TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA ĐIỆN – ĐIỆN TỬ

BỘ MÔN ĐIỆN TỬ

----------o0o----------

RED-STACK USER GUIDE

GVHD: Bùi Quốc Bảo

SVTH: Nguyễn Hoàng Phụng

MSSV: 1710246

Tp Hồ Chí Minh, ngày 28 tháng 11 năm 2019

MỤC LỤC

LỜI NÓI ĐẦU

IOT - Internet Of Things là một kịch bản mà ở đó vạn vật được kết nối với nhau, mỗi thiết bị trong mạng có một định danh riêng để có thể quản lý truy cập và điều khiển. Trong những năm gần đây với sự phát triển của công nghệ 5G thì kéo theo đó là sự phát triển của các thiết bị có kết nối internet, điều đó đồng nghĩa với việc phát triển của IOT. Vì thế ngoài việc các phần cứng của các thiết bị có kết nối internet phát triển thì việc phát triển phần mềm có thể sử dụng để hỗ trợ lập trình cũng là rất cần thiết.

Tài liệu này được viết như một tài liệu đính kèm trong báo cáo đồ án môn học của em. Phần core của phần mềm được viết và học hỏi từ một phần mềm đã rất phổ biến hiện nay là node - RED. Do kiến thức còn hạn chế nên việc viết toàn bộ một phần mềm được nhiều lập trình viên thực hiện gần như là điều khó khăn, vì thế nên đã tham khảo cách viết từ họ. Tuy nhiên, việc viết ra phần mềm còn có những thiếu sót trong việc lập trình cũng như xây dựng cấu trúc nên mong bạn đọc có thể bỏ qua hoặc góp ý phát triển phần mềm này.

Github: https://github.com/Creator250/red-stack-v7.0.1

Cuối cùng xin gửi lời cảm ơn đến thầy Bùi Quốc Bảo, đã dẫn dắn và hướng dẫn em thực hiện đồ án này. Bên cạnh đó đã dẫn dắt đưa ra ý tưởng cho việc thực hiện phần mềm này.

Tp. Hồ Chí Minh, ngày 28 tháng 11 năm 2019.

Sinh viên

Nguyễn Hoàng Phụng

CHAPTER 1: GETTING STARTED

I. Install program on local host:

1. Download Source from github:
   https://github.com/Creator250/red-stack-v7.0.1
2. Open folder red-stack-master-v7.0.1
3. Open terminal at here after that type in terminal:
   + npm install
   + npm start
   + It's running on localhost port 1999
4. If you want to draw graph: Please take folder .redstack and find folder have same name in your root folder and replace that.

II. Install program on Android:

This program is developing on Android. It will be run on the future.

CHAPTER 2: CREATING YOUR FIRST FLOW

Overview

This tutorial introduces the RED-Stack editor and creates a flow the demonstrates the Inject, Debug and Function nodes.

I. Access the editor

With RED-Stack running, open the editor in a web browser.

If you are using a browser on the same computer that is running RED-Stack , you can access it with the url: http://localhost:1999

If you are using a browser on another computer, you will need to use the ip address of the computer running RED-Stack: http://<ip-address>:1999.

II. Add an Inject node

The Inject node allows you to inject messages into a flow, either by clicking the button on the node, or setting a time interval between injects.

Drag one onto the workspace from the palette.

Select the newly added Inject node to see information about its properties and a description of what it does in the Information sidebar pane.

III. Add a Debug node

The Debug node causes any message to be displayed in the Debug sidebar. By default, it just displays the payload of the message, but it is possible to display the entire message object.

IV. Wire the two together

Connect the Inject and Debug nodes together by dragging between the output port of one to the input port of the other.

V. Deploy

At this point, the nodes only exist in the editor and must be deployed to the server.

Click the Deploy button.

With the Debug sidebar tab selected, click the Inject button.

VI. Add a Function node

The Function node allows you to pass each message though a JavaScript function.

Delete the existing wire (select it and press delete on the keyboard).

Wire a Function node in between the Inject and Debug nodes.

Double-click on the Function node to bring up the edit dialog. Copy the following code into the function field:

```
// Create a Date object from the payload

var date = new Date(msg.payload);

// Change the payload to be a formatted Date string

msg.payload = date.toString();

// Return the message so it can be sent on

return msg;
```

Click Done to close the edit dialog and then click the deploy button.

