Innovation Action

H2020-LC-SC3-SCC-1-2018

D2.8 - Adaptation of Oulu ICT platform

WP2; Task 2.7

November 2021 [M36]

Author(s): Jari Palosaari (VTT), Jari Rehu (VTT), Timo Kinnunen (VTT), Jussi Kiljander (VTT)

www.makingcity.eu

@MakingCity_EU

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Grant Agreement n°824418.

Disclaimer

Copyright notice

Document Information

Grant agreement	824418
Project title	Energy efficient pathway for the city transformation: enabling a positive future
Project acronym	MAKING-CITY
Project coordinator	Cecilia Sanz-Montalvillo (cecsan@cartif.es)- Fundación CARTIF
Project duration	1st December 2018 – 30th November 2023 (60 Months)
Related work package	WP2 – Demonstration of positive energy district concept in Oulu
Related task(s)	Task 2.7 – ICT City platform services integration
Lead organisation	20-VTT
Contributing partners	13-OUK, 14-UOU
Due date of deliverable	30th November 2021
Actual submission date	27th December 2021
Dissemination level	Public

History

Date	Version	Submitted by	Reviewed by	Comments
02/10/2020	0.1	VTT	Jussi Kiljander	ToC
30/11/2020	0.9	VTT	Jari Palosaari, Jari Rehu, Timo Kinnunen, Jussi Kiljander	Draft version for internal review
19/11/2021	1.0	VTT	Jari Rehu	Update Ouka school building

21/12/2021	1.1	VTT	Dharmendra Sharma, Jari Rehu	Updated based on comments from review
21/12/2021	2.0	VTT	Jari Rehu	Final version for submission

Table of content

List of figures

List of tables

Abbreviations and Acronyms

Acronym	Description
DSO	Distribution System Operator
FMI	Finnish Meteorological Institute
ICT	Information and Communication Technology
KPI	Key Performance Indicator
PED	Positive Energy District

Executive Summary

This deliverable documents the final version of D2.8 Adaptation of Oulu ICT Platform. It is the outcome of T2.7 and subtask 2.7.1, which focuses on the back-end implementation of the Oulu ICT platform. This deliverable documents the interfaces of the Oulu ICT Platform, including open application programming interfaces and graphical user interfaces.

The technical details of data from the different demonstrative integrations and open APIs implementations are also documented. The uniform structure of URLs and API is described. Moreover, this deliverable describes the server infrastructure designed and developed for the Oulu ICT Platform and its technology readiness level.

1 Introduction

1.1 Purpose and target group

This report constitutes Deliverable *"D2.8* Adaptation of Oulu ICT Platform*"* which is the outcome of the *"Task 2.7 ICT City platform services integration*".

The main objective of this deliverable is to represent open interfaces to services and modules of the Oulu ICT Platform. This deliverable presents also the server infrastructure of the Oulu ICT Platform in more detail.

It should be noted that in addition to this deliverable the description of the Oulu ICT Platform is divided into following final deliverables (and their intermediate versions): D2.9 - Services and Modules for Oulu ICT Platform, D2.6 - Positive District Energy Flows, and D2.5 - Smart Energy Systems in Oulu. The overall architecture of the Oulu ICT Platform is presented in D2.9. D2.6 presents the details on the ICT-based approach for energy management and monitoring at the district level. The D2.5 describes the details on the services and modules for building-level energy management.

The role of this deliverable is to document the open interfaces and server infrastructure of the Oulu ICT Platform. It is targeted for stakeholders interested in developing and deploying similar PED systems. This deliverable thus spans across the deliverables D2.5 and D2.6, but focuses more to the data monitoring side as that is in the core of the open data. That is, the control functionality described in D2.5 and D2.6 will not be part of the open interfaces but instead part of the internal logic and functionality executed by the Platform. The interfaces for energy and flexibility management related services are thus outside of the scope of this deliverable and only described in D2.5 and D2.6.

All the interfaces and implementations described in this deliverable are part of the new Energy Monitoring and Management Platform as such functionality was not provided by the existing Oulu Urban Platform. Please refer to D2.9 (and its intermediate versions) for details on the interplay between the existing and new platforms.

The rest of the deliverable is structured as follows. Section 2 introduces open interfaces, as well as, visualizations about the energy consumption and production at the building and district levels. Section 3 represents the server infrastructure of the Oulu ICT Platform. Section 4 concludes the deliverable.

1.2 Contribution partners

The following Table 1 depicts the main contributions from participant partners in the development of this deliverable.

Table 1: Contribution of partners

Partner nº and short name	Contribution
20-VTT	All sections.

1.3 Relation to other activities in the project

The following Table 2 depicts the main relationship of this deliverable to other activities (or deliverables) developed within the MAKING-CITY Project and that should be considered along with this document for further understanding of its contents.

Table 2: Relation to other activities in the project

Deliverable nº	Relation
D2.6	D2.6 Presents the details on the district-level energy monitoring and management approach.
D2.5	D2.5 presents the details on Building-level energy management approach in Oulu PED pilot.
D2.9	D2.9 presents the overall architecture of the Oulu ICT Platform in high abstraction level focusing on the context and functional views of the platform.
D5.7	D5.7 presents the implementation of the monitoring programme in Oulu
D5.9	D5.9 describes the principles of common open specifications for city ICT platforms
D3.20	D3.20 presents the new services and modules of the Groningen ICT Platform

2 Open interfaces

This section describes the open interfaces and server infrastructure of the Oulu ICT Platform. In addition to this deliverable the description and other details of the Oulu ICT Platform is divided into the final deliverables D2.9, D2.6, and D2.5. This deliverable describes technical details of User database backend, User interface frontend and server infrastructure in detail. The following template was defined to unify the descriptions.

2.1 Application programming interfaces

The open interfaces described in this deliverable build upon open standards including Hyper Text Transfer Protocol (HTTP), Representational State Transfer (REST), JavaScript Object Notation (JSON), Comma Separated Values (CSV) and MQTT. The SensorThings API presented in D5.9 was also considered for the semantic level interoperability, but we decided to utilize the REST API presented in this section for the open interface instead. This design choice was done for following reasons: 1) the format presented in this deliverable is more compact and more streamlined for accessing large amounts of data being collected from the Oulu PED area, 2) SensorThings API is mainly targeted for machine-to-machine communication whereas there was a need for a format that is easily interpretable and usable by non-expert humans, 3) we had existing tools for processing the data in the specified REST API format, which made the development and deployment faster.

Project database contains energy data series for individual buildings (Arina, Sivakka and YIT) and combined energy data series for whole PED area in 15 minutes interval. Any data in the databases can be accessed via a dedicated REST API.

REST	**Re**presentational **S**tate **T**ransfer is an architectural style designed as a request/response model that communicates over HTTP protocol.
API	**A**pplication **P**rogramming **I**nterface is a computing interface that defines interactions between multiple software intermediaries. It defines the kinds of requests that can be made, how to make them, the data formats that should be used, a conventions to follow, etc.

The API URL and API query format is as follows:

https://makingcity.vtt.fi/ped_data/feeds.<format>?<source>&<data_set>&...

When retrieving data from a database, you may specify one of the following formats:

JSON	JavaScript Object Notation. **JSON** is an open standard format that is human-readable, originally used for asynchronous browser-server communication. Widely used format derived from JavaScript.
CSV	Comma separated files. **CSV** is very compact format but requires the data to be a flat table. No hierarchical data supported. Accurate adherence to the structure for every line is very important, as an issue can cause the whole file to become unreadable.

