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](#_Toc102584804)

[**1. Описание на проблемната област** 5](#_Toc102584805)

[**1.1. Най-известните уреди за измерване и промяна на температура и влажност** 5](#_Toc102584806)

[**А) Умен термостат** 5](#_Toc102584807)

[**Б) Овлажнител на въздух** 8](#_Toc102584808)

[**В) Климатик** 11](#_Toc102584809)

[**Г) Контролер на температура и влажност** 14](#_Toc102584810)

[**1.2. Каква е целта и мотивацията за създаването на нов продукт, който ще служи за регулиране на температура и влажност?** 14](#_Toc102584811)

[**1.3. Workflow диаграма на основното използване на продукта** 15](#_Toc102584812)

[**2. Проектиране на архитектура на системата** 16](#_Toc102584813)

[**2.1. Основните модели в системата (domain model)** 17](#_Toc102584814)

[**2.2. ER (entity relationship) диаграма** 22](#_Toc102584815)

[**2.3. Цялостна архитектура на приложението** 25](#_Toc102584816)

[**2.4. Анализ на входни данни на потребител, модератор и администратор.** 28](#_Toc102584817)

[**2.5. Използвани технологии** 32](#_Toc102584818)

[**3. Особености на програмната реализация** 40](#_Toc102584819)

[**3.1. Какво представлява клиентът?** 40](#_Toc102584820)

[**3.2. Какво представлява сървърът?** 44](#_Toc102584821)

[**3.3. Видове използвани релации** 56](#_Toc102584822)

[**3.4. Блокови схеми на основните операции** 58](#_Toc102584823)

[**А) Блокова схема на регистрация на потребител** 58](#_Toc102584824)

[**Б) Блокова схема на добавяне на курс** 58](#_Toc102584825)

[**В) Блокова схема на свързване на акаунт с GitHub** 59](#_Toc102584826)

[**Д) Блокова схема на регистрация на потребител** 59](#_Toc102584827)

[**4. Валидация на системата** 60](#_Toc102584828)

[**4.1. Unit тестове** 60](#_Toc102584829)

[**4.2. Ръчни тестове** 64](#_Toc102584830)

[**Заключение** 69](#_Toc102584831)

[**Код** 70](#_Toc102584832)

[**Източници** 84](#_Toc102584833)

# **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.

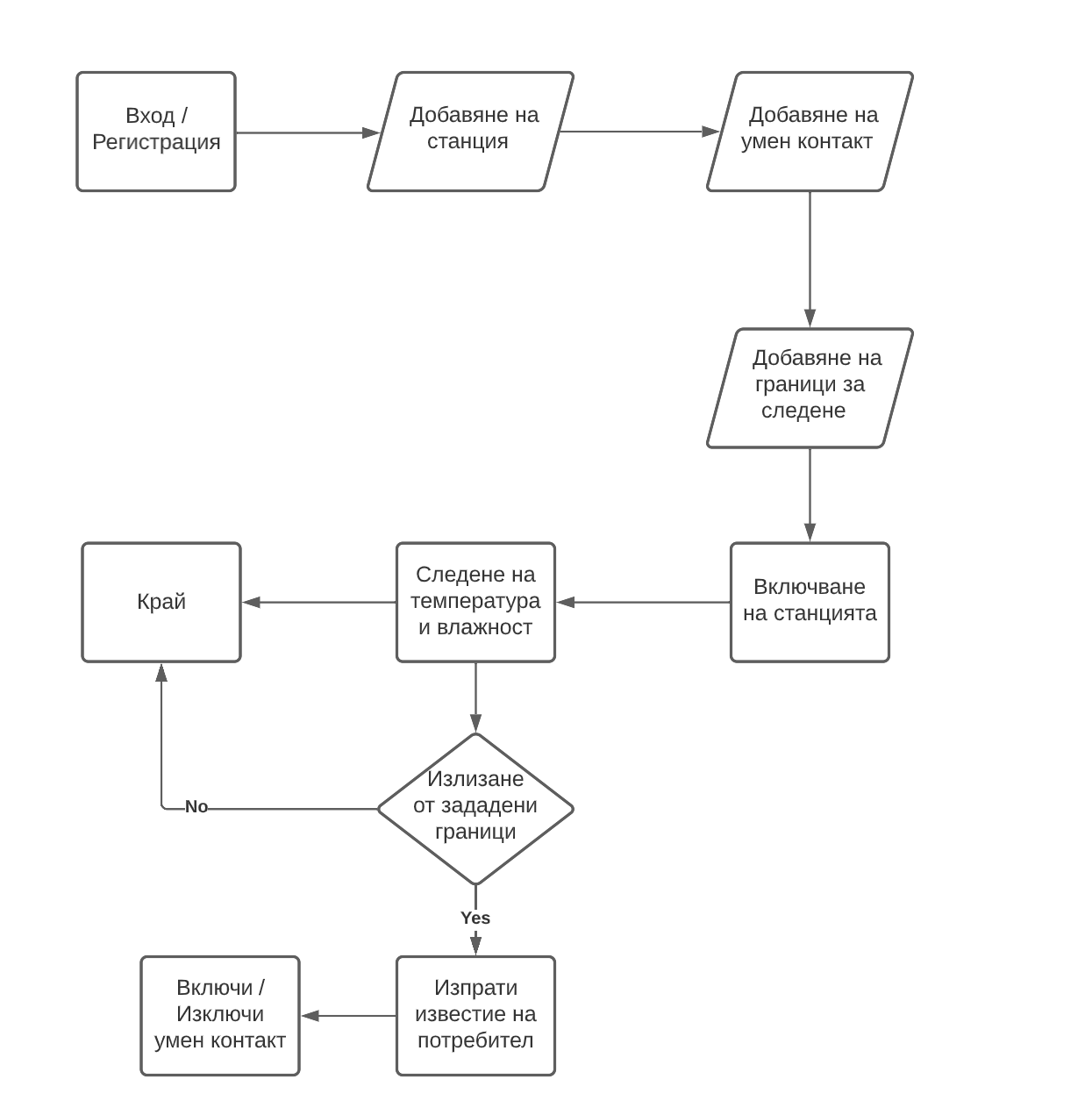
Друга цел на проекта да се предостави възможност на потребителите да следят графиката на температурата за всеки час, като средната температура във всеки един момент от денонощието е запазена.

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.

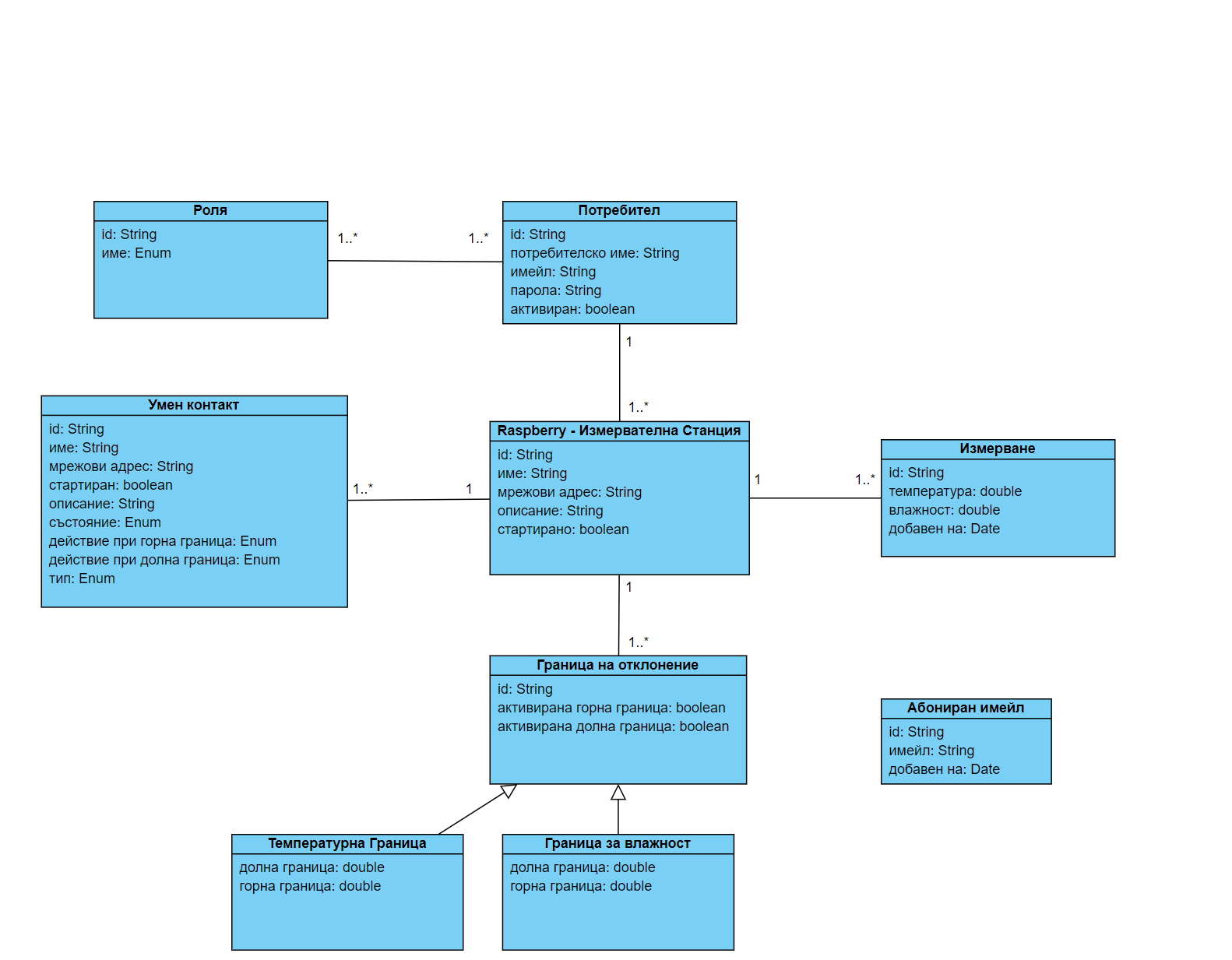
# **2. Проектиране на архитектура на системата**

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. В тази схема на базата данни не е използвана връзка едно-към-едно. 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:

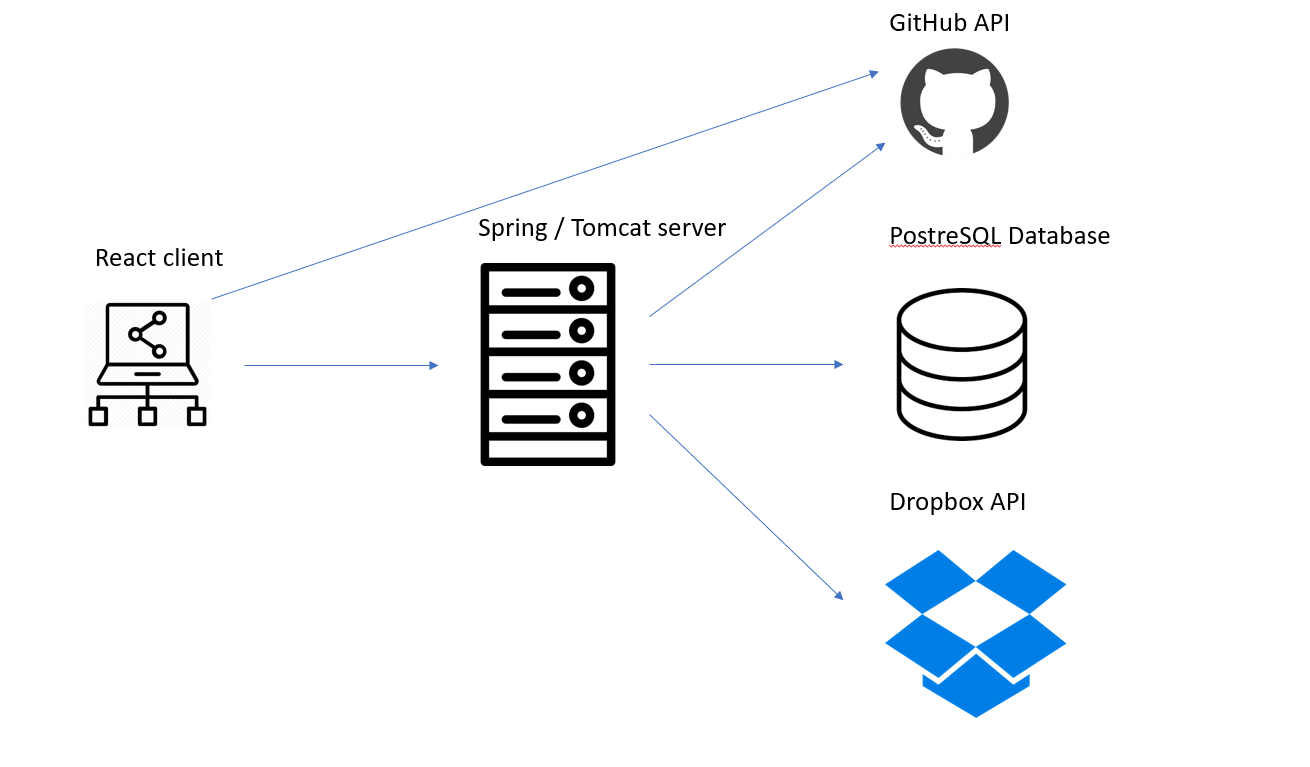
Картина, която съдържа маса

Описанието е генерирано автоматично

Фиг. 3 ER диаграма на продукта

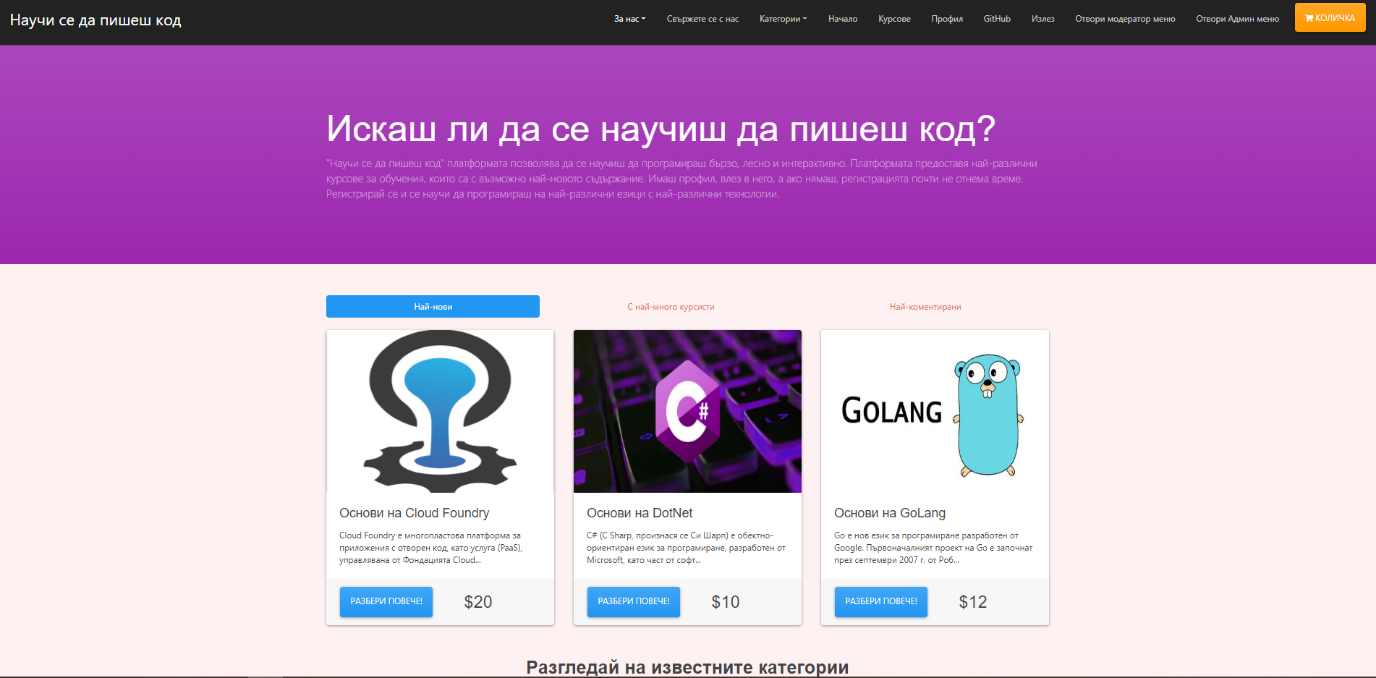
1. “users” – Таблицата пази информация за потребителите на приложението. Целта е да се съхранява информация за потребителя. Чувствителната информация, която в случая е паролата, не се пази в чист вид. Използва се хеширащ алгоритъм, който също така добавя и сол към паролата и следователно чак тогава се 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.

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



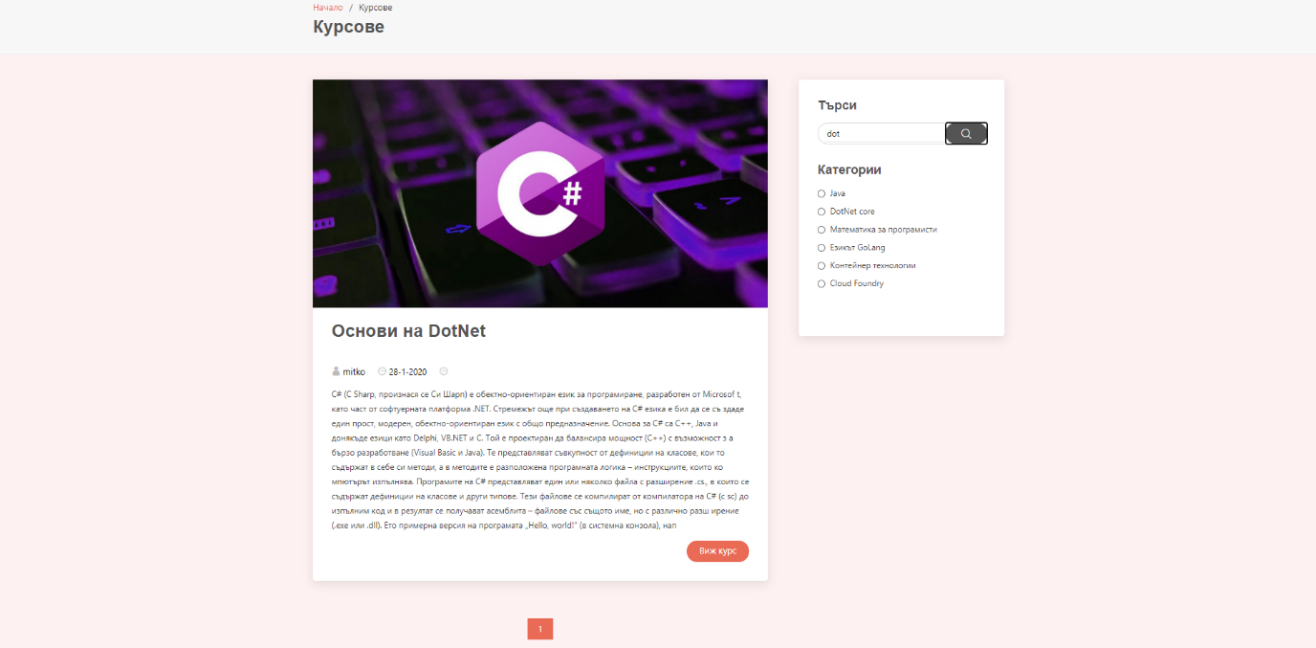
Фиг. 4 Архитектура на приложението

Потребителки интерфейс на някои от основните страници, с които потребителят най-често работи.



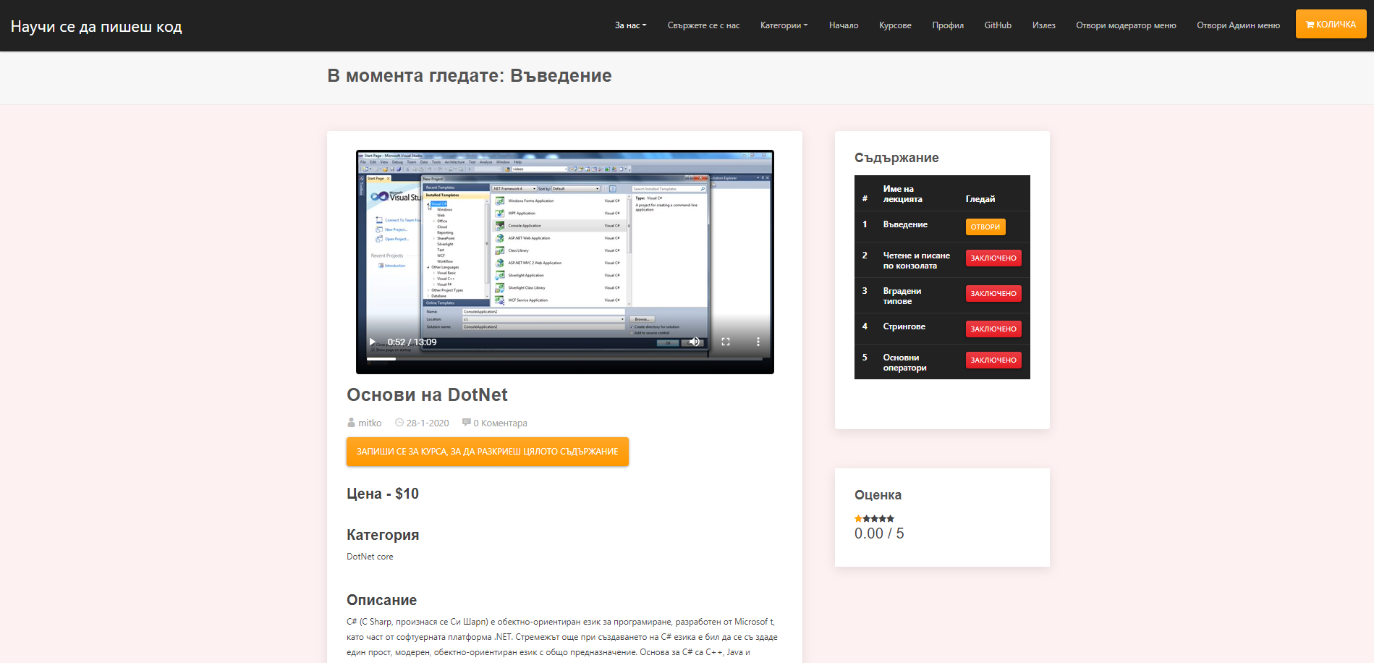
Фиг. 5 Начална страница на приложението

Това е началната страница на приложението. На нея се визуализират информация за уебсайта, топ курсовете, както и категориите, които има най-много курсове. Трите курса, които са визуализират могат да бъдат най-новите, с най-много записани курсисти или най-коментираните.

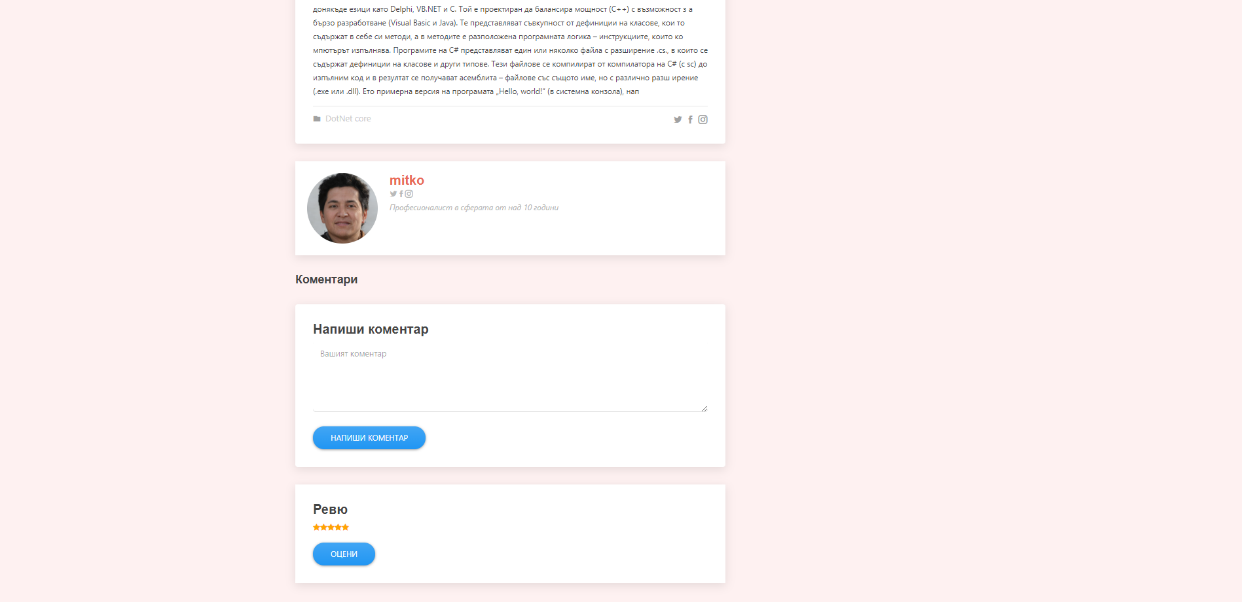


Фиг. 6 Страница за преглед на курс

В страницата за преглед на всички курсове потребителят има възможността разглежда наличните на сайта курсове. Също така има възможността да търси или да избира по определена категория. При опит за търсене се вземат всички курсове, които в заглавието си съдържат търсения текст без значения от това дали буквите са главни или малки. Селектирането по категория става като се избере категория и се натисне бутонът за търсене. Търсенето по категория и по текст могат да бъдат комбинира като по този начин влизат в сила и двата критерия.