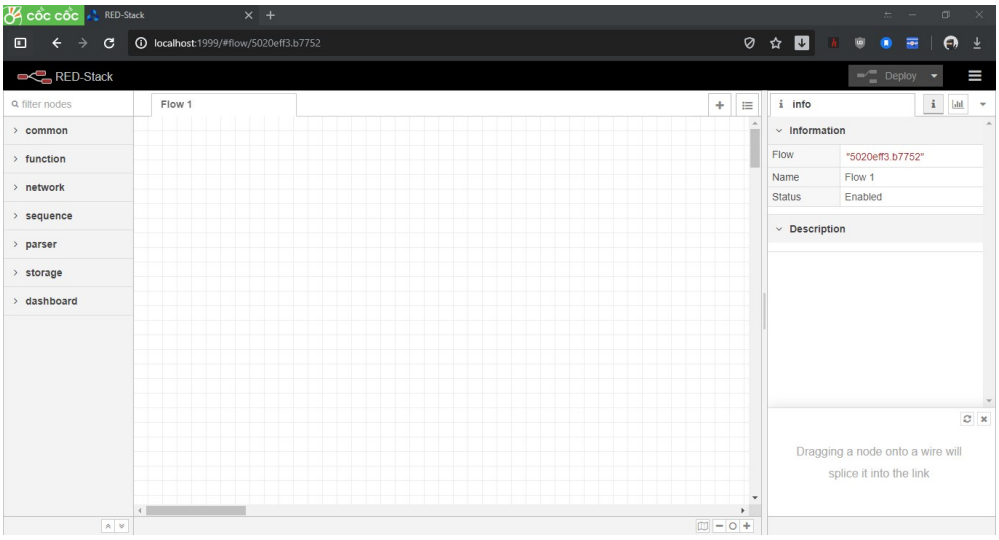
Now when you click the Inject button, the messages in the sidebar will now be formatted is readable timestamps.

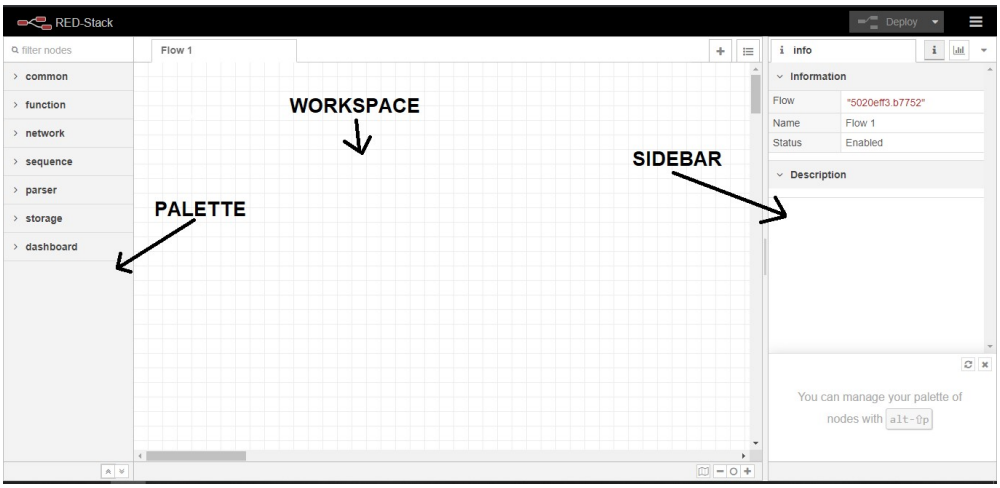
CHAPTER 3: EDITOR USER GUIDE

The editor window consists of four components:

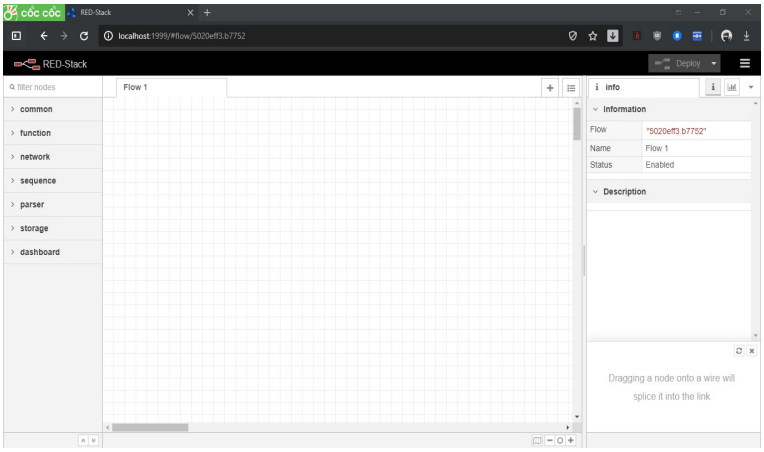
- The header at the top, containing the deploy button, main menu, and, if user authentication is enabled, the user menu.
- The palette on the left, containing the nodes available to use.
- The main workspace in the middle, where flows are created.
- The sidebar on the right.

Follow the links above to learn more about each component.