- e.g. https://makingcity.vtt.fi/ped_data/feeds.json

Other parameters:

• source=[arina,sivakka,yit,ped]	- Arina,Sivakka,Yit or combined PED data
• data_set=[energy]	- Data set returning: energy,...
• limit=[1 … n]	- Number of entries to retrieve

days=[days]	- Days from now to include in feed
start=[start date]	– YYYY-MM-DD%20HH:NN:SS
end=[end date]	– YYYY-MM-DD%20HH:NN:SS

2.2 User interfaces

The goal of the developed user interfaces is to see the consumption of every asset available along with bottom-up aggregation at the building and PED level and provide valuable insights about the energy use.

This section can be divided into two parts: **User database backend** and **User interface frontend**. User database backend is running on server and stores all user related data and handles the authentication process. User database backend also redirects measurement requests to **measurement server**, where measurement data is stored. User interface frontend is running on browser displaying all information and responding to user interaction.

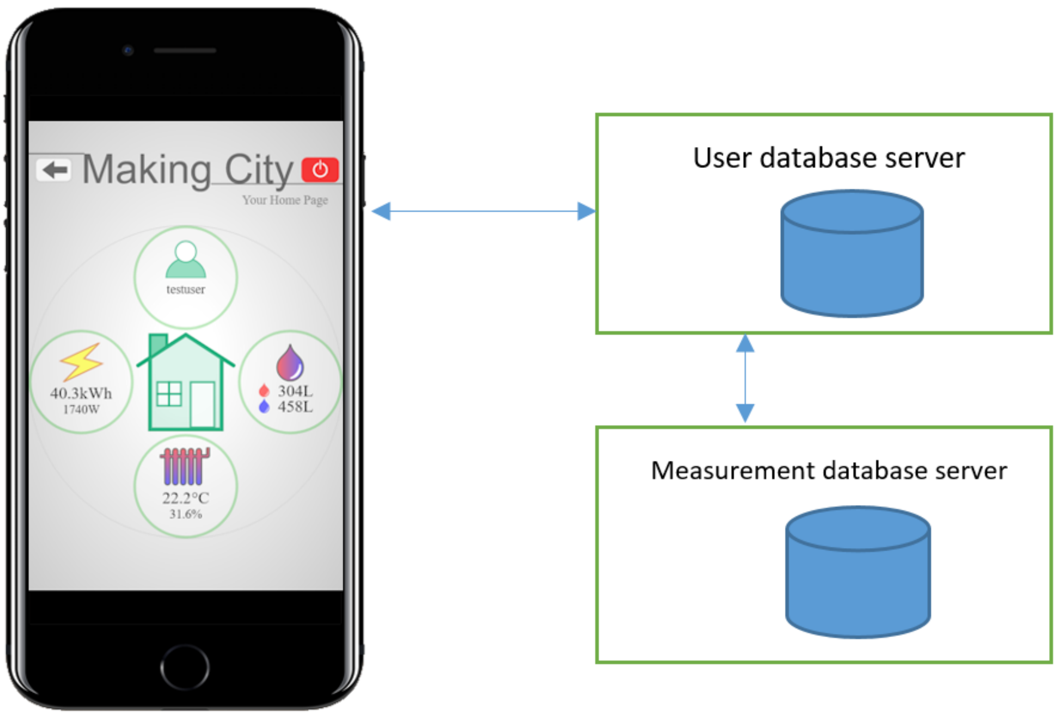


Figure 1: User interface and servers.

User database backend

Backend is written with **Node.js** [1]. Node.js is an open-source, cross-platform, back-end, JavaScript runtime environment that executes JavaScript code outside a web browser. Node.js represents a "JavaScript everywhere" paradigm, unifying web-application development around a single programming language, rather than different languages for server- and client-side scripts.

Backend stores user data into **MongoDB** [2] database. Backend uses **Mongoose**, MongoDB object modeling for Node.js. "Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box." [3]

Almost all backend API calls are verified with **JSON Web Token** (JWT) [4] to protect from unauthorized use. The JWT is created at server and sent to client during successful login. Client stores and uses it in following communication between client and server. At login the token is created. It is valid for 24 hours:

```
const token = jwt.sign(
  {
    email:user[0].email,
    userId: user[0]._id
  },
  process.env.JWT_KEY,
  {
    expiresIn: "24h"
  }
)
```

References

1. https://nodejs.org/en/

2. https://www.mongodb.com/

3. https://mongoosejs.com/

4. https://www.npmjs.com/package/jsonwebtoken

User database backend REST-API

Backend is made of models and routes, which are the same as REST-API endpoints. We have 6 models: **User**, **Regcode**, **Readkey**, **Feedback**, **Log** and **Visitorcount**. API endpoints with details are summarized in following figures.

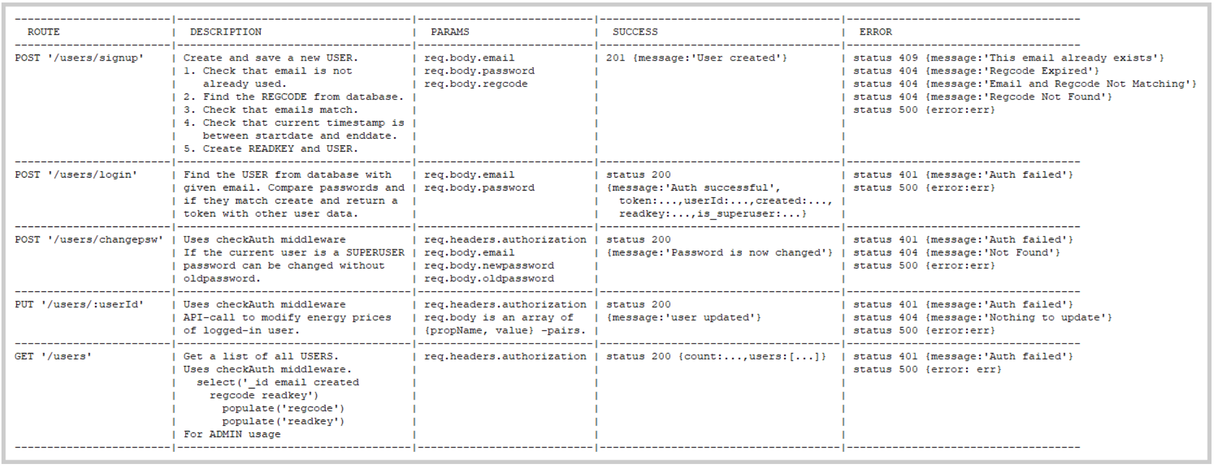
When client sends a request to server, it usually includes the authorization token in header part of the HTTP request, one special **middleware** function ("checkAuth") is always called first to verify and decode the user data encoded into token:

```
----------------------------------|--------------------------|-----------------------|----------------------------------
  DESCRIPTION                     |  PARAMS                  |  SUCCESS              |  ERROR
----------------------------------|--------------------------|-----------------------|----------------------------------
 Verifies token given in request. | req.headers.authorization | req.userData['userId'] | status 401 {message:'Auth failed'}
                                  |                          | req.userData['email']  |
----------------------------------|--------------------------|-----------------------|----------------------------------
```

User model:

_id	ObjectId
email	String
password	String
created	Date
regcode	reference to Regcode
readkey	reference to Readkey
price_energy_monthly	Number
price_energy_basic	Number

price_energy_transfer	Number
is_superuser	Boolean



ROUTE	DESCRIPTION	PARAMS	SUCCESS	ERROR

POST '/users/signup'	Create and save a new USER.
	1. Check that email is not already used.
	2. Find the REGCODE from database.
	3. Check that emails match.
	4. Check that current timestamp is between startdate and enddate.
	5. Create READKEY and USER.
		req.body.email
		req.body.password
		req.body.regcode
			201 {message:'User created'}	status 409 {message:'This email already exists'}
				status 404 {message:'Regcode Expired'}
				status 404 {message:'Email and Regcode Not Matching'}
				status 404 {message:'Regcode Not Found'}
				status 500 {error:err}

POST '/users/login'	Find the USER from database with given email. Compare passwords and if they match create and return a token with other user data.	req.body.email
		req.body.password
			status 200
			{message:'Auth successful',
			token:...,userId:...,created:...,
			readkey:...,is_superuser:...}
				status 401 {message:'Auth failed'}
				status 500 {error:err}

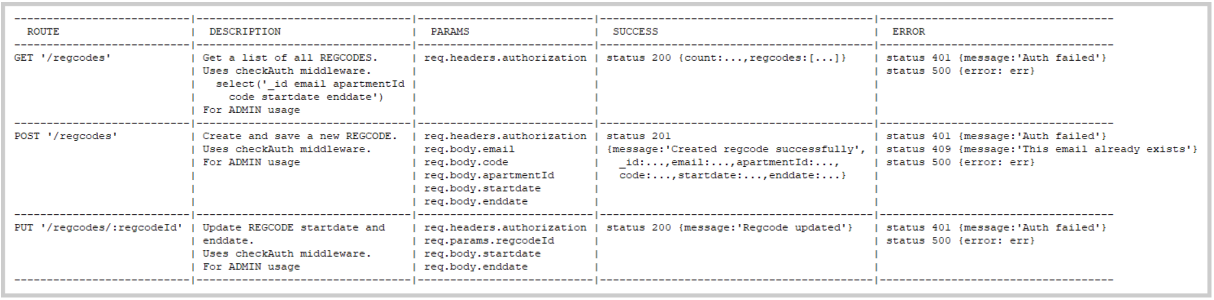
POST '/users/changepsw'	Uses checkAuth middleware
	If the current user is a SUPERUSER password can be changed without oldpassword.
		req.headers.authorization
		req.body.email
		req.body.newpassword
		req.body.oldpassword
			status 200
			{message:'Password is now changed'}
				status 401 {message:'Auth failed'}
				status 404 {message:'Not Found'}
				status 500 {error:err}

PUT '/users/:userId'	Uses checkAuth middleware
	API-call to modify energy prices of logged-in user.
		req.headers.authorization
		req.body is an array of {propName, value} -pairs.
			status 200
			{message:'user updated'}
				status 401 {message:'Auth failed'}
				status 404 {message:'Nothing to update'}
				status 500 {error:err}

GET '/users'	Get a list of all USERS.
	Uses checkAuth middleware.
	select('_id email created regcode readkey')
	populate('regcode')
	populate('readkey')
	For ADMIN usage
		req.headers.authorization	status 200 {count:...,users:[...]}	status 401 {message:'Auth failed'}
				status 500 {error: err}

Regcode model:

_id	ObjectId
email	String
apartmentId	String
code	String
startdate	Date
enddate	Date



ROUTE	DESCRIPTION	PARAMS	SUCCESS	ERROR

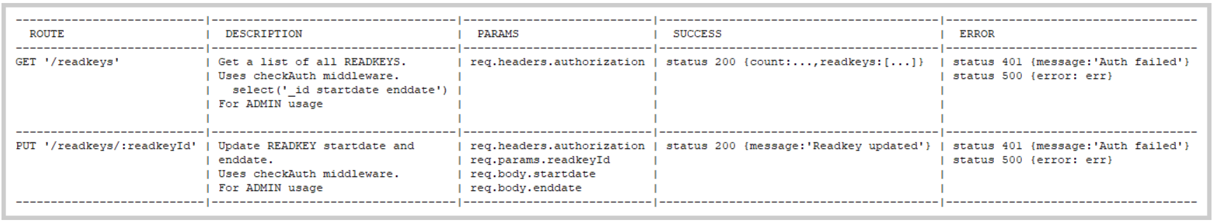
GET '/regcodes'	Get a list of all REGCODES.
	Uses checkAuth middleware.
	select('_id email apartmentId code startdate enddate')
	For ADMIN usage
		req.headers.authorization	status 200 {count:...,regcodes:[...]}	status 401 {message:'Auth failed'}
				status 500 {error: err}

POST '/regcodes'	Create and save a new REGCODE.
	Uses checkAuth middleware.
	For ADMIN usage
		req.headers.authorization
		req.body.email
		req.body.code
		req.body.apartmentId
		req.body.startdate
		req.body.enddate
			status 201
			{message:'Created regcode successfully',
			_id:...,email:...,apartmentId:...,
			code:...,startdate:...,enddate:...}
				status 401 {message:'Auth failed'}
				status 409 {message:'This email already exists'}
				status 500 {error: err}

PUT '/regcodes/:regcodeId'	Update REGCODE startdate and enddate.
	Uses checkAuth middleware.
	For ADMIN usage
		req.headers.authorization
		req.params.regcodeId
		req.body.startdate
		req.body.enddate
			status 200 {message:'Regcode updated'}	status 401 {message:'Auth failed'}
				status 500 {error: err}

Readkey model:

_id	ObjectId
startdate	Date
enddate	Date



ROUTE	DESCRIPTION	PARAMS	SUCCESS	ERROR

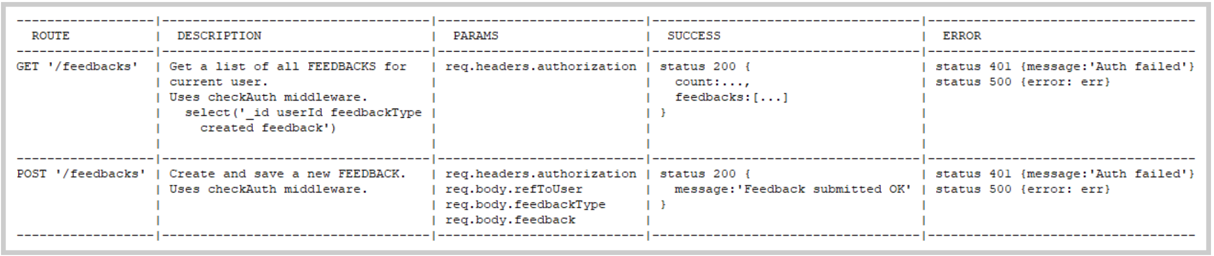
GET '/readkeys'	Get a list of all READKEYS.
	Uses checkAuth middleware.
	select('_id startdate enddate')
	For ADMIN usage
		req.headers.authorization	status 200 {count:...,readkeys:[...]}	status 401 {message:'Auth failed'}
				status 500 {error: err}

PUT '/readkeys/:readkeyId'	Update READKEY startdate and enddate.
	Uses checkAuth middleware.
	For ADMIN usage
		req.headers.authorization
		req.params.readkeyId
		req.body.startdate
		req.body.enddate
			status 200 {message:'Readkey updated'}	status 401 {message:'Auth failed'}
				status 500 {error: err}

NOTE that there is no API-call to create READKEY. This is because READKEY is automatically created when the USER registers (at signup). The READKEY is a MongoDB ObjectId of the Readkey model (_id), 12-byte hexadecimal string, for example: "5f75d08b251f6e38b8a6a736".

