



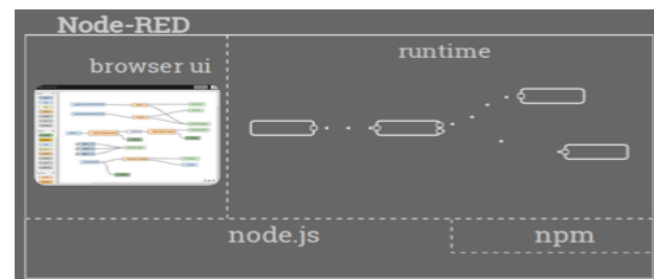
Sistemi e Tecnologie Industriali Intelligenti
per il Manifatturiero Avanzato
Consiglio Nazionale delle Ricerche

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/>

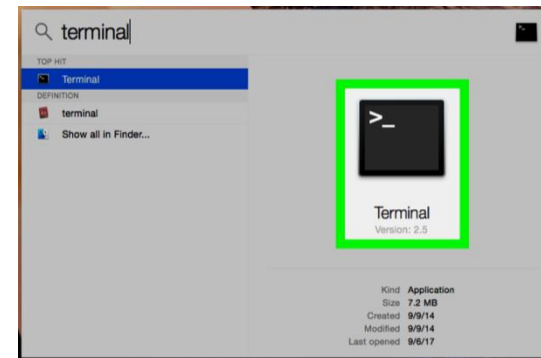


Linux / OsX

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):

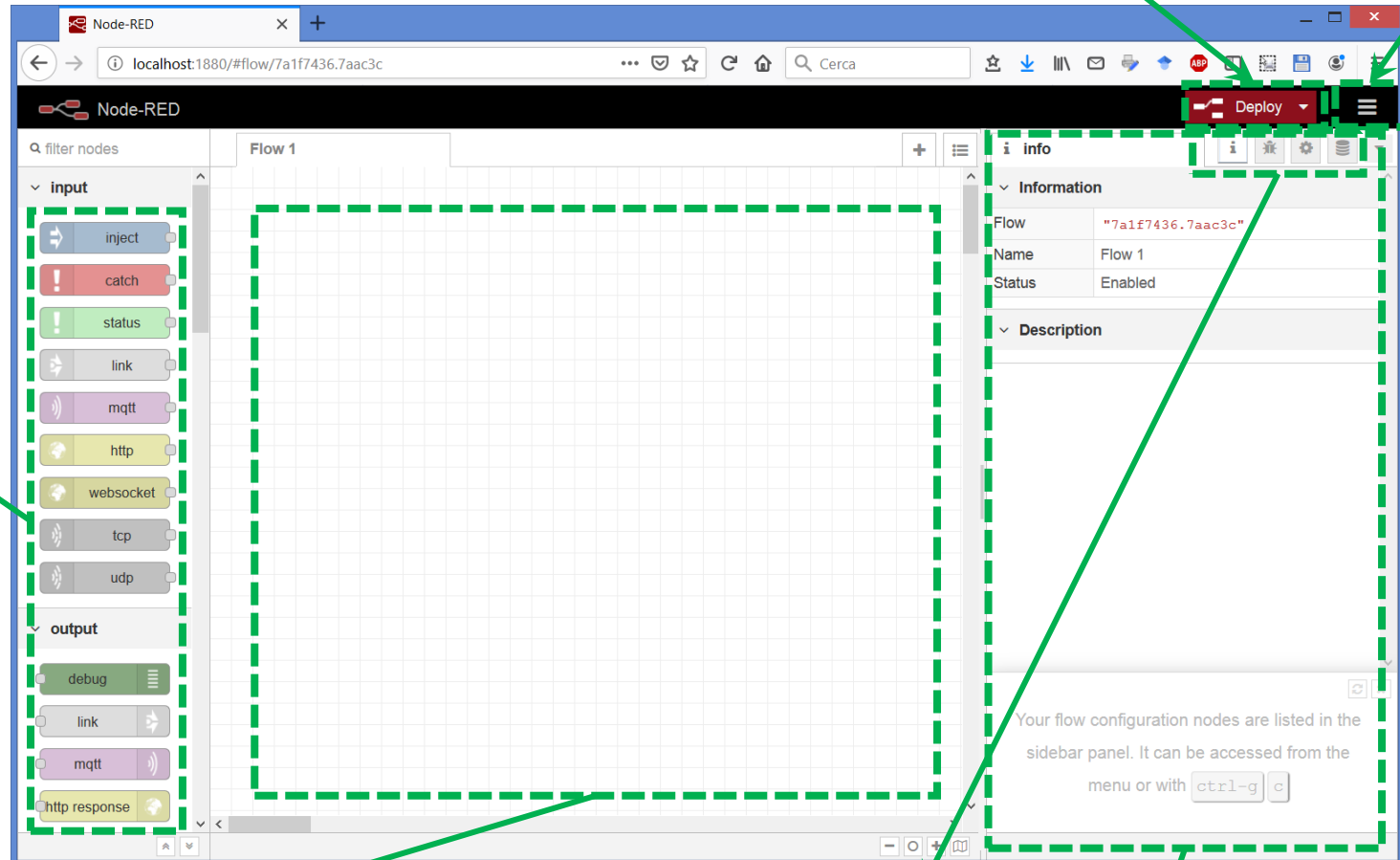
```
$ node-red
```

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



Panel showing contents:

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

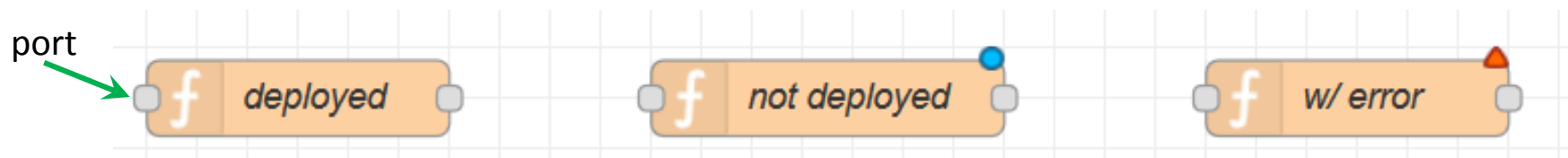
A Node-RED program is developed as a **FLOW** of messages running through a sequence of **NODES**. Each node can implement an elaboration of the message.

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.

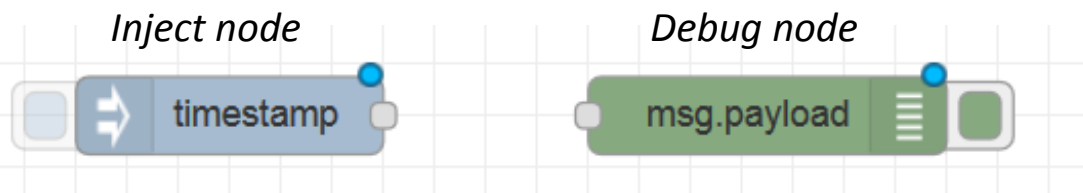
Nodes are added to a flow by simple **drag&drop**.

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



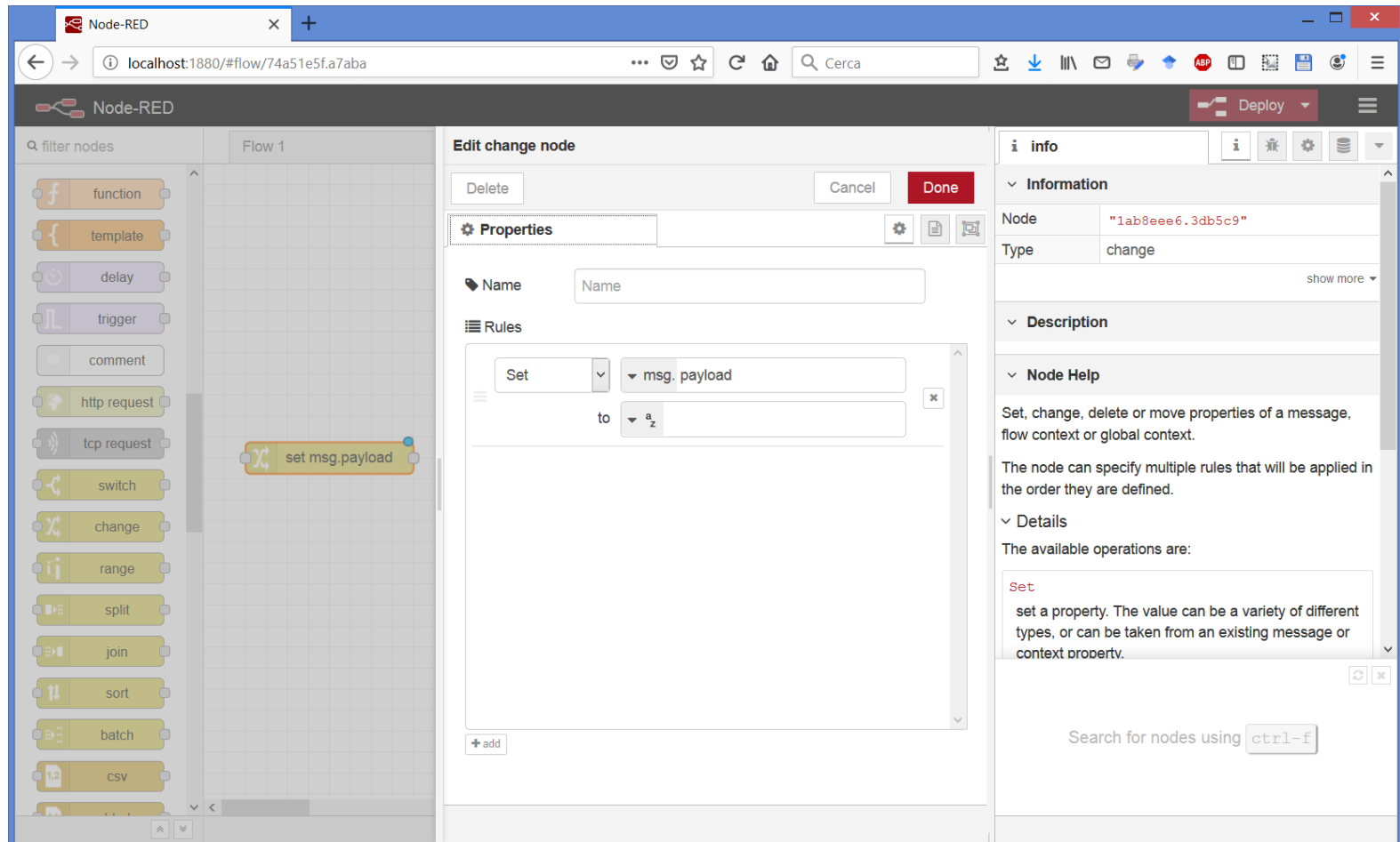
If a node has been change after the latest deployment, then it displays a blue circle above it. If there are errors with its configuration, it displays a red triangle.

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.



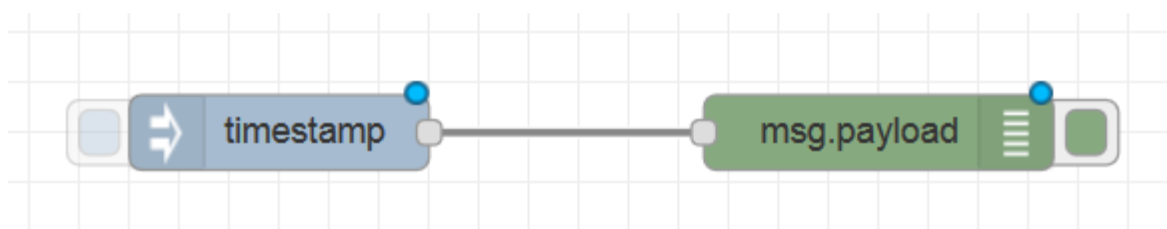
Wires define the connections between node input and output endpoints in a flow.

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.

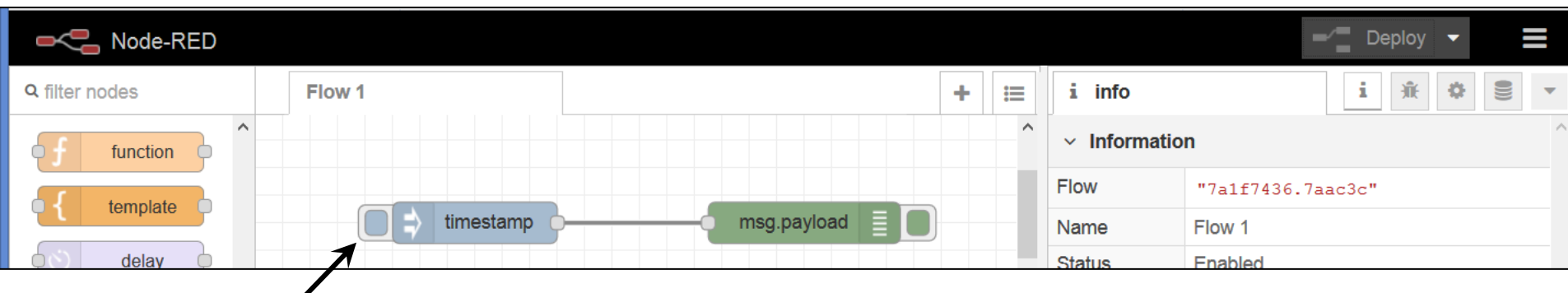
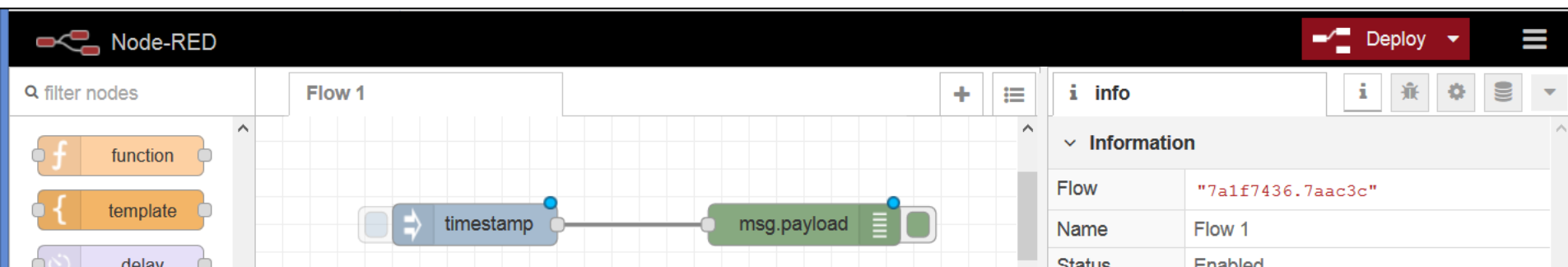
How to create wires?