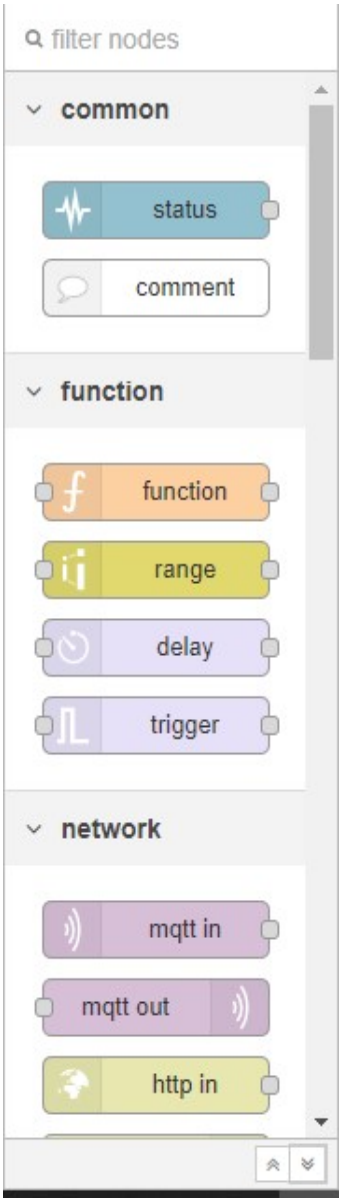
WORKSPACE



The main workspace is where flows are developed by dragging nodes from the palette and wiring them together.

The workspace has a row of tabs along the top; one for each flow and any subflows that have been opened.

PALETTE



The palette contains all of the nodes that are installed and available to use.

They are organised into a number of categories, with inputs, outputs and functions at the top. If there are any subflows, they appear in a category at the top of the palette.

Categories can be expanded or collapsed by clicking its header.

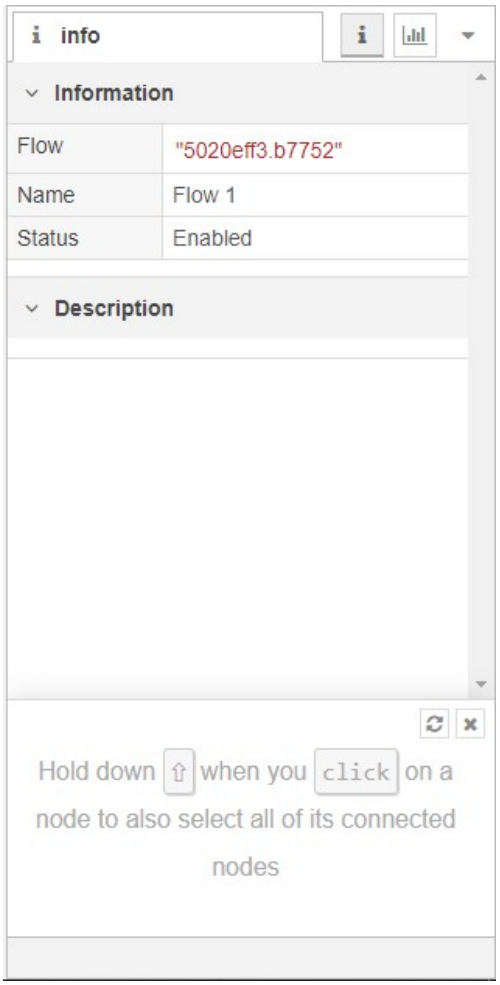
The and buttons at the bottom of the palette can be used to collapse or expand all categories.

Above the palette is an input that can be used to filter the list of nodes.

Palette toggle

The entire palette can be hidden by clicking the palette toggle that is shown when the mouse is over the palette.

SIDEBAR



The sidebar contains panels that provide a number of useful tools within the editor.

- Information - view information about nodes and their help
- Debug - view messages passed to Debug nodes
- Configuration Nodes - manage configuration nodes
- Context data - view the contents of context

Some nodes contribute their own sidebar panels, such as red-stack-dashboard.

The panels are opened by clicking their icon in the header of the sidebar, or by selecting them in the drop-down list shown by clicking the button.

The sidebar can be resized by dragging its edge across the workspace.

If the edge is dragged close to the right-hand edge, the sidebar will be hidden. It can be shown again by selecting the ‘Show sidebar’ option in the View menu, or using the Ctrl/Space shortcut.

CHAPTER 4: CONFIGURE RED-STACK

Settings file

- You can configure RED-Stack using the settings file.
- **So, Where is my settings file?**

When the RED-Stack started, it looks for a file called settings.js in your folder ~/.red-stack and in the link folder: red-stack-7.0.1/red-stack-master-v7.0.1/packages/node_modules/red-stack

P/s: if you are using windows, Please replace '/' by '\'.

If it does not find one there, it will copy in a default settings file to that directory and use it.

Alternatively, the --settings command-line argument can be used when starting RED-Stack to point at a different file.

If you have not yet run RED-Stack and want to edit the settings file, you can copy the default settings file in manually from here.

Editing the settings file

The settings file is loaded into the runtime as a Node.js module that exports a JavaScript object of key/value pairs.

The default settings file comes with many options commented out. For example, the option to format your flow file to make it easier to read:

```
//flowFilePretty: true,
```

To enable that option, remove the // at the start of the line.

If you add a new option to the file, be sure to add a comma to separate it from any options before or after it.

If there is a syntax error in the file, RED-Stack will not be able to start. The log output will indicate where the error is.

Configuration

The following properties can be used to configure RED-Stack.

When running as a normal application, it loads its configuration from a settings file. For more information about the settings file and where it is read this guide.

