

# Chapter 7

## KIS.API: Towards External Communication



### 7.1 Introduction to KIS.API

KIS.API is defined as a particular instance of REST API [1–3], which is designed for the purpose of unified and convenient communication with KIS.Devices and the associated environment operating within KIS.MANAGER.

The crucial components of KIS.API are the resources, which can be defined in a very broad sense. In fact, anything having a name can be perceived as a resource, e.g., a user, a workspace, an asset, etc. The crucial information associated with a resource pertains to the service being realized, i.e., the transfer of data as well as the associated actions. A general structure of the resource is given in Table 7.1 [1].

Let us start with the . As an example, one can consider a KIS.Device, which can be characterized by, e.g.,

- an ID,
- a name,
- a URN,
- associated asset group (workspace) IDs,

while its possible representation can be given in an intuitive JSON [1] form:

```
{
  "id": 10102,
  "urn": "urn:rafi:sbox:9c65f93cbcd6",
  "name": "KIS.BOX_9C65F93CBED6",
  "assetGroupIDs": [
    9757,
    9758
  ]
}
```

**Table 7.1** A typical structure of a resource

Property	Description
Representation	The way and structure of the data representation, e.g., JSON, XML
Identifier	A URL that refers to a specific resource at any time
Metadata	The content type, modification time, etc
Control data	Any data identifying the status of a resource, e.g., last modification date

**Table 7.2** KIS.API actions

HTTP verb	Action
GET	Access a resource in a read-only way
POST	Send a new resource
PUT	Update a resource
DELETE	Delete a resource

**Table 7.3** KIS.API response

Status code	Description
200	Successful response
204	No content returned
400	Bad request structure
404	Unsuccessful response

Now let us proceed to the resource identifier, which provides a unique way to identify the resource at any time instance. In other words, it should provide a full and unique path to the resource. Since the REST structure is based on HTTP, a sample path can look as follows:

```
https://api.kisme.com/kisapi/v1/assets/assetUrn,
```

which uniquely identifies a given KIS.Device using its URN. Having a resource identifier, one can proceed to realize some actions with it. A general set of verbs, which defines specific actions, is given in Table 7.2. Finally, an important standard that is inherited by REST from HTTP pertains to the status code. The most common status codes are given in Table 7.3. Under the above preliminary information, one can proceed to the registration and authorization of a new KIS.API user using KIS.MANGER.



### 7.1.1 User Registration and Authorization

The primary objective of this part is to define a new KIS.API user along with an appropriate credentials. The process starts with selecting the Main menu → Portal

### KIS.API Credentials




Fig. 7.1 Defining KIS.API credentials

admin and then pressing the KIS.API icon . Subsequently, a new KIS.API user can be created with . The process is fully automatic and the only information required is to provide a user description, i.e., a name. A sample KIS.API credentials generation process is presented in Fig. 7.1. As a sample, a KIS.API access control triplex is obtained, which can be summarized as follows:

- the client ID,
- the API key,
- the baseUrl.

Note that baseUrl is simply the base resource identifier detailed in the preceding section, i.e.,

```
https://api.kisme.com/kisapi/v1/
```

The objective of the subsequent sections is to provide a concise introduction to KIS.API. Thus, to simplify this process, the Postman (<https://www.postman.com/>) application is employed. It is a dedicated platform for building and using APIs. In other words, Postman can be perceived as an HTTP client for testing web services, which makes it easy to test APIs by providing a simple interface for issuing API requests and viewing responses. The Postman registration and configuration process is very intuitive, and hence it is omitted. The subsequent step is to proceed to the KIS.API documentation (<https://docs.kisme.com>) and then select Run are Postman button. As a result, KIS.API collection is loaded into the Postman workspace. The final step is to provide the above defined KIS.API credentials using . This process is illustrated in Fig. 7.2. As of that moment, all KIS.API commands can be accessed and tested using the intuitive Postman graphical user interface. This process boils down to selecting an appropriate option and then sending a desired request to KIS.API. Thus, the objective is to provide a concise review of all available KIS.API functionalities.

Globals <span style="float: right;">Edit</span>		
VARIABLE	INITIAL VALUE	CURRENT VALUE
api-key		●●●●●●●●●●●●●●●●●●●●●●●● ...
client-id		●●●●●●●●●●●●●●●●●●●●●●●● ...
baseUrl	https://rafi-stage-api-mgmt.azure-api.net/kisapi/v1	https://rafi-stage-api-mgmt.azure-api.net/kisapi/v1

**Fig. 7.2** Postman configuration with KIS.API credentials

## 7.2 Essential Functionalities

The objective of this section is to provide essential KIS.API functionalities. This starts from access to assets being simply KIS.Devices, users and asset groups up to the related Datapoints. Finally, a set of recipes concerning an access to calculated Datapoints and KPIs is provided.

### 7.2.1 *Obtaining Information About Asset Groups, Assets and Users*

As introduced in Chap. 2, KIS.Devices constitute the core KIS.ME components. Thus, knowing all of them, along with their membership to particular asset groups (see Chap. 2), is of paramount importance. Table 7.4 presents the list of all available KIS.API requests concerning assets along with the designated actions. As can be observed, most requests require additional path variables:

- **assetUrn**: directly printed on an asset (KIS.Device) and can be retrieved through request no. 1 from Table 7.4 or through KIS.MANAGER;
- **assetgroupId**: can be retrieved through the request no. 1 in Table 7.5.

The response pertaining to the request detailed in Table 7.4 may contain the following parameters:

- ID: the asset identifier,
- URN: the asset URN,
- name: the name of the asset,
- isOnline: the asset `isOnline` Datapoint value,
- hardware: information about the asset hardware,
- software: information about the asset software,
- certificate: information about the asset certificate,
- network: information about the asset network,
- firmwareUpdate: information about the asset firmware updates,


**Table 7.4** List of possible requests associated with an asset

No.	Verb	Path	Action
1	GET	baseUrl/assets	Obtain a list of all assets
2	GET	baseUrl/assets/assetUrn	Obtain the asset device information
3	PUT	baseUrl/assets/assetUrn	Update the name of the asset
4	PUT	baseUrl/assets/assetUrn/assetgroupId/ assetgroups	Add the asset to an asset group
5	DEL	baseUrl/assets/assetUrn/assetgroupId/ assetgroups	Remove the asset from an asset group

**Table 7.5** List of possible requests associated with an asset group

No.	Verb	Path	Action
1	GET	baseUrl/assetgroups	Obtain a list of all asset groups
2	GET	baseUrl/assetgroups/assetgroupId	Obtain the asset group information
3	PUT	baseUrl/assetgroups/assetgroupId	Update the asset group information

assetGroupIDs: an array containing numerical values of the asset groups associated with the asset.

Note that most of the above features can be directly accessed through KIS.MANAGER by selecting Main menu → Assets and then choosing a desired asset. Subsequently, the information about the asset can be retrieved by pressing  (see Fig. 2.16).

Having all the above information, let us proceed to two simple examples of using the discussed requests.

**Obtaining a list of assets**

This example concerns a response to the request no. 1 listed in Table 7.4. As a result, the following JSON structure can be obtained:

```

]
  {
    "id": 10102,
    "urn": "urn:rafi:sbox:9c65f93cbcd6",
    "name": "KIS.BOX 9C65F93CBED6",
    "assetGroupIDs": [
      9757,
      9758
    ]
  },
  {
    "id": 10153,

```

```

    "urn": "urn:rafi:sbox:9c65f93cbeb8",
    "name": "KIS.BOX 9C65F93CBEB8",
    "assetGroupIDs": [
        9757,
        9758
    ]
}
]

```

As can be observed, the response contains information about two KIS.BOXes, which are assigned to two asset groups (9757 and 9758).

