



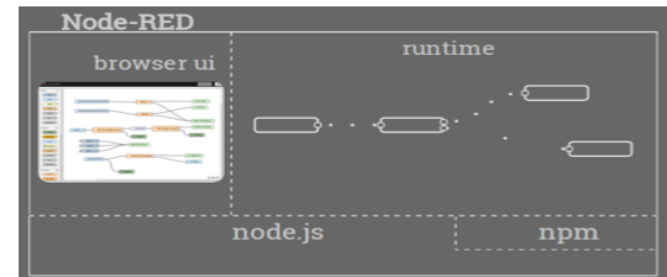
Sistemi e Tecnologie Industriali Intelligenti
per il Manifatturiero Avanzato
Consiglio Nazionale delle Ricerche

Node-RED Tutorial for linked-data

Walter Terkaj, STIIMA-CNR

*Summer School of LDAC
Lisboa, 18/06/2019*

- Node-RED is a powerful tool for building Internet of Things (IoT) applications with a focus on simplifying the 'wiring together' of code blocks to carry out tasks. It uses a visual programming approach that allows developers to connect predefined code blocks, known as 'nodes', together to perform a task. The connected nodes, usually a combination of input nodes, processing nodes and output nodes, when wired together, make up 'flows'. ([Link](#))
- Node-RED provides a web browser- based flow editor, which can be used to create JavaScript functions. Elements of applications can be saved or shared for re-use. The runtime is built on Node.js. The flows created in Node-RED are stored using JSON which can be easily imported and exported for sharing with others. By understanding Node-Red, IoT development can be accelerated without unnecessary coding.



References

- Part of the contents in this tutorial can be found on the Node-Red website: <https://nodered.org/>
- Installation instructions can be found at: <https://nodered.org/docs/getting-started/installation>
- Node RED Programming Guide: <http://noderedguide.com/>
- Introduction to Node-RED: <http://www.steves-internet-guide.com/node-red-overview/>

Installation of Node-RED

In order to be able to work with Node-RED, you should first install node.js. It is recommended the use of Node.js **LTS 8.x or 10.x**. Node-RED no longer supports Node.js 6.x or earlier. You can download the latest version of node.js from its website choosing which operating system you are using:

<https://nodejs.org/en/download/>

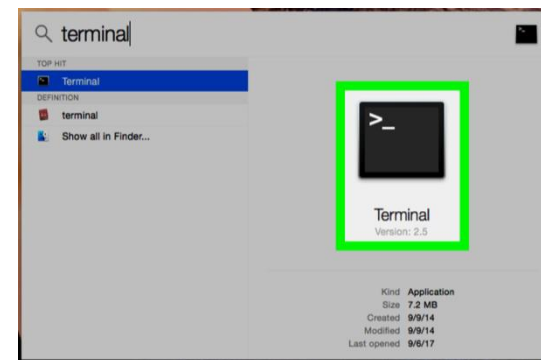


Linux / OsX

Once installed node.js, open the *terminal* window and run the following commands.

To check your version of Node.js

```
node -v
```



The easiest way to install Node-RED is to use the node package manager, npm, that comes with Node.js.

Installing as a global module adds the command node-red to your system path:

```
sudo npm install -g --unsafe-perm node-red
```

Windows

Run the downloaded MSI installer of Node.js. Local administrator rights are needed. Accept the defaults settings when installing. After installation completes, close any open command prompts and re-open to ensure new environment variables are picked up.

Once installed, open a command prompt and run the following command to ensure Node.js and npm are installed correctly.

```
node --version && npm --version
```

You should receive back output that looks similar to:

```
v8.9.0  
5.5.1
```

Installing Node-RED as a global module adds the command node-red to your system path. Execute the following at the command prompt:

```
npm install -g --unsafe-perm node-red
```

If you have installed Node-RED as a global npm package, you can launch node-red in the command prompt (Windows):

```
C: \>node-red
```

or in the terminal (Linux):

```
$ node-red
```

This will output the Node-RED log to the terminal. You must keep the terminal or command prompt open in order to keep Node-RED running. Note that running Node-RED will create a new folder in your %HOMEPATH% folder called .node-red. This is your userDir folder, think of it as the home folder for Node-RED configuration for the current user.

You can then open the Node-RED graphical editor by pointing your browser at <http://localhost:1880>

Deploy button (to push before executing a flow)

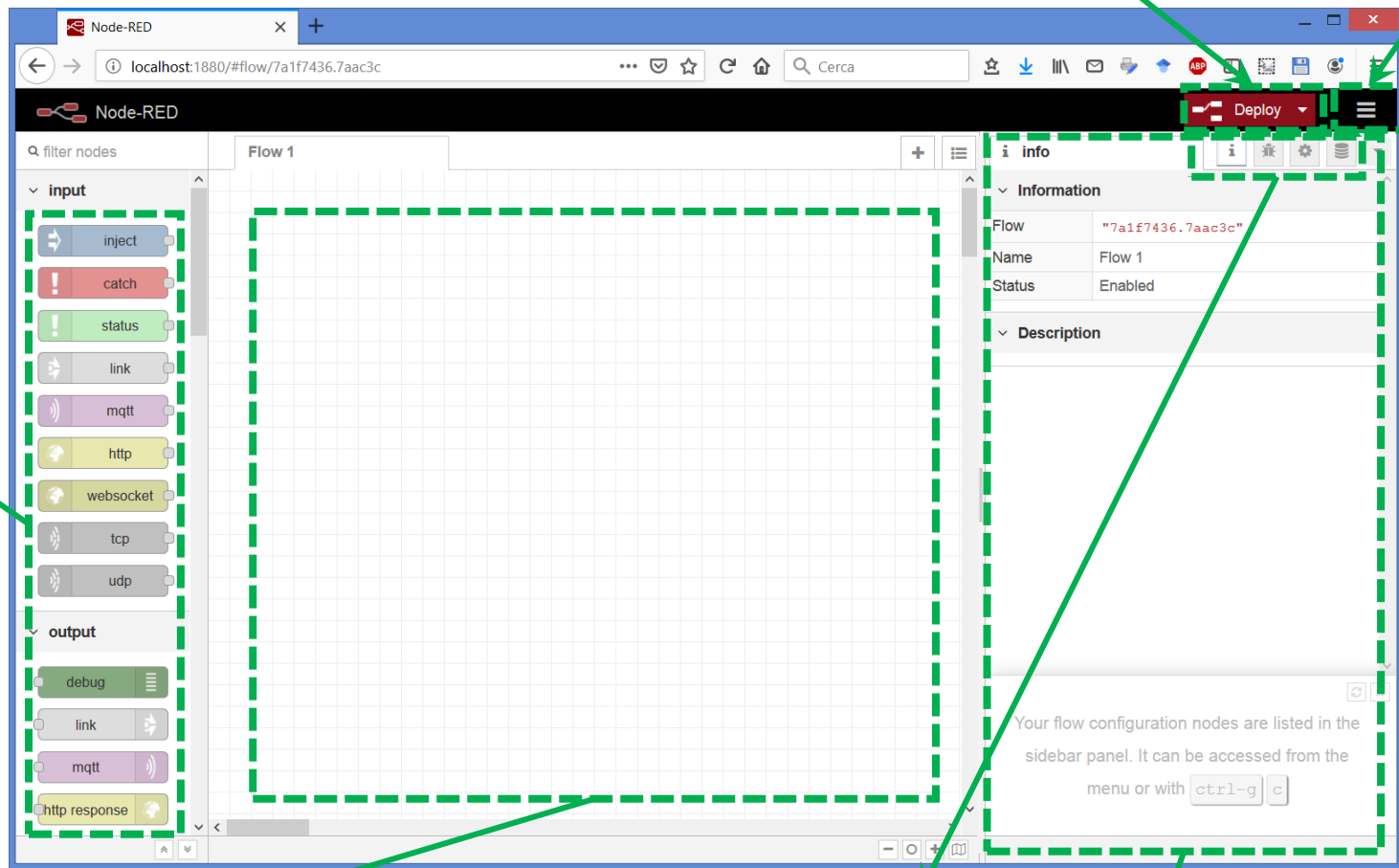
Menu button

Palette of node types

Layout to draw flows

Panel showing contents:

- **Info of selected node**
- **Debug**
- Configuration Nodes
- Context Data



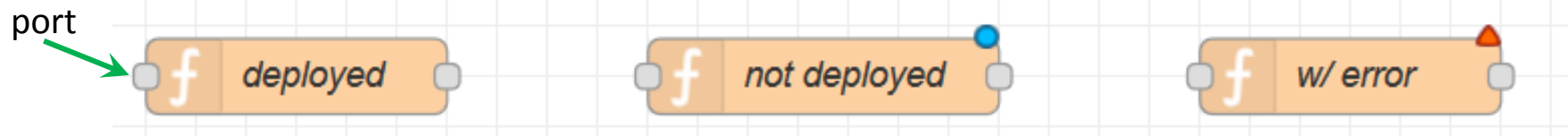
A Node-RED program is developed as a **FLOW** of messages running through a sequence of **NODES**. Each node can implement an elaboration of the message.

Nodes

Nodes consist of code that runs in the Node-RED service (javascript .js file), and an HTML file consisting of a description of the node, so that it appears in the node pane with a category, colour, name and an icon, code to configure the node, and help text.

Nodes are added to a flow by simple **drag&drop**.

A node can be linked to (multiple) input and (multiple) output via its ports which enable messages to be passed between nodes.



If a node has been change after the latest deployment, then it displays a blue circle above it. If there are errors with its configuration, it displays a red triangle.

Some nodes include a button on either its left or right edge. These allow some interaction with the node from within the editor. The **Inject** and **Debug** nodes are the only core nodes that have buttons.



A node configuration can be edited by double clicking on the node, or pressing Enter when the workspace has focus. If multiple nodes are selected, the first node in the selection will be edited.

The node edit dialog has typically three sections: Properties, Description, Appearance. The Properties section is used to set what the node does.

The screenshot shows the Node-RED web interface in a browser window. The address bar shows 'localhost:1880/#flow/74a51e5f.a7aba'. The main workspace displays a flow with a 'set msg.payload' node. The 'Edit change node' dialog is open, showing the 'Properties' section with a 'Name' field and a 'Rules' section. The 'Rules' section contains a rule: 'Set' with 'msg.payload' as the source and 'a-z' as the target. The right sidebar shows the 'info' panel with 'Information', 'Description', and 'Node Help' sections. The 'Node Help' section provides details about the 'Set' node: 'Set, change, delete or move properties of a message, flow context or global context. The node can specify multiple rules that will be applied in the order they are defined. Details The available operations are: Set set a property. The value can be a variety of different types, or can be taken from an existing message or context property.'

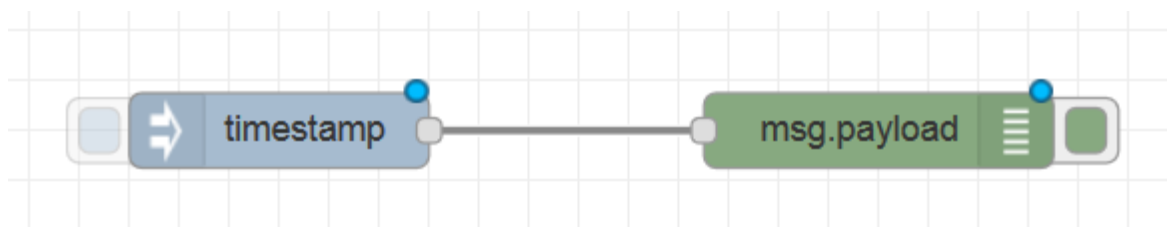
Wires define the connections between node input and output endpoints in a flow.

They (typically) connect the output endpoints of nodes to inputs of downstream nodes indicating that messages generated by one node should be processed by the connected node next.

It is possible to connect more than one node to an endpoint using wires. It is also possible to connect downstream nodes to upstream nodes to form loops.

When multiple nodes are connected to an output endpoint, messages are sent to each connected node in turn in the order they were wired to the output.

When more than one node output is connected to an input endpoint, messages from any of those nodes will be processed by the connected node when they arrive.



A simple flow generating a timestamp that is received by a debug node.

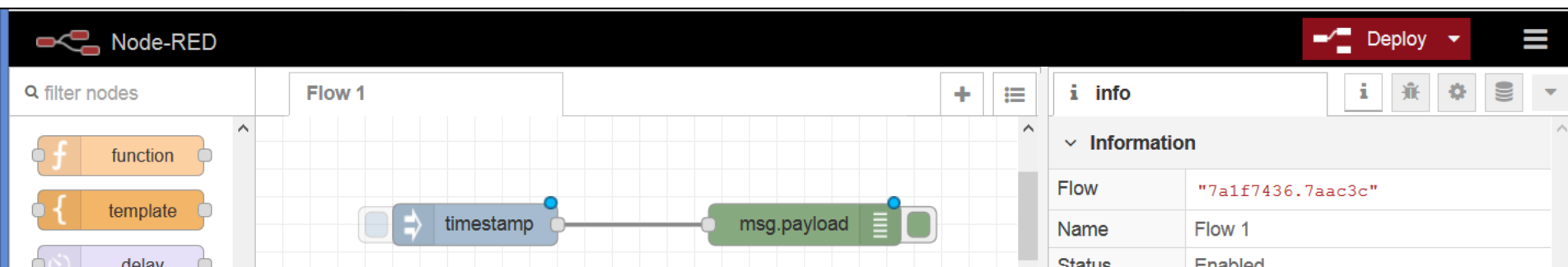
How to create wires?

Left-click on an output node port and, while holding down the mouse button, move to the destination input node port. Release the mouse button.

Flow Deployment is needed before executing the flow itself.

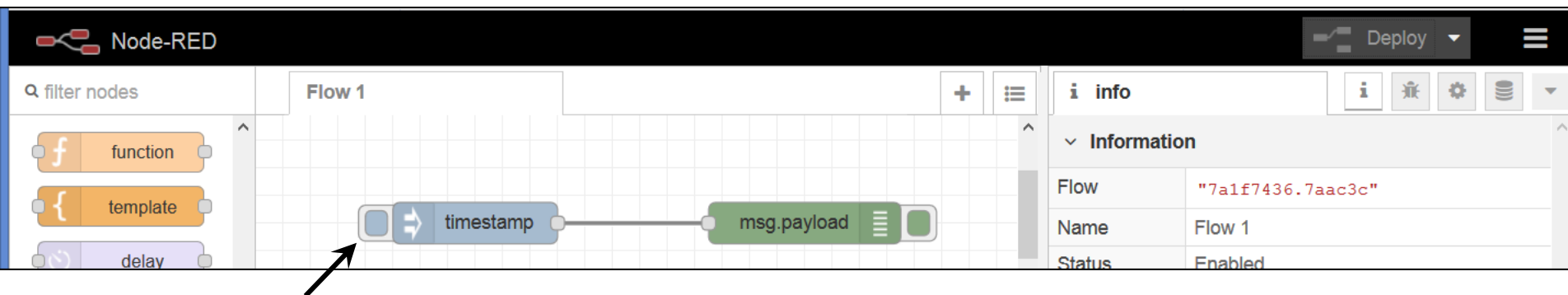
The Deploy button is on the top right corner and changes colour from grey to red when any change has been made to a flow to indicate that it needs to be deployed.

After deployment the flow can be run, e.g. clicking on the button of an Inject node.



The screenshot shows the Node-RED interface. On the left, there is a palette with nodes: function, template, and delay. The main workspace contains a flow named "Flow 1" with two nodes: a blue "timestamp" node and a green "msg.payload" node. On the right, the "info" panel is open, showing the flow's ID, name, and status. The "Deploy" button in the top right corner is grey.

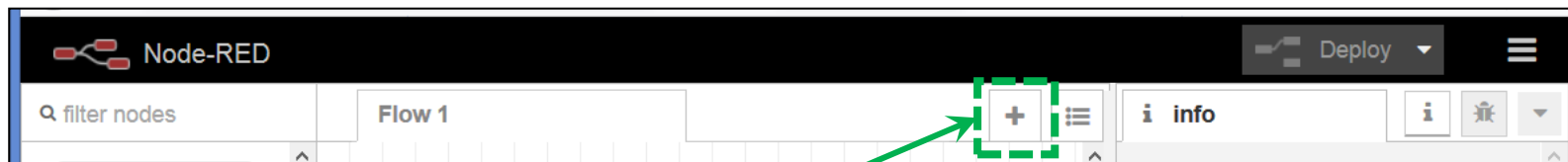
Information	
Flow	"7a1f7436.7aac3c"
Name	Flow 1
Status	Enabled



This screenshot is identical to the one above, but with an arrow pointing to the active button of the "timestamp" node. The "Deploy" button remains grey.

