TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Madis Jaagup Laurson

Asjade Interneti API loomine targa linna andmete halduseks

Bakalaureusetöö (9 EAP)

Juhendaja:
Pelle Jakovits, PhD

Tartu 2025

Asjade Interneti API loomine targa linna andmete halduseks

Lühikokkuvõte:

Töö eesmärk oli luua targa linna andmete haldamise platvorm. Töö käigus analüüsiti erinevaid andmebaaside ja veebiraamistike võimalusi rakenduse loomiseks. Lõpptulemusena valmis FastAPI'l ja QuestDB'l põhinev rakendus, millega on võimalik seadmeid ja kasutajaid hallata, seadmete poolt saadetud sündmuseid ja mõõtetulemusi vastu võtta ja neid hiljem pärida. Arutleti selle üle, miks osa nõuetest täitmata jäi, kuidas ja kas üldse neid täita. Rakendus on Tartu linna poolt kasutusele võetud. See aitas lisaks lokaalsetele testidele valideerida, et rakendus töötab ja tõi välja tekkinud murekohad, mille peale arendusprotsessi käigus ei tuldud.

Võtmesõnad: Tark linn, API, aegread, QuestDB, FastAPI

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Designing Internet of Things API for Smart City Data Management

Abstract:

The objective of the thesis was to create a smart city data management platform. It includes the analysis of various databases and web frameworks possibly suitable for the project. As a final result, an application was created using FastAPI and QuestDB that enables managing devices and users, receiving events and measurements sent by the devices, and querying them. There was also a discussion about the unfulfilled requirements, whether and how they should be fulfilled. The city of Tartu is already using the application. In addition to local tests, this helped to validate that the application is working as intended. It also brought light to the problems that were not noticed during the development process.

Keywords: Smart city, API, time series, QuestDB, FastAPI

CERCS: P170 Computer science, numerical analysis, systems, control

Sisukord

1. Sissejuhatus

Tark linn (ingl *smart city*) tähendab linnapiirkonda, kus kasutatakse tehnoloogiat selle elanike heaoluks. Üheks selliseks targa linna tehnoloogiliseks näiteks võiks olla busside jälgimise süsteem. Kui buss jõuab peatusesse, saadetakse teade, millega pannakse kaasa kellaaeg, peatuse nimi, bussi number ja reisijate arv. Selle põhjal on võimalik vaadata, kas buss püsib graafikus. Terve linna peale võib erinevaid sensoreid olla tuhandeid ja ka info saatmise sagedus võib olla oluliselt suurem, kui bussi puhul kord iga mõne minuti tagant.

Andmeid kogutakse väga erinevates valdkondades. Lisaks eelmainitud busside poolt saadetud infole loendatakse näiteks jalakäijaid ja sõidukeid, mõõdetakse temperatuuri ja mürataset. See omakorda tähendab, et andmeid liigub suurtes kogustes ja tihti. On olemas platvormid, mis on loodud täpselt selliste juhtudeks. Näiteks Tartu linnal oli varem kasutusel Cumulocity platvorm.

Cumulocity puhul on ilmnenud kitsaskoht, et keerukate andmepäringute sooritamiseks tuleb andmed kõigepealt andmebaasist välja võtta ja alles siis saab neid agregeerida. Lisaks on andmemahtude kasvades tekkinud andmekvaliteedi probleemid.

Linn soovib kasutusele võtta uue andmebaasi, aga sellele on vaja luua API ehk rakendusliides, mis teeb Tartu linna andmete saatmise ja integratsioonide loomise lihtsaks. Töö üheks eesmärgiks on luua rakendus, mis saab hakkama vähemalt sama andmete ja päringute mahuga, millega Cumulocity. Tähtis on ka see, et loodav rakendus toetaks eelneva platvormi poolt kasutatava struktuuriga andmete vastuvõtmist.

Suurem osa rakendusele seatud soovidest ja eesmärkidest tuligi Tartu linnavalitsuse poolt, sest nemad hakkavad seda haldama. Kasutajateks on firmad, kellel on Tartu linnaga leping ja kes hakkavad platvormi kasutama targa linna andmete edastamiseks Tartu linnavalitsusele. Osa nõuetest oli paigas juba hanke koostamisel. Lisaks korraldati üle nädala linnavalitsuse esindajaga koosolekuid, et jooksvalt tulemusi tutvustada ja vajadusel nõudeid täpsustada. Peamiselt vedas neid koosolekuid juhendaja, kuid autor võttis neist tihti osa, et vajadusel kaasa rääkida ja paremini kliendi soove mõista.

Andmed on olemuselt aegread (ingl *time series*) ehk andmepunktide kronoloogilised jadad[1], seega saab neid salvestada kronoloogilises andmebaasis. Selles töös on kasutusel QuestDB, mis on tuntud sisestuskiiruse poolest, mis on selle rakenduse puhul hädavajalik.

Andmebaasiga suhtlemiseks on vaja APIt ehk rakendusliidest[1], mis saaks ülalmainitud mahtude vahendamisega hakkama. Esialgne API arhitektuur pandi juhendaja poolt kirja OpenAPI spetsifikatsioonina. Sellise lahenduse loomiseks on väga laialt levinud variandid Pythoni raamistikud Flask ja Django. Viimastel aastatel on populaarsust kogunud kolmas Pythoni raamistik FastAPI. See pakub mugavat ja kiiret arenduskogemust. Pythoni kasuks räägib veel see, et seda kasutatakse näiteks aines Veebiteenuste ja hajussüsteemide arendus LTAT.06.018, mis võimaldab tulevikus selle lõputöö põhjal koostada aine jaoks praktikumiülesandeid.

Arendusprotsess algab juhendaja poolt OpenAPI spetsifikatsiooni prototüübi loomisega, mida autor edaspidi muudab ja täiendab. Arenduse käigus muutuste tekkimisel uuendatakse kõigepealt OpenAPI spetsifikatsiooni, seejärel genereeritakse FastAPI kood ja viiakse vajadusel sisse genereerimisel tekkinud vigade parandused. Siis kirjutatakse kood uue funktsionaalsuse tarbeks, autor testib seda enda arvutis ja seejärel testib juhendaja seda testserveris. Kui kõik on sobiv, asendatakse päriselt kasutatav rakendus uue versiooniga. Kui lõppkasutaja või juhendaja avastab rakenduse töös vigu, antakse neist autorile teada ja ta viib sisse parandused.

Töö jaguneb kolmeks sisupeatükiks. Esimeses neist tutvustatakse lugejale targa linna olemust, töö mõistmiseks vajalikke termineid, eelnevat lahendust ja olulisemaid töös kasutatavaid tehnoloogiaid. Teises sisupeatükis kirjeldatakse nõudeid, mis on rakendusele seatud, tuuakse välja kõik tehnoloogiad, mida kasutatakse, antakse ülevaade loodud andmemudelitest ja loodava rakenduse arhitektuurist. Kolmas sisupeatükk keskendub valminud rakendusele, jõudluse võrdlusele eelmise lahendusega, täitmata nõuete puhul selgitustele, miks need täitmata jäid ja võimalikele edasiarendustele. Töö lõpeb kokkuvõttega. Autor kinnitab, et ei ole töös kasutanud tehisintellekti.

[1]Andmekaitse ja infoturbe leksikon (AKIT). https://akit.cyber.ee/.

2. Tausta ülevaade

Käesolev peatükk annab ülevaate töös kasutatavatest mõistetest, töö teoreetilisest taustast, sarnases valdkonnas või sarnase eesmärgiga kirjutatud töödest ja olemasolevatest lahendustest.

2.1 Tark linn

IBM'i poolt avaldatud tarka linna tutvustavas blogipostituses [1] on kirjutatud järgmiselt (originaaltekst on inglise keeles): "Tark linn on linnapiirkond, kus tehnoloogia ja andmete kogumine aitavad parandada nii elukvaliteeti kui ka linna toimimise jätkusuutlikkust ja tõhusust."Linnaelaniku jaoks tähendab see seda, et ta näeb linna peal tähelepanelikult ringi vaadates erinevaid andureid ja kaameraid. See pole iseenesest midagi uut, sest avalikus ruumis on turvakaamerad väga tavaline ja vana nähtus, kuid neid andureid ja kaameraid kasutatakse peaasjalikult statistikate ja analüüside koostamiseks. Näiteks loendatakse sõidukeid, et aru saada, millistel tänavatel ja millal on kõige suurem liiklustihedus. Sarma ja Sunny täpsustavad enda töös [2], et targa linna kasutamise levinud eesmärke on majanduskasvu ja sotsiaalne arengu saavutamine.

Eestis on üheks targa linna näiteks Ülemiste City, kus tehakse koostööd paljude erinevate ülikoolidega Eestist ja mujalt, et lahendada linnakeskkonna planeerimise ja mobiilsuse probleeme[2]. Teine arvestatav näide on Tartu linn[3], kus keskendutakse ökoloogilise jalajälje vähendamisele. Näiteks renoveeritakse selle saavutamiseks SmartEnCity projekti raames maju, võetakse kasutusele nutikodu lahendusi ja suurendatakse elanike teadlikkust, et uuenduslikest süsteemidest maksimaalselt kasu oleks[3] (vt joonis 1). Järgmises alapeatükis kirjutataksegi lähemalt sellest, millist tüüpi andmeid Tartu linn kogub.

[2]Ülemiste City. https://www.ulemistecity.ee/tark-linn/.
[3]Tark Tartu. https://tarktartu.ee/.

Joonis 1. Tark Tartu pilootala kaart [3].

2.2 Mõisted

Selles alapeatükis seletatakse lahti mõisted, millel on käesoleva töö raames mingi kindel tähendus, mis võib erineda selle igapäevasest tähendusest, ja mida teadmata on raske töö sisu mõista.

Seade on üksus, mis saadab API kaudu sündmuseid ja/või mõõtetulemusi andmebaasi. Tartu puhul on need näiteks sensorid, mis mõõdavad temperatuuri, loendavad jalakäijaid või sõidukeid. Seadmeid saab olla ühes kohas mitu tükki ja sama funktsionaalsusega seadmeid saab olla erinevates kohtades. Ainuke piirang, mida nad üksteisele seavad, on see, et seadmete identifikaatorid peavad olema süsteemis unikaalsed. Cumulocity platvormi seadme struktuur on välja toodud koodinäites 1.

```
{
  "creationTime": "2019-06-28T09:24:39.543+03:00",
  "type": "DAL_series_target",
  "lastUpdated": "2023-08-22T11:05:46.742+03:00",
  "name": "11 AVC controller (Lammi)",
  "self": "http://tartu.platvorm.iot.telia.ee/inventory/"
    "managedObjects/117925746",
  "id": "117925746",
  "c8y_Availability": {
    "lastMessage": "2025-02-21T14:58:13.505+02:00",
    "status": "UNAVAILABLE"
  },
  "cepDirection_Lane1": "out",
  "dal_live": 2586,
  "c8y_Position": {
    "lng": 26.77393,
    "lat": 58.362385
  },
  "cep_correspondingMeasurementsDeviceId": 73157772,
  "dal_location": {
    "address": null,
    "lng": 26.77393,
    "groupingTag": "LIIKLUSLOENDURITE SÜNDMUSED (EVENTS)",
    "adsOid": null,
    "type": "COORDS",
    "uuid": null,
    "apartment": null,
    "room": null,
    "lat": 58.362385
  },
  "cepDirection_Lane2": "in"
}
```


Koodinäide 1. Seadme andmed Cumulocity APIs.

Tegemist on liiklusloenduriga, mis käesoleval juhul loendab kindlatel sõiduradadel liiklevaid sõidukeid.

Mõõtetulemus on tavaliselt perioodiliselt saadetav JSON-objekt, mis sisaldab mõõtmise aega, seadme identifikaatorit, mõõtetulemuse tüüpi ja fragmente. Fragment on sisuliselt mõõdetavata väärtuse tüüp, mõõtühik ja mõõdetud väärtus. Koodinäite 2 puhul on seadme ID (identifikaator) 681200. See seade saadab näiteks mõõtetulemusi, mis paigutuvad temperatuuri kategooriasse. Käesoleval juhul on tegemist fragmendiga, mis sisaldab näiteks mingi vabriku seadmete puhastuseks mõeldud auru kohta käivaid omadusi. Mõõdetakse selle temperatuuri ja sensori asukohas olevat niiskust.

```
{
    "source": {
        "id": "681200"
    },
    "time": "2020-03-19T12:03:27.845Z",
    "type": "temperatureMeasurement",
    "c8y_SteamFragment": {
        "Temperature": {
            "value": 42.7,
            "unit": "C"
        },
        "Humidity": {
            "value": 13.37,
            "unit": "%RH"
        }
    }
}
```

Koodinäide 2. Mõõtetulemuse andmed Cumulocity APIs [4].

Sellise lähenemise probleem on see, et võtmed ise on ka väärtused, näites *c8y_SteamFragment* ja *Temperature*. Süsteemi vaates on see loogiline, sest *c8y_SteamFragment* on fragment ja *Temperature* üks selle fragmendi raames mõõdetav väärtus ehk moodustub hierarhia, mille alusel hiljem andmeid pärida. Samas eksisteerib veel väärtus *type*, mis peaks ka midagi tähendama. See peaks määrama suure kategooria, mille alla mingid fragmendid kuuluvad, kusjuures dokumentatsioonis ei tundu selle kohta üldse mingeid täpsustusi olevat. Kõik see annab arendajale suure vabaduse (ja kohustuse) määrata, milline peaks olema tüüp, milline fragment ja mida selle fragmendi raames mõõdetakse.

Sündmus on JSON-objekt, mis saadetakse mingi välise teguri muutumisel. See sisaldab seadme identifikaatorit, sündmuse tüüpi, sündmuse aega ja vabateksti, kus saab näiteks sündmust lisaks täpsustada. Cumulocity platvormi sündmus on välja toodud koodinäites 3.

```
{
    "source": {
        "id": "117925736"
    },
    "type": "avc_VehicleEvent",
    "time": "2025-02-08T07:06:52.870Z",
    "text": "Vehicle detected",
    "avc_Vehicle": {
        "vehicle_class": "car",
        "occupancy": 2,
        "gap": 12220,
        "length": 12.9,
        "lane_id": 2,
        "vehicle_id": 123,
        "speed": 70
    }
}
```

Koodinäide 3. Sündmuse andmed Cumulocity APIs [4].

Koodinäite 3 puhul on tegemist seadmega, mille ID on 117925736 ja see saadab teate iga kord, kui tuvastab sõiduki. Sündmuse tüüp on seega põhimõtteliselt sõiduki tuvastamine ja vabatekst annab selle kohta inimsõbralikuma kirjelduse. Lisaks on kirjas, et tuvastati auto ja info sõiduki kohta, näiteks et kiirus on 70 km/h ja selles on 2 reisijat.

