POLITECHNIKA ŁÓDZKA
Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki
Instytut Automatyki

PRACA DYPLOMOWA MAGISTERSKA

Projekt i realizacja systemu Internet Rzeczy w oparciu o chmurę obliczeniową

(Design and implementation of Internet of Things System based on cloud computing model)

Marcin Jahn
214331

Opiekun pracy:
dr inż. Jarosław Kacerka

Łódź, Lipiec 2018

Spis treści

1 Wstęp

Internet Rzeczy (ang. *IoT – Internet of Things*) to termin, który w ostatnich latach zyskał wielką popularność - słyszał o nim niemal każdy. Nawet osoby niezwiązane profesjonalnie z technologią kojarzą go i są w stanie mniej więcej wyjaśnić na czym polega. Wynika to z faktu, że dziedzina Internetu Reczy cieszy się obecnie ogromnym zainteresowaniem - zarówno ze strony producentów sprzętu elektronicznego, jak i jego nabywców. Powodem tego trendu jest fakt, że idea Internetu Rzeczy okazuje się być przydatna w ogromnej ilości zastosowań. Obecnie implementacje IoT znaleźć można w szerokiej gamie produktów konsumenckich oraz w zastosowaniach profesjonalnych, w przemyśle. WSTAWIĆ INFO O JAKICHŚ BADANIACH. Mimo popularności samego terminu, warto w niniejszej pracy, która traktuje przecież o budowie systemu IoT, przedstawić definicję Internetu Rzeczy. Powinno to stanowić pewnego rodzaju fundament, na podstawie którego wyjaśniane będą założenia i funkcjonalność opisywanego projektu.

Okazuje się, że niełatwo jest przedstawić definicję, która w pełni opisywałaby zagadnienie. Jako dowód tego niech posłuży 86-stronicowy dokument przygotowany przez IEEE (ang. *Institute of Electrical and Electronics Engineers*) [1]. Publikacja ta została stworzona jako próba przedstawienia czym jest IoT, z różnych punktów widzenia. Spośród setek definicji, które zawiera, wybrane zostały dwie, które w moim odczuciu trafnie i wystarczająco opisują temat na potrzeby niniejszego projektu:

> *The Internet of Things (IoT) consists of things that are connected to the Internet, anytime, anywhere. In its most technical sense, it consists of integrating sensors and devices into everyday objects that are connected to the Internet over fixed and wireless networks. The fact that the Internet is present at the same time everywhere makes mass adoption of this technology more feasible. Given their size and cost, the sensors can easily be integrated into homes, workplaces and public places. In this way, any object can be connected and can manifest itself over the Internet. Furthermore, in the IoT, any object can be a data source. This is beginning to transform the way we do business, the running of the public sector and the day-to-day life of millions of people.* [2]

> *The Internet of Things refers to the unique identification and 'Internetization' of everyday objects. This allows for human interaction and control of these 'things' from anywhere in the world, as well as device-to-device interaction without the need for human involvement.* [3]

Ostatecznie więc, na podstawie przedstawionych cytatów, można wyróżnić kilka punktów, które charakteryzują system Internetu Rzeczy:

- obecność „rzeczy" (urządzeń, sensorów, itd.), które mają stałe połączenie z Internetem;
- łatwość obsługi oraz dostępność elementów składających się na system;
- każda końcówka systemu stanowi źródło danych;
- sterowanie urządzeń przez Internet, bez bezpośredniego kontaktu człowieka;
- możliwość komunikacji między urządzeniami.

W związku z tym, że IoT może być definiowane na tak wiele sposobów, nie dziwi różnorodność oferowanych rozwiązań, które z tej idei korzystają. Jednocześnie jednak można zwrócić uwagę na fakt, że sama etykieta 'IoT' niewiele mówi nam o tym jakie funkcjonalności oferuje dany produkt. Czy chodzi o wysyłanie danych telemetrycznych? Czy może dane urządzenie potrafi komunikować się z innymi urządzeniami? A może jeszcze coś innego? Internet Rzeczy może reprezentować szeroki wachlarz rzeczywistych możliwości danego urządzenia i sam opisywany termin jest zbyt ogólny by był wystarczający do opisu konkretnego produktu czy usługi. W związku z powyższym, kolejny podrozdział ma na celu wyjaśnienie czym właściwie ma być rezultat niniejszej pracy magisterskiej.

1.1 Cel i zakres pracy

Celem pracy jest zaprojektowanie oraz praktyczna realizacja architektury systemu Internet of Things (IoT) bazującego na wykorzystaniu chmury obliczeniowej. System ma umożliwić tworzenie wirtualnych połączeń pomiędzy urządzeniami sterującymi oraz wykonawczymi. Każde z urządzeń zostanie stworzone w oparciu o mikrokontroler wyposażony w łączność wi-fi, w celu komunikacji obustronnej z częścią systemu znajdującą się w chmurze. System powinien pozwolić na dostęp wielu użytkowników, gdzie każdy ma swoją własną pulę urządzeń oraz zestaw reguł z nimi związanych. Konfiguracja urządzeń powinna odbywać się poprzez dedykowaną dla rozwiązania aplikację, dostępną z poziomu smartfonu/tabletu bądź komputera.

Powyższy akapit to opis pracy przedstawiony przy zgłaszaniu tematu dla Komisji. Wymaga on jednak pewnego rozszerzenia.

Budowany system (dalej często nazywany „platformą") ma bazować na wykorzystaniu chmury obliczeniowej. Oznacza to, że główna funkcjonalność i logika projektu ma być umieszczona „w chmurze", czyli na zewnętrznych serwerach, których utrzymaniem i ciągłością działania zajmuje się dostawca usług chmurowych. Zadaniami części chmurowej są:

- zapis danych, które przesyłane są z urządzeń podłączonych do Internetu;
- pośredniczenie w komunikacji między urządzeniami.

Drugi z wymienionych punktów jest rezultatem wymagania, aby końcówki systemu miały możliwość przesyłania między sobą informacji. Każde z urządzeń powinno posiadać pewien określony zbiór właściwości. Właściwości te, to konkretne cechy urządzenia opisujące jego stan. Istotą systemu jest możliwość łączenia poszczególnych właściwości różnych urządzeń, aby wpływały na siebie. Określenie tychże właściwości to część procesu modelowania urządzeń, czyli zbudowania listy wymagań na temat tego, jakie cechy powinny charakteryzować dany typ urządzenia. Więcej na temat modelowania można przeczytać w rozdziale „Modelowanie oraz łączenie urządzeń". Ogólna idea polega na tym, aby użytkownik miał możliwość tworzenia połączeń między wybranymi urządzeniami – w taki sposób, że jedno staje się sterownikiem, a inne aktuatorem. Ważną kwestią jest fakt, że połączenia takie powinny być możliwe do określenia w relacji wiele-do-wielu, to znaczy dowolna ilość urządzeń może sterować dowolną ilością innych urządzeń.

Opis przedstawiony na początku informuje, że każde z urządzeń powinno być oparte o mikrokontroler (wyposażony w łączność wi-fi). Jednak w trakcie realizacji projektu, pomysł ten został rozszerzony – jako urządzenie rozumiany jest dowolny „element" będący w stanie podłączyć się do platformy. Może to więc być nie tylko mikrokontroler, ale także np. aplikacja komputerowa, która z wykorzystaniem pewnych bibliotek programistycznych może uzyskać dostęp do systemu. W pracy przyjęta została nomenklatura, według której każda „rzecz" podłączona do systemu jest urządzeniem – niezależnie od tego czy jest to fizyczny mikrokontroler, czy program komputerowy.

Użytkownicy systemu powinni mieć możliwość dokonywania jego konfiguracji – dotyczy to głównie definiowania połączeń między własnymi urządzeniami. W tym celu należało stworzyć aplikacje dostępowe, za pośrednictwem których użytkownik mógłby się zalogować z dowolnej platformy (PC (Windows/Linux), Mac, iOS, Android). Lepszym w utrzymaniu rozwiązaniem

byłoby jednak utworzenie jednej aplikacji, która byłaby dostępna niezależnie od platformy jaką posiada użytkownik. Taka realizacja była możliwa dzięki obecności uniwersalnych i responsywnych technologii webowych.

Poszczególne części, które ogólnikowo zostały przedstawione w poszczególnych akapitach powyżej, zostaną znacznie bardziej szczegółowo opisane w następnych rozdziałach. Autor pracy traktował jej realizację jako okazję do poznania nowych narzędzi i zagadnień, w związku z czym starał się przedstawić proces tworzenia projektu z uwzględnieniem poczynionych obserwacji na ten temat.

Ze względu na fakt, że owocem pracy jest pewien produkt/usługa, należało go nazwać. W dalszej części pracy pojawiać się więc będzie określenie *MJIoT*, które wybrane zostało jako wspomniana nazwa. Pochodzi ono od moich inicjałów (MJ) oraz członu IoT.

1.2 Przegląd rozwiązań obecnych na rynku

Zanim przejdę do opisu własnej implementacji systemu Internetu Rzeczy, warto poświęcić chwilę na zapoznanie się z rozwiązaniami, które są obecnie dostępne.

1.2.1 IFTTT

Ilustracja 1.1 Logo usługi IFTTT

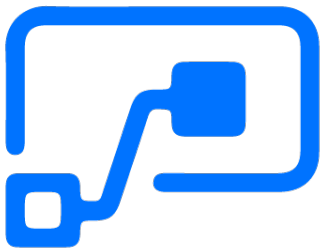
IFTTT (If This Then That) to platforma, która powstała w 2011 roku. Jej funkcjonalność polega na tworzeniu połączeń między wybranymi usługami przy wystąpieniu określonych warunków. Przykładowo, można zdefiniować połączenie między serwisem pogodowym a usługą mailową, w taki sposób, aby zawsze w przeddzień deszczowej pogody, użytkownik był o niej informowany. Siła rozwiązania polega na tym, że w tworzenie poszczególnych, dostępnych użytkownikowi, serwisów, zaangażowały się liczne firmy. Przykładem jest Google, które udostępnia usługi związane, między innymi, z Google Assisstant czy YouTube.

Jakiś czas temu, po zdobyciu popularności, do listy dostępnych serwisów dołączone zostały urządzenia (m. in. żarówki, przyciski). Pozwoliło to IFTTT stać się w wielu przypadkach rozwiązaniem z kategorii Smart Home, czyli automatyki domowej.

Omówione rozwiązanie było jedną z głównych inspiracji mojego projektu. Idea łączenia różnych usług/przedmiotów ze sobą, według upodobania użytkownika, jest bardzo ciekawym pomysłem. Pewnym minusem platformy jest jednak fakt, że jest to rozwiązanie zamknięte – użytkownik ma dostęp jedynie do serwisów i akcji, które przeszły proces certyfikacji IFTTT i zostały udostępnione globalnie. Tworzenie własnych urządzeń nie jest możliwe, zamiast tego należy kupić gotowe, dosyć drogie komponenty przygotowane przez firmy trzecie.

IFTTT jest niewątpliwie rozwiązaniem innowacyjnym i ciekawym. Jest to usługa komercyjna, która oferuje gotowe rozwiązania, bez możliwości tworzenia własnych komponentów systemu.

1.2.2 Microsoft Flow



Ilustracja 1.2 Logo usługi Microsoft Flow

Powstały w 2016 roku serwis Flow jest niewątpliwie odpowiedzią firmy Microsoft na przedstawiony wcześniej IFTTT. Zasada działania projektu jest praktycznie ta sama, jednak dostępność usług, które można ze sobą łączyć jest, w porównaniu do IFTTT, dosyć niewielka. Microsoft Flow został umieszczony w niniejszym zestawieniu, aby pokazać, że koncepcja łączenia ze sobą różnych serwisów/produktów jest tematem, którym interesują się obecnie największe korporacje świata technologii.

Szukając informacji na temat podobnych rozwiązań okazuje się, że, oprócz opisanych, dostępnych jest wiele mniejszych rozwiązań, które bazują na tym samym schemacie, m. in.: Zapier, Automate.io, Huggin.

1.2.3 Ubidots

Ilustracja 1.3 Logo platformy Ubidots

Kolejna z omawianych usług to, w przeciwieństwie do poprzednich, rozwiązanie kierowane głównie do przemysłu. System ten umożliwia kolekcję oraz wizualizację danych. Wyróżnia się mnogością dostępnych bibliotek programistycznych oraz łatwością dostosowania rozwiązania do swojej marki. Oprócz tego istnieje możliwość ustawienia powiadomień (mail, sms, itp.) w reakcji na określone zdarzenia.

Ważną kwestią w przypadku tego systemu jest jego cena. W przypadku potrzeby podłączenia do 60 urządzeń, koszt wynosi $99 miesięcznie – jest to usługa oferowana na zasadzie subskrypcji. Cena rośnie wraz z ilością urządzeń, które posiada klient.

Dwie pierwsze z przedstawionych ofert są bardzo podobne. Różnią się jednak znacznie od rozwiązania trzeciego (Ubidots), które skupia się na zbieraniu danych z podłączonych urządzeń i wizualizacji ich. Udowadnia to jedynie, że świat IoT jest bogaty w różne spojrzenia na to, czym tak naprawdę Internet Rzeczy jest. Wybrane rozwiązania nie zostały zaprezentowane przypadkowo. Projekt będący owocem niniejszej pracy stanowi próbę połączenia obu z przedstawionych scenariuszy:

- łączenie z sobą różnych „rzeczy”;
- gromadzenie danych z każdej z „rzeczy” i ich wizualizację.

2 Usługi chmurowe

Istotnym jest by przybliżyć czytelnikowi czym są usługi chmurowe. Stanowią one dużą część projektu MJIoT, zostały zresztą wspomniane w samym tytule pracy. Chcąc omówić temat tego podrozdziału, najlepiej chyba odwołać się do definicji zagadnienia sformułowanej przez jednego z największych graczy na tym rynku – firmę Amazon:

> *Cloud computing is the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing [4]*

2.1 Dostępne serwisy

Usługi chmurowe można więc najprościej określić jako zasoby informatyczne do wynajęcia. Zasoby te udostępniane są jako konkretne serwisy, najpopularniejsze z nich to np.:

- maszyny wirtualne,
- serwery baz danych bądź inne opcje przechowywania danych,
- pakiety biurowe,
- systemy wizualizacji danych.

Nie sposób wymienić wszystkich możliwych serwisów, gdyż każdy dostawca tego rodzaju rozwiązań stara się zaproponować coś, czego nie posiada konkurencja (bądź coś co byłoby lepsze). Obecnie mamy do czynienia z sytuacją, gdzie na rozwój chmur przeznaczane są ogromne inwestycje i w niedługich odstępach czasu pojawiają się kolejne nowości. Wystarczy np. śledzić wiadomości w kanale informacyjnym firmy Microsoft, by dowiedzieć się, że mniej więcej co kilka tygodni oferta platformy Azure (usługi chmurowej w ofercie Microsoftu) wzbogacana jest o nowe funkcjonalności.

Istotną cechą usług chmurowych jest metoda naliczania opłat. Zazwyczaj dostępne są dwie opcje:

- stała subskrypcja na określoną kwotę pieniężną, której nie można przekroczyć,
- tzw. Pay-As-You-Go, czyli płatność kwoty zgodnej z rzeczywistym użyciem.

Zaletami stosowania modelu chmurowego w stosunku do wcześniejszych są przede wszystkim:

- przeniesienie odpowiedzialności za utrzymanie infrastruktury na inny podmiot,
- możliwość większego skupienia się na budowie właściwego rozwiązania,

- łatwość skalowania rozwiązania w przypadku zmiany potrzeb na zasoby (również automatycznego, bez manualnej kontroli ze strony człowieka),
- dostępność ogromnej ilości serwisów w jednym miejscu – często z możliwością łatwej komunikacji między nimi.

Uważam, że ostatni punkt jest bardzo ważny. Dostawcy usług chmurowych, ze względu na silną konkurencję, starają się oferować swoje usługi w sposób, który jest jak najłatwiejszy do wykorzystania przez klienta. Oznacza to m. in. łatwość konfiguracji, która przekłada się bezpośrednio na mniejszy czas na nią poświęcony. Czas ten można przeznaczyć na dodatkowy rozwój własnego rozwiązania, co może wpłynąć na jego lepszą jakość czy konkurencyjność. Obecnie w informatyce mamy do czynienia z ogromną popularnością architektury mikro-serwisów, które idealnie wpisują się w warunki oferowane przez dostawców usług chmurowych. Izolacja poszczególnych komponentów aplikacji możliwa jest poprzez oddelegowanie oddzielnych zasobów/serwisów do obsługi konkretnego elementu. Połączenie tychże komponentów ze sobą jest znacznie ułatwione, jeśli znajdują się one na tej samej platformie. Często takie połączenia wymagają jedynie wybrania z listy konkretnych usług, które mają się komunikować – odpowiedzialność za zapewnienie poprawnej współpracy między nimi leży więc w dużej części po stronie dostawcy usługi, ponownie zwalniając konkretnego jej klienta z części obowiązków.

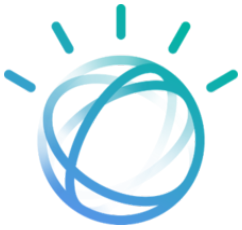
2.2 Dostawcy usług chmurowych

Oferowanie usług chmurowych wiąże się z ogromnymi inwestycjami. W związku z tym nie dziwi, że najwięksi gracze w tej dziedzinie to firmy, które istniały często na długo zanim nastąpiła „rewolucja” chmurowa. Magazyn Forbes wylistował ich wg przychodów (dane za ostatni kwartał 2017 roku) [5]:

1. IBM Watson ($5,5B)
2. Microsoft Azure ($5,3B)
3. Amazon AWS ($5,1B)
4. Salesforce Cloud ($2,68B)
5. Oracle Cloud ($1,5B)
6. SAP ($1,24B)
7. Google Cloud ($1B)

Warto zastanowić się, czym wyróżniają się poszczególni dostawcy, aby być w stanie podjąć decyzję nt. platformy, o którą oparty zostanie projekt.

2.2.1 IBM Watson



Ilustracja 2.1 Logo IBM Watson

Oferta korporacji IBM wyróżnia się przede wszystkim dużym naciskiem na uczenie maszynowe. Jest więc ona przeznaczona do rozwiązań, które mają bazować na „inteligentnym" operowaniu na danych wejściowych. Przykładowe wykorzystanie to m. in.:

- chatboty potrafiące przeprowadzić naturalną rozmowę z człowiekiem,
- analiza różnego rodzaju danych.

IBM Watson został przeze mnie odrzucony we wczesnym etapie planowania projektu ze względu na inny sposób rozumienia Internetu Rzeczy. Wizja IBM skupia się przede wszystkim na analizie danych z wykorzystaniem zaawansowanego uczenia maszynowego, co nie jest częścią mojego projektu. Ogromne zyski firmy IBM wskazują jednak, że przyjęła ona słuszny kierunek rozwoju swojej platformy, prawidłowo odczytując zapotrzebowania rynku.

2.2.2 Microsoft Azure

Ilustracja 2.1 Logo Microsoft Azure

Firma Microsoft znana jest z wielu inicjatyw związanych z informatyką, nie mogło zabraknąć jej również w tym zestawieniu. Platforma Microsoft Azure jest obecnie źródłem największego dochodu firmy i stanowi cel jej największych inwestycji. Start usługi nastąpił w lutym 2010 roku. Spośród zalet tego usługodawcy można podkreślić:

- ogromną ilość oferowanych usług,

- gwarantowana dostępność na poziomie ok. 99,9% [6],
- dostępność interfejsów programistycznych dla wielu środowisk/języków, przede wszystkim: .NET, Python, Java, NodeJs,
- możliwość korzystania z platformy z opcją płatności Pay-As-You-Go,
- dostępność w ogromnej ilości regionów na całym świecie.

W przeciwieństwie do oferty IBM, propozycja Microsoftu zawiera niezbędne komponenty do realizacji projektu MJIoT. Ważną kwestią jest dostępność usług związanych z IoT (IoT Hub, który omówiony zostanie w dalszej części pracy) oraz dobre wsparcie dla programistów w postaci pakietów SDK (ang. *Software Develpment Kit*) oraz dokumentacji.

2.2.3 Amazon AWS

Ilustracja 2.2 Logo Amazon AWS

Amazon, dawniej kojarzony głównie jako księgarnia internetowa, znajduje się bardzo wysoko na przedstawionej wcześniej liście. Wynika to przede wszystkim z faktu, że jest to najbardziej doświadczony z przedstawionych usługodawców – AWS (Amazon Web Services) pojawił się na rynku w marcu 2006 roku. Z tego powodu cieszy się sporym zaufaniem klientów, dla których liczy się „dojrzałość" wykorzystywanego środowiska.

Po krótkim omówieniu platformy Azure, ciężko tak naprawdę wskazać cechy odróżniające AWS - obie propozycje są do siebie dosyć podobne, głównie z powodu starań Microsoftu by dorównać Amazonowi. Spośród dostępnych serwisów, duża część z nich dostępna jest na obu platformach. Wydaje się, że obie nadają się do zastosowania w niniejszym projekcie. Należy jednak dokonać konkretnego wyboru, przyjrzyjmy się więc dokładniej ofercie związanej z IoT w przypadku obu platform.

2.3 Modele dostarczania usług chmurowych

Mając do czynienia z chmurą, nie sposób nie natknąć się na określenia: SaaS, PaaS czy IaaS. Wyjaśnienie tych terminów pomoże w lepszym zrozumieniu czym tak naprawdę są serwisy

dostępne w chmurze. Jest to ważne również z tego względu, że projekt opisywany przez niniejszą pracę jest przykładem rozwiązania dostarczanego w modelu *SaaS*.

2.3.1 SaaS

SaaS to skrótowiec od angielskiego: *Software as a Service*. Jest to obecnie bardzo popularna forma dostarczania oprogramowania i różni się znacznie od modelu tradycyjnego. Kilka-kilkanaście lat temu, standardem było, że wszelkiego rodzaju aplikacje należało zainstalować na komputerze docelowym. Rozwiązanie takie stwarzało wiele problemów, szczególnie w kwestiach programistycznych i utrzymaniowych. Po pierwsze programiści musieli brać pod uwagę, że różni klienci posiadają różne komputery – zarówno jeśli chodzi o sprzęt jak i system operacyjny. Ze względu na koszty, niektóre platformy były często wykluczane z możliwości uruchomienia na nich danego oprogramowania – mogło więc ono trafić do ograniczonego grona użytkowników. Drugą sprawą była potrzeba opracowania procedur związanych z utrzymaniem takiego oprogramowania. Zazwyczaj co jakiś czas pojawiała się potrzeba dokonania aktualizacji, co nie jest trywialną operacją, jeśli mamy do czynienia z dużą ilością klientów. Wszystko to było często powodem dużej ilości problemów, których rozwiązanie wiązało się z kolejnymi kosztami i stratą czasu.

Obecnie panuje inny model dostarczania oprogramowania. SaaS, któremu przeznaczony jest ten podrozdział, właściwie eliminuje opisane problemy. Oprogramowanie nie wymaga już instalacji (bądź wymaga instalacji jedynie niewielkiego klienta dostępowego), lecz jest dostarczane za pośrednictwem Internetu – jako aplikacja webowa. Rzeczywista instancja programu (a raczej mikro-serwisów) działa na serwerach (najczęściej w chmurze). W związku z tym proces rozwoju i utrzymania rozwiązania znacznie uprasza się. Zmiana taka stała się możliwa dzięki kilku czynnikom:

- zwiększenie prędkości łącz internetowych,
- pojawienie się technologii chmurowych, które ułatwiły zarządzanie aplikacjami SaaS,
- pojawienie się technologii webowych, które pozwalają tworzyć zaawansowane aplikacje uruchamiane przez przeglądarkę internetową.

Oczywiście SaaS nie spowodował, że całkowicie zniknęło klasyczne oprogramowanie. Nowy model dystrybucji wykorzystywany jest w sytuacjach, które nie wymagają dużej mocy obliczeniowej (technologie webowe nie zapewnią wymaganej wydajności). Są to więc

najczęściej rozwiązania, które opierają na prezentacji różnego rodzaju danych, chociaż istnieją również bardziej zaawansowane przykłady, jak np. pakiety biurowe.

Inną zaletą SaaS, związaną z tym, że opiera się na przeglądarkach internetowych, jest możliwość uruchamiania aplikacji na szerokiej gamie urządzeń, np. smartfonach, na co w dzisiejszych czasach kładziony jest duży nacisk.

Przykładami rozwiązań typu SaaS są: Microsoft Office Online, SAP, Skype Online, projekt MJIoT.

