Warsztaty IoT w chmurze obliczeniowej z Globallogic

Jacek Choroszy

14 grudnia 2022

Spis treści

1 Wprowadzenie

Celem tego warsztatu jest zaprezentowanie, jak może wyglądać prosty– ale i obszerny– fragment modelu rozwiązania w chmurze obliczeniowej IoT na poziomie zbliżonym do warunków komercyjnych. Warsztat został tak zaprojektowany, aby poruszyć wszystkie kluczowe elementy obszaru prawdziwego świata IoT, chmury obliczeniowej, DevOps i projektowania dobrego oprogramowania. Zatem partycypant zostanie poprowadzony przez wszystkie główne etapy projektu rozwiązania, programowania, tworzenia infrastruktury, wdrażania i wreszcie– testowania. Etapy te są nieodłączne w rzemiośle inżynierii oprogramowania.

Dlaczego warto Partycypanci po ukończeniu niniejszego warsztatu poza ogólną satysfakcją ukończenia kreatywnego, działającego dzieła i towarzyszącej temu dopaminie, pozyskają *bezcenne* umiejętności samodzielnego wytwarzania oprogramowania od zera, do gotowego rozwiązania, co pozwoli im także odpowiednio nakierować ich karierę, nabyć dojrzałego wejrzenia, oraz pojąć dogłębnie rzemiosło wytwarzania oprogramowania.

Niestety, ale w warunkach komercyjnych, zwłaszcza na poziomie czeladnika, możliwości poprowadzenia projektu od A do Z wraz z elementami DevOps są niewielkie, jeżeli niemożliwe. Ten warsztat to okazja do „wejścia w buty" architekta i leada technicznego, a także– mam taką nadzieję– zachęci do własnego eksperymentowania. Bo właśnie podczas eksperymentów w domu, wykuwa się prawdziwy mistrz (i po wielu latach nauki w rzemiośle „komercyjnym").

Stos techniczny Do niniejszego przedsięwzięcia wykorzystano następujące technologie i narzędzia:

- Java 11
- Gradle v7
- Terraform
- Git
- AWS CLI

Jak widać, niewiele trzeba, aby dostarczyć funkcjonujące rozwiązanie na skalę świata biznesu.

Biblioteki użyte do oprogramowania paczek zostały pominięte, albowiem część z nich będzie opisana w dalszej części instrukcji.

Chmura Jako dostawcę usług chmury obliczeniowej wybrano AWS. Proszę się w przyszłości nie sugerować tym wyborem podczas projektowania własnych (lub dla swoich przyszłych klientów) rozwiązań. Dobrze jest nie uzależniać się od konkretnego dostawcy ze względu na brak gwarancji ceny usług. Lepiej jest

zawsze mieć możliwość zmiany, np. w celu optymalizacji kosztów. Naturalnie, rozwiązania natywne zawsze będą tańsze (w perspektywie bieżącego użycia), lecz długoterminowo koszt może być słony– tranzycja na innego dostawcę z natywnego rozwiązania wymaga praktycznie przebudowę systemu od zera.

Jednym z kroków wprowadzających warstwę abstrakcji do definiowania jest narzędzie Terraform, pozwalając na agnostyczne zamodelowanie dowolnej infrastruktury.

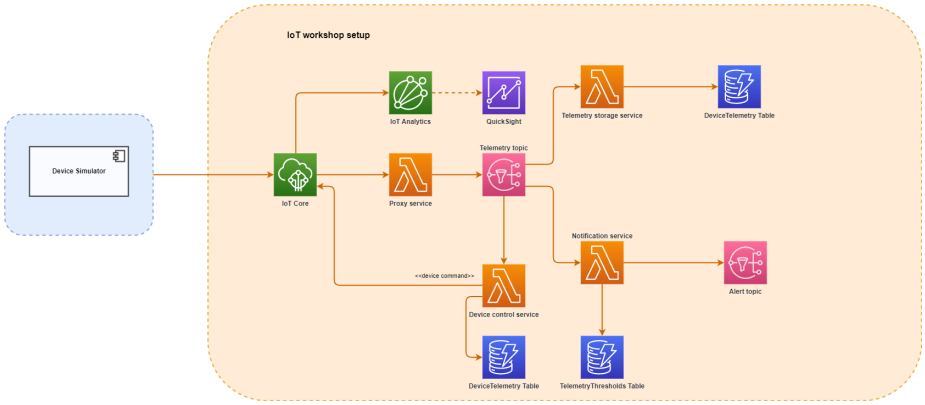
Dobrym sposobem jest odseparowanie infrastruktury od warstwy programowej (*software*)– oprogramowanie nie powinno koncentrować się gdzie działa, ale na realizacji procesów biznesowych. Zmiana infrastruktury, np. AWS na Azure, powinna wyłącznie objąć obszar samej infrastruktury– kod powinien zostać bez zmian (w idealnym założeniu).

W zastosowanym rozwiązaniu naturalnie zastosowano zdrowy kompromis, albowiem nie zawsze powyższe pryncypium ma rację bytu. Tutaj na przykład, celem jest pokazanie poszczególnych obszarów usług AWSa, co zakceleruje rozwój oprogramowania w tej technologii, jak i zaznajomi z pewnymi uniwersalnymi cechami, które także występują u innych dostawców. Ponadto, niestety część usług AWS może działać za pośrednictwem bibliotek opublikowanych przez AWS.

Co będzie zrobione Niniejszy warsztat opiewa prosty przypadek biznesowy dla urządzenia IoT udającego termostat z czujnikiem temperatury. W regularnych odstępach czasu wysyła dane telemetryczne, zawierające m.in. odczyt temperatury w formie surowej (nieczytelnej dla człowieka).

Chmura ma za zadanie przechować te dane– w celu późniejszej analizy– czy temperatura nie przekroczyła wartości krytycznej– wówczas wysyłana jest wiadomość do użytkownika z ostrzeżeniem– oraz sterować stanem klimatyzacji, by pomieszczenie osiągnęło temperaturę docelową.

Architektura zwizualizowana jest na rys. 1.



Rysunek 1: Ogląd infrastruktury

2 Konfiguracja środowiska pracy

Przed przystąpieniem do realizacji serwisów *IoT*, należy przygotować środowisko pracy (*warsztat*). Każda grupa otrzymała region, w którym będzie wdrożona infrastruktura, repozytorium oraz dane do logowania do konsoli AWS.

Formalizując, należy mieć następujące dane:

- Adres i dostęp do swojego repozytorium
- Dane do logowania na konto AWS
- Klucze do konta AWS (*access key* i *secret access key*)
- kod regionu, gdzie będzie wdrożona infrastruktura

2.1 Kroki do realizacji

1. Najpierw zajmiemy się miejscem, gdzie będziemy publikować swoją pracę.
2. Proszę utworzyć repozytorium o dostępie *Private*, i o nazwie *iot-workshop-grupa-<nr-grupy>*.
3. Utwórz folder, gdzie będzie składowana praca związana z warsztatami.

   ```
   mkdir ~/iot-workshop
   ```

   Tam będą przechowywane wszystkie moduły naszego ekosystemu, od kodu serwisów po infrastrukturę.
4. Będąc w folderze *~/iot-workshop*, pobierz swoje repozytorium.

   ```
   git clone <adres_repo_swojej_grupy>
   ```

5. W ramach uproszczenia, pobierzemy także repozytorium z materiałami do warsztatów.

   Uwaga! Proszę pobrać repozytorium do folderu *~/iot-workshop Adres HTTPS*:

   ```
   https://github.com/jacek-choroszy-gl/master-iot-workshop.git
   ```

   SSH

   ```
   git@github.com:jacek-choroszy-gl/master-iot-workshop.git
   ```

   W następnych krokach, jeżeli będzie wzmianka o sklonowaniu repozytorium do danego modułu i zrobieniu *forka*, proszę po prostu skopiować folder z nazwą modułu do przestrzeni swojego repozytorium.

Uwaga! Proszę jednak czynić to tylko wtedy, kiedy wskazuje na to instrukcja.

6. Aby móc pracować z AWSem, należy skonfigurować zmienne środowiskowe, gdzie przechowywane będą *access key* i *secret access key*. *AWS CLI* posługuje się nimi do wykonywania operacji w imieniu właściciela konta.

   Dodaj następujące zmienne środowiskowe: *AWS_ACCESS_KEY_ID* i *AWS_SECRET_ACCSESS_KEY* wprowadzając jako wartość dane przydzielone grupie.

   Przykładowa inwokacja w konsoli:

   ```
   export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
   export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI
   ```

7. W konsoli wpisujemy

   ```
   aws configure
   ```

   To spowoduje sczytanie danych przez *AWS CLI*, co umożliwi komunikację programatyczną z usługami AWSa. Po wpisaniu komendy, kolejno będą pojawiać się wartości konfiguracyjne. Te, które nie są skonfigurowane, należy uzupełnić. Jako domyślny region, wprowadzamy ten, który został przydzielony grupie. Jako *output*, wpisz *json*.

   Te dwie ostatnie opcje nie mają wpływu na dalszy przebieg warsztatów, albowiem później można arbitralnie ustalić wybrany region wdrożenia infrastruktury.

2.2 Oczekiwany rezultat

Po zrealizowaniu tej sekcji, powinny być osiągnięte następujące warunki:

- Narzędzia takie jak: git, Gradle, AWS CLI i Terraform działają
- Dodano do zmiennych środowiskowych klucze do komunikacji z usługami AWS
- Skonfigurowano AWS CLI do pracy.
- Pobrano repozytorium swojej grupy do folderu *~/iot-workshop*
- Pobrano repozytorium warsztatów do folderu *~/iot-workshop*

Spełnienie wszystkich punktów jest konieczne do sukcesywnego wykonania dalszych etapów warsztatu!

2.3 Do przemyślenia

1. Jak nazywa się *CLI* do zarządzania usługami w *Azure*?

3 Aprowizacja

Kluczowym obszarem bezpieczeństwa w obszarze IoT jest uwierzytelnienie „aktorów" biorących udział w komunikacji z infrastrukturą, w tym przypadku urządzeń. Każde urządzenie musi być rozpoznane przez system w chmurze– jest realizowane to przez serwis typu „Authoriser".

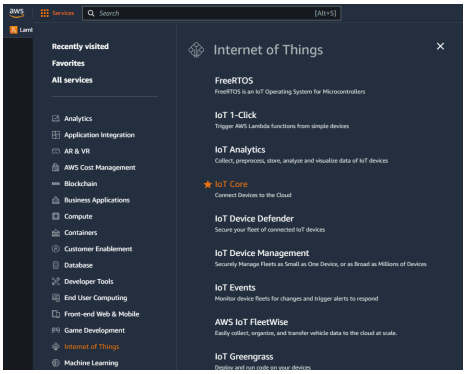
Jednym z klasycznych sposobów na uwierzytelnienie w świecie IoT jest posiadanie certyfikatu i klucza. Te są generowane po stronie chmury i wgrywane w pamięć urządzenia podczas produkcji. Proces przydziału urządzenia (jego metadanych i unikatowego certyfikatu) nazywany jest aprowizacją (*ang* „provisioning").

