--- T-H-I-N-X ---

A „Thing Server” concept for the Internet of Things (IoT)

Abstract resource nodes model, abstract architecture model, concrete implementation mappings and IoT ecosystem usage patterns

White Paper

Version 1.5 (May 2015)

ANDREAS RIEGG

Change log

Version	Date	Author	Changes
1.0	May 21st, 2014	Andreas Riegg	Initial version
1.1	July 15th, 2014	Andreas Riegg	Added content and modified model
1.2	Dec 22nd, 2014	Andreas Riegg	Completed HTTP/REST Mapping, added chapter on IoT ecosystems, added reference to RDF/OWL, changed overall chapter structure
1.3	Feb 4th, 2015	Andreas Riegg	Added chapter on IoT architectures and references to WebIOPi architecture and Weaved add-on, updated chapter sequence, some typos and small text updates and additions
1.4	Mar 20th, 2015	Andreas Riegg	Added <property> to the TouchPoint resource node and updated REST mapping and JSON mapping for that, shortened {val,ue(s)} to {value}. Added Swagger spec for API. Unified API proposals for /run and /configure. Added Swagger UI and swagger.ed screenshots.
1.5	May 11th, 2015	Andreas Riegg	Renamed <property> to <aspect> due to work with RDF to avoid confusion with RDF modelling "properties". Added geometrical, chemical and biological sensors to concept and REST mapping. Renamed <physicalUnit> to <unit> to be consistent with more types of sensors. Added references to more M2M protocols.

Content

Licensing & Copyright

This paper is ruled by the following Copyright statement:

The source code in this paper is ruled by the following Copyright statement:

Preface

This paper is derived from ideas introduced by WebIOPi from Eric Ptak [WEBIOPI_2014]. It extends those ideas and abstracts them to a concept at a higher level of generalization. Parts of this work are also inspired by the PhD thesis of Dominique Guinard [GUIN_2011]. Basic concepts of the RESTful concept are described in the PhD thesis of Roy Fielding [FIELD_2000]. Some concepts are similar to the Device Tree [DTREE_2014] concept (nodes and sub nodes plus attached attributes) but also at a much higher abstraction level and far away from all the hardware implementation details covered by Device Tree. It's much more an outside consumer's view towards hardware. Citation in this white paper is simplified compared to scientific papers. To avoid cluttering the text with inline citations you will find all interesting references just as a link list in the "References" chapter.

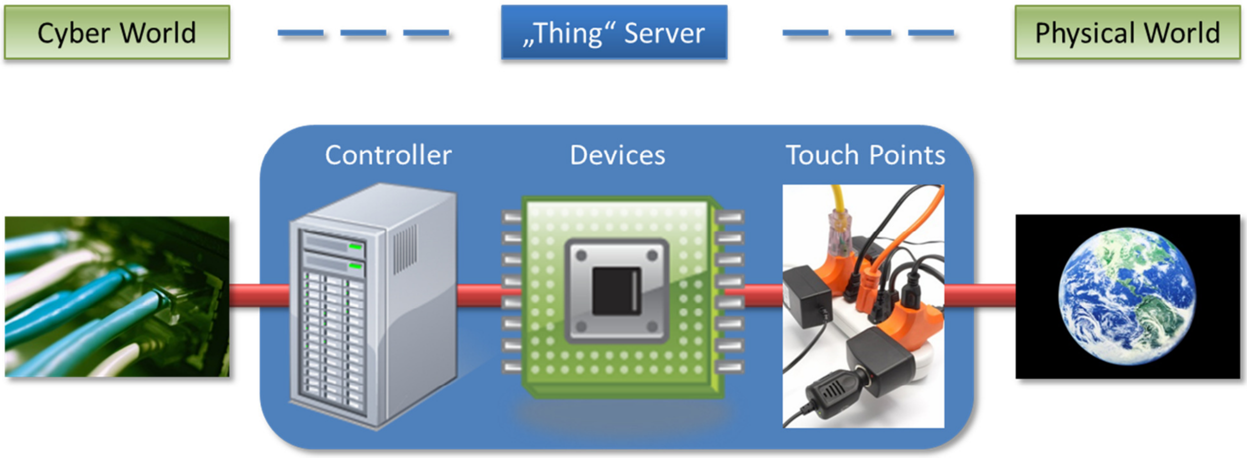
Text in ***bold italics*** is preliminary and to be done or to be completed.

Physical Architecture

The generic physical architecture consists of a “thing” that spans a bridge between the cyber world (aka “Internet”) and the physical world (aka “Reality”). This pattern is also called a “Cyber-Physical System”. The “Cyber World” and the “Physical World” are tightly connected:

- Changes in the physical world (e.g. a rising temperature) are mirrored into the cyber world (e.g. the new temperature is displayed on a web page instantly),
- and … vice versa … some action in the cyber world (e.g. a button “ON” is pressed within a mobile app) leads to a reaction in the real world (e.g. a light goes on).

More technically, this “Thing” Server concept looks like in the following picture:



The connection to the cyber world on the left most part of the picture is typically implemented using standard Web-Technology like Wifi/Ethernet and the TCP/IP protocol stack, but much more technical possibilities exist especially in situations where very low energy consumption is required. The processing required at the cyber world communication endpoint in the left part of the picture is done by some kind of controller. The technical implementation of this controller ranges from very small one-chip microcontrollers to full-size data center machines but this does not matter with respect to the model explained here.

The connection to the physical world on the right most part of the picture is realized by

- **Sensors** that touch the real world in some way and measure any effects reaching from simple things like temperature and pressure up to complex electromagnetic wave patterns and even biological indicators.
- **Actors** that touch the real world in some other way and have the ability to influence any effect reaching from switching an electrical value from being low to high up to sending complex electromagnetic wave patterns.

Some touch points are mainly unidirectional (e.g. most sensors), some others are bi-directional (e.g. most basic digital I/O ports).

To summarize, inside the “Thing” Server are three main parts that play an important role in the resources model:

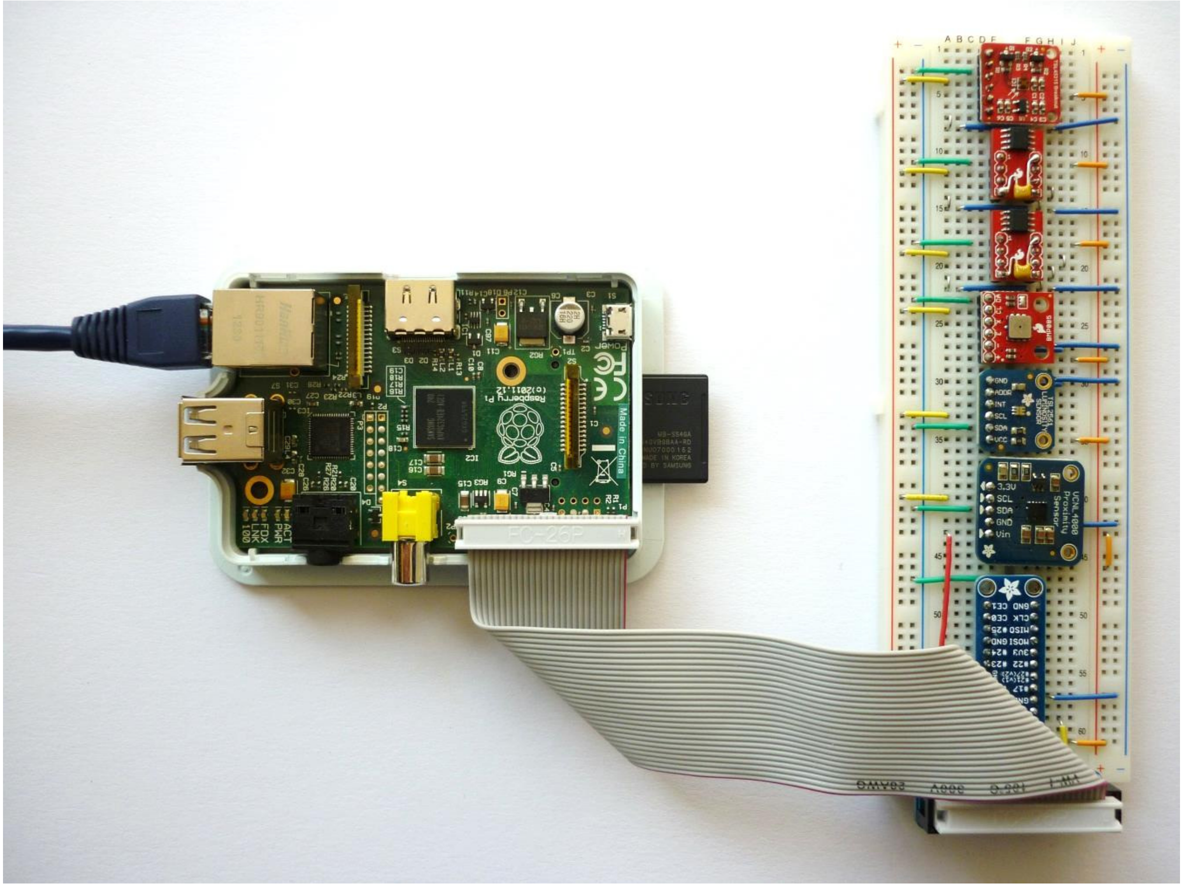
- The **Controller** that realizes the complete communication endpoint to and from the cyber world; it also controls the devices below and has the ability to be the runtime platform for additional executable “Thing” features.
- A number of **Touch Points** that implement the concrete connection to the physical world along with the effects that they tamper.

- And a set of **Devices** that transform effects captured at the **Touch Points** into some digital representation and vice versa.

Typically, one single Thing Server has one controller, some devices and more touch points than devices. If a device is connected to a set of touch points handling the same effect (e.g. a number of analog I/O ports) this is often implemented by calling this “channels”.

In order to provide ways of well-defined security at the cyber world endpoint, some kind of authentication, authorization and very often also encryption is used. This is typically implemented as a part of the controller functionality.

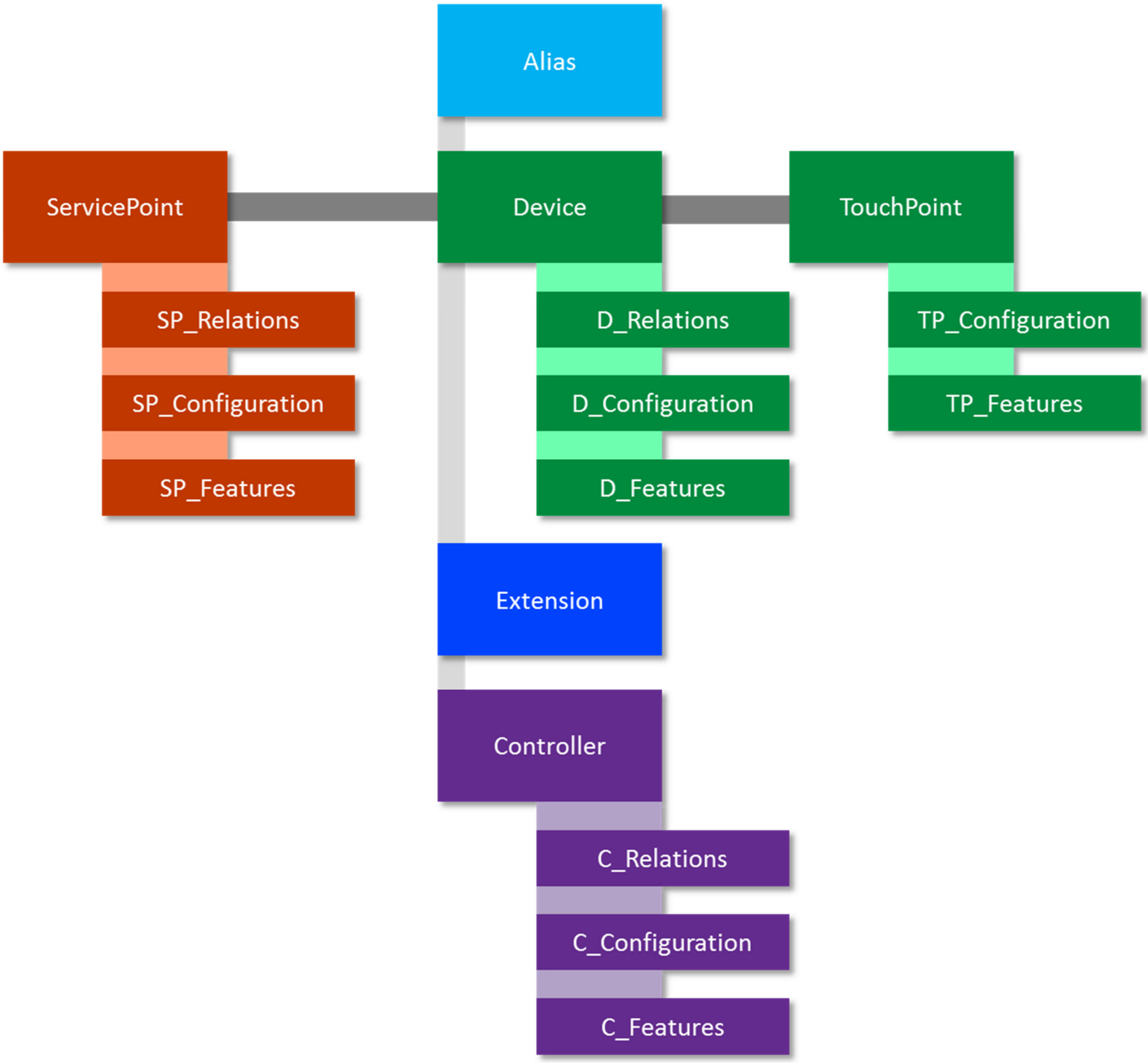
The following picture shows a concrete example of a Thing Server realized with a Raspberry Pi computer [RASP_2014], a breadboard, some sensors and a wired network connection.



Abstract Resource Nodes Domain Model

Overview

Transforming the physical architecture of the previous chapter into an abstract resource node domain model results in the following picture:



The big rectangles represent the primary resource "nodes" of the model. The main information and influence stream is simply abstracted to the primary logical node chain

ServicePoint <-> Device <-> TouchPoint

that is marked by the dark link in the picture above. This means that there is an "influence" flow from the ServicePoint via the Devices to the TouchPoints and also in the backwards direction.

The small rectangles attached to the primary nodes represent separated properties of them. This separation will be explained a bit later.

ServicePoint

The node ServicePoint summarizes all aspects that have to do with the reachability of the Things-Server from the cyberspace. You can look at the ServicePoint as being the out- and inbound facade of the Things-Server towards cyberspace. However, this facade is a logical construct and has nothing to do with the internal implementation or runtime architecture of this interface. But it is assumed that all communication that touches the Things-Server is managed through this node.

TouchPoint

A TouchPoint is, as its name implies, the point where the Things-Server has any kind of connection to the physical (real) world. At the very basic level this consists of a real atomic connection where some material of the Things-Server has a junction to any other material of the real world. If the “signal” or “force” flow along this junction is outbound, this is called an actor, if the flow is inbound, this is called a sensor.

Device

Devices are nodes that implement technically the information link between the TouchPoints and the ServicePoints. In order to do this they realize the “signal” transformation between the physical world and the cyber world. As the cyber world consists logically only of digital information and the physical world only contains physical effects, the Devices are the place where the conversion between those two worlds takes place. Technically spoken this is done at the very basic layer via

- digital-to-analog (DAC) and analog-to-digital (ADC) conversions and
- hardware elements that translate physical effects into the change of an electrical voltage and/or current and vice versa.

In real implementations often the TouchPoints are an intrinsic hardware feature of the devices (e.g. the case of a temperature sensor chip is the place where the environment temperature gets being touched and measured). However, in the logical model discussed here this does not play any role as chips exist that sense more than one physical effect in parallel (e.g. the Bosch pressure sensors that also measure temperature) and the logical model wants to see this as two separate influence streams to become independent from any concrete chip implementations. And, this also helps in cases where one single device (chip) has a big set of different TouchPoints like a complete microcontroller SoC with its bunch of native analog and digital I/O channels.

The three main nodes are accompanied by three assisting nodes. Their purpose is to be the anchor for additional aspects of the resource model that lie outside the main influence flow but still play an important role in the overall model.

Alias

Sometimes the designer of things wants to hide (at least optional) some of the complex technical details of the internal thing structure. This can be achieved by using some kind of “alias” (abbreviation) concepts that allow building a logical simple facade interface.

Extension

Sometimes it is necessary to have some additional functionality implemented in the thing to allow local processing. This avoids time- and energy-consuming round-trips from the thing back and forth to the cyber world (e.g. some kind of central control server). This functionality is implemented also within the Things-Server and thus called a server “extension”. In standard web architectures this is often called a servlet (Java) or a module (Apache HTTP). Extensions need some runtime container where they can be executed on request. In principal, more than one runtime container for different flavors of extensions can be provided by a Things-Server.

Controller

So far we have seen mostly logical resource nodes - but where does the real computation happen? The boundary between the Controller and the Devices is a bit fluent and not as sharp as the pictures show. In brief, the Controller is the complete runtime infrastructure inside the Things-Server that makes all the other elements being able to run or provide their services. Typically a Controller comprises

- an operating system,
- a network stack within this OS that provides web connectivity and all the needed security features,
- a hardware drivers environment for the Devices,
- the runtime container(s) for extensions,
- some storage to save digital assets like web pages and/or sensor/actor data.

A (traditional) example for such a controller implementation is the popular Arduino family of single -board computers [ARDU_2014]. Typical examples for more up-to-date controllers are the Linux-based single-board computers like Raspberry Pi [RASP_2014] or Beagle boards [BEAGLE_2014]. Very recent also brand new special cases exist where parts of the Controller live at runtime in the cloud like with the Electric Imp [EIMP_2014].

_Relations, _Configuration and _Features

The last elements of the abstract model are the X_Relations, X_Configuration and X_Features sub nodes. All properties of the primary nodes with a more static nature are gathered in the “X_Configuration” sub nodes; all properties with a more dynamic nature are gathered in the “X_Features” sub nodes. Information that links the primary nodes between each other is contained in the X_Relations sub nodes. This resembles a bit the concept of Object-oriented Development with their objects (to **be** something), internal members (to **know** something) and internal methods (to **do** something) but at a higher abstraction level:

- **Being** something has to do with the type and existence (and thus identity) of a node and
- **Knowing** something denotes all internal values that are stored permanent inside the nodes and the other nodes it is related to
- **Doing** something allows these nodes to change something inside them or also outside

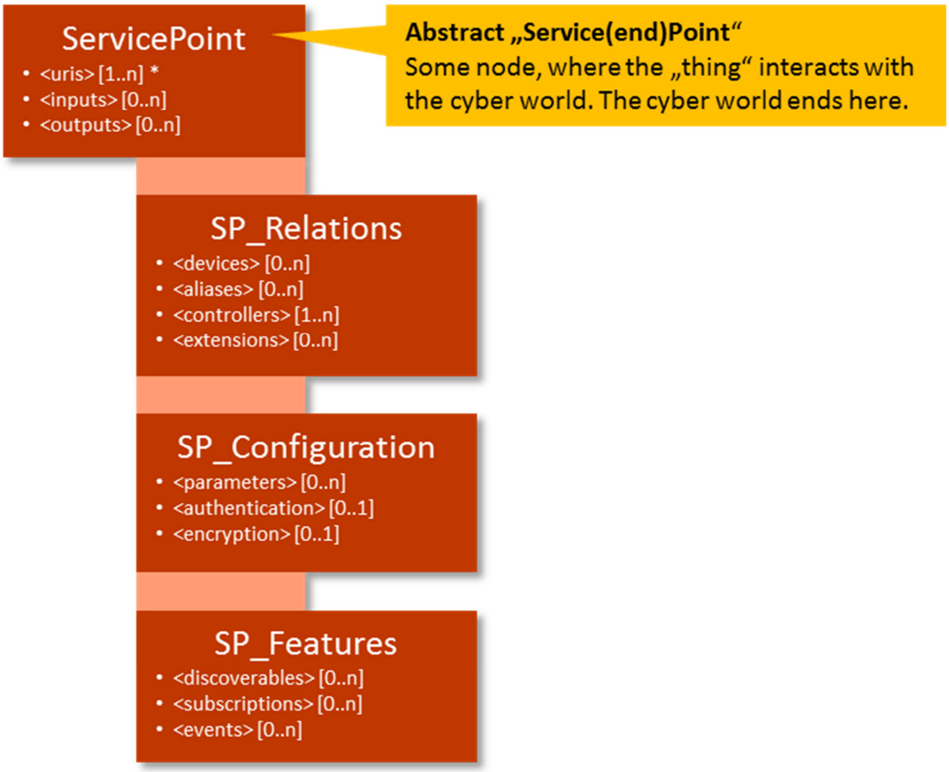
What **being**, **knowing** and **doing** means for the individual nodes is explained now in detail by expanding the six nodes and their sub nodes. The detailed information is modeled by using “slots” within the nodes. A slot means a particular piece of information but does not make any assumptions about its implementation. Very often, in simple cases the content of a slot is just a textual string or basic number. In slightly more complex cases it may be some kind of object that can be easy represented as a JSON or XML structure with moderate deepness or a list of strings or numbers. In very complex cases the content of a slot may be a list of objects or a self-contained net of objects.

Anyway, the concrete representation and implementation of these slots is out of scope for now. It will play some role when concrete mappings are described in chapters later in this document.

The slots of the resources that have an “*” at the end are used for the identification of these resource nodes. Navigating between the nodes can be handled using the slots within the sub nodes X_Relations and taking the identifying slots of the resource nodes as keys.

ServicePoint

The following picture shows the details of a “ServicePoint”.



<uris>

As a ServicePoint is the connection to the cyber world it needs some kind of address scheme for this universe. This is modeled by the <uris> slot which is a very common concept for this purpose. One concrete example (subclass) for a URI is the URL http://your.domain:12345. In such a case some protocol (HTTP), some address (your.domain) and some port (12345) is specified. As a Thing Server may have different ServicePoints at the same time (e.g. it may be reachable via HTTP and CoAP simultaneously) there can exist [1..n] URI’s. Minimum number is 1 to have at least one possibility to reach the Things Server at all.

<inputs>

If the cyber world wants to provide some information or influence flow into the ServicePoint, this is modeled by an input slot. Inputs are optional and can be multiples so their cardinality is [0..n]. Examples for inputs are commands that are sent to the Things-Server that change some state of a TouchPoint which is connected via a device (e.g. like a light bulb that goes on). Another input of a more indirect nature may be a device parameter that can be set from the outside like the conversion rate for an A/D converter chip.

<outputs>

If the cyber world wants to consume some information or influence flow out of the ServicePoint, this is modeled by an output slot. Outputs are also optional and can be multiples so their cardinality is [0..n]. Examples for outputs are state values or state updates from TouchPoints that ca be accessed from outside. Outputs can also provide “out-of-band” information about the Things-Server himself like the number of threads its controller is running currently.

SP_Relations

<devices>

This slot allows navigating from the ServicePoint to its connected devices. This slot has a cardinality of [0..n] as ServicePoints without Devices can exist (e.g. by bearing only some Extensions that may realize a kind of gateway).

<aliases>

This slot allows navigating from the ServicePoint to its known Aliases. As Aliases are optional this slot has a cardinality of [0..n].

<controllers>

This slot allows navigating from the ServicePoint to its implementing Controllers. As a ServicePoint without any Controllers makes not much sense this slot has a cardinality of [1..n]. In most cases just one Controller will be in place, but multi-core variants and setups that use e.g. 2-out-of-3 redundant architectures may be existent in the future.

<extensions>

This slot allows navigating from the ServicePoint to its known Extensions. As Extensions are optional this slot has a cardinality of [0..n].