Feedback model:

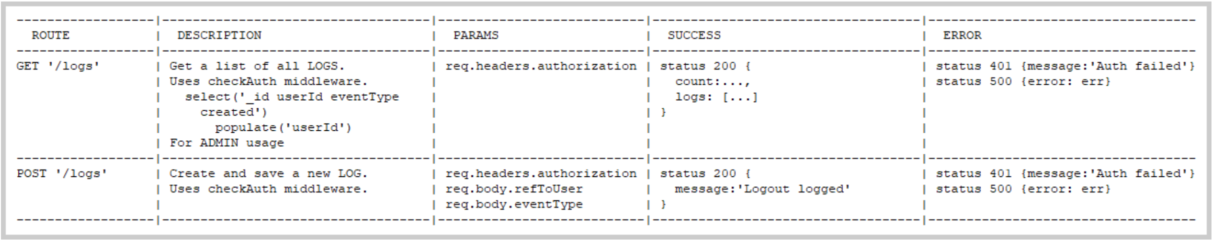
_id	ObjectId
userId	reference to User
feedbackType	String
created	Date
feedback	Number



```
-----------------|---------------------------------|-------------------------|--------------------------------|-------------------------------
  ROUTE          |  DESCRIPTION                    |  PARAMS                 |  SUCCESS                       |  ERROR
-----------------|---------------------------------|-------------------------|--------------------------------|-------------------------------
GET '/feedbacks' | Get a list of all FEEDBACKS for | req.headers.authorization | status 200 {                 | status 401 {message:'Auth failed'}
                 | current user.                   |                         |   count:...,                   | status 500 {error: err}
                 | Uses checkAuth middleware.      |                         |   feedbacks:[...]              |
                 |   select('_id userId feedbackType |                       | }                              |
                 |     created feedback')          |                         |                                |
                 |                                 |                         |                                |
-----------------|---------------------------------|-------------------------|--------------------------------|-------------------------------
POST '/feedbacks' | Create and save a new FEEDBACK. | req.headers.authorization | status 200 {                 | status 401 {message:'Auth failed'}
                 | Uses checkAuth middleware.      | req.body.refToUser      |   message:'Feedback submitted OK' | status 500 {error: err}
                 |                                 | req.body.feedbackType   | }                              |
                 |                                 | req.body.feedback       |                                |
-----------------|---------------------------------|-------------------------|--------------------------------|-------------------------------
```

Log model:

_id	ObjectId
userId	reference to User
eventType	String
created	Date

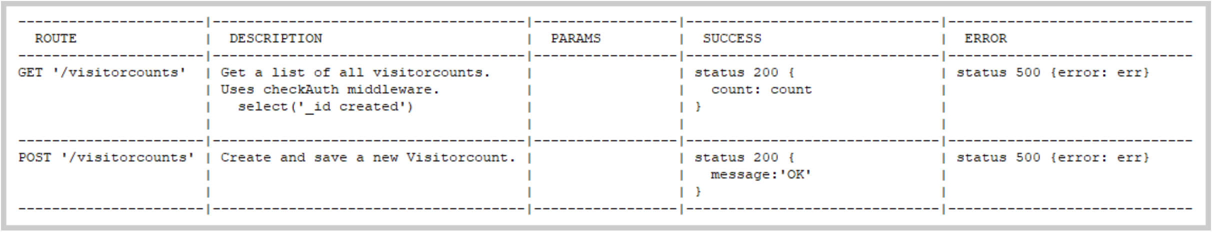


```
-----------------|---------------------------------|-------------------------|--------------------------------|-------------------------------
  ROUTE          |  DESCRIPTION                    |  PARAMS                 |  SUCCESS                       |  ERROR
-----------------|---------------------------------|-------------------------|--------------------------------|-------------------------------
GET '/logs'      | Get a list of all LOGS.         | req.headers.authorization | status 200 {                 | status 401 {message:'Auth failed'}
                 | Uses checkAuth middleware.      |                         |   count:...,                   | status 500 {error: err}
                 |   select('_id userId eventType  |                         |   logs: [...]                  |
                 |     created')                   |                         | }                              |
                 |       populate('userId')        |                         |                                |
                 | For ADMIN usage                 |                         |                                |
-----------------|---------------------------------|-------------------------|--------------------------------|-------------------------------
POST '/logs'     | Create and save a new LOG.      | req.headers.authorization | status 200 {                 | status 401 {message:'Auth failed'}
                 | Uses checkAuth middleware.      | req.body.refToUser      |   message:'Logout logged'      | status 500 {error: err}
                 |                                 | req.body.eventType      | }                              |
-----------------|---------------------------------|-------------------------|--------------------------------|-------------------------------
```

NOTE: 'Login' is now submitted directly from "users/login" route. No need to POST via this API.

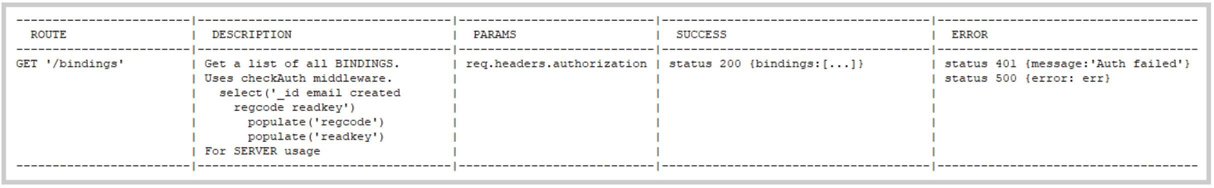
Visitorcount model:

_id	ObjectId
created	Date



```
---------------------|-------------------------------------|----------------|-------------------------------|----------------------------
  ROUTE              |  DESCRIPTION                        |  PARAMS        |  SUCCESS                      |  ERROR
---------------------|-------------------------------------|----------------|-------------------------------|----------------------------
GET '/visitorcounts' | Get a list of all visitorcounts.    |                | status 200 {                  | status 500 {error: err}
                     | Uses checkAuth middleware.          |                |   count: count                |
                     |   select('_id created')             |                | }                             |
                     |                                     |                |                               |
---------------------|-------------------------------------|----------------|-------------------------------|----------------------------
POST '/visitorcounts' | Create and save a new Visitorcount. |               | status 200 {                  | status 500 {error: err}
                     |                                     |                |   message:'OK'                |
                     |                                     |                | }                             |
---------------------|-------------------------------------|----------------|-------------------------------|----------------------------
```

When user requests measurement data, it includes its unique READKEY, which links apartmentId to user. This relation is stored only in the user database, so measurement server must call REST-API function “**bindings**” to have a fresh list of readkey – apartmentId relations.



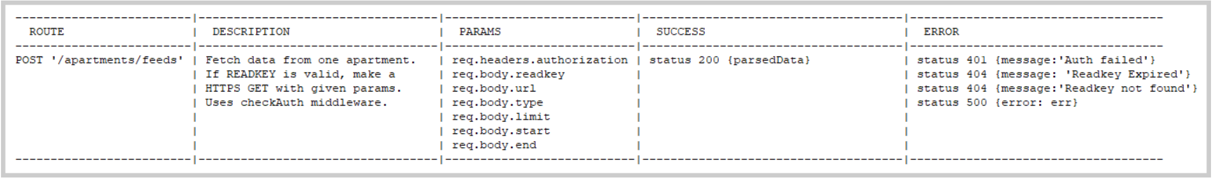
ROUTE	DESCRIPTION	PARAMS	SUCCESS	ERROR
GET '/bindings'	Get a list of all BINDINGS. Uses checkAuth middleware. select('_id email created regcode readkey') populate('regcode') populate('readkey') For SERVER usage	req.headers.authorization	status 200 {bindings:[...]}	status 401 {message:'Auth failed'} status 500 {error: err}

NOTE: Return ONLY those bindings that have a readkey and apartmentId and have a valid readkey (current time is between startdate and enddate).