2.3 Eelnev lahendus

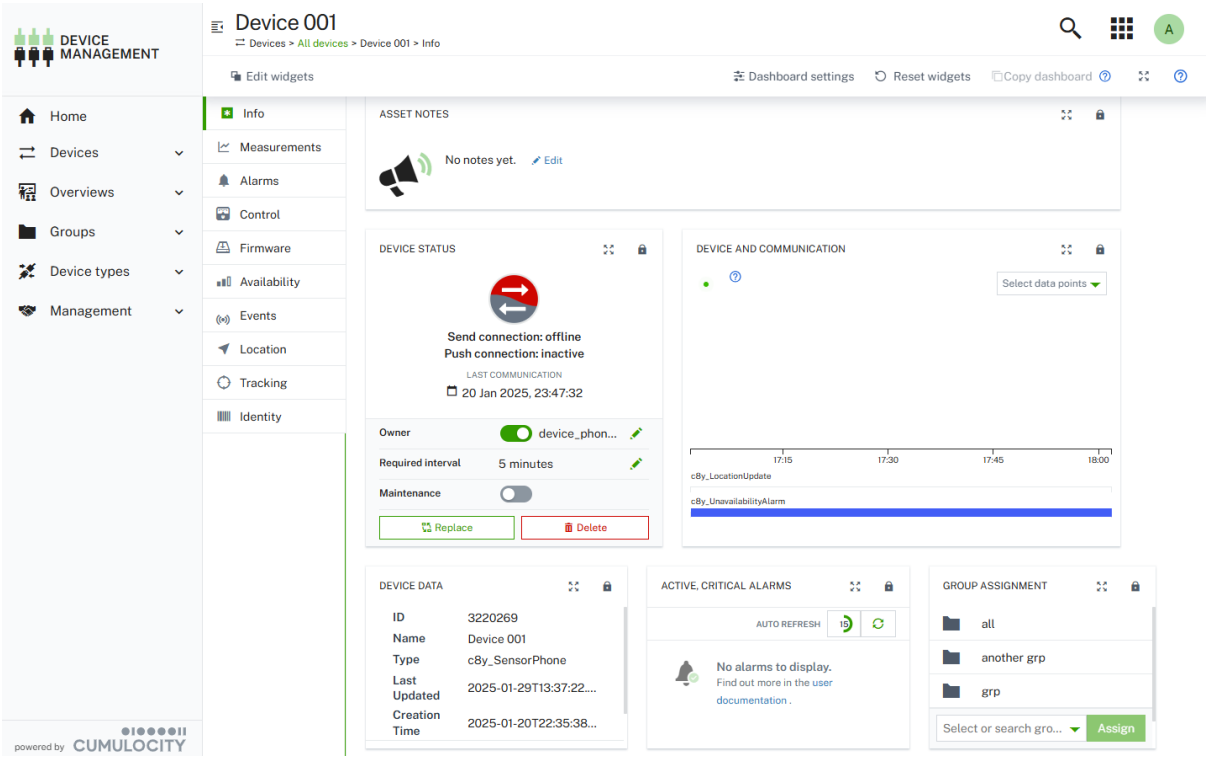
Cumulocity[4] on IoT platvorm, mida Tartu linn enne selle töö raames arendatavat rakendust kasutas. Sellega on võimalik seadmeid hallata ja seirata, hallata nende tarkvara. Platvorm võimaldab seadmete poolt saadetud andmeid analüüsida ja luua reegleid, et mingitel tingimustel

[4]Cumulocity. https://www.cumulocity.com/.

saadetakse teateid, näiteks antakse kellelegi teada, kui mingi seade on 10 minuti jooksul saatnud ohtlikult kõrgeid temperatuure.

Mitmekülgsust lisab võimalus jooksutada platvormi pilves või enda serveris. Cumulocity pakub HTTPS ühendust seadmega, et tagada andmete saatmise turvalisus. Platvormi on võimalik kiiresti kasutama hakata, nende enda väitel peaks äriliselt kasutatava lahenduse valmis saama vähem kui 90 päevaga. Leier toob välja [5], et Cumulocity eelis on veel see, et see võimaldab luua erinevate projektide tarbeks erinevaid vaateid vastavalt sellele, mis on oluline.

Cumulocity pakub väga mugavat kasutajaliidest, kus ühele ekraanile on paigutatud näiteks kõik ühte seadet puudutav (vt joonis 1). Seal saab hallata seadme tarkvara, seadistada ja vaadata teateid. Cumulocity pakub ka põhjaliku dokumentatsiooniga [4] API võimalust, et seadmed saaksid saata näiteks eelmises alapeatükis mainitud mõõtetulemusi ja sündmusi. Samuti saab selle kaudu teha paljusid toiminguid, mida võimaldab graafiline kasutajaliides.



Joonis 2. Cumulocity graafiline kasutajaliides [6].

2.4 Pythoni veebiraamistikud

Pythoni API-raamistikest on pikka aega olnud väga populaarsed Flask ja Django [7–9]. Flask on raamistik, mis võimaldab luua veebirakendusi. Ghimire sõnul ei kasuta see ise ühtegi turvalisuse

või andmebaasioperatsioonide lihtsustamise jaoks loodud teeki [10]. Küll aga antakse turvalisuse tagamiseks juhiseid[5]. Seega on arendajal vabadus vajalikud vahendid ise valida.

Django pakub ametlikku tuge erinevatele andmebaasidele, nagu näiteks PostgreSQL [10]. Seega on selles vallas rohkem tööd arendaja eest ära tehtud. Samuti annavad nad ise soovitusi rakenduste turvamiseks[6]. Samas paistab nende kõrval silma FastAPI. Kõigis kolmes edetabelis on kolm parimat just need raamistikud.

FastAPI on Pythoni raamistik, mis selle arendajate sõnul paistab välja jõudluse poolest[7]. See on oluline, sest targa linna seadmeid on palju ja need saadavad palju andmeid (näiteks iga kord, kui buss jõuab peatusesse).

