

Importing of live parking data from  
several external sources into bloTope  
data model

<b>Summary of the work performed</b>	<b>4</b>
<b>Detailed Description</b>	<b>5</b>
Concept, Objectives, Set-up, and Background	5
Motivation	5
Challenges	5
Solution	6
Technical Results	8
Solution Architecture overview	8
Generic Software Components	9
O-MI writer	9
ODF Converter	10
API Connectors	10
Native API Connector	10
Authenticator and Authorizer	10
Runtime Environment	10
Problems with performance observed	11
Parkki Hubi Implementation	12
Description of data source	12
Software components	14
The Parkkihubi connector	15
The Parkkihubi data converter	15
The Parkkihubi data storage	15
Park and Ride Implementation	16
Description of data source	16
Software components	18
The Park and Ride connector	19

The Park and Ride converter	19
Residential and Private Parking Case	20
Description of data source	20
Architecture and Software Components	21
Webapp	21
Rest Interface	22
Authentication storage	23

## A. Summary of the work performed

The result of the work performed have been reported in two deliverables.

The first deliverable “Proof-of-concept: Importing of live parking data from external sources into bloTope data model”<sup>1</sup> has been delivered and approved in October 2018. The deliverable covers the following work items:

- The initial implementation and deployment architecture has been developed and documented;
- The related technologies and external components has been studied and selected;
- Several potential data sources have been identified and studied;
- Meetings/communications with the selected data providers have been performed;
- Potential integration points with other bloTope ecosystem’s participants have been identified and discussed on face-to-face workshops and events;
- Local development and testing environments for the parts of the system, including O-MI node and Parkkihubi, have been set up;.
- Some initial ideas about business model have been developed and documented.

The second deliverable (this document) “Importing of live parking data from several external sources into bloTope data model” covers the following tasks, functionalities and activities:

- Final architecture overview, including common software components and runtime environment;
- Pakki HUBI integration: architecture, implementation, examples;
- Park and Ride integration: architecture, implementation, examples;
- Residential and other private parking (Airbnb-like service for parking): architecture, implementation, examples;
- Supporting documentation: cookbook for the developers;
- Overview of business impact and business models;
- Overview of dissemination activities and material.

---

<sup>1</sup> Can be found in temporary project storage:  
[https://drive.google.com/open?id=1uBugMgjW-5PL9jrpOOnSusev0Y\\_Aca264CYjHDG7fps](https://drive.google.com/open?id=1uBugMgjW-5PL9jrpOOnSusev0Y_Aca264CYjHDG7fps)

## B. Detailed Description

### B1. Concept, Objectives, Set-up, and Background

#### Motivation

There are two main drivers for integrating different sources of parking and charging data into bloTope ecosystem.

The first one is related to “silo”- development of existing charging and parking solutions. Even already existing parking and charging produce large amount of data about real-time parking. However, despite some initiatives to integrate different systems, agree on some standards, and deliver information to the end-user in a convenient and friendly way, there are still a lot of business obstacles and technical challenges, e.g. resistance of the service suppliers to integrate with the competitors, lack of understanding of the benefits from such an integration, lack of clearly defined business models, and number of challenges related to different data models and interfaces. At the moment charging and parking service suppliers are mostly using generated data for their operational needs. However, bioTope ecosystem allows using of data for developing new data-driven services, new business based on this data and thus better monetization the data.

Another driver is a need to integrate charging and parking use cases for electric vehicle. Though for “normal” petrol cars these two use cases are completely different, electrical cars need charging and parking often at the same time. When EV is using slow charging services it at the same time using a parking slot, which implies different needs for integration of data and services, including information on availability and location, authorization to use the services, payment systems. On the other hand, electric vehicles are more limited in time and area for driving and searching for parking places and charging slots. EV drivers have urgent needs to know where is a free parking slot and the closest possibility to charge.

#### Challenges

The challenges that have been addressed and solved in this project can be summarized as the following:

- To overcome the “silos” of parking and charging service by providing a framework and reference implementation for adding new charging and parking service providers into the bloTope ecosystem without programming effort from the service and data suppliers;
- To provide proof-of-concept on at least two real data and service providers and demonstrate results and benefits of such integration;
- To develop a concept and provide implementation for integration of residential and other parking facilities, to create Airbnb-like service for parking;
- To provide instructions for the developers how to use the developed solutions, how to integrate new data sources and services;
- To suggest business models for sustainable businesses based on bloTope-based technologies and ecosystem, illustrated with the parking and charging examples;
- To disseminated information about the developed solution and framework through different activities.

In fact, there is no application for the drivers that would provide all information from all possible parking and charging supplier and the same level of services from different charging and parking service supplier. Apart of bloTope, there is no common framework, platform, APIs, concepts that would enable the development of such an application in an open and standardized way.

