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 sugestiach 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:
   - operatorNotAssingned = this.**ownership == null ll this.ownership.operator == null**
   - 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 spełniony 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: Persystencja obiektu jako seria zdarzeń

Rozszerz klasę **Device** by konstruowała zdarzenia przy każdej modyfikacji swojego stanu, następnie każde zdarzenie zapisz w bazie danych. Ponad to zaimplementuj klasę **DeviceRepository,** która wczyta z bazy danych istotne zdarzenia i skonstruuje klasę **Device**:

1. W każdej metodzie klasy **Device**, modyfikującej stan urządzenia, skonstruuj odpowiednie zdarzenie mówiące o zmianie poszczególnych informacji. np: w metodzie updateLocation skonstruuj zdarzenie **LocationUpadated** posiadające id urządzenia oraz nową lokację
2. Do klasy **Device** dodaj pole package scope List<DomainEvent> events, gdzie DomainEvent to marker interfejs dla wszystkich typów zdarzeń
3. Zaimplementuj klasę **DeviceRepository** posiadającą dwie metody:
   - Device get(deviceId)
   - void save(Device)
4. Zaimplementuj encję **DeviceEventEntity** oraz repozytorium Spring Data JPA **DeviceEventRepository**.

```
@Data

@Entity

@Table(name = "device_events")

class DeviceEventEntity {

    @Id

    private UUID id;

    private String deviceId;

    private Instant time;

    @Type(type = "jsonb")

    @Column(columnDefinition = "jsonb")

    private DomainEvent event;
}
```

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

5. By Jackson potrafił odczytywać polimorficzne obiekty event-ów dodaj poniższe adnotacje do interfejsu **DomainEvnet:**

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME)
@JsonSubTypes({
        @JsonSubTypes.Type(value = LocationUpdated.class, name = "LocationUpdated"),
        ...
```

6. W metodzie save **DeviceRepository** wyjmij z instancji klasy Device listę wyemitowanych event-ów i zapisz każdy z event-ów w bazie.
7. W metodzie get **DeviceRepository** pobierz z bazy wszystkie eventy dla danego deviceId pogrupuj je po typie i wybierz ostatni event z każdego typu, pamiętaj że pewne eventy mogły nie występować inne wielokrotnie. Grupowanie „ostatnich” event-ów po stronie Postgresa:

```
@Query(value = "select distinct on (event ->> '@type') *" +
        " from device_events" +
        " where deviceId = :deviceId" +
        " order by event ->> '@type', time desc", nativeQuery = true)
List<DeviceEventEntity> findLastEvents(String deviceId);
```

8. Skonstruuj instancję Device przekazując do konstruktora wybrane wartości z ostatnich event-ów. Możesz też przekazać przez konstruktor całą listę event-ów lub posłużyć się fabryką czy budowniczym.

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 przejmuje bardziej selektywny PATCH.
2. Sam zaproponuj strukturę ścieżki endpointu.

Na potrzeby odczytu konfiguracji:

3. Dodaj do klasy **Device** metodę toSnapshot
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

   UWAGA na etapie tego ćwiczenia nie implementujemy wyraźnej separacji Odczytów od Zapisów (Device nie powinno być wykorzystywane przy odczycie w CQRS, ale w tym momencie pozwolimy sobie na to).

Na potrzeby aktualizacji konfiguracji:

6. Zaimplementuj klasę **UpdateDevice** posiadającą identyczne pola jak klasy **Device**
7. Uzupełnij metodę PATCH w **DevicesController** używając **UpdateDevice** jako body request-u
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:

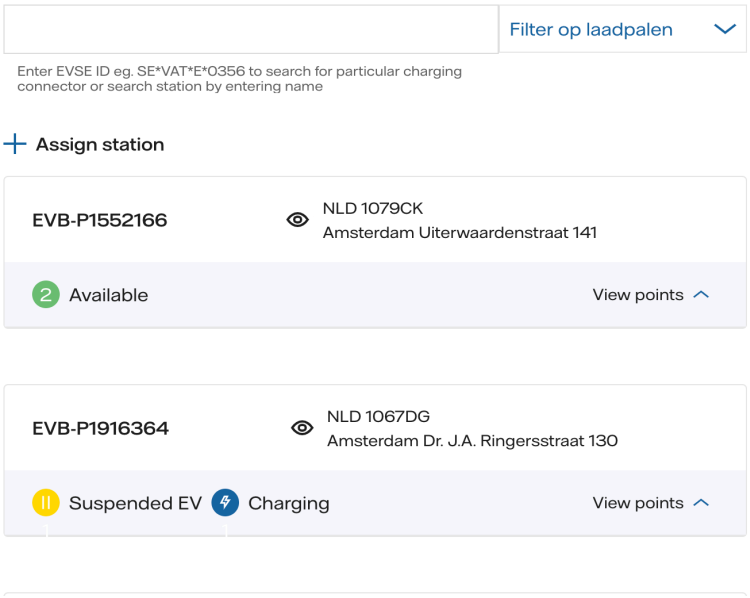
   if (updateDevice.location != null) device.updateLocarion(updateDevice.location)
   if (updateDevice.openingHours != null) device.updateOpeningHours(updateDevice.openingHours)
   …

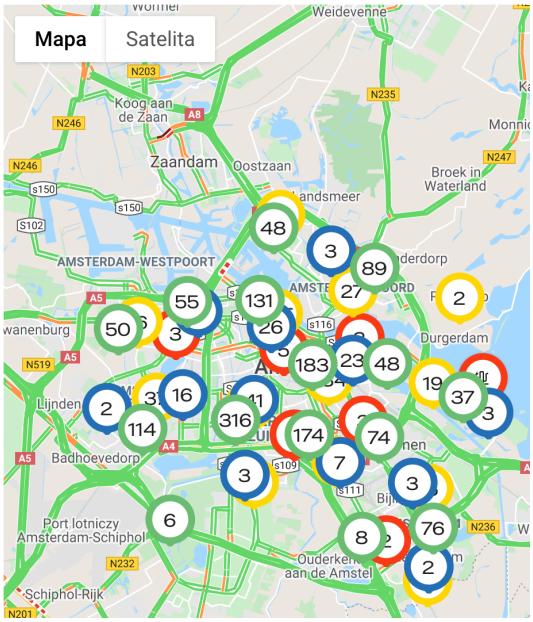
   Dzięki temu umożliwimy patch-owanie selektywnie dowolnego value objectu lub ich grupy poprzez jeden endpoint z metodą PATCH. W przypadku value objectu settings selektywność schodzi do poziomu pojedynczego pola.

   UWAGA użycie metody PATCH jest to przydatna technika przy edycji modeli definicyjnych (obiekty typu draft) jednak nie wszystkie obiekty mają tą cechę.

Ćwiczenie: Implementacja Read Modeli (CQRS)

Zaimplementuj persystenty model reprezentujący podstawowe informacje o urządzeniach na potrzeby listy urządzeń oraz mapy
To Read Model i API jest projektowane na potrzeby naszego zespołu i będzie wykorzystywane przez GUI webowe.





1. Zaimplementuj dodatkowe metody klasy **DevicesController** z pozwalającą na:
   odczyt z paginacją wszystkich urządzeń w skróconej postaci **DeviceSummary** (Http GET method)
   odczyt wszystkich urządzeń na potrzeby pinu na mapie **DevicePin** (Http GET method)

   UWAGA na potrzeby ćwiczenia pomijamy kwestie security i filtrowania rekordów dostępnych dla użytkownika - udostępniamy zawsze wszystkie urządzenia.
2. Sam zaproponuj formę mapowania requestu na właściwy endpoint np:
   po query params: ?view=pins ; ?view=summary
   po mime type: application/vnd…pins+json ; application/vnd…summary+json
3. Zmodeluj klasę **DevicePin** jako deviceId, coordinates, statuses
   gdzie statuses to List enumów AVAILABLE, CHARGING, FAULTED
4. Zmodeluj klasę **DeviceSummary** jako deviceId, location, statuses
   gdzie statuses to List stringów (by pokazać 1-1 wartości przychodzących z urządzenia)
5. Stwórz encję DeviceReadModels z polami:
   String deviceId;
   long version;
   String operator;
   String provider;
   DevicePin pin;
   DeviceSummary summary;
   DeviceSnapshot details;
   oraz spring data repository do niej
6. Uzupełnij odpowiednio metody **DevicesController**
7. Zadbaj by zmiany na obiekcie Device wpłynęły na read modele.
8. Zmiany statusów pochodzą bezpośrednio z urządzenia (pakiet remote)
   na potrzeby zadania pomijamy tą część problemu.

Ćwiczenie: Api dla innych Bounded Context-ów

Zaimplementuj notyfikację za pomocą Kafki o zmianach konfiguracji Device.
To API jest projektowane na potrzeby innych zespołów i musi trzymać kompatybilność wsteczną / być wersjonowane.

1. Do nowego pakietu np. published skopiuj klasę **DeviceSnapshot** jako **DeviceSnapshotV1** oraz wszystkie klasy wchodzące w jej skład jak **Location**, **Ownership**, **OpeningHours**, **Visibility**, umieść te klasy jako static inner class wewnątrz **DeviceSnapshotV1** i do ich nazwy dodaj sufix Snapshot pomijamy składowe **Settings** oraz **Violations** one nie są publikowane w API
2. Do klasy **DeviceSnapshotV1** dodają metodę statyczną:
   public static ConfigurationSnapshotV1 from(DeviceSnapshot)
   w której przekopiuj w głąb wszystkie wymagane pola
   dla wygody i poprawy wyglądu kodu dodaj analogiczne metody w poszczególnych klasach:
   public static LocationSnapshot from(Location)

By udostępnić API jako event driven:

3. W każdej metodzie, jeśli zmienił się stan **Device** utwórz właściwy event i dodaj do kolekcji events.
4. Utwórz klasę **DomainEventListener** z metodą:

```
@TransactionalEventListener

public void publish(DeviceSnapshot event) {

    // topic, key, payload

   kafka.send(topic, key, payload)
      .addCallback(
          success -> {},
          exception -> log.error(
          "Could not send message topic: {}, key: {}, payload: {}",
          topic, key, payload, exception)

      );

}
```

UWAGA jeśli topić jest Kafki kompaktowany można uniknąć udostępniania API REST-owego dla innych microserviceów.