G. De Luca kirjutab enda FastAPI-t õpetava raamatu alguses järgmiselt: ”FastAPI paistab silma arenduskiiruse- ja mugavuse ning põhjaliku dokumentatsiooni poolest” [11:1]. Lubanovic demonstreerib koodinäites 4, et viie koodireaga on võimalik luua rakendus, millega saab suhelda.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet():
        return "Hello? World?"
```

Koodinäide 4. Lubanovici lühike töötav FastAPI rakendus [12:28].

Dokumentatsiooni jaoks on kasutusel OpenAPI standard[8] (vt Lisa 1), mis selle eestvedajate sõnul aitab ühtlustada API rakenduste kirjeldamist ja ei nõua uue rakenduse mõistmiseks näiteks selle lähtekoodi lugemist. FastAPI lehel on välja toodud, et seda dokumentatsiooni genereerib nende raamistik automaatselt [7].

[5]Flask - Security Considerations. https://flask.palletsprojects.com/en/stable/web-security/.
[6]Security in Django. https://docs.djangoproject.com/en/5.2/topics/security/.
[7]FastAPI. https://fastapi.tiangolo.com/.
[8]OpenAPI. https://www.openapis.org/.

Andmete struktuuri valideerimiseks on FastAPI valinud Pydantic raamistiku ja kasutab seda automaatselt [13]. See on hea, sest üldjuhul on võimalik defineerida andmemudel ja FastAPI kontrollib ise Pydanticu abil iga sissetuleva päringu struktuuri ja viskab erindi, kui see ei vasta defineeritud mudelile. Andmete struktuuri valideerimisest ja võimalikest eranditest on pikemalt kirjutatud peatükis 4.3.

Talitlust iseloomustavaid andmeid (ingl *metrics*) on võimalik edastada Prometheusse[9]. Prometheus võimaldab neid andmeid koguda ja salvestada. Lisaks saab andmeid visualiseerida nii nende enda kasutajaliidese kui ka teiste selleks ette nähtud rakenduse, nagu näiteks Grafana[10], abil.

2.5 QuestDB

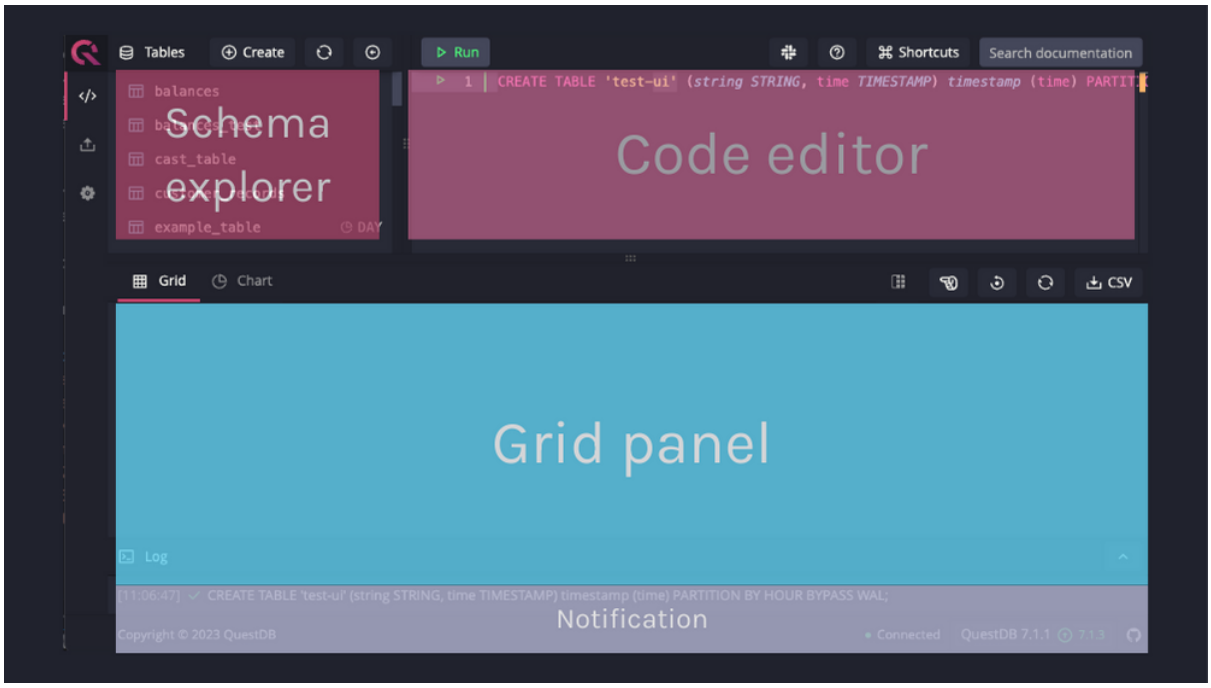
QuestDB on avatud lähtekoodiga kronoloogiline andmebaas (ingl *time series database*), mis toetab selle arendajate enda poolt kohandatud SQL (ingl *structured query language*) päringuid, et arendajatel oleks mugavam andmebaasi kasutada[11].

QuestDB ametlik dokumentatsioon on väga põhjalik, mis teeb selle kasutamise võrdlemisi lihtsaks. Andmebaasiga suhtlemisel on kasutusel PostgreSQL'le sarnane keel [14], mida on natuke vastavalt aegridade vajadustele muudetud. Seetõttu on sellel oma "veidrused", mille peale on alguses raske tulla, olles varem kasutanud peamiselt relatsioonilisi andmebaase. Ühe sellise näitena võiks välja tuua andmete kustutamise. Selle jaoks pole olemas eraldi käsku DELETE, nagu võiks eeldada. Samas on dokumentatsioonis selle jaoks mitu alternatiivi, mis annavad samasuguse või sarnase tulemuse [15]. Lisaks on samas dokumendis välja toodud, et QuestDB eesmärk on pakkuda võimalikult suurt sisestusvõimsust, nii et selle jõudluse saamine on neid ebamugavusi targa linna kontekstis väärt. Tegelikult on see nii ka teiste omapärade korral, mis sellel andmebaasil on. Kui midagi tundub ebatavaline või imelik, siis ilmselt on dokumentatsioonis selle kohta midagi kirjas või leidub Internetis keegi, kes on sarnase probleemi avastanud ja sellele lahenduse leidnud.

QuestDB'l on olemas API, mida saab kasutada andmete importimiseks, eksportimiseks või SQL-päringute sooritamiseks[12]. Lisaks on olemas veebiliides (vt joonis 3), mille abil saab andmeid

[9]Prometheus. https://prometheus.io/.
[10]Grafana. https://grafana.com/
[11]QuestDB. https://questdb.io/.
[12]QuestDB - REST API https://questdb.com/docs/reference/api/rest/.

pärida ja visualiseerida[13]. Seda on mugav kasutada testandmete sisestamiseks ja kontrollimiseks. Lisaks on hea selle liidese abil pikki päringuid koostada, sest neid saab kohe jooksutada ja vajadusel andmebaasi poolt antud tagasiside põhjal muuta.



Joonis 3. QuestDB graafiline kasutajaliides [16].

[13]QuestDB - Web Console. https://questdb.com/docs/web-console/.

3. Tarkvaralahenduse planeerimine

Selles peatükis kirjeldatakse loodava rakenduse nõudeid, põhjendatakse tehnoloogiavalikuid ja tutvustatakse rakenduse arhitektuuri.

3.1 Tarkvaranõuded

Töö valmis projekti Tartu linna aegrida andmete platvormi lahendus[14] raames, mis tähendab, et enamus nõuetest tuli nende poolt. Hanke raames väljastati dokument koos algsete nõuetega, mis vajasid jooksvalt täpsustamist. Selleks toimusid ülenädalased koosolekud, kus sai arutada, mis oli vahepeal valmis saanud ja samas nõudeid täpsustada. Autorit kaasati nendele koosolekutele, et ta saaks parema arusaama, mida klient soovib ja samas saaks vajadusel kaasa rääkida.

Loodav lahendus võimaldab lisada täpselt selliseid funktsionaalsusi, nagu vaja ja mitte tekitada liigset segadust, mis võib Cumulocity puhul tekkida. Lisaks omab Tartu linn täpset ülevaadet sellest, mis rakenduse sees toimub, sest kood on erinevalt eelnevast lahendusest nende käes. Lisaks saab andmemudeleid muuta vastavalt soovidele ja vajadustele.

3.1.1 Osapooled

Seade on üksus, mis saadab API kaudu sündmuseid ja/või mõõtetulemusi andmebaasi. Tartu puhul on need näiteks sensorid, mis mõõdavad temperatuuri, loendavad jalakäijaid või sõidukeid.

Tavakasutaja on keegi, kellel on õigus teatud seadmete, enamasti enda ettevõtte omade, poolt saadetud informatsiooni pärida ja kasutada selleks soovi korral filtreid. Tal ei ole õigust kasutajaid luua, muuta ega kustutada. Iga kasutaja kohta on täpselt üks API võti.

Administraator on keegi, kellel on õigus näha kõiki kasutajaid, seadmeid ja nende poolt saadetud informatsiooni. Lisaks saab administraator seadmeid ja tavakasutajaid ehk eelmises lõigus kirjeldatuid luua, muuta ja kustutada. Iga administraatori kohta on täpselt üks API võti.

3.1.2 Funktsionaalsed nõuded

Tabelis 1 on välja toodud rakenduse funktsionaalsed nõuded. Nõude prioriteet määrati peamiselt selle järgi, mida soovis klient ja mida tehti lisaarendusena. Kliendi poolt määratu on kõrgema prioriteediga.

[14]Tartu linna aegrida andmete platvormi lahendus. https://www.etis.ee/Portal/Projects/Display/9e86195f-bf54-42c1-b4c6-ded0821609bd.

Nõude number	**Kirjeldus**	**Prioriteet**
F1	Kasutaja peab saama seadmeid süsteemi registreerida.	Kõrge
F2	Kasutaja peab saama seadmete infot pärida.	Kõrge
F3	Seade peab saama mõõtetulemusi andmebaasi saata.	Kõrge
F4	Seade peab saama sündmuseid andmebaasi saata.	Kõrge
F5	Kasutaja peab saama seadme mõõtetulemusi andmebaasist pärida.	Kõrge
F6	Kasutaja peab saama seadme sündmuseid andmebaasist pärida.	Kõrge
F7	Kasutaja peab saama seadme mõõtetulemuste ja sündmuste pärimisel kasutada järgnevaid parameetreid: ajavahemik, seadmetüüp, seadmegrupp, seadme ID.	Kõrge
F8	Kasutaja peab saama seadme andmeid muuta.	Kõrge
F9	Kasutaja peab saama seadet kustutada.	Kõrge
F10	Kasutaja peab saama seadme sündmuste pärimisel kasutada järgnevaid parameetreid: sündmuse tüüp, fragmendi tüüp.	Kõrge
F11	Kasutaja ei tohi saada kasutajaid luua.	Kõrge
F12	Kasutaja ei tohi saada kasutajaid muuta.	Kõrge
F13	Kasutaja ei tohi saada kasutajaid kustutada.	Kõrge
F14	Administraator peaks saama uusi kasutajaid luua.	Keskmine
F15	Administraator peaks saama kasutajaid vaadata.	Keskmine
F16	API peaks võimaldama andmete väljastamist lehekülgede kaupa.	Keskmine
F17	API võiks võimaldada lehekülgede kasutamisel määrata lehekülje suurust.	Madal
F18	API võiks võimaldada päringu tulemust sorteerida kronoloogiliselt kahanevas järjekorras.	Madal
F19	Kasutaja võiks saada seadme mõõtetulemuste pärimisel kasutada järgnevaid parameetreid: mõõtetulemuse tüüp, seeria tüüp.	Madal
F20	Administraator võiks saada kasutajaid kustutada.	Madal

Tabel 1. Funktsionaalsed nõuded.

3.1.3 Mittefunktsionaalsed nõuded

Tabelis 2 on välja toodud rakenduse mittefunktsionaalsed nõuded. Prioriteetide loogika on sama, mis funktsionaalsete nõuete puhul.

Nõude number	**Kirjeldus**	**Prioriteet**
MF1	API peab olema võimeline hakkama saama suurema arvu päringutega kui Cumulocity.	Kõrge
MF2	Ligipääs otspunktidele peab olema kontrollitud API võtme ja/või parooli abil.	Kõrge
MF3	API võtmeid peab olema võimalik ründe korral muuta ja/või deaktiveerida.	Kõrge
MF4	API peab kasutama suhtluseks HTTPS protokolli.	Kõrge
MF5	API võtme nimi peab olema unikaalne.	Kõrge
MF6	Seadme nimi peab olema unikaalne.	Kõrge
MF7	Seadme nimi tohib sisaldada ainult suuri ja väikeseid inglise tähestiku tähti, numbreid ja alakriipsu.	Kõrge
MF8	API ei tohi võimaldada SQL-süsti (ingl *SQL injection*).	Kõrge
MF9	API võiks koguda ja edastada talitlust iseloomustavaid andmeid.	Madal

Tabel 2. Mittefunktsionaalsed nõuded.

3.2 Kasutatud tehnoloogiad

Selles alapeatükis tutvustatakse tarkvaralahenduse loomiseks kasutatud tehnoloogiaid.

3.2.1 Python

Python on kõrgetasemeline programmeerimiskeel, mida on lihtne kirjutada ja lugeda[15]. K. R. Srinath sõnul saab seda kasutada erinevate eesmärkide saavutamiseks, näiteks on see sobilik algajatele oma lihtsuse ja inglise keelele sarnase süntaksi poolest. Samas saab seda tema sõnul kasutada edukalt andmeteaduses [17].

[15] Python Software Foundation. https://www.python.org/.

Keel osutus valituks juhendajaga peetud arutelu tulemusena, peamine argument Pythoni poolt oli, et seda kasutatakse Tartu ülikoolis näiteks aines "Veebiteenuste ja hajussüsteemide arendus" (LTAT.06.018)[16]. Tulevikus on võimalik töö põhjal koostada selle ja teiste ainete jaoks praktikumijuhendeid.

Rakenduse poolt töödeldavate andmete näol on tegu aegridadega (ingl *time series*), seega on võimalik, et tulevikus tekib vajadus neid kasutada andmeteaduses levinud tööriistadega. Üheks selliseks näiteks on Pandas[17], mis on samuti kirjutatud Pythonis, seega vajaks selle tööriista kasutuselevõtt vähem lisatööd võrreldes sellega, kui rakendus oleks kirjutatud mõnes muus programmeerimiskeeles.

Arvestades taustapeatükis välja toodud plusse ja aine Veebiteenuste ja hajussüsteemide arendus LTAT.06.018 õppejõu võimalikku soovi tulevikus selle töö põhjal koostada praktikumiülesandeid, sai raamistikuks valitud FastAPI.

3.2.2 Pydantic

Projektiga seotud inimestega peetud koosolekutel selgus, et arendatava rakenduse ja sellest sõltuvate rakenduste töötamiseks on oluline, et andmed oleks kindla struktuuriga. Näiteks ei tohi tekkida olukorda, kus seadmest tuleva mõõtetulemusega ei ole kaasas mõõtetulemuse tüüpi, sest siis ei ole võimalik seda hiljem tüübi põhjal pärida.

Selleks saab kasutada näiteks Pythoni teeki Pydantic, mis on levinuim viis selles keeles andmete valideerimiseks[18]. See toetab JSON (ingl *JavaScript object notation*) vormingut, mida rakendused kasutavad omavahel suhtlemiseks, oskab sellest koostada Pythoni objekte ja vastupidi, seega on see API loomiseks hea lahendus. Lisaks võimaldab see andmemudeleid defineerida Pythoni klassidele väga sarnaste struktuuridena, mis teeb programmikoodi suure tõenäosusega arusaadavaks kõigile, kes oskavad Pythoni koodi lugeda.

3.2.3 OpenAPI spetsifikatsioon

OpenAPI spetsifikatsioon[19] aitab kirjeldada, millised otspunktid peaksid APIl olema, millise struktuuriga andmeid need otspunktid peavad oskama saata ja vastu võtta. Selle tugevus on

[16]ÕIS II. https://ois2.ut.ee/#/courses/LTAT.06.018/details.
[17]Pandas. https://pandas.pydata.org/.
[18]Pydantic. https://docs.pydantic.dev/latest/.
[19]OpenAPI. https://www.openapis.org/.

tehnoloogiatest sõltumatus[20]. Pole vahet, kas rakendus luuakse Pythoni või näiteks Go abil, OpenAPI spetsifikatsioon on mõlema puhul sama. Lisaks on võimalik sellise dokumendi põhjal genereerida sobivate tööriistade, näiteks OpenAPI Generatori[21] või Swaggeri[22], abil serverikood. Selline koodi genereerimine aitab vähendada tüüpsisu (ingl *boilerplate*) kirjutamist ja võimaldab keskenduda konkreetse rakenduse äriloogikale.

3.2.4 QuestDB

QuestDB on kronoloogiline andmebaas, mille pikem kirjeldus on leitav peatükis 2.5.

Seadmete ja andmete rohkuse tõttu on oluline, et andmete sisestamine toimuks kiiresti ja tõhusalt. QuestDB tiimi poolt korraldatud testis [18] suutis QuestDB oma peamise konkurendi ja teise väga levinud kronoloogilise andmebaasi InfluxDB'ga[23] võrreldes saavutada 3-10 korda parema sisestuskiiruse.

QuestDB on andmete sisestamiseks loonud Pythoni kliendi [19]. See tähendab, et on olemas ametlik dokumentatsioon ja tugi, mis aitavad API rakendust andmebaasiga ühendada.

3.2.5 InfluxDB Line Protocol

InfluxDB Line Protocol (ILP) võimaldab ühe tekstireaga väljendada ühte aegrea punkti ja seda kasutatakse aegridade edastamiseks kronoloogilisse andmebaasi [20]. QuestDB poolt loodud Pythoni klient suudab samuti seda kasutada [19]. Näiteks kirjeldab järgnev ILP rida ühte temperatuuri mõõtmist USA kindlas piirkonnas kindlal ajahetkel [20]: `weather,location=us-midwest temperature=82 1465839830100400200`. ILP kasutamine aitab tõsta rakenduse sisestusvõimet [19].

3.2.6 PostgreSQL

PostgreSQL on relatsiooniline andmebaas, millel on kõrge maine selle töökindluse ja võimekuse tõttu[24]. Sellel on suur kogukond[25], mis tähendab, et esiteks märkab keegi suure tõenäosusega tekkivaid vigu ja teiseks on lihtne leida enda probleemidele lahendusi, sest ilmselt on keegi juba sama asjaga kokku puutunud. Selles töös on see kasutusel kasutajate andmete hoidmiseks.

---

[20] OpenAPI Specification v3.1.1. https://spec.openapis.org/oas/latest.html#introduction.
[21] OpenAPI Generator. https://github.com/OpenAPITools/openapi-generator.
[22] Swagger Editor. https://editor.swagger.io/.
[23] InfluxData https://www.influxdata.com/time-series-database/.
[24] PostgreSQL. https://www.postgresql.org/.
[25] PostgreSQL Community https://www.postgresql.org/community/.

3.2.7 Psycopg

Psycopg[26] on põhimõtteliselt vahekiht Pythoni ja PostgreSQL andmebaasi vahel. See toetab suhteliselt keerukaid operatsioone[27], aga selle töö puhul on oluline, et see võimaldab SELECT käsu abil andmeid pärida ja UPDATE käsu abil andmeid uuendada.

3.2.8 Docker

Dockeri tutvustuses [21] lubatakse, et see võimaldab rakendusi arendada ja käitada mugavalt, kiiresti ja erinevatel platvormidel. Samuti kirjeldatakse seal, et iga rakenduse jaoks luuakse isoleeritud keskkond, mis jookseb samamoodi kõigil platvormidel, mis toetavad Dockerit. Dockeri tutvustuses tuuakse välja, et eesmärk on luua rakendus, mis teeb täpselt seda, mida temast tahetakse ja millel pole kaasas muid ebavajalikke osi (näiteks graafiline kasutajaliides). Ühte Dockeri poolt loodud konkreetset keskkonda nimetavad nad konteineriks (ingl *container*).

QuestDB ja FastAPI on võimalik paigutada konteineritesse ja neil mõlemal on selle jaoks koostatud põhjalik dokumentatsioon [22, 23]. See teeb arenduse mugavamaks ja kiiremaks. Näiteks kui juhendaja või klient soovib rakendust katsetada, saab ta ühe käsuga kogu rakenduse koos andmebaasiga enda masinas tööle panna. Lisaks ei pea mõtlema süsteemide erinevustele, sest Docker jooksutab konteinereid isoleeritud keskkonnas.

3.3 Andmemudelid

Käesolevas alapeatükis kirjeldatakse nii API poolt kasutatavaid JSON-tüüpi andmestruktuure kui ka nende salvestamiseks vajalikke andmebaasitabeleid.

3.3.1 QuestDB andmestruktuurid

QuestDB puhul on kõik tabelid ühes andmebaasis, erinevalt näiteks PostgreSQL'ist, mille puhul ühe isendi raames on võimalik luua erinevaid nimeruume [24]. Igal tabelil peab olema määratud ajatempel. Seda kasutatakse näiteks partitsioneerimisel. Kasutaja saab määrata, millise ajavahemiku alusel partitsioneerimine toimub, näiteks on võimalik iga päeva või tunni kohta teha eraldi partitsioon, mida hoitakse eraldi failides. See on kasulik näiteks andmete pärimise kiirendamiseks, sest korraga ei ole vaja mällu lugeda kõiki andmeid.

---

[26] Psycopg. https://www.psycopg.org/.

[27] Comparing psycopg2 vs psycopg in Python. https://www.geeksforgeeks.org/comparing-psycopg2-vs-psycopg-in-python/.

On võimalik määrata erilisi veergusid, mille tüüp on *symbol*[28]. Neid kasutatakse partitsioneerimisel nagu ajatemplite vahemikkegi. Seega on võimalik näiteks sündmuseid partitsioneerida seadme ID alusel. Lisaks hoitakse sellist tüüpi veeru puhul andmete tabelis vaid täisarvulist viidet sõnele, mida ennast hoitakse eraldi teises tabelis. See aitab optimeerida korduvate väärtuste salvestamist. Teine veergude hea omadus on see, et iga tabeli veerg on omakorda iga partitsiooni puhul eraldi failis või kahes failis [24]. See tähendab, et andmete pärimisel peab QuestDB avama ainult need failid, mille veerge päriti.

3.3.2 Mõõtetulemus

Cumulocity mõõtetulemuse struktuuri kirjeldati peatükis 2.2. Cumulocity pakub lisaks SmartREST protokolli kasutamise võimalust [25]. Nii on võimalik registreerida seadme jaoks mall, milliseid andmeid ta saadab ja siis saata lihtsalt õiges järjekorras komaga eraldatud väärtused. Need lähenemised on väga efektiivsed, aga inimese jaoks raskesti mõistetavad.

Tihti on tüübi ja fragmendi määramine üleliigne, mis viib selleni, et ühte neist pannakse lihtsalt midagi, et see tingimus täidetud oleks. Arendaja jaoks on mure lahendatud, aga siis tekib küsimus, et mida nende andmetega peale hakata, kui tegelikult ei ole aru saada, mida mõõdetud on. Näiteks oli eelmises süsteemis fragment nimega *PedestrianOut*. Mida see tähendab? Mitu õues olevat jalakäijat mõõtis seade mingil ajahetkel? Mitu jalakäijat on linnast väljas? Hea küll, anname ka tüübi: *dal_series_target_measurement*. Tõenäoliselt ei läinud pilt oluliselt selgemaks. Seega on näiteks analüütikul vaja pidevalt dokumentatsioonis näpuga järge ajada, et leida, mida need väärtused tähendavad, eeldusel, et see on üldse dokumenteeritud. Inimesel, kes nende andmetega igapäevaselt ei tegele, ei ole erilist lootust üldse midagi aru saada.

Tausta peatükist ja eelmisest lõigust tuleb seega välja kaks probleemi. Esiteks on Cumulocity mõõtetulemuste JSON-objektide võtmetes ja väärtustes raske orienteeruda ja teiseks on infovälju vähemalt selle projekti raames liiga palju. Koodinäites 6 olev uus struktuur lahendab võrreldes koodinäites 5 oleva Cumulocity struktuuriga mõlema probleemi või vähemalt leevendab neid:

[28] QuestDB - Symbol. https://questdb.com/docs/concept/symbol/.

```
{
    "source": {
        "id": "681200"
    },
    "time": "2020-03-19T12:03:27.845Z",
    "type": "temperatureMeasurement",
    "c8y_SteamFragment": {
        "Temperature": {
            "value": 42.7,
            "unit": "C"
        },
        "Humidity": {
            "value": 13.37,
            "unit": "%RH"
        }
    }
}
```

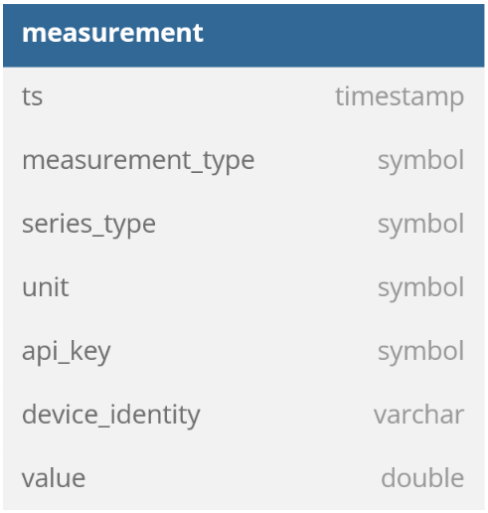
Koodinäide 5. Mõõtetulemuse andmed Cumulocity APIs.

