DataSites: a Simple Solution for Providing Building Data to Client Applications

M. Nurminen, M. Saari, and P. Rantanen
Tampere University/Faculty of Information Technology and Communication Sciences, Pori, Finland
Mikko.nurminen@tuni.fi

Abstract - **Nowadays, energy consumption and especially energy saving are important issues. The concern over global warming has increased the need to save energy in many areas of our living community. One way of slowing global warming is to reduce energy consumption and the related emissions. IoT systems can be utilized to monitor living conditions in residential and commercial buildings. These IoT systems continuously produce huge amounts of data. Similarly, various vendor-specific building automation systems generate data on an equal scale. Combining these data sources can be challenging, as protocols and data are seldom in a unified and interoperable format. In this paper, we present the DataSites service, which integrates back-end systems and client-oriented REST APIs. The high-level view of DataSites architecture, how to relay building data to client applications, and the required interfaces are described in this paper. Use of the service is illustrated using two real-life use cases.**

Keywords - Building data, IoT, API, Integrating services

I. INTRODUCTION

Research shows that in many countries the cooling and heating of buildings can consume significant amounts of energy. For example, in the colder climate of Finland, the heating of buildings uses over a quarter of the total energy produced [1]. Furthermore, studies show that a big potential for achieving energy savings exists in older, existing buildings, mainly because of the large building stock and lower energy efficiency of buildings constructed in past decades (1950-1990) [2].

This paper presents the preliminary results of the KIEMI ("Vähemmällä Enemmän – Kohti Kiinteistöjen Energiaminimiä", or "Less is More: Towards Energy Minimum Properties" in English) project, which aims to develop proof-of-concept demonstrations and prototype applications that illustrate how cost-effective, open, and modular solutions could be utilized to improve the energy efficiency of existing, older buildings.

In recent years, there has been a huge increase in studies that focus on Internet of Things (IoT) devices, smart buildings, and wireless sensor nodes. Our previous research generated a more in-depth review of applications used for apartment energy monitoring and how rapid prototyping with off-the-shelf devices and open source software could be utilized to collect environmental data [3].

A Wireless Sensor network (WSN) is a common implementation of IoT for gathering data. A typical concept of sensor networks was presented by Akyildiz et al. [4]. The architecture of a data gathering IoT system can be presented by a model of five layers [5]. This study focuses on the back-end layers: an objects layer with sensors, an object abstraction layer which transfers the data, and a service management layer for the storing and serving of collected data.

We presented various ways to store and visualize sensor data in our previous study [6]. This study continues in the same data gathering and serving focus area by developing a more sophisticated implementation. The construction of the developed system consists of back-end and front-end. We divide the research into two research papers: This paper introduces the back-end implementation. The developed system front-end with user interfaces (UI) is introduced in another study [7].

The purpose of this paper is to introduce a simple solution for providing building data to client applications. Further, the goal is to utilize the collected data for optimizing energy usage in existing building stock while preserving good living and working conditions. The study continues our previous research [8] on using cost-effective, off-the-shelf components and devices for measuring indoor air quality.

The structure of the paper is as follows. Section II presents the architecture of the developed system. Section III presents two real-life use cases for providing building data by using the developed system. Finally, Section IV summarizes this paper.

II. THE ARCHITECTURE

Fig. 1 illustrates the high-level architecture of the system. The system can be considered to consist of five major components (or collections of components): building systems (or back-ends, in the context of this paper); adapters; software components that are used to compose the actual service; service APIs (Application Programming Interfaces) of the Sites and Data service; and client applications. The last component - client applications - can be mobile or desktop software and is primarily used to visualize data collected from buildings or produce added value through deeper analysis of the data. The applications are client only from the perspective of the service presented in this paper and can themselves be other services or complex systems. The first four components are explained in more detail in the following subsections A, B, C, and D, respectively. The back-end use cases described in Section