SP_Configuration

<parameters>

A ServicePoint can have some parameters that tailor its behavior and which are relevant to the cyber world. These parameters are contained in this slot. Examples for such parameters are service quality it provides, the geolocation of a ServicePoint or if it uses authentication and encryption or not. As parameters are optional this slot has a cardinality of [0..n].

<authentication>

If authentication is activated for a ServicePoint this slot contains the information to handle this. Examples for authentication information may be the authentication modes the ServicePoint supports like Basic Auth or OAuth2. As authentication is optional this slot has a cardinality of [0..n].

<encryption>

If encryption is activated for a ServicePoint this slot contains the information to handle this. Examples for encryption information may be the type encryption algorithms it supports (e.g. AES 512). As encryption is optional this slot has a cardinality of [0..n].

SP_Features

<discoverables>

A Things-Server can respond to requests that try to discover some of the services it offers. These requests will hit the ServicePoint. This slot models the information that can be provided. An example for a discovering request is the call to the path /.well-known by the CoAP discovery protocol. As discovery mechanisms are optional this slot has a cardinality of [0..n].

<subscriptions>

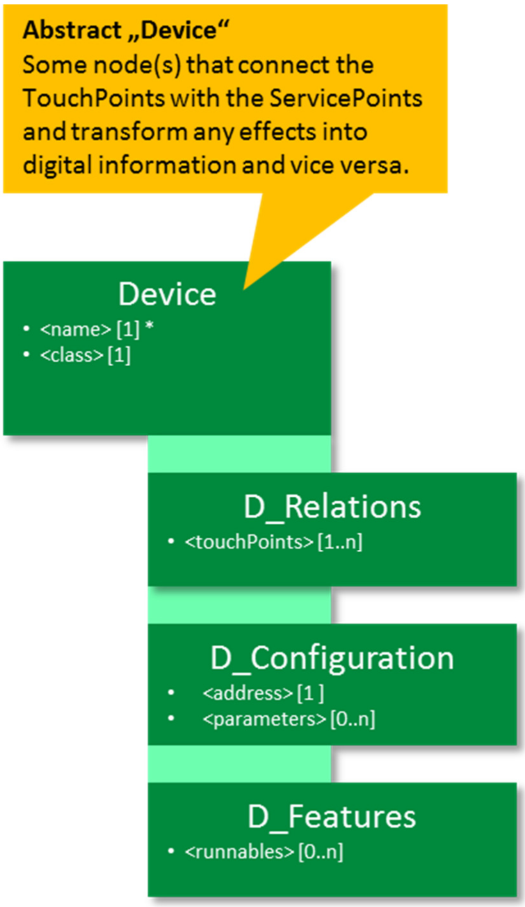
Sometimes publish/subscribe mechanisms are used within a M2M communication scenario. A Things-Server can offer such a service at its ServicePoint or subscribe itself at some other ServicePoint. This slot holds all active subscribers from outside and all subscriptions the Things-Server has made external. The publish/subscribe mechanism of CoAP is an example for this functionality. As publish/subscribe mechanisms are optional as they are not provided by every M2M communication protocol this slot has a cardinality of [0..n].

<events>

Within the ongoing operation of a Things-Server events can occur like an interrupt that has been detected. This slot gives access to these events. However, no assumptions are made about the mechanism how these events are advertised as this is dependent on the concrete mappings and their technical possibilities. As event detecting mechanisms are optional this slot has a cardinality of [0..n].

Device

The following picture shows the details of a “Device”.



A “Device” has the following slots:

<name>

This is the name of a Device that identifies it. Every device must have a name so this slot has the cardinality [1].

<class>

Every Device must have some kind of type that describes its nature and allows creating a set of Devices with different names but the same type. In analogy to Object Oriented Technology as mentioned above this type of Device is called a <class>. This mandatory slot takes this kind of information. Using the concept of class here does not necessarily mean that class inheritance has to be in place for every Device driver implementation. However, most modern implementation languages allow the usage of class inheritance and this will make the code for the drivers more compact through reuse.

D_Relations

<touchPoints>

From the overview picture it is shown that every Device implements one or more TouchPoints. Navigating from Devices to their TouchPoints is managed by this slot. As a Device without any TouchPoints makes not much sense this slot has a cardinality of [1..n].

D_Configuration

<address>

One of the most important configuration information of a Device is its address. The address here means all parts of an address on an abstract level. The concrete layout of addresses differs very much and is hardware dependent. In the case of the I2C bus, this would be the number of the I2C bus plus the I2C slave address of a chip on this bus. The address is mandatory so this slot has a cardinality of [1]. In the case that a device has more than one address (like the PCA9685 with its “all call” and “sub call” slave addresses) it may be necessary to create a unique device instance for each address.

<parameters>

Devices can have some additional parameters like timing constants, bit resolutions or reference voltages. This slot holds that information and has a cardinality of [0..n].

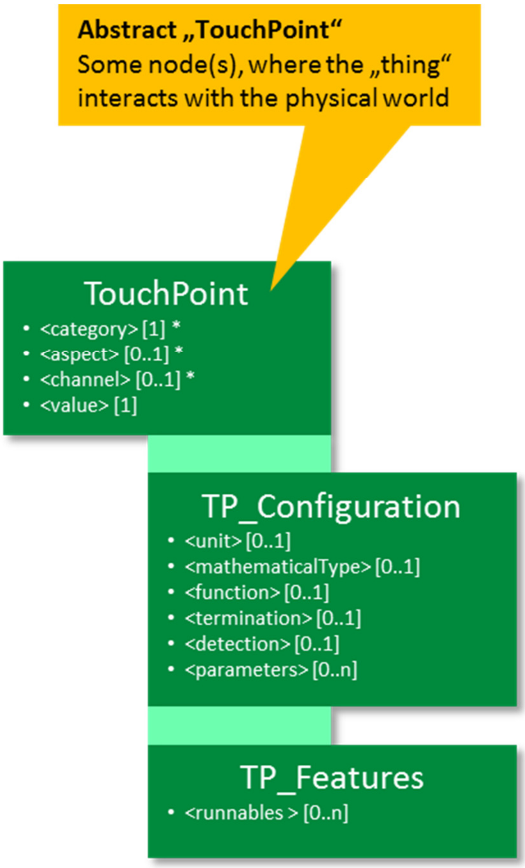
D_Features

<runnables>

Devices can have some functions that can be triggered from outside. An example for such a function is a calibration routine that can be started from time to time to make sure the device delivers correct values. Another example is the possibility to send a Device to a sleep mode and reawake it later when it’s needed again. This slot holds that information and has a cardinality of [0..n] as not every device has such functions.

TouchPoint

The following picture shows the details of a “TouchPoint”.



<category>

Each TouchPoint must have a <category> that denotes which kind of physical effect it is dedicated to. In order to be consistent every TouchPoint must have exactly one category so its cardinality is [1]. For <category> the following possibilities exist:

- Static digital I/O -> “digital”
- Pulsed digital I/O (one output variant also known as pulse width modulation “PWM”) -> “bitstream”
- Static analog I/O -> “analog”
- Modulated analog I/O (known as classic AM and FM modulation) -> “waveform”
- Sensors for any physical effect (e.g. temperature), any geometrical entity (e.g. position), any chemical effect (e.g. molarity) or any biological indicator (e.g. glucose) -> ”sensor”,

By adding some generalization other kinds of logical TouchPoints also exist:

- Time information (e.g. from a RTC chip) -> “clock”
- Unique identifications (e.g. ID chips, built-in serial numbers, GUIDs) -> “id”
- Interfaces (e.g. Serial) -> “interface”
- Memory (e.g. RAM, EEPROM) -> “memory”
- Hardware buses (e.g. I2C, SPI, 1-wire) ->“bus”

It is possible that a single Device has a set of TouchPoints that belong to different categories. This may be a RTC chip (-> category “clock”) that measures also the temperature (-> category “sensor”) (e.g. the DS3231 is such a chip) or a hardware extension shield that carries a set of ports (e.g. digital and analog I/O chips).

<aspect>

If a TouchPoint is the member of a list of TouchPoints for one Device that have all the same category but touch e.g. different physical effects, an additional aspect identifier is necessary. Especially for sensors a large number of aspects exist.

For <aspect> the following possibilities exist (list is not complete):

- Geometrical entities -> “position”, “distance”, ... (many possibilities)
- Physical effects -> “temperature”, “pressure”, “current”, ... (dozens of possibilities)
- Chemical effects -> “molarity”, “volumetric-flow”, “ph-value”, ... (many possibilities)
- Biological indicators -> “glucose”, “cholesterol”, ... (dozens of possibilities)
- Direction indicator ->”input”, “output”, “inout”
- Special signal forms ->”pwm”, “am”, “fm”
- Sub data slots->”time”, “date”, “datetime”

<channel>

If a TouchPoint is the member of a list of TouchPoints for one Device that have all the same category and aspect, an additional channel identifier is necessary. Typically these lists are sorted and the channel identifiers are integer numbers starting at 0 or 1 like the n A/D channels for a typical A/D chip. But this is not mandatory from an abstract point of view. Cases can exist where a <channel> may indicate some kind of different measuring way like a light sensor that can have visible and infrared light measuring channels. And this also includes the case where a channel also uses some kind of logical sub categorization where this makes sense like e.g. “16/infrared”.

For <channel> the following possibilities exist:

- (Sequential) Number -> “5”

- Any identifying string -> “a”, “port-b”
- Geometrical reference identifier ->”x”, “y”, “z”
- Physical reference identifier ->”sea”, “infrared”, “relative”
- Chemical reference identifiers -> “rvs”, “ps”, “os” (some examples for pH value measurements)
- Biological reference identifiers -> “fbs”, “ogtt”, “ivgtt”, “hba1c” (some examples for blood glucose tests)

At the bottom line, it is sufficient if all TouchPoints of one Device, which belong to the same category and optional to the same aspect, have different channel identifiers.

If a Device has

- only one TouchPoint or
- multiple TouchPoints of different categories or
- multiple TouchPoints of different category/aspect pairs,

then the <channel> slot can be empty.

For this reason, all slots sequenced together (<category>, <aspect> and <channel>) are used for the identity of TouchPoints including the case where <aspect> and/or <channel> is empty. The sequential combination of category, aspect and channel must be unique for each Device. Putting all this together leads to a cardinality of [0..1] for the <aspect>and <channel> slot.

<value>

This is the digitized value of the physical effect that is measured by this TouchPoint. If it has not been measured before the first access, it may be null. In simple cases, this value may just be a Boolean (0/1), Integer, Floating-Point or short String. In complex cases, the value may be a long String being some kind of text or even a binary object representing some blob media elements.

TP_Configuration

<unit>

Sometimes a TouchPoint (most notably a sensor) can offer its results in different physical, chemical, etc. units. To handle this, the slot <unit> is used. An example would be a temperature sensor that can offer its values in the units Celsius, Kelvin or Fahrenheit. As optional item it has the cardinality of [0..1].

<mathematicalType>

Sometimes a TouchPoint can handle its values in different mathematical types like integer, float or boolean etc. To handle this, the slot <mathematicalType> is used. An example would be an A/D converter device that can offer its values in the units Integer (being the raw A/D conversion value) and Float (being the raw A/D conversion value divided by the maximum A/D conversion value). As optional item it has the cardinality of [0..1].

For < mathematicalType > the following possibilities exist:

- Integer -> “INTEGER”
- Decimal -> “FLOAT”
- Boolean -> “BOOL”
- Binary -> “BINARY”
- N-ary -> “OCT”, “HEX”
- Date -> “DATE”
- Time -> “TIME”

- String -> “TEXT”

<function>

Sometimes a TouchPoint can be set to different operating modes. An example for such a behavior is a digital I/O port that can be configured alternatively as input, output or bidirectional. This aspect is covered by the slot <function>. As optional item it has the cardinality of [0..1].

<termination>

Sometimes a TouchPoint (most notably for digital I/O devices) can have a termination towards some defined signal level. Technically this is realized by pull-up or pull-down resistors. For some chips, these resistors can be set from remote via commands. This configuration option is covered by the <termination> slot. As optional item it has the cardinality of [0..1].

<detection>

Sometimes a TouchPoint (most notably for digital I/O devices) can have a kind of edge detection mechanism and generate an interrupt in some way. For some chips, this detection can be set from remote via commands. This configuration option is covered by the <detection> slot. As optional item it has the cardinality of [0..1].

<parameters>

Like Devices also TouchPoints can have individual additional parameters that are covered here. An example for such a parameter is the outside reference temperature (sensor) used to calculate the pressure at sea level for an inside pressure sensor. As optional item it has the cardinality of [0..n].

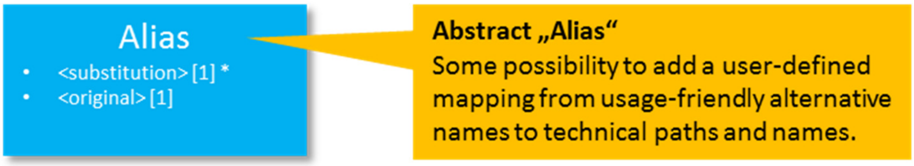
TP_Features

<runnables>

Like Devices also TouchPoints can have some functions that can be triggered from outside. An example for such a function is a bit sequence that is being output on a digital output channel. This slot holds that kind of information and has a cardinality of [0..n] .

Alias

The following picture shows the details of an “Alias”.


Alias
<substitution> [1] *
<original> [1]
Abstract „Alias“
Some possibility to add a user-defined mapping from usage-friendly alternative names to technical paths and names.

<substitution>

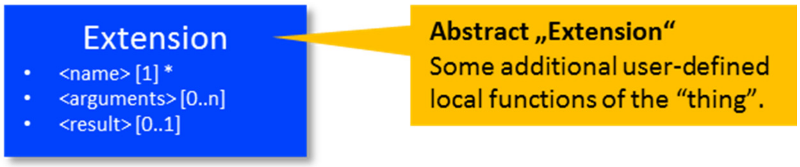
This is the new name (or resource path) and it has a cardinality of [1] as exact one substitution is possible per Alias. The <substitution> is the identifying slot of an Alias. Technically, when getting a service request to the new name, it is completely substituted by the original name. In practice, the substitutions are either much shorter that the possibly long names resulting from the hardware configuration. Or they use usage-oriented words to have better names that give hints about their purpose. Aliases can also be used to hide some internal resource structures to the outside world. Changing the original name but keeping the substitution name can allow modifying some TouchPoint settings without affecting the calls from outside. This is something very similar to the link concept within UNIX.

<original>

This is the original resource name resulting from the technical path derived from the relations between the Devices and their TouchPoints. It has a cardinality of [1] as exact one name translation is possible for each Alias. In principal more than one Alias can exist for the same original name as this is still logically consistent. The practical benefit of doing so is however limited.

Extension

The following picture shows the details of an “Extension”.



As mentioned, Extensions allow adding additional functionality to the Things-Server. They can run inn their own processes and/or threads if desired and possible in the runtime environment of the Things-Server.

An “Extension” has the following slots.

<name>

This is the calling signature of the Extension. Its concrete textual layout is somewhat dependent on the restrictions and conventions of the implementing language.

<arguments>

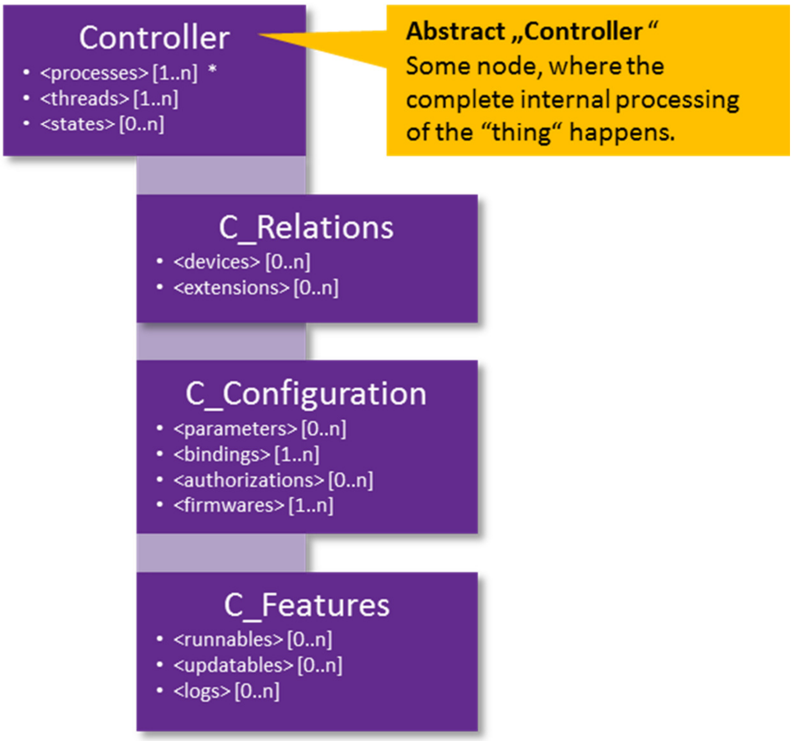
An Extension can have [0..n] arguments that either have a specific sequence or a naming convention for named arguments or both (as in Python). Arguments themselves can be very simple objects like strings or numbers up to complex object models like a complete object tree marshaled to e.g. JSON or XML.

<result>

An Extension can have [0..1] result. For the type of results the same applies as for arguments.

Controller

The following picture shows the details of a "Controller".



A "Controller" has the following slots.

<processes>

Within a Controller the computing tasks are implemented. The aspect of a running computing function is modeled by the process information within this slot. The minimum number of processes is 1, otherwise nothing would happen. In most cases more than one process exists within a controller (e.g. the drivers process and the communication process) so this slot has a cardinality of [1..n].

<threads>

To improve performance and quasi parallel computing mechanisms very often light-weight processes called threads are used. The minimum number of threads is 1 which is just the main thread of the possibly single process. In most cases more than one thread exists within a controller so this slot has a cardinality of [1..n].

<states>

Attached to the process(es) and thread(s) are informations that hold some data about those. These are called states and are modeled with this slot. States are optional so this slot has a cardinality of [0..n].

C_Relations

<devices>

This slot allows navigating from the Controller to its Devices. This slot has a cardinality of [0..n].

<extensions>

This slot allows navigating from the Controller to its Extensions. This slot has a cardinality of [0..n].

C_Configuration

<parameters>

Controllers can have some parameters that can be changed. Examples for such parameters are the amount of memory used for some processes and threads or the paths used to provide static data like web pages.

<bindings>

In order to realize that the Controller functionality which is executed is connected to the outside it is necessary to bind the Controller process(es) to some interfaces that can be reached via the URIs at the ServicePoint. As it is possible that one Controller serves more than one URI (e.g. one for a HTTP binding and another for a CoAP binding) this slot has a cardinality of [1..n].

<authorizations>

Accompanied with the <authentications> of the ServicePoint are authorization rules. As these rules are tightly coupled to the available resources inside the Things-Server and as they are implemented within the Controller functionality they are put into this slot. As authorization is optional this slot has a cardinality of [0..n].

<firmwares>

The code that runs within the controller can be seen as being the firmware of a Things-Server. More than one logical part of a firmware may exist so this slot has a cardinality of [1..n].

C_Features

<runnables>

Within a Controller some functions that can be activated exist. They are modeled with this slot. Examples for such runnables at the Controller level are the typical setup() and loop() parts of an Arduino-like Controller code. In contrast to Extensions which are user-defined, they live predefined (maybe with empty bodies) at the controller level. Another example would be the possibility to trigger a garbage collection for a virtual machine that runs inside the Controller to do some memory housekeeping when appropriate. And, finally, creating and deleting Devices at runtime would also be a runnable feature of a Controller. As runnables are optional this slot has a cardinality of [0..n].

<updatables>

Sometimes parts or all of the controller software can be updated. This is covered by this slot with a cardinality of [0..n].

<logs>

In order to be able to follow the actions of a Controller from an operational view the usage of logs is very common. This slot covers this aspect with a cardinality of [0..n].

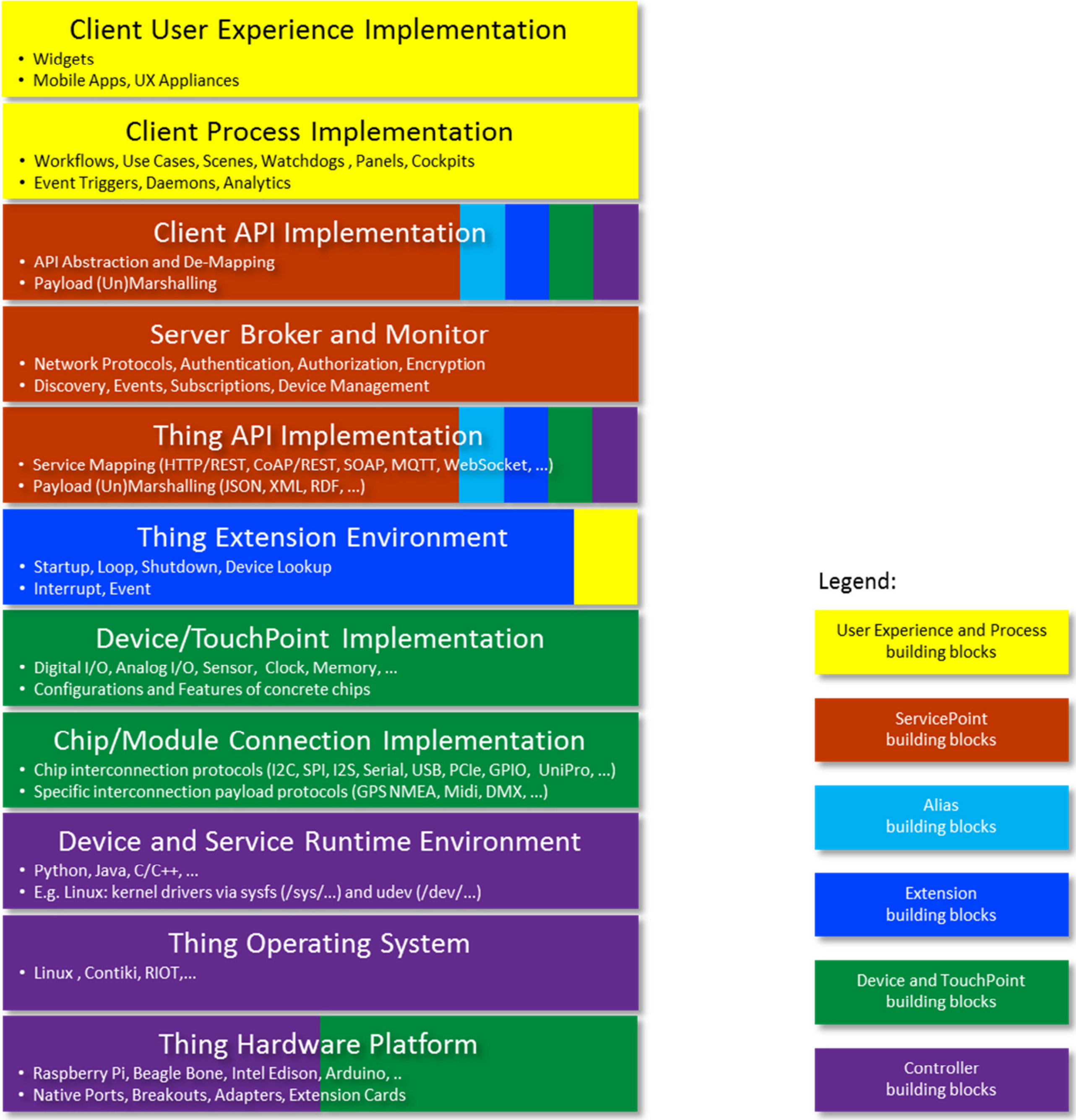
The following chapters now look at possibilities how this resource model can be used when concrete web interfacing technologies are used to communicate with the Things-Server

Where applicable, the corresponding mapping rules to the different concepts and/or languages are described in detail.

