View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AIS Electronic Library (AISeL)

Proceedings of the 51st Hawaii International Conference on System Sciences | 2018

Capability-based communication for green buildings and homes - a REST-like API within the conex.io project -

Olaf Droegehorn, Philipp Trenz, Benjamin Brausse, Timo Schwan, Christin Voscort, Marcel Wemmer
Harz University of Applied Sciences, 38855 Wernigerode, Germany
[odroegehorn, u30054, u30583, u30357, u30001, u29737]@hs-harz.de

Abstract

Within the Kyoto protocol and the Paris agreement the world's countries have agreed to limit global warming to a maximum of 2°C. The European Union has passed directives to mitigate emissions from buildings, as around 36% of the EU's total CO_2 emissions stem from them. To implement these directives, the use of home automation systems can be a significant contribution installed in existing, even renovated households. Looking to the global home automation market it becomes clear that none of the available vendors/solutions can cover a sufficient end-user scenario alone. And even with a multitude of technologies the integration of different systems is a tedious work as most of the systems are technically incompatible to each other. Tackling this challenge with open source software promises an easier integration but usually comes along with issues of heterogenic command syntaxes and parameter sets. This paper outlines a REST-like API and an abstraction mechanism, enabling user-interfaces and front-ends to communicate with smart home systems based on capabilities instead of protocols and technologies. The API decouples front-ends from specific smart home technologies and allows for a seamless integration of new protocols without touching the code of a front-end again.

1. Introduction

The primary purpose of building automation systems (BAS) is to optimize cost and energy efficiency in operating building spaces through the automatic and remote control of indoor environmental conditions. This can be done by regulating the heating, air-condition, ventilation and lighting systems of buildings through the deployment of interconnected sensors and actuating devices. A home automation system (HAS) is a specialization of BAS where, besides optimizing energy consumption, the comfort and peace of mind of the home inhabitants are of similar priority.

In recent years, reducing energy consumption in buildings has gained increased interest amongst researchers, due to the growing global awareness about the need to achieve long-term environmental sustainability. Beyond this, the numerous national legislations being approved to reduce CO_2 emissions demand immediate actions on this topic. According to the European Commission, buildings account for 40% of energy consumption and 36% of CO_2 emissions in the EU. The European Commission estimates that by using proven and commercially available automation products in buildings, it is possible to reduce the total EU energy consumption by 5-6% and the CO_2 emissions by about 5% [1].

However, despite the wide availability of home automation technologies, a significant number of repeated commercial failures has been noted and the reluctance of customers to invest into these technologies still remains relatively high. Many reasons have been proposed to explain this phenomenon and these include: lack of flexibility and scalability to adapt to new technologies, the diverse availability of products, not being compatible with one another, and last but not least, the low usability of HAS technologies.

This paper describes the design and implementation of a REST-like API together with an abstraction mechanism allowing finally HAS infrastructures to adapt in a faster manner to new technologies and to provide better user experiences.

A. Background

Modern home automation system (HAS) architectures are usually distributed across a three-level hierarchy namely: a field level, an automation level and a management level. Figure 1 shows the main functions that are associated with each level.

Currently there is no single standard technology that covers all three levels of the architecture and as a result heterogeneous technologies and design solutions have proliferated with no standard principles for interoperability. This greatly affects the development of

URI: http://hdl.handle.net/10125/50612
ISBN: 978-0-9981331-1-9
(CC BY-NC-ND 4.0)

a management-level application, such as a visualization- and control user interface, as each technology comes with its own data representation that is tightly coupled to its internal requirements. Since it is difficult to integrate all these sources of data into a single information model it becomes complicated for engineers to build a good and universal user interface for the HAS.

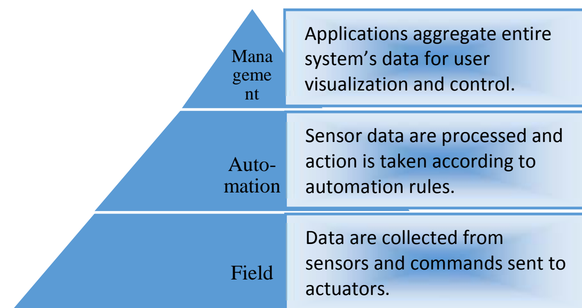

Figure 1: typical architecture of a HAS [3]

With the increased availability of different automation technologies and Software-as-a-Service business models, home automation has evolved from being an industrial application retrofitted for domestic environments to a highly consumer-focused Internet of Things application.

As home automation is now targeting a wider range of customers, especially non-technology enthusiasts, it is important to include the user needs, wishes and expectations into the design considerations of this technology. Since it is commonly accepted that a home environment holds an emotional attachment for its inhabitants HAS application developers should mainly concentrate on design aspects and user experience instead of addressing technology in many different ways.

