Laboratory Data Networks

In this laboratory, each student dispose of a smart plug with an interruptible load connected behind it.

The goal of the laboratory is to build a complete IoT architecture allowing to monitor the plugs and control them from a cloud interface.

Your job is made up of two main two parts:

1. develop a program hosted by your Raspberry Pi, to interface your smart plug and the cloud, and
2. develop a Grafana interface that will allow you to visualize your data and control all the plugs.

shows a schema of the installation.

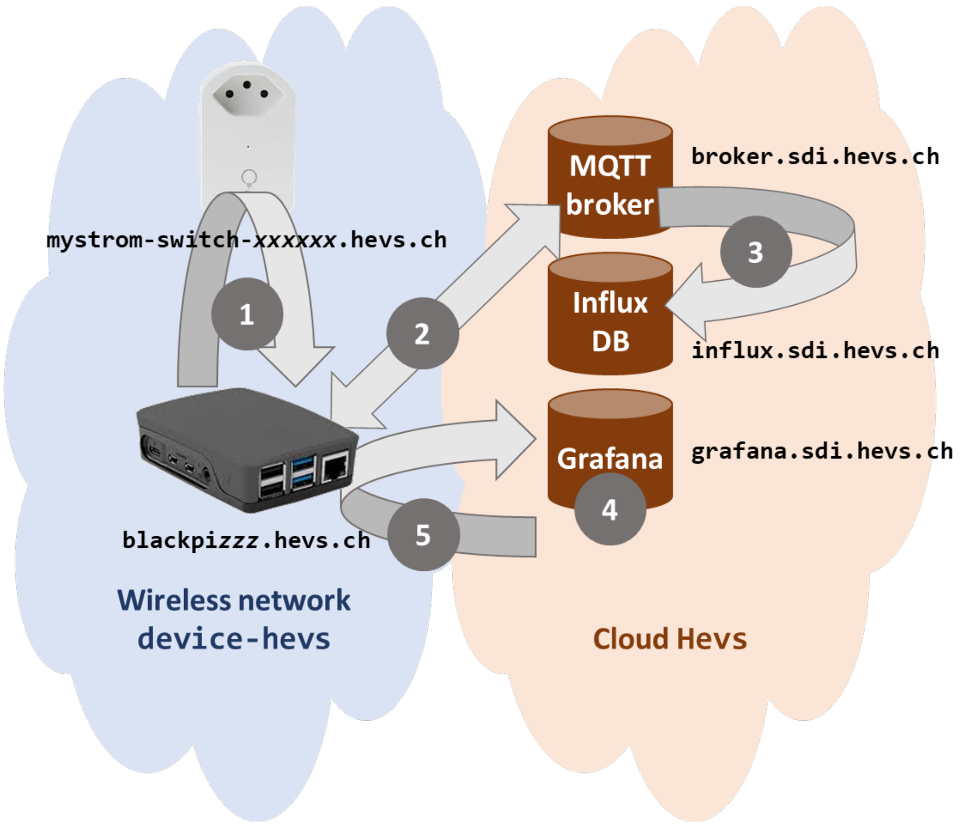


Figure 1 Overview of the lab's installation

Part 1 - Install and test your "myStrom Switch" smart plug

A "myStrom WiFi Switch" smart plug (https://mystrom.ch/wifi-switch-ch/):

- measures the consumed electrical power drawn from the plug,
- turns on/off a relay to enable/disable power consumption when requested, and
- provides an estimated value of the ambient temperature.

A myStrom plug is connected on the WiFi access point "device-hevs". It has a static IP address and is accessible through a permanent DNS name from the school IP network, including VPN. The DNS name for a given plug is:

> `mystrom-switch-`*`xxxxxx`*`.hevs.ch`
>
> where *`xxxxxx`* are the six last hexadecimal digits of the plug's MAC address (example: `mystrom-switch-944cd0.hevs.ch`)
>
> You can find the plug's MAC address on the back of the plug.