2.3.2 PaaS

Kolejnym z opisywanych skrótowców jest PaaS – *Platform as a Service*. Platforma to słowo, które często pojawia się w niniejszej pracy, głównie jako synonim ‘dostawcy usług chmurowych’. Jest to nieprzypadkowe określenie, gdyż właśnie do tego odnosi się PaaS. Dostawcy, tacy jak Microsoft czy Amazon, oferują komponenty chmurowe jako elementy budulcowe dla konkretnych rozwiązań, budowanych przez zainteresowane jednostki. Dostarczana jest więc pewnego rodzaju platforma, na której każdy może oprzeć swoje rozwiązania informatyczne. Zalety takiego rozwiązania zostały już przedstawione w podrozdziale „Dostępne serwisy” przy ogólnym omawianiu usług chmurowych.

Przykładami rozwiązań typu PaaS są: Microsoft Azure, Amazon AWS, Google Cloud.

2.3.3 IaaS

IaaS, czyli *Infrastructure as a Service* stanowi nieco inny model dostarczania usług. Tym razem mamy do czynienia z udostępnianiem samej mocy obliczeniowej, określonej jako parametry serwera, który wynajmujemy. Spośród wymienionych modeli dostarczania usług chmurowych, IaaS to model, który charakteryzuje się najniższym poziomem abstrakcji. Klient otrzymuje maszynę, na której może uruchomić dowolne oprogramowanie. Zazwyczaj jest to rozwiązanie stosowane w przypadku, gdy PaaS nie oferuje wymaganych funkcjonalności – np. systemu baz danych, który jesteśmy zmuszeni wykorzystywać.

Aby lepiej wytłumaczyć czym jest IaaS, można wyobrazić sobie sytuację, w której ktoś udostępnia nam swój komputer (o określonej mocy obliczeniowej) na pewien czas. Mamy możliwość dostosowania jego oprogramowania i wykonania swojej pracy. To my bierzemy odpowiedzialność za to, że system operacyjny wraz z dodatkowym oprogramowaniem zostały

poprawnie skonfigurowane. Jedyne o co nie musimy się martwić, to utrzymanie fizycznych komponentów, które zapewniają moc obliczeniową – tym nadal zajmuje się dostawca usług chmurowych.

IaaS zapewnia największą swobodę spośród trzech opisanych modeli. Jest wykorzystywany w różnych sytuacjach, np. do wykonywania obliczeń naukowych bądź jako podstawa do dostarczania rozwiązań typu SaaS przez różne firmy.

2.4 Usługi IoT

Przy omawianiu usług IoT dostępnych na platformach chmurowych skupię się na ofercie firm Amazon (IoT Core) oraz Microsoft (IoT Hub). Pominę innych usługodawców, głownie ze względu na:

- mniejszą popularność (co przekłada się na mniejsze wsparcie),
- niedostępność stabilnej wersji usługi w trakcie prac nad projektem – np. Google opublikowało IoT Core dopiero w lutym 2018 roku,
- niską dostępność bibliotek programistycznych.

2.4.1 Azure IoT Hub

Ilustracja 2.2 Logo Azure IoT Hub

IoT Hub to usługa, która publicznie wystartowała w kwietniu 2016 roku. Poniżej wylistowane zostały jej najważniejsze możliwości:

- komunikacja D2C (Device-to-Cloud) oraz C2D (Cloud-to-Device) z wykorzystaniem protokołów: MQTT, AMQP, HTTPS;
- retencja wiadomości, które nie mogły zostać dostarczone ze względu na status offline urządzenia (do 7 dni);
- autentyfikacja urządzeń;
- możliwość łączenia z wieloma innymi usługami, również z zastosowaniem własnych zasad biorących pod uwagę zawartość przesłanej wiadomości;

- wsparcie dla wielu środowisk programistycznych - C (w tym Arduino), .Net (Framework + Core), Python, NodeJS, Java, Swift, REST API.

2.4.2 AWS IoT Core

Ilustracja 2.3 Logo AWS IoT Core

Data premiery AWS IoT Core to grudzień 2015 roku. Oto dostępne funkcjonalności:

- komunikacja urządzeń z chmurą oraz z innymi urządzeniami za pomocą protokołów: MQTT, HTTP, Websocket;
- autentyfikacja urządzeń;
- możliwość łączenia z innymi usługami AWS, z uwzględnieniem własnych zasad;
- wsparcie dla środowisk: C (w tym Arduino), JavaScript.

2.5 Wybór dostawcy

Jak widać, obie firmy oferują dosyć podobne funkcjonalności. W swoim projekcie ostatecznie zastosowałem rozwiązania Microsoft (chmura Azure), głównie ze względu na:

- dobrą dokumentację z licznymi przykładami,
- bardzo dobrą dostępność pakietów SDK dla dużej ilości środowisk – w tym miejscu warto zwrócić szczególną uwagę na środowisko .NET, z którym mam już pewne doświadczenie, a które zostało stworzone i jest silnie wspierane przez Microsoft,
- wcześniejszą znajomość platformy.

Przemyślany wybór docelowej platformy ma duże znaczenie również ze względu na koszty – korzystanie z zasobów chmurowych nie jest darmowe, warto więc dobrze się zastanowić, aby poświęcone czas i pieniądze nie poszły na marne.

3 Modelowanie oraz łączenie urządzeń

Istotą działania systemu jest obsługa podłączonych do niego urządzeń – są one więc w centrum uwagi projektu. W związku z tym należało dobrze przemyśleć sposób ich definiowania i obsługi przez platformę. Oprócz tego należało się zastanowić jak tak naprawdę zrealizować łączenie urządzeń ze sobą. Czy zawsze jest to możliwe? Jakie dane powinny być przesyłane?

Informacje podane w niniejszym rozdziale zostaną rozszerzone w dalszej części pracy, szczególnie podczas omawiania bazy danych urządzeń.

3.1 Definiowanie urządzeń

Każde urządzenie wewnątrz platformy MJIoT charakteryzowane jest przez dwa zbiory informacji:

- model określający zbiór informacji jaki powinien je reprezentować,
- instancję zawierającą informacje o konkretnym urządzeniu.

W celu lepszego zobrazowania zagadnienia, posłużę się przykładem prostego urządzenia – lampki elektrycznej. Zanim stworzymy konkretną lampkę (np. lampa w pokoju), musimy zdefiniować jego model – samą lampę w ogóle. Model ten powinien zawierać informacje określające zbiór właściwości jakie każda LAMPA będzie posiadać. W tym prostym przykładzie możemy przyjąć, że właściwości te przedstawiają się następująco:

- NAZWA – określa nazwę konkretnej lampy,
- STAN ŻARÓWKI – określa czy żarówka jest zapalona.

Nasz model posiada więc dwie właściwości. Na razie znamy jednak jedynie ich nazwy. Należy dodatkowo zdefiniować pewien zbiór meta-danych, które wzbogacą ich charakterystykę. Platforma pozwala określić następujące dane:

- Typ wartości – format jaki przyjmują wartości danej właściwości. Ma on znaczenie w momencie tworzenia połączenia między urządzeniami, co zostanie omówione w dalszej części pracy. Możliwe opcje to:
  - liczba (*numer*),
  - łańcuch tekstowy (*string*),
  - wartość logiczna (*boolean*).

- Historyzowanie – określa czy dana właściwość powinna być przechowywana na zasadzie telemetrii, tzn. czy interesuje nas każda indywidualna jej wartość, która została zaraportowana przez urządzenie do chmury. Każda z tych wartości posiada dodatkowo informację o czasie jej przesłania (tzw. timestamp). Jeśli nie interesują nas wartości z przeszłości, przechowywana będzie jedynie ostatnia znana wartość. Parametr ten ma znaczenie ze względu na koszty przechowywania danych w chmurze;
- Zdolność sterowania – określa czy dana właściwość może być traktowana jako potencjalna wartość sterująca przy łączeniu z innym urządzeniem. W praktyce, parametr ten decyduje o tym, czy dane z pewnej konkretnej właściwości urządzenia mogą być przesyłane do innego urządzenia, by w jakiś sposób wpłynąć na jego stan;
- Zdolność bycia sterowanym – określa czy dana właściwość może być wykorzystywana w połączeniach jako odbiorca informacji z innych urządzeń.

Kontynuując przykład lamki, możemy wzbogacić informacje na temat modelu. Nasze dwie właściwości można opisać następująco:

- NAZWA:
  - Typ wartości: łańcuch tekstowy (*string*)
  - Historyzowanie – NIE
  - Zdolność sterowania – NIE
  - Zdolność bycia sterowanym – NIE
- STAN ŻARÓWKI:
  - Typ wartości: wartość logiczna (*boolean*)
  - Historyzowanie – TAK
  - Zdolność sterowania – NIE
  - Zdolność bycia sterowanym – TAK

Pozostają jeszcze jedynie dwa parametry dotyczące modelu:

- Abstrakcyjność – określa czy na podstawie danego modelu można tworzyć instancje;
- Aktywna komunikacja offline – określa czy do urządzenia, będącego instancją danego modelu, powinny być przesyłane komunikaty, kiedy nie ma ono połączenia z platformą.

Znaczenie obu parametrów zostanie wyjaśnione w dalszej części pracy. Na razie załóżmy, że nasz model lampki:

- nie jest abstrakcyjny,

- nie akceptuje wiadomości będąc offline.

Dopiero mając definicję modelu możemy przystąpić do stworzenia w bazie platformy konkretnej jego instancji. Przykładowo:

Lampa w pokoju:

- NAZWA: „Lampa w pokoju”;
- STAN ŻAROWKI: domyślnie wyłączona (wartość false).

Oprócz tego należy również zdefiniować:

- do kogo należy urządzenie (numer identyfikacyjny użytkownika),
- jaki jest klucz urządzenia do połączenia z IoT Hub’em.

Określony powyżej zbiór informacji jest przydatny na różnych poziomach działania platformy, zarówno w części „wewnętrznej”, niewidocznej dla użytkownika, jak i w części udostępnionej publicznie – w interfejsie aplikacji klienckiej.

3.2 Dziedziczenie właściwości

Przedstawiony proces tworzenia modeli jest wystarczający by utworzyć wiele typów urządzeń wewnątrz platformy. W niektórych przypadkach może się zdarzyć, że nowozdefiniowany model jest bardzo podobny do encji, która została już wcześniej utworzona. W takim przypadku możliwe jest dziedziczenie właściwości istniejącego modelu, aby wykorzystać istniejącą konfigurację i ją rozbudować. Przykładem może być tutaj urządzenie typu pilot. Przyjmijmy, że posiadamy zdefiniowany w bazie model PILOT10, który posiada 10 przycisków. Każdy przycisk reprezentowany jest przez właściwość o nazwie „Stan przycisku N”, gdzie N oznacza numer przycisku. W pewnym momencie jednak chcielibyśmy dodać do platformy inny rodzaj pilota – taki, który posiada 15 przycisków. Moglibyśmy od nowa definiować każdy z 15 przycisków nowego pilota. Lepszym rozwiązaniem będzie jednak wykorzystanie istniejących 10 właściwości, które stworzyliśmy na potrzeby modelu PILOT10 i dodać brakujące 5 – w taki sposób powstanie nowy model – PILOT15.

Dziedziczenie modeli nie jest mechanizmem niezbędnym do działania platformy. Mimo to jest to ciekawa funkcjonalność, która może się przydać przy definiowaniu wielu typów urządzeń o podobnych właściwościach.

Zdefiniowana w poprzednim podrozdziale LAMPA jest bardzo prostym modelem. Również w jej przypadku jednak możemy zastosować dziedziczenie – dotyczy to właściwości NAZWA.

Bez względu na to czym jest dane urządzenie, powinno ono posiadać jakąś nazwę, dzięki której mogłoby być szybko kojarzone przez użytkownika podczas zarządzania nim z poziomu aplikacji klienckiej. Jest to więc dobra okazja do tego, by utworzyć model bazowy – nazwiemy go MODEL_BAZOWY (wewnątrz platformy MJIoT został on nazwany BaseModel). Jedyną właściwością, którą posiada jest NAZWA, a jej charakterystyka jest taka sama jak przedstawiona wcześniej podczas omawianiu przykładu modelu lampy.

MODEL_BAZOWY różni się od modeli LAMPA czy PILOT10 pewną istotną cechą – nigdy nie stworzymy jego instancji, gdyż jest to typ stworzony jedynie na potrzeby dziedziczenia. W związku z tym w bazie danych zostanie on oznaczony jako model abstrakcyjny.

Opisany mechanizm umożliwia przejmowanie właściwości po tylko jednym typie. Nie ma jednak przeciwskazań, a nawet wydaje się to dobrym pomysłem, aby umożliwić dziedziczenie z wielu modeli. W ten sposób jeszcze bardziej przyspieszyć można by tworzenie nowych typów urządzeń.

Przedstawiony sposób definicji modeli oraz instancji został zainspirowany podejściem stosowanym w programowaniu obiektowym, gdzie definiujemy klasy oraz obiekty (instancje klas). Pokazać to można, wracając na chwilę do przykładu modelu LAMPA i definiując go z wykorzystaniem składni obiektowego języka programowania, np. C#:

```
//Nie można utworzyć instancji klasy ModelBazowy
public abstract class ModelBazowy
{
    public string Nazwa { get; set; }
}

//Lampa dziedziczy z ModelBazowy
public class Lampa : ModelBazowy
{
    public bool StanZarowki { get; set; }
}
…
//Konkretna instancja klasy Lampa
var lampkaWPokoju = new Lampa {
    Nazwa = "Lampa w pokoju",
    StanZarowki = false
};
```

Jak zostanie pokazane w rozdziale omawiającym bazę danych urządzeń, obiektowe podejście zostało zastosowane w praktyce, za sprawą zastosowania systemu ORM do budowy struktury bazy i wydobywania z niej informacji.

Wspomniane zostało, że proces modelowania mógłby zostać rozbudowany o możliwość wielokrotnego dziedziczenia. Inną funkcjonalnością, która wydaje się być jeszcze bardziej istotna to możliwość definiowania metod dla urządzeń. Bardziej naturalnym byłoby np. wydać polecenie „włącz żarówkę" niż ustawienie właściwości STAN_ŻARÓWKI na wartość TRUE. Niestety jednak nie przewidziałem takiej możliwości podczas planowania projektu. Wskazuje to jedynie na fakt, że etap definiowania założeń jakiegokolwiek projektu jest bardzo istotny i należy mu poświęcić wiele uwagi. W pewnym momencie jest już zbyt późno by wprowadzać do rozwiązania nieplanowane wcześniej funkcjonalności bez istotnego zaburzenia zaplanowanego terminu jego ukończenia.

3.3 Łączenie urządzeń

Zagadnienie, któremu niewątpliwie należy poświęcić uwagę jest sposób łączenia urządzeń ze sobą. W podrozdziale „Definiowanie urządzeń" wspomniane zostały dwa parametry, które posiada każda właściwość modelu urządzenia:

- Zdolność sterowania,
- Zdolność bycia sterowanym.

Są to kluczowe parametry do wyjaśnienia tematu tego podrozdziału. Zastosowałem koncepcję według której łączone są ze sobą nie same urządzenia, lecz ich właściwości. Dla przykładu, przyjmijmy, że posiadamy urządzenia: przycisk oraz lampka, o następującym zestawie właściwości:

PRZYCISK:

- NAZWA (łańcuch tekstowy)
- STAN PRZYCISKU (wartość logiczna, posiada zdolność sterowania)

LAMPKA:

- NAZWA (łańcuch tekstowy)
- STAN ŻARÓWKI (wartość logiczna, posiada zdolność bycia sterowanym)
- KOLOR ŚWIECENIA (łańcuch tekstowy, posiada zdolność bycia sterowanym)

Chcąc je ze sobą połączyć, nie możemy po prostu zdefiniować relacji w sposób następujący:

Połącz urządzenie PRZYCISK z urządzeniem LAMPKA

Taka definicja nie daje wystarczających informacji o tym, czego tak naprawdę oczekujemy jako użytkownicy platformy. PRZYCISK posiada tylko jedną właściwość sterującą, lecz LAMPKA wyposażona jest w dwie właściwości sterowalne. System musi wiedzieć dokładnie, która z nich ma zostać wybrana jako sterowana.

Zamiast tego połączenie powinno być więc zdefiniowane np. w poniższy sposób:

> *STAN PRZYCIKU urządzenia PRZYCIK ma wpływać na STAN ŻARÓWKI urządzenia LAMPKA.*

Taka definicja opisuje już znacznie lepiej, co chcemy osiągnąć. Określone zostały konkretne właściwości obu urządzeń, które mają zostać połączone. Po dodaniu takiej definicji połączenia do bazy danych, otrzymamy następujące zachowanie lampki:

- przy wciśnięciu przycisku – lampka zapali się;
- przy wyłączeniu przycisku – lampka zostanie zgaszona.

STAN ŻARÓWKI odwzorowuje więc STAN PRZYCISKU. W wielu przypadkach taki charakter relacji PRZYCISK-LAMPKA jest oczekiwany i wystarczający. Co jednak w sytuacji, kiedy chcielibyśmy np. odwrócić reakcję lampki, tzn.:

- przy wciśnięciu przycisku – lampka ma zostać zgaszona;
- przy wyłączeniu przycisku – lampka ma się zapalić.

Rozwiązanie takiego problemu (oraz innych) zostało zrealizowane za pomocą **kalkulacji**.

3.3.1 Kalkulacje

Sytuacja przedstawiona pod koniec ostatniego podrozdziału jest jednym z wielu możliwych przypadków, kiedy użytkownik chciałby wpłynąć na działanie zdefiniowanego połączenia. Inne przykładowe możliwości to:

- zapalanie lampki za każdym razem, niezależnie od stanu samego przycisku
- gaszenie lampki za każdym razem, niezależnie od stanu samego przycisku.

Przedstawione scenariusze wymagają wprowadzenia dodatkowego mechanizmu, który byłby w stanie przekształcić wysłaną wiadomość (po jej wysłaniu), zanim dotrze ona do drugiego urządzenia. Platforma posiada taki mechanizm – są to *kalkulacje*.

Idea kalkulacji polega na tym, aby dać możliwość modyfikowania wysyłanej wartości z urządzenia sterującego. W związku z tym, że platforma obsługuje trzy formaty właściwości - wartość logiczna, liczba, łańcuch tekstowy – należało przygotować trzy zestawy możliwych kalkulacji. Wynika to z faktu, że nie każdy rodzaj działania pasuje do dowolnego z wymienionych typów wartości. Oto zbiory zaimplementowanych kalkulacji:

- dla właściwości typu LOGICZNEGO:

NOT	AND	OR

- dla właściwości typu LICZBOWEGO:

Dodawanie	Odejmowanie	Mnożenie	Dzielenie

Oprócz wymienionych opcji, użytkownik ma również możliwość definicji połączenia bez kalkulacji. Właściwość typu ŁAŃCUCH TEKSTOWY nie posiada w systemie żadnych działań, gdyż uznałem, że nie byłoby to szczególnie przydatne. Jako przykład możliwego działania, można jednak podać operację odwracania łańcucha tekstowego bądź np. skracanie go do określonej liczby znaków. Operacje te dosyć łatwo można by dodać do systemu.

Spośród podanych rodzajów działań można podzielić je na dwie kategorie:

- wymagające dodatkowego parametru (np. dodawanie bądź AND),
- niewymagające dodatkowego parametru (np. NOT).

Podczas definicji połączenia, użytkownik ma możliwość wygodnego definiowania parametrów, w razie potrzeby. Dokładniejsza prezentacja procesu dodawania połączenia zostanie zaprezentowana w rozdziale omawiającym aplikację kliencką do obsługi platformy.

3.3.2 Filtry

Przedstawiony mechanizm kalkulacji to nie jedyna konfiguracja, jaka możliwa jest przy definiowaniu połączeń. Inną przydatną opcją jest *filtrowanie*. Ponownie najlepiej posłużyć się przykładem, aby wyjaśnić na czym polega ta funkcjonalność. Załóżmy, że posiadamy dwa urządzenia: czujnik temperatury oraz głośnik. Zadaniem systemu składającego się z tych dwóch „rzeczy” jest alarmowanie (za pomocą dźwięku) w razie wystąpienia zbyt wysokiej temperatury - np. ponad 40^{o}C. Chcemy więc, aby głośnik (będący urządzeniem sterowanym) otrzymał z systemu wiadomość tylko wtedy, kiedy czujnik temperaturowy (urządzenie sterujące) zaraportuje liczbę wyższą niż 40. Taki scenariusz nie byłby możliwy do realizacji

bez możliwości definiowania filtrów. W tym konkretnym przypadku filtr przepuszczałby jedynie wartości większe od 40.

Podobnie jak w przypadku omówionych już kalkulacji, lista dostępnych filtrów zależy od typu właściwości urządzenia sterującego. Są to:

- dla właściwości typu LOGICZNEGO:

Równy (=)	Nierówny (≠)

- dla właściwości typu LICZBOWEGO:

Równy (=)	Nierówny (≠)	Większy (>)
Większy bądź równy (>=)	Mniejszy (<)	Mniejszy bądź równy (<=)

- dla właściwości typu ŁAŃCUCH TEKSTOWY:

Równy (=)	Nierówny (≠)

W przypadku filtrów, każdy z nich do działania wymaga podania dodatkowego parametru.

3.3.3 Łączenie właściwości różnych typów

Podczas prac nad platformą, jednym z założeń było, aby rezultat był jak najbardziej generyczny, tzn. nie związany ściśle z żadnym konkretnym rodzajem czy zestawem urządzeń ani nie ograniczający użytkownika podczas konfiguracji. Rezultatem tego jest możliwość łączenia bardzo różnych właściwości, co niekoniecznie zawsze może mieć sens. Wróćmy na chwilę do przykładowej pary urządzeń przedstawionej w podrozdziale „Łączenie urządzeń": są to PRZYCISK oraz LAMPKA. Drugie z nich posiada dwie sterowalne właściwości:

- STAN ŻARÓWKI (wartość logiczna),
- KOLOR ŚWIECENIA (łańcuch tekstowy).

STAN PRZYCISKU, będąc właściwością sterującą, może zostać połączony z dowolną z powyższych opcji. O ile połączenie STAN PRZYCISKU – STAN ŻARÓWKI ma sens i jest logiczne, to druga możliwość (STAN PRZYCISKU – KOLOR ŚWIECENIA) z dużym prawdopodobieństwem sensu nie ma. Mimo wszystko, platforma dopuszcza takiego rodzaju połączenia. Wymagało to jednak wprowadzenia pewnych procedur, dzięki którym nawet

przedstawiony przed chwilą, na pierwszy rzut oka bezsensowny, scenariusz staje się realizowalny i poprawny. Procedury te zostaną omówione w następnych dwóch podrozdziałach.

3.3.3.1 Konwersja typów właściwości

Pierwszą istotną kwestią jest umożliwienie komunikacji między właściwościami, które zamodelowane zostały jako posiadające różne typy (wartość logiczna, liczba, łańcuch tekstowy). W tym celu logika platformy posiada konwerter, który potrafi zamieniać typy otrzymanych wartości w taki sposób, aby na wyjściu otrzymać typ oczekiwany przez urządzenie sterowane. Należy do razu zaznaczyć, że taka konwersja nie zawsze jest możliwa. Generalnie jednak można przedstawić zestaw zasad jakie obowiązują podczas konwersji:

Typ wejściowy	**Typ wyjściowy**	**Rezultat**
Liczba	Łańcuch tekstowy	Liczba jako łańcuch tekstowy
Liczba	Wartość logiczna	Jeśli liczba > 0: true Jeśli liczba <= 0: false
Łańcuch tekstowy	Liczba	Jeśli łańcuch tekstowy stanowi liczbę, będzie ona rezultatem. W innym przypadku konwersja nie powiedzie się.
Łańcuch tekstowy	Wartość logiczna	Jeśli łańcuch == ‘true’: true Jeśli łańcuch == ‘false’: false W innym przypadku konwersja nie powiedzie się
Wartość logiczna	Liczba	Jeśli wejście == true: 1 Jeśli wejście == false: 0
Wartość logiczna	Łańcuch tekstowy	Wejście jest serializowane do postaci łańcucha tekstowego

Tabela 3.1 Zestaw zasad przyjęty podczas implementacji konwertera typów właściwości

Jak widać w powyższej tabeli, problem może wystąpić w sytuacji, kiedy na wejściu konwertera znajduje się łańcuch tekstowy. W takim przypadku nie zawsze istnieje możliwość konwersji na pożądany typ wyjściowy. W celu ograniczenia niepotrzebnej komunikacji (każda wiadomość to pewien koszt w usłudze chmurowej, a odbiorca i tak nie zrozumie komunikatu) zwracany jest błąd. Tabela nie uwzględnia sytuacji, kiedy wartości wejściowa i wyjściowa mają ten sam typ – w takiej sytuacji wejście jest bezpośrednio przekazywane na wyjście konwertera.

