Smarthives - система за мониторинг и генериране на статистка на кошери за пчели

Проект по „Софтуерни технологии“, 2018 г.

Факултет математика и информатика, Софийски университет

Изготвили:

45031**, **Иван Александров Филипов**, **vanaka1189@gmail.com
Информатика, 4 курс, 1 поток, 1 група
45075**, **Николай Тихомиров Коцев**, **nikolay.t.kotzev@gmail.com
Информатика, 4 курс, 1 поток, 2 група
45085**, **Димитър Илиянов Димитров**, **mitko.bg.ss@gmail.com
Информатика, 4 курс, 1 поток, 1 група
45126**, **Никола Момчилов Николов**, **nikola.nikolov.11.11.11@gmail.com
Информатика, 4 курс, 1 поток, 1 група

Ръководител: доц. Димитър Биров

2018 г.

Съдържание

1 Увод

1.1 Тема на проекта

Проекта представлява софтуерно и хардуерно решение за създаване на клъстери от умни-кошери(SmartHives). Пчеларите имат брой кошери, често разпръснати по различни региони. Поради това те имат възможност да преглеждат един кошер на една седмица, дори по рядко. Това обаче не достатъчно понеже процесите в пчелните колонии се изменят драстично в малки периоди от време. Така възниква нуждата от по-стабилен и по-бърз начин за проверка. Това обаче не може да бъде изпълнено от човек за кратко време затова е нужна модернизация на метода на проверка.

1.2 Изисквания

- Wifi свързаност
- Енергонезависима памет
- I2C комукикационна шина
- Сензор за отчитане на температура на обект и околна среда
- Сензор за засичане на движение (акселерометър)
- Захраневане чрез батерия и/или постоянно захранване
- Възможността за смяна на батериите без специализирано техническо образование
- Възможност за управлнение на сензори, през поне 12 GPIO
- Да може да се регистрират нови потребители в системата.
- Да може потербителя да регистрира закупените от него устройства.
- При недостъпност към мрежата да се запазва информацията събрана от сензорите за период от поне три месеца. При възможност този период може да бъде одължен.
- Управление на локалното хранилище
- Възможност за управление на OTA (over the air update)

- Да може да се изпрати заявка през библиотека към приложно-програмения интерфейс на Telegram, така се оповестява собственика за проблем с кошера.
- Възможност за изпращане на заявки и изчакване за обработка на отговорите им
- Бърз обектно-ориентиран системен език
- Front-facing сървъра да издържа на множество едновременни заявки.
- Да не се предоставя потребителска информация преди автентикация.
- Потребителите да имат достъп само до информация предоставена само от техните устройства.
- Да може сървъра да приема информация за кошер, преди той да е присвоен от потребител. Когато той е присвоен от потребител да запазва вече събраната информация за този кошер.

1.3 Цели на проекта

Този проблем може да се реши много лесно. Единственото нужно е пчелната индустрия да се адаптира към бързоразвиващите се дигитални технологии. Именно това е целта на нашия проект – дигитализиране на мониторинга на кошери, като това включва:

- Измерване на важни данни за състоянието в кошерите
- Онлайн проверка на текущото състояние на кошерите
- Статистик за настъпилите промени представени чрез графики
- Security модул – алармиране при настъпване на извънредни ситуации

1.4 Резултат

Резултатите от подобна адаптация са лесно забележими. Предвижда се по-добро познание на външните влияния върху кошерите, както и придобиване на по-добра представа за ефективността и устойчивостта на процесите в пчелните

колонии. Така може да се положи по-адекватна грижа за здравето на пчелите и също така да се предотвратят или спрат навреме извънредни ситуации.

Технологиите, с които се постига това са:

- ESP8268 контролер, който управлява:
  - Термометър
  - Барометър
  - Акселометър
- Access point, базиран на 3G/LTE технология
- Server, DBMS и Web Interface за събиране, запазване и репрезентиране на данните

2 Описание на:

2.1 Чипът

Главният управляващ микрочип е: esp8266!

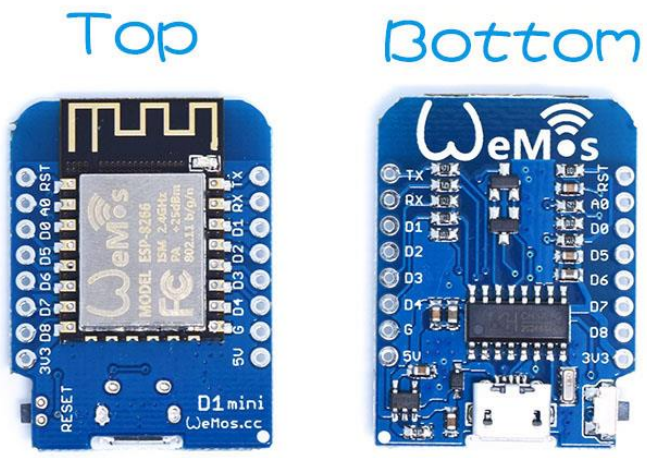
Неговото предназначение е да събира данните от всички свързани към него сензори, да ги обработва и пакетира в подходящи пакети, а след това посредством безжична връзка да ги изпраща на сървър, където да бъдат съхранявани и анализирани. Освен това предлага възможност за запазване на данните локално, при проблем с интернет комуникацията и известяване на собственика за опасности свързани с кошера посредством чатбот.

Технически данни и особености:

- разполага с 32-битов RISC-базиран микроконтролер
- 64К-байта RAM за програмните инструкции +
  96К-байта оперативна памет за програмата

- около 4M-байта FLASH памет, която може да се използва за дълготайно хранилище
- 16 пина вход / изходни пина с общо предназначение
- 1 ADC (преобразовател на аналогов към дигитален сигнал)
- поддържа протоколи WiFi, Serial, I2C, SPI, PWM
- работно напрежение 5/3.3V
- 1 micro - USB вход (предвиден за програмиране и захранване)

два бутона - един рестартиращ и един за програмни цели вграден WiFi модул 802.11 b/g/n , Wi-Fi Direct (поддържа P2P директна връзка)



Фиг. 1 Микрочипът погледнат от две страни

Разход на батерия:

Чипът поддържа два режима на работа - активен и пасивен.

При активният режим разходът на батерия е осезаемо по - голям от 30ma до 300ma, докато при пасивния (или още спящ режим) Esp8266 харчи само 0.001 - 0.1ma. Спящият режим на esp може да бъде постигнат по два начина, чрез извикване на функцията sleep, която поставя микроконтролера в почиващ режим. За определен брой милисекунди той не се занимава с нито една задача. Този вариант не е одачен, тъй като при него все пак се харчи високо количество батерия - около 0.5ma. Esp

поддържа така нареченето дълбоко спане - режим при, който единствено работи таймер отброяващ колко време остава до събуждането на чипа.

Една стандартна литиево - полимерна батерия LiPo с работно напрежение 3.3 – 4.2V (3.7) и капацитет 10 – 5000mAh би издържала около ден и половина при непрекъсната работа на ESP в режим "активен". Ако ESP през по - голямата част от времето си е в пасивен режим, то очакваните резултати за броя на дните на работа е около 14 - 15.

Това означава, че за да има оптимално отношение работа - разход на батерия, трябва в програмното осигуряване да бъде предвидено режимите на работа да бъдат редувани.

№	ТИП	ВОЛТАЖ(V)	КАПАЦИТЕТ	СЪПРОТИВЛЕНИЕ	САМОРАЗРЕЖДАНЕ
1	**Coin Cell 2032**	3.00	200	weВисоко	Ниско
2	**AAA/AA/C/D**	1.2-1.5	1000-10000	Средно	Ниско
3	**AAA/AA/C/D (NiMh)**	0.2 – 1.3	1000-10000	Средно	Средно/Ниско
4	**AAA/AA/C/D (Ni-Zn)**	1.3 – 1.6	1000-10000	Високо	Средно
5	**LiPo**	3.3 – 4.2	10 – 5000	Ниско	Ниско
6	**LiPo - 16850**	3.3 – 4.2	1000 – 5000	Ниско	Ниско
7	**LiPo – 14500**	3.3 – 4.2	500 – 1000	Ниско	Ниско

Таблица 1. Типове батерии

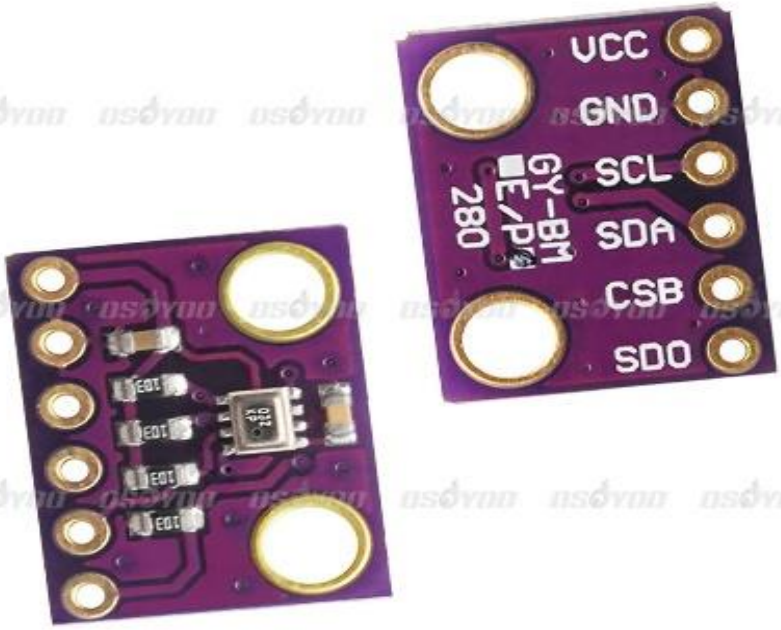
2.2 Сензори

Към всяко едно от устойствата ще бъдат свързани следните сензори, информацията събирана от тях ще се изпраща за обработка от сървъра. След дълго проучване и консултиране със специалисти се установява, че е нуждно да следи

състоянието на околната среда в и около кошера на пчелите. Според специалисти пчелари от най-голямо значение са температурата и влагата.

2.2.1 СЕНЗОР ЗА ТЕМПЕРАТУРА/ВЛАГА:

- **BME280**
  - Точност: +/- 0.5^0
  - Резолюция: 0.01 (14 бита)
  - Протокол: I2C
  - Сензор на шина
  - $-25^0 : 125^0$
  - цена : ~9 лв



Плюсове: измерва температура, влага, налягане, има подходящи библиотеки, възможност за навързване на повече от 1 сензор към същата информационна шина, лесен за монтаж към устойството.

