One Remote Control to Command Them all! Building a Hypermedia API for ESP8266-Based Devices

Alexey Andreev, Daniil Garayzuev, Maxim Kolchin, Nikita Chursin, Ivan Shilin
ITMO University
Saint Petersburg, Russia
{a_andreev, garayzuev}@corp.ifmo.ru, kolchinmax@niuitmo.ru, {chursin.nikita, shilinivan}@corp.ifmo.ru

***Abstract*—Lack of commonly accepted standards for the connected devices' APIs caused the situation when each manufacturer creates its own mobile application to control their devices. We propose an approach to the design of a self-descriptive CoAP-based API using Hydra Core Vocabulary (http://hydra-cg.com) which allows to create an adaptive mobile application to control any device. In this paper we describe the approach, evaluate it on two exemplified devices build using ESP8266 Wi-Fi module and describe the architecture of a mobile application controlling these devices.**

I. Introduction

ESP8266 is a Wi-Fi module which become very popular because of its price that even more lowered the barrier to start building Internet of Things devices. Such module is a good way for hobbyists to create a useful device for their homes. But since humans still can't talk to devices in its language they need to use a more human-friendly interface. Therefore hobbyists also have to spend sometime to create a mobile (or web) interface for each device. We show in this work how to even more lower the barrier using simple guidelines to design the device API so any device could be commanded by a single mobile application.

To achieve the desired goal to control any connected device through a single user interface, two requirements should be met:

- First of all, a device and a commander should be able to communicate through a commonly supported way. That is to use the same communication protocol, e.g. HTTP, MQTT, CoAP, etc.
- Secondly, they should be able interpret each others' messages. It means that they need to use a common message model and vocabulary.

In this work we show how both requirements could be met by using a standard communication protocol, a standard messaging model and a commonly shared vocabulary. Once the requirements are met by a connected device, we're able to command it through a single user interface.

A. Related work

Heterogeneous devices are devices which use different network, communication and application protocols. The existence of such devices is mainly caused by the lack of commonly agreed standards. To command them, a commander should support all the protocols used by the devices at home. It can be achieved by using a middleware layer which implements all the protocols and provides a unified API to the devices. Such approach is employed in works [1], [2], where for each protocol a separate adapter is created.

In this paper instead of creating an adapter for each device in the commander, we assume that the devices use the same protocol, message model and vocabulary. This allows to get rid of a middleware between devices and the commander, but still use the same interface to command functionally different devices.

Our work is inspired by the approach proposed in [3] which describes a way for implementing a Web API following all REST principles [4]. This approach was already employed in the project called SmallHydra [9]. SmallHydra is a C++ library for the ESP8266 Wi-Fi module using Arduino libraries and an asynchronous HTTP server to provide a Hypermedia API.

B. Structure

Section II describes the requirements which our approach should met for building devices and a mobile application. Overview of the approach is presented in Section III. Section IV presents the architecture of a mobile application which is able to control any devices following the described requirements. Case study described in Section V presents examples ESP8266-based devices and a mobile application implementing the suggested approach. And Section VI concludes our work.

II. Requirements

The requirements we are providing for the device server and the client are needed for implementing dynamically generated user interface. This interface could visualise and interact with the device without additional changes in the client and the device based on reasoning of the semantically annotated description of the data.

Below are the requirements we impose on the API for connected devices:

R1 *Device discovery mechanism* is needed to provide a way to automatically discover available connected devices. It should be possible to find a new device at the local network by request or such device should notify the commander about itself.

R2 *Model–Instance representation.* Since devices has limited resources there should be a way to refer a commander to additional information on Internet or local network. This information should describe a device model, common characteristics, supported functionality, etc.

R3 *Authentication* support is required to authenticate access to device's data and controls. Request packet should contain hashed concatenated login and password strings. Basic authentication is needed to regulate access in secured network for administration and working with private resources.