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.



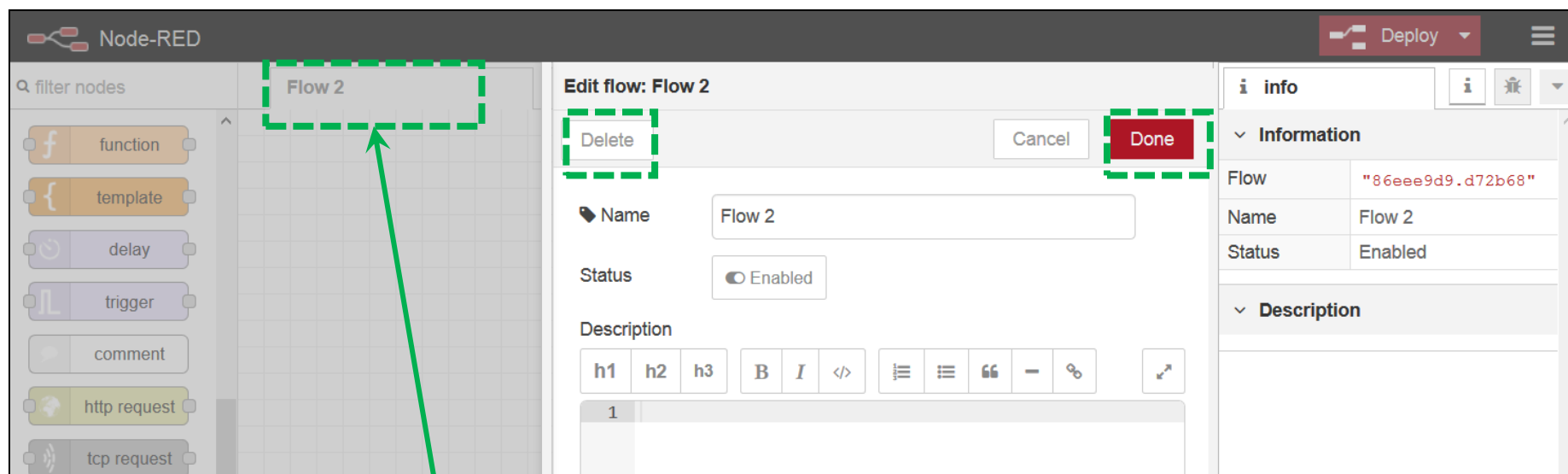
Active button of
Inject node

Create a Flow: click on the '+' tab



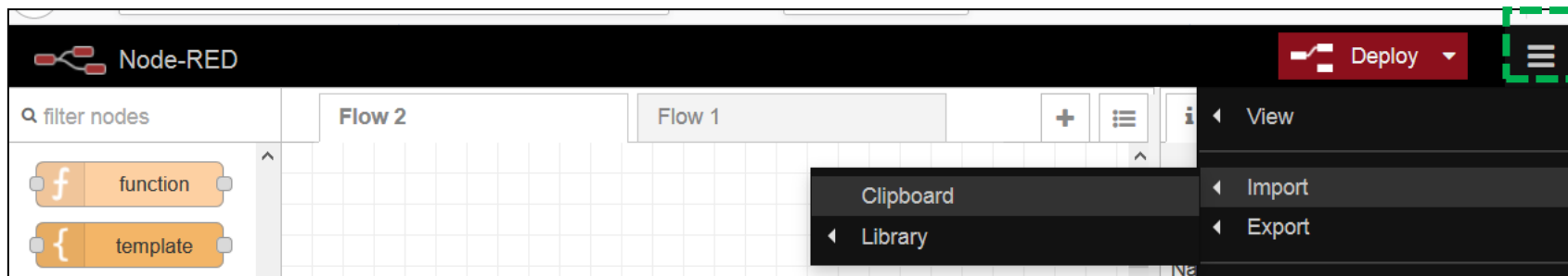
Delete a Flow: double-click on the flow tab and press the button «Delete»

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

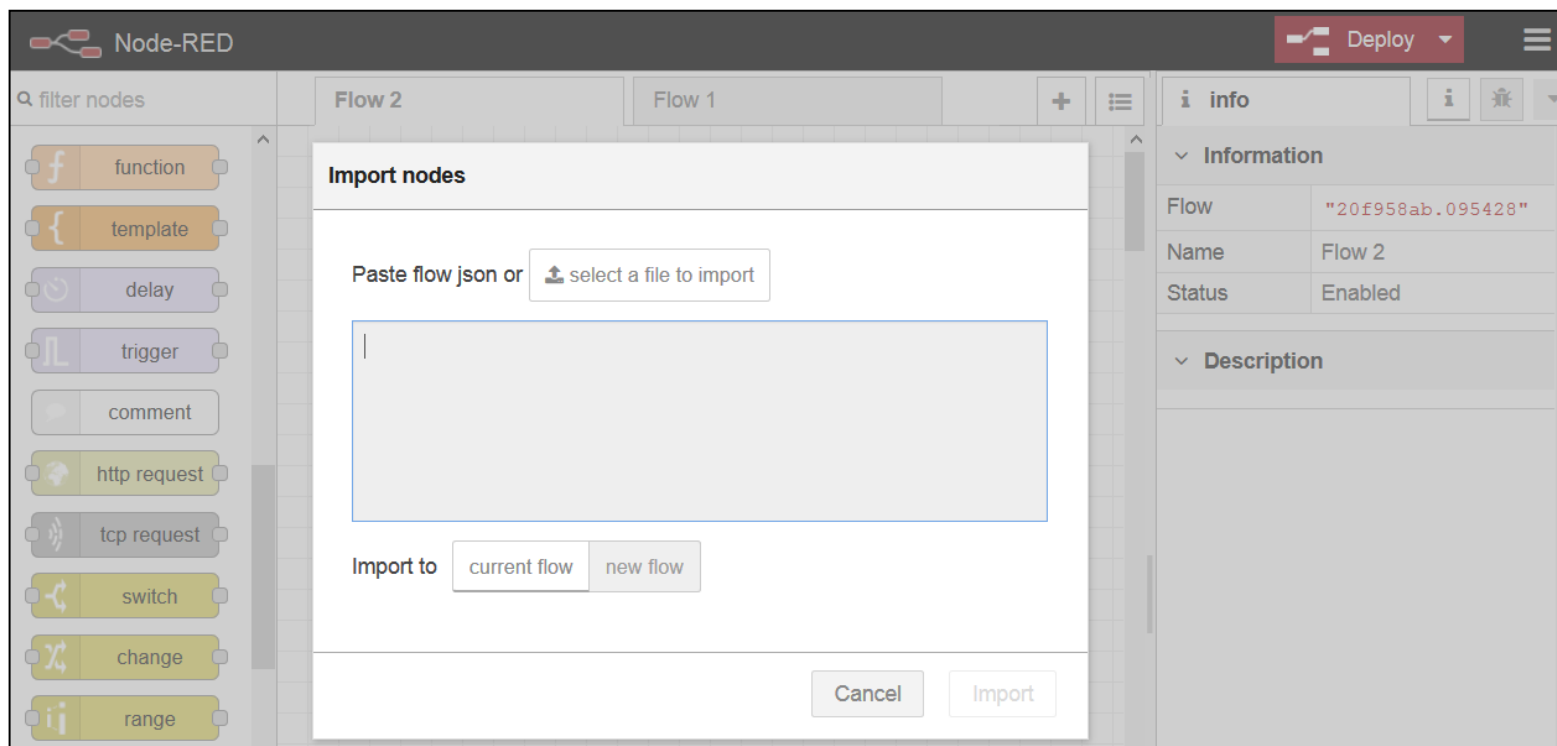


double-click on the flow tab

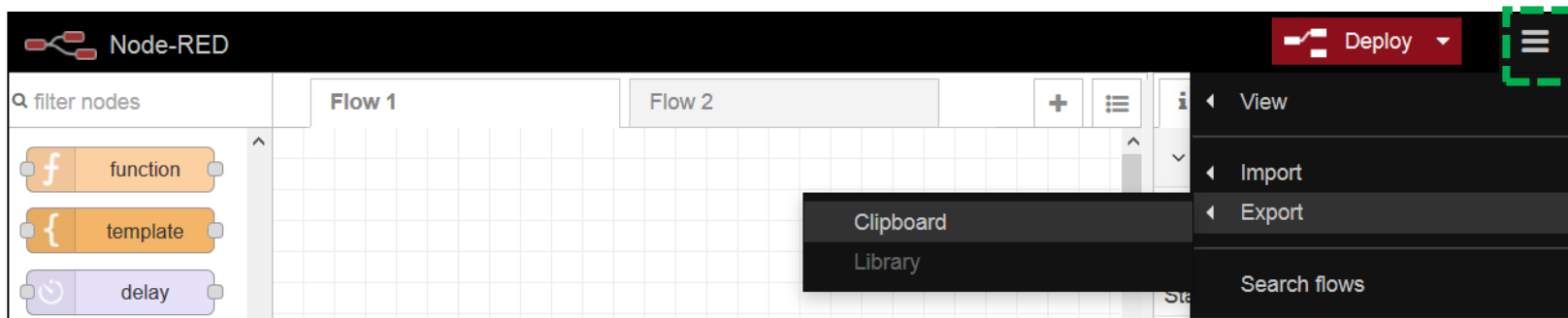
Import a Flow: click on the main menu and select «Import», «Clipboard»



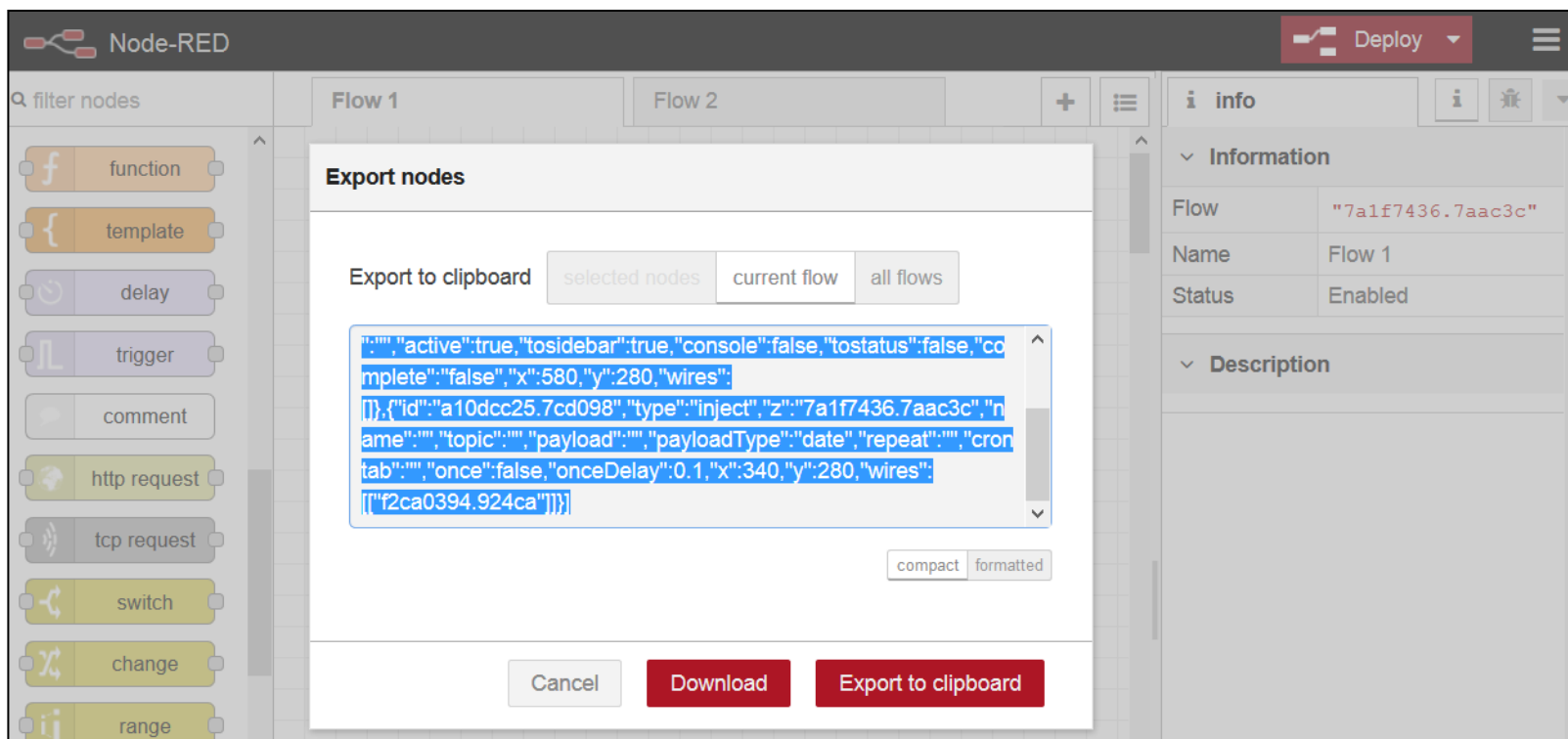
Then copy&paste the flow or select an input Json file