W tej sekcji wygenerowany zostanie zestaw certyfikatów wraz z kluczami do umożliwienia komunikacji z infrastrukturą IoT w AWSie. Serwis realizujący funkcję tzw. „Authorisera" jest wbudowany w *IoT Core*.

3.1 Kroki do zrealizowania

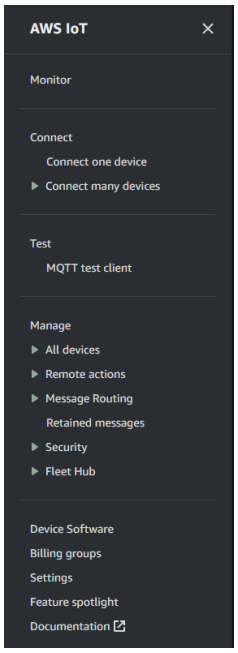
1. Zaloguj się na konto AWS
2. Zlokalizuj w lewym górnym rogu ekranu wybór *Services* i z listy wybierz dział *Internet of Things*.
3. Po prawej stronie pojawi się lista usług z tej dziedziny. Wybierz *IoT Core*.

   Alternatywnie w pasku *Search* wpisz „IoT Core" i kliknij w znaleziony wynik.



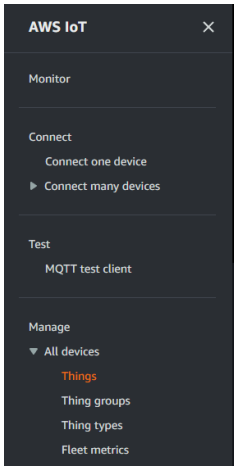
Rysunek 2

Po lewej stronie znajduje się menu otwartej usługi (jeżeli go nie ma, należy wówczas kliknąć w trzy poziome paski lewym górnym rogu ekranu).



Rysunek 3

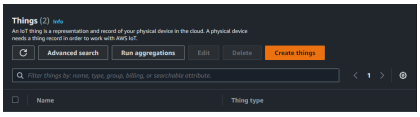
4. W sekcji *Manage* wybierz *All devices* -> *Things*



Rysunek 4

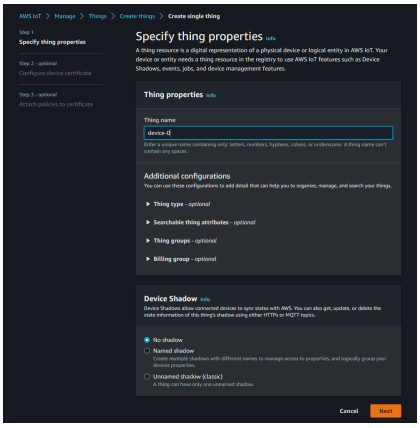
Thing w nomenklaturze AWSa jest wirtualnym ambasadorem konkretnego urządzenia. W dalszych krokach utworzony zostanie *thing* dla naszego urządzenia.

5. kliknij w *Create things*. Jako że interesuje nas aprowizacja jednego urządzenia, wybierz opcję *Create single thing*, a potem *Next*.



Rysunek 5

6. W tym kroku wystarczy tylko podać unikatową nazwę urządzenia. Niech będzie to *ID* urządzenia, np. „device-0”. Proszę wprowadzić jakąś nazwę. Będzie ona wykorzystywana do końca tych warsztatów.

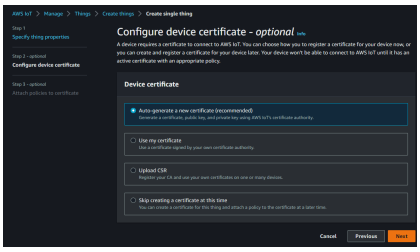


Rysunek 6

W sekcji „Additional configurations” można wprowadzić dodatkowe ustawienia pozwalające na grupowanie urządzeń wg. typu i tagów. Ułatwia to w zarządzaniu flotą. To można pominąć.

Uwaga! W sekcji *Device Shadow* proszę ustawić opcję na „No shadow”. Po wpisaniu nazwy urządzenia można nacisnąć *Next*.

7. Zaznaczamy *Auto-generate a new certificate (recommended)*. AWS zajmie się utworzeniem bezpiecznych certyfikatów.



Rysunek 7

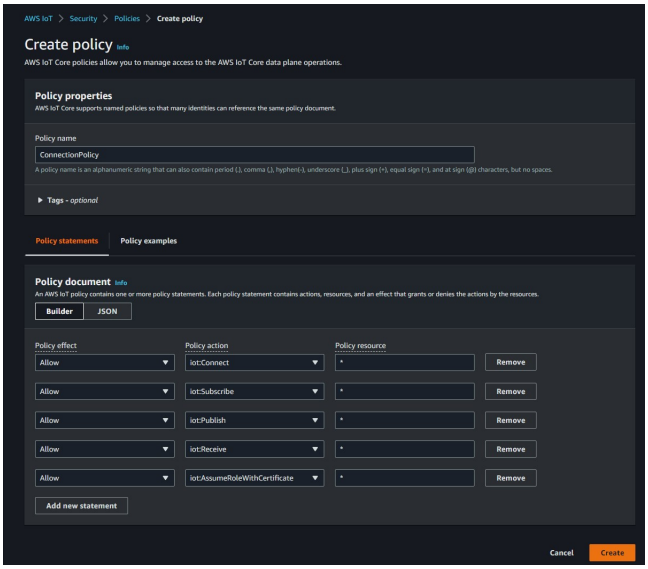
8. W tym kroku utworzone zostaną polisy dostępu, tzw. „policies”.

   Polisa może być powiązana z bytem (zasobem) w infrastrukturze AWSa i pozwala na kontrolę jego dostępu do poszczególnych obszarów serwisu. Dzięki temu rozwiązaniu, można powiązać jedną polisą z wieloma bytami i zarządzać nimi.

   Zalecane jest nadawania minimalnych dostępów do każdego zasobu, tj. tylko na te działania, które są konieczne do zrealizowania przeznaczonego mu celu. Co więcej, konieczną praktyką jest definiowanie uprawnień dla konkretnych zasobów (lub grupy)– należy unikać liberalnego *.

   W celu utworzenia polisy, należy nacisnąć na *Create policy*. Spowoduje to otwarcie nowej zakładki w przeglądarce.

   Polisę proszę wygenerować jak na zamieszczonym rysunku. Dla uproszczenia, godzimy się na liberalny dostęp *.

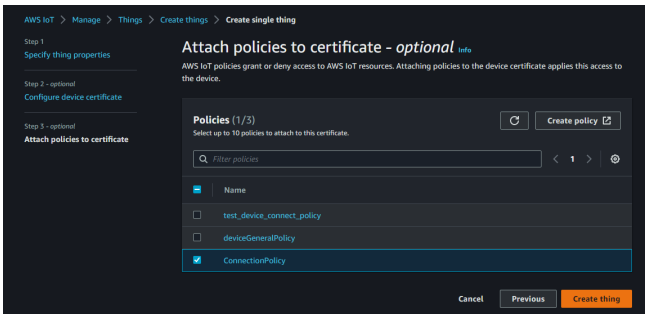


Rysunek 8

Po zakończeniu, klikamy w *Next*.

9. Wygenerowana polisa jest widoczna w panelu *Security/Policies*. Zakładkę przeglądarki można zamknąć, w celu powrotu do kreatora *Thing*.

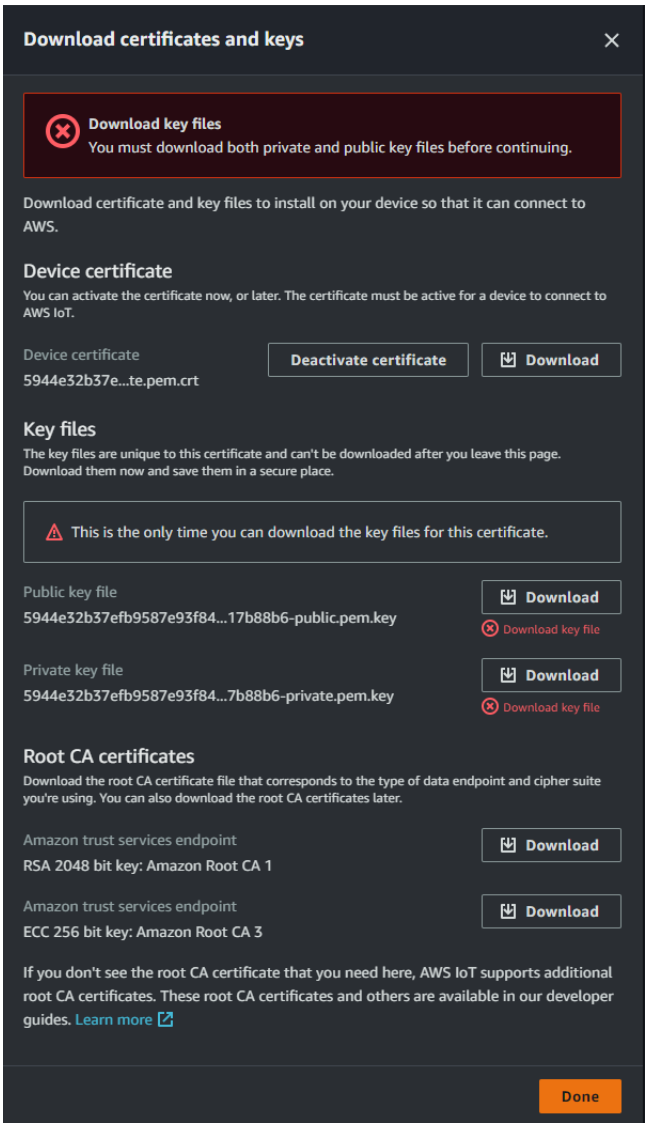
10. Można zauważyć, że wygenerowana polisa jest widoczna na liście.



Rysunek 9

W celu finalizacji procesu tworzenia *rzeczy*, naciskamy na *Create Thing*.

11. Otworzy się okno z certyfikatami i kluczami. Proszę jeszcze ich nie pobierać.



Rysunek 10

12. W celu utworzenia folderu na certyfikaty, wpisz komendę

```
mkdir ~/certs
```

13. Certyfikaty pobierz do utworzonego folderu, zachowując następujące nazewnictwo:

Nazwy certyfikatów	
Plik	Ścieżka
Klucz prywatny	~/certs/private.pem.key
Certyfikat urządzenia	~/certs/device.pem.crt
Certyfikat CA	~/certs/AmazonRootCA1.pem

3.2 Oczekiwany rezultat

Po zrealizowaniu tej sekcji, powinny być osiągnięte następujące warunki:

- Utworzona *Rzecz* w *IoT Core*
- Wygenerowana polisa dostępu i podpięta do zaprowizowanej *Rzeczy*
- Wygenerowane certyfikaty dla urządzenia i podpięte do jego wirtualnej reprezentacji
- Certyfikaty pobrane do folderu *~/certs*
- Umiejętność nawigacji po *IoT Core*

Spełnienie wszystkich punktów jest konieczne do sukcesywnego wykonania dalszych etapów warsztatu!

3.3 Do przemyślenia

1. Co to jest *thing* i jakie kryje się za nim założenie?
2. Jak nazywa się paralelne rozwiązanie po stronie chmury Azure?
3. czym charakteryzują się certyfikaty?