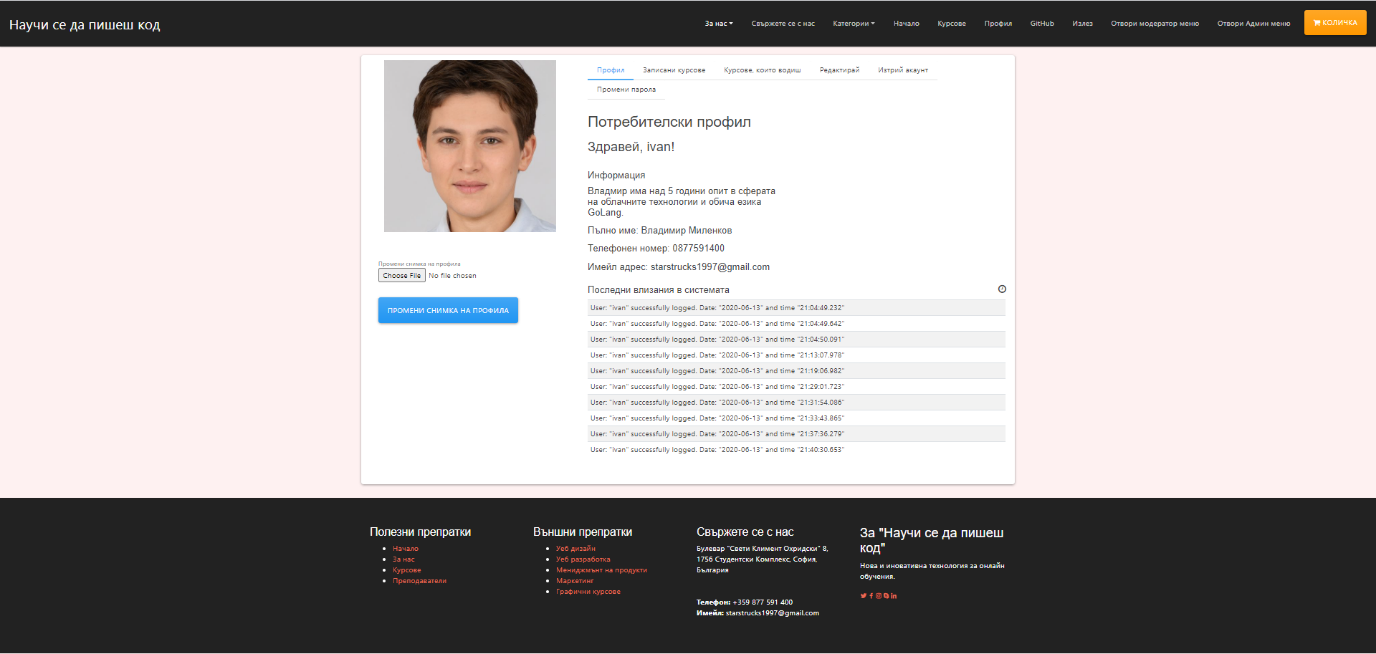


Фиг. 7 Страница за преглед на курс



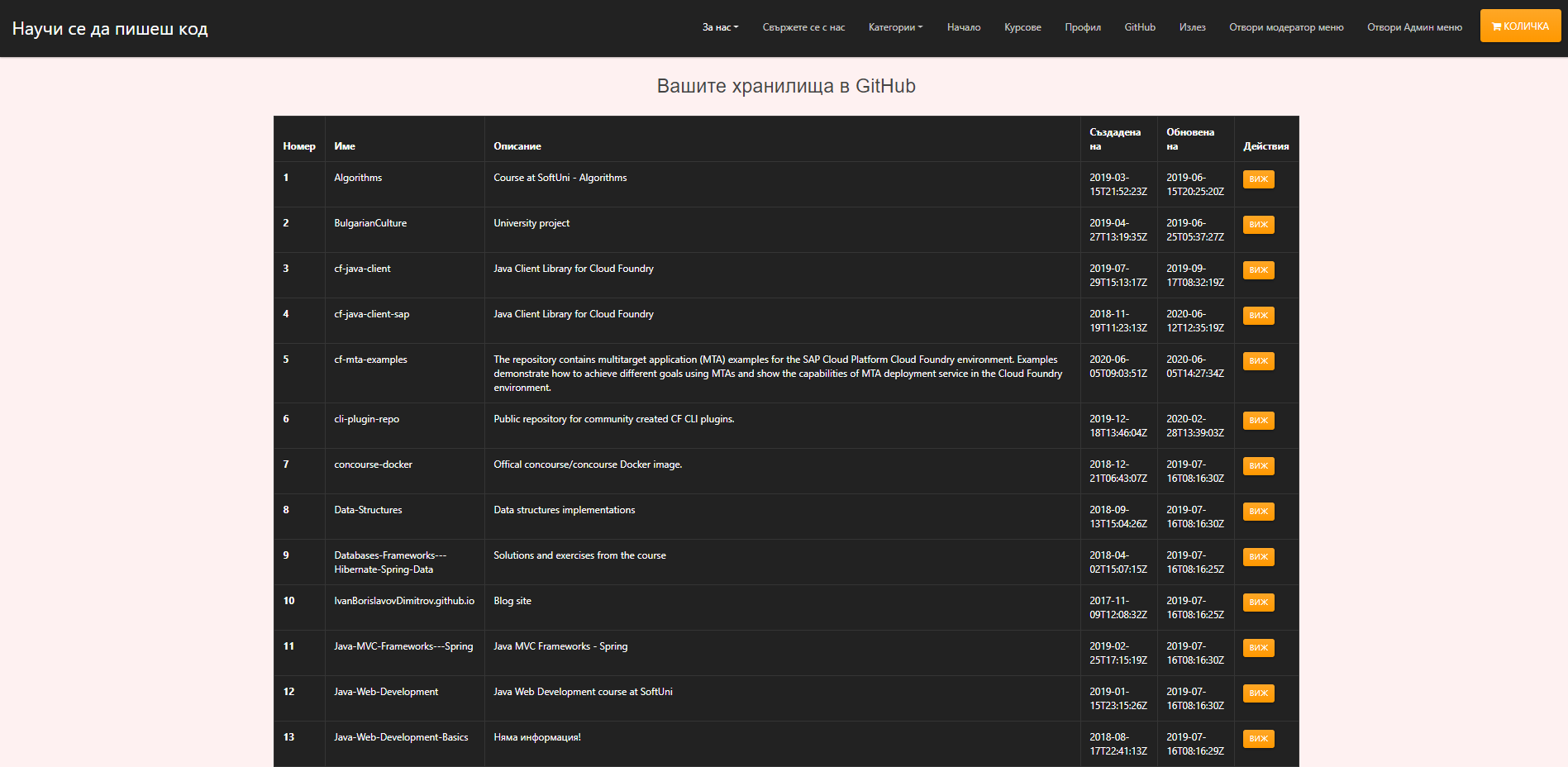
Фиг. 8 Продължение на страница за преглед на курс

Прегледът на определен курс може да бъде направен от всеки потребител без значение дали той е регистриран или не, дали се е записал или не, но има някои особеност, които трябва да се имат в предвид. Ако потребител не влязъл в профила си, той има достъп единствено до въвеждащото в курса видео. Това е направено с цел, потребител да може да погледна съдържанието на курса и да разбере за какво е той без да се е записал за него. Нерегистриран потребител няма право да коментира, оценява, да се записва за изпит или да добавя курс в количката си. Регистриран потребител има възможността да се запише за курс, да го добави в количката и да пише коментари. След като се запише за курс, потребителят получава привилегията да оценява курс, но все още съдържанието на курса не е достъпно. То става достъпно след като модератор одобри, че потребителят е заплатил съответната цена за курса.



Фиг. 9 Преглед на потребителски профил

Потребителят има възможността да проверява своя профил по всяко време. В профила си може да провери последните си 10 влизания като е посочено на коя дата и в кой час някой е влязъл в този профил. Също така може да се види на кои курсове е записан и кои курсове води, ако има такива. Също може да си подменя профилната снимка както името, фамилията, телефонния номер, паролата и т.н. Потребителят има ѝ правото да изтрива акаунта си.



Фиг. 10 Преглед на хранилищата на потребител

Потребител, който е влязъл в профил и е свързал профила си с GitHub може да разглежда своите хранилища. Те са изобразени в таблица и могат да бъдат преглеждани и поотделно, като така може да види допълнителна информация за всяко едно от тях. Целта на интеграцията с GitHub е да мотивира потребителя да си направи регистрация и да използва и GitHub тъй като това е най-известната технология за контрол на кода.

## **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.

1. Регистрация
2. Имейл – имейлът на потребителя, трябва да бъде валиден и да отговаря на правилата за валиден имейл. Той трябва също така да е уникален и да не е използван от друг потребител. След регистрация, потребителят трябва да влезе в своята поща с дадения имейл и да достъпи линкът изпратен от приложението, за да потвърди своята регистрация и да може да се автентикира по-късно.
3. Първо име – името на потребителя.
4. Фамилия – фамилията на потребителя.
5. Потребителски име – това е уникален псевдоним на потребителя и се използва при автентикация.
6. Описание – описание, което потребителят предоставя за себе си. Своите интереси и познания.
7. Телефонен номер – телефонният номер на потребителя. Той трябва е уникален, да не е използван от друг потребител и да отговаря на следните изисквания. Да започва с “+359” и след това да има 9 цифри или да започва с “0” и след това да има 9 цифри.
8. Дата на раждане – както се има предвид от полето дата на раждане
9. Парола – няма изисквания за дължината на паролата и за нейната сложност.
10. Потвърждаване на парола – трябва да съдържа същата стойност като горното поле.
11. Профилна снимка – ако потребителят желае може да качи собствена снимка, която се показва, когато някой друг потребител достъпва чужд профил.
12. Вход
13. Потребителско име – потребителското име въведено при регистрация
14. Парола – паролата, която потребителят е избрал при регистрация.
15. Свържете с нас (регистрация и вход не са задължителни)
16. Име – име на потребителят, който иска да запрати запитване или обратна връзка
17. Имейл – имейл на потребителя
18. Тема – тема на съобщението
19. Съобщения – съдържанието на съобщението
20. Курсове
21. Име на курс – името на курс, който потребител иска да намери. Търсенето работи на принципа, ако написаният текст се съдържа в заглавието на курса, то курса се визуализира.
22. Категория – “radio” бутони, използват се селектиране на определена категория. С натискане на символа за търсене се визуализират само курсовете от тази категория.
23. Единичен курс
24. Коментар – регистриран потребител може да оставя коментари върху курсовете.
25. Ревю – регистриран потребител дава оценката на курса като селектира звездички и натиска “оцени”.
26. Профил – може да се променят информация за потребители или да се променя паролата. Също така може да се изтрива акаунт. Полетата са сходни с тези при регистрация.

Модератор – има достъп до всички страници, които потребителят както към още няколко. В “модератор меню” има някои допълнителни функционалности.

1. Добавяне на курс
2. Име на курс – името на курса, кратко и ясно.
3. Дата на създаване – датата през, която е създаден курса.
4. Цена на курса
5. Описание – пълно и подробно описание на курса.
6. Категория – категорията на курса. Избира се някоя от създадените до момента.
7. Име на преподавател – името на човекът, който води. Той трябва задължително да е регистриран в сайта.
8. Части на курса – видеоклип и име на частта от курса. Може да са неопределен брой.
9. Изображение на курса – изображението, което се вижда, когато потребител разглежда курсовете.
10. Добавяне на категория за курс
11. Име на категорията
12. Описание на категорията
13. Изображение на категорията
14. Потвърждаване, че курсист е заплатил курс
15. С едно натискане на бутона потвърди се потвърждава, че даден курсист е записан за определен курс.

