

Wstęp

Urządzenia podłączone do systemu zgłaszają się po restarcie w następującej formie:

Dla protokołu w wersji 1.6.x:

wysyłając komunikat REST na endpoint `/protocols/iot16/bootnotification/{deviceId}`

```
class BootNotificationRequest {  
    String deviceVendor;  
    String deviceModel;  
    String deviceSerialNumber;  
    String firmwareVersion;  
    ...  
}
```

Dla protokołu w wersji 2.0.x:

wysyłając komunikat REST na endpoint `/protocols/iot20/bootnotification/{deviceId}`

```
class BootNotificationRequest {  
    Device device;  
    Reason reason;  
    static class Device {  
        String serialNumber;  
        String model;  
        Modem modem;  
        String vendorName;  
        String firmwareVersion;  
    }  
    ...  
}
```

W przypadku obu protokołów odpowiedzią na komunikat jest:

```
class BootNotificationResponse {  
    String currentTime;  
    int interval;  
    Status status;  
    enum Status { Accepted, Pending, Rejected }  
}
```

Gdzie **interval** to ilość sekund, co którą urządzenie ma przysyłać komunikat Heartbeat.

UWAGA: Wszystkie komunikaty oraz struktura ścieżek endpointów są zestandaryzowane i nie podlegają zmianom czy rozszerzeniom.

Ćwiczenie: Modelowanie Value Objects i Policy

Zaimplementuj logikę wyliczania pola **interval** komunikatu zwrotnego **BootNotificationResponse**.

Wymagania:

Heartbeat interwał może być określony przez administratora systemu dla:

- konkretnego podzbioru urządzeń wskazanych po **deviceId**,
- wszystkich urządzeń określonego modelu (**vendor + model**) gdzie model może być określony za pomocą wyrażenia regularnego
- wszystkich urządzeń komunikujących się danym protokołem
- pozostałych urządzeń (interwał domyślny).

Obecne reguły:

Interwały dla podzbioru urządzeń:

Interval	DeviceIds
600s	EVB-P4562137, ALF-9571445, CS_7155_CGC100, EVB-P9287312, ALF-2844179
2700s	t53_8264_019, EVB-P15079256, EVB-P0984003, EVB-P1515640, EVB-P1515526

Interwały dla modeli:

Interval	Vendor	Model (regex)
60s	Alfen BV	NG920-5250[6-9]
60s	ChargeStorm ABI	Chargestorm Connected
120s	EV-BOX	G3-M5320E-F2.*

Interwał dla protokołu IoT2.0 wynosi: 600s

Interwał domyślny: 1800s

Przebieg ćwiczenia:

1. Zaimplementuj metodę **calculateInterval(Deviceish)** klasy **IntervalRules** enkapsulującą wszystkie reguły.
2. Implementację możesz zacząć od testów, zgodnych z opisem reguł w akapicie „Obecne reguły” sugerowane scenariusze:
Test dla `deviceish.deviceId = EVB-P4562137` oczekujemy 600s
Test dla `deviceish.deviceId = t53_8264_019` oczekujemy 2700s
Test dla `deviceish.vendor = ChargeStorm ABI` i `deviceish.model = Chargestorm Connected` wtedy 60s
Test dla `deviceish.vendor = EV-BOX` i `deviceish.model = G3-M5320E-F2-5872` oczekujemy 120s
3. By odizolować logikę wyliczania interwału od różnorodności protokołów komunikacyjnych, zaimplementuj Value Object (na potrzeby ćwiczenia nazwijmy go **Deviceish**), który posiada wyłącznie pola niezbędne do wyliczenia interwału.
4. Dodaj do obu klas **BootNotificationRequest** metodę, która wyprodukuje Value Object **Deviceish**. Do metody produkującej przekaz brakujące informacje jako parametry.