R4 *Publish/Subscribe pattern* should be implemented to support the mechanism when a commander is notified about new data, such as new observation or state of the device's control. The pattern allows to subscribe for a resource and get updates once their appear.

R5 *On-demand configuration mode.* A possibility to configure a device through a public API is not always a good idea, because in that case it's vulnerable to different attacks. Therefore it'd be better to allow it be configured only if a user has a physical access to the device.

R6 *Sleeping nodes support:* is a requirement to support devices that could not be directly discovered at any time due to their sleep modes. This requirement makes possible to provide permanent access to impermanent resources via registration and updating device's data at the additional permanent proxy server.

III. Overview of the approach

Our approach is a set of guidelines that the developer of an ESP8266-based device should follow, so he or she could use the mobile application to command it.

A. Device software guidelines

To satisfy defined requirements The Constrained Application Protocol (CoAP) was selected [5]. CoAP is a specialised web transfer protocol working upon UDP (but could support TCP, SMS or any other channels and packets standards). CoAP is binary protocol and designed for use with constrained nodes and constrained networks in the Internet of Things. Observe option is standardised too and discovery and resource directory are in progress of standardisation, so draft version is used. CoAP is also using the RFC 6690 Constrained RESTful Environments (CoRE) Link format to describe the access to the resources.

To provide access to devices which are sleeping most of the time, several methods are provided. First of all, there is a configurable system parameter for every device, that could contain predefined system URI that could gather the data from the updating resources constantly and could be changed during initialisation or wake-up cycles. Secondly, support of the Resource Directory CoRE standard draft[10] is provided where Resource Directory resource could be registered at permanent client. Such resource directory is recognisable automatically in local network by any device and could register and update information about inpermanent devices to provide unified access to them.

Every device is executing CoAP server logic. CoAP server is building correct answers for the requests for some resources and also support subscribing to them and is recognisable in local network via GET-requests to "/.well-known/core"-resource.

Private resources requires basic authorisation option in normal mode. Fig. 1 represents interaction between device with the CoAP server and clients.

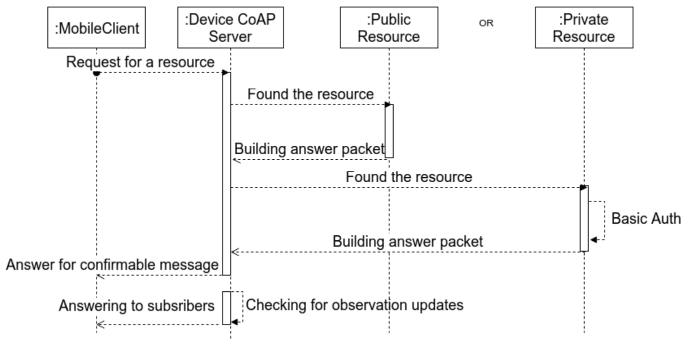

Fig. 1. Device's interaction diagram

This logic could also be represented upon the HTTP or some other protocol similar to SmallHydra project. But, according to Senior IT Architect at IBM India [11], HTTP for Internet of Things embedded devices is a lot slower, less reliable and uses more battery. CoAP fits the defined requirements best at the moment.

We have selected Arduino libraries for the ESP8266 platform, because they're compiling directly to binary firmware for the device with the native C++ SDK [13]. Prepared Arduino-based solution could be simply ported to other Arduino-compatible devices with enough resources, such as ESP32 or Arduino Nano and the communication modem could be changed too without changing other parts of the solution. Fig. 2 describes firmware model. CoAP Arduino server is implemented from scratch due to lack of functionality in other C-based implementations.