Конкуренция:

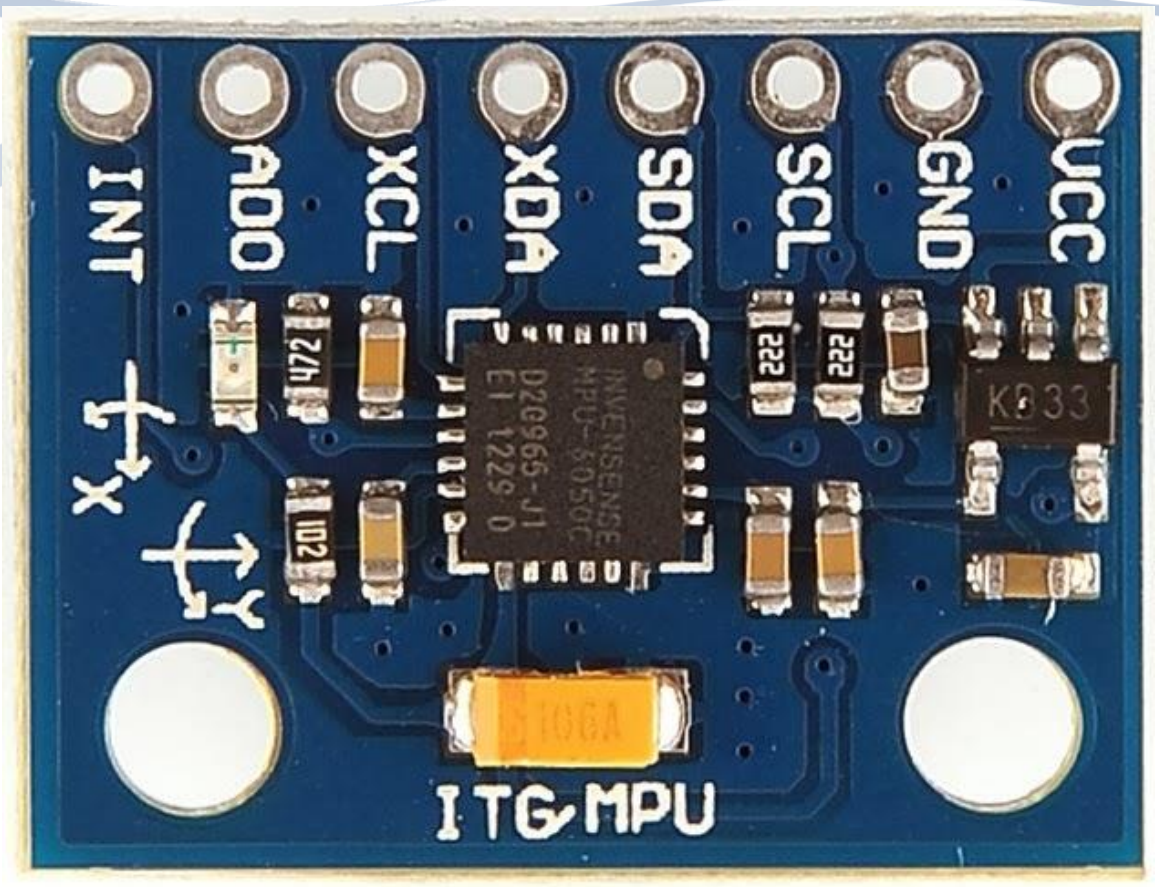
- DS18B20
  - Точност: +/- 0.5^0
  - Резолюция: 0.06 (12 бита)
  - Протокол: 1-Wire
    - Минуси - само за температура, висока неточност при четене на стойностите
- SI7021
  - Точност: +/- 0.3^0
  - Резолюция: 0.01 (14 бита)
  - Протокол: I2C
  - 1 Сензор на шина
  - $-25^0 : 125^0$
    - Минуси - висока цена, липса на подходящи библиотеки за софтуер, само 1 на информационна шина
- PT100
  - Протокол: Съпротивление
  - $-20^0 : 500^0$
    - Минуси - само за температура, голяма грешка при отчитането

2.2.2 АКСЕЛЕРОМЕТЪР/ЖИРОСКОП:

Предназначение - за отчитане на движение използва се в модула за сигурност

избор:

- MPU6050 – 6 DOF (Degrees Of Freedom)
  - SPI комуникация
    - Плюсове - високо ниво на обновяване на данните, добра цена, наличност на програмни ресурси



Фиг. 3. MPU6050 сензор

2.3 Комуникация със сървъра

Всяко едно от устройствата, чрез своя Wi - Fi ESP чип се свързва към AP (mtel - netbox or something) посредством предварително известни SSID (име на мрежата) и парола.

Така се осигурява интернет свързаността на всяко едно устойство.

Комуникацията устройство - сървър е еднопосочна, всяко едно от устройствата изпраща събраните от него данни към сървъра посредством HTTP POST заявка, във всеки изпратен пакет устройстото изпраща : своето фабрично ID, данни за температурата, данни за налягането на въздуха, данни за състоянието на

батерията си и време, в което е извършено замерването. Една примерна заявка към сървъра би изглеждала така:

```
POST  HTTP/1.1
    Host:
    Content-Type: application/x-www-form-urlencoded
    Cache-Control: no-cache
    battery=50&humidity=12&temperature=24%2C6&unic_id=9001415
```

След като получи отговор от сървъра, че всичко е наред устройството продължава със своята нормална работа, обаче ако няма отговор от сървъра или пък е получен отговор съдържащ код за грешка устройството преминава към запазване на данните в своята памет.

2.4 Локално хранилище

Всяко едно от устойствата базирани на esp8266 има 4МБ флаш памет, която ще наричаме локално хранилище. Предназначение:
В тази памет ще бъде съхранявана някоя от същинско важната за работата на устойствата информация, а именно - IP адрес на сървъра, където да изпращат данните събирани от сензорите им , SSID на мрежата към която да се свържат за да имат достъп до интернет и парола на тази мрежа. Допълнително ще бъде съхранена информацията за работа с Telegram chatbot-а, за целта са необходими и BOT TOKEN + CHATID. Освен тези конфигурации за работата на устройствата в локалното хранилище ще бъдат запазвани и също всички неуспешно изпратени данни към сървъра (устойството е нямало връзка или сървърът е паднал) .При първа възможност (отново е осъществена комуникация със сървъра) всички запазени данни в хранилището ще бъдат повторно изпратени и ако успешно бъдат обработени в сървара ще бъдат премахвани от хранилището.

Примерна организация на хранилището:

Можем да си представим, че имаме клетки на непекъснат болок от памет, всяка една клетка е един байт (осем бита) следователно имаме общо **4 194 304**

клетки. Може да ги адресираме от 0 до 4194303. Първите 32 клетки - от 0 до 31 ще резервираме за IPv4 адреса на сървъра. Следващите 64 клетки запазваме за символния низ, които ще представлява името на мрежата, която създава всеки AP. Те ще бъдат последвани от 64 символа за паролата на тази мрежа.За BOT TOKEN и CHATID са необходими допълнителни 32клетки. Остават ни 4 194 304 - 192 = 4 194 112 свободни клетки.

Записите от данни, които трябва да правим за всяка неуспешно обработена заявка от сървъра са в следния вид:

поле за температура: стойности -60 до 60, които ще преобразуваме в число от 0 до 120 => ще ни стигне една клетка за числото преди десетичната запетая и още една за това след нея.

поле за влажност в проценти: стойности от 0 до 100 => една клетка

поле за състояние на батерията в проценти: стойност от 0 до 100 => една клетка

=> за всяка неуспешна заявка ще трябва да запазвам по общо 4 клетки =>

можем на локалното хранилище да запазим цели 1 048 536 записа.

Всеки запис се прави в интервал от 15 минути => максимум можем да натрупаме 96 записа за денонощие => хранилището ще може да запази данните за повече от една астрономическа година, в която няма комуникация със сървъра.

Заключение: хранилището е енергонезависимо и с достатъчно голяма вместимост => че е надежден заместител при липса на интернет комуникация.

2.5 Защитен модул и чатбот

Предвижда се включването на опционален модул против кражби на кошери. Целта на модула е при минимална вероятност за кражба на някои от кошерите, собственика им моментално да бъде уведомен.

Включването на защитен модул е според желанието на клиента, то би довело до по - голям разход на батерия, но по - голямо спокойствие у клиента.

Работа на защитния модул:

Модулът прави рутинни проверки на всеки 5 секунди - всяка проверка е свързана с това да бъдат прочетени 10 пъти данните от жироскопичния сензор, чрез тези данни и математическо моделиране върху тях може с точност да се пресметне движението на устойството във всяка една от посоките на триземрното пространство. Ако по време на някоя от тези проверки възникне разлика по някои от осите над допустимото, то се приема, че кошерът е претърпял някакво движение (краде се, бутнат е, някои го мести), което кара модула веднага да сигнализира на собственика на кошера за това явление.

Защитния модул комуникира директно със клиента посредством чатбот в Telegram приложението за чат. Комуникацията е двупосочна като клиента може да праща команди на устойството, а пък устойството от своя страна може да изпраща информация и да алармира за възникнало събитие.

За всеки от клиентите се предвижда по един чатбот за всеки негов клъстер.Чрез този чатбот той ще може да комуникира с всяко едно от устойства в този клъстер, комадите който собственика праща се приемат от всички устойства, а пък всяко едно от тях автономно може да сигнализира за различно събитие.

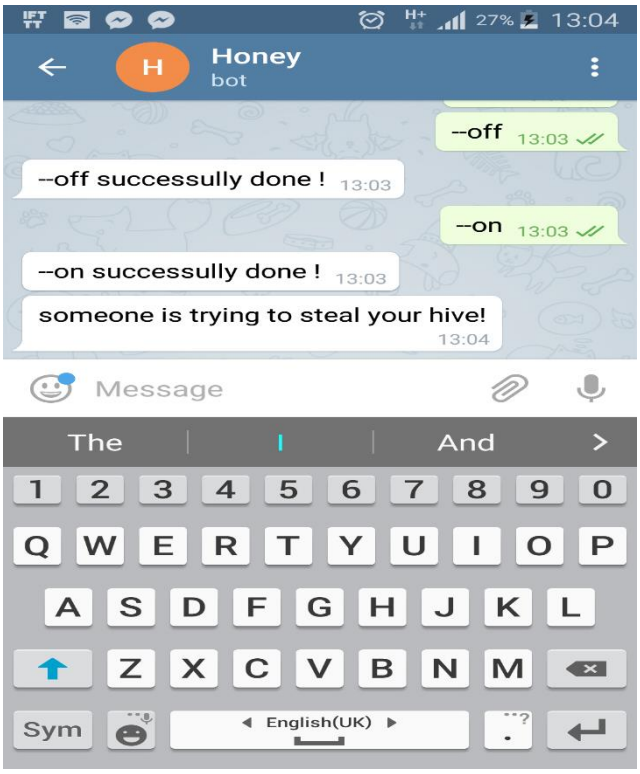
При засичане на движение от някоя от рутинните проверки устойството на кошера изпраща съобщение към собственика със съдържание: “Някой краде кошерът Ви!”, след което устойството на кошера се заключва докато не получи съответната команда --unlock от собственика.

Други събития за, които всяко устойство може да предупреди собственика си са: прекалено висока темература / прекалено високо налягане, също така може да информира когато някое от устойствата успешно е обновило версията на софтуера си.

Командите от срана на клиента могат да бъдат следните:

- off казва на всички устойства от даден клъстер веременно да изключат защитния си модул - това означава, че ще пестят повече батерия, а пък собственикът ще може да работи върху тях без да получава събощения, че някои се опитва да ги краде.

