Spis treści

[1 Wstęp 4](#_Toc405341629)

[1.1 Cel i założenia projektu 5](#_Toc405341630)

[2 Środowisko sprzętowe 6](#_Toc405341631)

[2.1 Raspberry PI 6](#_Toc405341632)

[2.2 ATmega 328 8](#_Toc405341633)

[2.3 nRF24l01+ 9](#_Toc405341634)

[3 Środowisko programowe 9](#_Toc405341635)

[3.1 Raspbian 9](#_Toc405341636)

[3.2 Visual Studio 10](#_Toc405341637)

[3.3 Wykorzystane technologie 11](#_Toc405341638)

[3.3.1 Platforma .NET 11](#_Toc405341639)

[3.3.2 Mono 11](#_Toc405341640)

[3.3.3 AngularJS 11](#_Toc405341641)

[3.3.4 Typescript 12](#_Toc405341642)

[3.3.5 MySql 12](#_Toc405341643)

[4 Implementacja 13](#_Toc405341644)

[4.1 Konfiguracja platormy RaspberryPi 14](#_Toc405341645)

[4.2 Warstwa prezentacji 14](#_Toc405341646)

[4.2.1 AutoMapper 15](#_Toc405341647)

[4.2.2 Unity 16](#_Toc405341648)

[4.2.3 Architektura MVC 17](#_Toc405341649)

[4.2.4 Kontroler 17](#_Toc405341650)

[4.2.5 Model 18](#_Toc405341651)

[4.2.6 Widok 18](#_Toc405341652)

[4.3 Warstwa aplikacji 21](#_Toc405341653)

[4.3.1 DeviceDomain 23](#_Toc405341654)

[4.3.2 AuthenticationDomain 26](#_Toc405341655)

[4.3.3 HistoryDomain 27](#_Toc405341656)

[4.3.4 DomainEvents 28](#_Toc405341657)

[4.4 Warstwa komunikacji 28](#_Toc405341658)

[4.4.1 WebSocket 28](#_Toc405341659)

[4.4.2 SerialPort 30](#_Toc405341660)

[4.5 Warstwa harmonogramu 32](#_Toc405341661)

[4.6 Warstwa dostępu do bazy danych 34](#_Toc405341662)

[5 Wnioski 35](#_Toc405341663)

[6 Bibliografia 37](#_Toc405341664)

[7 Spis rysunków 38](#_Toc405341665)

# Wstęp

Automatyka budynkowa jest dziedziną, która istnieje od wielu lat. Już w latach 60 ubiegłego wieku pojawił się pomysł inteligentnego budynku, chociaż sam termin powstał w roku 1984. W latach 70 powstała potrzeba zbierania danych i zarządzania energią w wieżowcach w Stanach Zjednoczonych, by móc rozliczać najemców niezależnie od siebie. Od tego był tylko krok do tworzenia inteligentnych instalacji w mniejszych budowlach.

Inteligentny budynek dostarcza niezwykłego komfortu, niespotykanego w domach o standardowych instalacjach. Dzisiaj zwraca się olbrzymią uwagę na ekologię oraz koszty utrzymania. System zarządzania inteligentnym budynkiem daję możliwość lepszego zarządzania zasobami. Poza komfortem i dbaniem o środowisko największą zaletą jest zwiększenie bezpieczeństwa mieszkańców. Inteligentny budynek posiada wiele sensorów dbających o to, by wszystko działało prawidłowo. Mogą one wykryć ogień, wyciek gazu, a także włamanie i zachować się adekwatnie do sytuacji. Na przykład uruchomić alarmy dźwiękowe i świetlne, a także powiadomić odpowiednie służby. Dom nie jest już tylko biernym elementem, ale wręcz żywą istotą, dostosowującą się do naszych zachowań, oraz trybu życia.

Zwłaszcza w ostatnich latach automatyka budynkowa gwałtownie rozkwita. Zaczęły się nią interesować firmy, które dotąd obejmowały zupełnie inny segment rynku. *Google* wykupiło firmę *Nest,* producenta inteligentnych termostatów. Niedawno zapowiedziało także zunifikowany system, wraz ze współpracą największych producentów nie tylko sprzętu RTV i AGD, ale także samochodów. Pojawiła się idea, a nawet ruch zwany *Internet of Things* [1]. Opiera się na idei rzeczy które są podłączone to internetu. Wysyłają one, przetwarzają i odbierają wszelkiego rodzaju informacje. W dzisiejszych czasach, dzięki postępowi miniaturyzacji, mikrokontrolery i czujniki są instalowane w ubraniach, biżuterii, a nawet ludzkim ciele. Na stronie firmy *Cisco* można znaleźć licznik elementów podłączonych do sieci [2]. Na dzień dzisiejszy wskazuje on ponad 14 miliardów. Ilość tego typu urządzeń rośnie diametralnie i przewiduje się, że licznik w roku 2020 sięgnie 50 miliardów. Inteligentny budynek jest zbiorem tego typu elementów, które ze sobą współpracują i wymieniają informacje. Dzięki podłączeniu go do Internetu zyskujemy nieograniczone możliwości. Możemy kontrolować wszystkie urządzenia z drugiego końca świata, albo na przykład wykorzystywać chmurę do skomplikowanych obliczeń, którym nie podołają domowe komputery. Wszystko to oznacza olbrzymie ilości wytwarzanych informacji, które mogą zostać zanalizowane i wykorzystane, by uczynić życie wygodniejszym i bezpieczniejszym.

Dziedzina inteligentnych budowli rozwija się z niezwykłym rozmachem, niestety głównie w krajach zachodnich. Nasz rodzimy rynek nie jest tak rozwinięty, a ilość budynków korzystających z dobrodziejstw automatyki budynkowej jest niewielka. Chciałbym za pomocą tego projektu rozpropagować idee inteligentnego budynku, oraz pokazać, że jest możliwość stworzenia systemu nie ingerującego w gotową instalacją naszego domu i dającego duże możliwości konfiguracji, przy jednoczesnym zachowaniu małego budżetu.

Inteligentnymi budynkami interesuję się od ponad roku. Przez poprzedni semestr brałem udział w zajęciach koła Automatyki Budynkowej, gdzie poznałem wiele stosowanych na dziś rozwiązań.

Wiele firm tworzy urządzenia do inteligentnych budynków. *ABB*, SIEMENS, *Satel*, *Gira* i *Theben* to tylko część z nich. Istnieje także wiele protokołów, za pomocą których urządzenia mogą się ze sobą komunikować. Jednym z największych systemów, zwłaszcza w tej części świata, jest niemiecki *KNX* [3]. Jest to system bardzo dojrzały i niezawodny. Powstał w latach 90 ubiegłego wieku, początkowo nosząc nazwę *EIB* (ang. *European Installation Bus*). Każde urządzenie musi spełniać określone wymagania i mieć certyfikat od stowarzyszenia *KNX*. Testy urządzeń trwają wiele miesięcy. Protokół *KNX* posiada wiele zalet, takich jak bycie systemem rozproszonym, który w przeciwieństwie do systemów scentralizowanych, nie posiada jednostki sterującej wszystkimi urządzeniami. Koszt, takiego systemu jest wyższy, jednak posiada ważną zaletę. W momencie awarii jednego urządzenia, reszta systemu może sprawnie funkcjonować. System dopuszcza możliwość podłączenia 57484 urządzeń. Jest to ogromna liczba i z tego powodu *KNX* jest wykorzystywany do zarządzania wieżowcami, fabrykami, a nawet miastami.

Kolejnym dużym systemem jest *X10*, którego cechą charakterystyczną jest wymiana informacji pomiędzy urządzeniami, za pośrednictwem istniejącej już sieci zasilającej. Wadą jest duży pobór prądu przez urządzenia, a także wprowadzanie zakłóceń do sieci elektrycznej.

Konkurencją dla protokołów typu *KNX, czy X10* są protokoły bezprzewodowe. Największe to *ZigBee* i *Z-Wave*. Drugi z wymienionych protokołów został zaimplementowany przez Polską firmę *Fibaro*, która propaguje idee bezprzewodowych instalacji automatyki budynkowej. Dużą zaletą urządzeń tworzonych przez tą firmę jest ich wygląd, energooszędność, a także niewielkie rozmiary.

## Cel i założenia projektu

Celem projektu było stworzenie systemu zarządzania inteligentnym budynkiem, wraz ze wszystkimi jego elementami. W skład tych elementów wchodził serwer nadrzędny, serwer zarządzający multimediami, oraz urządzenia wykonywujące różne prace, charakterystyczne dla inteligentnego budynku.

Motywacją do stworzenia tego projektu był wysoki koszt dostępnych na rynku rozwiązań. Projekt miał się charakteryzować niewielką ceną, oraz możliwością implementacji do każdego rodzaju budynku, bez ingerencji w instalację elektryczną. Jest to olbrzymia wada większości dostępnych na rynku rozwiązań, które oprócz ceny, wymagają położenia dodatkowych linii służących do zasilania i komunikacji pomiędzy urządzeniami.

W zakres projektu zawierało się zebranie wymagań, zaprojektowanie architektury, a następnie jego wdrożenie i przetestowanie.

Jednym z głównych założeń projektu było oparcie serwerów na komputerze *Raspberry Pi*, który opiera się architekturze *ARM (*ang. *Advanced RISC Machine*), jako taniej, niewielkiej i energooszczędnej alternatywie do serwerów opartych na architekturze x86/x64. Należało przygotować przykładowe urządzenia, które będzie można podłączyć i zaprogramować z poziomu interfejsu strony *WWW* (ang. *World Wide Web*). Celem było także danie użytkownikowi możliwości tworzenia między nimi zależności i powiązań. Należało znaleźć i zaimplementować sposób komunikacji pomiędzy serwerem zarządzającym, a urządzeniami. Zdecydowano się na wykorzystania fal radiowych. Protokoły bezprzewodowe dają olbrzymie możliwości, często pokonując swoje przewodowe odpowiedniki w cenie i prostocie instalacji.

Kolejnym elementem projektu był serwer multimedialny, który pozwalał na uruchamianie plików multimedialnych, takich jak filmy, czy muzyka. Częścią systemu zarządzania multimediów był także system „ambilight”, odpowiadający za odpowiednie podświetlanie monitora, barwami odpowiadającymi tym, które są wyświetlane na ekranie. Powoduje on optyczną iluzje poszerzenia ekranu. Jest to rozwiązane montowane fabrycznie w niektórych telewizorach.

Częścią projektu, za którą byłem odpowiedzialny było stworzenie serwera nadrzędnego sterującego wszystkimi elementami składającymi się na ten system. Serwer miał zostać oparty na *Raspberry Pi* i mieć możliwość komunikacji zarówno z serwerem multimediów jak i podłączonymi urządzeniami. Serwer będący sercem systemu, miał za zadanie dać użytkownikowi całkowitą kontrolę nad konfiguracją systemu i podległymi mu urządzeniami. Do jego obowiązków należy także zbieranie od nich informacji i przedstawienie ich w przystępny sposób.

# Środowisko sprzętowe

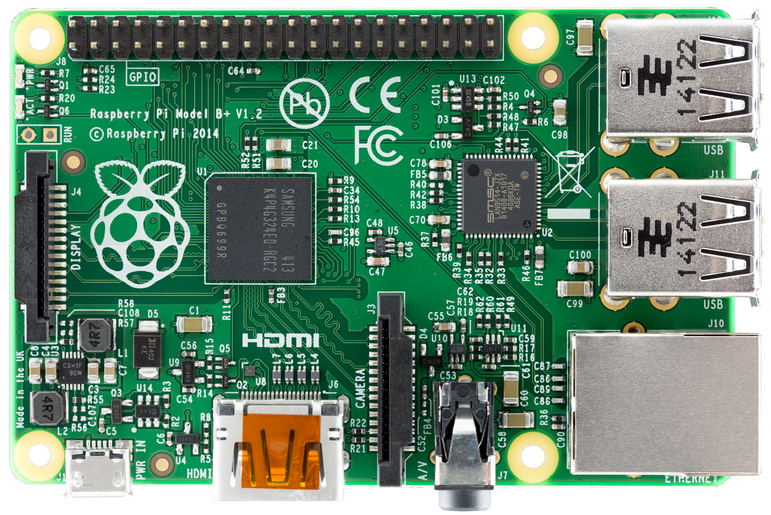
W tym rozdziale zostaną przedstawione środowiska sprzętowe na jakich oparto cały projekt. Są to komputery *RaspberryPI,* będące platformą, na jakiej działają serwery, mikrokontrolery firmy ATMEL, będące sercem wszystkich urządzeń, oraz moduły nRF24l01+ odpowiadające za komunikację pomiędzy serwerem a urządzeniami.