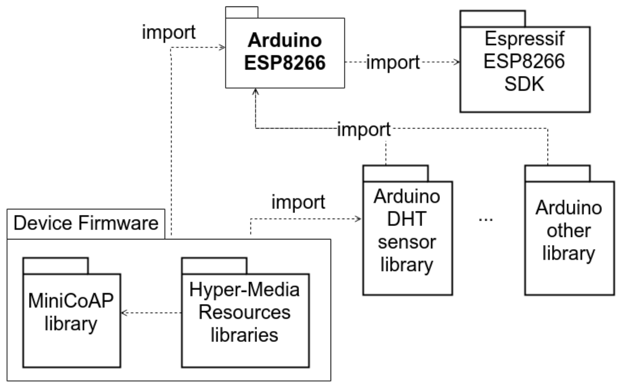

Fig. 2. Device's model diagram

Every resource of the device is independent and handled by the server. Server's implementation class diagram is represented at Fig. 3. MiniCoAP server is our CoAP server for Arduino-based devices with observe option and basic authentication support.

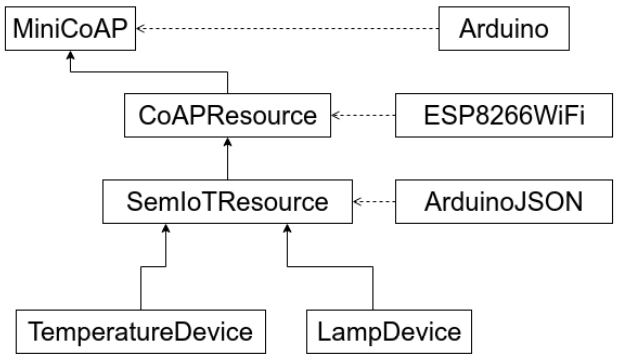

Fig. 3. Server's class diagram

According to the defined requirements, resources should be annotated and provide model-instance representation. Transfer protocol is not enough, so semantics is defined at the next chapter.

B. Device design guidelines

To met the requirement R5 the device should implement a way to physically switch to the configuration mode. Since ESP8266 module can work as a Wi-Fi node as well as a Wi-Fi access point, we suggest to use the access point mode as the configuration mode. The mode should be switched by pressing a hardware button on a device.

When a user presses the button, the device should switch to the Wi-Fi access point mode (in the documentation for ESP8266 it's called "SoftAP mode") and create it's own local network. Then a user should be able to connect to the network with a smart phone or computer and interact with the configuration API. The guidelines for the API is listed in Section III-D.

Algorithm 1 presents a code snippet that shows how to switch to the configuration mode.

Algorithm 1 Switching to the configuration mode

```
isButtonPressed = digitalRead(BUTTON PIN);
if (isButtonPressed) {
    if (WiFi.getMode() != WIFI AP) {
        WiFi.mode(WIFI_AP);
        WiFi.softAP(SOFT_AP_SSID, SOFT_AP_PASS);
    }
}
```

Others should have access only to the public API of a device. The public API provides access to device's measurements and controls. The guidelines are listed in Section III-C.

C. Device Public API Guidelines

Mobile application will be able to communicate with a device only if it describes itself. Such descriptions should include characteristics of a device (e.g. location, label, identifier, etc.) and operations it supports. Our approach is based on using JSON-LD [17] as the messaging model and Hydra Core [7], Schema [15] and partially Semantic Sensor Networks [8] as vocabularies for describing the APIs of ESP8266-devices.

The description of a device should consist of several parts:

- *The Well-Known Core* - it's the main resource (`/.well-known/core`) described in RFC5785 [16] specification. It lists all the resources supported by the API. The Device Resource should have property `rt` set to http://schema.org/EntryPoint which means that the resource is the entry point of the API.
- *API Documentation* - contains the definition of the device model, its supported properties and operations, types of supported links to other resources. To simplify the development of the API Documentation an external context were prepared, it available on https://w3id.org/semiot/device/commoncontext#. Examples of the documentation are in Algorithm 2 and 3.
- *Device Resource* - describes the device itself as an instance of the device model defined in the API Documentation. It may have links to actions and values. Examples are in Algorithm 4 and 5.
- *Value Resource* - describes the latest value provided by the device. It may be an observation, measurement, etc. It should be an observable resource.
- *Action Resource* - describes the result of an action performed by the device. E.g. switched on/off a lamp. It may support the `POST` requests to activate the action. It should be an observable resource.