Generic Architecture Models for “Thing-Server” and “Thing-Clients” Infrastructures

Basic/single-tier implementation architecture

The following picture shows a layered architecture model that compiles all components for a Thing-Server and a Thing-Client into one abstract structure:


Client User Experience Implementation
• Widgets
• Mobile Apps, UX Appliances
Client Process Implementation
• Workflows, Use Cases, Scenes, Watchdogs , Panels, Cockpits
• Event Triggers, Daemons, Analytics
Client API Implementation
• API Abstraction and De-Mapping
• Payload (Un)Marshalling
Server Broker and Monitor
• Network Protocols, Authentication, Authorization, Encryption
• Discovery, Events, Subscriptions, Device Management
Thing API Implementation
• Service Mapping (HTTP/REST, CoAP/REST, SOAP, MQTT, WebSocket, ...)
• Payload (Un)Marshalling (JSON, XML, RDF, ...)
Thing Extension Environment
• Startup, Loop, Shutdown, Device Lookup
• Interrupt, Event
Device/TouchPoint Implementation
• Digital I/O, Analog I/O, Sensor, Clock, Memory, ...
• Configurations and Features of concrete chips
Chip/Module Connection Implementation
• Chip interconnection protocols (I2C, SPI, I2S, Serial, USB, PCIe, GPIO, UniPro, ...)
• Specific interconnection payload protocols (GPS NMEA, Midi, DMX, ...)
Device and Service Runtime Environment
• Python, Java, C/C++, ...
• E.g. Linux: kernel drivers via sysfs (/sys/...) and udev (/dev/...)
Thing Operating System
• Linux , Contiki, RIOT,...
Thing Hardware Platform
• Raspberry Pi, Beagle Bone, Intel Edison, Arduino, ..
• Native Ports, Breakouts, Adapters, Extension Cards
Legend:
User Experience and Process building blocks
ServicePoint building blocks
Alias building blocks
Extension building blocks
Device and TouchPoint building blocks
Controller building blocks

This implementation architecture stacks all building blocks of the resource model (see legend) extended by some blocks that represent the user “frontend” and process control blocks (see legend, yellow boxes). This model does not represent any kind of exact layered called-by communication model. It more stacks the building blocks together in the sense that each layer is “using” the layer below it or “living” inside it. The additional text in the boxes gives examples for what this building block stands for. It is meant to give hints and explanations, it is not meant to be complete.

In principle, this kind of architecture can be implemented running all parts on a single physical device. This would be the case when the whole stack were implemented on a single Raspberry Pi and the end user would use this from a monitor and keyboard/mouse directly attached to this individual Raspberry Pi.

However, in practice this would be only useful for e.g. testing purposes. More common, the full monolithic stack from above will be split between different devices that are connected via some kind of wired or wireless network. These scenarios will be explained in the following chapters.

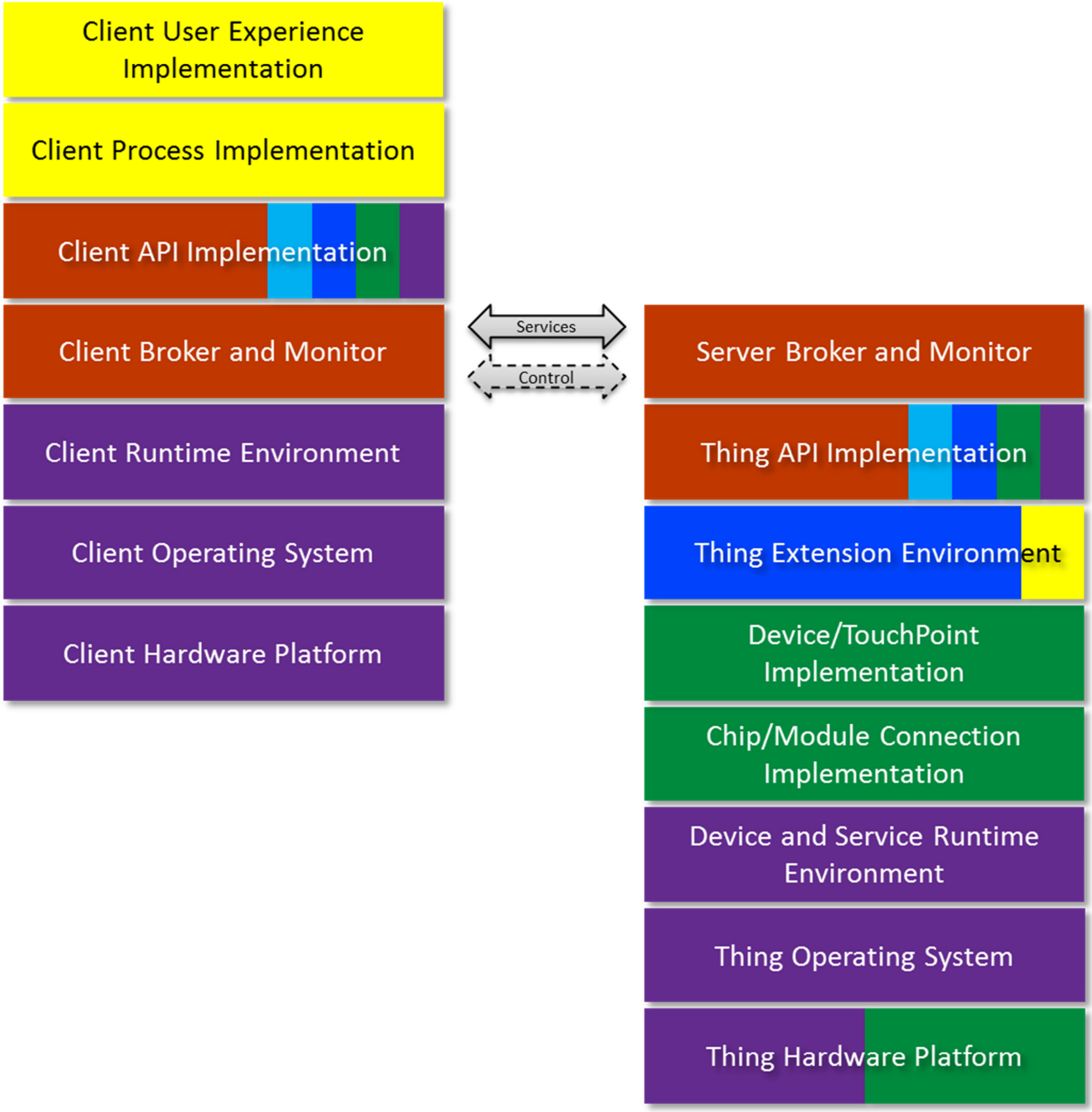
2-tier implementation architecture

In simple cases, one of the most common used architecture patterns is the 2-tier (or classical client/server) architecture pattern. Here, some (smaller) part is running on one device that is directly exposed to the end user (”client”). All the (bigger) remainder parts are running on another device that sits somewhere near the things that have to be connected to the cyber world (“server”). Between the both sits the network connection which has basically two kinds of channels:

- a “service” channel that is used for standard communication and
- a “control” channel that is used for special-purpose communication.

For the physical communication, both channels may use the same connection or use separate communication means. Even completely out-of-band communication for the control channel (e.g. for performance or security reasons) is sometimes used.

The following picture illustrates the 2-tier architecture:


Client User Experience Implementation
Client Process Implementation
Client API Implementation
Client Broker and Monitor
Client Runtime Environment
Client Operating System
Client Hardware Platform
Services
Control
Server Broker and Monitor
Thing API Implementation
Thing Extension Environment
Device/TouchPoint Implementation
Chip/Module Connection Implementation
Device and Service Runtime Environment
Thing Operating System
Thing Hardware Platform

The single-tier architecture is split within the Broker and Monitor layer into a client part and a server part plus a connection between them. As the separate client needs also to live on some kind of device, some client blocks for runtime environment, operating system and hardware platform are added. It's obvious that an unlimited number of clients can be connected to a single thing server. Problems may occur only when too much clients try to concurrently access the server or when different clients maintain simultaneous connections that interfere with each other by sending contradictory messages.

A typical example for 2-tier architecture would be a Raspberry Pi than hosts the complete server part and runs headless (no keyboard and monitor is attached) and some kind of PC or Smartphone that accesses this server from its browser.

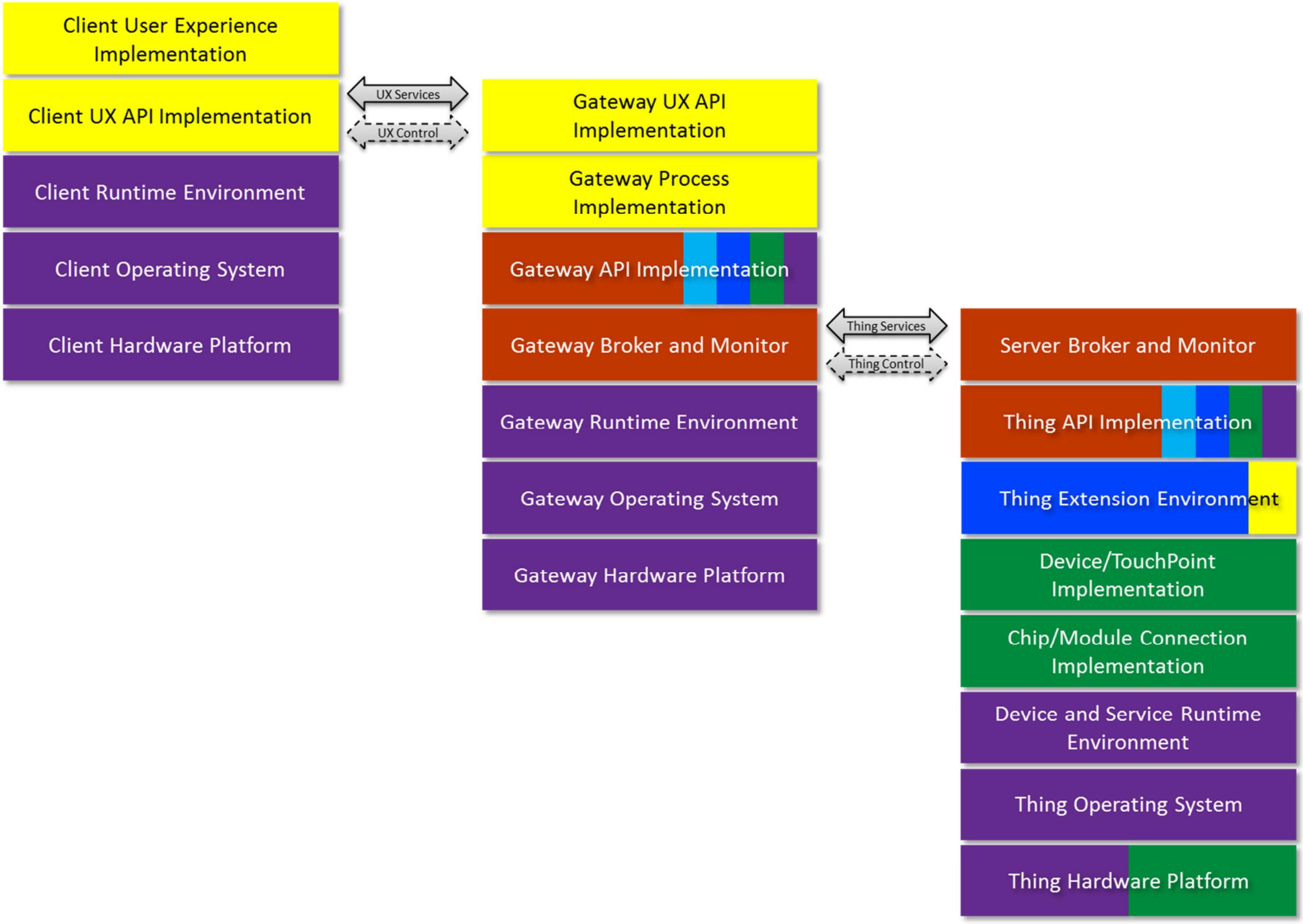
3-tier implementation architecture

In enhanced/professional cases the most common architecture that is used is the 3-tier architecture pattern. Here, some (probably small) part is running on the device that is directly exposed to the end user (”client”). More or less it's only the part that is needed to provide some kind of basic user experience and an API that connects this user experience to the second part, which is the gateway in the middle between the client and the server.

The main purpose of the gateway is to provide more complex process implementations for e.g. multiple clients or some kind of mashups/portals and connect this to the things server backend again. The gateway also contains a part that implements the counterpart for the user experience API and that communicates via a UX service and UX control channel with the client. As with the 2-tier architecture, the broker and monitor layer is split, now into a gateway and server part. Also, the same rules for the service and control channel apply.

All the remainder "thing" parts are running on a third device that sits like in the 2-tier variant somewhere near the things that have to be connected to the cyber world.

The following picture illustrates the 3-tier architecture:


Client User Experience Implementation
Client UX API Implementation
Client Runtime Environment
Client Operating System
Client Hardware Platform
UX Services
UX Control
Gateway UX API Implementation
Gateway Process Implementation
Gateway API Implementation
Gateway Broker and Monitor
Gateway Runtime Environment
Gateway Operating System
Gateway Hardware Platform
Thing Services
Thing Control
Server Broker and Monitor
Thing API Implementation
Thing Extension Environment
Device/TouchPoint Implementation
Chip/Module Connection Implementation
Device and Service Runtime Environment
Thing Operating System
Thing Hardware Platform

It is important to note that now we have some kind of communication split. The interface/API between the gateway and the thing server now uses the API that covers the thing service functions. In many cases, this will be the same interface that is used within the 2-tier architecture. For this reason, the thing server will be mostly unaware if it is used in a 2-tier or 3-tier architecture. A small exception occurs for the User Experience and Process building blocks, these may be missing or just be unnecessary in most 3-tier scenarios as those functions can be completely implemented within the gateway.

The interface between the gateway and the client however now uses some other kind of API that handles UX services and UX control. As this API covers different functions it will be in most cases look different from the thing server API. It may even be the case that for different clients, different UX APIs can be used (and fully hiding this from the thing server). Plus, it is also possible to have multiple thing servers with different thing service APIs being connected to the same gateway.

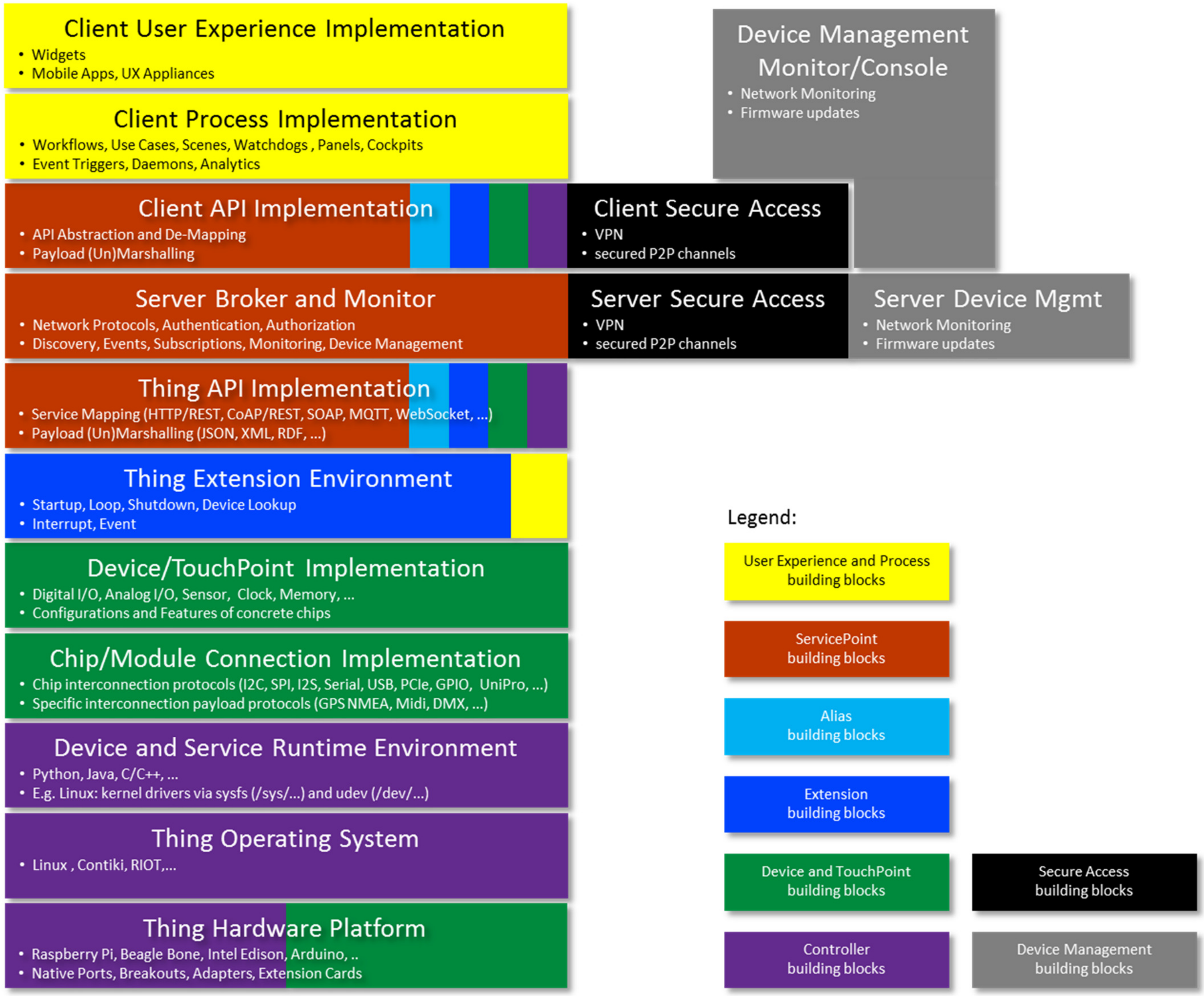
In other words, the big advantage of a 3-tier architecture is the possibility to integrate and decouple a set of different clients and different thing servers in an overall IoT solution scenario using this kind of gateway architecture. Most of the current “professional”-grade solutions from IoT software component libraries (e.g. OpenHAB, Eclipse Smarthome, ...) support such an architecture scenario just out of the box, more – they regard it as their preferred implementation pattern. It’s also worth to note that the upcoming IoT Mashup solutions and Microservice-based architectures will also use this kind of 3-tier approach.

A simple example for a 3-tier architecture would be

- a number of Raspberry Pi devices that host the thing server part sitting next to the TouchPoints that they have to physical connect to (e.g. one of them in each room of your house) and again run headless (no keyboard and monitor is attached)
- another Raspberry Pi (or some more capable hardware device like a Beagle Bone or any kind of Linux PC) that hosts the gateway part and runs maybe also headless
- and a set of PCs and/or smartphones/tablets that access this gateway from their browsers or rich/native clients apps installed on them.

Basic/single-tier implementation architecture with security and device management extensions

The basic architecture can be enhanced with a number of security and device management features. The following illustration shows how this can be achieved:



Client User Experience Implementation
- Widgets
- Mobile Apps, UX Appliances

Client Process Implementation
- Workflows, Use Cases, Scenes, Watchdogs , Panels, Cockpits
- Event Triggers, Daemons, Analytics

Client API Implementation
- API Abstraction and De-Mapping
- Payload (Un)Marshalling

Server Broker and Monitor
- Network Protocols, Authentication, Authorization
- Discovery, Events, Subscriptions, Monitoring, Device Management

Thing API Implementation
- Service Mapping (HTTP/REST, CoAP/REST, SOAP, MQTT, WebSocket, ...)
- Payload (Un)Marshalling (JSON, XML, RDF, ...)

Thing Extension Environment
- Startup, Loop, Shutdown, Device Lookup
- Interrupt, Event

Device/TouchPoint Implementation
- Digital I/O, Analog I/O, Sensor, Clock, Memory, ...
- Configurations and Features of concrete chips

Chip/Module Connection Implementation
- Chip interconnection protocols (I2C, SPI, I2S, Serial, USB, PCIe, GPIO, UniPro, ...)
- Specific interconnection payload protocols (GPS NMEA, Midi, DMX, ...)

Device and Service Runtime Environment
- Python, Java, C/C++, ...
- E.g. Linux: kernel drivers via sysfs (/sys/...) and udev (/dev/...)

Thing Operating System
- Linux , Contiki, RIOT,...

Thing Hardware Platform
- Raspberry Pi, Beagle Bone, Intel Edison, Arduino, ..
- Native Ports, Breakouts, Adapters, Extension Cards

Device Management Monitor/Console
- Network Monitoring
- Firmware updates

Client Secure Access
- VPN
- secured P2P channels

Server Secure Access
- VPN
- secured P2P channels

Server Device Mgmt
- Network Monitoring
- Firmware updates

Legend:

User Experience and Process building blocks

ServicePoint building blocks

Alias building blocks

Extension building blocks

Device and TouchPoint building blocks

Secure Access building blocks

Controller building blocks

Device Management building blocks

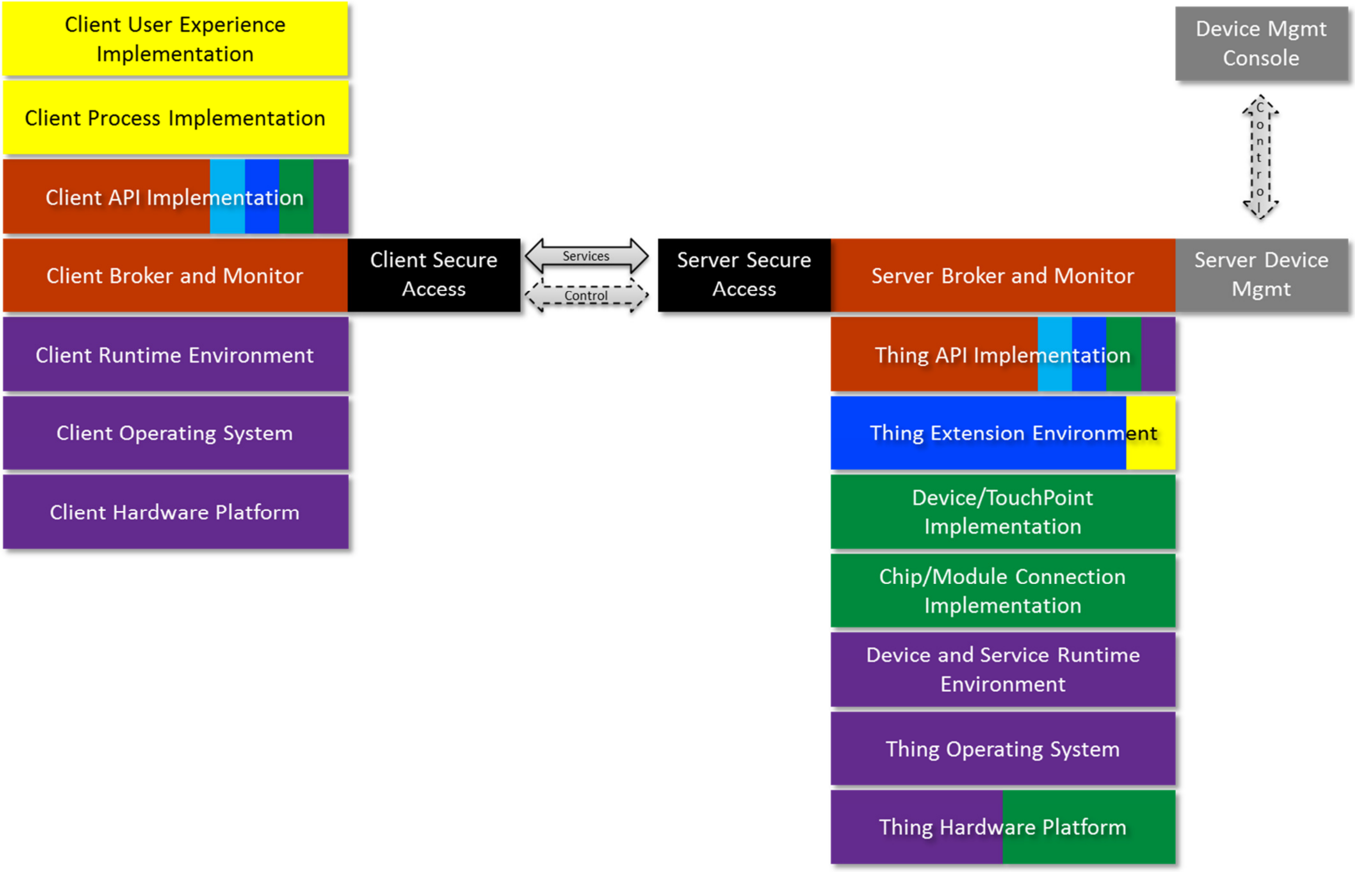
Mainly both additional building blocks (Secure Access in black color, Device Management in grey color) are directly attached to the Broker and API building blocks. Like for some other blocks, also a server and a client part exists.