## Raspberry PI

Serwer zarówno nadrzędny, zarządzający automatyką budynkową, jak i serwer multimedialny są oparte na platformie komputerowej Raspberry Pi. Jest to sprzęt stworzony przez *Raspberry Pi Foundation,* jego premiera odbyła się 29 lutego 2012 roku. Celem utworzenia fundacji była edukacja młodego pokolenia w programowaniu, oraz nauki obcowania ze sprzętem za ułamek dotychczasowej kwoty. W kwietniu tego roku fundacja ogłosiła wsparcie w postaci miliona funtów brytyjskich na rozwijanie projektów promujących zrozumienie technologii, oraz rozwijania kreatywności wśród dzieci.

Do dnia pisania pracy wyszło kilka modeli tego urządzenia. Najnowszym modelem jest *Raspberry Pi B+* i to na nim oparty jest serwer nadrzędny. W stosunku do pierwszych wersji urządzenia została zwiększona ilość *GPIO* (ang. *General Purpose Input/Output*), czyli wyprowadzeń, mogących pełnić role zarówno wejść jak i wyjść, służących do komunikacji z innymi elementami systemu, czy urządzeniami peryferyjnymi. Model ten także pobiera mniej energii, oraz ma większą ilość wyjść z interfejsem *USB* (ang. *Universal Serial Bus*).

Urządzenie ma bardzo małe rozmiary, jego wymiary to ledwie 85 mm długości i 56 mm szerokości. Jest także całkowicie bezgłośne, gdyż nie wymaga żadnego rodzaju chłodzenia. Ważnym elementem jest koszt urządzenia. Oficjalna cena producenta wynosi 35$.



Rys. 1 Raspberry Pi

Niestety, aby osiągnąć te możliwości, twórcy *Raspberry Pi* użyli podzespołów, których wydajność jest nieporównywalnie niższa, niż tych znajdujących się we współczesnych komputerach.

*Raspberry Pi* jest wyposażone w *SoC* (ang. *System-On-Chip*) *Broadcom BCM2835*, który posiada procesor 700 MHz i jest oparty na architekturze ARM11. Komputer posiada stosunkowo wydajny procesor graficzny *Broadcom VideoCore IV,* zdolny do płynnego odtwarzania filmóww rozdzielczości 1080p*.* Użyta w projekcie wersja *Raspberry Pi* ma 512 MB pamięci RAM. Cały mikro-komputer jest zasilany przez port *microUSB* o napięciu zasilania 5V.

Dlaczego to urządzenie zostało wybrane, jako podstawa projektu? Serwer zarządzający automatyką budynkową musi chodzić 24 godziny na dobę. Brak aktywnego systemu chłodzenia, oraz korzystanie z pamięci typu flash sprawia, że domownicy nie będą narażeni na nieprzyjemne dźwięki w zaciszu domowym. Komputer jest bardzo mały, co pozwala umieścić go w miejscu, którym jego obecność nie będzie się rzucać w oczy. Ważnym elementem jest także cena – *Raspberry Pi* potrafi być dziesięć a w wyjątkowych sytuacjach nawet sto razy tańszym rozwiązaniem, niż gotowe serwery zaproponowane przez firmy zajmujące się automatyką budynkową. Ważny jest także łatwy dostęp do *GPIO*, pozwalający podłączyć do systemu dowolne urządzenia.

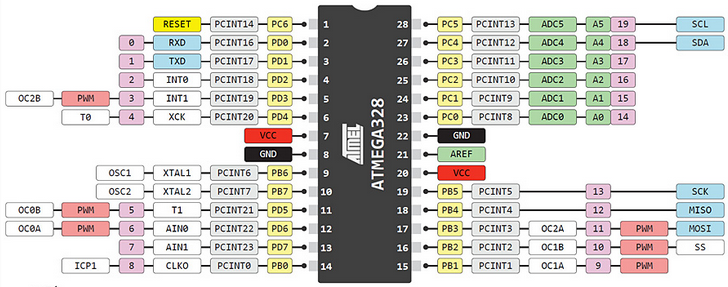
Raspberry Pi jako pełnoprawny komputer pozwala na uruchomienie pełnej dystrybucji Linuxa. Daje to niemal nieograniczone możliwości oprogramowania go w dowolnym języku.

Wokół fundacji, oraz samego urządzenia powstała olbrzymia społeczność, która jest w stanie doradzić i wspomóc każdą osobę, która chce sprawdzić swoich sił w poznawaniu systemu Linux, oraz programowania sprzętowego.

## ATmega 328

Każde urządzenie łączące się z serwerem nadrzędnym jest oparte na mikrokontrolerze firmy *Atmel* – *Atmega328*. Ośmiobitowe mikrokontrolery tej firmy, są bardzo popularne, niezwykle opłacalne cenowo i energooszczędne, dostępne są także w dużej różnorodności pod względem ilości pamięci, oraz portów. Wykorzystaliśmy model ze stosunkową dużą ilością pamięci wewnętrznej, oraz portów, by móc bez żadnych ograniczeń sprostać wszystkim wymaganiom postawionym przed projektem. Jednak w celu ograniczenia kosztów można by się zdecydować, na mikrokontroler z mniejszą ilością portów i pamięci.

Rys. 2 Atmega328

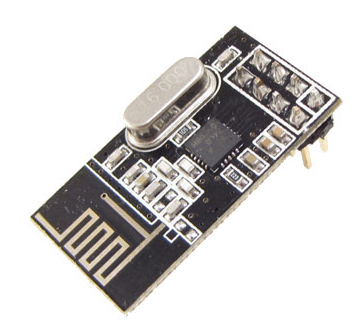


Mikrokontroler posiada wiele interfejsów za pomocą których można się z nim komunikować. Na potrzebu projektu niezwykle ważne były interfejsy *Serial Port*, oraz *SPI* (ang. *Serial Peripheral Interface*).

## nRF24l01+

Jednym z wymagań postawionych przed projektem było, że urządzenia muszą komunikować się z serwerem w sposób bezprzewodowy. Doskonałe urządzenie dla celów projektu zostały wytworzone przez firmę *Nordic Semiconductor*. Z modułem można komunikować się za pośrednictwem interfejsu *SPI*, więc jest możliwość wysterowania go z nimal dowolnego mikrokontrolera. Producent podkreśla jego wyjątkową energooszczędność i określa go, jako urządzenie *ULP* (ang. *Ultra Low Power*).

Wielką zaletą modułu *nRF24l01+* jest cena, można go dostać nawet poniżej 2$ za sztukę, co sprawia, że jest to rozwiązania bezkonkurencyjne, jeśli chodzi o cenę.

Moduł zdobył olbrzymią popularność wśród społeczności związanych z *Arduino* i *Raspberry Pi*, dzięki czemu powstało wiele bibliotek i gotowych rozwiązań kompatybilnych z tymi urządzeniami.

Rys. 3 nRF24l01+

# Środowisko programowe

Ten rozdział zostanie poświęcony systemu operacyjnemu, jaki obsługuje *Raspberry Pi*, a także środowisku programistycznemu, w jakim tworzony był system zarządzający automatyką budynkową. Zostaną także opisane technologie, z użyciem jakich system został zaprojektowany.

## Raspbian

Jest to darmowy system operacyjnym. Został on zoptymalizowany i dostosowany do Raspberry PI. Jest on oparty na systemie Debian. Został stworzony z potrzeby ograniczonych środków dostępnych na Raspberry Pi, oraz z wycofania wsparcia dla ARM11 przez systemy Debian i Ubuntu. Posiadały one także oprogramowanie niekompatybilne i niedostosowane do jego architektury. Jako, że celem Raspberry Pi i towarzyszącej mu fundacji jest propagowanie nauki, system ten posiada gotowe środowiska, które można wykorzystać do nauki programowania. Domyślnie można znaleźć tam wszelkie narzędzia, by zacząć programować w języku Python, czy Mathematica. Aby móc uruchamiać programu napisane w języku C#, trzeba doinstalować parę elementów, jednak wszystkie znajdują się w repozytorium systemu Raspbian.

## Visual Studio

Jest to środowisko programistyczne utworzone przez firmę Microsoft. Pierwsza wersja wyszła w roku 1995. W momencie pisania pracy ostatnią stabilną wersją jest *Visual Studio 2013* i w tym właśnie środowisku powstawało oprogramowanie mające za zadanie sterowanie inteligentnym budynkiem. Oprogramowanie dostępne jest w wielu rodzajach licencji, tak by wedle potrzeb, każdy programista mógł wybrać środowisku dostosowane do swoich potrzeb.

Visual Studio służy przede wszystkim do programowania w językach należących do platformy .*NET*, są to między innymi *Visual* *Basic*, *Visual* C++, F#, *Iron* *Python* i C#. Właśnie ten ostatni został wybrany, jako podstawa głównego systemu.

Środowisko programistyczne firmy Microsoft dostarcza wiele gotowych rozwiązań i funkcji. Do Visual Studio można instalować dodatki, znacznie rozszerzające jego możliwości. Przykładem takiego dodatku jest *ReSharper* firmy *JetBrains*, który między innymi wprowadza dodatkowe możliwości przeszukiwania i refaktoryzacji kodu, dodaje także wiele skrótów klawiszowych i wprowadza tworzenie szablonów, za pomocą, których możemy szybko utworzyć fragment kodu.

Środowisko daje olbrzymie możliwości nie tylko w zakresie programowania, ale także zarządzania bazami danych. Z poziomu IDE (ang. *Integrated Development Environment*) można połączyć się z bazą danych i edytować zarówno ją jak i zapisane w niej wartości. Można to robić z poziomu wygodnych edytorów, oraz kodu SQL (ang. *Structured Query Language*).

Visual Studio wspomaga programistę za pomocą niezwykle przydatnego systemu zwanego IntelliSense, który potrafi w bardzo trafny sposób podpowiadać programiście. Służy jednocześnie za swego rodzaju dokumentację, gdyż każda funkcja, którą nam podpowiada posiada krótki opis działania.

Dodatkowo daje dostęp do wielu narzędzi służących do poprawiania błędów i testowania kodu, budowania diagramów UML, oraz zarządzania projektem.

## Wykorzystane technologie

### Platforma .NET

Daje niezwykłe możliwości, podobne do platformy *Java*. Kod napisany i skompilowany w środowisku *Windows* na architekturze x86 można przenieść i uruchomić w środowisku Linux na architekturze *ARM*, za pomocą *mono*. Pozwala to na wygodę programowania i testowania kodu w środowisku *Visual* *Studio* na systemie *Windows*, a następnie przeniesienie go na inną platformę. Na platformę .*Net* składa się wiele języków. Jednym z nich jest C# i to właśnie za jego pomocą powstawał system zarządzającym inteligentnym domem. Język ten jest wzorowany na języku *Java* i na ten moment jest głównym językiem platformy .*NET*. Na platformę .*NET* składa się wiele bibliotek i szkieletów aplikacji. Największe to *Windows* *Presentation* *Foundation*, służący do tworzenia aplikacji okienkowych dla systemu operacyjnego *Windows*, *Windows* *Communication* *Foundation* dający dostęp do tworzenia serwisów obsługiwanych przez system *Windows*, oraz *ASP*.*NET* *MVC,* będący szkieletem dla aplikacji typu klient-serwer. Biblioteki zawarte w platformie .*Net* dają wiele gotowych funkcji i rozwiązań, takich jak parsowanie dokumentów w formacie *JSON* (ang. *JavaScript Object Notation)* i *XML (*ang. *Extensible Markup Language)*, komunikację z bazami danych na wiele sposobów, operacje matematyczne, obsługę plików itp.