API Documentation can be stored on the device or available on Internet, so the mobile application could download it. To define the API Documentation, Hydra Core and Schema.org vocabularies are used.

D. Device configuration API guidelines

The configuration API is similar to the public one, it reuses the API Documentation and has a single resource `/config` which should support `GET` and `PUT` operations. In Algorithm 6 the configuration of the Temperature Sensor is presented.

IV. ARCHITECTURE OF THE MOBILE APPLICATION

Architecture of the mobile application is based on Hydra API. The mobile application communicates with devices using CoAP protocol and gets a description of the devices from an external documents. This approach allows to change the information about device without interfering in his work, it is useful, for example, for extending of the supported languages in the description of the device or edit the available properties. Also this approach allows to create adaptive application that will change the displayed information depending on each device properties. It also simplifies configuring available device, for example, impose a the string length limit of the field or accepted characters range and so on without being attached to any particular field or type of the actual device.

The architecture of application was built by using this approach and is presented on Fig. 4. As can be seen from this figure, when new device is being added and configured, the device configuration module is refereed and this module invokes the configuration subsystem. This subsystem requests the configuration data from the device and builds page which

Algorithm 2 API Documentation of the Temperature Sensor API

```
 {
   ”@context”: [
       ”https://w3id.org/semiot/device/commoncontext#”,
       { ”doc”: ”http://external/doc#” }
   ],
   ”@id”: ”doc:ApiDocumentation”, 0
”@type”: ”ApiDocumentation”, 0:
”supportedClass”: [
       { ”@id”: ”doc:TempDevice”,
         ”subClassOf”: ”Device”,
         ”supportedProperty”: {
           ”property”: ”location”,
           ”writable”: ”true”,
           ”label”: ”Location”
         },
         ”supportedProperty”: {
           ”property”: ”label”,
           ”writable”: ”false”,
           ”label”: ”Label”
         },
         ”supportedOperation”: {
           ”method”: ”PUT”,
           ”expects”: ”doc:TempDevice”,
           ”returns”: ”doc:TempDevice”
         }
       },
       { ”@id”: ”doc:temperature”,
         ”@type”: ”Link”,
         ”range”: { ”@id”: ”doc:TemperatureValue”,
           ”subClassOf”: ”QuantitativeValue”,
           ”label”: ”Temperature (C)”
         }
       }
   ]
}
```

is displaying to the user. Displayed fields have a certain type, such as text or numeric, and the description of them. But it depends on the received configuration data. If any limiting data such as the maximum number of input characters or numeric range are specified, it is also taken into account in constructing these fields.

The data display subsystem is used to show the information from the device. It consists of a data acquisition module and building forms module for the final displaying to the user. Building forms module requests the description of the device that is located in the external document and pre-determines the type of data (observations or operations). Depending on the type invokes a special module for constructing the necessary forms. The required information for such a construction is also taken from the description of the device, on the basis of received information an application determines what data is used and the display format. For example, the possible ways to work with them (for example, where to send requests for changing the status of the device and which settings are necessary for the transmission) are determined for the operations. At the same time the data acquisition module determines the type of the device (active or sleeping) and