Runtime Configuration

flowFile

The file used to store the flows. Default: flows_<hostname>.json

userDir

The directory to store all user data, such as flow and credential files and all library data. Default: $HOME/.red-stack

nodesDir

A directory to search for additional installed nodes. RED-stack searches the nodes directory under the userDir directory. This property allows an additional directory to be searched, so that nodes can be installed outside of the RED-Stack

Install structure. Default: $HOME/.red-stack/nodes

uiHost

The interface to listen for connection on. Default: 0.0.0.0 = all IPv4 interfaces.

Standalone only.

uiPort

The port used to serve the editor UI. Default: 1999.

Standalone only.

httpAdminRoot

The root url for the editor UI. If set to false, all admin endpoints are disabled. This includes both API endpoints and the editor UI. To disable just the editor UI, see the disableEditor property below. Default: `/` .

httpAdminAuth

Deprecated: see adminAuth.

enables HTTP Basic Authentication on the editor UI:

```
httpAdminAuth: {user:"nol", pass:"5f4dcc3b5aa765d61d8327deb882cf99"}
```

The pass property is the md5 hash of the actual password. The following command can be used to generate the hash:

```
node -e "console.log(require('crypto').createHash('md5').update('YOUR PASSWORD
 HERE','utf8').digest('hex'))"
```

Standalone only.

httpNodeRoot

the root url for nodes that provide HTTP endpoints. If set to false, all node-based HTTP endpoints are disabled. Default: `/`

httpNodeAuth

enables HTTP Basic Authentication. See httpAdminAuth for format.

httpRoot

this sets the root url for both admin and node endpoints. It overrides the values set by httpAdminRoot and httpNodeRoot.

https

enables https, with the specified options object, as defined here.

Standalone only.

disableEditor

if set to true, prevents the editor UI from being served by the runtime. The admin api endpoints remain active. Default: false.

httpStatic

a local directory from which to serve static web content from. This content is served from the top level url, /. When this property is used, httpAdminRoot must also be used to make editor UI available at a path other than /.

Standalone only.

httpStaticAuth

enabled HTTP Basic Authentication on the static content. See httpAdminAuth for format.

httpNodeCors

enables cross-origin resource sharing for the nodes that provide HTTP endpoints.

httpNodeMiddleware

an HTTP middleware function that is added to all HTTP In nodes. This allows whatever custom processing, such as authentication, is needed for the nodes. The format of the middleware function is documented here.

```
httpNodeMiddleware: function(req,res,next) {
    // Perform any processing on the request.
    // Be sure to call next() if the request should be passed
    // to the relevant HTTP In node.
}
```

logging

currently only console logging is supported. Various levels of logging can be specified. Options are:

- **fatal** - only those errors which make the application unusable should be recorded
- **error** - record errors which are deemed fatal for a particular request + fatal errors
- **warn** - record problems which are non fatal + errors + fatal errors
- **info** - record information about the general running of the application + warn + error + fatal errors
- **debug** - record information which is more verbose than info + info + warn + error + fatal errors
- **trace** - record very detailed logging + debug + info + warn + error + fatal errors

The default level is info. For embedded devices with limited flash storage you may wish to set this to fatal to minimise writes to “disk”.

Editor Configuration

adminAuth

Enables user-level security in the editor and admin API.

palletteCategories

Defines the order of categories in the pallette. If a node's category is not in the list, the category will get added to the end of the pallette. If not set, the following default order is used:

```
['subflows', 'input', 'output', 'function', 'social', 'storage', 'analysis', 'advanced'],
```

Note: Until the user creates a subflow the subflow category will be empty and will not be visible in the pallette.

Editor Themes

The theme of the editor can be changed by using the following settings object. All parts are optional.

```
editorTheme: {
    page: {
        title: "RED-Stack",
        favicon: "/absolute/path/to/theme/icon",
        css: "/absolute/path/to/custom/css/file",
        scripts: [ "/absolute/path/to/custom/script/file", "/another/script/file"]
    },
    header: {
        title: "RED-Stack",
        image: "/absolute/path/to/header/image", // or null to remove image
        url: "http://nodered.org" // optional url to make the header text/image a link to this url
```

```
    },
    deployButton: {
        type:"simple",
        label:"Save",
        icon: "/absolute/path/to/deploy/button/image" // or null to remove ima
ge
    },
    menu: { // Hide unwanted menu items by id. see packages/node_modules/@red-
stack/editor-client/src/js/red.js:loadEditor for complete list
        "menu-item-import-library": false,
        "menu-item-export-library": false,
        "menu-item-keyboard-shortcuts": false,
        "menu-item-help": {
            label: "Alternative Help Link Text",
            url: "http://example.com"
        }
    },
    userMenu: false, // Hide the user-menu even if adminAuth is enabled
    login: {
        image: "/absolute/path/to/login/page/big/image" // a 256x256 image
    },
    logout: {
        redirect: "http://example.com"
    },
    palette: {
        editable: true, // Enable/disable the Palette Manager
        catalogues: [   // Alternative palette manager catalogues
            'https://catalogue.nodered.org/catalogue.json'
        ],
        theme: [ // Override node colours - rules test against category/type b
y RegExp.
```

```
            { category: ".*", type: ".*", color: "#f0f" }
        ]
    },
    projects: {
        enabled: false // Enable the projects feature
    }
},
```

