роля администратор или модератор. Имейлите могат да бъдат използвани за бъдеща връзка с тези потребители.

## **2.4. Анализ на входни данни на потребител, модератор и администратор.**

Потребител – има достъп до някои от страниците в приложението и въвеждането на данни понякога е наложително.

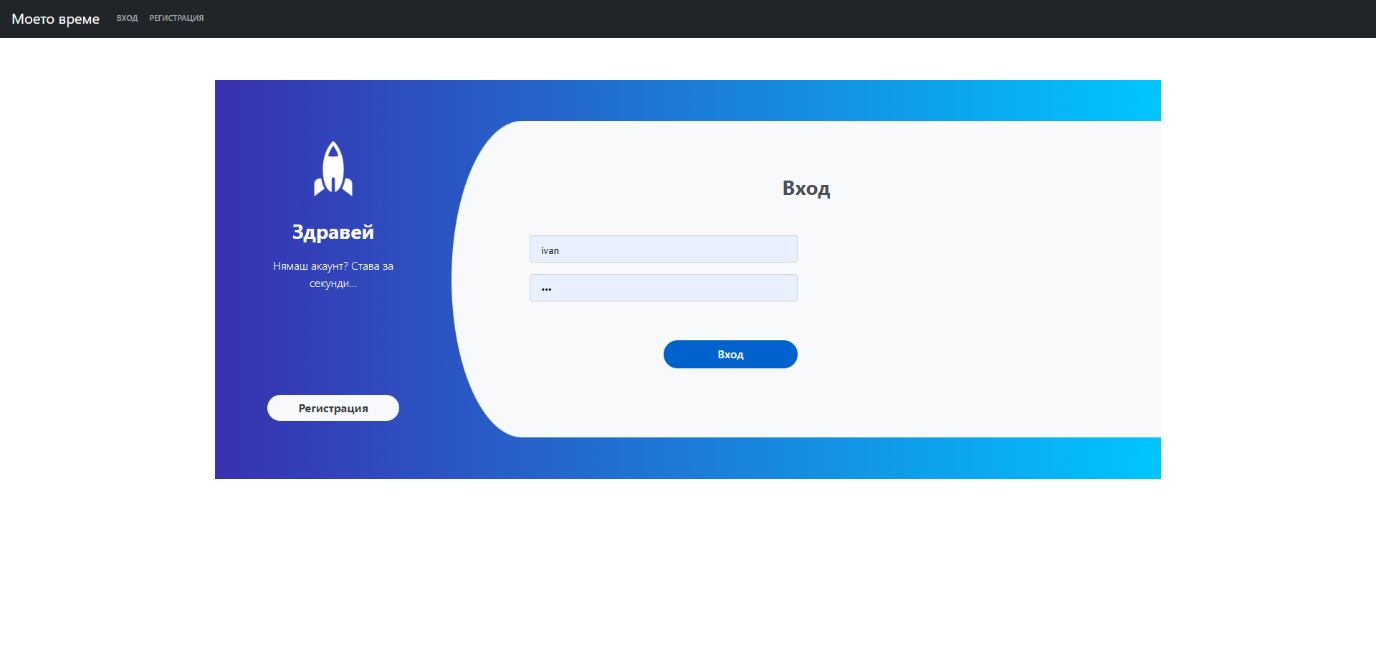


Фиг. 12 Страницата за регистрация на потребител

Когато потребител не е регистриран в приложението, това е първото нещо, което трябва да направи. За да се регистрира той трябва да въведе следните данни:

1. Имейл – имейл адресът на потребителя. Трябва да бъде уникален в цялата система, в противен случай потребителят ще бъде известен с съобщение за грешка, че въведеният имейл е вече използван от друг потребител. След регистрацията потребител ще получи имейл, който съдържа линк за активация на акаунта. Потребителят трябва да отвори своята имейл поща и да отвори линка, след което той ще може да влезе в профила си. Ако не е активиран акаунтът, потребителят не може да влезе в своя профил и ще получи съобщение за грешка.
2. Потребителско име – потребителското име на потребителя. Също трябва да бъде уникално в цялата система. То се избира от потребителя и това е името, с което потребителят се автентикира на следващ етап в приложението.
3. Парола – паролата на потребителя. Избрана от него и няма валидация за специална символи. Може да бъде произволна.
4. Потвърди парола – същата парола, валидация че потребителят е написал вярно своята парола тъй като се вижда като скрит текст, когато се въвежда.

Ако потребителят вече има профил, от този страница той може да отвори също страницата за вход.

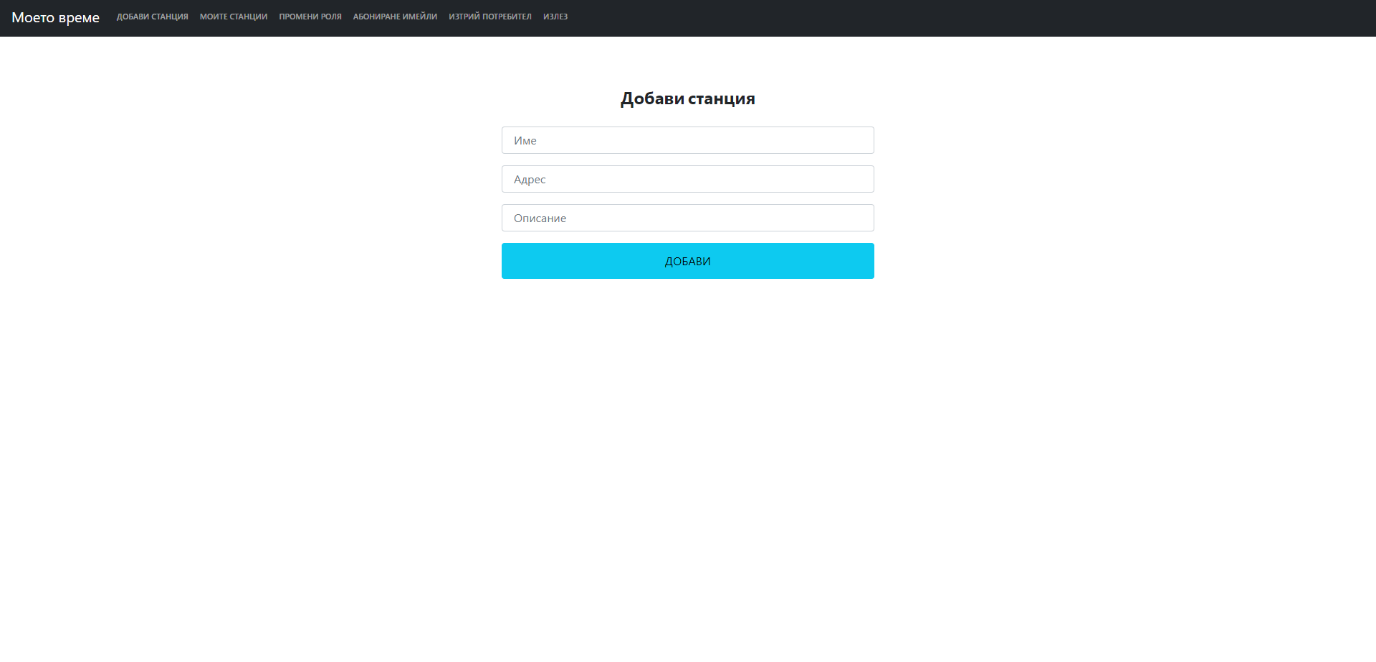


Фиг. 13 Страницата за вход на потребител

За да влезе в профила си, потребителят трябва първо да е активирай акаунта си през имейлът изпратен на пощата му. След това трябва да въведе слените данни:

1. Потребителско име – потребителското име на потребителя, което той е въвел по време на регистрацията.
2. Парола – паролата на потребителя.

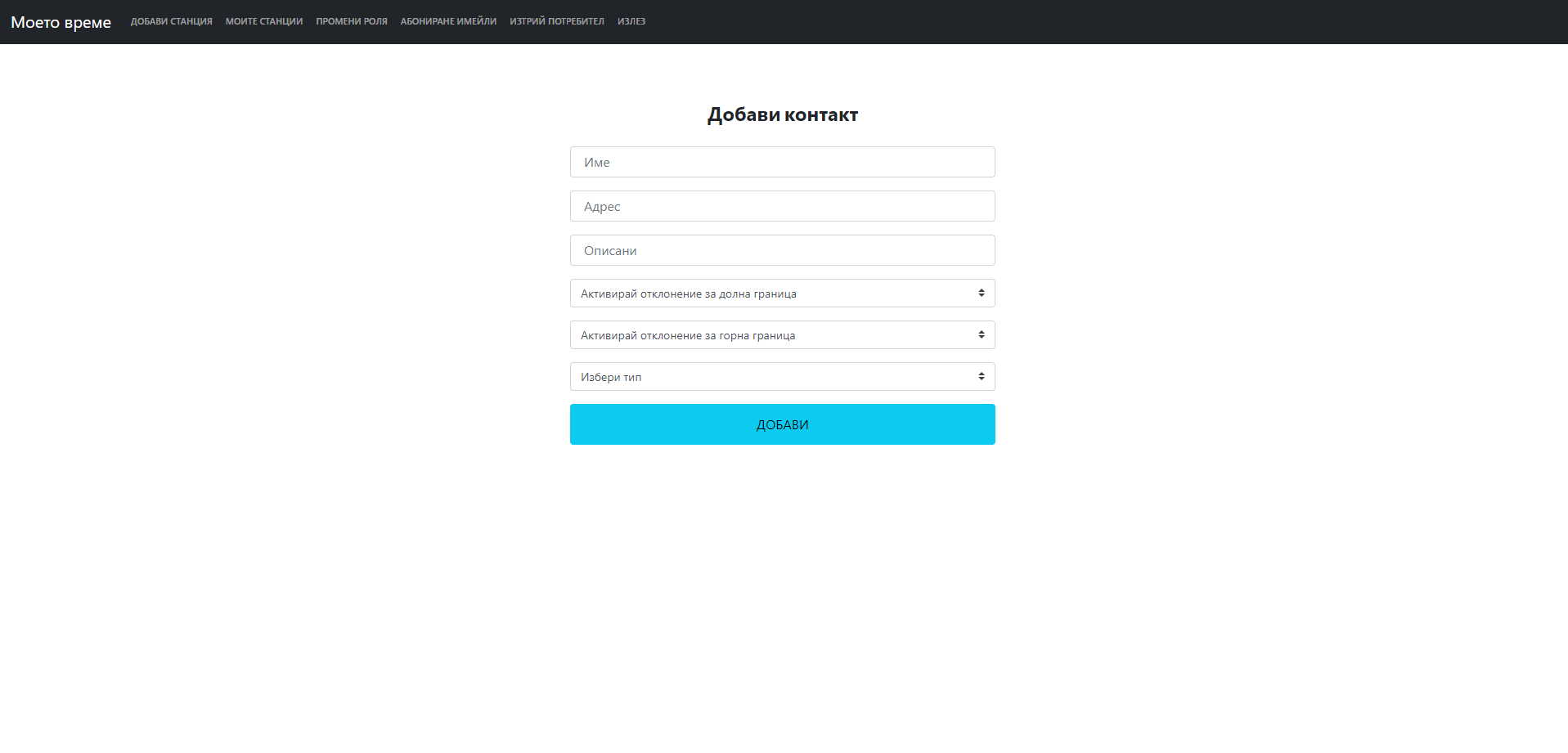
След като потребителят влезе в профила си, той има достът до приложението и е пренасочен към главната страници за потребители.



Фиг. 13 Страницата за добавяне на измервателна станция

Потребител, който е влязъл в профила си може да достъпи страницата за добавяне на измервателни станции. Данни, които потребителят въвежда са следните:

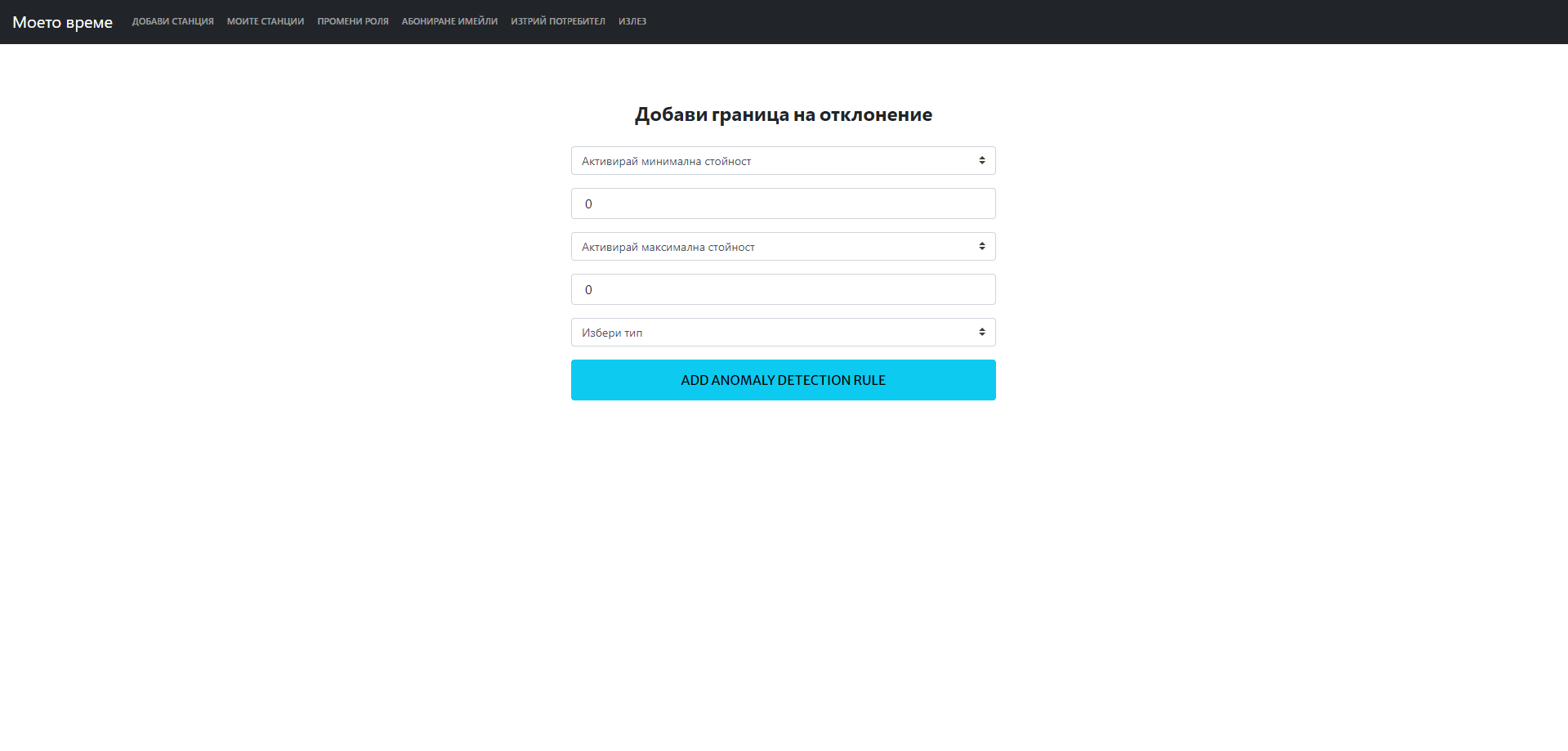
1. Име – името на измервателната станция. Произволно име на измервателната станция. Произволно избрано от потребителя.
2. Адрес на измервателната станция – мрежовият адрес на измервателната станция. Ако за нея няма закачен домейн, може да се използва IP адресът на станцията. Препоръчително е да е зададен предварително на raspberry PI компютъра (измервателната станция) и да е статичен. Когато системата се ползва в локална мрежа адресът има вид „192.168.1.100“.
3. Описание – произволно описание на измервателната станция. Това поле може да бъде оставено празно и служи само за улеснение на потребителя.



Фиг. 14 Страницата за добавяне на умен контакт

Това е страницата, която потребителят използва, за да добавя умни контакти. Следователно информацията, която е необходима е:

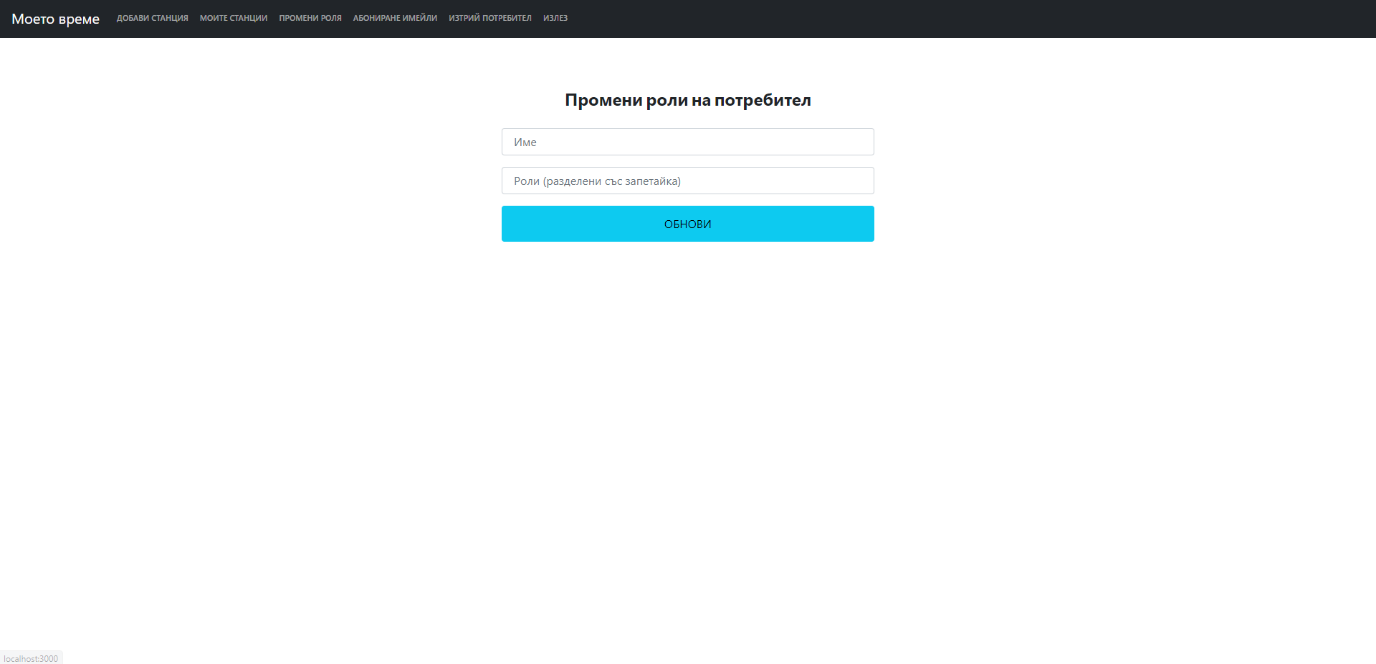
1. Име – името на умния контакт. Произволно име избрано от потребителя. Служи за лесното идентифициране на контакта от потребителя.
2. Адрес – мрежовият адрес на умният контакт. Както при адреса на измервателната станция, когато системата работи в локална мрежа е препоръчително да е зададен статичен такъв.
3. Описание – произволно описание на умния контакт. Това поле може да бъде оставено празно и служи само за улеснение на потребителя.
4. Активирай отклонение за долна граница – служа за действието, което ще бъде извършено от умния контакт, ако се засече отклонение от някоя от границите зададени от потребителя. Възможните стойности са: „TURN\_OFF“ -> контактът ще се изключи, ако тази граница се достигне, „TURN\_ON“ -> контактът ще се включи, ако тази граница се достигне. Представлява максимума на отклонението.
5. Активирай отклонение за горна граница – служа за действието, което ще бъде извършено от умния контакт, ако се засече отклонение от някоя от границите зададени от потребителя. Възможните стойности са: „TURN\_OFF“ -> контактът ще се изключи, ако тази граница се достигне, „TURN\_ON“ -> контактът ще се включи, ако тази граница се достигне. Представлява минимумът на отклонението.
6. Тип – типът на умния контакт. Стойностите са “Temperature” или “Humidity”. Типът представлява за какво отговаря умният контакт, за температура или влажност. Това е друг филтър, кой контакт ще се включи или изключи в зависимост от границата, която е в отклонение от зададените показатели. Например, ако се надвиши граница от тип температура, следва да бъде приложението действието на всички контакти, които са от тип температура и имат зададено действие за горно граница на отклонение.