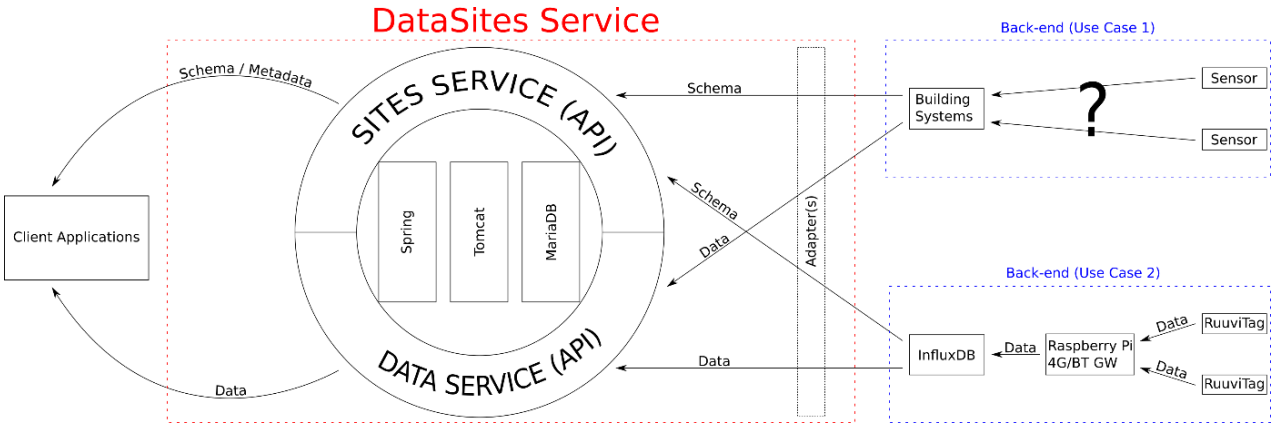

Figure 1 Visualization of the high-level architecture.

III are also shown in Fig. 1. The arrows in Fig. 1 illustrate the flow of data (or information). In practice, the actual data is polled from the back-end systems, as the back-ends can seldom be configured to send or relay data to external services automatically. Similarly, the client applications retrieve data from the service using REST (REpresentational State Transfer) APIs.

In our use cases, as well as in the context of this paper, the utilization of data is the number one priority, and controlling the actual building systems is secondary. In principle, similar architecture could also be used to control building automation systems, but this would increase the overall complexity. This would also require deeper understanding of the inner workings of the building systems, and tighten the security requirements for the service.

A. Back-ends

The back-ends represent the information, automation, or sensoring systems installed in a building, which collect or generate data within a building or a larger building complex. A back-end could also be a cloud-based service, which provides the required interfaces for retrieving data. During the KIEMI project, several buildings (or larger building complexes) have been utilized for running pilots and prototype trials. Even in Finland, there are several companies offering building automation systems with varying inter-compatibility. In other countries, many other systems are sure to be found and often the systems will not be directly compatible. In other words, implementing a service that can access a specific building automation system is not guaranteed to work with a system produced by another company. On a higher level, in our experience, the systems can be categorized into four different classes based on the interfaces available:

- **Cloud-based interfaces**. In these systems the data collected by, for example, building automation, can be accessed over the public Internet. The cloud service provides a machine readable interface (e.g., REST API) for reading data and controlling building automation systems. Integration-wise, cloud-based interfaces provide the easiest path and do not require the modifying (or even direct access) of the systems installed inside the buildings.
- **On-location interfaces**. It is quite common for older and/or smaller buildings to have limited or non-existing interfaces for remote access. On-location interfaces are often vendor-specific and require extra work for software and hardware integration. The actual interface might be, for example, an Ethernet or serial connection located somewhere within the premises. In some cases, open standards such as Modbus or BACnet could be available, making the integration work slightly easier, but usually at least some amount of location-specific customization of the utilized hardware/software is required. Additional challenges might be created by the physical location of the interface. Often the interface was not originally designed for remote access, and thus, it might be located underground or behind thick walls, hindering or preventing radio communications and/or requiring extensive re-wiring.
- **Web interfaces**. The buildings that provided only web interfaces also included on-location interfaces, but accessing the interfaces was impossible or impractical. There could be multiple reasons for this, for example: organizational issues - the building automation could be managed by a third party company making ad-hoc installations problematic; economic issues - the physical interface could be configured or located in a such way that remote access implementation would be expensive; or management issues - for whatever reason, the owner or manager of the building does not want to allow physical access to the existing on-location systems. Implementation-wise, web interfaces are problematic at best. It is, in general, possible to scrape the required data from the web page or reverse-engineer the underlying interfaces by inspecting the web page (when the page's terms of use allow this). However, there is no guarantee that an implementation built upon this kind of approach will work in the future - usable for testing and prototyping, but not for the production environment.
- **Legacy interfaces**. Cases where manufacturers, documentation, specifications, and personnel with appropriate knowledge have all gone out of