### Removing an asset from the asset group

This example concerns a response to the request no. 5 listed in Table 7.4. Let us consider the first asset (KIS.BOX) given in the preceding example. Its URN is `urn:rafi:sbox:9c65f93cbcd6`, and it is assigned to two asset groups 9757 and 9758. The objective is to remove it from the asset group 9757, and hence the `assetgroupId` parameter should be set to 9757. As a result, the following JSON structure can be obtained:

```

{
  "message": "The operation is in conflict with the
  relationship constraint 'assetsMustBeMemberOfInventory'",
  "code": 409
}

```

which simply means that it is impossible to remove an asset from the list of all available assets (see Table 2.5 for a comprehensive explanation). Thus, let us change the `assetgroupId` parameter to 9758 (the second available asset group). As a result, the following JSON structure is obtained:

```

{
  "id": 10102,
  "urn": "urn:rafi:sbox:9c65f93cbcd6",
  "name": "KIS.BOX 9C65F93CBED6",
  "assetGroupIDs": [
    9757
  ],
  "isOnline": false,
  "hardware": {
  },
  "software": {

```

```

    },
    "certificate": {
    },
    "network": {
    },
    "firmwareUpdate": {
    }
  }
}

```

As can be observed, the assignment of this asset to the asset group 9758 was removed.

Let us proceed to the asset groups for which the available request list is given in Table 7.5. As can be observed, most requests require an additional parameter `assetgroupId`, which can be retrieved through the request no. 1 in Table 7.5. The response pertaining to the request detailed in Table 7.5 may contain the following parameters:

- ID: the asset group identifier,
- `assetIDs`: an array containing numerical values of asset IDs associated with the asset group,
- `name`: the name of the asset,
- `isOnline`: the asset `isOnline` Datapoint value,
- `description`: a detailed description of the asset group,
- `name`: the name of the asset group.

Let us proceed to two simple examples explaining the application of the above requests.

### Obtaining a list of asset groups

This example concerns a response to the request no. 1 listed in Table 7.5. As a result, the following JSON structure can be obtained:

```

[
  {
    "id": 9757,
    "name": "My Devices",
    "assetIds": [
      10102,
      10153
    ]
  },
  {
    "id": 9758,

```

```

        "name": "Workspace 1",
        "assetIds": [
            10153
        ]
    }
]

```

As can be observed, the response contains information about two asset groups and the assets associated with them (IDs: 10102, 10153).

---

### Updating the asset group description

This example concerns a response to the request no. 3 listed in Table 7.5. Let us provide a new description of Workspace 1 (ID 9758) in the JSON form:

```

{
    "name": "Workspace 1",
    "description": "Main Workspace"
}

```

In the case of the Postman application, to provide such a description one should go to the `Body` tab of the request and enter the above JSON structure. As a result, the following JSON structure can be obtained:

```

{
    "id": 9758,
    "name": "Workspace 1",
    "assetIds": [
        10153
    ],
    "description": "Main Workspace"
}

```

---

Having access to assets and the associated asset groups, let us proceed to the user management functionalities, which are listed in Table 7.6. As can be observed in Table 7.6, requests no. 2 and 3 require an additional path parameter called `accountNumber`. Note that in KIS.MANAGER the user is identified by its name and email. Thus, at least one of these parameters should be known while realizing the request no. 1 in Table 7.6. Thus, the obtained response can be used to obtain the associated `accountNumber`.



**Table 7.6** List of possible requests associated with users

No.	Verb	Path	Action
1	GET	baseUrl/users	Obtain a list of all users
2	GET	baseUrl/users/accountNumber	Obtain user information
3	DEL	baseUrl/users/accountNumber	Delete a user

### Displaying all users and their accountNumber

This example pertains to realisation of the request no. 1 in Table 7.6. As a result, the following JSON structure can be obtained:

```
[
  {
    "name": "Jack Cactus",
    "email": "j.cactus@controlintech.pl",
    "accountNumber": "3d5997bb-f6ee-4681-8adf-0ce7366e2964"
  },
  {
    "name": "Hans Wurst",
    "email": "h.wurst@controlintech.pl",
    "accountNumber": "4f4009c2-1a7b-4531-b780-702a19cd62a1"
  }
]
```

The structure contains information about two users and their associated accountNumber.

### Deleting a user

The JSON structure obtained in the preceding example contains information about two users. The objective of the current example is to delete the user identified:

```
"accountNumber": "3d5997bb-f6ee-4681-8adf-0ce7366e2964"
```

For that purpose, the above number has to be provided as a path parameter in the Postman application. Note that, after sending a request to KIS.API, no JSON structure is received (cf. code 204 in Table 7.3).

**Table 7.7** List of possible requests associated with Datapoints

No.	Verb	Path	Action
1	GET	BaseUrl/assets/assetUrn/ datapointDefinitions	Obtain a list of Datapoints
2	GET	BaseUrl/assets/assetUrn/ datapointDefinitions/datapointValues	Obtain Datapoint values

## 7.2.2 Accessing Data Through Datapoints

The objective of this point is to provide a way of accessing the data associated with Datapoints. For a comprehensive description of Datapoints, the reader is referred to Sect. 2.7 and Appendix. B. The possible requests associated with Datapoints are provided in Table 7.7. It should be also noted that the above requests require the following path parameters:

- `assetUrn`: directly printed on an asset (KIS.Device), can be retrieved through the request no. 1 from Table 7.4 or through KIS.MANAGER;
- `datapointDefinition`: the Datapoint name, e.g., `button1Pressed`.

Additionally, request no. 2 in Table 7.7 can be executed with the following query parameters:

- `from`: an ISO timestamp indicating a lower bound of the required time range;
- `to`: an ISO timestamp indicating an upper bound of the required time range;
- `limit`: a maximum number of required Datapoint values.

Note that it is not compulsory to use all of the above-listed query parameters simultaneously. For example, the `limit` parameter can be employed as a standalone one.

### Obtaining a list of Datapoints

The example is concerned with the request no. 1 in Table 7.7. As a result of using it, a full list of Datapoints along with their types is returned. A sample form of such a couple is given as follows:

```
{
  "name": "button2Pressed",
  "datatype": "BOOLEAN"
}
```

### Obtaining five recent Datapoint values

For the purpose of this example, a set of two rules is implemented (see Sect. 2.9 for more details), i.e.,

1. If the KB Button1 operational LED color is black, then the KB Button 1 operational LED color is red.
2. If the KB Button 1 operational LED color is red, then the KB operational LED color is black.

Thus, the purpose of the above rules is to switch the KB Button1 operational LED color from red to black (no illumination) and vice versa. This means that the resulting effect should be the KB Button 1 operational LED blinking in red. However, to achieve such an effect, the KB Button 1 operational LED color should be initiated using its digital twin (see Sect. 2.6) by setting the above color to either black or red. Subsequently, the path parameters should be defined, i.e., `assetUrn` and `datapointDefinition`. The latter one is set to `button1ColorKpi`. Finally, the query parameter `limit` is set to five. As a result, a JSON structure is obtained containing the five recent values of the indicated Datapoint:

```
[
  {
    "timestamp": "2022-08-30T11:50:07.478Z",
    "value": "2"
  },
  {
    "timestamp": "2022-08-30T11:50:06.478Z",
    "value": "5"
  },
  {
    "timestamp": "2022-08-30T11:50:05.524Z",
    "value": "2"
  },
  {
    "timestamp": "2022-08-30T11:50:04.712Z",
    "value": "5"
  },
  {
    "timestamp": "2022-08-30T11:50:03.649Z",
    "value": "2"
  }
]
```

The recorded Datapoint values simply indicate that the KB Button 1 operational LED changes its color from red (5) to black (2) and vice versa. As can be observed,

the switching process takes more or less one second. However, this time is data transfer-dependent, and hence its is not uniform.