### Mono

Jest to implementacja platformy .*NET* na systemy *Linux* i *Unix*. *Microsoft* oficjalnie wspiera tylko system *Windows*, dlatego by uruchomić oprogramowanie napisane w .*NET* należy użyć *mono*. Jest ono rozwijane przez firmę *Xamarin*, która dostarcza przede wszystkim narzędzia do tworzenia aplikacji na platformę *Android* i *iOS* za pomocą języka C# i platformy *mono*. Pierwsze wydanie nastąpiło w roku 2004. W tym momencie *mono* jest dojrzałym projektem, używanym także do rozwiązań komercyjnych. W ostatnim czasie rozwój mono znaczący przyśpieszył z powodu współpracy firmy *Xamarin* z korporacją *Microsoft*. Ogłosiła ona także, że najnowsza wersja platformy .*NET* będzie całkowicie otwarta i kompatybilna z mono. W skład *mono* wchodzi między innymi kompilator języka C# oraz środowisko *Common Language Runtime,* czyli maszyna wirtualną platformy .*NET*. *Mono* posiada zbiór własnych narzędzie, takich jak swoje własne środowisko programistyczne *XamarinStudio*, czy serwer *XSP* hostujący aplikacje *ASP*.*NET*.

### AngularJS

Jest to szkielet aplikacji, oparty na języku *JavaScript,* rozwijany i wspierany przez Google. Został stworzony z myślą o tworzeniu aplikacji typu *Single Page Application* wykorzystując wzorzec *Model-View-Controller* [4]*.* Jest to wyjątkowy rodzaj aplikacji internetowej, która w przeciwieństwie do tradycyjnego rozwiązania nie wymaga serwera do generowania widoku dla klienta. Dzięki temu możemy usunąć najbardziej irytujący składnik tego rodzaju aplikacji, czyli przeładowywanie stron internetowych. Biblioteka posiada specyficzne dla siebie znaczniki i atrybuty w pliku HTML (ang. *HyperText Markup Language*), dlatego przed uruchomieniem wczytuje dany plik HTML i wypełnia instrukcje zawarte we wcześniej wspomnianych elementach pliku generując odpowiednie dane wyjściowe, a także tworzy niezbędne elementy w modelu *JavaScript*.

*AngularJS* rozszerza możliwości języka *HTML* pozwalając na automatyczną synchronizację pomiędzy widokiem, a modelem. Pozwala to na uniknięcie jawnej manipulacji na DOM (ang. *Domain Object Model*)i rozgraniczenie logiki biznesowej od warstwy prezentacji. Stara się także wymusić na programiście dobre praktyki, takie jak *Inversion-Of-Control* poprzez wstrzykiwanie zależności, oraz dzielenie aplikacji na warstwy.

### Typescript

Język stworzony i rozwijany przez *Microsoft*, jako alternatywa dla języka *JavaScript* [9]. Kompiluje się on do *JavaScript*, przez co zachowuje kompatybilność wsteczną i nie musi być zaimplementowany przez przeglądarki. Kod napisany w *JavaScript* jest także pełnoprawnym kodem języka *TypeScript*, co daje możliwość wykorzystania olbrzymiej bazy już dostępnych bibliotek. Jest językiem w pełni obiektowym, wprowadza dziedziczenie, interfejsy oraz tworzenie klas w sposób znajomy z języków typu *C#,* czy *Java*. Pozwala także na wprowadzenie do kody statycznego typowania, co pozwala kompilatorowi na sprawdzania błędów. Wprowadza także duże wsparcie w postaci *IntelliSense*, jeżeli korzystamy z *Visual Studio*.

*TypyScript* wprowadza także pliki nagłówkowe, co pozwala na korzystanie z gotowych bilbiotek napisanych w języku *JavaScript*, tak, jakby były one tworzone w *TypeScript*. Pozwala to na wprowadzenie statycznego typowania i systemu podpowiedzi. *TypeScript* pozwala także na tworzenie powiązań pomiędzy plikami, co sprawia, że kompilator wie, gdzie szukać konkretnych bibliotek.

### MySql

Jest to relacyjna baza danych rozwijana przez firmę *Oracle*. Była pisana z myślą o szybkości, a nie funkcjonalności, przez co zdobyła duże grono przeciwników. Jest kompatybilna ze środowiskiem *Linux*, oraz posiada zbiór oficjalnych dodatków pozwalających jej na działania z *Visual Studio* *2013*, oraz kompatybilność z *Entity Framework 6*.

# C:\Users\Rafal\Downloads\project(1).pngImplementacja

Rys. 4 Architektura projektu

Cały projekt składa się z trzech elementów:

* Serwer nadrzędny
* Serwer multimedialny
* Urządzenia wyposażone w moduły *nrf24l01*+

Urządzenia wyposażone w moduły *nrf24l01*+ dzielą się na trzy kategorie: sensory, przekaźniki i bramki.

Sensory to urządzenia typu czujnik temperatury, ściemniacz światła, przełącznik itp. Są to wszystkie fizyczne urządzenia, którymi chcemy sterować, lub od których chcemy pobierać dane.

Przekaźniki to urządzenia których zadaniem jest przesyłanie sygnału na dalsze odległości. Powtarzają one sygnał dla sensorów, które nie są w stanie komunikować się bezpośrednio z serwerem nadrzędnym.

Bramka jest modułem połączonym bezpośrednio z serwerem nadrzędnym za pomocą interfejsu *Serial Port.*

Serwer wymienia informacje bezpośrednio tylko z dwoma elementami składającymi się na projekt. Z serwerem multimedialnym za pomocą protokołu *WebSocket*, oraz właśnie z bramką*.* Bramka jest modułem, z którym porozumiewają się wszystkie inne urządzenia. Jest ona zbudowana z mikrokontrolera *Atmega328*, niezbędnych urządzeń peryferyjnych, oraz podłączonego do niej modułu *nRF24l01*+. Moduły radiowe porozumiewają się ze sobą za pomocą biblioteki *MySensors.* Jest to rozbudowany protokół przeznaczony dla modułów *nRF24l01*+, oraz platformy *Arduino*. Jeśli serwer przekaże odpowiednią wiadomość poprzez *Serial Port* do bramki, zostanie ona zanalizowana i wysłana do odpowiedniego urządzenia zdefiniowanego w wiadomości. Gdy do sieci zostanie podłączone nowe urządzenie, automatycznie poprosi ono o nadaniu mu adresu. Wiadomość z tą prośbą zostanie wysłana do bramki, która przekaże ją do serwera. Serwer zarezerwuje adres i wyśle go z powrotem do urządzenia, poprzez bramkę. Kiedy adres zostanie zarezerwowany, będzie go można wybrać z list, podczas procesu dodawania urządzeń na stronie internetowej.

Podlegającą pode mnie częścią projektu było stworzenie serwera nadrzędnego zarządzającego całym systemem inteligentnego budynku. System został zaprojektowany wedle wymagań, by dać możliwość użytkownikowi dużą swobodę podczas konfiguracji, jednocześnie zachowując prostotę.

System został podzielony na wiele warstw, co ułatwia jego utrzymywanie i testowanie. Warstwy, jakie zostały wydzielone to:

* Warstwa Aplikacji
* Warstwa Prezentacji
* Warstwa Komunikacji
* Warstwa Bazy Danych
* Warstwa Planowania Zadań

## Konfiguracja platormy RaspberryPi

Na komputerze został zainstalowany system *Raspbian.* Rozwiązania, na jakie zdecydowano się w projekcie wymagały zainstalowania odpowiednich paczek z oficjalnego repozytorium. Były to:

* *mono-complete -* wszystkie paczki niezbędne do uruchamiania aplikacji opartych na platformie .*NET*
* *mono-xsp4 – XSP* jest serwerem WWW, potrafiącym uruchamiać i dawać użytkownikowi dostęp do aplikacji opartych o platformę *ASP.NET*
* *mysql-server* – jest to serwer bazy danych *MySql*

Należało także zwolnić *Serial Port*, który domyślnie jest używany w *RaspberryPi* do logowania informacji. Można to było uczynić poprzez usunięcie wszystkich referencji do *Serial Port* w plikach */boot/cmdline.txt* i */etc/inittab***.**

## Warstwa prezentacji

Warstwa prezentacji została oparta na bibliotece ASP.NET MVC, wchodzącej w skład pakietu ASP.NET platformy .NET.

Jest to projekt startowy dla całej aplikacji, znajdują się w nim pliki konfiguracyjne, oraz są inicjowane wszystkie niezbędne wątki dla działania aplikacji.

public partial class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

System.Data.Entity.Database.SetInitializer(

new HomeCenterDbInitializer());

PostAuthenticateRequest += Application\_PostAuthenticateRequest;

CreateDatabaseIfNotExisting();

AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

AutoMapperConfig.RegisterMappings();

var container = Bootstrapper.Initialise();

container.Resolve<IWebSocket>().Initialise();

container.Resolve<IJobScheduler>().Initialise();

container.Resolve<ISerialPortConnection>().Initialise();

}

}

Metoda *Application*\_*Start* jest uruchamiana w momencie startu aplikacji. Ona jest odpowiedzialna za uruchomienie niezbędnych dla aplikacji elementów. Możemy dostrzec wywołanie metody *CreateDatabaseIfNotExisting*, odpowiedzialnej za stworzenie bazy danych i zainicjowanie jej początkowymi wartościami, w wypadku, gdy nie istnieje ona.

### AutoMapper

Biblioteka *AutoMapper* jest odpowiedzialna za automatyczne mapowanie obiektów. Jest to przydatne, gdy chcemy utworzyć obiekt typu *DTO* (ang. *Data* *Transfer* *Object*) z encji. Wystarczy raz skonfigurować odpowiednie mapowanie, by potem móc z niego dowolnie korzystać. Wszystkie mapowania zdefiniowane są w metodzie *AutoMapperConfig*.*RegisterMappings*.

Mapper.CreateMap<DeviceRelation, DeviceRelationDto>()

.ForMember(dest => dest.StateRelationDescription,

opt => opt.MapFrom(src => src.StateRelation

.GetDescription()));

Tak wygląda przykładowa konfiguracja mapowania z klasy DeviceRelation na DeviceRelationDto. Jeżeli właściwości posiadają taką samą nazwę, oraz typ, nie trzeba ich sprecyzować. Klasa DeviceRelationDto posiada pole StateRelationDescription, którego nie ma bezpośrednio w klasie DeviceRelation. W łatwy sposób możemy skonfigurować mapowanie, tak, by wartość była wyciągana z innego elementu.

var deviceRelationDto = Mapper.Map<DeviceRelationDto>(deviceRelation);

Same mapowanie obiektów jest także bardzo proste. Jak widać wystarczy podać typ, jaki będzie zwracany, oraz obiekt, z jakiego będziemy mapować. Resztę pracy wykona za nas AutoMapper.

### Unity

Kolejną ważną biblioteką jest *Unity*. Jest to kontener *DI* (Dependency Injection), odpowiedzialny za wstrzykiwanie zależności. Jest to jeden ze sposobów implementowania paradygmatu zwanego odwróceniem sterowania. „Odwrócenie sterowania przenosi drugorzędne odpowiedzialności z obiektu do innego obiektu, dedykowanego do tych zadań, przez co zapewnia się zachowanie zasady pojedynczej odpowiedzialności [4]. W kontekście zarządzania zależnościami obiekt nie powinien być odpowiedzialny za samodzielne tworzenie zależności. Powinien przekazać tę odpowiedzialność do innego autorytarnego mechanizmu, odwracając w ten sposób sterowanie.” Mówiąc prostymi słowami chcemy, by obiekty były tworzone nie samodzielnie przez klasę, a przez inny, wyspecjalizowany do tego element, który potem przekaże utworzoną przez siebie instancję. Może to być na przykład wzorzec projektowy fabryka, albo kontener DI.