B. Statement of the Problem

To encourage house owners or residents to install and operate home automation systems the applied solution should cover at least conceptually all the requirements of the users. But looking into existing vendor-driven solutions like Philips Hue®, Insteon, eQ3 HomeMatic®, Buderus, etc. it gets obvious that each of those vendors is covering only a subset of how a real smart home should look like. This observation typically leads to a multitude of installed systems, being isolated to each other, as only very few vendors are providing bridges to standardized protocols.

Some of them try to integrate with interaction systems from service providers, typically residing in the management level (see figure 1), like HomeKit from Apple® or Alexa from Amazon®. But this integration is usually produced by a very specific gateway, connected to the internet and offering only the vendor specific elements to the denoted services. This again leads to isolated systems with many gateways, open ports in the Firewall, and at best a partial integration in one of the internet-based services. But no common control element, no central in-house management and especially no integration of services that offer only their own control software, like central heating systems, garden watering elements, etc. (see Buderus, Gardena), can be achieved in this way. This doesn't help the end-users for an overall systematic approach and leads to a high level of uncertainty, if a HAS can achieve something for - and can be handled by individuals.

In order to integrate and to take up technologies, either being already available or showing up on the market, open source approaches seem to be the most promising solutions for end-users not willing to limit themselves to a specific vendor. Especially as these solutions are built by many motivated developers and therefore benefit from a quite fast integration of new products into their systems. This is quite important because the market has seen many new vendors during the past 12-18 months, delivering very specific solutions for controlling different devices and measuring specific values, but being mainly incompatible to each other.

Open source approaches are usually including new, specific modules into their infrastructures in order to integrate and communicate with new technologies, based on very specific commands related to the hardware or interpretation of measured values. As an example the FHEM project [13], being the largest German open source project on home automation, uses a dedicated software module for each vendor technology. These modules, written by members of the open source community, implement the specific protocols to communicate with their related devices like lamps, sensors, heat radiators, etc. Although there are guidelines on how to integrate a module in the HAS infrastructure, typically no guidance is given on how to interpret values or how to name interaction mechanisms in a common way, as the functions of these modules might be completely different. Ultimately no common semantic or ontology is used as a ground work within these projects, as all the vendors and the motivated developers are using their own terms and definitions. This holds for many open source projects like FHEM, openHAB, Home Assistant, OpenMotics, Domoticz, etc. [20].

In the case of the FHEM project this leads to a syntax for addressing a dimmer of the FS20 technology like "set dimmer 43%" whereas the same command for a HomeMatic device looks like "set dimmer PCT 43%". This is only a very simple example but already shows the different syntax, based on the different interpretation and command structure of the integrated technologies.

The same applies to large vendor solutions based on third party technologies like the "SmartHome" from Deutsche Telekom [19], which tries to integrate HomeMatic and EnOcean technology. Here also the

only point of integration is the front-end addressing different devices with a completely different syntax.

Observing this it becomes clear that, although open source projects as well as vendor solutions based on third party technology are trying to bring different technologies together, the lack of at least common syntaxes and data interpretation requires a change in the code of front-ends and service bridges whenever a new technology is integrated. This leads to bad user experiences due to complicated or outdated user interfaces and a slower uptake of home automation systems overall.

C. Aim and organization of the paper

Within this paper the conex.io project is outlined, in which a REST-like API has been developed, residing on an abstraction layer in order to eliminate any technology specific syntax and introducing a capability oriented interaction mechanism. This enables user-interfaces and front-ends to communicate with smart home systems based on capabilities instead of protocols and technologies from different vendors and allows easy integration of new vendors without modifying front-ends and user interfaces. Therefore, the REST-like API in this paper can be seen to reside between the automation- and the management level depicted in figure 1.

Within this paper section two presents a literature review whereas section three explains the architecture of the intended system. Chapter four introduces the REST-like API, which allows for capability based interaction and its unique filter architecture. Section five describes the mapping layer and its abstraction mechanism from technology-specific aspects and the needs to make this abstraction possible. Section six discusses the advantages of the used toolchain and the benefits of this API whereas section seven concludes the paper.

2. Literature Review and related Work

The report given in [4] provides a summary of the energy usage for residential and non-residential buildings in EU states and a comprehensive analysis of how the effects of the economic, energy prices and occupant's behaviors affect energy usage. The analysis is based on the energy usage data and energy efficiency indicators provided by the ODYSSEE database and website. The energy usage in buildings may vary per country, however this consumption represents in average a total of 41% of the energy usage in the European Union (EU) and from this lot, residential buildings accounts for 65.9% of the total energy usage of EU buildings and 27% of the energy consumption in the EU. For Finland, Spain, Portugal and Cyprus building energy usage represents 33.33% of their total energy usage while for Germany, Denmark, France and Poland building energy usage represent 45% of the overall energy consumption. Also, while the distribution of building energy consumption between residential and non-residential buildings may vary per country, the share for residential building from the total building consumption for Germany and Finland ranges between 60-70% and the annual consumption per (kWh/m^2) for these two countries are 210 and 325 respectively. This disparity is associated to climatic differences between the two countries and therefore in Germany not so much energy is used overall but the percentage of space heating is larger due to the lack of other needs as water heating or lighting. A breakdown of the energy consumption per household for both Finland and Germany in table I reveals that space heating represents the largest share of the total household energy usage.