4 Konfiguracja urządzenia

W tym rozdziale przygotujemy nasze urządzenie, które zaprowizowaliśmy w rozdziale poprzednim, do połączenia z ekosystemem w chmurze AWS.

Dla ułatwienia, pominięto krok wdrażania kodu na prawdziwym urządzeniu, jednakże, zastosowane tutaj metody są uniwersalne– w celu komunikacji z chmurą, urządzenie to musi posiadać klienta *MQTT*– wybrany protokół komunikacji. Proszę zwrócić uwagę, że jest to arbitralny wybór projektowy, równie dobrze protokołem komunikacji może być *TCP* lub *HTTP*.

Chmura AWS daje dużą dowolność co do sposobu aprowizacji urządzenia, jak i komunikacji. W tym przypadku wybrano *IoT Core* wraz z wbudowanym brokerem *MQTT*. Implementacja klienta *MQTT* w celu połączenia i dwukierunkowej komunikacji została wykonana przy użyciu biblioteki *aws-iot-device-sdk*. Jest ona dystrybuowana przez AWS, więc nie wymaga dodatkowej konfiguracji przy wyborze certyfikatu i klucza jako metody uwierzetelnienia. Należy zwrócić uwagę, że wybór biblioteki może być dowolny.

Uwaga na boku *W przypadku fizycznego urządzenia, wystarczyłoby pobrać SDK od AWS do wybranej platformy, np. C++ lub Python. W SDK zawarty jest gotowy stos komunikacyjny.*

4.1 Kroki do realizacji

1. Najpierw pobierzemy kod źródłowy symulujący urządzenie. Przejdź do folderu głównego swojego repozytorium: *~/iot-workshop*

2. Będąc w folderze *~/iot-workshop*, pobierz repozytorium *device-simulator*

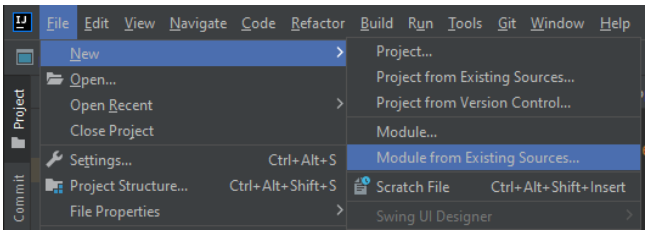
   ```
   git clone
       https://github.com/jacek-choroszy-gl/master-iot-workshop.git
   ```

   Jako jest to zewnętrzne repozytorium, i nie chcemy mieć konfliktu podczas swoich commitów, musimy je sobie *przywłaszczyć*, czyli formalnie, uczynić tzw. *forka*. Wpisz następującą komendę:

   ```
   git remote set-url origin <adres_repo_grupy>
   git push --mirror
   ```

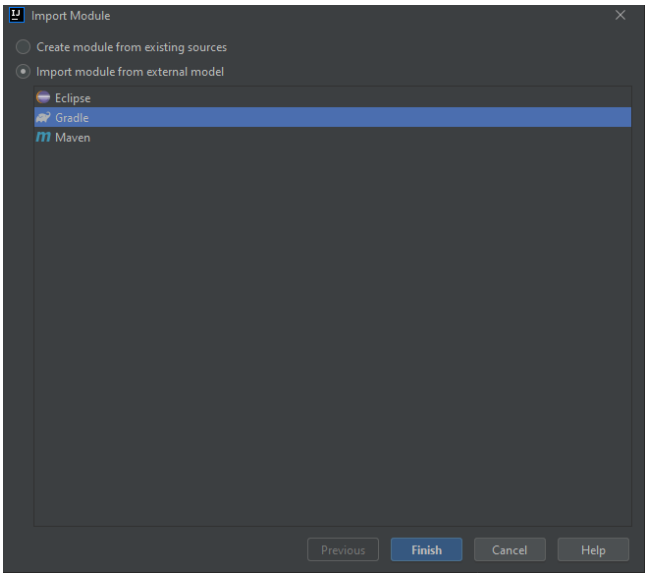
3. Otwórz środowisko programistyczne (tzw. *IDE*) *InteliJ*. Jest to program graficzny do pracy w środowisku Java.

4. *File -> New -> Module from Existing Source...*



Rysunek 11

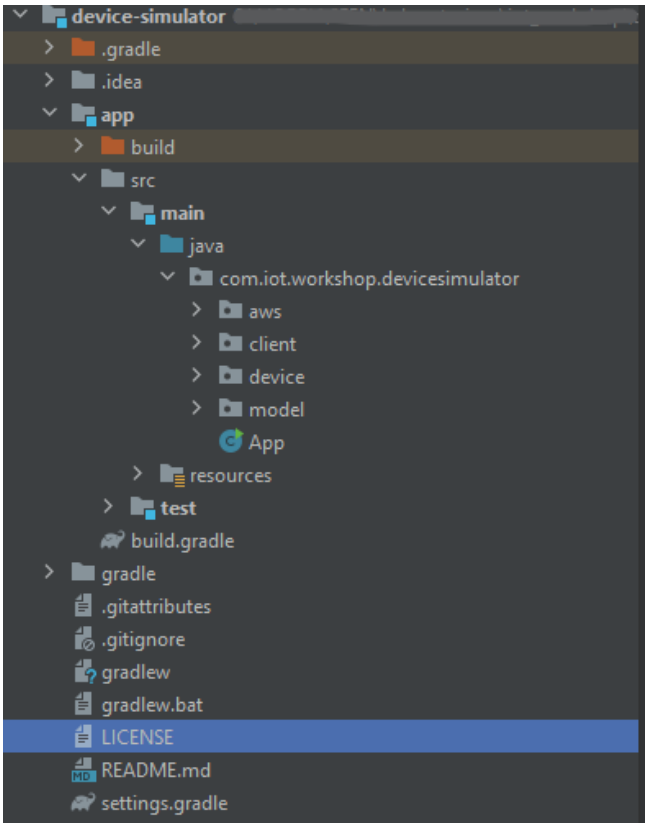
5. Wybieramy folder *~/iot-workshop/device-simulator*
6. Kliknij w *Import module from external model* i zaznacz *Gradle* jako narzędzie do budowania i zarządzania dependencjami



Rysunek 12

Naciskamy na *Finish*. Moduł powinien pojawić się w widoku *Project* po lewej stronie i powinien automatycznie się zbudować.

Końcowy efekt powinien mieć następującą formę:



Rysunek 13

7. Do folderu *resources* w ścieżce *src/main* umieść certyfikaty z poprzedniego rozdziału.

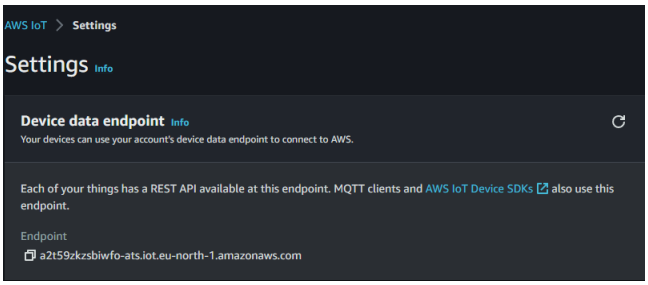
   Podczas działania programu (*runtime*), są one wczytywane do pamięci, symulując działanie prawdziwego urządzenia. Są one umieszczane nagłówku negocjacji połączenia *MQTT* z chmurą (brokerem *MQTT* i systemem typu *Authoriser* zawarte w *IoT Core*).

8. Otwórz plik *App.java* umieszczony w głównym pakiecie programu.

9. Zdefiniuj wartość pola klasy *DEVICE_ID* wartością *ID* zaprowizowanego urządzenia (to tak naprawdę nie ma znaczenia, albowiem w zastanej konfiguracji, AWS rozpoznaje urządzenie po certyfikacie, pozwoli to na zwiększenie przejrzystości logów).

10. To prawie koniec. Pozostaje jeszcze adres bramy frontowej systemu w chmurze, tzw. *endpoint*.

    W tym celu wracamy do *IoT Core* i w menu głównym tej usługi wybieramy *Settings*.



Rysunek 14

Poszukiwana wartość jest zawarta pod kluczem *endpoint*. Kopiujemy ją.

11. Wklejamy skopiowany adres jako wartość pola *ENDPOINT* w *App.java*.

```
public class App {

    private static final String ENDPOINT = "a2t59zkzsbiwfo-ats.iot.eu-north-1.amazonaws.com";

    private static final String DEVICE_ID = "DEVICE-12";
```

Rysunek 15

Gotowe. Urządzenie jest gotowe do pracy. W przypadku fizycznego urządzenia, należałoby wgrać aplikację, która jest w stanie komunikować się za pomocą protokołu *MQTT*, tj. posiada implementację klienta *MQTT*, np. z paczki SDK od AWSa i zawiera certyfikaty, jak i skonfigurowany adres do łączenia z chmurą.

Uwaga! *Certyfikatów nigdy nie należy publikować na repozytorium! Zawsze muszą być one zawarte w .gitignore.*

4.2 Oczekiwany rezultat

Po zrealizowaniu tego rozdziału, powinny być osiągnięte następujące warunki:

- Pobrano i zaimportowano *device-simulator*
- Zastosowano tzw. *forka* na sklonowanym repo, w celu włączenia go do swojego rozwiązania
- Przeniesiono certyfikaty do folderu *resources*
- Skonfigurowano *DEVICE_ID* i *ENDPOINT* w urządzeniu

Spełnienie wszystkich punktów jest konieczne do sukcesywnego wykonania dalszych etapów warsztatu!

4.3 Do przemyślenia

1. Jakie urządzenia *IoT* są *cloud-ready* na rynku?
2. Do czego można wykorzystać wirtualne urządzenie?
3. Gdzie zazwyczaj są przechowywane certyfikaty urządzenia (tego fizycznego)?

5 Wysłanie pierwszych wiadomości

W tym rozdziale dokonamy połączenia z chmurą i wyślemy pierwsze dane telemetryczne. Dysponując certyfikatem uwierzytelniającym dane urządzenie– aktora wchodzącego w interakcję z ekosystemem chmury– oraz nadanymi mu uprawnieniami– poprzez tzw. *policies*– jaki i wreszcie skonfigurowanym urządzeniem (lub programem), nic nie stoi na przeszkodzie w rozpoczęciu komunikacji.