Dashboard

ui

The home path for the RED-Stack-Dashboard add-on nodes can specified. This is relative to any already defined **httpNodeRoot**

ui : { path: “mydashboard” },

Node Configuration

Any node type can define its own settings to be provides in the file.

functionGlobalContext

Function Nodes - a collection of objects to attach to the global function context. For example,

```
functionGlobalContext: { osModule:require('os') }
```

can be accessed in a function node as:

```
var myos = global.get('osModule');
```

debugMaxlength

Debug Nodes - the maximum length, incharacter, of any message sent to the debug sibar tab. Default: 1000.

mqttReconnectTime

MQTT Nodes - if the connection is lost, how long to wait, in milliseconds, before attempting to reconnect. Default: 5000.

serialReconnectTime

Serial Nodes - how long to wait, in milliseconds, before attempting to reopen a serial port. Default: 5000.

socketReconnectTime

TCP Nodes - how long to wait, in milliseconds, before attempting to reconnect. Default: 10000.

socketTimeout

TCP Nodes - how long to wait, in milliseconds, before timing out a socket. Default: 120000.

CHAPTER 5: CREATING NODES

Creating Nodes:

The main way RED-Stack can be extended is to add new nodes into its palette.

The following sections exist and are largely complete:

- Creating your first node
- JacaScript File
- HTML File
- Node context
- Node properties
- Node appearance
- Node status

To do:

1. Library
2. Custom http endpoints

General guidance

There are some general principles to follow when creating new nodes. These reflect the approach taken by the core node and help provide a consistent user-experience.

The Node has writen should:

- **be well-defined in purpose.**

- **Be simple to use, regradless of the underlying functionality.**
- **Be forgiving in what types of message properties it accepts.**
- **Be consistent in what they send.**
- **Sit at the beginning, middle or end of a flow - not all at once.**
- **Catch errors.**

Creating your first node

Nodes get created when a flow is deployed, they may send and receive some messages while the flow is running and they get deleted when the next flow is deployed.

They consist of a pair of files:

+ a JavaScript file to defines what the node does,

+ an html file to defines the node's properties, edit dialog and help text

A package.json file is used to package it all together as an npm module:

- Creating a simple node
  - Package.json
  - lower-case.js
  - lower-case.html
- Testing your node in RED-Stack: it for tester, and we don't need care about this problem.

Creating a simple node

This example will show how to create a node that converts message payloads to all lower-case character.

Ensure you have the lastest version of NodeJS.

Create a directory where you will develop your code. Within that directory, create the following files:

- Package.json
- lower-case.js

- ✧ lower-case.html

Package.json

This is a standard file used by NodeJS modules to descirbe their contents.

To generate a standard package.json file you can use comand or powerShell(if you use window) **npm init.** This will ask a series of questions to help create the initial content for the file, using sensible defaults where it can. When prompted, give it the name red-stack-contrib-example-lower-case.

Once generated, you must added a red-stack section:

```
{
    "name" : "red-stack-contrib-example-lower-case",
    ...
    "red-stack" : {
        "nodes": {
            "lower-case": "lower-case.js"
        }
    }
}
```

This tell runtime what node files the module contains.

For more information about how to package your node, including requirements on naming and other properties that should be set before publishing your code, refer to the packaging guide.

P/s: Please do not publish this example node to npm!

lower-case.js

```
module.exports = function(RED) {
    function LowerCaseNode(config) {
        RED.nodes.createNode(this,config);
        var node = this;
        node.on('input', function(msg) {
            msg.payload = msg.payload.toLowerCase();
            node.send(msg);
        });
    }
    RED.nodes.registerType("lower-case",LowerCaseNode);
}
```

- The node is wrapped as a NodeJS module. The module exports a function that gets called when the runtime loads the node on start-up. The function is called with a single argument, RED, that provides the module access to the RED-Stack runtime api.
- The node itself is defined by a function, LowerCaseNode that gets called whenever a new instance of the node is created. It is passed an object containing the node-specific properties set in the flow editor.
- The function calls the RED.nodes.createNode function to initialise the features shared by all nodes. After that, the node-specific code lives.
- In this instance, the node registers a listener to the input event which gets called whenever a message arrives at the node. Within this listener, it changes the payload to lower case, then calls the send function to pass the message on in the flow.
- Finally, the LowerCaseNode function is registered with the runtime using the name for the node, lower-case.
- If the node has any external module dependencies, they must be included in the dependencies section of its package.json file.

For more information about the runtime part of the node, see JavaScript File guide.