3.3.3.2 Interpretacja wiadomości przez odbiorcę

Kolejną procedurą mającą na celu zapewnienie funkcjonowania systemu przy nietypowych połączeniach jest odpowiednia implementacja logiki urządzenia w momencie otrzymania wiadomości. Jak wynika z tabeli przedstawionej w poprzednim podrozdziale, problematyczny scenariusz to taki, gdzie odbiorca oczekuje łańcucha tekstowego. Problem wynika z faktu, że może on otrzymać w praktyce dowolny zestaw znaków. Ponownie możemy przywołać przykład PRZYCISKU ze STANEM PRZYCISKU i LAMPKI z KOLOREM ŚWIECENIA. Jeśli połączymy te właściwości ze sobą, LAMPKA otrzymywać będzie komunikaty: TRUE bądź FALSE, skierowane do właściwości KOLOR ŚWIECENIA!

W tym momencie, warto zatrzymać się na chwilę przy właściwości typu ŁAŃCUCH TEKSTOWY. Możemy tak naprawdę wydzielić dwa rodzaje oczekiwań wobec wiadomości, która zawiera ten typ wartości:

- zdolność obsługi dowolnego łańcucha tekstowego (np. ekran, który wyświetla każdą otrzymaną wiadomość),
- zdolność obsługi pewnego zbioru łańcuchów tekstowych (np. lampka, która może się świecić w pewnych kolorach).

W pierwszym przypadku nie ma żadnego problemu - każda wiadomość zostanie obsłużona poprawnie. W drugim przypadku natomiast mamy do czynienia z typem wyliczeniowym (w programowaniu często określanym jako *enum*), gdzie akceptowany jest jedynie pewien zbiór możliwości (np.: {zielony, czerwony, niebieski}). Dla tego rodzaju właściwości należy zastosować następujące podejście:

- Jeżeli otrzymana wartość stanowi liczbę, ustaw właściwość wyliczeniową na pozycji wynikającej z wyniku działania: $(x \ mod \ z)$, gdzie: x oznacza liczbę otrzymaną przez urządzenie, a z to ilość dostępnych opcji typu wyliczeniowego. Przykładowo, jeśli

przyjmiemy, że dostępne wyliczenie to: {zielony, czerwony, niebieski}, a otrzymana została wartość 7, to rezultatem powinno być zapalenie lampki na kolor zielony ($7\ mod\ 3 = 1$).

- Urządzenie powinno zawierać obsługę wartości logicznych. Programista powinien więc określić np., że wartość wejściowa TRUE da rezultat w postaci świecenia kolorem zielonym, a wartość FALSE spowoduje zapalenie światła czerwonego.

Przedstawione sposoby radzenia sobie z typem wyliczeniowym to jedynie wskazówki, które twórca urządzenia powinien rozważyć. Nie ma bowiem rozwiązania, które sprawdziłoby się w każdej sytuacji. Co na przykład uczynić w sytuacji, gdy LAMPKA otrzyma rozkaz zapalenia żarówki w kolorze „xyz”? Decyzja należy do programisty.

3.3.4 Schemat blokowy przetwarzania wartości

Omówione zostały mechanizmy, które zaimplementowane zostały w procesie obsługi komunikatów przesyłanych z urządzeń sterujących do urządzeń sterowanych. Najlepiej podsumować te wiadomości, prezentując schemat blokowy, który pokazuje jaka jest kolejność działania omówionych funkcji:

Ilustracja 3.1 Schemat blokowy przetwarzania wartości podczas obsługi połączenia między urządzeniami

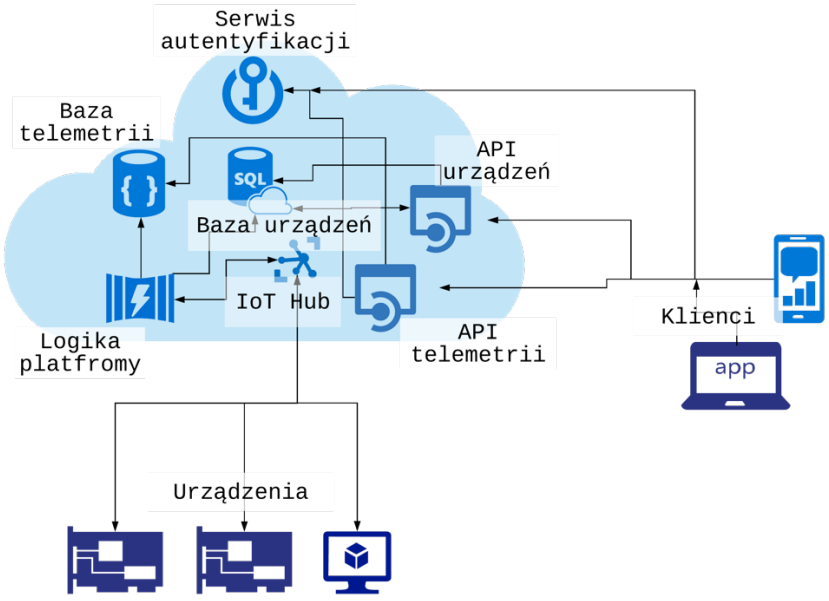
Najpierw następuje filtracja, aby wykluczyć przetwarzanie wiadomości, które nie powinny w ogóle zostać przekazane dalej. Kolejnym krokiem jest wykonanie zadanej kalkulacji (jeśli została zadana). Ostatnim etapem jest dokonanie konwersji, czego rezultatem jest uzyskanie wartości wyjściowej.

Każdy z bloków może zgłosić błąd w swoim działaniu – w takim przypadku wiadomość nie zostanie wysłana, a logika platformy odnotuje taki incydent, aby można było określić źródło problemu.

4 Implementacja platformy w chmurze

Wiedząc czym jest Internet Rzeczy oraz usługi chmurowe i jakie możliwości oferują, możemy przejść do opisu realizacji projektu, któremu poświęcona jest niniejsza praca. W tym rozdziale zaprezentowana zostanie lista komponentów z jakich składa się system, wraz ze schematem, który prezentuje architekturę całej platformy.

4.1 Schemat platformy



Ilustracja 4.1 Schemat platformy MJIoT

Schemat na ilustracji 3.1 jest uproszczonym, ogólnym spojrzeniem na platformę. Strzałki odchodzące od poszczególnych elementów wskazują komponenty, z których dany element korzysta w celu wykonania swojej funkcji. Wspomniane uproszczenie schematu wynika z faktu, że każdy ze zilustrowanych komponentów można podzielić na mniejsze części, które zostaną omówione w odpowiednich podrozdziałach.

Rysunek możemy go podzielić na trzy części:

- komponenty chmurowe, które stanowią wewnętrzną część platformy i pozostają „ukryte” przez użytkownikiem;
- komponenty, które znajdują się w chmurze, lecz stanowią punkt dostępowy dla użytkowników systemu (są dla nich widoczne);
- urządzenia, które dany użytkownik posiada.

4.2 Komponenty umieszczone w chmurze

Pierwsza z wymienionych kategorii zawiera elementy, które tworzą sam fundament platformy, dzięki którym możliwe jest jej funkcjonowanie. Oto ich lista:

- IoT Hub – przekazywanie wiadomości między chmurą a urządzeniami;
- Baza urządzeń – baza danych, która zawiera informacje na temat:
    - istniejących modeli urządzeń,
    - istniejących instancji urządzeń,
    - połączeń między urządzeniami,
    - użytkowników zarejestrowanych w systemie.
- Baza telemetrii – baza danych, która gromadzi wszelkie informacji przesyłane z urządzeń do chmury w celu ich odczytu bądź wizualizacji;
- Logika systemu – programy, które wykonują odpowiednie akcje przy otrzymaniu komunikatu z urządzenia:
    - wysyłanie sygnałów sterujących do połączonych urządzeń,
    - zapis danych telemetrycznych do bazy danych.

Z punktu widzenia użytkownika, nie potrzebuje on żadnej informacji na temat tego jakie są to komponenty. Co więcej, mogłyby one zostać przebudowane bądź kompletnie wymienione na inny zestaw serwisów, czego użytkownik nie powinien w ogóle zauważyć. Ponownie odwołam się do zasad znanych z programowania obiektowego – tworząc klasę, udostępniamy na zewnątrz jedynie część publiczną jej funkcjonalności. Część prywatna pozostaje ukryta, chociaż publiczny interfejs korzysta z niej w celu wykonania odpowiednich funkcji. Dokładnie na tej samej zasadzie zrealizowany został projekt MJIoT.

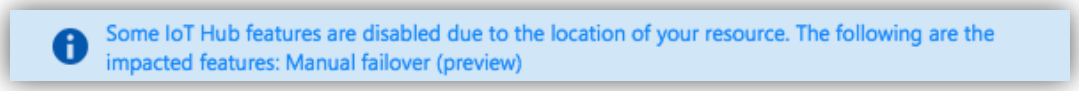
4.3 IoT Hub

Usługa IoT Hub została już przedstawiona w jednym z wcześniejszych rozdziałów. W tej części pracy omówione zostanie jednak jej praktyczne wykorzystanie w projekcie. Zacząć należy od samej czynności dodania usługi do swojego konta. W trakcie tego procesu określić należy kilka ustawień, które w tym przypadku są uniwersalne względem praktycznie wszystkich usług platformy Azure. Są to m. in.:

- region usługi,
- próg cenowy
- grupa zasobów

Przy ustawianiu pierwszej z wymienionych opcji należy zastanowić się, gdzie usługa będzie wykorzystywana, gdyż nieodpowiedni wybór regionu może spowolnić jej działanie. W związku z tym w przypadku MJIoT wybór padł na region *West Europe*. Generalnie jednak, wybór regionu może mieć w niektórych przypadkach większe znaczenie. Są ku temu dwa powody:

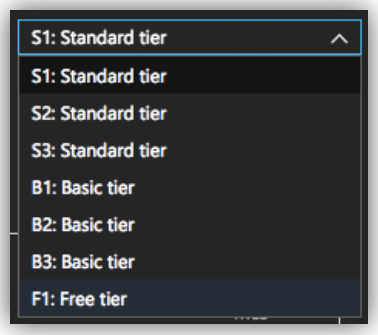
- w niektórych regionach niedostępne są niektóre funkcjonalności Azure. Przykładowo w przypadku regionu West Europe, usługa IoT Hub posiada dezaktywowaną funkcję „Manual failover” (zabezpieczenie przed katastrofami w regionach, nieistotne w niniejszym projekcie):



Ilustracja 4.1 Komunikat nt. braku dostępu do funkcjonalności "Manual failover" w regionie West Europe

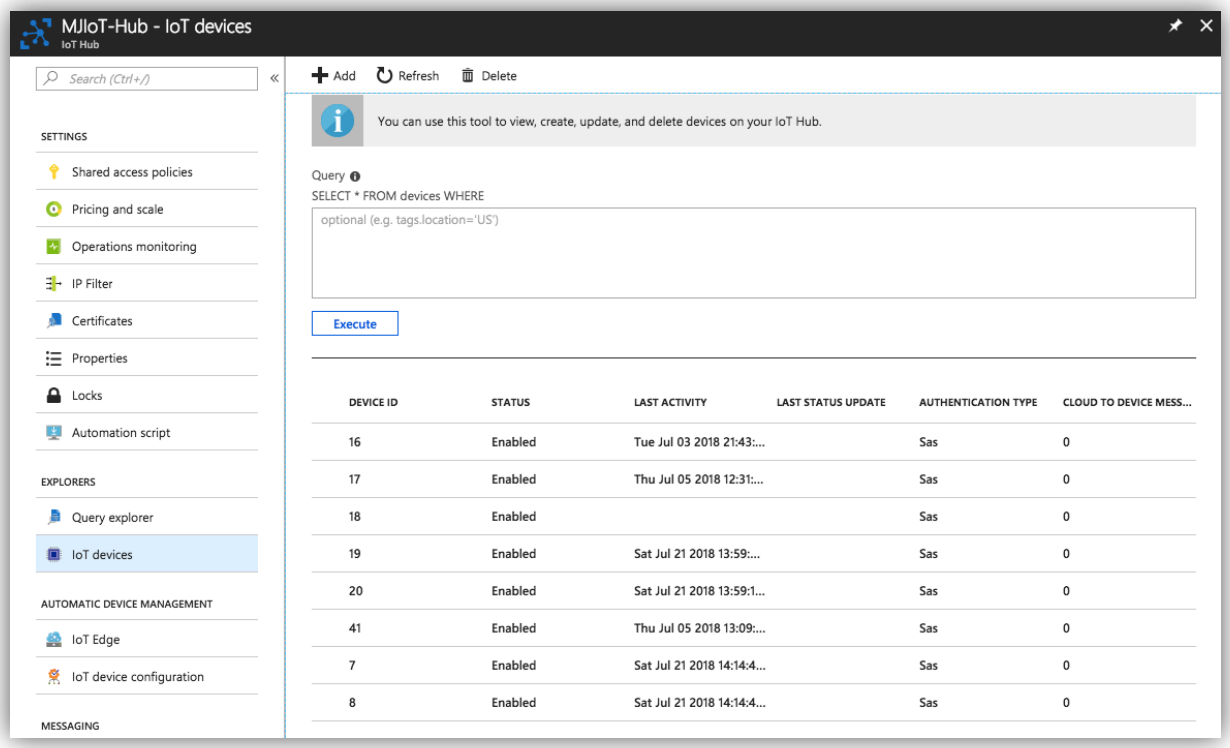
- względy geopolityczne – niektóre państwa (np. Niemcy oraz Chiny) legislacyjnie zabraniają przechowywania danych procesowych swoich przedsiębiorstw poza granicami kraju.

Kolejny z punktów – próg cenowy – określa ilość dostępnych zasobów oraz przewidywaną kwotę pieniężną, jaką właściciel subskrypcji będzie musiał za nią zapłacić. W przypadku serwisu IoT Hub dostępnych jest siedem progów cenowych, a różnią się one maksymalną ilością wiadomości jakie będą mogły być wysłane poprzez serwis w ciągu dnia. W tym przypadku dostępny jest próg F1, który jest darmowy i oferuje 8000 wiadomości w ciągu dnia. Jest to ilość niewielka, szczególnie w trakcie rozwoju projektu.



Ilustracja 4.2 Wybór progu cenowego usługi IoT Hub

Po stworzeniu instancji usługi na swoim koncie, mamy dostęp do panelu konfiguracyjnego:



Ilustracja 4.3 Lista zarejestrowanych urządzeń jako jedno z wielu dostępnych okien w panelu IoT Hub

Na powyższej ilustracji (4.3) widać okno z listą zarejestrowanych urządzeń. Dla każdego z nich można wyświetlić szczegółowe dane, m. in. Klucz dostępu do IoT Huba. Każde urządzenie musi znać swój klucz, jest to warunek konieczny połączenia się z usługą.

Istotnym punktem, jeśli chodzi o konfigurację, są tzw. grupy konsumentów. Mają one znaczenie w przypadku, kiedy wiadomości odbierane są przez więcej niż 1 odbiorcę. Taka sytuacja zachodzi ze względu na zastosowanie w projekcie architektury mikro-serwisów. Platforma zawiera 2 serwisy nasłuchujące:

- odpowiedzialny za obsługę połączeń między urządzeniami;
- odpowiedzialny za zapis telemetrii.

Okazuje się, że jedna grupa może tylko raz odebrać jedną wiadomość wysłaną przez IoT Hub. W przypadku, kiedy oba z wymienionych serwisów korzystały z tej samej grupy, pojedynczy komunikat był procesowany przez tylko jeden z tych serwisów. Przez pewien czas nie byłem w stanie zdiagnozować istoty problemu i podejrzewałem, że wada występuje w napisanym przeze mnie kodzie logiki platformy, podczas gdy w rzeczywistości należało stworzyć dodatkową grupę.

Ilustracja 4.4 Grupy konsumentów usługi IoT Hub zdefiniowane na potrzeby serwisów nasłuchujących

IoT Hub to jeden z najważniejszych elementów systemu. Jest tak, ponieważ to za jego pośrednictwem urządzenia mogą nawiązywać komunikację z chmurą. Każda „rzecz" będąca podłączona do platformy posiada zależności w postaci bibliotek programistycznych IoT Hub. W projekcie wykorzystane zostały biblioteki:

- *Microsoft.Azure.Devices.Client* (.NET) – pozwala m. in. wysyłać komunikaty do chmury z poziomu aplikacji napisanej w platformie .NET (np. z wykorzystaniem języka C#);
- *Microsoft.Azure.Devices* (.NET) – zastosowana wewnątrz logiki platformy w chmurze, pozwala m. in. przesyłać komunikaty do urządzeń;
- *Azure IoT Device SDK* (C) – pozwala podłączyć do IoT Hub'a urządzenia z kategorii mikrokontrolerów.

Dla środowiska .NET, na potrzeby niniejszego projektu, utworzona została dodatkowo biblioteka *MjIot.Devices.Common*, która udostępnia uproszczony zestaw API do obsługi urządzeń. Szczegóły omówione zostaną w rozdziale dotyczącym urządzeń.

4.4 Baza urządzeń Azure SQL

W jednym z poprzednich rozdziałów omówione zostało modelowanie urządzeń. Dane uzyskane w tym procesie należy gdzieś przechować – dla tego celu wykorzystana została kolejna usługa udostępniana w chmurze Azure – relacyjna baza danych Azure SQL. Jest to baza stworzona w oparciu o oprogramowanie Microsoft SQL Server. Obecnie jednak systemy te rozwijane są osobno i każdy z nich posiada pewien oddzielny zbiór funkcjonalności. W przypadku bazy Azure SQL chodzi tu przede wszystkim o zdolność do skalowania zasobów w ramach potrzeb – jest to cecha wspólna usług dostępnych w chmurze.

4.4.1 Skalowanie serwisów

W tradycyjnym modelu publikowania rozwiązania informatycznego, osoba odpowiedzialna za wdrożenia miała dwie opcje [7]:

- przydzielić zasoby w oparciu o wykorzystanie szczytowe produktu,
- przydzielić zasoby pokrywające zwyczajne/najczęstsze zapotrzebowania.

Oba rozwiązania mają swoje wady. Pierwsze wiąże się z nadmiernymi kosztami w trakcie, kiedy dodatkowe zasoby nie są tak naprawdę wykorzystane. Druga opcja natomiast wiąże się z ryzykiem utraty klientów, którzy nie będą w stanie skorzystać z naszego produktu z powodu przeciążenia serwerów. Przedstawiona sytuacja jest jednym z głównych powodów atrakcyjności rozwiązań chmurowych. Posiadają one mechanizmy, które w prosty sposób pozwalają skonfigurować mechanizmy skalowania. Można tu wymienić dwa rodzaje:

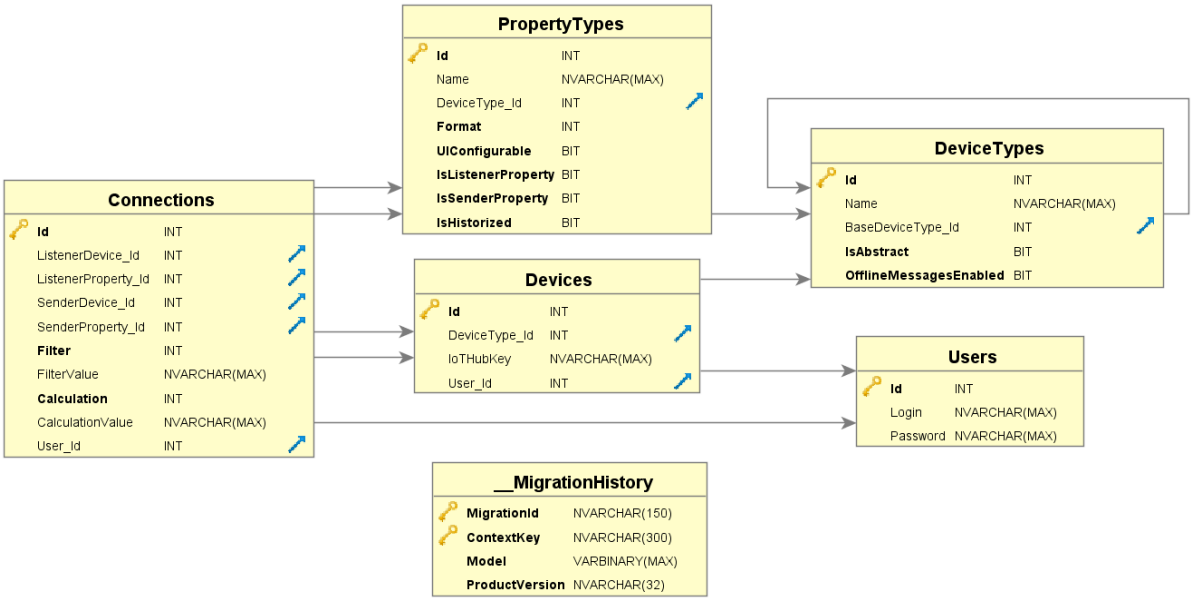
- skalowanie pionowe (ang. *scaling-up*) – polega na zwiększeniu zasobów maszyny obsługującej dany serwis, co najczęściej oznacza zastosowanie maszyny o większej ilości pamięci oraz dodatkowej mocy obliczeniowej;
- skalowanie poziomowe (ang. *scaling-out*) – polega na rozdzieleniu pracy serwisu na wiele serwerów.

Projekt MJIoT nie korzysta z mechanizmów skalowania ze względu na koszty takiego rozwiązania. Warto było jednak wspomnieć o takiej możliwości, ponieważ jest to ważna funkcjonalność udostępniana przez dostawców usług chmurowych. Projekty oparte o chmurę zdecydowanie mogą zyskać na zastosowaniu tego podejścia. Również niniejszy projekt prawdopodobnie korzystałby ze skalowania, gdyby był wykorzystywany na szerszą skalę.

4.4.2 Struktura bazy danych

Baza danych typu SQL składa się z tabel, które połączone są często pewnymi relacjami. Każda z tabel zawiera pewien zestaw kolumn o określonych typach danych. Zaplanowanie postaci takiej bazy jest dosyć istotne i warto poświęcić temu etapowi wystarczającą ilość czasu. Błędy w konstrukcji bazy mogą prowadzić do potrzeby zmian nie tylko samej bazy, ale często również rozwiązań, które mają na niej bazować. Innym problemem może być utrata lub problemy z migracją zapisanych wcześniej danych.

Dobrym sposobem przedstawienia takiej struktury jest diagram UML:



Ilustracja 4.5 Schemat bazy danych SQL

Baza zawiera 6 tabel powiązanych relacjami tak jak zostało to przedstawione powyżej. Jedynie „__MigrationHistory" nie zawiera żadnych połączeń z innymi tabelami – jej przeznaczenie zostanie wyjaśnione nieco dalej. Biorąc na razie pod uwagę pozostałe tabele, wyjaśnione zostanie ich przeznaczenie.

4.4.2.1 Modele urządzeń

W podrozdziale *3.1 Definiowanie urządzeń* opisany został sposób tworzenia typów urządzeń wewnątrz platformy MJIoT. Dane wygenerowane w procesie tworzenia nowego typu są przechowywane w, omawianej w niniejszym rozdziale, bazie danych. Służą do tego dwie tabele: *DeviceTypes* oraz *PropertyTypes*. Pierwsza z nich służy do przechowywania samych typów jako kolejnych encji. Spoglądając na powyższy schemat widać, że tabela ta zawiera 5 kolumn, które odpowiadają potrzebom definiowanych modeli. Mamy więc przede wszystkim kolumnę *Name*, która określa nazwę modelu. Kolejna – *BaseDeviceTypeId* – to referencja do modelu bazowego, z którego właściwości dany typ urządzenia ma korzystać. Relacja ta jest zdefiniowana jako jeden-do-jednego i odwołuje się do tej samej tabeli. Kolumna *IsAbstract* informuje czy dany model jest abstrakcyjny. Przeznaczenie *OfflineMessagesEnabled* zostanie wyjaśnione w innym rozdziale.