Администратор – има достъп до всички страници, които потребителят и модераторът имат както към още няколко. В “администратор меню” има някои допълнителни функционалности.

1. Промяна на роля на потребител
2. Добавяне на роли или премахване на роли за даден потребител
3. Изтрий потребител
4. Име на потребител за изтриване

Данните се съхраняват на две места. Данните, които представляват текстово съдържание се запазват в базата данни в таблиците показани по-горе. Всички останали файлове като изображения и видеоклипове се пазят в Dropbox, с цел надежден и бърз достъп до тях. За целта се използва специален клиент, който комуникира със техния сървър и качва, сваля и буферира в случай на гледане на видеоклип. Информация за имената на данните се държат в базата, за да могат да се вземат от Dropbox.

Потребителят може да навигира в приложението чрез горната лента на апликацията. Оттам могат да бъдат достъпвани почти всички основни страници в приложението.

Стартовият екран, който вижда всеки потребител, когато отвори приложението за пръв път вижда начален екран, който разполага с информативна информация за нивата и цените на курсовете, както и бутон, с който може да процедира към началната страница на приложението. В началната страници първоначално се показват последните 3 най-скоро добавените курсове. Потребителят може да погледне също така и 3-те курса с най-много записани курсисти и 3-те най-коментирани. След това са показани 6-те категории, които имат най-много свързани с тях курсове. Най-отдолу се намира панел с полезна информация и полезни линкове за бърза навигация в приложението. Потребителят може да види допълнителна информация за уебсайта в страницата „за нас“. Също в това меню може да се види информация за всички преподаватели в уебсайта. След като бъде отворена тази информация може да се отвори профилът на всеки от преподавателите. Там се съдържа частична информация за дадения потребител.

В страницата „свържете се с нас“ има информация подробна информация за сайта, както и карта с местоположението на седалище му. Също така има форма, която може да се попълни и изпрати към модераторите на уебсайта.

След това има плъзгащо се поле, в което се отварят всички категории и съответно при натискането на всяка една от тях се отваря страница само с курсове, които са от тази категория.

Следващото поле в горната лента са „курсове“, те отварят страницата с курсовете. В нея се визуализира съдържание с кратка информация за курсовете и навигация към пълната информация за тях, поле за търсене и избор на категория за търсене. Най-отдолу има страници тъй като броят на показваните курсове на страница е 3.

Потребителят може да вижда и променя информация за профила си от навигационното поле „Профил“. В него може да се види подробна информация за самия профил, последни 10 влизания в системата, както и промяна на снимката. Потребителят може да види за какви курсове е записан и какви курсове води, ако има такива. Може да променя информация за профила, както и да променя паролата или да изтрие профила си.

След курсовете следва „GitHub“ навигационният линк, който предоставя възможността потребителят да свърже своя акаунт с GitHub или да види своите хранилища. Също така може да види подробна информация за всяко едно от тези хранилища.

В зависимост от това дали влезлият потребител има роля модератор, той вижда или съответно не вижда линк към „модератор меню“. В него се съдържат навигационни линкове към: добавяне на курс, редактиране на курс, изтриване на курс, добавяне на категория, редактиране на категория, проверка за получени съобщения от потребители и потвърждаване на заплатен курс.

Следва, отново в зависимост от това дали потребителят притежава роля администратор. То предоставя възможността за промяна на роля на потребител и за изтриването на потребител.

Най-накрая следва бутонът количка, в който се пази информация за това кой потребител какви курсове е добавил в количката. Тя предоставя възможността за премахване на курсове от нея, както и за записването на всички курсове от нея.

Потребители, които не са влезли в профилите си виждат полета за вход и регистрация. Когато потребителят е забравил своята парола, той може да изиска подмяната с нова. Това става като си въведи потребителското име и следователно парола с изпраща на неговия имейл, който съдържа линк към страница за подмяна на паролата. Това става чрез генерирането на жетон, който се запазва в базата и се праща като част от линка от изпратения имейл. Следователно при смяна на паролата този жетон и се изпраща и ако съвпада със записания в базата паролата се подменя.

## **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”. [17]

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”. [4]

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. [6]

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. [26]

1. Dropbox е услуга за хостинг на файлове, управлявана от американската компания Dropbox, със седалище в Сан Франциско, Калифорния, която предлага облачно съхранение, синхронизация на файлове, личен облак и клиентски софтуер. Dropbox е основан през 2007 г. от студентите на MIТ като стартираща компания.

Dropbox е класиран като един от най-ценните стартъпи в САЩ и света с оценка от над 10 милиарда щатски долара и е описан като една от най-успешните инвестиции на Y Combinator до момента. Dropbox обаче също изпита критики и породи спорове по въпроси, включително нарушения на сигурността и опасения за поверителност.

Dropbox е блокиран в Китай от 2014 г. Има петзвездна поверителност от правителствения рейтинг от Фондация за електронни граници. [25]

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. [3]

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. [5]

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.

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. [7]

1. React – 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. [19]

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. [18]

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. [2]
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 (вижте Заявка).

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. [14]

# **3. Особености на програмната реализация**

За реализацията на платформата са използваните технологии посочени във втора точка. В основата си цялата платформа е разделена на две части: клиент и сървър.

## **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 34 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 работи на друг събдомейн. Събират се данни като, имена, телефонни номера, които се въвеждат от потребителите, при регистрация, при вход, при добавяне на курсове и т.н. Следователно те биват валидирани, ако данните не са коректни, например, липсва име при регистрация, или видео при добавяне на курс, 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 повторна, 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 някой от моментите. 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 домейн.

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. Пример за това може да се види още в началната страница, където могат да се избират топ 3 курсове по определен критерий. Тяхната смяна се извършва чрез динамична промяна на съдържанието на HTML-а.

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 да бъде установена сесия между клиента и сървъра се издава уникално ID от сървъра, което се запазва като т.н. бисквитки, които се пазят единствено в браузъра. При всяка заявка от клиента те се добавят към нея и така се удостоверява, че заявката идва от автентикиран потребител.

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

<Route exact path="/users/authorize" component={UserAuthorize} /> - При извикването на “/users/authorize” се взима код за жетон от GitHub и препраща към началната страница.

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

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

<Route exact path="/course-categories/add" component={AddCourseCategory} /> - При достъпването на “/course-categories/add” се отваря страницата за добавяне на нов курс.

<Route exact path="/admin/index" component={AdminUI} /> - При достъпването на “/admin/index” се отваря началната страница на администратор менюто, ако потребителят има достъп.

<Route exact path="/first/enter" component={FirstEnter} /> - Тази крайна точка “/first/enter” служи на потребителите, които за пръв път влизат в сайта, те автоматично са пренасочени към тази страница с цел по-обстойно запознаване със съдържанието на сайта.

<Route exact path="/admin/change-user-role" component={ChangeUserRole} /> - Този път “/admin/change-user-role” служи за отварянето на страницата за смяна на роля на потребител. Достъпна е само, ако потребителят притежава роля администратор.

<Route exact path="/admin/delete-user" component={DeleteUser} /> - При достъпването на “/admin/delete-user” се отваря страницата за изтриване на потребител. Страницата се отваря само ако потребителят има администратор роля.

<Route exact path="/moderator/index" component={ModeratorUI} /> - Тази страница “/moderator/index” служи за отварянето на началната страница на модератор менюто. Достъпна е само, ако потребителят има роля модератор или администратор.

<Route exact path="/courses/add-new-course" component={AddNewCourse} /> - Тази страница “/courses/add-new-course” представлява менюто за добавяне на нов курс.

<Route exact path="/courses" component={Courses} /> - Тази страница “/courses” служи за отварянето на всички курсове, разпределени в страници.

<Route path="/courses/view" component={ViewCourse} /> - Страницата “/courses/view” служи за отварянето на единичен курс. При нея е характерно подаването на следваща път променлива, която посочва името на курса.

<Route exact path="/courses/cart/finish-cart-purchase" component={FinishCartPurchase} /> - Страницата “/courses/car/finish-cart-purchase” служи за записването и заплащането на всички курсове, които потребителят е добавил в количката си.

<Route exact path="/course-categories/update" component={UpdateCourseCategory} /> - Страницата, която се достъпва на “/course-categories/update” служи за промяната на съществуваща категория.

<Route exact path="/github/check-repositories" component={CheckRepositories} /> - При достъпването на пътя “/github/check” се отваря страница със всички хранилища, които притежава потребителят, ако е свързал свой акаунт с уебсайта.

<Route path="/github/single-repository/" component={SingleRepository} /> - Страницата “/github/single-repository” служи за показването на данни за едно конкретно хранилище от GitHub.

<Route exact path="/information/team" component={Team} /> - При достъпа до “/information/team” може да бъде достъпена информация за преподаватели в уеб приложението.

<Route exact path="/information/about-us" component={AboutUs} /> - Страницата “/information/about-us” служи за достъпването на допълнителна информация относно уебсайта.

<Route exact path="/courses/update-course" component={UpdateCourse} /> - При достъп до “/courses/update-course” може да бъде достъпена страницата, която позволява редактиране на курс. Страницата може да бъде достъпена само от администратор или модератор.

<Route exact path="/courses/delete-course" component={DeleteCourse} /> - Достъпването на “/courses/delete-course” отвежда потребителя до страницата за изтриване на курс, ако потребителят притежава роля администратор или модератор.

<Route path="/comments/edit/" component={EditComment} /> - С достъпването на страницата “/comments/edit” може потребител да редактира коментар за определен курс, който той е написал.

<Route path="/comments/delete/" component={DeleteComment} /> - Страницата “/comments/delete” се използва за изтриване на коментар. Коментар може да бъде изтрит единствено от потребител написал коментара или от модератор или администратор.

<Route path="/users/activate/" component={ActivateAccount} /> - Страницата достъпвана на “/users/activate/” се използва за активирането на нов потребител.

<Route path="/users/new-password/" component={ChangePassword} /> - Страницата “/users/new-password” се използва за промяната на парола на потребителя.

<Route exact path="/users/profile/" component={UserProfile} /> - “/users/profile” представлява страницата за достъп до профила на потребител.

<Route exact path="/contact-us/" component={ContactUs} /> - С достъпването на страницата “/contact-us/” потребител може да изпрати съобщение до администраторите и до модераторите.

<Route exact path="/contact-us/show/" component={ContactUsForms} /> Когато “/contact-us/show/” страницата е достъпена се визуализира страницата с получени съобщения от горната форма. <Route exact path="/users/\*/profile" component={OtherUserProfile} /> - Страницата “/users/<USERNAME>/profile” служи за достъп до даден профил на потребител.

<Route exact path="/github/" component={GithubUI} /> - Когато “/github/” бъде достъпена се визуализира информация за функциите, с които разполага потребителят за установяване на връзка с GitHub.

<Route exact path="/courses/confirm-payment" component={ConfirmUserHasPaid} /> - Страницата “/courses/confirm-payment” може да бъде достъпвана само от модератори и администратори. В нея може да бъде потвърдено, че даден доказано е заплатил своя изпит.

<Route path="\*" component={NotFound} /> - Страницата “\*” означава всяка страница, която не е намерена от по-горните, която обозначава, че търсеният от потребителя адрес не съществува.

## **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, Dropbox). 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, сигурно така и да достъпва други сървъри с цел събиране на допълнителна информация. Той работи пряко с клиента, 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.

Сървърът предоставя така наречените “крайни точки”, които представлява местата, от които може да се вземе или прати информация. Данните се изпращат и приемат от сървъра като JSON форматиран текст или като бинарни файлове при видеоклиповете или изображенията.

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 курс се изисква името му 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.

Следват валидации дали името е свободно:

private void validateCourseName(String courseName) {

if (courseService.isNameTaken(courseName)) {

throw new CourseNameTakenException(courseName);

}

}

Това се извършва от тъй наречения “CourseService”, който се ползва като посредник на класовете, които директно комуникират с базата данни. Такъв клас е т.н. “CourseDaoImpl”, който отваря сесия с базата данни, ако такава не инициализирана, стартира транзакция, при която данните се записват в базата, когато тя бъде приключена успешно. Ако транзакцията пропадне, поради някаква причина, например са пропуснати валидации, при опит за техния запис се хвърля изключение, при което транзакцията се прекратява без да се записват данни, т.н. състояние “Rollback”. Това гарантира допълнителна сигурност при валидацията на данните, въпреки че е лоша практика да се оставят, валидации да се извършват от базата, тя служи като краен валидатор.