W przypadku tego projektu, zdecydowano się przenieść odpowiedzialność za tworzenie obiektów na bibliotekę Unity. Gdy tworzymy instancję klasy za jej pomocą, Unity automatycznie stara się rozwiązać wszystkie zależności, i przekazać niezbędne obiekty do konstruktora. Aby kontener mógł rozwiązać konkretne zależności, trzeba je najpierw skonfigurować. Polega to na przypisaniu odpowiedniej klasy do interfejsu.

container.RegisterType<IDeviceService, DeviceService>();

Inicjalizujemy obiekt typu *UnityContainer*, a następnie wykorzystujemy go do rejestracji zależności. W przykładowym kodzie pokazano tylko jedną zależność, jednak w całym projekcie zdefiniowano ich aż dwadzieścia osiem. Następnie obiekt można stworzyć za pomocą kontenera, korzystając z metody *Resolve*. Przykłady widać w metodzie *Application\_Start*. Biblioteka rozwiązuje automatycznie zależności w kontrolerach, więc nie trzeba ich inicjować jawnie, wystarczy podać w konstruktorach, jakiego typu obiektów oczekujemy.

public class AccountController : Controller

{

private readonly IAuthenticationService \_authenticationService;

private readonly IDomainLoggerService \_domainLoggerService;

public AccountController(IAuthenticationService authenticationService,

IDomainLoggerService domainLoggerService)

{

\_authenticationService = authenticationService;

\_domainLoggerService = domainLoggerService;

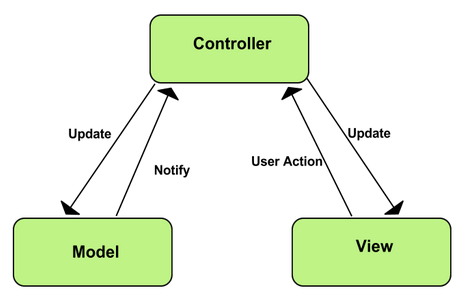
}

}

Powyższy kod przedstawia tylko fragment kodu kontrolera AccountController. Jako, że jest od odpowiedzialny między innymi za tworzenie użytkowników, musi mieć dostęp do serwisu znajdującego się w warstwie aplikacji, który to umożliwia. Odpowiedzialność za tworzenie kontrolera zostało przeniesione na bibliotekę Unity, więc to ona go inicjuje, jednocześnie rozwiązując zależności i przekazując do konstruktora niezbędne instancje. Oczywiście one też często posiadają inne zależności, które także muszą zostać rozstrzygnięte.

### Architektura MVC

Framework ASP.NET MVC korzysta z wzorca architektonicznego *Model-View-Controller*, wymuszając na programiście rozgraniczenie tych elementów [6]. Tworzenie osobnych warstw, które są odpowiedzialne za konkretne zadnia, jest dobrym podejściem w programowaniu. Nowoczesne biblioteki często wymuszają na programiście dobre praktyki, ułatwiając tworzenie i utrzymywanie kodu.

Graf przedstawia trzy powiązane ze sobą elementy. W przypadku tego projektu istnieją także inne warstwy aplikacji, jednak dla uproszczenia nie będą one tu przedstawione.

Rys. 5 Wzorzec MVC

### Kontroler

Kontroler jest elementem odpowiedzialnym za generowanie widoku, oraz obsługę akcji. Ma dostęp do warstwy aplikacji za pomocą udostępnionych przez nią serwisów. Jest kimś w rodzaju koordynatora, który ma dostęp do wielu warstw aplikacji. Otrzymuje dane wejściowe od użytkownika i za ich pomocą zleca wykonywanie odpowiednich operacji warstwom odpowiedzialnym za logikę biznesową, czy dostęp do bazy danych. Następnie zwraca odpowiednie informacje do widoku.

W projekcie wyróżniono siedem kontrolerów i każdy z nich jest odpowiedzialny za inną dziedzinę:

* AccountController
* AddDeviceController
* CreateBuildingController
* DeviceControlController
* DeviceRelationsController
* HomeController
* MediaController

Warto zwrócić uwagę na to, że każda nazwa kontrolera kończy się na Controller. Jest to konwencja przyjęta przez framework i jest wykorzystywana na przykład do tworzenia adresów do konkretnych akcji. Oprócz charakterystycznej nazwy, każdy kontroler musi dziedziczyć po abstrakcyjnej klasie Controller.

public class MediaController : Controller

{

public ActionResult Index()

{

return View();

}

}

Jako przykład, możemy użyć klasę *MediaController*. Posiada ona tylko jedną akcję o nazwie Index. Adres to tej akcji wygląda następująco: */Media/Index*. Jest to domyślny sposób adresowania przyjęty w *ASP*.*NET* *MVC*. Akcja w tym przypadku zwraca widok, jeśli nie zostaną podane żadne parametry, domyślnym widokiem dla akcji jest plik o nazwie *Index* znajdujący się w folderze */Views/Media*.

### Model

Model w mniej rozbudowanych aplikacjach często zawiera dane, oraz logikę biznesową. Jako, że w przypadku tego projektu te elementy zostały odseparowane w warstwie aplikacji, pojawia się pojęcie modelu prezentacji. Dane znajdujące się bazie danych nie zawsze są w formacie nadającym się do zaprezentowania użytkownikowi, mogą na przykład posiadać nadmiar informacji, które chcemy przed użytkownikiem ukryć. Dlatego też, wykorzystując dane pochodzące z warstwy aplikacji, tworzymy wyspecjalizowane modele prezentacji, których zadaniem jest przedstawienie, oraz odpowiednie sformatowanie danych. Do tworzenia obiektów tego typu można wykorzystać bibliotekę *AutoMapper*.

### Widok

Widok jest warstwą, która jest wyświetlana i obsługiwana po stronie klienta. Standardowo są to pliki *HTML* z zawartością strony, *CSS* (ang. *Cascade Style Sheets*) definiującymi wygląd strony i skryptami *JavaScript*, pozwalającymi na manipulację *DOM*, oraz łączenie się z serwerem w celu wysłania, bądź odebrania informacji. Jest to sposób komunikacji użytkownika z serwerem. To właśnie poprzez widok serwer może zaprezentować dane dla użytkownika, a użytkownik, na przykład wypełniając odpowiednie formularze i wysyłając je, komunikuje się z serwerem.

#### Bootstrap

W warstwie prezentacji zastosowano *front-end framework* zwany *Bootstrap*. Wprowadza on tak zwany *responsive design*, czyli płynne dostosowanie się układu strony, w zależności od rozdzielczości urządzenia, na jakim działa. Daje nam możliwość korzystania z gotowych klas *CSS*, które można przypisywać do elementów. Całość przypomina układanie klocków. Strona jest podzielona na dwanaście kolumn i szerokość elementów ustalamy w kolumnach, a nie jak przyjęło się w tradycyjnych rozwiązaniach, w pikselach. Sprawia to, że niezależnie od rozdzielczości element zajmuję taką samą część dostępnej przestrzeni. Bardzo łatwo zmieniać wygląd strony, gdyż wystarczy podmienić tylko plik *CSS* z innym schematem typu *Bootstrap*.

#### AngularJS i Typescript

Struktura widoku w warstwie prezentacji tej aplikacji jest specyficzna. Został zastosowany tutaj *AngularJS*, jednak nie jako framework *SPA* (ang. *Single-Page-Aplication*). Dynamicznie generowane są tylko fragmenty, a nie całe strony. *AngularJS* został zaimplementowany w języku *TypeScript*, co dało wiele zalet wynikających ze struktury tego języka*.*

Szkielet aplikacji *AngularJS* wymusza wzorzec architektoniczny *MVC.* W przypadku tego projektu dostęp do modelu jest możliwy za pomocą odpowiednich serwisów, w których są zaimplementowane zapytania *AJAX* (ang. *Asynchronous JavaScript and XML).*

Kontroler jest zwykłą klasą, jednak niezwykle ważnym jej elementem jest pole *scope.* Wszystkie obiekty znajdujące się w tym polu są bezpośrednio dostępne w widoku. Każdy widok ma zdefiniowany kontroler, który go obsługuje i z którego danych korzysta.

Elementy widoki i kontrolera są ze sobą połączone. Dzięki temu nie jest potrzebne jawne modyfikowanie *DOM*. Element wyświetlający tekst może być połączony ze zmienną typu *string,* znajdującą się w kontrolerze. Jeśli zmienna zostanie zmodyfikowana, widok się uaktualni. Działa to także w drugą stronę. Na przykład zdarzenie odpowiadające za wciśnięcie przycisku jest połączone z metodą, która wyświetla adekwatne powiadomienie, po wykonaniu tej czynności.

Kontroler wykonuje wiele innych działań, wykonuje niezbędne walidacje danych po stronie klienta, a także komunikuje się z serwerem za pośrednictwem zapytań *AJAX.* Jest totechnologia umożliwiająca asynchroniczne komunikowanie się klienta z serwerem. Dzięki temu wymiana informacji może odbyć się w tle, bez przeładowania strony internetowej. Klient wysyła zapytania typu *POST*, gdy chce przesłać informacje na serwer, oraz typu *GET,* gdy chce jakieś otrzymać. Podstawowym formatem, jaki wysyłają pomiędzy sobą klient i serwer jest *JSON*. Jest to bardzo lekki format danych, jeden z konkurentów *XML*. Pozwala na przedstawienie obiektu w formie łańcucha znaków.

{"Name":"Second Floor",

"Id":2,

"Rooms":[{"Name":"Bathroom","Id":3},

{"Name":"Bedroom","Id":4}]}

Jest to naturalny format dla języka *JavaScript* i radzi on sobie z nim bardzo dobrze. Biblioteki umożliwiające tworzenie obiektów z formatu *JSON* istnieją dla niemal każdego języka.

*AngularJS* umożliwia definiowanie serwisów. Są to zazwyczaj zbiory metod, których zadaniem jest komunikacja z serwerem. Przykładem takiego serwisu jest *HistorySevice*, którego zadaniem jest pobieranie logów z serwera.

Jest to przykład kodu w języku *TypeScript*. Jak można zobaczyć struktura kody przypomina C#. Aby stworzyć klasę, używamy słowa kluczoweg *class.* Za pomocą słowa *extends* dziedziczymy po klasie *Common.Service.* W klasie umieszczamy metody i pola, tak, jak w wielu znajomych językach obiektowych. Klasa posiada także konstruktor, który definiujemy za pomocą słowa *constructor.* Możemy określić także typy zmiennych, oraz typy zwracane przez funkcję. Dzięki językowi *TypeScript* możemy korzystać z dobrodziejstwa języka obiektowego po stronie klienta, a cały kod jest znacznie bardziej czytelny, niż gdybyśmy korzystali z języka *JavaScript.*

export class HistoryService extends Common.Service

{

static $inject = ['$http'];

constructor(httpRequest: ng.IHttpService)

{

super(httpRequest);

}

getSpecificLogs(page: number,

count: number,

username: string,

fromDateTime: number,

toDateTime: number) : ng.IHttpPromise<Array<IHistoryLog>> {

return this.HttpRequest.post("/Home/GetLogsForPage",

{

page: page,

count: count,

username: username,

fromDateTime: fromDateTime,

toDateTime: toDateTime

});

}

getLogsLength(username: string,

fromDateTime: number,

toDateTime: number): ng.IHttpPromise<number> {

return this.HttpRequest.post("/Home/GetLogsLength",

{

username: username,

fromDateTime: fromDateTime,

toDateTime: toDateTime

});

}

}

Warto zauważyć zmienną statyczną *$inject,* jest ona odpowiedzialna za wstrzykiwanie zależności do klas, w tym przypadku przekazywany jest serwis pozwalający na tworzenie zapytań *HTTP* (ang. *Hypertext Trasfer Protocol*)*.* Jest to element biblioteki *AngularJS.* Serwis *HistoryService* posiada dwie metody. Obie wysyłają zapytanie typu *POST* na serwer, przekazując w parametrach informacje na temat sposobu filtrowania. Metoda *getSpecificLogs* dodatkowo określa parametry stronicowania. Filtry są wyświetlane dla użytkownika w stronach o długości od 25 do 100 elementów, w zależności od preferencji. Serwer przesyła klientowi tylko niezbędne logi do wyświetlenia na stronie, czyli tylko konkretną stronę. W metodach jest także określony typ, jakie metody zwracają. W przypadku pierwszej metody jest to *ng.IHttpPromise<Array<IHistoryLog>>,* początkowo może to wyglądać skomplikowanie, jednak *ng.IHttpPromise* jest niczym innym jak interfejsem określającym zapytanie *HTTP,* do którego można przypisać zdarzenia *success* i *error,* wywołujące się w zależności od powodzenia zapytania.  *Array<IHistoryLog>* oznacza, że w razie sukcesu zapytanie zwraca tablice obiektów implementujących interfejs *IHistoryLog*.