---

### Obtaining five recent Datapoint values from a given time frame

The objective of this example is to focus on the reader attention to ISO data-time format, which is given, e.g., by

```
2022-08-30T14:02:07.478Z
```

Its construction is obvious: however, it contains characters which are not permitted in a URL construction, i.e. '.', which should be simply replaced by its equivalent equal to '%3A', yielding

```
2022-08-30T14%3A02%3A07.478Z
```

Thus, by setting the `from` query parameter according to the above form one can obtain:

```
[
  {
    "timestamp": "2022-08-30T14:02:32.304Z",
    "value": "5"
  },
  {
    "timestamp": "2022-08-30T14:02:31.304Z",
    "value": "2"
  },
  {
    "timestamp": "2022-08-30T14:02:30.336Z",
    "value": "5"
  },
  {
    "timestamp": "2022-08-30T14:02:29.507Z",
    "value": "2"
  },
  {
    "timestamp": "2022-08-30T14:02:28.475Z",
    "value": "5"
  }
]
```

---

### 7.2.3 KPIs and Calculated Datapoints

This section constitutes a continuation of the preceding one. Indeed, CDPs and KPIs (see Sect. 4.1.2 and Appendices A and B) employ Datapoints as a basis for forming desired answers pertaining to the system state and performance. The CDP requests are similar to those for Datapoints (see Table 7.7), and they are presented in Table 7.8. It should be also noted that the above requests require the following path parameters:

- `assetUrn`: directly printed on an asset (KIS.Device), can be retrieved through the request no. 1 from Table 7.4 or through KIS.MANAGER;
- `calaculatedDatapointDefinition`: CDP name, e.g., `kg2lb`.

Additionally, the request no. 2 in Table 7.8 can be executed with the following query parameters:

- `from`: an ISO timestamp indicating a lower bound of the required time range;
- `to`: an ISO timestamp indicated an upper bound of the required time range;
- `limit`: a maximum number of required CDP values.

Now, let us proceed to KPI requests available through KIS.API. They are given in Table 7.9, and it is not surprising that they are similar to those presented in Table 7.8. Additionally, they path parameters are given by

- `assetUrn`: directly printed on an asset (KIS.Device), can be retrieved through request no. 1 from Table 7.4 or through KIS.MANAGER;
- `kpiDefinition`: the CDP name, e.g., `kg2lb`.

Unlike Datapoints and CDP, the KPI request no. 2 (see Table 7.9) has to be executed with the following query parameters:

- `from`: an ISO timestamp indicating a lower bound of the required time range;
- `to`: an ISO timestamp indicate an upper bound of the required time range.

**Table 7.8** List of possible requests associated with CDPs

No.	Verb	Path	Action
1	GET	<code>baseUrl/assets/assetUrn/calaculatedDatapointDefinitions</code>	Obtain a list of CDPs
2	GET	<code>baseUrl/assets/assetUrn/calaculatedDatapointDefinition/calaculatedDatapointValues</code>	Obtain CDP values

**Table 7.9** List of possible requests associated with KPIs

No.	Verb	Path	Action
1	GET	<code>baseUrl/assets/assetUrn/kpiDefinitions</code>	Obtain a list of KPIs
2	GET	<code>baseUrl/assets/assetUrn/kpiDefinition/kpiValues</code>	Obtain KPI values

### Accessing the list of KPIs

Let us continue with the example presented in Sect. 7.2.2 pertaining to a KIS.BOX associated with two rules. These two rules perform cyclically one after the other. The first one change the KIS.BOX Button 1 operational LED color from black to red while the second one realizes an opposite situation. It is assumed that no KPIs are defined for this KIS.BOX (see Sect. 4.1.2 for a detailed tutorial on KPIs). Thus, let us define the KPI counting the entire time period for which the above mentioned color is red. For that purpose, the following KPI is implemented:

```
y = Round[Sum[If[x == 5, Duration[x], 0]]/1000];
```

where `x` stands for the `button1ColorKpiDuration` Datapoint while the number five signifies the red color (see Table 2.2). Finally, let us assume that this KPI is named `KBButton1Red` while its processing period is set to 15 min. Having the above KPI, let us proceed to performing the request no. 1 in Table 7.9. As a result, the following JSON structure is obtained:

```
[
  {
    "name": "KBButton1Red"
  }
]
```

### Accessing KPI values

Let us continue with the above example. Now, the task is to obtain `KBButton1Red` (path parameter `kpiDefinition`) values within the time period defined by the parameters `'from'` and `'to'` given by

```
2022-08-31T08:02:50.046Z
2022-08-31T12:22:50.046Z
```

which, as discussed in Sect. 7.2.2, are formatted according to

```
2022-08-31T08%3A02%3A50.046Z
2022-08-31T12%3A22%3A50.046Z
```

As a result, the following JSON structure is obtained:

```
{
  "to": "2022-08-31T12:22:50.046Z",
  "from": "2022-08-31T08:02:50.046Z",
  "values": [
    447,
    443,
    448,
    447,
    451,
    451,
    448,
    451,
    450,
    453,
    447,
    454
  ]
}
```

It can be easily observed that there are 12 values corresponding to 15-minute processing periods. It is also straightforward to observe that 15 min are equivalent to 900s. Thus, it is evident that all the above-presented values should oscillate around 450s, which is actually the case.

---

### Accessing data from CDPs

The last task of this point concerns obtaining an information about predefined CDPs as well as finding their values. For that purpose, it is assumed that no CDPs are defined. Moreover, the preceding example is continued. Thus, a new CDP is defined according to the approach presented in Sect. 4.1.1 with  $x$  equivalent to `button1ColorKpiDuration` Datapoints. The developed CDP aims at bounding the minimum numerical value of  $x$  to 3, which represents the green color (see Sect. 2.6). As a result, the following simple implementation is obtained:

$$z = \text{If}[x < 3, 3, x];$$

while CDP itself is named `KBmaxcolorCDP`. Let us start with the request no. 1 in Table 7.8, which pertains to obtaining a list of all available CDPs. As a result of using it, the following JSON structure is arrived at:

```
[
  {
    "name": "KBmaxcolorCDP",
    "datatype": "DOUBLE"
  }
]
```

which contains the existing CDP names as well as their data types. Having the CDP name, let us proceed to obtaining its values. In fact this process is identical to the one presented in Sect. 7.2.2. According to the adjustment performed in the preceding examples,  $x$  can have the values representing either the red or the black color, i.e.,  $x = 5$  or  $x = 2$  (cf. Table 2.2). The request No. 2 in Table 7.8 is executed with the query parameter `limit` only, which is equal to 10. As a result, the following JSON structure is obtained, which provides the desired results:

```
[
  {
    "timestamp": "2022-09-02T10:45:22.804Z",
    "value": "5"
  },
  {
    "timestamp": "2022-09-02T10:45:22.335Z",
    "value": "3"
  },
  {
    "timestamp": "2022-09-02T10:45:21.554Z",
    "value": "5"
  },
  {
    "timestamp": "2022-09-02T10:45:20.929Z",
    "value": "3"
  },
  {
    "timestamp": "2022-09-02T10:45:20.054Z",
    "value": "5"
  }
]
```

---



**Table 7.10** List of possible requests associated with rules

No.	Verb	Path	Action
1	GET	baseUrl/rules	Obtain a list of rules
2	GET	baseUrl/rules/ruleId/assetgroupId	Obtain a rule info

## 7.2.4 Accessing Information About Rules

Rules (cf. Sect. 2.9) constitute the last component which can be accessed through KIS.API. The currently possible requests are provided in Table 7.10.