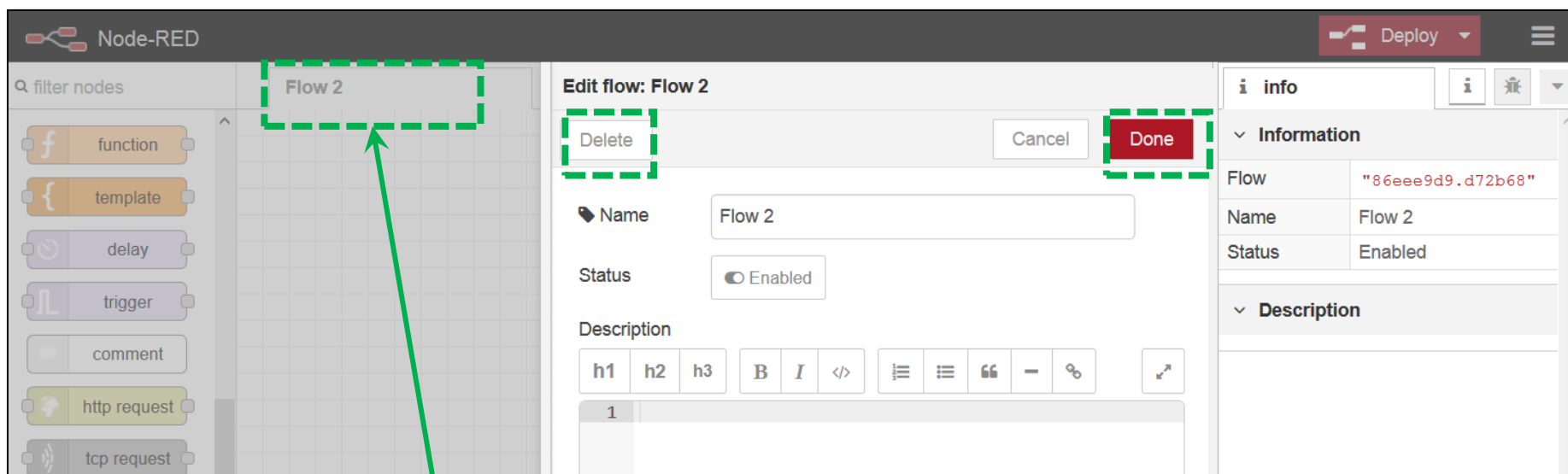
Active button of
Inject node

Create a Flow: click on the '+' tab



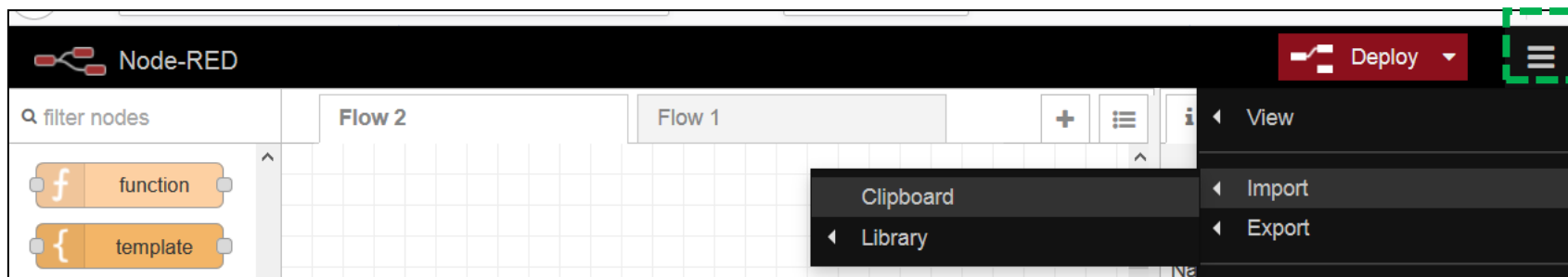
Delete a Flow: double-click on the flow tab and press the button «Delete»

Rename a Flow: double-click on the flow tab, change the name and press the button «Done»

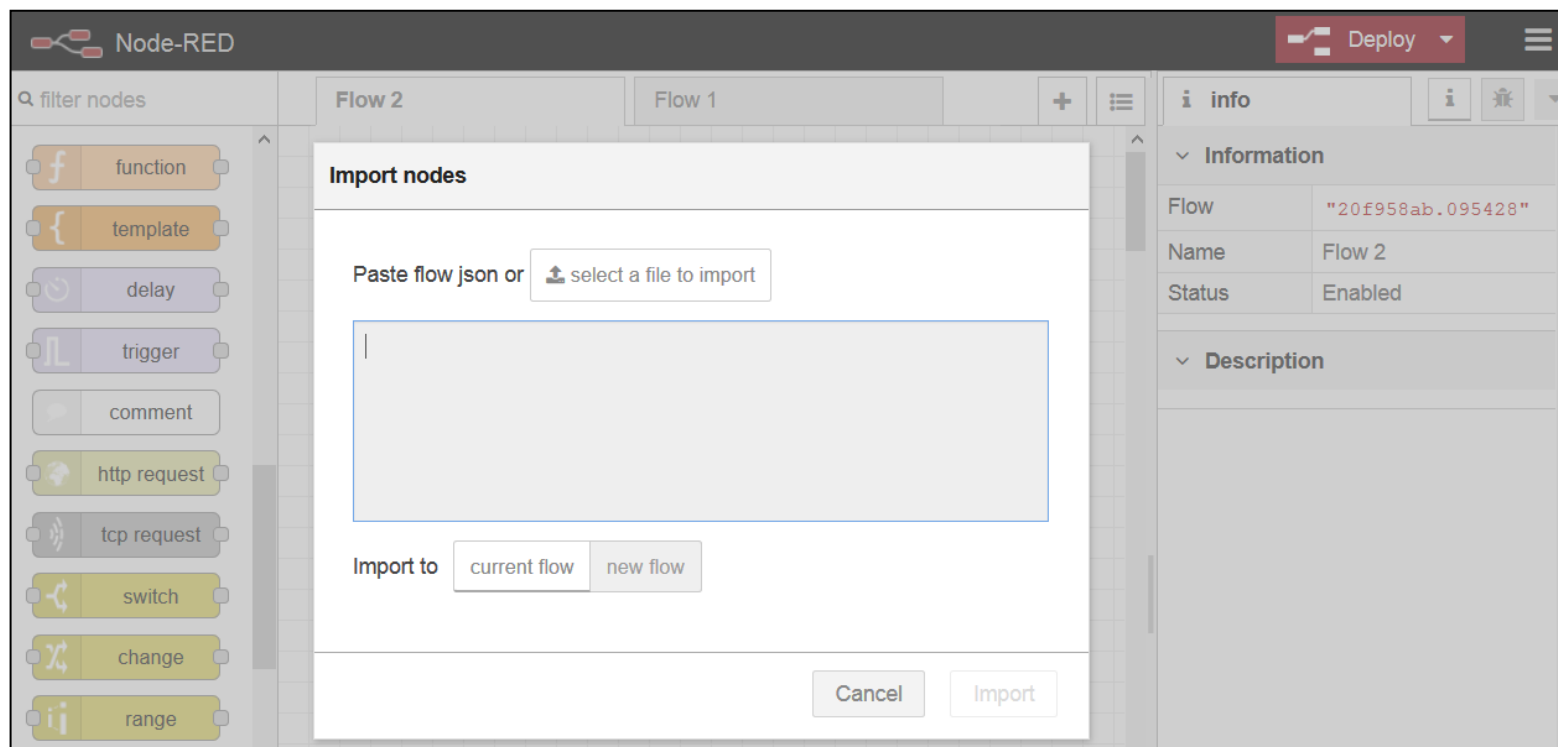


double-click on the flow tab

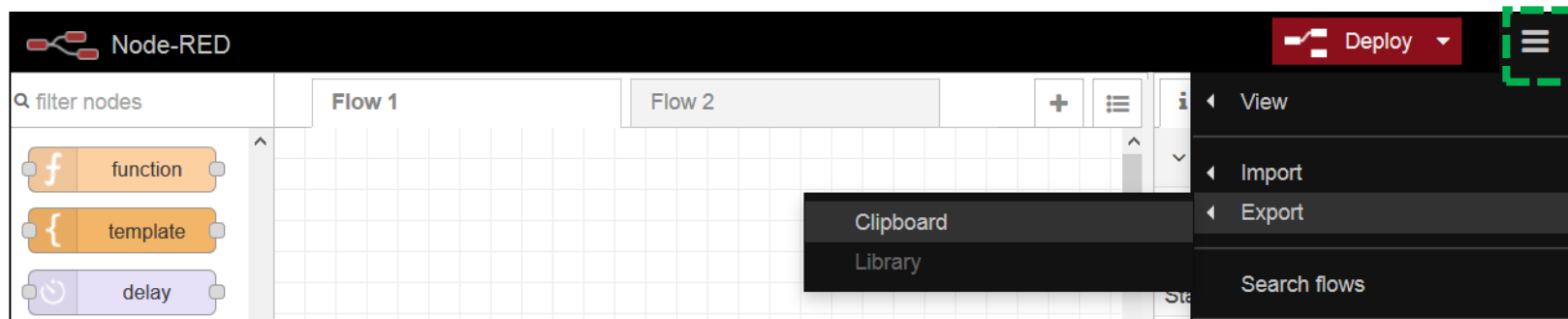
Import a Flow: click on the main menu and select «Import», «Clipboard»



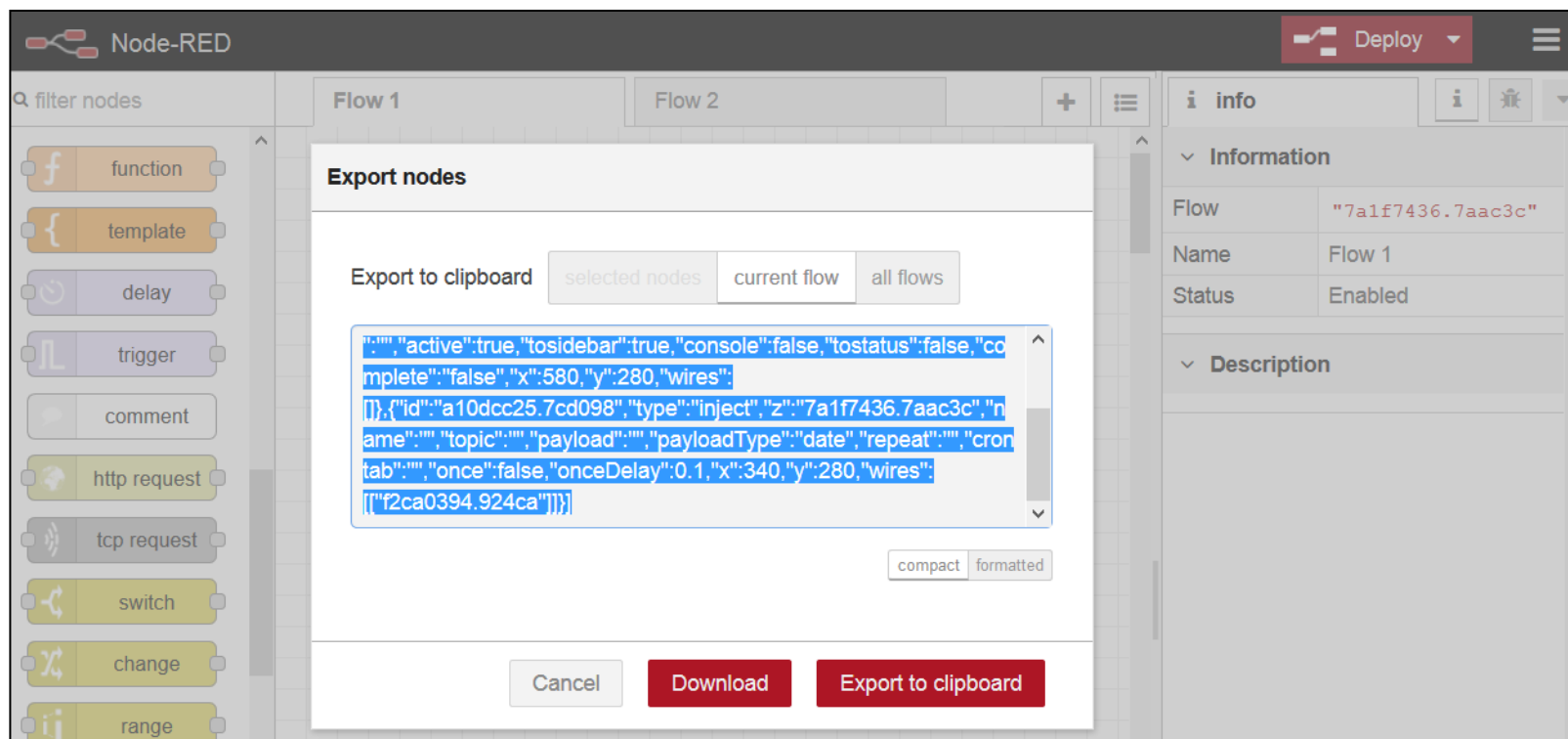
Then copy&paste the flow or select an input Json file



Export a Flow: click on the main menu and select «Export», «Clipboard»

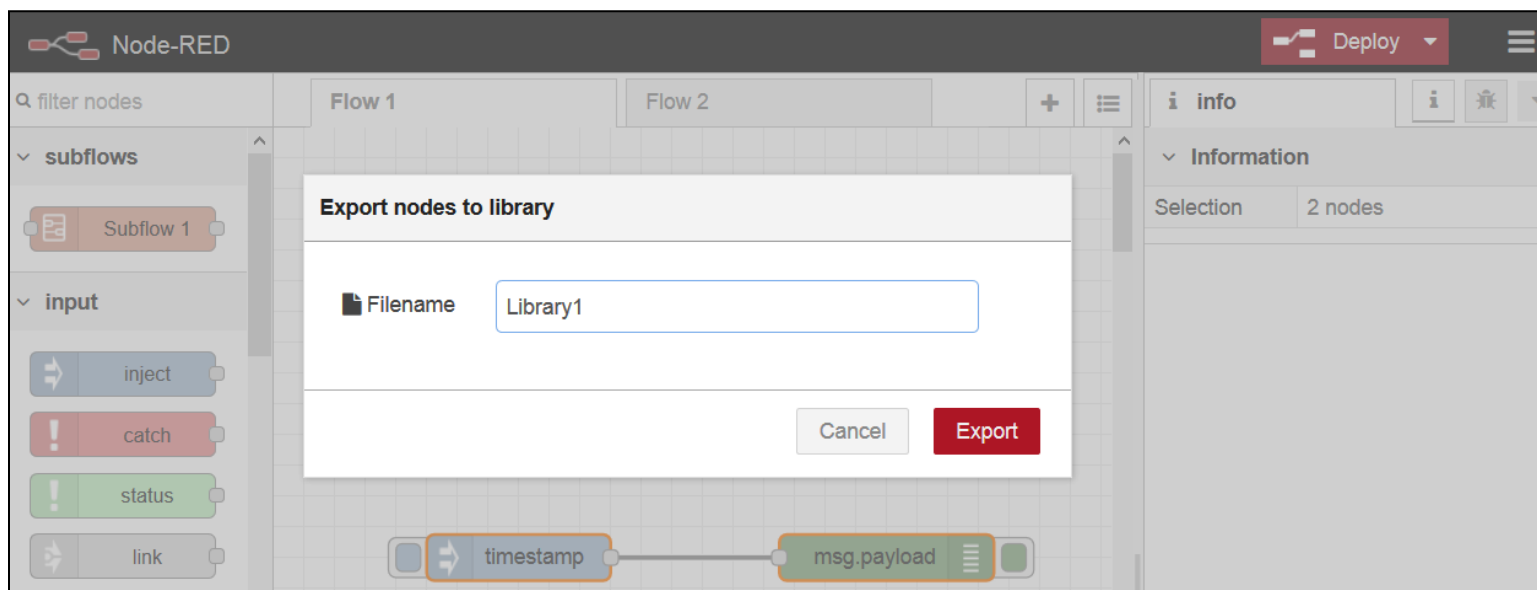


Then copy the flow or download it as a Json file.

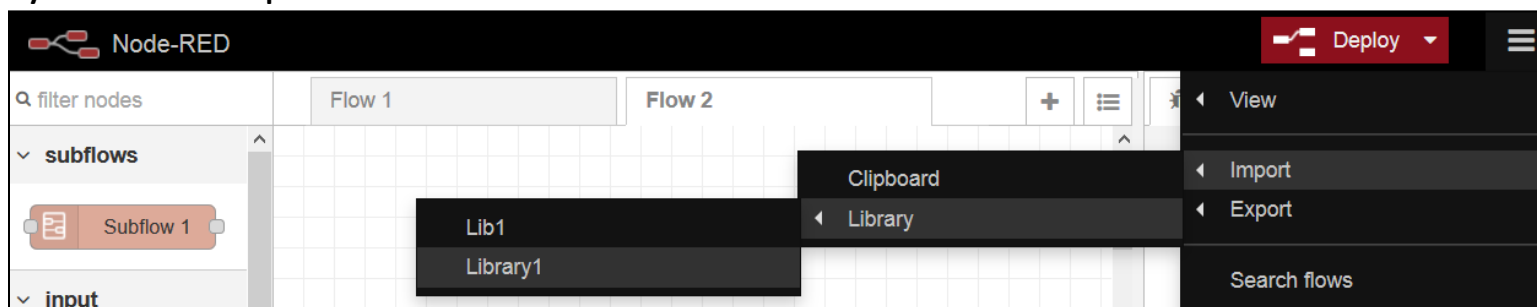


Libraries can be created to save a flow or function for reuse it in another flow .

Save a flow to a library: select the relevant nodes in the flow, then click on the main menu > **Export > Library**. Assign a name to the library



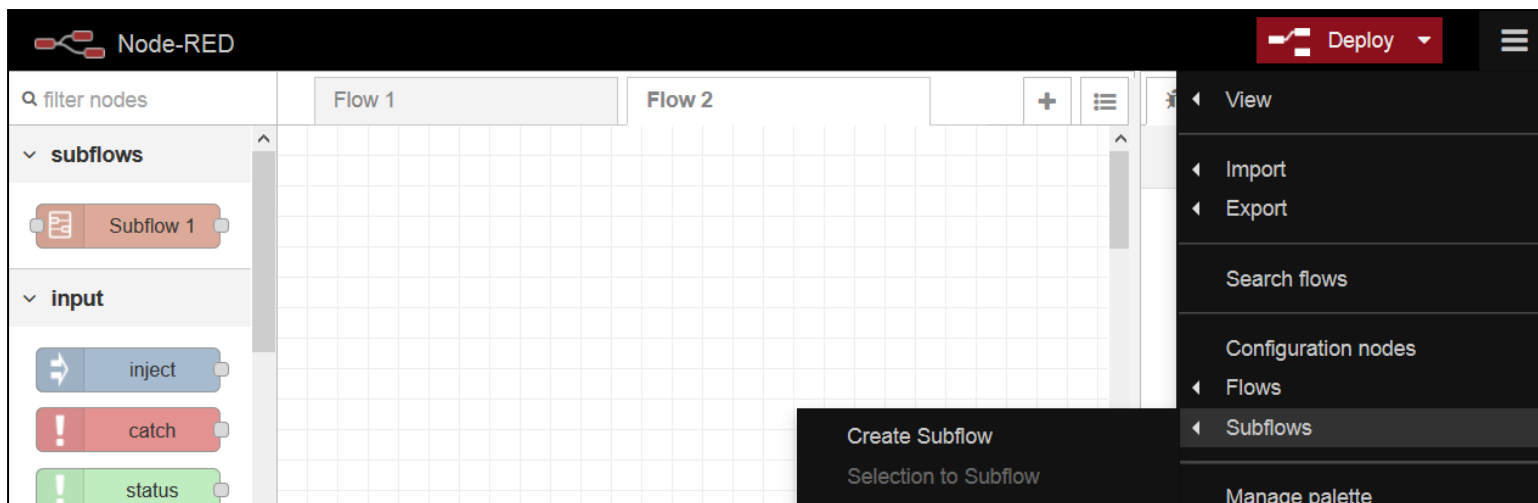
Import a flow from a library: click on the main menu > **Import > Library**. Click on the chosen library name and place the nodes in the flow.