The Secure Access blocks are added to impose some additional security features on the client connection. Additionally means here that the “standard” security measures (e.g. using https instead of http) are considered to be too weak or too hard to manage. This may apply for the service and the control channel simultaneously or for just one of them. The pair of client and server Secure Access blocks ensures in this context some kind of highly secured point-to-point connection. If security related management features are also needed they are considered to be an intrinsic part of the two building blocks.

The Device Management blocks are added to add some additional device management features to the whole IoT infrastructure. This is especially important when a large number of IoT devices are part of an IoT solution. While the Device Management server part sits on the top level of the thing server, the standard thing client does not have any connection to this. In typical solution architectures, some kind of separate device management “console” or “monitor” is handling this kind of functionality. In principal, the device management blocks may use the secure access blocks or they may use them not. This mainly depends on the overall architecture of the device management solution which is not part of this document.

2-tier implementation architecture with security and device management extensions

Like the base architecture also the 2-tier architecture can be enhanced with a number of security and device management features. The following illustration shows how this can be achieved:

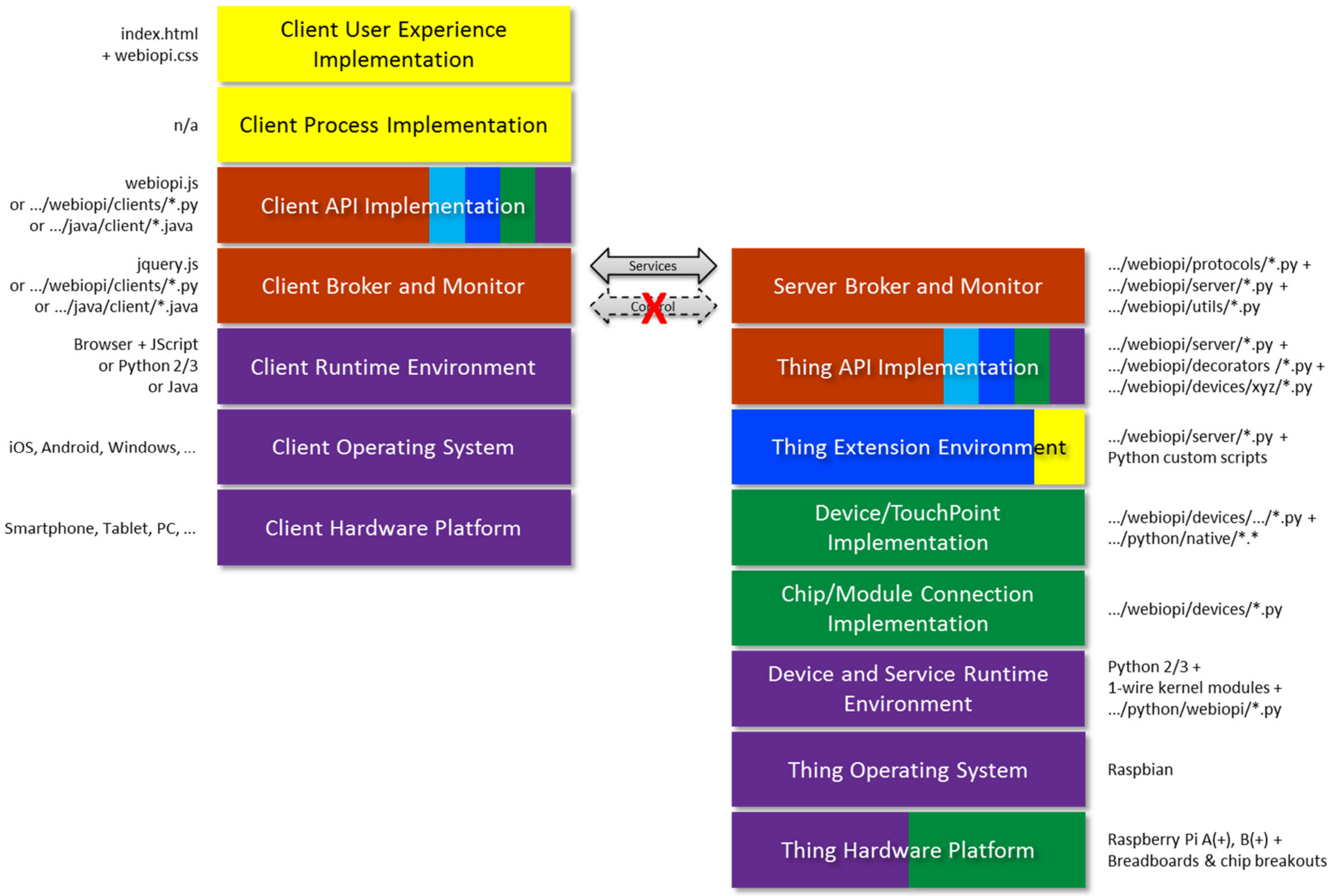

Client User Experience Implementation
Client Process Implementation
Client API Implementation
Client Broker and Monitor
Client Runtime Environment
Client Operating System
Client Hardware Platform
Client Secure Access
Services
Control
Server Secure Access
Server Broker and Monitor
Server Device Mgmt
Device Mgmt Console
Control
Thing API Implementation
Thing Extension Environment
Device/TouchPoint Implementation
Chip/Module Connection Implementation
Device and Service Runtime Environment
Thing Operating System
Thing Hardware Platform

In this case, the secure access is only used for the clients while the device management just uses the "normal" connection (as it may sit only within the internal already secure network segments).

Obviously, also the 3-tier architecture can be enhanced with these extensions, but this follows the same principles as the 2-tier architecture so this is not shown explicitly.

2-tier implementation architecture for WebIOPi

As mentioned in the introduction, WebIOPi is an implementation of the resource model. While 1-tier usage of it is possible (using the "localhost/127.0.0.1" network interface), the typical usage pattern is 2-tier. The following picture illustrates the various components of WebIOPi in relation to the 2-tier architecture from above:


index.html
+ webiopi.css
Client User Experience Implementation
n/a
Client Process Implementation
webiopi.js
or .../webiopi/clients/*.py
or .../java/client/*.java
Client API Implementation
jquery.js
or .../webiopi/clients/*.py
or .../java/client/*.java
Client Broker and Monitor
Browser + JScript
or Python 2/3
or Java
Client Runtime Environment
iOS, Android, Windows, ...
Client Operating System
Smartphone, Tablet, PC, ...
Client Hardware Platform
Services
Control
X
Server Broker and Monitor
.../webiopi/protocols/*.py +
.../webiopi/server/*.py +
.../webiopi/utils/*.py
Thing API Implementation
.../webiopi/server/*.py +
.../webiopi/decorators /*.py +
.../webiopi/devices/xyz/*.py
Thing Extension Environment
.../webiopi/server/*.py +
Python custom scripts
Device/TouchPoint Implementation
.../webiopi/devices/.../*.py +
.../python/native/*.*
Chip/Module Connection Implementation
.../webiopi/devices/*.py
Device and Service Runtime Environment
Python 2/3 +
1-wire kernel modules +
.../python/webiopi/*.py
Thing Operating System
Raspbian
Thing Hardware Platform
Raspberry Pi A(+), B(+) +
Breadboards & chip breakouts

The picture is self-explanatory, some aspects that are worth to be noted additionally are:

- WebIOPi does not implement a control channel, so this part of the 2-tier architecture is stroked out.
- The thing server part is mainly implemented in Python 2/3 and running on Raspbian, so the main device service runtime environment is Python 2/3.
- Some very small parts (the native GPIO port connection and the 1-wire device drivers) are implemented using the C language. The source code for the native GPIO connection is from WebIOPi and included, the source code for the 1-wire kernel modules is from standard Debian Linux and can be found in those repositories.
- The Python directory source paths (.../python/webiopi/devices/...) and the *.py file names reflect the basic relationship to the architecture building blocks. However, this is just convenience, this is neither mandatory nor perfect for each file and could be implemented completely different.
- The thing service API is identical to the WebIOPi REST API.
- The Thing Extension (blue) and Process implementation (yellow) building blocks are implemented within custom Python server scripts and their possibility to host setup(), loop(), destroy() methods and an arbitrary number of user-defined “macros”.
- The Alias building block (cyan) is implemented using the “routes” entries within the WebIOPi config file.
- On client side, a set of components exists. The main component is the JScript library (webiopi.js and jquery.js) plus some standard and example HTML pages that allow building browser-based clients for WebIOPi servers.
- Other client components which are provided are client libraries for Python and Java clients, however they are currently not fully implemented (as of Release 0.7.0) and may be more seen as examples.

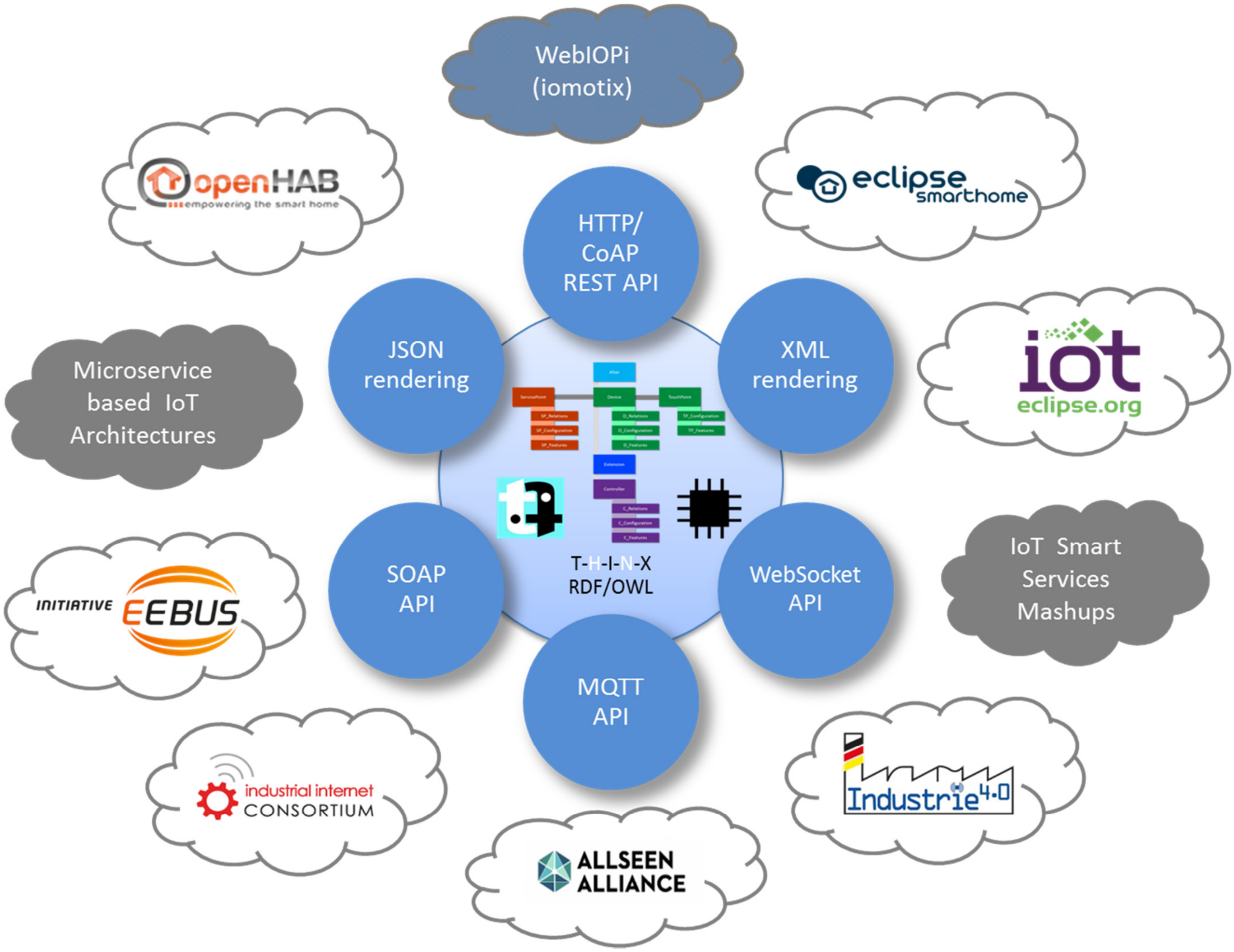
WebIOPi does provide no dedicated components to build a 3-tier architecture. However, it is possible to easily integrate the server part of WebIOPi with 3rd party 3-tier IoT architecture components from e.g. Eclipse or commercial vendors. In such a scenario, each WebIOPi instance would be a concrete thing server participating in a 3-tier infrastructure.

In order to support a possibility for easy secure access, WebIOPi can be combined with an add-on component called IoT toolkit [WEAVED_2015] developed in cooperation with Weaved Inc that provides a security enhanced communication channel from browsers or mobile apps to a WebIOPi server using the IoT secure cloud access service from Weaved. See the Weaved website for more information on this.

Positioning the IoT Model in the Landscape of an API-centric Digital Economy

Context

The following picture shows a schematic illustration what role the IoT model can play in the landscape of the upcoming API-centric digital economy ecosystems. The picture lists FOSS as well as non-FOSS activities.



In the core is the resources model with its textual and graphic representation form this document extended by a formal representation using RDF/OWL. From this conceptual core a set of different usage scenarios and patterns can be derived. This leads in the first step to the anticipated secondary representations named in the blue circles. These are not considered to be complete, just the most popular ones for upcoming IoT ecosystems are named. In the second step, the grey clouds denote some concrete ecosystems, usage scenarios and/or implementation patterns. On top, WebIOPi (and iomotix [IOMOTIX_2014]) are named as they are ancestors and the most prominent examples of an implementation for the IoT model. Also, most code examples in this document refer to this implementation.

RDF/OWL core

One of the most exiting web concepts that will gain much adoption in the upcoming next evolution of the internet are the technologies that are summarized using the keywords “Sematic Web” and “Linked Data”. The conceptual core of these activities is RDF. One of the most important implementation technologies for that is OWL. RDF/OWL is suited to represent all kinds of conceptual resource models. So it’s just natural that the T-H-I-N-X IoT resource model can also be represented in the form of an ontology based on RDF/OWL. It’s important to know that there is not one unique representation of the IoT model with that. In fact, RDF/OWL is by its nature a very generic concept so consequently many possible representations of the IoT model with RDF/OWL exist. The exemplary representation in the next chapter is just one concrete proposal for this.

The RDF/OWL core ontology allows representing the state of each IoT things server instance with those means at runtime. This state can then be inserted into any kind of non-relational database, e.g. one of the “triples stores” (e.g. Virtuoso) that are mainly suited to carry RDF content. And, resulting from that, it will be possible to retrieve and navigate that IoT instance state content with query tools that are perfect suited for such content like SPARQL.

At the bottom line, this will allow to search and navigate real-time IoT server content with tools from the next evolution of the internet. It may also allow implementing very advanced interaction concepts like natural language communication and Q&A sessions on IoT data using some of the upcoming cognitive computing engines. Imagine you could answer questions like “Are all lights shut off in my house?” and give commands like “Please switch off all lights in my office!” without the need to create and manage explicit lists of all your lights or implement special hand-crafted code that iterates all your lights and does some action based on that hard-wired navigation vehicle. Future? Yes, but a kind of future we can expect to become true in the next evolution steps of the internet, namely also of the internet of things. The RDF/OWL model will be the base for that.

JSON rendering

One of the more simple and immediate benefits of the IoT model is its rendering in JSON. JSON is nowadays very popular in the context of API’s, namely REST API’s. JSON allows describing and transferring the state of an IoT server instance. It can be used on a very fine grained level by just carrying the state of a single sensor but also on a very coarse-grained level by carrying the complete state of an IoT server plus all levels of granularity in between those extreme variants.

As with RDF/OWL, JSON also has some kind of schema language to describe the JSON rendering on a formal base. If such a schema is available, this will allow the usage of tools consuming the schema and JSON instance data to do verifications and/or augmented documentation (like e.g. with Docson). It will even be possible to generate code for applications that produce or consume JSON content based on that schema. And, in the meanwhile even some non-SQL databases exist that allow the native storage of JSON content for later (real-time) retrieval (e.g. MongoDB)

The chapter on JSON mapping and serialization contains a representation of the IoT model in JSON language concepts.

XML rendering

One of the more advanced but still very common benefits of the IoT model is its rendering in XML. XML is up till now very popular in the context of service-oriented architectures, namely SOAP API’s. Similar to JSON also XML allows describing and transferring the state of an IoT server instance. It can be used on a very fine grained level by just carrying the state of a single sensor but also on a very coarse-grained level by carrying the complete state of an IoT server plus all levels of granularity in between those extreme variants. On a high-level conceptual view there is no difference between JSON and XML, it’s just another formal language. The most important difference is that XML allows more formal details (allowing finer-grained verification) and that XML is somewhat much more verbose compared to JSON.

As with JSON, XML also has a schema language to describe the XML rendering on a formal base. If such a schema is available, this will allow the usage of tools consuming the schema and XML instance data to do verifications and/or augmented documentation. It will even be possible to generate code for applications that produce or consume XML content based on that schema. And, in the meanwhile even some non-SQL databases exist that allow the native storage of XML content for later (real-time) retrieval and analysis.

The chapter on XML mapping and serialization contains a representation of the IoT model in XML language concepts.

HTTP/CoAP REST API

One of the most popular concept and technology for APIs in the context of modern web based architectures are RESTful APIs. They have been introduced some years ago and are now very broadly used all over the internet.

The chapters on different HTTP/CoAP REST API's show how the IoT model can be used to map it into directly into a set of RESTful API calls. It also refers to special API tools that can be used to document and test a set of formal defined API's like the swagger toolset [SWA_2014].

SOAP API

The most popular concept and technology for APIs in the context of service oriented architectures (SOA) is the usage of the SOAP protocol [SOAP_2000] in combination with WSDL [WSDL_2001] and XML payloads.

The chapter on SOAP API shows how the IoT model can be used to map it into directly into a set of SOAP API calls that carry XML rendered IoT payload.

WebSocket API

WebSockets (WS) [WEBSOCK_2014], [IETF RFC6455] is one of the next evolution of the HTTP protocol. In extension of the standard HTTP protocol WebSockets allow the establishment of a permanent bi-directional communication channel between two networked nodes based on TCP/IP. This is very similar to the good old serial communication line and its very popular network-based successor the so-called sockets that derive form work made for UNIX (hence the name web-"sockets"). The biggest advantage is the fact that once the bi-directional communication channel has been established, the (things) server can send back answers to the client (e.g. state changes and/or event/interrupt notifications) without the need for any continuous polling mechanism on the client side. As you can imagine, such an enhanced communication scheme will be of high benefit for IoT client/server scenarios.

Due to its close relation to the HTTP protocol WebSockets are implemented as an extension/upgrade of the HTTP handshake and their standard port is still 80.

The chapter on WebSockets will show how the IoT model can be used in WebSocket communication architectures that implement a WebSocket API.

MQTT API

MQTT (MQ Telemetry Transport) [MQTT_2014] is another network protocol that looks to have an important impact in the IoT ecosystem landscape. It has been developed by IBM and others and has been handed over to OASIS to make it an IoT world official standard. MQTT is simple, lightweight and thus optimized for communication with embedded devices that are limited in their processor or memory resources in an IoT infrastructure. It has variants based on TCP/IP and non-TCP/IP networks.

The chapter on MQTT API will show how the IoT model can be used in MQTT communication architectures that implement a MQTT API.

Other candidate IoT API protocols and payload rendering options

Some common used IoT M2M protocols in broad use are:

- XMPP/Jabber (http://wiki.xmpp.org/web/Tech_pages/IoT_systems)
- Efficient XML Interchange (EXI) (http://www.w3.org/TR/exi/)
- OMA Lightweight M2M (http://openmobilealliance.hs-sites.com/lightweight-m2m-specification-from-oma)
- OPC Unified Architecture (UA) (https://opcfoundation.org/about/opc-technologies/opc-ua/)

- ETSI SmartM2M (http://www.etsi.org/technologies-clusters/technologies/m2m)

Mode text tbd.

IoT Ecosystems and Consortiums

Text tbd.

TODO: Reference to EEBUS, IIC, AA and I4.0.

IoT Architecture Patterns

Text tbd.

TODO: Reference to:

- Microservice Architectures and
- Service Mashups

Formalizing the Model with RDF/OWL

Text tbd.

TODO list:

- Defining an ontology using RDF/OWL with Protégé
- Check references to other sematic W3C standards and/or ontologies and/or namespaces related to IoT resources and objects, candidates are
  - Clock -> http://www.w3.org/TR/owl-time/
  - ...
- Check usage of JSON-LD -> http://www.w3.org/TR/json-ld/
- Provide unique identifiers for resource nodes; it is tbd for what kind of nodes this makes sense (external exposed granularity)
- Provide HTTP addresses for sematic lookup

THINX HTTP/CoAP REST API Mapping

Overview

This chapter aims to map the complete IoT model with all its facets to an HTTP/CoAP based IoT service API. The proposed mapping has currently no concrete implementation but may get one in the future. The complete mapping is only dependent on the IoT model and completely independent from any concrete implementation language.

Basic mapping concepts and main resource node path

The most important concepts for the REST API mapping are:

- HTTP/CoAP verbs are used according to the following rules:
  - **GET** -> get/read values
  - **PUT** -> set/write/update values
  - **POST** -> create (temporary) resource objects
  - **DELETE** -> delete (temporary) resource objects
- If submitting of request values is necessary (e.g. for PUT, POST) they are transferred either URL-encoded within the request path (simple cases) or using the request body (complex cases). Valid mime types are “text/plain”, “application/json” and “application/xml”. Mime types have to be set correct for all requests. Within this document, other chapters describe proposals for JSON and XML renderings of the IoT model that can be used here. Depending on the affected resource node for a request, also partly serialized fragments of the resource model can be used.
- The main resource path from ServicePoint via Device to TouchPoint (being the dark gray line in the IoT model) is mapped by navigating along the SP_Relations slot <devices> with this fundamental schematic URL path pattern `{SP_URI}/devices/` and
- being directly extended by `{SP_URI}/devices/{devicename}/{category}[/{aspect}][/{chan/nel}][/{value}]`
- `{SP_URI}` is just a schematic placeholder for a valid ServicePoint URI like e.g. `http://my.domain:8000` or `coap://224.0.1.123:5683`.
- According to the slot cardinalities of the IoT model, the path parameters `{devicename}` and `{category}` are mandatory; the path parameters `{aspect}`, `{chan/nel}` and `{value}` are optional. Please note that channel path parameter can add one or more additional `/` inside. For URL-encoded PUT and POST values, they are added after an additional `/`.
- Navigating along the main path leads only to the TouchPoint values or the full/partial (JSON) state of the selected resource node. All other API slots are handled using special request parameter verbs. This is valid for reading, writing, creating and deleting type of requests.