All challenges above are characterised by high complexity, both technical and business, novelty due to relatively new technology and market immaturity, and high level of uncertainty due to a lack of evidence in implementing similar approaches on the market and unpredictable reaction of business and industries.

## Solution

AVRORASoft developed a mechanism and instructions for importing and hosting of external live parking data from different available sources into bloTope ecosystem, made it available in O-DF MobiVoc (Biotope standards) format, published and hosted in O-MI nodes. In scope of this project data from Parkki HUBI and Park and Ride services have been integrated to bloTope. Moreover, the mechanism to integrate different residential and other private parking places has been implemented and explained.

The developed code is pair-reviewed and published on Github. AVARORASoft has provided instructions for the developers how to access the developed code, how to use

APIs, how to create a new solution based on the developed framework, e.g. how to add a new data source to the ecosystem.

## B2. Technical Results

The overview and description for the proposed architecture of general software components have been provided in detail in the previous deliverable “Proof-of-concept “Importing of live parking data from external sources into bloTope data model”, part B.2. This deliverable focuses on the **realized architecture and implementation** of the specific components developed for different data sources.

### Solution Architecture overview

In a nutshell, the solution integrates different data sources and deploys data to O-MI nodes.

The implementation follows the application design of serverless architecture as it shown on Figure 1, which implies using third party backends as a service. The realized architecture is slightly different than the one proposed in the first deliverable. The main difference is that common components have been implemented as libraries, not lambdas, as it was originally planned. The business logic is divided into small functions and implemented as a code that runs in managed and short living containers. These functions perform specific tasks and have no internal state. The execution of these functions is triggered by events generated from external and internal events.

Serverless platforms are available as a service from the most of cloud providers, such as Amazon, Microsoft, and Google

The current solution uses AWS Lambda, because it is the oldest and the most mature, it has a rich ecosystem for open third party solutions.

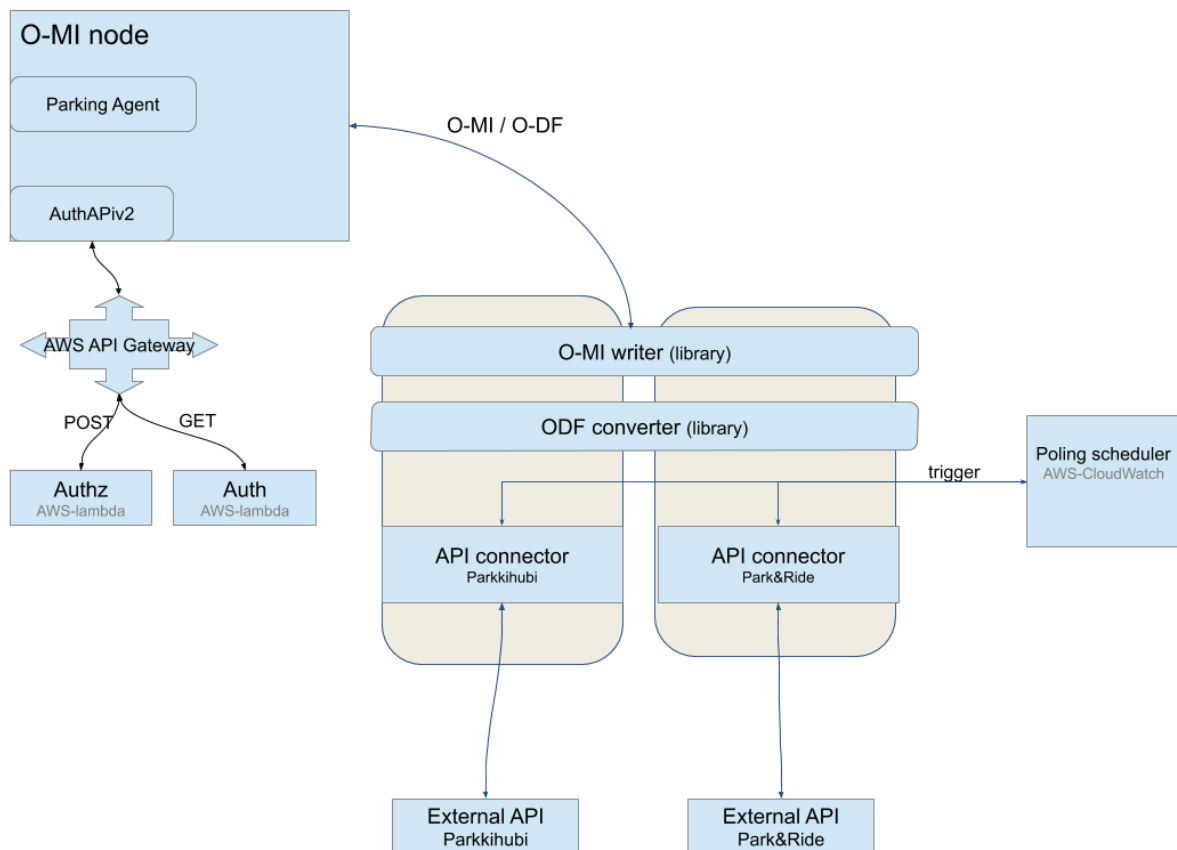
AWS Lambda functions can be written using Node.js, Java, C#, Go or Python.