Przykładowe encje wewnątrz tabeli DeviceTypes:

Id	Name	BaseDeviceType_Id	IsAbstract	OfflineMessagesEnabled
1	DeviceBase	(NULL)	1	0
7	Chatter	1	0	1
4	LED Diode	1	0	0
6	SimNumSenderNumListener	1	0	1
5	Switch	1	0	0
11	LED and Potentiometer	4	0	0

Ilustracja 4.6 Widok tabeli DeviceTypes

Każdy typ urządzenia może posiadać wiele właściwości. W związku z tym zdefiniowana została relacja jeden-do-wielu pomiędzy *DeviceTypes* a *PropertyTypes*. Każda właściwość ma szereg kolumn, które ją definiują. Oto znaczenie kolumn, które mogą być niejasne:

- Format – opisuje typ wartości danej właściwości (liczba, wartość logiczna bądź łańcuch tekstowy);
- UIConfigurable – w założeniach kolumna ta miała informować o tym, czy dana właściwość powinna być konfigurowalna z poziomu aplikacji klienckiej. Projekt nie posiada jednak obecnie takiej funkcjonalności;
- IsListenerProperty – określa czy dana właściwość może być sterowalna;
- IsSenderProeprty – określa czy dana właściwość może być sterownikiem;
- IsHistorized – określa czy dana właściwość powinna być zapisywana w czasie w formie telemetrii (w przeznaczonej do tego bazie danych).

Przykładowe encje wewnątrz tabeli PropertyTypes:

Id	Name	DeviceType_Id	Format	UIConfigurable	IsListenerProperty	IsSenderProperty	IsHistorized
1	Name	1	1	1	0	0	0
6	LED State	4	0	0	1	0	1
5	Switch State	5	0	0	0	1	1
8	SimSenderProp	6	2	0	0	1	1
9	SimListenerProp	6	2	0	1	0	1
10	Sent Message	7	1	0	0	1	1
11	Received Message	7	1	0	1	0	1
12	Message History	7	1	0	0	0	0
15	Potentiometer Value	11	2	0	0	1	1

Ilustracja 4.7 Widok tabeli PropertyTypes

4.4.2.2 Urządzenia

Tabela Devices służy do zapisu danych nt. poszczególnych urządzeń. Jak widać na schemacie, zawiera referencję do swojego typu (relacja jeden-do wielu, tzn. jeden typ urządzenia może być implementowany przez wiele urządzeń), klucz dostępu do IoT Hub'a oraz referencję do właściciela danego urządzenia (ponownie jeden-do wielu – jeden użytkownik może posiadać 0 lub więcej urządzeń).

W trakcie prototypowania projektu, tabela ta zawierała również kolumnę *IsDeviceOnline*. W trakcie prac okazało się jednak, że przetrzymywanie tego typu informacji w bazie danych nie ma sensu, gdyż stan połączenia urządzenia jest sprawdzany dopiero gdy użytkownik o to poprosi. Dane wewnątrz bazy byłyby więc często nieaktualne i tak naprawdę niepotrzebne (ponieważ informacja o stanie online jest pobierana z oddzielnego serwisu). Więcej informacji na temat sprawdzania połączenia urządzeń z platformą zostanie podane podczas omawianiu interfejsu dostępowego API.

4.4.2.3 Połączenia

Tabela *Connections* zawiera najwięcej danych i definiuje ona połączenia między urządzeniami. Spoglądając na poszczególne kolumny można zauważyć, że odwzorowują one kolejne cechy połączenia opisane w rozdziale „Modelowanie oraz łączenie urządzeń". Jak zostało wspomniane w tymże rozdziale, łączone są ze sobą tak naprawdę właściwości urządzeń, a nie same urządzenia. W związku z tym tabela zawiera referencje zarówno do urządzeń jak i odpowiednich właściwości. W związku z tym, że każde połączenie należy do konkretnego użytkownika, tabela zawiera również odpowiednią referencję.

4.4.2.4 Użytkownicy

Tabela o najmniejszej ilości kolumn to *Users*. Jest to prosty zestaw danych na temat użytkowników platformy. Nie zawiera żadnych referencji, lecz sama jest często w relacji z innymi tabelami. Zawarte informacje to loginy i hasła, gdzie druga z kolumn zawiera tzw. *hash*, tzn. hasła w zaszyfrowanej postaci. Jest to zabezpieczenie na wypadek, gdyby tabela użytkowników została ujawniona publicznie. Wykorzystana metoda szyfrowania (SHA-256) sprawia, że danych oryginalnych nie można odzyskać. Więcej informacji na ten temat zostanie podanych podczas omawiania zabezpieczeń platformy.

4.4.3 Entity Framework

Przedstawiona baza danych została stworzona z pomocą Entity Framework (w skrócie często nazywane *EF*), czyli narzędzia typu ORM (ang. *Object-Relational Mapping*) dla środowiska .NET. Można wymienić kilka zalet zastosowania takiego podejścia:

- projektowanie oraz aktualizacje struktury bazy danych z wykorzystaniem składni języka obiektowego,
- wygodny dostęp do bazy danych z poziomu kodu źródłowego, z wykorzystaniem zdefiniowanych wcześniej klas oraz LINQ to Entities.

4.4.3.1 Zdefiniowane klasy

Przy wykorzystaniu narzędzia ORM, zmienia się sposób tworzenia struktury danych SQL. Zamiast pisać zapytania SQL, programista może napisać klasy, gdzie każda reprezentuje osobną tabelę. W przypadku projektu MJIoT, stworzone zostały następujące klasy oraz typy wyliczeniowe (składnia C#):

```
public class DeviceType
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DeviceType BaseDeviceType { get; set; }
    public bool IsAbstract { get; set; }
    public bool OfflineMessagesEnabled { get; set; }
}

public class PropertyType
{
    public int Id { get; set; }
    public DeviceType DeviceType { get; set; }
    public string Name { get; set; }
    public PropertyFormat Format { get; set; }
    public bool UIConfigurable { get; set; }
    public bool IsListenerProperty { get; set; }
    public bool IsSenderProperty { get; set; }
}

public class User
{
    public int Id { get; set; }
    public string Login { get; set; }
    public string Password { get; set; }
}
```

```
public class Device
{
    public int Id { get; set; }
    public string IoTHubKey { get; set; }
    public DeviceType DeviceType { get; set; }
    public User User { get; set; }
}
public class Connection
{
    public int Id { get; set; }
    public Device SenderDevice { get; set; }
    public Device ListenerDevice { get; set; }
    public PropertyType SenderProperty { get; set; }
    public PropertyType ListenerProperty { get; set; }
    public ConnectionFilter Filter { get; set; }
    public string FilterValue { get; set; }
    public ConnectionCalculation Calculation { get; set; }
    public string CalculationValue { get; set; }
    public User User { get; set; }
}
public enum PropertyFormat : int
{
    Boolean,
    String,
    Number
}
public enum ConnectionFilter
{
    None,
    Equal,
    Greater,
    GreaterOrEqual,
    Less,
    LessOrEqual,
    NotEqual
}
public enum ConnectionCalculation
{
    None,
    Addition,
    Subtraction,
    Product,
    Division,
    BooleanNot,
    BooleanAnd,
    BooleanOr
}
```

Kod źródłowy 4.1 Typy zdefiniowane w celu utworzenia bazy danych urządzeń

Można zauważyć, że przedstawione klasy odpowiadają tabelom, które zostały zaprezentowane na ilustracji 4.5. Jest tak, ponieważ tabele te zostały wygenerowane przez Entity Framework na podstawie powyższego kodu.

4.4.3.2 Migracje

Podczas omawiania struktury bazy danych, na schemacie z ilustracji 4.5 można było zauważyć tabelę „__MigrationHistory”. Jest to tabela tworzona przez Entity Framework automatycznie podczas inicjowania bazy. Same migracje, to kolejne aktualizacje bazy danych – nie chodzi tu o aktualizacje danych wewnątrz tabel, lecz o zmiany w samej strukturze bazy. Zmian takich dokonuje się w klasach przedstawionych w poprzednim podrozdziale. Generowana jest nowa migracja do aktualizacji bazy danych. Zanim baza zostanie jednak zmodyfikowana, porównywana jest lokalna historia oraz historia zapisana w bazie danych (wewnątrz __MigrationHistory). W razie konfliktów, EF nie przeprowadzi aktualizacji, gdyż skutki mogą być niekoniecznie takie jak życzyłby sobie użytkownik.

Oto kilka migracji zapisanych w bazie:

MigrationId	ContextKey	Model	ProductVersion
201710011432075_InitialModel	MJIoT_DBModel.Migrations.Configuration	0x3F08	6.1.3-40302
201710011504026_AddIoTHubKeyPropertyToDevicesTable	MJIoT_DBModel.Migrations.Configuration	0x3F08	6.1.3-40302
201710071741351_FixForConnectedDevicesToBeManyToMany	MJIoT_DBModel.Migrations.Configuration	0x3F08	6.1.3-40302
201710071758217_AddedUsersTable	MJIoT_DBModel.Migrations.Configuration	0x3F08	6.1.3-40302
201710071821369_FixPropertyTypesTableToIncludeTypeEnum	MJIoT_DBModel.Migrations.Configuration	0x3F08	6.1.3-40302
201710220954417_AdditionOfSenderAndListenerPropertiesToDeviceType	MJIoT_DBModel.Migrations.Configuration	0x3F08	6.1.3-40302

Ilustracja 4.8 Fragment tabeli __MigrationHistory

4.4.3.3 LINQ to Entities

Podczas opisu Entity Framework nie sposób pominąć LINQ to Entites. Sama technologia LINQ (Language INtegrated Query) to sposób operowania na kolekcjach w środowisku .NET, który przypomina do pewnego stopnia składnię SQL. Użyteczność LINQ najlepiej zaprezentować na przykładzie, w porównaniu z klasycznym sposobem dostępu do danych.

Załóżmy, że mamy zdefiniowaną kolekcję obiektów klasy Person (np. `List<Person>`), z której chcemy wydzielić osoby powyżej 18 roku życia. Oto klasa Person:

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Kod źródłowy 4.2 Przykładowa klasa Person

Klasyczny sposób, z wykorzystaniem instrukcji *foreach*:

```
var adults = new List<Person>();
foreach (var person in collection)
{
    if (person.Age >= 18)
        adults.Add(person);
}
```

Kod źródłowy 4.3 Operacja na kolekcji z wykorzystaniem foreach

Wykorzystanie LINQ:

```
var adults = collection.Where(person => person.Age >= 18).ToList();
```

Kod źródłowy 4.4 Operacja na kolekcji z wykorzystaniem LINQ

Przedstawiony przykład jest bardzo prosty i ma na celu jedynie zaprezentowanie składni LINQ. Technologia ta jednak umożliwia dokonywanie w prosty i czytelny sposób wielu operacji, które w klasycznej pętli wymagałyby napisania wielu linii kodu.

Jednym z wariantów technologii jest LINQ to Entites, który operuje na kolekcjach *DbSet<T>* zwracanych przez Entity Framework. Dzięki temu, w sposób podobny jak w powyższym przykładzie, możemy dokonywać zapytań na bazie danych. W przypadku niniejszego projektu wykorzystane zostają wtedy przedstawione już w tym rozdziale klasy, które w pierwszej kolejności użyte były do stworzenia struktury bazy. Jest to bardzo wygodne podejście, programista operuje na kolekcjach obiektów z wykorzystaniem przystosowanej do tego składni.

4.4.3.4 Abstrakcja dostępu do bazy danych

Każdy projekt, który wymaga dostępu do bazy SQL (i został stworzony w środowisku .NET) posiada zależność od projektu *MjIot.Storage.Models*, który z kolei korzysta z jednej z dwóch bibliotek: *MjIot.Storage.Models.EF6Db* lub *MjIot.Storage.Models.EFCoreDb*, które korzystają z Entity Framework. Istnienie dwóch wersji tej biblioteki wynika z faktu, że część projektów korzysta ze środowiska .NET Framework (dostarczanego jedynie dla platformy

Windows), a część z .NET Core (multiplatformowa implementacja .NET). Każde z tych środowisk wymaga stosowania osobnych wersji EF:

- .NET Framework wymaga Entity Framework 6;
- .NET Core wymaga Entity Framework Core.

Wspomniany projekt *MjIot.Storage.Models* został stworzony w oparciu o wzorzec projektowy Repozytorium oraz Unit of Work.

4.4.3.4.1 Repozytorium

Wzorzec repozytorium polega na stworzeniu osobnych klas dostępowych dla każdej tabeli bazy danych. Klasy te oparte są o interfejs IRepository:

```
public interface IRepository<T> where T : class
{
    T Get(int id);
    IEnumerable<T> GetAll();

    void Remove(int id);
    void Remove(T entity);
    void RemoveRange(IEnumerable<T> entities);
    void RemoveAll();

    void Add(T entity);
    void AddRange(IEnumerable<T> entities);

    T Find(Expression<Func<T, bool>> predicate);
}
```

Kod źródłowy 4.5 Generyczny interfejs IRepository

Interfejs ten definiuje kontrakt jaki realizować musi każde repozytorium. Wymagana jest więc możliwość odczytu, usuwania i dodawania danych. Interfejs przedstawiony powyżej jest generyczny, oznacza to, że nie jest ściśle związany z żadnym typem danych, co jest dobrą praktyką programistyczną. Dzięki temu może on być implementowany przez wszystkie repozytoria. Gdyby nie korzystać z generyczności interfejsów, należałoby zwracać za każdym razem typ *object*, co wymuszałoby na programiście stosowanie rzutowania za każdym razem podczas wydobywania danych z repozytorium.

Poniżej zaprezentowany jest interfejs repozytorium urządzeń, który implementuje *IRepository*:

```
public interface IDeviceRepository : IRepository<Device>
{
    List<Device> GetDevicesOfUser(int userId);
    string GetDeviceName(Device device);
    DeviceRole GetDeviceRole(Device device);
    DeviceType GetDeviceType(int deviceId);
}
```

Kod źródłowy 4.6 Interfejs IDeviceRepository

Przedstawiony został interfejs zamiast klasy, która go implementuje, ze względu na objętość kodu źródłowego. Widać, że *IDeviceRepository* podczas implementacji *IRepository* wskazuje, że operacje będą wykonywane na obiektach klasy *Device*, co jest rezultatem wspomnianej generyczności interfejsu. Dodatkowo interfejs ten dodaje kilka metod pomocniczych, które będą przydatne dla klientów tego repozytorium.

4.4.3.4.2 Unit of Work

Kolejnym wzorcem jest Unit of Work, który stanowi warstwę dostępu do wszystkich zdefiniowanych repozytoriów. Najlepiej przedstawić interfejs:

```
public interface IUnitOfWork : IDisposable
{
    IDeviceRepository Devices { get; }
    IDeviceTypeRepository DeviceTypes { get; }
    IPropertyTypeRepository PropertyTypes { get; }
    IUserRepository Users { get; }
    IConnectionRepository Connections { get; }

    int Save();
}
```

Kod źródłowy 4.7 Interfejs IUnitOfWork

Jak widać, Unit of Work stanowi „opakowanie” na repozytoria wraz z opcją zapisu modyfikacji jakie zostały dokonane na danych z ich wykorzystaniem.

Za każdym razem, kiedy któraś z aplikacji wchodzących w skład projektu MJIoT musi podłączyć się do bazy danych, wykorzystywana jest klasa UnitOfWork, która implementuje powyższy interfejs. Jest to znacznie wygodniejszy i czytelniejszy sposób pozyskiwania/zapisu danych niż korzystanie bezpośrednio z kontekstu Entity Framework.

Przykładowo, poniższe wywołanie:

```
var unitOfWork = new UnitOfWork();
var deviceType = unitOfWork.Devices.GetDeviceType(8);
```

Kod źródłowy 4.8 Wykorzystanie UnitOfWork

jest bardziej czytelne niż:

```
var context = new MJIoTDBContext();
var deviceType = context.Devices
                    .Include(n => n.DeviceType)
                    .Where(n => n.Id == 8)
                    .Select(n => n.DeviceType)
                    .FirstOrDefault();
```

Kod źródłowy 4.9 Bezpośrednie wykorzystanie Entity Framework

Pierwszy z przedstawionych fragmentów to przykład wykorzystania UnitOfWork. Drugi natomiast to bezpośrednia operacja na obiekcie DbContext z Entity Framework. Przedstawiony przykład ma na celu zdobycie obiektu DeviceType (który reprezentuje model urządzenia) dla urządzenia o identyfikatorze równym „8”.

Sama czytelność kodu nie jest jednak w tym przypadku najważniejsza. Istnieje inny powód, dla którego zastosowano taki sposób kontaktu z bazą danych. Powodem tym jest enkapsulacja oraz abstrakcja. Wewnątrz aplikacji wchodzących w skład projektu MJIoT wielokrotnie występuje potrzeba łączenia się z bazą urządzeń. Załóżmy, że każda z tych aplikacji bezpośrednio działałaby w oparciu o DbContext Entity Framework. Oznaczałoby to, że są one dosyć ściśle powiązane z implementacją bazy danych. Wynika z tego poważny problem: w przypadku jakiejkolwiek zmiany struktury bazy danych, wymagana jest zmiana kodu oraz rekompilacja w każdej z tych aplikacji. Tworząc warstwę pośrednią (w tym przypadku Unit of Work) zabezpieczamy się przed taką sytuacją. Zmiana bazy danych będzie wymagać zmian jedynie w tej warstwie. Aplikacje, które z niej korzystają nie będą wymagały rekompilacji, a jedynie podmiany biblioteki *MjIot.Storage.Models*.

W tym miejscu warto też krótko wspomnieć o testowaniu aplikacji. Dzięki temu, że dostęp do danych w bazie jest zawsze realizowany z wykorzystaniem interfejsu IUnitOfWork, a konkretna implementacja interfejsu jest wstrzykiwana do klas (podać jakieś źródło w postaci książki), które go wymagają, łatwo można stworzyć implementację testową. Dzięki temu w trakcie testów konkretnych klas, kontakt z bazą danych będzie symulowany. Więcej na temat testowania można dowiedzieć się w dalszej części pracy.

4.5 Baza telemetrii CosmosDB

Niniejszy rozdział poświęcony jest kolejnej bazie danych jaka znalazła się w platformie MJIoT – bazie telemetrii. Tym razem wykorzystany został inny system bazodanowy – usługa CosmosDB z oferty Azure. Jest to baza typu NoSQL – wybór tego typu rozwiązania został podyktowany moją ciekawością w stosunku do tego typu rozwiązania i chęcią jego poznania.

4.5.1 Radzenie sobie ze zmianą bazy danych

Podczas tworzenia prototypu projektu, wszystkie dane znajdowały się w bazie Azure SQL, która przedstawiona została nieco wyżej. Z czasem jednak podjąłem decyzję, że telemetria przechowywana będzie w bazie innego typu. Tego rodzaju zmiana prowadzi często do wielu problemów spowodowanych licznymi zmianami w kodzie źródłowym. Dlatego tak istotna jest, wspomniana już, abstrakcja oraz interfejsy, które pomagają to osiągnąć. Oto zastosowany interfejs dostępu do bazy telemetrii:

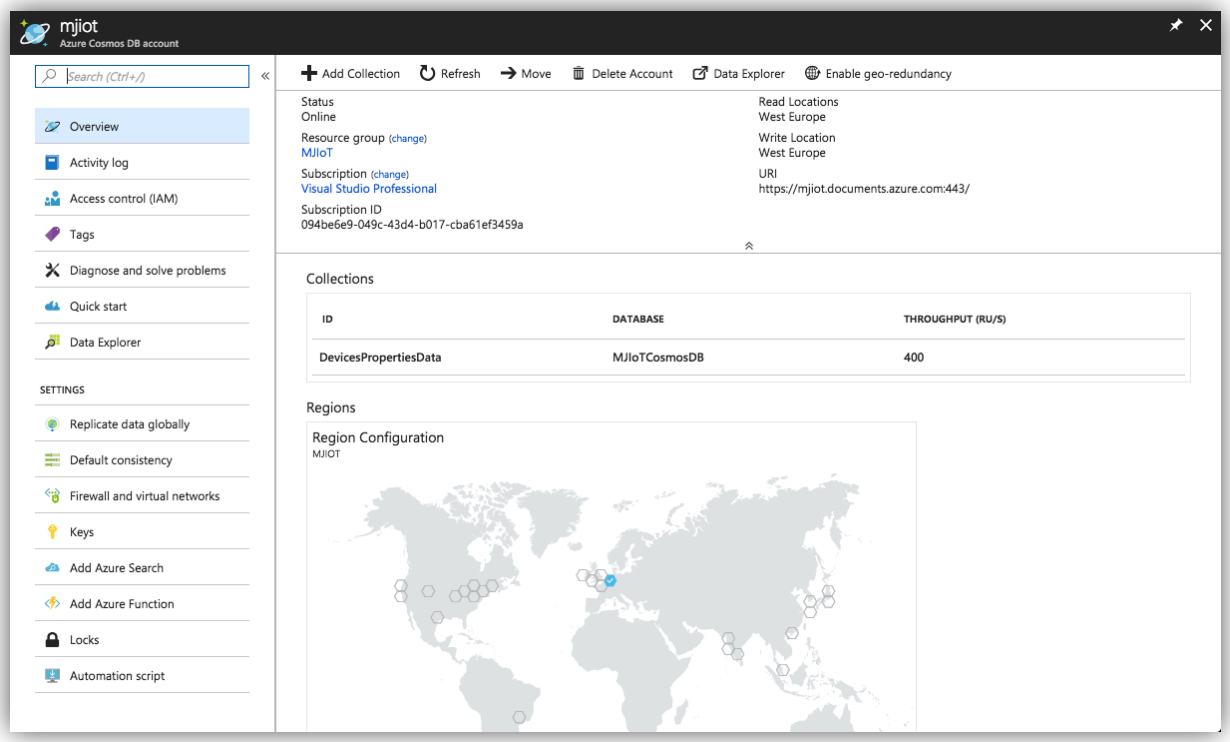
```
public interface IPropertyStorage
{
    Task<string> GetPropertyValueAsync(int deviceId, string propertyName);
    string GetPropertyValue(int deviceId, string propertyName);
    Task SetPropertyValueAsync(int deviceId, string propertyName,
                               string propertyValue);
}
```

Kod źródłowy 4.10 Interfejs IPropertyStorage

Powyższy kod nie zawiera żadnych powiązań z konkretnym Systemem Zarządzania Bazą Danych. Jest to jedynie kontrakt, który określa konkretne wymagania, jakich oczekujemy wobec klasy, której będziemy używać. W rzeczywistości dane mogą być trzymane nawet w pliku tekstowym – z punktu widzenia użytkownika interfejsu nie ma to żadnego znaczenia.

Wracając więc do migracji z SQL do NoSQL, dostosowanie projektu do nowej bazy daych sprowadzało się do:

1. stworzenia instancji CosmosDB w Azure:



Ilustracja 4.9 Widok panelu instancji CosmosDB w Azure Portal

2. stworzenia klasy implementującej interfejs *IPropertyStorage*:

```
public class CosmosPropertyStorage : IPropertyStorage
{
    ...
}
```

Kod źródłowy 4.11 Klasa CosmosPropertyStorage implementująca IPropertyStorage (bez szczegółów implementacyjnych)

3. wstrzyknięcia nowej klasy *CosmosPropertyStorage* zamiast poprzedniej, obsługującej SQL, w konstruktorach klas, które tego wymagają, np. wewnątrz projektu Web API (które zostanie omówiony w dalszej części pracy):

```
public DevicesController()
{
    _handler = new RequestHandler(new UnitOfWork(), new
    CosmosPropertyStorage());
}
```

Kod źródłowy 4.12 Wstrzykiwanie konkretnej instancji IPropertyStorage

4.5.2 Charakterystyka NoSQL

Tradycyjnie w projektach opartych o wykorzystanie bazy danych, najpopularniejsze Systemy Zarządzania Bazami Danych to te oparte o język SQL i przechowywanie danych w relacyjnych tabelach (np. MS SQL Server, MySQL, PostgreSQL, itd.). Obecnie coraz większą popularność zyskują bazy typu NoSQL, które, nawet nazwą, odcinają się od sposobu działania kojarzonego z SQL. W tym podrozdziale przedstawię porównanie dobrze znanych SZBZ SQL z NoSQL.

Przede wszystkim ważnym jest, aby stwierdzić, że nie ma żadnych przeciwskazań, aby rolę przechowywania telemetrii przejął system typu SQL. Tak samo w drugą stronę – nie ma powodu, dla którego baza urządzeń powinna być zaimplementowana z wykorzystaniem klasycznego SQL, a nie, omawianego w tym rozdziale, NoSQL. Technologie te różnią się między sobą, co zostanie wyjaśnione, jednak obie spełniają bardzo dobrze swoje główne zadanie – przechowywanie danych. Nie należy więc traktować NoSQL jako następcę lub lepszy wybór wobec SQL – są to raczej alternatywy. Pojawienie się NoSQL nie spowodowało zatrzymania rozwoju SQL, a wręcz przeciwnie – obecnie niektóre tradycyjne systemy adoptują niektóre rozwiązania NoSQL oferując nowe funkcjonalności. Podobnie implementacje NoSQL oferują czasami rozwiązania kojarzone dotychczas z relacyjnymi bazami danych.