Additionally, the request no. 2 in Table 7.10 should be executed with the following path parameters:

- ruleId: the rule identification number which can be retrieved through the request no. 1 from Table 7.10.
- assetgroupId: asset group identification.

The parameters that can be accessed through the process of executing these requests are

- name: the name of the rule provided in KIS.MANGER Rule engine;
- enabledAPI: a logical property stating if it is possible to trigger the rule from an external application using KIS.API.

### Obtaining information about rules

The objective of this example is to show how to access information about rules. Let us start with the request no. 1 in Table 7.10, which does not require any path or query parameters while its execution results in the following JSON structure:

```
[
  {
    "name": "rblack2red",
    "assetGroupId": 9758,
    "id": "b92080c3-55aa-410e-a417-9efe90637515",
    "enabledAPI": false
  },
  {
    "name": "rred2black",
    "assetGroupId": 9758,
    "id": "62124944-a606-4b80-b906-94d6d5ae8d38",
    "enabledAPI": false
  }
]
```

Contrarily, having `assetGroupId` and `id` signifying the rule, one can obtain the name of the rule and the logical property `enabledAPI`. Indeed, by using them as the path parameters and then executing the request no. 2 in Table 7.10, one can arrive at the following JSON structure:

```
{
  "name": "rblack2red",
  "assetGroupId": 9758,
  "id": "b92080c3-55aa-410e-a417-9efe90637515",
  "enabledAPI": false
}
```

### 7.2.5 Triggering Rules from External Applications

The objective of this section is to introduce a very important feature of KIS.API, which makes it possible to trigger a rule from an external application. However, as mentioned in Sect. 7.2.4, such an operation is possible for the rules having the `enabledAPI` property set to the logical truth. For example, the rule considered at the end of Sect. 7.2.4 should have the following JSON structure:

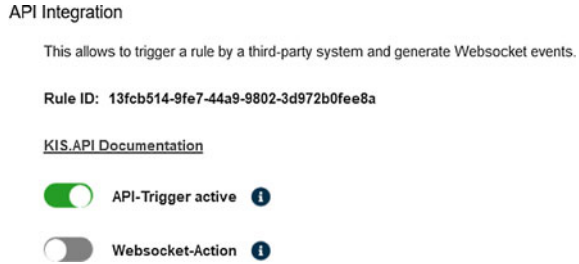
```
{
  "name": "rblack2red",
  "assetGroupId": 9758,
  "id": "b92080c3-55aa-410e-a417-9efe90637515",
  "enabledAPI": true
}
```

Note that the modification of the `enabledAPI` property is possible through KIS.MANGER only. For that purpose, one should use the Main menu → Asset groups and then select an appropriate asset group. Subsequently, by going to Rule engine and selecting the desired rule, one can see the property editor called API integration, which is presented in Fig. 7.3. As can be observed, there is another property, which is called Websocket-Action. However, it will be discussed in Sect. 7.5. Finally, the rule triggering request is described in Table 7.11.

**Table 7.11** a requests associated with a rule trigger

No.	Verb	Path	Action
1	POST	<code>baseUrl/rules/RuleId/assetgroupId/Trigger</code>	trigger a rule

**Fig. 7.3** Setting enabledAPIproperty



## 7.3 KIS.API in Practice

The purpose of Sect. 7.2 was to introduce to the reader the essential functionalities concerning KIS.API. All of them were carefully described while their practical usage was explained with the Postman (<https://www.postman.com/>) application. The objective of this section is to provide practical guidelines concerning KIS.API application using some popular software. Indeed, the software selection being used in this section is not accidental. The first candidate is employed widely both in the industry for presenting various kinds of data in tabular order. The second one is commonly used for research, analysis, development and deployment of new practical concepts based on data gathered from a given system. Thus, these two popular software instances are

- Microsoft Excel (<https://www.microsoft.com/>),
- MathWorks Matlab (<https://www.mathworks.com/>).

Both of them have several different and freely-available counterparts, which can provide similar functionalities. Thus, the objective of the subsequent point is to provide a short practical tutorial on feeding MS Excel and Matlab with KIS.ME data. Although the current section is restricted to MS Excel and Matlab, the reader possessing the knowledge about the KIS.ME essential functionalities can integrate it with more advanced and dedicated software. An enterprise resource planning system can be a good example of such software (see, e.g., SAP and its API functionalities at <https://api.sap.com/>).

### 7.3.1 Feeding MS Excel with KIS.ME Data

Starting with MS Excel 2013 it is possible access any REST API using the so-called power query. Thus, the entire recipe for accessing data from KIS.API boils down the following steps:

1. Select and push the `from Web` power query icon.
2. Provide the URL associated with the desired request.

Convert	Manage Items	Sort	Numeric List
Queries	>		
		List	
	1	Record	
	2	Record	
	3	Record	

Fig. 7.4 Result of a KIS.API request in MS Excel

ABC 123	Column1.name	ABC 123	Column1.assetGroupId	ABC 123	Column1.id	ABC 123	Column1.enabledAPI
1	KBOutHigh		9758	52739801-29c0-4318-9edd-0774fd3d0c44			FALSE
2	rbblack2red		9758	b92080c3-55aa-410e-a417-9efe90637515			FALSE
3	rred2black		9758	62124944-a606-4b80-b906-94d6d5ae8d...			FALSE

Fig. 7.5 Result of a KIS.API request presented as a table

3. Use advanced options to provide appropriate headers, i.e., X-CLIENT-ID and X-API-KEY (see Fig. 7.1).
4. Perform the desired data request.

**Obtaining a list of all rules**

The objective of this example is to obtain a list of all rules present in KIS.MANAGER Rule engine. According to Sect. 7.2.4, a URL should be defined as follows:

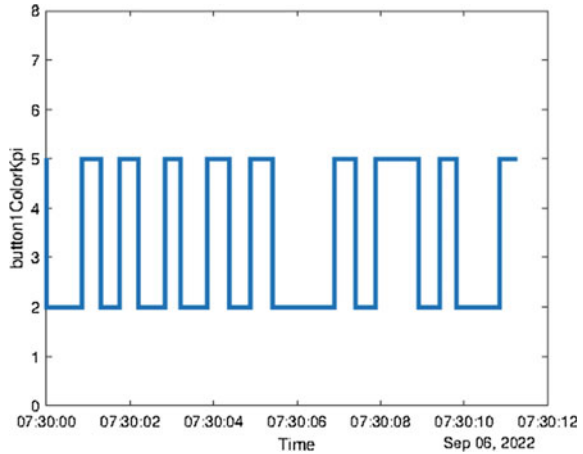
```
baseUrl/rules
```

while the required headers may have the following structure:

```
X-API-KEY 2b84bd4de38b4e92bb7bb283364477e1
X-CLIENT-ID 708ce7cc-fd57-416d-991f-f68704385c22
```

Finally, the desired data request is performed and the obtained result is given in Fig. 7.4. As can be observed, there are three records, which simply correspond to three different rules. To make the obtained result more transparent, the option Convert to Table can be used used, which after suitable expansion yields the view presented in Fig. 7.5.