AWS Serverless Platform is used for computing tasks like mapping the data between different object models. The chosen solution implements “pay as you use” pricing strategy.

The lambdas connected to the external APIs are triggered either by a timer or in case the external API supports push request the trigger is through AWS API gateway as in the case of the native API.

The lambda in this context refers more to related functionality that has been implemented as many separate AWS lambdas. For example the native API lambda is a collection of AWS lambdas as the API consists of not only the data update functionality but also some administrative functions are needed.





**Figure 1. Simplified high level architecture of the solution**

The solution can contain more than one O-MI node as some data providers might want to have a O-MI node dedicated to them. Although the O-MI node itself hasn't been in the scope of the proposal, some O-MInode related work has been done. This includes the containerization of the O-MI node and the runtime configuration.

The developed software components are divided to the common components that are generic for all types of integrated data sources and specific for certain data sources and APIs.

## Generic Software Components

### O-MI writer

The O-MI writer is a javascript library function that performs a write request to the correct O-MI node. The OMI writer does not do any data format conversions.

## ODF Converter

The ODF Converter is a javascript library function that creates a valid XML- format O-DF message out of its input.

## API Connectors

Every data provider has its own connector that has been implemented as an Amazon lambda function. The connectors take care of obtaining the data from the external source and adapting it the rest of the system. Depending on the configuration, the connector can be triggered by a timer or by push from the provider.

The connectors have two different operations:

- I. The initial data load including a new charging station and a parking place discovery.
- II. The update of the current real-time status of the charging station and the parking place.

## Native API Connector

A REST interface for residential data providers, triggered from API gateway. The native API connector allows creation of a new parking facility and parking places. The attributes of the parking facilities and places can also be updated using the connector. Nativ API connector also updates the authentication storage when creating a parking facility.

## Authenticator and Authorizer

The implement the needed functionality for the O-MI node authAPI.v2 authentication and authorization. For the 'bulk' data connectors (parkkihubi, and park and ride) the access token is simply read from the execution environment and the authorization allows all access for the ParkingFacilities handled by the agent. For the Residential parking case the access control is more fine grained access allowing access only for the invoking users own parking facilities.

## Runtime Environment

The O-MI node is deployed to Amazon Elastic Container Service (ECS) as a single docker container. This allows easy scaling of the runtime environment and takes care of the monitoring and recovery of the O-MI node. It also allows an easy way to start additional O-MI node instances if needed for example development purposes. The configuration and

persistent data of the O-MI node are on separate docker volumes backed by the local storage the Amazon elastic computing cloud (EC2) instance running the ECS.

The container approach also allows easy changes to the deployment, for example adding a proxy server in front of the O-MI node to shield it from certain types of attacks.

The O-MI node has been configured to use the authAPI.v2 interface. The authAPI.v2 interface uses HTTP GET-method for authentication and HTTP POST-method for authorization. The authentication and authorization endpoint is an Amazon API gateway connecting to two Amazon lambda functions, one for authentication, another for authorization. The API gateway endpoint is protected with an API-key that has been configured for the authAPI.v2 and is send as on HTTP-header in every request. The actual authentication and authorization is based on a token that is extracted from the authorization header of the O-MI request. The Parkkihubi and Park&Ride integration obtain this token from the process environment. In the Residential and Private parking case the token is obtained from Amazon Cognito user pool based on the credential supplied by the user.

The integration functions have been written using ECMAScript 2017. The source code is transpiled with babel and packaged with webpack into a single file to be used as Amazon lambda function. The transpiling is needed because the Amazon labda javascript runtime does not directly support ECMAScript2017.

## Problems with performance observed

During the course of the project implementation two O-MI node performance problems has been discovered. Both of which were related to the parking agent implementation. The first problem was write performance degradation after the system contained several hundreds of parking facilities. This problem is fixed in O-MI node version 1.0.8.

Second problem is slow read performance with the findParking action of the parking agent. When querying the parking facilities for an area having many (in order of hundreds) parking facilities the processing of the reply is too slow for interactive use. The reply is also big (around 11MB with the parkkihubi and park and ride data), this is mainly due to the verbosity of the XML-format and to the fact that findParking method does not allow limiting the amount of data returned.