API version indicator

As a rule for handling API releases and upgrades it is recommended to add a version indicator to the resource path. The preferred place for such an indicator is at the very root of the main resource path like in this pattern:

```
{SP_URI}/{versionIndicator}/{restOfPath}
```

In the simplest case, the value for `{versionIndicator}` could just be something like “1” or “v2”. The main resource path would be in this case:

```
{SP_URI}/v2/devices/{devicename}/{category}[/{aspect}]
/{chan/nel}][/{value}]
```

In order to keep the patterns and examples in the remainder of this chapter short, the version indicator is removed from all paths but should be added in a real final implementation.

Device and {devicename}

The path parameter {devicename} is used for the <name> slot of the Device nodes. For that reason all device names must comply with the rules valid for URL paths. This means that letters and digits can be used without problems. However, care must be taken with all special characters.

An example for device selection using the devicename "mydevice" is:

```
http://my.domain:8000/devices/mydevice/...
```

Especially for modern SOC microcontrollers with rich embedded device functionalities the predefined names "native" or "built-in" should be used to address those special devices.

An example for addressing the native GPIO ports of such a microcontroller would be:

```
http://my.domain:8000/devices/native/digital/25
```

TouchPoint and {category}

The path parameter {category} is used for the <category> slot of the TouchPoints. The following possibilities (the list names the most important ones but can't be complete) for categories exist:

- digital
- bitstream
- analog
- waveform
- sensor
- clock
- memory
- id
- interface
- bus
- ...

An example for combined device and category selection is:

```
http://my.domain:8000/devices/mydevice/digital/...
```

TouchPoint and {aspect}

The path parameter {aspect} is used for the <aspect> slot of the TouchPoints. The following possibilities (the list names the most important ones but can't be complete) for aspects exist:

- Geometrical entities that sensors measure; this expands unique to sensor/{geometricalentity} and has the following possibilities (also this list can't be complete, in principal all known measurable geometrical entities are possible)
  - sensor/position
  - sensor/distance
  - ...
- Physical effects that sensors measure; this expands unique to sensor/{physicaleffect} and has the following possibilities (also this list can't be complete, in principal all known measurable physical effects are possible)
  - sensor/acceleration

    - sensor/rotation
    - sensor/orientation
    - sensor/velocity
    - sensor/temperature
    - sensor/pressure
    - sensor/altitude
    - sensor/luminosity
    - sensor/humidity
    - sensor/current
    - sensor/voltage
    - sensor/power
    - sensor/tension
    - sensor/radiation
    - ...
- Chemical effects that sensors measure; this expands unique to sensor/{chemicaleffect} and has the following possibilities (also this list can't be complete, in principal all known measurable chemical effects are possible)
    - sensor/molarity
    - sensor/ ph-value
    - ...
- Biological indicators that sensors measure; this expands unique to sensor/{biologicalindicator} and has the following possibilities (also this list can't be complete, in principal all known measurable biological indicators are possible)
    - sensor/glucose
    - sensor/cholesterol
    - ...
- Direction indicators:
    - digital/input
    - analog/output
    - ...
- Special signal form indicators
    - analog/pwm
    - waveform/am
    - waveform/fm
    - bitstream/ac3
    - bitstream/dts96
    - bitstream/pcm48
    - ...
- Protocol or technology indicators:
    - interface/serial
    - interface/rs485
    - interface/midi
    - interface/dmx
    - bus/i2c
    - bus/spi
    - bus/1wire
    - ...
- Logical sub data slots or format indicators:

    - clock/time
    - clock/date
    - clock/datetime
    - clock/second
    - clock/minute
    - clock/hour
    - memory/bit
    - memory/byte
    - memory/word
    - memory/long
    - id/uuid
    - id/raw
    - ...
- Predefined keywords for specific attributes of a TouchPoint category:
    - digital/count
    - digital/banks
    - analog/count
    - bitstream/channels
    - ...

TouchPoint and {channel}

The path template parameter {channel} is used for the <channel> slot of the TouchPoints. The following possibilities for channels exist:

- Plain enumeration with (sequential) positive integer numbers including 0:
    - digital/0
    - analog/15
    - sensor/temperature/3
    - ...
- Plain identification with a string:
    - digital/output/relais1
    - analog/input/port-b
    - bitstream/pcm48/center
    - ...
- Geometrical coordinate or reference point identifiers:
    - position/x
    - position/y
    - position/z
    - position/lat
    - position/long
    - position/alt
    - ...
- Physical sub effect, coordinate or reference point identifiers:
    - pressure/sea
    - luminosity/visible
    - luminosity/ir
    - luminosity/uv
    - humidity/relative

  - humidity/absolute
  - acceleration/x
  - acceleration/y
  - acceleration/z
  - rotation/roll
  - rotation/pitch
  - rotation/yaw
  - .../resolution
  - .../maximum
  - .../vref
  - ...
- Chemical reference identifiers:
  - ph-value/rvs
  - ph-value/ps
  - ph-value/os
  - ...
- Biological reference identifiers:
  - glucose/blood/fbs
  - glucose/blood/ogtt
  - glucose/blood/ivgtt
  - glucose/blood/hba1c
  - ...

Every TouchPoint value has an optional default unit and/or mathematical type that is used unless otherwise specified (NONE is allowed). In order to choose other variants (held by the TouchPoint slots <unit> and <mathematicalType>) than these default unit and types, a special mechanism has to be used that will be explained later.

Examples for combined device, category, aspect and channel selection are:

```
http://my.domain:8000/devices/mydevice/digital/output/12
http://my.domain:8000/devices/mydevice2/sensor/temperature/3
http://my.domain:8000/devices/mydevice2/sensor/pressure/sea
```

Just for completeness, to demonstrate the version indicator recommendation, this would result in final paths like these:

```
http://my.domain:8000/v2/devices/mydevice/digital/output/12
http://my.domain:8000/v2/devices/mydevice2/sensor/temperature/3
http://my.domain:8000/v2/devices/mydevice2/sensor/pressure/sea
```

ServicePoint

As mentioned in the base mapping concept, the ServicePoint is the root anchor point for the resource path and is mapped via its URI(s) using just this placeholder:

```
{SP_URI}
```

An example for accessing a ServicePoint root node is:

```
http://my.domain:8000/...
```

X_Relations

Basically, the selection along the resource path for X_Relations to an individual node is implemented by putting an identifying name at the end of a path (segment) derived from the ServicePoint root like

- `{SP_URI}/devices/myname` gives access to the whole Device named “myname” (as already explained above)
- `{SP_URI}/controllers/somecontroller` gives access to the whole Controller named “somecontroller”
- `{SP_URI}/extensions/an-extension` gives access to the whole Extension named “an-extension”

All identifying slots of the abstract IoT model are marked with an asterix “*” after the name of the resource node slot names. More details for controllers and extensions will be explained later in separate sub chapters.

In order to map the [o..n] cardinality of X_Relations for a selection along the resource path for all slots with a maximum cardinality higher as 1 the help of the commonly known wildcard character “*” is used, like

- `{SP_URI}/devices/*` which gives the list of all [0..n] existing devices or
- `{SP_URI}/devices/mydevice/*` which gives access to all TouchPoints (i.e. analog and digital channels) of the Device named “mydevice”.
- `{SP_URI}/devices/mydevice/digital/*` which gives access to all digital TouchPoints (i.e. digital channels) of the Device named “mydevice”.
- `{SP_URI}/devices/mydevice/sensor/temperature/*` which gives access to all temperature sensor TouchPoints of the Device named “mydevice”.

The standard results of those requests are JSON objects or JSON arrays. The nesting deepness of the JSON results may be configurable by additional request parameters.

In order to be able to submit additional selection criteria it is allowed to add this by query parameters like

- `{SP_URI}/devices/*?category=digital` which gives the list of all digital TouchPoints or
- `{SP_URI}/devices/*?class=TMP275` which gives the list of all “TMP275” class Devices

X_Configuration

X_Configuration has to do with configuring the resource nodes. This is mapped to

- the standard main resource node path to navigate to the resource which is being configured
- and extending this with the special request “verb” `/configure` being added to the very end of the path
- **GET** -> get/read configuration slot (values)
- **PUT** -> set/write/update configuration slot (values)

After this, the detailed further selection of all X_Configuration slots is mapped via request query parameters and their values. The names of the query parameters reflect the X_Configuration slot names.

Two different cases exist:

- Setting configuration slots
- Retrieving the current values of them

Setting values

The basic pattern for setting configuration values is like this:

```
{SP_URI}/{resourcepath}/configure?{resourceslot}={RESOURCEVALUE}
```

The parameter `{resourcepath}` is to be specified correct to access the Devices, TouchPoints, Controllers and Extensions. For the ServicePoint, it is just empty.

For better readability predefined slot names `{resourceslot}` use lowercase and slot values `{RESOURCEVALUE}` use uppercase (if applicable, digits remain digits). Configuration slots that are <parameters> use the parameter name and value with such a pattern:

```
{SP_URI}/{resourcepath}/configure?parameter={paramName}&value={PARAMVALUE}
```

An example for a configuration value setting of a TP_Configuration slot <function> is:

```
http://my.domain:8000/devices/mydevice/digital/1/configure?function=IN
```

More than one slot/parameter is possible like this for the slots <termination> and <detection>:

```
.../digital/1/configure?termination=PULLUP&detection=ONDOWN
```

An example for a configuration value setting of a D_Configuration and a parameter named “resolution” is:

```
http://my.domain:8000/devices/.../sensor/temperature/configure?
parameter=resolution&value=11
```

An example for a configuration value setting of a C_Configuration for a controller named “main” is:

```
http://my.domain:8000/controllers/main/configure?parameter=looptime&value=500
```

If some value is too long for the URL it may be put into the request payload as JSON object and have an according hint “PAYLOAD” in the query parameter value like this SP_Configuration:

```
http://my.domain:8000/configure?parameter=encryption&value=PAYLOAD
```

Retrieving values

The basic pattern for getting configuration values is like this:

```
{SP_URI}/{resourcepath}/configure?query={resourceslot,s}
```

Here, the predefined query parameter `query` is used and its value is just the name(s) of the configuration slot(s) `{resourceslot,s}` used in the same way as for setting the slot values. The result value is in the response payload as plain text or JSON object.

An example for a configuration value reading form a TP_Configuration is:

```
http://my.domain:8000/devices/mydevice/digital/1/configure?query=function
```

The result payload in this case would be IN, OUT or INOUT.

If more than one configuration value is requested this is possible by requesting them with a list of slot names:

```
{SP_URI}/{resourcepath}/configure?query={resourceslot1},{resourceslot2}
```

or by just requesting them all via a special query value of “*”:

```
{SP_URI}/{resourcepath}/configure?query=*
```

The result for multiple slot retrieving is a JSON object containing all configuration key/value pairs.

An example for multiple configuration value reading from a TP_Configuration is:

```
http://my.domain:8000/.../digital/1/configure?query=function,termination
```

TP_Configuration

Within TouchPoint configurations two special slots exist that have to do with units <unit> and mathematical types <mathematicalType>. In order to utilize these slots dedicated query parameters are used to map this.

Units can be used with the dedicated query parameter “unit”. Many TouchPoint (channels) have a fixed default unit (e.g. Kelvin for temperature). When reading and setting the value of such TouchPoints and no unit is given, then the values are assumed to be in that default unit. If any other unit should be used, this can be achieved by adding this unit via a query parameter resulting in this pattern:

```
{SP_URI}/.../{category}[/{aspect}][/{chan/nel}][/{value}]?unit={Unit}
```

Examples for units are:

- C (Celsius)
- mm (Millimeter)
- A (Ampere)
- ...

Units should be written in the case that is common sense enforced by international physics etc. standards (recommended is ISO 80000 series standard). An example for value reading from a temperature sensor TouchPoint is:

```
http://my.domain:8000/devices/mydevice/sensor/temperature
```

which will read the temperature in the default unit (assuming this is Kelvin).

In contrast, this

```
http://my.domain:8000/devices/mydevice/sensor/temperature?unit=C
```

will read the temperature in the specified unit (Celsius).

Mathematical types can be used with the dedicated query parameter “type”. Many TouchPoint (channels) have a fixed default mathematical type (e.g. Floating point for DAC/ADC converters). When reading and setting the value of such TouchPoints and no mathematical type is given, then the values are assumed to be in that default type. If any other type should be used, this can be achieved by adding this type information as query parameter resulting in this pattern:

```
{SP_URI}/.../{category}[/{aspect}][/{chan/nel}][/{value}]?type=
{MATHEMATICALTYPE}
```

Examples for mathematical types are:

- INTEGER
- FLOAT
- BOOLEAN
- RATIO
- ANGLE
- ...

Mathematical types should be written in uppercase. An example for value reading from an analog channel (with number 2) TouchPoint is:

```
http://my.domain:8000/devices/mydevice/analog/2
```

This will read the analog value in the default type (Float).

In contrast, this

`http://my.domain:8000/devices/mydevice/analog/2?type=INTEGER`

will read the analog value in the specified type (Integer).

The same applies when setting the analog value:

`http://my.domain:8000/devices/mydevice/analog/2/2.5`

versus

`http://my.domain:8000/devices/mydevice/analog/2/511?type=INTEGER`

Situations may exist where both mechanisms (units and mathematical types) are being used in combination like:

`{SP_URI}/.../...?unit={Unit}&type={MATHEMATICALTYPE}`

The common rule is that setting and getting values has to be possible without any additional query parameters for the most frequently standard used cases.

Creating and destroying a Device at runtime

It is possible to create a Device node at runtime. This is mapped to the basic pattern:

- `{SP_URI}/devices/create?class={ClassName}&name={devicename}`
- and the usage of the HTTP verb **POST**.

In the case that an additional address is necessary this can be attached at the end. An example for the dynamic creation of a temperature sensor Device of class “TMP75” with the name “tmp1” and the I2C slave address “0x4A” would be:

`{SP_URI}/devices/create?class=TMP75&name=tmp1&address=0x4A`

It is possible to destroy a Device node at runtime. This is mapped to the basic pattern:

- `{SP_URI}/devices/destroy?name={devicename}`
- and the usage of the HTTP verb **DELETE**.

An example for the dynamic destruction of the temperature sensor Device just created above would be:

`{SP_URI}/devices/destroy?name=tmp1`

The practical benefit of creating and destroying Devices at via the REST API would be the implementation of a web-based configuration console for an IoT server instance. During a test or configuration phase, all Devices could be created and configured until they fit to all needs. After that, the configuration setup could be saved to any kind of static resource (e.g. written to a file) and then reused without console-based manual configuration.

X_Features

X_Features has to do with activating or triggering something on the resource nodes (like running some command or firing some event). This is mapped to

- the standard main resource node path to navigate to the resource which provides the intended feature,
- extending this with the special request “verbs” being added to the very end of the path,
- in contrast to X_Configuration, the verbs are now derived from the feature slot names (like e.g. /run for the <runnables> slots,
- plus all other information being submitted by a set of query parameters,
- and finally **POST** for the HTTP/CoAP verb as something is being created (temporarily) on the resource node.

<runnables>

For all <runnables> slots, this looks like this:

```
{SP_URI}/{resourcepath}/run?method={methodName}
```

As `{methodName}` is the identifier of some kind of executable code is has to be written exactly like the case-sensitive name of that piece of code (e.g. some procedure name).

When parameters are needed for the methods, this can be extended with either sequential values separated by commas:

```
{SP_URI}/{resourcepath}/run?method={methodName}&arguments={ARG,VALU,ES}
```

or explicit named parameters like this:

```
{SP_URI}/{resourcepath}/run?method={methodName}&arguments={VALUE1,VALUE2}&argument_names={arg1,arg2}
```

If named parameters are used, they have also to be written in the exact case-sensitive way as in the original source code. As a rule of thumb, the values of those parameters should be either numbers or strings that are case-insensitive.

An example for a runnable feature of a TP_Feature is:

```
http://my.domain:8000/.../digital/1/run?method=sequence&arguments=10,1,0,1,0
```

An example for a runnable feature of a D_Feature is:

```
http://my.domain:8000/devices/mydevice/run?method=calibrate
```

or

```
http://my.domain:8000/devices/mydevice/run?method=wake
```

or

```
http://my.domain:8000/devices/mydevice/run?method=sleep&arguments=2000&argument_names=time
```

<events>

For <events> slots, this looks like this:

```
{SP_URI}/{resourcepath}/event?name={eventName}&action={FIRE/CANCEL/HIDE/...}
```

<logs>

For <logs> slots, this looks like this:

```
{SP_URI}/{resourcepath}/log?name={logName}&read=100
```

The same pattern can be used for all other X-Features slots. But not every X_Feature has to be exposed via the REST API.

Extension

The extension resource node is mapped with the basic resource path

- `{SP_URI}/extensions/{extensionName}`
- and the usage of the HTTP verb **GET** and **POST**.

The `{extensionName}` is the <name> slot value of the Extension node. To some extent, extensions have some kind of similarity with runnable features. The main difference is that runnable features are directly attached to their respective sub nodes and really execute a bit of code whereas Extensions are a more abstract concept with their own primary node type and may be much more than just a small piece of code that gets executed. Big Extensions may even represent a complete deployable package or a complete subsystem ***(-> reference to TOSCA)*** running on the Things Server in its own threads that gets started.

The Extension resource node is mapped with the basic resource path:

```
{SP_URI}/extensions/{extensionName}
```

which answers the whole Extension is any kind of appropriate format.

The activation/execution of an Extension is mapped to the path `/run` in the same way as the features with the difference that now the `method` query parameter is not needed. And, in some cases the activation of an Extension also needs one or more parameters. This is mapped identically to the other `/run` paths so we end up with:

```
{SP_URI}/extensions/{extensionname}/run?arguments={...}&argument_names={...}
```

Once again, the usage of argument_names is optional.

Controller

The Controller resource node is mapped with the basic resource path:

```
{SP_URI}/controllers/{controllername}
```

If only one (default) Controller node does exist, the predefined name "main" is used for `{controllername}`. Otherwise the name/id of the main process of a distinct controller has to be used but this will occur very seldom.

An example to retrieve some default Controller parameters from C_Configuration would be:

```
{SP_URI}/controllers/main/configure?query=board-revision
```

or

```
{SP_URI}/controllers/main/configure?query=coap-multicast
```

An example to set some default Controller parameters for C_Configuration would be:

```
{SP_URI}/controllers/main/configure?parameter=coap-multicastval&value=TRUE
```

An example to activate some default controller <runnables> from C_Features would be:

```
{SP_URI}/controllers/main/run?method=setup
```

or

```
{SP_URI}/controllers/main/run?method=destroy
```

If just the complete state of the default controller is requested and enforced to a JSON result, this can be done by:

```
{SP_URI}/controllers/main/*.json
```

Alias

After all the dedicated mappings above, there is finally a special case to simplify things a bit. An Alias has no specific resource path prefix but MUST NOT overlap with all other mappings. For this reason, the first rule is that aliases starting with the reserved names `/devices`, `/controllers` and `/extensions` are forbidden. The second rule is that every alias must have an original path that is valid within the current resource tree. The third rule is that for the sake of simplicity an alias can only substitute a resource path without any additional query parameters.

An example for a valid Alias would be to substitute this technical path

```
http://my.domain:8000/devices/mydevice/digital/output/12
```

with such an “aliased” path:

`http://my.domain:8000/lights/livingroom/ceiling`

It could be also valid to substitute this alternatively by

`http://my.domain:8000/livingroom/lights/ceiling`

and have another Alias being

`http://my.domain:8000/livingroom/temperature/floor`

which could be an Alias for the technical path

`http://my.domain:8000/devices/myotherdevice/sensor/temperature/2`

At the bottom line the concept of “Aliasing” will allow to have a separate real/physical/logical resource path model of some entity (e.g. a building) and to decouple this from the technical resource path model described above. Plus, it allows giving some resources a constant logical name whereas their technical mapping can be modified over time.

Result and payload type enforcement

In order to allow the enforcement of result and payload mime types it is possible to do this by adding a payload type marker to the URL. This is done by extending the URL path with this pattern:

`{SP_URI}/{resourcepath}.{payloadmarker}...`

The `{payloadmarker}` gets directly attached post fixed to the resource path with a leading `.` separator and has to be inserted BEFORE any query parameters. Possible payload markers are

- txt ->plain text result
- json -> JSON object result
- xml -> XML result

As noted already above, a possibility to get the list of all TouchPoints for a Device is:

`{SP_URI}/devices/mydevice/*`

By default, this will answer the result as JSON Object. To enforce the result being a XML structure, this can be specified in such a way:

`{SP_URI}/devices/mydevice/*.xml`

Or, it could be also possible to request this in plain text:

`{SP_URI}/devices/mydevice/*.txt`

Error and exception handling

Text tbd.

Additional resources

Swagger Specification of the REST API

Remarks

The mapped API from above can be described formally with a Swagger spec [SWAGGER_2014]. The spec below relies on Version 2 of the Swagger spec and uses the YAML [YAML_2015] representation of it as YAML is better human readable and less verbose that pure JSON. It has been edited and validated with the Swagger editor online tool [SWAGGER_EDITOR_2015] as long as possible but the latest version from below comes with some restrictions that hopefully disappear over time:

- Currently it can’t be edited with the online tool as this tool does not support the “$ref” element for parameter elements on operations level correct at the time of this release. This is a confirmed bug of Swagger.
- Some other parts of the Swagger 2.0 spec are officially not supported by the Swagger editor at the time of this writing but do not cause any problems (e.g. Enums). For this, these parts are within the spec but they could not be validated so far.
- Some elements of the resource path are optional (e.g. {aspect}) but the Swagger spec does not allow this so far. For this reason some of the paths have been documented in the full breadth of all possible combinations as a (hopefully) temporary workaround.
- Some concepts of the API are currently not included in the spec. Namely the APIs for the events and logs resource slots and the payload enforcement indicator.
- The (JSON) schema part of the Swagger spec (“definitions: ...”) is currently only rudimentary contained to show its basic purpose. As soon as the final JSON schema for T-H-I-N-X is available it will be substituted by this version.
- The error and response handling is currently very simplified.