Distribution	**Germany (%)**	**Finland (%)**
Space Heating	75	66,7
Water Heating	12	14
Electric Appliances and Lighting	12	19
Cooking	1	0,3

TABLE I: Building energy consumption per usage category

A comparison of the energy usage for space heating from the year 1990 to 2009 reveals a reduction trend for the EU average usage with a ratio of 30-60%. This reduction was attributed to the implementation of thermal regulations from EU countries for new buildings. However, the data provided by [5] for heat consumption per square meter (m^2) at normal climate conditions reveals that between the year 2000 and 2012, Germany recorded a 17.38% decrease in energy usage with figures 17.472koe/m^2 and 12.436koe/m^2 - respectively while Finland recorded a 2.18% increase with figures 15.583koe/m^2 and 15.923koe/m^2 - respectively. This implies a 21% energy usage difference for space heating for Finland and Germany for the year 2012.

Comparing the energy usage for electric appliances per dwelling for the year 2000 and 2012, the data given in [5] reveals that Germany recorded a slight 8.81% increase from 2078kWh to 2261kWh respectively and Finland recorded a significant 30.23% decrease from 4548kWh to 3173kWh respectively. This implies a 29% energy usage difference for electricity for Finland and Germany for the year 2012.

The ecoMOD project by the University of Virginia given in [6] entails the design, construction and evaluation of houses for energy efficiency. This project aims to achieve three objectives: academic, environmental, and social. An energy monitoring

system was installed to retrieve sensory and actuation data every second. This monitoring system comprised of cost effective sensors that measure temperature, humidity, air quality, water flow, electric usage for appliances, carbon dioxide level and wind speed. Sensory and actuation data were retrieved through a wireless connection and these were stored on a remotely accessible database. A detailed data analysis was conducted on a 20-day stored data using a custom developed web data-analytical application software and the data analysis results indicates that the Heating, Ventilation and Air Conditioning (HVAC) and water heating system contributed the larger portion of the energy consumption with both measuring 38% and 21% total energy consumption respectively. Also the result indicates a 50% and 45% reduction in the envisaged energy consumption of the building. The discrepancies between the envisaged consumption and the analysis result for the hot water heater and HVAC was not justified with measured data, however it correlated with the result of a similar study given by [2].

Utilizing various wired and wireless media approaches for implementing smart gateway architectures for home automation were extensively discussed in [7], [8], [9] and [10]. However, setting up an architecture doesn't necessarily provide a way how to implement that architecture.

From the home/technology perspective, one of the main technology-related problems that HAS application developers focus on is the interoperability of the heterogeneous automation technologies. According to [11] these diverse technologies and protocols have caused a problem of integration at the information level due to the lack of interoperability of their data representation of devices and building layouts. The lack of interoperability in data description mechanisms makes it difficult to integrate different sources of information at the management level which in turn affects the development of high-level applications to process, visualize and control the entire automation system. Within [12] the problem to manage and aggregate useful information from a large amount of data, being generated by technology indifferent devices, has been described. This design problem was approached by implementing a standardized template system or information model that defines a common language for data representation and storage for both devices and buildings, also called Building Information Model(ling) (BIM).

3. Capabilities and the system architecture

Based on existing approaches and identified solutions of the literature review it seems a valid starting point to build a system architecture, which can communicate based on capabilities rather than on technology-oriented protocols. Open source projects already integrate a multitude of available technologies; therefore, different home automation systems should be taken into account. In our example the open source project FHEM [13] was selected, but the architecture and the system layout was designed in such a way, that any underlying HAS could be used.

The architecture for providing a capability-based interface was designed in the conex.io project [21], which aims at fostering the uptake of home automation systems. Within the project a system architecture has been laid out, an API was designed as well as a mapping layer, which translates the abstract devices and their capabilities into specific syntactic objects for the underlying HAS. The project itself consists also of a team for live demos, web casts and physical presentations for developers and even end-users, in order to foster the uptake of the API and of a related HAS itself. It connects to several activities in Germany together with the Harz University of Applied Sciences, the IBM Client Innovation Center Germany GmbH, the DHS-Computertechnik GmbH and the FHEM community overall, in order to ease the use and development of an open HAS, which provides a good and hassle-free user experience while integrating new technologies when they appear [22], [23], [24].