Сесията с базата данни се осъществява чрез т.н. “session per request approach”. Той представлява осъществяването на сесия с базата и се стартира транзакция при всяка заявка. При пристигане на заявката се отваря сесия и съответно преди връщането на отговор сесията и транзакцията. При върнат статус за грешка транзакцията се прекратява и никакви данни не се запазват в базата. За целта е използван “Interceptor”, който служи за менежирането на сесията и транзакцията. На места също се отварят и затварят сесии и транзакции, когато заявките до базата не са обвързани с заявки от клиента.

След като получените данни са успешно записани в базата с данни, следва записването на бинарните файлове като изображения и видеоклипове, в случая на добавянето на курс. Тези данни не се държат в самата база данни или локално на машината, тъй като това винаги крие рискове от тяхната загуба и води до допълнителна трудност за управлението им. Бинарните файлове се държат на облачна инфраструктура за данни наречена Dropbox. Тя предоставя безплатно 2GB място за данни, като повече се изисква заплащане. В момента уеб приложението работи с безплатната версия, като при нарастването на размера на данните ще се наложи закупуването на план.

За използването на Dropbox се използва клиент имплементиран от тях.

За да се използва първоначално трябва да бъде добавен в проекта. За целта се използва само Maven dependency и съответно Maven инструментът за автоматизация, автоматично сваля библиотеката и я добавя към проекта. Библиотеката се добавя в секцията с “dependencies”:

<dependency>

<groupId>com.dropbox.core</groupId>

<artifactId>dropbox-core-sdk</artifactId>

<version>${dropbox.client.version}</version>

</dependency>

Добавянето на този код е достатъчно за свалянето на библиотеката. За инициализирането на Dropbox клиент са необходими следните две неща. Client identifier – представлява името на проекта в Dropbox и Access token – жетон за достъп до Dropbox, който се издава от техния сайт и е свързан с уникален потребител. Той няма определен живот и може да бъде използван за неопределено време.

Качването на файл става по следния начин:

@Override

public FileMetadata uploadFile(InputStream inputStream, String filename) {

try {

return client.files().uploadBuilder(insertFrontSlash(filename))

.withMode(WriteMode.*OVERWRITE*)

.uploadAndFinish(inputStream);

} catch (IOException | DbxException e) {

throw new LCException(e.getMessage(), e);

}

}

Първоначално се извиква методът “files()”, който връща нов клиент само за работа с данни. След това се извиква методът “uploadBuilder”, който връща обект, на който се подават име на файл, вид на запис на данните и накрая се извиква “uploadAndFinish” и следователно файлът се качва на сървъра на Dropbox.

За вземането на файл се използва следният подход

@Override

public InputStream getFileAsInputStream(String filename, Long offset) {

try {

return client.files()

.downloadBuilder(insertFrontSlash(filename))

.range(offset)

.start()

.getInputStream();

} catch (DbxException e) {

throw new LCException(e.getMessage(), e);

}

}

Свалянето се извършва като се взима “InputStream” на файла. Това представлява поредицата от байтове на файла като тя се чете динамично. При четенето на изображенията тя се изчита напълно и се връща байт масив със съдържанието на изображението. При четенето на видеоклип се връща стрийм от видеоклипа, който се чете и визуализира едновременно. Методът “range” е използван само заради видеоклиповете и представлява начина за превъртана на видео, или с други думи откъде да започне четенето на файла, тъй като понякога това може да не е началото. Тази стойност се изпраща от браузъра, като тя зависи на общия размер на файла в байтове. Той се изпраща при първия достъп до видеоклипът, като част от HTTP заглавието “Content-Range”.

Сървърът също така предоставя връзка и с GitHub, където при съгласието на клиента GitHub може да издаде жетон, с който сървърът може да достъпва информация за потребителя и да я връща. За целта с използва “OAuth” автентикация. Тя се осъществява, като първоначално клиентът извиква GitHub URL, който изисква потвърждение от потребителя, че е съгласен негова информация да бъде достъпвана от другаде. След като се съгласи, GitHub пренасочва заявката към зададен URL, в случая той е на клиента с определен уникално издаден на потребителя код, който е добавен като параметър в URL-а. След това клиентът изпраща този код до сървъра, който прави нова заявка до GitHub като подава три задължителни параметъра: clientId, clientSecret, code. Това се извършва по следния начин в кода:

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;

}

Използва се Apache HTTP клиентът за изпращането на заявки. След като се върне в отговора жетон за достъп до потребител той се записва в базата за дадения потребител.

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”
2. Регистрация на потребител:

@PostMapping(value = "/register", consumes = MediaType.*MULTIPART\_FORM\_DATA\_VALUE*)

public ResponseEntity<UserResponseModel> register(@Valid UserBindingModel userBindingModel) {

return userServiceApi.register(userBindingModel);

}

1. Вземане на потребител по име

@GetMapping(value = "/{username}")

public ResponseEntity<UserResponseModel> getUser(@PathVariable String username) {

return userServiceApi.findUser(username);

}

1. Активиране на профил на потребител

@PostMapping(value = "/activate/{username}")

public ResponseEntity<UserResponseModel> activateAccount(@PathVariable String username) {

return userServiceApi.activateAccount(username);

}

1. Активиране на профил на потребител

@PostMapping(value = "/activate/{username}")

public ResponseEntity<UserResponseModel> activateAccount(@PathVariable String username) {

return userServiceApi.activateAccount(username);

}

1. Смяна на профилната снимка

@PostMapping(value = "/change-profile-picture/{username}", consumes = MediaType.*MULTIPART\_FORM\_DATA\_VALUE*)

public ResponseEntity<UserResponseModel> changeProfilePicture(@PathVariable String username, @RequestParam MultipartFile profilePicture) {

return userServiceApi.updateProfilePicture(username, profilePicture);

}

1. Деактивиране на профил

@PostMapping(value = "/update/deactivate")

public ResponseEntity<UserResponseModel> deactivateCurrentUser(HttpSession httpSession) {

String loggedUserUsername = usernameGetter.getLoggedInUserUsername();

ResponseEntity<UserResponseModel> userResponseModelResponseEntity = userServiceApi.deactivateProfile(loggedUserUsername);

httpSession.invalidate();

return userResponseModelResponseEntity;

}

1. Промяна на парола

@PatchMapping(value = "/update/password", consumes = MediaType.*APPLICATION\_JSON\_VALUE*)

public ResponseEntity<?> changeUserPassword(@RequestBody @Valid UserPasswordChangeBindingModel userPasswordChangeBindingModel) {

String loggedUserUsername = usernameGetter.getLoggedInUserUsername();

userPasswordChangeBindingModel.setUsername(loggedUserUsername);

return userServiceApi.changeUserPassword(userPasswordChangeBindingModel);

}

1. Курсове – всички пътища започват с “/courses”
2. Добавяне на курс

@PostMapping(consumes = MediaType.*MULTIPART\_FORM\_DATA\_VALUE*)

public ResponseEntity<CourseResponseModel> addCourse(@Valid CourseBindingModel courseBindingModel) {

return courseServiceApi.add(courseBindingModel);

}

1. Взимане на курс по зададени критерии

@GetMapping

public ResponseEntity<List<CourseResponseModel>> getCourses(@RequestParam(defaultValue = "1") int page,

@RequestParam(required = false) String name,

@RequestParam(required = false) String category) {

return courseServiceApi.getByPageNameCategory(page - 1, name, category);

}

1. Взимане на курс по име

@GetMapping(value = "/{courseName}")

public ResponseEntity<CourseResponseModel> getCourse(@PathVariable String courseName) {

return courseServiceApi.get(courseName);

}

1. Записване на потребител за курс

@PostMapping(value = "/enroll/{courseName}", consumes = MediaType.*APPLICATION\_JSON\_VALUE*)

private ResponseEntity<CourseResponseModel> enrollUserForCourse(@PathVariable String courseName) {

String loggedUser = usernameGetter.getLoggedInUserUsername();

return courseServiceApi.enrollUserForCourse(loggedUser, courseName);

}

1. Проверка дали потребител е записан за курс

@GetMapping(value = "/is-enrolled/{courseEnrolledFor}")

public ResponseEntity<UserEnrolledForCourse> isLoggedUserEnrolledForCourse(@PathVariable String courseEnrolledFor) {

String loggedUser = usernameGetter.getLoggedInUserUsername();

return courseServiceApi.isUserEnrolledForCourse(loggedUser, courseEnrolledFor);

}

1. Проверка дали потребител е заплатил за курс

@GetMapping(value = "/has-paid/{courseName}")

public ResponseEntity<UserPaidForCourse> hasUserPaidForCourse(@PathVariable String courseName) {

String loggedUser = usernameGetter.getLoggedInUserUsername();

return courseServiceApi.hasUserPaidForCourse(loggedUser, courseName);

}

1. Потвърждаване, че потребител е заплатил за курс

@GetMapping(value = "/has-paid/{courseName}")

public ResponseEntity<UserPaidForCourse> hasUserPaidForCourse(@PathVariable String courseName) {

String loggedUser = usernameGetter.getLoggedInUserUsername();

return courseServiceApi.hasUserPaidForCourse(loggedUser, courseName);

}

1. Вземане на всички потребители, които има неплатени курсове

@GetMapping(value = "/unpaid")

public ResponseEntity<List<UnpaidCourseResponseModel>> getCoursesThatAreStillNotPaid() {

return courseServiceApi.getCoursesThatAreStillUnpaid();

}

1. Добавяне на курс в количката

@PostMapping(value = "/cart/add/{courseName}")

public ResponseEntity<CourseResponseModel> addToCart(@PathVariable String courseName) {

String loggedUser = usernameGetter.getLoggedInUserUsername();

return courseServiceApi.addToCart(loggedUser, courseName);

}

1. Показване на курсове в количката

@GetMapping(value = "/cart/all")

public ResponseEntity<List<CourseResponseModel>> getCoursesInCart() {

String loggedUser = usernameGetter.getLoggedInUserUsername();

return courseServiceApi.getCoursesInCart(loggedUser);

}

1. Премахване на курс от количката

@PostMapping(value = "/cart/enroll-from-cart")

public ResponseEntity<List<CourseResponseModel>> enrollForCourses() {

String loggedUser = usernameGetter.getLoggedInUserUsername();

return courseServiceApi.enrollFromCart(loggedUser);

}

1. Изтриване на курс

@DeleteMapping(value = "/delete/{courseName}")

public ResponseEntity<CourseResponseModel> deleteCourse(@PathVariable String courseName) {

return courseServiceApi.deleteCourse(courseName);

}

1. Оценяване на курс

@PostMapping(value = "/rate")

public ResponseEntity<CourseResponseModel> rateCourse(@RequestBody @Valid CourseRatingBindingModel courseRatingBindingModel) {

String loggedUserUsername = usernameGetter.getLoggedInUserUsername();

return courseServiceApi.rateCourse(courseRatingBindingModel, loggedUserUsername);

}

1. Коментари – всички пътища започват с “/comments”
2. Добавяне на коментар

@PostMapping(value = "/add")

public ResponseEntity<CommentResponseModel> add(@RequestBody @Valid CommentBindingModel commentBindingModel) {

commentBindingModel.setAuthor(usernameGetter.getLoggedInUserUsername());

return commentServiceApi.add(commentBindingModel);

}

1. Взимане на коментари по име на курс

@GetMapping(value = "/all")

public ResponseEntity<List<CommentResponseModel>> getCommentsByCourseName(@RequestParam String courseName) {

return commentServiceApi.getCommentsByCourseName(courseName);

}

1. Редактиране на коментар

@PutMapping(value = "/update")

public ResponseEntity<CommentResponseModel> edit(@RequestBody @Valid CommentBindingModel commentBindingModel) {

String loggedUser = usernameGetter.getLoggedInUserUsername();

return commentServiceApi.update(commentBindingModel, loggedUser);

}

1. Изтриване на коментар

@DeleteMapping(value = "/delete/{commentId}")

public ResponseEntity<CommentResponseModel> delete(@PathVariable String commentId) {

String loggedUser = usernameGetter.getLoggedInUserUsername();

return commentServiceApi.delete(commentId, loggedUser);

}

1. Взимане на коментар по ID номер

@GetMapping(value = "/{commentId}")

public ResponseEntity<CommentResponseModel> get(@PathVariable String commentId) {

return commentServiceApi.get(commentId);

}

1. Свържете се с нас – всички пътища започват с “/contact-us”
2. Добавяне на форма

@PostMapping(consumes = MediaType.*APPLICATION\_JSON\_VALUE*)

public ResponseEntity<ContactUsResponseModel> add(@RequestBody @Valid ContactUsBindingModel contactUsBindingModel) {

return contactUsServiceApi.add(contactUsBindingModel);