Specification in YAML format

```
#
#   Copyright 2015 Andreas Riegg - t-h-i-n-x.net
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.
#
#   This is the YAML source of the T-H-I-N-X REST API based on Swagger 2.0
#
#   Changelog
#
#   1.0.0    2015-03-17    Initial release.
#

---
swagger: '2.0'

# Basic information and copyright
# -------------------------------

info:
  title: T-H-I-N-X REST API
  description: API proposal suited for the generalized T-H-I-N-X IoT resources model.
  termsOfService: n/a
  version: "1.0.0"
  contact:
    name: Andreas Riegg
    url: http://www.t-h-i-n-x.net/
    email: brain@t-h-i-n-x.net
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html

# Top-level attributes of the API
# -------------------------------
# The domain of the service, change this for any other URL when using any kind of live request generating
toolset
host: api.t-h-i-n-x.net:8000

# E.g. for a connection via Weaved this may be something like
# host: abcxyz.p2.yoics.net:8000

schemes:
- http
- https
- ws
- wss
# - coap (added for later inclusion, currently not supported by the Swagger spec)

basePath: /v100

securityDefinitions:
  basicAuth:
    type: basic
    description: HTTP Basic Authentication with userid and password.

security:
- basicAuth: []

consumes:
- text/plain
- application/json

produces:
- text/plain
- application/json

tags:
- name: ServicePoint
  description: ServicePoint resource operation
- name: Device
  description: Device resource operation
- name: TouchPoint
  description: TouchPoint resource operation
- name: Controller
  description: Controller resource operation
- name: Extension
  description: Extension resource operation
- name: Alias
  description: Alias resource operation
- name: Relations
  description: Relations navigating operation
- name: Configuration
  description: Configuration handling operation
- name: Feature
  description: Feature management operation
- name: (C)
  description: Create operation
- name: (R)
  description: Read operation
- name: (W)
  description: Write operation
- name: (D)
  description: Delete operation
- name: (X)
  description: Execution operation

# Parameters defined at top level for reuse via ref-element in many APIs
# ------------------------------------------------------------------------

parameters:
  devicenamePathParam:
    name: devicename
    in: path
    description: The value of the &lt;name&gt; slot of a Device.
    required: true
    type: string
  categoryPathParam:
    name: category
    in: path
    description: >
      The value of the &lt;category&gt; slot of a TouchPoint in lowercase characters. Examples for
categories are
      digital, bitstream, analog, waveform, sensor, clock, memory, id, interface, bus.
    required: true
    type: string
    enum:
    - digital
    - bitstream
    - analog
    - waveform
    - sensor
    - clock
    - memory
    - id
    - interface
    - bus
  categoryQueryParam:
    name: category
    in: query
    description: >
      The value of the &lt;category&gt; slot of a TouchPoint in lowercase characters. Examples for
categories are
      digital, bitstream, analog, waveform, sensor, clock, memory, id, interface, bus.
    required: false
    type: string
    enum:
    - digital
    - bitstream
    - analog
    - waveform
    - sensor
    - clock
    - memory
    - id
    - interface
    - bus
  aspectPathParam:
    name: aspect
    in: path
    description: >
      The value of the &lt;aspect&gt; slot of a TouchPoint. Examples for aspects are input, output,
      pwm, temperature, pressure, date, time, byte, word etc.
    required: true
    type: string
  aspectQueryParam:
    name: aspect
    in: query
    description: >
      The value of the &lt;aspect&gt; slot of a TouchPoint. Examples for aspects are input, output,
      pwm, temperature, pressure, date, time, byte, word etc.
    required: false
    type: string
  channelPathParam:
    name: channel
    in: path
    description: >
      The value of the &lt;channel&gt; slot of a TouchPoint. Channels are strings and can be any mixture
of
      alphanumeric letters. Examples for channels are 2, 13, relais1, relative, x, y, z, visible etc.
    required: true
    type: string
  valuePathParam:
    name: value
    in: path
    description: The new value of the TouchPoint channel.
    required: true
    type: string
  valueBodyParam:
    name: b_value
    in: body
    description: The new value of the TouchPoint channel.
    required: false
    schema:
      type: string
  unitQueryParam:
    name: unit
    in: query
    description: >
      The value of the &lt;unit&gt; slot of a TouchPoint in mixed case characters and noted
      according to the ISO 80000 series standards. Units are strings and can be any mixture of
      alphanumeric letters. Examples for units are mm, s, V, A.
      If unit is omitted, the TouchPoint value will be answered in its default unit.
    required: false
    type: string
```

```
typeQueryParam:
  name: type
  in: query
  description: >
    The value of the &lt;mathematicalType&gt; slot of a TouchPoint in uppercase characters.
    If omitted, the value will be answered in its default type.
    Possible values are INTEGER, FLOAT, BOOL, BIN, OCT, HEX, DATE, TIME, TEXT.
  required: false
  type: string
methodQueryParam:
  name: method
  in: query
  description: The identifier of a &lt;runnable&gt; of a Resource node.
  required: true
  type: string
argumentsQueryParam:
  name: arguments
  in: query
  description: A comma-separated list of arguments for a Resource node feature.
  required: false
  type: array
  items:
    type: string
  collectionFormat: csv
argumentNamesQueryParam:
  name: argument_names
  in: query
  description: A comma-separated list of argument names for a Resource node feature.
  required: false
  type: array
  items:
    type: string
  collectionFormat: csv
functionQueryParam:
  name: function
  in: query
  description: The &lt;function&gt; slot of a TouchPoint channel.
  required: false
  type: string
  enum:
  - IN
  - OUT
  - INOUT
terminationQueryParam:
  name: termination
  in: query
  description: The &lt;termination&gt; slot of a TouchPoint channel.
  required: false
  type: string
  enum:
  - PULLUP
  - PULLDOWN
  - NONE
detectionQueryParam:
  name: detection
  in: query
  description: The &lt;detection&gt; slot of a TouchPoint channel.
  required: false
  type: string
  enum:
  - ONUP
  - ONDOWN
  - NONE
queryQueryParam:
  name: query
  in: query
  description: >
    The name(s) of a configuration slot of a Resource node. If more than one slot is requested
    the names are separated by commas. If all slots are required then use "*" for this parameter.
  required: false
  type: array
  items:
    type: string
  collectionFormat: csv
parameterQueryParam:
  name: parameter
  in: query
  description: The name of a &lt;parameter&gt; configuration slot of a Resource node.
  required: false
  type: string
  valueQueryParam:
    name: value
    in: query
    description: The value of a &lt;parameter&gt; configuration slot of a Resource node.
    required: false
    type: string
  extensionnamePathParam:
    name: extensionname
    in: path
    description: The value of the &lt;name&gt; slot of an Extension.
    required: true
    type: string
  controllernamePathParam:
    name: controllername
    in: path
    description: The value of the &lt;name&gt; slot of a Controller.
    required: true
    type: string


# REST API resources paths
# ------------------------

paths:

# ServicePoint node
# -----------------

  /configure:
    get:
      summary: ServicePoint configuration retrieval
      description: This request answers the value(s) of ServicePoint configuration slot(s).
      parameters:
      - $ref: '#/parameters/queryQueryParam'
      tags:
      - ServicePoint
      - Configuration
      - (R)
      responses:
        200:
          description: The new value of the parameter.
          schema:
            type: string
        default:
          description: Any kind of error

    put:
      summary: ServicePoint configuration update
      description: This request updates the value of a ServicePoint configuration slot.
      parameters:
      - $ref: '#/parameters/parameterQueryParam'
      - $ref: '#/parameters/valueQueryParam'
      tags:
      - ServicePoint
      - Configuration
      - (W)
      responses:
        200:
          description: The new value of the parameter.
          schema:
            type: string
        default:
          description: Any kind of error

# Device node
# -----------

  /devices/*:
    get:
      summary: Device listing
      description: >
        This request answers a complete or selected list of all existing Devices in the form of a JSON
array.
        The simple list just contains array entries with all unique device names.
        The abstracted list contains array entries with JSON objects that give the Device name and
resource abstractions of a device {name:type:}.
        The abstraction results from the &lt;category&gt; and &lt;aspect&gt; of the TouchPoints of each
Device.
        If a device provides more than one abstraction, then a separate JSON object is returned for each
abstraction type.
```

```
    The full list contains array entries with complete JSON representations of each Device.
   parameters:
   - name: scope
    in: query
    description: The scope of the list. Possible values are simple, abstracted and full. If omitted,
the list will be answered in its default scope which is abstracted.
    required: false
    type: string
    enum:
    - simple
    - abstracted
    - full
   - name: class
    in: query
    description: The Device class to be selected for the list.
    required: false
    type: string
   - $ref: '#/parameters/categoryQueryParam'
   - $ref: '#/parameters/aspectQueryParam'
   tags:
   - Device
   - TouchPoint
   - Relations
   - (R)
   responses:
    200:
     description: A Device list.
     schema:
      type: array
      items:
       $ref: '#/definitions/AbstractedDevice'
    default:
     description: Any kind of error

 /devices/create:
  post:
   summary: Device creation
   description: This request creates a Device.
   parameters:
   - name: class
    in: query
    description: The Device class to be created.
    required: true
    type: string
   - name: name
    in: query
    description: The name of the Device to be created.
    required: true
    type: string
   - name: address
    in: query
    description: The address of the Device to be created.
    required: false
    type: string
   tags:
   - Device
   - (C)
   responses:
    200:
     description: Successful Device creation
    default:
     description: Any kind of error

 /devices/destroy:
  delete:
   summary: Device deletion
   description: This request deletes a Device.
   parameters:
   - name: name
    in: query
    description: The name of the Device to be destroyed.
    required: true
    type: string
   tags:
   - Device
   - (D)
   responses:
    200:
     description: Successful Device destruction
    default:
          description: Any kind of error


  /devices/{devicename}:
    get:
      summary: Device retrieval
      description: This request answers a named Device in the form of a JSON object.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      tags:
      - Device
      - (R)
      responses:
        200:
          description: The named Device object.
          schema:
            $ref: '#/definitions/SimpleDevice'
        default:
          description: Any kind of error


  /devices/{devicename}/configure:
    get:
      summary: Device configuration retrieval
      description: This request answers the value(s) of Device configuration slot(s).
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/queryQueryParam'
      tags:
      - Device
      - Configuration
      - (R)
      responses:
        200:
          description: The new value of the parameter.
          schema:
            type: string
        default:
          description: Any kind of error

    put:
      summary: Device configuration update
      description: This request updates the value of a Device configuration slot.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - name: address
        in: query
        description: The value of a &lt;address&gt; configuration slot of a Device.
        required: false
        type: string
      - $ref: '#/parameters/parameterQueryParam'
      - $ref: '#/parameters/valueQueryParam'
      tags:
      - Device
      - Configuration
      - (W)
      responses:
        200:
          description: The new value of the parameter.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/run:
    post:
      summary: Device feature execution
      description: This request triggers the execution of a Device feature.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/methodQueryParam'
      - $ref: '#/parameters/argumentsQueryParam'
      - $ref: '#/parameters/argumentNamesQueryParam'
      tags:
      - Device
      - Feature
      - (X)
      responses:
        200:
          description: Device feature result object.
          schema:
            type: string
        default:
          description: Any kind of error


# TouchPoint node
# ---------------

  /devices/{devicename}/*:
    get:
      summary: TouchPoint listing
      description: >
        This request answers a complete or selected list of all existing TouchPoints for a
        named Device in the form of a JSON array. The array entries are complete JSON representations of
each TouchPoint.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - name: category
        in: query
        description: The TouchPoint category to be selected for the list.
        required: false
        type: string
        enum:
        - digital
        - bitstream
        - analog
        - waveform
        - sensor
        - clock
        - memory
        - id
        - interface
        - bus
      - name: aspect
        in: query
        description: The TouchPoint aspect to be selected for the list.
        required: false
        type: string
      tags:
      - Device
      - TouchPoint
      - Relations
      - (R)
      responses:
        200:
          description: A TouchPoint list.
        default:
          description: Any kind of error

  /devices/{devicename}/{category}/*:
    get:
      summary: TouchPoint category specific channel listing.
      description: >
        This request answers the list of all existing channels for a TouchPoint with a specific category
for a named Device in
        the form of a JSON array. The array entries are JSON objects that give the channel names and
        channel values {channel:value:}.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      tags:
      - TouchPoint
      - (R)
      responses:
        200:
          description: The current channel list of the selected TouchPoint.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{aspect}/*:
    get:
      summary: TouchPoint channel listing.
      description: >
        This request answers the list of all existing channels for a TouchPoint for a named Device in
        the form of a JSON array. The array entries are JSON objects that give the channel names and
        channel values {channel:value:}.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/aspectPathParam'
      tags:
      - TouchPoint
      - (R)
      responses:
        200:
          description: The current channel list of the selected TouchPoint.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{aspect}/{channel}:
    get:
      summary: This request answers a TouchPoint value (full c/a/c path).
      description: >
        This request retrieves the current value of a TouchPoint. The
        concrete TouchPoint is addressed by the path category/aspect/channel.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/aspectPathParam'
      - $ref: '#/parameters/channelPathParam'
      - $ref: '#/parameters/unitQueryParam'
      - $ref: '#/parameters/typeQueryParam'
      tags:
      - TouchPoint
      - (R)
      responses:
        200:
          description: The current value of the selected TouchPoint channel.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{aspect}/{channel}/{value}:
    put:
      summary: This request updates the current value of a TouchPoint (full c/a/c path).
      description: >
        This request writes the current value of a TouchPoint. The
        concrete TouchPoint is addressed by the path category/aspect/channel.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/aspectPathParam'
      - $ref: '#/parameters/channelPathParam'
      - $ref: '#/parameters/valuePathParam'
      - $ref: '#/parameters/valueBodyParam'
      - $ref: '#/parameters/unitQueryParam'
      - $ref: '#/parameters/typeQueryParam'
      tags:
      - TouchPoint
      - (W)
      responses:
        200:
          description: The new value of the selected TouchPoint channel.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{aspect}:
    get:
      summary: This request answers a TouchPoint value (partial c/a path).
      description: >
        This request retrieves the current value of a TouchPoint. The
        concrete TouchPoint is addressed by the path category/aspect.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/aspectPathParam'
      - $ref: '#/parameters/unitQueryParam'
      - $ref: '#/parameters/typeQueryParam'
      tags:
      - TouchPoint
      - (R)
      responses:
        200:
          description: The current value of the selected TouchPoint channel.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{aspect}/{value}:
    put:
      summary: This request updates the current value of a TouchPoint (partial c/a path).
      description: >
        This request writes the current value of a TouchPoint. The
        concrete TouchPoint is addressed by the path category/aspect.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/aspectPathParam'
      - $ref: '#/parameters/valuePathParam'
      - $ref: '#/parameters/valueBodyParam'
      - $ref: '#/parameters/unitQueryParam'
      - $ref: '#/parameters/typeQueryParam'
      tags:
      - TouchPoint
      - (W)
      responses:
        200:
          description: The new value of the selected TouchPoint channel.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{channel}:
    get:
      summary: This request answers a TouchPoint value (partial c/c path).
      description: >
        This request retrieves the current value of a TouchPoint. The
        concrete TouchPoint is addressed by the path category/channel.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/channelPathParam'
      - $ref: '#/parameters/unitQueryParam'
      - $ref: '#/parameters/typeQueryParam'
      tags:
      - TouchPoint
      - (R)
      responses:
        200:
          description: The current value of the selected TouchPoint channel.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{channel}/{value}:
    put:
      summary: This request updates the current value of a TouchPoint (partial c/c path).
      description: >
        This request writes the current value of a TouchPoint. The
        concrete TouchPoint is addressed by the path category/channel.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/channelPathParam'
      - $ref: '#/parameters/valuePathParam'
      - $ref: '#/parameters/valueBodyParam'
      - $ref: '#/parameters/unitQueryParam'
      - $ref: '#/parameters/typeQueryParam'
      tags:
      - TouchPoint
      - (W)
      responses:
        200:
          description: The new value of the selected TouchPoint channel.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}:
    get:
      summary: This request answers a TouchPoint value (partial c path).
      description: >
        This request retrieves the current value of a TouchPoint. The
        concrete TouchPoint is addressed by the path category/channel.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/unitQueryParam'
      - $ref: '#/parameters/typeQueryParam'
      tags:
      - TouchPoint
      - (R)
      responses:
        200:
          description: The current value of the selected TouchPoint channel.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{value}:
    put:
      summary: This request updates the current value of a TouchPoint (partial c path).
      description: >
        This request writes the current value of a TouchPoint. The
        concrete TouchPoint is addressed by the path category/channel.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/valuePathParam'
      - $ref: '#/parameters/valueBodyParam'
      - $ref: '#/parameters/unitQueryParam'
      - $ref: '#/parameters/typeQueryParam'
      tags:
      - TouchPoint
      - (W)
      responses:
        200:
          description: The new value of the selected TouchPoint channel.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{aspect}/{channel}/run:
    post:
      summary: TouchPoint feature execution (full c/a/c path)
      description: This request triggers the execution of a TouchPoint feature.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/aspectPathParam'
      - $ref: '#/parameters/channelPathParam'
      - $ref: '#/parameters/methodQueryParam'
      - $ref: '#/parameters/argumentsQueryParam'
      - $ref: '#/parameters/argumentNamesQueryParam'
      tags:
      - TouchPoint
      - Feature
      - (X)
      responses:
        200:
          description: TouchPoint feature result object.
          schema:
            type: string
        default:
          description: Any kind of error
```

```
/devices/{devicename}/{category}/{aspect}/run:
  post:
    summary: TouchPoint feature execution (partial c/a path)
    description: This request triggers the execution of a TouchPoint feature.
    parameters:
    - $ref: '#/parameters/devicenamePathParam'
    - $ref: '#/parameters/categoryPathParam'
    - $ref: '#/parameters/aspectPathParam'
    - $ref: '#/parameters/methodQueryParam'
    - $ref: '#/parameters/argumentsQueryParam'
    - $ref: '#/parameters/argumentNamesQueryParam'
    tags:
    - TouchPoint
    - Feature
    - (X)
    responses:
      200:
        description: TouchPoint feature result object.
        schema:
          type: string
      default:
        description: Any kind of error


/devices/{devicename}/{category}/{channel}/run:
  post:
    summary: TouchPoint feature execution (partial c/c path)
    description: This request triggers the execution of a TouchPoint feature.
    parameters:
    - $ref: '#/parameters/devicenamePathParam'
    - $ref: '#/parameters/categoryPathParam'
    - $ref: '#/parameters/channelPathParam'
    - $ref: '#/parameters/methodQueryParam'
    - $ref: '#/parameters/argumentsQueryParam'
    - $ref: '#/parameters/argumentNamesQueryParam'
    tags:
    - TouchPoint
    - Feature
    - (X)
    responses:
      200:
        description: TouchPoint feature result object.
        schema:
          type: string
      default:
        description: Any kind of error


/devices/{devicename}/{category}/run:
  post:
    summary: TouchPoint feature execution (partial c path)
    description: This request triggers the execution of a TouchPoint feature.
    parameters:
    - $ref: '#/parameters/devicenamePathParam'
    - $ref: '#/parameters/categoryPathParam'
    - $ref: '#/parameters/methodQueryParam'
    - $ref: '#/parameters/argumentsQueryParam'
    - $ref: '#/parameters/argumentNamesQueryParam'
    tags:
    - TouchPoint
    - Feature
    - (X)
    responses:
      200:
        description: TouchPoint feature result object.
        schema:
          type: string
      default:
        description: Any kind of error


/devices/{devicename}/{category}/{aspect}/{channel}/configure:
  get:
    summary: TouchPoint configuration retrieval (full c/a/c path)
    description: This request answers the value of a TouchPoint configuration slot.
    parameters:
    - $ref: '#/parameters/devicenamePathParam'
    - $ref: '#/parameters/categoryPathParam'
```