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

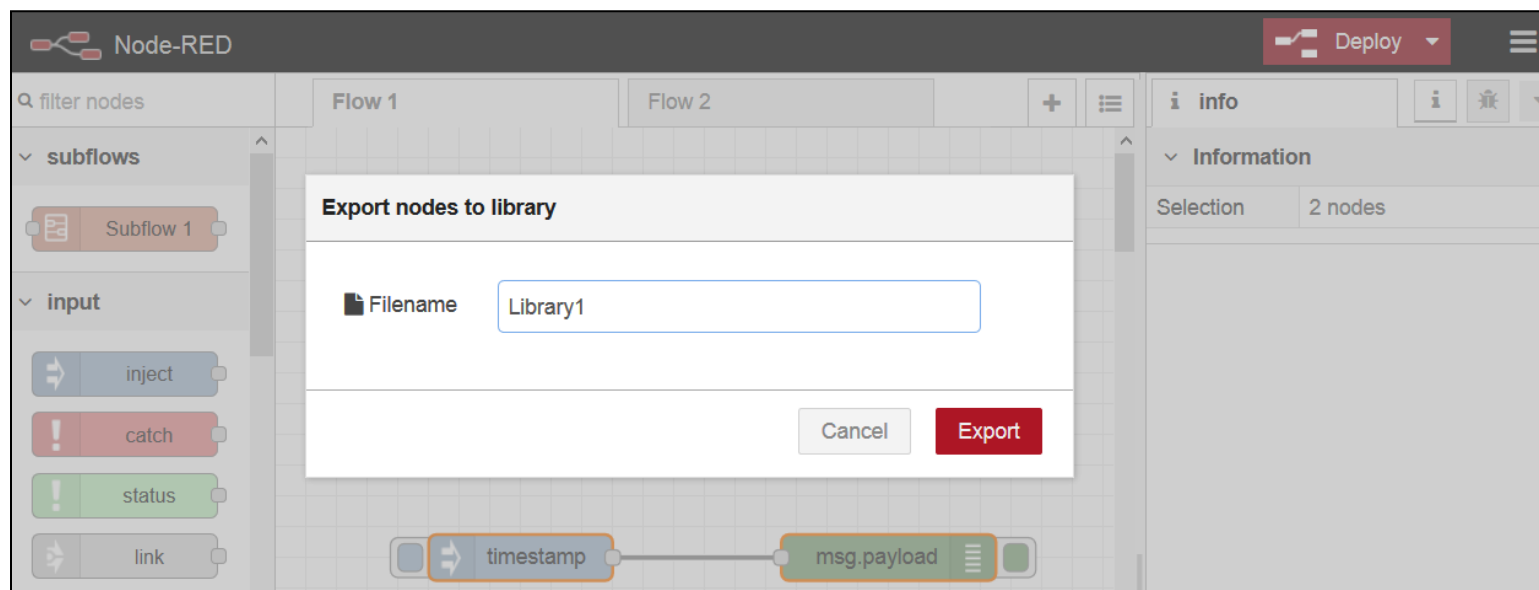


Then copy the flow or download it as a Json file.

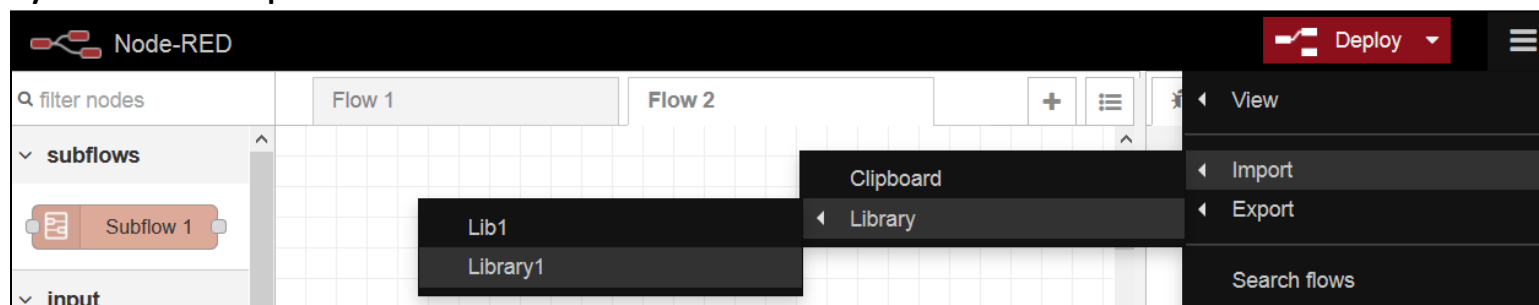


Libraries can be created to save a flow or function for reuse it in another flow .

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



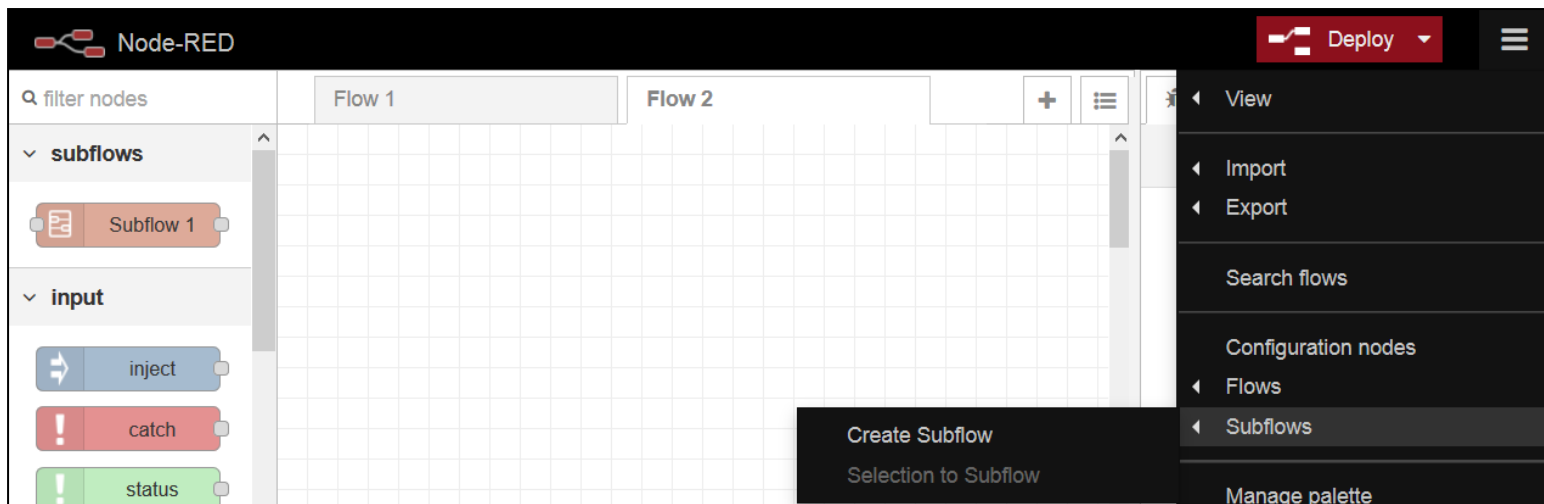
Import a flow from a library: click on the main menu > **Import > Library**. Click on the chosen library name and place the nodes in the flow.



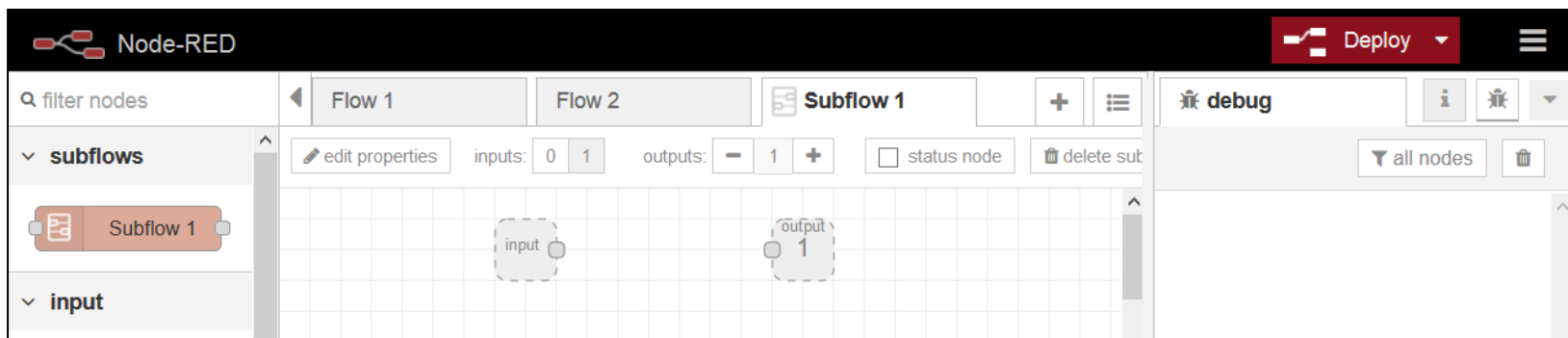
A Subflow is an aggregation of nodes and wires with input and output ports.

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 - eg 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 interface with two flows. Flow 1 contains a 'timestamp' node connected to a 'msg.payload' node. The 'msg.payload' node is highlighted with an orange border. The debug sidebar on the right shows the message content:

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

The screenshot shows a Node-RED interface with two flows. Flow 1 contains a 'timestamp' node connected to a 'msg' node. The 'msg' node is highlighted with a green border. The debug sidebar on the right shows the 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.



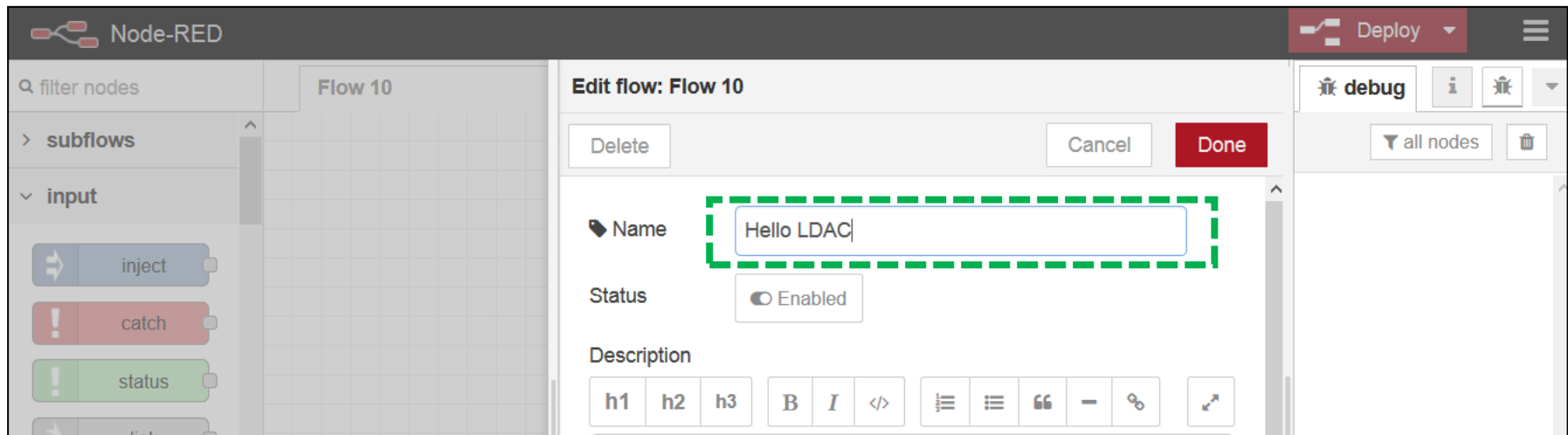
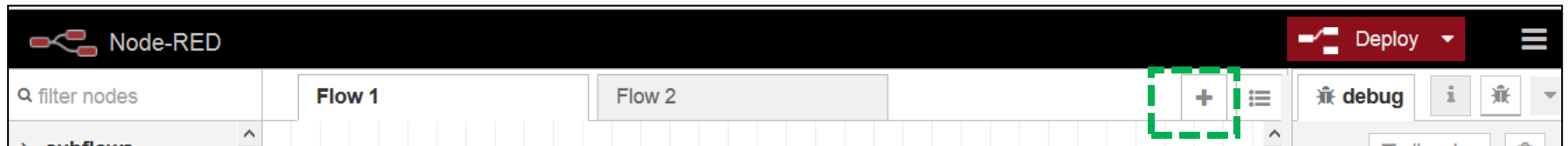
copies the value of the element to your clipboard as a JSON string.