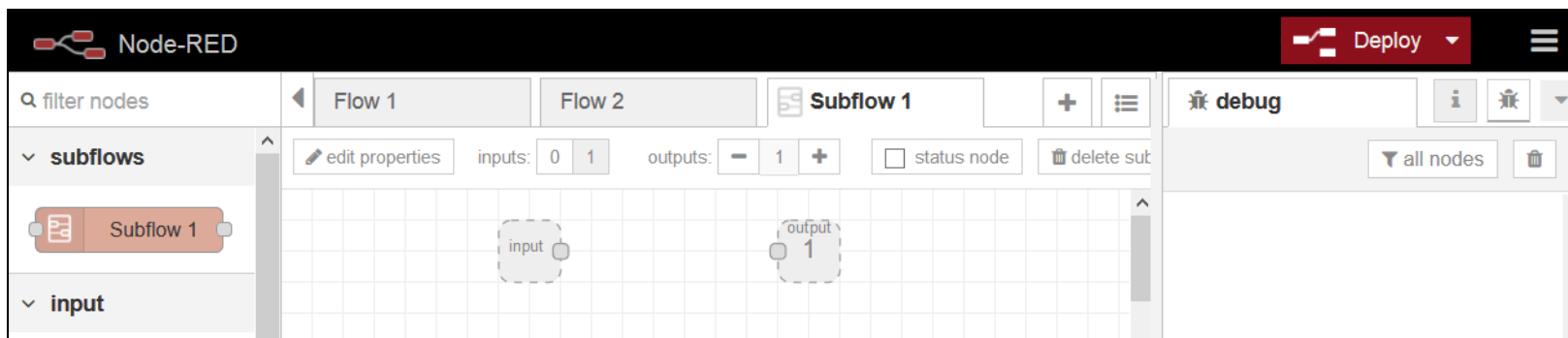
A Subflow is an aggregation of nodes and wires with input and output ports.

The creation of subflows useful to reuse flows and keep the overall flow manageable.

Creat a subflow: click on the main menu > **Subflows** > **Create Subflow**. Assign a name to the library



Subflows are available in the Palette and can be later added to another flow like a simple node.



A Node-RED flow works by passing messages between nodes.

The messages are simple **JavaScript objects** that can have any set of properties.

The message is passed in as an object called **msg**. Messages usually have a **payload property (msg.payload)**, i.e. the default property containing the body of the message.

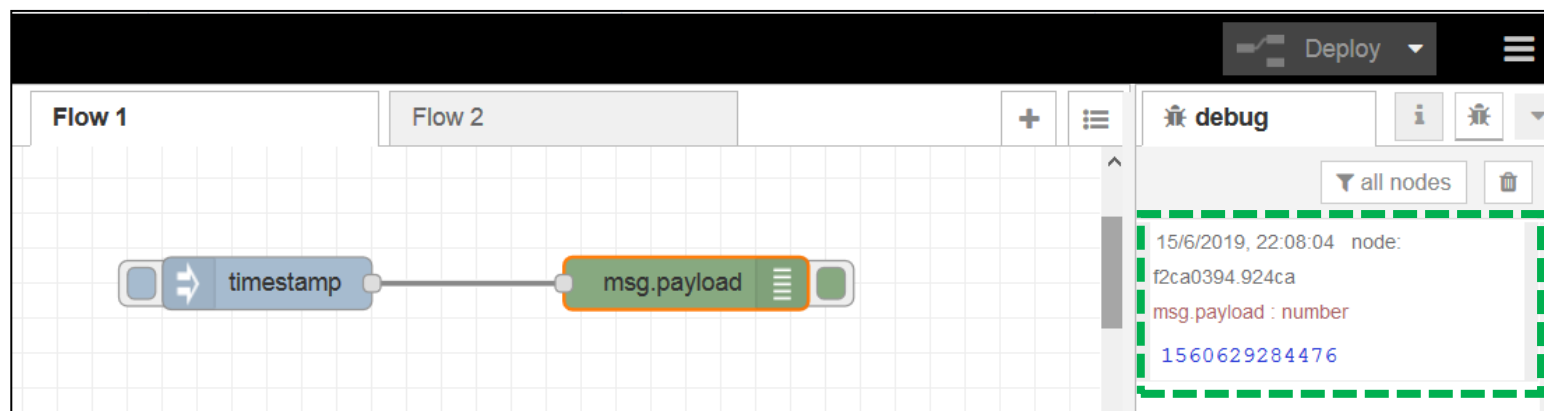
Node-RED also adds a property called **_msgid** to each message, i.e. an identifier of the message.

The value of a property can be typically any valid JavaScript type, such as:

- Boolean - true, false
- Number - e.g. 0, 123.4
- String - "hello"
- Array - [1,2,3,4]
- Object - { "a": 1, "b": 2 }
- Null

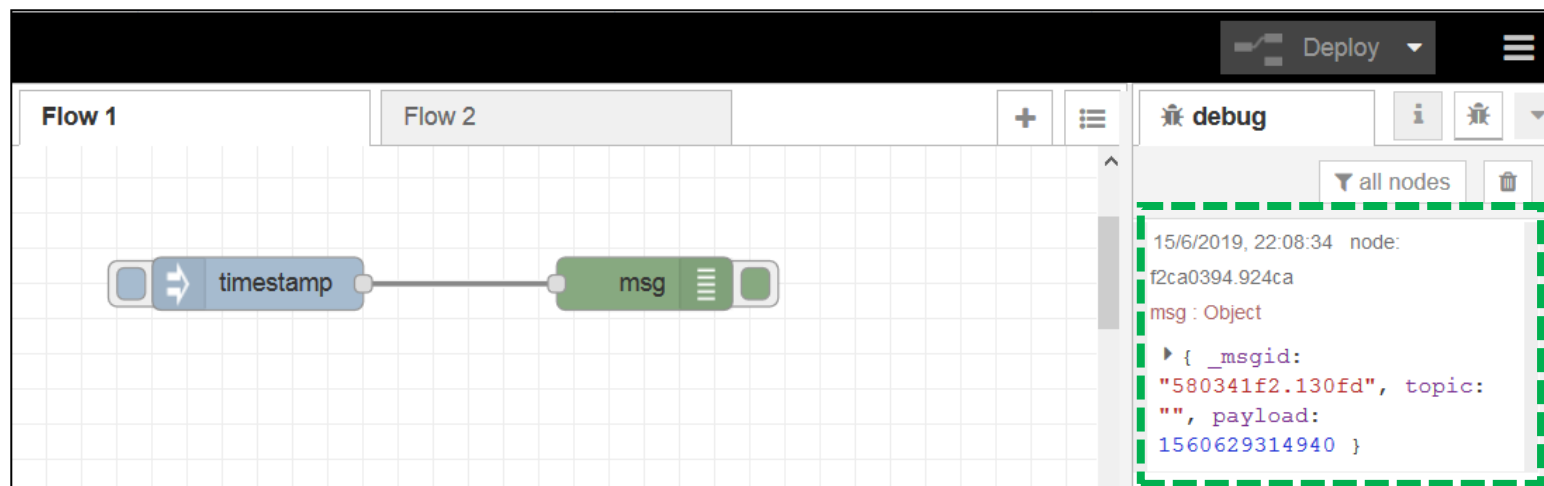
The structure of a message can be better understood if it is passed to a Debug node. Thus, the contents of the message can be viewed in the Debug sidebar.

By default, the Debug node will display the **msg.payload** property, but it can be configured to display any property or the whole message.



The screenshot shows a Node-RED flow with two nodes: 'timestamp' and 'msg.payload'. The 'msg.payload' node is highlighted with an orange border. The debug sidebar on the right shows the following message content:

```
15/6/2019, 22:08:04 node:  
f2ca0394.924ca  
msg.payload : number  
1560629284476
```



The screenshot shows a Node-RED flow with two nodes: 'timestamp' and 'msg'. The 'msg' node is highlighted with a green border. The debug sidebar on the right shows the following message content:

```
15/6/2019, 22:08:34 node:  
f2ca0394.924ca  
msg : Object  
  {  
    _msgid:  
      "580341f2.130fd", topic:  
      "", payload:  
      1560629314940 }  
}
```

When you hover over any element in Debug, a set of buttons appear on the right:



copies the path to the selected element to your clipboard.



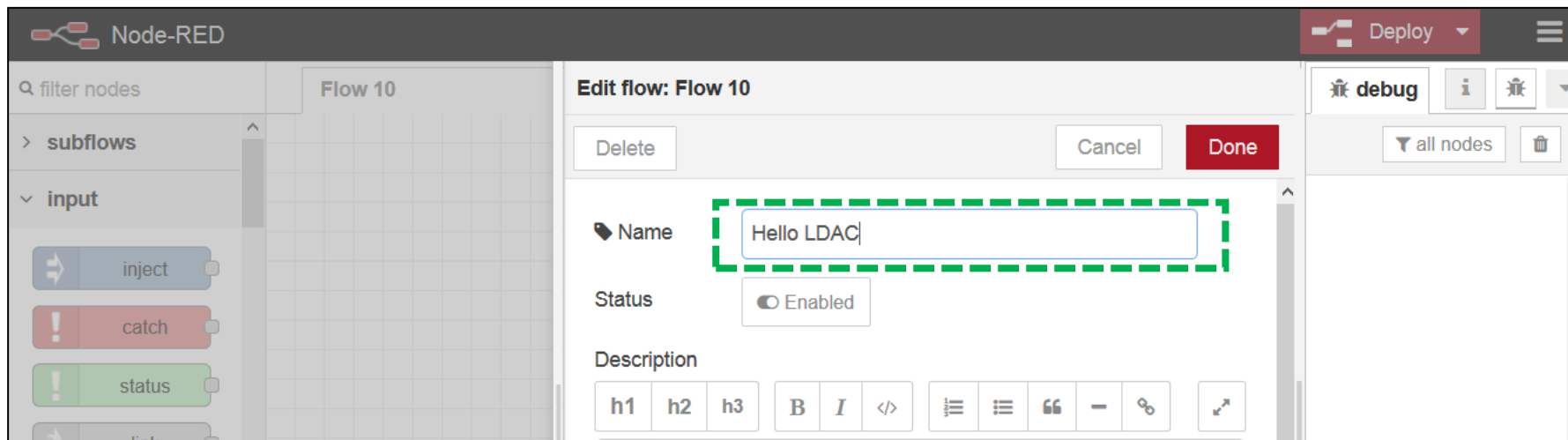
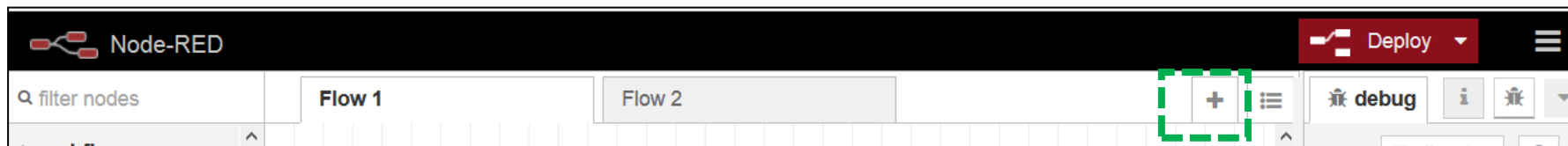
copies the value of the element to your clipboard as a JSON string.



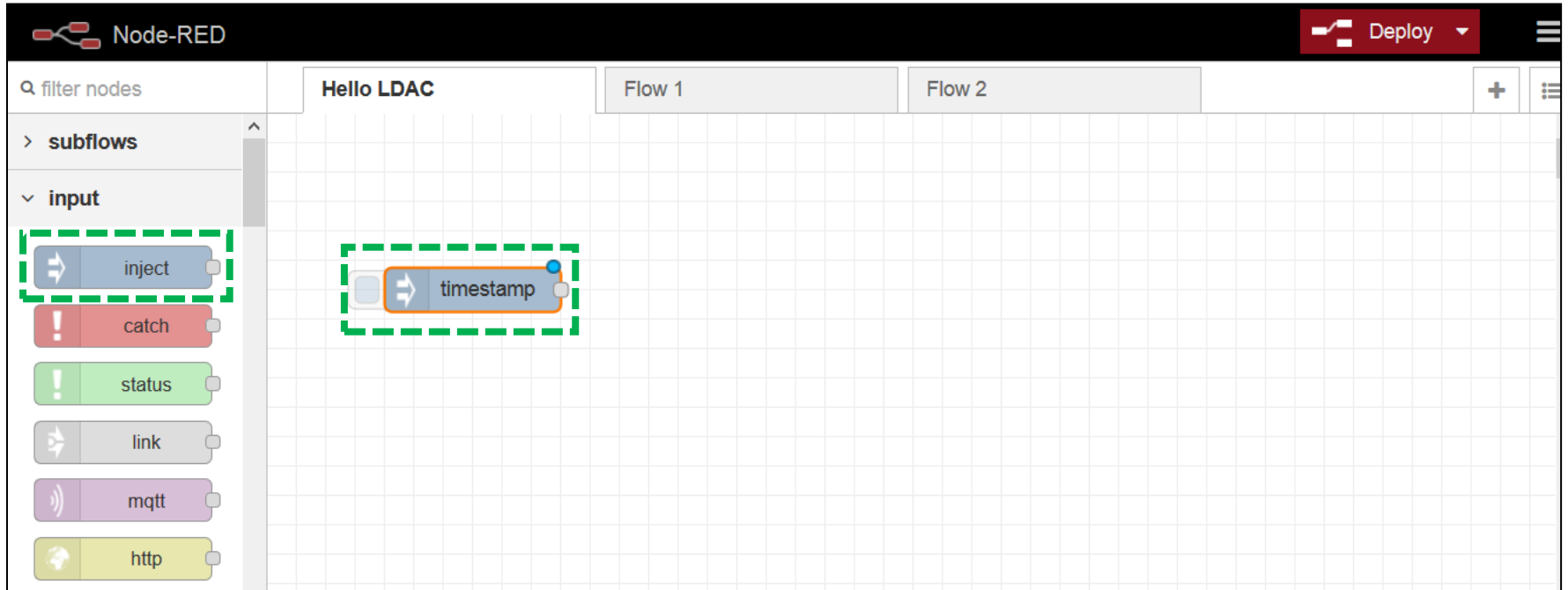
pins the selected element so it is always displayed. When another message is received from the same Debug node, it is automatically expanded to show all pinned elements.

Create a simple routine that prints the message 'Hello LDAC Summer School!'
(Note: several alternative options are possible)

1. Create a new Flow and rename it



2. Add Inject node (from input nodes)



The screenshot shows the Node-RED web interface. The top bar includes the Node-RED logo, a search bar for nodes, and a 'Deploy' button. The main workspace is titled 'Hello LDAC' and contains two flow tabs: 'Flow 1' and 'Flow 2'. On the left sidebar, the 'input' category is expanded, and the 'inject' node is highlighted with a dashed green border. In the main workspace, a 'timestamp' node is also highlighted with a dashed green border, indicating it is being added to the flow.

3. Add Change node (from function nodes)

The screenshot shows the Node-RED web interface. On the left, the 'function' category in the node palette is expanded, and the 'change' node is highlighted with a green dashed box. In the main workspace, a flow named 'Hello LDAC' contains a 'timestamp' node followed by a 'set msg.payload' node, which is also highlighted with a green dashed box. On the right, the 'Edit change node' dialog is open. The 'Name' field contains 'Set message payload'. Under the 'Rules' section, a rule is configured with the action 'Set' applied to the 'msg. payload' property, with the value 'Hello LDAC Summer School!'.

4. Add Debug node (from output nodes)

The screenshot shows the Node-RED interface with the following components:

- Left Sidebar:** A list of nodes under the 'output' category. The 'debug' node is highlighted with a green dashed box.
- Main Workspace:** A flow diagram titled 'Hello LDAC' containing a 'timestamp' node, a 'Set message payload' node, and a 'msg.payload' node. The 'msg.payload' node is highlighted with a green dashed box.
- Right Panel:** The 'Edit debug node' dialog is open. It features a 'Delete' button, 'Cancel' and 'Done' buttons, and a 'Properties' section. The 'Properties' section is highlighted with a green dashed box and contains:
 - Output:** A dropdown menu set to 'msg.payload'.
 - To:** Checkboxes for 'debug window' (checked) and 'system console' (checked). There is also an unchecked checkbox for 'node status (32 characters)'.
 - Name:** A text input field containing 'Debug msg.payload'.

5. Connect wires and deploy

The screenshot shows the Node-RED web interface. At the top, a notification box says "Successfully deployed". The main workspace contains a flow with three nodes: a "timestamp" node, a "Set message payload" node, and a "Debug msg.payload" node. Wires connect the "timestamp" node to the "Set message payload" node, and the "Set message payload" node to the "Debug msg.payload" node. The "timestamp" node and the "Set message payload" node are circled with green dashed lines. In the top right corner, the "Deploy" button is highlighted with a green dashed box. Below it, the "debug" console is also highlighted with a green dashed box. The left sidebar shows the "output" category with nodes like "debug", "link", "mqtt", "http response", "websocket", "tcp", and "udp". The "function" category is also visible.