UWAGA w tym ćwiczeniu:

- Pomiń funkcjonalności związane z edycją / dodawaniem reguł przez administratora, skup się na wyliczaniu interwałów
- Pomiń warstwę persystencji oraz inne technologie

Ćwiczenie: Persystencja obiektu jako dokument

Zaimplementuj i przetestuj testem integracyjnym zapis oraz pobieranie reguł wyliczania interwału z poprzedniego zadania.

1. Zaimplementuj klasę **IntervalRulesRepository**, zaimplementuj jedynie metodę: `IntervalRules get()` nie przyjmującą argumentu
2. W implementacji klasy **IntervalRulesRepository** posłuż się poniższym kodem:

```
@Data
@Entity
@Table(name = "features_configuration")
class FeaturesConfigurationEntity {
    @Id
    private String name;

    @Type(type = "jsonb")
    @Column(columnDefinition = "jsonb")
    private IntervalRules configuration;
}
```

Persystencja obiektu jako JSON
w kolumnie typu binary json
w bazie Postgresql

```
public interface FeaturesConfigurationRepository    Interface repozytorium Spring Data
    extends CrudRepository<FeaturesConfigurationEntity, String> {
    Optional<FeaturesConfigurationEntity> findByName(String name);
}
```

3. Posłuż się wstępnie przygotowanym testem **IntervalRulesRepositoryTest** - dokończ test. Test wykorzystuje bazę Postgresql uruchamianą automatycznie w kontenerze Docker-a. Upewnij się, że na Twoim komputerze docker jest uruchomiony. Za zarządzanie kontenerem w trakcie testów odpowiada biblioteka <https://www.testcontainers.org>
4. Zadbaj o scenariusz w którym w bazie danych nie ma żadnej konfiguracji, w tym przypadku skonstruuj i zwróć domyślną konfigurację z domyślnym interwałem.

Ćwiczenie: REST dla prostej logiki CRUD

Zaimplementuj kontroler REST-owy i przetestuj testem Spring MockMvc endpoint REST-owy pozwalający na zapis oraz odczyt aktualnych reguł wyliczania interwału z poprzednich zadania (**IntervalRules**).

1. Zaimplementuj klasę **FeaturesConfigurationController** z pozwalającą na:
odczyt (Http GET method)
zapis (Http PUT method)
2. Sam zaproponuj strukturę ścieżki endpointu.
3. Przetestuj przypadek z przekazaniem błędnego wyrażenia regularnego dla reguły opartej o model urządzeń, zadбай by kod błędu zwracany w tym przypadku to BAD REQUEST (400).

Ćwiczenie: Modelowanie Aggregate i Value Objects

Rozszerz klasę **Device** pozwalającą docelowo konfigurować informacji o urządzeniu takich jak:

- lokalizacja urządzenia
- godziny dostępności urządzenia
- ogólne ustawienia
- oraz pozwala na przypisanie urządzenia do operatora urządzeń (**operator**) i dostawcy usługi (**provider**).

Ponad to klasa docelowo udostępnia informacje o aktualnych brakach i błędach w konfiguracji oraz o widoczności urządzenia dla klientów końcowych.

UWAGA agregat demonstrowany w tym ćwiczeniu to typowy model definicyjny (obiekty typu draft) i posiada następujące cechy:

- relatywnie prosta logika CRUD-owa, metody typu get / set
- miękka weryfikacja reguł podczas edycji:
 - złamanie reguł jest dopuszczalne
 - posiada ciągłą informację o aktualnych błędach, żądaniach lub sugestjach naprawy
- złamane reguły mogą blokować pewne funkcje, tutaj publikację / widoczność urządzenia dla klientów

Zapoznaj się z obecnym stanem klasy **Device** oraz **OpeningHours** i **Settings** oraz dodaj kolejne konfigurowalne informacje o urządzeniu:

1. Zaimplementuj klasę **Location** z polami tekstowymi street, houseNumber, city, postalCode, state, country oraz coordinates, gdzie **Coordinates** to klasa o polach longitude, latitude typu BigDecimal
2. Do klasy **Device** dodaj metodę updateLocation oraz pole location
3. Zaimplementuj klasę **Ownership** z polami tekstowymi operator i provider
4. Do klasy Device dodaj metodę assign(**Ownership**) oraz pole ownership