To be able to provide such an API it was needed to abstract from specific technological elements and their protocols. Nevertheless, a HAS and its users want to manage devices, as they represent physical objects and related functions to them. Therefore, the conex.io project uses the same concept of devices like an underlying HAS but in a more abstract manner. The (physical) devices themselves will be kept, enriched with some attributes being useful for management and navigation (see section 4) but stripped from any detail of the underlying technology. Instead of this the capabilities of a devices will be described by so called functions, which are abstract mechanisms for controlling a device or getting data from it. Figure 2 depicts the idea where an (abstract) device has certain capabilities and is a container for several functions. E.g. a room-controller for a heat radiator has the capability to control the room temperature based on measured values and programmed schedules. As a consequence, this device, in an abstract view, contains several functions like the temperature sensor, a humidity sensor as well as an actuator to open a valve and commands to program a schedule. This concept is the baseline of the conex.io API-project and allows the control and usage of abstract devices based on functions building up their capabilities.

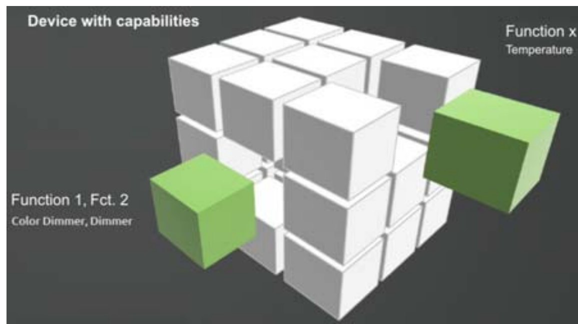

Figure 2: Device with capabilities and related functions

These functions, being the generalizing elements in this approach, can be defined by the API designer or maintainer. Of course they can be based on ontologies or taxonomies, and several research projects have tackled this, but it can be observed that none of the realistic solutions (being in practical use) have reached a common semantic ground represented by an ontology. And especially within the domain of open source only very few participants have even the knowledge to use or follow an ontology. Because of this, the conex.io project has designed an API description mechanism, allowing the maintainer (-community) of an HAS to define these functions and their names according to their own wording, preferably maintained in a dictionary. Simple examples for these functions are "onoff" for a switch, "dimmer" for dimming devices or "temperature" for sensors to get data or valves to set a target temperature. Although this seems to be quite simple it appears that these very basic functions are programmed with different syntaxes for different technologies in most cases. Based on this concept the architecture of the conex.io project, as shown in Figure 3, incorporates several elements.

Figure 3: Architecture for a Capability based interface

The API itself is built in a REST-like manner. The details of the API and its design will be explained in the following section. To build the API the Spring framework (spring.io) was chosen, as this gives an easy entry for designing and implementing the API. The API itself incorporates several endpoints, as usual for a REST API, and is stateless, meaning that each request needs to contain all necessary information and is not related to any other request. This API allows front-end developers to address all desired devices or capabilities independent from any underlying technology. Under the API itself, a database of abstract devices, their capabilities, built by a set of functions, actual states and values has been foreseen. This database is mainly intended to reduce the amount of interactions with the HAS, as requests from the API about state evaluations can be answered directly from the interface layer. The database should therefore store the actual states of devices in an abstract manner and also values of related sensors and actuators. In which way this database should be implemented has not been specified on this level, as several possibilities are available. Either a real database-system can be incorporated, storing each change of values communicated from the HAS for the devices, or a real-time update mechanism, leading to a change of simple state-variables within the objects for the abstract devices within the interface can be used. This highly depends on the level of complexity and available mechanisms of the HAS to communicate state updates.

For the implemented case, being FHEM as the HAS, this was realized as a simple Web-Socket connection to the FHEM-system, delivering all state changes from the HAS to the interface and leading to a state change of related Java-Objects, representing the abstract devices. This is mainly necessary as the API is designed in a REST-like manner, meaning it is completely stateless and therefore each request for a state of a device would lead to a newly formed request towards the HAS. This can be reduced significantly by storing the relevant states in the database, or in this case in variables of the Java object for the corresponding device, of the API.

The mapping layer maps the technology specific procedures and values into abstract devices with functions and supported values of the API.

The descriptions of functions as well as of the FHEM modules are done in a technology agnostic way, using YAML and JSON respectively. This is shown in Figure 4, where the YAML descriptions denotes the available functions for the device capabilities (denoted in figure 3 as capability descriptions) and the JSON files for the technology specific HAS modules.

The descriptions are further detailed in section five, explaining how module developers within open source communities could make use of the API and its mechanisms. The mapping layer translates the functions, describing the capabilities of the abstract devices, specified in the YAML file, into specific technological commands for the HAS. According to the JSON description the correct command syntax will be derived to address the HAS and the used technology in there. With the use of JSON-files, describing the technology specific aspects of the HAS, the mapping layer generates the technology agnostic abstraction using a parser to read out data from the HAS.