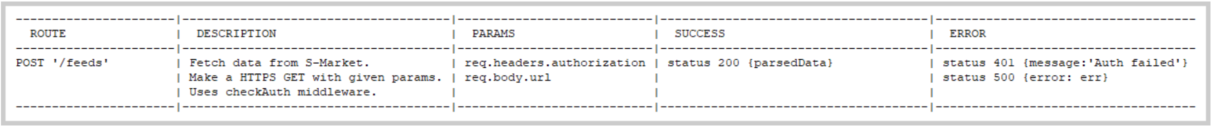
Here is an example of response:

```
{ "bindings" : [{ "apartmentId" : "123", "readkey" : "5f75d08b251f6e38b8a6a736" },
{ "apartmentId" : "333ded", "readkey" : "5f8589ff1e6aae446ca9443a" }] }
```

Following two routes ('/apartments' and '/feeds') just redirect the request adding Authorization into call.



ROUTE	DESCRIPTION	PARAMS	SUCCESS	ERROR
POST '/apartments/feeds'	Fetch data from one apartment. If READKEY is valid, make a HTTPS GET with given params. Uses checkAuth middleware.	req.headers.authorization req.body.readkey req.body.url req.body.type req.body.limit req.body.start req.body.end	status 200 {parsedData}	status 401 {message:'Auth failed'} status 404 {message: 'Readkey Expired'} status 404 {message:'Readkey not found'} status 500 {error: err}



ROUTE	DESCRIPTION	PARAMS	SUCCESS	ERROR
POST '/feeds'	Fetch data from S-Market. Make a HTTPS GET with given params. Uses checkAuth middleware.	req.headers.authorization req.body.url	status 200 {parsedData}	status 401 {message:'Auth failed'} status 500 {error: err}

User interface frontend

Frontend is a **web application** accessed by the user through a web browser with an active internet connection. It is written with **JavaScript**, **HTML**, **SVG** and **CSS**. It utilizes modular programming techniques facilitated by ECMAScript 6 (**ES6**) and **MVC** and **observer** patterns. It is made with **Responsive Web Design** (RWD) approach, so it naturally supports different devices: **smartphones**, **tablets**, **laptops** and **desktops**.

Frontend is also a **single-page application** (**SPA**) [1]. A single-page application (SPA) is a web application or website that interacts with the user by dynamically rewriting the current web page with new data from the web server, instead of the default method of the browser loading entire new pages. The goal is faster transitions that make the website feel more like a native app. In a SPA, all necessary HTML, JavaScript, and CSS code is either retrieved by the browser with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does it transfer control to another page, although the location hash or the HTML5 History API can be used to provide the perception and navigability of separate logical pages in the application.

Frontend is written with ECMAScript (AKA **JavaScript**) [2]. ECMAScript is a general-purpose programming language, standardized by Ecma International according to the document ECMA-262. It is a JavaScript standard meant to ensure the interoperability of Web pages across different Web browsers. ECMAScript is commonly used for client-side scripting on the World Wide Web, and it is increasingly being used for writing server applications and services using Node.js.

The 6th edition, initially known as **ECMAScript 6 (ES6)** then and later renamed to ECMAScript 2015, adds significant new syntax for writing complex applications, including class declarations, ES6 modules like import and export. ES6 is therefore well suited to implement modular software design patterns like Model–view–controller (usually known as MVC) [3].

MVC is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. Traditionally used for desktop graphical user interfaces (GUIs), this pattern has become popular for designing web applications.

Another pattern used in MAKING-CITY frontend is the **observer pattern** [4]. It is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

In our case Models are Subjects, which call notify()-method on Views (or Controllers) which are Observers. Model calls notify()-method when new data is fetched or updated, then those views and controllers which are registered as observers to that model will execute their notify()-method.

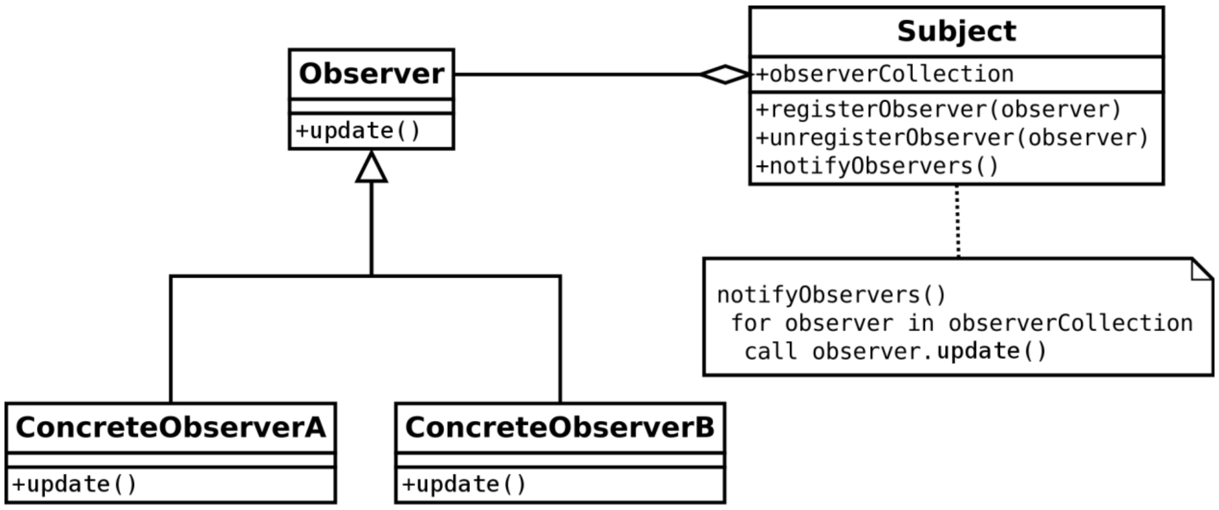


Figure 2: Observer Design Pattern [5].

Scalable Vector Graphics (SVG) [6] is an Extensible Markup Language (XML)-based vector image format for two-dimensional graphics with support for interactivity and animation. The SVG specification is an open standard developed by the World Wide Web Consortium (W3C) since 1999.