```
{
  "time": "2020-03-19T12:03:27.845Z",
  "device_identity": "681200",
  "measurement_type":
  ↪  "temperatureMeasurement",
  "series": [
    {
      "series_type": "Temperature",
      "unit": "C",
      "value": 42.7
    },
    {
      "series_type": "Humidity",
      "unit": "%RH",
      "value": 13.37
    },
  ]
}
```

Koodinäide 6. Mõõtetulemuse andmed loodud rakenduses.

Esiteks on siin kolmetasandilise JSON'i asemel kaks tasandit ja eraldi võti *series_type*, mis teevad objekti lihtsamini hoomatavamaks. Teiseks on tüübi, fragmendi ja mõõdetava väärtuse asemel jäänud tüüp ja fragmendiga siin samamoodi käituv *series*, seega ei pea mingeid väärtuseid välja mõtlema lihtsalt selleks, et platvormi poolt määratud tingimused oleks täidetud. Lisaks on mitme väärtuse korraga saatmiseks kasutusel järjend, mis annab võimaluse peale vaadates kohe aru saada, et väärtuseid võib olla mitu. Cumulocity puhul ei oleks see nii intuitiivne, kui näites juhtuks olema üks väärtus, sest seal ei kasutata järjendit.

Sellist struktuuri on mugav andmebaasitabeliks teisendada (vt joonis 4). Juhul, kui *series* järjend sisaldab mitut elementi, on võimalik igaüks neist salvestada eraldi reale. Tulenevalt sellest, et mõõtetulemusi saadetakse kindla ajavahemiku tagant, on neid hea andmebaasis partitsioonideks jaotada nii, et need on sarnase suurusega. Seetõttu valiti *symbol*-tüüpi veergudeks kõik need väärtused, mis ajas ei muutu. Edaspidi saab partitsioonide suurust reguleerida partitsiooni ajavahemikku muutes. Tabelisse lisati ka API võti (veerg *api_key*), sest andmeid peab saama siduda nende saatjaga.



Joonis 4. Mõõtetulemuste tabeli struktuur andmebaasis.

3.3.3 Sündmus

Cumulocity poolt pakutava sündmuse struktuuri on kirjeldatud peatükis 2.2. Mõõtetulemusega võrreldes on siin sarnane see, et võti on ise ka väärtus. Samas on siin vaja määrata ainult sündmuse tüüp, mitte veel mingi alamtüüp ja siis fragment. See ei tähenda, et kasutaja elu oleks kuidagi lihtsam. Fragment, siin *avc_Vehicle*, on vabatahtlik ja võib olla ükskõik milline JSON-objekt ehk näiteks sõne, arv või teine JSON-objekt, nagu ka siin näites. Teisest küljest on see vajalik, sest iga sündmuse kohta on vajalik saata erineva struktuuriga andmeid.

Seega võivad erinevat tüüpi sündmused olla üksteisest väga erinevad ja seda olukorda on väga raske muuta. Arenduse käigus töötati välja koodinäites 7 välja toodud andmestruktuur.

```
{
  "device_identity": "Riia_77_jalgratta_loendur",
  "time": "2020-06-24T21:00:00.000Z",
  "fragments": [
    {
      "fragment_type": "car-detected",
      "vehicle_type": "Car",
      "occupancy": "2",
      "gap": 3829,
      "length": 2,
      "lane_id": 1,
      "speed": 73
    }
  ],
  "event_type": "avc_VehicleEvent",
  "text": "Vehicle detected"
}
```

Koodinäide 7. Sündmuse andmed loodud rakenduses.

See näeb väga Cumulocity poolt pakutavale struktuurile väga sarnane välja, kuid lahendab samu probleeme, mis esinesid mõõtetulemuse puhul, kui välja arvata liigsed tüübid, sest siin neid ei olnud. Andmetüüpe ei kasutata enam võtmetena. See standardiseerib struktuuri ja kirjeldab, millist tüüpi väärtus peaks selle võtme juurde käima. On lisatud eraldi järjend, mis annab märku, et sinna peaks käima fragmendid. Lisaks sai JSON kolmetasandilisest kahetasandiliseks, mis parandab loetavust.

Tulenevalt sellest, et sündmustel saab põhimõtteliselt olla lõputu arv erinevaid struktuure, on selle andmebaasi salvestamine raskem, kui mõõtetulemuste puhul. Iseenesest oleks olnud võimalik luua liides, mille kaudu saaks luua kindla sündmuse jaoks õige struktuuriga sündmuse tabeli ja siis sinna andmeid saata. Selle projekti raames otsustati lasta andmebaasil tabelid ise genereerida (vt joonis 5). Selle tingis asjaolu, et automaatne genereerimine töötab kirjutamise hetkel teadaolevalt kõigil juhtudel peale selle, kui mingi väärtus saadetakse alguses täisarvuna, kuigi see võib vahel olla ka ujukomaarv. Näiteks kui sõiduki kiirus on esimesel korral 70, siis genereeritakse see veerg täisarvuna, aga edaspidi saadetakse näiteks väärtus 70,5, siis tagastatakse veateade ja sündmust andmebaasi ei lisata. Nii juhtuks ka joonisel 5 oleva tabeli puhul, sest *long* on täisarvutüüpi.

Joonis 5. Andmebaasi poolt genereeritud sündmuse tabel.

Selliste probleemide vältimiseks on mõistlik lasta kasutajatel anda uue sündmusetüübi jaoks tabeli loomisel API kaudu selle sündmuse struktuur, et andmete tüübid rangemalt ette paika panna.

Sündmuse puhul on osa partitsioneerimise tööst juba ära tehtud. QuestDB hoiab kõiki tabeleid eraldi kataloogides [26] ja iga sündmus on eraldi tabelis. Sarnaselt mõõtetulemusele on *symbol*-tüüpi veerud need, mis ajas ei muutu ja edaspidi saab partitsiooni suurust muuta ajavahemiku abil.

3.3.4 Seade

Seadmetega seotud andmeid peab olema võimalik tema poolt saadetud mõõtetulemustest ja sündmustest sõltumatult muuta ja pärida. Seega on selleks vaja eraldi JSON-struktuuri (vt koodinäide 8).

```
{
  "device_identity": "Riia_77_jalgratta_loendur",
  "type": "mobility-counter",
  "group": [
    "electricity"
  ],
  "description": "Asutuse jalgratta loendur Riia tanaval",
  "location_type": "address",
  "lat": 66.82124124,
  "long": 344.8212412,
  "address": {
    "house": 77,
    "street": "Riia tn",
    "city": "Tartu",
    "apartment": 2
  }
}
```

Koodinäide 8. Seadme andmed loodud rakenduses.

Seadme puhul on võimalik muuta kõike peale tema unikaalse tunnuse, sest see on tema poolt saadetud väärtustega seotud. Seadmeid hoitakse lihtsuse ja jõudluse huvides kronoloogilises andmebaasis, seega on kohustuslik kasutada tulpa, mis hoiab ajalist väärtust. See võimaldab näiteks andmete visualiseerimisel automaatselt võtta (andmebaasi JOIN operatsiooni kaudu) ka seadme nime või asukoha. Seega hoitakse alles seadme süsteemi lisamise aega (vt joonis 6), mis ei olnud nõue, aga võib tulevikus kasulik olla. Lisaks salvestatakse seadmete gruppide järjend ja aadress lihtsalt sõnena, et vältida andmebaasis liigselt keerukate tabelite loomist. Aadressi saaks põhimõtteliselt jaotada eraldi veergudeks, aga juba projekti ajal on tekkinud kahtlus, kas nii keerukat aadressi on päriselt vaja, seega edaspidi on võimalik ilma tabelit muutmata panna aadressiks näiteks ”Kesklinn”, kui API seda toetab.

Joonis 6. Seadmete tabeli struktuur andmebaasis.

3.3.5 Kasutaja

Kasutaja struktuur on väga lihtne (vt koodinäide 9).

```
{
  "key_name": "voti1",
  "first_name": "John",
  "last_name": "James",
  "email": "john.james@gmail.com",
  "phone": "12345",
  "key_status": "active"
}
```

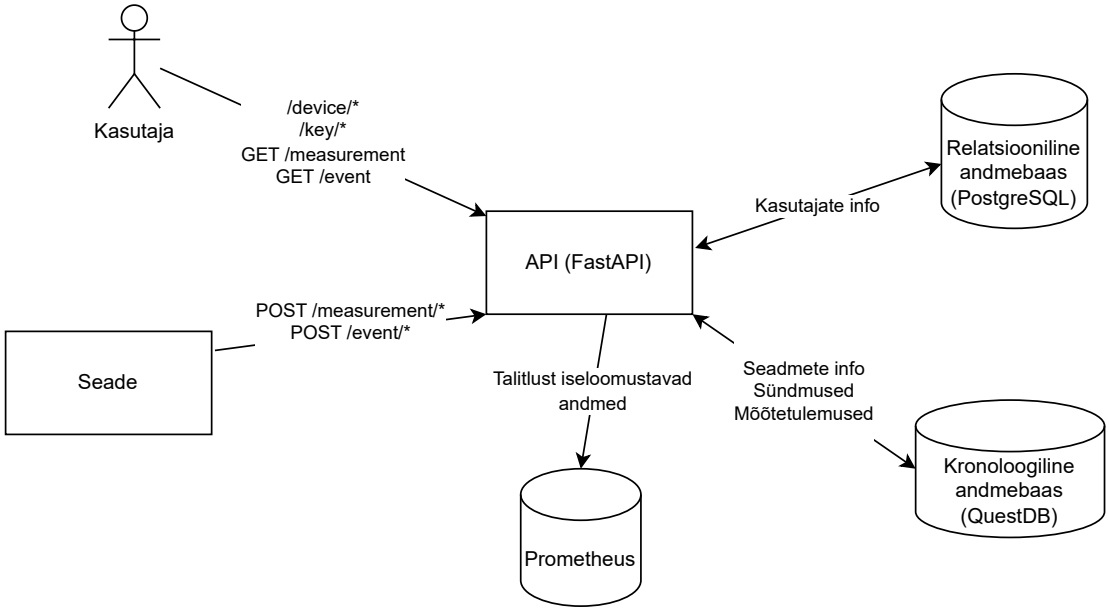
Koodinäide 9. Kasutaja andmed loodud rakenduses.

Kasutajaid hoitakse relatsioonilises andmebaasis (selle projekti puhul PostgreSQL), seega ei ole vaja kunstlikult tekitada aegrida. Iga kasutaja kohta on ka API võti, mis viitab üheselt temale.

Võti luuakse automaatselt kasutaja loomisel. Täpsemalt võikski öelda, et kasutaja on API võti, millel on kaasas metaandmed.

4. Rakenduse arhitektuur ja teostus

Selles peatükis kirjeldatakse, millised on rakenduse komponendid ja kuidas need omavahel koos töötavad (vt joonis 7).



Joonis 7. Rakenduse arhitektuuri visuaalne kirjeldus.

4.1 Andmebaasid

4.1.1 QuestDB

Kronoloogilises andmebaasis on vaja hoida mõõtetulemusi ja sündmuseid. Sellena on kasutusel QuestDB ja andmemudelid said välja töötatud just aegridasid silmas pidades. Küll aga lisas keerukust vajadus seadmeid enda andmetega siduda. Näiteks peab saama pärida mingit tüüpi seadmete poolt saadetud mõõtetulemusi. Iseenesest ei ole mitte mingit põhjust hoida seadmeid kronoloogilises andmebaasis, sest nende kohta ei hoita meeles ühtegi väärtust, mis oleks aegrida. See aga tähendaks, et kui päringus kasutada seadmetüüpi, peaks iga mõõtetulemuse või sündmuse kohta tegema päringu teise andmebaasi, kus kontrollitaks, kas nende andmete saatjal on vajalik seadmetüüp. Seega võeti jõudluse huvides vastu otsus, et kuigi seadmed pole loomult kronoloogiliselt järjestatud, hoitakse neid liigsete päringute vältimiseks sündmuste ja mõõtetulemustega samas (kronoloogilises) andmebaasis.

4.1.2 PostgreSQL

Lisaks kronoloogilisele andmebaasile on kasutusel relatsiooniline andmebaas PostgreSQL, kus hoitakse kasutajaid ja API võtmeid. Otsus kahe andmebaasi kasuks langetati peamiselt turvakaalutlustel, sest seal salvestatakse kasutajate isikuandmete ja ligipääsuvõtmete näol tundlikku informatsiooni. Nendele andmetele pole ühelgi API kasutajal põhjust ligi pääseda.

Teine põhjus on see, et QuestDB vabavaralise versiooni puhul (mis on selles töös kastusel) ei saa ametliku dokumentatsiooni kohaselt [27] teha erinevate õigustega kasutajaid. See on oluline näiteks siis, kui kasutada QuestDB enda veebiliidest[29] või nende enda poolt arendatud REST API't[30]. Kui näiteks anda analüütikule ligipääs API'le, saaks ta lisaks seadmete poolt saadetud andmetele ligi ka kasutajatele ja nende andmetele.

Ühest küljest tähendab kahe andmebaasi kasutamine enamjaolt kahe erineva liidese kasutamist, et nende andmebaasidega suhelda. See lisab rakendusele sõltuvusi, mis omakorda suurendavad keerukust ja riski, et midagi "läheb katki". Sellise ühe tabeliga andmebaasi jaoks pole tegelikult oluline, kas on kasutusel PostgreSQL või näiteks MySQL[31], mis on samuti väga laialt levinud relatsiooniline andmebaas. Samas on just liideste poolest PostgreSQL'l üks suur eelis. Nagu eelnevalt mainitud, kasutab QuestDB päringute jaoks keelt, mis on ehitatud PostgreSQLi peale, seega on võimalik sealt andmete pärimiseks kasutada samu liideseid, millega saab päringuid teha PostgreSQL andmebaasi. Ühe sellisena pakutakse QuestDB dokumentatsioonis välja psycopg [14], mis on ka selles töös kasutusel.

4.2 OpenAPI spetsifikatsioon

API kirjeldamiseks loodi nõuete põhjal OpenAPI spetsifikatsioon (vt Lisa 2)[32]. See võimaldab osapooltele arusaadavalt kirja panna, mida API võimaldab, milliseid parameetreid on võimalik kasutada ja tuua näiteid võimalike parameetrite ja päringute kohta. See on kasulik dokument terve arendusprotsessi käigus. Alguses aitab see kliendi ja arendusmeeskonna vahel nõuete osas möödarääkimisi vältida, sest tegu on *de facto* standardiga API'de kirjeldamises ja selle mõistmine ei eelda teostusest (ingl *implementation*) arusaamist. Arendusprotsessi käigus on see

[29] QuestDB - Web Console. https://questdb.com/docs/web-console/.
[30] QuestDB - REST API https://questdb.com/docs/reference/api/rest/.
[31] MySQL. https://www.mysql.com/.
[32] OpenAPI spetsifikatsiooni täisversioon. https://reaalajaandmed.tartu.ee/docs.

kasulik abivahend jälgimaks, et rakendus vastab nõuetele. Kui rakendus on tehtud selle järgi, siis on see ka väga hea dokumentatsioon.

FastAPIt kasutades on sellel veel kaks eelist. Esiteks genereerib FastAPI automaatselt OpenAPI spetsifikatsiooni ehk on võimalik võrrelda varem kirjutatud spetsifikatsiooni loodud rakenduse omaga ja vaadata, kas need kattuvad. Teine, selle töö kontekstis veelgi kasulikum omadus on see, et Swaggeri[33] abil on võimalik genereerida OpenAPI spetsifikatsiooni põhjal FastAPI serverikood. See teeb arenduse kiiremaks ja mugavamaks, sest ei ole vaja kirjutada sarnaseid funktsioonide päiseid palju kordi, vaid saab keskenduda nende sisule. Lisaks saab spetsifikatsiooni muutumisel koodi uuesti genereerida, mille käigus uuendatakse ka päiseid.

Vaatamata sellele, et kood genereeritakse, tuleb ettevaatlik olla, sest mingi osa genereeritud koodist tuleb käsitsi üle käia. Seda põhjusel, et genereerimiseks kasutatav programm OpenAPI Generator[34] (versioon 7.10.0) ei saanud ametliku dokumentatsiooni põhjal hakkama kõigi spetsifikatsioonis kirjeldatud omadustega. Näiteks on sündmuste fragmentide puhul kasutusel JSONi *additionalProperties*[35] (vt 3.3.3, täpsemalt *fragments* järjendi sisu), mis tähendab, et ei ole võimalik ette määrata võtit. OpenAPI Generator oskab küll selle välja Pythoni koodi lisada, kuid ei oska seda välja õigesti kasutada.

4.3 API

Kogu kliendi ja andmebaaside vaheline suhtlus toimub läbi FastAPI raamistikule loodud rakenduse. Seadmed saavad sellele andmeid saata. Selleks, et nende andmetega midagi teha, on vaja rakendusel suhelda andmebaasidega. Kasutajad saavad seadmeid ja administraatoriõiguste korral teisi kasutajaid hallata. Lisaks saavad nad seadmete poolt saadetud andmeid pärida. Ka selleks peab rakendus suhtlema andmebaasidega. Ülevaate API otspunktidest ja meetoditest on saadaval lisas 2. Järgnevates lõikudes antaksegi ülevaade rakenduse toimimise tehnilisest poolest.

---

[33] Swagger Editor. https://editor.swagger.io/.

[34] OpenAPI Generator. https://github.com/OpenAPITools/openapi-generator.

[35] JSON - Additional Properties. https://json-schema.org/understanding-json-schema/reference/object#additionalproperties.

4.3.1 Andmete valideerimine

Oletame, et seade on saatnud mõõtetulemused või sündmused (korraga saab saata ühte tüüpi andmeid; neid võib olla üks või mitu, loogika jääb samaks). Need võetakse API poolt vastu. Järgmisena kontrollitakse, kas päringuga on kaasas API võti ja kas andmed on õige struktuuriga. API võtme puhul tehakse psycopg abil päring PostgreSQL andmebaasi, kus hoitakse kasutajate andmeid. Selline päring tehakse tegelikult iga kord igasse API otspuntki päringu tegemisel, sest kõik otspunktid on turvatud. Kui võtmele vastavat kasutajat ei leita, lükatakse kogu seadme poolt tehtud päring tagasi ja saadetud andmeid ei sisestata. Sama kehtib ka juhul, kui saadud kasutaja on deaktiveeritud. Saadetud andmete enda puhul kontrollib Pydantic automaatselt, kas need vastavad peatükis 3.3.2 või 3.3.3 defineeritud struktuurile. Vea tekkimisel tagastatakse, mis oli struktuuri puhul valesti (vt koodinäide 10).