Фиг. 15 Страницата за добавяне на граница на отклонение

Тази страница отговаря за задаването на граници на станции. Те отговарят, какво трябва да стане с контактите свързани с тази измервателна станция, ако се засече, че има отклонение от зададените показатели. Съответно необходимите данни са:

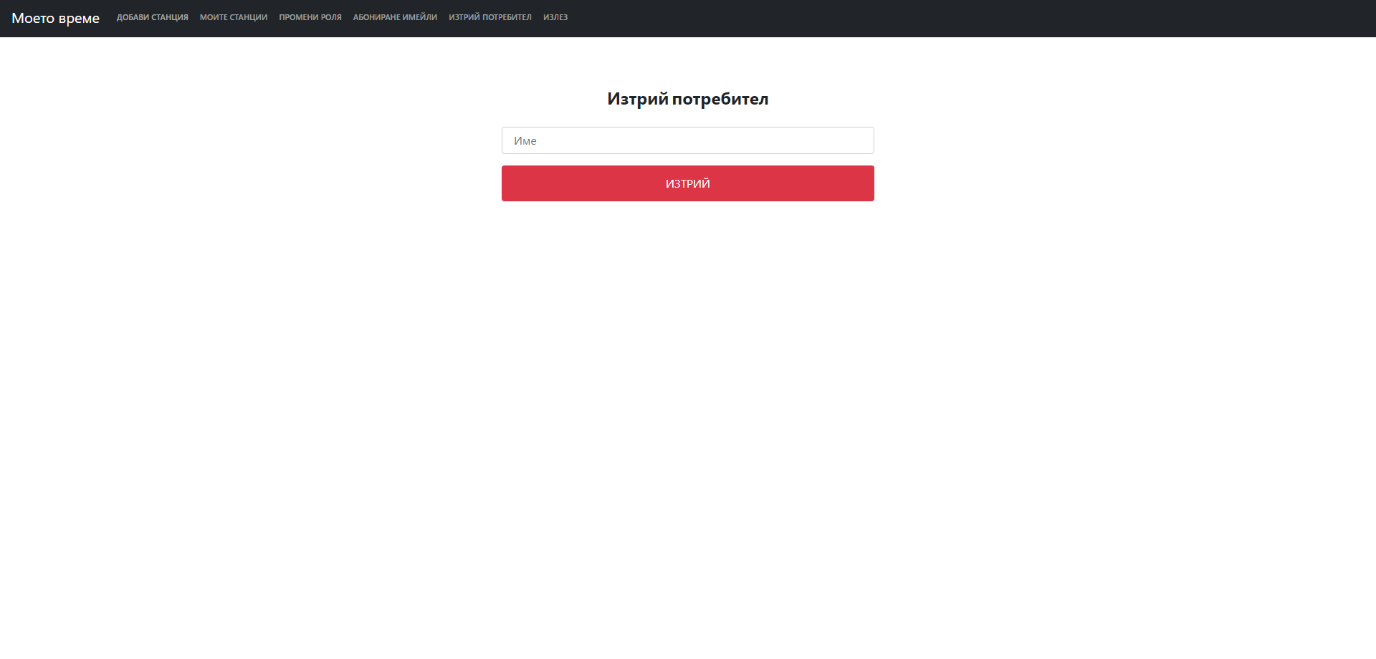
1. Активирай минимална стойност – показва дали трябва да се вземе предвид долната граница на отклонение. Тоест дали нещо трябва да стане, ако се достигне минималната стойност на границата.
2. Минимална стойност – минималната гранична стойност. Взима се предвид само ако е активирана.
3. Активирай максимална стойност - показва дали трябва да се вземе предвид горната граница на отклонение. Тоест дали нещо трябва да стане, ако се достигне максималната стойност на границата.
4. Максимална стойност – максималната гранична стойност. Взима се предвид само ако е активирана.
5. Избери тип – типът на границата на отклонение. Може да бъде температура или влажност.



Фиг. 17 Страницата за смяна на роля на потребител

Тази страница може да се достъпи само от потребители, които има роля модератор или администратор. Служи за промяната на ролята на потребител. Нужната информация е:

1. Име – името на потребител
2. Роли – ролите трябва да бъдат разделени със запетая, като възможните роли са: “user”, “moderator”, “admin”.



Фиг. 18 Страницата за изтриване на потребител

Тази страница може да се достъпва само от потребител с роля администратор. Служи за изтриването на потребител от системата, включително и всичките му измервателни станции, умни контакти и записани измервания. Данните, които са нужни са само потребителското име.

## **2.5. Използвани технологии**

1. Spring framework – Spring е фреймуърк и контейнер за инверсия на контрола за Java платформата. Той може да се използва във всяко Java приложение, но има и допълнения, които са налични само при използването на “Java Enterprise Edition”. Предоставя лесен и гъвкав начин за хлабаво обвързани уеб приложения. Чрез наречената “Model-View-Controller” архитектура, където моделът служи за управление на данните. Управлява логиката и правилата на приложението. Контролерът е клас, който управлява входа от потребителя и го преобразува до команди и след това връща информацията на клиента, като го подава “View”. [20]

Spring MVC е разработен около DispatcherServlet, който изпраща заявки към класовете, които се грижат за управлениеto на приложението, както. Основният контрол се основава на анотациите @Controller и @RequestMapping, предлагайки широк спектър от гъвкави методи на работа. С въвеждането на Spring 3.0 механизмът @Controller също позволява да се създават RESTful уеб сайтове и приложения, чрез анотацията @RestController.

В Spring MVC може да се използва всеки обект като команда, която представя форма. Не е необходимо да се прилага специфичен интерфейс или базов клас. Свързването на данни на Spring е много гъвкаво: например, той разглежда несъответствията на типа като грешки при валидиране, които могат да бъдат прихванати от приложението, а не като системни грешки. [20]

Представянето на изгледите е изключително гъвкаво. Имплементацията на контролера може дори директно да пише по потока на отговора. Предимно „ModelAndView“ инстанцията се състои от име на изгледа, както и от „Map“. Името на изгледа е лесно се конфигурира, през имена т.н. “beans” или файлове със свойства, или чрез собствена имплементация. Спринг е много съвместим с “template engines” чрез “ModelAndView”. [21]

1. Spring Security – Java / Java EE, фреймуърк, който осигурява автентикация и авторизация, и други функции за защита. Проектът е стартиран през 2003 и през 2008 и с търговски лиценз.

В основата си Spring Security представлява купчина от „Servlet“ филтри, които спомагат за автентикацията и авторизацията на уеб приложението. Той също така се интегрира добре със Spring фреймуърка и също така поддържа автентикация чрез Oauth2 и SAML. Също така авто-генерира страници за вход и изход и има защита срещу CSRF. [20]

1. PostgreSQL – е мощна, обектна релационна база с отворен код, която използва и разширява SQL езика в комбинация с много функции, които безопасно съхраняват и мащабират най-сложните натоварвания с данни. Произходът на PostgreSQL датира още от 1986г. като част от проекта POSTGRES в Калифорнийския университет в Бъркли и има повече от 30 години активно развитие.

PostgreSQL има силна репутация заради доказаната си архитектура, надеждност, цялост на данните, стабилен набор от функции, разширяемост и отдаденост на общността с отворен код зад софтуера за постоянно предлагана на иновативни решения. PostgreSQL работи на всички операционни системи и е съвместима с ACID от 2001г. и има добавки като геопространствени бази данни с PostGIS. Не е изненада, че PostgreSQL се е превърнала в релационна база данни с отворен код за избор на много хора и организации.

PostgreSQL предлага много функции, насочени към подпомагане на разработчиците да изграждат приложения, администратори за защита на целостта на данните и изграждане на устойчиви на откази среди и да помогнат с управлението на данните без значение колко е голям или малък наборът от данни. Освен че е безплатен и с отворен код, PostgreSQL е силно разширяващ се. Например, може да се определят собствени типове данни, да се изграждат персонализирани функции, дори и да се пише код от различни езици на програмиране, без да се прекомпилира базата данни!

PostgreSQL се опитва да съответства на стандарта SQL, когато такова съответствие не противоречи на традиционните характеристики или би могло да доведе до лоши архитектурни решения. Поддържат се много от функциите, изисквани от SQL стандарта, макар и понякога с леко различаващ се синтаксис или функция. С течение на времето могат да се очакват по-нататъшни стъпки към съответствие. Към версията на версия 12 през октомври 2019 г. PostgreSQL отговаря на поне 160 от 179 задължителни функции за съответствие със SQL: 2016 Core. Към този текст никоя релационна база данни не отговаря на пълното съответствие с този стандарт. [22]

1. Hibernate – позволява на разработчиците по-лесно да пишат приложения, чиито данни надживяват процеса на кандидатстване. Като рамка на обектно / релационно картографиране (ORM) Hibernate се грижи за постоянството на данните, тъй като се прилага за релационни бази данни (чрез JDBC).

В допълнение към собствения си „основен“ API, Hibernate е също и внедряване на спецификацията на Java Persistence API (JPA). Като такъв, той може лесно да се използва във всяка среда, поддържаща JPA, включително Java SE приложения, Java EE сървъри, контейнери Enterprise OSGi и т.н.

Hibernate позволява развитие на устойчиви класове, следвайки естествени обектно-ориентирани идиоми, включително наследяване, полиморфизъм, асоциация, композиция и рамката на колекциите на Java. Hibernate не изисква интерфейси или базови класове за устойчиви класове и дава възможност на всеки клас или структура на данни да бъдат устойчиви.

Hibernate поддържа “lazy” инициализация, многобройни стратегии за извличане и оптимистично заключване с автоматична версия и печат на времето. Hibernate не изисква специални таблици на база данни или полета и генерира голяма част от SQL по време на инициализация на системата, а не по време на изпълнение.

Hibernate постоянно предлага изключителна производителност спрямо директен JDBC код, както по отношение на производителността на разработчиците, така и по време на изпълнение.

Hibernate е проектиран да работи в клъстер на сървър на приложения и да предоставя високо мащабируема архитектура. Фреймуъркът се мащабира добре във всяка среда: използва се, за да задвижва своя вътрешен интранет, който обслужва стотици потребители или за критични за мисията приложения, които обслужват стотици хиляди.

Hibernate е добре известен със своята отлична стабилност и качество, доказана с приемането и използването на десетки хиляди разработчици на Java. Той е силно конфигурируем. [23]

1. Maven – буквално означава събирач на знания и започва като опит за опростяване на процесите на изграждане в проекта за Джакарта. Програмистите са искали стандартен начин за изграждане на проектите, ясно определение от какво се състои проектът, лесен начин за публикуване на информация за проекта и начин за споделяне на JAR в няколко проекта.

Резултатът е инструмент, който може да се използва за изграждане и управление на всеки проект, базиран на Java.

Основната цел на Maven е да позволи на програмиста да намали усилията за разработка в най-кратки срокове. За да постигне тази цел, Maven се занимава с няколко проблемни области:

Улесняване на процеса на изграждане.

Осигуряване на еднаква система за изграждане.

Предоставяне на качествена информация за проекта.

Насърчаване на по-добри практики за развитие. [24]

1. Tomcat – софтуерът Apache Tomcat е реализация с отворен код на Java Servlet, JavaServer Pages, Java Expression Language и Java WebSocket технологиите.

Apache Tomcat е разработен в отворена среда с участието на много програмисти и е пуснат под лиценза на Apache версия 2. Проектът Apache Tomcat е предназначен да бъде колаборация на най-добрите разработчици от целия свят.

Apache Tomcat захранва многобройни мащабни, критични уеб приложения в най-различни индустрии и организации. [25]

Компоненти:

* 1. Catalina – контейнерът на сървлетите на Tomcat. Catalina внедрява спецификациите на Sun Microsystems за сервлет и JavaServer Pages (JSP). В Tomcat елементът Realm представлява "база данни" с потребителски имена, пароли и роли, присвоени на тези потребители. Различните реализации на Realm позволяват на Catalina да бъде интегрирана в среди, където такава информация за автентификация вече се създава и поддържа и след това използва тази информация за внедряване на Container Managed Security, както е описано в спецификацията на сървлетите.
  2. Coyote - компонент на Connector за Tomcat, който поддържа HTTP 1.1 протокол като уеб сървър. Това позволява на Catalina или JSP, да действа и като обикновен уеб сървър, който обслужва локални файлове като HTTP документи. Coyote слуша за входящи връзки към сървъра на конкретен TCP порт и препраща заявката към Tomcat Engine, за да обработи заявката и да изпрати обратно отговор на искащия клиент. Друг Coyote Connector, Coyote JK, слуша подобно, но вместо това препраща заявките си към друг уеб сървър, например Apache, използвайки JK протокола. Това обикновено предлага по-добри резултати.
  3. Jasper – Джаспър е JSP Engine на Tomcat. Jasper анализира JSP файловете, за да ги компилира в Java код като сървлети (които могат да се обработват от Catalina). По време на изпълнение Jasper открива промени в JSP файловете и ги прекомпилира.

От версия 5 Tomcat използва Jasper 2, което е изпълнение на JSP 2.0 спецификацията на Sun Microsystems. От Jasper до Jasper 2 са добавени важни характеристики:

Обединяване на библиотека с маркери на JSP - Всяко маркиране на маркери в JSP файл се обработва от клас обработчик на маркери. Обектите от клас за обработка на маркери могат да се обединят и използват повторно в целия JSP сървлет.

Фон JSP компилация – Докато се прекомпилира модифициран JSP Java код, по-старата версия все още е достъпна за заявки на сървъра. По-стария сървър на JSP се изтрива, след като новият JSP сървлет приключи с рекомпилирането.

Рекомпилира се JSP при включени промени в страницата - страниците могат да бъдат вмъкнати и включени в JSP по време на изпълнение. JSP не само ще бъде прекомпилиран с промени в JSP файловете, но и с включени промени в страницата.

JDT Java компилатор - Jasper 2 може да използва Java компилатора Eclipse JDT (Java Development Tools) вместо Ant и javac.

1. React JS – е JavaScript библиотека, използвана в уеб разработката за изграждане на интерактивни елементи на уебсайтове. JavaScript (или JS) е скриптов език, използван за създаване и контрол на динамично уеб съдържание. Динамичното уеб съдържание включва неща като анимирани графики, слайдшоу за снимки и интерактивни форми. Всеки път, когато се посети уебсайт, където нещата се движат, обновяват или променят по друг начин на екрана ви, без да се изисква ръчно да презаредите уеб страница, има много добър шанс JS да е езикът, който го прави.

От дефиницията по-горе можете да се види как JavaScript играе решаваща роля в разработването на уебсайтове и уеб приложения. Но има моменти, когато е необходим JavaScript, за изпълняваne на повтарящи се функции - неща като ефекти за анимация или функции за автоматично завършване на лентата за търсене. Повторното писане на тези функции всеки път би довело да много излишно повторение.

JavaScript библиотеките са колекции от предварително написан JavaScript код, който може да се използва за общи задачи на JS, като позволява да се заобиколят ръчно интензивния и ненужен процес на кодиране. Ако има функция за изпълнение на JavaScript, която непрекъснато трябва се пренаписва, вероятно има JS библиотека, която да върши тази дейност.

Има много различни JS библиотеки и React JS е една от тях, но поради лесната си употреба, бързата разработка на код, стабилността и функционалността прави React JS една от топ трите най-използвани технологии. [26]

1. Bootstrap – е мощен фреймуърк, използван за създаване на модерни уебсайтове и уеб приложения. Той е с отворен код и безплатен за използване, но разполага с многобройни HTML и CSS шаблони за елементи на интерфейса на интерфейса, като бутони и форми. Bootstrap също поддържа разширения за JavaScript. Също така вътрешно използва Jquery, за динамичното променяне на HTML и CSS шаблоните. [27]
2. Fetch API – предоставя интерфейс за извличане на данни от интернет ресурси. То е сходно със по старите технологии за извършване на заявки по мрежата, но предоставя улеснена и по-надеждна функционалност.

Fetch предоставя обща дефиниция на обекти за запитване и отговор (и други неща, свързани с мрежови заявки).

Той също така дава дефиниция за свързани понятия като CORS и семантиката на заглавието на HTTP начало, замествайки техните отделни дефиниции другаде.

За отправяне на заявка и извличане на ресурс, използва метода WindowOrWorkerGlobalScope.fetch(). Той се реализира в множество интерфейси, по-специално Window и WorkerGlobalScope. Това го прави достъпно в почти всеки контекст, в който трябва да се извличат ресурси.

Методът fetch() взема един задължителен аргумент - пътят към ресурса, който искате да извлечете. Той връща обещание, което се решава на Отговорът на тази молба, независимо дали е успешно или не. Може също по желание да се прехвърли в обекта с опции на init като втори аргумент.

След като бъде получен отговор, съществуват редица методи за определяне какво е съдържанието на тялото и как трябва да се борави.

Може да се създава заявка и отговор директно, като се използват конструкторите Request() и Response(), но е рядкост да се прави това директно. Вместо това те са по-често се създаване като резултати от други действия на API. [28]

1. Сензор за температура, влажност и налягане BME 280 – BME280 е сензор за влажност, специално разработен за мобилни приложения и преносими устройства, където размерът и ниската консумация на енергия са ключови параметри на дизайна. Устройството съчетава сензори с висока линейност и висока точност и е напълно осъществимо за ниска консумация на ток, дългосрочна стабилност и висока устойчивост на ЕМС. Сензорът за влажност предлага изключително бързо време за реакция и следователно поддържа изисквания за производителност за нововъзникващи приложения, като осъзнаване на контекста и висока точност в широк температурен диапазон. [29]
2. Библиотека за работа с сензор Bosch BME 280 - Интерфейс на цифров сензорен модул Bosch BME280 (способен да отчита температура, влажност и налягане) в Python 2 или 3, използвайки I2C на Raspberry Pi. [30]
3. Python Flask - Flask е микро уеб фреймуърк, написан на Python. Той е класифициран като микрорамка, защото не изисква специални инструменти или библиотеки. Той няма абстракционен слой на база данни, валидиране на формуляри или други компоненти, където вече съществуващите библиотеки на трети страни предоставят общи функции. Въпреки това, Flask поддържа разширения, които могат да добавят функции на приложението, сякаш са внедрени в самия Flask. Съществуват разширения за обектно-релационни карти, валидиране на формуляри, обработка на качване, различни отворени технологии за удостоверяване и няколко общи инструменти, свързани с рамката. [31]
4. Shelly Plug s – Умен контакт, който може автоматично да наблюдава и контролира осветлението, отоплението и всеки друг свързан електрически уред у дома, независимо къде се намирате. Когато електрически уред или лампа е включен в Shelly Plug, можете да го включвате и изключвате чрез приложението Shelly Cloud по всяко време и навсякъде. Поддържа мощност до 2500W. Поддържа Wi-Fi и Bluetooth връзка. [32]

# **3. Особености на програмната реализация**

За реализацията на системата са използваните технологии посочени във втора точка. В основата си цялата платформа е разделена на четири части: клиент, сървър, Raspberry PI измервателна станция със сензор за температура и влажност и Shell Plug S умен контакт.

## **3.1. Какво представлява клиентът?**

Клиентът представлява JavaScript + HTML 5 приложение, което използва библиотеките React, Bootstrap и Fetch API. Неговата идея е да се достъпва от потребителите, поради което той трябва да е удобен за употреба от потребителите, да има красив външен вид и да бъде интуитивен за употреба. Да не се налага да четеш дълги упътвания как се работи с него, а да може само при влизането и от бутоните и текста да е достатъчно потребителят да се ориентира и да работи с приложението. Включва общи 20 HTML страници. Потребителят няма достъп до абсолютно всички тъй като някои от тях са предназначени единствено за участници с по-големи привилегии като модератори и администраторите. Всяка една от страниците е свързана с всяка една друга, като така няма нужда от достъпването през URL-и.