Responsive web design (RWD) is an approach to web design that makes web pages render well on a variety of devices and window or screen sizes.[7][8]

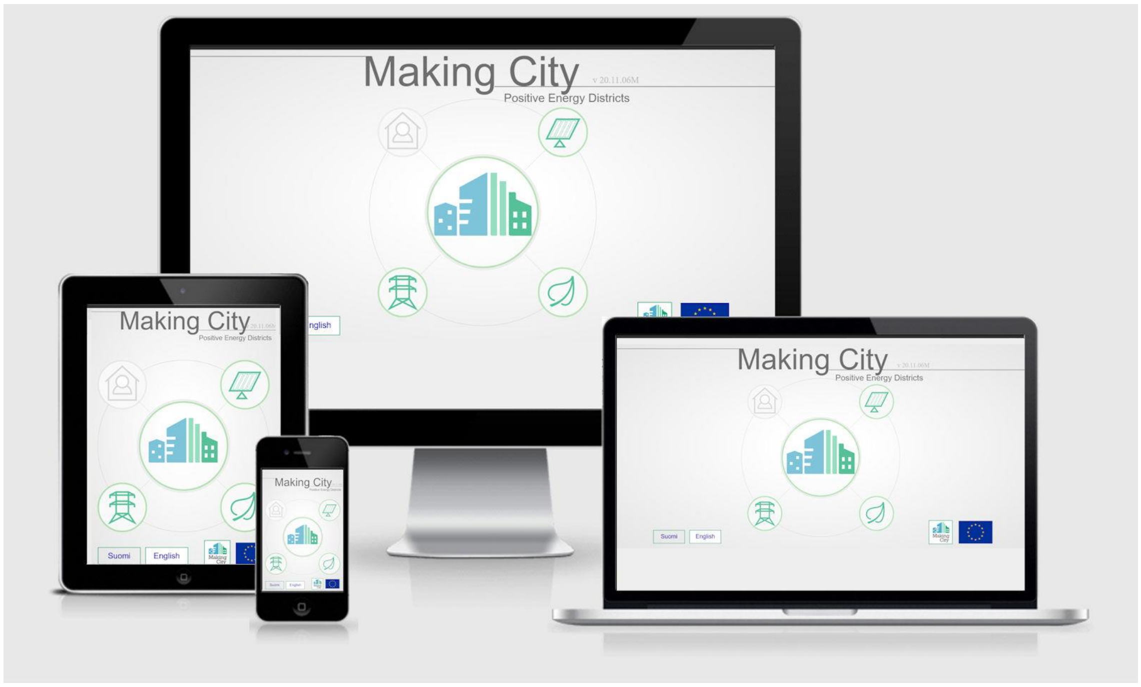


Figure 3: Responsive Web Design approach.

In frontend Master Controller keeps track of ALL controllers and Models in the system. It has two instance properties to do that:

```
this.controllers = {};
this.modelRepo = new ModelRepo();
```

The ModelRepo is just a collection of (key, value) pairs, such that each key can appear only once in the collection. Master Controller creates all controllers and adds them to its controllers hash. Each controller (a sub module) creates its models and adds them to “global” model repository. Each controller creates also a view or a wrapper view, if there are for example many charts in one view. Each view subscribes as a listener of notifications from all relevant models (see Figure 4 Class diagram).

When a view is shown (=rendered), it first creates a skeleton of HTML markup, where empty data-placeholders are inserted into parent element. If all models are ready, then also the data is available and it is appended into placeholders.

But if all models are not ready, we show a Wait-spinner and wait for Model to trigger "fetched" notification. The models (one or many) also notify View whenever new data is fetched, so the chart update can happen "out-of-sync", one model at a time.

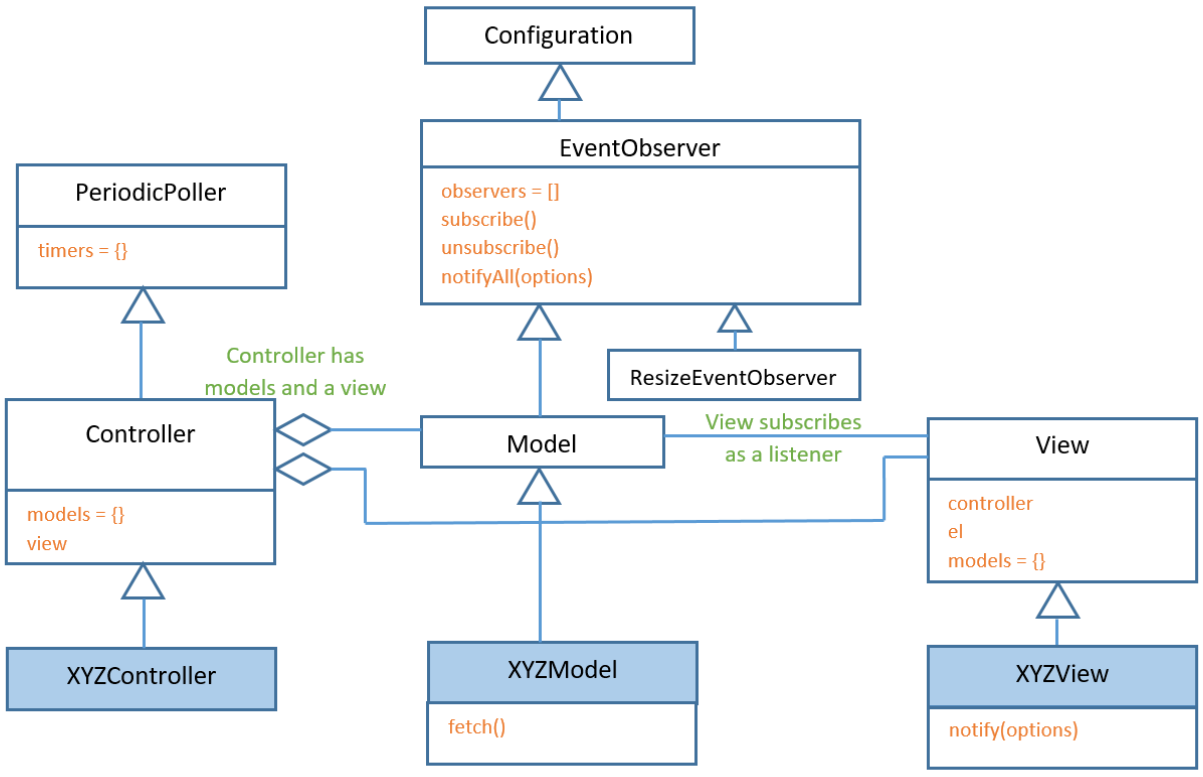


Figure 4: Class diagram.

The user interface contains following views and sub views (see Figure 5 and Figure 6). Boxes with white color are already implemented and boxes with orange color are under construction. The user interface can be divided into **private** and **public** parts. Private part contains resident's data, and it is for authenticated users only. Public part contains information about solar energy, environmental load of energy used and aggregated load from grid. Also PED area buildings and PED-level metrics are visualized to detail suitable for unauthenticated users (= general public).