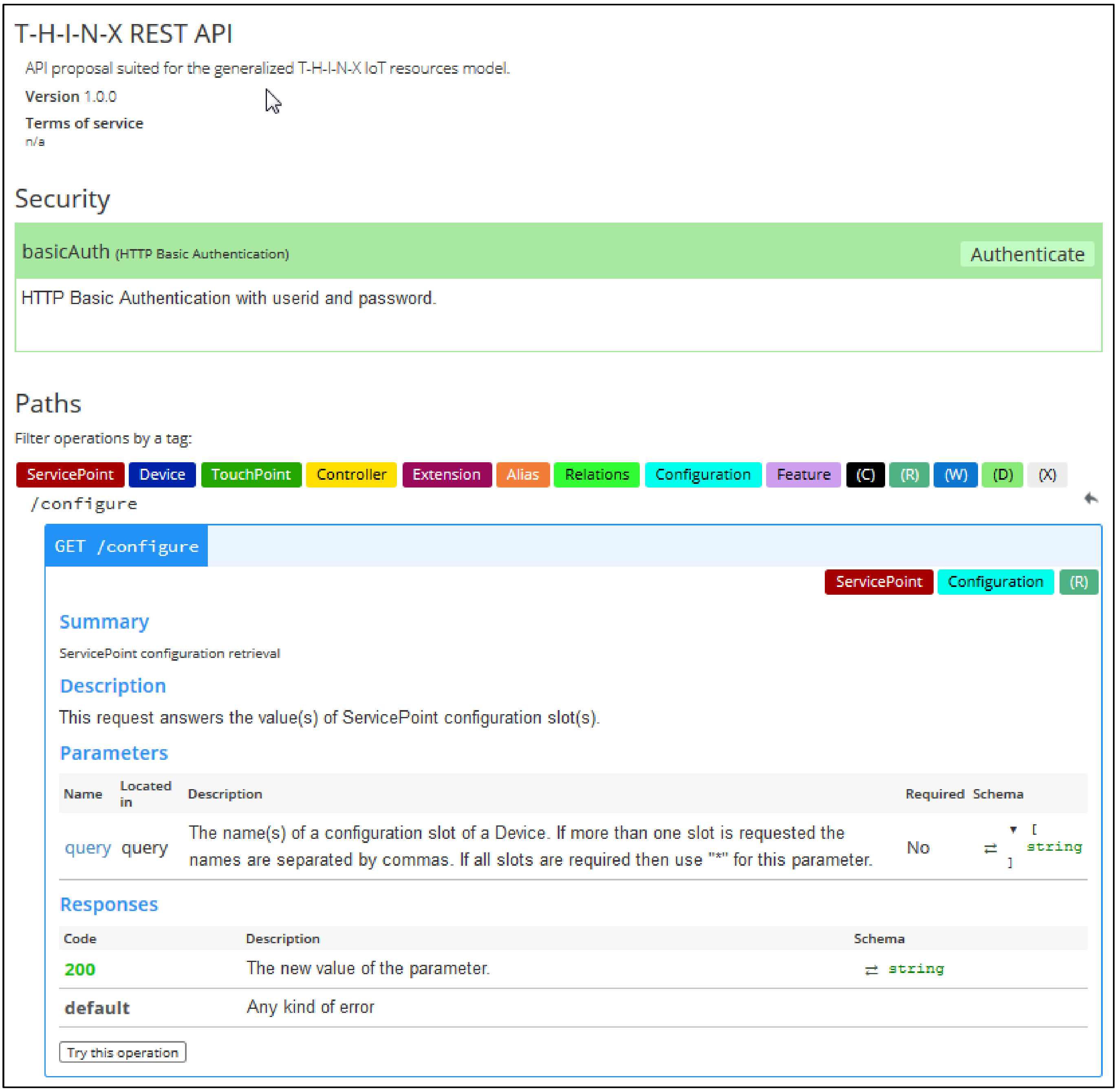
```
      - $ref: '#/parameters/aspectPathParam'
      - $ref: '#/parameters/channelPathParam'
      - $ref: '#/parameters/queryQueryParam'
      tags:
      - TouchPoint
      - Configuration
      - (R)
      responses:
        200:
          description: The new value of the parameter.
          schema:
            type: string
        default:
          description: Any kind of error
    put:
      summary: TouchPoint configuration update (full c/a/c path)
      description: This request updates the value of a TouchPoint configuration slot.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/aspectPathParam'
      - $ref: '#/parameters/channelPathParam'
      - $ref: '#/parameters/functionQueryParam'
      - $ref: '#/parameters/terminationQueryParam'
      - $ref: '#/parameters/detectionQueryParam'
      - $ref: '#/parameters/parameterQueryParam'
      - $ref: '#/parameters/valueQueryParam'
      tags:
      - TouchPoint
      - Configuration
      - (W)
      responses:
        200:
          description: The new value of the parameter.
          schema:
            type: string
        default:
          description: Any kind of error


  /devices/{devicename}/{category}/{aspect}/configure:
    get:
      summary: TouchPoint configuration retrieval (partial c/a path)
      description: This request answers the value(s) of TouchPoint configuration slot(s).
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/aspectPathParam'
      - $ref: '#/parameters/queryQueryParam'
      tags:
      - TouchPoint
      - Configuration
      - (R)
      responses:
        200:
          description: The value(s) of the parameter(s).
          schema:
            type: string
        default:
          description: Any kind of error
    put:
      summary: TouchPoint configuration update (partial c/a path)
      description: This request updates the value of a TouchPoint configuration slot.
      parameters:
      - $ref: '#/parameters/devicenamePathParam'
      - $ref: '#/parameters/categoryPathParam'
      - $ref: '#/parameters/aspectPathParam'
      - $ref: '#/parameters/functionQueryParam'
      - $ref: '#/parameters/terminationQueryParam'
      - $ref: '#/parameters/detectionQueryParam'
      - $ref: '#/parameters/parameterQueryParam'
      - $ref: '#/parameters/valueQueryParam'
      tags:
      - TouchPoint
      - Configuration
      - (W)
      responses:
        200:
          description: The new value of the parameter.
          schema:
```

```
          type: string
      default:
        description: Any kind of error


/devices/{devicename}/{category}/{channel}/configure:
  get:
    summary: TouchPoint configuration retrieval (partial c/c path)
    description: This request answers the value of a TouchPoint configuration slot.
    parameters:
    - $ref: '#/parameters/devicenamePathParam'
    - $ref: '#/parameters/categoryPathParam'
    - $ref: '#/parameters/channelPathParam'
    - $ref: '#/parameters/queryQueryParam'
    tags:
    - TouchPoint
    - Configuration
    - (R)
    responses:
      200:
        description: The new value of the parameter.
        schema:
          type: string
      default:
        description: Any kind of error
  put:
    summary: TouchPoint configuration update (partial c/c path)
    description: This request updates the value of a TouchPoint configuration slot.
    parameters:
    - $ref: '#/parameters/devicenamePathParam'
    - $ref: '#/parameters/categoryPathParam'
    - $ref: '#/parameters/channelPathParam'
    - $ref: '#/parameters/functionQueryParam'
    - $ref: '#/parameters/terminationQueryParam'
    - $ref: '#/parameters/detectionQueryParam'
    - $ref: '#/parameters/parameterQueryParam'
    - $ref: '#/parameters/valueQueryParam'
    tags:
    - TouchPoint
    - Configuration
    - (W)
    responses:
      200:
        description: The new value of the parameter.
        schema:
          type: string
      default:
        description: Any kind of error


/devices/{devicename}/{category}/configure:
  get:
    summary: TouchPoint configuration retrieval (partial c path)
    description: This request answers the value of a TouchPoint configuration slot.
    parameters:
    - $ref: '#/parameters/devicenamePathParam'
    - $ref: '#/parameters/categoryPathParam'
    - $ref: '#/parameters/queryQueryParam'
    tags:
    - TouchPoint
    - Configuration
    - (R)
    responses:
      200:
        description: The new value of the parameter.
        schema:
          type: string
      default:
        description: Any kind of error
  put:
    summary: TouchPoint configuration update (partial c path)
    description: This request updates the value of a TouchPoint configuration slot.
    parameters:
    - $ref: '#/parameters/devicenamePathParam'
    - $ref: '#/parameters/categoryPathParam'
    - $ref: '#/parameters/functionQueryParam'
    - $ref: '#/parameters/terminationQueryParam'
    - $ref: '#/parameters/detectionQueryParam'
    - $ref: '#/parameters/parameterQueryParam'
    - $ref: '#/parameters/valueQueryParam'
```

```
      tags:
      - TouchPoint
      - Configuration
      - (W)
      responses:
        200:
          description: The new value of the parameter.
          schema:
            type: string
        default:
          description: Any kind of error


# Extension node
# --------------

  /extensions/{extensionname}:
    get:
      summary: Extension retrieval
      description: This request answers a named Extension in the form of a JSON object.
      parameters:
      - $ref: '#/parameters/extensionnamePathParam'
      tags:
      - Extension
      - (R)
      responses:
        200:
          description: The named Extension object.
          schema:
            type: string
        default:
          description: Any kind of error


  /extensions/{extensionname}/run:
    post:
      summary: Extension execution
      description: >
        This request triggers the execution of an Extension.
        If the Extension has one ore more arguments their values can be submitted as comma-separated list.
        If additionally argument names can be submitted, this can be added by a comma-separated
        list of those names. The sequence of the argument values and argument names has to match.
      parameters:
      - $ref: '#/parameters/extensionnamePathParam'
      - $ref: '#/parameters/methodQueryParam'
      - $ref: '#/parameters/argumentsQueryParam'
      - $ref: '#/parameters/argumentNamesQueryParam'
      tags:
      - Extension
      - (X)
      responses:
        200:
          description: Extension execution result object.
          schema:
            type: string
        default:
          description: Any kind of error


# Controller node
# ---------------

  /controllers/{controllername}/configure:
    get:
      summary: Controller configuration retrieval
      description: This request answers the value of a Controller configuration slot.
      parameters:
      - $ref: '#/parameters/controllernamePathParam'
      - $ref: '#/parameters/queryQueryParam'
      tags:
      - Controller
      - Configuration
      - (R)
      responses:
        200:
          description: The new value of the parameter.
          schema:
            type: string
        default:
          description: Any kind of error
    put:
      summary: Controller configuration update
      description: This request updates the value of a Controller configuration slot.
      parameters:
      - $ref: '#/parameters/controllernamePathParam'
      - $ref: '#/parameters/parameterQueryParam'
      - $ref: '#/parameters/valueQueryParam'
      tags:
      - Controller
      - Configuration
      - (W)
      responses:
        200:
          description: The new value of the parameter.
          schema:
            type: string
        default:
          description: Any kind of error


  /controllers/{controllername}/run:
    post:
      summary: Controller feature execution
      description: This request triggers the execution of a Controller feature.
      parameters:
      - $ref: '#/parameters/controllernameParam'
      - $ref: '#/parameters/methodQueryParam'
      - $ref: '#/parameters/argumentsQueryParam'
      - $ref: '#/parameters/argumentNamesQueryParam'
      tags:
      - Controller
      - Feature
      - (X)
      responses:
        200:
          description: Device feature result object.
          schema:
            type: string
        default:
          description: Any kind of error


# Alias node
# ----------

  /{substitutionpath}:
    get:
      summary: Alias retrieval
      description: This request answers the value of an Alias path.
      parameters:
      - name: substitutionpath
        in: path
        description: The value of the &lt;substitution&gt; slot of an Alias.
        required: true
        type: string
      tags:
      - Alias
      - (R)
      responses:
        200:
          description: The result of the aliased request.
          schema:
            type: string
        default:
          description: Any kind of error


  /{substitutionpath}/{value}:
    put:
      summary: Alias update
      description: This request updates the value of the target of an Alias path.
      parameters:
      - name: substitutionpath
        in: path
        description: The value of the &lt;substitution&gt; slot of an Alias.
        required: true
        type: string
      - name: value
        in: path
        description: The new value of the target of an aliased path.
        required: true
        type: string
      tags:
      - Alias
      - (W)
      responses:
        200:
          description: The new value of the aliased request.
          schema:
            type: string
        default:
          description: Any kind of error


# Object model JSON schema definitions
# --------------------------------------

# Remark: Just a few for now to show how this works, the final version has to refer to the full T-H-I-N-X
JSON schema.

definitions:

# Device node
# -----------

  SimpleDevice:
    required:
    - name
    - class
    - address
    properties:
      name:
        type: string
        description: The value of the &lt;name&gt; slot of a Device.
      class:
        type: string
        description: The value of the &lt;class&gt; slot of a Device.
      address:
        type: string
        description: The value of the &lt;address&gt; slot of a Device.

  AbstractedDevice:
    required:
    - name
    - type
    properties:
      name:
        type: string
        description: The value of the &lt;name&gt; slot of a Device.
      type:
        type: string
        description: The value of the abstraction of a TouchPoint which is a unique value derived from the
&lt;category&gt; and/or &lt;aspect&gt; slot of a TouchPoint.
        enum:
        - Digital
        - Analog
        - PWM
        - Memory
        - Temperature
        - Pressure
        - Luminosity
        - Distance
        - Position
        - Acceleration
        - Gear
        - Orientation
        - Velocity
        - Humidity
        - Power
        - Current
        - Voltage
        - Frequency
        - Clock
        - ID
        - Interface
        - Bus
```

Swagger UI toolset representation

The following picture shows a partial screenshot of the visual representation that gets generated out of the Swagger spec:



T-H-I-N-X REST API

API proposal suited for the generalized T-H-I-N-X IoT resources model.

Version 1.0.0

Terms of service
n/a

Security

basicAuth (HTTP Basic Authentication)

HTTP Basic Authentication with userid and password.

Paths

Filter operations by a tag:

/configure

GET /configure

Summary

ServicePoint configuration retrieval

Description

This request answers the value(s) of ServicePoint configuration slot(s).

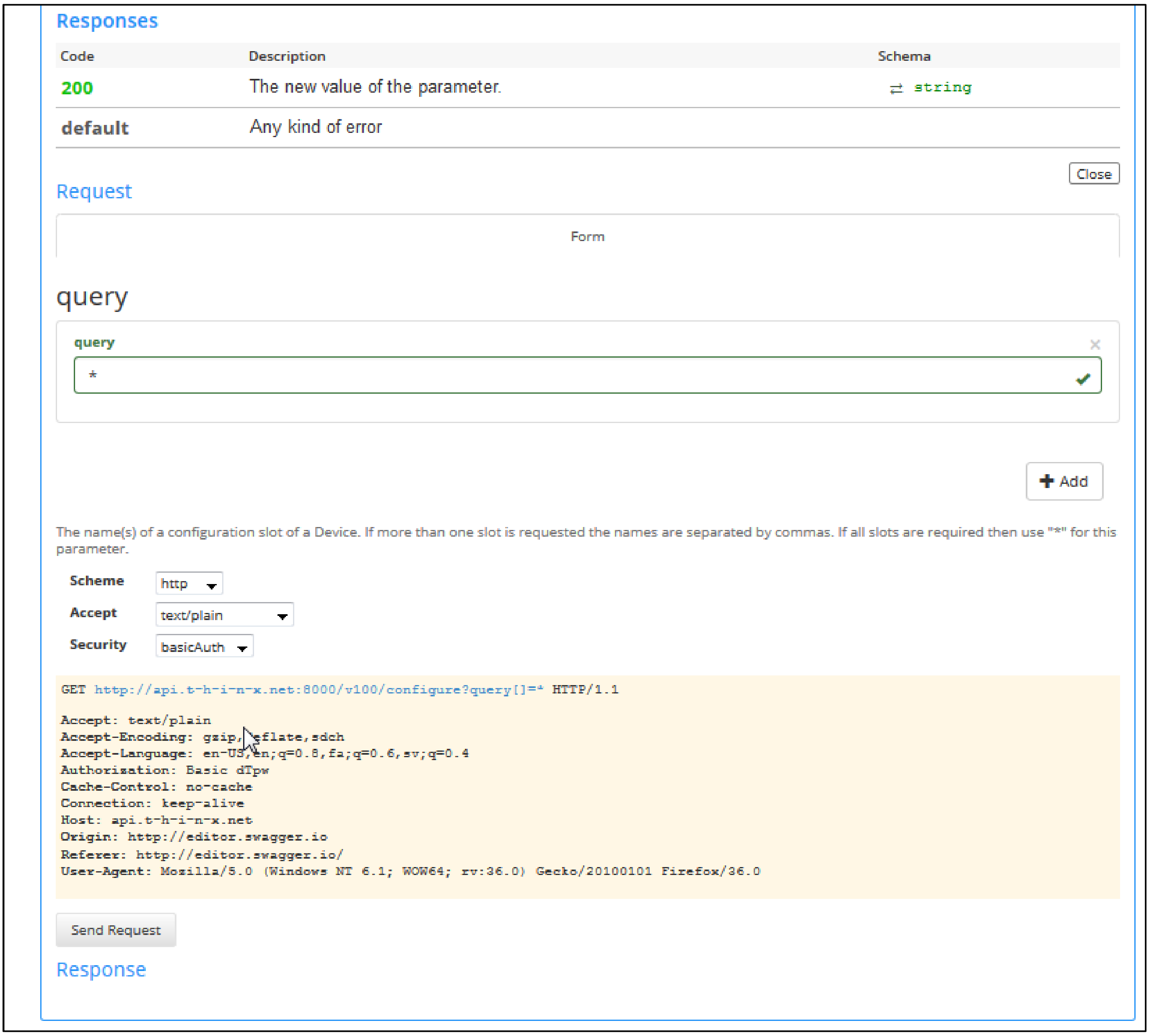
Parameters

Name	Located in	Description	Required	Schema
query	query	The name(s) of a configuration slot of a Device. If more than one slot is requested the names are separated by commas. If all slots are required then use "*" for this parameter.	No	⇄ ▾ [string]

Responses

Code	Description	Schema
200	The new value of the parameter.	⇄ string
default	Any kind of error	

If you click on “Try this operation” the following screen gets folded down for every API path:



Responses

Code	Description	Schema
200	The new value of the parameter.	⇄ string
default	Any kind of error	

Request

query

query: *

The name(s) of a configuration slot of a Device. If more than one slot is requested the names are separated by commas. If all slots are required then use "*" for this parameter.

Scheme: http

Accept: text/plain

Security: basicAuth

```
GET http://api.t-h-i-n-x.net:8000/v100/configure?query[]=* HTTP/1.1

Accept: text/plain
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,fa;q=0.6,sv;q=0.4
Authorization: Basic dTpw
Cache-Control: no-cache
Connection: keep-alive
Host: api.t-h-i-n-x.net
Origin: http://editor.swagger.io
Referer: http://editor.swagger.io/
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:36.0) Gecko/20100101 Firefox/36.0
```

Response

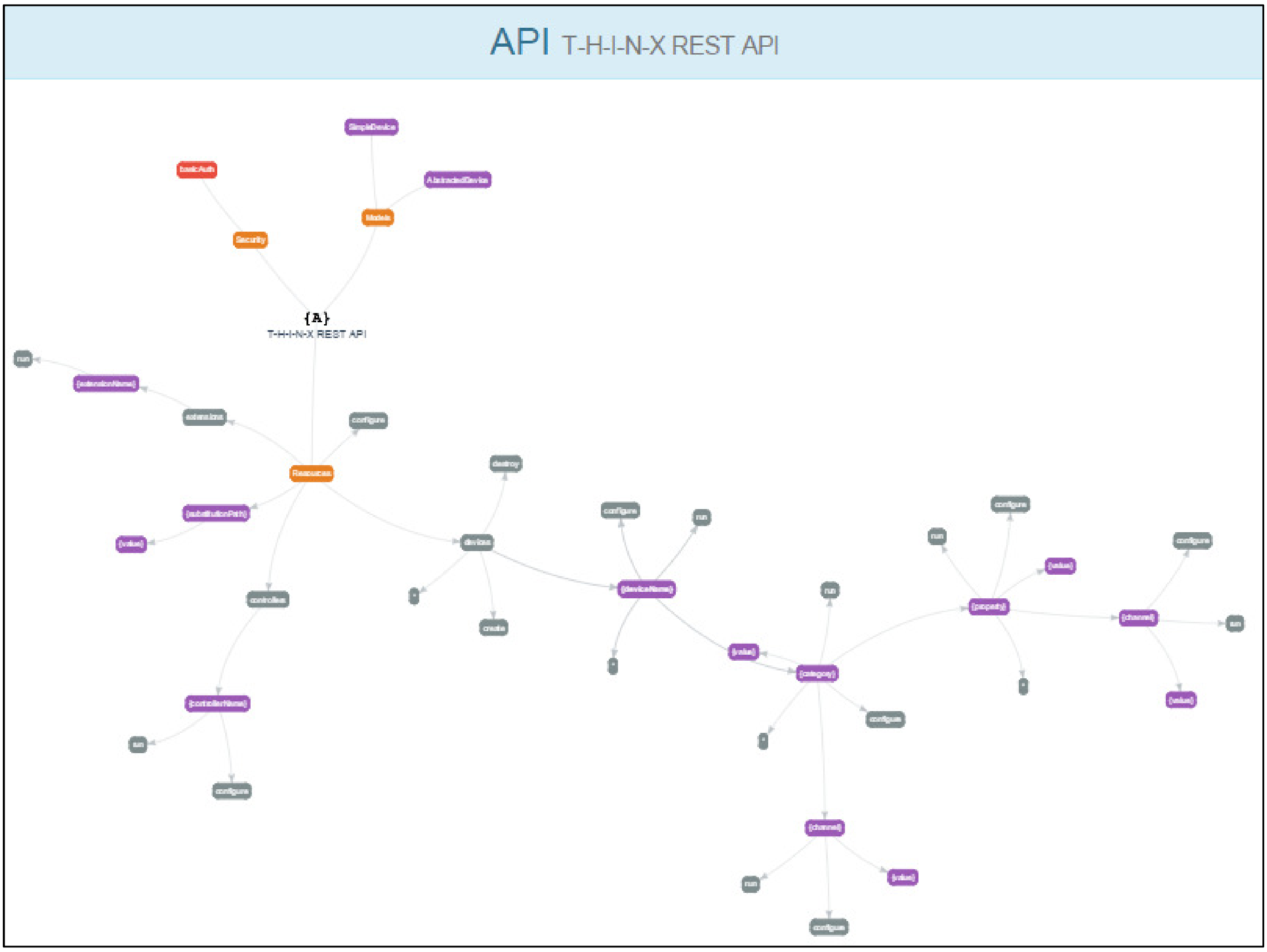
Here you can add the values for parameters (here, the “*” for the query parameter has been entered), see the generated resulting API request, and, if clicking on “Send Request”, you can see the live result from this request (which does not work in the example from above as the host “api.t-h-i-n-x.net” is just a placeholder and does not exist in reality).

TODO: Update screenshot when [] bug gets corrected from Swagger UI.

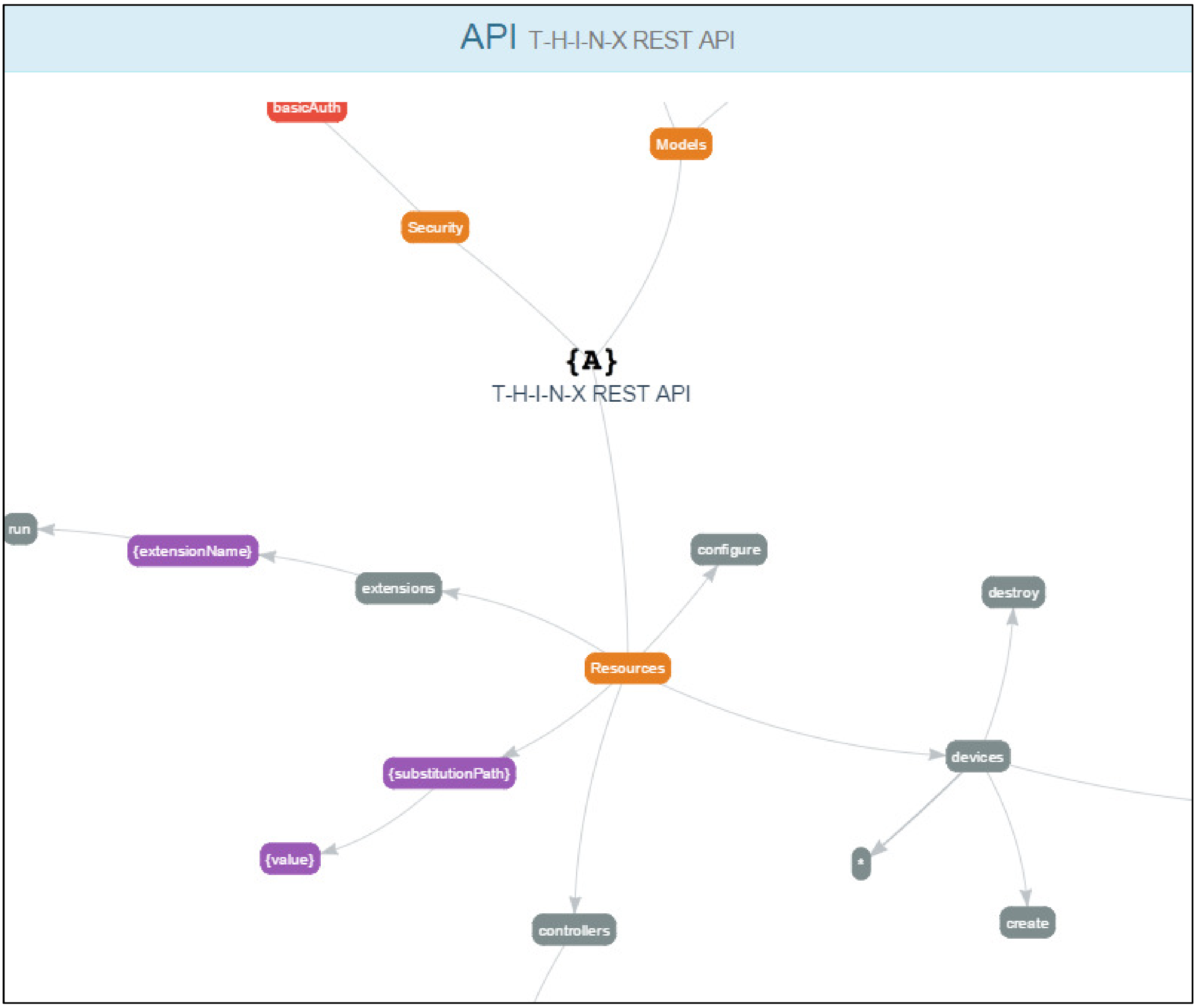
Swagger.ed toolset representation

In addition to the tool from above another tool exists that allows a very dynamic interaction with Swagger specs called “swagger.ed” [SWAGGER.ED_2015]. It gets delivered in the form of an extension for the Chrome browser.