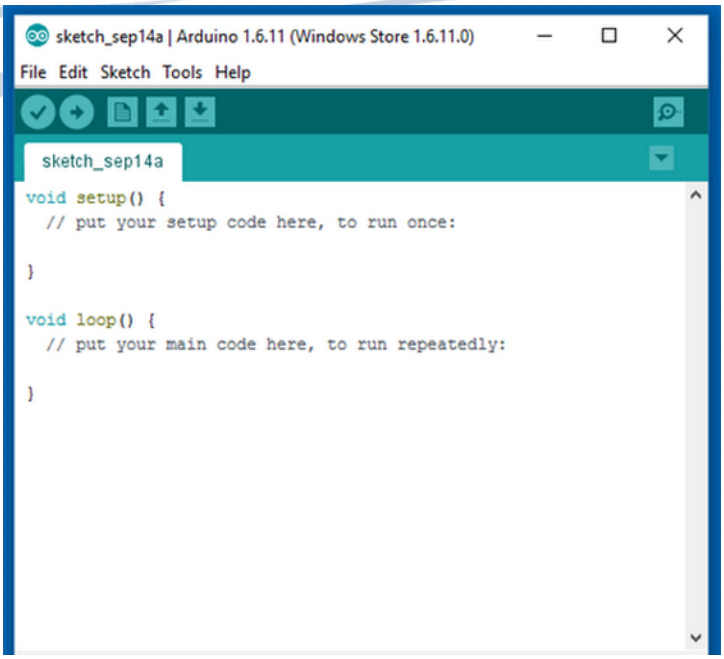
- on активира отново изключения защитен модул
- unlock активира отново устойствата, които са се заключили защото са били заплашени от някакво събитие



Фиг. 4. Примерно изпълнение на командите в чатбота

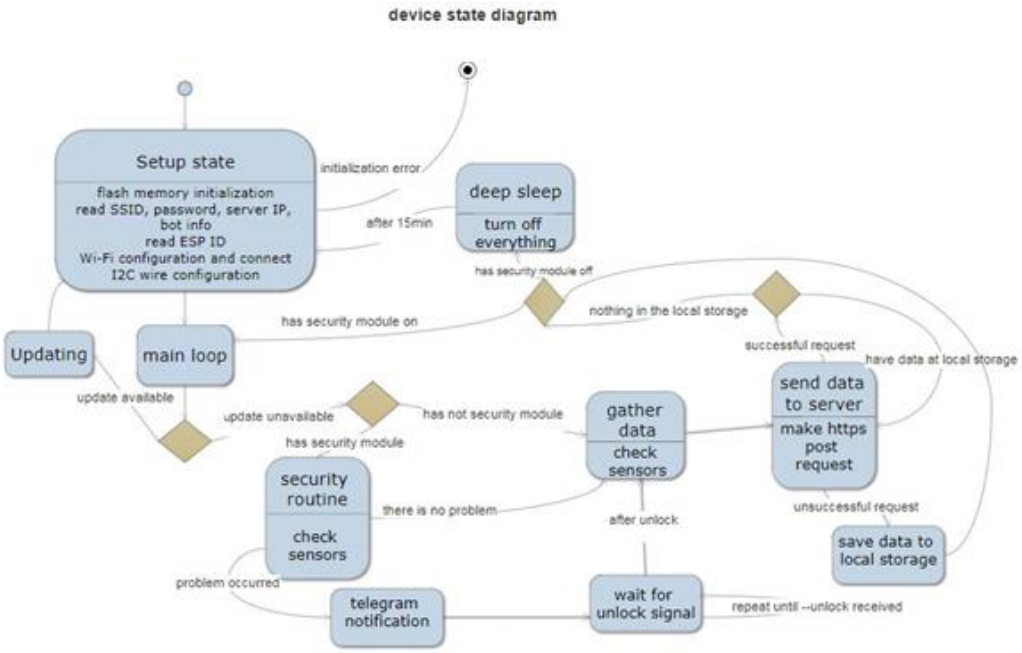
2.6 Софтуерно осигуряване

Чипът може да бъде програмиран директно използвайки езикът C. Другата опция е да се използва платформата за програмиране на Arduino(Arduino IDE), което позволява програмиране на по - високо ниво посредством Arduino програмният език, който е силно повлиян от C++. Качването на програма на чипа се случва посредством USB - портът му, друг вариант е да се направи така нареченото Over-the-air-update(дистанционно актуализиране на софтуера).

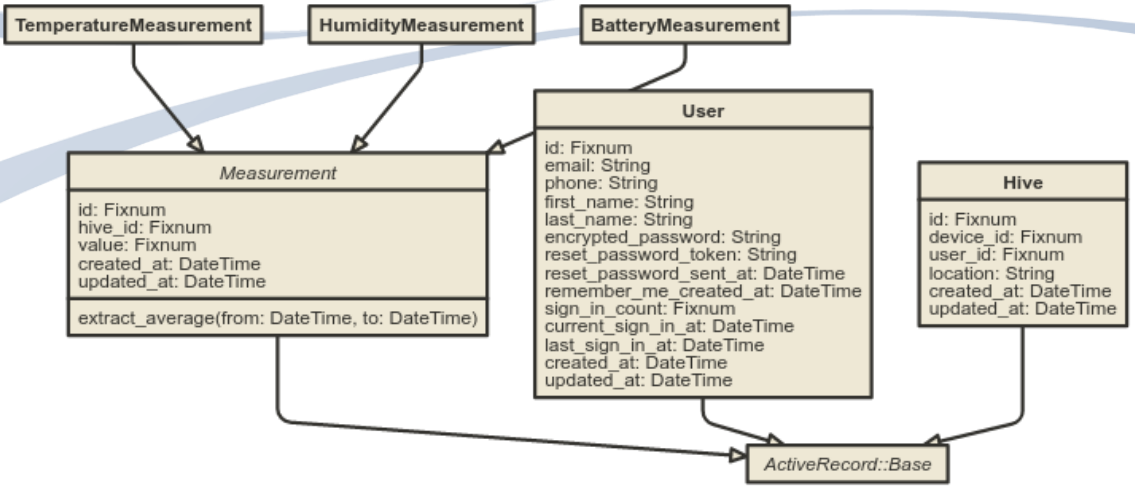


Фиг. 5 Среда за програмиране – Arduino 1.6.11

Начинът на работа на програмата, която ще бъде инсталирана във всяко едно устойство подробно се описва от следният UML.



Фиг. 6 UML за състоянието на устройството



Фиг. 6 Пространство на класовете

3 Уеб интерфейс

3.1 Избор на уеб технология

Уеб интерфейсът е основна част от нашето приложение отговаряща за репрезентиране на данните събрани от устройствата на клиента и запазването на тяхната сигурност. Преди да започнем разработката на нашето приложение ще трябва да разгледаме популярните технологии използвани в момента.

3.1.1 EXPRESS (NODE.JS) + ANGULAR

Комбинацията от Express и Angular е, може би, най-модерната за предоставяне на динамични и красиви уеб страници. Но първо, какво се крие зад двете имена.

Express.js или просто, Express, е open source framework за уеб апликации за Node.js, достъпна под MIT лиценза. Направена е за бързото изграждане на уеб апликации и API-и. Оригиналния и автор я описва като сървър, вдъхновен от Sinatra, което означава че целта му е била да създаде минималистична библиотека,чиито способност лесно да бъдат допълвани от plugins, по избор на разработчиците.

Angular (често наричан Angular2) е TypeScript (език транспилиращ се до Javascript) базирана, open source front-end платформа за уеб апликации разработена

от екип на Google. Той предоставя удобен начин за структуриране на front-end кода и много удобства за опитните с него, front-end разработчици, но заедно с това носи overhead и малко boilerplate code, който трябва да се копира.

Не случайно комбинацията от тези два framework-а е популярна. Express помога за невероятно бърза разработка на back-end частта на приложението, докато Angular рава възможността за най-модерните и добре изглеждащи динамични уеб страници. Факта че всичко се пише на един и същи език, също е голям бонус. Но тази комбинация, не е толкова утопична, колкото звучи на пръв поглед. Самият Javascript все още се разработва активно, и често голяма част от feature-ите които би очаквал да са част от езика, са все още nightly. Ако ги използваш то техния интерфейс може да се промени за следващата версия за езика и да се наложи да пренаписване на големи части от приложението на базата на промените.

3.1.2 C♯ И ASP.NET

ASP.NET (на английски: Active Server Pages за .NET) е технология за създаване на уеб приложения и уеб услуги, разработена от Microsoft. За първи път е публикувана през януари 2002 г. с версия 1.0 на .NET Framework, и е наследник на Microsoft Active Server Pages (ASP) технологията. ASP.NET е изградена въз основа на Common Language Runtime (CLR), което позволява на програмистите да пишат ASP.NET код като използват .NET език по избор. В нашия случай, избора би бил най-популярния, а именно C♯.

CLR и компилацията при първата потребителска заявка биха направили приложението много по-бързо, но ние не очакваме особено голям трафик към web приложението ни, тъй като към текущия момент плановете ни са да обслужваме само български производители. Също обвързването към дадена корпорация и липсата на сертифициран от Microsoft разработчик, биха довели до overhead при лицензирането на софтуера.

3.1.3 RUBY ON RAILS

Ruby on Rails (често съкращавано в английски като Rails или RoR) е популярна софтуерна рамка под MIT лиценз. Rails е framework основан на модела за дизайн

Модел-Изглед-Контролер (MVC), написана изцяло на програмния език Ruby. Тя включва в себе си множество предварително зададени структури за бази данни, уеб услуги и уеб страници. Ruby on Rails е разделен на различни под-библиотеки, така че разработчиците сами да избират кои части от него да използват. Обществото около Rails е вече съзряло и има достатъчно de-facto стандарти от който бихме могли да се възползваме. Това, че framework-а включва в себе си ORM (object-relational mapping) и удобни и лесни за разбиране конвенции е голям бонус за разработчиците.

3.1.4 ИЗБОР

В края на краищата се спираме на Ruby on Rails. Тя е силно развита и стабилна рамка за разработване на уеб приложения с развита и стабилна общност около нея и има достатъчно добре развити и тествани библиотеки които биха ни помогнали значително при разработката на уеб базираната част на проекта.

3.2 Избор на система за управление на база от данни

Важна част от технологичните избори пред които е изправен един екип при започване на един нов проект, е избора на СУБД (за улеснение наричано база данни, по-надолу). Тъй като се опитваме да намалим възможно най-много разходите по-проекта и тук най-вероятно ще изберем open source проект.

3.2.1 MARIADB (MYSQL)

MariaDB е fork на MySQL разработван от оригиналните автори на така популярната система за управление на бази данни, чиято цел е да предостави open-source drop in заместител на закупената от Oracle система. До скоро разработката на двете беше тясно свързана като Maria DB напълно репликираше всичко което бе достъпно дори в Enterprise версията на MySQL. Към текущия момент, това вече не е така, след като общността около MariaDB решиха да не репликират част от нишовите функционалности въведени в MySQL, които силно влияят на производителността на цялата система.

3.2.2 POSTGRESQL

PostgreSQL, често наричана Postgres, е обектно-релационна система за управление на бази от данни, фокусирана върху разширяемостта и спазването на изискванията на стандартите. Тя е подходяща и използвана за приложения, вариращи от най-прости сървъри за домашна употреба до огромни "складове от данни". Тя е (ACID), поддържа транзакции и всички други основни функционалности от MySQL, MariaDB, DB2, Oracle, MSSQLS.