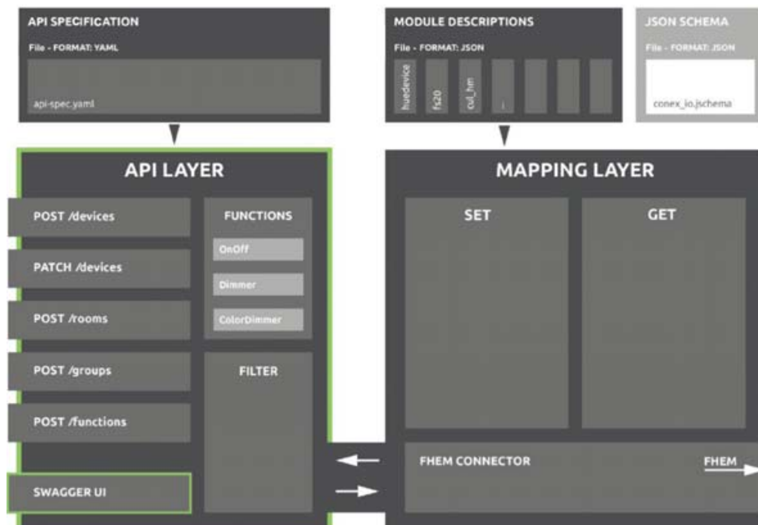

Figure 4: The API and related description files

Whenever a new technology is integrated in the HAS only the corresponding JSON file needs to be added to the API architecture to empower front-ends to use these devices at the API layer.

The connection to the HAS is built by a connector (in this case for FHEM), which could be implemented in any desired way to fit to an available system. This doesn't only apply for open source solutions, but also to vendor specific or third party servers like the system from Deutsche Telekom, etc.

4. REST-like API based on Capabilities

The typical design of a RESTful API is to define an endpoint and add a specific path for each resource to it. This is in principle doable but whenever a new device is integrated in the HAS a new path needs to be added to the endpoint, like "/device/new_device_id". This would be conforming with the REST architectural approach as defined in [14]. But for front-end developers this is quite complicated as each front-end needs to be able to formulate and parse new paths whenever a new device and therefore a new path-element is added. This is not practical and doesn't lead to simplifications for front-end developers, which is the ultimate goal of the REST-like API here.

Therefore, the conex.io project has selected a REST-like approach, in which only a few endpoints have been defined but equipped with a powerful filter architecture. This API defines the following endpoints:

"/devices", "/rooms", "/groups", "/functions"

These four endpoints are able to address either devices, defined in the HAS and abstracted by the interface, rooms in which those devices are located (if supported by the HAS), specific groups, which may be used to organize sets of devices, and functions characterizing the capabilities, which are applicable to numerous devices. As this omits a very specific URI per resource, meaning here per device, this violates the RESTful requirements but is still REST-like, as the request scheme is still stateless and JSON is used for data transport.

Each of these endpoints is using a filter mechanism based on JSON, which will be delivered in the BODY of the requests. Therefore, the HTTP-GET method can't be used, as according to the HTTP standard everything has to be encoded in the URL. This isn't doable with complex filter-objects and therefore the requests for status towards the API are encoded by the HTTP-POST method and state updates are communicated by HTTP-PATCH methods. This again violates the RESTful approach, but as HTTP-methods are used, although not the semantic obvious one (like GET for status request), it still remains to be REST-like.

For each of the requests a JSON based filter can be added in the BODY-part, which limits the selected devices or functions. This is shown in Figure 5, where the different filters for the "/devices" endpoint are listed.

POST /devices

Description

Get a list of device objects based on the specified filter.

Parameters

Name	Located in	Description	Required	Schema
filter	body	The user specified filter	Yes	▼Filter { device_ids: ▶[] room_ids: ▶[] group_ids: ▶[] function_ids: ▶[] }

Figure 5: REST-Endpoint for "/devices" with filters

What can be seen is, that although the endpoint itself is designed for devices, the filter allows to limit the response to a very specific set of device_ids, a set of room_ids, group_ids and even function_ids. By this an application can request a list of all devices being together in a specific room or group or supporting a specific function as part of their capabilities, like dimming or switching. As this endpoint delivers the device objects, and not only the IDs of the devices, including the capabilities represented as function objects in a JSON representation, the states of the different devices can be retrieved.

With the endpoints "/rooms", "/groups", "/functions" an application can request IDs of rooms, groups and functions, which apply to the filtered objects. By this each front-end can derive the desired

subsets of devices and their states, according to the specific needs. Additionally, the PATCH method, being implemented for the "/devices" endpoint, can be used to alter the state of a single or a multitude of devices. This leads to possibilities like switching on or off all devices that might support these capabilities, like switches, dimmers, etc. or to set all dimmers in a room or a group to a specific value. With this HTTP-method also the JSON filter object can be passed, so that a front-end developer can issue a single command to a multitude of devices, depending on their capabilities, their physical or logical organization.