6. Execute: click on Inject button and check results in Debug pane

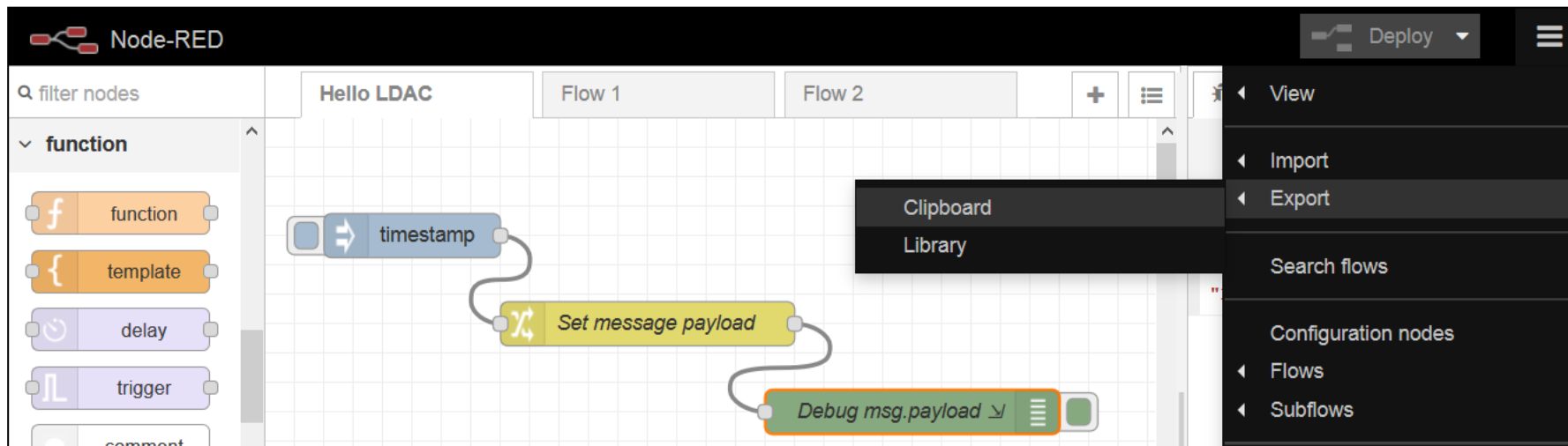
Start flow

The screenshot shows the Node-RED interface with a flow titled "Hello LDAC". The flow consists of three nodes: a "timestamp" node, a "Set message payload" node, and a "Debug msg.payload" node. A green dashed box highlights the "timestamp" node, and a green arrow points to it with the text "Start flow". A notification banner at the top says "Successfully injected: timestamp". On the right, the "debug" pane is open, showing the following output:

```

16/6/2019, 13:43:53 node: Debug
msg.payload
msg.payload : string[25]
"Hello LDAC Summer School!"
    
```

7. Export flow



```
[{"id":"148e3965.53767f","type":"debug","z":"283e160d.c1304a","name":"Debug msg.payload","active":true,"tosidebar":true,"console":true,"tostatus":false,"complete":"payload","targetType":"msg","x":440,"y":200,"wires":[]},{id:"478a9031.b808","type":"inject","z":"283e160d.c1304a","name":"","topic":"","payload":"","payloadType":"date","repeat":"","crontab":"","once":false,"onceDelay":0.1,"x":100,"y":80,"wires":[["112be1b8.535ec6"]]},{"id":"112be1b8.535ec6","type":"change","z":"283e160d.c1304a","name":"Set message payload","rules":[{"t":"set","p":"payload","pt":"msg","to":"Hello LDAC Summer School!","tot":"str"}],"action":"","property":"","from":"","to":"","reg":false,"x":260,"y":140,"wires":[["148e3965.53767f"]]}]
```

Alternative flow (2): inject part of the message

The screenshot shows the Node-RED web interface. On the left, a palette of nodes includes 'batch', 'csv', 'html', 'json', 'xml', and 'yaml'. The main workspace contains a flow with three nodes: 'Inject "Hello"', 'Add "LDAC Summer School"', and 'Debug msg.payload'. The 'Inject' node is connected to the 'Change' node, which is connected to the 'Debug' node. On the right, a 'debug' sidebar shows a log entry: '16/6/2019, 13:54:11 node: Debug msg.payload "Hello LDAC Summer School"'. The top of the interface shows 'Hello LDAC (2)', 'Flow 1', and 'Flow 2' tabs.

```
{
  "id": "4855565d.e62278",
  "type": "debug",
  "z": "85679f4e.4e529",
  "name": "Debug msg.payload",
  "active": true,
  "tosidebar": true,
  "console": true,
  "tostatus": false,
  "complete": "payload",
  "targetType": "msg",
  "x": 440,
  "y": 200,
  "wires": [],
  "id": "67c10546.ab0594",
  "type": "inject",
  "z": "85679f4e.4e529",
  "name": "Inject \"Hello \"",
  "topic": "",
  "payload": "Hello",
  "payloadType": "str",
  "repeat": "",
  "crontab": "",
  "once": false,
  "onceDelay": 0.1,
  "x": 110,
  "y": 80,
  "wires": [
    [
      "cea55947.537d68"
    ]
  ],
  "id": "cea55947.537d68",
  "type": "change",
  "z": "85679f4e.4e529",
  "name": "Add \"LDAC Summer School\"",
  "rules": [
    [
      {
        "t": "set",
        "p": "payload",
        "pt": "msg",
        "to": "payload & \" LDAC Summer School\"",
        "tot": "jsonata"
      }
    ]
  ],
  "action": "",
  "property": "",
  "from": "",
  "to": "",
  "reg": false,
  "x": 240,
  "y": 140,
  "wires": [
    [
      "4855565d.e62278"
    ]
  ]
}
```

Alternative flow (3): use function node

The screenshot shows the Node-RED interface with a flow named "Hello LDAC (3)". The flow consists of three nodes connected in sequence: an "Inject 'Hello'" node, a "function" node with the text "Add 'LDAC Summer School!'", and a "Debug msg.payload" node. The "function" node is highlighted with a green dashed box in the original image. The left sidebar shows the "function" category selected, with the "function" node icon also highlighted. The right sidebar shows the "debug" console with the following output:

```

16/6/2019, 13:59:17 node: Debug
msg.payload
msg.payload : string[26]
"Hello LDAC Summer School!"
    
```

```

[{"id":"1f21a9f.25f0f56","type":"debug","z":"810f777b.83fa2","name":"Debug
msg.payload","active":true,"tosidebar":true,"console":true,"tostatus":false,"complete":"payload","targetType":"msg","x":440,"y":180
,"wires":[]},{id":"cc423328.2163a","type":"inject","z":"810f777b.83fa2","name":"Inject \"Hello\" ","topic":"","payload":"Hello
","payloadType":"str","repeat":"","crontab":"","once":false,"onceDelay":0.1,"x":110,"y":60,"wires":[["1ff89f23.5a8e11"]]},{"id":"1ff89
f23.5a8e11","type":"function","z":"810f777b.83fa2","name":"Add \"LDAC Summer School!\"","func":"msg.payload += \" LDAC
Summer School!\"\\nreturn msg;","outputs":1,"noerr":0,"x":260,"y":120,"wires":[["1f21a9f.25f0f56"]}]}
    
```

input

- inject
- catch
- status
- link
- mqtt
- http
- websocket
- tcp
- udp

output

- debug
- link
- mqtt
- http response
- websocket
- tcp
- udp

function

- function
- template
- delay
- trigger
- comment
- http request
- tcp request
- switch
- change
- range
- split

function

- join
- sort
- batch
- csv
- html
- json
- xml
- yaml
- rbe
- http auth

social

- email
- twitter
- email
- twitter

storage

- file
- tail
- file

analysis

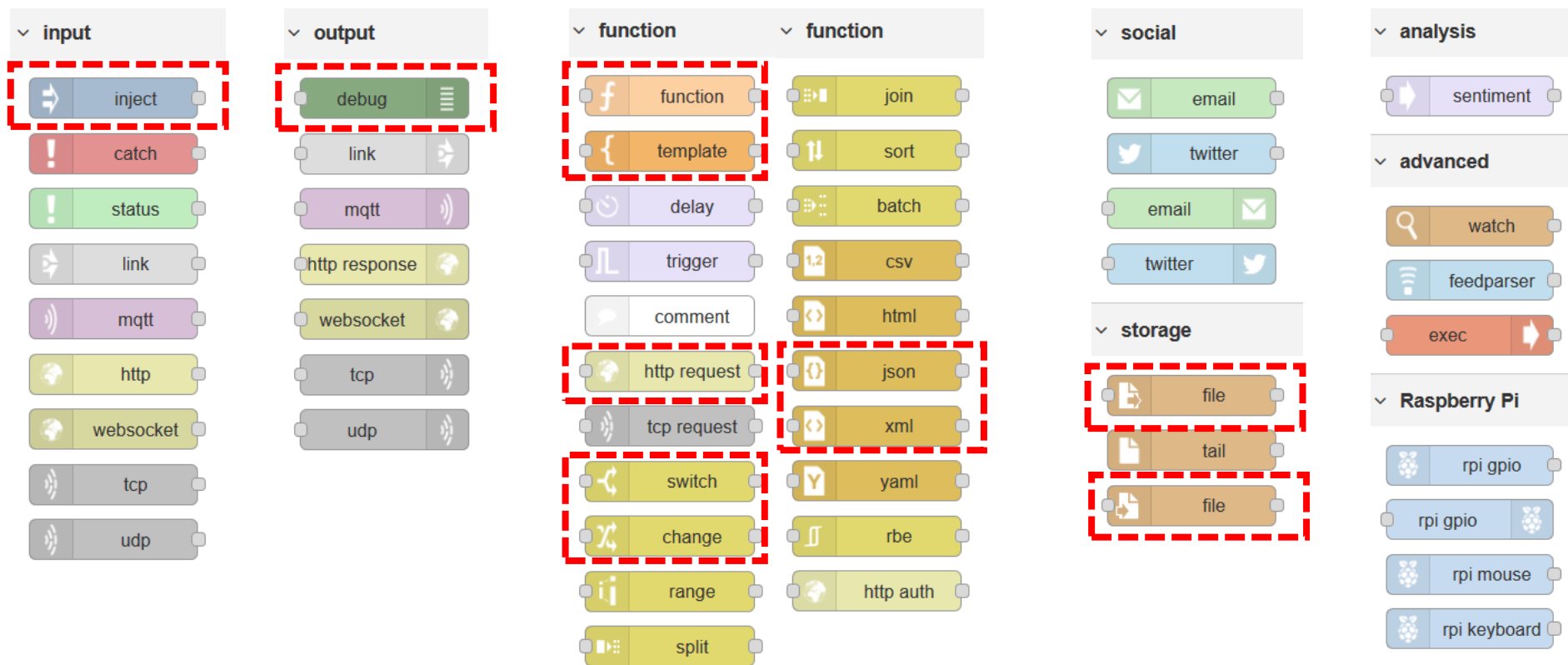
- sentiment

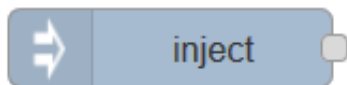
advanced

- watch
- feedparser
- exec

Raspberry Pi

- rpi gpio
- rpi gpio
- rpi mouse
- rpi keyboard

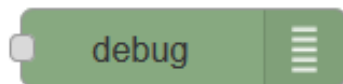




Injects a message into a flow either manually or at regular intervals. The message payload can be a variety of types, including strings, JavaScript objects or the current time.

The Inject node can initiate a flow with a specific payload value. The node also supports injecting timestamp , strings, numbers, booleans, JavaScript objects, or flow/global context values.

By default, the node is triggered manually by clicking on its button within the editor. It can also be set to inject at regular intervals or according to a schedule.




Displays selected message properties in the debug sidebar tab and optionally the runtime log. By default it displays **msg.payload**, but can be configured to display any property, the full message or the result of a **JSONata expression**.


The debug sidebar provides a structured view of the messages it is sent, making it easier to understand their structure.

JavaScript objects and arrays can be collapsed and expanded as required. Alongside each message, the debug sidebar includes information about the time the message was received, the node that sent it and the type of the message.

The button on the node can be used to enable or disable its output.

 function

A JavaScript function block to run against the messages being received by the node. The messages are passed in as a JavaScript object called **msg**. By convention it will have a **msg.payload** property containing the body of the message. The function is expected to return a message object (or multiple message objects), but can choose to return nothing in order to halt a flow.

 template

The Template node can be used to generate text using a message's properties to fill out a template. It uses the Mustache templating language to generate the result.


For example, a template of:

```
This is the payload: {{payload}} !
```

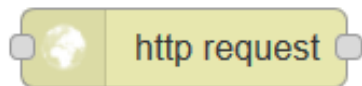
Will replace `{{payload}}` with the value of the message's payload property.

 switch

The Switch node allows messages to be routed to different branches of a flow by evaluating a set of rules against each message. The node is configured with the property to test - which can be either a message property or a context property.

 change

The Change node can be used to modify a message's properties and set context properties without having to resort to a Function node. Each node can be configured with multiple operations that are applied in order. The available operations are: Set, Change, Move, Delete

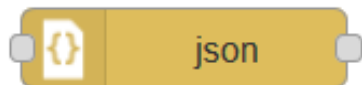


Sends HTTP requests and returns the response.

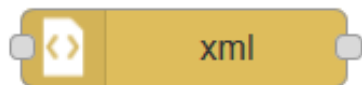
Key Inputs:

- url (string): url of the request.
- method (string): HTTP method of the request, i.e. one of GET, PUT, POST, PATCH or DELETE.
- headers (object): HTTP headers of the request.
- payload: body of the request

in output, payload contains the body of the response. The node can be configured to return the body as a string, attempt to parse it as a JSON string or leave it as a binary buffer.



Converts between a JSON string and its JavaScript object representation, in either direction.



Converts between an XML string and its JavaScript object representation, in either direction.



Reads the contents of a file as either a string or binary buffer.



Writes msg.payload to a file, either adding to the end or replacing the existing content. Alternatively, it can delete the file.

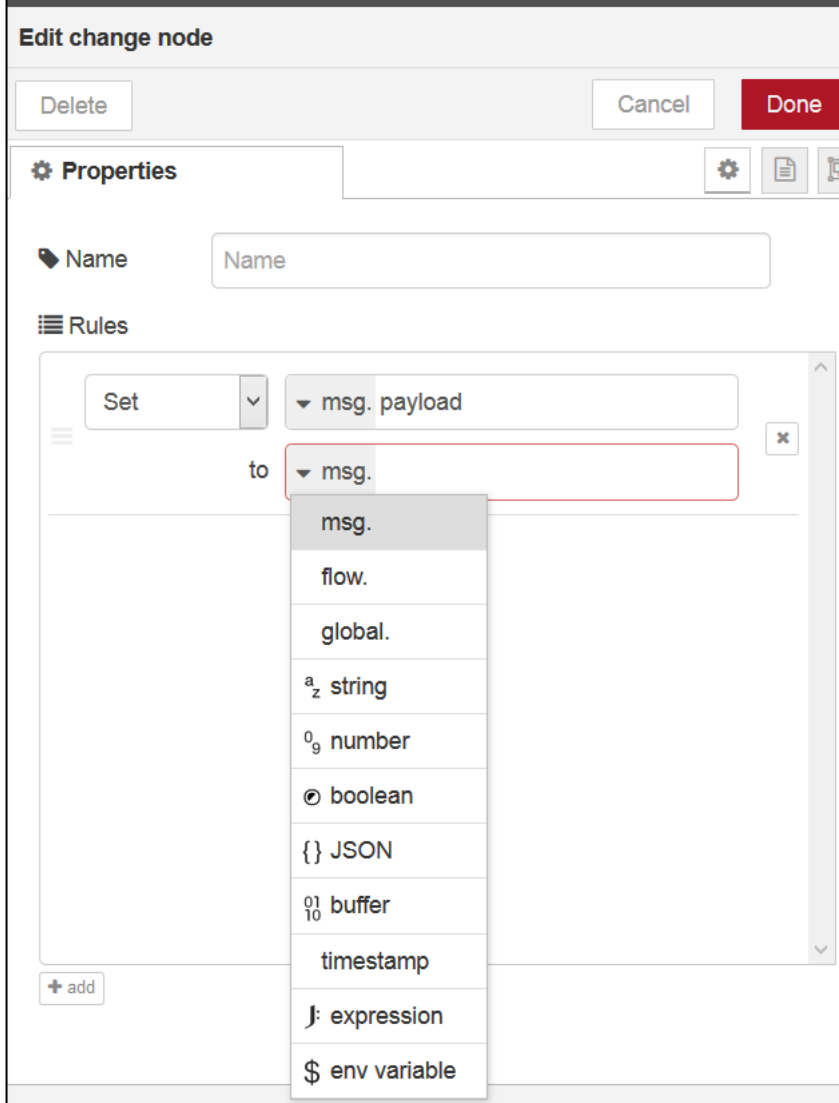
There are two main nodes for modifying a message, the Change node and the Function node.

The **Change node** provides a lot of functionality without needing to write JavaScript code. Not only can it modify message properties, but it can also access flow- and global-context.

The Change node provides four basic operations:

- **Set** a property to a value,
- **Change** a String property by performing a search and replace,
- **Delete** a property,
- **Move** a property.

More than one change can be assigned to one node. The resulting value can be hardcoded (e.g. string, number) or defined by elaborating another message.



JSONata expressions can be exploited to make complex elaborations based on input messages.

Edit change node

Delete Cancel Done

Properties

Name Name

Rules

Set msg. payload

to j: payload & " abc"

Edit change node

Delete Cancel Done

Properties

Name Name

Rules

Set msg. payload

to j: payload + 10

More information about JSONata can be found at <http://jsonata.org/>

JSONata

JSON query and transformation language

J	Address.City
	• FirstName & ' ' & Surname
	• Phone[type = 'mobile'].number
	\$sum(Order.Product.(Price * Quantity))

Go play in the [JSONata Exerciser](#)

The **Function node** allows you to run any **JavaScript** code against the message. This gives you complete flexibility in what you do with the message, but does require familiarity with JavaScript and is unnecessary for many simple cases.

In particular, you can use function node when there is no existing node dedicated to your task at hand.

```
var inputpayload = msg.payload; // get the message payload

msg.payload = newpayload; // modify the message payload

var newproperty;
msg.newproperty= newvalue; // the contents of the message can be enriched with properties

var msg1 = { payload: msg.payload, topic:msg.topic}; // create a new message object
return msg1;
```

A function must return a message, otherwise the flow stops.

The **Function node** allows you to run any **JavaScript** code against the message. This gives you complete flexibility in what you do with the message, but does require familiarity with JavaScript and is unnecessary for many simple cases.