Dodaj weryfikację braków i błędów informacji oraz kalkulację widoczność urządzenia:

1. Zaimplementuj metodę getViolations zwracającą nową klasę **Violations** konstruowaną za pomocą wzorca budowniczy
2. **Violations** niech posiada serię pól typu Boolean dla każdego weryfikowanego błędu:
 - operatorNotAssigned
 - providerNotAssigned
 - locationMissing
 - showOnMapButMissingLocation
 - showOnMapButNoPublicAccess
3. W metodzie getViolations klasy **Device**, która przy każdym jej wywołaniu wylicz wszystkie weryfikowane błędy
4. Zaimplementuj klasę **Visibility** z polami forCustomer oraz roamingEnabled forCustomer to enum o 3 wartościach: USABLE_AND_VISIBLE_ON_MAP, USABLE_BUT_HIDDEN_ON_MAP, INACCESSIBLE_AND_HIDDEN_ON_MAP roamingEnabled to pole typu boolean
5. Do klasy **Device** dodaj metodę getVisibility zwracającą obiekt typu **Visibility**, która przy każdym jej wywołaniu wylicz widoczność według poniższych reguł:
 1. roamingEnabled = true kiedy nie ma błędów i braków w danych oraz settings.publicAccess == true
 2. analogicznie reguły muszą być spełnione by klient mógł użyć urządzenia część USABLE / INACCESSIBLE wartości enuma ForCustomer ponadto settings.showOnMap decyduje czy urządzenie jest pokazywane na mapie część VISIBLE_ON_MAP / HIDDEN_ON_MAP wartości enuma ForCustomer

Ćwiczenie: Implementacja REST i Application Service

Zaimplementuj kontroler REST-owy pozwalający na odczyt oraz edycję aktualnej konfiguracji urządzenia z poprzednich zadania (**Device**).

To API jest projektowane na potrzeby naszego zespołu i będzie wykorzystywane przez GUI webowe.

1. Zaimplementuj klasę **DevicesController** z pozwalającą na:
 - odczyt pojedynczego urządzenia (Http GET method)
 - odczyt z paginacją wszystkich urządzeń (Http GET method)
 - aktualizację konfiguracji pojedynczego urządzenia (Http PATCH method)

W tym ćwiczeniu pominiemy metody POST i DELETE ale jak najbardziej miały by one zastosowanie, metodę PUT wykluczamy a rolę edycji przejmie bardziej selektywny PATCH.

2. Sam zaproponuj strukturę ścieżki endpointu.

Na potrzeby odczytu konfiguracji:

3. Dodaj do klasy **Device** metodę toConfigurationSnapshot
4. Zaimplementuj klasę **DeviceSnapshot** posiadającą identyczne pola jak klasy **Device** oraz pola z **Violations** i **Visibility**
5. Uzupełnij metody odczytowe (GET) w **DevicesController**
Paginację zaimplementuj przy pomocy Pageable i Page ze Springa

Na potrzeby aktualizacji konfiguracji:

6. Zaimplementuj klasę **UpdateDevice** posiadającą identyczne pola jak klasy **Device**
7. Uzupełnij metodę PATCH w **DevicesController** używając **UpdateDevice** jako body requestu
8. Zaimplementuj klasę **DevicesService** posiadającą metodę update(**UpdateDevice**)
Jeżeli pole klasy **UpdateDevice** jest różne od null to wykonaj odpowiednią metodę aktualizującą na klasie **Device**, np:

```
if (updateDevice.location != null) device.updateLocation(updateDevice.location)
if (updateDevice.openingHours != null) device.updateOpeningHours(updateDevice.openingHours)
...
```

Dzięki temu umożliwimy patch-owanie selektywnie dowolnego pola lub ich grupy poprzez jeden endpoint z metodą PATCH