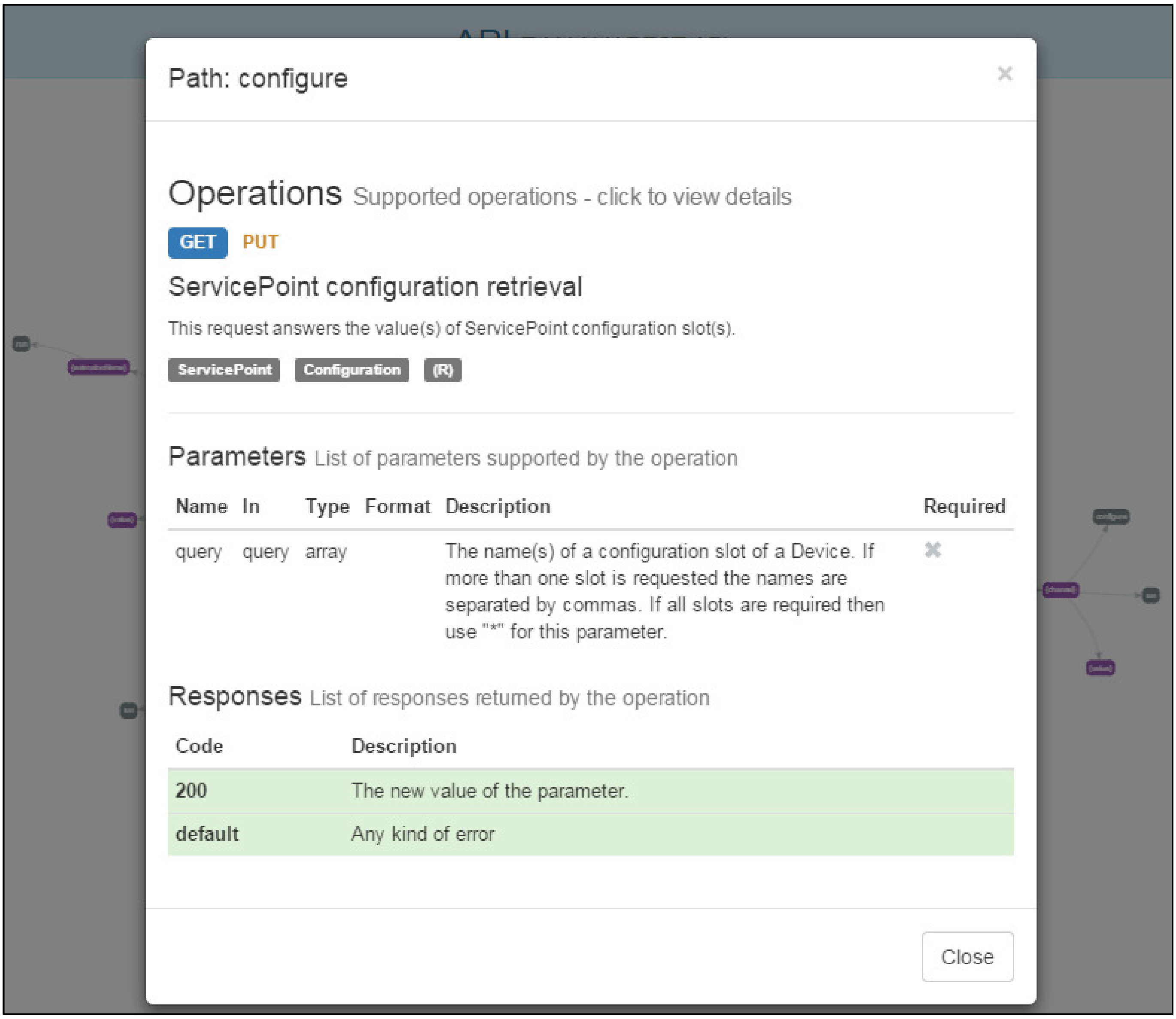
The following picture shows a screenshot of the visual representation that gets generated by this tool out of the Swagger spec:



You get the complete API visualized in the form of a dynamically interactive graph. As the T-H-I-N-X API is already very rich in details it is a bit hard to recognize that in this screenshot. However, it is possible to zoom into the view like this (upper left quadrant of the view above) and see much more details in a readable form:



Now, if you double-click on one of the nodes (e.g. onto the node “configure” north-east from the “Resources” node that represents the root path), you will get a detail view that resembles the API view of the Swagger UI toolset:



Path: configure

Operations Supported operations - click to view details

ServicePoint configuration retrieval

This request answers the value(s) of ServicePoint configuration slot(s).

Parameters List of parameters supported by the operation

Name	In	Type	Format	Description	Required
query	query	array		The name(s) of a configuration slot of a Device. If more than one slot is requested the names are separated by commas. If all slots are required then use "*" for this parameter.	✖

Responses List of responses returned by the operation

Code	Description
200	The new value of the parameter.
default	Any kind of error

This allows a very comfortable exploring navigation of any API.

JSON mapping and serialization

The abstract resource model can be mapped to a corresponding JSON object structure.

JSON instance

An example for a possible JSON structure starting at the ServicePoint would be as follows. The nodes of the resource model are marked in the color of the nodes from the resource model. X_Relation information is marked yellow. X_Features aspects are not covered.

The example data for the JSON node values is derived from the WebIOPi implementation where possible.

```
{
   "thinx" : {
      "$version" : "2.0",
      "$copyright" : "Copyright 2014 Andreas Riegg",
      "$license" : "Apache 2.0",
      "servicepoint" : {
         "aliases" : [
            {
               "alias" : {
                  "original" : "/devices/mcp1/digital/2",
                  "substitution" : "/kitchen/switch/ceiling"
               }
            },
            {
               "alias" : {
                  "original" : "/devices/mcp1/digital/5",
                  "substitution" : "/kitchen/switch/wall"
               }
            },
            {
               "alias" : {
                  "original" : "/devices/mcp1/digital/4",
                  "substitution" : "/kitchen/lamp/ceiling"
               }
            },
            {
               "alias" : {
                  "original" : "/devices/mcp1/digital/6",
                  "substitution" : "/kitchen/lamp/wall"
               }
            }
         ],
         "controllers" : [
            {
               "controller" : {
                  "bindings" : [
                     {
                        "connection" : "http://192.168.0.2:8000"
                     },
                     {
                        "connection" : "coap://192.168.0.2:5683"
                     },
                     {
                        "connection" : "coap://224.0.1.123:5683"
                     }
                  ],
                  "firmwares" : [
                     {
                        "firmware" : {
                           "os" : "linux-wheezy-armv6l",
                           "platform" : "armhf-raspberrypi",
                           "server" : "WebIOPi-0.7.0-py2.7"
                        }
                     }
                  ],
                  "parameters" : [
                     {
                        "name" : "board_revision",
                        "value" : "2"
                     },
                     {
                        "name" : "http-doc-root",
                        "value" : "/home/pi/webiopi/examples/scripts/macros"
                     },
                     {
                        "name" : "http-welcome-file",
                        "value" : "index.html"
                     },
                     {
                        "name" : "coap-multicast",
                        "value" : true
                     },
                     {
                        "name" : "http-doc-root",
                        "value" : "/home/pi/webiopi/examples/scripts/macros"
                     }
                  ],
                  "processes" : [
                     {
                        "process" : {
                           "name" : "main",
                           "pid" : "1234"
                        }
                     }
                  ],
                  "threads" : [
                     {
                        "thread" : {
                           "ppid" : "1234",
                           "tid" : "__MAIN__"
                        }
                     },
                     {
                        "thread" : {
                           "ppid" : "1234",
                           "tid" : "HTTPThread"
                        }
                     },
                     {
                        "thread" : {
                           "ppid" : "1234",
                           "tid" : "COAPThread"
                        }
                     },
                     {
                        "thread" : {
                           "ppid" : "1234",
                           "tid" : "LoopTask_0"
                        }
                     },
                     {
                        "thread" : {
                           "ppid" : "1234",
                           "tid" : "LoopTask_1"
                        }
                     }
                  ]
               }
            }
         ],
         "devices" : [
            {
               "device" : {
                  "address" : {
                     "busType" : "I2C",
                     "busId" : "1",
                     "deviceId" : "0x48"
                  },
                  "class" : "TMP275",
                  "name" : "temp1",
                  "parameters" : [
                     {
                        "name" : "resolution",
                        "value" : 11
                     }
                  ],
                  "touchpoints" : [
                     {
                        "touchpoint" : {
                           "category" : "sensor",
                           "aspect" : "temperature",
                           "channel" : null,
                           "unit" : "C",
                           "value" : 24.69
                        }
                     }
                  ]
               }
            },
            {
               "device" : {
                  "address" : {
                     "busType" : "I2C",
                     "busId" : "1",
                     "deviceId" : "0x40"
                  },
                  "class" : "BMP085",
                  "name" : "bmp1",
                  "parameters" : [
                     {
                        "name" : "altitude",
                        "value" : 150
                     },
                     {
                        "name" : "external",
                        "value" : "temp1"
                     }
                  ],
                  "touchpoints" : [
                     {
                        "touchpoint" : {
                           "category" : "sensor",
                           "aspect" : "pressure",
                           "channel" : null,
                           "unit" : "hPa",
                           "value" : 1003.69
                        }
                     },
                     {
                        "touchpoint" : {
                           "category" : "sensor",
                           "aspect" : "pressure",
                           "channel" : "sea",
                           "unit" : "hPa",
                           "value" : 996.91
                        }
                     },
                     {
                        "touchpoint" : {
                           "category" : "sensor",
                           "aspect" : "temperature",
                           "channel" : null,
                           "unit" : "K",
                           "value" : 280.0
                        }
                     }
                  ]
               }
            },
            {
               "device" : {
                  "address" : {
                     "busType" : "I2C",
                     "busId" : "1",
                     "deviceId" : "0x20"
                  },
                  "class" : "MCP23008",
                  "name" : "mcp1",
                  "touchpoints" : [
                     {
                        "touchpoint" : {
                           "category" : "digital",
                           "aspect" : null,
                           "channel" : 0,
                           "function" : "out",
                           "value" : 1
                        }
                     },
                     {
                        "touchpoint" : {
                           "category" : "digital",
                           "aspect" : null,
                           "channel" : 1,
                           "function" : "out",
                           "value" : 0
                        }
                     },
                     {
                        "touchpoint" : {
                           "category" : "digital",
                           "aspect" : null,
                           "channel" : 2,
                           "function" : "in",
                           "value" : 1
                        }
                     },
                     {
                        "touchpoint" : {
                           "category" : "digital",
                           "aspect" : null,
                           "channel" : 3,
                           "function" : "out",
                           "value" : 1
                        }
                     },
                     {
                        "touchpoint" : {
                           "category" : "digital",
                           "aspect" : null,
                           "channel" : 4,
                           "function" : "out",
                           "value" : 0
                        }
                     },
                     {
                        "touchpoint" : {
                           "category" : "digital",
                           "aspect" : null,
                           "channel" : 5,
                           "function" : "in",
                           "value" : 1
                        }
                     },
                     {
                        "touchpoint" : {
                           "category" : "digital",
                           "aspect" : null,
                           "channel" : 6,
                           "function" : "out",
                           "value" : 1
                        }
                     },
                     {
                        "touchpoint" : {
                           "category" : "digital",
                           "aspect" : null,
                           "channel" : 7,
                           "function" : "in",
                           "value" : 0
                        }
                     }
                  ]
               }
            }
         ],
         "extensions" : [
            {
               "extension" : {
                  "name" : "myExtension1"
               }
            },
            {
               "extension" : {
                  "arguments" : [
                     {
                        "arg1" : "myArg1"
                     }
                  ],
                  "name" : "myExtension2"
               }
            },
            {
               "extension" : {
                  "arguments" : [
                     {
                        "arg1" : "myArg1"
                     },
                     {
                        "arg2" : "myArg2"
                     },
                     {
                        "arg3" : "myArg3"
                     }
                  ],
                  "name" : "myExtension3",
                  "result" : {
                     "class" : "Integer",
                     "mime-type" : "text/plain"
                  }
               }
            }
         ],
         "uris" : [
            {
               "uri" : "http://mydomain.local:8000"
            },
            {
               "uri" : "coap://mydomain.local:5683"
            },
            {
               "uri" : "coap://224.0.1.123/5683"
            }
         ]
      }
   }
}
```

Remark 1: This just one possible structure, but not the only one. However it is one that tries to cover as much as possible elements of the resource model.

Remark 2: Some JSON nodes could be simplified. In the version above, “named” JSON objects are used for the 1:n relations. This can be simplified by omitting those Object “names”, e.g. instead of this solution for “touchpoints:” (see cursive parts)

```
            ...
            {
               "device" : {
                  "address" : {
                     "busId" : "1",
                     "busType" : "I2C",
                     "slave" : "0x48"
                  },
                  "class" : "TMP275",
                  "name" : "temp1",
                  "parameters" : [
                     {
                        "name" : "resolution",
                        "value" : 11
                     }
                  ],
                  "touchpoints" : [
                     {
                        "touchpoint" : {
                           "category" : "sensor",
                           "aspect" : "temperature”,
                           "channel" : null,
                           "unit" : "c",
                           "value" : 24.69
                        }
                     }
                  ]
               }
            },
            ...
```

this one could be used:

```
...
{
   "device" : {
      "address" : {
         "busId" : "1",
         "busType" : "I2C",
         "slave" : "0x48"
      },
      "class" : "TMP275",
      "name" : "temp1",
      "parameters" : [
         {
            "name" : "resolution",
            "value" : 11
         }
      ],
      "touchpoints" : [
         {
            "category" : "sensor",
            "aspect" : "temperature”,
            "channel" : null,
            "unit" : "c",
            "value" : 24.69
         }
      ]
   }
},
...
```

The JSON structure above is a kind of verbose form as it uses explicit names for the relation sub nodes like “device” : { ... } for entries in the “devices” list. This can be simplified by omitting the extra JSON node and just putting entries of JSON objects having the device-slots as keys.

When using the JSON structure above together with a REST API, navigation paths of the resource model like “ServicePoint -> Device -> TouchPoint” can be addressed via a mapping URI path and the remainder is serialized to JSON object structures. Ideally the paths from the URI and the nested JSON structure are very similar. 1:n relations can be mapped to paths by using all identifying slots as parameters leading to the already mentioned scheme

URI/device/{name}/{category}/{aspect}/{channel}

and by accessing all objects of a 1:n relation with the wildcard “*”

`URI/devices/*` -> all devices

`URI/devices/{deviceName}/*` .> all touch points of a device

`URI/devices/{deviceName}/{category}/{aspect}/*` -> all channels of a touch point of a device

JSON schema

The JSON structure above can be described formal with the following JSON schema [JSCHEM_2014]:

More text tbd.

Docson reference

jsonschema.net reference

JSON pointers?

WebIOPi REST API mapping and Python implementation for HTTP and CoAP

Overview

WebIOPi is an Internet of Things framework that has been developed by Eric Ptak. It is implemented in Python and available from this source in version 0.7.0 [WEBIOPI]. Currently it supports bindings for the HTTP and CoAP protocol. WebIOPi runs on the Raspberry Pi and allows connecting around 30 types of devices to the web.

Mapping concepts

The primary mapping concepts used in WebIOPi are closely related to the abstract resource model as it is an abstraction of what was done first with WebIOPi. This means that the six main resource nodes exist, however not having all the details of them. Additionally, X_Configuration and X_Features are partly implemented with WebIOPi config file options and Python server methods but they are currently not mapped via REST APIs.

ServicePoint

Three (each optional) URIs exist: One for HTTP and two for CoAP (one like HTTP bound to the local IP address and another bound on the multicast address “224.0.1.123”). From a ServicePoint can be navigated to Devices, Aliases, Controller and Extensions. For authentication just “HTTP Basic Authentication” is supported. Encryption is not available as well as no discoverables, no subscriptions and no events.

The path encoded mappings from the ServicePoint URIs are:

- The root for the REST API is just `URI/...`
- **ServicePoint <-> Device:** `URI/devices/...` with exception for native GPIOs which are direct linked to `URI/GPIO/...`
- **ServicePoint <-> Alias:** Available as user-definable routes but no prefix is used `URI/...`
- **ServicePoint <-> Extension:** `URI/macros/...`
- **ServicePoint <->Controller:** Some direct paths (e.g. `URI/version`)

Device

Devices are called devices in WebIOPi so there is a direct 1:1 implementation of this node. Their <names> are user-defined text strings. They also have a <class> which is defined within the Python device driver class hierarchy.

Device objects are mapped to the REST API in this way `URI/devices/{deviceName}/...` with the exception of native GPIO ports as mentioned above.

Devices implement all of the X_Configuration and X_Features sub node slots, but they are not mapped via the REST API. They are partly available with WebIOPi config file options and/or Python server calls.

TouchPoint

TouchPoints don’t exist as separate objects in WebIOPi but they are an intrinsic element of the WebIOPi devices. They are mapped to the REST API in this way:

`URI/devices/{deviceName}/{category}/{channel}/{unit}...`

Within this pattern the following rules apply:

- For <category> the variants “analog”, “pwm”, “sensor/temperature”, “sensor/luminosity”, “sensor/distance” and “sensor/humidity” are available like here: `URI/devices/myLightSensor/sensor/luminosity/...`. The direct category “digital” for native GPIO ports and digital I/O devices is missing. In the case of the driver for the PiFace shield, the categories “digital/input”, “digital/output” are added and channels with integer numbers are existent.

- <channel> is used with integer numbers starting at 0 for analog and digital I/O devices being multichannel `.../analog/2/...`. For native GPIO ports the channel number is the GPIO number of the chip `.../GPIO/17/...`. They are also integers but not all integers in the natural sequence are used. In the case of the pressure sensor an additional channel giving the pressure at sea level has been implemented `.../pressure/sea/...`. In this case, the channel is a text string.

- <unit> is used in selected cases with
  - (partly abbreviated) physical units for sensors `.../sensor/luminosity/lux` and
  - complete strings of mathematical units for others `.../analog/3/float`.

- <runnables> are implemented only for native GPIO ports to
  - output a single pulse `URI/GPIO/17/pulse` or
  - a bit sequence `URI/GPIO/17/sequence/10,01001`.

- <function> can be set for native GPIO ports and digital I/O ports. Usage is encoded to the URI path `.../GPIO/17/function/...`

<termination> and <parameters> can be partly set via WebIOPi server config file or Python server library calls but are not mapped to REST API.

Alias

Alias are implemented in the form of logical routes that are defined in the WebIOPi server config file.

Extension

Extensions are called macros in WebIOPi. They are implemented by separate Python source files that are configured via entries in the WebIOPI server config file. Macro support is very rich, it is possible to use macros with our without arguments. It is also possible to get the results of macros. The mapping is using this pattern `URI/macros/{macroName}/{macroArguments}`. The usage of macroArguments is optional, if more than one argument is needed, then the arguments are separated by commas `URI/macros/myMacro/1,second,THIRD`.

Controller

Naturally, a Controller exists within WebIOPi (this is the running WebIOPi server itself) but it is not explicitly supported with dedicated resource paths. Only some direct paths exist that provide very basic information as already mentioned above. Besides the REST API, some slots of the Controller can be set via WebIOPi config file options. Navigation from the Controller to the Devices is possible with Python server calls.

Miscellaneous

Other important concepts used related to the REST API via HTTP and CoAP are:

- Reading values is mapped to the HTTP/CoAP verb “GET”

- Writing values is mapped to the HTTP/CoAP verb “POST” with URL-encoded data, no HTTP payload for POST is used
- The special path component “*” is partly used to be a wildcard when selecting resources like `URI/devices/*` which answers all configured devices as JSON list
- HTTP/CoAP results are either “text/plain” or “application/json”. This is fixed for all calls; no way of selecting the result type is available.

Resources

For the complete mapping of WebIOPi some formal specs and derived representations are available. They are listed here.

The full API specification based on WADL [WADL_2009; WADL_2014] can be found within the appendices document [THINX_2014] and ***here***. With the help of a modified XSLT-Stylesheet [WADL_2012] it can be transformed to a HTML documentation that can be found within the appendices document [THINX_2014] and ***here***. The WADL specification is also “ApiGee”-augmented so that it can be also used to populate an ApiGee Console-To-Go [APIGEE_2014].

Additionally, the full API specification based on Swagger [SWA_2014] can be found within the appendices document [THINX_2014] and ***here***. It can also be used to provide a self-contained WebIOPi-hosted console with this package that is available ***here***.

XML mapping and serialization

XML instance

Text tbd.

XML schema

Text tbd.

WebSockets API usage pattern

Text tbd.

SOAP API usage pattern

WSDL

XML (rendered) payload

Text tbd.

MQTT API usage pattern

Text tbd.

Relation to openHAB, eclipse smarthome and iot.eclipse.org

openHAB

A Things-Server can be used as binding provider for an openHAB based central control instance. The simplest possibility is to do this via a HTTP binding. Another possibility may be a CoAP binding but this will have an openHAB CoAP support as prerequisite which is not available so far.

The natural point for a Things-Server to interact with openHAB is its “Event Bus” and the associated bindings. The <output> slot of the ServicePoint is related to the state updates of the Event Bus. The <input> slot of the ServicePoint is related to the commands of the Event Bus.

More text tbd.

eclipse smarthome

Relation to Eclipse Smart Home Gateway initiative

Text tbd.

iot.eclipse.org

Text tbd.

Relation to IoT targeting public (industrial) initiatives

Initiative EEBUS

Text tbd.

industrial internet CONSORTIUM

Text tbd.

Industrie 4.0

Text tbd.

ALLSEEN ALLIANCE

Text tbd.

Relation to XPath

Text tbd.

Relation to OData

[ODATA_2014]

Text tbd.

Appendices

In order to keep this document small the printout of all referenced resources is gathered in a separate document [THINX_2014]. These printouts are:

- WADL spec for WebIOPi
- HTML-rendered documentation (excerpts) of WebIOPi REST API
- Swagger 1.2 spec for WebIOPi
- Swagger console (excerpts) of WebIOPi

References

- [WEBIOPI_2014]: WebIOPi project homepage (http://code.google.com/p/webiopi/)
- [THINXNET_2014]: T-H-I-N-X.NET resources homepage (http://www.t-h-i-n-x.net)
- [THINX_2014]: IoT Model Appendices document
- [IOMOTIX_2014]: iomotix project homepage (http://www.iomotix.com/)
- [GUIN_2011]: PhD thesis of Dominique Guinard (http://www.webofthings.org/dom/thesis.pdf)
- [FIELD_2000]: PhD thesis of Roy Fielding (https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf)
- [DTREE_2014]: Device Tree specifications (http://www.devicetree.org)
- [JSON_2014]: JSON specification (http://json.org)
- [JSCHEM_2014]: JSON schema specifications and tools (http://json-schema.org)
- [CORE_2014]: Core and CoAP specifications (https://datatracker.ietf.org/wg/core/)
- [ODATA_2014]: OData resources and specifications (http://www.odata.org/)
- [WEBSOCK_2014]: WebSockets specifications (http://www.websocket.org)
- [OHAB_2014]: openHAB project (http://www.openhab.org)
- [OSGI_2014]: OSGi specifications (http://www.osgi.org)
- [ESH_2014]: Eclipse Smart Home project (http://www.eclipse.org/smarthome/)
- [MQTT_2014]: MQTT specifications (http://mqtt.org/)
- [SOAP_2000]: SOAP specifications (http://www.w3.org/2000/xp/Group)
- [WSDL_2001]: WSDL specifications (http://www.w3.org/TR/wsdl)
- [WADL_2014]: WADL infos (http://wadl.java.net/)
- [WADL_2009]: WADL specification (http://www.w3.org/Submission/wadl/)
- [WADL_2012]: WADL.XSL tool (https://github.com/ipcsystems/wadl-stylesheet)
- [APIGEE_2014]: APIGEE Papers and Console-To-Go tool (http://apigee.com/)
- [SWAGGER_2014]: Swagger project (http://github.com/wordnik/swagger-core/wiki)
- [SWAGGER_EDITOR_2015]: Swagger editor online tool (http://editor.swagger.io)
- [SWAGGER.ED_2015]: swagger.ed toolset http://chefarchitect.github.io/swagger.ed/features/apis-json-support/
- [YAML_2015]: YAML Ain't Markup Language (http://www.yaml.org)
- [SOAPUI_2014]: SoapUI tool (http://www.soapui.org)
- [ARDU_2014]: Arduino platform (http://www.arduino.cc/)
- [RASP_2014]: Raspberry PI product (http://www.raspberrypi.org/)
- [BEAGLE_2014]: Beagle board products (http://beagleboard.org/)
- [EIMP_2014]: Electric Imp product (https://electricimp.com)
- [WEAVED_2015]: Weaved IoT toolkit (http://www.weaved.com/in-action/weaved-iot-kit)
- [XMPP_2014]: XMPP/Jabber (http://wiki.xmpp.org/web/Tech_pages/IoT_systems)
- [EXI_2014]: Efficient XML Interchange (EXI) (http://www.w3.org/TR/exi/)
- [LWM2M_2014]: OMA Lightweight M2M (http://openmobilealliance.hs-sites.com/lightweight-m2m-specification-from-oma)
- [OPCUA_2015]: OPC Unified Architecture (UA) (https://opcfoundation.org/about/opc-technologies/opc-ua/)
- [ETSIM2M_2015]: ETSI SmartM2M (http://www.etsi.org/technologies-clusters/technologies/m2m)