The "REST API" mode" of the myStrom WiFi Switch provides a so-called "REST API", concretely an integrated HTTP (web) server bound to local port 80 that allows fetching the current power, ambient temperature and relay state, and also to set the relay state.

URLs and their roles are specified in the myStrom documentation available at: https://mystrom.ch/wp-content/uploads/REST_API_WSE-11.txt.

1. Use a web browser to read the current consumed power, ambient temperature and relay state.
2. Use a web browser to change the relay state.
3. Use the `curl` or `wget` command to perform the same operations.

Part 2 - Access your myStrom smart plug in Python with your RaspberryPi 1

Your smart plug is identified by a number "*`xxx`*" ("`001`", "`002`", …). Name your smart plug "`my_strom-`*`xxx`*". This name is required for further steps of the project.

2.a) UML model for a myStrom smart plug

1. Define a `MyStromSwitch` UML class diagram with methods and attributes providing encapsulation for the smart plug.
2. Let an instructor validate your UML model.

2.b) HTTP access to the myStrom smart plug

In this task, you will develop a initial version of the `MyStromSwitch` class. The class will implement the access to your smart plug using HTTP.

1. Develop and test the `MyStromSwitch` class on your Raspberry Pi.

Part 3 - Interface the Raspberry Pi with the cloud 2

Your Raspberry Pi only serves as gateway to the cloud, i.e.:

- all measured values are pushed to the cloud right after measurement, and
- the cloud may request opening / closing the relay; such an order must be executed immediately.

For this purpose, you will make use of the MQTT based platform operated by the HEI. Topics to be used are summarized in Table 1.

Message topic	Direction	Description	Sample message body
`@update/my_strom-xxx/monitoring/power`	Pi to cloud	Instantaneous power consumed by appliances behind the smart plug	`{“value” : 24.56, “unit” : “W”}`
`@update/my_strom-xxx/monitoring/temperature`	Pi to cloud	Instantaneous temperature measured by the smart plug	`{“value” : 24.56, “unit” : “C”}`
`@set/my_strom-xxx/control/relay`	Cloud to pi	Relay command (“open”, “close”)	`{“value” : “open”}`

Table 1 MQTT topics

Take care when using the broker that just topics starting with @update/ or with @set/ are accepted by the broker. **If you publish or subscribe to other topics, the broker will cut the connection.**

The coordinates to access the MQTT broker are:

Broker	broker.sdi.hevs.ch, port 1883 (default port for MQTT) (public access) (also available on port 8883 with SSL/TLS, requires a certificate)
Username	“SInxx”, with xx between 01 and 40. Use the number of your Raspberry Pi box. Multiple access with the same login can produce errors.
Password	The MD5 hash of the username “SInxx” as given by http://md5.cz

Table 2 Coordinates of the MQTT broker

1. Install the paho.mqtt.client package for MQTT programming.
2. Instantiate a Client object and pass the appropriate parameters to its constructor to connect to the broker. Don’t forget to call the method loop_forever() at the end of the constructor, to initiate the consumption of MQTT messages.
3. Read periodically every second the power and the temperature on the smart plug and publish the corresponding messages (using qos 0).
4. Subscribe to the relay’s topic and register a call-back method (for example on_relay_set()). Develop this method and link it to the smart plug’s relay.

5. To validate your MQTT programming, install the MQTT Explorer program (http://mqtt-explorer.com). Use the same credentials to login on the broker. Subscribe to the smart plug's topic and display the received messages. Publish a message to change the state of the relay.
6. You can use the example file *MqttConnectTest.py* provided on Cyberlearn.

Part 4 - Monitoring of data on InfluxDB & Grafana

Finally, we would like to implement a graphical supervision of all the plugs with all their historical values. To do it, we will implement a new class that is permanently listening to the broker and that is pushing all data from the plugs it receives directly to the cloud. To do it, we will use InfluxDB, a time series database, and Grafana, a tool that can be used to visualize the data from the DB.