5.1 Kroki do realizacji

1. Przed uruchomieniem „urządzenia", należy sprawdzić połączenie z samą chmurą. Wprowadź następującą komendę w terminalu:

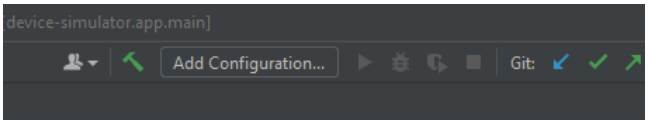
```
ping -c 5 <adres_iot_grupy>
```

   W rezultacie powinen pojawić się wynik podobny do tego:

```
PING ts.iot.eu-west-1.amazonaws.com 56(84) bytes of data.
64 bytes from ec2-.eu-west-1.compute.amazonaws.com:
    icmp_seq=1 ttl=231 time=127 ms
64 bytes from ec2-.eu-west-1.compute.amazonaws.com:
    icmp_seq=2 ttl=231 time=127 ms
64 bytes from ec2-.eu-west-1.compute.amazonaws.com:
    icmp_seq=3 ttl=231 time=127 ms
64 bytes from ec2-.eu-west-1.compute.amazonaws.com:
    icmp_seq=4 ttl=231 time=127 ms
64 bytes from ec2-.eu-west-1.compute.amazonaws.com:
    icmp_seq=5 ttl=231 time=127 ms
```

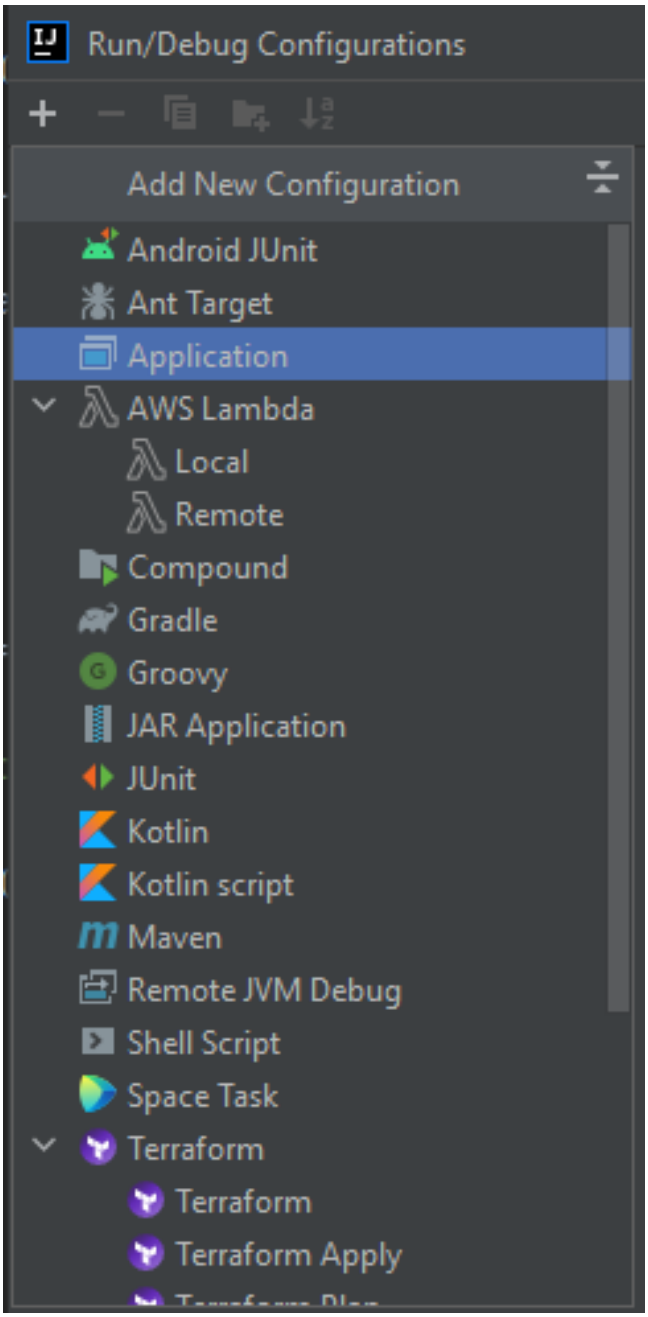
   Jeżeli rezultat jest poprawny, punkt połączeniowy dla urządzenia jest dostępny i można przejść do następnych kroków.

2. Mając otwarty *InteliJ*, klikamy w *Add Configuration...* w pasku nad edytorem.



Rysunek 16

3. Naciśnij w znak + i z listy wybierz *Application*

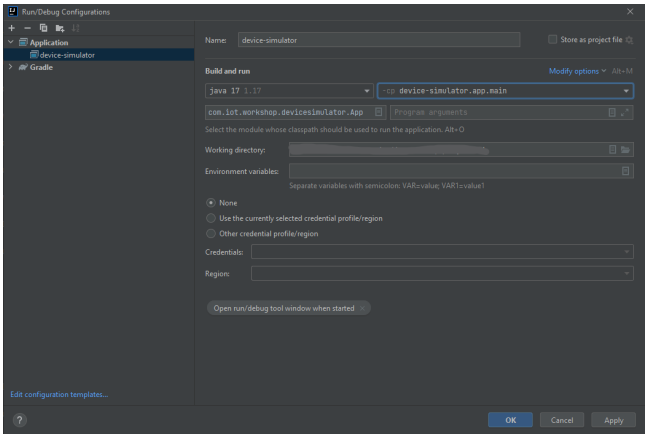


Rysunek 17

4. W polu *Name* wpisujemy „device-simulator”. Później, zainstalowaną wer-

się Javy i moduł *Gradle'a*, gdzie znajduje się kod źródłowy symulatora– *device-simulator.app.main*.

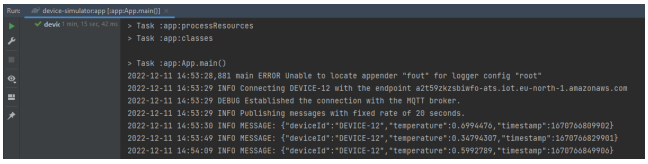
Na koniec, wprowadzamy pełną nazwę klasy (*fully-qualified name*), jak na załączonym rysunku.



Rysunek 18

5. *Apply* i *OK*

6. Konfiguracja uruchomieniowa jest gotowa. Program może wystartować. W tym celu klikamy w zieloną strzałkę po po prawej stronie od wyświetlanej konfiguracji. Na dole, w konsoli powinny pojawić się logi działającego programu.



Rysunek 19

Proszę nie zamykać programu– po określonym czasie sam jawnie zamknie połączenie z chmurą. Wysyła on telemetrię w regularnych odstępach czasu.

Jeżeli nie wystąpiły żadne problemy z połączeniem (powinny być widoczne w logach), wówczas można przejść dalej. W przeciwnym przypadku, należy dokładnie sprawdzić nazwy certyfikatów, czy są one umieszczone we właściwym miejscu lub polisę dostępu.

7. Wiemy, że urządzenie emituje dane sensoryczne, w tym przypadku temperaturę, do chmury.

Dane te są przesyłane protokołem *MQTT* na *topic* o nazwie *telemetry/%s*, gdzie *%s* to *ID* urządzenia. Umożliwia to potencjalnym konsumentom w systemie zasubskrybować się na ten *topic* i odbierać dane. Zastosowanie znaku # w nazwie *topica* pozwala na stosowanie filtra– # oznacza dowolny ciąg znaków. Dzięki temu na przykład, broker będzie wysyłał do konsumentów dane telemetryczne wszystkich urządzeń.

AWS IoT Core ma wbudowany broker *MQTT* wraz z graficznym interfejsem klienta *MQTT*. Nazywa się on *MQTT test client*.

W *AWS IoT* wybierz *Test -> MQTT test client*.

8. W polu *Topic filter* wpisz nazwę *topica*, na który nadaje twoje urządzenie.
9. *Subscribe*

   W sekcji *Subscriptions* powinien być widoczny zasubskrybowany *topic*. Po prawej będą wyświetlać się wiadomości.
10. Uruchom program jeszcze raz i wróć do klienta w *AWS IoT*. Wiadomości powinny pojawić się po krótkim czasie.
11. Przejdź do zakładki *Publish to a topic*.
12. Uruchom program jeszcze raz i kiedy nastąpi połączenie, w *AWS IoT* naciśnij *Publish*. Urządzenie powinno odebrać przesłaną wiadomość.

    Jak widać, komunikacja w protokole *MQTT* jest dwukierunkowa.

5.2 Oczekiwany rezultat

Po zrealizowaniu tego rozdziału, powinny być osiągnięte następujące warunki:

- Dodano konfigurację uruchomieniową dla *device-simulator*, która poprawnie startuje program
- Urządzenie poprawnie komunikuje się z wystawionym węzłem *AWS IoT*
- Wysłane wiadomości przez urządzenie zostały wyświetlone w *MQTT test client*
- Urządzenie odebrało wiadomości wysłane przez *MQTT test client*

Spełnienie wszystkich punktów jest konieczne do sukcesywnego wykonania dalszych etapów warsztatu!

5.3 Do przemyślenia

1. Jaki jest interwał wysyłania danych telemetrycznych przez urządzenie?
2. Jaką strukturę ma wysyłana wiadomość?
3. Na jaki port łączy się urządzenie?
4. Jak sprawdzić, czy wystawiona brama połączeniowa chmury ma otwarte porty?

6 Konstrukcja pierwszego serwisu

W tym rozdziale zostanie utworzony pierwszy serwis naszego ekosystemu *IoT*– *proxy-service*. Jak nazwa wskazuje, jest on odpowiedzialny za przekierowywanie wiadomości z brokera *MQTT* do pozostałych partycypantów, biorących udział w operacjach biznesowych.

Z poziomu samego *AWS* nie jest to konieczne, jednakże, aby nie tworzyć stricte natywnego rozwiązania pod tego poddostawcę, dobrze jest taki serwis wprowadzić. Może w przyszłości zajdzie potrzeba przeniesienia się na typ infrastruktury *on-premise*.

Co więcej, taka architektura pozwala nam na dokonanie pierwszego etapu operacji na danych odebranych przez system– dekodowania. Zazwyczaj w świecie *IoT* urządzenia ślą wiadomości w formie surowej, tj. zakodowane np. binarnie– w celu redukcji *payloadu*, a tym samym obciążenia sieci. Co więcej, niektóre czujniki mogą przesyłać odczyty w formie wymagającej właściwego odkodowania.

W tym przypadku nasze urządzenie wysyła temperaturę jako liczbę zmiennoprzecinkową mniejszą niż *1*. Musi ona zostać przetransformowana do właściwej postaci.

Drugim założeniem *proxy-service* jest kontrola napływających danych– w postaci logowania ich– oraz słanie ich w odkodowanej formie do kolejki rozgłoszeniowej *SNS– Simple Notification Service–* która pozwoli na emisję wiadomości do pozostałych serwisów.

Warto tutaj wspomnieć, że zalecana jest integracja systemów za pomocą kolejek, aniżeli poprzez bezpośrednią komunikację, gdyż pozwala to na tzw. *decoupling*, czyli separacje warstw.

6.1 Kroki do realizacji

1. Na początek postawimy infrastrukturę. Do tego posłużymy się narzędziem *Terraform*– jest to agnostyczny zasobnik do definiowania infrastruktury systemów w chmurach obliczeniowych za pomocą metajęzyka.

   Pozwala to na uniezależnienie się od konkretnego poddostawcy, poprzez wprowadzenie warstwy abstrakcji. Warto wspomnieć o pojęciu *IaC– Infrastructure as a Code–* infrastruktura zamodelowana w ten sposób jest czytelniejsza, zarządzalna i możliwa do modyfikacji przyrostowej, a nawet testowanie.

   Całą infrastrukturę można „wyklikać" w konsoli AWS lub utworzyć programatycznie za pomocą *AWS CLI*. Byłoby to jednak czasochłonne, a sam proces podatny jest na błąd.

Ta droga zostanie pominięta ze względu na oszczędność czasu, niemniej zachęca się do zrealizowania jej na własną rękę w procesie eksperymentu (poza tym warsztatem).