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)

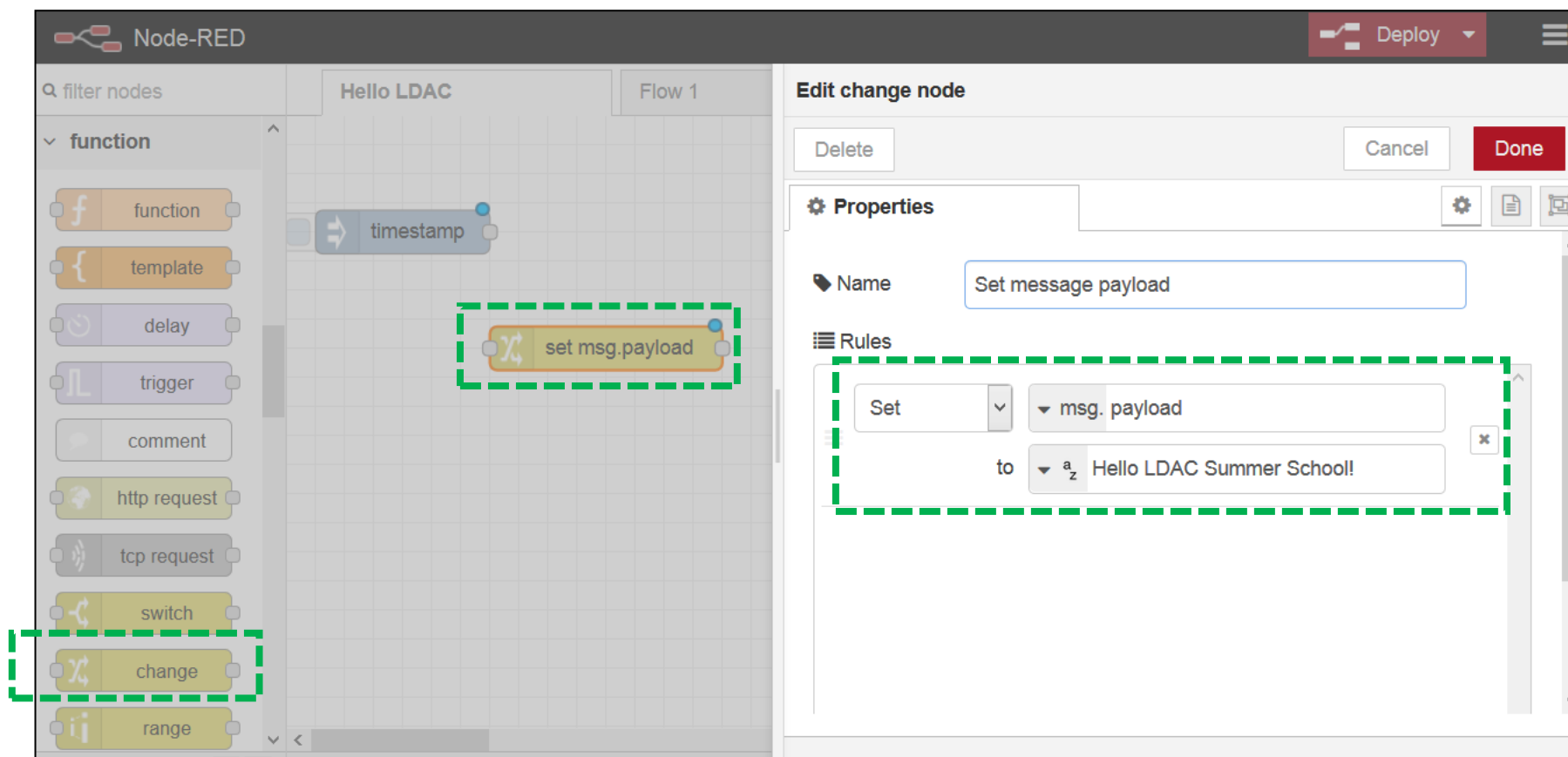
1. Create a new Flow and rename it



2. Add Inject node (from input nodes)

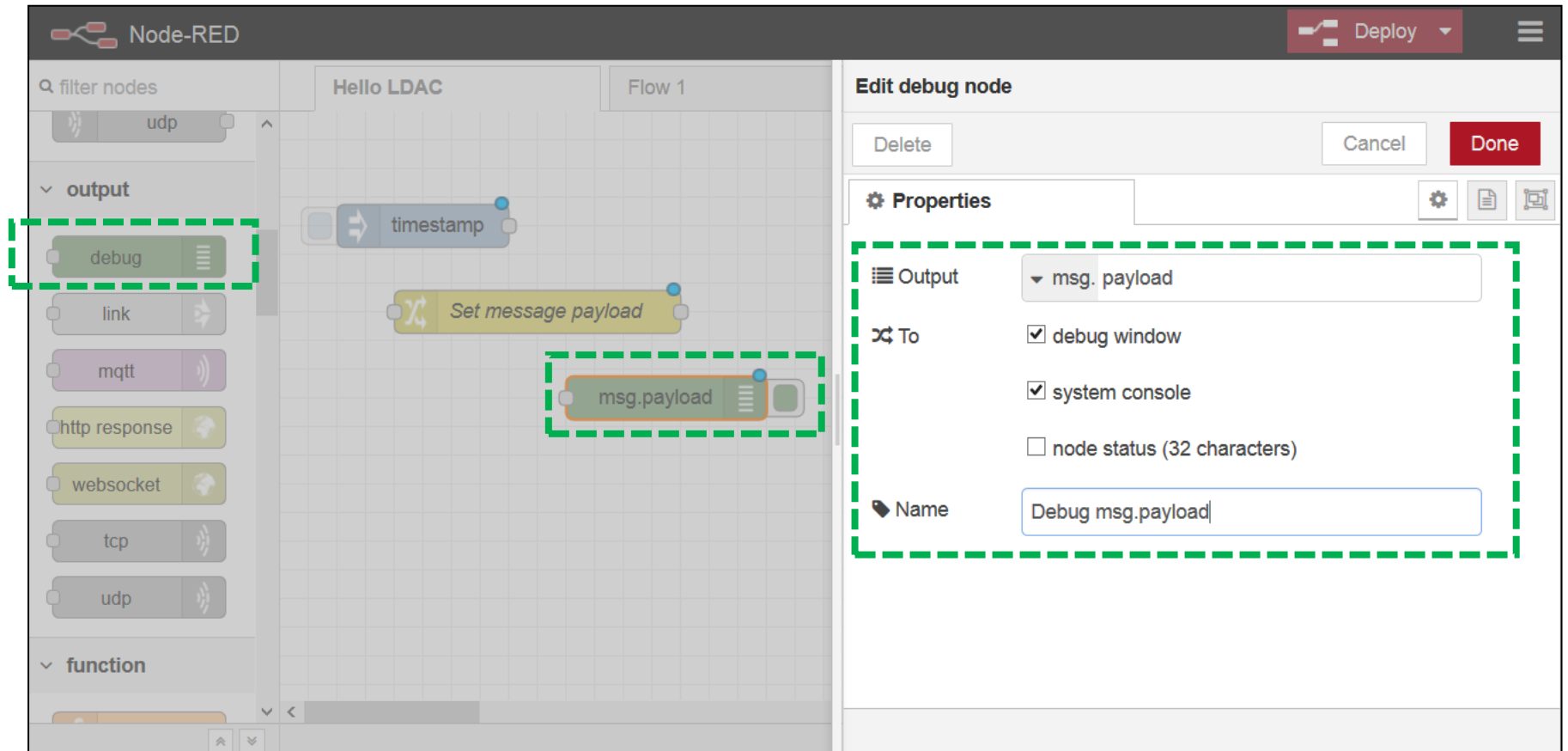
The screenshot shows the Node-RED web interface. The top bar includes the Node-RED logo, a search bar, and a 'Deploy' button. The main workspace is titled 'Hello LDAC' and contains two flow tabs: 'Flow 1' and 'Flow 2'. On the left sidebar, the 'input' category is expanded, showing a list of nodes: 'inject',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

3. Add Change node (from function nodes)



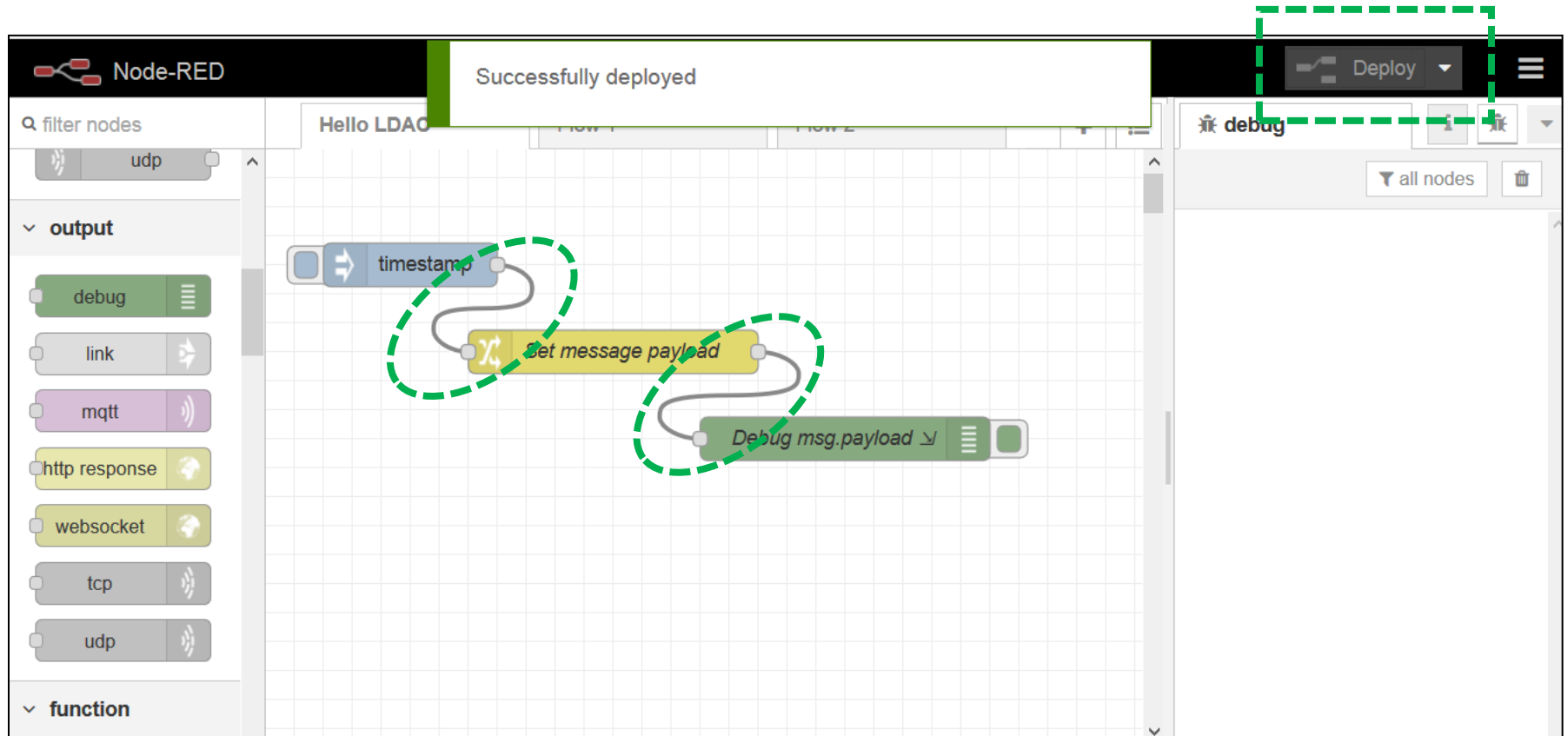
The screenshot shows the Node-RED web interface. On the left, the 'function' category in the node palette is expanded, and the 'change' node is highlighted with a green dashed box. In the center workspace, a flow named 'Hello LDAC' is shown with a 'timestamp' node followed by a 'set msg.payload' node,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

4. Add Debug node (from output nodes)



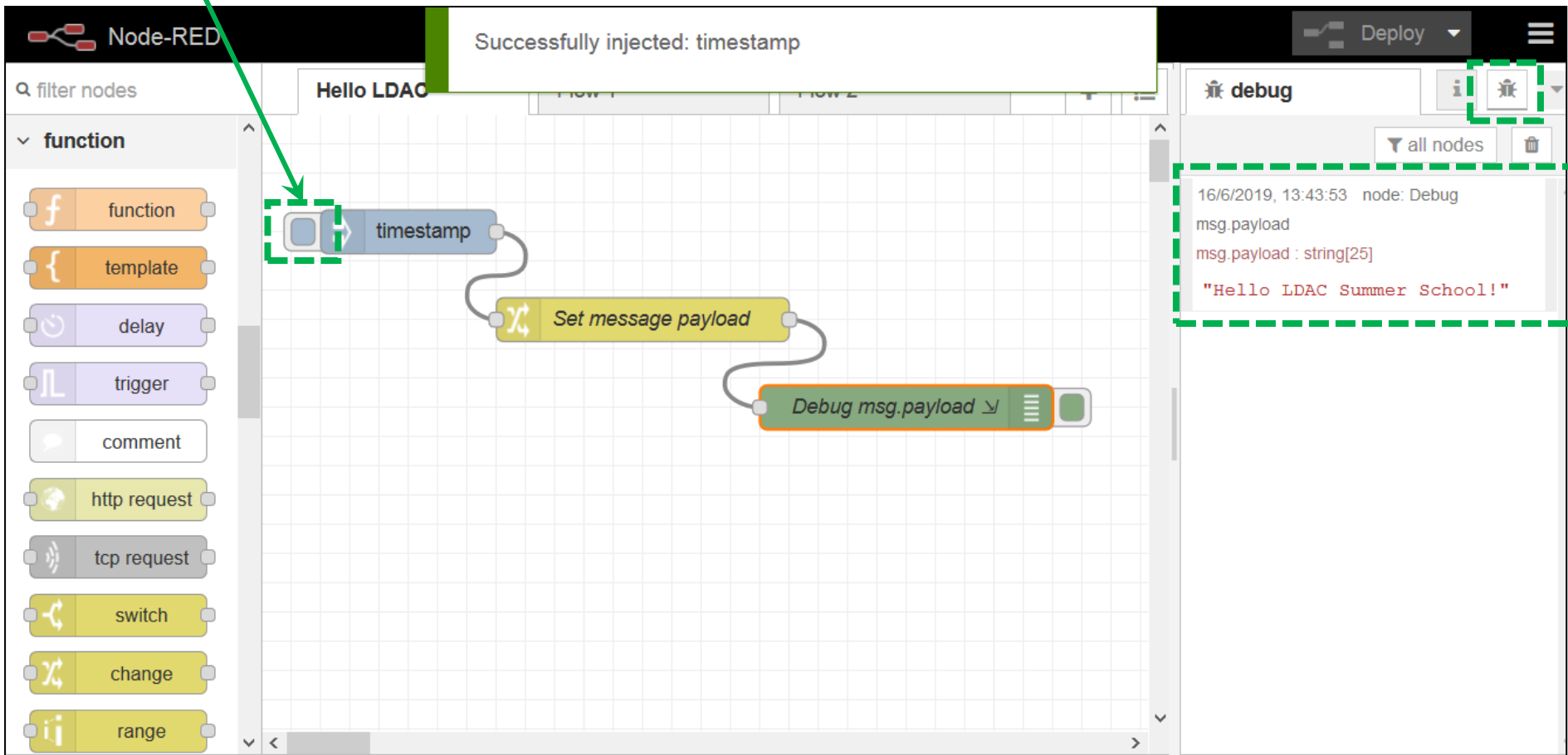
The screenshot shows the Node-RED web interface. On the left, the 'output' category is expanded, and the 'debug' node is highlighted with a green dashed box. In the center workspace, a flow named 'Hello LDAC' is visible, containing a 'timestamp' node, a 'Set message payload' node, and a 'msg.payload' node, which is also highlighted with a green dashed box. On the right, the 'Edit debug node' panel is open. The 'Properties' section is expanded, and the 'Output' dropdown is set to 'msg. payload'. The 'To' section has 'debug window' and 'system console' checked. The 'Name' field contains 'Debug msg.payload'.