```
{
  "detail": [
    {
      "type": "missing",
      "loc": [
        "body",
        "measurements",
        0,
        "series",
        0,
        "series_type"
      ],
      "msg": "Field required",
      "input": {
        "series_tye": "Bicycle-towards-center",
        "unit": "count",
        "value": 34.2
      }
    }
  ]
}
```

Koodinäide 10. API vastus vigase struktuuriga mõõtetulemuse saatmisel.

Rakendusel on tugi eelneva lahenduse ehk Cumulocity struktuuriga andmete vastuvõtmiseks. Kui andmeid saadetakse selle liidese kaudu, ei ole kasutusel Pydanticu poolt pakutav valideerimine. Selle asemel loodi just sellise struktuuriga objektide valideerimiseks mõeldud lahendus. Pythoni sõnastik ja JSON-objekt on struktuurilt väga sarnased, seega on struktuuri läbimine mugav, sest

saab kasutada sõnastikust tuttavaid operatsioone ja ei pea juurde tooma täiendavaid sõltuvusi. Kui vajalikud tingimused on täidetud, antakse andmed edasi andmete andmebaasi sisestamisega tegelevale komponendile. Siiani toimub sündmuse puhul kõik samamoodi.

4.3.2 Andmete sisestamine kronoloogilisse andmebaasi

Pärast sisestatavate andmete struktuuri valideerimist on vaja iga saadetava väärtuse korral kontrollida, kas sellega kaasasoleva seadme ID vastab päriselt süsteemi registreeritud seadmele. Kui ei vasta, siis ühtegi väärtust selle päringu raames andmebaasi ei lisata ja vigane ID lisatakse selleks ettenähtud järjendisse. Vigase seadme ID leidmisel kontrollitakse üle kõik sellele järgnevad väärtused ja kui ka neis leidub vigane ID, siis lisatakse see samuti järjendisse. Need tagastatakse koos veateatega (vt koodinäide 11).

```
{
  "detail": {
    "Device IDs not found": [
      "123abc"
    ]
  }
}
```

Koodinäide 11. API vastus vigase seadme ID'ga mõõtetulemuse saatmisel.

Kui kõik seadmed on süsteemi registreeritud, jaotatakse need vastavalt andmebaasi tabeli struktuurile ridadeks ja read saadetakse andmebaasi. Selleks on kasutusel QuestDB ametlik andmete sisestamiseks loodud Pythoni teek [19]. See saadab read ILP-vormis HTTP ühenduse kaudu QuestDB andmebaasi. Mõõtetulemuste puhul salvestatakse kõik andmed ühte tabelisse. Sündmuste puhul on iga sündmusetüüp omaette tabelis ja nende struktuurid on erinevad, seega seal on vaja natuke dünaamilisemat lähenemist (vt koodinäide 12).

```
columns = dict()
columns["text"] = event.text
for fragment_value in fragment.additional_properties:
    columns[fragment_value] =
    ↪  fragment.additional_properties[fragment_value]
sender.row(
    event_type,
    symbols={
        "API_KEY": api_key,
        "device": device_identity,
        "fragment_type": fragment.fragment_type,
    },
    columns=columns,
    at=datetime.fromisoformat(time.rstrip("Z"))
    .replace(tzinfo=timezone.utc)
)
```

Koodinäide 12. Sündmuse salvestamine QuestDB Pythoni teegi kaudu.

4.3.3 Andmete pärimine

Andmete pärimiseks QuestDB eraldi Pythoni teeki ei paku, kuid nagu peatükis 4.1 ilmnes, saab selleks kasutada teeki psycopg. See võimaldab koostada SQL-päringu Pythoni sees, seda jooksutada ja siis saadud tulemusi kasutada. Mõõtetulemuste puhul koosneb enamus loogikast sellest, et vastavalt kasutaja poolt antud parameetritele tuleb need SQL SELECT päringu WHERE osa külge panna (vt koodinäide 13).

```
if device_identity:
    query += " AND measurement.device_identity = %s"
    params.append(device_identity)
```

Koodinäide 13. Päringu koostamine mõõtetulemuste andmebaasist pärimiseks.

Sündmuste puhul on põhimõte sama, aga tabelite struktuuri tõttu on vaja tabelinimi päringusse dünaamiliselt lisada (vt koodinäide 14).

```
if device_identity:
    query += " AND {event_table}.device_identity = %s"
    params.append(device_identity)
```

Koodinäide 14. Päringu koostamine sündmuste andmebaasist pärimiseks.

4.3.4 Käitamine ja seire

Rakendust saab jooksutada Dockeri konteinerites, mis lihtsustab mitme isendi loomist, et API saaks korraga rohkem andmeid vastu võtta. Lisaks saadab rakendus Prometheuse ametliku teegi[36] kaudu talitlust iseloomustavaid andmeid (vt koodinäide 15).

```
metrics_app = make_asgi_app()
app.mount("/metrics", metrics_app)

# https://fastapi.tiangolo.com/tutorial/middleware
# /#before-and-after-the-response
@app.middleware("http")
async def metrics_middleware(request: Request, call_next):
    start_time = time.time()
    response: Response = await call_next(request)

    duration = time.time() - start_time
    endpoint = request.url.path
    method = request.method
    status_code = response.status_code

    prometheus.REQUEST_DURATION.labels(endpoint, method,
    ↪  status_code).observe(duration)

    return response
```

Koodinäide 15. Talitlust iseloomustavate andmete salvestamine.

[36] Prometheus Python Client. https://prometheus.github.io/client_python/.

Täpsemalt salvestatakse iga päringu puhul selle kestvus, otspunkti nimetus, HTTP-meetod ja saadud olekukood. Need tehakse Prometheuse jaoks kättesaadavaks */metrics* otspunkti kaudu. Need andmed osutusid valituks, sest nende põhjal on võimalik kindlaks teha, kas rakendus töötab korrektselt ja kui ei tööta, siis saab olekukoodide ja otspunktide põhjal teada, millises rakenduse osas on probleem. Päringute kestvuse põhjal saab jälgida näiteks seda, kas rakendusel ja andmebaasil on piisavalt riistvaralisi ressursse, et päringuid teenindada. Kui päringute kestvused lähevad suuremaks, saab uurida, kas probleem on ressursis või käitub rakendus ise saadud andmetega ebakorrektselt. Ühe saadetava väärtuse struktuur on välja toodud koodinäites 16.

```
REQUEST_DURATION = Histogram(
    name='request_duration_seconds',
    documentation='Duration of requests in seconds',
    labelnames=['endpoint', 'method', 'status_code']
)
```

Koodinäide 16. Päringuid iseloomustavate andmete struktuur.

Koodinäites 17 on välja toodud osa rakenduse poolt kogutavatest talitlusandmetest:

```
request_duration_seconds_count{endpoint="/measurement",method="POST",
status_code="201"} 4234.0
request_duration_seconds_sum{endpoint="/measurement",method="POST",
status_code="201"} 22094.012771606445
request_duration_seconds_count{endpoint="/device",method="POST",
status_code="201"} 2000.0
request_duration_seconds_sum{endpoint="/device",method="POST",
status_code="201"} 3558.955129146576
request_duration_seconds_count{endpoint="/docs.json",method="GET",
status_code="404"} 1.0
request_duration_seconds_sum{endpoint="/docs.json",method="GET",
status_code="404"} 0.0003132820129394531
```

Koodinäide 17. Päringuid iseloomustavad talitlusandmed.

Selliste andmete põhjal on võimalik arvutada näiteks päringute keskmist aega, loendada nurjunud ja õnnestunud päringuid. Kuna tegelikult on kogutavate andmete puhul tegemist

histogrammidega, on võimalusi veel, aga näite lihtsuse huvides ei ole see hetkel oluline. Prometheusel on histogrammide[37] ja teiste andmetüüpide[38] kohta põhjalik dokumentatsioon.

[37] Prometheus - Histogram. https://prometheus.io/docs/concepts/metric_types/#histogram.
[38] Prometheus - Metric Types. https://prometheus.io/docs/concepts/metric_types/.

5. Tulemused

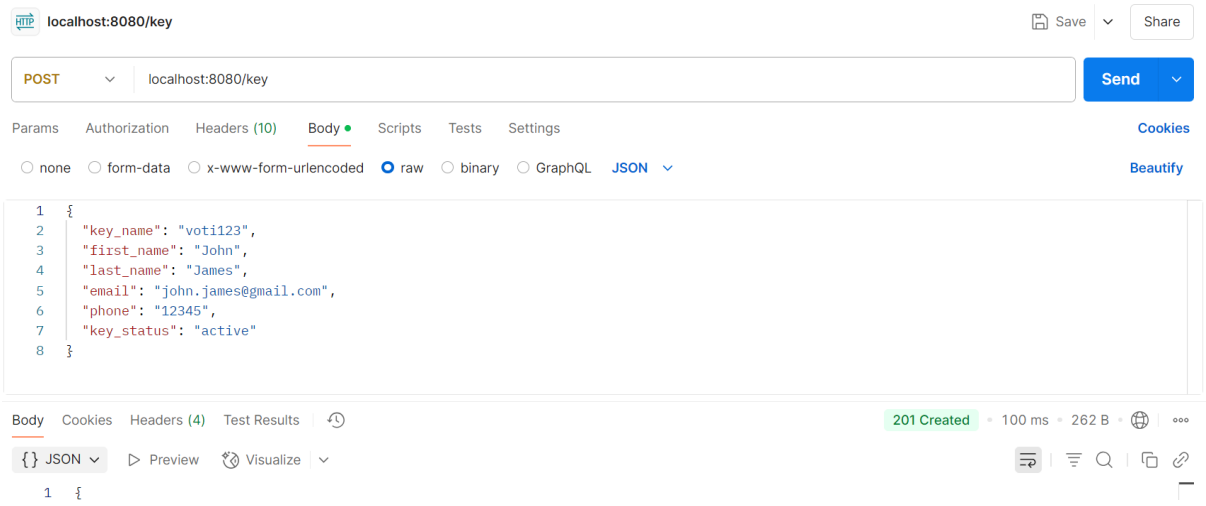
Selles osas võrreldakse rakendust tema eelkäijaga, tuuakse välja koormustesti tulemused ja analüüsitakse nõuete täitmist. Täitmata nõuete puhul arutletakse täitmata jäämise põhjuste ja puuduste parandamise võimaluste üle.

5.1 Valminud rakendus

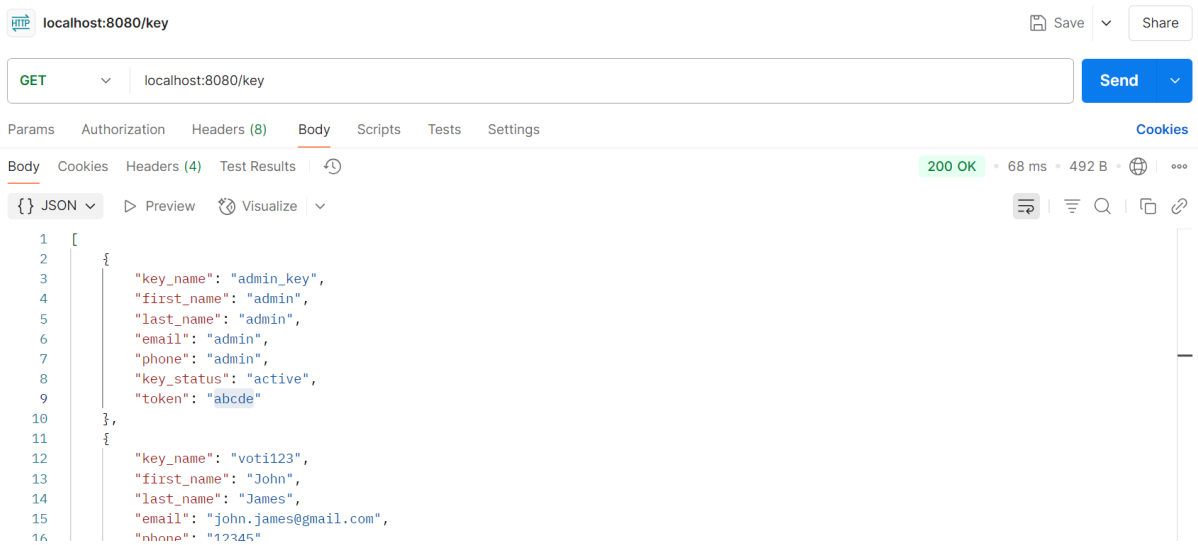
Selles osas antakse üldine ülevaade sellest, millised on valminud rakenduse omadused ja milliseid funktsionaalsusi see sisaldab.

Kasutajate ja seadmete haldus

Loodud rakendus võimaldab ainult administraatoril (F11, F12) luua uusi kasutajaid (F14, vt joonis 8) ja kõiki kasutajaid korraga pärida (F15, vt joonis 9). API võti genereeritakse rakenduse poolt ja kui juhtub tekkivat duplikaat, genereeritakse uus võti (MF5). Süsteem ei võimalda kasutajaid kustutada (F13).

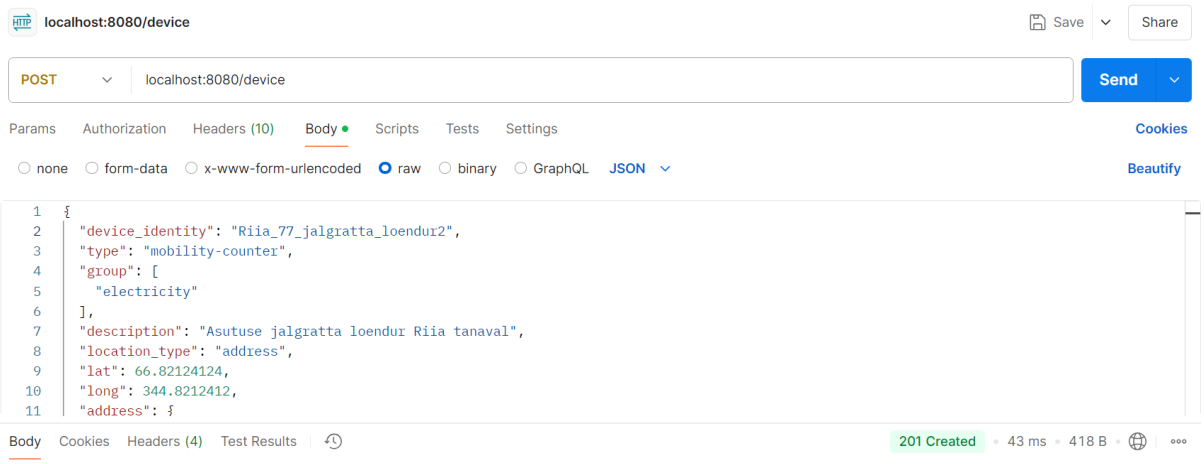


Joonis 8. Kasutaja loomine.

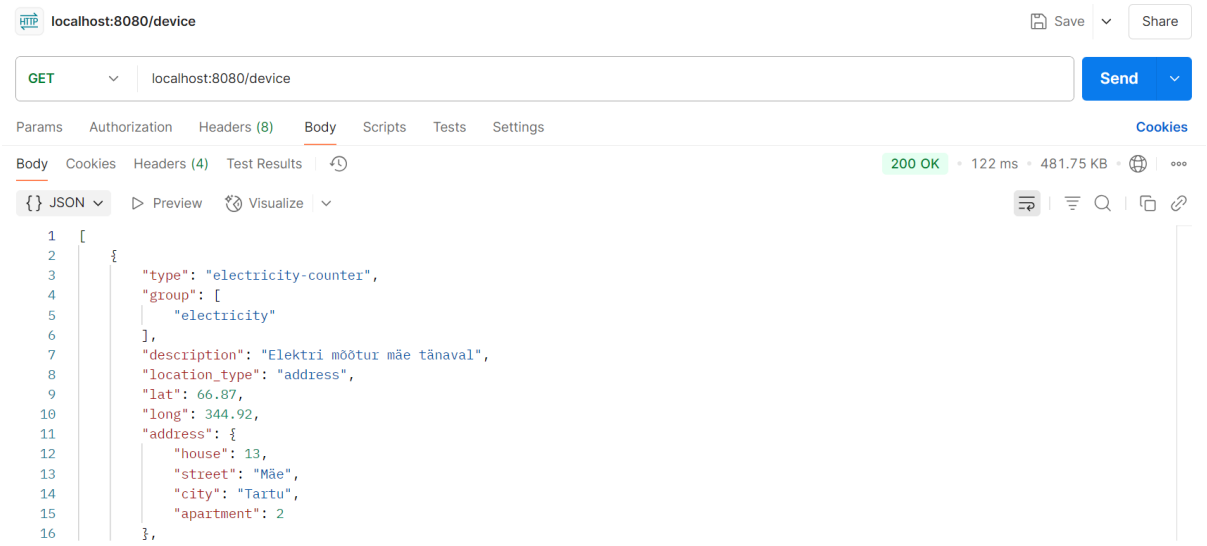


Joonis 9. Kasutajate pärimine.

Seadmeid on võimalik luua (F1, vt joonis 10) ning kõiki seadmeid korraga pärida (F2, vt joonis 11).

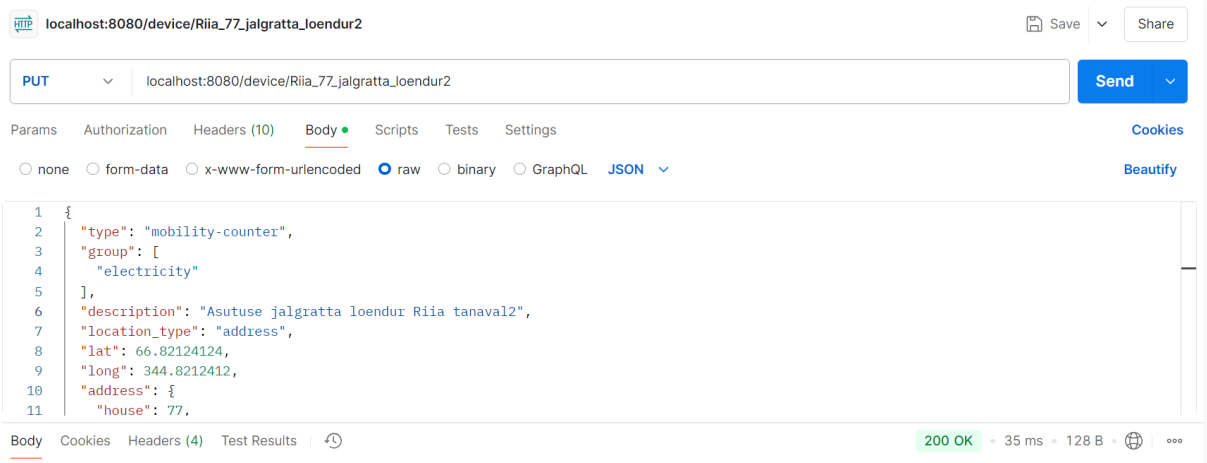


Joonis 10. Seadme loomine.



Joonis 11. Kõigi seadmete pärimine.

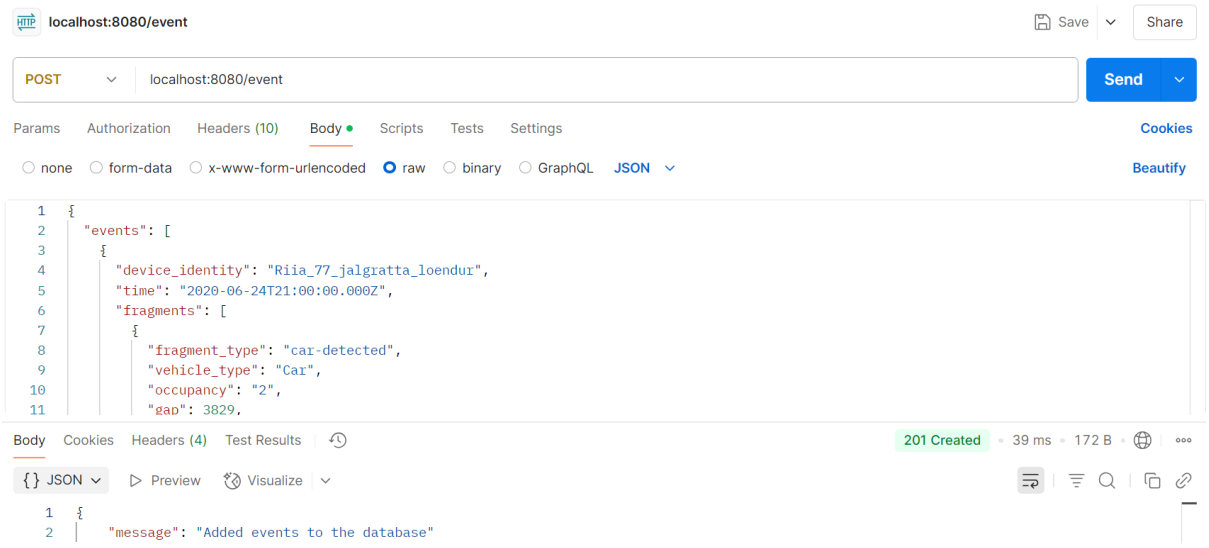
Lisaks on võimalik muuta enda poolt loodud seadmete infot (F8, vt joonis 12). Seadme loomisel kontrollitakse, et selle nimi oleks unikaalne (MF6) ja et see sisaldaks vaid suuri ja väikeseid inglise tähestiku tähti, numbreid ja alakriipsu (MF7).



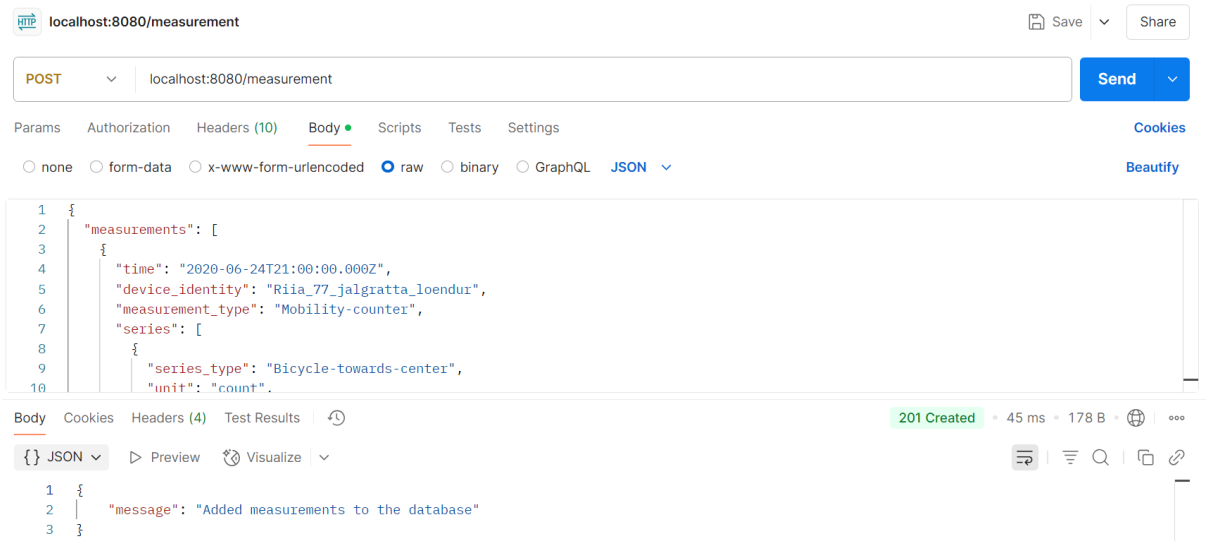
Joonis 12. Seadme info muutmine.

Sündmuste ja mõõtetulemuste sisestamine

Rakendus võimaldab seadmetel sündmusi (F4, vt joonis 13) ja mõõtetulemusi (F3, vt joonis 14) sisestada. Andmete saatmisel kontrollitakse API võtme olemasolu, korrektsust ja kehtivust (MF2).



Joonis 13. Sündmuse sisestamine.



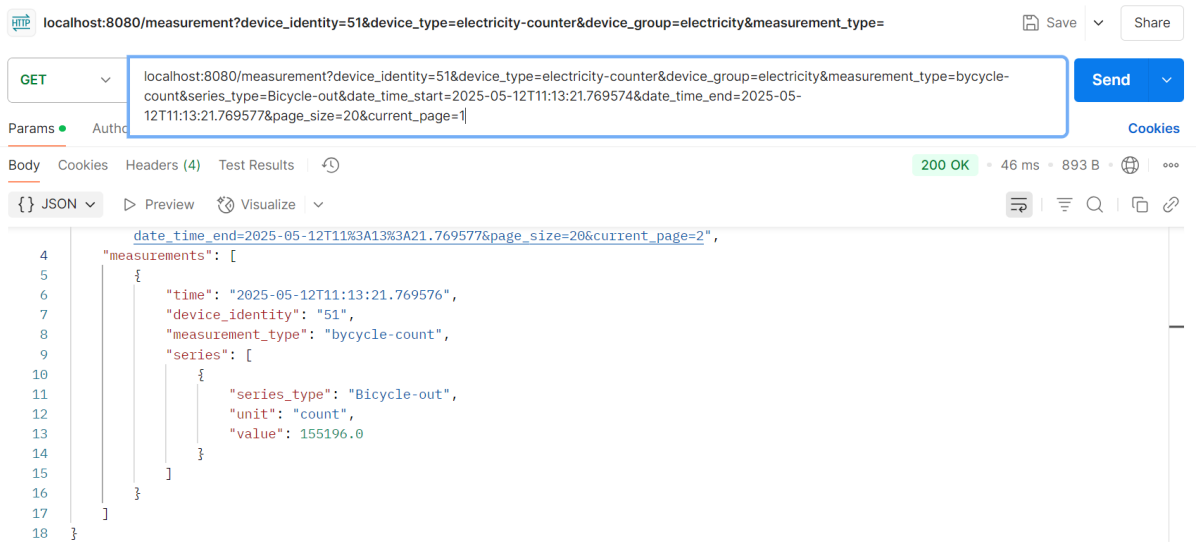
Joonis 14. Mõõtetulemuse sisestamine.

Mõõtetulemuste pärimine

Mõõtetulemusi saab API kaudu pärida (F5). Rakendus ei võimalda parameetreid kasutada SQL-süsti läbiviimiseks (MF8). Mõõtetulemuste pärimisel on võimalik kasutada järgnevaid parameetreid (F7, F16, F17, F19) (vt joonis 15):

- Seadme ID
- Seadmetüüp
- Seadmegrupp

- Mõõtetulemuse tüüp
- Seeria tüüp
- Ajavahemik (algus ja/või lõpp)
- Lehekülje suurus
- Leheküljenumber



Joonis 15. Mõõtetulemuste pärimise tulemus.

Sündmuste pärimine

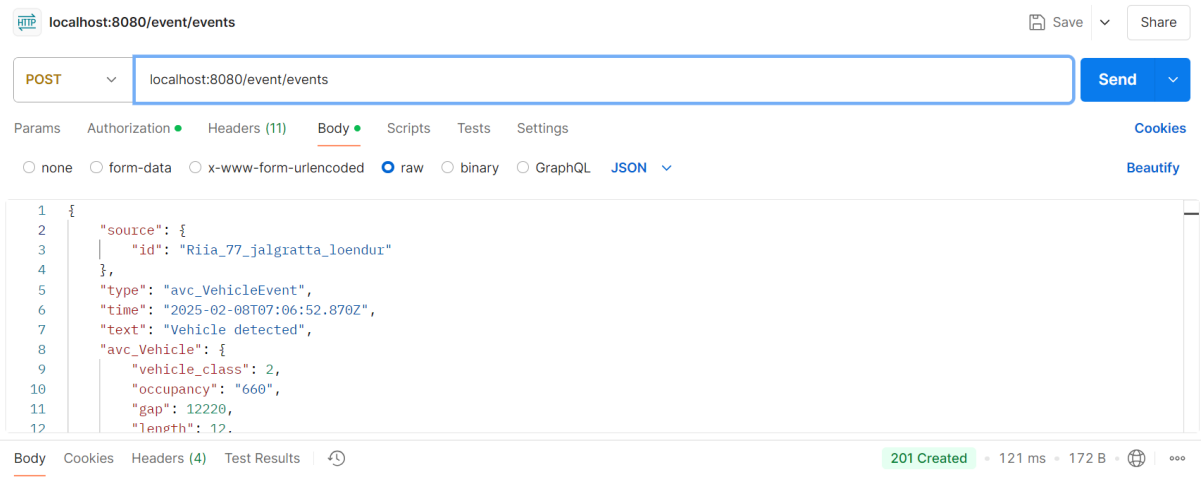
Sündmuseid saab API kaudu pärida (F6). Rakendus ei võimalda parameetreid kasutada SQL-süsti läbiviimiseks (MF8). Sündmuste pärimine töötab analoogselt mõõtetulemuste pärimisele. Sündmuste pärimisel on võimalik kasutada järgnevaid parameetreid (F7, F10, F16, F17):

- Sündmuse tüüp
- Seadme ID
- Seadmetüüp
- Seadmegrupp
- Ajavahemik (algus ja/või lõpp)
- Fragmendi tüüp
- Lehekülje suurus

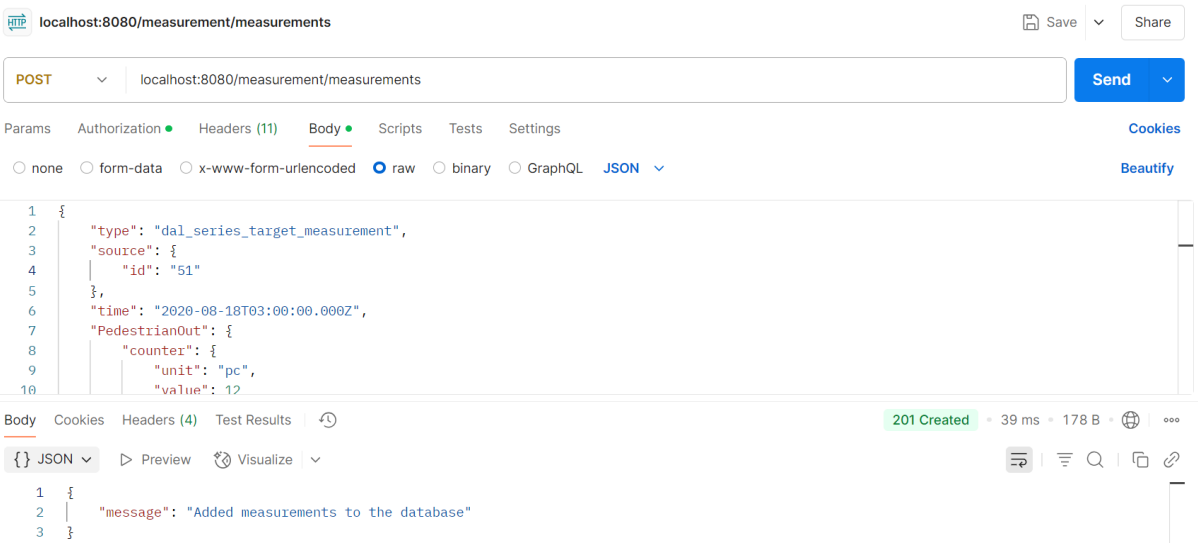
- Leheküljenumber

Tagasiühilduvus ja seire

Rakendus on võimeline vastu võtma Cumulocity-struktuuriga sündmusi (vt joonis 16) ja mõõtetulemusi (vt joonis 17), et lihtsustada üleminekut uuele platvormile. Rakendus kogub ja edastab Prometheuse abil andmeid päringute arvu, kestvuse ja tagastatud olekukoodi kohta (MF9).



Joonis 16. Cumulocity-struktuuriga sündmuse sisestamine.



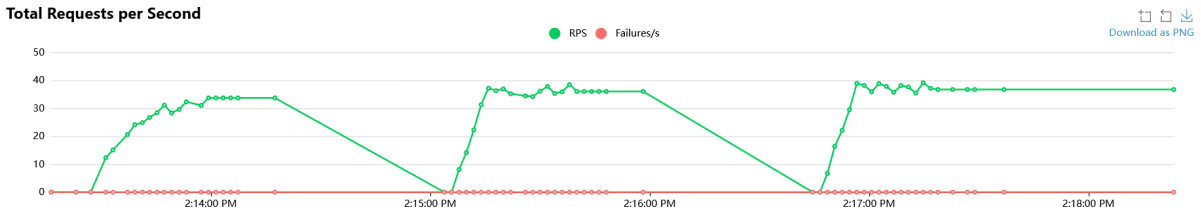
Joonis 17. Cumulocity-struktuuriga mõõtetulemuse sisestamine.

5.2 Võrdlus varasema lahendusega

Eelmine lahendus võttis 2024. aasta oktoobris vastu umbes 27,7 miljonit päringut[39]. Arvestades, et need arvud muutuvad pidevalt ja et arvutusi lihtsustada, ümardame selle 30 miljoniks. See teeb päeva kohta umbes 1 miljon päringut, arvestades, et kuus on 30 päeva. Nõue MF1 eeldab, et valmiv rakendus saaks hakkama suurema arvu päringutega, kui Cumulocity.

Välja selgitamaks, kas uus rakendus sellele nõudele vastab, korraldas autor katse. Selleks kasutas ta Locust tarkvara[40], kus on võimalik näiteks simuleerida päringuid, nagu tuleks need päris seadmelt. Katse korraldati HP EliteBook 840 G8 sülearvutil. Samas Dockeri keskkonnas jooksid QuestDB, PostgreSQL, API ja Locust. Neile anti kokku 8 gigabaiti muutmälu ja 4 protsessorituuma. Üks test kestis viis minutit ja päringuid tegi 2000 kasutajat. Kasutajate arv otsustati selle põhjal, et 2024. aasta detsembris kasutas rakendust umbes 1500 seadet[41]. Tõsi, nad ei saada kõik korraga andmeid ja päeval saadetakse rohkem andmeid, kui öösel, sest inimesi ja sõidukeid liigub siis rohkem.

Locusti jaoks loodi Pythoni skript, mis lõi 20 sekundi jooksul 2000 seadet, mis hakkasid saatma mõõtetulemusi. Kokku kestis üks test 5 minutit. Joonisel 19 on näha, mitu päringut sekundis testi vältel tehti. Mõõtetulemuste aeg ja mõõdetud väärtus genereeriti iga mõõtetulemuse jaoks uuesti.

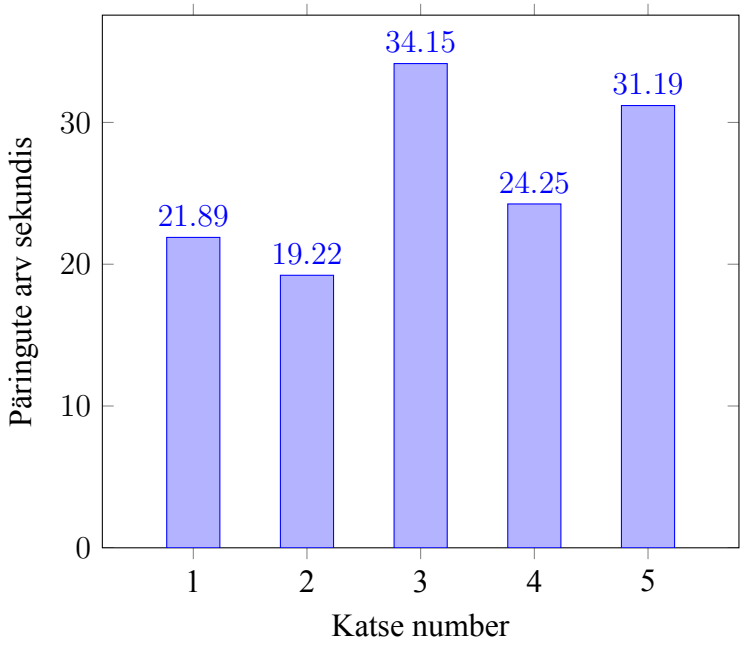


Joonis 18. Päringute arv sekundis Locusti testi vältel.

Selleks, et täita nõuet MF1, peaks API olema võimeline vastu võtma keskmiselt rohkem kui 11,57 päringut sekundis (päevas on 86400 sekundit ja 1 miljoni päringu jaoks peab igas sekundis käitlema 11,57 päringut). Katse tulemustest (vt joonis 19) selgus, et see läks täide igal korral.

[39] Info pärineb eelmise keskkonna kasutusandmete väljavõttest oktoobris 2024. Autorile näitas seda juhendaja.
[40] Locust. https://locust.io/.
[41] Väljavõte eelmise keskkonna kasutusandmetest.

Arvestades, et päriselt hakkab rakendus jooksma eraldi keskkonnas ja mitmes konteineris, on see tulemus positiivne. Saab tõdeda, et MF1 on täidetud.



