Exercise: Aggregate i Value Objects

Extend **Device** class for enabling configuration of following information about device:
- The location of the device
- Opening hours of location (availability of device)
- Generał settings
- Operator and provider of device.

Moreover Device class should makes available information about currently missing or inconsistent informations and settings and visibility and availability of device for end customers.

NOTE aggregate demonstrated in that exercise is typical definition (draft) model and has the following features:
- Relatively easy CRUD like logic, methods get / set for value objects included
- Soft verification of rules while editing data:
  - Protected rules can be broken after edit
  - Provides continuous information about broken rules or suggestion of fixes
- Broken rules can lock some functionality, in that example visibility and availability of device for end customers.

Other example of definition (draft) model: Shop Cart

Examine current implementation of **Device**, **OpeningHours** and **Settings** classes add another configurable information about device like:

1. Implement **Location** class with following properties
   street, houseNumber, city, postalCode, state, country
   and coordinates, where **Coordinates** class has following properties: longitude, latitude of type BigDecimal
2. To **Device** class add method updateLocation and location property
3. Implement **Ownership** class two with string properties operator i provider
4. To **Device** class add method assign(**Ownership**) and ownership property

Implement calculation of missing or inconsistent informations and visibility / availability of device:

1. Implement method of getViolations returning new class **Violations** constructed with Builder pattern
2. **Violations** should contain fields of type boolean for each verified rule:
   - operatorNotAssingned calculated as **this.ownership == null ll this.ownership.operator == null**
   - providerNotAssigned calculated as **this.ownership == null ll this.ownership.provider == null**
   - locationMissing calculated as **this.location == null**
   - showOnMapButMissingLocation calculated as **this.settings.showOnMap && this.location == null**
   - showOnMapButNoPublicAccess **this.settings.showOnMap && ! this.settings.publicAccess**
3. In method getViolations of **Device** class, calculate all rules and build and return **Violations** instance
4. Implement **Visibility** class with properties forCustomer and roamingEnabled
   forCustomer define as enum of 3 values: USABLE_AND_VISIBLE_ON_MAP, USABLE_BUT_HIDDEN_ON_MAP, INACCESSIBLE_AND_HIDDEN_ON_MAP
   roamingEnabled should have boolean type
5. Add method getVisibility to **Device** class calculate and return **Visibility** instance,
   rules to calculate **Visibility**:
   1. roamingEnabled = true if there is no violations and settings.publicAccess == true
   2. Same rule need to be meet to altowe customer to use device
      portion USABLE / INACCESSIBLE of ForCustomer enum values
      moreover settings.showOnMap influence portion VISIBLE_ON_MAP / HIDDEN_ON_MAP of ForCustomer enum values

Exercise: Persistence as series of events

Extend **Device** class to create and emit event by each data modification, after that persist each event in database. Implement **DeviceRepository** class, responsible for persisting and fetching events from database and creation of **Device** instances:

1. In each method modifying state of **Device** class, instantiate proper domain event informing about change of particular information ex: in method updateLocation create **LocationUpadated** event with id of device and new value of location
2. To **Device** class add package scope field List<DomainEvent> events, where DomainEvent is a marker interface of all domain events
3. introduce **DeviceRepository** class with two methods:
   - Device get(deviceId)
   - void save(Device)
4. Implement entity **DeviceEventEntity** and 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;

}
```

Persistence of object as JSON
in column of type JSONB (binary JSON)
in Postgresql database

5. To instruct Jackson library how to create proper concrete (polymorphic) instance of domain event annotate **DomainEvnet** class as follows:

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

6. In save method of **DeviceRepository** get from **Device** instance a list of emitted domain events and persist them in database.
7. In get method of **DeviceRepository** fetch all (relevant) events for given deviceId. In can be done in few different ways depending of structure of events and persisted class, ex. **Device** instance can be recreated based on last event of each type.
   You can fetch tem with postgres native query:

```
@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. Create instance of **Device** based on fetched events, you can give full list of events to Device class or fold left events list and create Device with natural all argument constructor.