In particular, you can use function node when there is no existing node dedicated to your task at hand.

```
var inputpayload = msg.payload; // get the message payload

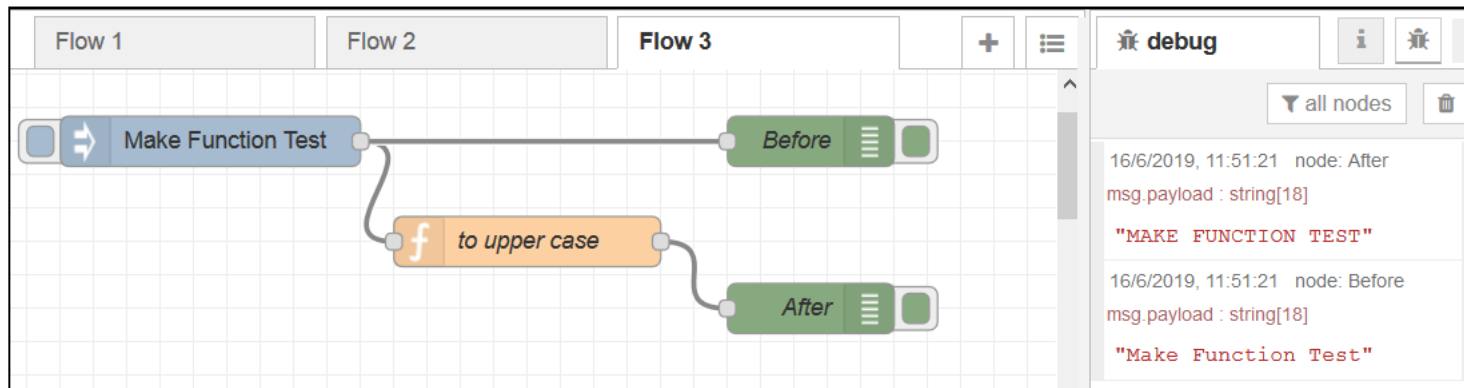
msg.payload = newpayload; // modify the message payload

var newproperty;
msg.newproperty= newvalue; // the contents of the message can be enriched with properties

var msg1 = { payload: msg.payload, topic:msg.topic}; // create a new message object
return msg1;
```

A function must return a message, otherwise the flow stops.

Example



Function body:

```

msg.payload = msg.payload.toUpperCase ()
return msg;

```

```

[{"id":"c6a5d62f.46d2f","type":"tab","label":"Flow
3","disabled":false,"info":""},{id":"ec2f106.b8260f","type":"function","z":"c6a5d62f.46d2f","name":"to upper case","func":"msg.payload =
msg.payload.toUpperCase()\nreturn
msg;","outputs":1,"noerr":0,"x":320,"y":180,"wires":[["c011ed5d.75eda"]]}, {"id":"cdefe980.e69b4","type":"inject","z":"c6a5d62f.46d2f","name":"","top
ic":"","payload":"Make Function
Test","payloadType":"str","repeat":"","crontab":"","once":false,"onceDelay":0.1,"x":130,"y":120,"wires":[["ec2f106.b8260f","91e5aad8.61ea78"]]}, {"id"
:"c011ed5d.75eda","type":"debug","z":"c6a5d62f.46d2f","name":"After","active":true,"tosidebar":true,"console":false,"tostatus":false,"complete":"payl
oad","targetType":"msg","x":490,"y":220,"wires":[]}, {"id":"91e5aad8.61ea78","type":"debug","z":"c6a5d62f.46d2f","name":"Before","active":true,"tosi
debar":true,"console":false,"tostatus":false,"complete":"payload","targetType":"msg","x":490,"y":120,"wires":[]}

```

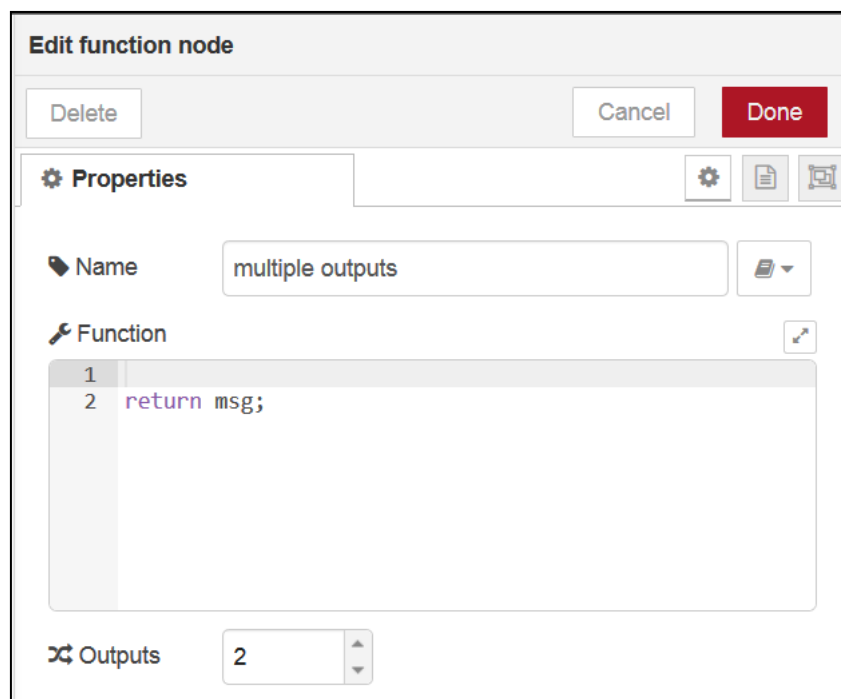
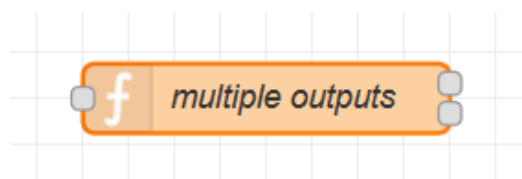
The function node can be configured with multiple outputs. This is useful when the flow splits into separate paths depending on a message property. To configure multiple outputs open the function node and use the up/down arrows to adjust the number of outputs.

To return messages to multiple outputs you need to return an array, e.g.

```
return [msg1, msg2];
```

msg1 will appear on output1 and msg2 on output2. To stop a flow you return null, e.g.

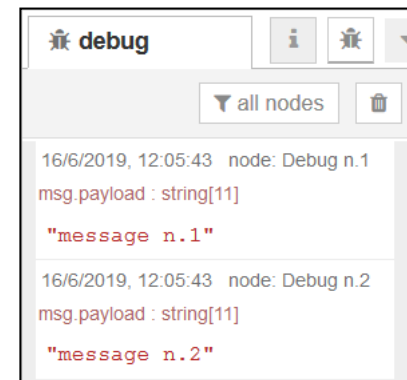
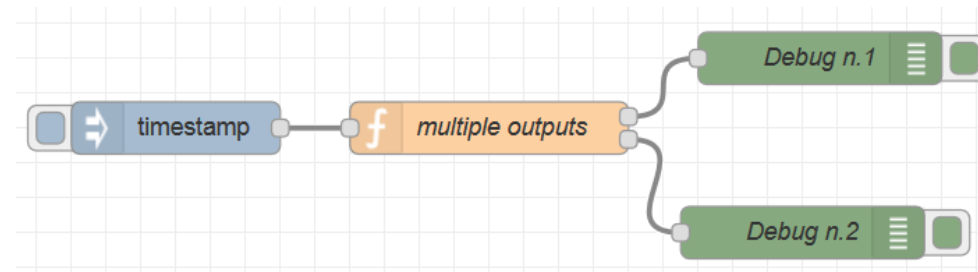
```
return [msg1, null];
```



Multiple outputs example

Function body:

```
var msg1 = {payload: "", topic: ""}
msg1.payload = "message n.1 "
var msg2 = {payload: "", topic: ""}
msg2.payload = "message n.2"
return [msg1, msg2];
```



```
{
  "id": "439af97a.44ef38",
  "type": "tab",
  "label": "Flow 4",
  "disabled": false,
  "info": "",
  "z": "439af97a.44ef38",
  "name": "multiple outputs",
  "func": "var msg1 = {payload: \"\", topic: \"\"}\nmsg1.payload = \"message n.1 \"\nvar msg2 = {payload: \"\", topic: \"\"}\nmsg2.payload = \"message n.2 \"\nreturn [msg1, msg2];",
  "outputs": 2,
  "noerr": 0,
  "x": 280,
  "y": 80,
  "wires": [
    [
      "31f9f142.ac463e",
      "1af074e3.f0c223"
    ]
  ],
  "id": "8dc40387.f59c88",
  "type": "inject",
  "z": "439af97a.44ef38",
  "name": "",
  "topic": "",
  "payload": "",
  "payloadType": "date",
  "repeat": "",
  "crontab": "",
  "once": false,
  "onceDelay": 0.1,
  "x": 100,
  "y": 80,
  "wires": [
    [
      "dc28df5c.459198"
    ]
  ],
  "id": "31f9f142.ac463e",
  "type": "debug",
  "z": "439af97a.44ef38",
  "name": "Debug n.1",
  "active": true,
  "tosidebar": true,
  "console": false,
  "tostatus": false,
  "complete": "payload",
  "targetType": "msg",
  "x": 470,
  "y": 40,
  "wires": [],
  "id": "1af074e3.f0c223",
  "type": "debug",
  "z": "439af97a.44ef38",
  "name": "Debug n.2",
  "active": true,
  "tosidebar": true,
  "console": false,
  "tostatus": false,
  "complete": "payload",
  "targetType": "msg",
  "x": 460,
  "y": 140,
  "wires": []
}
```


Multiple outputs in array

Function body:

```
var msg1 = {payload: "", topic: ""}
msg1.payload = "message n.1"
var msg2 = {payload: "", topic: ""}
msg2.payload = "message n.2"
var allmsg = []
allmsg.push(msg1)
allmsg.push(msg2)
return [allmsg];
```

```
[{"id":"439af97a.44ef38","type":"tab","label":"Flow
4","disabled":false,"info":""},{"id":"dc28df5c.459198","type":"function","z":"439af97a.44ef38","name":"multiple outputs in
array","func":"var msg1 = {payload: \"\", topic: \"\"}\nmsg1.payload = \"message n.1\"\nvar msg2 = {payload: \"\", topic:
\"\"}\nmsg2.payload = \"message n.2\"\nvar allmsg = []\nallmsg.push(msg1)\nallmsg.push(msg2)\nreturn
[allmsg];","outputs":1,"noerr":0,"x":290,"y":120,"wires":[["31f9f142.ac463e"]]}, {"id":"8dc40387.f59c88","type":"inject","z":"439af97a.44ef
38","name":"","topic":"","payload":"","payloadType":"date","repeat":"","crontab":"","once":false,"onceDelay":0.1,"x":100,"y":80,"wires":[[
"dc28df5c.459198"]]}, {"id":"31f9f142.ac463e","type":"debug","z":"439af97a.44ef38","name":"Debug","active":true,"tosidebar":true,"cons
ole":false,"tostatus":false,"complete":true,"targetType":"full","x":490,"y":160,"wires":[]}]
```

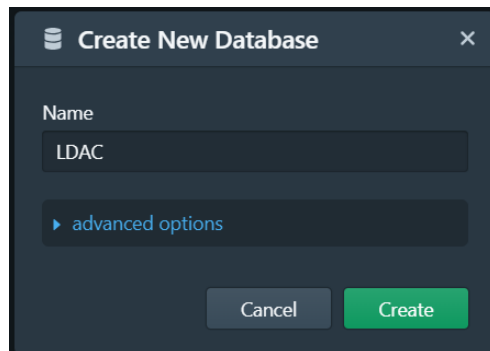
Node-RED can be employed to create flows involving the query and update of triple stores (RDF stores) while integrating other functionalities.

Herein we show how basic nodes can be used to make SPARQL queries/updates while accessing an RDF Store.

SETUP

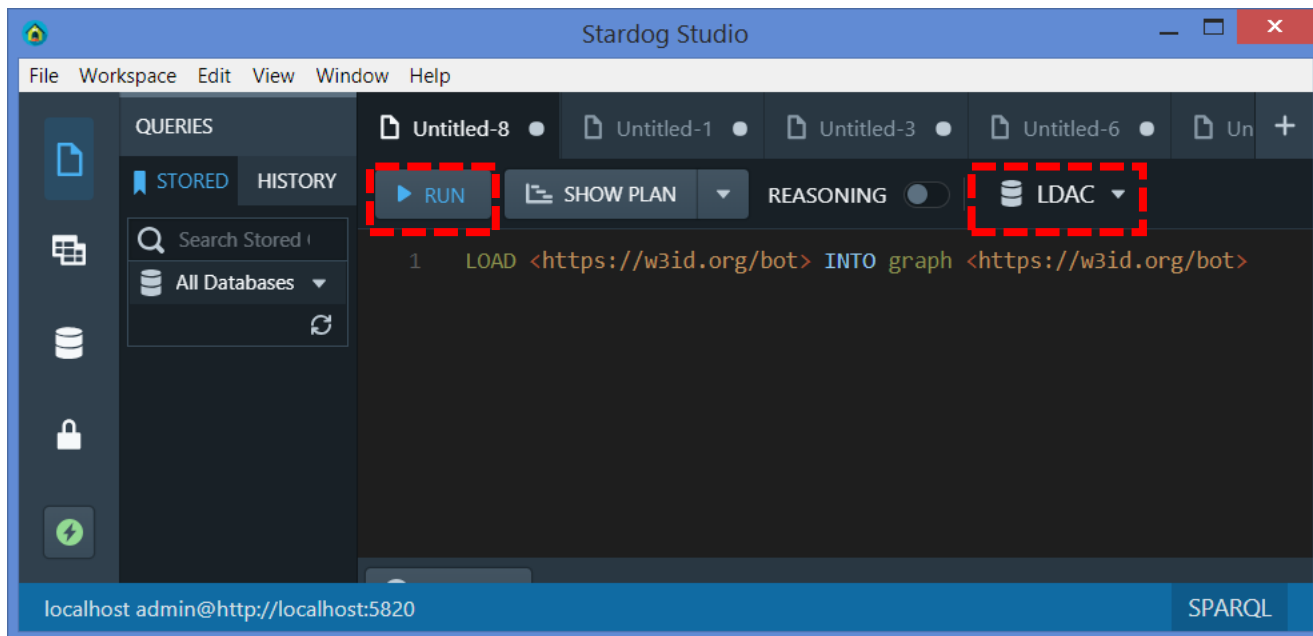
- Stardog is employed as triple store
- The examples refers to a Stardog installation available in localhost
- The examples are based on the BOT ontology (<https://github.com/w3c-lbd-cg/bot>) that must loaded on Stardog database "LDAC"

Create LDAC database in Stardog Studio



Load BOT ontology by running this query in Stardog Studio:

```
LOAD <https://w3id.org/bot> INTO graph <https://w3id.org/bot>
```



Setting to execute a SPARQL Query (in Stardog) via HTTP request

URL (server, DB, request)

<http://localhost:5820/LDAC/query>

Headers

`content-type = application/x-www-form-urlencoded`

`Accept = application/sparql-results+json`

Get all classes defined
in BOT ontology

```
select distinct ?botclass
FROM <https://w3id.org/bot>
where {
    ?botclass rdf:type owl:Class .
}
```

The screenshot displays the Node-RED interface with a workflow for executing a SPARQL query. The workflow consists of the following nodes:

- make request** (blue node)
- Set URL & Headers** (yellow node)
- SPARQL query** (orange node)
- finalize query** (yellow node)
- http request** (yellow node)
- Bindings** (green node)
- URIs** (green node)

The workflow is connected as follows: **make request** connects to **Set URL & Headers**, which connects to **SPARQL query**, which connects to **finalize query**. **finalize query** connects to **http request**, which then branches to **Bindings** and **URIs**.

The **debug** console on the right shows the following output:

```
16/6/2019, 23:20:47 node: Bindings
msg.payload : Object
  ▶ { head: object, results:
    object }

16/6/2019, 23:20:47 node: URIs
msg : array[9]
  ▶ [ "http://purl.org
    /voccommons/voaf...",
    "http://xmlns.com/foaf/0.1
    /Pers...", "https://w3id.org
    /bot#Zone", "https://w3id.org
    /bot#Element",
    "https://w3id.org
    /bot#Interface",
    "https://w3id.org/bot#Site",
    "https://w3id.org
    /bot#Building",
    "https://w3id.org
    /bot#Storey",
    "https://w3id.org/bot#Space"
  ]
```

Default Inject node

The screenshot displays the Node-RED web interface. On the left, a workflow is visible on a grid background. It consists of several nodes: a 'make request' node (highlighted with a red dashed box), a 'Set URL & Headers' node, a 'SPARQL query' node, and an 'http request' node. The 'make request' node is connected to the 'Set URL & Headers' node, which is connected to the 'SPARQL query' node. The 'http request' node is also connected to the 'SPARQL query' node.

On the right, the 'Edit inject node' configuration panel is open. It features a 'Delete' button, 'Cancel' and 'Done' buttons, and a 'Properties' section with a gear icon. The configuration includes:

- Payload:** A dropdown menu set to 'timestamp'.
- Topic:** An empty text input field.
- Inject once after:** A checkbox that is unchecked, followed by a text input field containing '0.1' and the text 'seconds, then'.
- Repeat:** A dropdown menu set to 'none'.
- Name:** A text input field containing 'make request'.

A yellow note box at the bottom of the configuration panel contains the following text:

Note: "interval between times" and "at a specific time" will use cron. "interval" should be less than 596 hours. See info box for details.

Set URL and Headers using **Change node**

The screenshot displays the Node-RED interface. On the left, a flow is visible with a 'make request' node connected to a 'Set URL & Headers' node, which is highlighted with a red dashed box. This node is connected to an 'http request' node. On the right, the 'Edit change node' configuration panel is open, also with a red dashed box around the 'Rules' section. The configuration includes:

- Name:** Set URL & Headers
- Rules:**
 - Set `msg. url` to `http://localhost:5820/LDAC/query`
 - Set `msg. headers['content-type']` to `application/x-www-form-urlencoded`
 - Set `msg. headers['Accept']` to `application/sparql-results+json`

Define SPARQL query as **payload** using **Template node**.