Друга важна характеристика на клиента е да събира и валидира данни от потребителя и да ги запраща след успешна начална валидация към сървъра, който може и да работи на друг събдомейн. Събират се данни като, имейли, имена, които се въвеждат от потребителите, при регистрация, при вход, при добавяне на измервателни станции и т.н. Следователно те биват валидирани, ако данните не са коректни, например, липсва име при регистрация, или мрежови адрес при добавяне на умен контакт, се показват допълнителни грешки, които показват на потребителя какво е пропуснато или какво е въведено неправилно, например, заето потребителско име. Част от тези данни се валидират от самия клиент, тъй като това е възможно и забързва процеса на валидация, но това не винаги е възможно. Пример за това е заето потребителско име. Такива валидации се правят след като данните са пратени към сървъра, където са валидират повторно, тъй като част от валидациите винаги могат да бъдат пренебрегнати от „лоши“ потребители и така да направят необратими последствия върху реалните данни. След валидацията на данните на сървъра се връща или валиден отговор, с който се означава, че не са изникнали никакви проблеми при валидацията и запазването на данните, или се връща отговор грешка, който означава, че е имало проблеми в някоя от валидациите. След такава грешка отново се визуализира проблемът на потребителя и той коригира своите данни и следователно продължава използването на приложението. То бива достъпвано през модерен интернет браузър на определен адрес.

Приложението доставя удобен дизайн за работа. Също така включва тъй наречения “responsive design”, който представлява визуализирането на текста, както и изображенията, и форматът на полетата винаги по удобен начин на потребителя независимо от размера на екрана или устройството, което потребителят ползва.

Също така благодарение на React библиотеката HTML съдържанието е динамично и може да бъде подменяно по всяко време без да се изисква презареждането на страницата. Това не само забързва работата на приложението, но и прави работата на потребителя с приложението много по-приятна, тъй като всичко и динамично и бързо.

Клиентът изпраща заявки до сървъра по няколко причини. Първата е за вземане на данни. Това е най-често използваната заявка т.н. “GET” заявка, и нейната цел е единствено за вземане на данни от сървъра. Следващата доста използвана заявка е т.н. “POST” заявка, тя се използва за изпращане на данни. Най-често форматът, в който се изпращат данни е JSON (JavaScript object notation), тъй като лесен за четене и малък по размер. Друг начин е чрез т.н. “form-urlencoded”, при който данните се изпращат като поредица от ключ-стойност обекти в тялото на HTTP заявката. Друг вид заявки, които клиентът изпраща са PUT/PATCH използвани за подмяна на целия ресурс или част от него. Други видове DELETE използван за изтриване на ресурс. Също използван е OPTIONS, за да се удостовери дали сървърът е безопасен да се извиква.

За автентикация се използват тъй наречените JWT tokens. Това е кодиран JSON обект със секретен ключ, който се пази в самото приложение и съдържа достатъчна информация за потребителят. В token-а се държи информация за ролите на потребителя, като така уеб клиентът разбира, какви страници може да покаже. Този token се генерира, когато потребителят влиза в приложението с потребителско име и парола. Той се запазва в т.н. “localStorage” на уеб браузъра и се изпраща към сървъра, когато има нужда, като така се гарантира, че потребителят е автентикиран и има право да праща определената заявка. Token-ът се валидира на от главния сървър. Валидността му е 24 часа, след което потребителят трябва отново да влезе в профила си.

Клиентът се достъпва няколко крайни точки, изброени долу: <Route exact path="/" component={Landing} /> - На “/” път може да се достъпи началната страница на приложението.

<Route exact path="/login" component={UserLogin} /> - На “/login” може да се достъпи страницата за вход на потребител.

<Route exact path="/register" component={UserRegister} /> - При достъпването на “/register” се достъпва страницата за регистрация на потребител.

<Route exact path="/add-raspberry" component={AddRaspberry} /> - При достъпването на “/add-raspberry” се зарежда страницата за добавяне на нова измервателна станция.

<Route exact path="/my-raspberries" component={MyRaspberries} /> - При достъпването на “/my-raspberries” пътя се отворя страницата за всички измервателни станции, които потребителят притежава.

<Route path="/raspberry/chart/" component={RaspberryCharts} /> - При достъпването на “/raspberry/chart/<id>” се зарежда графиката за средната температура и влажност за определена измервателна станция.

<Route path="/user/activate/" component={UserActivation} /> - При достъпването на “/user/activate/<username>” се активира акаунтът на потребителя. Линкът е изпратен на имейла на потребителя като тази страница се отваря само оттам. Тя веднага пренасочва към началото, като преди да пренасочи към началото тя изпраща заявка за активация на акаунта.

Route path="/anomalies/raspberry/" component={RaspberryAnomalies} /> - При достъпването на “/anomalies/raspberry/<id>” се отваря страницата с границите на отклонение за дадената измервателната станция.

<Route path="/raspberry/power-plugs/" component={MyPowerPlugs} />- При отварянето на “/raspberry/power-plugs/<id>” се отварят умните контакти за дадената измервателна станция.

<Route path="/power-plug/add/" component={AddPowerPlug} /> - При отварянето на “/power-plug/add/<id>” се отваря страницата за добавяне на умни контакти.

<Route path="/raspberry/edit/" component={EditRaspberry} /> - При достъпването на “/raspberry/edit/<id>” се отваря страницата за редакция на измервателна станция.

<Route path="/power-plug/edit/" component={EditPowerPlug} /> - При достъпването на “/power-plug/edit/<id>” се отваря страницата за редакция на измервателна станция.

<Route path="/power-plug/delete/" component={DeletePowerPlug} /> - При отварянето на “/power-plug/delete/<id>” се отваря страницата за изтриване на умен контакт.

<Route path="/anomaly/edit/" component={EditAnomalyDetectionRule} /> - При отварянето на “/anomaly/edit/<id>” се отваря страницата за редактиране на граница на отклонение.

<Route path="/anomaly/delete/" component={DeleteAnomalyDetectionRule} /> - При отварянето на “/anomaly/delete/<id>” се отваря страницата за изтриване на граница на отклонение.

<Route exact path="/admin/update-roles/" component={UpdateUserRoles} /> - При отварянето на “/admin/update-roles” се отваря страницата за промяната на роли на потребител. За отварянето на тази страница се изисква потребителят да има роля модератор или администратор.

<Route exact path="/admin/delete-user/" component={DeleteUser} /> - При отварянето на “/admin/delete-user/” се отваря страницата за изтриване на потребител по име. За отварянето на тази страница се изисква роля администратор в системата.

<Route exact path="/subscribe/emails/" component={SubscribedEmails} /> - При отварянето на “/subscribe/emails” се отваря страницата с изпратените имейли на все още нерегистрирани потребители. За отварянето на тази роля се изисква роля администратор или модератор.

## **3.2. Какво представлява сървърът?**

Сървърът представлява приложение написано на езика Java и използва по-горе посочените технологии за разработката (Spring, Tomcat, Maven, Hibernate, PostgreSQL). Той е много по-голямо и по-сложно приложение от клиента и неговата идея е както да пази данните на потребителя, както и данните, с които потребителят работи, също комуникира с измервателните станциите, като им праща заявки за включване и изключване, а те изпращат измервания към него. Освен това главният сървър работи и с умните контакти, като ги контролира кога да се включват и изключват. Той работи пряко с клиента, измервателните станции и контактите, като или предава данни или приема. Съответно при предаването на данни трябва потребителят, който се опитва да се снабди с тези данни да има правото да го направи. Например обикновен или нерегистриран потребител не трябва да може изпраща данни, за които само модераторът би могъл да има достъп. Друга задача на сървъра е да валидира данните изпратени от клиента, тъй като някой валидации не е възможно да бъдат направени от страната на клиента или да бъдат изпратени, от други клиенти като, например Postman, който представлява клиент, който може да изпраща всякакъв вид заявки и може да бъде използван за злонамерено изпращане или вземане на данни. Също така сървърът се грижи за периодичното зачистване на стари данни. Това са измервания, които са направи преди 7 или повече дни. Таза функционалност се включва веднъж дневно. Също така сървърът изпраща известия на потребителите по формата на имейли, когато се спира или включва умен контакт или с други думи, когато измерваните показатели излизат извън зададените граници.

Сървърното приложение е разделено на 4 основни слоя. Първият слой, който се достига, когато е пратена заявка е т.н. “Filter”. В този тип класове се извършват основни валидации върху изпратената от уеб клиентът заявка. Ако има някаква нередност в заявката, например, изисква се в заявка да има „Authorization“ header, който носи гореспоменатия JWT token, а той липсва тогава сървърът ще прекъсне по-нататъшното обработване на заявката и в случая ще върне статус код на отговора “401 Unauthorized”, което означава, че пратената няма автентикация. Пример за такъв Filter e:

@Component  
public class JwtFilter extends OncePerRequestFilter {  
  
 public static final Logger *LOGGER* = LoggerFactory.*getLogger*(JwtFilter.class);  
  
 private final UserDetailsService userService;  
  
 @Autowired  
 public JwtFilter(UserDetailsService userService) {  
 this.userService = userService;  
 }  
  
 @Override  
 protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)  
 throws ServletException, IOException {  
 String authorizationHeader = request.getHeader(HttpHeaders.*AUTHORIZATION*);  
 if (!StringUtils.*startsWithIgnoreCase*(authorizationHeader, "Bearer")) {  
 filterChain.doFilter(request, response);  
 return;  
 }  
 String jwt = authorizationHeader.substring(7);  
 Optional<String> username = JwtUtil.*extractUsername*(jwt);  
 if (username.isPresent() && SecurityContextHolder.*getContext*()  
 .getAuthentication() == null) {  
 UserDetails userDetails;  
 try {  
 userDetails = userService.loadUserByUsername(username.get());  
 } catch (UsernameNotFoundException e) {  
 *LOGGER*.error(e.getMessage(), e);  
 filterChain.doFilter(request, response);  
 return;  
 }  
 if (JwtUtil.*validateToken*(jwt, userDetails)) {  
 UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(userDetails,  
 null,  
 userDetails.getAuthorities());  
 usernamePasswordAuthenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
 SecurityContextHolder.*getContext*()  
 .setAuthentication(usernamePasswordAuthenticationToken);  
 }  
 }  
 filterChain.doFilter(request, response);  
 }  
}

Следващият слой е т.н. Controller слой и тези класове отговарят за транслирането на body-то изпратената заявка към Java класове. Върху тези данни се извършват базови валидации, като дали е подадено име на потребител и ако не е следва да се върне друг служебен код „400 Bad request“. Ако заявката е валидна, новите класове се пращат към следващия слой от приложението.

Този слой е т.н. Service слой и в него се правят подробни валидации на данните, изпращат се имейли, стартират се и се спират станции и умни контакти. Това е слоят с най-много бизнес логика в системата и следва да бъде най-добре тестван. Също в този слой класовете изпратени от Controller слоя се транслират до най-базовия клас, който представлява таблица от базата данни описана като Java клас. Пример за такъв клас е ролята:

@Entity  
@Table(name = "roles")  
public class Role extends IdEntity {  
  
 @Column  
 @Enumerated(EnumType.*STRING*)  
 private RoleType roleType;  
  
 @LazyCollection(LazyCollectionOption.*FALSE*)  
 @Fetch(value = FetchMode.*SUBSELECT*)  
 @ManyToMany  
 @JoinTable(name = "roles\_users", joinColumns = @JoinColumn(name = "role\_id"), inverseJoinColumns = @JoinColumn(name = "user\_id"))  
 private List<User> users = Collections.*emptyList*();  
  
 public enum RoleType {  
 *ADMIN*, *MODERATOR*, *USER*;  
  
 @Override  
 public String toString() {  
 return name().toLowerCase(Locale.*ROOT*);  
 }  
 }  
  
}

Последният слой е т.н. Repository слой, в който данните се записват в базата с помощта на библиотеката Hibernate. Отваря се връзка с базата, стартира се транзакция и се съхранява нужната информация, след което транзакцията се commit-ва и се затваря връзката с базата данни. И отговорът се връща нагоре по веригата докато стига Controller слоя, където данните отново се превръщат в JSON текст и се пращат като отговор на изпратената заявка от уеб клиента. Пример за Repository. Това е базов клас на всички хранилища (показана е само малка част):

public abstract class BaseRepositoryImpl<E extends IdEntity> implements BaseRepository<E> {  
  
 protected final SessionFactory sessionFactory;  
  
 protected BaseRepositoryImpl(SessionFactory sessionFactory) {  
 this.sessionFactory = sessionFactory;  
 }  
  
 @Override  
 public void save(E obj) {  
 executeInTransaction(session -> {  
 session.save(obj);  
 return null;  
 });  
 }

}

Сървърът предоставя така наречените “крайни точки”, които представлява местата, от които може да се вземе или прати информация. Те се намират в Controller слоя описан по-горе. Данните се изпращат и приемат от сървъра като JSON форматиран текст.

При изпращането само на текстове данни от клиента към сървъра се извършват следните стъпки за тяхното запазване. Първоначално клиентът извършва определена валидация и ги праща към сървъра, след това сървърът ги прима и ги транслира до класове, като предварително извършва някои основни валидации. Например, при добавянето на измервателна станция се изисква името ѝ да бъде подадено или с други думи да не е null. За такъв вид валидации си използват анотации, които се поставят на член-променливата, в случая “@NotNull”, които се четат от Spring фреймуърка, който използва класове от JavaEE, за да извърши съответната валидация. В случай, че валидацията е неуспешна, в случая стойността на полето е null, автоматично Spring връща отговор със код 400 (“Bad Request“), което означава, че клиентът не е подал правилни данни. Пример за такова поле в кода:

@NotNull

private String name;

Също така са използвани и други анотации, които подсигуряват коректността на данните като @Min, @Max за минимална или максимална стойност или съответно, някои създадени ръчно, с цел персонализирана валидация. Пример за максимална или минимална стойност са измерената температура и влажност, която трябва да е в диапазон от [-5000; 5000].

@Min(-5000)  
@Max(5000)  
private double temperature;

Следват валидации дали името потребител е свободно, които се извършват като се провери дали съществува такъв потребител в базата данни:

private void validateUsernameIsFree(UserServiceModel userServiceModel) {  
 Optional<User> user = userRepository.findByUsername(userServiceModel.getUsername());  
 if (user.isPresent()) {  
 throw new IllegalArgumentException(String.*format*("Username \"%s\" is taken!", userServiceModel.getUsername()));  
 }  
}

Такива проверки има за всички полета (колони в базата), които изискват уникалност на данните. Проверки се правят в т.н. “Service” слой.

Сесията с базата данни се стартира и commit-ва всеки път, когато се налага работа с нея. Тоест дори и при една заявка от клиента е възможно да се отворят и затворят повече от една сесия с базата данни, затова трябва всички необходими проверки да бъдат извършена преди да започне писането по базата данни, тъй като ако заявка пропадне по време на писане по базата и вече имаме нещо друго запазено в базата, което зависи на текущия пропаднал запис може да се наруши целостта на данните.

За да комуникира сървърът с измервателните станции се използва WebClient, който идва с Spring framework, който е много удобен клиент за работата с HTTP заявки. Надежден е тъй има и вградена устойчивост при пропадане на заявка. За да се стартира измервателна станция се изпраща заявка до нея:

public void startRaspberry(String raspberryRoute) {  
 webClient.post()  
 .uri(getRaspberryFullRoute(raspberryRoute, Constants.*RASPBERRY\_ACTION\_START\_PATH*))  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .body(BodyInserters.*fromValue*(new StartRaspberryRequest(environmentGetter.getLocalHostname())))  
 .retrieve()  
 .bodyToMono(String.class)  
 .block();  
}

Изпраща се заявка на адресът на измервателната станция и също така в body-то на заявката се добавя адресът на сървъра, който се взима е средата на приложението, тоест трябва да е предварително зададен. След като се изпрати заявката, измервателната станция започва да измерва температурата и влажността и ги изпраща на сървъра. За спиране на измервателната станция се праща отново POST заявка и измервателната станция спира да отчита и праща измерванията:

public void stopRaspberry(String raspberryRoute) {  
 webClient.post()  
 .uri(getRaspberryFullRoute(raspberryRoute, Constants.*RASPBERRY\_ACTION\_STOP\_PATH*))  
 .retrieve()  
 .bodyToMono(String.class)  
 .block();  
}

За комуникацията с умните контакти отново е използван WebClient-ът от Spring framework-а. За стартирането и спирането се използва един и същ метод като се сменя единствено параметъра „turn“, като неговите стойности са “on” или “off”:

public void executeAction(String powerPlugRoute, String action) {  
 try {  
 webClient.get()  
 .uri("http://" + powerPlugRoute + "/relay/0?turn=" + action)  
 .retrieve()  
 .bodyToMono(String.class)  
 .block();  
 } catch (Exception e) {  
 System.*err*.println(e.getMessage());  
 }  
}

Сървърът предоставя следните “крайни точки” за комуникация, като за целта се използва Spring “@RestController”. По-долу са показани някой от най-ключовите крайни точки:

1. Потребители – всеки от пътищата започва с “/users”
   1. Регистрация на потребител:

@PostMapping(value = "/register")  
public ResponseEntity<UserRegistrationResponseModel> register(@RequestBody UserRegistrationBindingModel userRegistrationBindingModel) {  
 if (!Objects.*equals*(userRegistrationBindingModel.getConfirmPassword(), userRegistrationBindingModel.getPassword())) {  
 return ResponseEntity.*badRequest*()  
 .build();  
 }  
 UserServiceModel userServiceModel = modelMapper.map(userRegistrationBindingModel, UserServiceModel.class);  
 userService.save(userServiceModel);  
 return ResponseEntity.*ok*(modelMapper.map(userRegistrationBindingModel, UserRegistrationResponseModel.class));  
}

* 1. Вземане на измервателна станции на потребител

@GetMapping("/raspberries")  
public ResponseEntity<List<RaspberryResponseModel>> findUserRaspberries() {  
 Authentication authentication = SecurityContextHolder.*getContext*()  
 .getAuthentication();  
 if (!(authentication instanceof AnonymousAuthenticationToken)) {  
 String currentUserName = authentication.getName();  
 return ResponseEntity.*ok*(userService.findUserRaspberries(currentUserName));  
 }  
 return ResponseEntity.*status*(403)  
 .build();  
}

* 1. Активиране на профил

@PostMapping(value = "/activate/{username}")  
public ResponseEntity<Void> activate(@PathVariable String username) {  
 userService.activate(username);  
 return ResponseEntity.*ok*()  
 .build();  
}

* 1. Автентикация на потребител

@PostMapping(value = "/authenticate")  
public ResponseEntity<JwtTokenResponse> authenticate(@RequestBody AuthenticationRequest authenticationRequest) {  
 UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(authenticationRequest.getUsername(),  
 authenticationRequest.getPassword());  
 User userServiceModel = (User) userService.loadUserByUsername(authenticationRequest.getUsername());  
 authenticationManager.authenticate(usernamePasswordAuthenticationToken);  
 String jwt = JwtUtil.*generateToken*(userServiceModel, userServiceModel.getRoles()  
 .stream()  
 .map(Role::getRoleType)  
 .map(Role.RoleType::toString)  
 .collect(Collectors.*toList*()));  
 return ResponseEntity.*ok*(new JwtTokenResponse(jwt));  
}

1. Измервателни станции – всеки от пътищата започва с “/raspberry”:
   1. Вземане на всички