4.5.2.1 Sposób przechowywania danych

SQL jest dobrze znany z operowania na tabelach, gdzie przechowywane są dane. NoSQL natomiast korzysta z dokumentów w formacie JSON. W tym przypadku każda encja stanowi osobny dokument. Porównajmy pojedynczy rekord SQL oraz NoSQL dla przykładowych danych:

- **SQL**:

Id (INT)	Imię (NVARCHAR)	Data urodzenia (DATETIME)	Subskrypcja (BIT)
4	Kosma	2006-08-29T00:00:00	1

- **NoSQL**:

```
{
    "Id": 4,
    "Imię": "Kosma",
    "Data urodzenia": "2006-08-29T00:00:00",
    "Subskrypcja": true
}
```

Pewnym odpowiednikiem tabel ze świata SQL, są w NoSQL kolekcje, które stanowią zbiór dokumentów JSON.

4.5.2.2 Format danych

Zanim zapiszemy jakiekolwiek dane do bazy SQL, musimy zdefiniować tabelę. Określa ona zbiór danych jaki musi się znaleźć w pojedynczym rekordzie wraz z ich typami. W przykładzie powyżej, przedstawiona tabela wymaga, aby każdy rekord posiadał cztery wartości określonych typów. Zupełnie inaczej zaprojektowany został NoSQL który nie nakłada żadnych wymagań na ilość i typ danych jakie przechowuje. Tworząc kolekcję, określamy po prostu jej nazwę oraz ewentualnie klucze unikalne (pomijam tu konfigurację związaną np. z dostępną przepływnością kolekcji, która wynika raczej z optymalizacji kosztów w chmurze niż samego wykorzystania NoSQL). Nie ma żadnych przeciwskazań, aby, oprócz przedstawionego w przykładzie dokumentu, w tej samej kolekcji znalazł się np. JSON:

```
{
    "Id": 7,
    "Imię": "Barbara",
    "Data urodzenia": "26.09.2016",
    "Kolor oczu": "#3d87ff"
}
```

Nowy dokument różni się brakiem klucza „Subskrypcja” oraz dodatkiem w postaci klucza „Kolor oczu”. Oprócz tego zmienił się format zapisu daty urodzenia.

W związku z tym, że kolekcja nie wymaga stałości typów danych pomiędzy dokumentami, w każdym kluczu może się znaleźć dowolny typ wartości (o ile jest wspierany przez konkretną bazę). Typy danych wspierane przez CosmosDB:

- String,
- Number,
- Boolean,
- Null,

- Tablica,
- Zagnieżdżony JSON.

4.5.2.3 Relacje

Cechą wspólną obu systemów jest możliwość określania relacji. Pewną różnicą jest jednak fakt, że wewnątrz tablicy SQL musimy wskazać istnienie takiej relacji „dosłownie" wiążąc tabele ze sobą. W systemie NoSQL wystarczy podać wewnątrz jednego dokumentu referencję do innego, podając np. jego id:

```
{
    "id": 68,
    "book": "The Catcher in the Rye",
    "author": 24
},
{
    "id": 24,
    "name": "Jerome David Salinger"
}
```

W powyższym przykładzie konkretna książka posiada referencję do jej autora poprzez jego unikalny numer identyfikacyjny. Taki sposób tworzenia relacji nie jest jednak w żaden sposób wspierany przez silnik bazy – dla systemu jest to po prostu kolejny klucz wewnątrz dokumentu.

4.5.2.4 Zapytania

Bazy korzystają oczywiście z języka SQL (ang. *Structured Query Language*), który umożliwia dokonywanie skomplikowanych zapytań na bazie danych wraz z możliwością łączenia danych z wielu tabel. Składnie różnią się nieco między systemami ze względu na dostępne funkcjonalności, np. produkty firmy Microsoft korzystają ze składni T-SQL.

Postać zapytań w bazach NoSQL zależy od twórców danego systemu bazodanowego. Przykładowo CosmosDB udostępnia kilka sposobów manipulacji danymi:

- SQL
- MongoDB
- Graph
- Tables
- Cassandra

Nadal możemy więc korzystać np. z języka SQL. Oprócz tego istnieje możliwość korzystania z interfejsów innych baz danych, np. wylistowanego MongoDB, które stanowi konkurencyjny system bazodanowy NoSQL.

Pewnym utrudnieniem w przypadku baz NoSQL jest brak możliwości dokonywania operacji JOIN na wielu kolekcjach. Należy w tym celu pobierać dane oddzielnie i łączyć je ręcznie. W przypadku SQL jest to normalna praktyka – pobieranie danych relacyjnych z wielu tabel.

Oprócz różnic opisanych w przedstawionych podrozdziałach, należy dodatkowo zwrócić uwagę na:

- Brak wsparcia dla transakcji w systemach NoSQL;
- Szybkość wykonywania zapytań na korzyść NoSQL;
- Łatwiejsze skalowanie bazy NoSQL ze względu na prostszy model przechowywania danych.

4.5.3 Struktura bazy

Mówiąc o NoSQL nie można tak naprawdę powiedzieć, że baza ma jakąś ustaloną strukturę, ze względu na charakterystykę tego typu bazy, opisaną we wcześniejszych podrozdziałach. Można jednak przedstawić postać dokumentów jakie są wykorzystywane w niniejszym projekcie.

Baza zawiera jedną kolekcję – *DevicesPropertiesData*. Wewnątrz niej znajdują się dokumenty, które zawierają następujący zestaw par klucz-wartość:

- DeviceId – numer identyfikacyjny urządzenia, z którego pochodzi dana wartość,
- PropertyName – nazwa właściwości, której wartość dotyczy,
- PropertyValue – wartość właściwości,
- Timestamp – czas powiązany z wartością. Może zostać określony przez urządzenie, które wysyła daną telemetrię lub przez serwis zapisujący telemetrię do bazy.

Warto odnotować, że wartość (*PropertyValue*) przechowywana jest zawsze jako łańcuch tekstowy, niezależnie od rzeczywistego formatu danej właściwości. Powodem tego jest fakt, że początkowo magazynem telemetrii była baza typu SQL, która (jak wspomniano w poprzednim podrozdziale) wymaga określenia konkretnych typów wartości dla każdej z kolumn. Łańcuch tekstowy to jedyna opcja, która pozwala bezstratnie przechować (oraz odzyskać) wartości typu:

liczba, wartość logiczna oraz oczywiście łańcuch tekstowy. Co prawda, NoSQL nie ma opisanego ograniczenia, jednak całość rozwiązania była już stworzona z założeniem, że wartości przesyłane są jako wartości tekstowe. Takie rozwiązanie działało poprawnie, postanowiłem go więc nie zmieniać.

Dokument dodany do CosmosDB w rzeczywistości zawiera pewne dodatkowe klucze, zdefiniowane w momencie dodawania dokumentu. Przykładowy dokument opisujący nazwę pewnego urządzenia wewnątrz platformy MJIoT:


```
{
    "DeviceId": 7,
    "PropertyName": "Name",
    "PropertyValue": "Switch 1",
    "Timestamp": "2018-07-21T11:53:16.5088751Z",
    "id": "790758d6-7809-4fd4-9554-ce2c6f1250de",
    "_rid": "1oUHAJhiGgABAAAAAAAAAA==",
    "_self": "dbs/1oUHAA==/colls/1oUHAJhiGgA=/docs/1oUHAJhiGgABAAAAAAAAAA==/",
    "_etag": "\"47007d6c-0000-0000-0000-5b531eac0000\"",
    "_attachments": "attachments/",
    "_ts": 1532173996
}
```


Powyższy dokument zawiera informacje, że urządzenie o identyfikatorze „7” posiada nazwę „Switch 1”. W związku z tym, że nazwa to parametr określany przy dodawaniu urządzenia do platformy (chociaż nie jest to wymagane), możemy stwierdzić, że urządzenie zostało dodane 21 lipca 2018 roku o godzinie 11:53.

Klucze zaczynające się od „_” oraz klucz „id” są generowane automatycznie dla każdego nowego dokumentu. Ich znaczenie jest następujące [8]:

- *_rid* – unikalny identyfikator zasobu w skali konkretnej instancji bazy danych,
- *_self* – unikalny adres URI dokumentu,
- *_etag* – klucz do zapewnienia tzw. *optimistic concurrency control*, tzn. zabezpieczenia przed jednoczesną modyfikacją dokumentu,
- *_attachments* – referencja do innych dokumentów powiązanych z danym dokumentem,
- *_ts* – czas ostatniej aktualizacji dokumentu,
- *id* – unikalny identyfikator dokumentu, który może zostać określony przy dodawaniu dokumentu do bazy. W razie jego braku, system sam przydziela id.

4.6 Serwis zapisu telemetrii

Przechodząc do części związanej z logiką platformy MJIoT, zacznę od omawiania serwisu obsługi telemetrii. Jego rola to zapis komunikatów otrzymywanych od urządzeń do przedstawionej w poprzednim podrozdziale bazy danych telemetrii. Kod serwisu napisany został w języku C# w środowisku .NET Core.

4.6.1 Azure Functions

Ilustracja 4.10 Logo Azure Functions

Zanim zostanie przedstawiony algorytm aplikacji, warto omówić sposób jej hostowania w chmurze. Do tego celu wykorzystana została kolejna z usług platformy Microsoftu – Azure Functions. Jest to typowy przykład rozwiązania SaaS, dodatkowo określany jako *architektura bezserwerowa* (ang. *Serverless architecture*) [9]. Jest to dosyć nowe pojęcie, które sprowadza się do uproszczenia procesu publikowania aplikacji. Programista zajmuje się w tym przypadku jedynie tworzeniem kodu źródłowego, a inne kwestie, jak np.: wybór serwera, instalacja środowiska uruchomieniowego, itp., zostają rozwiązane za niego. Proces tworzenia aplikacji sprowadza się więc (w uproszczeniu) do:

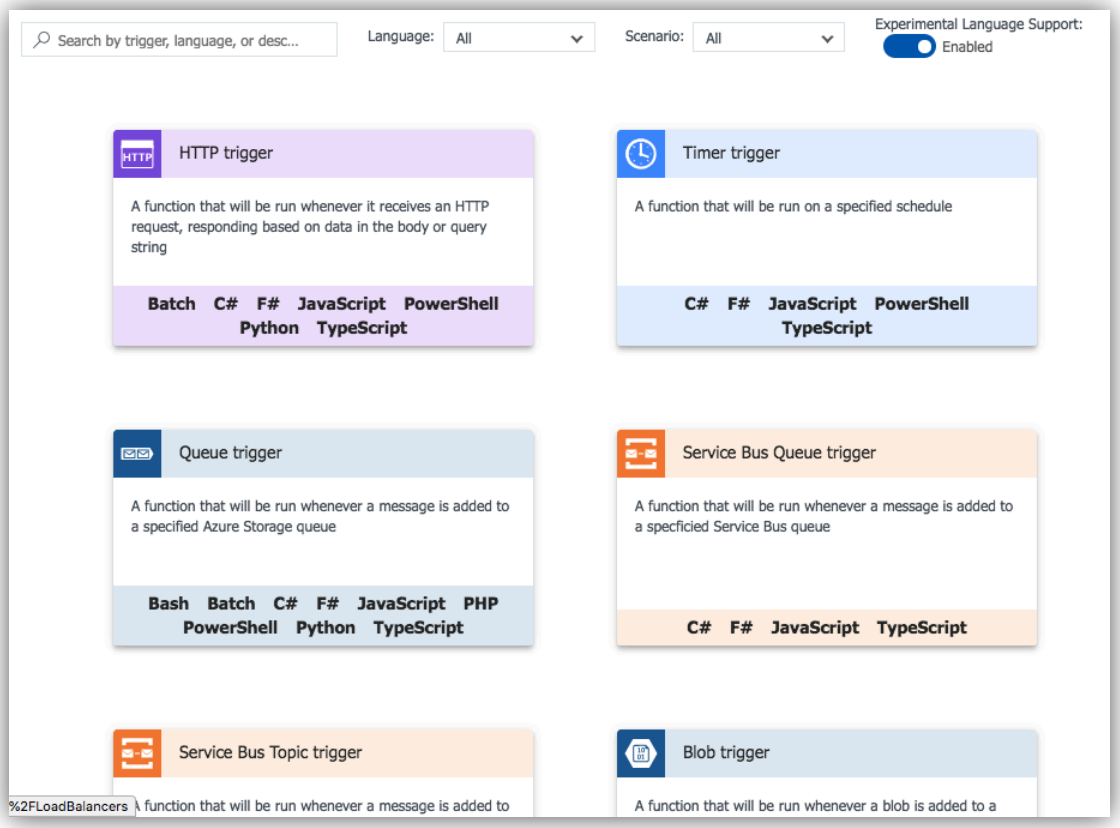
1. napisania kodu źródłowego,
2. wysłania kodu do chmury.

Kompilacja przeprowadzana jest automatycznie po stronie serwera za każdym razem, kiedy kod zostanie zaktualizowany.

Istotną kwestią w przypadku tej usługi jest możliwość łączenia jej z innymi serwisami Azure na zasadzie zdarzeń. Programista ma możliwość konfiguracji, kiedy jego funkcja powinna zostać uruchomiona. Niektóre z dostępnych możliwości to:

- zapytanie HTTP,
- uruchamianie cykliczne z określonym interwałem czasowym,
- nowy dokument w bazie CosmosDB,

- nowy „commit” w serwisie GitHub
- i wiele innych…



Ilustracja 4.11 Kreator tworzenia nowej instancji Azure Functions

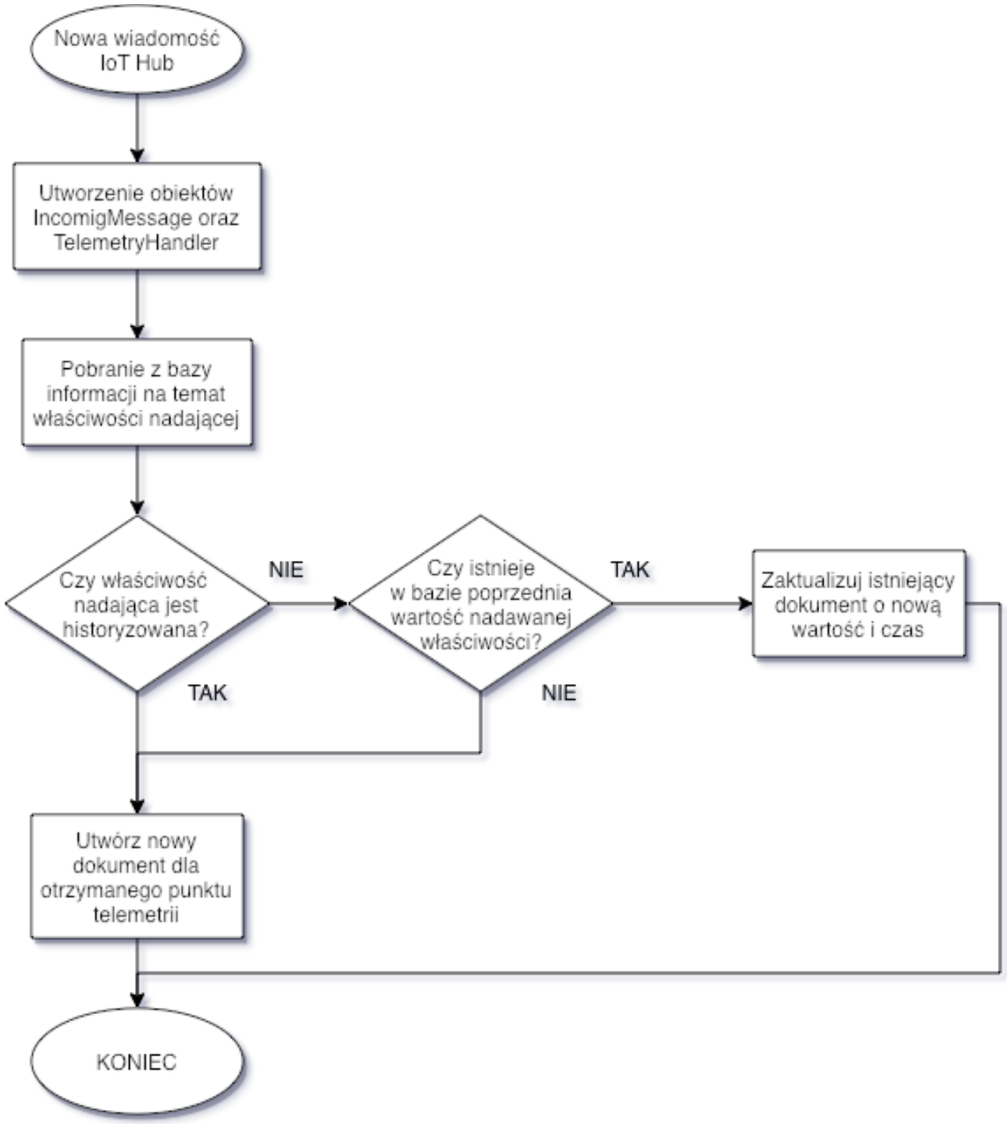
W przypadku dwóch funkcji stworzonych na potrzeby niniejszego projektu wybrane zostało zdarzenie „IotT Hub Message”. Oznacza to, że funkcje zostaną uruchomione, kiedy tylko dowolne urządzenie prześle telemetrię. Jest to bardzo wygodne rozwiązanie, które upraszcza kod źródłowy i umożliwia skupienie się na samej obsłudze wiadomości.

Podczas tworzenie nowej funkcji istnieje również możliwość wyboru języka programowania jaki chcemy wykorzystać. Obecnie najlepsze wsparcie zapewnione jest dla: C#, F# oraz JavaScript. Niektóre zdarzenia dostępne są jednak również dla takich języków jak: Bash, Batch, PHP, Powershell, Python, TypeScript. Jest to jednak wsparcie eksperymentalne i obecnie nie zalecane jest stosowanie wymienionych technologii w rozwiązaniach produkcyjnych.

Warto również wspomnieć o dostępności oficjalnego dodatku Azure Functions dla środowiska Visual Studio, które umożliwia publikowanie nowego kodu do chmury bez opuszczania edytora kodu źródłowego. Ta oraz wymienione w poprzednich akapitach cechy Azure Functions

powodują, że tworzenie nowych rozwiązań programistycznych opartych o obsługę zdarzeń jest znacznie przyspieszone w stosunku do klasycznych rozwiązań.

4.6.2 Schemat blokowy



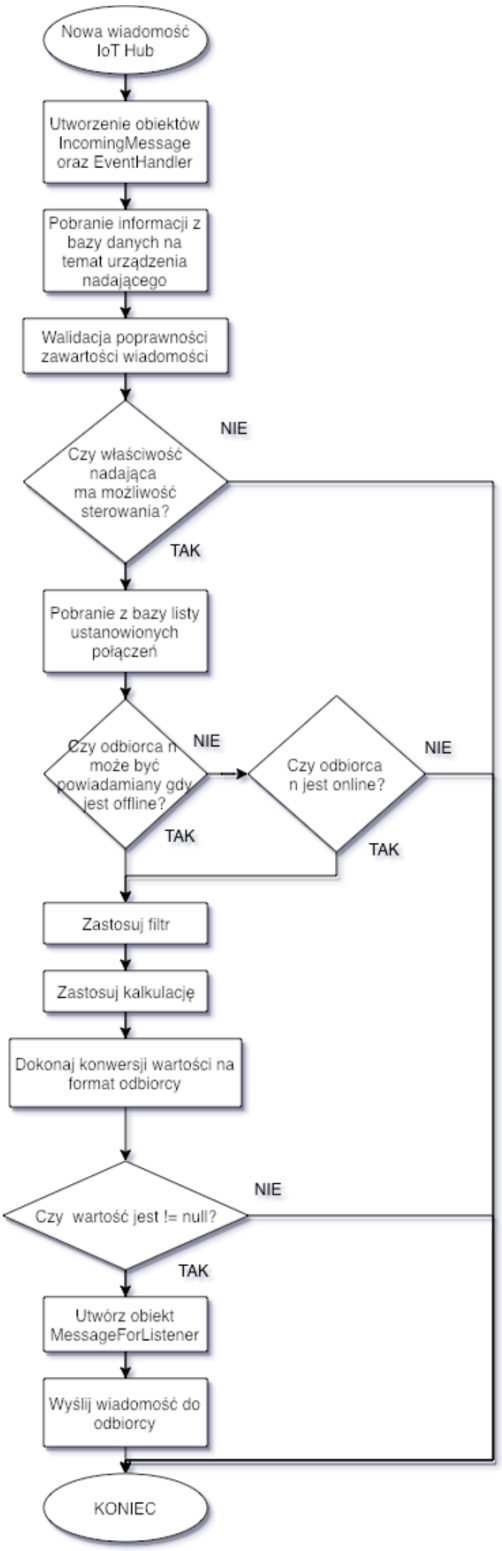
Ilustracja 4.12 Schemat blokowy algorytmu zapisu telemetrii do bazy danych

4.7 Serwis obsługi połączeń

Kolejnym serwisem wchodzącym w skład tzw. logiki platformy jest serwis obsługi połączeń między urządzeniami. Ponownie wykorzystana została usługa Azure Functions (osobna funkcja niż opisana w rozdziale *4.6 Serwise zapisu telemetrii*), która jest bardzo dobrym wyborem dla tego typu zastosowań. Ponownie została ona skonfigurowana jako serwis nasłuchujący

wiadomości przesyłane od urządzeń poprzez IoT Hub. W porównaniu do serwisu zapisu telemetrii jednak, wybrana została inna grupa konsumencka – kwestia ta została opisana w rozdziale dotyczącym IoT Hub'a. Aplikacja została napisana w języku C# z wykorzystaniem środowiska .NET Framework.

4.7.1 Schemat blokowy



Ilustracja 4.13 Schemat blokowy działania serwisu obsługi połączeń

Algorytm przedstawiony na ilustracji 4.10 w sposób ogólny prezentuje obsługę pojedynczej wiadomości, która trafia do platformy. Poszczególne kroki zostaną bardziej szczegółowo omówione w poniższym podrozdziale.

4.7.2 Opis działania

4.7.2.1 Walidacja wiadomości

Główną częścią aplikacji jest klasa *EventHandler*, która realizuje kolejne bloki schematu przedstawionego w poprzednim podrozdziale. Jednym z pierwszych kroków jest pobranie informacji z bazy danych na temat urządzenia nadającego. W tym celu wykorzystana została, przedstawiona wcześniej, klasa UnitOfWork, która umożliwia dostęp do bazy danych urządzeń. Zanim wyszukane zostaną ewentualne połączenia z innymi urządzeniami, walidowana jest zawartość wiadomości w celu wykluczenia wadliwych komunikatów. Sprawdzanie wiadomości realizowane jest przez statyczną klasę MessageValidator, której kod przedstawiono poniżej:

```
public static class MessageValidator
{
    public static bool IsMessageValid(IncomingMessage message,
        PropertyFormat propertyFormat)
    {
        if (propertyFormat == PropertyFormat.Boolean)
        {
            if (message.PropertyValue == "true" ||
                message.PropertyValue == "false")
                return true;
        }
        else if (propertyFormat == PropertyFormat.Number)
        {
            if (double.TryParse(message.PropertyValue.Replace('.', ','),
                                out double result))
                return true;
        }
        else if (propertyFormat == PropertyFormat.String)
            return true;

        return false;
    }
}
```

Kod źródłowy 4.13 Klasa MessageValidator służąca do sprawdzania poprawności wiadomości wysyłanych z urządzeń do chmury

Działanie metody *IsMessageValid* sprowadza się do weryfikacji wartości przesyłanej właściwości z uwzględnieniem jej typu (łańcuch tekstowy, liczba bądź wartość logiczna). Typ ten jest pobierany z bazy danych podczas zbierania informacji o nadawcy.

Sprawdzeniu podlegać powinna również sama struktura wiadomości (przedstawiona w rozdziale 4.5.3 *Struktura Bazy* – wiadomości przesyłane przez urządzenia mają ten sam format). Dzieje się to jednak już na samym początku (pierwszy blok schematu), podczas tworzenia obiektu IncomingMessage, który reprezentuje komunikat z urządzenia. Jeśli wiadomość posiada inny forma od wymaganego, zgłoszony zostanie wyjątek działania programu (zapisany w logach), a wiadomość nie będzie dalej procesowana.