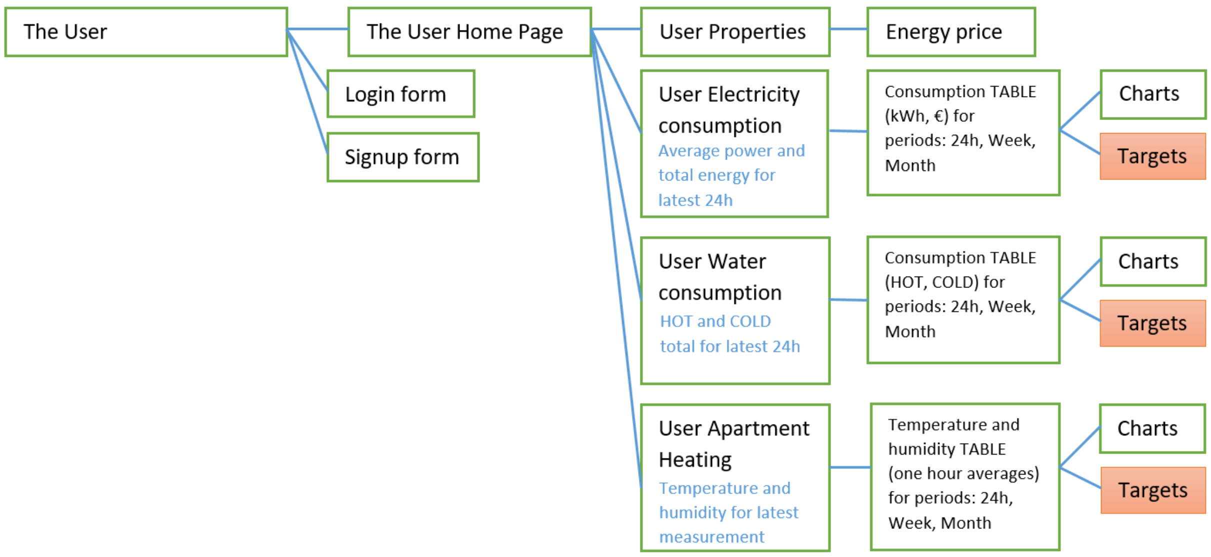

The User
The User Home Page
Login form
Signup form
User Properties
Energy price
User Electricity consumption
Average power and total energy for latest 24h
Consumption TABLE (kWh, €) for periods: 24h, Week, Month
Charts
Targets
User Water consumption
HOT and COLD total for latest 24h
Consumption TABLE (HOT, COLD) for periods: 24h, Week, Month
Charts
Targets
User Apartment Heating
Temperature and humidity for latest measurement
Temperature and humidity TABLE (one hour averages) for periods: 24h, Week, Month
Charts
Targets

Figure 5: The user interface for residents.

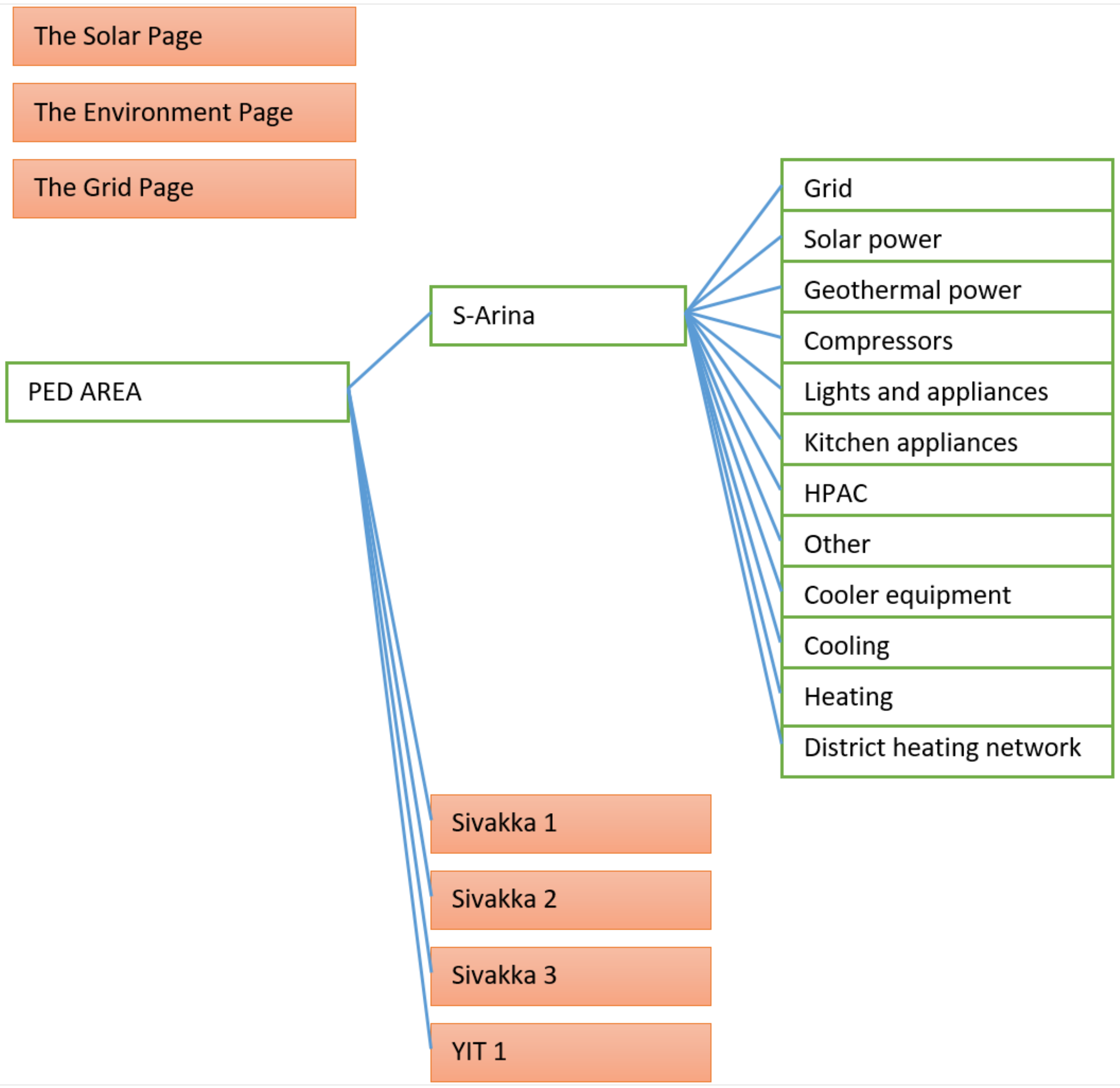


Figure 6: The user interface for general public.

References

1. https://en.wikipedia.org/wiki/Single-page_application

2. https://en.wikipedia.org/wiki/ECMAScript

3. https://en.wikipedia.org/wiki/Model-view-controller

4. https://en.wikipedia.org/wiki/Observer_pattern

5. Observer Design Pattern SVG By Gregorybleiker - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=65617132

6. https://en.wikipedia.org/wiki/Scalable_Vector_Graphics

7. https://en.wikipedia.org/wiki/Responsive_web_design

8. https://alistapart.com/article/responsive-web-design/

3 Server infrastructure

Oulu ICT Platform consist of two physical server that have been virtualized. Virtualization means dividing a physical server into one or more virtual servers with the help of virtualization software as shown in Figure 7. Each virtual machine acts as a server that have its own operating system, own services and databases.

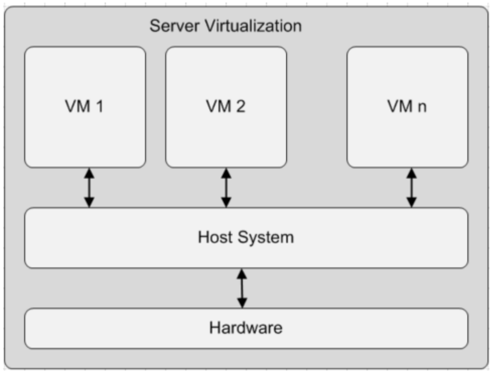


Figure 7: Server virtualization

Oulu ICT Platform servers are powerful Dell 1 unit rack servers that are located in a same cabinet. Both server have a connection to the internet and they are also connected together through a local network connection as shown in Figure 8.