5. Connect wires and deploy



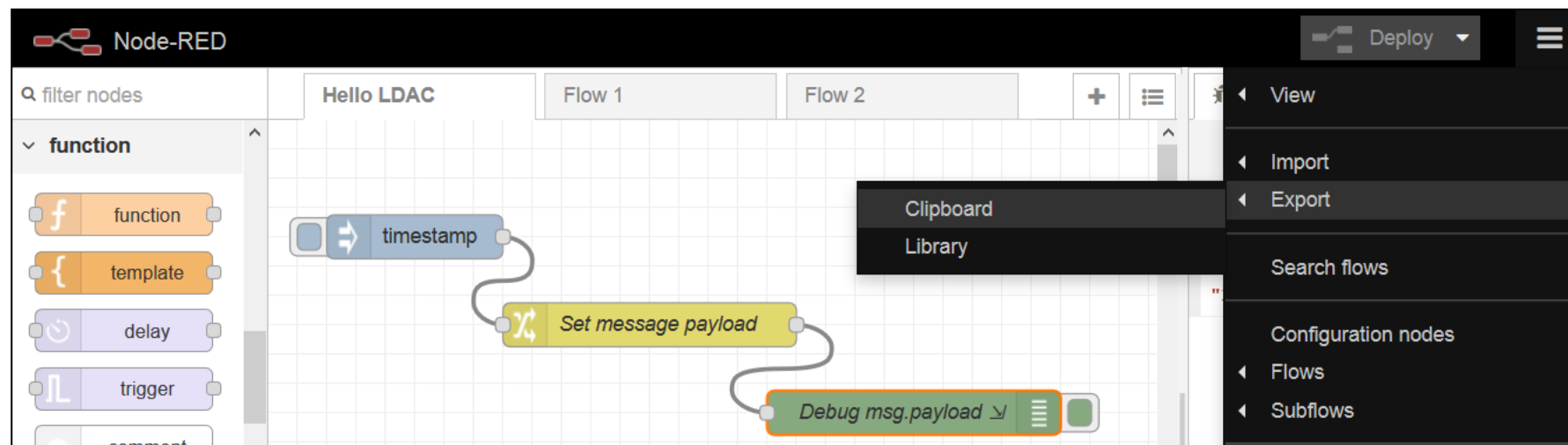
6. Execute: click on Inject button and check results in Debug pane

Start flow



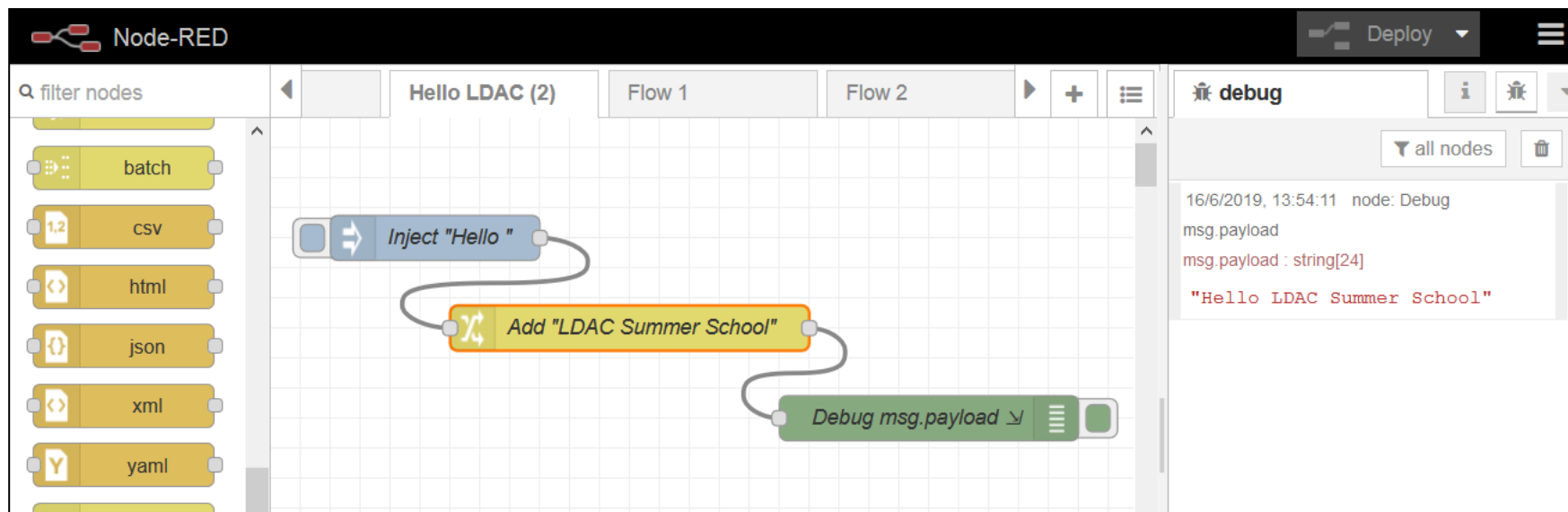
The screenshot shows the Node-RED web interface. A flow titled "Hello LDAC" is visible. The flow consists of three nodes: an inject node (highlighted with a green dashed box and a green arrow pointing to it with the text "Start flow"), a "timestamp" node, and a "Set message payload" node. The flow is connected to a "Debug msg.payload" node. The top status bar shows "Successfully injected: timestamp". On the right, the "debug" pane is open, showing the output of the debug node: "16/6/2019, 13:43:53 node: Debug", "msg.payload", and "msg.payload : string[25]" followed by the string "Hello LDAC Summer School!". The "Deploy" button is visible in the top right corner.

7. Export flow



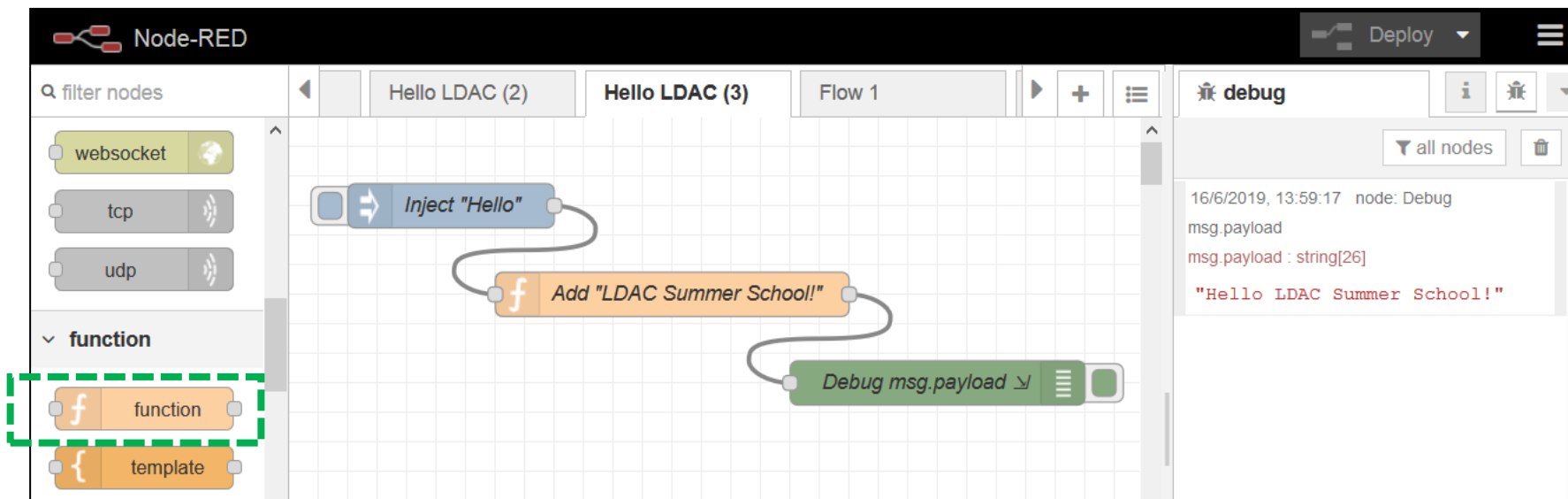
```
[{"id":"148e3965.53767f","type":"debug","z":"283e160d.c1304a","name":"Debug msg.payload","active":true,"tosidebar":true,"console":true,"tostatus":false,"complete":"payload","targetType":"msg","x":440,"y":200,"wires":[]},{id:"478a9031.b808","type":"inject","z":"283e160d.c1304a","name":"","topic":"","payload":"","payloadType":"date","repeat":"","crontab":"","once":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



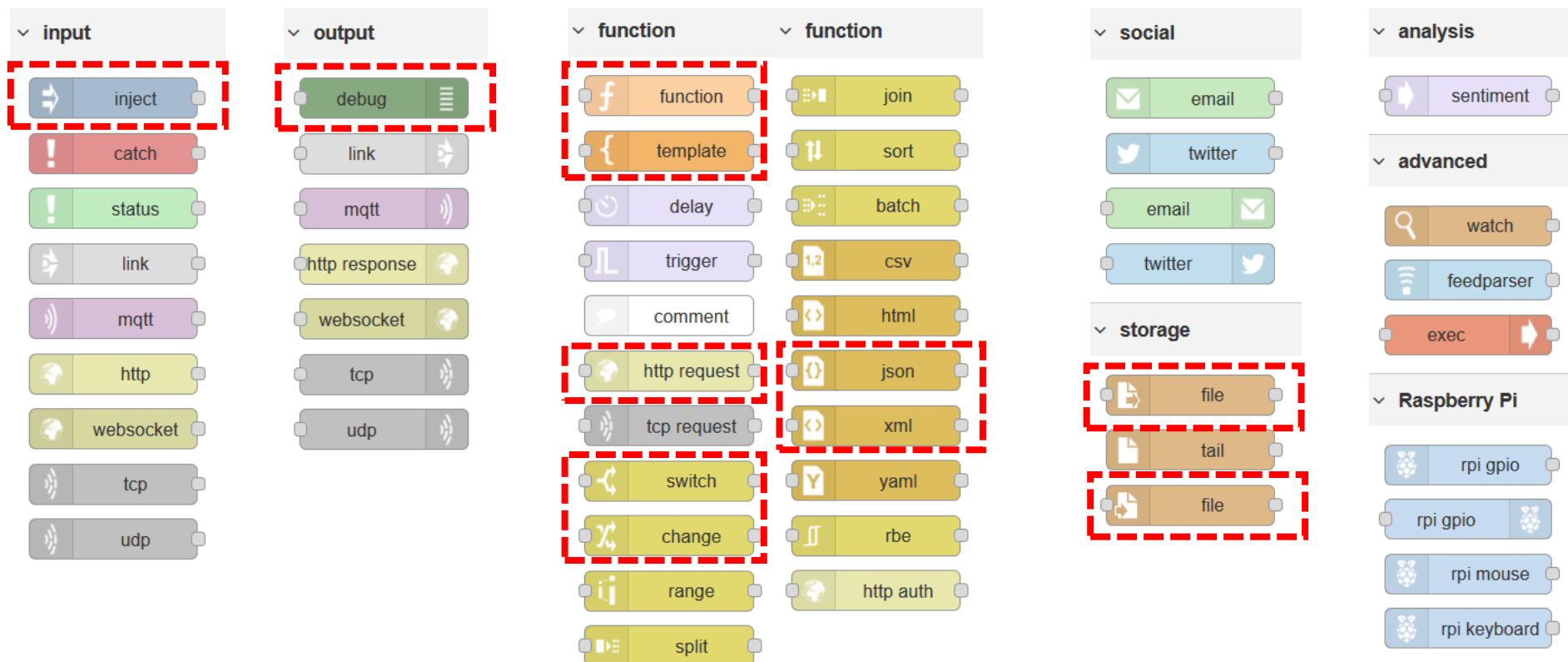
```
{
  "id": "4855565d.e62278",
  "type": "debug",
  "z": "85679f4e.4e529",
  "name": "Debug msg.payload",
  "active": true,
  "tosidebar": true,
  "console": true,
  "tostatus": false,
  "complete": "payload",
  "targetType": "msg",
  "x": 440,
  "y": 200,
  "wires": []
}, {
  "id": "67c10546.ab0594",
  "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"
    ]
  ]
}
```

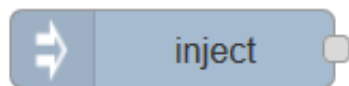
Alternative flow (3): use function node