business a long time ago. In the best case scenario, the systems are simple enough to reverse-engineer, but in the worst case it would be more cost-effective and future-proof to simply replace the existing systems with more modern ones if remote access is required.

B. Adapters

Adapters are software components, located within the DataSites service, responsible for communicating with the back-ends and converting data into a format understandable by the service. Usually, one adapter cannot be used for another back-end because of data format and protocol incompatibilities.

Two types of data are integral to back-end systems. The first type includes the layout of the building: blueprints, floorplans, sensor locations, etc. - called the "schema" of the building in the context of this paper. The second type is the actual data, which is, in general, time series data representing the sensor measurements collected within the building. The data could be anything from temperature readings to the air pressure statistics of an HVAC unit. Internally, the adapters use the Sites and Data APIs described in subsection D below. Schemas are fully converted by the adapters to the format understandable by the Sites APIs, but data is only partially converted.

Data is converted to simple key-value pair JSON object arrays regardless of the original format. Only a simple common key unification is performed. In other words, if back-ends A and B both provide temperature data, but with data keys "t" and "temp", respectively, the keys are renamed "temperature" in the final product. If back-end C provides a "hvac_air_temperature_t" key, the original key is preserved, as no common key mapping exists. Similarly, the data values are converted to SI units when possible. This approach has both advantages and disadvantages. The advantages are simpler implementation and preservation of unknown raw data in the final output. The disadvantage is the possibility of key collisions: multiple back-ends might use the same unknown key, but use it to describe two different measurements, or simply use different units of measurements. If only simple data is required and there is no need to see the "raw data" on the client side, the unknown keys could simply be ignored. The alternative would be to create a unified format, which contains all possible data types produced by all supported building systems. It could also be assumed that a user requesting the rarer and more unique data types would already know what to look for and have some kind of understanding of the building in question.

In some cases, other information such as alerts or value set-points/thresholds could be available. For these, "virtual sensors" are created, and the data is converted to timestamped data points. I.e., natively (unsupported) information can be presented in the model as “sensors” even though the data sources may not be actual physical sensors located in the building. More complex, which cannot be easily represented as simple textual notifications or key-value paired data, is currently ignored. In practice, complex data is often generated and managed by higher-level services built on top of building system interfaces and are thus out of reach for our implementation. Furthermore, the basic purpose of our system is to provide simple data APIs for client applications, making processing complex data out-of-scope.

In Fig. 2, for the sake of simplicity, both the actual "data" and "schema" data are visualized as originating from one single back-end. In practice, it is quite common that the layout of the building must be retrieved from another system or is not directly available in a structured format, and must be constructed based on blueprints, and inserted into the service using the service APIs. If a service providing the building schema is available, the service could be periodically polled for changes, but building layouts and sensor locations usually change only in the case of renovations or when replacing broken or malfunctioning sensors or devices.

C. Service Components

The service was created with the Java programming language and built upon the Spring framework, Apache Tomcat, and MariaDB (MySQL). These can be easily built into a Spring boot application using, for example, Maven. In our case, the service runs in a Linux-based virtual machine (Debian Bullseye), and all of the components are readily available in most Linux distributions. The API specifications are written in Yaml and are based on the OpenAPI specification [9]. There exist several code generators for creating both client and service stubs for practically every commonly used programming language. In our case, we used the swagger codegen [10]. In summary, the service was built on commonly used components, and a similar approach has worked well in our previous projects.