Algorithm 3 API Documentation of the Lamp API

```
 {
@context”: [
       ”https://w3id.org/semiot/device/commoncontext#”,
       { ”doc”: ”http://external/doc#” }
   ],
   ”@id”: ”doc:ApiDocumentation”,
@type”: ”ApiDocumentation”, 0:
”supportedClass”: [
       {”@id”: ”doc:LampDevice”,
         ”subClassOf”: ”Device”,
         ”supportedProperty”: {
           ”property”: ”location”,
           ”writable”: ”true”,
         ”label”: {”@value”: ”Location”, ”@language”: ”en”}
         },
         ”supportedProperty”: {
           ”property”: ”label”,
           ”writable”: ”false”,
           ”label”: {”@value”: ”Label”, ”@language”: ”en”}
         },
         ”supportedOperation”: {
           ”method”: ”PUT”,
           ”expects”: ”doc:LampDevice”,
           ”returns”: ”doc:LampDevice”
         }
       },
       {”@id”: ”doc:shineAction”,
         ”@type”: ”Link”,
         ”range”: ”ControlAction”,
         ”supportedOperation”: {
           ”method”: ”GET”, ”returns”: ”ControlAction”
         },
         ”supportedOperation”: {
           ”method”: ”POST”, ”expects”: ”doc:TurnOnAction”
         },
         ”supportedOperation”: {
           ”method”: ”POST”, ”expects”: ”doc:TurnOffAction”
         }
       },
       {”@id”: ”doc:TurnOnAction”,
         ”subClassOf”: ”ControlAction”,
         ”label”: {”@value”: ”Turn on”, ”@language”: ”en”},
         ”supportedProperty”: {
           ”@type”: ”PropertyValueSpecification”,
           ”property”: ”doc:brightness”,
             ”label”: {”@value”: ”Brightness”, ”@language”:
  ”en”},
           ”valueRequired”: ”false”,
           ”defaultValue”: ”50”,
           ”minValue”: ”0”,
           ”maxValue”: ”100”
         }
       },
       {”@id”: ”doc:TurnOffAction”,
         ”subClassOf”: ”ControlAction”,
         ”label”: {”@value”: ”Turn off”, ”@language”: ”en”}
       }
   ]
 }
```

Algorithm 4 Device Resource of the Temperature Sensor

```
{
  ”@context”: ”http://external/doc#”,
  ”@id”: ”coap://1.1.1.1/”,
  ”@type”: ”TempDevice”,
  ”identifier”: ”e9704”,
  ”label”: ”Temperature Device”,
  ”location”: {”@type”: ”Place”, ”label”: ”1010”},
”temperature”: ”/temperatureValue”
}
```

Algorithm 5 Device Resource of the Lamp

```
{
  ”@context”: ”http://external/doc#”,
  ”@id”: ”coap://1.1.1.1/”,
  ”@type”: ”LampDevice”,
  ”label”: {”@value”: ”Lamp #1010”, ”@language”: ”en”},
”identifier”: ”e9704”,
  ”location”: {”@type”: ”Place”, ”label”: ”1010”},
  ”shineAction”: ”/shine”
}
```

depending on these data requests information from the local store or send the request directly to the device.

The module for receiving data from the sleeping device is a service job and it’s run in the background. When the device wakes up, it transmits data with the state to the mobile application and this module, when the module has received it and saves it in the local store for future access.

For building a mobile application is based on the architecture that was described above, need to provide user interaction with the above modules, as well as to implement these modules. Processes were analysed for the design of subsystems and we compiled data flow diagrams.

The diagram which is presented on Fig. 5, shows the display of all available devices which are connected to current Wi-Fi network. Sleeping devices had transmitted data on this network are also displayed. For displaying all devices run a query to a local data store to get out the received data from the sleeping devices. Also broadcast CoAP request was sent to the network. All devices that are received this query return their id and label. Then a list of available devices are created and returned to the user.

Fig. 6 shows a diagram of adding a new device. To do this, the user connects to a device’s Wi-Fi network. Once

Algorithm 6 The configuration of the Temperature sensor

```
{
  ”@context”: ”http://external/doc#”,
”@id”: ”/config”,
  ”@type”: ”Configuration”,
  ”wifiName”: ”wifi-network”,
  ”wifiPassword”: ”password”,
  ”username”: ”superuser”,
  ”password”: ”strongpass”,
  ”deviceName”: ”My Device”
}
```