```
{
  "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": "1ff89f23.5a8e11",
  "type": "function",
  "z": "810f777b.83fa2",
  "name": "Add \"LDAC Summer School!\"",
  "func": "msg.payload += \" LDAC Summer School!\\n\\nreturn msg;",
  "outputs": 1,
  "noerr": 0,
  "x": 260,
  "y": 120,
  "wires": [
    [
      "1f21a9f.25f0f56"
    ]
  ]
}
```



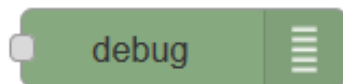




Injects a message into a flow either manually or at regular intervals. The message payload can be a variety of types, including strings, JavaScript objects or the current time.

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.



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.

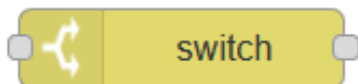


The Template node can be used to generate text using a message's properties to fill out a template. It uses the Mustache templating language to generate the result.

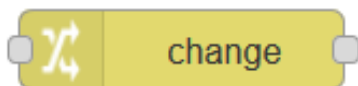
For example, a template of:

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

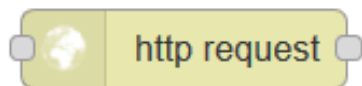
Will replace `{{payload}}` with the value of the message's payload property.



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.



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

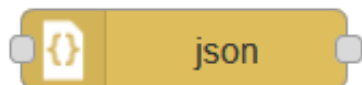


Sends HTTP requests and returns the response.

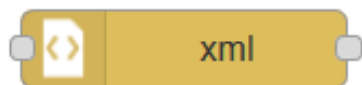
Key Inputs:

- 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.

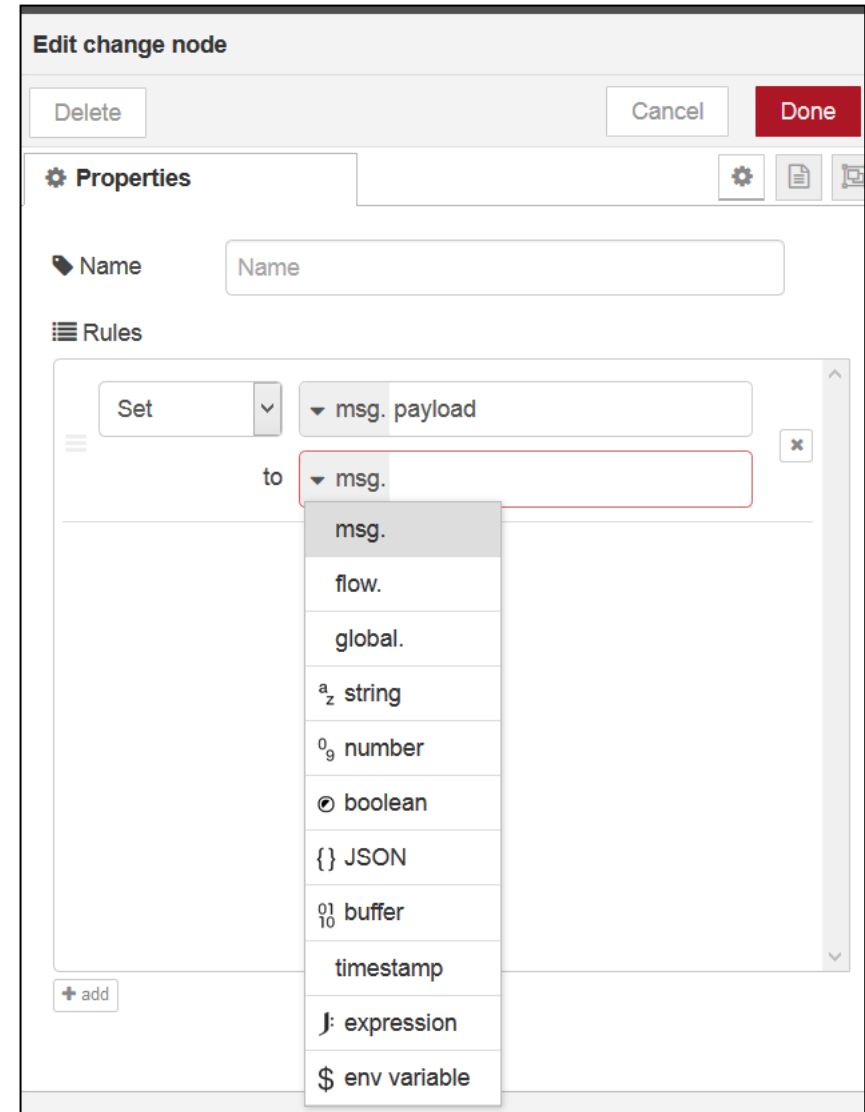
There are two main nodes for modifying a message, the Change node and the Function node.

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.

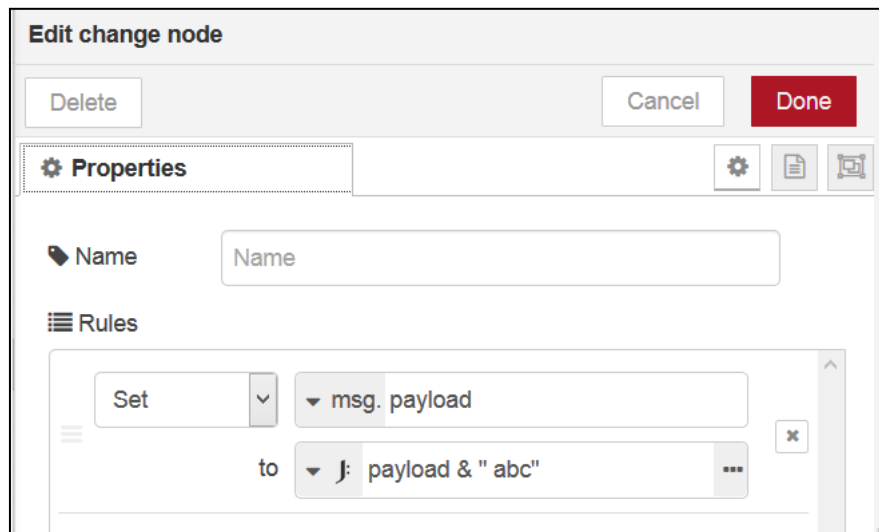
The Change node provides four basic operations:

- **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

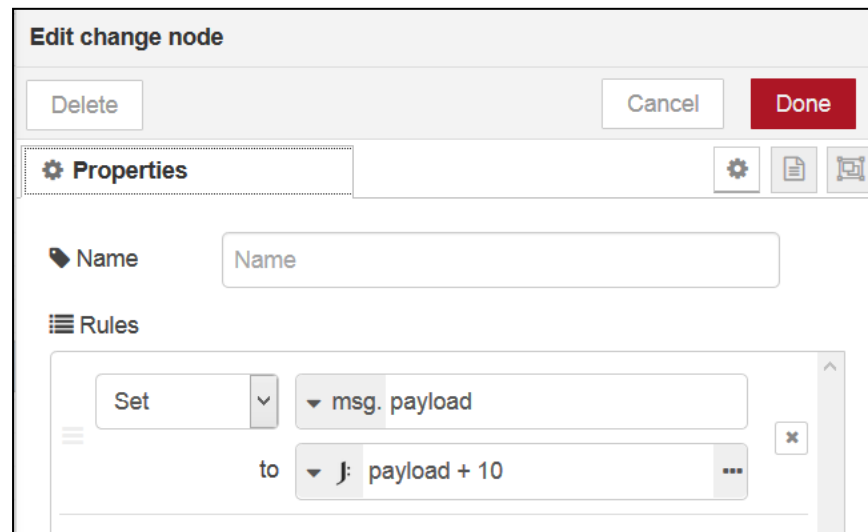
Properties

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

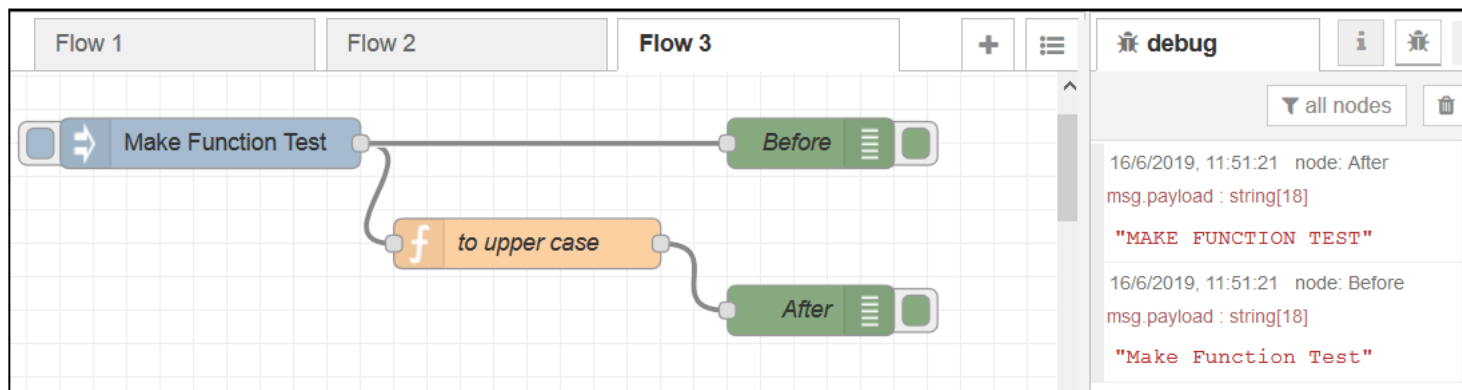
msg.payload = newpayload; // modify the message payload

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":"","topic":"","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":[]}

```

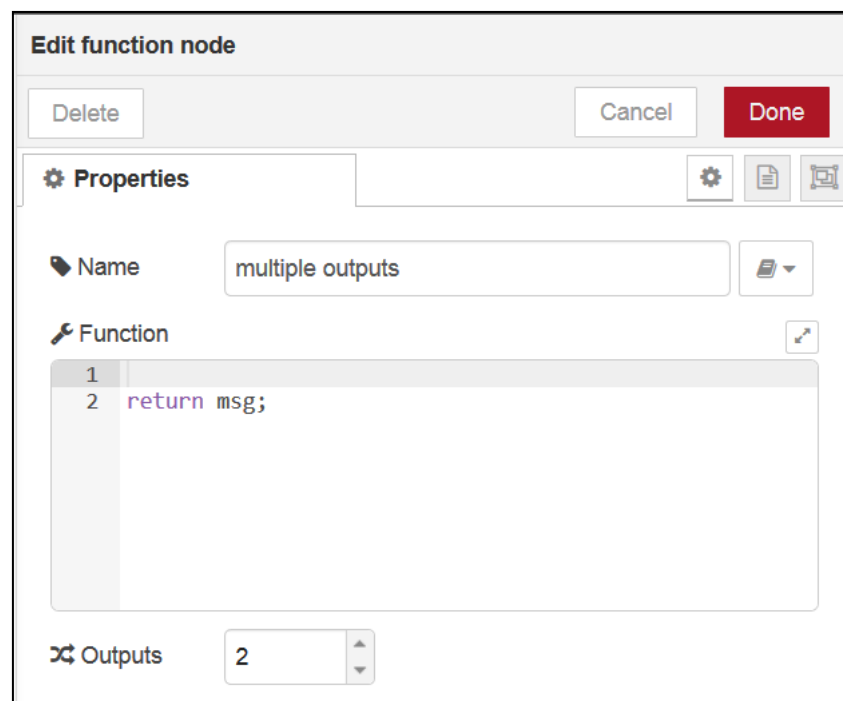
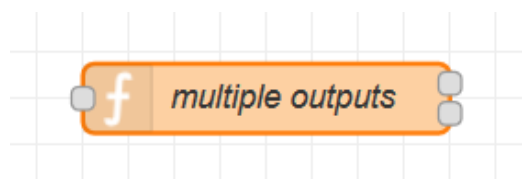
The function node can be configured with multiple outputs. This is useful when the flow splits into separate paths depending on a message property. To configure multiple outputs open the function node and use the up/down arrows to adjust the number of outputs.

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.