Przed przejściem do następnego kroku, przeczytaj zawartość dokumentu „Infrastructure as Code - Basics" (autorstwa DevOpsa Piotra Pietruszki), który stanowi podstawowe wprowadzenie w świat *IaC*. Dokument powinien znajdować się w folderze głównym warsztatów.

2. *Ad rem*. Utwórz folder *infrastructure* w głównym folderze swojego projektu.

3. Otwórz konsolę w tym folderze i wpisz

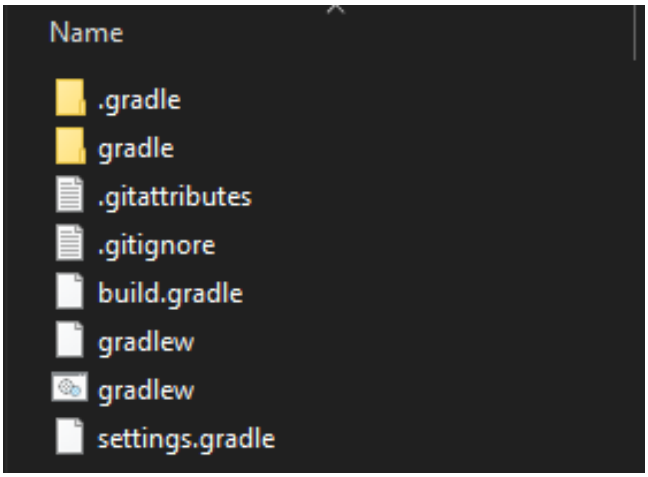
```
gradle init
```

4. Wybieramy *1: basic*

5. *1: Groovy*

6. Nazwę projektu zostawiamy domyślnie jako nazwa folderu (wystarczy wcisnąć *Enter*)

7. Spowoduje to utworzenie projektu podpiętego pod *Gradle* jako narzędzie do budowania.



Rysunek 20

8. Otwieramy plik *build.gradle* i wpisujemy:

```
group 'com.iot.workshop'
```

9. Teraz możemy zaimportować moduł, identycznie jak *device-simulator*

10. Następnie tworzymy folder *terraform*. Tam będziemy umieszczać kolejno pliki z infrastrukturą.

11. W folderze *terraform* tworzymy plik *main.tf*. Będzie on punktem wyjściowym infrastruktury.

12. *InteliJ* powinien mieć wsparcie do *HCL*, zapewniając podświetlanie jego struktury, jak i podpowiedzi.

    Otwieramy *main.tf* i wpisujemy magiczną inwokację:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.0.0"
    }
    archive = {
      source = "hashicorp/archive"
      version = "~> 2.2.0"
    }
  }

  required_version = "~> 1.0"
}

provider "aws" {
  region = var.aws_region
}

resource "aws_s3_bucket" "lambda_bucket" {
  bucket = "iot-workshop-lambda-<nr-grupy>"
}

resource "aws_s3_bucket_acl" "bucket_acl" {
  bucket = aws_s3_bucket.lambda_bucket.id
  acl    = "private"
}
```

Uwaga na boku Warto wspomnieć, że infrastruktura w *Terraform* zakodowana jest w języku *HCL*.

W miejscu *<nr-grupy>* wpisujemy nr swojej grupy.

Jak można się domyślić, zawarliśmy definicję kubełka *S3*– usługa składowania plików w świecie AWS. Drugi zasób– *resource*– determinuje kontrolę dostępu. Tutaj *private*, dzięki czemu nikt niepożądany nie będzie miał tam wstępu.

InteliJ powinien zaalarmować o błędach– *var.aws_region*.

Kolejna uwaga W celu wdrożenia serwisu w środowisku uruchomieniowym *Lambda*, zbudowana paczka serwisu musi być przesłana do *S3*.

13. Tworzymy następny plik *variables.tf* i wpisujemy:

```
variable "aws_region" {
 description = "AWS region for all resources."

 type    = string
default = "eu-north-1"
}
```

Określa to zmienną, którą będziemy wykorzystywać w pozostałych plikach. W polu *default* wpisujemy kod przydzielonego grupie regionu.

14. Otwieramy plik *.gitignore* i dodajemy następujące linie:

```
.gradle
build
gradlew
gradlew.bat
.idea
gradle
build.gradle
*.tfstate
*.iml
.terraform
*.tfstate.backup
```

Pliki *tfstate* zawierają dane krytyczne (m.in. klucze do AWS), dlatego nie powinny się znaleźć na repozytorium. Dodano też pozostałe niechciane pliki.

15. Czas na wygenerowanie pierwszej infrastruktury!

    W folderze *terraform* włączamy terminal i wpisujemy:

```
terraform init
```

To zainicjuje infrastrukturę.

16. Jeżeli nie wystąpił żaden błąd, czynimy dalej:

```
terraform apply
```

Ta komenda formalnie wdraża zdefiniowaną infrastrukturę w naszym imieniu. Podczas działania, program zapyta nas o potwierdzenie. Wpisujemy *yes*.

17. Jeżeli operacja zakończyła się sukcesem, warto sprawdzić efekt zastosowanych „czarów".

    W polu wyszukiwania konsoli AWS wpisujemy *S3* i wybieramy z listy wyświetloną usługę z wiadrem (kubełkiem).

18. W sekcji *Buckets* powinien znajdować się wpis ze zdefiniowaną przez nas nazwą.

    Uwaga na boku w związku z ograniczoną ilością kont, będą też wyświetlone kubełki innych grup– z innych regionów. Proszę się tym nie przejmować.

19. Teraz utworzona zostanie infrastruktura dla naszego serwisu.

    Tworzymy plik *proxy-service.tf* i otwieramy go.

20. Dodajemy kubełek, gdzie będzie przechowywana zbudowana paczka serwisu:

```
resource "aws_s3_object" "lambda_proxy_service" {
  bucket = aws_s3_bucket.lambda_bucket.id

  key    = var.proxy_service_filename
  source = var.proxy_service_zip_path

  etag = filemd5(var.proxy_service_zip_path)
}
```

21. Następnie definicję lambdy:

```
resource "aws_lambda_function" "proxy_service" {

  function_name = "ProxyService"

  s3_bucket = aws_s3_bucket.lambda_bucket.id
  s3_key    = aws_s3_object.lambda_proxy_service.key

  runtime = var.lambda_runtime
  handler = var.proxy_service_handler

  source_code_hash =
      base64sha256(filebase64(var.proxy_service_zip_path))

  role = aws_iam_role.proxy_service_role.arn
```

```
  memory_size = var.lambda_memory
  timeout = var.lambda_timeout
}
```

22. Dorzucamy logi do *CloudWatch*– system do monitorowania i przechowywania logów w usługach AWS.

    Dodatkowo, definiujemy rolę *IAM*, która wirtualnie reprezentuje zasób AWS i pozwala na definiowanie polis kontroli.

```
resource "aws_cloudwatch_log_group" "proxy_service_clw" {
  name =
      "/aws/lambda/${aws_lambda_function.proxy_service.function_name}"

  retention_in_days = 1
}

// Creating IAM role
resource "aws_iam_role" "proxy_service_role" {

  name = "proxy_service_role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Sid    = ""
      Principal = {
        Service = "lambda.amazonaws.com"
      }
    }
    ]
  })
}

// Allows Lambda to store logs to the CloudWatch
// For further read refer to:
    https://docs.aws.amazon.com/lambda/latest/dg/lambda-intro-execution-role.html
resource "aws_iam_role_policy_attachment" "proxy_service_policy" {
  role       = aws_iam_role.proxy_service_role.name
  policy_arn =
      "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
}
```

23. Do *variables.tf* dodajemy następujące definicje zmiennych:

```
variable "lambda_runtime" {
  default = "java11"
```

```
}

variable "lambda_memory" {
  default = 256
}

variable "lambda_timeout" {
  default = 10
}

// PROXY SERVICE
variable "proxy_service_filename" {
  default = "proxy-lambda-1.0.0-SNAPSHOT.zip"
}

variable "proxy_service_zip_path" {
  default =
      "../../proxy-lambda/build/distributions/proxy-lambda-1.0.0-SNAPSHOT.zip"
}

variable "proxy_service_handler" {
  default = "com.iot.workshop.lambda.proxy.Handler"
}
```

Warto wspomnieć, że tzw. *handler* serwisu w nomenklaturze AWS, to program główny w paczce, który zostanie uruchomiony przez środowisko AWS. W przypadku Javy, jest to pełna nazwa publicznej klasy, z publiczną metodą i domyślnym konstruktorem.

Przyjmujemy umownie, że każdy program uruchamiany w AWS będzie nazywał się *Handler*.

Warto tutaj zauważyć, że nie podano nazwy metody, która ma zostać uruchomiona– jest to spowodowane faktem, że klasa *Handler* implementuje interfejs z SDK AWS:

```
com.amazonaws.services.lambda.runtime.RequestHandler
```

24. Teraz zajmiemy się lambdą.

    W głównym folderze repozytorium pobieramy gotowy kod serwisu: *proxy-lambda*.

    Uwaga! Proszę zastosować procedurę *forka* jak w kroku z *device-simulator*!

25. Importujemy moduł do *InteliJ*, jak w poprzednich krokach.

26. Proszę dogłębnie zapoznać się z plikiem *build.gradle* i porównać go z tym w module *infrastructure*.

Szczególnie interesujący jest fragment *dependencies*, gdzie zawarto wszystkie dependencje wykorzystane w konstrukcji serwisu. Dependencje dodawane są automatycznie do tzw. *classpath* i są dołączane do końcowego produktu kompilacji. Można je znaleźć na repozytorium *Maven*, wpisując nazwy paczek w pole wyszukiwania.

Dodatkowo dodano *task* o nazwie *buildZip*. Służy on do spakowania wybudowanej paczki w *zipa*, który będzie mógł być wysłany do właściwego kubełka *S3*.

Ponadto, zawarto *task* o nazwie *release*– wykonuje on komendy *clean*, *build*, *test* i *buildZip*. To dobra praktyka, aby przed każdą publikacją na repozytorium uruchomić tę komendę– jej produktem finalnym jest zbudowana paczka. Jeżeli którykolwiek test nie przejdzie, wówczas build zostanie przerwany.

Później zostanie pokazane, jak uruchamiać poszczególne taski.

27. Klasa *Handler.java* jest klasą główną serwisu. Proszę zwrócić uwagę na zadeklarowany typ w *wildcard operator* o nazwie *TelemetryEvent* (linia nr 24).

    Jest to model oczekiwanej telemetrii tożsamej z tą odbieraną przez brokera *MQTT*.

    AWS automatycznie zajmuje się mapowaniem *JSONów* na tzw. *POJO*– czyli obiekty Javy. Jest to duże ułatwienie.

28. Przeanalizuj dokładnie wszystkie klasy w module.

29. W klasie *Handler.java*, dla pola *snsClient*, gdzie tworzony jest za pomocą *buildera*– warto sobie zapamiętać tej jeden z najpopularniejszych wzorców projektowania obiektowego– klient do wysyłania wiadomości *SNS*.

    Dodaj przed wywołaniem łańcuchowym *build()* metodę *.withRegion()*. Korzystając z klasy *com.amazonaws.regions.Regions* dodaj kod swojego regionu.

30. Założeniem *proxy-service* jest dekodowanie danych telemetrycznych urządzenia. Do tego celu utworzona został interfejs *TemperatureDecoder* w pakiecie *decoder*. Funkcjonalność dekodowania nie została dostarczona.

    Dodaj klasę *TemperatureImpl*, która realizuje następującą operację:

$$T = t_{raw} \cdot |t_{min} - t_{max}| + t_{min} \tag{1}$$

$$\text{gdzie:} \quad t_{min} = 10.00$$
$$t_{max} = 31.00$$

    Zadanie można uznać za wykonane, jeżeli wszystkie testy serwisy przejdą pomyślnie. Aby uruchomić testy, wystarczy wpisać w terminalu projektu komendę:

```
gradle test
```

31. Kolejnym zadaniem *proxy-service* jest wysłanie danych na kolejkę rozgłoszeniową *SNS*.

    W tym celu przejdź do klasy *SnsPublisher* w pakiecie *sns* i uzupełnij brakujący kod. Zwróć uwagę, jak pobierany jest identyfikator kolejki *ARN*, na którą mają być wysyłane wiadomości.

    Wskazówka pomocna może okazać się dokumentacja AWS na temat SNS dla Javy.

32. Wracamy do infrastruktury. Należy dodać definicję kolejki.

    W pliku *proxy-service.tf* wstawiamy:

```
// output SNS
resource "aws_sns_topic" "telemetry_sns" {
  name = "telemetry_topic"
}
```

33. W tym samym pliku dodajemy do zasobu *„aws_lambda_function”* następujący wpis:

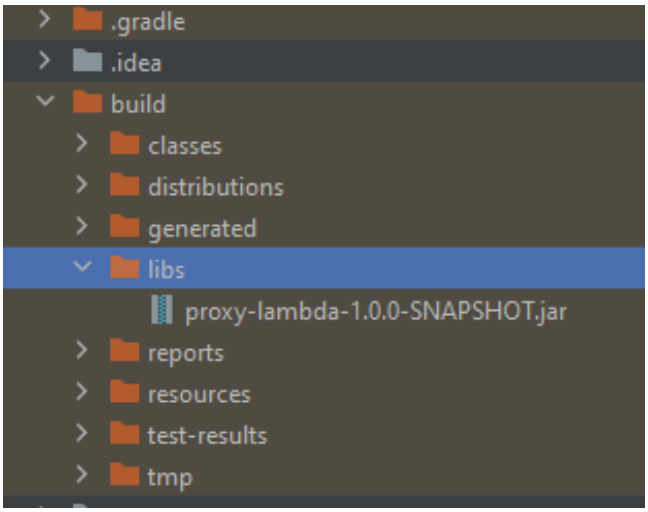
```
environment {
  variables = {
    TOPIC_ARN = "${aws_sns_topic.telemetry_sns.arn}"
  }
}
```

34. Teraz możemy zbudować paczkę serwisu.

    Otwieramy konsolę w pliku *proxy-lambda*– lub w *InteliJ*– i wpisujemy komendę:

```
gradle release
```

    Sprawdzamy, czy paczka się zbudowała.



Rysunek 21

35. Uruchamiamy komendę

```
terraform apply
```

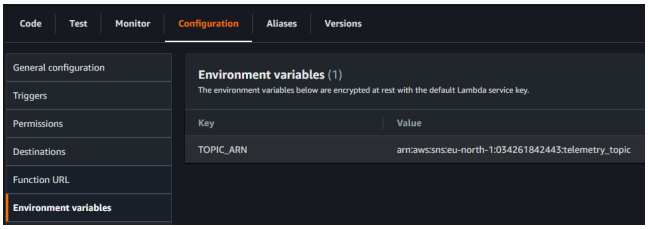
Przypomnienie: należy to uczynić w konsoli z poziomu *infrastructure/terraform*.

Jeżeli nigdzie nie został popełniony błąd, nowa infrastruktura powinna wylądować na AWS.

W celu potwierdzenia, sprawdzamy paczkę w naszym folderze na *S3*. Dodatkowo, przechodzimy do usługi *Lambda* (wpisując „lambda" w polu wyszukiwania) i tam odnajdujemy naszą lambdę– *ProxyService*.

36. Klikamy na naszą lambdę i w zakładkę *Configuration*. Odnajdujemy *Envrionment variables*.

Sprawdzamy, czy *ARN* naszego *topica* zostało nadane.



Rysunek 22

37. Na koniec, integracja z brokerem *MQTT*.

    Przejdź do *IoT Core*.

38. *Security -> Policies*.

39. Zaznacz polisę, którą utworzyłeś, i usuń ją. Utworzymy ją w *Terraform*.

40. Tworzymy plik *iot.tf* w folderze z infrastrukturą.

41. Wprowadzamy następujący kod:

```
data "aws_iot_endpoint" "iot_core" {
}

resource "aws_iot_policy" "deviceGeneralPolicy" {
  name = "deviceGeneralPolicy"

  # Terraform's "jsonencode" function converts a
  # Terraform expression result to valid JSON syntax.
  policy = jsonencode({
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": "iot:Connect",
        "Resource": "*"
      },
      {
        "Effect": "Allow",
        "Action": "iot:Subscribe",
        "Resource": "*"
      },
      {
        "Effect": "Allow",
        "Action": "iot:Publish",
        "Resource": "*"
      },
      {
        "Effect": "Allow",
        "Action": "iot:Receive",
        "Resource": "*"
      },
      {
        "Effect": "Allow",
        "Action": "iot:AssumeRoleWithCertificate",
        "Resource": "*"
      }
    ]
  })
}
```

```
resource "aws_iot_topic_rule" "rule_telemetry" {
  name        = "TelemetryRule"
  description = "Rule for sending telemetry data"
  enabled     = true
  sql         = "SELECT * FROM 'telemetry/#'"
  sql_version = "2016-03-23"

  lambda {
    function_arn = aws_lambda_function.proxy_service.arn
  }
}
```

Zwróć uwagę, że *deviceGeneralPolicy* jest dokładną kopią utworzonej ręcznie polisy.

Przypatrz się też *rule_telemetry*. Dane z *topica* każdego urządzenia *telemetry* wysyłane są do lamdby *proxy-service*.

42. Dodaj polisę do certyfikatu urządzenia.
43. Uruchom *Terraform*.
44. Jeżeli nie wystąpił żaden błąd, uruchom urządzenie.
45. Rezultat przetwarzania powinien być widoczny w *CloudWatch -> Log groups*.
46. Przeanalizuj dokładnie logi i zlokalizuj wszelkie błędy, jeżeli takowe wystąpiły.
47. Gratulacje! Właśnie zbudowany został pierwszy serwis w chmurze przy użyciu *Terraform*!

6.2 Oczekiwany rezultat

Po zrealizowaniu tego rozdziału, powinny być osiągnięte następujące warunki:

- Utworzono infrastrukturę dla *proxy-service* w *Terraform* i dla *IoT*
- Cała infrastruktura przy użyciu *Terraform* została wdrożona na AWS
- Zaimportowano i uzupełniono brakujące luki w *proxy-service*
- Zbudowano paczkę dla *proxy-service*
- Dane telemetryczne urządzenia są odbierane i przetwarzane przez *proxy-service*

Spełnienie wszystkich punktów jest konieczne do sukcesywnego wykonania dalszych etapów warsztatu!

6.3 Do przemyślenia

1. Jaka jest inna możliwość przesłania wiadomości z *IoT Core* na *SNS*?
2. Co cechuje lambdę, że dobrze sprawdza się do realizacji roli *proxy-service*?
3. Jak nazywa się ekwiwalent lambdy AWS na Azure?

7 Utrwalanie danych

W tym rozdziale będziemy zapisywać dane telemetryczne do bazy danych. Będą one mogły posłużyć do analizy, a także na rzecz operacji biznesowych innych serwisów.

W dziedzinie *IoT* rozgranicza się bazy na tzw. *hot storage*– bazy dużej wydajności i składujące dane przez określony czas– oraz *cold storage*– *data lake*, gdzie wydajność odczytu nie stanowi newralgicznego elementu systemu.

Warto wspomnieć, że ten rozdział jest także utrwaleniem nabytej wiedzy w rozdziale poprzednim. W konsekwencji, zostanie zupełnie samodzielnie utworzony serwis *telemetry-storage-service* do zapisu danych do *DynamoDB* odebranych z kolejki *telemetry_topic*.

Uwaga! Platformą bazodanową będzie *DynamoDB*.

7.1 Kroki do realizacji

1. Utwórz infrastrukturę dla *telemetry-storage-service*, podobnie, jak dla *proxy-service*.

2. Nadaj nazwę lambdzie „TelemetryStorageService”

3. Aby serwis był wpięty w kolejkę rozgłoszeniową *SNS*, dodaj następujący wpis:

```
data "aws_iot_endpoint" "iot_core" {
resource "aws_lambda_permission"
    "allow_invocation_from_sns_telemetry_service" {
  statement_id = "AllowExecutionFromSNS"
  action       = "lambda:InvokeFunction"
  function_name =
      aws_lambda_function.telemetry_storage_service.function_name
  principal    = "sns.amazonaws.com"
  source_arn   = aws_sns_topic.telemetry_sns.arn
}
```

4. Aby umożliwić serwisowi zapis danych do bazy, musimy nadać uprawnienia:

```
resource "aws_iam_role_policy" "lambda_store_policy" {

  name = "dynamoStorePolicy"
  role = aws_iam_role.storage_service_role.name

  policy = jsonencode({
```

```
    "Version": "2012-10-17",
    "Statement":[
      {
        "Effect": "Allow",
        "Action": "dynamodb:PutItem",
        "Resource": aws_dynamodb_table.telemetry-data.arn
      },
      {
        "Effect": "Allow",
        "Action": "dynamodb:UpdateItem",
        "Resource": aws_dynamodb_table.telemetry-data.arn
      }
    ]
  })
}
```

5. Tworzymy plik *datastore.tf* w folderze z infrastrukturą o treści:

```
resource "aws_dynamodb_table" "telemetry-data" {

  name          = "TelemetryData"
  billing_mode  = "PAY_PER_REQUEST"

  hash_key      = "deviceId"
  range_key     = "arrivalTimestamp"

  attribute {
    name = "deviceId"
    type = "S"
  }

  attribute {
    name = "arrivalTimestamp"
    type = "N"
  }

  tags = {
    Name       = "iot-telemetry"
    Environment = "alpha"
  }
}
```

Rezultat można podziwiać w konsoli AWS po otwarciu *DynamoDB*. Utworzone table powinny już tam być (po uruchomieniu *Terraform*).

6. Zaleca się, aby kod handlera miał następującą formę:

```
package com.iot.workshop.lambda.storage;
```

```
import com.iot.workshop.lambda.storage.data.TelemetryRepository;
import com.iot.workshop.lambda.storage.decoder.JsonToEntityDecoder;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * This class must be public with a default constructor, so AWS
     (or any other cloud provider) can wrap it upon
 * invocation request
 * <p>
 * Here is example with implementation of {@link RequestHandler}
     provided by aws sdk, where the 'I' stands for expected
 * input and 'O'-- output
 */
public final class Handler implements RequestHandler<SNSEvent,
    Void> {