4.a) Storage on InfluxDB 3

An InfluxDB server hosts usually several databases. Users – defined by a username and a password – have read and/or write access to one or more databases.

- An InfluxDB server is available at `influxdb.sdi.hevs.ch` on port TCP 8086.
- It hosts 40 databases `SIn01`, `SIn02`… `SIn40`.
- There are also 40 usernames `SIn01`, `SIn02`… `SIn40`. The password is given by the MD5 hash of the username (lower case hex characters, see http://md5.cz)
- A user `SIn`*xx* has read and write access to the database `SIn`*xx*.

Use a personal database `SIn01`…`SIn10` and make a small client that will subscribe to all power messages managed by the broker (`@update/+/monitoring/power`) and push the power values to InfluxDB.

1. Secure a personal username `SIn01`… `SIn10`.
2. Install the package `influxdb-client` using the Thonny packet manager. See its documentation here: https://github.com/influxdata/influxdb-client-python.
3. Develop a `DbConnector` class with an `InfluxDBClient` object as attribute.
4. Develop a specific method in the `DbConnector` class to log the current power (and temperature and relay state) for a given message. Make use of the `InfluxDBClient` method `write_api`, as you did in the previous laboratory.
5. You can then make a small program that will then connect to the broker and subscribe to all the data coming from the plugs. This program will then use your `DbConnector` class and methods to push the values received to InfluxDB.

4.b) Grafana visualization 4

1. Similarly, as in the laboratory on computer monitoring, prepare a workbench in Grafana that is presenting the load of all the different plugs.
2. Add a display for each plug showing if they are on or off.
3. As an addition, add a curve that is the aggregated load of all the plugs.
4. Finally, add the extended measurement from the plug (temperature) if it exists.

Part 5 - Control of the plugs through Grafana 5

Buttons can be added in Grafana dashboards. In this part, you will use this feature to manually open, close, or toggle the relay of a smart plug. For this, two operations are needed:

1. associate an HTTP request to be sent upon button pressing, and
2. setup an HTTP server interpreting the HTTP request to control the relay through MQTT.

The HTTP server must be hosted by station with a DNS name / static IP address. Your Raspberry Pi fulfils this condition, and you are asked to use it for that purpose.

You may use one or more of the HTTP requests indicated in Table 3 to control the plugs' relay. As you control both the client or the server, you may use the URL and bodies that you find appropriate.

HTTP method	URL (local path only)	Body in HTTP request
`POST, (GET*)`	`/plugs/<plug_name>/relay?value=<open|close>`	-
`POST, (GET*)`	`/plugs/<plug_name>/relay`	`{"value":"<open|close>"}`
`POST`	`/plugs/relays`	`[{"name":"<plug_name>","value":"<open|close>"}, ...]`

*The philosophy of HTTP REST requires to use the POST method to modify a resource, what we do in this case. However, it is technically possible to use the GET method.

Table 3 Sample HTTP requests to control relays

5.a) Developing an HTTP server in Python

A simple way to develop an HTTP server in Python is to use the `flask` package.

An object of the class Flask must be instantiated and associated with a server port (e.g., 8080):

```
app = Flask(__name__)
app.run(port=8080)
```

The basic operation of a flask HTTP server is explained in Table 4. `@app.route` annotations associate local paths of URLs with a method handling the requests.

HTTP operation, local path, request body (if any)	flask code	Comment

`GET /a`	`@app.route('/a', methods=['GET'])`
	`def a():`
		The name of the method ("a") can be freely chosen.

`GET|POST /c?d=e`	`@app.route('/c', methods=['POST'])`
	`def c():`
	`  request.args.get(d) # is "e"`
		`request` is an object representing the HTTP request.