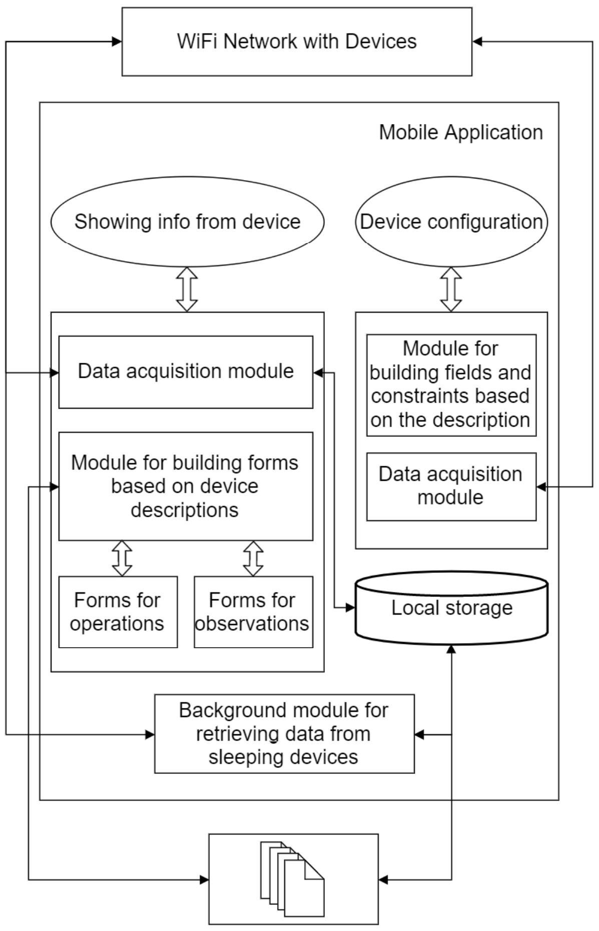
WiFi Network with Devices
Mobile Application
Showing info from device
Device configuration
Data acquisition module
Module for building fields and constraints based on the description
Module for building forms based on device descriptions
Data acquisition module
Forms for operations
Forms for observations
Local storage
Background module for retrieving data from sleeping devices

Fig. 4. Architecture of the mobile application

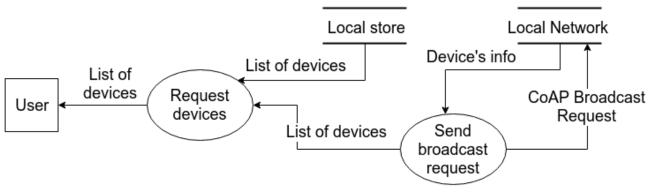

Fig. 5. Data Flow Diagram. Show list of devices

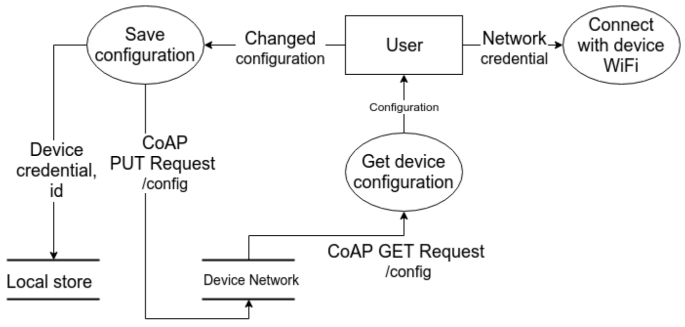

Fig. 6. Data Flow Diagram. Configuration

connected, the GET request was sent to the CoAP for obtaining configuration data from the device. The response to this query contains available for configuration data, such as network name and password which the device should be connected after it had been configured, the label of the device, and so on., as well as limiting the field data, for example, the maximum length of the string, and so on. Once the user page is generated with the configuration data, and the user can edit them. When the user saves them, then a part of the data from the device is saved locally (data such as the username and password for future access to a private device resources, its id) and sent PUT request with the data on the device.