Although the use of a JSON-based filter object violates explicitly the RESTful requirements, as no endpoint/URI is built per resource and the methods for requesting data needs to be POST instead of GET, this seems for the purpose of easing the front-end developers' life more than meaningful. No drawbacks are arising from this filter architecture but giving the user of this API a great amount of power by addressing several devices or functions at the same time. This also reduces traffic between mobile front-ends and the API itself, as same commands don't have to be repeated for different devices, saving therefore resources, lines of code and air time. And beyond this all these commands can be issued completely independent from the underlying technology or technology-specific syntax of the used HAS.

5. The Mapping Mechanism

For being able to abstract from technology specific protocols and commands of the HAS the API architecture incorporates a mapping layer. This layer is based on two description mechanisms, outlining the different levels of abstraction. A JSON description for each of the HAS specific modules defines the different commands, values and their interpretation as well as specific paths of those values in the data structure of the HAS.

The functions supported by the API-layer are described in the YAML API-specification. The mapping layer has to translate technology specific commands, described in their syntactic structures in the different JSON-files, into the abstract functions provided via the API, building the capabilities of the devices and the overall system (see figure 4).

The communication with the HAS should deliver a JSON structure of defined devices in the HAS. This can be implemented for each desired HAS but should return the data structures of the devices in the HAS via JSON in a meaningful way. In the case of FHEM being the HAS figure 6 shows the JSON-structure for a specific device of the FS20 technology.


```
{
  "Name": "FS20_Lampe",
  "PossibleSets": "dim06% dim100% dim12% dim18% ... off ... on ...",
  "PossibleAttrs": " ... ",
  "Internals": {
        "BTN": "56",
        "DEF": "1234 56",
        "NAME": "FS20_Lampe",
        "NR": "85",
        "STATE": "dim75%",
        "TYPE": "FS20",
        "XMIT": "1234"
  },
  "Readings": {
        "state": { "Value":"dim75%", "Time":"2017-08-06 12:43:47" } },
  "Attributes": {
        "IODev": "CCD",
        "model": "fs20di",
        "room": "Wohnzimmer"
  }
}
```


Figure 6: JSON data from the HAS for a FS20 device

To allow the mapping layer to parse the JSON data from the HAS corresponding JSON-description files for each used technology in the HAS are needed, as the syntax, paths, data representation and -interpretation of each technology might be different. The conex.io project has built a JSON-Schema that defines the syntactic structure and rules for these files.

Element	Key	Value
information (description: Mapping instruction...)	filename	fs20.json
	author	Philipp Trenz, Benj
	last_edited	2017-04-10
device_id	key_path	Name
type_id	key_path	Internals/TYPE
rooms	key_path	Attributes/room
	delimiters	,
groups	key_path	Attributes/group
	delimiters	,
functions	get	
	set	

Figure 7: Structure of JSON-description file, example for FS20

They are built in such a way, that the representation of a technology in the HAS can be parsed and all related values, SET/GET commands and identifiers can be found. The beginning of the structure is depicted in figure 7, showing the information fields needed and sample data for the FS20 technology.

As shown the paths for the name and the type of a device in the delivered JSON data is denoted in such a way that the parser can derive the name of the delivered device "FS20_Lampe" and the type "FS20" in a straight forward way (apply fig. 7 to fig. 6). This is quite simple

for string-based data, as those only need to be read out. As soon as state variables, ranges or even units are part of the data, more complex mapping modes can be applied. Here several possibilities like regular expressions, range converters and min/max rules can be used to derive the data from the HAS and to convert it to a common scale used in the conex.io project.

```
"device_id": {
  "key_path": "Name"
},
"rooms": {
  "key_path": "Attributes/room",
  "delimiters": ","
},
"groups": {
  "key_path": "Attributes/group",
  "delimiters": ","
},
"functions": {
  "get": [
    {
      "function_id": "OnOff",
      "requirements": [
        {
          "attributes": [
            "on",
            "off"
          ],
          "delimiters": " "
        }
      ]
    },
    {
      "function_id": "Dimmer",
      "requirements": [
        {
          "mode": "contains_all",
          "key_path": "PossibleSets",
          "attributes": [
            "on",
            "off",
            "dim([0-9]{1,2}|100)%"
          ],
          "delimiters": " "
        }
      ],
    },
  ],
  "set": [
    {
      "function_id": "OnOff",
```

Figure 8: Excerpt of the FS20.json file

This leads overall to several JSON files, one for each technology. An excerpt of the FS20.json file is shown in figure 8 depicting also the function identifiers of the API, that are supported by the FS20 technology. When parsing the above sample data (fig. 6) based on the FS20.json description an internal representation of the device in an abstract form will be generated. This is shown in figure 9.

```
{
    "device_id": "FS20_Lampe",
    "type_id": "fs20",
    "room_ids": [
        "Wohnzimmer"
    ],
    "group_ids": [],
    "functions": [
        {
            "function_id": "OnOff",
            "isOn": true,
            "timestamp": "2017-08-06T10:43:47.000Z"
        },
        {
            "function_id": "Dimmer",
            "value": 191,
            "timestamp": "2017-08-06T10:43:47.000Z"
        }
    ]
}
```

Figure 9: Internal abstract device representation

By this an abstract device representation in the API-infrastructure is derived for each device in the HAS, if a technology specific JSON file exists. If a new technology is integrated into the HAS only the JSON description file needs to be produced and integrated into the API infrastructure in order to allow the mapping layer to use it. So given the case that a new dimmer or switch, using another different technology and therefore a different syntax and scale is integrated, the JSON description can be added and the device is abstracted in the same way as the others using the abstract functions (here “onoff” and “dimmer”) from the YAML specification. In this way the open source community can easily integrate new technologies without changing any code in the API-infrastructure, just adding the JSON files, and even without touching any front-end code whatsoever.