Oprócz *HistoryService,* zostały utworzone także serwisy takie jak *AccountService,* odpowiedzialny za tworzenie i modyfikowanie kont, *BuildingService*, pozwalający na tworzenie, usuwanie i zwracanie budynków, oraz *DeviceService*, którego rolą jest dodawanie, usuwanie i kontrolowanie wszystkich elementów związanych z urządzeniami.

## Warstwa aplikacji

Warstwa aplikacji została zaprojektowana za pomocą podejścia zwanego DDD (ang. *Domain-Driven Design*)*.* Taki sposób projektowania jest bardzo powszechny w firmach, gdzie kładzie się duży nacisk na zmniejszania kosztów utrzymywania aplikacji. Twórcą tego terminu jest Eric Evans, który po raz pierwszy użył go w swojej książce *Domain-Driven Design: Tackling Complexity in the Heart of Software* [5]*.* Tytuł tej książki dobrze określa cel stworzenia tej metodyki – okiełznanie złożoności projektu. Podejście to wymusza podzielenie aplikacji na warstwy, domeny, które są od siebie odseparowane i zajmują pewną sferę tematyczną i są odpowiedzialne za określone czynności. Na przykładzie mojej aplikacji można wyróżnić *DeviceDomain*, zajmującą się między innymi dodawaniem, usuwaniem i zmianą stanu urządzeń, oraz *HistoryDomain,* której celem jest logowanie zdarzeń do bazy danych. Takie rozwarstwianie aplikacji ma wiele zalet. Dużo rzeczy jest od siebie niezależnych, przez co możemy testować je osobno i łatwiej znajdywać, oraz naprawiać błędy. Gdy aplikacja jest duża, różne osoby, albo nawet zespoły mogą zajmować się różnymi częściami aplikacji niezależnie od siebie. Dla osoby zaznajomionej z tym podejściem, zrozumienie nowej aplikacji będzie łatwiejsze, ponieważ sama struktura kodu będzie znajoma. Znacznie ułatwia to także przyszłe rozbudowywanie aplikacji, ponieważ można dodawać kolejne elementy obok już istniejących. Dlatego czasem warto tworzyć nawet mniej rozbudowane domeny, jeśli jest możliwość, że w przyszłości dojdą do nich nowe funkcje. Unikniemy dzięki temu przepisywania i rozdzielania jakiejś domeny na dwie inny, by zachować porządek.

Domeny mogą się ze sobą komunikować za pomocą *DomainEvents.* Gdy domena wykona jakieś zadanie, może opublikować zdarzenie na przykład ogłaszające koniec wykonania jakiejś czynności. Zdarzenie zostanie obsłużone poprzez specjalną klasę, która wywoła odpowiednie akcje w innej domenie.

Każda domena dzieli się na trzy warstwy:

* Infrastruktury
* Serwisu
* Modelu Biznesowego

Najważniejsza jest warstwa modelu biznesowego. Zawiera ona w sobie przede wszystkim encje z logiką biznesową, oraz definiuje interfejsy dla warstwy infrastruktury. Jest to spowodowane tym, że dostęp do warstwy persystencji był podyktowany wymaganiami biznesowymi. Encje tworzą agregaty. Jest to swego rodzaju graf złożony z obiektów, który ma wspólny *Aggregate Root,* czyli encje główną. Jest jednostka, którą pobieramy z bazy danych poprzez warstwę infrastruktury, wykonujemy zawartą w niej logikę biznesowej, a następnie zapisujemy jej stan w bazie danych. Jedynie poprzez encje główną możemy dostać się do innych elementów danego grafu i wykonać na nich dane operacje. Model biznesowy jest to zespół reguł określonych przez zlecających aplikację. *DDD* zostało stworzone, by stworzyć wspólny język, platformę porozumienia pomiędzy programistami, a ludźmi biznesu. Dlatego model biznesowy jest nieświadomy warstwy bazy danych, oraz innych warstw aplikacji. Jego zadaniem jest odwzorowanie rzeczywistych działań biznesowych.

Warstwa infrastruktury zawiera w sobie repozytoria, będące swego rodzaju abstrakcją stworzoną wokół warstwy dostępu do bazy danych [8]. W niej zaimplementowane są metody zdefiniowane w warstwie modelu biznesowego, których daniem są operacje na bazie danych, na przykład zapisanie stanu danej encji. Repozytorium może posiadać tylko encja główna, inne elementy są edytowane poprzez nią i zawartą w niej logikę.

Warstwa serwisu ma wstrzyknięte do siebie repozytoria, przez co ma możliwość wyciągania encji z bazy danych, a następnie przeprowadzanie na nich określonych operacji, zdefiniowanych w warstwie modelu biznesowego. Serwisy w przeciwieństwie do elementów warstw infrastruktury i modelu są dostępne dla reszty aplikacji, na przykład warstwy prezentacji. Są czymś w rodzaju bramy dostępu do warstwy aplikacji dla innych warstw systemu. Jako, że dostęp do modelu biznesowego, albo repozytoriów jest ograniczony, a warstwy te działają na encjach, zostały stworzone obiekty DTO. Są to często encje, jednak bez żadnych metod biznesowych. Służą tylko do przenoszenia danych pomiędzy warstwami aplikacji. W idealnym wypadku DTO powinny zawierać tylko typy proste, albo, chociaż typy niecharakterystyczne dla jednego języka, ponieważ czasem musimy przenosić dane poza dany język. W przypadku tego projektu, chcemy wykorzystać dane z serwera, gdzie mamy język C#, po stronie klienta, gdzie jest *TypeScript.*

W warstwie aplikacji wyróżniono 3 główne domeny i jedną *CommonDomain,* zawierającą zestawy generycznych klas abstrakcyjnych i interfejsów dla pozostałych domen.

public abstract class GenericRepository<T>

: IGenericRepository<T> where T : class, IAggregateRoot

{

protected readonly IDbContext \_contextDb;

protected readonly IDbSet<T> \_set;

protected GenericRepository(IDbContext contextDb)

{

\_contextDb = contextDb;

\_set = \_contextDb.Set<T>();

\_contextDb.EnableLazyLoading();

}

public void Save()

{

\_contextDb.SaveChanges();

}

public void RollbackChanges()

{

\_contextDb.Rollback();

}

}

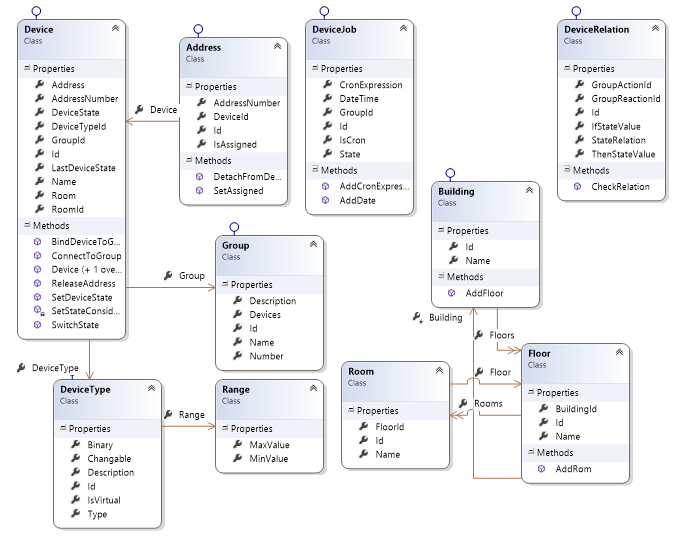
Przykładem takiej klasy jest *GenericRepository*, klasa abstrakcyjna, po której dziedziczy każde repozytorium w aplikacji. Zawiera ona elementy wspólne dla każdej klasy tego typu. Jak można zobaczyć na poniższym fragmencie kodu, klasa *GenericRepository* jest typem generycznym, przyjmującym, jako parametry tylko klasy implementujące interfejs *IAggregateRoot*. Jest to ważne, ponieważ jak wspomniano wyżej, tylko *AggregateRoot* może posiadać swoje repozytorium. Należy zwrócić uwagę, na fakt, że klasa nie implementuje często spotykanych metod typu *CRUD* (*Create Read Update Delete),* czyli podstawowych działań na bazie danych. Nie każde repozytorium powinno według mnie implementować te metody automatycznie, za każdym razem powinna to być świadoma decyzja programisty. Nie zawsze chcemy dać możliwość na przykład modyfikowania danej encji, lub jej usunięcia.

Główne domeny wyszczególnione w warstwie aplikacji to

* DeviceDomain
* AuthorizationDomain
* HistoryDomain

### DeviceDomain

Jest to najbardziej rozbudowana domena w warstwie aplikacji. Zajmuje się ona tworzeniem struktury budynku, oraz zarządzaniem urządzeniami. Za pomocą serwisów zawartych w tej domenie można także dodawać urządzenia, modyfikować ich grupy, oraz zmieniać ich stan. Na zdjęciu przedstawiono wszystkie encje modelu biznesowego, ich właściwości, oraz metody zajmujące się logiką biznesową.

Najważniejszą klasą tego modelu jest *Device*. Posiada ona wiele metod, które pozwalają na wysterowania urządzenia, wedle określonych reguł definiowanych przede wszystkim przez typ urządzenia. Oprócz grupy, niezwykle ważna jest dla urządzenia grupa, do której należy. W systemie urządzenia nie łączą się ze sobą bezpośrednio, a właśnie przez grupę, do której należą.

Rys. 6 Diagram zależności modelu biznesowego DeviceDomain

Urządzenia w jednej grupie synchronizują ze sobą swój stan, przez co oddziałują ze sobą. Działa to w sposób podobny do systemu KNX, gdzie urządzenia posiadają adres grupowy i urządzenia w obrębie jednej grupy mogą ze sobą współpracować. Jeśli fizyczny przycisk i aktor sterujący oświetleniem znajdują się w jednej grupie, zmiana stanu wywołana kliknięciem w przycisk sprawi, że nastąpi także zmiana stanu aktora. Tworzenie powiązań w ten sposób sprawia, że możemy wiązać ze sobą dowolną ilość urządzeń. Urządzenia można przemieszczać pomiędzy grupami, jednak w danej chwili urządzenie może należeć tylko do jednej grupy.

Każde urządzenie ma swój typ, który jest określony przez szereg wartości. Oprócz nazwy i opisu, można znaleźć tam inne własności. Takie jak, czy urządzenie jest binarne, czyli czy może przyjmować tylko dwa stany. *Changeable* określa, czy możemy mieć wpływ na stan urządzenia. Chcemy mieć możliwość zmiany jasności oświetlenia, ale na przykład nie chcemy wpływać na termometr, gdyż jego stan nie zależy od nas, a od temperatury otoczenia i chcemy jedynie odczytywać tą wartość. Ważnym polem jest *IsVirtual*. Urządzenia są podzielone na dwa zasadnicze typy, urządzenia wirtualne i fizyczne. Urządzeniami wirtualnymi są wszelkiego rodzaju przyciski, suwaki, oraz monitory wyświetlane na stronie internetowej, a urządzeniami fizycznymi są urządzenia umieszczone w naszym domu, takie jak przełączniki na ścianie, albo aktory.