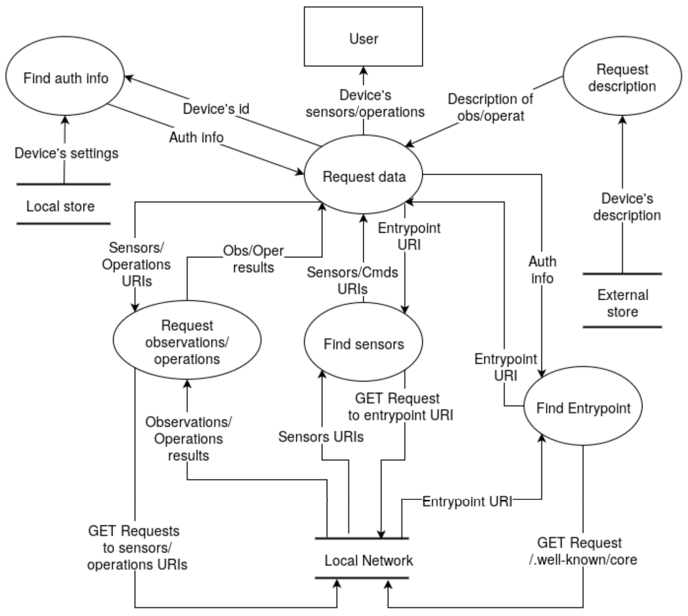

Fig. 7. Data Flow Diagram. Show device information

On Fig 7 presented a model that using for viewing an information from device. Page with such data, are generated based on the description of the device. The description is a response for request which is sent to an external document containing this description. Authentication data is requested from local storage for this device to be able to request information from private resources. After that device's entrypoint URI is requested by making a GET request to `/.well-known/core`. URI, having `http://schema.org/EntryPoint` type is a sought-for entrypoint. Next, the GET request is done to entrypoint URI, there are obtained data on the external URL of the document describing the relative address and device resources. The document with describing the device stores a description for all operations are possible with this device, and measured them evidence (if present). Further, the GET request is performed to CoAP resource. Response to this request contains information on the latest operation and/or the last observation. Since this resource is associated with the description of the device, the data associated with the mapping.

To perform operation on the device fields with property are generated based on description (if they are presented). After user starts a operation execution then the appropriate CoAP-method are sent.

V. CASE STUDY

Here we are describing specific devices and mobile application, corresponding provided approach and characterises details of the implementation.

We have implemented two independent separated devices: temperature sensor and bulb lamp actuator. Both of them are based on Espressif ESP8266 system on chip controller, which have a Wi-Fi modem.

Temperature device is based on ESP8266 Wi-Fi module and DHT22 digital temperature and humidity sensor. Bulb lamp actuator is based on ESP8266 Wi-Fi module and digital voltage relay. Temperature device is represented in Fig. 8.

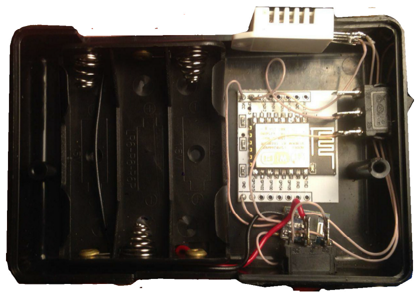

Fig. 8. Temperature ESP8266-based device

Every device is recognisable at the local network by the standard `/.well-known/core` resource. All available devices could be found and controlled by the client via multicast request or via recognisable server containing Resource Directory resource.

The GET multicast request with the `/.well-known/core?rt=core.rd` request URI message will not be ignored only by resource directory because it is qualified by the query string.

The resource type of the / root URI is marked as `http://schema.org/EntryPoint` at the `/.well-known/core` answer to provide Entry Point link for the client. Root resource contains main information about the device as a system. The example of the answer description is provided at Algorithms 5 and 4. PUT-request is also supported for configuration according to the defined requirements.