Exercise: REST & Application Service (without CQRS)

Implement REST controller enabling read and edit of current **Device** configuration.
This API is designed only for usage internally by Web based UI developed inside team (not external API).

1. implement **DevicesController** class with following functionality:
   Read single Device resource (Http GET method)
   Read with pagination of all Device resources (Http GET method)
   Update of configuration of single Device (Http PATCH method)

   In that exercise we skip usage of POST and DELETE methods, but they for sure can by applied for creation and deletion of device, PUT method is replaced by edits with more fine-grained PATCH method.
2. Propose path structure of that endpoint.

To expose Reads methods:

3. Add method toSnapshot in **Device** class
4. Implement class **DeviceSnapshot** with identical fields like **Device** plus fields of types **Violations** and **Visibility**
5. Implement read method (GET) w **DevicesController** by fetching **Device** instances and converting them to snapshot. Use Pageable and Page classes from Spring to implement pagination in easy way.

   NOTE that exercise is not showing any separation from Reads and Writes (Device should be not fetched from database in case of CQRS reads).

To expose updates of **Device**:

6. Implement (command) **UpdateDevice** with identical fields like **Device**
7. Implement PATCH method i **DevicesController** taking **UpdateDevice** as body of request
8. Implement class **DevicesService** with method update(**UpdateDevice**)
   With repeatable logic like: If field of **UpdateDevice** is not null, them execute updateXYZ method on **Device**, ex:

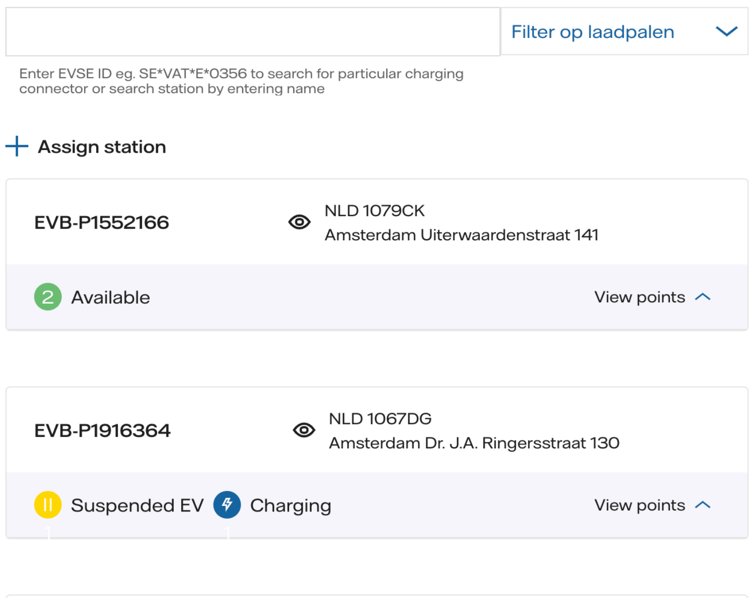
   if (update.location != null) device.updateLocarion(updateDevice.location)
   if (update.openingHours != null) device.updateOpeningHours(updateDevice.openingHours)
   …

   With that pattern UI can selectively change any value object of device configuration or subgroup of them depending on UI design. There is no need to expose multiple endpoints.
   In case of **Settings** value object updates are even more fine-grained, by merging old and new **Settings** instances.

   NOTE that PATCH pattern is very useful with definition type models (draft objects) but not all domain objects are of that kind.

Exercise: Read Models (CQRS)

Implement persistned model representing basic informations about device needed to present list of devices and present them on map.
This API is designed only for usage internally by Web based UI developed inside team (not external API).