Podczas dodawania urządzenia do systemu, Administrator, albo osoba z odpowiednimi uprawnieniami musi zdefiniować, adres, oraz do jakiego pokoju i grupy należy dodać urządzenie.

W domenie można dostrzec *aggregate,* zawierający w sobie klasy *Room, Floor,* oraz *Building*. Ostatnia z wymienionych klas jest elementem typu *aggregate root* i to właśnie poprzez nią możemy mamy dostęp do pięter i pokoi.

Celem istnienia hierarchii budynku jest łatwiejsze sterowanie urządzeniami. Dzięki temu możemy stworzyć swojego rodzaju prostą wizualizację, gdzie łatwo możemy wybrać część budynku, którą chcemy sterować. Wiedząc, gdzie znajduje się urządzenie, znacznie łatwiej nam je zlokalizować i sprawdzić, lub ustawić jego stan.

Nie jest to jedyny sposób układania zależności pomiędzy urządzeniami. Kolejnym sposobem jest tworzenie relacji pomiędzy grupami. Jest odpowiedzialna za to klasa *DeviceRelation.* Pozwala nam to na utworzenie zależności, tak, że gdy w jednej grupie stan zostanie zmieniony na określony, to wtedy w innej grupie stan zmieni się na taki, jaki zdefiniowaliśmy przy tworzeniu relacji. Można wybrać parę rodzajów zależności, takich jak większy, mniejszy, równy stan grupy. Całość działa według popularnego schematu *if this then this,* którego idea jest bardzo prosta. Jeśli zostanie spełniony jakiś warunek, wtedy wykonaj określoną akcję. Jednak łącząc kilka takich zależności ze sobą, można tworzyć mocno zaawansowane sceny. Przykładem może być połączenie grupy, odpowiedzialnej z wysuwaniem ekranu dla projektora, z grupą odpowiedzialną za ściemnianie światła. Efektem może być zmniejszenie jasności światła na 20 procent w momencie, gdy wciśnięty zostanie przycisk odpowiedzialny za wysuwanie ekranu.

Zmiana stanów urządzeń może zostać wywoła w jeszcze jeden sposób. Oprócz tworzenia relacji pomiędzy urządzeniami, może stworzyć dla urządzenia zadanie. Zadanie zostanie wykonane, w zdefiniowanym przez nas czasie. Może to być jednorazowe zdarzenie wykonane w danym czasie, albo zadanie wykonywujące się cyklicznie. Daje to duże możliwości, jeśli np. chcemy symulować naszą obecność w domu, gdy pracujemy i tak można kazać, by światła w domu zapalały się każdego dnia od poniedziałku do piątku o godzinie 13.

Zadania *DeviceJob* są obsługiwane za pomocą innej warstwy, o której można przeczytać w dalszej części pracy.

### AuthenticationDomain

Aplikacja posiada system zarządzania użytkownikami. Każdy użytkownik może posiadać dowolną ilość ról. Role określają, do jakiej części strony użytkownik ma prawo dostępu. Role zaimplementowane w aplikacji:

* Administrator
* User
* Builder
* Scheduler
* AccountManager

*Administrator* ma dostęp do każdej części systemu. Każde działanie system, takie jak zmiana stanu urządzenia, utworzenie urządzenia, dodanie użytkownika itd. tworzy w bazie danych logi, które tylko użytkownik z rolą administratora może podejrzeć.

*AccountManager* ma dostęp tylko do części pozwalającej na zarządzania kontami użytkowników. Może je tworzyć, oraz zmieniać im prawa. Nie posiada dostępu do logów.

*Builder* może dodawać urządzenia, oraz tworzyć strukturę budynku. Nie ma dostępu do pozostałych funkcji.

*Scheduler* ma prawo do ustawiania zależności pomiędzy grupami, oraz specjalnych wyzwalaczy, które uruchomią urządzenia po spełnieniu określonych warunków.

*User* ma prawo do kontrolowania urządzeń z poziomu *UI* (ang. *User Interface*). Z poziomu panelu zarządzania kontami, do którego ma dostęp *AccountManager* i *Administrator*, można ograniczyć dostęp użytkownika z rolą *User* tylko do określonych pokoi. Daje to możliwość nadania dostępu do systemu gościom i dania im możliwość sterowania urządzeniami w ich pokoju z poziomu aplikacji.

Dzięki robudowanemu systemowi ról, z dobrodziejstw systemu inteligentnego domu mogą korzystać nie tylko właściciele domu, ale także ich przyjaciele, dzieci, czy krewni.

Dane o użytkownikach są przechowywane w bazie danych, jednak nie zawierają one hasła, tak by w razie włamania do bazy danych, hasła użytkowników systemu nie zostały poznane.

*Platforma .NET* udostępnia przestrzeń nazw *System.Security,* dzięki której mamy dostęp do wielu certyfikowanych narzędzi do kryptografii. Przy tworzenia użytkownika dla każdego hasła generowana jest sól. Jest ciąg losowy składający się 24 znaków. Do wylosowania ciągu wykorzystywana jest klasa *RNGCryptoServiceProvider.* W przeciwieństwie do *Random,* może pochwalić się znacznie większą losowością, ciąg losowy jest wyliczany nie tylko na bazie jednej zmiennej losowej, jak w *Random*, w którym zwykle jest to czas, ale wielu innych, takich jak identyfikatory procesu i wątku, cyklu zegara procesora, czasu systemowego, wolnej przestrzeni na dysku itp. Wszystko obliczane za pomocą specjalnych algorytmów, przez co zbliżamy się do „prawdziwej” losowości.

Wylosowany ciąg znaków dodajemy do hasła a następnie hashujemy za pomocą klasy *SHA256CryptoServiceProvider* algorytmem *SHA-256*, używanym powszechnie przy ochronie haseł, także przez rząd Stanów Zjednoczonych*.* Jest powszechnie uważany za bezpieczny algorytm.Idea przechowywania haseł w ten sposób polega na tym, że nie ma możliwości, albo jest to niezwykle trudne, by odzyskać hasło, posiadając tylko hash i sól.

Kiedy użytkownik loguje się do systemu z bazy danych dla danej nawy użytkownika wyciągany jest hash i sól. Do hasła podanego podczas logowania dodawana jest pobrana z bazy sól, a następnie jest wyliczany hash, który jest przyrównywany do tego, który jest przechowany w bazie.

### HistoryDomain

*HistoryDomain* jest odpowiedzialna za tworzenie i przechowywanie logów aplikacji. Kiedy serwer wykona jakąś czynność, na przykład zmieni stan jakiegoś urządzenia, zaloguje to w bazie danych, na przykład, aby administrator mógł sprawdzić aktywność urządzeń, podczas swojej nieobecności. Zdarzenia, jakie są logowane to między innymi:

* Utworzenie urządzenia
* Stworzenie budynku, lub jego modyfikacja
* Dodanie nowego użytkownika
* Zalogowanie, oraz wylogowanie użytkownika

Log przechowuje podstawowe informacje o wydarzeniu. Użytkownika, który jest zalogowany do danej sesji, czas wydarzenia, oraz krótki opis.

public List<HistoryLog> GetPagedLogsFor(int i,

int count,

string username,

DateTime fromDate,

DateTime toDate)

{

return \_set.Where(l => l.Username == username

&& l.DateTime >= fromDate

&& l.DateTime <= toDate )

.OrderByDescending(l => l.DateTime)

.Skip(i \* count)

.Take(count).ToList();

}

Logi potrafią zajmować bardzo dużo miejsca i zawierać setki tysięcy, a nawet miliony wpisów. Dlatego ważne jest, by sortowanie i stronicowanie danych działo się po stronie bazy danych, a zwracana przez repozytorium była tylko niezbędna część wszystkich logów. Na fragmencie kodu widać zapytanie *LINQ (*ang. *Language INtegrated Query*) znajdujące się w repozytorium. Jest one tłumaczone na zapytanie *SQL* przez *Entity Framework* i wysyłane do bazy danych, która zwraca nam odpowiednie dane.

### DomainEvents

Jest to sposób komunikacji pomiędzy warstwami aplikacji, albo różnymi domenami. Domeny nie powinny komunikować się ze sobą w sposób bezpośredni.

Stan urządzeń jest synchronizowany za pomocą *DomainEvents*. Gdy użytkownik na przykład przesunie suwak wysterywujący śniemniacz, wtedy zostanie uruchomione zdarzenie *DeviceStateHasChangedFromUi.* Inna możliwość, to, gdy zostanie zmieniony stan urządzenia fizycznego, na przykład z powodu zmiany temperatury, albo przyciśnięcia fizycznego przycisku. Wtedy uruchomi się zdarzenie *DeviceStateHasChangedFromSerialPort.* Zdarzenia mają za zadania zalogować zmianę stanu, jej powód, oraz ewentualnego użytkownika powodującego zmianę, a także uruchomić funkcję w *ChangeDeviceStateWith* znajdującą się *DeviceService* odpowiedzialną za zmianę stanu urządzeń, za pomocą innego urządzenia. Ta metoda ma dwa zadania, zmienić stan wszystkich urządzeń w danej grupie, o ile logika biznesowa na to pozwala, oraz zmienić stan we wszystkich grupach połączonych za pomocą relacji.

Uruchomienie powyższych metod powoduje lawinową zmianę stanów wielu urządzeń. Każda zmiana stanu uruchamia zdarzenie *DeviceStateHasChanged,* którego zadaniem jest uaktualnienie stanu urządzeń. Gdy nastąpi zmiana stanu w bazie danych, stan musi się także zmienić na fizycznym urządzeniu, oraz na stronie internetowej. Zdarzenie jest odpowiedzialne, za wysłanie odpowiedniego komunikatu za pomocą interfejsów *SerialPort* oraz *WebSocket.*

## Warstwa komunikacji

Warstwa ta odpowiedzialna jest za komunikację za pomocą protokołu WebSocket, oraz interfejsu Serial Port.

### WebSocket

*UI* dostępny na stronie internetowej jest generowany dynamicznie, dużo informacji jest przesyłanych od klienta na serwer, oraz w przeciwną stronę. *AJAX* jest bardzo dobrym rozwiązaniem, jeśli to klient za każdym razem jest elementem, który wysyła, albo żąda danych. Jednak nie zawsze tak jest. W części strony *WWW*, odpowiedzialnej za kontrole urządzeń, znajduje się wizualizacja przedstawiająca stany urządzeń w domu. Jednym z wymagań projektu, było, aby stany te zmieniały się w czasie rzeczywistym. Często zdarza się tak, że stan urządzenia zmienia się i to serwer musi powiadomić o tym klienta. Jedynym sposobem, by to rozwiązać za pomocą zapytań *AJAX*, jest by klient, co jakiś czas wysyłał zapytania do serwera i pobierał informację na temat stanu urządzeń. Nie jest to najbardziej optymalne rozwiązanie, mimo, że spełnia swoją funkcję. W tym przypadku postanowiono zdecydować się na serwer *WebSocket*, który będzie wysyłał informację o zmianie stanów urządzeń do wszystkich swoich klientów. Wszystkie współczesne wyszukiwarki wspierają protokół WebSocket, a samo rozwiązanie ma wiele zalet. Komunikacja za pomocą tego protokołu jest bardzo szybka i nie obciąża połączenia, a co najważniejsze jest typu *full duplex*, co oznacza możliwość przesyłania wiadomości w dwie strony jednocześnie.

webSocketServer = new WebSocketServer(4649);

AddWebSocketServiceWithAuthentication("/Control");

webSocketServer.Start();

Zdecydowano się na bibliotekę *WebSocketSharp* pozwalającą na stworzenie serwera w bardzo prosty sposób, jednocześnie dając duże możliwości konfiguracji. Stworzenie serwera polega na trzech krokach. Należy zainicjalizować serwer podając port, na jakim ma działać, następnie dodać dowolną liczbę serwisów obsługiwanych przez serwer, a na końcu uruchomić serwer, który automatycznie stworzy sobie wątek, na którym będzie obsługiwać wszystkie zapytania.

private void AddWebSocketServiceWithAuthentication(string url)

{

webSocketServer.AddWebSocketService(url, () => new DeviceControlHub()

{

CookiesValidator = (req, res) => (req.Cast<Cookie>()

.Where(cookie => cookie.Name == ".ASPXAUTH"

|| cookie.Name == ".MONOAUTH")

.Select(cookie => FormsAuthentication.Decrypt(cookie.Value))

.Where(formsAuthenticationTicket =>

formsAuthenticationTicket != null)

.Select(authTicket =>

\_authenticationService.GetRolesForUser(authTicket.Name)))

.Any(roles => roles.Contains("Administrator")

&& roles.Contains("User"))

});

}

Metoda *AddWebSocketServiceWithAuthemtication* dodaje serwis, jednocześnie konfigurując go w taki sposób, by możliwe było podłączenie się do serwera dopiero po zalogowaniu się na stronie i posiadania odpowiednich uprawnień. Tutaj dochodzi kolejna wielka zaleta protokołu WebSocket. Podczas połączenia klient wysyła w nagłówku pliki *cookies*, które serwer może zanalizować i podjąć decyzję o zezwoleniu na połączenie.