Root resource description directs us to the external documentation resource that could be shared between different devices. Examples of the answer for the GET request for the temperature and lamp devices are provided in Algorithms 2 and 3.

All sensitive resources' methods could be private. Configuration resource `/config` is private too. It only available for GET and PUT requests in configuration mode or in normal mode with the correct basic authentication credentials.

Additional CoAP option with number 40 is provided during the research. The option payload should contain SHA-1 hash from concatenated login and password strings to provide authorisation. Config resource request provided on Fig. 6. Software Wi-Fi Access point's SSID for initial configuration is based on the chip identifier and the password is standard and the same for the devices. CoAP headers is minimal and JSON payload could be compressed with the MessagePack solution [12] for about 50% to maximise the transfer rate.

Mobile application is based on the Android SDK. The solution is available at public project repository [14].

VI. Conclusion

In this paper we showed how to build a mobile application which dynamically generates its user interface based on the self-descriptive API of connected devices. The guidelines for building connected devices with self-descriptive API are presented.

In case study we applied the guidelines in two devices and found them useful and appropriate for these types of devices. If believe that the similar approach can be applied for MQTT and similar protocols.

Acknowledgment

This work has been supported by the Ministry of Education and Science of the Russian Federation (Grant #RFMEFI57514X0101).

References

[1] J.E. Kim, G. Boulos, J. Yackovich "Seamless Integration of Heterogeneous Devices and Access Control in Smart Homes" in *Proc. of 8th International Conference on Intelligent Environments (IE)*, 2012, pp. 206-213

[2] P. Desai, A. Sheth, P. Anantharam "Semantic Gateway as a Service Architecture for IoT Interoperability" in *Proc of IEEE International Conference on Mobile Services*, 2015, pp. 313–319

[3] M. Lanthaler, and C. Gütl, "Hydra: A Vocabulary for Hypermedia-Driven Web APIs", *LDOW*, vol.996, 2013

[4] R.T. Fielding "Architectural Styles and the Design of Network-based Software Architectures", PhD thesis, University of California, 2000

[5] Bormann, C., Castellani, A. P., Shelby, Z. "CoAP: An application protocol for billions of tiny internet nodes", *IEEE Internet Computing*, vol.16(2), 62, 2012.

[6] Ankolekar, Anupriya, et al. "DAML-S: Web service description for the semantic web." International Semantic Web Conference. *Springer Berlin Heidelberg*, 2002.

[7] Lanthaler M. "Creating 3rd Generation Web APIs with Hydra" *Proceedings of the 22nd International Conference on World Wide Web*, 2013, pp. 3538.

[8] M. Compton, P. Barnaghi, L. Bermudez, R. Garca-Castro, O. Corcho, S. Cox, et al., 'The SSN ontology of the W3C semantic sensor network incubator group', *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 17, December 2012, pp. 25-32.

[9] GitHub, A small Hydra library for the ESP8266 using the ESPAsyncWebServer, Web: https://github.com/bergos/smallhydra.

[10] IETF Tools, CoRE Resource Directory Draft 8, Web: https://tools.ietf.org/html/draft-ietf-core-resource-directory-08.

[11] IBM developerWorks, Why HTTP is not enough for the Internet of Things, Web: https://www.ibm.com/developerworks/community/blogs/mobileblog/entry/why_http_is_not_enough_for_the_internet_of_things.

[12] MessagePack, It's like JSON. But fast and small. Arduino C implementation, Web: http://msgpack.org/.

[13] GitHub, ESP8266 core for Arduino, Web: https://github.com/esp8266/Arduino.

[14] GitHub, SemIoT project, Web:https://github.com/semiotproject

[15] Schema.org, Web: http://schema.org

[16] RFC 5785 - Defining Well-Known Uniform Resource Identifiers (URIs), Web: https://tools.ietf.org/html/rfc5785

[17] JSON-LD 1.1 - A JSON-based Serialization for Linked Data, Web: http://json-ld.org/spec/latest/json-ld/