@GetMapping  
public ResponseEntity<List<RaspberryBindingModel>> getAll() {  
 List<RaspberryServiceModel> raspberryServiceModels = raspberryService.findAll();  
 List<RaspberryBindingModel> raspberryBindingModels = raspberryServiceModels.stream()  
 .map(raspberryServiceModel -> modelMapper.map(raspberryServiceModel,  
 RaspberryBindingModel.class))  
 .collect(Collectors.*toList*());  
 return ResponseEntity.*ok*(raspberryBindingModels);  
}

* 1. Вземане по идентификационен номер

@GetMapping("/{raspberryId}")  
public ResponseEntity<RaspberryBindingModel> getById(@PathVariable String raspberryId) {  
 RaspberryServiceModel raspberryServiceModel = raspberryService.findById(raspberryId);  
 return ResponseEntity.*ok*(modelMapper.map(raspberryServiceModel, RaspberryBindingModel.class));  
}

* 1. Добавяне на измервателна станция

@PostMapping  
public ResponseEntity<RaspberryBindingModel> add(@RequestBody RaspberryBindingModel raspberryBindingModel) {  
 RaspberryServiceModel raspberryServiceModel = modelMapper.map(raspberryBindingModel, RaspberryServiceModel.class);  
 attachUserToRaspberry(raspberryServiceModel);  
 raspberryService.save(raspberryServiceModel);  
 return ResponseEntity.*ok*(raspberryBindingModel);  
}

* 1. Редактиране на измервателна станция

@PutMapping("{raspberryId}")  
public ResponseEntity<RaspberryBindingModel> update(@RequestBody RaspberryBindingModel raspberryBindingModel,  
 @PathVariable String raspberryId) {  
 RaspberryServiceModel raspberryServiceModel = modelMapper.map(raspberryBindingModel, RaspberryServiceModel.class);  
 attachUserToRaspberry(raspberryServiceModel);  
 raspberryServiceModel.setId(raspberryId);  
 raspberryService.update(raspberryServiceModel);  
 return ResponseEntity.*ok*(raspberryBindingModel);  
}

* 1. Стартиране на измервателна станция

@PostMapping(value = "/start")  
public ResponseEntity<Void> startRaspberry(@RequestBody RaspberryBindingModel requestBindingModel) {  
 raspberryService.start(requestBindingModel.getId());  
 return ResponseEntity.*ok*()  
 .build();  
}

* 1. Спиране на измервателна станция

@PostMapping(value = "/stop")  
public ResponseEntity<Void> stopRaspberry(@RequestBody RaspberryBindingModel requestBindingModel) {  
 raspberryService.stop(requestBindingModel.getId());  
 return ResponseEntity.*ok*()  
 .build();  
}

1. Умни контакти – всеки от пътищата започва с “/power-plug”
   1. Добавяне на умен контакт

@PostMapping  
public ResponseEntity<PowerPlugResponseModel> add(@RequestBody @Valid PowerPlugBindingModel powerPlugBindingModel) {  
 PowerPlugServiceModel powerPlugServiceModel = modelMapper.map(powerPlugBindingModel, PowerPlugServiceModel.class);  
 powerPlugService.save(powerPlugServiceModel);  
 PowerPlugResponseModel powerPlugResponseModel = modelMapper.map(powerPlugBindingModel, PowerPlugResponseModel.class);  
 return ResponseEntity.*ok*(powerPlugResponseModel);  
}

* 1. Вземане по идентификационен номер

@GetMapping("/{id}")  
public ResponseEntity<PowerPlugResponseModel> get(@PathVariable String id) {  
 PowerPlugServiceModel powerPlugServiceModel = powerPlugService.findById(id);  
 PowerPlugResponseModel powerPlugResponseModel = modelMapper.map(powerPlugServiceModel, PowerPlugResponseModel.class);  
 return ResponseEntity.*ok*(powerPlugResponseModel);  
}

* 1. Изтриване по идентификационен номер

@PutMapping("/{id}")  
public ResponseEntity<PowerPlugResponseModel> edit(@PathVariable("id") String powerPlugId,  
 @RequestBody @Valid PowerPlugBindingModel powerPlugBindingModel) {  
 PowerPlugServiceModel powerPlugServiceModel = modelMapper.map(powerPlugBindingModel, PowerPlugServiceModel.class);  
 powerPlugServiceModel.setId(powerPlugId);  
 powerPlugServiceModel.setActionOnAboveAnomaly(Action.*from*(powerPlugBindingModel.getActionOnAboveAnomaly()));  
 powerPlugServiceModel.setActionOnBelowAnomaly(Action.*from*(powerPlugBindingModel.getActionOnBelowAnomaly()));  
 powerPlugService.update(powerPlugServiceModel);  
 PowerPlugResponseModel powerPlugResponseModel = modelMapper.map(powerPlugBindingModel, PowerPlugResponseModel.class);  
 return ResponseEntity.*ok*(powerPlugResponseModel);  
}

* 1. Взимане на всички контакти по идентификационен номер на измервателна станция

@GetMapping  
public ResponseEntity<List<PowerPlugResponseModel>> getByRaspberryId(@RequestParam String raspberryId) {  
 List<PowerPlugServiceModel> powerPlugServiceModels = powerPlugService.findAllByRaspberryId(raspberryId);  
 return ResponseEntity.*ok*(powerPlugServiceModels.stream()  
 .map(powerPlugServiceModel -> modelMapper.map(powerPlugServiceModel,  
 PowerPlugResponseModel.class))  
 .collect(Collectors.*toList*()));  
}

1. Измервания – всеки от пътищата започва с “/measurement”
   1. Добавяне на измерване

@PostMapping  
public ResponseEntity<MeasurementBindingModel> create(@RequestBody MeasurementBindingModel measurementBindingModel) {  
 MeasurementServiceModel measurementServiceModel = MeasurementServiceModel.*from*(measurementBindingModel);  
 measurementService.save(measurementServiceModel);  
 return ResponseEntity.*ok*(measurementBindingModel);  
}

* 1. Взимане на измервания за час (средноаритметично)

@GetMapping(value = "/charts/hour")  
public ResponseEntity<List<DateWithMeasurementsResponseModel>>  
 getMeasurementFor24Hours(AveragedMeasurementBidingModel averagedMeasurementBidingModel) {  
 Map<LocalDateTime, MeasurementResponseModel> averagedMeasurements = measurementCalculator.calculateMeasurementsBetween(averagedMeasurementBidingModel.getStartPeriod(),  
 averagedMeasurementBidingModel.getEndPeriod(),  
 averagedMeasurementBidingModel.getRaspberryId());  
 List<DateWithMeasurementsResponseModel> dateWithMeasurements = new ArrayList<>();  
 averagedMeasurements.forEach((key, value) -> dateWithMeasurements.add(new DateWithMeasurementsResponseModel(key, value)));  
 return ResponseEntity.*ok*(dateWithMeasurements);  
}

1. Граници на отклонение – всеки от пътищата започва с “/anomaly”
   1. Взимане на граници

@GetMapping  
public ResponseEntity<List<AnomalyDetectionRuleResponseModel>> getAll() {  
 List<AnomalyDetectionRuleServiceModel> anomalyDetectionRuleServiceModels = anomalyDetectionRuleService.findAll();  
 return ResponseEntity.*ok*(anomalyDetectionRuleServiceModels.stream()  
 .map(AnomalyDetectionRuleServiceModel::toResponseModel)  
 .collect(Collectors.*toList*()));  
}

* 1. Добавяне на нова граница

@PostMapping  
public ResponseEntity<AnomalyDetectionRuleResponseModel>  
 add(@RequestBody AnomalyDetectionRuleBindingModel anomalyDetectionRuleBindingModel) {  
 AnomalyDetectionRuleType anomalyDetectionRuleType = AnomalyDetectionRuleType.*from*(anomalyDetectionRuleBindingModel.getType());  
 AnomalyDetectionRuleServiceModel anomalyDetectionRuleServiceModel = anomalyDetectionRuleType.getRuleBindingParser(anomalyDetectionRuleBindingModel)  
 .parse();  
 anomalyDetectionRuleService.save(anomalyDetectionRuleServiceModel);  
 AnomalyDetectionRuleResponseModel anomalyDetectionRuleResponseModel = anomalyDetectionRuleServiceModel.toResponseModel();  
 return ResponseEntity.*ok*(anomalyDetectionRuleResponseModel);  
}

* 1. Редактиране на граница

@PutMapping("/{id}")  
public ResponseEntity<AnomalyDetectionRuleResponseModel>  
 edit(@PathVariable String id, @RequestBody AnomalyDetectionRuleBindingModel anomalyDetectionRuleBindingModel) {  
 AnomalyDetectionRuleType anomalyDetectionRuleType = AnomalyDetectionRuleType.*from*(anomalyDetectionRuleBindingModel.getType());  
 AnomalyDetectionRuleServiceModel anomalyDetectionRuleServiceModel = anomalyDetectionRuleType.getRuleBindingParser(anomalyDetectionRuleBindingModel)  
 .parse();  
 anomalyDetectionRuleServiceModel.setId(id);  
 anomalyDetectionRuleService.update(anomalyDetectionRuleServiceModel);  
 AnomalyDetectionRuleResponseModel anomalyDetectionRuleResponseModel = anomalyDetectionRuleServiceModel.toResponseModel();  
 return ResponseEntity.*ok*(anomalyDetectionRuleResponseModel);  
}

* 1. Вземане по идентификационен номер

@GetMapping("/{id}")  
public ResponseEntity<AnomalyDetectionRuleResponseModel> get(@PathVariable String id) {  
 AnomalyDetectionRuleServiceModel anomalyDetectionRuleServiceModel = anomalyDetectionRuleService.findById(id);  
 return ResponseEntity.*ok*(anomalyDetectionRuleServiceModel.toResponseModel());  
}

* 1. Изтриване по идентификационен номер

@DeleteMapping("/{id}")  
public ResponseEntity<Void> delete(@PathVariable String id) {  
 anomalyDetectionRuleService.delete(id);  
 return ResponseEntity.*ok*().build();  
}

* 1. Вземане на всички по идентификационен номер на измервателна станция

@GetMapping("/raspberry/{raspberryId}")  
public ResponseEntity<List<AnomalyDetectionRuleResponseModel>> getAnomaliesByRaspberry(@PathVariable String raspberryId) {  
 List<AnomalyDetectionRuleServiceModel> anomalyDetectionRuleServiceModels = anomalyDetectionRuleService.findByRaspberryId(raspberryId);  
 return ResponseEntity.*ok*(anomalyDetectionRuleServiceModels.stream()  
 .map(AnomalyDetectionRuleServiceModel::toResponseModel)  
 .collect(Collectors.*toList*()));  
  
}

1. Абонирани имейли – всеки от пътищата започва с “/subscribe”
   1. Добавяне на измервателна станция

@PostMapping  
public ResponseEntity<SubscribedEmailResponseModel> add(@RequestBody @Valid SubscribedEmailBindingModel subscribedEmailBindingModel) {  
 SubscribedEmailServiceModel subscribedEmailServiceModel = modelMapper.map(subscribedEmailBindingModel,  
 SubscribedEmailServiceModel.class);  
 subscribedEmailService.save(subscribedEmailServiceModel);  
 return ResponseEntity.*ok*(modelMapper.map(subscribedEmailServiceModel, SubscribedEmailResponseModel.class));  
}

* 1. Вземане на всички измервателни станции

@GetMapping  
public ResponseEntity<List<SubscribedEmailResponseModel>> getAll() {  
 List<SubscribedEmailServiceModel> subscribedEmailServiceModels = subscribedEmailService.findAll();  
 List<SubscribedEmailResponseModel> response = subscribedEmailServiceModels.stream()  
 .map(subscribedEmailServiceModel -> modelMapper.map(subscribedEmailServiceModel,  
 SubscribedEmailResponseModel.class))  
 .collect(Collectors.*toList*());  
 return ResponseEntity.*ok*(response);  
}

За реализацията на таблиците в базата данни е използван фреймуъркът Hibernate, който улеснява поддържането на таблиците, като ги генерира по време на работата на приложението и също така служи за промяна по тях, ако са променени анотациите.

Пример за маркирането на поле като колона от таблицата в базата от данни се използва анотацията “@Column”.

@Column(nullable = false, unique = true)  
private String username;

Това е примерен код за маркирането на полето “username” като колона в таблицата за потребители. S “nullable = false” се забранява полето да има стойност “null”, а с “unique = true” означава че трябва да бъде уникално в таблицата в базата данни.

Някои колони изискват неопределена дължина на текста. Пример за такава колона е описанието на измервателната станция:

@Lob  
private String description;

С тази анотация се означава, че съдържанието на полето ще бъде запазено като бинарна информация с неопределена дължина.

## **3.3. Видове използвани релации използвани в главния сървър**

Релациите между връзки отново са реализирани чрез помощта на допълнителната библиотека наречена Hibernate. Пример за връзка едно-към-едно има само при наследените класове AnomalyDetectionRule -> HumidityAnomalyDetectionRule и TemperatureAnomalyDetectionRule. Връзката се прави автоматично от Hibernate, заради използваната стратегия за наследяване “Joined”. Тя означава, че всеки наследник ще има собствена таблица с неговите специфични колони, а общите колони ще се преизползват от наследената таблица:

@Entity  
@Inheritance(strategy = InheritanceType.*JOINED*)  
@Table(name = "anomaly\_detection\_rules")  
public abstract class AnomalyDetectionRule extends IdEntity {

…

}

@Entity  
@Table(name = "humidity\_anomaly\_detection\_rules")  
public class HumidityAnomalyDetectionRule extends AnomalyDetectionRule {

…

}

Едно-към-много връзка е реализирана при потребители и измервателни станции, тъй като един потребител може да има много измервателни станции, а една измервателна станция може да има само един потребител. Пример:

@LazyCollection(LazyCollectionOption.*FALSE*)  
@Fetch(value = FetchMode.*SUBSELECT*)  
@OneToMany(mappedBy = "owner", targetEntity = Raspberry.class, fetch = FetchType.*EAGER*, cascade = CascadeType.*ALL*)  
private List<Raspberry> raspberries = Collections.*emptyList*();

“OneToMany” анотацията е използвана в случая. С “targetEntity” се обозначава класът (таблицата), с която е направена релацията. С “mappedBy” се означава полето от другия клас, което е връзката от другата страна, тъй като всяка една от релациите между класовете в двупосочна (“bidirectional”). Тук “fetch” типът е “EAGER”, което означава, че при вземането на класа ще се направи “join” с другата таблица.

Връзка много-към-много е използвана за потребители и ролите, които те имат. Връзката е много-към-много, тъй като много потребители може да имат много от ролите, както всяка роля може да има много потребител. Тази връзка се изразява чрез “@ManyToMany” анотацията. Пример:

@LazyCollection(LazyCollectionOption.*FALSE*)  
@Fetch(value = FetchMode.*SUBSELECT*)  
@ManyToMany(mappedBy = "users", targetEntity = Role.class, fetch = FetchType.*EAGER*, cascade = CascadeType.*ALL*)  
private List<Role> roles = Collections.*emptyList*();

Връзката е сходна с едно към много, но от другата страна има различия:

@LazyCollection(LazyCollectionOption.*FALSE*)  
@Fetch(value = FetchMode.*SUBSELECT*)  
@ManyToMany  
@JoinTable(name = "roles\_users", joinColumns = @JoinColumn(name = "role\_id"), inverseJoinColumns = @JoinColumn(name = "user\_id"))

В един от двата класа трябва да се посочи името на допълнителната таблица, както и нейните колони и по кои колони се извършват “join”-овете. Анотацията “@JoinTable” е използвана в случая.

Както може би се забелязва, всички тези анотации идват JPA (“Java Persistence API”) и са доста известни и използвани. JPA предоставя интерфейсите и основните анотации, които фреймуъркът трябва да имплементира, в случая Hibernate, за да спази този стандарт. Има и други доставчици на тази имплементация и ако Hibernate бъде заменен с друг такъв, това не би трябвало да се отрази на кода и това го прави изключително гъвкав.

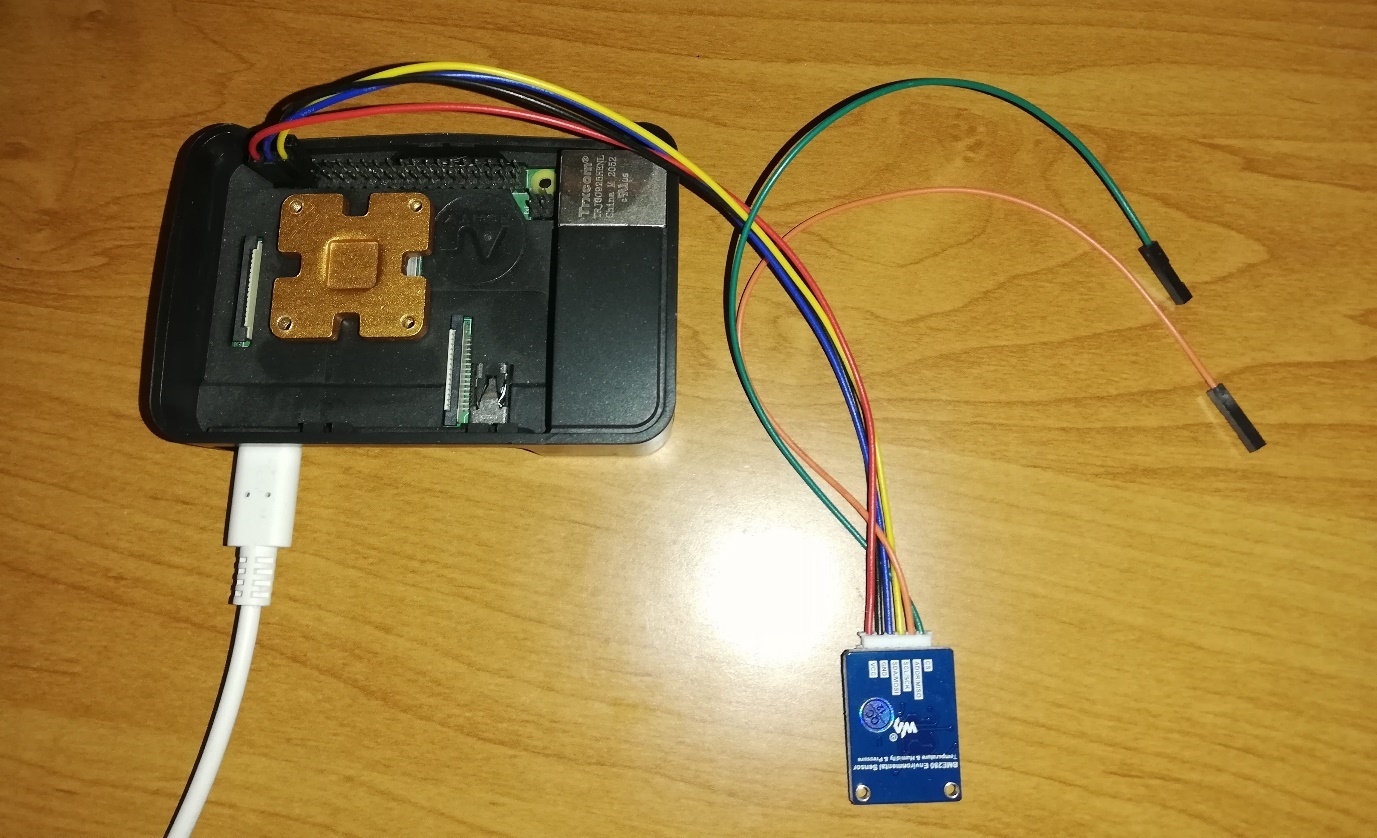
## **3.4. Какво представлява измервателната станция**

Измервателната станция представлява Raspberry PI компютър, върху който е инсталирана Raspbian PI OS, специфична Linux дистрибуция, предназначена за този тип компютри.

Една мощна характеристика на Raspberry Pi е редът от GPIO щифтове по горния ръб на дъската. GPIO означава вход/изход с общо предназначение. Тези щифтове са физически интерфейс между Raspberry Pi и външния свят. На най-простото ниво можете да мислите за тях като превключватели, които можете да включвате или изключвате (вход) или които Pi може да включва или изключва (изход).