**Fig. 7.6** Result of a KIS.API request in Matlab



### 7.3.2 Feeding Matlab with KIS.ME Data

The objective of this section is to show how to realise the KIS.API GET request with MATLAB. The entire process boils down to the following steps:

1. Provide the URL associated with the desired request.
2. Define the `weboptions` structure `HeaderFields` containing the returned content type, X-CLIENT-ID and X-API-KEY.
3. Define optional query parameters.
4. Execute `webread` to obtain data associated with the desired request:

```
data=webread(URL,web_options,query_parameters)
```

#### Accessing Datapoint values

The primary objective of this example is to obtain twenty recent values of the `button1ColorKpi` Datapoint of KIS.BOX defined by a given URN. For that purpose the example introduced in Sect. 7.2 is utilized. Let us start with defining an appropriate URL according to Sect. 7.2.2 (Table 7.7), which can be realized as follows:

```
baseUrl='https://api.kisme.com/kisapi/v1';
assetUrn='urn:rafi:sbox:9c65f93cbcd6';
DPdef='button1ColorKpi';
finalUrl=strcat(baseUrl,'assets/',assetUrn,'/',DPdef,
'/datapointValues');
```

Subsequently, the `weboptions` structure with `HeaderFields` is defined:

```
opt=weboptions;
content={'Content-Type' 'application/json'};
client={'X-CLIENT-ID' '708ce7cc-fd57-416d-991f-f68704385c22'};
key={'X-API-KEY' '2b84bd4de38b4e92bb7bb283364477e1'};
opt.HeaderFields=[content;client;key];
```

---

Finally, `webread` can be executed:

```
data=webread(finalUrl,opt,'limit','20');
```

However, the current example, apart from retrieving data, extracts the values and occurrence times of the KIS.API data. Moreover, such a request is repeated every second and the obtained results are suitably visualized. Such a time-driven loop is repeated until a user presses any key. The code realizing all the above mentioned operations is given as follows:

```
clc; clear; close all;
baseUrl='https://api.kiseme.com/kisapi/v1';
assetUrn='urn:rafi:sbox:9c65f93cbed6';
DPdef='button1ColorKpi';
finalUrl=strcat(baseUrl,'assets/',assetUrn,+'/',DPdef,
'datapointValues');
opt=weboptions;
content={'Content-Type' 'application/json'};
client=
{'X-CLIENT-ID' '708ce7cc-fd57-416d-991f-f68704385c22'};
key={'X-API-KEY' '2b84bd4de38b4e92bb7bb283364477e1'};
opt.HeaderFields=[content;client;key];
h = figure(1);
while isempty(get(h,'CurrentCharacter'));
    data=webread(finalUrl,opt,'limit','20');
    values=[cellfun(@str2num,{data.value})];
    ind=find(values>7);
    values(ind)=[];
    times=datetime({data.timestamp},'InputFormat',
'uuuu-MM-dd'T'HH:mm:ss.SSSZ','TimeZone','UTC');
    times(ind)=[];
    stairs(times,values,'LineWidth',2);
    xlabel('Time');
    ylabel(DPdef);
    ylim([0 8]);
    pause(1);
end;
```

The obtained results are given in Fig. 7.6. As can be observed, the value of `button1ColorKpi` Datapoints is switched between black (2) and red (5) colors.

**Remark 7.1** The example presented in this point corresponds to the so-called polling procedure in which KIS.API requests are repeated every single second. This process is, of course, inefficient as it is executed irrespective of the fact of having new KIS.API data. Indeed, even if there is no new data, the request is still executed. To settle this unappealing phenomenon, KIS.ME provides the so-called websockets, which are discussed in Sect. 7.5.

## 7.4 Triggering Rules from MATLAB

The objective of this section is to show how to realize a KIS.API POST request with Matlab. For that purpose, an example with triggering the rule is engaged. Generally, the entire process boils down to the following steps:

1. Provide the URL associated with the desired request.
2. Define the `weboptions` structure `HeaderFields` containing the returned content type, X-CLIENT-ID and X-API-KEY.
3. Define optional request parameters.
4. Execute `webread` to obtain the data associated with the desired request:

```
data=webwrite(URL,web_options,request_parameters)
```

### Triggering a rule from MATLAB

The objective of this example is to show how to trigger KIS.ME rule from MATLAB. For that purpose, let us define a new rule. This rule has no triggers or conditions defined in KIS.MANGER. It aims at switching the KIS.BOX Button 2 operational LED color to yellow. Thus, there is only one action which performs the above task. Subsequently, API Integration (see Fig. 7.3) is used to set the `enabledAPI` property by activating the API-Trigger active option. The remaining information required to formulate the rule triggering request reduces to collecting

Rule ID: 13fcb514-9fe7-44a9-9802-3d972b0fee8a,  
Asset group ID: 9758.

Having the above information, it is possible to formulate the triggering request according to Table 7.11. Finally, the resulting code is given as follows:

```

clc; clear; close all;
baseUrl='https://api.kisme.com/kisapi/v1';
ruleID='13fcb514-9fe7-44a9-9802-3d972b0fee8a';
assetgroupId='9758';
addpath='rules/';
finalUrl=strcat(baseUrl,addpath,ruleID,+'/',assetgroupId,
'/trigger');
opt=weboptions;
content={'Content-Type' 'application/json'};
client=
{'X-CLIENT-ID' '708ce7cc-fd57-416d-991f-f68704385c22'};
key={'X-API-KEY' '2b84bd4de38b4e92bb7bb283364477e1'};
opt.HeaderFields=[content;client;key];
data=webwrite(finalUrl,opt);

```

---

## 7.5 Websockets

In spite of an incontestable appeal of the communication strategies presented in the preceding part of this chapter, they frequently suffer from the lack of efficiency. This is particularly the case when there is a need for observing the changes in system behaviour expressed in the evolution of Datapoints associated with KIS.Devices. Indeed, such a problem was already discussed in Sect. 7.3.2. The example considered in the preceding section concerned the so-called polling procedure, in which KIS.API requests are repeated every single second. This process is, of course, inefficient as it is executed irrespective of the fact of having new KIS.API data. Indeed, even if there is no new data, the request is still executed. To settle this unappealing phenomenon, KIS.ME provides the so-called websockets, which are discussed in this section.

### 7.5.1 Brief Introduction to Websockets

Websocket [4] can be defined as a communication protocol, which permits bidirectional communication between the client and the web server. In a most common case, if a browser visits a web page, then an HTTP request is sent to the associated server. Subsequently, the web server replies by sending the response to the web browser. A similar strategy was realized in the preceding part of this chapter while using a different kind of applications, i.e., POSTMAN, MS Excel and Matlab. Thus, as was



shown in Sect. 7.3.2, if the application wants to receive recently released data, then it must constantly, e.g., every second, send a request to the server. This corresponds to the situation in which the user constantly refreshes the page within the browser. This is also the reason why HTTP is a half duplex, which denotes the fact that the traffic flows in a single direction at a time:

- the client releases a request to the server (one direction);
- the server replies to the request (one direction).

Therefore, it is an obvious fact that it is not an elegant solution, widely called polling. It is defined as a regularly timed synchronous call in which the client releases a request to the server to check if there is any new data available. Such requests are realized using regular time intervals, and the client receives a response irrespective of the availability of the new data. Thus, if there is no new data, then the server replies with a negative response and the connection closes. Hence, polling can be efficient when an exact time interval concerning the release of the new data is known. Unfortunately, as has already been mentioned, KIS.ME is used to model a discrete event system in which the occurrence time of events is not equally distributed over a time horizon. Another communication strategy is called long polling. In this case, the client sends request to the server and opens a connection within some time period. If the server has no new data, then it holds the request and connection open until it has a new data for the client (or a predefined timeout is reached). An alternative communication strategy is called streaming. In this case, the client sends a request to the sever, which maintains an open response that is continually updated. The connection can be open permanently or until a predefined timeout is reached. Note that the server never indicates the completion of the HTTP response, and hence the connection is open continuously.