The screenshot shows the Node-RED interface. On the left, a workflow is visible with nodes: 'make request', 'Set URL & Headers', 'SPARQL query' (highlighted with a red dashed box), and 'http request'. On the right, the 'Edit template node' configuration panel is open. The 'Name' field is 'SPARQL query'. The 'Property' dropdown is set to 'msg.payload' (highlighted with a red dashed box). The 'Format' dropdown is 'Plain text'. The 'Template' text area contains the following SPARQL query (highlighted with a red dashed box):

```
1 select distinct ?botclass
2 FROM <https://w3id.org/bot>
3 where {
4   ?botclass rdf:type owl:Class .
5 }
```

The 'Output as' dropdown is also set to 'Plain text'.

Finalize the query by adding the text "query=" in front of it.

A Change node can be used together with a JSONata expression.

Other options: use a Function node; include the missing text in the previous node (Template)

The screenshot displays the Node-RED web interface. On the left, a workflow is visible on a grid background. It starts with an 'http request' node, which connects to two nodes: 'Bindings' and 'URIs'. The output of these nodes connects to a 'headers' node, which then connects to a 'SPARQL query' node. Finally, the 'SPARQL query' node connects to a 'finalize query' node, which is highlighted with a red dashed box. On the right, the 'Edit change node' configuration panel is open. It has a title bar with 'Delete', 'Cancel', and 'Done' buttons. Below the title bar is a 'Properties' section with a gear icon and a 'Name' field containing 'finalize query'. The 'Rules' section contains a single rule, also highlighted with a red dashed box. This rule is a 'Set' node with 'msg. payload' in the 'to' field and a JSONata expression 'J: "query=" & payload' in the 'value' field. At the bottom of the panel, there is a '+ add' button.

Make a HTTP request with the **http request node**. Method POST is selected. Use authentication (unless Stardog is started w/o security) and specify user and password (default in Stardog: admin, admin). Specify that a JSON object is returned as results. URL is already received as msg.url.

The screenshot displays the Node-RED web interface. On the left, a workflow is visible with nodes: 'URL & Headers', 'SPARQL query', 'finalize query', 'http request', 'Bindings', and 'URIs'. The 'http request' node is highlighted with a red dashed box. On the right, the 'Edit http request node' configuration panel is open. The 'Method' dropdown is set to 'POST' (highlighted with a red dashed box). The 'URL' field contains 'http://'. The 'Use authentication' checkbox is checked, and the 'Type' is set to 'basic authentication'. The 'Username' field contains 'admin' and the 'Password' field contains a masked password. The 'Return' dropdown is set to 'a parsed JSON object' (highlighted with a red dashed box). The 'Name' field contains 'Name'. Buttons for 'Delete', 'Cancel', and 'Done' are visible at the top of the configuration panel.

Show the full output in the **payload** using a **Debug** node.

The screenshot displays the Node-RED web interface. On the left, a workflow is visible on a grid background. It starts with an 'http request' node, which branches into two nodes: 'Bindings' and 'URIs'. A 'SPARQL query' node is connected to a 'finalize query' node. The 'Bindings' node is highlighted with a red dashed box. On the right, the 'Edit debug node' configuration panel is open. It features a 'Delete' button, 'Cancel', and 'Done' buttons. Under the 'Properties' section, the 'Output' field is set to 'msg. payload' and is highlighted with a red dashed box. The 'To' section has 'debug window' checked, while 'system console' and 'node status (32 characters)' are unchecked. The 'Name' field is set to 'Bindings'.

Extract only the URIs of the result using a JSONata in a **Debug node**.

The screenshot displays the Node-RED web interface. On the left, a workflow is visible in the 'Query' workspace. It starts with an 'http request' node, which branches into two paths. One path goes through a 'SPARQL query' node and a 'finalize query' node. The other path goes through a 'Bindings' node. Both paths then merge into a 'URIs' node, which is highlighted with a red dashed box. The right-hand side of the interface shows the 'Edit debug node' configuration panel. The 'Output' field is set to the JSONata expression `payload.results.bindings.botclass.value`, also highlighted with a red dashed box. The 'To' section has 'debug window' checked, while 'system console' and 'node status (32 characters)' are unchecked. The 'Name' field is set to 'URIs'.

Query results:

```
array[9]
0: "http://purl.org/vocommons/voaf#Vocabulary"
1: "http://xmlns.com/foaf/0.1/Person"
2: "https://w3id.org/bot#Zone"
3: "https://w3id.org/bot#Element"
4: "https://w3id.org/bot#Interface"
5: "https://w3id.org/bot#Site"
6: "https://w3id.org/bot#Building"
7: "https://w3id.org/bot#Storey"
8: "https://w3id.org/bot#Space"
```

Exported flow

```
[{"id":"383e0c8c.5a7d0c","type":"inject","z":"f0f1a3db.d85df","name":"make request","topic":"","payload":"","payloadType":"date","repeat":"","crontab":"","once":false,"onceDelay":"","x":170,"y":140,"wires":[["bb2f0eb2.b3e71"]]}, {"id":"6a30fe9c.c16a78","type":"template","z":"f0f1a3db.d85df","name":"SPARQL query","field":"payload","fieldType":"msg","format":"text","syntax":"plain","template":"select distinct ?botclass\n\nFROM <https://w3id.org/bot>\n\nwhere { \n  ?botclass rdf:type owl:Class .\n\n}","output":"str","x":440,"y":220,"wires":[["5390e2c7.0b9c2c"]]}, {"id":"a50bda43.26d36","type":"http request","z":"f0f1a3db.d85df","name":"","method":"POST","ret":"obj","paytoqs":false,"url":"","tls":"","proxy":"","authType":"basic","x":370,"y":340,"wires":[["bf6562c1.d99d58","54b7c078.dcf43"]]}, {"id":"bf6562c1.d99d58","type":"debug","z":"f0f1a3db.d85df","name":"Bindings","active":true,"tosidebar":true,"console":false,"tostatus":false,"complete":"payload","targetType":"msg","x":620,"y":320,"wires":[]}, {"id":"bb2f0eb2.b3e71","type":"change","z":"f0f1a3db.d85df","name":"Set URL & Headers","rules":[{"t":"set","p":"url","pt":"msg","to":"http://localhost:5820/LDAC/query","tot":"str"}, {"t":"set","p":"headers['content-type']","pt":"msg","to":"application/x-www-form-urlencoded","tot":"str"}, {"t":"set","p":"headers['Accept']","pt":"msg","to":"application/sparql-results+json","tot":"str"}],"action":"","property":"","from":"","to":"","reg":false,"x":230,"y":220,"wires":[["6a30fe9c.c16a78"]]}, {"id":"54b7c078.dcf43","type":"debug","z":"f0f1a3db.d85df","name":"URIs","active":true,"tosidebar":true,"console":false,"tostatus":false,"complete":"payload.results.bindings.botclass.value","targetType":"jsonata","x":610,"y":360,"wires":[]}, {"id":"5390e2c7.0b9c2c","type":"change","z":"f0f1a3db.d85df","name":"finalize query","rules":[{"t":"set","p":"payload","pt":"msg","to":"\n\nquery=\n\n & payload","tot":"jsonata"}],"action":"","property":"","from":"","to":"","reg":false,"x":630,"y":220,"wires":[["a50bda43.26d36"]}]}]
```

Setting to execute a SPARQL Update (in Stardog) via HTTP request

URL (server, DB, request)

<http://localhost:5820/LDAC/update>

Headers

`content-type = application/x-www-form-urlencoded`

Add an individual of class bot:Zone in a new graph that imports BOT ontology

```
PREFIX bot: https://w3id.org/bot#
PREFIX ex: https://www.example.com#
INSERT DATA {
  GRAPH <https://www.example.com> {
    <https://www.example.com> owl:imports <https://w3id.org/bot> .
    ex:NewZone rdf:type owl:NamedIndividual , bot:Zone.
  }
}
```

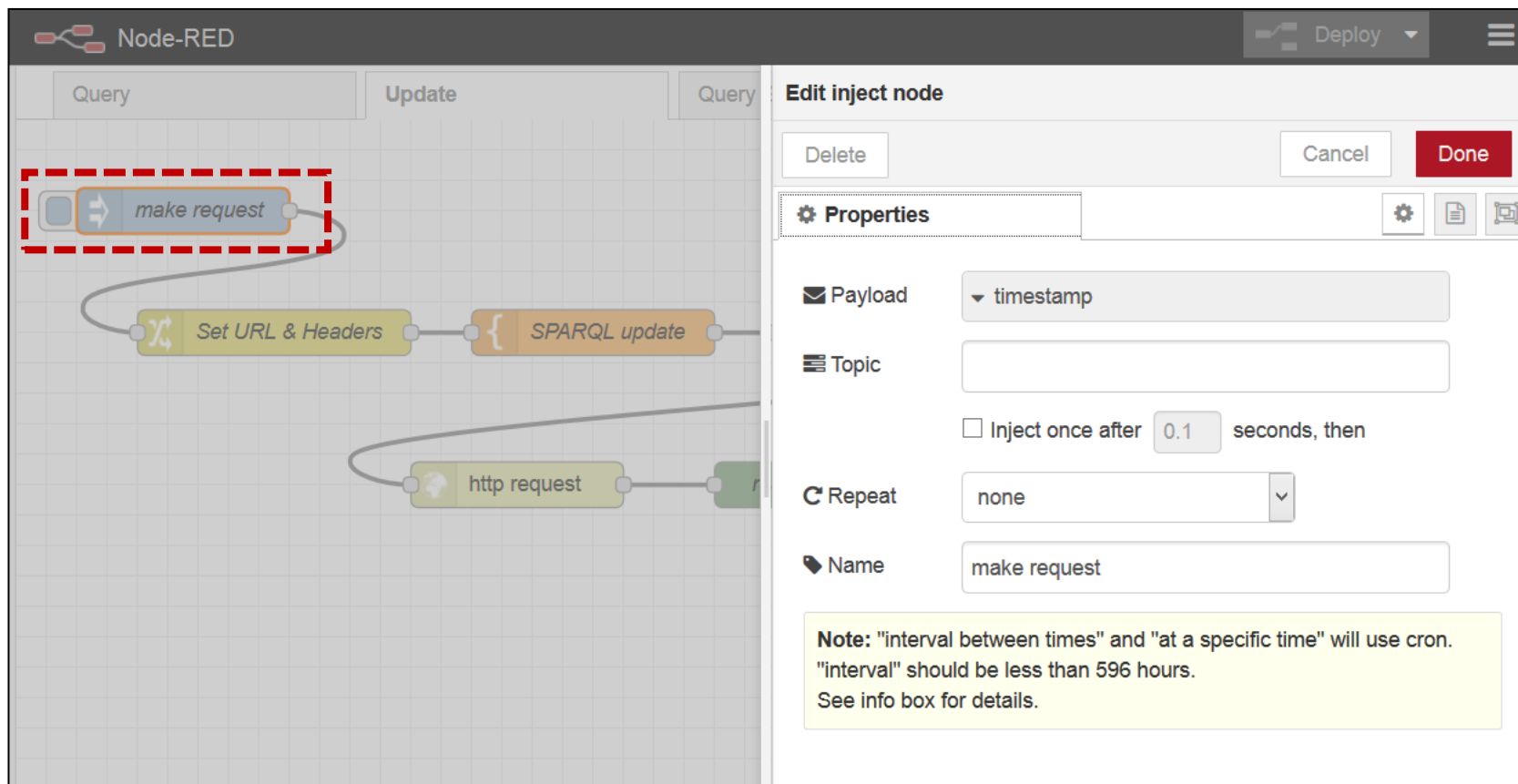
The screenshot shows the Node-RED interface with a workflow for sending a SPARQL update. The workflow consists of the following nodes:

- make request**: A blue node that initiates the process.
- Set URL & Headers**: A yellow node that configures the request.
- SPARQL update**: An orange node that contains the SPARQL update query.
- finalize query**: A yellow node that completes the query.
- http request**: A yellow node that sends the request.
- statusCode**: A green node that captures the response status code.

The **debug** console on the right shows the following output:

```
17/6/2019, 00:11:56 node: statusCode
msg.statusCode : number
200
```


Default Inject node



The screenshot shows the Node-RED interface with a flow in the 'Update' tab. The flow consists of the following nodes: 'make request' (highlighted with a red dashed box), 'Set URL & Headers', 'SPARQL update', and 'http request'. The 'Edit inject node' dialog is open on the right, showing the configuration for the 'make request' node.

Edit inject node

Delete Cancel Done

Properties

Payload timestamp

Topic

Inject once after 0.1 seconds, then

Repeat none

Name make request

Note: "interval between times" and "at a specific time" will use cron. "interval" should be less than 596 hours. See info box for details.

Set URL and Headers using **Change node**

The screenshot displays the Node-RED web interface. On the left, a workflow is visible in the 'Update' tab, featuring a 'make request' node connected to a 'Set URL & Headers' node (highlighted with a red dashed box), which is then connected to a 'SPARQL update' node. Below this, an 'http request' node is also shown. On the right, the 'Edit change node' configuration panel is open, showing the node's name as 'Set URL & Headers'. The 'Rules' section, also highlighted with a red dashed box, contains two configuration entries:

- Set `msg. url` to `http://localhost:5820/LDAC/update`
- Set `msg. headers['content-type']` to `application/x-www-form-urlencoded`

Define SPARQL update as **payload** using **Template node**.

The screenshot shows the Node-RED interface with a workflow in progress. In the 'input' section, a 'SPARQL update' node is highlighted with a red dashed box. Below it, an 'http request' node is connected. The right-hand panel is titled 'Edit template node' and contains the following configuration:

- Name:** SPARQL update
- Property:** msg.payload
- Format:** Plain text
- Template:**

```
1 PREFIX bot: <https://w3id.org/bot#>
2 PREFIX ex: <https://www.example.com#>
3
4 INSERT DATA {
5   GRAPH <https://www.example.com> {
6     <https://www.example.com> owl:imports <https://w3id.org/bot#> .
7     ex:NewZone rdf:type owl:NamedIndividual , bot:Zone.
8   }
9 }
10
```
- Output as:** Plain text

Finalize the update by adding the text "query=" in front of it.

A Change node can be used together with a JSONata expression.

Other options: use a Function node; include the missing text in the previous node (Template)

The screenshot shows the Node-RED interface. On the left, a workflow is visible with a 'SPARQL update' node connected to a 'finalize query' node. The 'finalize query' node is highlighted with a red dashed box. Below it, an 'http request' node is connected to a 'statusCode' node. On the right, the 'Edit change node' panel is open, showing the configuration for the 'finalize query' node. The 'Name' field is set to 'finalize query'. The 'Rules' section is highlighted with a red dashed box and contains a 'Set' rule with the following configuration:

- Operation: Set
- Target: msg. payload
- Expression: j: "query=" & payload

Make a HTTP request with the **http request node**. Method POST is selected. Use authentication (unless Stardog is started w/o security) and specify user and password (default in Stardog: admin, admin). URL is already received as msg.url.

The screenshot displays the Node-RED web interface. On the left, a workflow is visible in the 'Update' tab, consisting of a 'SPARQL update' node, a 'finalize query' node, and an 'http request' node. The 'http request' node is highlighted with a red dashed box. On the right, the 'Edit http request node' configuration panel is open, also with a red dashed box around its authentication settings. The configuration includes:

- Method: POST
- URL: http://
- Enable secure (SSL/TLS) connection
- Use authentication
 - Type: basic authentication
 - Username: admin
 - Password: [masked]
- Use proxy
- Return: a UTF-8 string

The http request returns the status code that is shown in **msg.statusCode** using a **Debug node**. Code 200 means the request was successful.

The screenshot displays the Node-RED web interface. On the left, a workflow is visible in the 'Update' workspace. It consists of three nodes: a yellow 'SPARQL update' node, a green 'finalize query' node, and a green 'http request' node. A red dashed box highlights the 'http request' node. A line connects the 'http request' node to a green 'statusCode' node, which is also highlighted with a red dashed box. On the right, the 'Edit debug node' configuration panel is open. It features a 'Delete' button, 'Cancel' and 'Done' buttons, and a 'Properties' section. The 'Output' dropdown menu is set to 'msg. statusCode' and is highlighted with a red dashed box. Below it, the 'To' section has three options: 'debug window' (checked), 'system console' (unchecked), and 'node status (32 characters)' (unchecked). The 'Name' field is set to 'statusCode'.

Update results:

```
msg.statusCode : number  
200
```

Exported flow

```
[{"id":"8e217d16.03ffc8","type":"inject","z":"7b9ab328.92ad0c","name":"make  
request","topic":"","payload":"","payloadType":"date","repeat":"","crontab":"","once":false,"onceDelay":"","x":110,"y":60,"wires":  
[["e37b680d.98a458"]]}, {"id":"389a0ae7.cfc906","type":"template","z":"7b9ab328.92ad0c","name":"SPARQL  
update","field":"payload","fieldType":"msg","format":"text","syntax":"plain","template":"PREFIX bot:  
<https://w3id.org/bot#>\nPREFIX ex: <https://www.example.com#>\n\nINSERT DATA { \n GRAPH <https://www.example.com>  
{\n <https://www.example.com> owl:imports <https://w3id.org/bot#> .\n \tex:NewZone rdf:type owl:NamedIndividual ,  
bot:Zone.\n }\n}\n","output":"str","x":380,"y":140,"wires":  
[["482eb33b.cadd8c"]]}, {"id":"6c88eba5.82a624","type":"http  
request","z":"7b9ab328.92ad0c","name":"","method":"POST","ret":"txt","paytoqs":false,"url":"","tls":"","proxy":"","authType":"  
basic","x":330,"y":240,"wires":  
[["745b9c05.eba1c4"]]}, {"id":"745b9c05.eba1c4","type":"debug","z":"7b9ab328.92ad0c","name"  
:"statusCode","active":true,"tosidebar":true,"console":false,"tostatus":false,"complete":"statusCode","targetType":"msg","x":53  
0,"y":240,"wires":[]}, {"id":"e37b680d.98a458","type":"change","z":"7b9ab328.92ad0c","name":"Set URL &  
Headers","rules":[{"t":"set","p":"url","pt":"msg","to":"http://localhost:5820/LDAC/update","tot":"str"}, {"t":"set","p":"headers[c  
ontent-type]","pt":"msg","to":"application/x-www-form-  
urlencoded","tot":"str"}],"action":"","property":"","from":"","to":"","reg":false,"x":170,"y":140,"wires":  
[["389a0ae7.cfc906"]]}, {"id":"482eb33b.cadd8c","type":"change","z":"7b9ab328.92ad0c","name":"finalize  
query","rules":[{"t":"set","p":"payload","pt":"msg","to":"\nquery=\n &  
payload","tot":"jsonata"}],"action":"","property":"","from":"","to":"","reg":false,"x":570,"y":140,"wires":  
[["6c88eba5.82a624"]]  
]
```

Get all classes defined in BOT ontology . Then for each class get the properties where that class is the domain.