GPIO щифтовете позволяват на Raspberry Pi да контролира и наблюдава външния свят, като е свързан към електронни схеми. Pi е в състояние да контролира светодиодите, да ги включва или изключва, да работи с двигатели и много други неща. Той също така може да открие дали е натиснат превключвател, температурата и светлината. Наричаме това като физически изчисления.

Има 40 щифта на Raspberry Pi (26 щифта при ранните модели) и те предоставят различни различни функции. [33]



Фиг. 16 Измервателна станция свързана за BME 280 през GPIO

Върху него е инсталиран Python Flask сървър, който служи като REST API, както и клиент, които позволяват комуникацията с главния сървър. Сървърът изглежда по следния начин:

sending=False

app = Flask(\_\_name\_\_)

app.config["DEBUG"] = True

CORS(app)

port = 1

address = 0x77

bus = smbus2.SMBus(port)

@app.route('/weather', methods=['GET'])

def get\_weather():

    calibration\_params = bme280.load\_calibration\_params(bus, address)

    data = bme280.sample(bus, address, calibration\_params)

    print(data)

    class Weather:

        def \_\_init\_\_(self, temperature, humidity, pressure):

            self.temperature = temperature

            self.humidity = humidity

            self.pressure = pressure

    w = Weather(data.temperature, data.humidity, data.pressure)

    jsonStr = json.dumps(w.\_\_dict\_\_)

    return Response(jsonStr, mimetype='application/json')

@app.route('/start', methods=['POST'])

def start\_sending():

    server\_name = json.loads(request.data)["hostname"]

    global sending

    sending=True

    bashCommand = "hostname -I"

    process = subprocess.Popen(bashCommand.split(), stdout=subprocess.PIPE)

    output, error = process.communicate()

    hostname = re.findall("\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}", output.decode('utf-8'))

    x = threading.Thread(target=thread\_function, args=(hostname[0], server\_name))

    x.start()

    return Response("", mimetype='application/json')

@app.route('/stop', methods=['POST'])

def stop\_sending():

    global sending

    sending=False

    return Response("", mimetype='application/json')

def thread\_function(local\_hostname, server\_hostname):

    while (sending):

        print("Thread starting! Localhost: " + local\_hostname + ". Server host: " + server\_hostname)

        calibration\_params = bme280.load\_calibration\_params(bus, address)

        data = bme280.sample(bus, address, calibration\_params)

        #print(data)

        w = {'temperature': data.temperature,'humidity': data.humidity, 'pressure': data.pressure, 'raspberryRoute': local\_hostname}

        headers = {'Content-Type': 'application/json'}

        try:

            r = requests.post('http://'+ server\_hostname + ':8080/measurement',json=w, headers=headers, timeout=10) # add authentication

            print(r.text)

        except:

            print("Error occurred while communicating with server")

        time.sleep(4)

        print("Thread finishing")

app.run(host="0.0.0.0", port=8080)

Предоставя три „крайни точки“, на които може да бъде достъпван. Първата „/weather“ връща информацията за температурата и влажността, която се чете през използваната python библиотека от сензора. На нея се подава адресът на сензора, който може да се види чрез командата „sudo i2cdetect -y 1“. Библиотеката разчита на I2C протокола.

I2C съчетава най-добрите характеристики на SPI и UART. С I2C може да се свържат множество подчинени устройства към едно главно устройство (като SPI) и може да има множество глави, контролиращи единични или няколко подчинени устройства. Това е наистина полезно, когато трябва да има повече от един микроконтролер, регистриращ данни на една карта с памет или показване на текст на един LCD дисплей. Подобно на UART комуникацията, I2C използва само два проводника за предаване на данни между устройства:

1. SDA (серийни данни) – линията за главен и подчинен за изпращане и получаване на данни.
2. SCL (Serial Clock) – Линията, която носи clock сигнал.

I2C е сериен комуникационен протокол, така че данните се прехвърлят бит по бит по един проводник (линията SDA).

Подобно на SPI, I2C е синхронен, така че изходът на битове се синхронизира със семплирането на битове чрез тактов сигнал, споделен между главния и подчинения. Сигналът на clock-a винаги се управлява от главния.

С I2C данните се прехвърлят в съобщения. Съобщенията са разбити на рамки от данни. Всяко съобщение има адресна рамка, която съдържа двоичния адрес на подчинения и един или повече кадри с данни, които съдържат данните, които се предават. Съобщението включва също условия за стартиране и спиране, битове за четене/запис и ACK/NACK битове между всеки фрейм от данни.

След като сме намерили адреса на сензора, следва отворим I2C bus връзка, което става през „smbus2.SMBus(port)“. След това се калибрира сензора и се взима информацията за температура и влажност, която се връща като HTTP отговор.

Другите 2 метода са за стартиране и спиране на станцията. Те отново ползват същата библиотека, за да взимат температурата и влажността. При тях разликата е, че сървърът изпраща своя IP адрес, измервателната станция стартира нова нишка и започва да праща измерванията на пратения IP адрес. Това става докато не се получи заявка за спиране на станцията. Заявките се пращат през няколко секунди, като максималното време, което може да се забави отговор от сървъра е 2 минути.

## **3.5. Какво представлява умният контакт**

Shelly Plug S е IoT (internet of things) устройство, което представлява контакт, който има Wi-Fi и Bluetooth връзка. През Wi-Fi мрежа, той може да бъде включван и изключван. Поддържа напрежение от 110 до 230V и издържа но мощност до 2500W. Може да бъде използвано самостоятелно. Също така контактът предоставя потребителски интерфейс, както и мобилно приложение, през което може да бъде управляван, но те не се ползват в системата. Единственото, което се ползва от умния контакт е REST API-я, който предоставя, като през него той се включва и изключва. За да се използва в системата, той има зададен статичен IP адрес. Допълнителни характеристики:

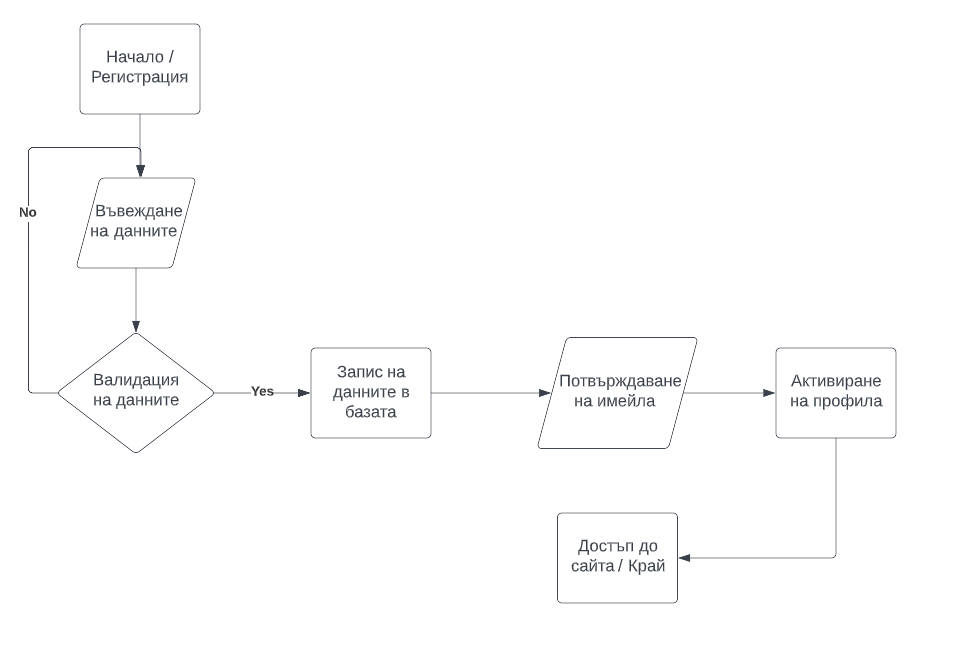
* Wi-Fi свързаност – Свържете се към Wi-Fi мрежа и се стартира чрез телефон, който се свързва към него чрез bluetooth.
* Управление на захранването - Shelly Plug може да изключи всеки уред, ако е достигнат предварително зададено ниво на потребление на енергия.
* Безплатен облак - Всичко, което трябва да управлявате вашия Shelly PLUG S е смартфон и безплатно мобилно приложение Shelly Cloud
* Последвайте слънчевата светлина - Shelly Plug може автоматично да се включи / да възвърне основата на часа на изгрева и залеза.
* Седмично планиране - Създавайте персонализирани дневни/седмични графики за вашите устройства.
* Мобилен контрол за използване - Можете да включвате и изключвате уредите си от всяка точка по света.
* Таймер за прекъсване - Няма вече забравени включени уреди. Интегрираният таймер за отброяване може автоматично да изключи фурната след час.
* До 2500W - с Shelly Plug можете да контролирате широка гама от устройства и устройства.
* Съвместим с Android, iOS, Amazon Alexa и Google Assistant



Фиг. 17 Умен контакт Shelly Plug S

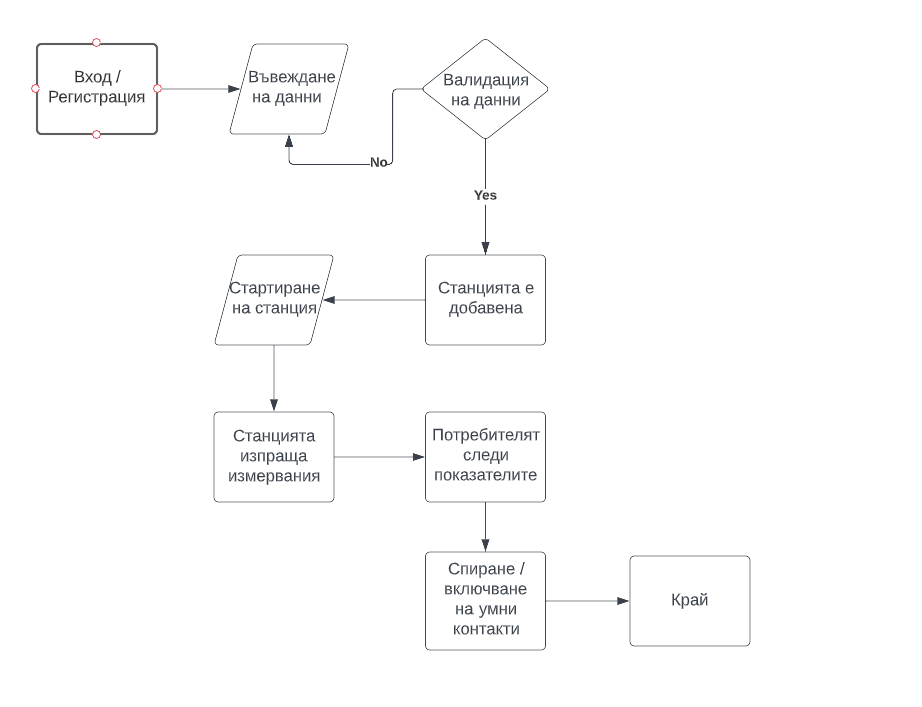
## **3.6. Блокови схеми на основните операции**

### **А) Блокова схема на регистрация на потребител**



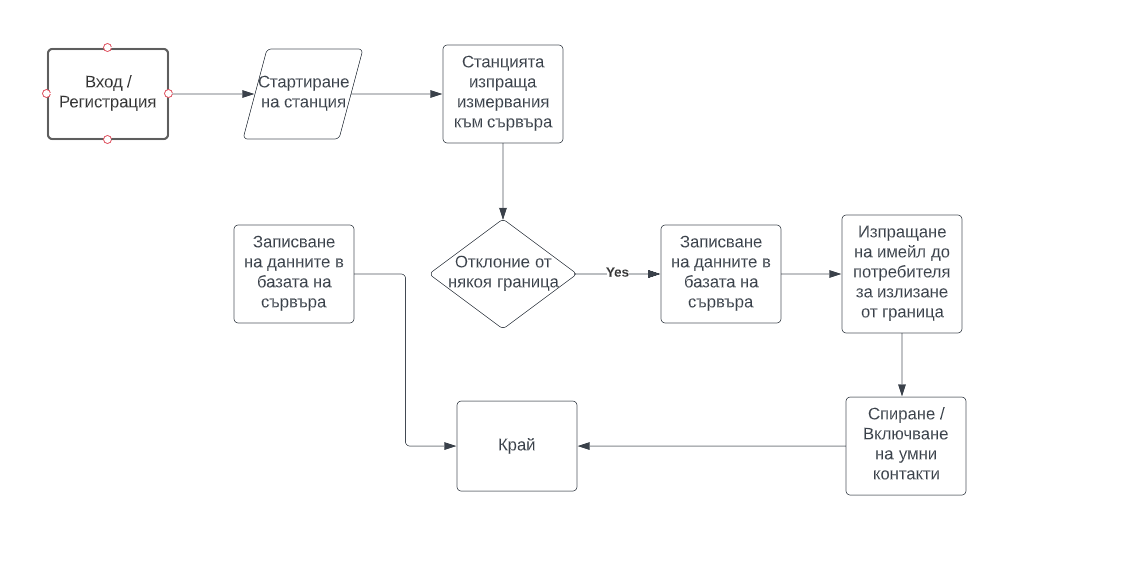
Фиг 18. Блокова схема – регистрация

### **Б) Блокова схема за добавяне измервателна станция**



Фиг 19. Блокова схема – добавяне на измервателна станция

### **В) Блокова схема на излизане от граница**



Фиг 20. Блокова схема – засичане на отклонение от граница

### **Д) Блокова схема на промяна на регистрация на потребител**



Фиг 21. Блокова схема – промяна на роля на потребител

# **4. Валидация на системата**

Валидирането на системата е извършвано както с ръчно тестване, така и с unit тестове, които се изпълняват преди стартирането на програмата. Unit тестовете са само на сървърното приложение, тъй като там се намира най-голямата част от логиката на цялата платформа. Колкото е добри да са unit тестовете, дори и да има 100% покритие на кода, те не са достатъчни, за да верифицират, че системата работи правилно от край до край, тъй като тяхната цел е да валидира клас по клас или съвкупност от няколко класа.

## **4.1. Unit тестове**

Тестовете, които приложението съдържа валидират работата на услугите, които са посредникът за работа с базата и другите части от приложението, които изискват достъп до базата.

1. AnomalyDetectionRuleServiceTests

@Test  
public void testSaveAnomalyDetection(…) {} – тества дали границата за отклонение е запазена правилно в базата данни. Връзката с измервателната станция също трябва да бъде добавена, което също е покрито от теста.

@Test  
public void testFindAll() {…} – тества дали правилно ще бъдат върнати всички граници на отклонения. Това също тества полиморфизма на границите.

@Test  
public void testFindById() {…} – тества дали границата ще бъде намерена правилно по нейния идентификационен номер.

@Test  
public void testFindByRaspberryId() {…} – тества дали правилно ще се намерят всички измервания по подаден идентификационен номера на измервателна станция.

@Test  
public void testUpdate() {…} – тества дали правилно ще бъде редактирана измервателна станция.

1. MeasurementServiceTests

@Test  
public void testSave() {…} – тества дали правилно се запазва измервателната станция в базата данни. Както и дали се спира/включва измервателна станция, както и дали се изпраща имейл на потребителя.

@Test  
public void testGetMeasurementsBetween() {…}- тества дали правилно се вземат всички измервания за определена измервателна станция за определен период от време.

@Test  
public void testDeleteMeasurementsOlderThan() {…}- тества дали правилно се изтривате всички стари неактуални измервания, които са добавени преди определена дата и час.

1. PowerPlugServiceTests

@Test  
public void testSave() {} – тества дали правилно се запазва умен контакт в базата.

@Test  
public void testUpdate() {}- тества дали правилно се редактира умен контакт, който след това се запазва в базата.

@Test  
public void testFindAllByRaspberryId() {}- тества дали правилно се намират всички умни контакти, по идентификационен номер на измервателна станция.

1. RaspberryServiceTest

@Test  
public void testSave() {} – тества дали правилно се запазва измервателна станция в базата данни.

@Test  
public void testUpdate() {} - тества дали правилно се редактира измервателна станция, която след това се запазва базата данни.

@Test  
public void testStart() {} - тества дали правилно се стартира измервателна станция и дали състоянието ѝ се редактира в базата.

@Test  
public void testStop() {} – тества дали правилно се спира измервателна станция и дали състоянието ѝ се редактира в базата.

1. SubscribedEmailServiceTest

@Test  
public void testSave() {} – тества дали правилно се запазва имейлът на потребител, който не е регистриран в системата.

1. UserServiceTests

@Test  
public void testSave() {} – тества дали правилно се запазва потребител в базата и дали се изпраща имейл за активация на акаунта на посочения имейл адрес.

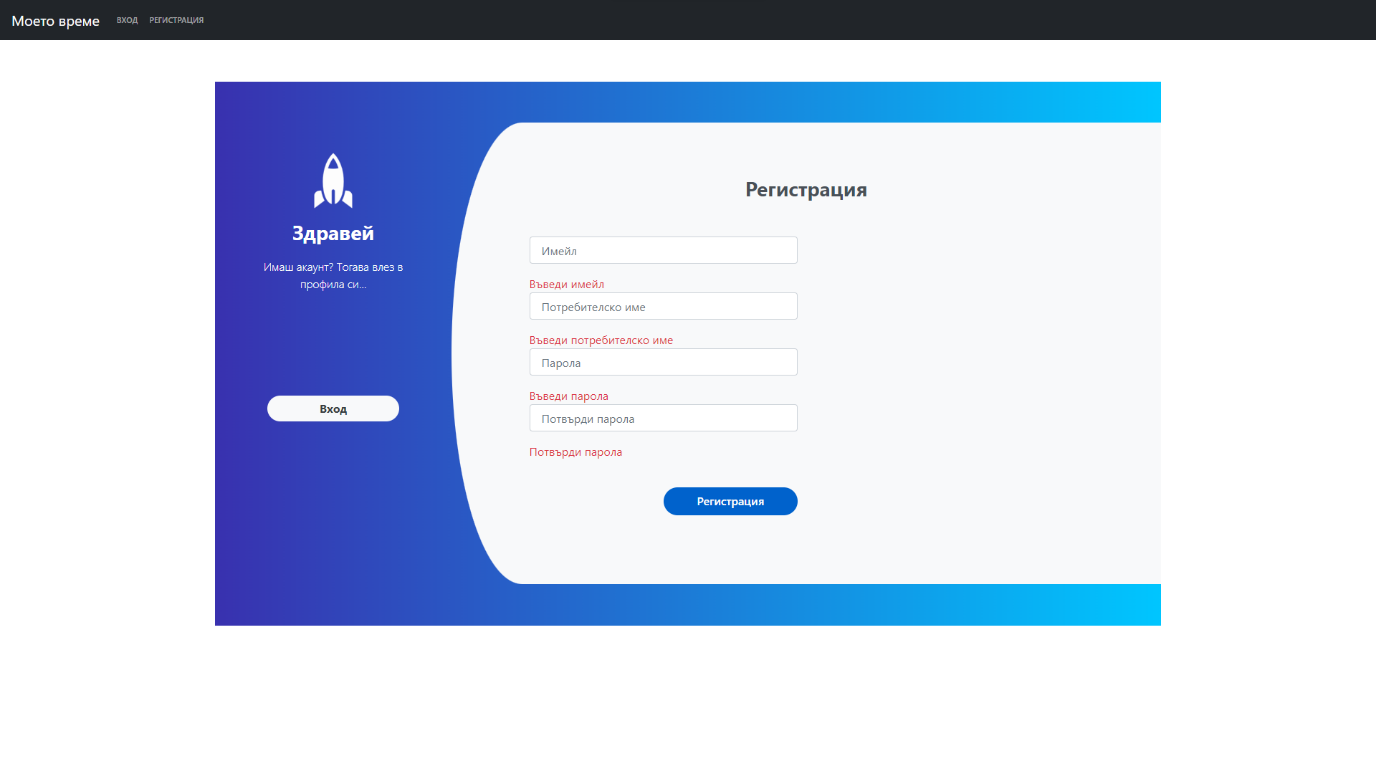
@Test  
public void testActivate() {} – тества дали правилно се активира акаунтът на потребител.