Joonis 19. Rakenduse poolt vastuvõetavate päringute arv sekundis.

5.3 Funktsionaalsete nõuete tätmine

Kirjutamise ajaks jäid funktsionaalsetest nõuetest täitmata F9, F18, F20. F9 ja F20 puudutavad seadmete ja kasutajate kustutamist. Kasutaja kustutamisel võib tekkida olukord, kus seadet ja tema poolt saadetud andmeid ei ole võimalik enam pärida. Teisest küljest seadme süsteemist kustutamisel tekib olukord, kus füüsiliselt eksisteeriv seade ei saa enam andmeid saata, sest andmete saatmisel kontrollitakse seadme olemasolu süsteemis. Keerukust lisab veel peatükis 4.1 väljatoodud QuestDB omadus, et sellel puudub eraldi DELETE märksõna. Potentsiaalsete probleemide vältimiseks otsustati need funktsionaalsused hetkel välja jätta. Õnneks ei häiri kustutamise võimaluse puudumine rakenduse tööd, sest vajadusel saab luua uue seadme või kasutaja, kui nende informatsiooni muutmine ei ole piisav.

F18 tähendab päringu tulemusena saadud väärtuste sorteerimist kronoloogiliselt kahanevas järjekorras. See on kasulik näiteks selleks, et pärida kõige uuemaid andmeid. Tehnilise poole pealt ei ole tegemist raske nõudega, sest QuestDB päringukeelde on selline funktsionaalsus sisse ehitatud. Selleks on kasutusel märksõna LATEST ON [28]. Selle abil saab valida ajatempli, mille alusel sorteeritakse tabelis olevad väärtused nii, et uuemad on eespool [28]. Nõue jäi täitmata, sest see oli kõige madalama prioriteediga ja kõrgema prioriteediga nõuded vajasid täitmist.

5.4 Mittefunktsionaalsete nõuete täitmine

Kirjutamise hetkeks on täitmata kaks mittefunktsionaalset nõuet (MF3, MF4). MF3 tähendab võimalust ründe korral API võtmeid muuta. APIga paralleelselt saab kasutada otse PostgreSQL andmebaasiga suhtlemiseks loodud lahendusi, mis on loodud rakenduse toimimisest sõltumatud. See tähendab, et selle liidese kaudu saab näiteks rünnaku korral API võtmed deaktiveerida.

MF4 on nõue, et API peab kasutama HTTPS protokolli. Põhimõtteliselt on see täitmata, sest API ise seda ei kasuta. Küll aga toimub suhtlus API ja klientide (kasutajad ja seadmed) vahel läbi nginx[42] serveri, millel on HTTPS tugi olemas[43]. Selline lähenemine ei ole midagi ebatavalist ja kuna nginx konfiguratsioon on sarnane olenemata tehnoloogiast, mida kasutatakse rakenduse arendamisel, ei ole mõistlik hakata eraldi selgeks tegema, kuidas seda sama asja näiteks FastAPI puhul teha. See seati üles juhendaja poolt ja APIst sõltumata, seega on API turvatud, aga autor ei saa seda enda nimele kirjutada.

5.5 Valideerimine

Funktsionaalseid nõudeid oli kokku 20, millest täidetud sai 17. Mittefunktsionaalseid nõudeid oli kokku 9 ja neist sai täidetud 7. Seega on täidetud umbes 83% nõuetest, mis iseenesest tundub suhteliselt väikese protsendina. Vaadates arvudest natuke sügavamale ja toetudes kahele eelmisele alapeatükile, ei olnud täitmata nõuded kriitilise tähtsusega. Kõige olulisem oli luua rakendus, mis on võimeline eelmist lahendust asendama, võtma vastu selle poolt defineeritud struktuuriga andmeid ja pakkuma suuremat sisestusvõimet. Need omadused on loodud rakendusel olemas.

5.6 Rakenduse kasutuselevõtmine

Selle peatüki kirjutamise hetkeks on rakendus Tartu linna poolt kasutuses. Rakendus jookseb kahes Dockeri konteineris ja sinna saadab andmeid kolm ettevõtet. Kokku on saadetud umbes 22 miljonit mõõtetulemust ja umbes 15,5 miljonit sündmust. Sündmuste saatmistihedus on suurem hommikul ja lõunal ning eriti suur õhtuse tipptunni ajal (vt joonis 20). See on loogiline, sest nendel aegadel liigub kõige rohkem jalakäijaid ja sõidukeid. Mõõtetulemuste puhul on näha, et 8. mail on ilmselt hakanud rohkem seadmeid andmeid saatma. See graafik võiks tegelikult olla

[42] nginx. https://nginx.org/.
[43] nginx. Configuring HTTPS servers. https://nginx.org/en/docs/http/configuring_https_servers.html.

ühtlasem, sest mõõtetulemusi peaks saadetama kindlate intervallidega. Graafikult tuleb aga välja, et kindlal hetkel päevas on saadetavate mõõtetulemuste arv ülejäänud ajast oluliselt väiksem. See ei pruugi iseenesest probleem olla, sest see võib olla põhjustatud seadmete tahtlikult nii seadistamisest. Siiski on tegemist hea näitega, miks peaks rakenduse talitluse kohta käivaid andmeid koguma ja neid analüüsima.

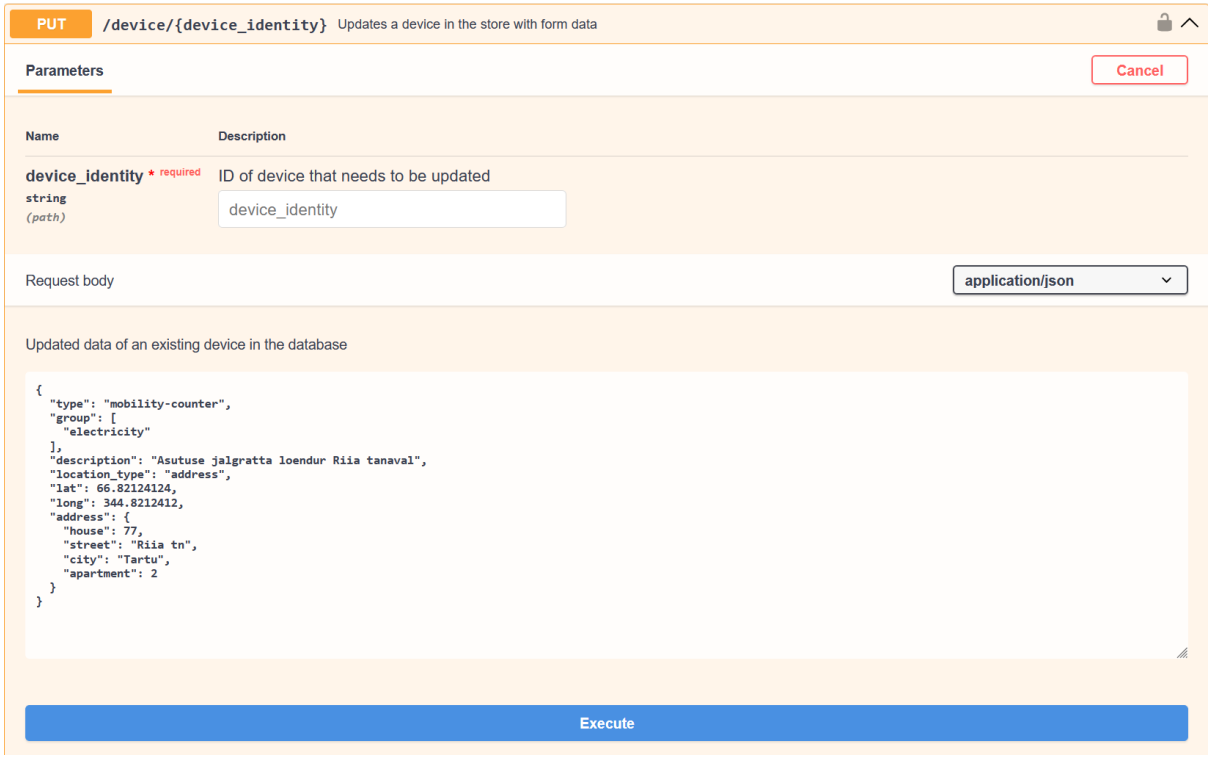
Joonis 20. Saadetud südmuste ja mõõtetulemuste arv 10 minuti jooksul nädala lõikes.

5.7 Edasiarendused

Selles peatükis tuuakse välja, mida võiks veel rakendusele lisada, et tõsta selle kasutusmugavust ja jõudlust.

5.7.1 Eessüsteem

Praeguse lahenduse puhul on rakendusega graafilise kasutajaliidese kaudu suhtlemine ebapraktiline. Näiteks seadme muutmisel tuleb käsitsi muuta JSON-objekti sisu (vt joonis 21), kasutajate muutmine töötab analoogselt.



Joonis 21. Seadme uuendamine Swagger UI kaudu.

Selle parandamiseks oleks mõistlik luua eraldiseisev eessüsteem. Seal saaks sisse logida, vaadata vastavalt oma õigustele seadmeid ja/või kasutajaid ja neid filtreerida. Seejärel saaks valida sobiva kasutaja või seadme ja selle peale klõpsates avaneks vaade, mis sisaldab vastava objekti infot sisaldavat blanketti, kus saab vastavalt vajadusele väärtuseid muuta.

5.7.2 Kubernetes

Kubernetese tutvustuses[44] tuuakse välja, et see aitab automatiseerida konteineriseeritud rakenduste käitamist. Näiteks saab see vajadusel automaatselt tekitada uusi konteinereid, kui ajutiselt on vajadus suurema jõudluse järele ja pärast need kontrollitult eemaldada.

[44] Kubernetes - Overview https://kubernetes.io/docs/concepts/overview/.

Käesoleva töö kontekstis on Kubernetesest kõige rohkem kasu andmebaasi konteinerite haldamisel. Näiteks saab andmeid dubleerida erinevates konteinerites, et suurendada nende säilimise tõenäosust, kui ühe konteineriga midagi juhtub. Need konteinerid saavad kindluse mõttes joosta ka näiteks erinevates füüsilistes masinates.

Lisaks on FastAPI jaoks hea omadus automaatne konteinerite loomine ja eemaldamine, sest teatud hetkedel võib koormus olla suurem. Näiteks suurkontsertide algus- ja lõppaja ümbruses liigub linnast välja ja vastupidi tunduvalt suurem hulk autosid, kui suvalisel muul päeval samal ajal. Sellistel hetkedel on oluline kiiresti lisajõudlust tekitada, et süsteemi ei koormataks üle ja muude allikate andmed ei jääks sisestamata.

6. Kokkuvõte

Töö eesmärk oli luua Tartu linnale targa linna andmete haldamise platvorm. Loodav lahendus pidi lihtsustama andmete saatmist ja integratsioonide loomist. Oluline oli toetada eelmise lahenduse poolt kasutatavaid andmestruktuure, kuid samas toetada suuremat arvu päringuid.