In order to eliminate the above issues, the concept of websocket was introduced. The websocket is by nature a bidirectional full duplex and single-socket connection. While using it, a single HTTP request is required to open a websocket connection. An appealing property of websocket is that it reuses the same connection in both ways, i.e., client–server and server–client. Owing to the fact that the server can send messages as they are available, the overall latency is reduced. Contrarily to polling, websocket communication is based on a single request, i.e., it is not necessary to wait for another request (along with headers, request parameters, etc.). A concise summary of using the websocket can be formulated as follows:

- it makes real-time communication much more efficient;
- it enables a simpler Web-based communication between the client and server;
- it is a network protocol that enables developing other standard protocols on top of it;
- it overcomes the drawbacks of HTTP with respect to real-time communication.

Similarly to HTTP and HTTPS, the websocket defines two URI schemes, namely, ws and wss, which correspond to standard and encrypted communication between the client and the server. The wss (Websocket Secure) URI scheme corresponds to the websocket connection over transport layer security (TLS). Note that TLS is also

known as SSL (Secure Socket Layer). Thus, the same security mechanism is used as the one employed for HTTPS. This means that while constructing websockets one should use a URL of either the `ws://` or `wss://` form.

## 7.5.2 *Obtaining a KIS.ME URI and Identifiers*

The objective of this point is to show how to retrieve a URI and an identifier necessary to construct a websocket. The first step on the way towards the above objective is to select the data of interest, which can be the following:

- Datapoints: one can obtain real-time data corresponding to Datapoints or calculated Datapoints (see Sects. 2.5 and 4.1.1) of an asset;
- KPIs: one can obtain data corresponding to KPIs (see Sects. 2.5 and 4.1.1) of an asset, which are calculated every 15 min assuming that the new underlying Datapoint values are available;
- Rules: one can obtain data associated with the triggered rules.

Thus, the required preliminary data is associated with the property `subscribeTo`, which may have the following values:

- datapoint,
- kpi,
- rule.

Subsequently, Datapoints and KPIs require a single or a list of `assetIds` (see Sect. 7.2) while rules require `assetGroupIds`. Finally, all these properties form a JSON data structure, which may look as follows:

```
{
  "assetIds": [
    37,
    66
  ],
  "assetGroupIds": [
    43,
    6
  ],
  "subscribeTo": "datapoint"
}
```

Having the above data along with KIS.API credentials (see Sect.7.1.1), one can perform a POST request according to Table 7.12. As a result, the following JSON structure can be obtained:

**Table 7.12** Requests of subscribing/unsubscribing to a websocket

No.	Verb	Path	Action
1	POST	baseUrl/websockets	Subscribe to websocket
1	DEL	baseUrl/websockets	Unsubscribe from a websocket

```
{
  "subscriptionUri": [
    "wss://pubsub.api.kisme.com/
    6d574a1f-4c37-4ab4-9fa6-86fb74a66375"
  ],
  "subscriptionId": "6d574a1f-4c37-4ab4-9fa6-86fb74a66375"
}
```

Analogously, the DEL request will unsubscribe from the websocket.

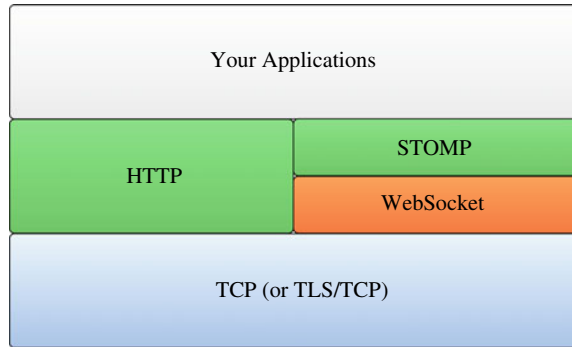
### 7.5.3 *Brief Introduction to STOMP*

Messaging [4] stands for an architecture associated with sending asynchronous messages between independent components. Such an appealing property makes it possible to develop relatively loosely coupled systems. The crucial components of messaging are the message broker and the client. In particular, the former can perform such actions:

- accepting connections of the clients,
- sending messages to the clients,
- distributing messages among the clients.

Note that the broker can also handle such operations as authorization, message encryption, etc. Thus, if clients are connected to the broker, then they can send messages to the broker as well as receive message distributed by the broker. Such a strategy is called publish/subscribe. Therefore, if a message broker publishes a number of messages, then the client can subscribe to either all or a subset of these messages. STOMP (simple text-oriented messaging protocol) is a good representative example of such a publish/subscribe protocol. Its layering relation with the websocket as well as with other protocols is detailed in Fig. 7.7. STOMP was also employed for the communication purposes within KIS.ME. Indeed, the websocket fits very well to a standard messaging architecture, in which there could be a large volume of potential messages distributed at high rates from the broker to the client. A good example is a client subscribing to Datapoints of an asset (see Sect. 7.5.2). Due to the relatively large number of Datapoints as well as their possible high rate of change,

**Fig. 7.7** STOMP over a websocket



receiving messages in real-time as well as with low latency is extremely important for the final performance of the entire application. STOMP is a very simple protocol, which resembles HTTP in its appearance. Each frame consists of a command, headers, etc. STOMP messages can represent any text or binary data. For further information about STOMP, the reader is referred to the STOMP protocol specification [5]. Additionally, STOMP [5] provides the so-called heart-beating mechanism, which can optionally be employed to verify the healthiness of the underlying TCP connection and to ensure that the remote end is still alive and kicking. Generally, it is defined by two integer values, separated by a comma. The first one represents outgoing heart-beats from the sender:

- 0 signifies the fact that it cannot send heart-beats;
- otherwise it is the smallest number of milliseconds between heart-beats that it can guarantee.

The second one represents incoming heart-beats, i.e., what sender would like to obtain:

- 0—stands for the fact that it does not want to receive heart-beats;
- otherwise it is the desired number of milliseconds between heart-beats that it can guarantee.

Note that enabling heart-beating is possible by adding a suitable heart-beating header during the beginning of the STOMP session, i.e., to CONNECT [5].

### 7.5.4 *Sample Websocket Implementations*

The objective of this section is to provide guidelines for practical implementation of KIS.ME-based websockets. In particular, the NODE.js [1] environment, which employs a widely employed JavaScript (JS) programming language is used. This section is composed of two examples, which aim at

1. reading KIS.ME data using STOMP over websocket,
2. enhancing the above example with a local Web server employed for publishing KIS.ME data.

Moreover, it is assumed that the reader has essential knowledge regarding NODE.js.

### Reading KIS.ME data using STOMP over a websocket

Let us start with providing suitable credentials and parameters, which will be located in the `.env` file:

```
SERVER_URL="https://api.kisme.com/kisapi/v1/websockets"
API_KEY="2b84bd4de38b4e92bb7bb283364477e1"
CLIENT_ID="708ce7cc-fd57-416d-991f-f68704385c22"
ASSET_ID="10102"
ASSET_GROUP_ID="9758"
```

For the purpose of further implementations, the following modules are required:

`dotenv`: loads environment variables from the `.env` file into `process.env` structure;

`websocket`: implements the websocket protocol;

`webstomp-client`: provides a STOMP client for Web browsers and NODE.js through websockets;

`axios`: is a promise-based HTTP client for the Web browser and NODE.js.