As the REST-like API is based on functions, building up the capabilities of the different devices, the specification of these functions should be done beforehand. All desired functions, being part of the capabilities of the technical devices integrated by the HAS, are described in a central YAML file [16] (see fig. 4). This file incorporates descriptions about the end-points, the filters added to the endpoints and the supported functions like “onoff”, “dimmer”, etc. of the API.

The format YAML is used because the JAVA code for instantiating and handling the endpoints can be automatically generated within the Spring Framework, using the Swagger-CodeGen tool [17]. This allows for a fast generation of API functions, being used in the mapping layer for device and function abstraction. An excerpt of the YAML file is shown in Figure 10. In such a way the functions and finally the capabilities supported by the API, the needed JAVA code, the endpoints and their filters can be specified and generated.

```yaml
Function:
  description: Base Class for all functions
  type: object
  discriminator: function_id
  properties:
    function_id:
      type: string
  required:
    - function_id
OnOff:
  description: Generic On-Off switch
  allOf:
    - $ref: '#/definitions/Function'
    - properties:
        isOn:
          type: boolean
          default: false
        timestamp:
          type: string
          format: date-time
Dimmer:
  description: Generic slider switch
  type: object
  allOf:
    - $ref: '#/definitions/Function'
    - properties:
        value:
          type: integer
          minimum: 0
          maximum: 255
          default: 0
        timestamp:
          type: string
          format: date-time

ColorDimmer:
  description: Generic color dimmer using the hsv color model
  type: object
  allOf:
    - $ref: '#/definitions/Dimmer'
    - properties:
        hue:
          description: Color value in degrees
          type: integer
          minimum: 0
          maximum: 359
          default: 0
        saturation:
          description: Saturation of color
          type: integer
          minimum: 0
          maximum: 255
          default: 0
```

Figure 10: YAML description for capabilities

Whenever a new function is added in the YAML specification the API-infrastructure needs to be recompiled and redeployed as the code for the new function needs to be added. If the newly integrated function is only used in a new technology from the HAS, this function just needs to appear in the newly produced JSON file. If any of the existing technologies also support the new function, it has to be added in the existing JSON files also.

This leads to a very flexible API system, being able to be supported and enhanced within any open-source community. The core maintainers of an open source community could be responsible for the specification of the API-functions and capabilities (the YAML specification) whereas each and every technology contributor could add JSON files into the API infrastructure.

6. Discussion

Within the conex.io project an API for abstracting different smart home technologies was designed, incorporating an underlying home automation system to communicate with different technologies.

This API is based on capabilities of abstract devices, split into several functions, which are defined in a YAML file and delivered via several REST-like endpoints for application programmers. The API itself is designed to be deployed in an application server like Apache TomCat [18]. As the design of the API is mainly controlled by the specifications in the YAML file a two-step build-process needs to be executed. First the JAVA code for the endpoints and the abstract devices and their capabilities need to be generated out of the YAML description. Afterwards the generated code and control-logic of the mapping layer need to be compiled into a Web-Application Archive (WAR) for being deployed in an Application Server. This process is based on a quite sophisticated tool chain based on the Spring Framework and needs to be installed in order to build the API.

This toolchain can, based on the YAML file, also generate the client code for front-ends and applications, which needs to be included to use the API. Therefore, the application developer doesn't have to address the API by himself but can use the pre-generated code from the toolchain in his front-end project.

This eases the task of front-end developers significantly as the front-end itself doesn't have to deal with differences stemming from different technologies in the HAS anymore. The same holds for gateways towards internet-based interaction systems like Alexa or HomeKit, as for now either each technology has to provide a specific gateway or an integrating HAS has to reformat each technology for those services manually. With the presented API a general abstraction and therefore gateway towards these systems can be derived without the need of changes for new technology integrations. This can easily lead to better user interfaces, as front-end developers now can focus on the user experience instead of dealing with communication and technology issues related to the underlying HAS. In the same way the integration of new technologies in existing systems and their front-ends can be faster as no changes in the code of front-ends and gateways need to be made. This might lead to a much faster uptake of home automation systems as the user experience could be much better and a more hassle-free installations and updates could be done.