The proposed solution splits the results of the first query, thus creating a set of sequential messages that are used to configure the second query. Therefore the second query can be executed several times.

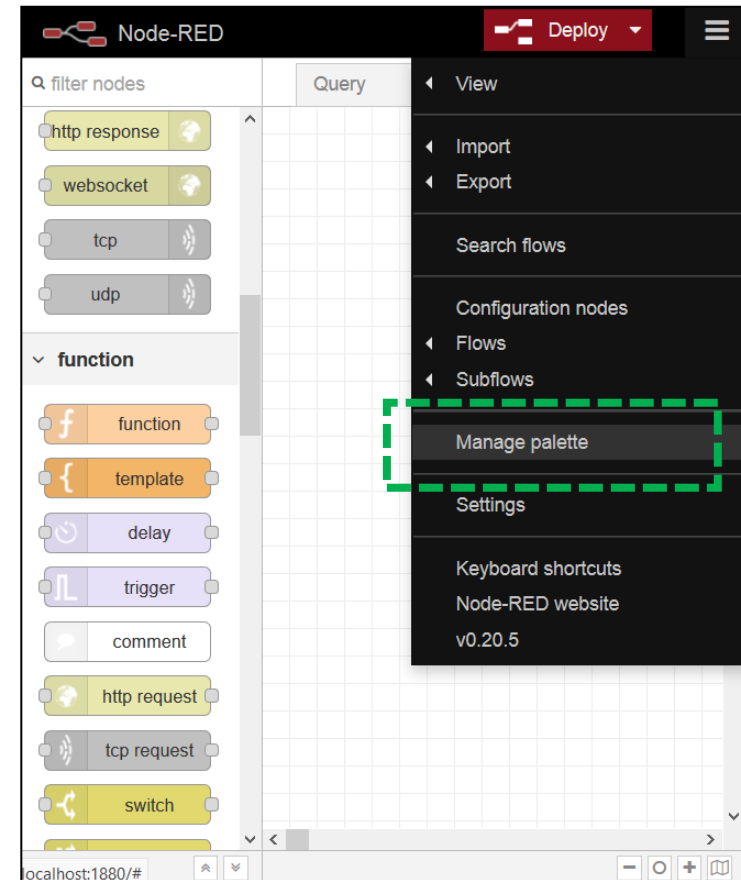
The screenshot displays the Node-RED web interface. The main workspace shows a flow titled "Query Sequence". The flow starts with a "make request" node, which triggers a "Set URL & Headers" node. This is followed by a "SPARQL query" node, then an "http request" node, and finally a "URIs" node. The output of the "URIs" node is connected to a "Move results to payload" node, which then feeds into a "split" node. The "split" node's output is connected to a "set msg.uri" node. This "set msg.uri" node then triggers a second "Set URL & Headers" node, followed by another "SPARQL query" node, an "http request" node, and finally a "Results" node. The "debug" console on the right shows the message flow: an empty array, a message with an empty array, a message with an array containing one object, an empty array, a message with an array containing seven objects, a message with an array containing one object, and finally an empty array.

Exported flow

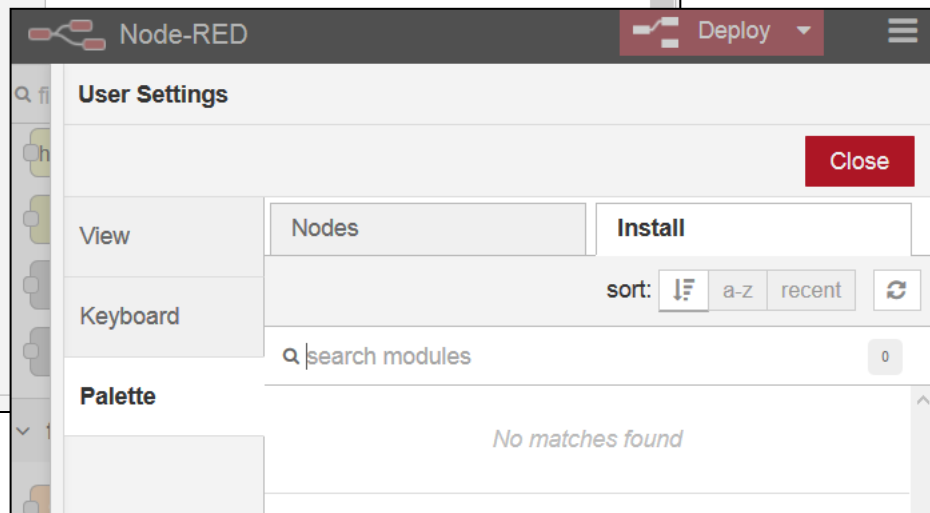
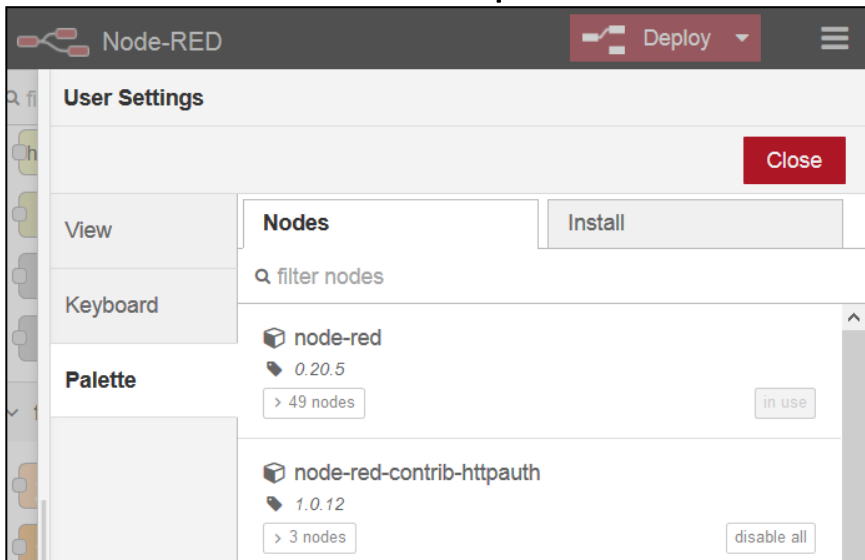
```
[{"id":"59bfaf1b.8937e","type":"tab","label":"Query
Sequence","disabled":false,"info":"","id":"b8f665a5.a6725","type":"inject","z":"59bfaf1b.8937e","name":"make
request","topic":"","payload":"","payloadType":"date","repeat":"","crontab":"","once":false,"onceDelay":"","x":130,"y":40,"wires":[["845281d5.018cf"]]},{"
id":"59d3a3ee.95f184","type":"template","z":"59bfaf1b.8937e","name":"SPARQL
query","field":"payload","fieldType":"msg","format":"text","syntax":"plain","template":"query=\nselect distinct ?botclass \nFROM
<https://w3id.org/bot>\nwhere { \n  ?botclass rdf:type owl:Class .
\n}","output":"str","x":420,"y":140,"wires":[["8248d7b0.962c08"]]},{"id":"8248d7b0.962c08","type":"http
request","z":"59bfaf1b.8937e","name":"","method":"POST","ret":"obj","paytoqs":false,"url":"","tls":"","proxy":"","authType":"basic","x":610,"y":140,"wir
es":[["c6e99b04.3067c","90c2c2e6.1e2ae"]]},{"id":"845281d5.018cf","type":"change","z":"59bfaf1b.8937e","name":"Set URL &
Headers","rules":[{"t":"set","p":"url","pt":"msg","to":"http://localhost:5820/LDAC/query","tot":"str"},{"t":"set","p":"headers[content-
type]","pt":"msg","to":"application/x-www-form-urlencoded","tot":"str"},{"t":"set","p":"headers[Accept]","pt":"msg","to":"application/sparql-
results+json","tot":"str"}],"action":"","property":"","from":"","to":"","reg":false,"x":190,"y":140,"wires":[["59d3a3ee.95f184"]]},{"id":"c6e99b04.3067c","t
ype":"debug","z":"59bfaf1b.8937e","name":"URLs","active":false,"tosidebar":true,"console":false,"tostatus":false,"complete":"payload.results.bindings.bo
tclass.value","targetType":"jsonata","x":810,"y":140,"wires":[]},{"id":"9f1ce207.58f23","type":"split","z":"59bfaf1b.8937e","name":"","splt":"\n","spltTyp
e":"str","arraySplt":1,"arraySpltType":"len","stream":false,"addname":"","x":470,"y":260,"wires":[["585fc625.2b85b"]]},{"id":"90c2c2e6.1e2ae","type":"ch
ange","z":"59bfaf1b.8937e","name":"Move results to
payload","rules":[{"t":"set","p":"payload","pt":"msg","to":"payload.results.bindings.botclass.value","tot":"jsonata"}],"action":"","property":"","from":"","to
":"","reg":false,"x":250,"y":260,"wires":[["9f1ce207.58f23"]]},{"id":"eef48833.0ad108","type":"debug","z":"59bfaf1b.8937e","name":"Results","active":tru
e,"tosidebar":true,"console":false,"tostatus":false,"complete":"payload.results.bindings","targetType":"jsonata","x":800,"y":380,"wires":[]},{"id":"73ff640
4.25f12c","type":"template","z":"59bfaf1b.8937e","name":"SPARQL
query","field":"payload","fieldType":"msg","format":"handlebars","syntax":"mustache","template":"query=\nselect distinct ?botclass ?prop \nFROM
<https://w3id.org/bot>\nwhere { \n  VALUES ?botclass <{{uri}}> \n  ?prop rdfs:domain ?botclass
\n}","output":"str","x":420,"y":380,"wires":[["f488988f.d24918"]]},{"id":"f488988f.d24918","type":"http
request","z":"59bfaf1b.8937e","name":"","method":"POST","ret":"obj","paytoqs":false,"url":"","tls":"","proxy":"","authType":"basic","x":610,"y":380,"wir
es":[["eef48833.0ad108"]]},{"id":"ecca5a34.e79b4","type":"change","z":"59bfaf1b.8937e","name":"Set URL &
Headers","rules":[{"t":"set","p":"url","pt":"msg","to":"http://localhost:5820/LDAC/query","tot":"str"},{"t":"set","p":"headers[content-
type]","pt":"msg","to":"application/x-www-form-urlencoded","tot":"str"},{"t":"set","p":"headers[Accept]","pt":"msg","to":"application/sparql-
results+json","tot":"str"}],"action":"","property":"","from":"","to":"","reg":false,"x":210,"y":380,"wires":[["73ff6404.25f12c"]]},{"id":"585fc625.2b85b","ty
pe":"change","z":"59bfaf1b.8937e","name":"","rules":[{"t":"set","p":"url","pt":"msg","to":"payload","tot":"msg"}],"action":"","property":"","from":"","to":
":"","reg":false,"x":650,"y":260,"wires":[["ecca5a34.e79b4"]]}]
```

- implement SPARQL queries as seen during previous lectures of the LDAC Summer School
- repeat the SPARQL query/update using another triple store, e.g. GraphDB by Ontotext (<http://graphdb.ontotext.com/documentation/standard/sparql-compliance.html>). Other options are listed in the lecture about triple stores. Possibly only URL and headers must be updated.
- Make a flow that integrates a SPARQL query and a SPARQL update
- Make (complex) elaborations to generate a SPARQL query
- Make (complex) elaborations to consume the results of a SPARQL query
- Integrate SPARQL query/update with other data sources and IoT technologies, e.g.
 - Input/Output files
 - MQTT connection (e.g. in a sensor network)
 - UDP connection
 - A HTTP end-point (different from SPARQL end-point)
 - Other HTTP requests

- New nodes can be installed using the Palette Manager
- Browse and install published nodes.



Another way of adding new nodes is via npm. Refer to Node-RED manual.




Connect to a MQTT broker and subscribe to messages from a specified topic.

Use free online demo broker provided by HiveMQ

<http://www.hivemq.com/demos/websocket-client/>

Click on **Connect** button.

**HIVEMQ**
ENTERPRISE MQTT BROKER

Websockets Client Showcase

Connection

● ⌵


Host	Port	ClientID	Connect
<input type="text" value="broker.mqttdashboard.com"/>	<input type="text" value="8000"/>	<input type="text" value="clientId-1lqxiyaXwU"/>	
Username	Password	Keep Alive	Clean Session
<input type="text"/>	<input type="text"/>	<input type="text" value="60"/>	<input type="checkbox"/> <small>x</small>
Last-Will Topic	Last-Will QoS		Last-Will Retain
<input type="text"/>	<input type="text" value="0"/> <small>▼</small>		<input type="checkbox"/>
Last-Will Message			
<input type="text"/>			

Publish ⌵

Subscriptions ⌵

Messages ⌵

Configure Publish section



HIVEMQ
ENTERPRISE MQTT BROKER

Websockets Client Showcas

Connection

● connected ⌵

Subscriptions

Add New Topic Subscription

Publish

Topic

QoS

0
⌵

Retain

Publish

Message

Room temperature is 20°C

Messages

In Node-RED create a new flow and add an MQTT input node. Set the server connection and topic (LDAC/roomsensor).

The screenshot displays the Node-RED web interface. On the left, the 'input' category is expanded, showing various nodes including 'mqtt'. In the main workspace, an 'LDAC/roomsensor' MQTT input node is placed on a grid and is marked as 'connected'. On the right, the 'Edit mqtt in node' configuration panel is open. It features a 'Properties' section with the following settings: 'Server' is 'broker.mqtdashboard.com:1883', 'Topic' is 'LDAC/roomsensor', 'QoS' is set to '2', 'Output' is 'auto-detect (string or buffer)', and 'Name' is 'Name'. The 'Server' and 'Topic' fields are highlighted with green dashed boxes.

Add a Debug node and Deploy.

The screenshot shows the Node-RED web interface. On the left, the 'input' category is expanded, showing nodes like 'inject', 'catch', 'status', and 'link'. The main workspace contains a flow with an 'LDAC/roomsensor' node (purple) connected to a 'msg.payload' node (green). A dashed green box highlights the 'msg.payload' node. In the top right corner, a 'Deploy' button is also highlighted with a dashed green box. The interface includes a search bar for nodes, a toolbar with 'Query', 'Update', and 'Query Sequence' buttons, and a right-hand sidebar with a 'debug' tab.

Publish the message in HiveMQ by clicking on the “Publish” button.

The screenshot shows the HiveMQ Websockets Client Showcard interface. The top left features the HiveMQ logo and the text 'ENTERPRISE MQTT BROKER'. The top right says 'Websockets Client Showcard'. The main area is divided into sections: 'Connection' (status: connected), 'Publish', and 'Subscriptions'. The 'Publish' section is active, showing a 'Topic' field with 'LDAC/roomsensor', a 'QoS' dropdown set to '0', and a 'Retain' checkbox. A dashed green box highlights the 'Publish' button. Below the 'Publish' section, there is a 'Message' field containing the text 'Room temperature is 20°C'. The 'Subscriptions' section has an 'Add New Topic Subscription' button.

The subscription made in Node-RED receives the message and goes through Debug.

The screenshot shows the Node-RED interface. On the left, the 'input' category is expanded, showing nodes like 'inject', 'catch', 'status', and 'link'. In the center workspace, an 'MQTT subscribe' node (purple) is connected to a 'msg.payload' node (green). The MQTT node is labeled 'LDAC/roomsensor' and has a 'connected' status indicator. On the right, the 'debug' console is open, showing a message received at 17/6/2019, 18:06:11 from node 5aad1055.c00fe8. The message content is 'LDAC/roomsensor : msg.payload : string[24]' followed by 'Room temperature is 20°C'.

```
[{"id":"355547da.7ad858","type":"tab","label":"MQTT
subscribe","disabled":false,"info":""},{
  "id":"27f1b8cf.b11c2","type":"mqtt
in","z":"355547da.7ad858","name":"","topic":"LDAC/roomsensor","qos":"2","datatype":"auto","broker":
"9b10bbd9.0639e8","x":230,"y":120,"wires":[["5aad1055.c00fe8"]]},
  {"id":"5aad1055.c00fe8","type":"de
bug","z":"355547da.7ad858","name":"","active":true,"tosidebar":true,"console":false,"tostatus":false,"c
omplete":false,"x":470,"y":120,"wires":[]},
  {"id":"9b10bbd9.0639e8","type":"mqtt-
broker","z":"","name":"","broker":"broker.mqttdashboard.com","port":"1883","clientId":"","usetls":false,
"compatmode":true,"keepalive":"15","cleansession":true,"birthTopic":"","birthQos":"0","birthPayload":"","
closeTopic":"","closePayload":"","willTopic":"","willQos":"0","willPayload":""}]
```


If the message is structured as a JSON, then further elaborations are possible

Publish

Topic

QoS

Retain

Message

```
{"sensor": "temperature", "value": 20}
```

The screenshot shows the Node-RED interface. On the left, a list of nodes includes 'change', 'range', 'split', 'join', and 'sort'. In the workspace, an MQTT subscribe node is configured with the topic 'LDAC/roomsensor' and is connected to a 'json' node, which is then connected to a 'msg.payload' node. The right-hand side shows the 'debug' console with the following output:

```
17/6/2019, 18:15:47 node: 2d989f3c.0ea25
LDAC/roomsensor : msg.payload : Object
  object
    sensor: "temperature"
    value: 20
```

```
[{"id":"b4daf526.ebf55","type":"tab","label":"MQTT subscribe JSON","disabled":false,"info":""},{"id":"46639621.cd2db8","type":"mqtt in","z":"b4daf526.ebf55","name":"","topic":"LDAC/roomsensor","qos":"2","datatype":"auto","broker":"9b10bbd9.0639e8","x":110,"y":140,"wires":[["f689ede5.af7ae8"]]},{"id":"2d989f3c.0ea25","type":"debug","z":"b4daf526.ebf55","name":"","active":true,"tosidebar":true,"console":false,"tostatus":false,"complete":"false","x":350,"y":260,"wires":[]},{"id":"f689ede5.af7ae8","type":"json","z":"b4daf526.ebf55","name":"","property":"payload","action":"","pretty":false,"x":240,"y":200,"wires":[["2d989f3c.0ea25"]]},{"id":"9b10bbd9.0639e8","type":"mqtt-broker","z":"","name":"","broker":"broker.mqttdashboard.com","port":"1883","clientId":"","usetls":false,"compatmode":true,"keepalive":"15","cleansession":true,"birthTopic":"","birthQos":"0","birthPayload":"","closeTopic":"","closePayload":"","willTopic":"","willQos":"0","willPayload":""}]
```

Connects to a MQTT broker and publishes messages.