In the current version there is no functionality for "hot loading" new adapters into the systems. In other words, a full system re-compilation and deployment is required when new adapters are added. Allowing new adapters to be added while the service is running would allow better customization and easier extendibility, but on the other hand it would increase the system complexity, and in practice, implementing such a feature is often not trivial. In our case the brief service downtimes caused by adding new adapters are not crucial, as new adapters are not created very often. Still, adapter hot loading is a feature that could be added in the future.

The MariaDB database is used to store both client-side and back-end side credentials as well as building schemas. No actual (measurement) data is stored in the database. If data caching is required, better databases exist for handling time series data (such as InfluxDB, which we used in ref. [6]). In practice, the client requests often need real-time data that has to be directly retrieved from back-ends, which in turn reduces the advantages of active data caching. HTTP (Hypertext Transfer Protocol) request caching can be performed on the client-side API to lower the back-end load.

The Spring Security framework is used for user management and authorization. In our testing scenarios, in client-side APIs HTTP basic authentication with or without encryption is used for easier testing, but for production use the Spring framework offers methods that have better

security. The client-side user credentials never allow direct access to the back-end systems. This is both for convenience (the user does not need to know back-end credentials) and for security (if client-side credentials are breached, a malicious user might gain access to the data, but not to the actual back-end credentials).

The back-end side authentication depends on the methods available in the back-end systems in question. In some cases, a VPN or other form of trusted zone can be created, or tokens could be generated for the purpose of accessing the back-ends. In other cases, the only option may well be to store credentials (username-password pairs) directly in the service's database, creating increased requirements for securing the system. Further, how "fine-tuned" credentials can be created varies between systems. That is, it may not be possible to create credentials with access limited only to certain data, or the same credentials - if they have fallen into the wrong hands - could also be used to modify the building systems' parameters even when the service itself does not provide such functionality. In principle, one should never store clear-text credentials in databases, but unfortunately this is not always possible when dealing with legacy systems.

In addition, older systems may not have been designed with user management in mind in the first place, with security control being implemented by a "physical barrier". The system access point is placed in a closed area or behind a locked door, with access granted only to authorized personnel. Accessing this kind of system remotely will essentially override all existing security practices implemented using location, and could potentially give full access to the back-end system in case of a security breach.

D. Service API and Data Formats

The REST APIs mimic the underlaying data models: the (measurement) data and the building schema. The client-side Data Service API (shown in Fig. 1) contains only a single end point (GET /data/sensors/{sensorId}) with commonly used filters (time interval from-to in ISO8601 formatted strings, and start index and maximum results for paging the data). Calling the API without parameters results in the newest data object to be returned. As described earlier, the data consists of simple JSON key-value pair object arrays. If the back-ends do not provide history data, the end point filters will simply return an empty array for non-matching parameters.

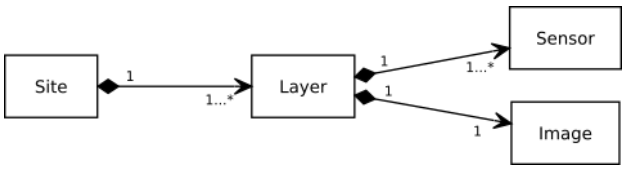

Figure 2. Object relations within the DataSites Service.

The Sites Service API contains three end points: /sites/{siteId}, /layers/{layerId}, and /sensors/{sensorId}. Each end point has HTTP methods for creating (POST), reading (GET), updating (PUT), and deleting (DELETE) schema data. Omitting {identifiers} will result in returning all data, and similarly to the Data Service API, all GET methods provide filters for paging the results. An optional /layers path can be appended to /sites/{siteId} and /layers/{layerId} for listing all child layers for the site and layer designated by the identifier.