Filter op laadpalen

Enter EVSE ID eg. SE*VAT*E*0356 to search for particular charging connector or search station by entering name

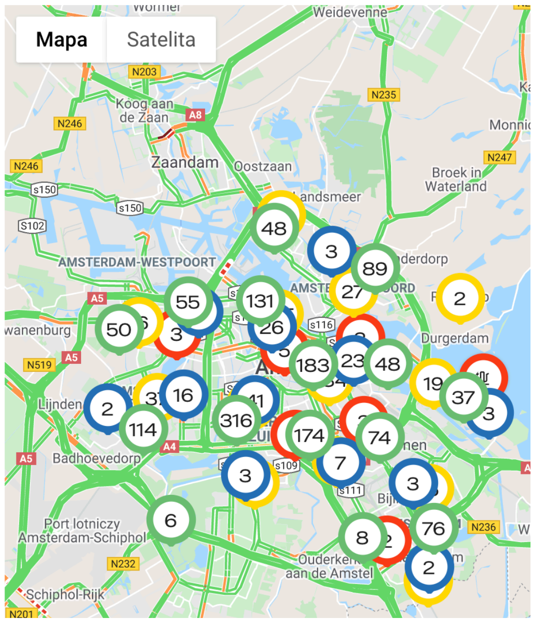
+ Assign station

EVB-P1552166 — NLD 1079CK Amsterdam Uiterwaardenstraat 141

2 Available — View points

EVB-P1916364 — NLD 1067DG Amsterdam Dr. J.A. Ringersstraat 130

Suspended EV — Charging — View points



1. Implement additional methods to **DevicesController** enabling read:
   - Paginated read of all devices in shorter form of **DeviceSummary** (Http GET method)
   - read of all devices form of **DevicePin** (Http GET method)

   NOTE in that exercise we skip security or filtering of records for given user - we expose all devices.
2. Propose form of request mapping to proper controller method alternatively:
   with use of query params: ?view=pins ; ?view=summary
   with use of mime type: application/vnd…pins+json ; application/vnd…summary+json
   or other
3. Introduce **DevicePin** class as deviceId, coordinates, statuses
   where field statuses is List or enums like: AVAILABLE, CHARGING, FAULTED
4. Introduce **DeviceSummary** class as deviceId, location, statuses
   where field statuses is List of strings (to present 1-1 values coming form device)
5. Create JPA entity **DeviceReadModels** with fields:
   String deviceId;
   long version;
   String operator;
   String provider;
   DevicePin pin;
   DeviceSummary summary;
   DeviceSnapshot details;
   and Spring Data JPA repository for that entity
6. Implement new methods in **DevicesController**
7. Ensure any change of **Device** will influence value of read models.
   Most effective approach is to emit **DeviceSnapshot** from **DeviceRepository**.save(Device) method in case of any change on **Device**.
8. Changes of statuses are coming directly from physical devices (package remote)
   we skep that issue in scope of that exercise.

Exercise: Api for external Bounded Context-ów

Implement event publication to Kafki after every change of **Device**.
This API is designed for other teams and it needs to keep backward compatibility and be versioned.

1. Create new package published copy there **DeviceSnapshot** class as **DeviceSnapshotV1** and all classes used in previous snapshot like: **Location**, **Ownership**, **OpeningHours**, **Visibility**, put all those classes as static inner classes inside **DeviceSnapshotV1** include Snapshot suffix to names of inner classes.
   Skip information of **Settings** and **Violations** this informations are not published over API
2. To **DeviceSnapshotV1** class add static method:
   public static ConfigurationSnapshotV1 from(DeviceSnapshot)
   do deep copy of all published fields from **DeviceSnapshot** to **DeviceSnapshotV1**
   For convenience and code clarity add analogical „from" factory methods in each inner class ex.:
   public static LocationSnapshot from(Location)

To expose API as event driven:

3. Create class **DomainEventListener** with method:

```
@EventListener

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)

      );

}
```

NOTE if Kafka topic is compacted we can skip entirely API REST for other teams using Kafka.