In Node-RED create a new flow and add an **MQTT output node**.

Set the server connection and topic (LDAC/officesensor).

The screenshot displays the Node-RED web interface. On the left, the 'input' category is expanded, showing various nodes including 'mqtt'. A 'mqtt' node is placed on the workspace and is marked as 'connected'. On the right, the 'Edit mqtt out node' configuration panel is open. The 'Server' field is set to 'broker.mqttdashboard.com:1883'. The 'Topic' field is empty. The 'QoS' field is set to 0, and the 'Retain' checkbox is checked. The 'Name' field is empty. A tip at the bottom of the configuration panel reads: 'Tip: Leave topic, qos or retain blank if you want to set them via msg properties.'

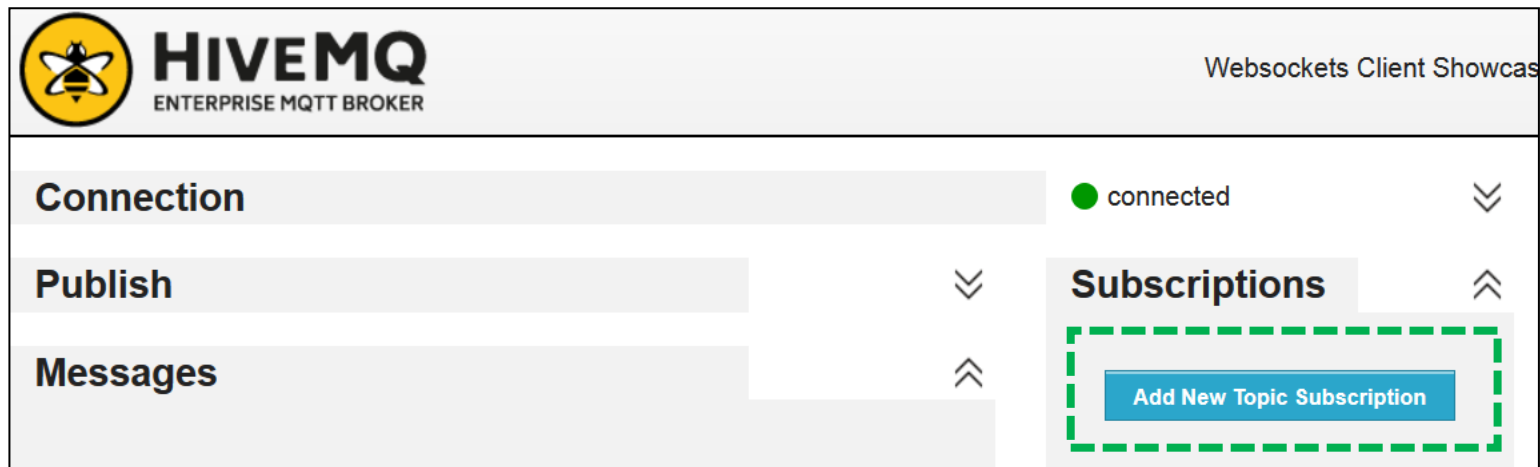
Add an Inject node that generates the message as JSON.

The screenshot shows the Node-RED interface with an 'inject' node selected in a flow. The 'Edit inject node' panel is open, displaying the following configuration:

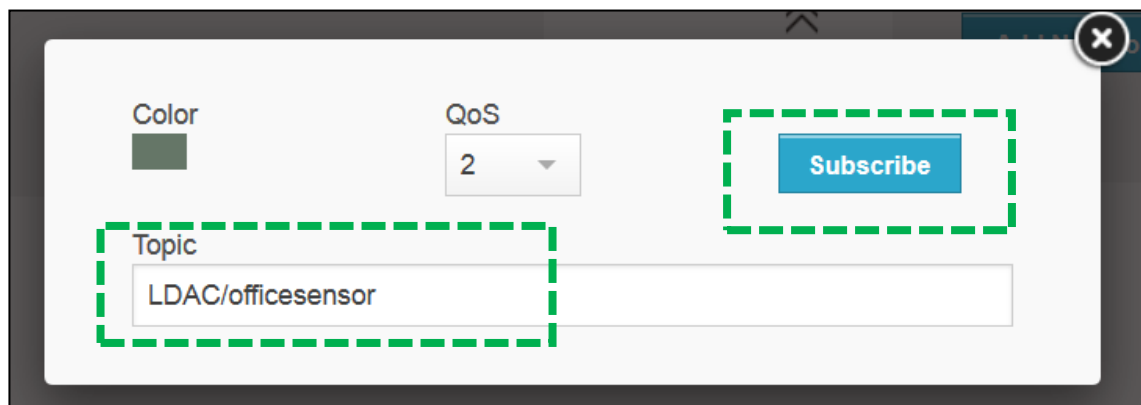
- Payload:** `{ "sensor": "temperature", "value": 18 }` (highlighted with a green dashed box)
- Topic:** `LDAC/officesensor` (highlighted with a green dashed box)
- Inject once after:** 0.1 seconds, then
- Repeat:** none
- Name:** Name

Note: "interval between times" and "at a specific time" will use cron. "interval" should be less than 596 hours. See info box for details.

In HiveMQ add a New Topic Subscription

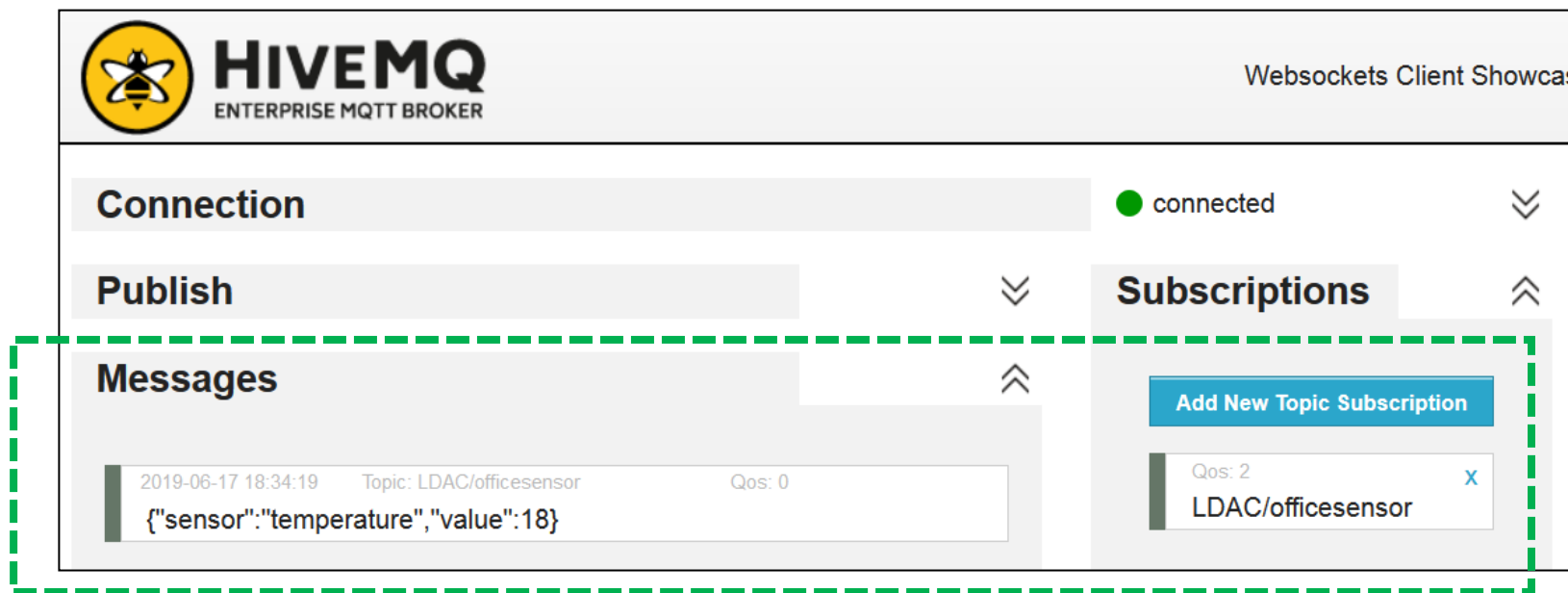
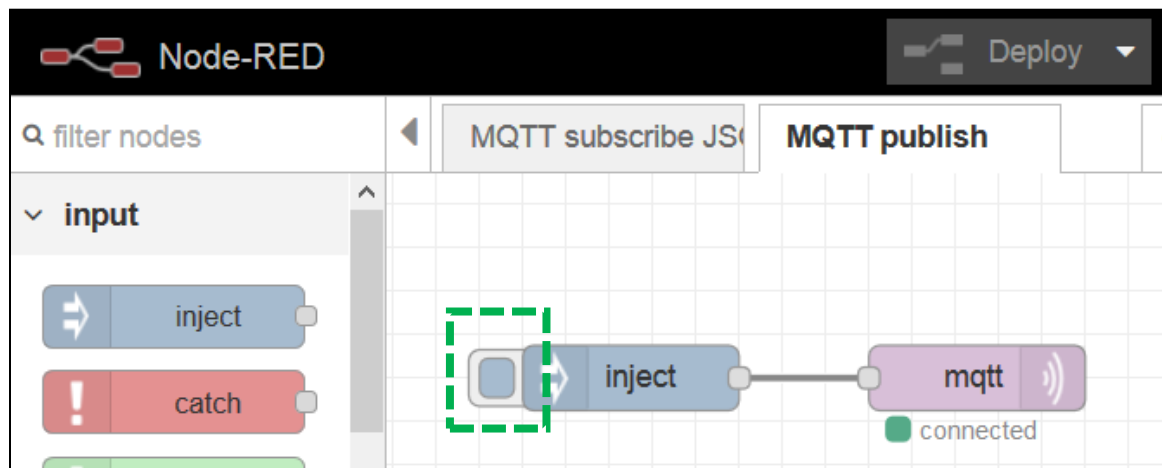


The screenshot shows the HiveMQ web interface. At the top left is the HiveMQ logo with the text "HIVEMQ ENTERPRISE MQTT BROKER". At the top right, it says "Websockets Client Showcas". Below the header, there are three main sections: "Connection" (with a green dot and "connected" status), "Publish", and "Messages". On the right side, there is a "Subscriptions" tab. A blue button labeled "Add New Topic Subscription" is highlighted with a green dashed border.



The screenshot shows a configuration dialog box for adding a new topic subscription. It has a close button (X) in the top right corner. The dialog contains the following elements: a "Color" field with a green square, a "QoS" dropdown menu set to "2", a "Topic" input field containing "LDAC/officesensor", and a blue "Subscribe" button. The "Topic" field and the "Subscribe" button are highlighted with green dashed borders.

In Node-RED click on Inject button and the message will be received in HiveMQ



```
[{"id":"88732d77.c66628","type":"tab","label":"MQTT
publish","disabled":false,"info":""},{id":"92745b13.80ea8","type":"mqtt
out","z":"88732d77.c66628","name":"","topic":"","qos":"","retain":"","broker":"ee3aa3bc.8ac2f","x
":530,"y":280,"wires":[]},{id":"9991b19.87de85","type":"inject","z":"88732d77.c66628","name":"","
","topic":"LDAC/officesensor","payload":"{\"sensor\": \"temperature\", \"value\":
18}","payloadType":"json","repeat":"","crontab":"","once":false,"onceDelay":0.1,"x":370,"y":280,"w
ires":[["92745b13.80ea8"]]},{"id":"ee3aa3bc.8ac2f","type":"mqtt-
broker","z":"","name":"","broker":"broker.mqttdashboard.com","port":"1883","clientId":"","usetls":
false,"compatmode":true,"keepalive":"60","cleansession":true,"birthTopic":"","birthQos":"0","birth
Payload":"","closeTopic":"","closePayload":"","willTopic":"","willQos":"0","willPayload":""}]
```

Setting to execute a SPARQL Query using **GraphDB** endpoint via HTTP request.

The reference example is making use of the OpenSmartHomeDataSet:

<https://github.com/linkedbuildingdata/SummerSchoolOfLDAC/blob/master/Notebooks/00-01-Check-Internet-Connection.ipynb>

URL of endpoint

<https://rdf.ontotext.com/4139541402/mydb/repositories/OpenSmartHomeDataSet>

Headers

`content-type = application/x-www-form-urlencoded`

`Accept = application/sparql-results+json`

Get sensor ID

```
PREFIX express: <http://purl.org/voc/express#>
PREFIX bot: <https://w3id.org/bot#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dog: <http://elite.polito.it/ontologies/dogont.owl#>
PREFIX seas: <https://w3id.org/seas/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX ifc: <http://www.buildingsmart-tech.org/ifcOWL/IFC4_ADD1#>

SELECT ?SensorIdent
WHERE{
  ?ifcglobID express:hasString "05i4VutGDCsQKCrT6CQvhu" .
  ?roomIFC ifc:globalId_IfcRoot ?ifcglobID .
  ?room skos:related ?roomIFC .
  ?room bot:containsElement ?TempSensor .
  ?TempSensor rdf:type dog:TemperatureSensor .
  ?TempSensor seas:connectsAt/dcterms:identifier ?SensorIdent .
}
```


The screenshot shows a Node-RED workflow for executing a SPARQL query. The workflow consists of the following nodes:

- make request**: A blue node that initiates the process.
- Set URL & Headers**: A yellow node that configures the request.
- SPARQL query**: An orange node containing the query text.
- finalize query**: A yellow node that processes the query results.
- http request**: A light green node that sends the request.
- Bindings**: A dark green node that outputs the query results.
- SensorID**: A dark green node that outputs the sensor identifier.

The debug console on the right shows the following JSON payload:

```

18/6/2019, 11:01:11 node: Bindings
msg.payload : Object
  object
    head: object
    results: object
      bindings: array[1]
        0: object
          SensorID: object
            type: "literal"
            value: "Room1Temp"
18/6/2019, 11:01:11 node: SensorID
msg : string[9]
"Room1Temp"
    
```

```
{["id":"a76e66b2.880158","type":"inject","z":"3aa979e2.2262ce","name":"make
request","topic":"","payload":"","payloadType":"date","repeat":"","crontab":"","once":false,"onceDelay":"","x":250,"y":100,"wir
es":[["4df97275.4a4a64"]],{"id":"7e5915c1.716c8c","type":"template","z":"3aa979e2.2262ce","name":"SPARQL
query","field":"payload","fieldType":"msg","format":"text","syntax":"plain","template":"PREFIX express:
<http://purl.org/voc/express#>\nPREFIX bot: <https://w3id.org/bot#>\nPREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-
ns#>\nPREFIX dog: <http://elite.polito.it/ontologies/dogont.owl#>\nPREFIX seas: <https://w3id.org/seas/>\nPREFIX dcterms:
<http://purl.org/dc/terms/>\nPREFIX skos: <http://www.w3.org/2004/02/skos/core#>\nPREFIX ifc: <http://www.buildingsmart-
tech.org/ifcOWL/IFC4_ADD1#>\n\nSELECT ?SensorIdent\nWHERE{\n?ifcglobID express:hasString \"05i4VutGDCsQKCrT6CQvhu\"
.\n?roomIFC ifc:globalId_ifcRoot ?ifcglobID .\n?room skos:related ?roomIFC .\n?room bot:containsElement ?TempSensor
.\n?TempSensor rdf:type dog:TemperatureSensor .\n?TempSensor seas:connectsAt/dcterms:identifier ?SensorIdent
.\n}","output":"str","x":480,"y":180,"wires":[["833db6ae.64fc98"]],{"id":"2cd033af.416474","type":"http
request","z":"3aa979e2.2262ce","name":"","method":"POST","ret":"obj","paytoqs":false,"url":"","tls":"","proxy":"","authType":
"basic","x":410,"y":300,"wires":[["90db6f97.854c2","98d08d2d.3bce"]],{"id":"90db6f97.854c2","type":"debug","z":"3aa979e2.
2262ce","name":"Bindings","active":true,"tosidebar":true,"console":false,"tostatus":false,"complete":"payload","targetType":
"msg","x":660,"y":280,"wires":[]},{"id":"4df97275.4a4a64","type":"change","z":"3aa979e2.2262ce","name":"Set URL &
Headers","rules":[{"t":"set","p":"url","pt":"msg","to":"https://rdf.ontotext.com/4139541402/mydb/repositories/OpenSmartHo
meDataSet","tot":"str"},{"t":"set","p":"headers['content-type']","pt":"msg","to":"application/x-www-form-
urlencoded","tot":"str"},{"t":"set","p":"headers['Accept']","pt":"msg","to":"application/sparql-
results+json","tot":"str"}],"action":"","property":"","from":"","to":"","reg":false,"x":270,"y":180,"wires":[["7e5915c1.716c8c"]],
{"id":"98d08d2d.3bce","type":"debug","z":"3aa979e2.2262ce","name":"SensorID","active":true,"tosidebar":true,"console":false
,"tostatus":false,"complete":"payload.results.bindings.SensorIdent.value","targetType":"jsonata","x":660,"y":320,"wires":[]},{"id
":"833db6ae.64fc98","type":"change","z":"3aa979e2.2262ce","name":"finalize
query","rules":[{"t":"set","p":"payload","pt":"msg","to":"\"query=\" &
payload","tot":"jsonata"}],"action":"","property":"","from":"","to":"","reg":false,"x":670,"y":180,"wires":[["2cd033af.416474"]]]}
```



Sistemi e Tecnologie Industriali Intelligenti
per il Manifatturiero Avanzato
Consiglio Nazionale delle Ricerche

That's all folks!

walter.terkaj@stiima.cnr.it