}

1. Четене на всички форми

@GetMapping

public ResponseEntity<List<ContactUsResponseModel>> getAll() {

return contactUsServiceApi.getAll();

}

1. Категории – всички пътища започват с “/course-categories”
2. Добавяне на категория

@PostMapping(consumes = MediaType.*MULTIPART\_FORM\_DATA\_VALUE*)

public ResponseEntity<CourseCategoryResponseModel> add(@Valid CourseCategoryBindingModel courseCategoryBindingModel) {

return courseCategoryServiceApi.add(courseCategoryBindingModel);

}

1. Взимане на топ 6 категории с най-много курсове

@GetMapping(value = "/top")

public ResponseEntity<List<CourseCategoryResponseModel>> getCategoriesWithMostCourses(@RequestParam(defaultValue = "6") int limit) {

return courseCategoryServiceApi.getCategoriesWithMostCourses(limit);

}

1. GitHub – всички пътища започват с “/github”
2. Взимане на жетон за потребител

@PostMapping(value = "/authorize")

public ResponseEntity<GithubAccessTokenResponseModel> authorizeUser(@RequestParam String code) {

return githubServiceApi.requestAccessTokenForUser(usernameGetter.getLoggedInUserUsername(), code);

}

1. Взимане на всички хранилища за потребител

@PostMapping(value = "/authorize")

public ResponseEntity<GithubAccessTokenResponseModel> authorizeUser(@RequestParam String code) {

return githubServiceApi.requestAccessTokenForUser(usernameGetter.getLoggedInUserUsername(), code);

}

1. Ресурси – всички пътища започват с “/resource”
2. Взимане на изображение

@GetMapping(value = "/images/{name:.+}", produces = MediaType.*IMAGE\_JPEG\_VALUE*)

public @ResponseBody

byte[] getImageResource(@PathVariable String name) {

return resourceServiceApi.getImageResource(name);

}

1. Възпроизвеждане на видеоклип

@GetMapping(value = "/videos/{courseName}/{name:.\*}", produces = "video/mp4")

public CompletableFuture<StreamingResponseBody> getVideoResource(@PathVariable String courseName,

@PathVariable String name,

HttpServletRequest request,

HttpServletResponse 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(length = 100, nullable = false)  
private String firstName;

Това е примерен код за маркирането на полето “firstName” като колона в таблицата за потребители. С “length” се означава максималната дължина, която може да съдържа ред от колоната, а с “nullable = false” се забранява полето да има стойност “null”.

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

@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**

Релациите между връзки отново са реализирани чрез помощта на допълнителната библиотека наречена Hibernate. Пример за връзка едно-към-едно:

@OneToOne(cascade = {CascadeType.*MERGE*, CascadeType.*PERSIST*, CascadeType.*REFRESH*, CascadeType.*DETACH*, CascadeType.*MERGE*}, fetch = FetchType.*EAGER*)  
@JoinColumn(name = "github\_access\_token\_id", referencedColumnName = IdEntity.*ID*)  
private GithubAccessToken githubAccessToken;

Такава връзка е потребител и GitHub жетон. С “@OneToOne” се означава такава релация. С ключовата дума “cascade” се означава, кои операции могат да бъдат извършвани каскадно. С “fetch” се означава как да се зареждат данните от релацията. В случая е “EAGER”, което означава, че още при взимането на ред от таблицата веднага се извършва “join” с другата таблица. С анотацията “@JoinColumn” се описват имената на колоната от текущата таблица и от другата таблица.

Едно-към-много връзка е реализирана при преподаватели и курсове, тъй като един преподавател може да води много курсове, но един курс може да се води само от един преподавател. Пример:

@OneToMany(targetEntity = Course.class, fetch = FetchType.*LAZY*, mappedBy = "teacher", cascade = CascadeType.*MERGE*)  
private List<Course> coursesThatTeaches = new ArrayList<>();

“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 “LAZY”, което означава, че при вземането на класа не се “join” с другата таблица. Той се извършва единствено, ако бъде извикан методът, който връща това поле. Тогава динамично се извършва “join” и се връщат резултатите.

Връзка много-към-много е използвана за потребители и курсовете в количката, които те имат. Връзката е много-към-много, тъй като много потребители могат да искат да се запишат за един и същи курс и един курс може да име повече от един записани курсисти. Тази връзка се изразява чрез “@ManyToMany” анотацията. Пример:

@ManyToMany(mappedBy = "futureAttendants", targetEntity = Course.class, fetch = FetchType.*LAZY*, cascade = CascadeType.*MERGE*)  
private List<Course> coursesInCart = new ArrayList<>();

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

@ManyToMany(targetEntity = User.class, fetch = FetchType.*LAZY*, cascade = CascadeType.*MERGE*)  
@JoinTable(name = "future\_attendants\_courses", joinColumns = @JoinColumn(name = "course\_id", referencedColumnName = "id"),  
 inverseJoinColumns = @JoinColumn(name = "future\_attendants\_id", referencedColumnName = "id"))  
private List<User> futureAttendants = new ArrayList<>();

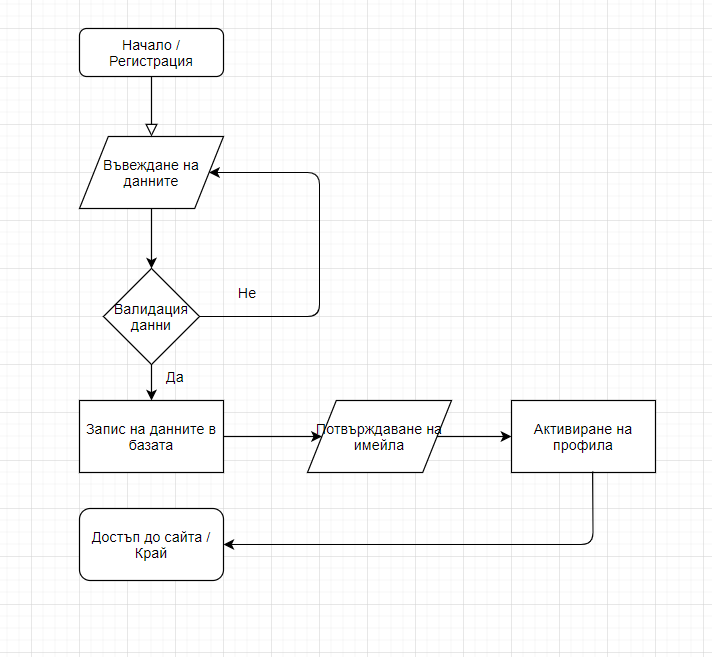
В един от двата класа трябва да се посочи името на допълнителната таблица, както и нейните колони и по кои колони се извършват “join”-овете. Анотацията “@JoinTable” е използвана в случая.

Има още доста места, на които се използват релации, които имат много сходна структура на по-горе посочените три. Те могат са видят в ER диаграмата в точка 2.

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.

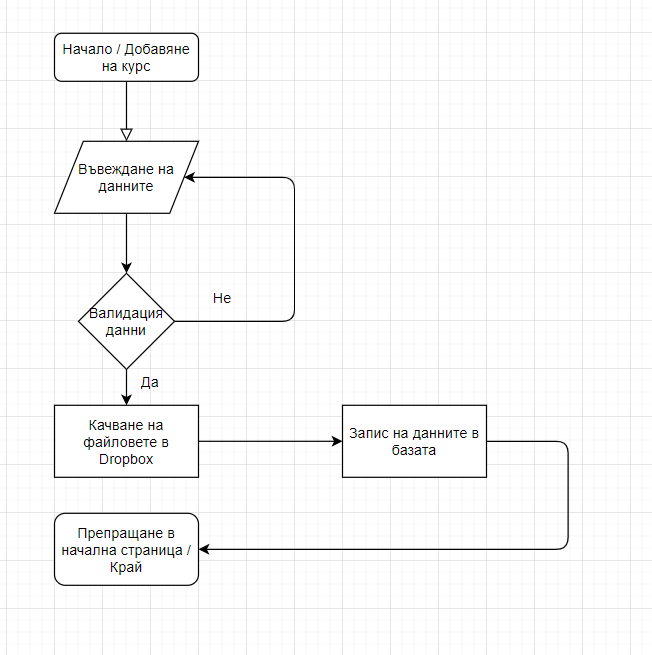
## **3.4. Блокови схеми на основните операции**

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



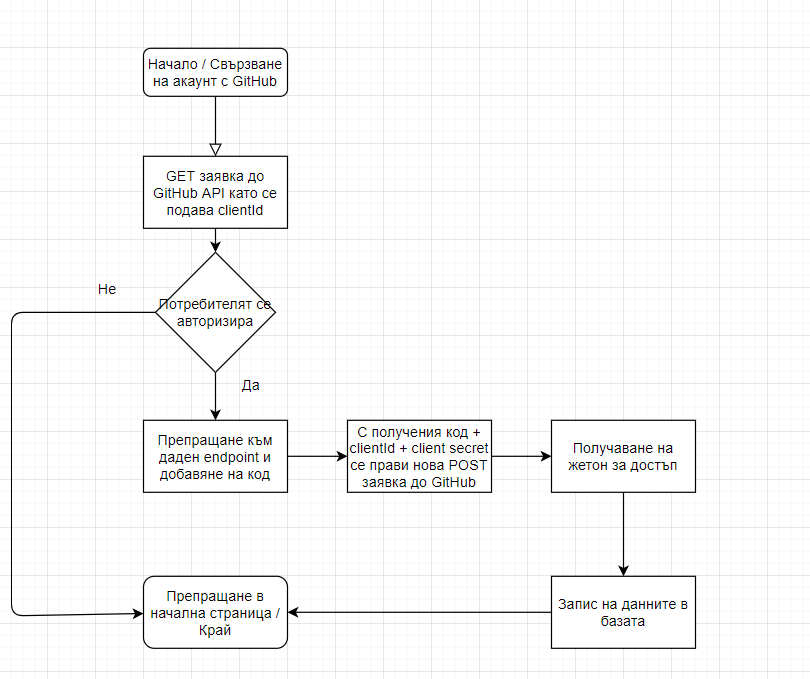
Фиг 11. Блокова схема – регистрация

### **Б) Блокова схема на добавяне на курс**



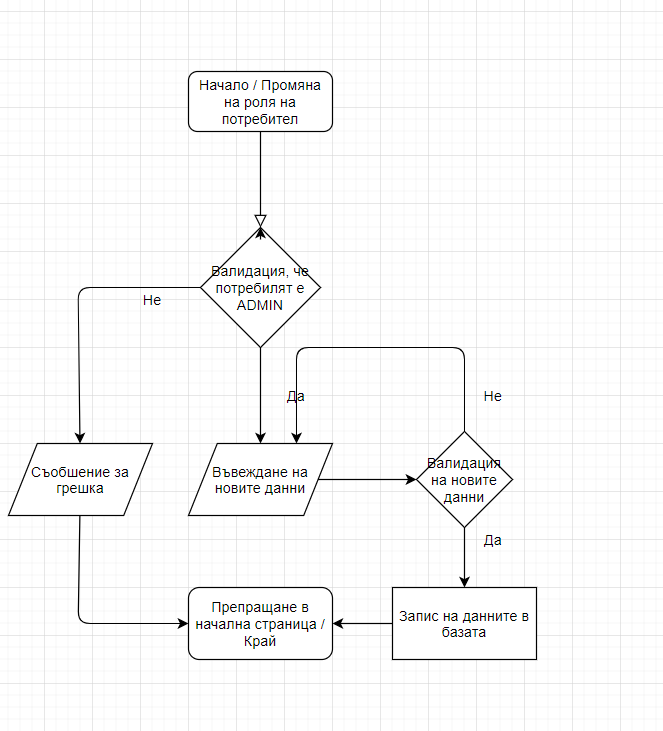
Фиг 12. Блокова схема – добавяне на курс

### **В) Блокова схема на свързване на акаунт с GitHub**



Фиг 13. Блокова схема – свързване акаунт с GitHub

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



Фиг 14. 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. CommentService

@Test

public void testFindCommentsByCourseName() {…} – Тества се дали, са върнати правилните коментари, по определено име на курс.

1. CourseCategoryService

@Test

public void testFindCategoriesWithMostCourses() {…} – Тества дали се връщат категориите с най-много курсове.

1. CourseService

@Test

public void testIsNameTaken() {…} – Тества се дали подаденото име на курс вече е заето от някой друг курс.

@Test

public void testFindCourses() {…} – Тества дали се връщат всички курсове по зададени критерии.

@Test

public void testCountByNameLike() {…} – Тества се дали са върнати точният брой записи, които имат име което прилича на зададен тест.