3.2.3 ИЗБОР

Екипа избра да изплзваме PostgreSQL. Приложението ни е достатъчно просто, че макар и важен избор, всички популярни системи за управление на бази данни, биха били достатъчно ефективни и избора ни в случая е изцяло базиран на персонални предпочитания и предишен опит с избраната система.

3.3 Уеб приложението

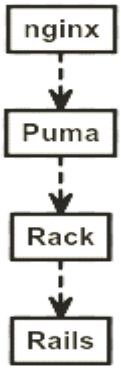
Уеб приложението е важна част от проекта отговаряща за даването на достъп и визуализирането на данните на клиентите ни. Без него информацията от кошерите щеше да е твърде трудно достъпна за клиентите ни. Чрез него ние даваме достъп до информацията на клиентите ни, агрегираме я и я визуализираме по удобен и добре изглеждащ начин.

3.3.1 APPLICATION STACK

Използвани библиотеки:

- nginx - front-facing server
- Puma - application server
- Rails - application framework
- Devise - Регистрация и аутентикация на потребители
- Highcharts - Генериране на динамични web диаграми

Към текущия момент nginx -> Puma -> Rack -> Rails е de-facto стандартния начин за подръжка на Rails приложения.



Фиг. 8.Flow

3.3.2 NGINX

nginx е изключително бърз, паралелизиран http server, който най-често бива използван като reverse proxy. Reverse proxy (обратно прокси) е модел при който преди заявките да стигнат до основния server който, ги прочита и на базата на тях генерира динамични отговори, те първо минават през сървър който да филтрира реалните заявки от тези които просто се опитват да причинят вреда. Част от отговорностите на един прокси сървър са:

- Да крият част от характеристиките на основния сървър, така че да бъде по-трудно откриването на уязвимости в основния сървър.
- Firewall, опазващ от DoS и DDoS атаки.
- Сервиране на статични ресурси (изображения, client side scripts (скриптове изпълнявани от браузъра на потребителя) и други).

3.3.3 PUMA

Puma е най-популярния application server (приложение зареждащо ruby кодът) за Rack приложения. За разлика от другите популярни application servers, Puma е изградена с мисълта за скорост, минимално използване на памет и

паралелелизъм. Тя е написана основно на C++ и подържа нужния брой ruby процеси които паралелно да изпълняват заявките на потребителите.

3.3.4 RACK

Rack е стандартът (и библиотеката придружаваща го) за комуникация между application server и приложението ни. Той бива използван от почти всички framework-ове за разработка на уеб приложения на ruby, а дори и на някои други езици.

3.3.5 RAILS

Rails е рамката която нашето приложение ще използва. Една от характерните черти на рамката е това че кара разработчиците да използват MVC design pattern-а.

3.3.6 MVC

MVC е много популярен модел за дизайн, който се използва за разделянето на бизнес логиката на едно приложение от тази отговаряща за репрезентирането на данните. Тя разглежда следните 3 вида обекти:

- Модела е главния вид компоненти на модела за дизайн. Той отговаря за съдържането на бизнес логиката на приложението и запазването на данните. Той директно обработва достъпната информация в апликацията приложението.
- Изгледа е репрезентацията на информацията менижирана от приложението. Обикновенно изгледите контролират използването на прозорците (и други примитиви) на операционната система и запълването им с потребителски интерфейс или генерирането на html страници които да бъдат изпратени на уеб браузъра на потребителя
- Третия вид компоненти са контролерите. Те приемат входни данни от потребителя и ги преобразуват до команди разбираеми от моделите и изгледите.

В нашия случай те ще приемат HTTP заявки и ще връщат на потребителите HTML страници, които ще бъдат визуализирани от техните уеб браузъри.

Освен основните компоненти MVC дефинира и основните взаимодействия между тях:

- Модела е отговорен за управлението на информацията в приложението. Той отговаря на заявки от изгледите и приема команди от контролер, който му задават как да моделира информацията си.
- Изгледа репрезентира информацията в определен формат, когато контролер го изисква.
- Контролера е отговорен да отговори на входните данни на потребителя и да контролира интеракциите между останалите компоненти в приложението.

3.3.7 ACTIVERECORD

Друг важен модел които е част от Rails е ActiveRecord. Той е архитектурен модел, получил името си от Мартин Фауър в книгата му от 2003 „Patterns of Enterprise Application Architecture“. Той се използва за моделиране на структури от данни в паметта към обекти в релационни бази от данни. Модела се състои в това че всяка таблица и всеки изглед от релационната база данни са обвити в клас от приложението. Всяка една инстанция на обекта репрезентира ред от таблицата. След като обект се създаде и се модифицира така че да съхранява релевантната информация за него, то на него се изпраща съобщение да бъде запазен. Всеки обект който не е нов за базата, бива генериран на базата на ред от таблицата за която отговаря разглеждания клас. Всяка модификация на обекта, бива завършвана с викането на `update` и така тя бива отразявана в базата данни.

Този pattern е използван за реализацията на ORM (Object-relational mapping) в библиотека носеща името на pattern-а. ActiveRecord е една от основните библиотеки в Rails и една от основните причини за популярността на рамката. Тя позволява лесна комуникация с база данни и повишава четимостта на кода.

Пример:

```
user = User.new()
user.name = 'John Doe'
user.email = 'john@example.com'
user.save()
```

Ще генерира и изпрати следната заявка:

```
INSERT INTO users (name, email) VALUES ('John Doe', 'john@example.com');
```

Листинг 1. Пример създаване на нов потребител

За сравнение, в езици на които подобен модел не е реализиран, щеше да трябва да имаме специална функция която взима произволни стрингове и ги интерполира в ръчно зададен стринг. Често това може да доведе до малки грешки от страна на програмистите, които да доведат до огромни пробиви в сигурността на приложението.

Друг пример за удобно използване на модела е лесното търсене на обекти. Вместо цялосното изписване на заявката:

```
User.find_by_email('john@example.com')
SELECT * FROM users AS u WHERE u.email == 'john@example.com'
```

Листинг 2. Пример за търсене на потребител чрез имейл

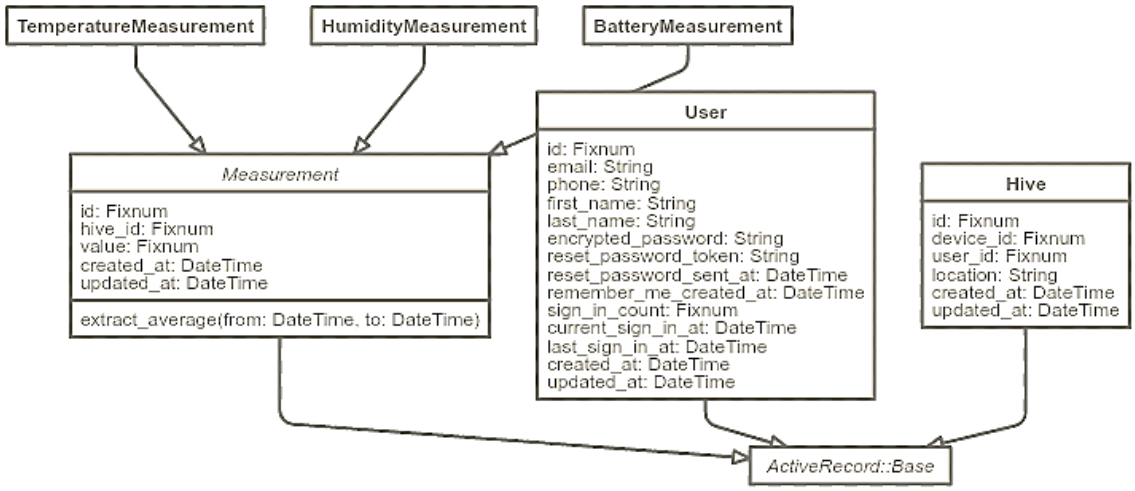
По конвенция ActiveRecord използва имената на полетата в базата данни, като имена на полетата в обектите в приложението. Имената на таблиците трябва да са множественото число на имената на Класовете. Освен удобството в писането на код, ActiveRecord предоставя възможността да се използват различни database driver-и които да модифицират вътрешната репрезентация на заявката до SQL код, подходящ за определената база данни. По този начин приложението не е обвързано за винаги със една и съща база от данни и прави миграцията, при нужда от такава много по-лесна от колкото във всеки друг случай.

3.4 Структура на приложението

До сега разгледахме някои от основните щрихи за уеб приложението. От тук нататък ще започнем да разглеждаме структурата на уеб приложението ни. Ще разгледаме приложението ни по компоненти от MVC модела. Първо ще разгледаме моделите в приложението, след което ще разгледаме едновременно контролери и изгледи (тъй като обикновенно публичните методи на контролерите отговарят на извиквания към изгледите).

3.4.1 МОДЕЛИ

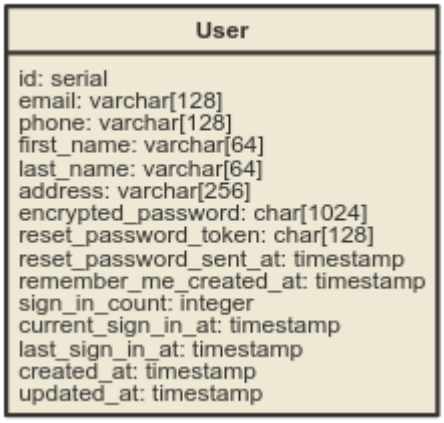
Моделите са класовете които отговарят за съхраняване на отделните видове информация в приложението и бизнес логиката директно свързана с тях.



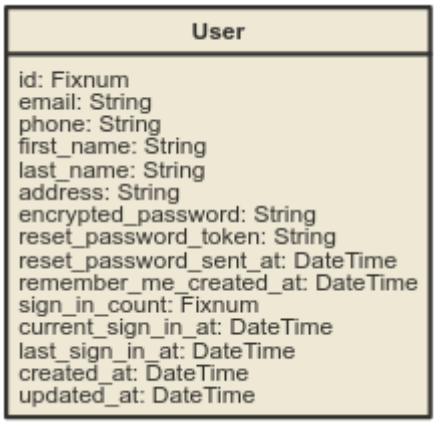
Фиг. 9. Йерархия на модела

3.4.1.1 Потребител

Моделът потребител съхранява информация за потребителите на приложението и част от логиката за модификация на потребителите.



Фиг. 9 Потребителски UML с типове от базата данни



Фиг.10. Потребителски UML с Ruby типове

Нека поговорим за полетата на потребителя:

- `id` е уникален идентификатор за потребителя. Оставяме базата данни да осигурява уникалността на идентификатора, затова типа му е `serial`
- `email` е адреса на електронната поща на потребителя. Тя трябва да е уникална в базата данни, тъй като ще бъде уникален идентификатор
- `phone`, `first\_name`, `last\_name`, `address` са просто информация за потребителя която би била удобна за администриране на приложението и контакт с потребителите.
- Полетата `created\_at` и `updated\_at` са полета които по конвенция Rails добавя към всяка таблица и се използват, както вътрешно в рамката, така и често за дебъгване.
- Останалите полета са генерирани от Devise, библиотеката за аутентикация на потребителите която ще използваме.