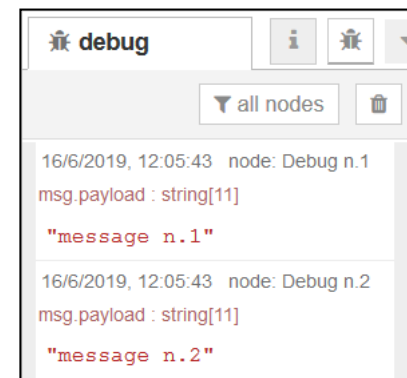
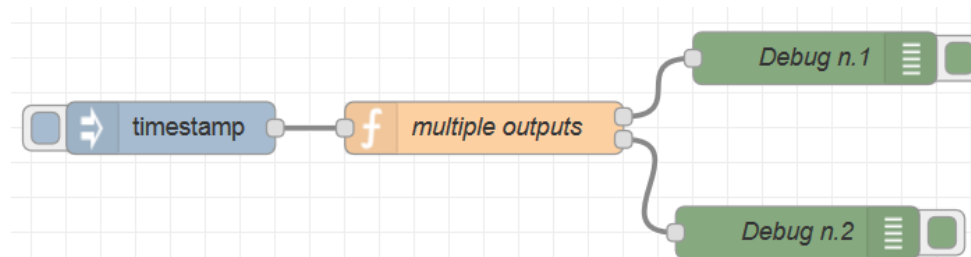
```
return[msg1,null];
```



Multiple outputs example

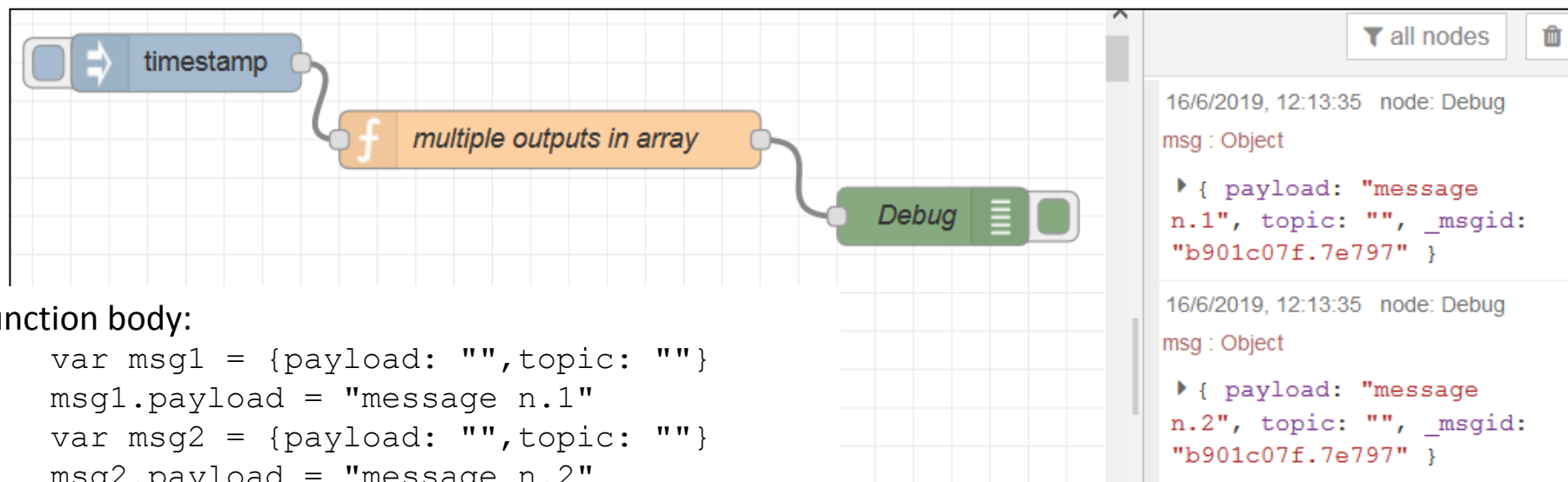
Function body:

```
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":""},{id":"dc28df5c.459198","type":"function","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\"\\n\\nreturn
[msg1,msg2];","outputs":2,"noerr":0,"x":280,"y":80,"wires":[["31f9f142.ac463e"],["1af074e3.f0c223"]]},{"id":"8dc40387.f59c88","type":"in
ject","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":[]}]
  
```

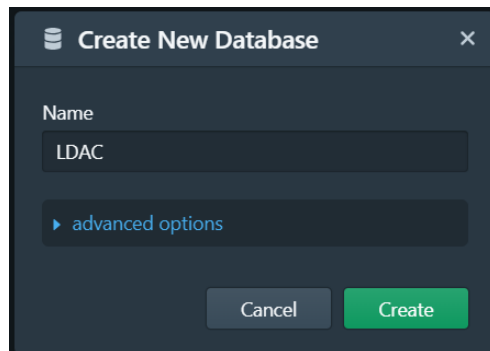
Node-RED can be employed to create flows involving the query and update of triple stores (RDF stores) while integrating other functionalities.

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

SETUP

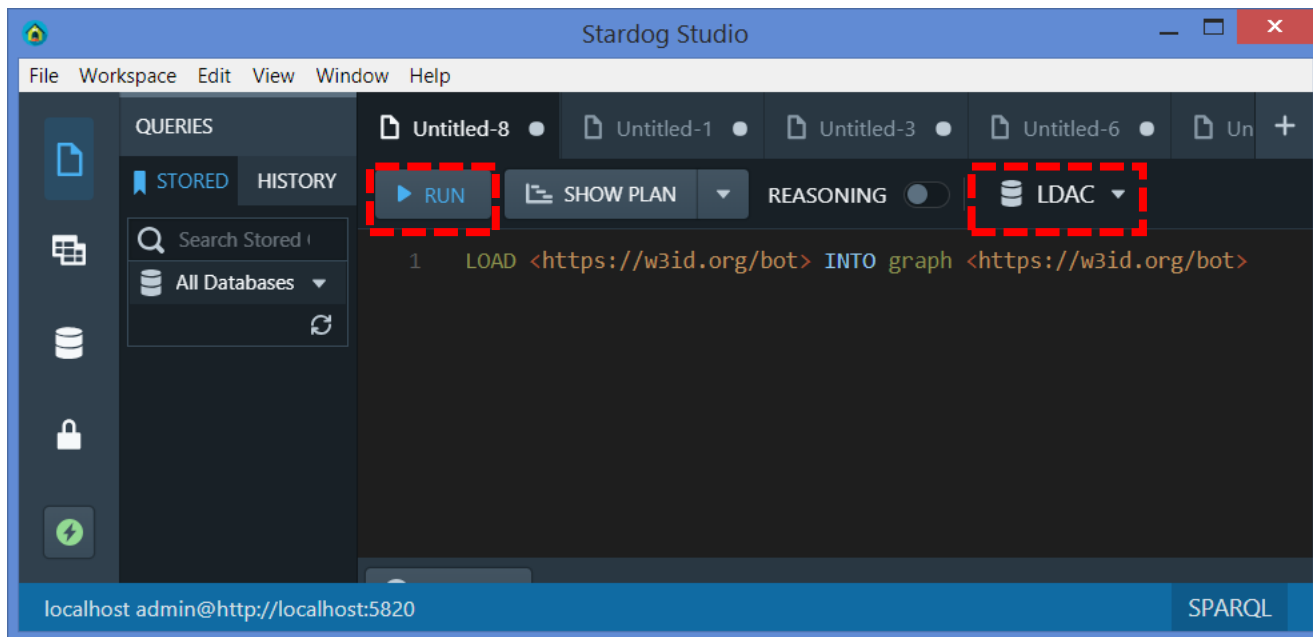
- Stardog is employed as triple store
- The examples refers to a Stardog installation available in localhost
- The examples are based on the BOT ontology (<https://github.com/w3c-lbd-cg/bot>) that must loaded on Stardog database "LDAC"