Here is an example of findParking action execution and the server logs. As it can be seen the execution took over a minute:

```
O-MI server logs:
2019-03-19 08:37:20,208 INFO ParkingAgent - Current indexes has: 1788 parking facilities
2019-03-19 08:37:20,208 INFO ParkingAgent - Found 1468 parking facilities close to destination
2019-03-19 08:38:07,864 INFO ParkingAgent - Found total 12383 of valid parking spaces and capacities

Command execution:
/usr/local/bin/node /Users/reenari/HEL/Avrora/avrora/parkki/dist/findParking.js
2019-03-19T08:37:20.141Z
send 1: undefined
2019-03-19T08:38:39.189Z
onFly 0, rcvCount 1, data.length 11485805
2019-03-19T08:38:39.502Z
```

## Parkki Hubi Implementation

### Description of data source

Parkkihubi is an open REST interface that provides real time information about street side parking spaces in the southern Helsinki. It gathers the information using the parking fee payments data. The original goal of Parkkihubi project was to provide information about parking to the companies that are checking parking and giving penalties for wrong parking. However, according to general Helsinki open data and open interfaces principles Parkkihubi has exposed their APIs and made the anonymised data open and available for everybody and free of charge. bloTope is the first project that started cooperation with Parkkihubi project even before APIs have been officially released. It has been discussed with Parkkihubi project manager Lauri Uski that if necessary Parkkihubi can collect data about charging spots available in the area and implement support for that in APIs. bloTope is the first project that is using APIs and Parkkihubi data.

The two most notably limitations of the data are:

1. There is no information about the parking spaces occupied by the resident parking permit holders and
2. There is no information during the time when parking is free of charge.

The API provides two endpoints 'Parking area' and 'Parking Area Statistics'.

'Parking Area' API endpoint provides general information about all available parking places. The information includes the parking area id, its estimated capacity and the

geometry of the parking area. With the API one can query information about all the parking places or of one specific area identified by its id. The geometry of the parking area is given as a multipolygon.

Here is an example request and reply:

```
https://pubapi.parkkiopas.fi/public/v1/parking_area/?format=json&page_size=1&page=14
```

```
{
  "type": "FeatureCollection",
  "count": 1830,
  "next": "https://pubapi.parkkiopas.fi/public/v1/parking_area/?format=json&page=15&page_size=1",
  "previous":
  "https://pubapi.parkkiopas.fi/public/v1/parking_area/?format=json&page=13&page_size=1",
  "features": [
    {
      "id": "0b4beedd-29bd-4894-8fc5-1668adee14b0",
      "type": "Feature",
      "geometry": {
        "coordinates": [
          [
            [24.925439053093953, 60.177038117819784],
            [24.926793124327016, 60.177187537845334],
            [24.9268163513672, 60.17723187860529],
            [24.925464761081148, 60.177082216790836],
            [24.925439053093953, 60.177038117819784]
          ]
        ]
      },
      "type": "MultiPolygon"
    },
    {
      "properties": {
        "capacity_estimate": 21
      }
    }
  ]
}
```

The 'Parking Area Statistics' API endpoint provides current parking information about parking places. The returned information has the number of parkings in the area (i.e possible free places are estimated capacity minus current parkings). With the API one can query information about all the parking places or of one specific area identified by its id. Also if there are less than 3 parking lots in area the number of parking lots returned is 0.

```
https://pubapi.parkkiopas.fi/public/v1/parking_area_statistics/?page_size=1&page=14&format=json
```

```
{
  "count": 1830,
  "next":
  "https://pubapi.parkkiopas.fi/public/v1/parking_area_statistics/?format=json&page=15&page_size=1",
  "previous":
  "https://pubapi.parkkiopas.fi/public/v1/parking_area_statistics/?format=json&page=13&page_size=1",
  "results": [
    {
      "id": "0b4beedd-29bd-4894-8fc5-1668adee14b0",
      "current_parking_count": 0
    }
  ]
}
```

There are neither change notifications nor a way to subscribe to data changes, which means that users of the API must be periodically polled to get the current information.

The documentation of the API can be found at <https://api.parkkiopas.fi/docs/public/>. The API itself is available at <https://pubapi.parkkiopas.fi/public/v1/> and is publicly available without authentication.

## **Software components**

The Parkkihubi integration to O-MI-node uses the following generic components (described in detail in the section Solution Architecture Overview):

- Polling scheduler
- O-MI writer
- ODF converter

Additionally, the following Parkkihubi specific components have been developed:

- Parkkihubi connector
- Parkkihubi data converter
- Parkkihubi data storage

### ***The Parkkihubi connector***

The Parkkihubi connector consists of two AWS (Amazon web services) lambda functions. First (readAll()) for reading the parking facility information (parking facility locations and capacity estimations), and a second (updateParkings())for reading the real time parking information. Both functions are periodically triggered by the polling scheduler. First one is triggered only at night time once a week or if the second function encounters an unknown parking facility. The second function is triggered every 15 minutes during daytime. The polling durations are based on the observations how often the data changes in the source.