    /**
     * Logger defined by the slf4j interface
     */
    private static final Logger logger = LoggerFactory.getLogger(
        Handler.class );

    /**
     * All dependencies are defined as fields of the class
     */
    private final TelemetryRepository repository;

    private final JsonToEntityDecoder jsonToEntityDecoder;

    /**
     * Default constructor is provided (it does not take any
         parameters), so it can be wrapped during the runtime by
         the
     * lambda container.
     * <p>
     * It all dependencies shall be initialized here
     */
    public Handler() {
        repository = TelemetryRepository.instance();
        jsonToEntityDecoder = JsonToEntityDecoder.instance();
    }

    @Override
    public Void handleRequest( SNSEvent event, Context context ) {
```

```
        logger.info( "Received an event: {}", event );

        event.getRecords()
                .stream()
                .map( record -> record.getSNS() )
                .map( sns -> sns.getMessage() )
                .map( jsonToEntityDecoder::decode )
                .forEach( repository::store );

        /*
         * It is recommended to return codes of the invocation,
             i.e. 200 OK when no error has been occurred
         */
        return null;
    }
}
```

Jak można zaobserwować, *RequestHandler* oczekuje typu *SNSEvent*– jest to POJO dla wiadomości *SNS*.

Pakiet *'com.amazonaws:aws-lambda-java-events:3.11.0'* powinien zawierać definicję klasy *SNSEvent*.

W lini nr *52-53* widać wypakowywanie payloadu *SNS*. Jest to wiadomość w formacie *JSON*. Musi ona być zmapowana do modelu w Javie.

Do mapowania używany zazwyczaj jest *Jackson*. Potrzebne są następujące dependencje:

```
// Jackson
//
    https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind
implementation group: 'com.fasterxml.jackson.core', name:
    'jackson-databind', version: '2.14.0'

//
    https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core
implementation group: 'com.fasterxml.jackson.core', name:
    'jackson-core', version: '2.14.0'

//
    https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-annotations
implementation group: 'com.fasterxml.jackson.core', name:
    'jackson-annotations', version: '2.14.0'
```

Sam proces mapowania można zrealizować w następujący sposób:

```
package com.iot.workshop.lambda.storage.decoder;
```

```
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.iot.workshop.lambda.storage.model.TelemetryData;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class JsonToEntityDecoderImpl implements
    JsonToEntityDecoder {

    private static final Logger logger = LoggerFactory.getLogger(
        JsonToEntityDecoderImpl.class );

    private final ObjectMapper objectMapper;

    public JsonToEntityDecoderImpl() {
        objectMapper = new ObjectMapper();
    }

    @Override
    public TelemetryData decode( String json ) {

        try {
            return objectMapper.readValue( json,
                TelemetryData.class );
        }
        catch ( JsonProcessingException e ) {
            logger.error( "Error while decoding JSON payload {}:
                {}", json, e.getMessage() );
        }

        /*
         * Returning NULL is never a good idea, unless You have a
             real reason to. For simplicity, it can be forgiven.
         * In a real world, wrap the exception into custom
             exception or return {@link java.util.Optional}.
         */
        return null;
    }
}
```

7. Przed realizacją połączenia z *DynamoDB*, warto zapoznać się z [1]. Propozycja implementacji repozytorium:

```
package com.iot.workshop.lambda.storage.data;

import com.iot.workshop.lambda.storage.model.TelemetryData;

import com.amazonaws.regions.Regions;
```

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import
    com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import
    com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

final class TelemetryRepositoryImpl implements TelemetryRepository
    {

    private static final Logger logger = LoggerFactory.getLogger(
        TelemetryRepositoryImpl.class );

    // How to initialize this?
    private final AmazonDynamoDB client;
    /**
     * This provides basic functionality of mapping DB entities
         into Java objects (aka POJOs-- Plain Old Java Objects)
     * and vice versa. Thanks to that it is not needed to write
         queries / requests and mapping manually. Still, for an
     * object to be mapped, proper annotations must be defined.
         Sadly, they do not fall into the standardized JPA.
     */
    private final DynamoDBMapper mapper = new DynamoDBMapper(
        client );

    @Override
    public void store( TelemetryData event ) {

        logger.debug( "Storing event in the database." );

        \\ What to do here?
    }
}
```

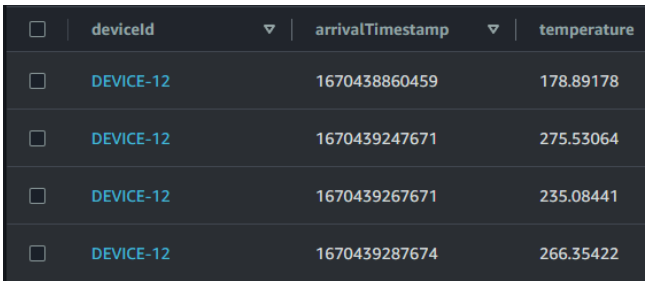
Uwaga! Proszę pamiętać o podaniu właściwego regionu w *builderze* klienta *DynamoDB*!

Dependencje potrzebne do komunikacji z bazą:

```
//
    https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-dynamodb
implementation group: 'com.amazonaws', name:
    'aws-java-sdk-dynamodb', version: '1.12.349'
```

8. Uruchomienie urządzenia powinno poskutować pojawieniem się nowych

wpisów w bazie.



deviceId	arrivalTimestamp	temperature
DEVICE-12	1670438860459	178.89178
DEVICE-12	1670439247671	275.53064
DEVICE-12	1670439267671	235.08441
DEVICE-12	1670439287674	266.35422

Rysunek 23

9. Gratuluję! To nie było trywialne zadanie!

7.2 Oczekiwany rezultat

Po zrealizowaniu tego rozdziału, powinny być osiągnięte następujące warunki:

- Utworzono infrastrukturę dla *telemetry-storage-service* w *Terraform*
- Cała infrastruktura przy użyciu *Terraform* została wdrożona na AWS
- Zaimplementowany został *telemetry-storage-service*
- Zbudowano paczkę dla *telemetry-storage-service*
- Dane telemetryczne urządzenia są odbierane i przetwarzane przez *telemetry-storage-service*
- *telemetry-storage-service* zapisuje dane w bazie danych

Spełnienie wszystkich punktów jest konieczne do sukcesywnego wykonania dalszych etapów warsztatu!

7.3 Do przemyślenia

1. Jaka platforma do tzw. *Cold Strage* nadałaby się najlepiej?
2. Czy *DynamoDB* to dobry wybór jako platforma do składowania danych telemetrycznych? Za i przeciw.
3. Co cechuje *DynamoDB* jako platformę bazodanową?

8 Continious Inegration

Ten rozdział stanowi o tworzeniu własnego *CI/CD– Continious Inegration / Continious Deployment*. Każde oprogramowanie w świecie biznesu (*enterprise*) powinno być zintegrowane z narzędziem do *CI/CD*. Pozwala to na kontrolę jakości kodu w sposób zautomatyzowany, co jest kluczowe w środowiskach o dużej kolaboracji, a także na dostarczanie paczek oprogramowania w stanie ciągłym, w dowolnym czasie.

To redukuje koszt poprzez nie tylko wykrywanie błędów przed publikacją paczki na produkcji, ale i też proces budowania i wdrażania nie wymaga człowieka.

Proszę wykonać kroki zawarte w dokumencie „CICD in Azure DevOps setup - runbook" Piotra Pietruszki, który powinien znajdować się w folderze z pobranymi materiałami do warsztatów. Następnie, wykonaj kroki z „Create your first CICD pipeline".

Zadanie do realizacji Postaw *CI/CD* dla całego ekosystemu wytworzonego na warsztatach.

8.1 Oczekiwany rezultat

Po zrealizowaniu tego rozdziału, powinny być osiągnięte następujące warunki:

- Pomyślnie zrealizowano proces tworzenia *CI/CD* z załączonych dokumentów.
- Postawiono *CI/CD* dla swojego ekosystemu.

Spełnienie wszystkich punktów jest konieczne do sukcesywnego wykonania dalszych etapów warsztatu!

8.2 Do przemyślenia

1. Wymień narzędzia realizujące koncept *CI/CD*?

9 Kontrola stanu urządzenia

Kluczowym zadaniem serwisów *IoT* jest komunikacja z urządzeniami, w celu np. wykonywania za nich obliczeń. W przedstawianym przypadku utworzymy serwis *device-control-service*, który będzie sterował stanem termostatu, aby ustawić go w jeden ze stanów: *COOLING*, *HEATING*, *IDLE*. To teoretycznie umożliwi osiągnięcie temperatury docelowej, ustawionej przez użytkownika.

9.1 Kroki do realizacji

1. Podobnie, jak w przypadku *device-simulator*, pobieramy kod źródłowy serwisu *device-control-service* z repozytorium.

   Uwaga! Proszę pamiętać o dokonaniu *forka*, wzorując się na rozdziale z *device-simulator*.

2. Importujemy moduł do *InteliJ*.

3. Dodatkowo, z folderu *device-control-service/terraform* przenosimy pliki **.tf* do miejsca, gdzie przechowywujemy infrastrukturę.

4. Proszę zapoznać się dogłębnie z implementacją serwisu.

   Zastosowano tutaj wzorzec projektowy *Dependency Injection* (*DI*), nazwany także *Inversion of Control (*IoC). Obowiązkowo (poza warsztatami) proszę poczytać materiał Martina Fowlera o tym [3].

   DI został zaimplementowany przy użyciu frameworka *Dagger* [5]. Wybór padł na to narzędzie, pomimo dodatkowej trudności korzystania z niego, albowiem jest on frameworkiem do *DI*, który działa w czasie kompilacji, a nie działania programu. W świecie usług działających w chmurze, każda nanosekunda się liczy, wobec tego *Dagger* oznacza dla nas oszczędność.

   Poszczególne dependencje– tutaj rozumiane nie jako te pobierane z repozytorium zewnętrznego, a jako konkretne klasy w systemie– wstrzykiwane są automatycznie przez *Dagger*, po zakomunikowaniu tego za pomocą adnotacji *javax.inject.Inject*. Na marginesie warto dodać, że *Inject* został dodany do Javy EE w ramach *JSR-330* [6]– stał się standardem. To ważne, albowiem należy pamiętać, aby projekt systemu był możliwie jak najbardziej oparty na standardach Javy, aniżeli na „niuansach" różnych frameworków.

   Proszę przede wszystkim przyjrzeć się jak wysyłane są wiadomości do urządzenia. Jest to realizowane przez klasę *DeviceMessengerImpl* pakietu *device*. Proszę porównać to z implementacją klienta *MQTT* w *device-simulator*.

5. W klasie *RepositoryModule* pakietu *di.module* proszę zmienić w linii 16. region na ten przydzielony grupie.

6. Proszę uzupełnić implementację klasy *TelemetryGetRepositoryImpl*.

   Metoda *getEventsFromRange* powinna zwracać wszystkie wiersze dla urządzenia o podanym *deviceId* i o *arrivalTimestamp* wcześniejszym, niż *timestampBoundInclusive*.

   Do realizacji zadania przyda się materiał stanowiący o kwerendach typu *scan*[7].