Töös anti ülevaade sellest, kuidas FastAPI raamistikku ja QuestDB andmebaasi kasutades sellist rakendust luua ja miks neid kasutada. Arutleti QuestDB ja üldiselt kronoloogiliste andmebaaside omapärade, andmemudelite koostamise ja andmete tabeliteks jaotamise üle. Anti ülevaade FastAPI tugevustest ja nõrkustest targa linna andmete käitlemisel.

Töö käigus valmis rakendus, mis võimaldab seadmete ja kasutajate haldust, andmete vastuvõtmist ja pärimist. Arutleti selle üle, miks osa nõuetest täitmata jäi, kuidas ja kas üldse neid täita.

Täitmata funktsionaalsetest nõuetest 2 olid seotud seadmete ja kasutajate kustutamisega. Mõlema tegevusega kaasnesid oma probleemid ja nende puudumine ei häiri rakenduse tööd, seega otsustati need töö raames kõrvale jätta. Kolmas nõue on andmete pärimisel nende kronoloogiliselt kahanevas järjekorras sorteerimine. Ka see ei olnud vajalik, seega otsustati nõue kõrvale jätta, kuid toodi välja, kuidas see funktsionaalsus lisada.

Mittefunktsionaalsete nõuete puhul jäid täitmata HTTPS protokolli kasutamine ja ründe korral API võtmete muutmine. HTTPS protokoll on küll kasutusel, kuid mitte rakenduse, vaid seda ja välismaailma ühendava vahekihi poolt. Seega on selle töö raames nõue ametlikult täitmata. API võtmete muutmise nõue jäi täitmata, sest selleks ei leitud head moodust, kuid pakuti välja alternatiiv.

Rakendus on juba suhteliselt pikalt Tartu linna poolt kasutuses olnud. Valideerimiseks saigi seega lisaks lokaalsetele testidele kasutada kliendi poolt tulnud tagasisidet. See on sellise projekti puhul hindamatu väärtusega, sest kõik probleemid ei pruugi arendusprotsessis ennast ilmutada. Üldiselt võib tulemusega rahule jääda, sest rakendus töötab ja suuremad probleemid said parandatud.

Tulevikus oleks mõistlik lisada rakendusele eesüsteem, et sellega suhtlemist mugavamaks teha. Kauges tulevikus oleks võib-olla kasu ka Kubernetese toest, aga selleks, et sellest kasu oleks, peaks koormus oluliselt suurenema.

Viited

[1] Gomstyn A. ja Jonker A. What is a smart city? https://www.ibm.com/think/topics/smart-city (Viimati külastatud: 21.04.2025).

[2] Sarma S. ja Sunny S. A. Civic entrepreneurial ecosystems: Smart city emergence in Kansas City. *Business Horizons* 60.6 (2017), lk 843–853.

[3] Tark Tartu. https://tarktartu.ee/ (Viimati külastatud: 06.05.2025).

[4] Cumulocity - API. https://cumulocity.com/api/core/ (Viimati külastatud: 17.04.2025).

[5] Leier M. Cumulocity IoT platform connects all devices into one integrated system. https://iot.ttu.ee/cumulocity-iot-platvorm-liidab-koik-seadmed-uhtsesse-susteemi/ (Viimati külastatud: 06.05.2025).

[6] Cumulocity - UI functionalities and features. https://cumulocity.com/docs/get-familiar-with-the-ui/gui-features/ (Viimati külastatud: 06.05.2025).

[7] Barić M. The Most Popular Python Frameworks in 2024. https://codeanywhere.com/blog/the-most-popular-python-frameworks-in-2024 (Viimati külastatud: 21.04.2025).

[8] Top 10 Python REST API Frameworks in 2025. https://www.geeksforgeeks.org/top-python-rest-api-frameworks/ (Viimati külastatud: 21.04.2025).

[9] Leapcell. 2025's Top 10 Python Web Frameworks Compared. https://dev.to/leapcell/top-10-python-web-frameworks-compared-3o82 (Viimati külastatud: 21.04.2025).

[10] Ghimire D. Comparative study on Python web frameworks: Flask and Django (2020).

[11] De Luca G. FastAPI Cookbook: Develop high-performance APIs and web applications with Python. en. Birmingham, England: Packt Publishing, august 2024.

[12] Lubanovic B. FastAPI. Ö'Reilly Media, Inc.", 2023.

[13] FastAPI - Dependencies. https : / / fastapi . tiangolo . com / #dependencies (Viimati külastatud: 08.12.2024).

[14] QuestDB - PostgreSQL & PGWire. https://questdb.com/docs/reference/api/postgres/ (Viimati külastatud: 29.04.2025).

[15] QuestDB - Modify Data. https://questdb.com/docs/guides/modifying-data/ (Viimati külastatud: 17.04.2025).

[16] QuestDB - Web Console. https : / / questdb . com / docs / web - console/ (Viimati külastatud: 11.05.2025).

[17] Srinath K. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology* 4.12 (2017), lk 354–357.

[18] Pechkurov A. Benchmark and comparison: QuestDB vs. InfluxDB. https://questdb.com/blog/2024/02/26/questdb-versus-influxdb/ (Viimati külastatud: 08.12.2024).

[19] QuestDB - Python Client Documentation. https://questdb.io/docs/clients/ingest-python/ (Viimati külastatud: 08.12.2024).

[20] InfluxData - InfluxDB line protocol tutorial. https://docs.influxdata.com/influxdb/v1/write_protocols/line_protocol_tutorial/ (Viimati külastatud: 08.12.2024).

[21] Docker - What is Docker? https://docs.docker.com/get-started/docker-overview/ (Viimati külastatud: 08.12.2024).

[22] QuestDB - Using Docker with QuestDB. https://questdb.io/docs/deployment/docker/ (Viimati külastatud: 08.12.2024).

[23] FastAPI in Containers - Docker. https://fastapi.tiangolo.com/deployment/docker/ (Viimati külastatud: 08.12.2024).

[24] QuestDB - Schema Design Essentials. https://questdb.com/docs/guides/schema-design-essentials/ (Viimati külastatud: 06.05.2025).

[25] Cumulocity - SmartREST. https://cumulocity.com/docs/smartrest/smartrest-introduction/ (Viimati külastatud: 17.04.2025).

[26] QuestDB - Root directory structure. https://questdb.com/docs/concept/root-directory-structure/ (Viimati külastatud: 06.05.2025).

[27] QuestDB - Role-based Access Control (RBAC). https://questdb.com/docs/operations/rbac/ (Viimati külastatud: 29.04.2025).

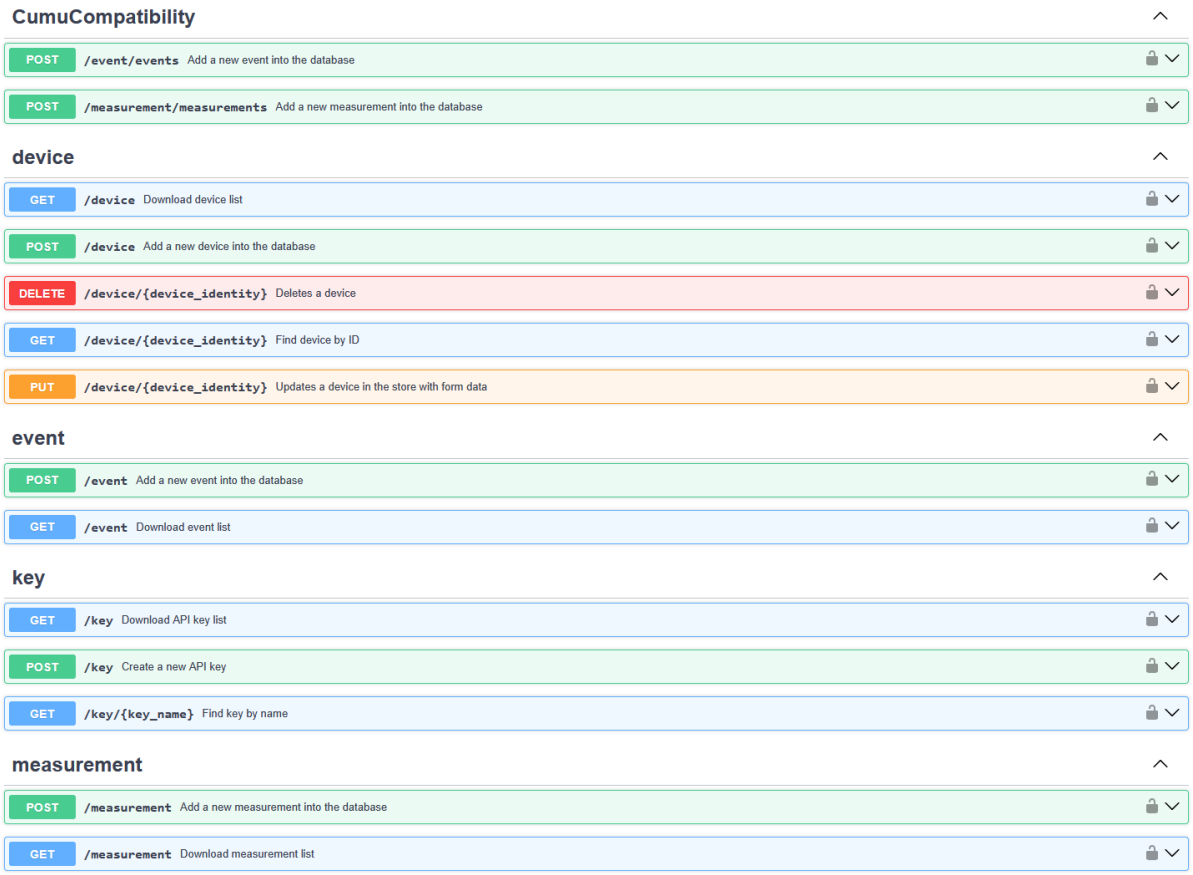
[28] QuestDB - LATEST ON keyword. https://questdb.com/docs/reference/sql/latest-on/ (Viimati külastatud: 02.05.2025).

Lisad

I Rakenduse lähtekood

Rakenduse lähtekood on kättesaadav GitLab keskkonnas: https://gitlab.cs.ut.ee/tartu-project-timeseries/tartu-timeseries-db-api. Ligipääsu saamiseks võib ühendust võtta autori (meiliaadressil madislaurson@gmail.com) või juhendajaga.

II OpenAPI spetsifikatsioon



Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Madis Jaagup Laurson,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

   Asjade Interneti API loomine targa linna andmete halduseks,

   mille juhendaja on Pelle Jakovits,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada Tartu Ülikooli digitaalarhiivi kuni autoriõiguse kehtivuse lõppemiseni;

2. annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni;

3. olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile;

4. kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Madis Jaagup Laurson

15.05.2025