### ***The Parkkihubi data converter***

The Parkkihubi data converter is a library function that converts the Parkkihubi data model to the Mobivoc format which is then converted by the ODF converter to ODF XML format and finally sent to O-MI writer. For each Parkkihubi parking\_area a corresponding Mobivoc parkingFacility is generated. The coordinates of the parking facility are simply the center coordinates of the parking\_area polygon. The parking\_area capacity estimate is then used to generate mobivoc parkingSpaces under the parkingFacility.

### ***The Parkkihubi data storage***

The Parkkihubi data storage is an AWS DynamoDb database. It provides lookup of the capacity estimate by the parking facility id and is used by the Parkkihubi connector. The capacity estimates are written by the readAll() function and the updateParkings() function is using them while updating the parkingSpace availability status.

The high level reference architecture can be seen in Figure 2

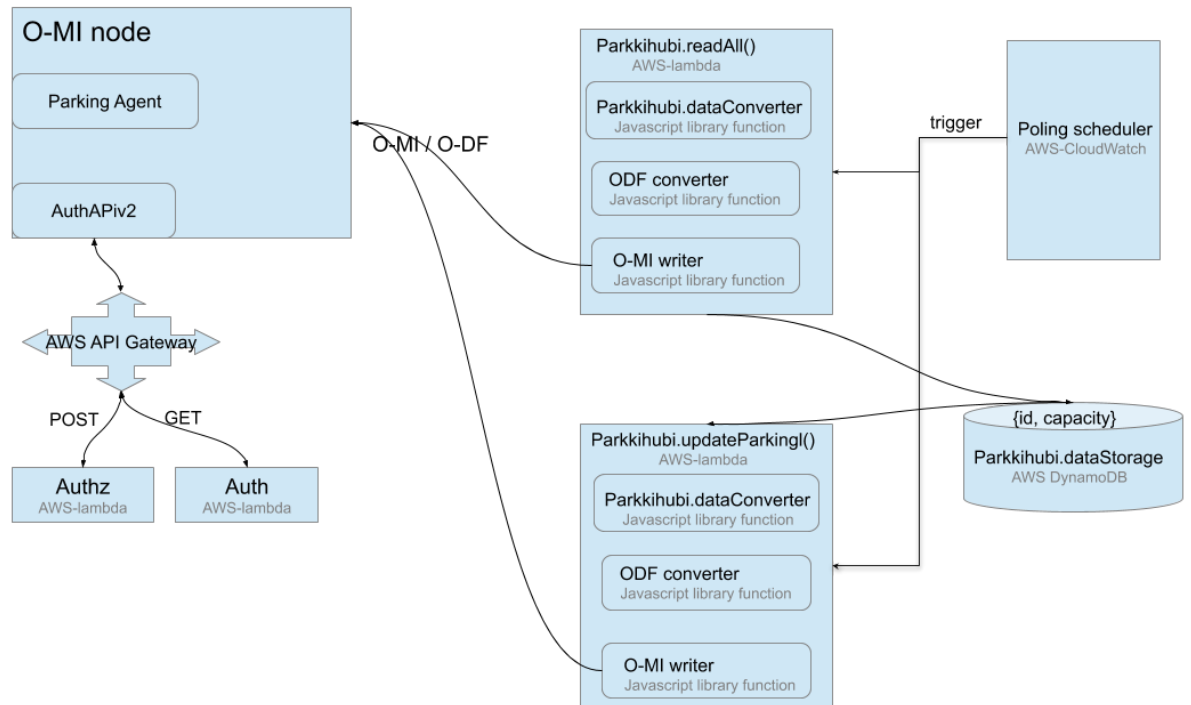


Figure 2. Simplified high level architecture of the Parkki hubi integration

## Park and Ride Implementation

### Description of data source

Park and ride is an open API that provides information about the commuter parking. It is best described as stated in its documentation: *Park and Ride is one of the solutions to tackle Helsinki regions growing population and traffic by encouraging citizens to leave their cars outside of the most congested areas and continuing their trip with public transportation. HSL Park and Ride application contains all the important information about parking facilities and their services near public transportation hubs. The application is used by HSL, parking operators, roadside displays and third party applications. All data is provided free of charge under the CC BY 4.0 license.*

Park and Ride provides rich REST-like interface to query information about parking facilities and their attributes (such as opening times, fees and utilization). Parking facilities can be searched by their id, status, or coordinates. Not all facilities provide the utilization



information. The facilities can have parking spaces for different kind of vehicles (i.e. bicycles, motorcycles, and cars)

There are neither change notifications nor a way to subscribe to data changes, which means that users of the API must be periodically polled to get the current information.

The integration uses two endpoints one to get the parking facility information and another to get the latest utilization of the facilities.