Note that the application of the above modules is not compulsory and there are several counterparts which can be employed instead. Moreover, their documentation can be easily found at <https://www.npmjs.com>. After such a preliminary step, it is possible to define suitable request headers and options, which are given as follows:

```
const options = {
  method: "POST",
  url: process.env.SERVER_URL,
  headers: {
    "X-API-KEY": process.env.API_KEY,
    "X-CLIENT-ID": process.env.CLIENT_ID,
    "Content-Type": "application/json",
  },
  data: JSON.stringify({
    assetIds: [parseInt(process.env.ASSET_ID)],
    assetGroupIds: [parseInt(process.env.ASSET_GROUP_ID)],
    subscribeTo: "datapoint",
  }),
};
```

while the underlying POST request concerns subscription to a websocket (see Sect. 7.5.2). Thus, the objective is to obtain (cf. `subscribeTo`) the values of

Datapoints and calculated Datapoints of KIS.Devices associated with assetIds and assetGroupIds. Note that the last property is not compulsory for obtaining Datapoint values. Subsequently, let us assume that both incoming and outgoing heart-beating is set to 1000 ms. Having all the above ingredients, the final code is developed, which is mostly included in the getSubscriptionId function:

```

require("dotenv").config();
const WebSocket = require("websocket").w3cwebsocket,
webstomp = require("webstomp-client");
const axios = require("axios");
const heartbeat = 1000;
const getSubscriptionId = () => {
  const options = {
    method: "POST",
    url: process.env.SERVER_URL,
    headers: {
      "X-API-KEY": process.env.API_KEY,
      "X-CLIENT-ID": process.env.CLIENT_ID,
      "Content-Type": "application/json",
    },
    data: JSON.stringify({
      assetIds: [parseInt(process.env.ASSET_ID)],
      assetGroupIds: [parseInt(process.env.ASSET_GROUP_ID)],
      subscribeTo: "datapoint",
    }),
  };
  axios(options)
    .then(function (response) {
      const subscriptionId=response.data.subscriptionId;
      const Server=
response.data.subscriptionUris[0].replace("///","//");
      const socket = new WebSocket(Server, null);
      const stomp = webstomp.over(socket, {
        heartbeat: {incoming: heartbeat, outgoing: heartbeat},
        protocols: ['v12.stomp'],
      });
      stomp.connect(
        {host: Server},
        function () {
          stomp.subscribe(`/topic/${subscriptionId}`,
            function (message) {
              const data = JSON.parse(message.body);
              console.log("Message data",data.info);
            });
        },
      );
    });
};

```

```

    })
    .catch(function (error) {
        console.log(error);
    })
    .finally(function () {
        // always executed
    });
});
};

getSubscriptionId();

```

The example considered is a continuation of the ones exploited in this chapter for which the KIS.BOX (`assetIds=10102`) Button 1 operational LED color switches between red and black. The above KIS.BOX has also associated calculated Datapoint and KPI. After running the above code, one can observe that `message.body` contains the JSON structure, which may look as follows:

```

{"jsonType":"centersightEvent",
 "type":"datapointValuesReceived",
 "nodeId":10102,"timestamp":"2022-10-06T10:47:47.833Z",
 "info":{"key":"button1Color","value":"#000000",
 "timestamp":"2022-10-06T10:47:47.833Z"}}

```

The above JSON structure is self-explained and it can be easily observed that it contains the `info` property, which covers another JSON structure involving

- key: the name of the (calculated) Datapoint,
- value: the value of the (calculated) Datapoint,
- timestamp: the timestamp associated with the value of the (calculated) Datapoint,

and hence, this structure is directly displayed in the console. Note that the above code can be easily adapted to the remaining possible settings of `subscribeTo`, i.e., `kpi` and `rule`. However, such an implementation is left out to be featured an exercise listed in the last section of this chapter.

### > Getting information about the rules

Contrarily, to KPIs and Datapoints, rules are directly associated with asset groups. Indeed, there are designed within each asset group. This implies the necessity of a reduced subscription data:

```

{
  assetGroupIds: [parseInt(process.env.ASSET_GROUP_ID)],
  subscribeTo: "rule",
}

```

while `assetIds` is excluded.

---

### Publishing KIS.ME data with a local Web server

The objective of this example is to extend the proceeding one in such a way as to provide the following functionalities:

- feeding the selected Datapoint data to another bi-directional third party API,
- a frontend displaying the selected Datapoint data obtained from the above API.

Let us start with selecting an API. Since the purpose of the example is to collect the data in the form of a JSON structure containing three properties (key, value and timestamp) the list of possible candidates is rather long. Thus, due to relative usage simplicity, the Pusher API was selected (<https://pusher.com/>). As was the case with KIS.API, the first step is to register with Pusher and then collect the list of credentials (App keys). A sample list of Pusher credentials is given as follows:

```

app_id = "1482901"
key = "8e17772867a31d055131"
secret = "b2632afdbd8419d40bf1"
cluster = "eu"

```

Thus, let us extend the `.env` file with the above data, which yields

```

SERVER_URL="https://api.kisme.com/kisapi/v1"
API_KEY="2b84bd4de38b4e92bb7bb283364477e1"
CLIENT_ID="708ce7cc-fd57-416d-991f-f68704385c22"
ASSET_ID="10102"
ASSET_GROUP_ID="9758"
app_id = "1482901"
key = "8e17772867a31d055131"
secret = "b2632afdbd8419d40bf1"
cluster = "eu"

```

Having the above information, let us simply extend the code from the previous example with a list of commands creating a Pusher instance:



```
const Pusher = require("pusher");
const pusher = new Pusher({
  appId: "1482901",
  key: process.env.key,
  secret: process.env.secret,
  cluster: process.env.cluster,
});
```

Now let us assume that the Datapoint of interest is called `button1ColorKpiDuration` (see Appendix. B for its description). Thus, the transfer of Datapoint values to the Pusher API reduces to the following:

```
if (data.info.key=="button1ColorKpiDuration")
  pusher.trigger("b1ColorKpiDuration", "b1ColorKpiDuration",
    {
      value: JSON.stringify(data.info),
    });
```

where `b1ColorKpiDuration` signifies both the so-called channel and event (see <https://www.npmjs.com/package/pusher> for a detailed explanation). The preparation of the backend concludes with including the code for the local Web server. For that purpose the `express` module is used, which can be simply characterized as a lightweight NODE.js Web server. The entire code reduces to adding the following lines:

```
const express = require("express");
const app = express();
app.use(express.static(__dirname + '/public'));
app.listen(3000, () => {
  console.log("Server running on: http://localhost:3000/");
});
```

which are responsible for creating an `express` Web server that will run on port 3000 and will communicate with server static files located in the `public` directory. Finally, the entire backend code can be implemented as follows:

```
require("dotenv").config();
const WebSocket = require("websocket").w3cwebsocket,
webstomp = require("webstomp-client");
const axios = require("axios");
const Pusher = require("pusher");
const pusher = new Pusher({
  appId: "1482901",
  key: process.env.key,
  secret: process.env.secret,
```

```

    cluster: process.env.cluster,
  });
const heartbeat = 1000;
const getSubscriptionId = () => {
  const options = {
    method: "POST",
    url: process.env.SERVER_URL,
    headers: {
      "X-API-KEY": process.env.API_KEY,
      "X-CLIENT-ID": process.env.CLIENT_ID,
      "Content-Type": "application/json",
    },
    data: JSON.stringify({
      assetIds: [parseInt(process.env.ASSET_ID)],
      assetGroupIds: [parseInt(process.env.ASSET_GROUP_ID)],
      subscribeTo: "datapoint",
    }),
  };
  axios(options)
    .then(function (response) {
      const subscriptionId=response.data.subscriptionId;
      const Server=
response.data.subscriptionUris[0].replace("///", "//");
      const socket = new WebSocket(Server, null);
      const stomp = webstomp.over(socket, {
        heartbeat: {incoming: heartbeat, outgoing: heartbeat},
        protocols: ['v12.stomp'],
      });
      stomp.connect(
        {host: Server},
        function () {
          stomp.subscribe(`/topic/${subscriptionId}`,
            function (message) {
              const data = JSON.parse(message.body);
              if (data.info.key=="button1ColorKpiDuration")
                pusher.trigger("b1ColorKpiDuration",
                  "b1ColorKpiDuration",
                  {
                    value: JSON.stringify(data.info),
                  });
            });
        });
    },
  );
})
.catch(function (error) {
  console.log(error);
});