@Test  
public void testFindUserByUsername() {} – тества дали правилно се намира потребителят по неговото потребителско име.

@Test  
public void testFindUserRaspberries() {} – тества дали правилно се намират всички измервателни станции, които потребителят е регистрирал в системата.

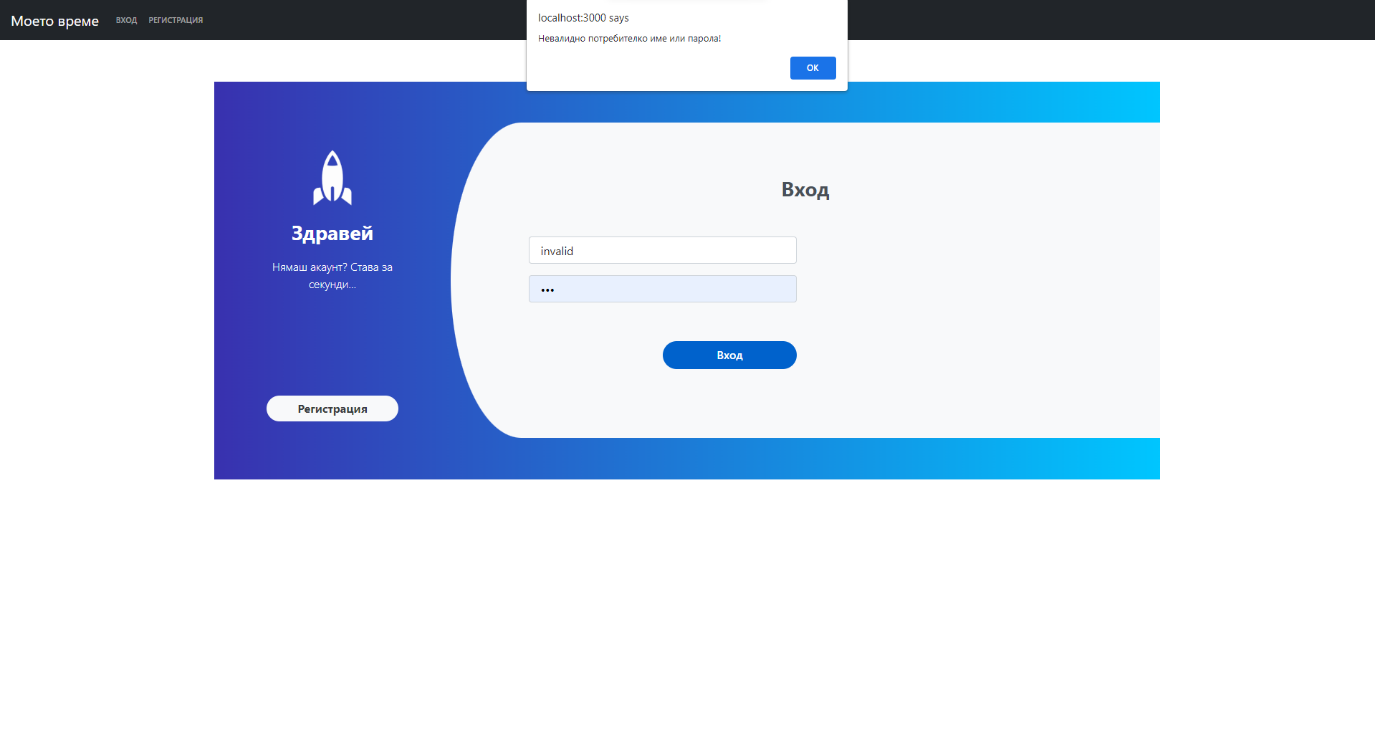
## **4.2. Ръчни тестове**

1. Регистрация с невалидни данни:



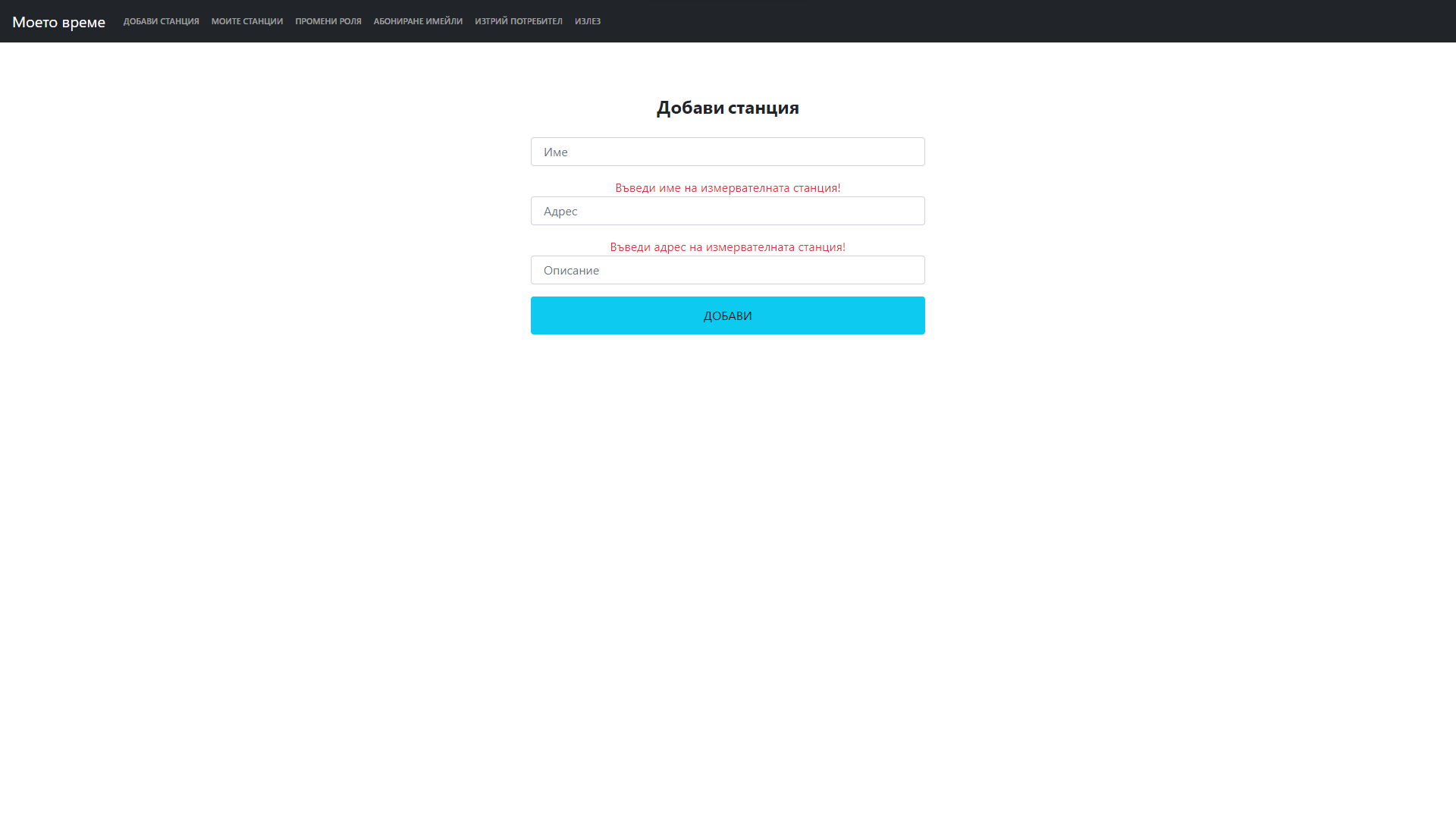
Фиг. 22 Форма за регистрация с невалидни данни

1. Вход с невалиден потребител и парола:



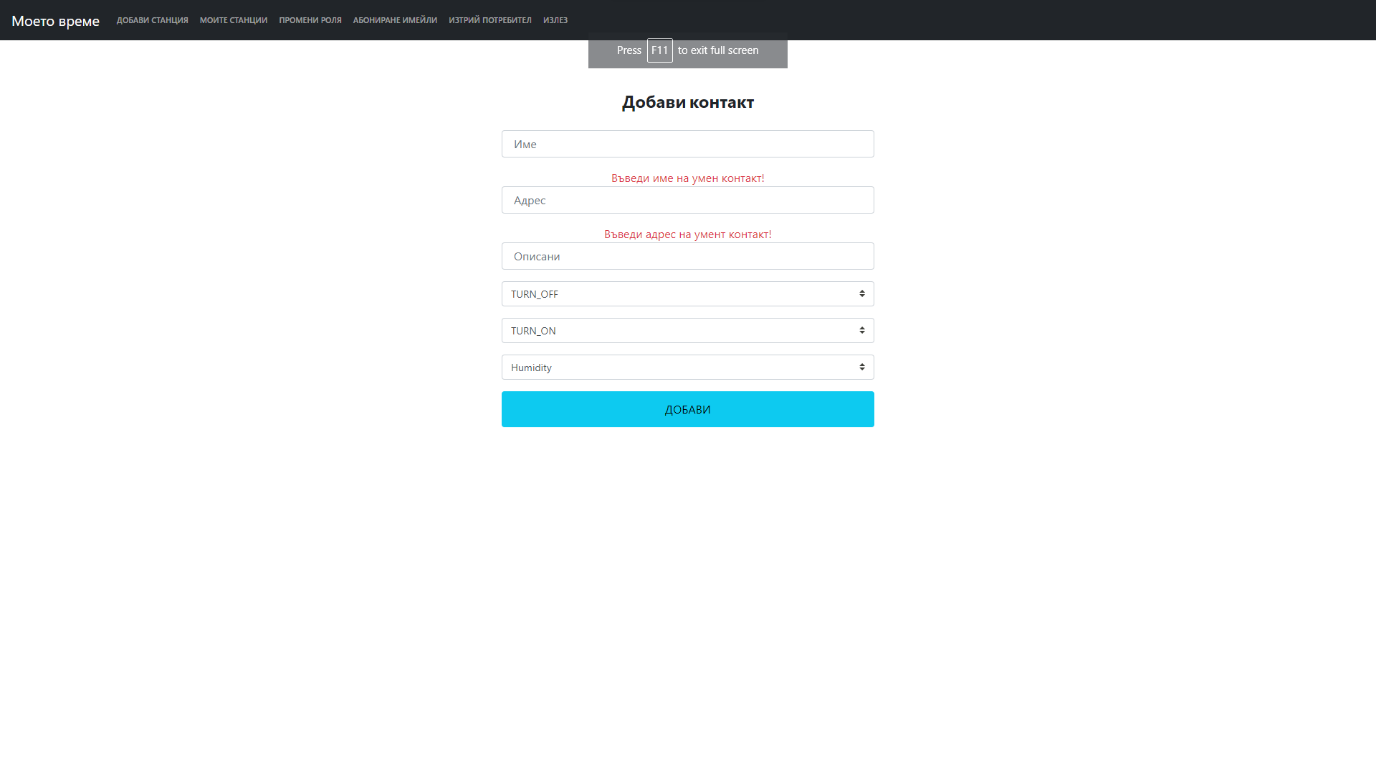
Фиг. 23 Форма за вход с невалиден потребител и парола

1. Добавяне на измервателна станция с невалидни данни:



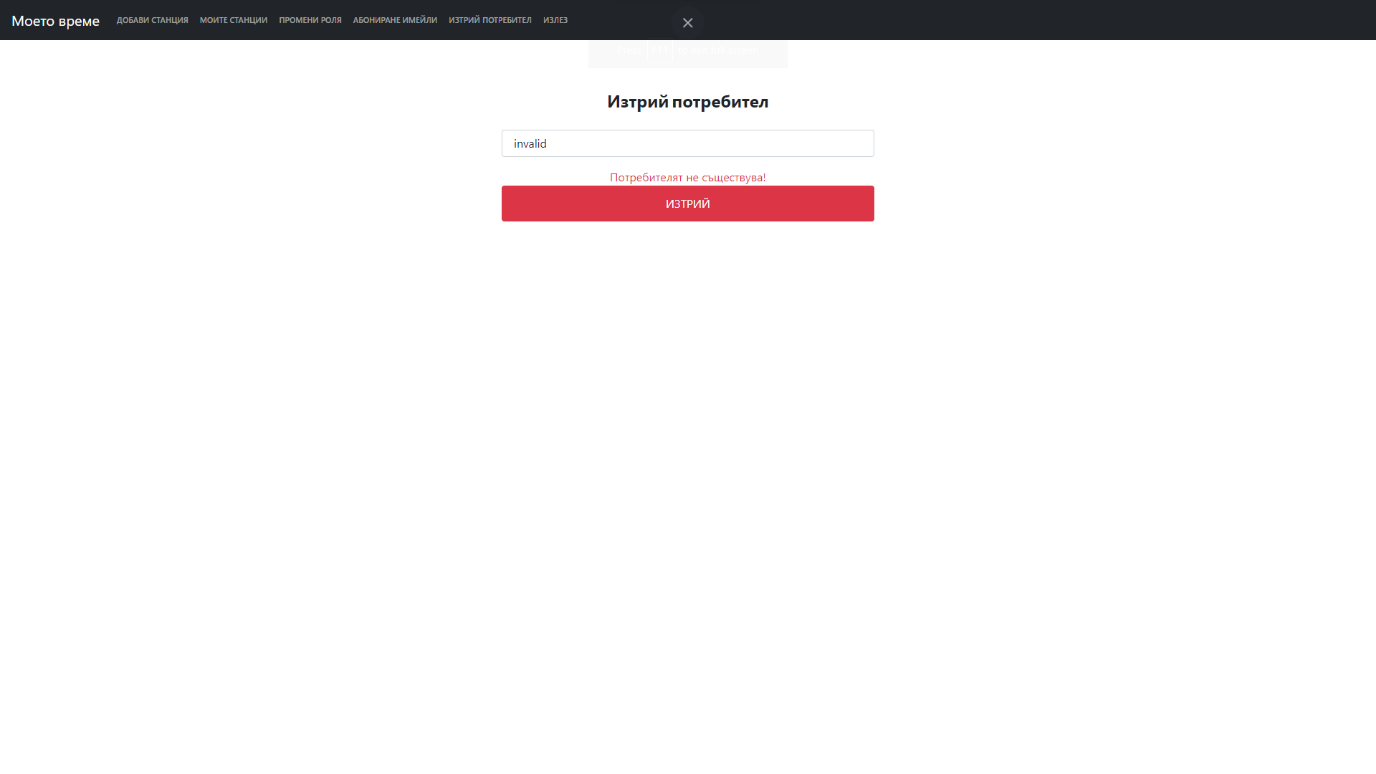
Фиг. 24 Неправилно добавяне на измервателна станция

1. Добавяне на умен контакт с невалидни данни:



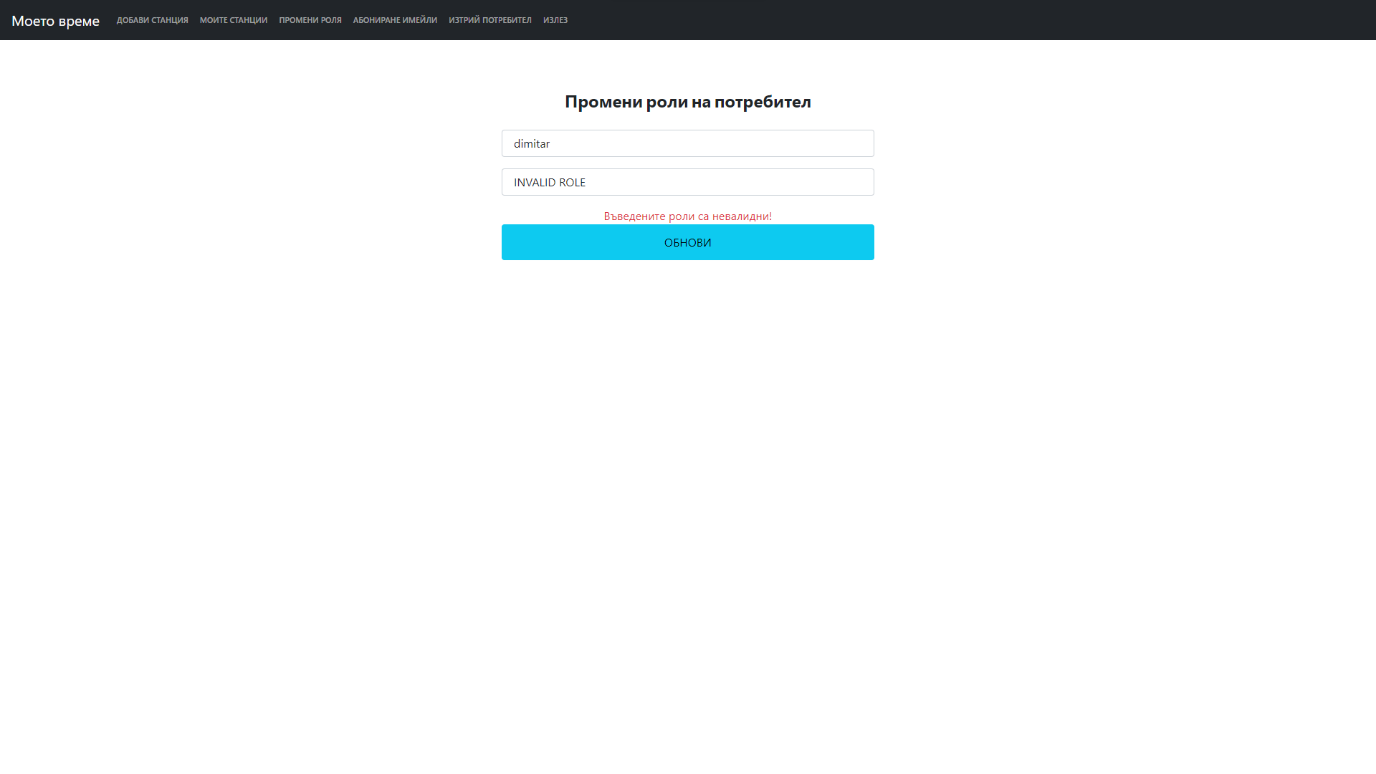
Фиг. 25 Неправилно добавяне на умен контакт

1. Изтриване на несъществуващ потребител от администратор:



Фиг. 26 Изтриване на потребител, който не съществува

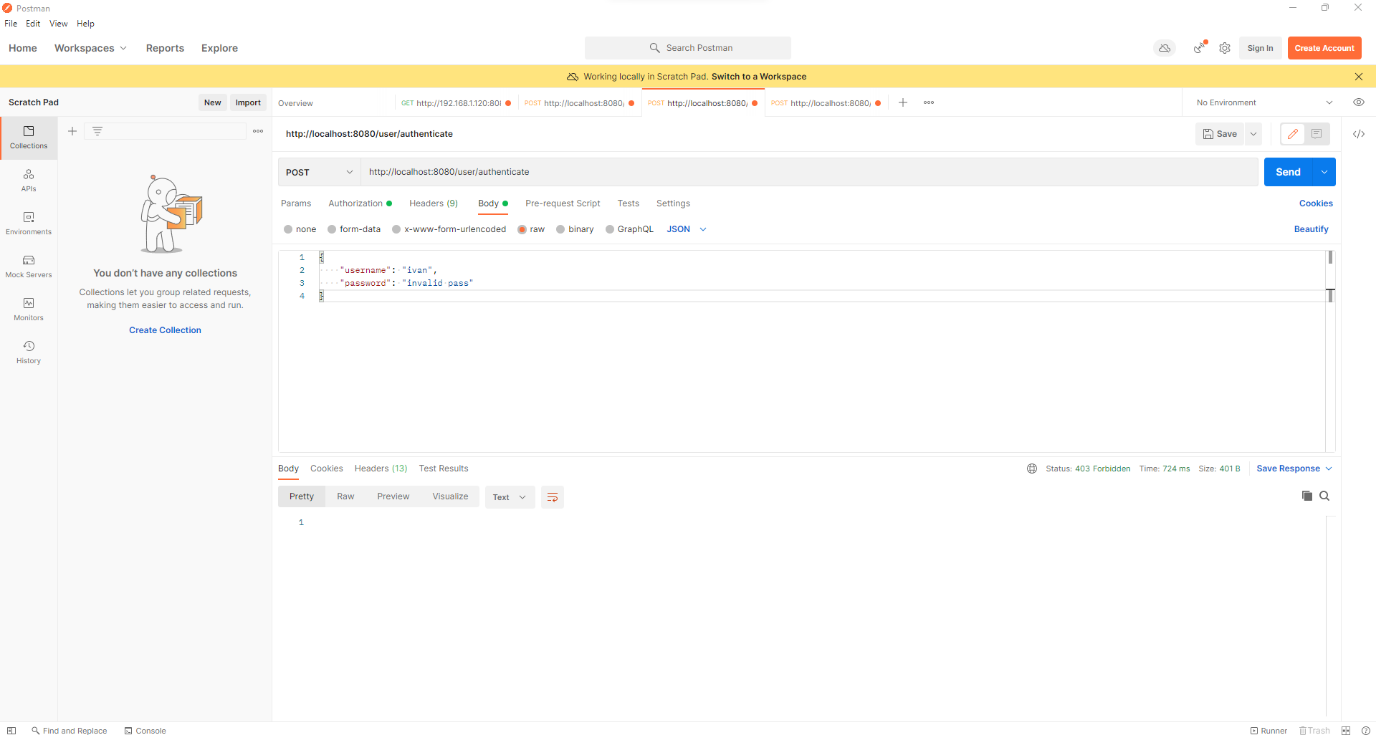
1. Промяна на роля на потребител с невалидна такава:



Фиг. 27 Промяна с невалидна роля

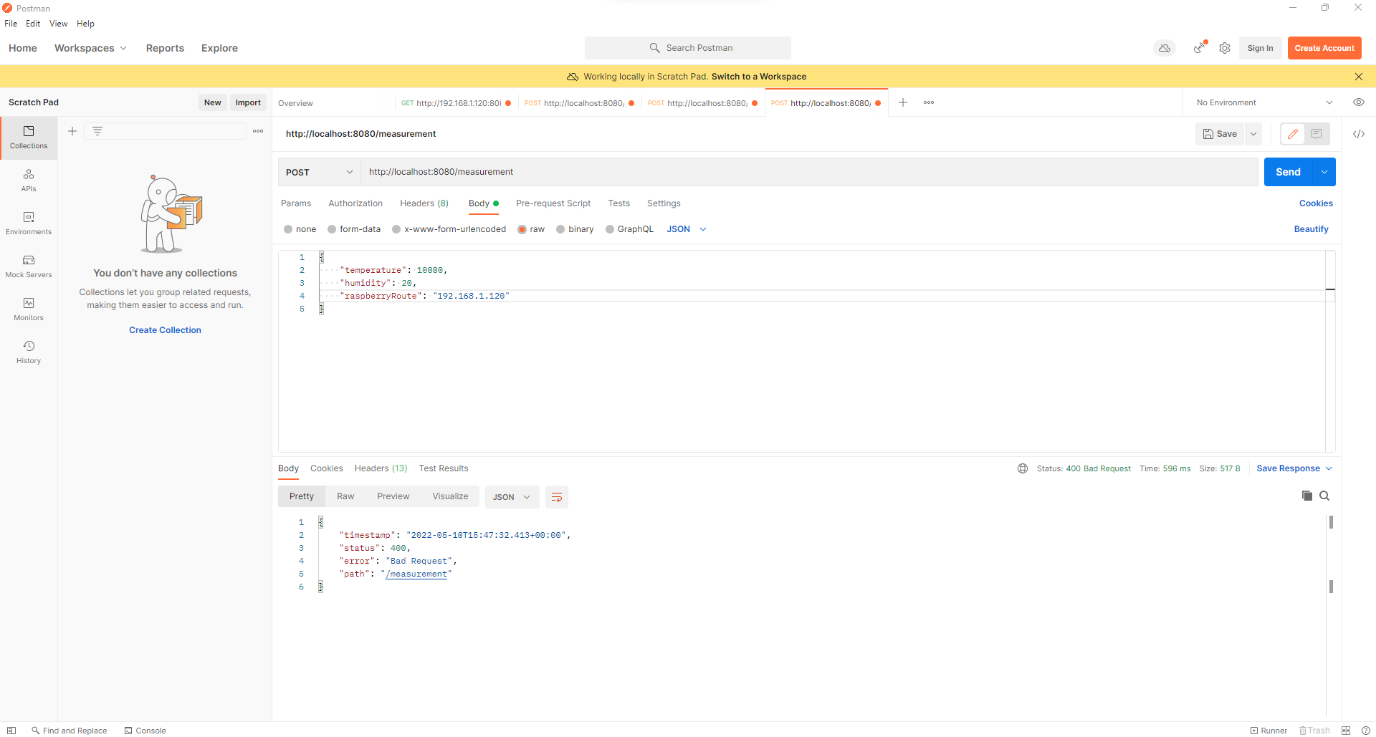
Заявки директно към сървъра с невалидни данни. Използван е един от най-известните клиенти за изпращането на REST заявки:

1. Вход с невалидни данни. Потребителят е подал невалидна парола:



Фиг. 21 Изпращане на заявка с невалидна парола на потребител

1. Добавяне на измерване, с твърде голяма температура (10000 градуса)



Фиг. 22 Изпращане на измерване с невалидна температура

# **Заключение**

Както става ясно от предходни точки, подробното описание на това какво прави системата, използваните технологии и програмната реализация, в основата си приложението представлява цялостна системата, която следи и управлява температура и влажност в затворени помещения с помощта на устройства, които биват включени в умните контакти. Освен, че следи и контролира показанията системата също така служи и за известяването на своите потребители за засечени отклонение на показатели спрямо зададените от тях норми. Освен че системата ще спре или включи умен контакт, тя ще извести също и нейните потребители, като така може още по-бързо може да се реагира, ако възникне някакъв проблем в помещение, където е нужна определена влажност и температура.

Целта на приложението е контролира условията на температура и влажност в помещение. Това е ключово на изключително много места, като музеи, парници, спа центрове, дори може да се използва и вкъщи като така могат да се поддържат най-оптималните нива на температура и влажност, като така се хората ще имат по-добро здравословно състояние и по приятен начин на живот.

Най-големият плюс на системата е, че ако тя бъде настроена веднъж повече, не налага никой да я следи. Тя е автономна, което я прави изключително удобно тъй като не налага необходимостта от оператор върху нея.

С отворения код се цели и включването на още хора, които имат желание да развиват системата. Чрез този подход може да се създаде общност, която да разработва и развива софтуер, което ще допринася както за развитието на цялата софтуерна и технологична индустрия, така и за подобряването и модернизацията на заобикалящия ни свят.

Както се вижда, проектът е все още в начален стадий и няма реални потребители, но е готов за началната употреба. След използването му от първите си клиенти, най-вероятно ще изникват проблеми, които ще могат да бъдат лесно поправени заради гъвкавостта на кода. В бъдеще ще бъдат добавени още функционалности като измерване и на другите показатели, като прахови частици, замърсяване на въздуха с различни видове химикали, както и още по-доброто известяване на потребители, като така се цели все по-добрата автоматизация на заобикалящия ни свят.

# **Код**

По-долу могат да бъдат разгледани някои извадки на по-интересен код. Показаните класове не са пълни.

Линкове към пълния код:

- Клиентско приложение (client) -<https://github.com/IvanBorislavovDimitrov/weather-station-client>

- Сървърно приложение (server) - <https://github.com/IvanBorislavovDimitrov/weather-station-server>

- Сървърно приложение на измервателна станция - <https://github.com/IvanBorislavovDimitrov/weather-station-sensor>

1. RaspberryRemoteControl

public class RaspberryRemoteControl {  
  
 private static final Logger *LOGGER* = LoggerFactory.*getLogger*(RaspberryRemoteControl.class);  
  
 private final WebClient webClient;  
 private final EnvironmentGetter environmentGetter;  
  
 @Autowired  
 public RaspberryRemoteControl(WebClient webClient, EnvironmentGetter environmentGetter) {  
 this.webClient = webClient;  
 this.environmentGetter = environmentGetter;  
 }  
  
 public void startRaspberry(String raspberryRoute) {  
 webClient.post()  
 .uri(getRaspberryFullRoute(raspberryRoute, Constants.*RASPBERRY\_ACTION\_START\_PATH*))  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .body(BodyInserters.*fromValue*(new StartRaspberryRequest(environmentGetter.getLocalHostname())))  
 .retrieve()  
 .bodyToMono(String.class)  
 .block();  
 }  
  
 public void stopRaspberry(String raspberryRoute) {  
 webClient.post()  
 .uri(getRaspberryFullRoute(raspberryRoute, Constants.*RASPBERRY\_ACTION\_STOP\_PATH*))  
 .retrieve()  
 .bodyToMono(String.class)  
 .block();  
 }  
  
 private String getRaspberryFullRoute(String raspberryRoute, String raspberryActionStart) {  
 return Constants.*HTTP\_PREFIX* + raspberryRoute + Constants.*RASPBERRY\_DEFAULT\_PORT* + raspberryActionStart;  
 }  
  
}

1. PowerPlugRemoteControlc

@Component  
public class ResourceServiceApiFacade implements ResourceServiceApi {  
  
 private final RemoteStorageFileGetter remoteStorageFileGetter;  
  
 @Autowired  
 public ResourceServiceApiFacade(RemoteStorageFileGetter remoteStorageFileGetter) {  
 this.remoteStorageFileGetter = remoteStorageFileGetter;  
 }  
  
 @Override  
 public byte[] getImageResource(String name) {  
 return remoteStorageFileGetter.getImageResource(name);  
 }  
  
 @Override  
 public InputStream openFileStream(String name, Long offset) {  
 return remoteStorageFileGetter.getFileStream(name, offset);  
 }  
}ce

1. GitHubServiceApiFacade

@Service  
public class GithubServiceApiFacade extends ExtendableMapper<GithubAccessTokenServiceModel, GithubAccessTokenResponseModel> implements GithubServiceApi {  
  
 private static final Logger *LOGGER* = LoggerFactory.*getLogger*(GithubServiceApiFacade.class);  
  
 private final Parser parser;  
 private final ResilientHttpClient resilientHttpClient;  
 private final ApplicationConfiguration applicationConfiguration;  
 private final UserService userService;  
 private final GithubService githubService;  
  
 @Autowired  
 public GithubServiceApiFacade(Parser parser, ResilientHttpClient resilientHttpClient,  
 ModelMapper modelMapper, ApplicationConfiguration applicationConfiguration, UserService userService, GithubService githubService) {  
 super(modelMapper);  
 this.parser = parser;  
 this.resilientHttpClient = resilientHttpClient;  
 this.applicationConfiguration = applicationConfiguration;  
 this.userService = userService;  
 this.githubService = githubService;  
 }  
  
 @Override  
 public ResponseEntity<GithubUserResponseModel> getGithubUserInfo(String username) {  
 UserServiceModel userServiceModel = userService.findByUsername(username);  
 GithubAccessTokenServiceModel githubAccessTokenServiceModel = getGithubAccessTokenServiceModel(username, userServiceModel);  
 HttpGet userRequest = new HttpGet(getUsernameResource());  
 userRequest.setHeader(HttpHeaders.*AUTHORIZATION*, "Bearer " + githubAccessTokenServiceModel.getAccessToken());  
 HttpResponse userResponse = resilientHttpClient.execute(userRequest);  
 validateRequest(username, userResponse);  
 GithubUserResponseModel githubUserResponseModel = parser.deserialize(UncheckedEntityUtils.*getResponseBody*(userResponse),  
 GithubUserResponseModel.class);  
 return ResponseEntity.*ok*(githubUserResponseModel);  
 }  
  
 private GithubAccessTokenServiceModel getGithubAccessTokenServiceModel(String username, UserServiceModel userServiceModel) {  
 GithubAccessTokenServiceModel githubAccessTokenServiceModel = userServiceModel.getGithubAccessToken();  
 if (githubAccessTokenServiceModel == null) {  
 throw new GithubException(MessageFormat.*format*(Messages.*NO\_GITHUB\_ACCESS\_TOKEN\_FOR\_USER*, username));  
 }  
 return githubAccessTokenServiceModel;  
 }  
  
 private void validateRequest(String username, HttpResponse userResponse) {  
 validateStatusNotFound(username, userResponse);  
 validateStatusUnauthorized(username, userResponse);  
 }  
  
 private void validateStatusNotFound(String username, HttpResponse response) {  
 if (response.getStatusLine().getStatusCode() == HttpStatus.*NOT\_FOUND*.value()) {  
 throw new GithubException(MessageFormat.*format*(Messages.*USER\_NOT\_FOUND*, username));  
 }  
 }  
  
 private void validateStatusUnauthorized(String username, HttpResponse response) {  
 if (response.getStatusLine().getStatusCode() == HttpStatus.*UNAUTHORIZED*.value()) {  
 throw new GithubException(MessageFormat.*format*(Messages.*ACCESS\_TOKEN\_FOR\_USER\_HAS\_EXPIRED*, username));  
 }  
 }  
  
 @Override  
 public ResponseEntity<GithubAccessTokenResponseModel> requestAccessTokenForUser(String username, String code) {  
 UserServiceModel userServiceModel = userService.findByUsername(username);  
 HttpResponse accessTokenResponse = executeAccessTokenRequest(code);  
 return processAccessTokenForUser(userServiceModel, accessTokenResponse);  
 }  
  
 private String getUsernameResource() {  
 return applicationConfiguration.getGithubApiUrl() + "/" + com.code.to.learn.web.constants.Constants.*USER\_RESOURCE*;  
 }  
  
 private HttpResponse executeAccessTokenRequest(String code) {  
 HttpPost accessTokenRequest = new HttpPost(applicationConfiguration.getGithubAccessTokenUrl());  
 List<NameValuePair> parameters = getAccessTokenParameters(code);  
 setFormEntity(accessTokenRequest, parameters);  
 return resilientHttpClient.execute(accessTokenRequest);  
 }  
  
 private List<NameValuePair> getAccessTokenParameters(String code) {  
 List<NameValuePair> parameters = new ArrayList<>();  
 parameters.add(new BasicNameValuePair(Constants.*CLIENT\_ID\_KEY*, applicationConfiguration.getClientIdValue()));  
 parameters.add(new BasicNameValuePair(Constants.*CLIENT\_SECRET\_KEY*, applicationConfiguration.getClientSecretValue()));  
 parameters.add(new BasicNameValuePair(Constants.*AUTHENTICATION\_CODE*, code));  
 return parameters;  
 }  
  
 private void setFormEntity(HttpPost httpPost, List<NameValuePair> parameters) {  
 try {  
 httpPost.setEntity(new UrlEncodedFormEntity(parameters));  
 } catch (UnsupportedEncodingException e) {  
 throw new LCException(e.getMessage(), e);  
 }  
 }  
  
 private ResponseEntity<GithubAccessTokenResponseModel> processAccessTokenForUser(UserServiceModel userServiceModel, HttpResponse accessTokenResponse) {  
 GithubAccessTokenResponseModel githubAccessTokenResponseModel = parseGithubAccessTokenResponse(accessTokenResponse);  
 updateGithubAccessTokenForUser(userServiceModel, githubAccessTokenResponseModel);  
 return ResponseEntity.*ok*(githubAccessTokenResponseModel);  
 }  
  
 private GithubAccessTokenResponseModel parseGithubAccessTokenResponse(HttpResponse accessTokenResponse) {  
 String accessTokenParametersQuery = UncheckedEntityUtils.*getResponseBody*(accessTokenResponse);  
 List<String> rawQueryParameters = Arrays.*asList*(accessTokenParametersQuery.split("&"));  
 Map<String, String> githubAccessTokenQueryParameters = getGithubAccessTokenParameters(rawQueryParameters);  
 verifyNotErrorIsReturned(githubAccessTokenQueryParameters);  
 return mapToGithubAccessToken(githubAccessTokenQueryParameters);  
 }  
  
 private Map<String, String> getGithubAccessTokenParameters(List<String> rawQueryParameters) {  
 return rawQueryParameters.stream()  
 .map(queryParameter -> queryParameter.split("="))  
 .collect(Collectors.*toMap*(accessTokenKeyValuePair -> accessTokenKeyValuePair[Constants.*ACCESS\_TOKEN\_KEY\_POSITION*],  
 this::getAccessTokenValueSafely));  
 }  
  
 private String getAccessTokenValueSafely(String[] accessTokenKeyValuePair) {  
 if (accessTokenKeyValuePair.length != Constants.*ACCESS\_TOKEN\_KEY\_VALUE\_SIZE*) {  
 return "";  
 }  
 return accessTokenKeyValuePair[Constants.*ACCESS\_TOKEN\_VALUE\_POSITION*];  
 }  
  
 public void verifyNotErrorIsReturned(Map<String, String> accessTokenQueryParameters) {  
 if (accessTokenQueryParameters.containsKey(Constants.*ERROR*)) {  
 throw new GithubException(com.code.to.learn.web.constants.Messages.*INVALID\_GITHUB\_VERIFICATION\_CODE*, accessTokenQueryParameters);  
 }  
 }  
  
 private GithubAccessTokenResponseModel mapToGithubAccessToken(Map<String, String> githubAccessTokenQueryParameters) {  
 return GithubAccessTokenResponseModel.*fromAccessTokenQueryParameters*(githubAccessTokenQueryParameters);  
 }  
  
 private void updateGithubAccessTokenForUser(UserServiceModel userServiceModel, GithubAccessTokenResponseModel githubAccessTokenResponseModel) {  
 GithubAccessTokenServiceModel githubAccessTokenServiceModel = toInput(githubAccessTokenResponseModel);  
 githubAccessTokenServiceModel.setUserServiceModel(userServiceModel);  
 userServiceModel.setGithubAccessToken(githubAccessTokenServiceModel);  
 githubService.save(githubAccessTokenServiceModel);  
 GithubAccessTokenServiceModel updateEntity = githubService.findByAccessToken(githubAccessTokenResponseModel.getAccessToken());  
 userServiceModel.setGithubAccessToken(updateEntity);  
 userService.update(userServiceModel);  
 }  
  
 @Override  
 public ResponseEntity<List<GithubRepositoryResponseModel>> getUserRepositories(String username) {  
 UserServiceModel userServiceModel = userService.findByUsername(username);  
 GithubAccessTokenServiceModel githubAccessToken = userServiceModel.getGithubAccessToken();  
 HttpGet repositoriesRequest = new HttpGet(getRepositoryResource());  
 repositoriesRequest.setHeader(HttpHeaders.*AUTHORIZATION*, "Bearer " + githubAccessToken.getAccessToken());  
 HttpResponse repositoriesResponse = resilientHttpClient.execute(repositoriesRequest);  
 validateRequest(username, repositoriesResponse);  
 String repositoriesResponseBody = UncheckedEntityUtils.*getResponseBody*(repositoriesResponse);  
 return ResponseEntity.*ok*(parser.deserialize(repositoriesResponseBody, new TypeReference<List<GithubRepositoryResponseModel>>() {  
 }));  
 }  
  