Kolejnym etapem jest pobranie listy zdefiniowanych połączeń. W razie ich braku, program zostaje zakończony. W przeciwnym wypadku następują kolejne kroki algorytmu, zgodnie z przedstawionym schematem.

4.7.2.2 Dostępność urządzenia odbiorczego

Istotną funkcjonalnością usługi IoT Hub jest retencja wiadomości do 7 dni. Oznacza to, że w razie nieodebrania wysłanego komunikatu (kiedy urządzenie nie jest połączone z Internetem), będzie on „czekał” w kolejce i zostanie dostarczony do urządzenia, kiedy podłączy się ono do platformy. Jest to funkcjonalność bardzo przydatna, ale jednocześnie stwarza pewne problemy w przypadku wykorzystania urządzeń małej mocy, o niewielkiej ilości pamięci – np. mikrokontrolerów. Nietrudno takie urządzenie zawiesić, przepełniając jego bufor odbiorczy. Sytuacja taka nastąpi w przypadku, kiedy wyślemy większą ilość komunikatów, podczas gdy mikrokontroler jest wyłączony. Po jego uruchomieniu, otrzyma on naraz wszystkie wiadomości, co, przy odpowiedniej ilości komunikatów, zawiesi go.

Z tego powodu wprowadzona została do platformy MJIoT funkcjonalność, która umożliwia blokadę komunikacji, gdy urządzenie jest offline. Aktywacja opcji odbywa się na poziomie definicji typu urządzenia i została przedstawiona w poprzednich rozdziałach.

Proces sprawdzania stanu połączenia musiał zostać zaplanowany i zaimplementowany „ręcznie” ze względu na brak takiego mechanizmu w IoT Hub’ie. Istnieje co prawda właściwość *ConnectionState*, która udostępniona jest dla każdego urządzenia zarejestrowanego w usłudze. Eksperymenty wykazały jednak, że wartości, które zwraca nie mogą być traktowane jako wiarygodne. Przykładowo, po wyłączeniu testowego urządzenia, *ConnectionState* było raportowane jako „Connected” jeszcze przez 6 minut. Jest to niedopuszczalne opóźnienie, jeśli

chcemy przeciwdziałać ewentualnej możliwości zawieszenia urządzenia. Z tego powodu zastosowany został inny pomysł. Usługa IoT Hub, oprócz wysyłania zwykłych komunikatów umożliwia również wywoływanie metod na urządzeniach. Co ważne, urządzenia mają możliwość zwrócić odpowiedź przy wywołaniu metody. Jeśli natomiast wywołanie zakończy się niepowodzeniem (z powodu problemów z połączeniem), zwrócony zostanie błąd. Sprawdzenie dostępności sprowadza się więc do wywołania metody nazwanej *CheckConnection*. W razie odpowiedzi z urządzenia docelowego, zostaje ono uznane za podłączone. W razie błędu natomiast, traktowane jest jako odłączone.

Proces ten wykonywany jest za każdym razem, kiedy urządzenie niskiej mocy jest adresatem wiadomości. W przypadku innych urządzeń, krok ten jest pominięty. Warto zaznaczyć, że funkcjonalność ta nie była w ogóle planowana na etapie ustalania założeń projektowych. Potrzeba jej implementacji wyszła na jaw dopiero podczas testów prototypu platformy. Ze względu na fakt, że urządzenia oparte o mikrokontrolery miały stanowić główny rodzaj urządzeń w platformie, postanowiłem dodać omawiany mechanizm. Trudność polegała głównie na niedostępności informacji na ten temat, opracowane rozwiązanie jest więc autorskim pomysłem.

4.7.2.3 Modyfikacja wartości

W kolejnym kroku następuje zastosowanie kolejnych operacji mających na celu przygotowanie nowej wiadomości, która zostanie wysłana do każdego z zarejestrowanych odbiorców (urządzeń sterowanych). Łańcuch ten przedstawiony został w rozdziale 3.3.4 *Schemat przetwarzania wartości*. Warto na chwilę zatrzymać się w tym miejscu, aby wyjaśnić realizację tego procesu. Każdy etap to osobna klasa, gdzie każda z nich implementuje interfejs *IValueModifier*. Instancje tych klas są elementami tablicy. W związku z tym, wywołanie procesu modyfikacji wartości sprowadza się do iterowania po tej kolekcji i wywoływania metody *Modify*(…), której istnienie wymusza wspomniany interfejs:

```
string modifiedValue = _message.PropertyValue;
foreach (var modifer in _valueModifiers)
    modifiedValue = modifer.Modify(modifiedValue, connection);
```

Kod źródłowy 4.14 Wywołanie kolejnych modyfikatorów wartości

Sama kolejność modyfikatorów jest natomiast ustawiana w konstruktorze klasy *EventHandler*:

```
_valueModifiers = new List<IValueModifier>
      { new Filter(), new Calculation(), new ValueFormatConverter() };
```

Kod źródłowy 4.15 Konfiguracja kolejności modyfikatorów wartości

4.7.2.4 Wysyłanie rozkazów

Ostatnią czynnością do wykonania jest wysłanie komunikatów do urządzeń sterowanych. Wykorzystana została biblioteka do obsługi IoT Hub, która udostępnia asynchroniczną metodę *SendAsync(...)*.

Postać komunikatów jest podobna do telemetrii otrzymywanej z urządzeń:

```
{
    "ReceiverId": 8,
    "PropertyName": "LED State",
    "PropertyValue": "true"
}
```

Różnicą jest brak klucza „Timestamp”, który w przypadku telemetrii jest opcjonalny.

4.8 Web API informacji o urządzeniach

Rozpoczynając opis kolejnej części systemu – Web API urządzeń – przechodzimy jednocześnie do omawiania „warstwy” platformy, która udostępniona jest publicznie. Oznacza to, że klienci/użytkownicy, korzystający z systemu, mają możliwość dostępu do niej, wiedzą o jej istnieniu. Ma to znaczenie głównie ze względu na fakt, że publiczna część jakiegokolwiek systemu powinna zachować pewną stałość, jeśli chodzi o sposób jej użytkowania. Projektant dowolnego rozwiązania nie może zakładać, że jego klienci będą skłonni w krótkich okresach czasu wielokrotnie zmieniać swoje przyzwyczajenia czy uczyć się od nowa obsługi produktu. Takie praktyki zazwyczaj zniechęcają większość użytkowników, dając im impuls do poszukiwania rozwiązania konkurencyjnego. W związku z powyższym, przed opublikowaniem np. interfejsu Web API, należy dobrze przemyśleć sposób dostępu do niego. Każda zmiana może być dla niektórych użytkowników sporym problemem.

4.8.1 Rola Web API

Opis serwisu warto zacząć od przybliżenia czym on w ogóle jest. Samo Web API to dobrze znany sposób komunikacji użytkowników z systemami informatycznymi z wykorzystaniem protokołu HTTP/S i jego metod (m. in. GET, POST, PUT, DELETE). Często mamy kontakt z

rozwiązaniami tego typu nie wiedząc o tym. Najprościej rzecz ujmując Web API stanowi punkt dostępowy dla użytkownika do informacji z danego systemu, często pozwalając również, aby to użytkownik pewne informacje udostępniał. Przykładowo, załóżmy, że posiadamy bogatą bazę danych, której zbiór chcielibyśmy udostępnić publicznie. Moglibyśmy np. stworzyć stronę internetową, która prezentowała by wszystkie dane. Nie zawsze jest to jednak wygodne rozwiązanie. Co, jeśli użytkownik chciałby mieć możliwość pobrania danych, przetworzenia ich i wykorzystania do swoich potrzeb? Dla takich scenariuszy idealnym rozwiązaniem jest stworzenie Web API o wygodnym interfejsie dostępowym. Dane zwracane są najczęściej w znanych formatach: XML lub JSON. Formaty te idealnie nadają się do dalszego przetwarzania.

Warto zaznaczyć, że często mamy do czynienia z sytuacją, kiedy firmy tworzą interfejsy Web API, które nie są dostępne publicznie – są one wykorzystywane np. przez stronę internetową firmy, która w ten sposób pobiera dane do wyświetlenia w ściśle określony sposób. Jako że niniejszy projekt stanowi próbę stworzenia otwartej platformy IoT, dostęp do Web API z założenia jest publiczny – użytkownik powinien mieć możliwość pobrania danych i prezentacji ich w dowolny sposób. Może jednak zamiast tego wykorzystać aplikację kliencką, która jest wygodniejsza w obsłudze (ponieważ posiada „klikalny" interfejs graficzny), ale w rzeczywistości sama korzysta z tego samego Web API. Aplikacja ta zostanie przedstawiona w dalszej części pracy.

Tak jak w poprzednich rozdziałach, przedstawię prosty przykład, który najlepiej wyjaśni zagadnienie. Załóżmy, że chcemy zdobyć informacje nt. dostępnych urządzeń wewnątrz platformy. Możemy wykorzystać URL:

https://{ADRES_IP}/GetDevices?includeConnections=false&includeAvailability=true&includeProperties=false

Powyższy adres zawiera m. in.:

- zasób (ang. *endpoint*), który nas interesuje, w tym przypadku *GetDevices*, który zwraca listę urządzeń wraz z ich charakterystyką;
- dodatkowe parametry, które określają zawartość odpowiedzi z serwera. W tym przypadku, dla uproszczenia, pominięte zostaną informacje o połączeniach zdefiniowanych dla poszczególnych urządzeń (includeConnections) oraz lista ich właściwości (includeProperties). Dołączone zostaną natomiast stany online (includeAvailability), ponieważ są to wartości logiczne, które nie komplikują niepotrzebnie przykładu.

Oto przykładowa odpowiedź z serwera:

WSTAWIĆ ODPOWIEDŹ Z SERWERA

Powyższe dane dają użyteczny zbiór informacji. Ich prezentacja nie jest zbyt atrakcyjna ani czytelna dla mniej zaawansowanych użytkowników, więc zazwyczaj są one przedstawiane w ciekawszej formie. Generalnie jednak rola Web API sprowadza się do tego, by obsługiwać zapytania zwracając właściwe informacje (bądź błędy, jeśli zapytanie nie jest prawidłowe).

4.8.2 Wybór technologii

Wybór technologii nie ma tak naprawdę wielkiego znaczenia. Obecnie tworzenie rozwiązań serwerowych jest możliwe z wykorzystaniem dziesiątek rozwiązań programistycznych. Nie można tak naprawdę jednoznacznie stwierdzić, które z nich będzie najlepsze, ponieważ najczęściej zależy to od preferencji programisty. W moim przypadku wybór padł na ASP .NET 5 Web API 2, które jest rozwiązaniem sprawdzonym i wykorzystywanym na całym świecie w profesjonalnych rozwiązaniach. Przede wszystkim jednak zwyciężyła chęć wykorzystania przygotowanej wcześniej biblioteki dostępowej do bazy danych urządzeń (wspomniana już klasa UnitOfWork), która została napisana w środowisku .NET Framework.

Stworzoną aplikację należy opublikować. Idealnym wyborem w tym przypadku jest usługa Azure App Service do wdrażania aplikacji serwerowych oraz stron internetowych.

Ilustracja 4.14 Logo usługi Azure App Service

Jest to serwis zbliżony funkcjonalnością do opisanego już Azure Functions z tą różnicą (nie jedyną), że Azure Functions nie może hostować projektów opartych o ASP .NET. Są inne rozwiązania, jednak zdecydowałem się pozostać przy ASP ze względu na chęć poznania choć części tego środowiska, które, oprócz projektów Web API, umożliwia tworzenie części serwerowych bogatych serwisów internetowych.

Jeśli chodzi o sam proces pisania aplikacji, wykorzystane zostało, tak jak w większości aplikacji składających się na niniejszy projekt, środowisko Visual Studio 2017. W związku z tym, że

całość wykorzystanego zbioru technologii pochodzi od firmy Microsoft, zostały one dobrze połączone ze sobą. Przykładowo więc publikowanie aplikacji ASP .NET jako zawartość Azure App Service, z poziomu Visual Studio to kwestia jedynie kilku kliknięć.

Warto wspomnieć też, że aplikacja wykorzystuje przygotowany zestaw bibliotek, które ułatwiają dostęp do poszczególnych części. Chodzi tu przede wszystkim o dostęp do baz danych. Dobrą praktyką jest wydzielanie wspólnych części większego rozwiązania i wykorzystywanie ich w razie potrzeby, zamiast tworzenia tej samej części za każdym razem od nowa. W przypadku jakiejkolwiek zmiany (np. sposobu wydobywania pewnych informacji z bazy danych) należy dokonać rekompilacji jedynie pojedynczej wspólnej biblioteki. Rozwiązania, które z niej korzystają wymagają podmiany jej plików – nie wymaga to kompilacji.

4.8.3 Dostępne zasoby

Każdy serwis Web API charakteryzuje się określoną listą dostępnych zasobów, tzn. punktów dostępowych, z których można uzyskać dane. W kolejnych podrozdziałach wylistowane zostaną dostępne rozkazy wraz z ich opisami i przykładami użycia. Należy zauważyć, że rozróżnić możemy dwa rodzaje rozkazów: komendy oraz kwerendy. Pierwsza kategoria dotyczy przypadków, kiedy naszym celem jest wykonanie pewnej akcji mającej trwałe skutki. Druga natomiast reprezentuje intencje pobrania pewnych danych – bez ich modyfikacji. Dobrą praktyką jest rozdzielenie tych dwóch kategorii zapytań, tzn. projektowanie API w taki sposób, aby każdy pojedynczy punkt dostępowy należał do jednej z wymienionych kategorii. Taki sposób projektowania powinien być stosowany nie tylko w przypadku Web API, ale również podczas definiowania metod klas programu napisanego obiektowo. Takie podejście nosi nazwę *Command-Query Separation* [10]. Punkty dostępowe opisane poniżej również można podzielić na komendy (nie mają przedstawionych odpowiedzi z serwera) i kwerendy (dla których zaprezentowano przykładowe odpowiedzi z serwera).

4.8.3.1 Lista urządzeń

Ten punkt dostępowy daje możliwość uzyskania listy urządzeń jakie dany użytkownik posiada zarejestrowane wewnątrz platformy MJIoT.

Nazwa URL: GetDevices

Typ zapytania: GET

Parametry:

- includeConnections – informacje o zdefiniowanych połączeniach dla każdego z urządzeń,
- includeAvailability – informacje o stanie online poszczególnych urządzeń,
- includeProperties – informacje o dostępnych właściwościach poszczególnych urządzeń

Przykładowe zapytanie:

https://{ADRES_IP}/GetDevices?includeConnections=true&includeAvailability=true&includeProperties=true

Przykładowa odpowiedź:

WSTAWIĆ ODPOWIEDŹ Z SERWERA

4.8.3.2 Lista właściwości

Ten punkt dostępowy służy do pobierania listy właściwości dla urządzenia o wskazanym numerze ID.

Nazwa URL: GetProperties

Typ zapytania: GET

Parametry:

- deviceId – ID urządzenia, którego dotyczy zapytanie

Przykładowe zapytanie:

https://{ADRES_IP}/GetProperties?deviceId=7

Przykładowa odpowiedź:

WSTAWIĆ ODPOWIEDŹ Z SERWERA

4.8.3.3 Dodawanie połączeń

Za pomocą niniejszego wywołania, użytkownik ma możliwość definiowania połączeń pomiędzy urządzeniami.

Nazwa URL: AddConnections

Typ zapytania: POST

Parametry:

Parametry przekazywane są jako ciało zapytania POST w postaci tablicy obiektów JSON, które tworzone są na podstawie poniższych klas:

```
public class ConnectionInfo
{
    public DevicePropertyPair Sender { get; set; }
    public DevicePropertyPair Listener { get; set; }
    public ConnectionFilter Filter { get; set; }
    public string FilterValue { get; set; }
    public ConnectionCalculation Calculation { get; set; }
    public string CalculationValue { get; set; }
}

public class DevicePropertyPair
{
    public int DeviceId { get; set; }
    public int PropertyId { get; set; }
}
```

Kod źródłowy 4.16 Klasy, na podstawie których tworzona jest zawartość POST AddConnections

Zawarte są więc informacje nt.: ID urządzeń oraz właściwości, które mają zostać połączone; zdefiniowany filtr i jego ewentualna wartość; kalkulacja oraz jej wartość. Jest to więc zbiór wszystkich potrzebnych informacji jakie wymagane są do określenia połączenia

Przykładowe zapytanie:

https://{ADRES_IP}/SetConnections

Ciało POST:

WSTAWIĆ CIAŁO POST

4.8.3.4 Lista połączeń

Kolejny URL pozwala pobierać listę zdefiniowanych połączeń pomiędzy urządzeniami.

Nazwa URL: GetConnections

Przykładowe zapytanie:

https://{ADRES_IP}/GetProperties?deviceId=7

Przykładowa odpowiedź:

WSTAWIĆ ODPOWIEDŹ Z SERWERA

4.8.3.5 Usuwanie połączenia

Kolejny URL pozwala usuwać połączenie wskazane przez użytkownika. Jako argument należy podać numer ID połączenia, które chcemy usunąć.

Nazwa URL: RemoveConnection

Przykładowe zapytanie:

https://{ADRES_IP}/RemoveConnection?connectionId=24

Omówione zostały niektóre punkty dostępowe spośród dostępnych możliwości stworzonego Web API. Ich wybór nie jest przypadkowy – właśnie ten podzbiór zapytań wykorzystywany jest przez aplikację kliencką, która omówiona zostanie w dalszej części pracy. Pozostałe funkcjonalności nie zostały wykorzystane, jednak można wymienić je poniżej:

- SetConnections – działa podobnie do AddConnections, z tą różnicą, że usuwa połączenia, które były zdefiniowane wcześniej;
- GetDeviceListeners – zwraca listę połączeń, w których rolę „sterownika" pełni urządzenie o wskazanym ID;
- GetPropertyListeners – zwraca listę połączeń, gdzie „sterownikiem" jest wskazana właściwość wybranego urządzenia;

Podczas projektowania Web API, wydawało mi się, że powyższe funkcje będą potrzebne bądź nawet niezbędne do stworzenia aplikacji klienckiej. Okazało się jednak, że ich użycie nie było wymagane. Mimo to, są one zdefiniowane i możliwe do wykorzystania przez użytkowników.

4.9 Web API telemetrii

Każda z właściwości, które posiadają urządzenia zarejestrowane w platformie MJIoT, raportuje zmiany swoich wartości do chmury. Dzięki temu realizowane są dwa cele:

- komunikacja między urządzeniami;
- zapis wartości.

Drugi z wymienionych celów nie miałby sensu, gdyby niedostępna była możliwość odczytu zapisanych wartości. Zadanie to realizowane jest z wykorzystaniem Web API telemetrii, któremu poświęcony został niniejszy rozdział.

4.9.1 Wybór technologii

Wiedząc już, z poprzedniego rozdziału, czym właściwie jest Web API, możemy do razu przejść do omówienia, na jakiej podstawie zbudowane zostało Web API telemetrii. Wybór technologii, podobnie jak w przypadku Web API urządzeń, nie ma wielkiego znaczenia – dostępne rozwiązania są na tyle rozwinięte, że wybór zależy właściwie jedynie od preferencji lub wygody programisty. Ponownie więc mógłbym skorzystać ze środowiska .NET oraz framework'a ASP. W tym przypadku jednak, sytuacja jest nieco inna. Poprzednie Web API silnie bazowało na dostępie do bazy danych urządzeń – naturalnym więc była chęć ponownego wykorzystania stworzonej przeze mnie już biblioteki UnitOfWork oraz jej repozytoriów. W przypadku Web API telemetrii jednak, przede wszystkim wykorzystywana jest druga baza danych - baza telemetrii. Pierwsza z baz wykorzystywana jest jedynie w niewielkim stopniu, co zostanie jeszcze wyjaśnione.

W odróżnieniu od poprzedniego projektu, nie ma żadnego argumentu, który wskazywałby na wybór ASP .NET, poza moją znajomością tej technologii. W związku z tym postanowiłem wykorzystać niniejszy projekt jako okazję do poznania nowego środowiska, które obecnie zdobywa ogromną popularność – chodzi o NodeJS oraz framework Express.

Ilustracja 4.15 Logo NodeJS

NodeJS to środowisko uruchomieniowe języka JavaScript, które bazuje na wykorzystaniu V8 – otwarto-źródłowego silnika JavaScript stworzonego przez Google (i wykorzystywanego w przeglądarce Chrome). Mimo że język JavaScript jest interpretowany, aplikacje stworzone z jego wykorzystaniem charakteryzują się lepszą wydajnością, niż rozwiązania zaimplementowane przy użyciu innych technologii. Jako przykład może służyć przypadek firmy PayPal [11], która dokonała migracji swoich rozwiązań serwerowych opartych o środowisko Java na platformę NodeJS. Oto powody jakimi firma motywuje swoją decyzję:

- wykorzystanie tego samego języka (JavaScript) po stronie serwerowej (ang. *back-end*) oraz interfejsu użytkownika (ang. *front-end*), co ułatwia komunikację między programistami, a nawet przejęcie obu ról przez jeden zespół (ang. *full-stack developers*);
- w porównaniu do rozwiązania opartego o Javę, budowanie aplikacji w Node.JS zakończyło się dwukrotnie szybciej, przy mniejszej ilości programistów;
- rezultat zawierał 33% mniej linijek kodu niż Web API w Javie. Co jednak ważniejsze, aplikacja była w stanie obsłużyć 2 razy więcej zapytań w ciągu sekundy, przy czasie obsługi pojedynczego zapytania mniejszym średnio o 35%.

Szybkość działania Web API w NodeJS wynika z asynchroniczności, która jest podstawą działania środowiska. Mimo wykorzystania tylko jednego wątku procesora, operacje bazujące na obsłudze interfejsów I/O (wejścia-wyjścia) realizowane są równolegle z głównym wątkiem, nie blokując go. Ta cecha NodeJS ma znaczenie, kiedy porównujemy to środowisko z Javą, która nie wspiera podejścia opartego o asynchroniczne operacje I/O. Kiedy jednak weźmiemy pod uwagę inne rozwiązania, jak np. .NET, okaże się, że wydajność jest w tym przypadku porównywalna – środowisko .NET od wersji 4.5 umożliwia programowanie w modelu asynchronicznym. Niezaprzeczalnym faktem jest jednak kwestia tzw. „progu wejścia", który w przypadku NodeJS jest znacznie niższy niż w konkurencyjnych rozwiązaniach. Przy wykorzystaniu framework'a Express (użytego w niniejszym projekcie), Web API typu „Hello World" można napisać w 4 linijkach kodu:

```
const express = require('express')
const app = express()

app.get('/', (req, res) => res.send('Hello World!'))
app.listen(3000, () => console.log('Example app listening on port 3000!'))
```

Kod źródłowy 4.17 Najprostszy program Web API z wykorzystaniem NodeJS + Express

4.9.2 Funkcjonalność

W omawianym Web API dostępny jest tylko jeden punkt dostępowy – służy on do pobierania telemetrii wybranej właściwości wybranego urządzenia.

Adres URL:

https://{ADRES_IP}/

/api/{deviceId}/{propertyName}?startTime={startTime}&endTime={endTime}

Adres posiada cztery parametry. W odróżnieniu od poprzedniego Web API, dwa z nich podawane są już w części stanowiącej ścieżkę URL. Są to *deviceId* oraz *propertyName*. Taka konwencja bliska jest założeniom REST (ang. REpresentational State Transfer) [12], które między innymi określają, że ścieżka do zasobu powinna hierarchicznie ukazywać jego logiczne położenie. W tym przypadku odwołujemy się do pewnej właściwości, która należy do konkretnego urządzenia. Dwa pozostałe parametry (*startTime* oraz *endTime*) znajdują się standardowo w części zapytań adresu.

Mimo dostępności tylko jednego punktu dostępowego, dostępne są tak naprawdę dwa rezultaty działania Web API.

- Pobranie jedynie ostatniej wartości

  W przypadku, kiedy klient API nie zdefiniuje ani czasu startowego (*startTime*) ani końcowego (*endTime*), aplikacja pobierze z bazy, i zwróci, jedynie ostatnią zaraportowaną wartość. Ta ścieżka działania jest jedyną opcją, kiedy operujemy na właściwości niehistoryzowanej, gdyż ostatnia wartość jest wtedy jedyną dostępną w bazie.