Gdy klient połączy się z serwer wysyła mu nagłówek z *cookies*, serwer następnie sprawdza, czy któreś z nich ma nazwę *ASPXAUTH* lub *MONOAUTH*. Zawiera ono zaszyfrowaną nazwę użytkownika, pierwsze to oficjalna implementacja znajdująca się w *ASP.NET*, drugie jest implementacją stworzoną dla *mono*. Następnie zadaniem serwera jest rozszyfrowanie wartości tego cookie, co skutkuje otrzymaniem nazwy użytkownika. Na koniec serwer odwołuje się do serwisu domeny autentykacji i pobiera dla danego użytkownika role, jakie posiada, by sprawdzić, czy ma odpowiednie uprawnienia dla połączenia. Tylko użytkownicy z rolami *User* i *Administrator* mogą kontrolować urządzenia, oraz sprawdzać ich stan, dlatego też, tylko oni mają dostęp do serwera *WebSocket*. Jeśli serwer wykryje inne role, zamknie połączenie z klientem.

### SerialPort

*Serial Port* jest odpowiedzialny za komunikację z modułem, posiadającym możliwość komunikacji za pomocą sieci radiowej po częstotliwości 2.4GHz. Odbiera on informację od innych urządzeń, a potem wysyła odpowiednie wiadomości do serwera, którego zadaniem jest zaktualizować stan urządzenia w bazie danych, oraz *UI*. Serwer musi także wykryć wszystkie inne urządzenia połączone za pomocą grupy, albo relacji, i także adekwatnie zaktualizować ich stan. Następnie musi go rozesłać nie tylko za pomocą protokołu WebSocket do *UI*, oraz zaktualizować danych w bazie danych. Jego zadanie polega także na wysłaniu odpowiednich wiadomości poprzez interfejs *Serial Port* do modułu, który roześle go do fizycznych urządzeń drogą radiową.

SerialPort = new SerialPort("/dev/ttyAMA0", 115200);

if (!SerialPort.IsOpen)

{

SerialPort.Open();

Thread.Sleep(500);

}

InitialiseDevice();

InitialiseSendingThread();

InitialiseReadThread();

W projekcie wykorzystano bibliotekę do obsługi interfejsu *Serial Port* znajdującą się w domyślnie *.NET framework*. Jest ona dostępna w przestrzeni nazw *System.IO.Port*. Korzystanie z niego jest bardzo proste, jedynymi wymaganym parametrem, jaki trzeba skonfigurować jest prędkość przesyłu danych, by go dostosować do prędkości, z jakiej korzysta moduł radiowy, oraz nazwa portu, z jakiego chcemy skorzystać.

void InitialiseReadThread()

{

var readingThread = new Thread(() =>

{

while (SerialPort.IsOpen)

{

if (SerialPort.BytesToRead > 0)

ReadAndParseData();

Thread.Sleep(200);

}

});

readingThread.Start();

}

Po zainicjalizowaniu portu, należy go otworzyć, jeśli nie jest otwarty, by móc rozpocząć komunikację. Metoda *InitialiseReadThread* tworzy wątek, odpowiedzialny za sczytywanie danych, które serwer otrzymuje za pomocą interfejsu *Serial Port*. Standardowym rozwiązaniem jest przypięcie metody do zdarzenia *DataReceived* zainicjalizowanego w klasie *Serial Port*. Uruchamia się ono za każdym razem, gdy pojawiają się dane do odczytania. Niestety w mono funkcja nie została zaimplementowana, dlatego w projekcie musiano znaleźć inne rozwiązanie do odczytywania danych.

void ReadAndParseData()

{

var deviceSerialized = SerialPort.ReadLine();

var msg = deviceSerialized.Split(';');

if (msg.Length != 6) return;

if (int.Parse(msg[2]) == 1 && int.Parse(msg[4]) == 24)

\_deviceService

.UpdateDeviceStateFromSerialPort(int.Parse(msg[0]),

int.Parse(msg[5]));

else if (int.Parse(msg[2]) == 3 && int.Parse(msg[4]) == 3)

{

var address = \_deviceService.GetAddresForPhisicalDevice();

SerialPort

.WriteLine(MessageFactory.CreateSetAddressMessage(address));

}

}

W metodzie tworzony jest nowy wątek z pętlą działającą dopóki port jest otwarty, sprawdzającą z interwałem 200 milisekund, czy pojawiły się nowe dane do odczytania. Jeśli tak, metoda *ReadAndParseData* sprawdza, czy dane są w prawidłowym formacie i podejmuje odpowiednie działanie w zależności od rodzaju danych.

Zadaniem metody jest zidentyfikowanie rodzaju przychodzących danych. Może to być wiadomość sygnalizująca zmian stanu urządzenia. Wtedy uruchamiana jest metoda z serwisu z warstwy aplikacji z domeny DeviceDomain, której zadaniem jest aktualizowanie stanów urządzeń, oraz uruchomienie odpowiednich zdarzeń.

Poprzez *Serial Port* można wysyłać jedną wiadomość na raz. Jednak mamy do czynienia z aplikacją internetową, do której może być podłączonych wielu klientów, i każdy z nich może komunikować się z urządzeniami. Jest to standardowy problem typu producent i konsument. W tym przypadku mamy wielu producentów, ale tylko jednego konsumenta. By rozwiązać ten problem została użyta kolekcja *BlockingCollections* z przestrzeni nazw *System.Collection.Concurrent*. Kolekcja jest zaimplementowana w taki sposób, że tylko jeden wątek na raz może dodawać elementy do kolekcji. Inne wątki czekają w kolejce, aż dostęp do kolekcji zostanie odblokowany.Gdy nowy element pojawi się w kolekcji, może on zostać skonsumowany.

void InitialiseSendingThread()

{

var thread = new Thread(() =>

{

foreach (var device in BlockingCollection.GetConsumingEnumerable())

{

if (SerialPort != null && SerialPort.IsOpen)

SerialPort.WriteLine(MessageFactory

.CreateChangeStateMessage(device.Address,

device.DeviceState));

}

});

thread.Start();

}

W *InitialiseSendingThread* jest tworzona pętla, której zadaniem jest przejście po elementach zwróconych przez metodę zawartą w kolekcji *GetConsumingEnumerable*. Jest ona o tyle specyficzna, że obieg pętli wykonuje się dopiero, gdy w kolekcji pojawi się element do skonsumowania. Jeśli kolekcja jest pusta, pętla nie wykonuje się i czeka na pojawienie się następnego obiektu, jednocześnie blokując wątek. Gdy wszystkie dostępne elementy zostaną skonsumowane, pętla ponownie wchodzi w stam wstrzymania. Jeśli zostanie dodany nowy element do kolekcji, pętla wykona się raz, w tym przypadku wysyłając wiadomość po interfejsie *Serial Port* do modułu zajmującego się komunikacją radiową, a następnie usunie ten konkretny element z kolekcji.

## Warstwa harmonogramu

Ważnym elementem dla użytkownika jest zautomatyzowanie pewnych zadań. Chcemy, by ogrzewanie wyłączyło się na noc, kiedy idziemy spać, oraz włączyło się odpowiednio wcześnie, by nagrzać dom, zanim wstaniemy. Wprowadza to znaczące oszczędności. Gdy jesteśmy na wakacjach, naszym zmartwieniem jest bezpieczeństwo domu, dlatego też wiele współczesnych systemów jest w stanie symulować obecność domowników, na przykład poprzez zapalanie świateł w ciągu dnia.

Biblioteka *Quartz*.*NET* służy do planownia i wykonywania zadań. Została stworzona na podstawie biblioteki *Quartz* napisanej w języku Java.

Użytkownik może stworzyć zadania, które będą się wykonywać o określonej dacie, lub zadanie, które się będzie wykonywać cyklicznie. Aby umożliwić tworzenie takich zadań, które będą powtarzalne i dać dużą możliwość ich konfiguracji, w bibliotece wprowadzono możliwość korzystania z wyrażeń *cron*. Zostały one wprowadzone w systemach typu Unix, by dać możliwość planowania zadań w systemie. Wyrażenie ma postać łańcucha znaków i może wyglądać następująco*: „0 15 10 ? \* MON-FRI 2014”.* Ten ciąg znaków definiuje, że zdarzenie będzie się wykonywać o 10.15, od poniedziałku do piątku, dopóki będzie trwał rok 2014. Użytkownik ma możliwość tworzenia tych wyrażeń, za pomocą interfejsu graficznego, więc nie musi posiadać umiejętności tworzenia wyrażeń *cron.*

public void Initialise()

{

SchedulerFactory = container.Resolve<ISchedulerFactory>();

Scheduler = SchedulerFactory.GetScheduler();

Scheduler.Start();

\_deviceService.GetAllJobs().ForEach(AddJob);

}

Obiekt *Scheduler*, którego zadaniem jest obsługa zadaniem jest tworzony za pomocą fabryki o interfejsie *ISchedulerFactory*. Fabryka to wzorzec projektowy, który ma za zadanie tworzyć obiekty należące do tej samej rodziny, w tym przypadku, obiekt typu *IScheduler*. Fabryka została tutaj stworzona za pomocą kontenera *DI*, by móc wprowadzić wstrzykiwanie zależności do utworzonych zadań. Następnie uruchamiamy wątek, na którym będzie działał *Scheduler* za pomocą zaimplementowanej w nim metody *Start*. Ostatnia linijka kody jest odpowiedzialna za pobranie wszystkich zadań z bazy danych i uruchomienie ich.

public void AddJob(DeviceJobDto dJob)

{

var job = JobBuilder.Create<ChangeGroupStateJob>()

.WithIdentity(dJob.Id.ToString())

.UsingJobData("jobId", dJob.Id)

.Build();

if (!dJob.IsCron)

{

var trigger = TriggerBuilder.Create()

.WithIdentity(dJob.Id.ToString())

.StartAt(dJob.DateTime)

.Build();

Scheduler.ScheduleJob(job, trigger);

}

else

{

var trigger = TriggerBuilder.Create()

.WithIdentity(dJob.Id.ToString())

.WithCronSchedule(dJob.CronExpression)

.Build();

Scheduler.ScheduleJob(job, trigger);

}

}

Metoda *AddJob* jest odpowiedzialna za dodawanie nowych zadań do obiektu *Scheduler*. Są potrzebne do tego dwa elementy: zadanie, które będzie wykonywane, oraz trigger definiujący, kiedy ma zostać uruchomione. Pierwszym krokiem jest stworzenie zadania za pomocą wzorca projektowego zwanego budowniczym. Proces kreacji obiektu za pomocą tego wzorca polega na definiowanie parametrów obiektu, który chcemy stworzyć w paru krokach. W tym przypadku, przed stworzeniem obiektu definiujemy jego identyfikator, oraz dane, z jakich będzie korzystał. Dopiero wtedy za pomocą metody *Build* zwracamy gotowy obiekt.