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>

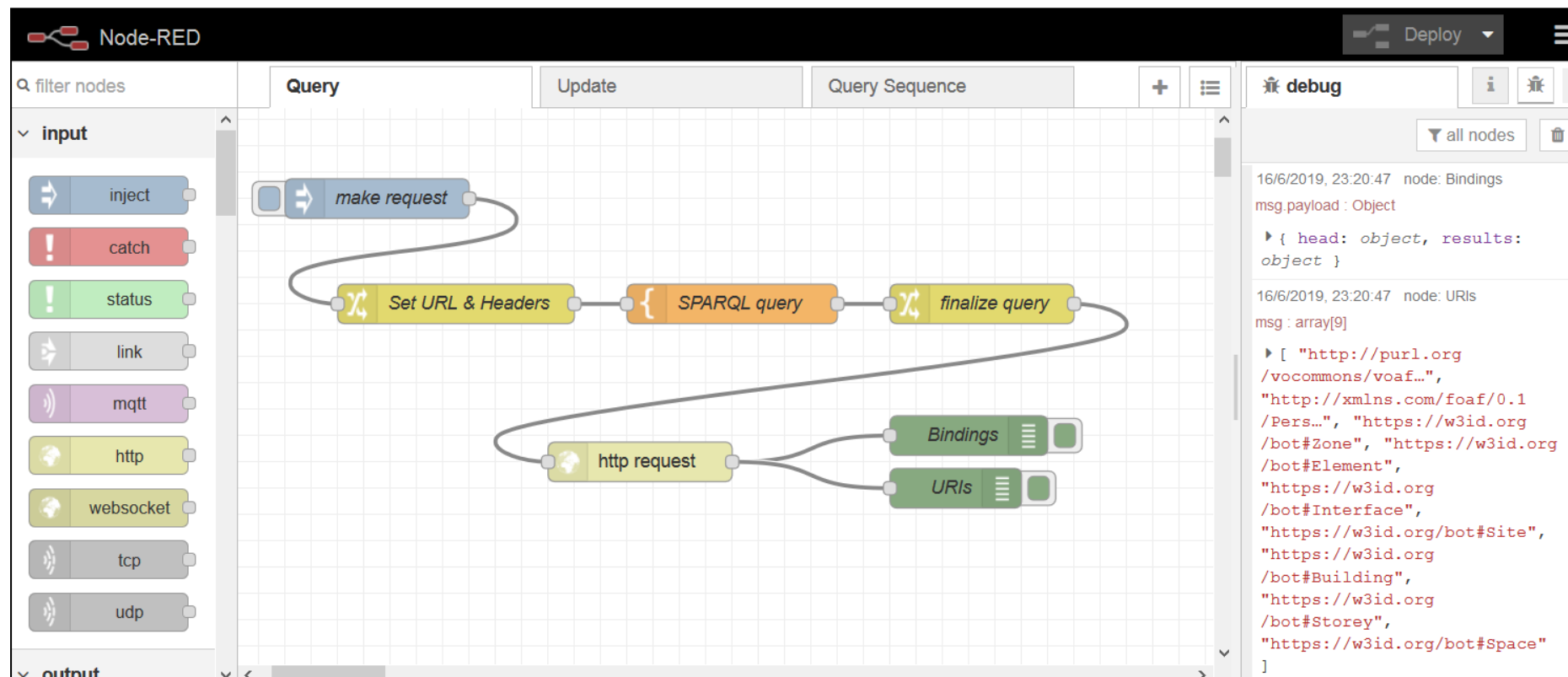
Headers

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 .
}
```



Default Inject node

The screenshot displays the Node-RED web interface. On the left, a workflow is visible on a grid background. It starts with a 'make request' node (highlighted with a red dashed box), followed by a 'Set URL & Headers' node, then a 'SPARQL query' node, and finally an 'http request' node. The right panel is titled 'Edit inject node' and contains the following configuration options:

- Delete** button
- Cancel** button
- Done** button
- Properties** section with icons for settings, documentation, and help.
- Payload**: A dropdown menu currently set to 'timestamp'.
- Topic**: An empty text input field.
- Inject once after**: A checkbox that is unchecked, followed by a text input '0.1' and the text 'seconds, then'.
- Repeat**: A dropdown menu currently set to 'none'.
- Name**: A text input field containing 'make request'.
- Note**: A yellow box containing the 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 web interface. On the left, a workflow is visible on a grid. It starts with a 'make request' node, followed by a 'Set URL & Headers' node (highlighted with a red dashed box), and then an 'http request' node. The 'Set URL & Headers' node is connected to the 'http request' node. On the right, the 'Edit change node' configuration panel is open. It has a 'Delete' button, 'Cancel', and 'Done' buttons. The 'Properties' section shows the node name 'Set URL & Headers'. The 'Rules' section, also highlighted with a red dashed box, contains three rules for setting message properties:

- 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`

At the bottom of the Rules section is a '+ add' button.

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

The screenshot shows the Node-RED web interface. On the left, a workflow is visible with three nodes: 'make request' (blue), 'Set URL & Headers' (yellow), and 'http request' (yellow). A red dashed box highlights a 'Template node' (orange) labeled 'SPARQL query' connected between 'Set URL & Headers' and 'http request'. On the right, the 'Edit template node' dialog is open. It has a 'Delete' button, 'Cancel' button, and a red 'Done' button. The 'Properties' section shows: 'Name' as 'SPARQL query', 'Property' as 'msg. payload' (highlighted with a red dashed box), and 'Format' as 'Plain text'. The 'Template' section shows a 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 'Syntax Highlight' dropdown is set to 'none'. At the bottom, 'Output as' is 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 in the 'Query' tab. It consists of a 'headers' node connected to a 'SPARQL query' node, which is then connected to a 'finalize query' node. The 'finalize query' node is highlighted with a red dashed box. Below this, an 'http request' node is connected to 'Bindings' and 'URIs' nodes. On the right, the 'Edit change node' configuration panel is open. It shows the node's name as 'finalize query'. Under the 'Rules' section, a rule is configured: 'Set' the 'msg. payload' to the JSONata expression '"query=" & payload'. This rule is also highlighted with a red dashed box.

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 shows the Node-RED web interface. On the left, a workflow is visible in the 'Query' workspace. It starts with a 'msg.url' input, followed by a 'URL & Headers' node, then a 'SPARQL query' node, and finally a 'finalize query' node. The output of 'finalize query' is connected to an 'http request' node, which is highlighted with a red dashed box. The 'http request' node's output is split into two paths: one leading to a 'Bindings' node and another to a 'URIs' node. On the right, the 'Edit http request node' dialog is open. It has tabs for 'Delete', 'Cancel', and 'Done'. The 'Properties' section is expanded. The 'Method' is set to 'POST' (highlighted with a red dashed box). The 'URL' is 'http://'. The 'Enable secure (SSL/TLS) connection' checkbox is unchecked. The 'Use authentication' checkbox is checked. Under 'Use authentication', the 'Type' is 'basic authentication', the 'Username' is 'admin', and the 'Password' is masked with dots. The 'Use proxy' checkbox is unchecked. The 'Return' type is set to 'a parsed JSON object' (highlighted with a red dashed box). The 'Name' is 'Name'.

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

The screenshot shows the Node-RED web interface. On the left, a workflow is visible: an 'http request' node connects to a 'Bindings' node and a 'URIs' node. The 'Bindings' node is highlighted with a red dashed box. Above it, a 'SPARQL query' node connects to a 'finalize query' node. The 'Edit debug node' panel on the right shows the 'Properties' tab. The 'Output' field is set to 'msg. payload' (highlighted with a red dashed box). The 'To' section has 'debug window' checked. 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 on a grid background. It starts with an 'http request' node, which connects to two 'debug' nodes labeled 'Bindings' and 'URIs'. The 'URIs' node is highlighted with a red dashed border. Above this, a 'SPARQL query' node connects to a 'finalize query' node, which then connects to the 'Bindings' node. On the right, the 'Edit debug node' panel is open. It features a 'Delete' button, 'Cancel', and 'Done' buttons. Below these is the 'Properties' section, which includes an 'Output' dropdown menu. The selected output path is 'J: payload.results.bindings.botclass.value', which is also highlighted with a red dashed border. Under the 'To' section, the 'debug window' checkbox is checked, while 'system console' and 'node status (32 characters)' are unchecked. At the bottom, 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\nFROM <https://w3id.org/bot>\nwhere { \n  ?botclass rdf:type owl:Class .\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":"\"query=\" & 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)

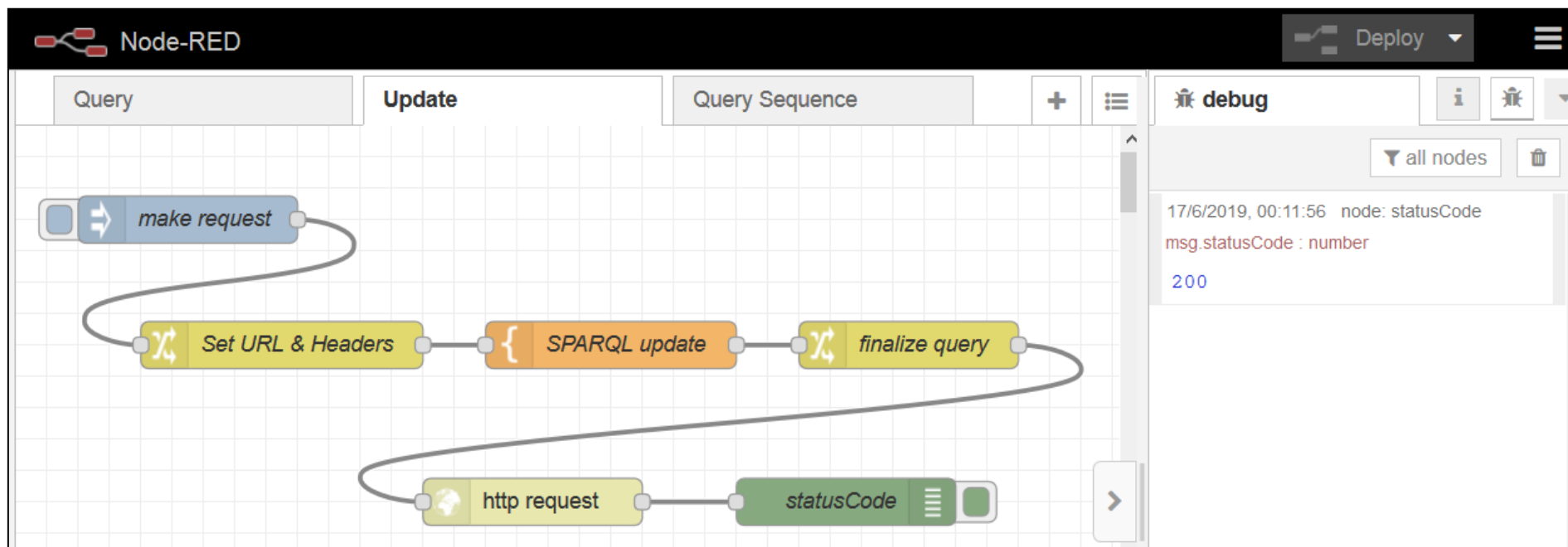
<http://localhost:5820/LDAC/update>

Headers

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

Add an individual of class bot:Zone in a new graph

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



Default Inject node

The screenshot shows the Node-RED web interface. On the left, a flow is visible with three nodes: an inject node (labeled 'make request'), a 'Set URL & Headers' node, and a 'SPARQL update' node. The inject node is highlighted with a red dashed box. Below these, there is an 'http request' node. On the right, the 'Edit inject node' panel is open. It contains the following fields:

- Delete** button
- Cancel** button
- Done** button
- Properties** section with a gear icon, a document icon, and a refresh icon.
- Payload**: A dropdown menu showing 'timestamp'.
- Topic**: A text input field.
- Inject once after**: A checkbox (unchecked) followed by a text input '0.1' and the text 'seconds, then'.
- Repeat**: A dropdown menu showing 'none'.
- Name**: A text input field containing 'make request'.
- Note**: A yellow box containing the 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 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. It includes a 'Delete' button, 'Cancel', and 'Done' buttons. The 'Properties' section shows the node name 'Set URL & Headers'. The 'Rules' section, also highlighted with a red dashed box, contains two configuration rules:

- Rule 1: Set msg. url to http://localhost:5820/LDAC/update
- Rule 2: Set msg. headers['content-type'] to application/x-www-form-urlencoded

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

The screenshot displays the Node-RED web interface. On the left, a workflow is visible in the 'Update' tab, featuring a 'Headers' node, a 'SPARQL update' node (highlighted with a red dashed box), and a 'finalize' node. Below this, an 'http request' node is connected to a 'statusCode' node. On the right, the 'Edit template node' dialog is open for the 'SPARQL update' node. The dialog includes a 'Delete' button, 'Cancel', and 'Done' buttons. The 'Properties' section shows the node's name as 'SPARQL update'. The 'Property' dropdown is set to 'msg.payload' (highlighted with a red dashed box). The 'Format' dropdown is set to 'Plain text'. The 'Template' section (highlighted with a red dashed box) contains a SPARQL update query:

```
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     ex:NewZone rdf:type owl:NamedIndividual , bot:Zone.
7   }
8 }
9
```

 The 'Syntax Highlight' dropdown is set to 'none'. At the bottom, the 'Output as' dropdown is set to '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 displays the Node-RED web interface. On the left, a workflow is visible on a grid. It starts with a 'SPARQL update' node (orange) connected to a 'finalize query' node (yellow). The 'finalize query' node is highlighted with a red dashed border. Below this, an 'http request' node (green) is connected to a 'statusCode' node (green). On the right, the 'Edit change node' configuration panel is open. It has tabs for 'Delete', 'Cancel', and 'Done'. The 'Properties' section shows the node name 'finalize query'. The 'Rules' section, also highlighted with a red dashed border, contains a 'Set' rule. The rule is configured to set 'msg. payload' to the JSONata expression '"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. It features a 'Delete' button, 'Cancel', and 'Done' buttons. The 'Properties' section is expanded, showing the following settings:

- Method:** POST (highlighted with a red dashed box)
- URL:** http://
- ☐ Enable secure (SSL/TLS) connection
- ☒ Use authentication (highlighted with a red dashed box)
 - Type:** basic authentication
 - Username:** admin
 - Password:** (masked with dots)
- ☐ 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 on a grid background. It starts with a 'Query' tab, followed by a 'SPARQL update' node, then a 'finalize query' node. A line connects the 'finalize query' node to an 'http request' node. The 'http request' node is connected to a 'statusCode' node, which is highlighted with a red dashed border. On the right, the 'Edit debug node' panel is open. It has a 'Delete' button, 'Cancel', and 'Done' buttons. Under the 'Properties' section, the 'Output' dropdown is set to 'msg. statusCode' (highlighted with a red dashed border). 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:

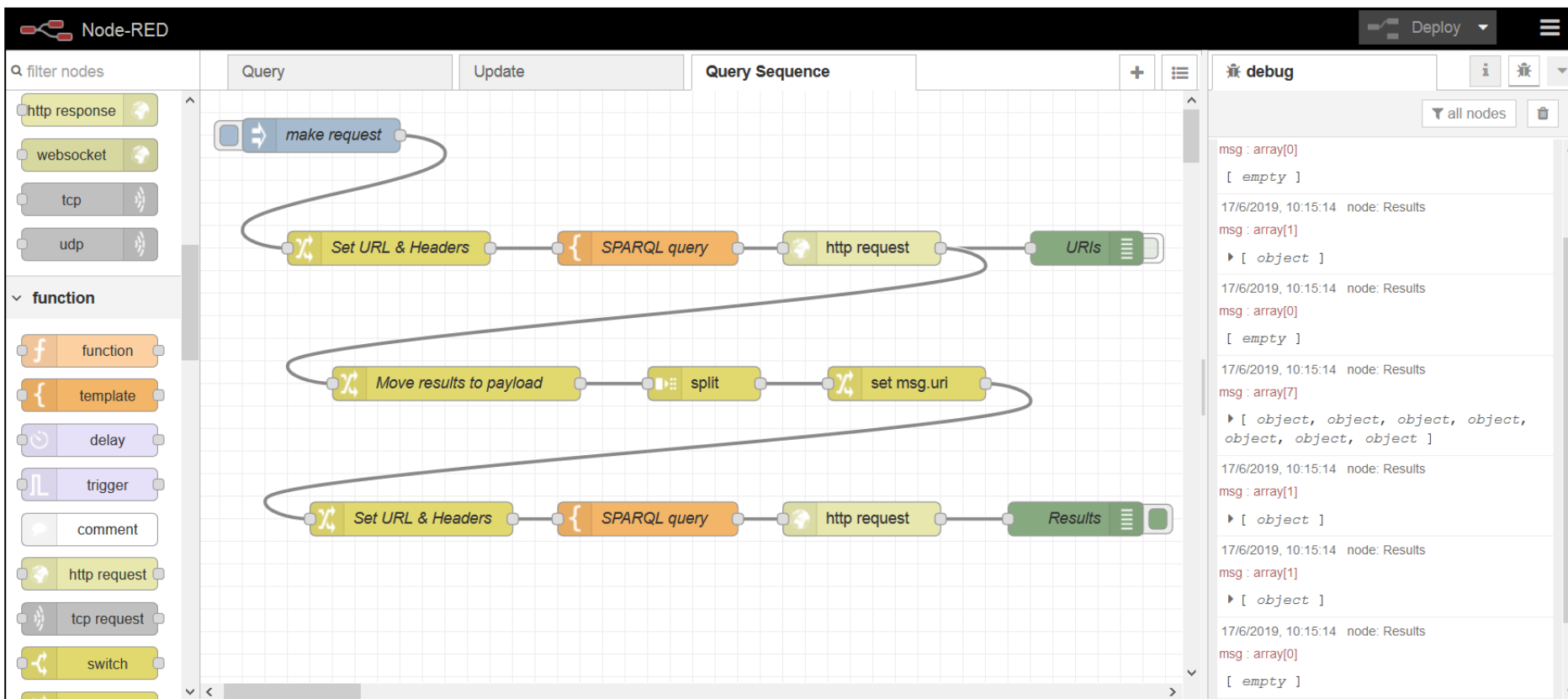
```
msg.statusCode : number
200
```

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  \t ex: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.




```
{
  "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\n<https://w3id.org/bot>\nwhere {\n  ?botclass rdf:type owl:Class.\n}\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,
  "wires": [
    [
      "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",
  "type": "debug",
  "z": "59bfaf1b.8937e",
  "name": "URIs",
  "active": false,
  "tosidebar": true,
  "console": false,
  "tostatus": false,
  "complete": "payload.results.bindings.botclass.value",
  "targetType": "jsonata",
  "x": 810,
  "y": 140,
  "wires": [
    [
      "9f1ce207.58f23"
    ]
  ],
  "id": "9f1ce207.58f23",
  "type": "split",
  "z": "59bfaf1b.8937e",
  "name": "",
  "spl": "\n",
  "splType": "str",
  "arraySpl": 1,
  "arraySplType": "len",
  "stream": false,
  "addname": "",
  "x": 470,
  "y": 260,
  "wires": [
    [
      "585fc625.2b85b"
    ]
  ],
  "id": "90c2c2e6.1e2ae",
  "type": "change",
  "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": true,
  "tosidebar": true,
  "console": false,
  "tostatus": false,
  "complete": "payload.results.bindings",
  "targetType": "jsonata",
  "x": 800,
  "y": 380,
  "wires": [
    [
      "73ff6404.25f12c"
    ]
  ],
  "id": "73ff6404.25f12c",
  "type": "template",
  "z": "59bfaf1b.8937e",
  "name": "SPARQL query",
  "field": "payload",
  "fieldType": "msg",
  "format": "handlebars",
  "syntax": "mustache",
  "template": "query=\nselect distinct ?botclass ?prop\nFROM\n<https://w3id.org/bot>\nwhere {\n  VALUES ?botclass {<{{uri}}>\n  ?prop rdfs:domain ?botclass.\n}\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,
  "wires": [
    [
      "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",
  "type": "change",
  "z": "59bfaf1b.8937e",
  "name": "",
  "rules": [
    {
      "t": "set",
      "p": "uri",
      "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



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

That's all folks!

walter.terkaj@stiima.cnr.it