Валидираме:

- `email` полето, че отговаря на много опростен регулярен израз проверяващ, че е въведения низ прилича на имейл. Официална валидация на електронна поща е твърде сложна за целите ни.
- `phone` полето, че съдържа само '+' и цифри.
- Device се грижи за валидациите и стойностите по подразбиране във неговите полета.

Атрибутите на Model-а са, макар и добре разписани по-горе биват автоматично генерирани от ActiveRecord. Функциите за взимане (getter) и задаване (setter) на стойностите автоматично се генерират от Ruby, затова не са описани като методи в UML диаграмата. Таблицата в базата данни е индексирана по `email`, тъй като потребителите ще влизат във системата с адреса на тяхната електронна поща и `reset_password_token`, за по-бързо намиране при опит за връщане на забравена парола от страна на потребителя.

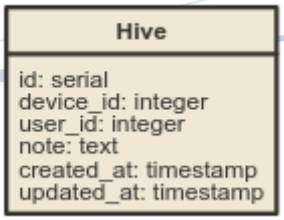
За authentication на потребителите ще използваме библиотеката на Plataformatec – Device. Тя е de-facto стандарта, избран от обществото около Rails. Разработва се от 2009-та година насам и е доказала надежността си в много различни приложения през годините и се използва до днес. Devise не само хешира паролата на потребителя, но добавя и salting към алгоритъма, така че ако злонамерени лица успеят да извлекат паролите на потребителите ни от базата данни, те да не могат да използват Rainbow таблици за да могат да разберат cleartext-а на паролата на протребителя ни.

3.4.1.2 Кошер

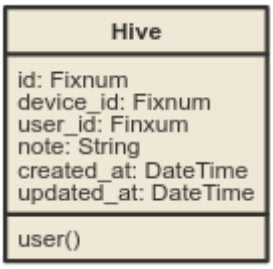
Моделът кошер ще отговаря за съхранението и модифицирането на информацията свързана с кошерите.

Обект от тип Hive:

- `id` е софтуерния идентификатор в апликацията за дадения кошер. Той се ипозлва за да бъдат генерирани URL-и за изгледите отговарящи за визуализирането на данните за дадения кошер.



Фиг. 12 Hive таблица

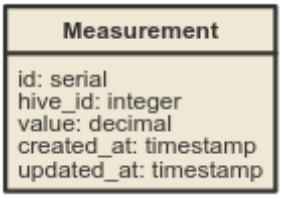


Фиг. 11 Hive обект

- `device\_id` е hardware-ния идентификатор на кошера. Той се използва при комуникация между кошера и сървъра за да се идентифицира. Не желаем да показваме хардуерния идентификатор в URL-ли, тъй като знанието на даден токен, доказва собствеността над даден кошер.
- `user\_id` е идентификатора, който бива използван като ключ (foreing key) показващ на кой потребител е дадени кошер.
- `note` е текстови стринг с който потребителя да може лесно да може да идентифицира своите кошери, по разбираем за него начин, без да му налагаме нашите идентификатори.
- Полетата `created\_at` и `updated\_at` са полета които по конвенция Rails добавя към всяка таблица и се използват, както вътрешно в рамката, така и често за дебъгване.

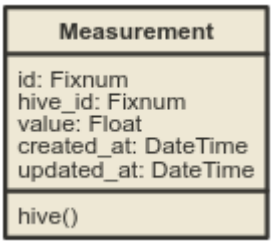
3.4.1.3 Измервания (Measurement)

За всеки тип измерване ще имаме отделен клас, съответно и таблица. Имаме възможността да събираме всички измервания в една таблица. Това би довело до намаляване на броя на заявките към базата данни, но увеличава значително големината на една таблица (което би довело до много тромави заявки) и работата която трябва да бъде свършена в Ruby, частта, което практически погледнато най-вероятно би довело до забавяне на приложението. По-долу е демонстриран общ модел на данните, където Measurement, ще може да бъде заменено със TemperatureMeasurement, HumidityMeasurement и BatteryMeasurement.

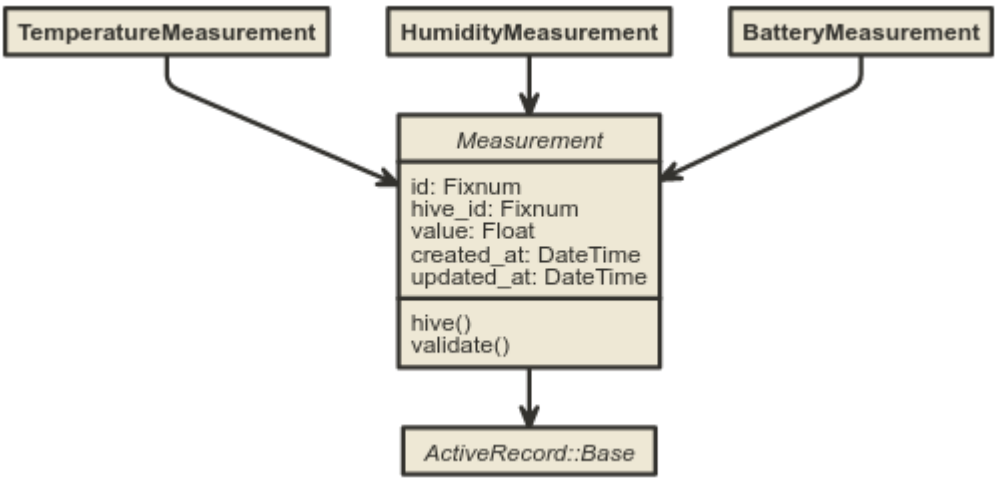


Фиг. 13 Измервания за таблицата

- id е униклания идентификатор за даденото измерване.
- hive_id е ключа с който свързваме кошер и измерване.
- value е стойността на измерването.
- Полетата created_at и updated_at са полета които по конвенция Rails добавя към всяка таблица и се използват, както вътрешно в рамката, така и често за дебъгване.

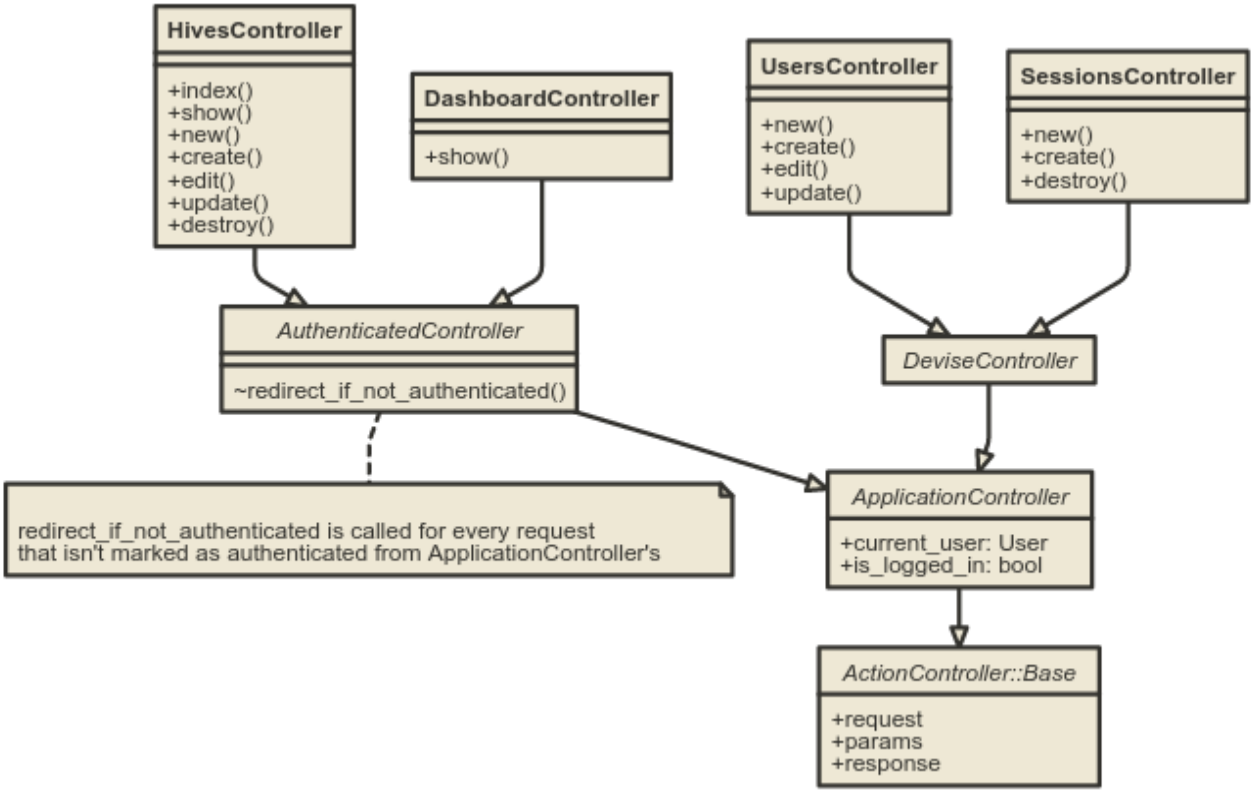


Фиг. 15 Измервания за обект от тип Hive



Фиг. 14 Стуктурата на класовете в Rails частта на приложението

3.4.2 КОНТРОЛЕРИ



Фиг. 16 Йерархия на контролерите

Контролерите са частта от кода която контролира хода на приложението и отговаря на заявките на потребителите. В Rails контролерите са класове който чиито инстанции биват създавани за да обслужат по една заявка. Това означава, че при пристигане на заявка към сървъра към даден url, сървърът създава от обект от класа на controller-а отговарящ за url-а, и вика зададения от url-а метод.

В конкретния случай когато сървъра приеме заявката, ще се създаде инстанция на UsersController, ще бъдат зададени началните стойности на променливите request, response и params.

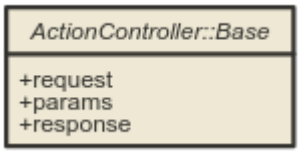
- request е обект носещ цялата информация за request-а на потребителя. Той съдържа HTTP header-ите на заявката, тялото на заявката, url-а на заявката,

бисквитките на потребителя, параметрите на заявката (Както тези от url-а, така и тези от тялото. Видими са и в params.) и много други. response е обекта представляващ отговора който бива изграждан от контролера.

- params са параметрите на заявката, в 'речник'. В примера по-горе, при пристигане на GET заявка за дадени адрес, ще бъде създадена нова инстанция на UsersController, на който ще бъде извикан метода new. В променливата с параметрите ще се съдържат: {controller: ‘users’;, action: ‘new’;, from: ‘facebook’, fid: ‘a75b42b1’ }

Разбира се, това може да бъде преконфигурирано, но това рядко се прави, тъй като една от основните идеи на Ruby е „convention over configuration“ Като начална страница на апликацията за хора без активна потребителска сесия ще действа формаза за вход, за сервирането на която отговаря действието new на SessionsController. За хора с активна потребителска сесия, началната страница ще е т.нар „табло“ (Dashboard).