7. Nareszcie, przechodzimy do *DeviceStateOperatorImpl* pakietu *state*. Jest to klasa mająca na celu określenie na podstawie danych telemetrycznych z ostatnich 5. minut na jaki stan zmienić termostat, aby osiągnąć temperaturę docelową ustawioną przez użytkownika.

   Temperatura docelowa jest zwracana przez repozytorium *DeviceConfigRepository*– jest to „wydmuszka", która została wprowadzona dla uproszczenia materii.

   Proszę uzupełnić brakujące fragmenty kodu. Potwierdzeniem poprawnej implementacji jest pomyślne przejście testu *DeviceStateOperatorTest*.

8. Jeżeli testy przechodzą, można zbudować paczkę za pomocą *Gradle* i uruchomić *Terraform*.

9. Pomyślnie wdrożona infrastruktura powinna być na AWS.

10. Uruchom urządzenie i odczekaj na pojawienie się wiadomości od *device-control-service*.

11. Dodatkowo, zastosuj *MQTT test client* i zasubskrybuj na *topicu* „commands/#". Konieczne będzie ponowne uruchomienie urządzenia.

9.2 Oczekiwany rezultat

Po zrealizowaniu tego rozdziału, powinny być osiągnięte następujące warunki:

- Utworzono infrastrukturę dla *device-control-service* w *Terraform*
- Cała infrastruktura przy użyciu *Terraform* została wdrożona na AWS
- Zaimportowany został *device-control-service*
- Zbudowano paczkę dla *device-control-service*
- Wszystkie testy modułu przechodzą pomyślnie
- Dane telemetryczne urządzenia są odbierane i przetwarzane przez *device-control-service*
- Urządzenie otrzymuje komendy od chmury i zmienia stan

Spełnienie wszystkich punktów jest konieczne do sukcesywnego wykonania dalszych etapów warsztatu!

9.3 Do przemyślenia

1. Czym jest spowodowana różnica w implementacji klienta $MQTT$ od tego do wysyłania wiadomości do urządzenia?

10 Kontrola bezpieczeństwa

Wysyłanie alertów w obwieszczających użytkownika (np. administratora) o potencjalnych problemach z flotą, jest częstym zastosowaniem w kontroli jej jakości. Dla użytkownika „domowego" użytecznym może być informacja, np. o tym, że w jego domu jest bardzo gorąc, jak choćby wskutek pożaru.

W tym rozdziale utworzymy serwis do wysyłania notyfikacji w postaci e-maila, w momencie, kiedy urządzenie prześle odczyt o alarmującej wartości.

10.1 Kroki do realizacji

1. Podobnie, jak w przypadku *device-simulator*, pobieramy kod źródłowy serwisu *notification-service* z repozytorium.

   Uwaga! Proszę pamiętać o dokonaniu *forka*, wzorując się na rozdziale z *device-simulator*.

2. Importujemy moduł do *InteliJ*.

3. Dodatkowo, z folderu *notification-service/terraform* przenosimy pliki **.tf* do miejsca, gdzie przechowujemy infrastrukturę.

4. Proszę zapoznać się dogłębnie z implementacją serwisu.

5. Zmień region w *ThresholdConfigGetRepositoryImpl* pakietu *data*.

6. Uzupełnij implementację klasy *ThresholdCrossedEvaluatorImpl* pakietu *evaluator*.

7. Uzupełnij implementację klasy *NotificationFactoryImpl* pakietu *notifications*.

8. w pliku *notification-service.tf* dla zasobu *topic_email_subscription* wprowadź swój email.

9. Po uruchomieniu *Terraform* na skrzynkę powinna przyjść wiadomość z AWS w celu potwierdzenia subskrypcji. Zgadzamy się na nią klikając w link.

10. Przed uruchomieniem urządzenia, przejdź do *DynamoDB*.

11. *Tables -> Explore items -> ThresholdConfig*.

12. *Create item*.

13. Wpisujemy *ID* naszego urządzenia i wartość progową, najlepiej jak najniższą, aby zagwarantować otrzymanie notyfikacji.

Rysunek 24

14. Uruchamiamy urządzenie.

10.2 Oczekiwany rezultat

Po zrealizowaniu tego rozdziału, powinny być osiągnięte następujące warunki:

- Utworzono infrastrukturę dla *notification-service* w *Terraform*
- Cała infrastruktura przy użyciu *Terraform* została wdrożona na AWS
- Zaimportowany został *notification-service*
- Zbudowano paczkę dla *notification-service*
- Notyfikacje zostają odebrane przez skonfigurowaną skrzynkę pocztową

10.3 Do przemyślenia

1. Jaka usługa odpowiada za wysyłanie e-maili w serwisie?
2. Jak nazywa się w świecie AWS dedykowana usługa do wysyłania e-maili?

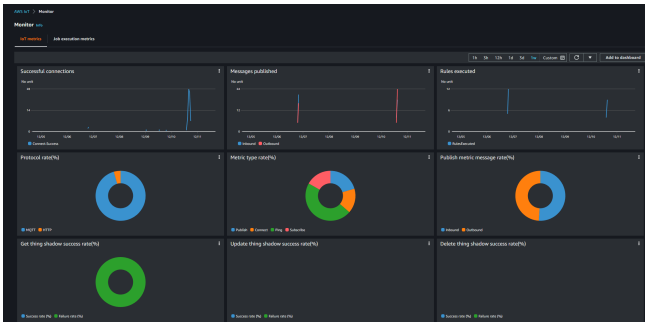
11 Wizualizacja i Metryki

W tym rozdziale zobrazujemy otrzymywane przez chmurę dane, jak i zwizualizujemy metryki. Pozwala to na kontrolę nad naszym systemem– dzięki temu jesteśmy w stanie określić, jak dużo wiadomości przepływa i wypływa z chmury.

11.1 Kroki do realizacji

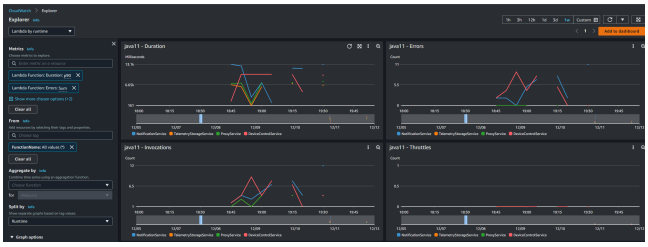
1. Przejdź do *AWS IoT Core*.
2. W menu po lewej stronie, wybierz *Monitor*.
3. Pojawi się zakładka *IoT metrics*.

   Tutaj możemy zobaczyć m.in. ile wiadomości przychodzących i wychodzących zostało opublikowanych przez brokera po stronie *IoT Core*.



Rysunek 25

4. Przejdźmy do *CloudWatch*. Tam też można wizualizować metryki, np. czas wykonywania programu na lambdach.
5. Klikamy w *Explorer*. To prosty „kreator”, w którym możemy utworzyć z listy gotowe panele z metrykami.
6. Z listy rozwijanej pod *Explorer* wybieramy np. *Lambda by runtime*.



Rysunek 26

7. W planach tych warsztatów było przejście przez konstrukcje wykresów za pomocą *IoT Analytics* i *QuickSight*, jednakże zostało to w ostatniej chwili „wycięte" z materiałów. Jest to związane z faktem, że usługa *QuickSight* wymaga dodatkowej rejestracji i jest słono płatna dla kreatorów.

   Proszę jednak pamiętać na przyszłość, że ów serwis w świecie enterprise służy właśnie do prezentacji i analizy zbiorów danych, m.in. do *IoT*.

   IoT Analytics realizuje funkcje zaimplementowane w ramach tych laboratoriów (przetwarzanie i zapis do bazy) w celu ich dalszej analizy i wizualizacji [8].

   Proszę zapoznać się z [9], aby zobaczyć, jak może wyglądać integracja z *QuickSight*, jak i *Jupyterem*.

11.2 Oczekiwany rezultat

Po zrealizowaniu tego rozdziału, powinny być osiągnięte następujące warunki:

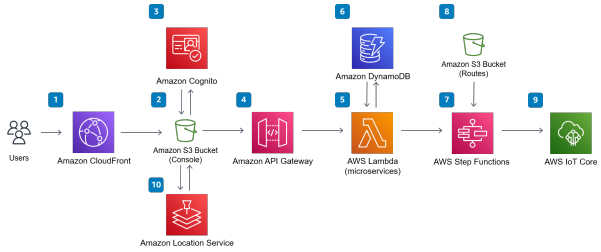
- Zobrazowano metryki w *IoT Core* i *CloudWatch*
- Zapoznano się ze stroną o *QuickSight*

11.3 Do przemyślenia

1. Czym różni się *IoT Analytics* od standardowej ścieżki przetwarzania i składowania danych?

12 Testy obciążeniowe

W tym rozdziale postawimy infrastrukturę do *IoT Simulator* [10] od AWS w celu utworzenia i zasymulowania floty testowej urządzeń.



Rysunek 27: [10]

Proszę wykonać kroki zawarte w dokumencie „IoT device simulator - CloudFormation runbook" Piotra Pietruszki, który powinien znajdować się w folderze z pobranymi materiałami do warsztatów.

Uwaga! Folder z *iot-device-simulator* powinien znajdować się w pliku pobranym na rzecz warsztatów.

Do wykonania: Po postawieniu narzędzia, proszę utworzyć 100 urządzeń symulujących termostat. Następnie proszę prześledzić metryki z poprzedniego rozdziału.

12.1 Oczekiwany rezultat

Po zrealizowaniu tego rozdziału, powinny być osiągnięte następujące warunki:

- Postawiono *iot-device-simulator*
- Utworzono konfigurację 100 urządzeń typu termostat
- Uruchomiono symulator

12.2 Do przemyślenia

1. W jakim celu tworzy się symulator floty urządzeń?

13 Sprzątanie

Na koniec pracy, proszę usunąć wszystkie utworzone ręcznie zasoby– przede wszystkim certyfikaty w *IoT Core*. Po wszystkim, proszę wpisać komendę:

```
terraform destroy
```

Literatura

[1] *Java Annotations for DynamoDB* https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBMapper.Annotations.html (dostęp 14 grudnia 2022)

[2] *Gradle* https://docs.gradle.org/current/userguide/what_is_gradle.html (dostęp 14 grudnia 2022)

[3] Martin Fowler *Inversion of Control* https://www.martinfowler.com/articles/injection.html (dostęp 14 grudnia 2022)

[4] *Lambda handlers* https://docs.aws.amazon.com/lambda/latest/dg/java-handler.html (dostęp 14 grudnia 2022)

[5] *Dagger* https://dagger.dev/ (dostęp 14 grudnia 2022)

[6] *JSR-330* https://jcp.org/en/jsr/detail?id=330 (dostęp 14 grudnia 2022)

[7] *DynamoDB scan* https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ScanJavaDocumentAPI.html (dostęp 14 grudnia 2022)

[8] *IoT Analytics* https://docs.aws.amazon.com/iotanalytics/latest/userguide/welcome.html (dostęp 14 grudnia 2022)

[9] *IoT Analytics how to* https://docs.aws.amazon.com/iotanalytics/latest/userguide/data-visualization.html (dostęp 14 grudnia 2022)

[10] *IoT Device Simulator by AWS* https://aws.amazon.com/solutions/implementations/iot-device-simulator/ (dostęp 14 grudnia 2022)