Example of a facility query and response:

```
https://p.hsl.fi/api/v1/facilities.geojson?ids=992
```

```
{
  "hasMore": false,
  "features": [
    {
      "id": 992,
      "geometry": {
        "crs": {
          "type": "name",
          "properties": {
            "name": "EPSG:4326"
          }
        },
        "bbox": [24.73429627113602, 60.15969295146101, 24.74047575860297, 60.16297965935033],
        "type": "Polygon",
        "coordinates": [
          [
            [24.73429627113602, 60.16234354799164],
            [24.737644761881658, 60.16297965935033],
            [24.74047575860297, 60.160192794590245],
            [24.73797961095622, 60.15969295146101],
            [24.736092279808677, 60.16202548769496],
            [24.734692001860505, 60.16169227836832],
            [24.73429627113602, 60.16234354799164]
          ]
        ]
      },
      "properties": {
        "name": {
          "fi": "Matinkylän asema",
          "sv": "Mattby station",
          "en": "Matinkylä station"
        },
        "status": "IN_OPERATION",
        "operatorId": 36,
        "builtCapacity": {
          "CAR": 350
        }
      }
    }
  ]
}
```

```

    },
    "usages": [
      "HSL_TRAVEL_CARD"
    ]
  },
  "type": "Feature"
}
],
"type": "FeatureCollection"
}

```

Example of a utilization query and a response:

<https://p.hsl.fi/api/v1/utilizations>

```

[
  ...,
  {
    facilityId: 992,
    capacityType: "CAR",
    usage: "HSL_TRAVEL_CARD",
    timestamp: "2019-05-07T19:17:40.000+03:00",
    spacesAvailable: 312,
    capacity: 350,
    openNow: true
  },
  ...
]

```

### Software components

The Park and Ride integration to O-MI-node consists of the following Park andRide specific components:

The Park and Ride integration to O-MI-node uses the following generic components (described in detail in the section Solution Architecture Overview):

- Polling scheduler
- O-MI writer
- ODF converter

Additionally, the following Park and Ride specific components have been developed:

- Park and Ride connector

- Park and Ride data converter

#### *The Park and Ride connector*

The Park and Ride connector consists of two AWS (Amazon web services) lambda functions. First (readAll()) for reading the parking facility information (parking facility locations and capacities), and a second (updateParkings()) for reading the real time utilization information. Both functions are periodically triggered by the polling scheduler. First one is triggered only at night time once a week. The second function is triggered every 15 minutes during daytime.

#### *The Park and Ride converter*

The Park and Ride data converter is a library function that converts the Park and Ride data model to the Mobivoc format which is then converted by the ODF converter to ODF XML format and finally sent to O-MI writer. For each Park and Ride parking facility with car parking spaces a corresponding Mobivoc parkingFacility is generated. The coordinates of the parking facility are simply the center coordinates of the parking\_area polygon. The parking\_area capacity estimate is then used to generate mobivoc parkingSpaces under the parkingFacility.