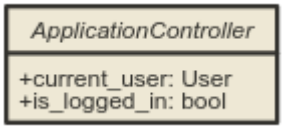
3.4.2.1 ActionController::Base



Фиг. 17 Контролер на действията

`ActionController::Base` е абстрактен контролер когото всички контролери наследяват. Той е основния контролер в приложението и върши по-голямата част от разбора на заявката. Чрез него се скрива голяма част от boilerplate кода и в контролерите, имплементирани от разработчиците, остава само код за контрол над бизнес логиката.

3.4.2.2 ApplicationController



Фиг. 18 ApplicationController

`ApplicationController` е първия клас във дървото на контролерите който се модифицира от разработчика. В него седи логика която е важна за всички контролери в приложението. Пример за това е намирането на идентификатор на потребителската сесия в бисквитките на потребителя или логиката, предпазваща ни от Cross Site Request Forgery атаки.

3.4.2.3 SessionsController

Това е контролера отговорен за създаване на на сесии.

Фиг. 19 SessionController

new и create

Действието `new`(GET) ще сервира страницата с формата за вход в приложението. При изпращане на формата от страницата `new`, информацията ще бъде изпратена до действието `create`(POST). Тук се се проверява дали в базата данни има потребител с това име. Ако няма, потребителя бива връщан на действието `new`, с допълнително съобщение, че входа е бил неуспешен заради грешно потребителско име или парола (не издаваме кое от двете е проблема).Ако съществува такъв потребител, за паролата му се проявява специално внимание. Тя не се пази ясен текст (cleartext), а е 'осолена' на базата на timestamp-а `created_at`, в което потребителя е създаден, след което бива хеширана на базата на алгоритъма (bcrypt) След като работата на алгоритъма е завършена, проверяваме дали стойността на полето `encrypted_password` е същата, като новополучения хеш. Ако това е така генерираме

нова сесия за потребителя и я записваме като бисквитка в browser-а му. Това се прави на базата на по-просто хеширане на email-а на потребителя първите 16 символа от хешираната му парола и стойността в полето `current_sign_it_at`, която се задава да е текущото време. Така потребителя вече има достъп до профила си в системата и бива изпратен на "таблото" му. Ако сравнението е било неуспешно, отново връщаме потребителя, без да издаваме на коя стъпка от автентикацията се прекъснали процеса. Ако потребител, който вече има потребителска сесия се опита да влезе в някоя от двете страници, то той ще бъде прехвърлен към потребителското му табло.

destroy

Действието `destroy`(DELETE) унищожава текущата потребителска сесия, като инвалидира стойността в полето `current_sign_it_at` на потребителя и го прехвърля към формата за аутентикация. Ако потребител който няма сесия се опита да достъпи това действие, то той пива пренасочен към формата за вход. Потребителите ще могат да достигат до това действие, чрез бутон който ще се намира в header-а на сайта и ще е видим само за потребителите с активна сесия.

3.4.2.4 UsersController

Фиг. 20 UsersController

Това е контролера който се грижи за заявките свързани със създаване и модифициране на потребители.

new u create

Действието `new`(GET) отговаря за сервирането на страницата съдържащата формата за създаването на нов потребителски профил. До него ще се достига от бутон намиращ се на началната страница на приложението. При опит за достъп до страницата от потребител с активна сесия, той бива връщан на потребителското му табло. `create` (POST) отоваря за създаването на записи в таблицата за потребители.

Паролите биват хеширани по начина описан при описанието на процеса за вписване в апликацията.

edit и update

`edit`(GET) използва същия изглед като `new`, но препопулира полетата със данните на потребителя с текущатата активна сесия. `update` (PUT) генерира UPDATE заявка за базата данни, на базата на формата подадена от потребителя в `edit` и го прилага за текущия потребител.

3.4.2.5 AuthenticatedController

Фиг. 21 AuthenticatedController

Това е абстрактен контролер, чиято основна цел е напълно за забрани достъпа на потребители без активна потребителска сесия до функционалностите, на контролерите, които наследяват от него. Това се дължи на `before_filter` функционалността която ActionController::Base предоставя и проста проверка за верността на `is_logged_in`. За всички заявки без бисквитка съдържаща потребителска сесия ще бъде извикан метода `redirect_if_not_authenticated` и потребителя ще бъде пренасочен към страницата за вход в системата.

3.4.2.6 DashboardController

Фиг. 22 Dashboard Controller

Основната цел на този контролер е да агрегира желаната от потребителя информация и да я предоставя под удобна за анализирането и форма. Единственото действие в този контролер е 'show'(GET). Динамичната част от страницата ще започва с най-актуалната информация за средните стойности на всички кошери на на

потребителя. Основната част от страницата сервирана от този контролер представява модерно изглеждаща таблица, която на всеки ред има следната информация, за всеки кошер който потребителя притежава:

- Полето 'location' на кошера, съдържащо кратък текстови низ, по който потребителя лесно да се ориентира за точно кой от кошерите му е последващата информация
- Area chart с измерванията на температурата и влажността в кошера, за зададен

период. По абцисата ще се променя времето, а по ординатата - измерените стойности.

- Gauge chart с батерията оставяща на кошера.

Между двете части, ще има форма, с два селектора за дата и час, при промяна, на които ще се изпраща нова заявка към сървъра, на базата на която ще се обновяват стойностите. За графиките ще се грижи библиотеката Highcharts. Тя приема данни в JSON формат.

```
{
        'dates': {
        'from': timestamp;
         'to':   timestamo;
        }
}
```

Листинг 3. Структура на JSON заявката

```
{
         'avarages': {
         temperature: float;
        humidity: float;
        lowest_battery: float;
         },
'hives': [
 {
```

```
 id: integer,
x: [timestamps],
y: [floats]
}, ...
]
}
```

Листинг 4. Структура на JSON отговора

Заедно с тялото на заявката показано по-горе, ще се изпращат и бисквитките на потребителя, така че да можем да идентифицираме за кои кошери ще трябва да излвечем информацията. Клик върху клетката съдържаща описанието на кошера, ще пренасочва потребителя към действието `show` на `HivesController`.

3.4.2.7 HivesController

Фиг. 23 HivesController

`HivesController` е контролера които отговаря за манипулацията и частично визуализацията на кошерите на потребителите ни. Той има следните действия:

index

`index`(GET) ще сервира страница, представляваща таблица на всички кошери, които потребителя е регистрирал. За всеки един от тях ще има бутон за редактиране на данните, показване на подробна информация за кошера, или "освобождаване" на кошера. Освен това ще има бутон пренасочващ потребителя към страница за регистрация на нов кошер.

show

`show`(GET) е действието при което визуализираме информацията за точно определен кошер на наш потребител. Идентификатора на кошера се намира в универсалния ресурсов локатор. Това id ще бъде достъпно в променливата `params` и така лесно ще можем да идентифицираме кой кошер трябва да визуализираме. Преди да го визуализираме трябва да проверм дали текущо активния потребител притежава кошера. Ако това не е така, той ще бъде пренасочен към началната страница и ще му бъде показано съобщение за грешка.

Информацията ще бъде визуализирана под формата на таблица, в която по подразбиране ще бъдат показвани последните 50 (или колкото налични има, ако са по-малко от 50), записа за даден кошер. На страницата ще има форма с дати: "от" и "до". Ако разширението на края на локатора е '.csv', ще изпратим файл с данните на потребителя в формат, който лесно се зарежда от софтуер за работа с таблици, като Microsoft Excel.

new u create

`new`(GET) е действието отговорно за сервирането на страницата с форма съдържаща две полета:

- `device_id` хардуерния идентификатор на устройството.
- `location` идентификатора по който потребителя ще разпознава устройствата.

При натискане на бутона за запис тази информация се подава на `create` действието.

`create`(POST) е действието отговорно за създаването на нови записи и пререгистрирането на други съществуващи такива. Когато получи, нов хардуерен идентификатор, системата създава нов запис в базата данни. Ако получи вече съществуващ такъв има два варианта. Ако полето `user_id` на записа със съществуващия идентификатор е нулирано, това означава че кошера е бил освободен или предотстъпен. Тогава то се запълва със идентификатора на текущия потребител и от тогава нататък, кошера става негов. Ако кошера не е бил 'освободен', то потребителя бива връщан на действието `new` с допълнително съобщение за грешка.

edit u update

При `edit`(GET) на потребителя се сервира същата форма като при new, но препопулирана с информацията достъпна в записа на дадения кошер. Полето за хардуерен идентификатор е деактивирано и не може да бъде редактирано. `update` действието прилага промените върху записа на дадения кошер.

destroy

Действието `destroy` в повечето случаи се използва за цялостно изтриване на записите от базата данни, но в случая ние няма да процедираме по този начин. При нас то ще нулира стойността на полето `user_id` на кошера и по този начин ще го освобождава. Преди полето да бъде нулирано, ще се извършва проверка, че потребителя, притежаващ дадения кошер е активния потребител.

С това завършихме всички котролери и съответните им изгледи в приложението, и заедно с това завършихме обзора на front-facing уеб апликацията.

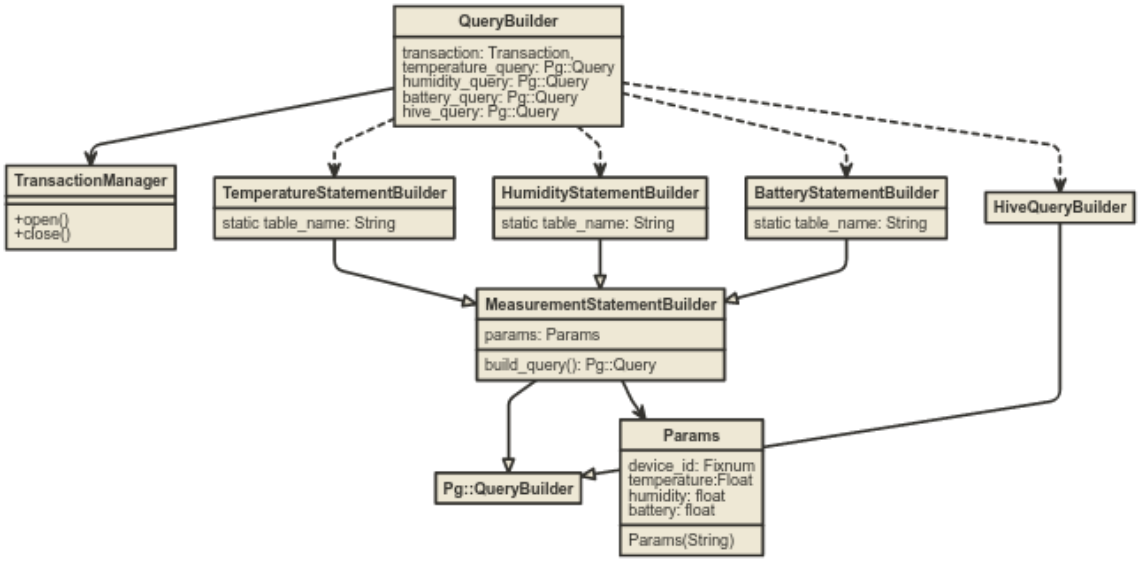
3.5 Приложение - приемник.