 @Override  
 public ResponseEntity<GithubRepositoryResponseModelExtended> getRepositoryForUser(String username, String repositoryName) {  
 UserServiceModel userServiceModel = userService.findByUsername(username);  
 GithubAccessTokenServiceModel githubAccessTokenServiceModel = userServiceModel.getGithubAccessToken();  
 GithubUserResponseModel githubUserInfo = getGithubUserInfo(username).getBody();  
 HttpGet repositoryRequest = new HttpGet(getCertainRepositoryResource(githubUserInfo.getLogin(), repositoryName));  
 repositoryRequest.setHeader(HttpHeaders.*AUTHORIZATION*, "Bearer " + githubAccessTokenServiceModel.getAccessToken());  
 HttpResponse repositoryResponse = resilientHttpClient.execute(repositoryRequest);  
 validateRequest(username, repositoryResponse);  
 String repositoryResponseBode = UncheckedEntityUtils.*getResponseBody*(repositoryResponse);  
 return ResponseEntity.*ok*(parser.deserialize(repositoryResponseBode, new TypeReference<GithubRepositoryResponseModelExtended>() {  
 }));  
 }  
  
 private String getRepositoryResource() {  
 return applicationConfiguration.getGithubApiUrl() + com.code.to.learn.web.constants.Constants.*REPOSITORIES\_RESOURCE*;  
 }  
  
 private String getCertainRepositoryResource(String username, String repositoryName) {  
 return applicationConfiguration.getGithubApiUrl() + com.code.to.learn.web.constants.Constants.*REPOSITORY\_RESOURCE* + "/" + username + "/" + repositoryName;  
 }  
  
 @Override  
 protected Class<GithubAccessTokenServiceModel> getInputClass() {  
 return GithubAccessTokenServiceModel.class;  
 }  
  
 @Override  
 protected Class<GithubAccessTokenResponseModel> getOutputClass() {  
 return GithubAccessTokenResponseModel.class;  
 }

1. CourseServiceApiFacade

@Component  
public class CourseServiceApiFacade extends ExtendableMapper<CourseServiceModel, CourseResponseModel> implements CourseServiceApi {  
  
 private final Object lock = new Object();  
 private final CourseService courseService;  
 private final RemoteStorageFileOperator remoteStorageFileOperator;  
 private final CourseValidator courseValidator;  
 private final UserService userService;  
 private final CourseCategoryService courseCategoryService;  
 private final ExecutorService executorService;  
 private final RemoteStorageFileGetter remoteStorageFileGetter;  
 private final ApplicationConfiguration configuration;  
 private final RatingEstimatorFactory ratingEstimatorFactory;  
  
 @Autowired  
 public CourseServiceApiFacade(CourseService courseService, ModelMapper modelMapper, RemoteStorageFileOperator remoteStorageFileOperator,  
 CourseValidator courseValidator, UserService userService,  
 CourseCategoryService courseCategoryService, ExecutorService executorService,  
 RemoteStorageFileGetter remoteStorageFileGetter, ApplicationConfiguration configuration, RatingEstimatorFactory ratingEstimatorFactory) {  
 super(modelMapper);  
 this.courseService = courseService;  
 this.remoteStorageFileOperator = remoteStorageFileOperator;  
 this.courseValidator = courseValidator;  
 this.userService = userService;  
 this.courseCategoryService = courseCategoryService;  
 this.executorService = executorService;  
 this.remoteStorageFileGetter = remoteStorageFileGetter;  
 this.configuration = configuration;  
 this.ratingEstimatorFactory = ratingEstimatorFactory;  
 }  
  
 @Override  
 public ResponseEntity<CourseResponseModel> add(CourseBindingModel courseBindingModel) {  
 courseValidator.validateCourseBindingModel(courseBindingModel);  
 List<FileToUpload> videosToUpload = getVideosToUpload(courseBindingModel);  
 List<FileMetadata> uploadedVideos = remoteStorageFileOperator.uploadFilesSync(videosToUpload);  
 CourseServiceModel courseServiceModel = toCourseServiceModel(courseBindingModel, uploadedVideos, true);  
 FileToUpload thumbnail = new FileToUpload(getThumbnailName(courseBindingModel), courseBindingModel.getThumbnail());  
 remoteStorageFileOperator.uploadFileAsync(thumbnail);  
 courseService.save(courseServiceModel);  
 return ResponseEntity.*ok*(toOutput(courseServiceModel));  
 }  
  
 private CourseServiceModel toCourseServiceModel(CourseBindingModel courseBindingModel, List<FileMetadata> uploadedVideos,  
 boolean shouldUpdateContent) {  
 CourseServiceModel courseServiceModel = getMapper().map(courseBindingModel, CourseServiceModel.class);  
 if (shouldUpdateContent) {  
 List<CourseServiceModel.CourseVideoServiceModel> videos = getVideos(courseBindingModel, uploadedVideos);  
 if (!videos.isEmpty()) {  
 courseServiceModel.setVideosNames(videos);  
 }  
 }  
 courseServiceModel.setThumbnailName(getThumbnailName(courseBindingModel));  
 UserServiceModel teacher = userService.findByUsername(courseBindingModel.getTeacherName());  
 courseServiceModel.setTeacher(teacher);  
 DateTimeFormatter formatter = DateTimeFormatter.*ofPattern*(*DATE\_PATTERN*);  
 courseServiceModel.setStartDate(LocalDate.*parse*(courseBindingModel.getStartDate(), formatter));  
 CourseCategoryServiceModel courseCategoryServiceModel = courseCategoryService.findByName(courseBindingModel.getCategoryName());  
 courseServiceModel.setCourseCategory(courseCategoryServiceModel);  
 return courseServiceModel;  
 }

1. GenericDaoImpl

public abstract class GenericDaoImpl<E extends IdEntity<E>> implements GenericDao<E> {  
  
 protected final SessionFactory sessionFactory;  
  
 public GenericDaoImpl(SessionFactory sessionFactory) {  
 this.sessionFactory = sessionFactory;  
 }  
  
 @Override  
 public List<E> findAll() {  
 Session session = DatabaseSessionUtil.*getCurrentOrOpen*(sessionFactory);  
 CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();  
 CriteriaQuery<E> criteriaQuery = criteriaBuilder.createQuery(getDomainClassType());  
 Root<E> root = criteriaQuery.from(getDomainClassType());  
 criteriaQuery.select(root);  
 Query<E> query = session.createQuery(criteriaQuery);  
 return query.getResultList();  
 }  
  
 @Override  
 public Optional<E> findById(String id) {  
 return findByField(IdEntity.*ID*, id);  
 }  
  
 @Override  
 public void persist(E entity) {  
 DatabaseSessionUtil.*getCurrentOrOpen*(sessionFactory)  
 .persist(entity);  
 }  
  
 @Override  
 public Optional<E> deleteById(String id) {  
 Optional<E> entity = findById(id);  
 if (entity.isPresent()) {  
 delete(entity.get());  
 return entity;  
 }  
 return Optional.*empty*();  
 }  
  
 @Override  
 public Optional<E> delete(E entity) {  
 DatabaseSessionUtil.*getCurrentOrOpen*(sessionFactory).delete(entity);  
 return Optional.*of*(entity);  
 }  
  
 @Override  
 public Optional<E> update(E entity) {  
 DatabaseSessionUtil.*getCurrentOrOpen*(sessionFactory).update(entity);  
 return Optional.*of*(entity);  
 }  
  
 @Override  
 public Optional<E> merge(E entity) {  
 DatabaseSessionUtil.*getCurrentOrOpen*(sessionFactory).merge(entity);  
 return Optional.*of*(entity);  
 }  
  
 @Override  
 public long count() {  
 Session session = DatabaseSessionUtil.*getCurrentOrOpen*(sessionFactory);  
 CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();  
 CriteriaQuery<Long> criteriaQuery = criteriaBuilder.createQuery(Long.class);  
 Root<E> root = criteriaQuery.from(getDomainClassType());  
 criteriaQuery.select(criteriaBuilder.count(root));  
 return getOrEmpty(session, criteriaQuery).get();  
 }  
  
 @Override  
 public Optional<E> findByField(String field, Object value) {  
 Session session = DatabaseSessionUtil.*getCurrentOrOpen*(sessionFactory);  
 CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();  
 CriteriaQuery<E> criteriaQuery = criteriaBuilder.createQuery(getDomainClassType());  
 Root<E> root = criteriaQuery.from(getDomainClassType());  
 criteriaQuery.select(root).where(criteriaBuilder.equal(root.get(field), value));  
 return getOrEmpty(session, criteriaQuery);  
 }  
  
 protected <T> Optional<T> getOrEmpty(Session session, CriteriaQuery<T> criteriaQuery) {  
 try {  
 return getSingleResult(session, criteriaQuery);  
 } catch (NoResultException e) {  
 return Optional.*empty*();  
 }  
 }  
  
 protected String buildContainsExpression(String value) {  
 return "%" + value + "%";  
 }  
  
 private <T> Optional<T> getSingleResult(Session session, CriteriaQuery<T> criteriaQuery) {  
 return Optional.*of*(session.createQuery(criteriaQuery).getSingleResult());  
 }  
  
 protected abstract Class<E> getDomainClassType();  
  
 protected SessionFactory getSessionFactory() {  
 return sessionFactory;  
 }  
}

1. HibernateSessionManagementInterceptor

@Component  
public class HibernateSessionManagementInterceptor implements HandlerInterceptor {  
  
 private final SessionFactory sessionFactory;  
  
 @Autowired  
 public HibernateSessionManagementInterceptor(SessionFactory sessionFactory) {  
 this.sessionFactory = sessionFactory;  
 }  
  
 @Override  
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
 DatabaseSessionUtil.*getCurrentOrOpen*(sessionFactory);  
 return true;  
 }  
  
 @Override  
 public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {  
 if (response.getStatus() == HttpStatus.*OK*.value()) {  
 DatabaseSessionUtil.*closeWithCommit*(sessionFactory);  
 return;  
 }  
 DatabaseSessionUtil.*closeWithRollback*(sessionFactory);  
 }  
}

1. Parser

public abstract class Parser {  
  
 private final ObjectMapper objectMapper = createMapper();  
  
 public <T> String serialize(T type) {  
 try {  
 return objectMapper.writeValueAsString(type);  
 } catch (JsonProcessingException e) {  
 throw new ParsingException(e.getMessage(), e);  
 }  
 }  
  
 public <T> T deserialize(String objectValue, Class<T> clazz) {  
 try {  
 return objectMapper.readValue(objectValue, clazz);  
 } catch (IOException e) {  
 throw new ParsingException(e.getMessage(), e);  
 }  
 }  
  
 public <T> T deserialize(String objectValue, TypeReference<T> typeReference) {  
 try {  
 return objectMapper.readValue(objectValue, typeReference);  
 } catch (IOException e) {  
 throw new ParsingException(e.getMessage(), e);  
 }  
 }  
  
 protected abstract ObjectMapper createMapper();  
}

1. NewestCoursesDailySender

@EnableAsync  
@Component  
public class NewestCoursesDailyEmailSender {  
  
 private final EmailClient emailClient;  
 private final UserDao userDao;  
 private final ApplicationConfiguration applicationConfiguration;  
  
 @Autowired  
 public NewestCoursesDailyEmailSender(EmailClient emailClient, UserDao userDao, ApplicationConfiguration applicationConfiguration) {  
 this.emailClient = emailClient;  
 this.userDao = userDao;  
 this.applicationConfiguration = applicationConfiguration;  
 }  
  
 @Scheduled(cron = *THREE\_AM\_EVERY\_DAY*)  
 @Async  
 public void sendDailyNewCourses() {  
 int page = 0;  
 List<User> userServiceModels = userDao.findUsersByPage(page, *MAX\_USERS\_BY\_PAGE*);  
 while (!userServiceModels.isEmpty()) {  
 sendDailyEmails(userServiceModels);  
 userServiceModels = userDao.findUsersByPage(++page, *MAX\_USERS\_BY\_PAGE*);  
 }  
 }  
  
 private void sendDailyEmails(List<User> users) {  
 for (User user : users) {  
 Email dailyEmail = new Email.Builder()  
 .setTitle(*CHECKOUT\_OUR\_NEW\_COURSE*)  
 .setContent(MessageFormat.*format*(*VISIT\_OUR\_SITE\_AND\_CHECK\_THE\_COURSES*, applicationConfiguration.getClientUrl()))  
 .setRecipient(user.getEmail())  
 .build();  
 emailClient.sendAsync(dailyEmail);  
 }  
 }  
}

1. SecurityJavaConfiguration

@Configuration  
@EnableWebSecurity  
@ComponentScan(basePackages = "com.code.to.learn")  
public class SecurityJavaConfiguration extends WebSecurityConfigurerAdapter {  
  
 private final PasswordEncoder passwordEncoder;  
 private final UserDetailsService userDetailsService;  
 private final ModelMapper modelMapper;  
 private final Parser parser;  
 private final SessionFactory sessionFactory;  
 private final UserService userService;  
  
 @Autowired  
 public SecurityJavaConfiguration(PasswordEncoder passwordEncoder, @Qualifier("userDetailsService") UserDetailsService userDetailsService,  
 ModelMapper modelMapper, Parser parser, SessionFactory sessionFactory, UserService userService) {  
 this.passwordEncoder = passwordEncoder;  
 this.userDetailsService = userDetailsService;  
 this.modelMapper = modelMapper;  
 this.parser = parser;  
 this.sessionFactory = sessionFactory;  
 this.userService = userService;  
 }  
  
 @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userDetailsService)  
 .passwordEncoder(passwordEncoder);  
 }  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 CharacterEncodingFilter encodingFilter = new CharacterEncodingFilter();  
 encodingFilter.setEncoding("UTF-8");  
 encodingFilter.setForceEncoding(true);  
  
 http.csrf()  
 .disable()  
 .cors()  
 .and()  
 .exceptionHandling()  
 .authenticationEntryPoint(getRestAuthenticationEntryPoint())  
 .and()  
 .authorizeRequests()  
 .antMatchers("/users/logout", "/github/\*\*", "/comments/add", "/comments/delete/\*\*",  
 "/courses/enroll/\*\*", "/courses/is-enrolled/\*\*", "/courses/cart/\*\*", "/users/user", "/courses/rate",  
 "/users/forgotten-password/\*\*", "/users/change-forgotten-password/\*\*", "/users/change-profile-picture/\*\*",  
 "/users/update/basic/\*\*", "/users/update/deactivate/\*\*", "/users/update/password/\*\*")  
 .authenticated()  
 .antMatchers("/api/admin/\*\*", "/users/change-roles/\*\*")  
 .hasRole(UserRole.*ROLE\_ADMIN*.getValue())  
 .and()  
 .authorizeRequests()  
 .antMatchers("/contact-us", "/courses/has-paid", "/course-categories/add", "/course-categories/update",  
 "/courses/unpaid", "/courses/add", "/courses/update", "/course/delete/\*\*")  
 .hasRole(UserRole.*ROLE\_MODERATOR*.getValue())  
 .and()  
 .formLogin()  
 .loginPage("/users/login")  
 .successHandler(getSuccessHandler())  
 .failureHandler(getFailureHandler())  
 .and()  
 .logout()  
 .invalidateHttpSession(true)  
 .permitAll()  
 .and()  
 .addFilterBefore(encodingFilter, CsrfFilter.class);  
 }

}

1. GenericExceptionHandler

@ControllerAdvice  
public class GenericExceptionHandler {  
  
 @ExceptionHandler({UserException.class, CourseException.class, GithubException.class, InvalidTokenException.class})  
 public ResponseEntity<ApiErrorResponse> userExceptionOccurred(Exception exception) {  
 ApiErrorResponse apiErrorResponse = new ApiErrorResponse.Builder()  
 .code(HttpStatus.*BAD\_REQUEST*.value())  
 .message(exception.getMessage())  
 .type(exception.getClass().getSimpleName())  
 .build();  
 return ResponseEntity.*badRequest*()  
 .header(*CONTENT\_TYPE*, MediaType.*APPLICATION\_JSON\_VALUE*)  
 .body(apiErrorResponse);  
 }  
  
}

# **Източници**

[1] Браян Хупи, Системи и методи за интеграция на сензори в термостати <https://patents.google.com/patent/US8727611>, 19.11.2010г.

[2] Агенция за защита на околната среда. Резюме на резултатите от изследванията от пазара на програмируеми термостати. Вашингтон, окръг Колумбия: Офис на централата, 2004 г

[3] Сакс. Програмируеми термостати. ACEEE, 2004 г

[4] Сервизни шампиони. 18.05.2018г <https://www.servicechampions.net/blog/manual-vs-programmable-smart-thermostats/>

[5] <https://americanhistory.si.edu/collections/search/object/nmah_1392743>

[6] Алън Мейер, <https://www.aceee.org/files/proceedings/2010/data/papers/1963>, 2010

[7] Джоусеф Лопез <https://www.aceee.org/files/proceedings/2010/data/papers/1953>

[8] Nest. <https://www.nest.com/support/article/>, 07.12.2018г.

[9] Лу, Дзяканг; Сукур, Тамим; Шринивасан, Виджай; Гао, Ге; Холбен, Брайън; Станкович, Джон; Филд, Ерик; Уайтхаус, Камин (2010). „Умният термостат“. Сборник доклади от 8-та конференция ACM за вградени мрежови сензорни системи - Sen Sys

[10] Ернандес, Грант; Ариас, Орландо; Буентело, Даниел; Джин, Йер (2014). <https://www.blackhat.com/docs/us-14/materials/us-14-Jin-Smart-Nest-Thermostat-A-Smart-Spy-In-Your-Home-WP.pdf>

[11] Общо отопление и климатизация. 2014-07-21 <https://genhvac.com/get-thermostat-geofencing-technology/>

[12] <https://www.ecobee.com/en-us/savings/>

[13] „Икономия на енергия от термостата Nest Learning: Резултати от анализа на сметките за енергия.“ Nest Labs, Inc., февруари 2015 г <https://storage.googleapis.com/nest-public-downloads/press/documents/energy-savings-white-paper.pdf>

[14] JS Овлажнители <https://web.archive.org/web/20091012112116/http://www.jshumidifiers.com/static-elimination-159-application/>

[15] Кларк, Джеф (27 март 2012 г.) <https://web.archive.org/web/20140718214253/http://www.datacenterjournal.com/facilities/humidity-in-the-data-center-do-we-still-need-to-sweat-it/>

[16] Уорд, Джералд В. Р. (2008). Енциклопедия на материалите и техниките на Grove в изкуството. Oxford University Press

[17] Център за устойчиви технологии на Earth Rangers <http://www.ercshowcase.com/hvac/earth-tubes/>

[18] Вараси, Джон (6 юни 2011 г.) <https://www.asme.org/topics-resources/content/global-cooling-the-history-of-air-conditioning>

[19] Грийн, Аманда (1 януари 2015 г.) <https://www.popularmechanics.com/home/a7951/history-of-air-conditioning/>

[20] Информация за Контролери за температура и влажност [https://www.globalspec.com/learnmore/manufacturing\_process\_equipment/process\_controllers/humidity\_controllers](https://www.globalspec.com/learnmore/manufacturing_process_equipment/process_controllers/humidity_controllersю)

[20] Официална документация на Spring framework <https://spring.io/blog/2004/03/24/spring-framework-1-0-final-released>

[21] Официална GitHub страница на Spring boot <https://github.com/spring-projects/spring-boot/releases/tag/v2.6.7>

[22] Официална документация на PostgreSQL <https://www.postgresql.org/>

[23] Официална документация на Hibernate <https://hibernate.org/orm/documentation/6.0/>

[24] Официална документация на Maven <https://maven.apache.org/guides/index.html>

[25] Официална документация на Tomcat <https://tomcat.apache.org/tomcat-8.5-doc/>

[26] Официална документация на React JS <https://reactjs.org/docs/create-a-new-react-app.html>

[27] Официална документация на Bootstrap <https://getbootstrap.com/docs/4.1/getting-started/introduction/>

[28] Официална документация на Fetch API <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>

[29] Официален сайт на BME 280 <https://www.bosch-sensortec.com/products/environmental-sensors/humidity-sensors-bme280/>

[30] Документация на библиотека за работа със сензор BME 280 <https://pypi.org/project/RPi.bme280/>

[31] Официална документация на Python Flask framework <https://flask.palletsprojects.com/en/2.1.x/>

[32] Официален сайт на Shelly Plug s <https://shelly.cloud/knowledge-base/devices/shelly-plug-s/>

[33] Raspberry GPIO статия <https://projects.raspberrypi.org/en/projects/physical-computing/1>

[34] Основи на I2C протокола <https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/>