7. Conclusion

This paper presented the API of the conex.io project, enabling a capability based communication with smart homes and buildings. The API resides on top of a selected Home Automation Server, meaning on top of the automation layer in figure 1, and abstracts the communication from any technology specific aspects the HAS may still have, especially considering open-source projects.

The API itself provides a REST-like interface with endpoints that support specific filters. These filters allow for each endpoint a selection of specific devices based on groups, rooms or functions of their capabilities. The API is built by the Spring Framework and its toolset. With this toolchain the API can be configured with a YAML file, specifying endpoints, filters and functions. The Framework, beside building the API, generates also client code for different environments to address the API. This helps front-end and application developers to ease the use of the API.

With this approach the conex.io project has built an API that abstracts smart home systems and their control from technology specific aspects and provides a REST-like API. This API allows filters to be applied and enables a capability-based communication with devices independent of their technology.

Acknowledgements

We would like to thank Axel Krüger and Alexander Schreiner for their support, technical help and continuous willingness to bring advances to the conex.io project.

8. References

[1] Chathura Withanage, “A comparison of the popular home automation technologies”, Dev. Pillar, Singapore Univ. of Technol. & Design, Singapore, 2014 IEEE Innovative Smart Grid Technologies – Asia, ISSN: 2378-8534

[2] Global eSustainability Initiative, “GeSI SMARTer2030: Enabling the low carbon economy in the information age,” UK, [Online], Available: http://smarter2030.gesi.org/, accessed 04.2016

[3] O.Drögehorn, J.Porras, F.Sangogboye, “Home Automation, using OpenSource to fulfill EU-Directives“, Proc. of the17th Int. Conf. on Internet Computing; ed. H.R. Arabnia, LasVegas Nevada, USA, July 2016

[4] Odyssee-Mure, “Energy Efficiency Trends in Buildings in the EU,” 2012. [Online]. Available:http://www.odyssee-mure.eu/publications/br /Buildingsbrochure-2012.pdf, accessed 02. March 201

[5] Enerdata, March 2015. [Online]. Available: http://www.indicators.odyssee-mure.eu/onlineindicators.html, accessed 04.2016

[6] S. Foster, A. Tramba and L. MacDonald, “ecoMOD:Analyzing Energy Efficiency in Affordable Housing,” Charlottesville, VA, 2007

[7] C. Wei and Y. Li, "Design of energy consumption monitoring and energy-saving management system of intelligent building based on the Internet of things," in Electronics, Communications and Control (ICECC), 2011 International Conference on, Zhejiang, 2011

[8] J. Skon, O. Kauhanen and M. Kolehmainen, “Energy consumption and air quality monitoring system,” Adelaide, SA, 2011

[9] C.-Y. Chen, Y.-P. Tsoul, S.-C. Liao and C.-T. Lin, “Implementing the design of smart home and achieving energy conservation,” Cardiff, Wales, 2009

[10] D.-M. Han and J.-H. Lim, “Smart home energy management system using IEEE 802.15.4 and zigbee,” Kongju, South Korea, 2010

[11] W. Granzer and W. Kastner, "Information modeling in heterogeneous building automation systems," in Factory Communication Systems (WFCS), 2012 9th IEEE International Workshop On, 2012, pp. 291-300

[12] T. Weng, A. Nwokafor and Y. Agarwal, "BuildingDepot 2.0: An integrated management system for building analysis and control," in Proceedings of the 5th ACM Workshop on Embedded Systems for Energy-Efficient Buildings, Roma, Italy, 2013, pp. 7:1-7:8

[13] The open source project FHEM, www.fhem.de, accessed 06.2017

[14] The REST-Cookbook, www.restcookbook.com, access 06.2017

[15] Jeremy Dorn, the web-based JSON-Editor, http://jeremydorn.com/json-editor/, access 06.2017

[16] YAML, another markup language, http://www.yaml.de/, accessed 06.2016

[17] The Swagger CodeGen, http://swagger.io/swagger-codegen/, accessed 06.2017

[18] Apache Foundation, the TomCat Project, http://tomcat.apache.org/, access 06.2017

[19] German Telekom Smart Home, www.smarthome.de, accessed 06.2017

[20] 5 open source home automation tools, https://opensource.com/life/16/3/5-open-source-home-automation-tools, accessed 08.2017

[21] conex.io project, prototype implementation of a REST-like API for home automation systems, https://github.com/philipptrenz/conex.io/, accessed 08.2017

[22] The conex.io project overview, project-descirption within the media-informatics projet landscape @ Harz University of Applied Sciences, http://www.medieninformatik.de/conex-io-entwickelt-appschnittstelle/, accessed 08.2017

[23] The IBM CIC Blog, https://mymagdeburgexperience.wordpress.com/2016/12/08/open-source-und-motivierte-studenten/, accessed 08.2017

[24] DHS-Computertechnik GmbH, Home Automation Services and Developments, http://www.dhs-computertechnik.de/home-automation, accessed 08.2017