Приложението приемник е микро услуга (microservice) чиято единствена цел е записването на данните изпращани му от устройствата на клиентите ни. Защо го разделяме от основното приложение? Краткият отговор на този въпрос е прост - бързодействие. Ако го направим част от основната Rails апликация,то за всяка една заявка ще минава през много филтрация. Дори самият Rails, без библиотеките за автентикация и допълнителните филтри които програмистите дефинират, прави твърде много операции върху заявката преди да я подаде на някой от контролерите. Затова за това просто приложение няма да използваме Rails, а една минималистична библиотека, която особено през последните няколко години, набра огромна популярност, наречена Sinatra.

3.5.1 SINATRA

Sinatra е свободна open-source библиотека за изграждане на уеб апликации със собствен domain-specific language. Тя, също както Rails, разчита на Rack, като интерфейс за работа с web server, но тъй като се стреми към простота и минимализъм, не използва MVC модела. За комуникация със сървъра ще използваме директно библиотеката Pg, която действа като database driver и в Rails апликацията. Там обаче тя е скрита зад ActiveRecord.

3.5.2 СТРУКТУРА НА АПЛИКАЦИЯТА



Фиг. 24 Класова структура

Апликацията има само един end-point, а именно този за предаване на информация от устройството, който се намира на / . Сървърът ще отговаря само на POST заявки към този адрес. При получаване на некоректно форматирана заявка, сървърът ще отговаря със 422 Unprocesable Entry. Когато сървърът получи информация от кошер който вече е регистиран, то той прочита заявката, отваря транзакция към базата данни, прави записи нови записи за измервания на температурата, влажността и батерията, запазва ги и затваря транзакцията. Когато сървърът получи информация от кошер който все още не е регистриран, то той ще прави запис в базата данни за този кошер, с user_id 0. Така в последствие той да може

да бъде регистриран от потребителя притежаващ го. След това продължава по нормалната процедура.

4 Управление на конфигурацията и инструменти за дистанционно управление

Едни от проблемите, които биха могли да възникнат при изграждането на проекта, са такива, свързани с успешните реализация, поддръжка, конфигурация и управление на множество сървъри без ненужното изразходване на изключително количество финансови и времеви ресурси. Начинът по който избираме да разрешим този проблем е използването на инструменти за автоматизирано снабдяване (automated provisioning), или така наречените инструменти за конфигурационно управление (Configuration Management –CM) и дистанционно изпълнение (Remote Execution – RE). Този тип инструменти позволяват на администратора да извърши определена дейност на няколко различни сървъра едновременно, а също и да разгърне многобройни приложения с минимално действие. В общи линии, те улесняват оформянето и поддръжката на десетки, стотици и даже хиляди на брой сървъри.

В днешно време можем да открием голямо количество такива инструменти, заливащи пазара, всеки със своите силни и слаби страни. За нас е важно да изберем най-добрия за нашия проект, за да можем да решим максимално ефективно вече споменатия проблем и да сведем до минимум вероятността за възникване на други такива, а също и да постигнем максимално бързо и лесно решаването им. В крайна сметка се стига до избор между две от най-използваните приложения, що се отнася до Automated Provisioning. По-долу ще разгледаме техните характеристики и ще обясним основните им предимства и недостатъци, а накрая ще обосновем решението си, а именно, защо избираме едното от тях пред другото.

Puppet е един от първите и най-старите инструменти в тази сфера, като е на пазара от 2005 година насам. Точно на него се спират и изключително голяма част от най-могъщите компании в различните браншове, като малка част от тях са, например, google, Reddit, Dell, PayPal, Oracle, Los Alamos Labs и Стандфордския Университет. Тези именити клиенти показват високото качество на продукта, както и стандарта, който поддържа Puppet вече дълги години. Приложението е с отворен код и е разработено чрез програмния език Ruby. Предимствата с които могат да се похвалят от Puppet са напълно развитият и удобен интерфейс, както и поддръжката на всички основни операционни системи – Linux, UNIX, Windows и дори MAX OS X. Инсталацията е изключително лесна за осъществяване. Puppet поддържа, както команден интерфейс, базиран на Ruby, така и графичен интерфейс.

Недостатъците на вече отдавна възникналия Puppet, също са никак не малко. На първо място, трудно можем да го определим, като особено гъвкав или пък „пъргав“. По конкретно, отстраняването на наличните т.нар. „бъгове“ отнема прекалено много време, а многократните искания от страна на клиентите на Puppet за нововъведения или подобрения обикновено биват игнорирани. Също така, голямо недоволство предизвиква подтикът от страна на PuppetLabs към преминаването към комерсиалната версия на продукта.

Ansible, от своя страна, е от по-късните конкуренти на Puppet, като започва работата си чак през 2012 година. Разработен е на Python и също е с отворен код. Не поддържа Windows и като по-нов продукт, не предлага такова ниво на комфорт и удобство, като своите по-опитни конкуренти. Все пак, Ansible има какво да предложи на своите клиенти, в лицето, както на големите, така и на по-малките компании. На първо място, благодарение на Python, като вграден в повечето UNIX и Linux системи, Ansible може да се похвали с отлично бързодействие. Също така е много лек и не се налага инсталирането на допълнителни „агенти“, които да натоварват системата. Може да бъде използван през командния интерфейс, като не се налага използването на конфигурационни файлове за изпълнението на по-обикновените и елементарни

функционалността. Този интерфейс приема команди на всякакъв програмен език, което освобождава клиента от ограничаването до един такъв.

Разбира се, колкото и способен да изглежда Ansible до тук, той притежава други важни недостатъци, които няма как да не вземем под внимание. Не бива да се пренабрегва липсата на поддръжка на Windows системи. Основен, в нашия случай, проблем би могло да се окаже не до там развитият графичен интерфейс, предлаган от Ansible. Освен това той е разграничен от командния интерфейс, което налага честото синхонизиране и на двата, за да може да представят едни и същи данни.

Въпреки че е все още нов продукт и е смятан за удобен само за по-малки или временни проекти, Ansible постепенно става избор на повече и по-големи компании, което следва от високият професионализъм, големият ентусиазъм и стремежът към подобрение на продукта. Също така се работи усърдно над подобряването на графичният интерфейс, което дава своя резултат. Повечето гъвкавост и доброто развитие , от страна на Ansible, както и по-експедитивното извършване на всички негови функции, ни кара да изберем да работим точно с него. Ние вярваме, че Ansible е изборът на бъдещето и че той ще реши споменатият още в началото проблем, който ни подтикна към използването на автоматизирано снабдяване. Смятаме че направеният избор ще улесни в пъти нашата работа и ще осигури високото качество и благонадеждността на нашата сървърна система.

5 Front facing server

За реализирането на прокта, се нуждаем от надежден уеб сървър, който да съдържа, обработва и доставя уеб страниците до клиентите. Добрата комуникация между клиент и сървър, която се осъществява посредством HTTP (Hypertext Transfer Protocol), е изключително важна за нас. Точно за това ще изберем един измежду двата най-популярни уеб сървъри с отврен код – Apache и Nginx. Заедно, те отговарят

за обслужването на над 50% от трафика в интернет, което е показателно и доказва благонадеждноста на тези два продукта.

Вместо да имплементира единична архитектура, Apache предлага набор от така наречените „многообработващи модули“ (MultiProcessing Modules) или просто MPMs. Това позволява по-добро съответствие в изискванията на всяка специфична инфраструктура. Това подсказва, че изборът на точните MPMs е изключително важен! Apache се облагодетелства от своята добра документация и интегрирана поддръжка от други софтуерни проекти. Администраторите обикновено избират тази технология заради нейните гъвкавост, способност и широкоразпространена поддръжка. Може да обработва голям брой интерпретируеми езици без да се налага да се свързва с отделен софтуер. С няколко думи, Apache осигурява гъвкава архитектура за избор на различни връзки и алгоритми за обработка на заявките.

Една от основните силни страни на Apache, а именно, силата да модифицира конфигурацията си, се оказва и един от големите недостатъци, тъй като може да доведе до сериозен риск в сигурността, ако не се изпълни правилно. Създаването на персонализиран протокол, пък, означава възникването на различни проблеми (бъгове) в системата. Apache изисква стриктна система за актуализация, която трябва да премине без пропадния, което може да бъде твърде неудобно.

Nginx е по-нова от Apache система, която обръща по-голямо внимание на проблемите със съгласуването, пред каквито се изправят разрастващите се сайтове. Конструирано е по начин, който позволява използването на асинхронни, неблокиращи, event-driven алгоритми за управление на връзки. Nginx създава работни процеси, всеки от които може да обработва хиляди връзки. Те постигат това, имплементирайки бърз механизъм, тип – цикъл, който регулярно проверява за процеси и ги управлява. Това позволява на работните процеси да се занимават с някоя връзка, само ако е настъпило ново събитие. Така се постига висока

производителност и разрастване на обема работа, и то използвайки ограничени ресурси. Дори и в моменти в които трябва да се извърши по-тежка от обикновено работа, нивата на използваните памет и CPU остават относително постоянни. Nginx изпъква и в обслужването на статично съдържание, като отново се справя изключително бързо с тази задача.

Ngnix може да се определи, като значително по-бърз, лек и ефективен, спрямо използваните ресурси, сравнен с Apache. Въпреки, че няма такъв голям набор от функционалностти, Ngnix изпълнява най-жизненоважните такива и го прави значително по-бързо и ефикасно от своите конкуренти. Няма причина да разполагаме с хиляди функции ако имаме нужда от само няколко, точно за това се спираме на Nginx, като по-надежден, по-обновен, бърз и отговарящ на нашите изисквания спрямо сигурността.

6 Сървърна структура

В тази част, ще обясним точно как е устроена нашата сървърна структура, за какво служи всеки един от съвърите, а също, какво поддържа и как спазваме изискванията за сигурност. Първо имаме нужда от сървър, който да хоства нашият Nginx – front facing. Вече се запознахме с характеристиките на Nginx и уеб сървърите. Целта на този сървър е всяка заявка към което и да е от приложенията ни, а минава първо през него. Той ще служи за обратно прокси (reverse proxy), тоест ще предава заявки от името на клиента към един или повече сървъри. Обикновено обратното прокси стои за firewall-а в частните мрежи и насочва заявките на клиента към правилните сървъри в по-задно ниво. Така се постига допълнително ниво на абстракция и контрол, за да се осигури гладкият поток на интернет трафика между клиентите и сървърите. Този сървър ще отговаря също и за поддръжката на TLS (Transport Layer Security). Това е криптографски протокол, който осигурява сигурност при комуникацията между две компютърни апликации. При вътрешна

комуникация, нашият първи сървър премахва TLS и тя ще се осъществява директно по HTTT (Hypertext Transfer Protocol), тъй като не искаме да имаме overhead-а на TLS и във вътрешната ни система. На същият този сървър ще стои и приложението – приемник.

Вторият основен сървър ще отговаря за базата данни, която ще бъде управлявана от PostgreSQL, както вече обяснихме. Всяко едно от двете приложения които комуникират с нашия database сървър ще има собствен акаунт. Приемника има право на достъп (четене и писане) до таблиците „hives" и таблиците за измервания, докато основното приложение има достъп до всички таблици, освен вътрешните конфигурации за PostgreSQL. На последно място имаме сървъра на потребителското приложение.

7 Уеб хостване

Задължителна част от плана за нашия проект е изборът на услуга за уеб хостване. Няма начин да избегнем този избор, при положение че имаме за цел наш сайт да е част от интернет пространството. За това ще разгледаме два от най-благонадеждните доставчици на тази услуга, а именно Heroku и SiteGround.