The end points are based on the three core data models of the building schema, and their relationship is also illustrated in Fig. 2:

- **Sites**. Contains high-level metadata of the location (description, external URI link, name and organization) and an identifier. Only name and identifier are required properties. A site can have only one layer.
- **Layers**. Represents a single "layer" within a building complex or a single building. Includes metadata properties such as address, postal code, country, description, external URI link, name, organization, and coordinates (latitude and longitude). It also contains the properties required by the schema functionality: identifier, index, and type. One layer can have any number of child layers. Type is an enumerated constant, which describes what kind of layer (unknown, outdoor, building, floor, room) is in question. This enables the creation of hierarchical schemas. For example, layer 1 (building) has 2 child layers (floors) and each of those have 3 more child layers (rooms). Using the index (number) the child layers could be sorted in a specific order. Additionally, an image object (with scale and URL properties) can be added to each layer. The scale is a text field currently used only for informative purposes and the URL is a link to an image file. The file could be, for example, an image of a floorplan. Only name, type, and identifier are required properties. A layer can have any number of sensors, but only a single image. If multiple images relate to a single layer (e.g., both a floorplan and electrical plan are available for a floor), a separate layer must be created for each image, and the layers must be grouped below an (imageless) parent layer.
- **Sensors**. The sensor object of the Sites API is the metadata representation of a sensor (or a virtual sensor). It contains an identifier, external identifier, name, service type, and coordinates (x,y,z). An external identifier is basically a string that adapters use to resolve the original back-end data source, whatever that might be. The service type is the name of the back-end system. The sensors are always hierarchically placed below a layer, and if that layer has an image attached, the coordinates can be used to mark the position on that image. The coordinate system is in pixels and they are in relation to the original image. Thus, replacing the image requires re-calculation of the coordinates, which can be performed automatically. The identifiers and service type are required properties. As can be seen in Fig. 2, a single sensor can only be associated with a single layer. First, this is done to reduce many-to-many connections, and second, as the sensor coordinates are in relation to the image, unless both images are exactly the same size and scale, the coordinates would not match. Further, the sensor object is, in general, simply a

relation object. The external identifier can be used to track the real sensor (or data source). Multiple sensor objects can have the same external identifier.

In the Sites Service API all three models are presented as JSON objects. Each layer could have different metadata, or the metadata can be considered to "cascade" downwards in the hierarchy, depending on the client application's interpretation of the data. In the service, no metadata is automatically copied or updated to child layers when parents are updated.

We also host a basic file upload service for storing images and creating image links for layer pictures, although external image links are also accepted to allow hosting of an image on a third party service. The implementation of the image hosting service or the user management features are not described in great detail in this paper as pre-made implementations are readily available both as open-source and commercial components, and they are not "core features" in the context of this paper.

III. Use Cases

To illustrate back-end integration, two real-life use cases are presented in this section. The use cases - a building automation system and RuuviTag data collection - are highlighted in the high-level architecture, in Fig. 1, and they can also be seen in Fig. 3 and Fig. 4. The first use case is a more modern system, which offers a cloud-based interface (REST API) for retrieving data, and the second case implements an on-location interface for data retrieval.

A. Building Automation System

The building automation use case is a large four-story building located in Helsinki, Finland. The building contains a modern building automation system for monitoring conditions within the building and for controlling HVAC. More specifically, we were interested in the sensor data produced by the heating system. Every radiator in the building is equipped with a smart thermostat, which both controls the heating and collects operational data from the unit. There are about 25 radiators per floor, but the data collection is concentrated only on floors 2-4, making the total thermostat (i.e., sensor) count about 75.

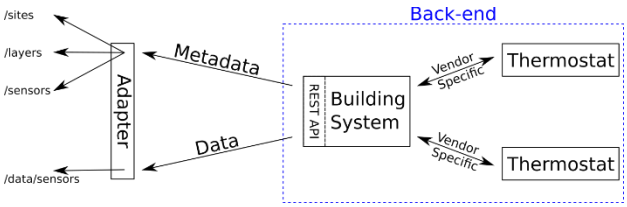

Figure 3. Illustration of the building automation system use case.