```

```

    })
    .finally(function () {
      // always executed
    });
  });
  getSubscriptionId();
  const express = require("express");
  const app = express();
  app.use(express.static(__dirname + '/public'));
  app.listen(3000, () => {
    console.log("Server running on: http://localhost:3000/");
  });

```

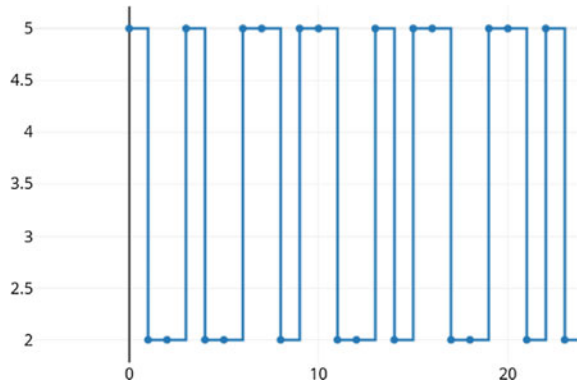
Let us proceed to the frontend development by creating the `public` directory and the `index.html` file inside it. The presentation of Datapoint values will be reduced to showing its consecutive values in the form of a plot. For that purpose the well known `plotly` is employed. Thus, the entire frontend reduces to the following code:

```

<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="content-type" content="text/html;
    charset=UTF-8" />
  <script src="https://cdn.plot.ly/plotly-latest.min.js">
  </script>
  <script src="https://js.pusher.com/7.2.0/pusher.min.js">
  </script>
</head>
<body>
  <div id="chart"></div>
<script>
  Pusher.logToConsole = true;
  var my_plot = {
    y: [],
    mode: 'lines+markers',
    name: 'hv',
    line: {shape: 'hv'},
    type: 'scatter',
  };
  Plotly.newPlot('chart', [my_plot]);
  const pusher = new Pusher(
    "8e1772867a31d055131", // Replace with your 'key'
    { cluster: "eu", }
  );
  const channel = pusher.subscribe("b1ColorKpiDuration");

```

**Fig. 7.8** Plotting datapoint values with a local Web server



```

var cnt=0;
var width_graph_window=23;
channel.bind("b1ColorKpiDuration", (data) => {
  const obj=JSON.parse(data.value);
  Plotly.extendTraces('chart', {y:[[obj.value]]}, [0]);
  cnt++;
  if( cnt > width_graph_window)
    Plotly.relayout('chart', {
      xaxis:
        { range: [cnt-width_graph_window, cnt] }
    });
});
</script>
</body>
</html>

```

The main part of the code starts with creating an empty plot located in the `chart` section of the HTML document (`newPlot`). Subsequently, a new `Pusher` instance is created with the above-defined credentials named `key` and `cluster`. This enables forming a new channel `b1ColorKpiDuration`. Finally, the `channel.bind` command is responsible for receiving cyclically arriving `Datapoint` values. It also invokes the `extendTrace` command, which updates the existing plot with the new data. Note that the plot is restricted to presenting 23 most up to date values, which requires appropriate scaling realized with the `relayout` command. The graphical result obtained after running the entire application, i.e. the one presented in the browser, is given in Fig. 7.8.

## 7.6 Training Exercises

### 7.1 Obtaining a list of assets

1. Obtain a JSON structure containing all KIS.Devices which are at your disposal.
2. Find all assets which are on-line.

### 7.2 Obtaining a list of asset groups

1. Obtain a JSON structure containing all asset groups.
2. Modify the name and description of a selected asset group.

### 7.3 Obtaining a list of users

1. Obtain a JSON structure containing all users and determine their `accountNumber`.
2. Obtain a user and determine the above JSON structure once again.
3. Delete the added user.

### 7.4 Obtaining a list of Datapoints and CDPs

1. Select a KIS.Device and determine its URN.
2. Obtain a JSON structure containing a list of all Datapoints.
3. Determine a list of CDPs.

### 7.5 Obtaining values of Datapoints and CDPs

1. Choose a KIS.Device and write a set of rules transferring automatically and cyclically its selected operational LED within a state-space: red, yellow, green.
2. Obtain a JSON structure containing a list of 10 recent values of Datapoints corresponding to the numerical values of the operational LED's colors.
3. Prepare CDP converting the numerical values of the operational LED's colors in such a way so that red corresponds to 0, yellow is signified by 1, while green yields 2.
4. Obtain a JSON structure containing a list of 10 recent values of the above CDP.

### 7.6 Obtaining values of KPIs

1. Continue Exercise 7.5 by implementing KPIs calculating mean times of each state, i.e., red, yellow and green.
2. Obtain a JSON structure containing all KPIs.
3. Obtain a JSON structure containing a list of 10 recent values of the above KPIs.

### 7.7 Obtaining information about rules

1. Continue Exercise 7.5 and display the information about rules used for color state transitions.
2. Import the information about the rules to MS Excel or any compatible software.

## 7.8 Triggering rules from an external application

1. Select a KIS.LIGHT.
2. Prepare two rules with API-Trigger active (see Fig. 7.3):
  - GoRed: with an action setting the KIS.LIGHT operational LED color to red;
  - GoGreen: with an action setting the KIS.LIGHT operational LED color to red;
3. Develop a Matlab (you can also use OCTAVE or Python) program, which will trigger these rules every second one after the other.

## 7.9 Websocket implementation

1. Continue Exercise 7.5 and develop websocket-based application according to the scheme presented in Sect. 7.5.4.
2. Use the above-developed software to get information about the triggered rules and KPI calculation. Hint: use the `subscribeTo` property.

## 7.10 Websocket implementation

1. Continue Exercise 7.9 and extend it according to Sect. 7.5.4 in order to get a local web server employed for presenting selected Datapoint values.
2. On the frontend side (`index.html`) implement a functionality calculating the total accumulated time of the red color state, i.e., the overall time in which the operational LED color is red.

## 7.7 Concluding Remarks

The aim of this section was to provide an overview of KIS.API functionalities. It was demonstrated that the software provides an effective way for communicating with external applications. In particular, the chapter started with KIS.API user registration and goes through KIS.API essential functionalities. These functionalities are strictly linked with the content of the preceding chapter. Indeed, it is shown how to prepare request for accessing the information about assets, asset groups, users, Datapoints, CDPs, KPIs as well as the rules. The crucial functionality, which makes KIS.API a fully bidirectional framework is the ability of triggering the rules from external applications. In particular, it was shown how to trigger such rules from Matlab, which is one of the most popular development tools in modern engineering. The last part of the chapter was devoted to the development of efficient websocket-based communication framework, which utilizes STOMP messaging architecture. Indeed, the websocket is an excellent for providing an efficient asynchronous bidirectional communication. It was also demonstrated how to prepare a local Web server for publishing Datapoint values. This crucially example clearly shows that with KIS.API there is an infinite spectrum of possible external extensions of KIS.ME. Finally, the chapter is summarized with a set of training exercises, which verify KIS.API-oriented skills.

## References

1. F. Doglio, *Pro REST API Development with Node.js* (Apress, Berkeley, 2015)
2. A. Tamboli, *Build Your Own IoT Platform* (Springer, Berlin, 2019)
3. S. Patni, *Pro RESTful APIs* (Springer, Berlin, 2017)
4. V. Wang, F. Salim, P. Moskovits, *The Definitive Guide to HTML5 WebSocket*, vol. 1 (Springer, New York, 2013)
5. Stomp. <http://stomp.github.io/stomp-specification-1.2.html#Heart-beating>. Accessed: 10 June 2022

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