7.1 Обща характеристика и функционалност на SITEGROUND

SiteGround е популярен и уважаван доставчик на уеб хостинг услуги. Той предлага точно тези отлични скорост и сигурност от които се нуждаем, а също и голям набор от изключително полезни функционалностти, като например CDN, автоматична актуализация, GIT контрол, кеширане и други. SiteGround предлага ежедневна 24-часова техническа поддръжка, неограничени трафик, имели и бази данни. Безплатни конфигуриране, дневен backup и трансфер, а и SSH – достъп, също са част от функционалността, която предлага SiteGround.

7.1.1 ДРУГИ ОСНОВНИ ПРЕДИМСТВА

Такива са първо, че работи с CloudFlare, което води до подобрени сигурност и производителност. Клиентите, пък, могат да изберат местоположението на сървърите, което може да бъди изключително удобно. SiteGround предлага и полезни материали за поддръжка. На последно място, но не и по важност, ще споменем отново високото ниво на сигурност, надеждност и цялостни услуги спрямо клиента.

7.1.2 НЕДОСТАТЪЦИ

SiteGround, за жалост, не предлага неограничена памет или Bandwidth. В ценови аспект, липсата на месечни планове, също може да ни накара да изберем друг доставчик. Цените все пак ще разгледаме в следващия абзац!

7.1.3 ЦЕНОВИ УСЛОВИЯ

SiteGround, предоставя на своите бъдещи клиенти избора измежду няколко различни месечни плана за споделено уеб хостване. Първият и най-обикновен, а и евтин план – StartUp, е на цена 3.95$ месечно, включващ един уебсайт, 10GB интернет пространство и всички най-съществени функции. Подходящ е за до около 10,000 посещения всеки месец. GrowBig планът, пък, е на цена от 6.45$ и предоставя 20GB пространство. Позволява до 25,000 посещения месечно, а освен основните функции, включва и допълнителни такива. Третият и последен план, който ще разгледаме – GoGeek е на цена от 11.95$. С 30GB пространство, над 100,000 посещения месечно и всички функционалностти, които SiteGround може да предложи, това е планът от най-висок клас, като разбира се е и с най-висока цена.

7.2 Обща характеристика и функционалност на HEROKU

Heroku е облачно базирана, сървърна технология, която позволява лесно управление и развитие на уеб приложения. Тя може лесно да се разраства, заедно с разрастването на приложението, чрез подобрение на базата данни и увеличение на количеството работа. Heroku е облачна платформа, което означава, че не трябва да се

тревожим за инфраструктурата и можем да се съсредоточи върху приложението. Можем да се възползваме от добре написаното упътване и да започнем буквално за минути. Heroku, за разлика от SiteGround, предоставя 750 безплатни изчислителни часа, което означава, че можем да разполагаме с единични процеси на абсолютно никаква цена. Производителността също е отлична, като, например, едно обикновено node.js приложение би могло да се справи с около 60-70 заявки в секунда.

7.2.1 ДРУГИ ОСНОВНИ ПРЕДИМСТВА

Незабавно вдигане на приложението се осъществява чрез Git push и различни скриптове. Heroku предлага множество различки добавки и ресурси. Също така, разрастването на приложението не би могло да повлияе на функционалността и производителността. Всички процеси са изолирани един от друг и така се постига високо ниво на абстракция. С Heroku можем да се възползваме от много добрата видимост, като имаме лесен достъп до всеки изход от всеки компонент и процес на приложението.

7.2.2 НЕДОСТАТЪЦИ

Първият е, че сме напълно обвързани с Heroku/EC2 и неговата стабилност. Ако то спре да работи, нашето приложение също спира. Въпреки че предлага безплатен start-up план, Heroku, като цяло е по-скъпа от обикновено услуга. Все пак цената спрямо качеството на тази услуга си заслужава. Безплатната версия пък може да се забави с до половин минута за отговор.

7.2.3 ЦЕНОВИ УСЛОВИЯ

Heroku, както вече споменахме, предлага безплатен план, който е идеален за експериментиране и изпробване на облачните приложения, с ограничени налични ресурси. Този план „заспива“ при над 30 минути липса на активност и използва индивидуални домейни. Друг план е така наречения Hobby. Той е идеален за слабо разрастващи се приложения, или пък лични проекти. Той е на цена от 7$ месечно, никога не „заспива“ и има всички основни функции. Предоставя безплатен SSL и автоматизиран сертификат за управление на индивидуални домейни. Следват

професионалните планове, които са по-мощни и по-скъпи, като се простират от 25$ до 500$ на месец, в зависимост от конкретното желание и избор на функционалностти от страна на клиента. Стандартният такъв има подобрена видимост, производителност и възможност за засилване популярността на приложенията. Вторият се съсредоточава върху изпълнителността и производителността, точно в най-критичните моменти и в моменти на силно разрастване на приложението, или пък при висок трафик.

8 Анализ, оценка и управление на риска

Оценка	Възможност/Вероятност	Въздействие
1	Почти невъзможен	Незначително въздействие
2	Не много вероятен	По-ниско от средното въздействие
3	Средна вероятност (50%)	Средно въздействие
4	Над средната вероятност	Над средното въздействие
5	Определено,вече настъпило събитие или почти сигурно	Катастрофа

Таблица 2. Класификация на оценките за риск анализ

Рейтинг = Вероятност * Въздействие

Класификация на рисковете:

- Нисък рейтинг – от 1 до 5
- Среден рейтинг – от 6 до 9
- Висок рейтинг – над 10

Идеята на класификация на рисковете е нуждата от специфичен план на изпълнение на управлението на риска. За да бъде създаден такъв план е нужно да се даде приоритет на рисковете и според този приоритет те да бъдат разрешени. В Табл.3 рисковете са сортирани по рейтинг. В същият ред се дава и приоритета на разрешаване на проблемите за съответния риск.

№	Риск	Вероятност	Въздействие	Рейтинг	План за управление
1	Продуктът не отговаря на очакванията на клиента	3	5	15	Консултации с клиента. Идентифициране на силни и слаби страни на продукта с цел подобрение
2	Забавяне от страна на подизпълнителна фирма	3	3	9	Предварително се закупуват допълнителни материали.
3	Климатични условия	3	3	9	Следене на прогнози за времето. Уведомяване на клиента
4	Неоткриване на грешките в приложението	2	4	8	Изготвяне на нов прототип. Повторно тестване
5	Членовете на екипа не са запознати с използваните технологии	2	4	8	Допълнително обучение насочено към използваните технологии
6	Технически дефект	2	4	8	След установяване на дефекта се подменя устройството
7	Подценяване на времето за разработка на продукта	3	2	6	Преразпределение на процесите и предвидените периоди за изпълнението им
8	Софтуерна атака	1	5	5	Допълнителна защита и повторно криптиране на данните.
9	Да отпадне необходимостта от продукта	1	5	5	Проучване и анализ на пазара.
10	Неефективна комуникация	2	2	4	Преразглеждане на плана за комуникация. Изискване на повече срещи
11	Изпреварване на пускането на продукта от конкуренти	2	2	4	По-добра реклама. Нови функциналности

Таблица 3. Оценка и управление на рисковете

9 Scrum

Обща информация

Scrum е итеративна, инкрементална рамка за управление на проекти. Scrum процесът се състой от отделни итерации, наречени спринтове. Спринтовете имат продължителност от една до четири седмици и в края на всеки спринт работещият екип разполага с работеща версия на проекта.

Ключови роли:

- **Собственик на продукта** – той е отговорното лице и прави връзката с клиента. Неговата задача е да комуникира с клиента и според обсъденото да поставя приоритет на задачите, които трябва да се извършат.
- **Scrum ръководител –** той е отговорен за контрола на изпълнението на задачите, т.е. той следи дали се постигат поставените цели за дадените спринтове и се грижи да държи екипът фокусиран върху разработката на даден продукт.
- **Екип разработка** – това е екипът, който създава продукта, анализира, създава архитектура, пише кода, тества го, документира го и т.н. Обикновено той се състой от три до девет души. Екипът е длъжен да изпълнява зададените цели за всеки спринт. Те се самоорганизират – никой не им разпределя задълженията освен самите тях, важното е да е постигната желаната цел за дадения спринт.

Всеки ден се провеждат срещи с продължителност от пет до петнадесет минути (stand-up meetings). По време на тези срещи екипът има за цел да съгласува докъде е стигнала работата през съответния спринт. Всеки член на екипа трябва да отговори на три въпроса:

1. Какво е работил по проекта предишния ден?
2. Какво планира за предстоящия ден?
3. Какви проблеми е срещнал по време на работа?

След завършване на даден спринт се провеждат две срещи:

- Среща за обзор на спринта – по време на тази среща се преглежда работата, която е била свършена и работата, която не е била свършена. Представя се демо на свършената работа досега на клиента.
- Ретроспекция на спринта – по време на тази среща екипа обсъжда предишния спринт като се задават два основни въпроса:
    - Какво мина добре по време на спринта?
    - Какво може да се подобри в процеса на работа за следващия спринт?

За нашия продукт

За нашия продукт сме избрали фрейм за разработка да е Scrum. Причината за този избор е проста – Scrum предлага силен и качествен контрол над процеса на работа по продукта. Допълнително, той разрешава във всеки даден момент, при спиране на разработка или незабавно искане за продажба на продукта, да бъде спряна работата, тъй като има работещ продукт в края на всеки спринт. Периодът определен за един спринт е две седмици. Ежедневните срещи на екипа (stand-by meetings) се провеждат онлайн през Google Hangouts.

Роли:

- Собственик на продукта – Николай Коцев
- Scrum ръководител – Димитър Димитров
- Екип разработка – Николай Коцев, Никола Николов, Иван Филипов

Библиография

[GH+94]E. Gamma, R. Helm, R. Johnson,J. Vlissides, „*Design Patterns: Elements of Reusable Object-Oriented Software“,* 1994

[Ore99] O'Reilly Media, „Programming Embedded Systems in C and C++”, January 1999

[Ore11] O'Reilly Media, Arduino Cookbook, 2nd Edition, December 2011

[MC13] Adrian McEwen, Hakim Cassimally, „Designing the Internet of Things“, Nov 2013

[TSD17]Dave_Thomas, Sam_Ruby, David_Copeland, „Agile Web Development with Rails 5.1“, 2017

[GITHB] Github lectures from FMI’s IOT course: https://github.com/vlast3k/FMI_IoT_2017, 18.01.2018
[TGRM] telegram website: https://telegram.org/, 18.01.2018
[ADSW] arduino website about software: https://www.arduino.cc/en/Main/Software, 19.01.2018

[PUPPT] puppet.com, 26.01.2018

[ANSBL] ansible.com, 26.01.2018

[UPGGR] upguard.com/articles/ansible-puppet, 26.01.2018

[NGIX] nginx.com/resources/glossary/reverse-proxy-server/, 27.01.2018

[FINCS] reviews.financesonline.com/h/siteground/, 28.01.2018
[SITGR] https://www.siteground.com, 28.01.2018
[HERO] heroku.com, 28.01.2018
[GITHB] gist.github.com/taybenlor/3684133, 28.01.2018