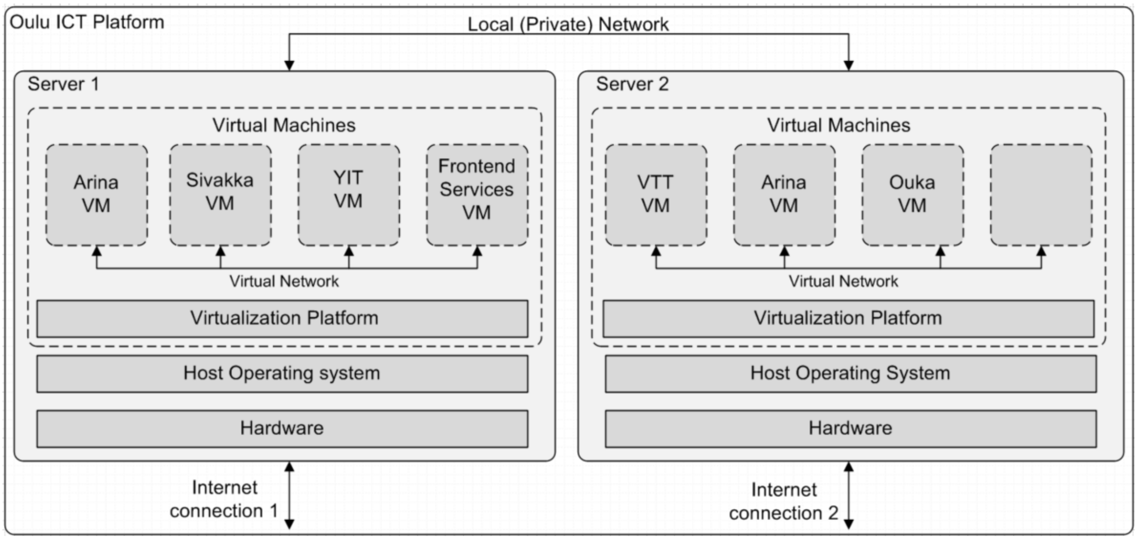


Figure 8: Oulu ICT Platform

Any of the (virtual) servers can also communicate with other servers thru a virtual network and local network if needed.

Host operating system on both machines is Red Hat Enterprise Linux (RHEL) based CentOS 7 Linux. CentOS Linux distribution is a stable, predictable, manageable and reproducible platform that is available in a free of charge.

On the host operating system is a KVM virtualization layer that is an open-source virtualization technology build on Linux. KVM allows a host machine to run multiple, isolated virtual environments called quests or virtual machines (WMs).

Guest operating system in the virtual machines is a Debian Linux based Ubuntu LTS Linux. LTS is an abbreviation of "Long Term Support" and it is a server release of Ubuntu Linux. Ubuntu LTS Linux is an excellent platform for site-data services because it is reliable and provides excellent support for latest tools and services.

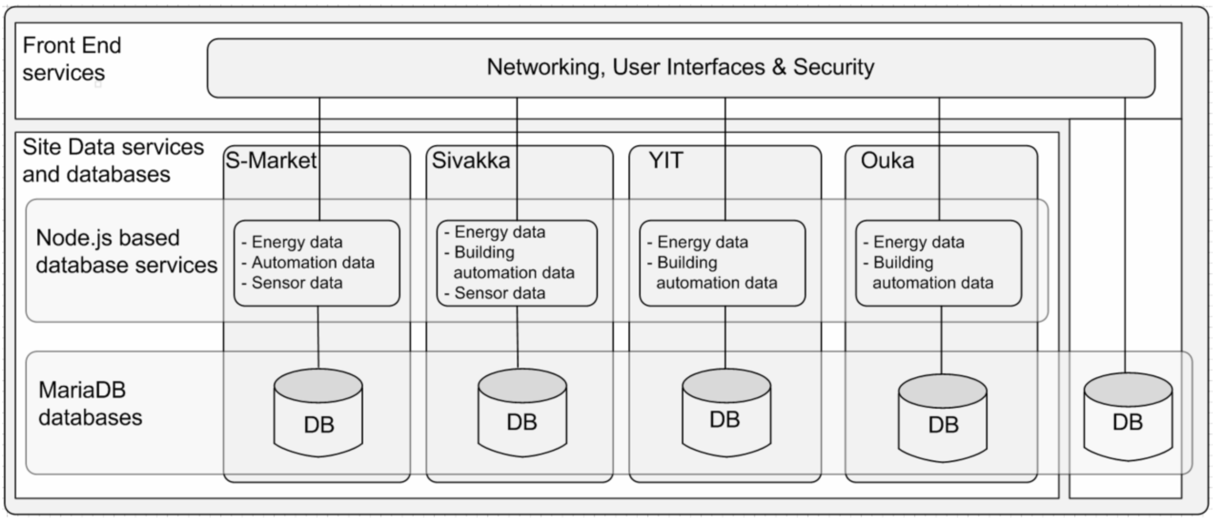

Front End services
Networking, User Interfaces & Security
Site Data services and databases
S-Market
Sivakka
YIT
Ouka
Node.js based database services
- Energy data
- Automation data
- Sensor data
- Energy data
- Building automation data
- Sensor data
- Energy data
- Building automation data
- Energy data
- Building automation data
MariaDB databases
DB
DB
DB
DB
DB

Figure 9: Virtual Machines & Services

Front end server runs also a service named Nginx. Nginx is a free, open source, high performance load balancer, http web server and reverse proxy server for HTTP, TCP and UGP traffic. Nginx is known for its high performance, stability, rich feature set, simple configuration and low resource consumption.

Nginx provides an SSL termination with a certificate for a secure https connection, and a reverse proxy for routing http requests to the appropriate Site data service.

Site-data services are written with Node.js. Node.js is an open-source, cross-platform, back-end that allows the creation of Web servers and networking tools using JavaScript and a collection of "modules" that handle various core functionalities. Modules are provided for file system I/O, networking (DNS, HTTP, TCP, TLS/SSL, or UDP), binary data (buffers), cryptography functions, data streams, and other core functions.

There are many benefits to using Node.js. It is fast, but also lightweight for the server when sending multiple requests. It is well supported and easy to learn. And it works perfectly with databases.

Database engine for Site-data is MariaDB relational database. MariaDB Server is one of the most popular open-source relational database. It's a community-developed, commercially supported fork of the MySQL relational database management system (RDBMS), intended to remain free and open-source software under the GNU General Public License. MariaDB is reliable, high performance and full-featured database server.

Each site data server contains multiple data services and databases for a variety of parameters and measurements that come from multiple sources.

Arina Site-data Services module contains services and database tables for Supermarket automation system data, Supermarket energy data and Supermarket sensor data.

Sivakka Site-data Services module contains services and database tables for building automation data, building energy data, apartments energy data and apartments sensor data.

YIT Site-data Services module contains services and database tables for building automation data and building energy data.

Ouka Site-data Services module contains services and database tables for school building automation data and building energy data.

Conclusions

This deliverable documented the Oulu ICT Platform that extends the existing Oulu Urban Platform with open interfaces as well as visualizations about the energy consumption and production at the building and district levels in PED area. The server infrastructure of the Oulu ICT Platform is documented at the technical level. Implementation view, information view and deployment view that complement the architecture documentation were documented in this deliverable.

The developed software solutions (backend and frontend) can be replicated outside of MAKING-CITY project as well with the help of the companies. The impact is that the user interfaces allow the users to have the exact details of what is happening with controlled consumption for each device and at overall level too. The APIs are uniform and scalable. The current user interface is available at [2] and the overall estimated TRL level of this solution (user interfaces and server infrastructure) is around TRL6-7.

Bibliography

[1] International Organization Of Standardization, "ISO/IEC/IEEE 42010:2011 - Systems and software engineering -- Architecture description," *ISOIECIEEE 420102011E Revis. ISOIEC 420102007 IEEE Std 14712000*, vol. 2011, no. March, pp. 1–46, 2011.

[2] https://makingcity.vtt.fi/authtest/auth/index.html