@Test

public void testEnrollUserForCourse() {…} – Тества дали записването на потребител за курс работи правилно.

@Test

public void testRemoveCourseFromCart() {…} – Тества се дали курс се премахва успешно от количката.

@Test

public void testEmptyCart() {…} – Тества се дали количката се изпразва изцяло, когато потребителят заяви записване на всички курсове от нея.

@Test

public void testFindMostCommented() {…} – Тества дали се връщат най-коментираните курсове.

@Test

public void testFindBestSellers() {…} – Тества се дали се намират курсовете, които имат най-много записани участници (те се считат за най-продаваните).

@Test

public void testMarkCourseAsRatedByUser() {…} – Тества се дали курс се маркира успешно оценен, когато потребител го оцени.

@Test

public void testHasUserRatedCourse() {…} – Тества се дали курс е оценен от потребител.

1. GenericService (Базов клас, който всеки service наследява)

@Test

public void testFindAll() {…} – Тества се дали се връщат всички обекти.

@Test

public void testFindById() {…} – Тества се дали даден обект е намерен при определен идентификационен номер.

@Test

public void testSave() {…} – Тества се дали даден обект се записва успешно в базата.

@Test

public void testDeleteById() {…} – Тества се дали даден обект се изтрива успешно при подаден идентификационен номер.

@Test

public void testDelete() {…} – Тества се дали даден обект се изтрива, когато е подаден.

@Test

public void testMerge() {…} – Тества се дали даден обект се слива успешно с друг.

@Test

public void testCount() {…} – Тества се дали се връща точният брой обекти.

1. NamedElementService (Базов клас, който всеки service наследява, ако се нуждае от функционалност за търсене по име)

@Test

public void testGetByName() {…} – Тества се дали се връща правилният обект по дадено име.

1. RoleService

@Test  
public void testGetByName() {…} – Тества се дали се връща правилната роля при подадено име на ролята.

1. UserService

@Test  
public void testRegisterUser() {…} – Тества се дали регистрацията на потребител е успешна. Прави се проверка, че всички полета са вярно попълнени.

@Test  
public void testIsUsernameTaken() {…} – Тества се дали даденото потребителско име е заето.

@Test  
public void testIsEmailTaken() {…} – Тества се дали даденият имейл е зает.

@Test  
public void testIsPhoneNumberTaken() {…} – Тества се дали е зает даденият телефонен номер на потребителя.

@Test  
public void testFindByUsername() {…} – Тества се дали правилно се намира потребителят по неговото потребителско име.

@Test  
public void testFindUsersCount() {…} – Тества са дали се намира правилният брой на потребителите.

@Test  
public void testFindUsersByUsernameContaining() {…} – Тества се дали се връщат правилно всички потребители, които имат потребителско име, което съдържа определен текст.

@Test  
public void testFindTeachers() {…} – Тества се дали се връщат правилно всички потребители, които водят поне един курс (това се приема за преподавател).

@Test  
public void testActivateAccount() {…} – Тества се дали акаунт на потребител се активира правилно и акаунтът се отключва.

@Test  
public void testGenerateResetPasswordToken() {…} – Тества се дали се издава и запазва в базата жетон за забравена парола на потребител.

@Test  
public void testChangeForgottenPassword() {…} – Тества се дали успешно се променя забравената парола на потребител след като е изпратен валидния жетон за забравена парола.

@Test  
public void testStoreUserLoginInformation() {…} – Тества се дали се запазва информация за влизане в профил.

## **4.2. Ръчни тестове**

1. Регистрация с невалидни данни:



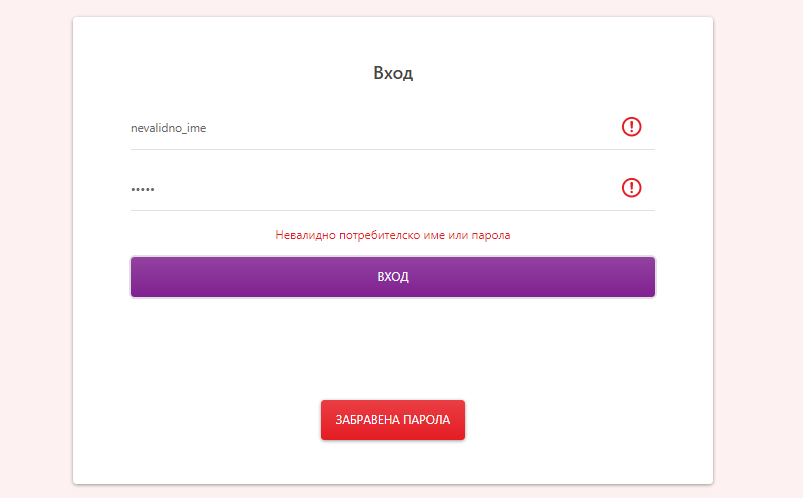
Фиг. 15 Форма за регистрация с невалидни данни

1. Регистрация със вече използван имейл:



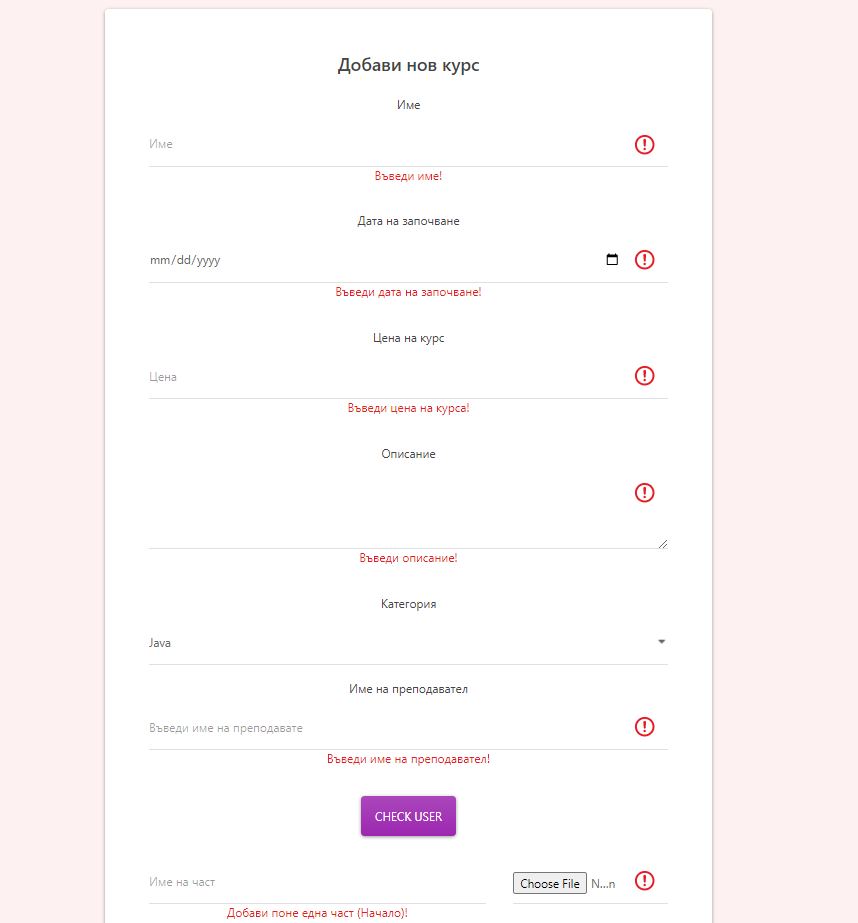
Фиг. 16 Форма за регистрация с заето потребителско име

1. Използване на невалидни име и парола при вход:



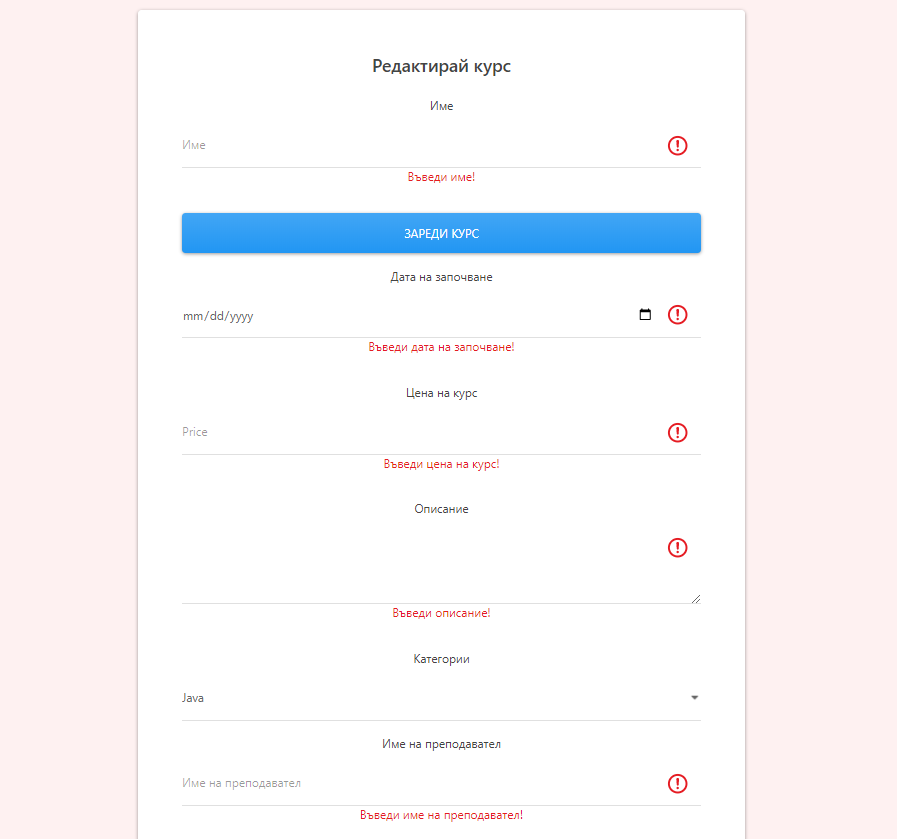
Фиг. 17 Форма за вход

1. Добавяне на курс с невалидни данни:



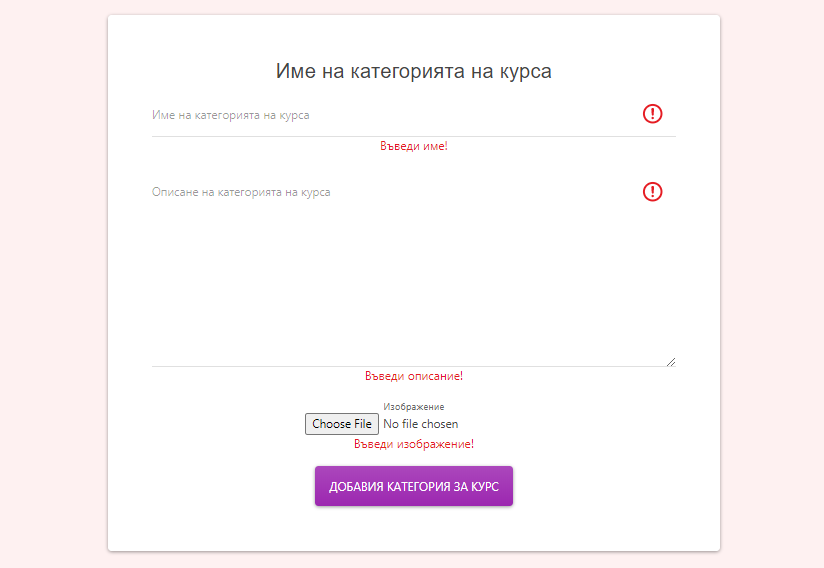
Фиг. 18 Форма за добавяне на курс

1. Редактиране на курс с невалидни данни:



Фиг. 19 Форма за редакция на курс

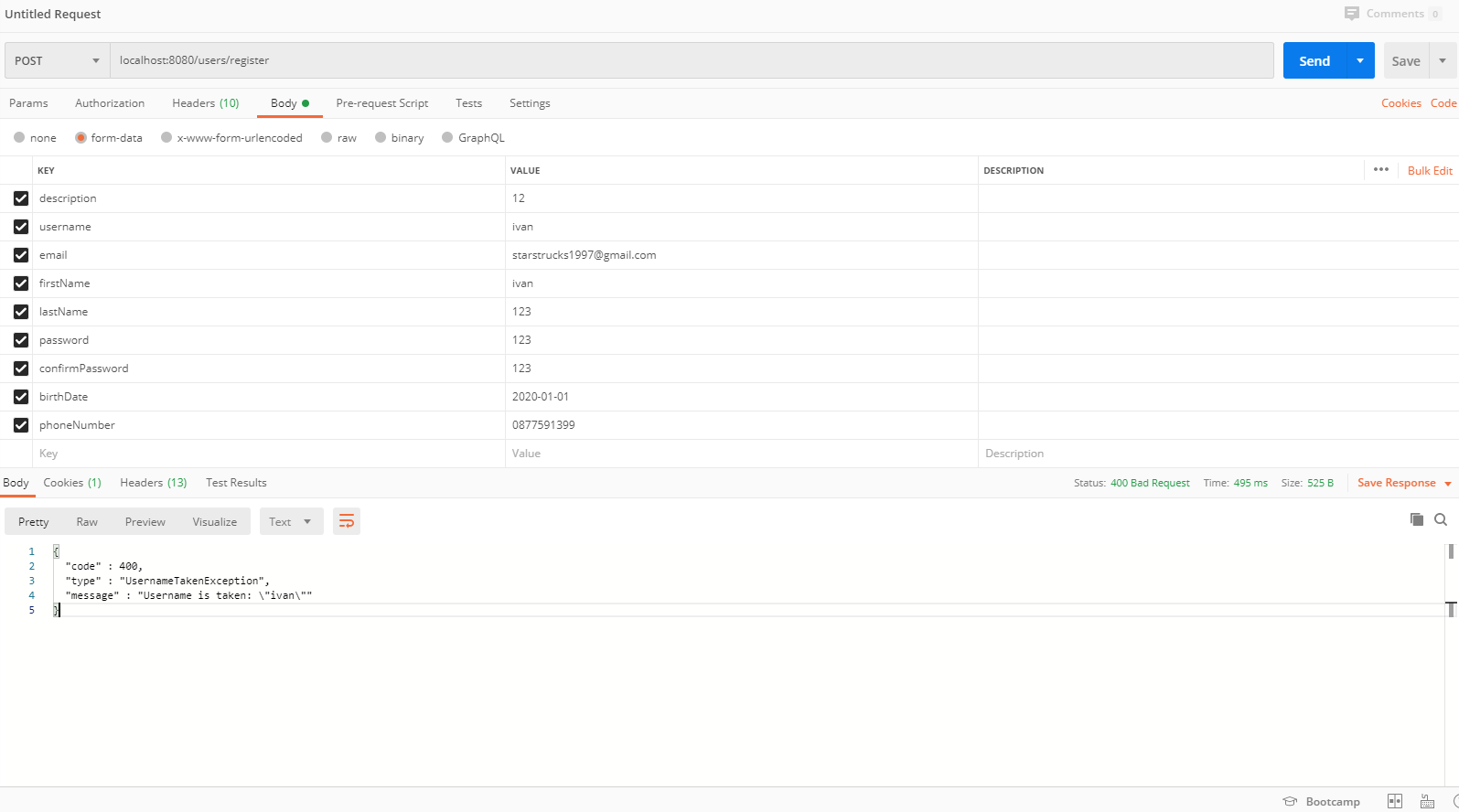
1. Добавяне на невалидна категория:



Фиг. 20 Форма за добавяне на категория

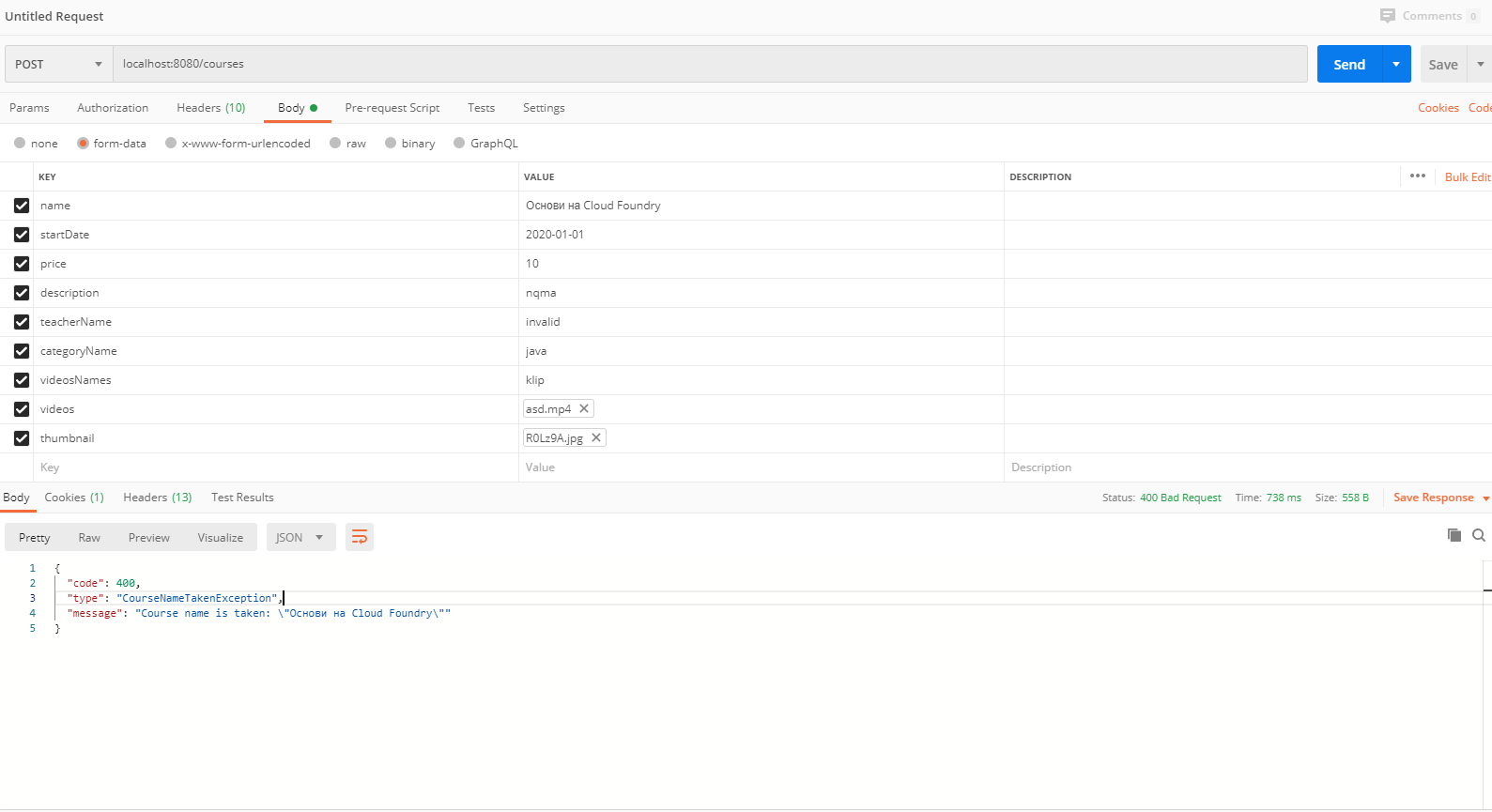
Заявки директно към сървъра с невалидни данни:

1. Регистрация с невалидни данни:



Фиг. 21 Изпращане на заявка с невалидни данни

1. Добавяне на курс с заето име:



Фиг. 22 Изпращане на заявка с невалидни данни

# **Заключение**

Както се вижда от предходните точки уеб приложението в основата си представлява уеб сайт за онлайн обучение. В него потребителите могат да се регистрират, да се записват за курсове, да оценяват, коментират и дават обратна връзка. Също така да получават информация, под формата на имейли, за най-актуалните курсове.

Целта на приложението е да помага на хората, които имат желание да се развиват в сферата на компютрите, технологиите и софтуера. То предоставя иновативни курсове, които потребители могат да използват с цел подобряване и развитието на техните умения. Курсовете са подходящи както за хора без опит, които имат желание за научаването на нови неща, чрез интерактивни курсове, така и за хора, които имат опит в сферата и се нуждаят от надграждане на своите знания. Приложението включва курсове от най-разнообразно естество, предназначени предимно в софтуера. Има курсове за най-различни езици, подходящи за начинаещи и напреднали. Също включва курсове за най-различни платформи, които са масово разпространени и използвани както в малки компании, така и в големи корпорации.

Има включени както платени, така и безплатни курсове. Безплатните целят привличането на повече хора, които в бъдеще да се възползват и от някои от професионалните курсове. Приложението също така насърчава своите курсисти да се възползват от възможността да видят какво представлява реален проект, тъй като source кодът на приложението е отворен. Така те не само могат да видят истински код, ни и да правят промени по него. Това е една от най-отличаващи характеристики на това приложение с всички други, тъй като никоя от големите и установили се платформи не е с отворен код.

С отворения код се цели и включването на още хора, които имат желание да работят по него. Чрез този подход може да се създаде “community”, което да разработва и развива софтуера и качва нови видео уроци, които да бъдат предлагани на потребителите, които ще допринасят 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 бъдат оправяни. В бъдеще ще бъдат добавени още функционалности като плащане през сайта, още интеграции и по-голяма стабилност, като разбира се помощ от обучаващите се потребители е винаги добре дошла.

# **Код**

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

Линкове към пълния код:

- Frontend (client) - <https://github.com/IvanBorislavovDimitrov/Learn-to-code-frontend>

- Backend (server) - <https://github.com/IvanBorislavovDimitrov/Learn-to-code>

1. UserServiceApiFacade

@Service  
public class UserServiceApiFacade extends ExtendableMapper<UserServiceModel, UserResponseModel> implements UserServiceApi {  
  
 private final UserValidator userValidator;  
 private final UserService userService;  
 private final PasswordEncoder passwordEncoder;  
 private final RoleService roleService;  
 private final RemoteStorageFileOperator remoteStorageFileOperator;  
 private final EmailClient emailClient;  
 private final ApplicationConfiguration applicationConfiguration;  
  
 @Autowired  
 public UserServiceApiFacade(ModelMapper modelMapper, UserValidator userValidator, UserService userService,  
 PasswordEncoder passwordEncoder, RoleService roleService, RemoteStorageFileOperator remoteStorageFileOperator, EmailClient emailClient, ApplicationConfiguration applicationConfiguration) {  
 super(modelMapper);  
 this.userValidator = userValidator;  
 this.userService = userService;  
 this.passwordEncoder = passwordEncoder;  
 this.roleService = roleService;  
 this.remoteStorageFileOperator = remoteStorageFileOperator;  
 this.emailClient = emailClient;  
 this.applicationConfiguration = applicationConfiguration;  
 }

@Override  
public ResponseEntity<UserResponseModel> register(UserBindingModel userBindingModel) {  
 userValidator.validateUserBindingModel(userBindingModel);  
 UserServiceModel userServiceModel = toUserServiceModel(userBindingModel);  
 if (userBindingModel.getProfilePicture() != null) {  
 remoteStorageFileOperator.uploadFileAsync(new FileToUpload(userServiceModel.getProfilePictureName(), userBindingModel.getProfilePicture()));  
 }  
 addRolesForUser(userServiceModel);  
 convertAndSetDateToUser(userBindingModel, userServiceModel);  
 userService.registerUser(userServiceModel);  
 sendActivationEmail(userBindingModel);  
 UserResponseModel userResponseModel = toOutput(userServiceModel);  
 return ResponseEntity.*ok*(userResponseModel);  
}

private void sendActivationEmail(UserBindingModel userBindingModel) {  
 String activateAccountUrl = applicationConfiguration.getActivateAccountUrl();  
 activateAccountUrl = String.*format*(activateAccountUrl, userBindingModel.getUsername());  
 Email activateAccountEmail = new Email.Builder()  
 .setContent(MessageFormat.*format*(Messages.*ACTIVATE\_ACCOUNT\_EMAIL*, activateAccountUrl))  
 .setTitle(Messages.*ACCOUNT\_ACTIVATION*)  
 .setRecipient(userBindingModel.getEmail())  
 .build();  
 emailClient.sendAsync(activateAccountEmail);  
}

@Override  
public ResponseEntity<UserResponseModel> updateProfilePicture(String username, MultipartFile profilePicture) {  
 UserServiceModel userServiceModel = userService.findByUsername(username);  
 remoteStorageFileOperator.removeFileSync(userServiceModel.getProfilePictureName());  
 userServiceModel.setProfilePictureName(getProfilePictureName(username, profilePicture));  
 remoteStorageFileOperator.uploadFileSync(new FileToUpload(userServiceModel.getProfilePictureName(), profilePicture));  
 UserServiceModel updatedUserServiceModel = userService.update(userServiceModel);  
 return ResponseEntity.*ok*(toOutput(updatedUserServiceModel));  
}

}

1. ResourceServiceApiFacade

@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);  
 }  
}

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_controllersю>