Lower-case.html

```
<script type="text/javascript">
    RED.nodes.registerType('lower-case',{
        category: 'function',
        color: '#a6bbcf',
        defaults: {
            name: {value:""}
        },
        inputs:1,
        outputs:1,
        icon: "file.png",
        label: function() {
            return this.name||"lower-case";
        }
    });
</script>

<script type="text/x-red" data-template-name="lower-case">
    <div class="form-row">
        <label for="node-input-name"><i class="icon-tag"></i> Name</label>
        <input type="text" id="node-input-name" placeholder="Name">
    </div>
</script>

<script type="text/x-red" data-help-name="lower-case">
    <p>A simple node that converts the message payloads into all lower-case ch
aracters</p>
</script>
```

A node’s HTML file provides the following things:

- The main node definition that is registered with the editor
- The edit template
- Help text

In this ex, the node has a single editable property, name. While not required,

There is a widely used convention to this property to help distinguish between multiple instances of a node in a single flow.

More information about editor see HTML file guide.

JavaScript file

The node .js file defines the runtime behaviour of the node.

- Node constructor
- Receiving messages
  - ◊ Multiple outputs
  - ◊ Multiple messages
- Closing the node
  - ◊ Time out behavior
- Logging events
  - ◊ Handling errors
- Settings status
- Custom node settings

Node constructor

Nodes are defined by a contructor function that can be used to create new instances of the node. The function gets registered with the runtime so it can be called when nodes of the corresponding type are deployed in a flow.

The first thing it must do is call the RED.nodes.createNode function to initialise the features shared by all nodes. After that, the node-specific code lives.

```
function SampleNode(config) {
    RED.nodes.createNode(this,config);
    // node-specific code goes here

}

RED.nodes.registerType("sample",SampleNode);
```

Receiving messages

Nodes register a listener on the input event to receive messages from the up-stream nodes in a flow.

```
this.on('input', function(msg, send, done) {
    // do something with 'msg'

    // Once finished, call 'done'.
    // This call is wrapped in a check that 'done' exists
    if (done) {
        done();
    }
});
```

Handling errors

If the node encounters an error whilst handling the message, it should pass the details of the error to the done function.

This will trigger any Catch nodes present on the same tab, allowing the user to build flows to handle the error.

In the case does not provide the done function, it should be use node.error:

```
let node = this;
this.on('input', function(msg, send, done) {
    // do something with 'msg'


    // If an error is hit, report it to the runtime
    if (err) {
        if (done) {
            done(err);
        } else {
            node.error(err, msg);
        }
    }
});
```

Sending messages

If the nodes sits at the start of the flow and produces messages in response to external events, it should use the send function on the Node object:

```
var msg = { payload:"hi" }
this.send(msg);
```

If the node wants to send from inside the input event listener, in response to receiving a message, it should use the send function that is passed to the listener function:

```
let node = this;
this.on('input', function(msg, send, done) {
    // For maximum backwards compatibility, check that send exists.
    send = send || function() { node.send.apply(node,arguments) }

    msg.payload = "hi";
    send(msg);

    if (done) {
        done();
    }
});
```

If msg is null, no message is sent.

If the node is sending a message in response to having received one, it should reuse the received message rather than create a new message object. This ensures existing properties on the message are preserved for the rest of the flow.

Multiple outputs

If the node has more than one output, an array of messages can be passed to send, with each one being sent to the corresponding output.

```
this.send([ msg1 , msg2 ]);
```

Multiple messages

It’s possible to send multiple messages to a particular output by passing an array of messages within this array:

```
this.send([ [msgA1 , msgA2 , msgA3] , msg2 ]);
```

Closing the node

Whenever a new flow is deployed, the existing nodes are deleted. If any of them need to tidy up state when this happens, such as disconnecting from a remote system, they should register a listener on the close event.

```
this.on('close', function(removed, done) {
    if (removed) {
        // This node has been deleted
    } else {
        // This node is being restarted
    }
    done();
});
```

If the node needs to do any asynchronous work to complete the tidy up, the registered listener should accept an argument which is a function to be called when all the work is complete.

Logging events

If a node needs to lo something to the console, it can use on of the follow function:

```
this.log("Something happened");
this.warn("Something happened you should know about");
this.error("Oh no, something bad happened");
```

The warn and error messages also get sent to the flow editor debug tab.

Setting status

Whilst running, a node is able to share status information with the editor UI. This is done by calling the status function:

```
this.status({fill:"red",shape:"ring",text:"disconnected"});
```

The details of the status api can be found at Node status.

Custom node settings

A node may want to expose configuration options in a user's settings.js file.

The name of any setting must follow the following requirements:

- the name must be prefixed with the corresponding node type.
- the setting must use camel-case - see below for more information.
- the node must not require the user to have set it - it should have a sensible default.

For example, if the node type sample-node wanted to expose a setting called colour, the setting name should be sampleNodeColour.

Within the runtime, the node can then reference the setting as RED.setting.sampleNodeColour.

HTML File

The node .html file defines the how the node appears with the editor. It contains three distinct part, each wrapped in its own <script> tag:

1. the main node definition that is registered with the editor. This defines things such as the palette category, the editable properties (defaults) and what icon to use. It is within a regular javascript script tag
2. the edit template that defines the content of the edit dialog for the node. It is defined in a script of type text/x-red with data-template-name set to the type of the node.
3. the help text that gets displayed in the Info sidebar tab. It is defined in a script of type text/x-red with data-help-name set to the type of the node.