The building automation system uses the thermostat data (in combination with weather forecasts, outdoor temperature data, and statistics from the HVAC units) to optimize the heating of the building. The implementation of the thermostats, data collection, and the building automation were unknown to us, and our knowledge was limited to the data provided by the REST API. The "sensor" data consists of: the room temperature (as measured by each thermostat unit), room target temperature (or optimum temperature), signal level (the thermostats are wireless), battery level, timestamp of last communication (hourly updates), mac address, thermostat name, and valve status (the building has central heating and each thermostat has a valve that controls the flow of hot water into the radiator). The data is provided in JSON.

The REST API can be used to retrieve the basic metadata of the building (e.g., address, name, organization), which the adapter uses to generate the top-level site container and to fill out the appropriate properties in layers. In the data model of the back-end all thermostats are children to "site nodes" and nodes can be children to other nodes, somewhat similar to layers in our model. The nodes could represent rooms, buildings, floors, or perhaps thermostat groups, but unfortunately they are not enumerated or typed, and it can be slightly challenging to reliably interpret which node is a room and which one is a floor. The adapter will simply read the node's textual name and use it to figure out the node types. For example, if the name has the word "kerros" (floor, in Finnish) in it, the node is assumed to be a floor node, if the node name has the letter H followed by numbers or the word "huone" (room, in Finnish) in it, the node is assumed to be a room node. The approach worked in this case, but as the names are simply text fields, the naming convention might be entirely different in another building, requiring modifications to our adapter.

The adapter creates a single site and one layer per node to describe floors and rooms and internally inserts these through appropriate interfaces found on our service, as illustrated in Fig. 3. Based on the thermostat data, sensors are created in our service, using the thermostat name as the sensor name, the mac address as an external identifier, and a back-end specific service type name is generated. There is no method for retrieving a building blueprint or floorplan in the back-end API, and the system does not contain exact thermostat locations within the rooms.

Finally, upon client request, the adapter will use the sensor mapping to resolve and retrieve the actual data from the building system. The company managing the building system also provided us with floorplans to use as layer images and for demonstration purposes the thermostats were placed on the image at their exact coordinates. Admittedly, manually setting nearly a hundred sensors can be tedious even though the work, in general, needs to performed only once.

B. RuuviTag Data Collection

The RuuviTag data collection case is a small, single-floor children's daycare center located in Pori, Finland. The back-end system consists of a combination of InfluxDB database, Raspberry Pi, a group of RuuviTag sensors [11] installed in various rooms in the buildings, and a possible on-location interface connection (Ethernet, BACnet) to the building automation system. The overall setup is illustrated in Fig. 4.

The manufacture of the building automation system also offers a web interface, which allows viewing of history data and various HVAC schematics, but no floorplans or blueprints of the building are available, nor is there any metadata that would help to discover the exact locations of

the various sensors or devices within the building. In this case, we asked the building owner to provide a blueprint and we set up the sensor locations manually for demonstration purposes. There were only around 15 sensors total, and figuring out the device locations manually was a straightforward task.

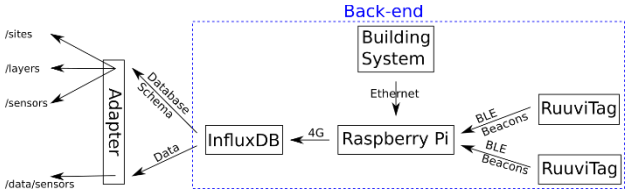

Figure 4. Illustration of the RuuviTag data collection use case.

In addition to the building automation data, the RuuviTags include temperature, relative humidity, atmospheric pressure, and accelerometer sensors. The building automation system includes temperature measurements from outside the building and within the HVAC unit, but no accurate room-based measurement. The Java application (RuuviCollector [12]) used to monitor the Bluetooth Low Energy (BLE) beacons used by the RuuviTags for transmitting measurement data also calculates certain derived values (e.g., dew point, absolute humidity). The same application also relays the data to the InfluxDB. The Raspberry Pi is located inside the building, but the InfluxDB is installed on a virtual machine accessible over the Internet using a 4G modem. The InfluxDB could also be installed on the Raspberry, but this would increase the CPU and memory load of the Raspberry Pi when data queries are performed. Also, in the current setup, the Raspberry Pi simply sends data and there is no need to allow inbound access to the building's network.