Kolejnym krokiem jest stworzenie dla zadania obiektu typu *ITrigger*. Jak wspomniano wcześniej, zadanie może być zdefiniowane za pomocą daty i wtedy wykonuje się jednorazowo, lub za pomocą wyrażenia *cron*. Na końcu dodajemy do obiekty *Scheduler* zadanie, wraz z odpowiadającym mu obiektem *ITrigger*.

W przypadku tego projektu tworzony jest tylko jeden rodzaj zadania, który jest zdefiniowany za pomocą klasy *ChangeGroupStateJob*. Implementuje ona interfejs *IJob*, posiadający tylko jedną metodę *Execute*. I to właśnie ta metoda jest wykonywana, gdy warunku określone przy tworzeniu obiektu *ITrigger* spełnią się. W zadaniu wstrzyknięta została zależność do serwisu typu *IDeviceService*, by dać dla zadania możliwość kontroli urządzeń. Metoda, posiadając identyfikator zdarzenia, wyciąga je z bazy danych za pomocą serwisu, a następnie wykonuje metodę *SetAllDevicesWithinGroup* z parametrami zdefiniowanymi przy tworzeniu zdarzenia.

public class ChangeGroupStateJob : IJob

{

private readonly IDeviceService \_deviceService;

public ChangeGroupStateJob(IDeviceService deviceService)

{

\_deviceService = deviceService;

}

public void Execute(IJobExecutionContext context)

{

var dataMap = context.JobDetail.JobDataMap;

var id = dataMap.GetLongValue("jobId");

var job = \_deviceService.GetDeviceJob(id);

if (job == null) return;

\_deviceService.SetAllDevicesWithinGroup(job.GroupId, job.State);

if (!job.IsCron)

\_deviceService.RemoveJob(id);

}

}

## Warstwa dostępu do bazy danych

Przy tworzeniu bazy danych postanowiono skorzystać z *ORM* (ang. *Object-Relational Mapping*), czyli sposobu pozwalającego na odwzorowanie obiektowej architektury warstwy aplikacji, na relacyjną architekturę bazy danych.

W projekcie zdecydowano się na wykorzystanie *Entity Framework*, czyli systemu *ORM* stworzonego przez firmę *Microsoft*. Zdecydowano się na to rozwiązania z powodu jego kompatybilności z mono, oraz z bazą danych *MySql*. Firma odpowiedzialna za rozwijanie tej bazy danych, *Oracle*, dostarcza narzędzi pozwalających na integrację bazy danych z *Visual Studio*, oraz biblioteki umożliwiające połączenie się z bazą danych za pomocą *Entity Framework.*

*Entity Framework* wprowadził innowacyjne podejście zwane *Code First* [6]. Umożliwia ono stworzenie bazy danych na podstawie obiektowej architektury warstwy aplikacji, oraz konfiguracji za pomocą *Fluent Api*, które umożliwia odseparowanie niezbędnych konfiguracji dla bazy danych, od modelu biznesowego.

Fragment kodu poniżej przedstawia jedynie część klasy *HomeCenterDb*. Najważniejsze metody w klasie to *OnModelCreating*, w niej znajdują się wszystkie konfiguracje dotyczące utworzenia tabel w bazie danych. Jako przykład wykorzystałem klasę *Building*. Na początku wymagana jest zmiana parametrów w procedurach tworzonych przez *Entity Framework*. Jest to niezbędne z powodu, że domyślnie nazwy parametrów pokrywają się z nazwami kolumn i baza danych *MySql* nie potrafi rozróżnić kolumn Id od parametru Id. W kolejnym kroku zaznaczamy, że klasa posiada klucz główny o nazwie Id. Na końcu ustalona została relacja jeden do wielu w stosunku do klasy *Floor*. Jak można zobaczyć składnia jest bardzo intuicyjna. Wszystkie tablice bazy danych zostały zainicjalizowane za pomocą tej metody.

public class HomeCenterDb : DbContext, IDbContext

{

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<Building>()

.MapToStoredProcedures(s => s

.Update(u => u.Parameter(p => p.Id, "building\_Id"))

.Delete(u => u.Parameter(p => p.Id, "building\_Id")))

.HasKey(b => b.Id)

.HasMany(b => b.Floors)

.WithRequired()

.HasForeignKey(f => f.BuildingId);

base.OnModelCreating(modelBuilder);

}

public new IDbSet<T> Set<T>() where T : class

{

return base.Set<T>();

}

}

Drugim ważnym elementem tej klasy jest generyczna metoda *Set*. Przesłania ona metodę znajdującą się w klasie abstrakcyjnej *DbContext* i zwraca typ implementujący interfejs *IDbSet* zawierający elementy żądanego przez nas typu. Działania, jakie możemy wykonać na tej kolekcji są bardzo podobne do tych, jakie wykorzystujemy do działań na liście. Mamy wiele możliwości, takich jak sortowanie, filtrowanie, dodawanie, czy usuwanie elementów. Ważnym jest, że operacje wykonane na kolekcji typu *IDbSet* są tłumaczone na język *SQL* i wykonywane na bazie danych. Dzięki temu nie musimy ładować wszystkich elementów z tablicy do pamięci i ich sortować, mamy możliwość wysłania zapytania i otrzymania jedynie niezbędnych wartości.

# Wnioski

Stworzony został system prosty w konfiguracji, nawet dla początkującego użytkownika, jednocześnie dostarczający dużych możliwości. Koszt elementów był bardzo mały w stosunku do gotowych rozwiązań. Zautomatyzowanie pewnych funkcji systemu, takich jak adresowanie urządzeń sprawia, że system nie wiele odstaje od swoich budżetowych odpowiedników. Interfejs *WWW*, który daje użytkownikowi możliwość komunikacji z systemem został zaprojektowany i zbudowany za pomocą najnowszych technologii i dostarcza nie tylko niezbędnych funkcjonalności, ale także jest przyjemny dla oka. Dzięki zastosowaniu *responsive design* można korzystać z interfejsu przystosowanego specjalnie dla różnych rozdzielczości i kontrolować urządzenia na przykład za pomocą telefonu komórkowego.

Wszystkie cele pracy zostały zrealizowane. Wdrożony system spełnia wszystkie wymagania, jakie zostały zebrane na początku projektu.

System został stworzony, by z łatwością można go było skalować, w zależności od wymagań. Struktura serwera została zaprojektowana w taki sposób, by dać możliwość rozbudowy systemu o dodatkowe funkcje w przyszłości. Ułatwia to wykorzystanie wysokopoziomowych, języków o statycznym typowaniu, które pozwalają na łatwiejsze zrozumienie zależności znajdujących się w projekcie. Warstwa aplikacji została zaprojektowana za pomocą podejścia *DDD*, co dodatkowo ułatwia rozbudowę systemu. Jest to architektura znajoma dla wielu programistów i pozwala łatwiej zrozumieć wewnętrzną logikę aplikacji. Został zastosowany podział na wiele warstw, które są w dużej mierze od siebie niezależne, co pozwala na modyfikację tylko niektórych elementów, bez ingerencji w inne.

System można rozbudowywać bez ingerencji w kod, poprzez dodawanie do niego kolejnych urządzeń. Urządzenie automatycznie podłączy się do sieci i zostanie zidentyfikowane, użytkownik musi tylko skonfigurować cechy urządzenia z poziomu interfejsu strony *WWW*. Dzięki temu użytkownik może np. kupić tylko niewielką ilość urządzeń, a potem dokupować, lub tworzyć kolejne wedle potrzeb i samemu je dodawać do sieci. Jest to niewątpliwa zaleta w stosunku do instalacji typu *KNX,* gdzie urządzenie musi zostać dodane przez certyfikowanego instalatora.

Wykorzystanie najnowszych technologii, stworzonych przez firmy takie jak *Oracle*, *Google* oraz *Microsoft* powoduje, że system jest w dużym stopniu niezawodny. Przygotowane przez nich oprogramowanie jest przystosowane dla dużych korporacji, dla których niezawodność i stabilność, jest bardzo ważna.

*Raspberry Pi* okazało się doskonałą platformą zarówno dla serwera nadrzędnego jak i multimedialnego. Mimo niewielkich zasobów, nie dało się odczuć wolnego działania serwera, w każdej sytuacji poza samym uruchamianiem, które zajmuje parę minut. Sprzyja temu cel urządzenia, które w przeciwieństwie do tradycyjnych serwerów nie jest mocno obciążane, a dostęp do niego ma ograniczona liczba ludzi. Opóźnienie pomiędzy zadaniem akcji, a reakcją systemu jest minimalne i niemal niezauważalne. Wynika głównie z powodu warstwy dostępu do bazy danych.

Wybrane na potrzeby projektu moduły *nRF24l01+,* zajmujące się komunikacją radiową sprawdziły się doskonale. Wiadomości były wysyłane bardzo szybko, nie stwierdzono także żadnych problemów z zasięgiem. Moduły były w stanie komunikować się ze sobą bez żadnych problemów, nawet jak znajdowały się w różnych pokojach.

Element, o jakie system można rozbudować, to integracja serwera z chmurą, co dałoby mu możliwość przetwarzania danych, na które *Raspberry Pi* ma mało mocy. Można by także stworzyć aplikacje dedykowane na telefony komórkowe, umożliwiającą kontrolę urządzeń, bez potrzeby ładowania strony internetowej.

# Bibliografia

|  |  |
| --- | --- |
| [1] | M. Riley, Inteligentny dom. Automatyzacja mieszkania za pomocą platformy Arduino, systemu Android i zwykłego komputera, Wydawnictwo Helion, 2013. |
| [2] | Cisco, „Internet of Things Counter,” [Online]. Available: http://newsroom.cisco.com/ioe. |
| [3] | A. Klajn, „Automatyka budynkowa - Wybrane systemy inteligentnych instalacji elektrycznych,” *ElektroInstalator,* 7-8 2007. |
| [4] | R. C. Martin, Czysty kod. Podręcznik dobrego programisty, Wydawnictwo Helion, 2010. |
| [5] | E. J. Evans, Domain-Driven Design. Tackling Complexity in the Heart of Software, Wydawnictwo Addison Wesley, 2003. |
| [6] | S. Klein, Pro Entity Framework 4.0, Wydawnictwo Apress, 2010. |
| [7] | B. Green i S. Shyam, AngularJS, Wydawnictwo O’Reilly Media, 2013. |
| [8] | A. Hunt i D. Thomas, Pragmatyczny programista. Od czeladnika do mistrza, Wydawnictwo Helion, 2011. |
| [9] | S. D. Ritchie, Pro .NET Best Practices, Wydawnictwo Apress, 2011. |
| [10] | S. Sobótka, „Domain Driven Design. Sposób na projektowanie złożonych modeli biznesowych,” *Software Developer's Journal,* 8 2011. |
| [11] | Microsoft, „TypesScript HandBook,” 2012. [Online]. Available: http://www.typescriptlang.org/handbook. |
| [12] | Microsoft, „Baza Wiedzy MSDN,” [Online]. Available: http://msdn.microsoft.com/library/. |

# Spis rysunków

[Rys. 1 Raspberry Pi 7](file:///C:\Users\rstraszewski\Dropbox\PracaInżynierska3.docx#_Toc405341556)

[Rys. 2 Atmega328 8](file:///C:\Users\rstraszewski\Dropbox\PracaInżynierska3.docx#_Toc405341557)

[Rys. 3 nRF24l01+ 9](file:///C:\Users\rstraszewski\Dropbox\PracaInżynierska3.docx#_Toc405341558)

[Rys. 4 Architektura projektu 13](file:///C:\Users\rstraszewski\Dropbox\PracaInżynierska3.docx#_Toc405341559)

[Rys. 5 Wzorzec MVC 17](file:///C:\Users\rstraszewski\Dropbox\PracaInżynierska3.docx#_Toc405341560)

[Rys. 6 Diagram zależności modelu biznesowego DeviceDomain 24](file:///C:\Users\rstraszewski\Dropbox\PracaInżynierska3.docx#_Toc405341561)