`POST /f/`*`any`*	`@app.route('/f/<x>', methods=['POST'])`
	`def f(x): # x takes the value of "any"`
		Variable elements of the local path can be recuperated as arguments

`POST /g`
 	
`{"h": "i"}`
	`@app.route('/g', methods=['POST'])`
	`def g():`
	`  request.data.decode()`
		`request.data.decode()` returns the HTTP body as string `({"h": "i"})`

Table 4 From HTTP to code

All the above methods should return the HTTP response to be sent back to server. The response can be crafted using the `response_class()` method of the app object:

```
response = app.response_class(body_to_send)
```

Be aware that the above-mentioned methods can use global variables (e.g., the `mqtt.Client` object) only if they are declared as `global` in each of the methods.

```
mqtt_connector = mqtt.Client(…)

@app.route(…)
def a():
   global mqtt_connector
   mqtt_connector.publish(…)
```

5.b) Specific configuration for the given situation

If you want to have a link between Grafana and your REST server, you should create a SSL secured REST server, as it is required by the also secured Grafana server. For that, you have to slightly change your Flask server with the following parameters

```
app.run(host='blackpi010.hevs.ch',port=8080,ssl_context=('cert.pem',
'key.pem'))
```

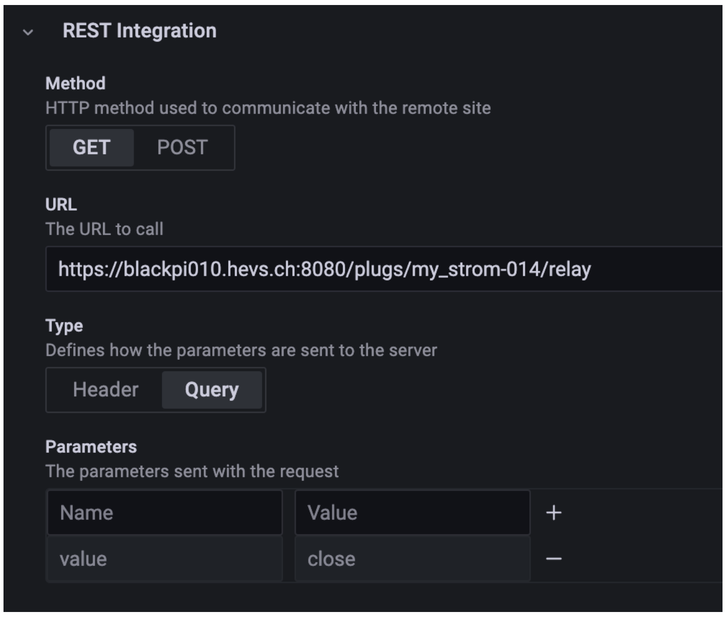
Having a verified SSL certificate would be too pricey and complicated. As a result, you will have to generate your own certificate. You can do it directly on your Rasbperry pi by using openssl, for example:

```
openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem -keyout key.pem -
days 365
```

Moreover, to make it work with the buttons from Grafana, you have to use the first version of the REST server, with parameters sent through the URL, not using JSON messages. You can test your server directly from your browser. Beware that your browser will show you an exception as the certificate is not officially validated. But as you have generated it, you can accept it.

5.c) Creation and configuration of buttons on Grafana

Finally, you can add buttons to your dashboard. For that, you can use button-panels. Information on how you can use them can be found on the webpage of the plugin[1].Hereunder stands an example how your button could be configured.



REST Integration

Method
HTTP method used to communicate with the remote site
GET [✓] POST

URL
The URL to call
https://blackpi010.hevs.ch:8080/plugs/my_strom-014/relay

Type
Defines how the parameters are sent to the server
Header [✓ Query]

Parameters
The parameters sent with the request

Name	Value
value	close

Create multiple buttons on your Dashboard corresponding to all the plugs. You can have a button for some or all the actions (open, close, toggle). You can also use a status element showing the state of the different plugs.

Finished the dashboard. You should now have both a display of historical data and status of all plugs of the class, but also a web interface to control your plugs directly.

[1] https://grafana.com/grafana/plugins/cloudspout-button-panel/