The data retrieval from InfluxDB was implemented within the adapter as a pre-created list of database query templates. To resolve the building schematics, in this case, the only option was a manual one. The building automation system is part of the building, but other components were implemented by us, and because of this, we knew certain internal details of the back-end implementation that were not available to us in the "Building Automation System" use case. To be precise, we knew the database "schema" of the InfluxDB database. The adapter uses the database name-measurement name-tag name structure of the database to automatically construct the site-layer-sensor hierarchy, requiring only address and organization details to be typed in manually if a similar setup is installed in another building. In practice, based on our experience, it is quite common that building schematics are not available as structured data, and especially with older buildings it is possible that only paper (or photographed) blueprints are available.

IV. Conclusions

The paper presented the DataSites service, which integrates back-end systems and client-oriented REST APIs. The high-level view of the DataSites architecture, how to relay building data to client applications, and the required interfaces, were described in this paper. Furthermore, the paper showed how simple, basic components can be used to construct a viable service for providing data building data to client applications. In test scenarios, the use of the service was illustrated using two real-life use cases.

The research gave rise to some ideas for future research: Hot loading new adapters without stopping, shutting down, or rebooting the system would be usable when using remote controlled systems. The second issue to be developed might be usability improvement with improved methods for setting sensor locations.

Acknowledgments

This work is part of the KIEMI project and has been funded by the European Regional Development Fund and the Regional Council of Satakunta.

References

[1] Statistics Finland, “Total energy consumption decreased and consumption of renewable energy grew by one per cent in 2019,” Official Statistics of Finland (OSF): Energy supply and consumption [e-publication]. ISSN=1799-7976. 2019. Helsinki: Statistics Finland, https://www.stat.fi/til/ehk/2019/ehk_2019_2020-12-21_tie_001_en.html, Retrieved: January 29, 2021.

[2] T. Niemelä, R. Kosonen, and J. Jokisalo, “Cost-effectiveness of energy performance renovation measures in Finnish brick apartment buildings,” Energy and Buildings, vol. 137, ISSN 0378-7788, 2017, pp. 60-75.

[3] M. Saari, P. Sillberg, J. Grönman, M. Kuusisto, P. Rantanen, H. Jaakkola, J. Henno, “Reducing energy consumption with IoT prototyping”, Acta Polytechnica Hungarica, Volume 16, Issue Number 9, 2019. http://www.uni-obuda.hu/journal/Issue96.htm .

[4] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci, “Wireless sensor networks: a survey”, Computer Networks, 38(4), pp. 393–422, 2002.

[5] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications,” IEEE Communications Surveys & Tutorials, vol. 17, no. 4, pp. 2347–2376, 2015.

[6] M. Saari, J. Grönman, J. Soini, P. Rantanen, and T. Mäkinen, “Experimenting with Means to Store and Monitor IoT based Measurement Results for Energy Saving,” in 2020 43rd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2020.

[7] M. Nurminen, A. Lindstedt, M. Saari and P. Rantanen, “The Requirements and Challenges of Visualizing Building Data” in 2021 44th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2021. Submitted.

[8] P. Rantanen and M. Saari, “Towards the utilization of cost-effective off-the-shelf devices for achieving energy savings in existing buildings”, Proceedings of 2020 IEEE 10th International Conference on Intelligent Systems (IS20), Varna, Bulgaria, 26-28 June 2020.

[9] OpenAPI Specification, https://swagger.io/specification/, Retrieved: January 29, 2021.

[10] Swagger Codegen, https://swagger.io/tools/swagger-codegen/, Retrieved: January 29, 2021.

[11] RuuviTag, https://ruuvi.com/files/ruuvitag-tech-spec-2019-7.pdf, Retrieved: January 29, 2021.

[12] RuuviCollector, https://github.com/Scrin/RuuviCollector, Retrieved: January 29, 2021.