  Przykład zapytania:

  https://{ADRES_IP}/ api/7/LampColor

  Przykładowa odpowiedź:

  WSTAWIĆ PRZYKŁADOWĄ ODPOWIEDŹ

- Pobranie wartości ze wskazanego przedziału czasowego

  Inna możliwością jest pobranie tablicy wartości. W tym celu należy zdefiniować czas początkowy (*startTime*) lub czas końcowy (*endTime*). W razie zdefiniowania tylko jednego z tych parametrów, drugi z nich przyjmie minimalną (*startTime*) bądź maksymalną (*endTime*) wartość. Oznacza to, że kiedy zdefiniujemy przykładowo jedynie czas końcowy, zwrócona tablica zawierać będzie wartości: od najstarszej do ostatniej, która mieści się we wskazanym czasie końcowym.

  Przykład zapytania:

  https://{ADRES_IP}/ api/7/LampColor?startTime=2018&endTime=2019

  Przykładowa odpowiedź:

WSTAWIĆ PRZYKŁADOWĄ ODPOWIEDŹ

4.9.3 Format czasu

Dosyć istotną kwestią jest format czasu, którym posługuje się platforma MJIoT. Z formatem tym mamy do czynienia w następujących sytuacjach:

- wysyłanie telemetrii z urządzeń (opcjonalnie, ponieważ urządzenia nie muszą definiować czasu);
- zapis telemetrii z urządzeń do bazy danych (jeśli urządzenie samo nie określi czasu wartości, przydzieli go platforma);
- wysyłanie zapytania do Web API telemetrii (wartości *startTime* oraz *endTime*);
- odbieranie danych z Web API telemetrii (zawartość *timestamp*).

Poprawne działanie rozwiązania wymaga, aby określony został jeden format czasu wykorzystywany podczas każdej z wymienionych czynności. Inaczej, interpretacja danych w bazie telemetrii byłaby problematyczna.

Podczas fazy prototypowania popełniłem błąd, nie zastanawiając się w ogóle nad wspomnianą kwestią. Problem wyszedł na jaw podczas testowania Web API telemetrii – okazało się, że dane historyczne zwracane są w złej kolejności z powodu formatu czasu, który nie daje się łatwo sortować. Początkowo format był następujący:

MM/DD/YYYY HH:MM:SS

Tak naprawdę występowały tu dwa problemy:

- wspomniana kwestia sortowania – nie wystarczy zastosować sortowania alfabetycznego w celu poprawnego ułożenia danych,
- niewystarczająca rozdzielczość – niektóre urządzenia mogą przesyłać dane kilka razy w ciągu sekundy, co spowodowałoby, że kilka wartości miałoby przypisany ten sam czas.

Rozwiązaniem, które zastosowałem było wykorzystanie formatu określonego przez normę *ISO 8601* [13], który przedstawia się następująco:

YYYY-MM-DD THH:MM:SS:fffffffK

Powyższy format rozwiązuje oba wskazane problemy. Ustawienie danych chronologicznie wymaga jedynie prostego sortowania alfabetycznego. Natomiast rozdzielczość czasu jest rzędu setek nanosekund, co w zupełności wystarczy dla potrzeb projektu MJIoT.

Inną kwestią jest region określania czasu – zastosowane zostało założenie, że czas zawsze powinien być określany jako UTC (Uniwersalny Czas Koordynowany). Po stronie klientów leży ewentualna konwersja tego czasu na lokalny.

4.10 Web API tokenów oraz uwierzytelnianie

Kwestią, która w żaden sposób nie była dotychczas poruszana jest autentyfikacja użytkownika oraz autoryzacja jego zapytań. Opisane zostały dwie aplikacje typu Web API – obie operują na danych użytkowników, które muszą pozostać zabezpieczone przed nieautoryzowanym dostępem. Tak też jest w rzeczywistości – poprzednie rozdziały nie wspominały o tym, aby temat ten omówić w jednym miejscu – niniejszym rozdziale.

4.10.1 Tokeny JWT

Wykorzystany sposób zabezpieczenia aplikacji Web API polega na wykorzystaniu tokenów JWT (ang. *JSON Web Token*). Sposób ich zastosowania zostanie przybliżony w tym podrozdziale.

Tokeny JWT to jeden ze sposobów zabezpieczenia rozwiązania typu Web API przed dostępem nieautoryzowanym. Istnieje również wiele innych metod, które spełniają to zadanie, np.:

- Basic HTTP Authentication
- OAuth2

Pierwsze z nich zostało przeze mnie odrzucone ze względu na niewielki stopień zabezpieczeń, który oferuje – hasło jest kodowane za pomocą Base64, co oznacza, że haker może z łatwością uzyskać hasło z szyfrogramu. Drugie zaś, w celu uwierzytelnianienia, korzysta z usług firm trzecich i wymaga logowania u zewnętrznego dostawcy (najpopularniejsi dostawcy: Google, Facebook, Microsoft), co dodaje, nie zawsze pożądaną, zależność od tychże firm trzecich.

4.10.1.1 Zasada wykorzystania tokenów

Najlepiej wytłumaczyć zagadnienie przedstawiając proces jego działania:

1. Użytkownik chce zalogować się do pewnego serwisu.
2. Użytkownik podaje swoje dane logowania (najczęściej login oraz hasło).

3. Aplikacja wysyła dane logowania użytkownika (najlepiej poprzez bezpieczny protokół HTTPS) do serwera wydającego tokeny.
4. Serwer wydający tokeny sprawdza poprawność otrzymanych danych i, jeśli dane są poprawnie, odsyła token – zaszyfrowany ciąg znaków, który używany będzie przez użytkownika do potwierdzenia jego tożsamości. Token ważny jest jedynie przez pewien czas (np. 15 minut).
5. Aplikacja, do której logował się użytkownik, otrzymuje token, i wykorzystuje go do kontaktu z właściwym serwerem obsługującym serwis będący punktem jego zainteresowania (np. w celu pobrania listy urządzeń).
6. Serwis ten, sprawdza token, który dołączony był do zapytania, i w razie jego poprawności, odsyła dane, o które prosił użytkownik.
7. Użytkownik otrzymuje dane, co było jego celem.

Powyższa lista dosyć ogólnie przedstawia proces otrzymania i wykorzystania tokena. Istotnym jest tutaj fakt, że aplikacja, z której korzystał użytkownik, kontaktowała się z dwoma punktami dostępowymi, zanim wyświetlone zostały wymagane dane. Co więcej, kontakt z serwerem tokenów został nawiązany tylko raz – w celu wygenerowania kodu. Serwer zwracający właściwe dane (interesujące użytkownika) jest w stanie samodzielnie, bez kontaktu z zewnętrznym serwisem, potwierdzić tożsamość użytkownika.

Wiedząc już, jak ogólnie wygląda proces uwierzytelniania z wykorzystaniem tokenów, warto zastanowić się, w czym tak naprawdę metoda ta jest lepsza od klasycznego sposobu, w postaci wysyłania pary login-hasło. Główną zaletą jest to, że token posiada określony czas ważności, co zmniejsza jego użyteczność dla ewentualnego hakera. Kolejną zaletą, która nie jest tak oczywista, jest ochrona użytkownika przed włamaniami na inne serwisy, których jest on klientem. Przy korzystaniu z tokena, nawet jeśli zostanie on przechwycony, haker otrzymuje dostęp jedynie do jednego serwisu. Jeśli przechwycone zostaną login oraz hasło, napastnik posiada potencjalny dostęp do innych serwisów, do których użytkownik być może loguje się tymi samymi danymi. Zauważyć można jeszcze co najmniej jedną zaletę – tokeny mogą zawierać dodatkowe informacje, np. zbiór zasobów, do których użytkownik może uzyskać autoryzację.

Poza korzystaniem z dobrej metody uwierzytelniania, nie należy zapominać o odpowiednim zabezpieczeniu samego procesu przesyłania danych – podstawą jest wykorzystanie protokołu HTTPS, co obecnie staje się standardem.

4.10.1.2 Generowanie JWT

Wiedząc już jak wygląda ogólny proces wykorzystania tokenów, warto nieco więcej uwagi poświęcić samym tokenom i dowiedzieć, jak są generowane. Ich postać przedstawia się następująco:

NAGŁÓWEK.ZAWARTOŚĆ.SYGNATURA

Token składa się więc z trzech części oddzielonych kropką – jest to więc po prostu łańcuch tekstowy. Najlepiej zacząć od omówienia czym jest *ZAWARTOŚĆ*. Jej przykład znajduje się poniżej. Zaprezentowany JSON to jednocześnie przykład *ZAWARTOŚCI*, która wykorzystywana jest w projekcie MJIoT.

```
{
    "iss" : "MJIoT Authentication Service",
    "sub" : 12,
    "exp" : "2018-08-16 T20:03:31.61475332"
}
```

Zawarte tutaj dane to zbiór informacji, który jest wymagany podczas autentykacji zapytania. Jest więc zawarta informacja o jednostce, która wygenerowała token (*iss*), identyfikator użytkownika (*sub*) oraz czas wygaśnięcia tokena (*exp*). Zbiór informacji, który się tu znajduje nie jest w żaden sposób wymuszony na programiście – można tu umieścić dowolne informacje, z jednym zastrzeżeniem – nie mogą one być poufne. Nie należy więc zapisywać haseł, kluczy ani innych informacji, które chcemy chronić. Wynika to z faktu, że *ZAWARTOŚĆ* nie jest szyfrowana!

Następnie omówię *SYGNATURĘ*, która stanowi ostatnią część klucza oraz jego najważniejszą część. Jest ona po prostu zaszyfrowaną *ZAWARTOŚCIĄ*. Często określa się tę część tokena jako MAC (ang. *Message Authentication Code*). Szyfrowanie odbywa się z wykorzystaniem klucza prywatnego, który znany jest jedynie przez serwer generujący token. Klucz ten stanowi jedną część pary kryptograficznej szyfrowania asymetrycznego. Drugi klucz – publiczny – służy do odszyfrowania *SYGNATURY*. Uwierzytelnienie polega więc na odszyfrowaniu *SYGNATURY* i porównaniu wyniku z *ZAWARTOŚCIĄ* – jeśli się one zgadzają, zapytanie uznawane jest jako poprawne.

Pozostaje jeszcze jedna kwestia – rodzaj wykorzystanego szyfrowania. W tym celu token zawiera *NAGŁÓWEK*. Oto jego postać:

```
{
    "alg": "RS256",
    "typ": "JWT"
}
```

Powyższy JSON informuje, że token został zaszyfrowany za pomocą RS-256. Mając taki zestaw informacji, można bezpiecznie obsłużyć zapytania użytkownika.

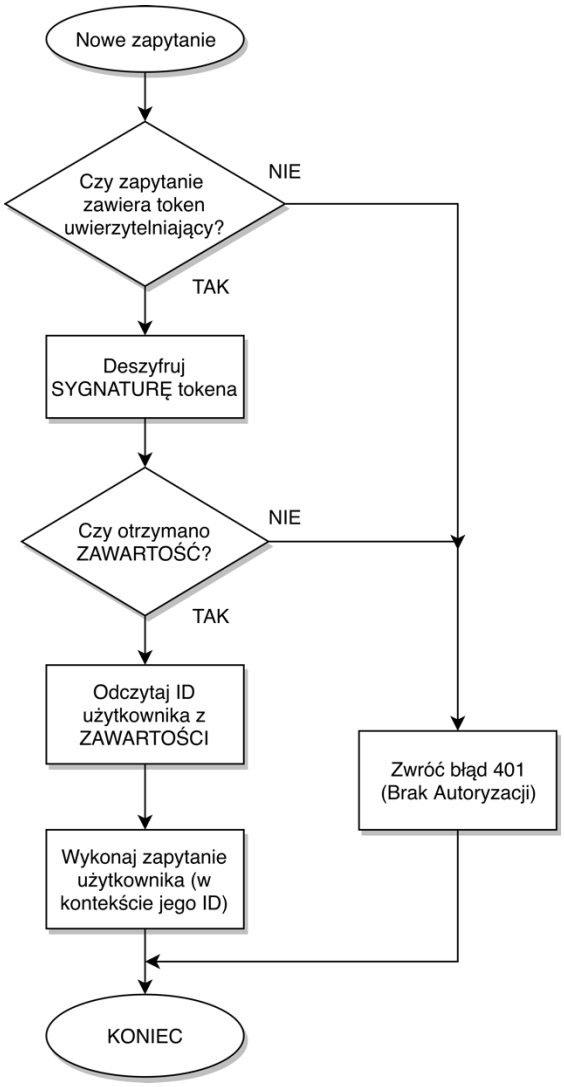
Oto przykładowy token JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Iktvc21
hIGkgQmFzaWEiLCJpYXQiOjE1MTYyMzkwMjJ9.au2s1GG1sj2cd70uQz1d5kFAblRW-
5SK6zaRkDteH6o
```

Kolor czerwony to *NAGŁÓWEK*, niebieski to *ZAWARTOŚĆ*, zielonym kolorem natomiast oznaczona jest *SYGNATURA*. Powyższy przykład nie zawiera żadnych obiektów JSON, mimo że przecież w takim formacie przedstawione zostały poszczególne części. Powodem tego jest fakt, że części tokena poddawane są kodowaniu Base64Url – jest to wersja Base64, która zawiera jedynie znaki możliwe do umieszczenia w adresie URL. Taki token może być więc łatwo przesyłany w Internecie.

4.10.1.3 Uwierzytelnienie użytkownika

Do tej pory omówiona została jedna strona całego procesu – to, co dzieje się po stronie serwera generującego token. Pozostaje przyjrzeć się jeszcze procedurze obsługi tokena przez Web API, stanowiące punkt dostępu do danych interesujących użytkownika (np. Web API urządzeń bądź Web API telemetrii). Tym razem algorytm przedstawiony zostanie za pomocą schematu blokowego:



Ilustracja 4.16 Schemat blokowy procesu weryfikacji tożsamości użytkownika za pomocą tokena

Powyższy algorytm został zaimplementowany w opisanych już Web API platformy. Mimo że niniejszy rozdział dotyczy Web API tokenów, to dopiero tutaj podane zostaną pewne szczegóły implementacji algorytmu z ilustracji 4.16. Dzięki temu temat uwierzytelniania zamknięty zostanie w jednym rozdziale.

4.10.2 Obsługa tokenów przez główne Web API

Ze względu na fakt, że Web API urządzeń korzysta ze środowiska .NET, a Web API telemetrii z Node.JS, obsługa tokenów musiała być stworzona osobno dla każdego z rozwiązań.

4.10.2.1 Web API urządzeń

W przypadku aplikacji .NET, wykorzystana została biblioteka *Jose-JWT* [14]. Jest to projekt, który znacznie ułatwia zarówno tworzenie jak i weryfikację tokenów JWT. Zawiera

implementację wielu algorytmów szyfrowania i posiada bardzo wygodne API. Zaletą jest też czytelna i rzetelnie napisana doumentacja. Swoją drogą, ta sama biblioteka wykorzystywana jest przez Web API generujące tokeny.

ASP.NET posiada pewien określony łańcuch obsługi zapytań. Oznacza to, że każde zapytanie do Web API przechodzi przez określone procedury, zanim wysłana zostanie odpowiedź. Istnieje możliwość dodania dodatkowych modułów do wspomnianego łańcucha. Wymaga to stworzenia klasy dziedziczącej z *DelegatingHandler* i zarejestrowania jej:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.EnableCors();

        // Web API routes
        config.MapHttpAttributeRoutes();

        config.MessageHandlers.Add(new TokenValidator());

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{action}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

Kod źródłowy 4.18 Klasa WebApiConfig, która pozwala konfigurować Web API

Sama klasa do obsługi tokenów (*TokenValidator*) przedstawiona jest poniżej:

```
    public class TokenValidator : DelegatingHandler
    {
        public TokenValidator()
        {
            _certificateLoader = new LocalCertificateLoader();
        }

        private ICertificateLoader _certificateLoader;

        protected override Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)
        {
            System.Diagnostics.Debug.WriteLine("Token started");
            //PREFLIGHT CALL
            if (request.Method.Method == "OPTIONS")
                return base.SendAsync(request, cancellationToken);

            string token;
            if (!TryRetrieveToken(request, out token))
                return Task<HttpResponseMessage>.Factory.StartNew(() => new
HttpResponseMessage(HttpStatusCode.Unauthorized) { });

            var publicKey = _certificateLoader.LoadCertificate().PublicKey.Key
as RSACryptoServiceProvider;

            try
            {
                string jsonString = Jose.JWT.Decode(token, publicKey);
                var json = JObject.Parse(jsonString);
                var userId = json["sub"];
                request.Properties.Add("userId", userId);
            }
            catch (Exception e)
            {
                return Task<HttpResponseMessage>.Factory.StartNew(() => new
HttpResponseMessage(HttpStatusCode.Unauthorized) { });
            }

            return base.SendAsync(request, cancellationToken);
        }

        private static bool TryRetrieveToken(HttpRequestMessage request, out
string token)
    }
```

Kod źródłowy 4.19 Klasa TokenValidator do weryfikacji tokenów wewnątrz Web API urządzeń

W kodzie źródłowym 4.19, metoda *TryRetrieveToken()* została pominięta ze względu na zbyt wiele linii kodu – jej funkcja polega na odcztaniu tokena z z obiektu *HttpRequestMessage* (jeśli token istnieje).

Na uwagę zasługuje interfejs *ICertificateLoader*:

```
public interface ICertificateLoader
{
    X509Certificate2 LoadCertificate();
}
```

Kod źródłowy 4.20 Interfejs ICertificateLoader, którego implememtacja ładuje certyfikat

Jego obecność wynika z problematyczną sytuacją jaką napotkałem podczas umieszczania Web API Urządzeń wewnątrz usługi App Service platformy Azure. Okazało się, że niemożliwy jest odczyt certyfikatu z systemu plików. Taka strategia działała prawidłowo podczas uruchamiania programu na lokalnym komputerze, Azure wymaga jednak, aby certyfikat był wysłany na serwer poprzez specjalny panel ustawień usługi App Service. W związku z tym, proces ładowania certyfikatu jest inny podczas uruchamiania Web API w środowiskach lokalnym i chmurowym. Dzięki obecności interfejsu, można stworzyć jego różne implementacje (np. *LocalCertificateLoader* oraz *AzureCertificateLoader*).

4.10.2.2 Web API telemetrii

Aplikacja Node.JS (Web API telemetrii) również wykorzystuje zewnętrzną bibliotekę - *express-bearer-token*. Jest to tak naprawdę dodatek do frameworka *express*, który obsługuje zapytania. Wykorzystanie biblioteki sprowadza się do włączenia jej w łańcuch obsługi pojedynczego zapytania:

```
this.app.use(bearerToken());

this.app.use(async (req, res, next) => {
    let isTokenValid = await this.authHandler.verifyToken(req.token);
    if (!isTokenValid)
        res.status(401).send(
            "The request's token is not correct. Request dropped.");
    next();
});
```

Kod źródłowy 4.21 Włączenie express-bearer-token do łańcucha obsługi zapytań

Nie wystarczy to jednak do działania weryfikacji. W kodzie źródłowym powyżej (4.18) widać również wywołanie `this.authHandler.verifyToken(req.token)`. Stworzona została klasa *AuthHandler*, która definiuje metodę *verifyToken*:

```
class AuthHandler {
    constructor() {
        this.userId = null;
    }

    verifyToken(token) {
        let cert = fs.readFileSync('./assets/cert.pem');
        let self = this;
        return new Promise (function(resolve, reject) {
            jwt.verify(token, cert, { algorithms: ['RS256']},
                function(err, decoded) {
                if (decoded !== undefined) {
                    self.userId = decoded.sub;
                    resolve(true);
                }
                else
                    resolve(false);
              });
        });
    }

    getUserId() {
        return this.userId;
    }
}
```

Kod źródłowy 4.22 Klasa AuthHandler, która weryfikuje tożsamość użytkownika

Do poprawnego działania, wymagany jest oczywiście publiczny certyfikat, który zapisany jest w pliku *cert.pem*.

4.10.3 Wybór technologii

Tak jak w przypadku poprzednich Web API, należało wybrać środowisko, w którym zostanie ono zaimplementowane. Wybór padł na ASP .NET Core, ponownie, tak jak przy okazji tworzenia Web API telemetrii, w celu poznania tego środowiska. Bazuje ono na .NET Core, które jest tworzoną przez Microsoft kontynuacją „klasycznego" .NET Framework. Jego wielkim atutem jest multi-platformowość. Aplikacje .NET Core uruchomimy (jeśli pozwolą na to wykorzystane zależności) na systemach: macOS, Linux, Windows. Jest to ogromna

przewaga w porównaniu z .NET Framework. Co więcej, firma Microsoft obecnie intensywnie rozwija wersję Core swojego środowiska (obecna wersja to 2.1), wzbogacając je o kolejne nowości. W związku z tym, .NET Core zdobywa ostatnio ogromną popularność. Zalecane jest zresztą tworzenie nowych rozwiązań z jego wykorzystaniem [15], co wskazuje, że jest to środowisko, które jest przez Microsoft traktowane poważnie i spodziewać należy się sporych inwestycji w jego rozwój. W związku z tym, że ASP .NET Core jest bardzo podobne do ASP .NET 5 (wykorzystane w Web API Urządzeń), rozpoczęcie pracy w tej technologii było dla mnie znacznie łatwiejsze niż podczas korzystania z Node.JS.

Jak wspomniane zostało w poprzednim podrozdziale, jedną z kluczowych zależności aplikacji jest *Jose-JWT*, czyli biblioteka do prostej obsługi tokenów JWT. Poza tym aplikacja korzysta z, opisanego już, Entity Framework (w wersji Core – przeznaczonej dla .NET Core).

4.10.4 Punkt dostępowy

Opisywane Web API posiada tylko jeden punkt dostępowy, zdefiniowany jako metoda POST:

Przykład ciała POST:

```
{
    "Username": "someUser",
    "Password": "somePassword"
}
```

Przykładowe zapytanie:

https://{ADRES_IP}/api/token

Przykładowa odpowiedź:

```
200
(eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ
SI6Iktvc21hIGkgQmFzaWEiLCJpYXQiOjE1MTYyMzkwMjJ9.au2s1GG1sj2cd70uQz1d
5kFAblRW-5SK6zaRkDteH6o)
```

Klient otrzymuje token i może go wykorzystać do uzyskania żądanych informacji z innych Web API.

4.11 Aplikacja kliencka

Kolejny komponent umieszczony w chmurze, i zarazem ostatni, to referencyjna aplikacja dostępowa. Stanowi ona tak naprawdę pewnego rodzaju nakładkę na trzy serwisy Web API,

które zostały zaprezentowane w poprzednich rozdziałach. Dla większości użytkowników tego typu rozwiązań jest to jednak jedyny interfejs dostępowy, z którego chcą bądź potrafią korzystać. Z tego powodu, budowa aplikacji klienckiej jest równie ważna jak budowa samej platformy. Aplikacja kliencka nie jest częścią platformy, jest jedynie wspomnianą „nakładką", bez której nadal możliwe byłoby wykonywanie wszystkich możliwych operacji. Nic nie stoi na przeszkodzie, aby klient stworzył swoją własną warstwę dostępową do platformy, jeśli ma taką ochotę. Prawdopodobnie miałoby to sens w wielu przypadkach, przede wszystkim, kiedy rozwiązanie byłoby wykorzystywane dla pewnej specyficznej grupy urządzeń. Aplikacja stworzona na potrzeby niniejszej pracy ma charakter jak najbardziej ogólny, tzn. ma umożliwić proste zarządzanie urządzeniami, nie bierze pod uwagę tego, czym te urządzenia tak naprawdę są. Kwestia ta zostanie wyjaśniona w dalszej części rozdziału.

4.11.1 Wybór technologii

Zgodnie z założeniami projektu, dostęp do MJIoT ma być możliwy z poziomu zarówno komputera jak i smartfona. Obecnie dostępna jest spora ilość różnych rozwiązań, które pozwalają tworzyć aplikacje z interfejsem graficznym. Końcowe aplikacje podzielić na dwie główne kategorie:

- działające na jednej platformie,
- uniwersalne/działające na wielu platformach.

Pierwsza z kategorii wymagałaby stworzenia co najmniej dwóch aplikacji, aby spełnić założenia projektowe (jedna aplikacja mobilna oraz jedna dla komputera). W związku z tym znacznie bardziej interesująca wydaje się być druga opcja. Tworzenie jednej aplikacji, działającej na wielu urządzeniach/platformach przyspiesza pracę, nie jest jednak pozbawione wad. Tworząc rozwiązanie uniwersalne, czynimy to kosztem m. in.: wydajności, dostępnych funkcjonalności.

Przykłady rozwiązań należących do pierwszej kategorii:

- natywna aplikacja iOS napisana w języku Swift lub Objective-C,
- natywna aplikacja Android napisana w języku Kotlin lub Java,
- natywna aplikacja WPF napisana w jezyku C# (system Windows).

Przykłady rozwiązań należących do drugiej kategorii:

- Xamarin – aplikacja dla Windows, iOS, Android napisana w języku C#,

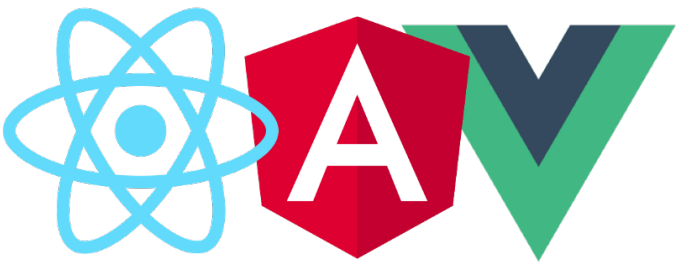
- QT – aplikacje C++ dla wszystkich głównych platform,
- rozwiązania webowe – aplikacja na wszystkie główne platformy, stworzona w oparciu o HTML, CSS oraz JavaScript.

Spośród przedstawionych rozwiązań, całkowicie odrzuciłem pierwszą kategorię ze względu na wspomnianą konieczność tworzenia oddzielnych rozwiązań dla poszczególnych systemów. Chcąc pokryć systemy: macOS, Windows, Linux, iOS, Android, należałoby stworzyć pięć aplikacji!

Pozostała druga kategoria. Spośród dostępnych rozwiązań wybrałem ostatecznie aplikację webową. Xamarin został przeze mnie odrzucony ze względu na negatywne doświadczenia związane z jego wykorzystaniem w przeszłości. QT natomiast wymaga programowania w C++ - stworzenie solidnej aplikacji wymaga doświadczenia w wykorzystaniu tego języka, którego nie posiadam.

Sam wybór – aplikacja webowa – nie rozwiązuje jeszcze problemu. Kolejnym krokiem jest wybór konkretnego sposobu tworzenia projektu. Oto niektóre z dostępnych możliwości:

- wykorzystanie „czystego" języka JavaScript,
- wykorzystanie React – frameworka opracowanego przez Facebook, Inc.,
- wykorzystanie Angular – rozwiązania opracowanego przez Google Inc.,
- wykorzystanie Vue.JS – frameworka opracowanego przez społeczność.



Ilustracja 4.17 Loga: React, Angular, Vue.JS

Poszczególne opcje charakteryzują się zbiorem różnych zalet oraz wad, jednak wyróżnia się tu przede wszystkim pierwszy z punktów, który został przeze mnie od razu odrzucony – ze względu na trudność utrzymania porządku w większym programie, który korzysta jedynie z JavaScript'u. Trzy wymienione frameworki powstały w celu wprowadzenia pewnego „ładu" do programu, co zdecydowanie jest istotną kwestią.

Sam wybór framework'a nie jest jednak najważniejszy. Warto wytłumaczyć raczej, co oznacza wspomniany „ład". Najważniejsze, co wprowadzają Angular, React, czy Vue, to MVC (ang. *Model-View-Controller*), czyli sposób tworzenia aplikacji, który rozdziela odpowiedzialności na oddzielne elementy:

- Model – są to dane, które chcemy wyświetlić,
- View – jest to sposób wyświetlenia danych,
- Controller – jest to logika, która operuje na danych.

Przykładowo, chcąc wyświetlić dane nt. urządzeń użytkownika, poszczególne odpowiedzialności możemy zdefiniować następująco:

- Model – klasa, która reprezentuje zbiór informacji nt. pojedynczego urządzenia,
- View – struktura HTML, która odpowiada za wyświetlenie danych nt. urządzenia (np. tabela bądź lista),
- Controller – program, który pobiera dane nt. urządzeń z serwera i tworzy na ich postawie instancje odpowiednich klas (tworzy modele).

Tworząc program za pomocą „czystego" JavaScriptu, bardzo trudno oddzielić te odpowiedzialności. W tym celu należałoby stworzyć dosyć zaawansowaną warstwę abstrakcji, która wprowadziłaby paradygmat MVC – w praktyce, powstałby prawdopodobnie nowy framework! Zamiast więc wymyślać koło od nowa, można skorzystać z istniejącego i, co istotne, dobrze przetestowanego rozwiązania.

4.11.1.1 Angular

Do tworzenia swojej aplikacji, wybrałem rozwiązanie proponowane przez Google – Angular. Spośród wymienionych framework'ów, wyróżnia się on przede wszystkim:

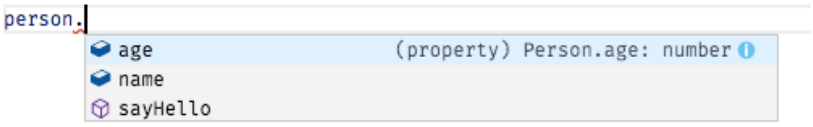
- domyślnym wsparciem dla języka TypeScript (opisanym w kolejnym podrozdziale),
- dostępnością Angular CLI – konsolowego narzędzia przyspieszającego tworzenie poszczególnych elementów aplikacji,
- bogatym zestawem modułów, które zapewniają szereg funkcjonalności (np. komunikacja HTTP(S), tworzenie i weryfikacja formularzy).

Budowa aplikacji jest oparta o znany również z innych rozwiązań sposób – definiowanie komponentów i umieszczanie ich w widoku. Przykład zastosowania komponentów zostanie

przedstawiony na podstawie aplikacji będącej przedmiotem tego rozdziału, w dalszej części pracy.

4.11.1.2 TypeScript

Na uwagę zasługuje język TypeScript, który został wykorzystany w projekcie. Najprościej język ten można opisać jako rozszerzenie JavaScript'u o możliwość określania typów. Wprowadza to porządek do programu i upodabnia go do innych, silnie typowanych języków, np. C#, Java czy C++. Dzięki statycznemu typowaniu, programista może uniknąć ewentualnych błędów w aplikacji, trzymając się zadeklarowanego typu. Nie można jednoznacznie stwierdzić, że taki sposób programowania jest lepszy od typowania dynamicznego, ponieważ zależy to od preferencji programisty i można w tym względzie znaleźć wiele opinii. Jedną z zalet takiego podejścia jest możliwość wyświetlenia podpowiedzi w trakcie pisania kodu w edytorze, który wspiera taką możliwość (np. IntelliSense wewnątrz Visual Studio). Dzięki znajomości typu, edytor jest w stanie zaproponować odpowiednie właściwości oraz metody:



Kod źródłowy 4.23 Przykład działania funkcji IntelliSense edytora Visual Studio Code

Przykład stosowania typów w języku TypeScript:

```
let data: string = "Zmienna zadeklarowana jako string";
let someNumber: number = 7; //zmienna zadeklarowana jako liczba

class Person {
    name: string;
    age: number;

    sayHello(): string {
        return `Hello. I am ${name}`;
    }
}

let person: Person = new Person(); //instancja klasy Person
```

Kod źródłowy 4.24 Przykład stosowania typów w języku TypeScript

Należy zaznaczyć, że typowanie jest opcjonalne. W razie nieprzestrzegania typów, zgłoszone zostanie jedynie ostrzeżenie, program nadal zostanie przetłumaczony na JavaScript.

Tłumaczenie (transpilacja) do języka JavaScript to sposób, w jaki zapewniono działanie programów pisanych w nowym języku w przeglądarkach. Nie istnieje przeglądarka, która bezpośrednio interpretuje kod TypeScript, nie robi tego nawet Edge (przeglądarka firmy Microsoft, która tworzy również język TypeScript). Przykład transpilacji kodu 4.23:

```
var data = "Zmienna zadeklarowana jako string";
var someNumber = 7; //zmienna zadeklarowana jako liczba
var Person = /** @class */ (function () {
    function Person() {
    }
    Person.prototype.sayHello = function () {
        return "Hello. I am " + name;
    };
    return Person;
}());
var person = new Person(); //instancja klasy Person
```

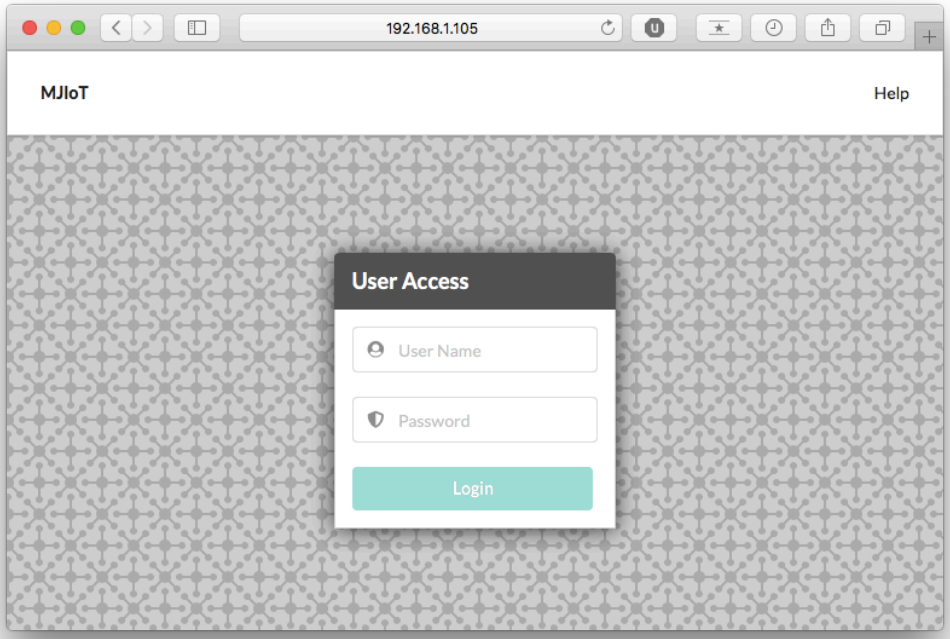
Kod źródłowy 4.25 Kod JavaScript będący rezultatem transpilacji kodu TypeScript 4.24

Inną zaletą TypeScript jest wsparcie dla najnowszych standardów JavaScript, które po transpilacji są tłumaczone na kod zrozumiały dla przeglądarek (twórcy silników JavaScript dodają wsparcie dla nowych wersji języka z dużym opóźnieniem, dlatego wskazane jest korzystanie ze składni w wersji ES5).

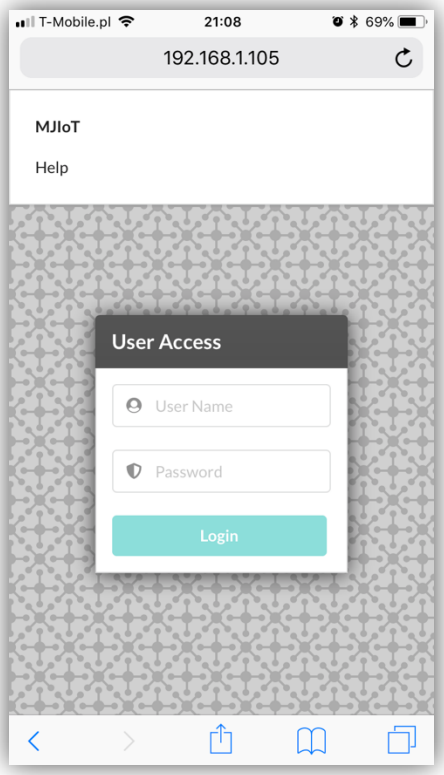
4.11.2 Zrzuty ekranu

Ze względu na fakt, że tworzone rozwiązanie to aplikacja o interfejsie graficznym, należy zaprezentować, jak przedstawia się ona dla potencjalnego użytkownika. Poniżej wstawione są dostępne ekrany aplikacji, przedstawione w dwóch wersjach:

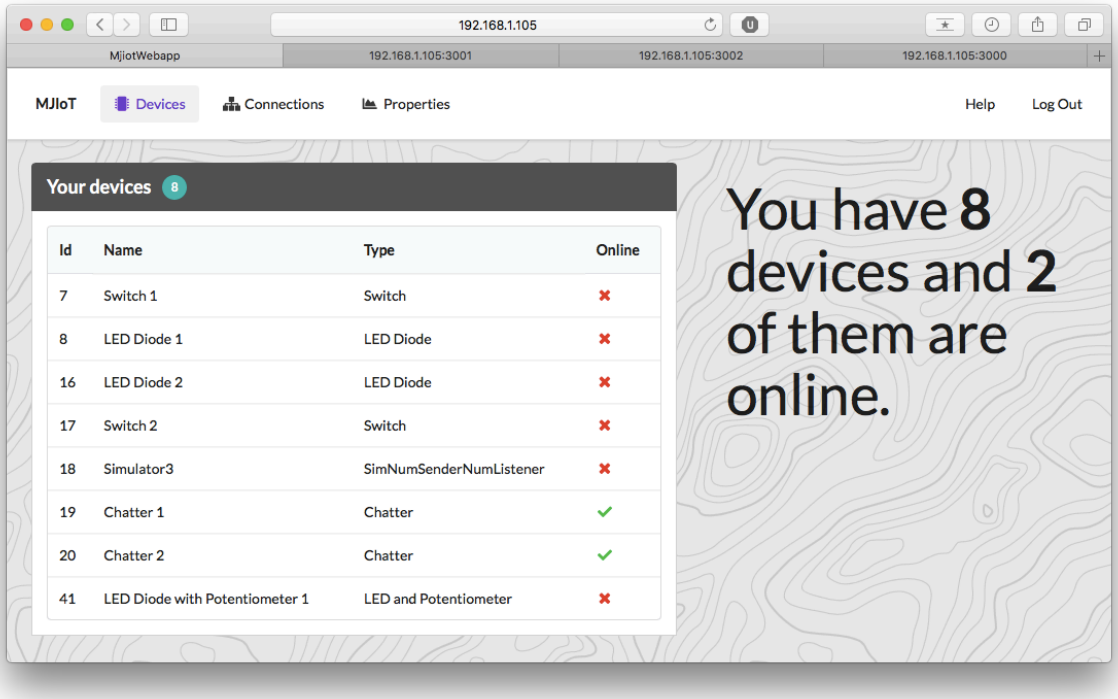
- wersja dla dużych ekranów,
- wersja mobilna.



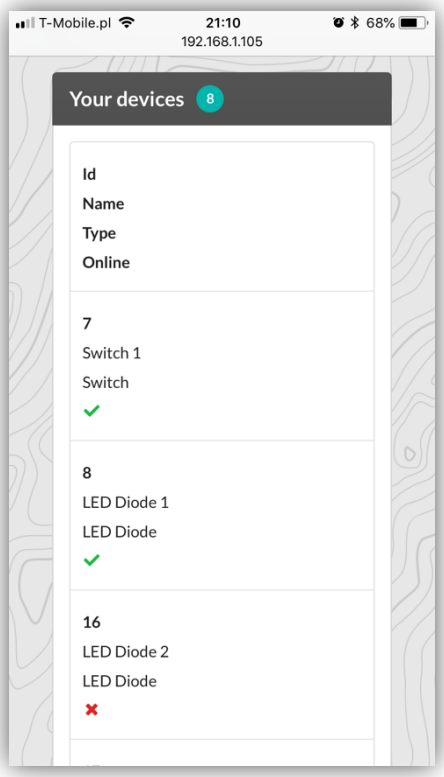
Ilustracja 4.18 Widok logowania – ekran komputera



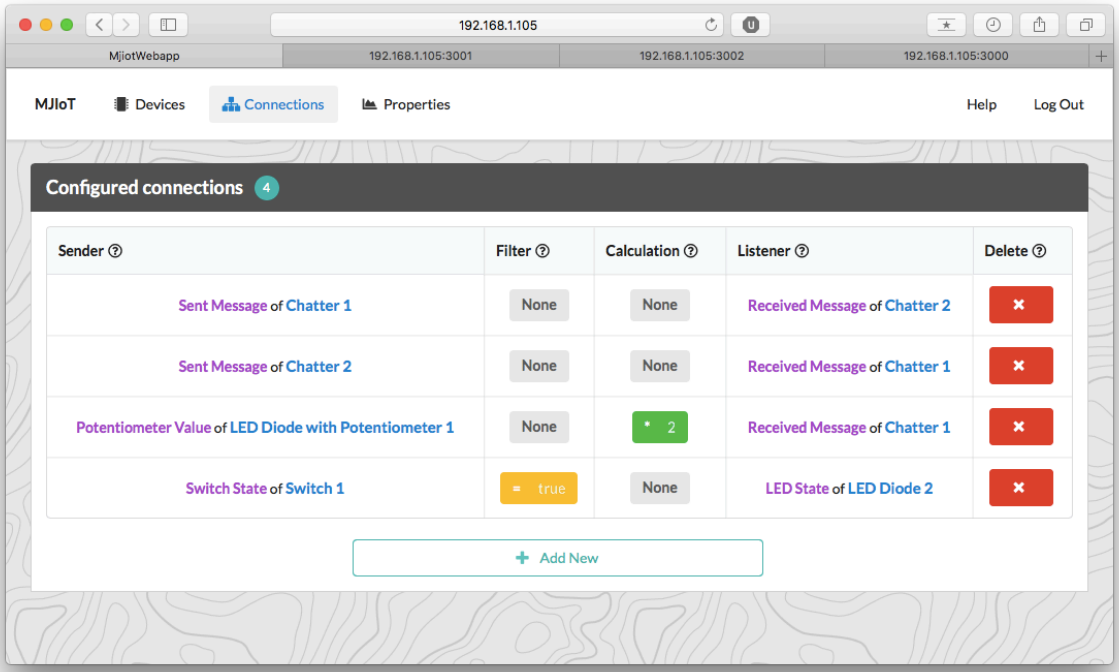
Ilustracja 4.19 Widok logowania - ekran smartfona



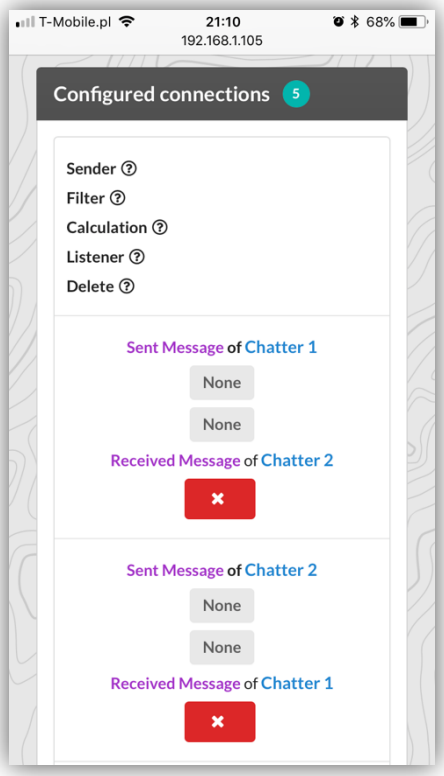
Ilustracja 4.20 Widok urządzeń - ekran komputera



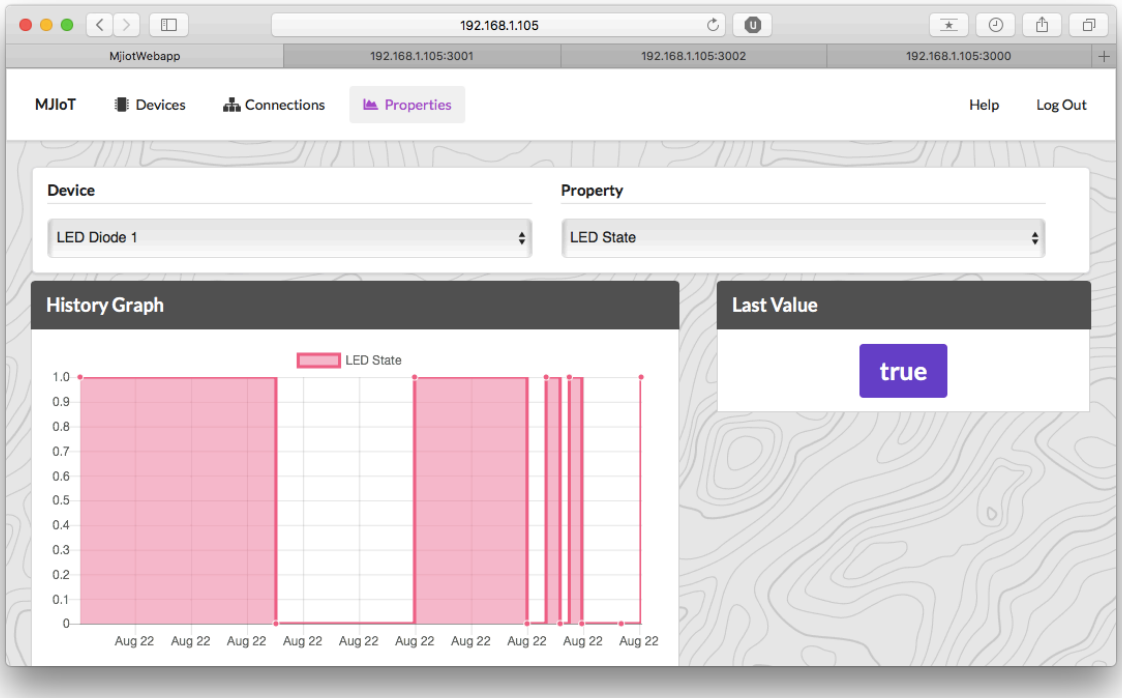
Ilustracja 4.21 Widok urządzeń - ekran smartfona



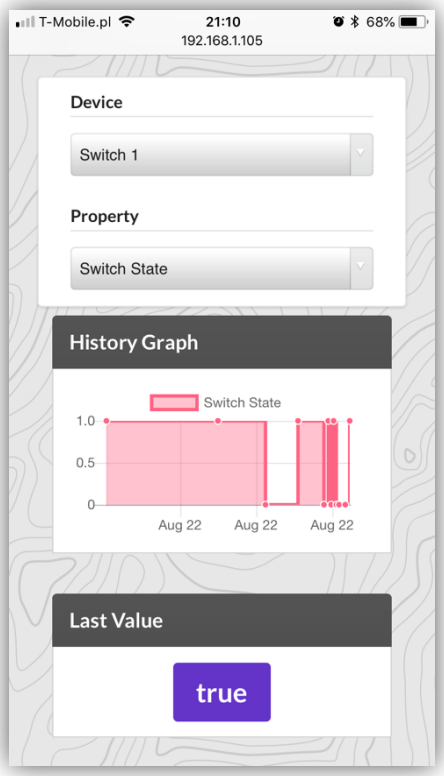
Ilustracja 4.22 Lista połączeń - ekran komputera



Ilustracja 4.23 Lista połączeń - ekran smartfona



Ilustracja 4.24 Widok telemetrii - ekran komputera



Ilustracja 4.25 Widok telemetrii - ekran smartfona

4.11.3 Dostępne funkcje

Przedstawione zrzuty ekranu (ilustracje 4.18 – 4.25) zawierają wszystkie dostępne możliwości jakie oferuje projekt. Wersja dla komputerów oraz smartfonów posiada ten sam zbiór funkcjonalności, gdyż jest to ta sama aplikacja, która responsywnie dopasowuje się do rozmiaru ekranu docelowego.

4.11.3.1 Ekran logowania

Ekran logowania (4.18 oraz 4.19) to prosty formularz, który pozwala wpisać użytkownikowi dane logowania.

5 Bibliografia

[1] https://iot.ieee.org/images/files/pdf/IEEE_IoT_Towards_Definition_Internet_of_Things_Revision1_27MAY15.pdf

[2] "The Internet of Things: In a Connected World of Smart Objects" (Accenture & Bankinter Foundation of Innovation, 2011)

[3] HP Security, Miessler, 2014

[4] https://aws.amazon.com/what-is-cloud-computing/

[5] https://www.forbes.com/sites/bobevans1/2018/04/09/microsoft-amazon-and-ibm-which-cloud-powerhouse-will-top-q1-revenue-charts/#2872148814dc

[6] https://azure.microsoft.com/en-us/support/legal/sla/summary/

[7] https://docs.microsoft.com/en-us/azure/sql-database/sql-database-elastic-pool

[8] https://docs.microsoft.com/en-us/azure/cosmos-db/sql-api-resources

[9] https://martinfowler.com/articles/serverless.html

[10] https://martinfowler.com/bliki/CommandQuerySeparation.html

[11] https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/

[12] "Architectural Styles and the Design of Network-based Software Architectures" by Roy Thomas Fielding

[13] https://www.iso.org/iso-8601-date-and-time-format.html

[14] https://github.com/dvsekhvalnov/jose-jwt

[15] https://www.microsoft.com/net/learn/what-is-dotnet