The high level reference architecture can be seen in Figure 3

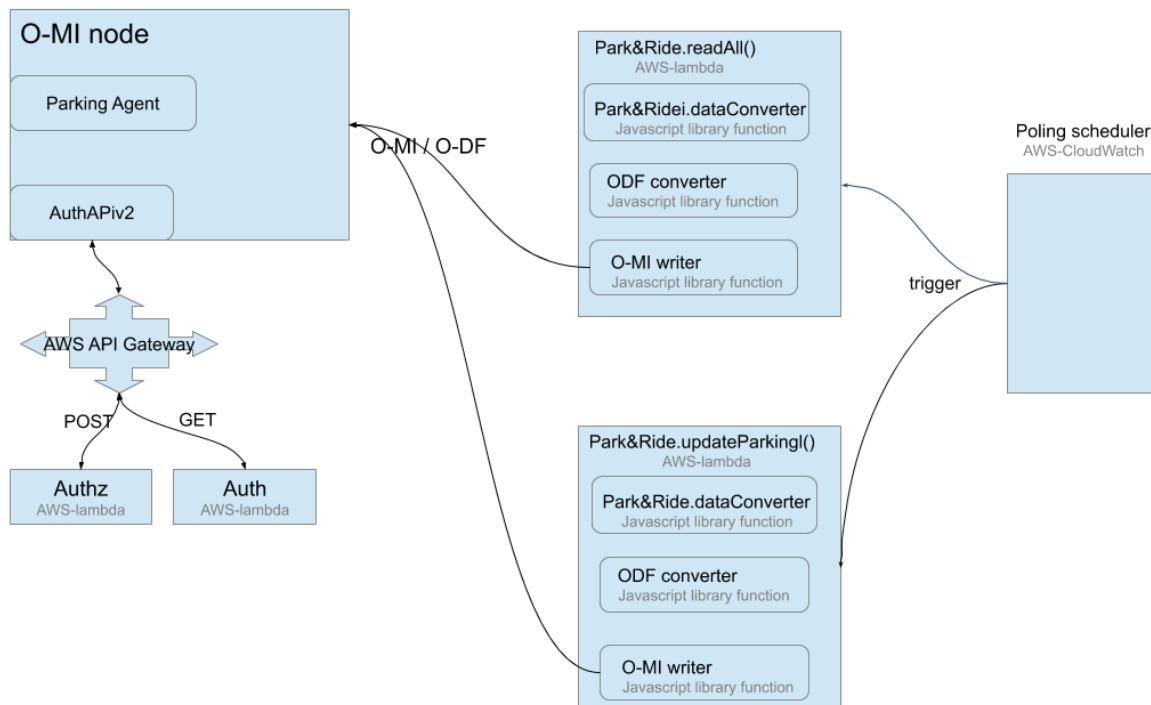


Figure 3. Simplified high level architecture of the Park and Ride integration

## Residential and Private Parking Case

### Description of data source

The purpose of the residential case is to provide a way to private individuals or housing companies to offer their parking spaces for others to use when they themselves do not have need for them.

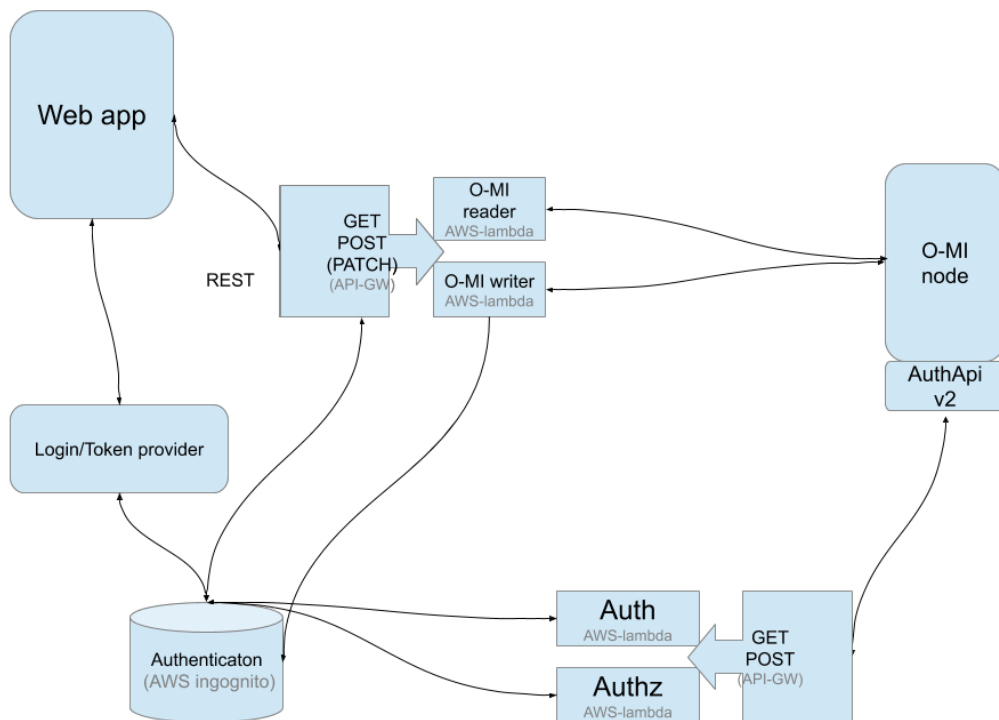
In scope of pilot residential case provides implementation for creation and managing of parking spaces. The future after pilot development implies implementation of support for payment systems, and implementation of the components that check data validity, quality, verify data supplier, and perform other needed control functions.

## **Architecture and Software Components**

The residential case integration to O-MI-node consists of the following Residential case specific components:

- Webapp
- Rest interface
- Residential data storage
- Authentication storage

The high level reference architecture can be seen in Figure 4



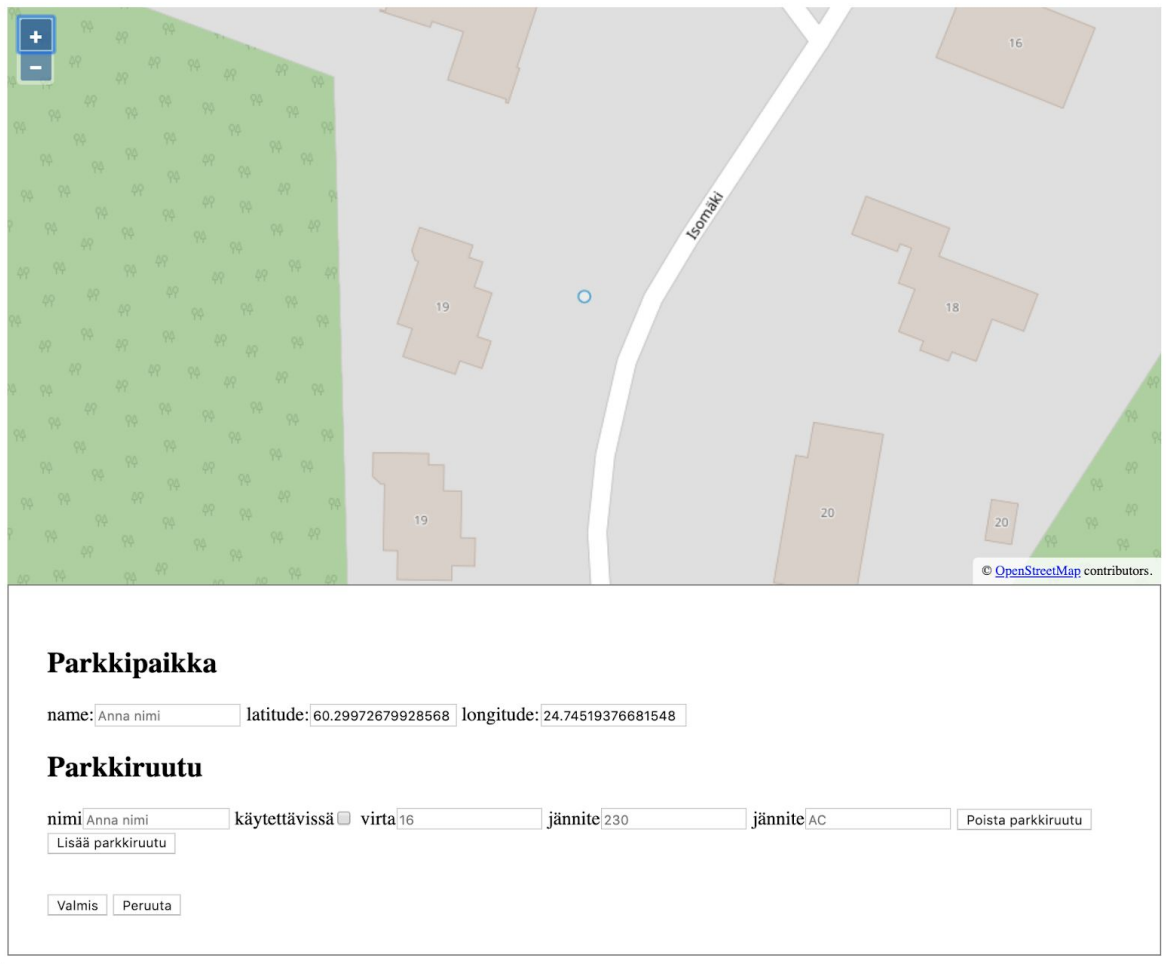
**Figure 4. Simplified high level architecture of the Parkki hubi integration**

### ***Webapp***

The Residential web app is hosted from amazon S3 and served with Amazon CloudFront. The webapp reads the existing parking facilities around the users location and show them on the map. After signin in it allows the user to create a new parking facility and parking spaces for it. The user is able to specify the attributes of the created parking facility

and the parking spaces. Once the user has created some parking facilities and parking spaces he is able to set the availability attribute of those parking spaces either form the webapp or via the REST-interface.

## Lisää uusi parkkipaikka



**Parkkipaikka**

name: Anna nimi latitude: 60.29972679928568 longitude: 24.74519376681548

**Parkkiruutu**

nimi: Anna nimi käytettävissä ☐ virta: 16 jännite: 230 jännite: AC Poista parkkiruutu

Lisää parkkiruutu

Valmis Peruuta

## Rest Interface

The residential rest interface is implemented using Amazon API Gateway and Amazon Lambda functions for creating and updating the parking facilities and parking spaces. The rest interface exposes methods for creating a parking facility and parking space, and modifying the attributes of parking facilities and parking spaces owned by the invoking user.

- GET ParkingFacilities/<facilityId>
- GET ParkingFacilities/<facilityId>/ParkingSpaces
- POST ParkingFacilities/

- POST ParkingFacilities/<facilityId>/
- POST ParkingFacilities/<facilityId>/ParkingSpaces/<spaceId>

New parking facility is created using the POST ParkingFacilities/ method. It expects the parking facility and its parking space as JSON as input. After successful creation it returns the same JSON structure augmented with “facilityId” attribute.

The POST ParkingFacilities/<facilityId>/ allows modification of the attributes of the parking facility.

The POST ParkingFacilities/<facilityId>/ParkingSpaces/<spaceId> allows modification of the attributes of a parking space.

### ***Authentication storage***

The Residential authentication storage is an Amazon Cognito User Pool. Users are authenticated against it and given an access token. This access token is then carried over the API Gateway and AWS lambda functions to the O-MI-node and handled over the auth-APIv2 of the O-mi node to the Authentication and Authorization functions. Authentication and Authorization functions use the token to verify the allowed access and get the list of parking facilities the user has access.