Defining a node

A node must be registered with the editor using the RED.nodes.registerType function.

This function takes two arguments; the type of the node and its definition:

```
<script type="text/javascript">

    RED.nodes.registerType('node-type',{

        // node definition

    });</script>
```

Node type

The node type is used throughout the editor to identify the node. It must match the value used by the call to RED.nodes.registerType in the corresponding .js file.

The type is also used as the label for the node in the palette. If the type ends with " in" or " out" this is stripped off label. For example, the "mqtt in" and "mqtt out" nodes are both labelled as "mqtt", with the display of the input and output ports providing the "in" or "out" context.

Node definition

The node definition contains all of the information about the node needed by the editor. It is an object with the following properties:

- category: (string) the palette category the node appears in
- defaults: (object) the editable properties for the node.
- credentials: (object) the credential properties for the node.
- inputs: (number) how many inputs the node has, either 0 or 1.
- outputs: (number) how many outputs the node has. Can be 0 or more.
- color: (string) the background colour to use.
- paletteLabel: (string|function) the label to use in the palette.
- label: (string|function) the label to use in the workspace.
- labelStyle: (string|function) the style to apply to the label.

- inputLabels: (string|function) optional label to add on hover to the input port of a node.
- outputLabels: (string|function) optional labels to add on hover to the output ports of a node.
- icon: (string) the icon to use.
- align: (string) the alignment of the icon and label.
- button: (object) adds a button to the edge of the node.
- oneditprepare: (function) called when the edit dialog is being built. See custom edit behaviour.
- oneditsave: (function) called when the edit dialog is okayed. See custom edit behaviour.
- oneditcancel: (function) called when the edit dialog is cancelled. See custom edit behaviour.
- oneditdelete: (function) called when the delete button in a configuration node's edit dialog is pressed. See custom edit behaviour.
- oneditresize: (function) called when the edit dialog is resized. See custom edit behaviour.
- onpaletteadd: (function) called when the node type is added to the palette.
- onpaletteremove: (function) called when the node type is removed from the palette.

Edit dialog

The edit template for a node describes the content of its edit dialog.

```
<script type="text/x-red" data-template-name="node-type">
    <div class="form-row">
        <label for="node-input-name"><i class="fa fa-tag"></i> Name</label>
        <input type="text" id="node-input-name" placeholder="Name">
    </div>
    <div class="form-tips"><b>Tip:</b> This is here to help.</div>
```

```
</script>
```

There are some simple conventions to follow:

- a with class form-row should be used to layout each row of the dialog.
- a typical row will have a <label> that contains an icon and the name of the property followed by an <input>. The icon is created using an <i> element with a class taken from those available from Font Awesome 4.7.
- if more interactivity is required, oneditprepare can be used to attach any event handlers on the dialog elements.

More information on how the edit template is used is available at Node properties.

Help text

When a node is selected, its help text is displayed in the info tab. This should provide a meaningful description of what the node does. It should identify what properties it sets on outgoing messages and what properties can be set on incoming messages.

The content of the first <p> tag is used as the tooltip when hovering over nodes in the palette.

```
<script type="text/x-red" data-help-name="node-type">
    <p>Some useful help text to introduce the node.</p>
    <h3>Outputs</h3>
        <dl class="message-properties">
        <dt>payload
            <span class="property-type">string | buffer</span>
        </dt>
    <h3>Details</h3>
    <p>Some more information about the node.</p>
 </script>
```

A complete style guide for node help is help style guide.

Node context

A node can store data within its context object.

For more information about context, read the Working with Context guide.

There are three scopes of context available to a node:

- Node - only visible to the node that set the value
- Flow - visible to all nodes on the same flow (or tab in the editor)
- Global - visible to all nodes

Unlike the Function node which provides predefined variables to access each of these contexts, a custom node must access these contexts for itself:

```
// Access the node's context object
var nodeContext = this.context();


var flowContext = this.context().flow;


var globalContext = this.context().global;
```

Each of these context objects has the same get/set functions described in the Writing Functions guide of Node Red because I write this program base on it.

Note: Configuration nodes that are used by and shared by other nodes are by default global, unless otherwise specified by the user of the node. As such it cannot be assumed that they have access to a Flow context.

Node properties

A node's properties are defined by the defaults object in its html definition. These are properties that get passed to the node constructor function when an instance of the node is created in the runtime.

In the example form the creating your first node selection, the node had single property called name. In this section, we'll add a new property called prefix to the node:

1. Add a new entry to the defaults object:

```
defaults: {
            name: {value:""},
            prefix: {value:""}
        },
```

The entry includes the default value to be used when a new node of this type is dragged onto the workspace.

2. Add an entry to the edit template for the node:

```
<div class="form-row">
    <label for="node-input-prefix"><i class="icon-tag"></i> Prefix</label>
    <input type="text" id="node-input-prefix">
</div>
```

The template should contain an <input> element with an id set to node-input-<propertyname>.

3. Use the property in the node

```
function LowerCaseNode(config) {
    RED.nodes.createNode(this,config);
```

```
    this.prefix = config.prefix;
    var node = this;
    this.on('input', function(msg) {
        msg.payload = node.prefix + msg.payload.toLowerCase();
        node.send(msg);
    });
}
```

Property definitions

The entries in the defaults object must be objects and can have the following attributes:

- value : (any type) the default value the property takes
- required : (boolean) optional whether the property is required. If set to true, the property will be invalid if its value is null or an empty string.
- validate : (function) optional a function that can be used to validate the value of the property.
- type : (string) optional if this property is a pointer to a configuration node, this identifies the type of the node.

Reserved property names

There are some reserved names for properties that must not be used. These are:

Type, x, y, z, wires, outputs

If a node wants to allow the number of outputs it provides to be configurable then outputs may be included in the default array. The function node is an example for how this works.

Property Validation

The editor attempts to validate all properties to warn the user if invalid values have been given.

The required attribute can be used to indicate a property must be non-null and non-blank.

If more specific validation is required, the validate attribute can be used to provide a function that will check the value is valid. The function is passed the value and should return either true or false. It is called within the context of the node which means this can be used to access other properties of the node. This allows the validation to depend on other property values. While editing a node the this object reflects the current configuration of the node and **not** the current form element value. The validator function should try to access the property configuration element and take the this object as fallback to achieve the right user experience.

There is a group of common validation functions provided.

- RED.validators.number() - check the value is a number
- RED.validators.regex(re) - check the value matches the provided regular expression

Both methods - required attribute and validate attribute - are reflected by the UI in the same way. The missing configuration marker on the node is triggered and the corresponding input is red surrounded when a value is not valid or missing.

The following example shows how each of these validators can be applied.

```
defaults: {
        minimumLength: { value:0, validate:RED.validators.number() },
        lowerCaseOnly: {value:"", validate:RED.validators.regex(/[a-z]+/) },
        custom: { value:"", validate:function(v) {
            var minimumLength=$("#node-input-minimumLength").length?$("#node-inp
ut-minimumLength").val():this.minimumLength;
```

```
                return v.length > minimumLength
            } }
},
```

Property edit dialog

When the edit dialog is opened, the editor populates the dialog with the edit template for the node.

For each of the properties in the defaults array, it looks for an <input> element with an id set to node-input-<propertyname>. This input is then automatically populated with the current value of the property. When the edit dialog is okayed, the property takes whatever value is in the input.

The <input> type can be either text for string/number properties, or checkbox for boolean properties. Alternatively, a <select> element can be used if there is a restricted set of choices.

Custom edit behaviour

The default behaviour works in many cases, but sometimes it is necessary to define some node-specific behaviour. For example, if a property cannot be properly edited as a simple <input> or <select>, or if the edit dialog content itself needs to have certain behaviours based on what options are selected.

A node definition can include two functions to customise the edit behaviour.

- oneditprepare is called immediately before the dialog is displayed.
- oneditsave is called when the edit dialog is okayed.
- oneditcancel is called when the edit dialog is cancelled.
- oneditdelete is called when the delete button in a configuration node's edit dialog is pressed.
- oneditresize is called when the edit dialog is resized.

For example, when the Inject node is configured to repeat, it stores the configuration as a cron-like string: `1,2 * * * *`. The node defines an `oneditprepare` function that can parse that string and present a more user-friendly UI. It also has an `oneditsave` function that compiles the options chosen by the user back into the corresponding cron string.

Node credentials

A node may define a number of properties as credentials. These are properties that are stored separately to the main flow file and do not get included when flows are exported from the editor.

To add credentials to a node, the following steps are taken:

1. Add a new credentials entry to the node's definition.

```
credentials: {
            username: {type:"text"},
            password: {type:"password"}
        },
```

The entries take a single option - their type which either text or password.

2. Add suitable entries to the edit template for the node.

```
<div class="form-row">
    <label for="node-input-username"><i class="icon-tag"></i> Username</label>
    <input type="text" id="node-input-username">
</div>
<div class="form-row">
```

```
    <label for="node-input-password"><i class="icon-tag"></i> Password</label>
    <input type="password" id="node-input-password">
</div>
```

3. In the node's .js file, the call to RED.nodes.registerType must be updated to include the credentials:

```
RED.nodes.registerType("my-node",MyNode,{
    credentials: {
        username: {type:"text"},
        password: {type:"password"}
    }
});
```

Accessing credentials

Runtime use of credentials

Within the runtime, a node can access its credentials using the credentials property:

```
function MyNode(config) {
    RED.nodes.createNode(this,config);
    var username = this.credentials.username;
    var password = this.credentials.password;
}
```

Credentials within the Editor

Within the editor, a node has restricted access to its credentials. Any that are of type `text` are available under the `credentials` property - just as they are in the runtime. But credentials of type `password` are not available. Instead, a corresponding boolean property called `has_<property-name>` is present to indicate whether the credential has a non-blank value assigned to it.

```
oneditprepare: function() {
            // this.credentials.username is set to the appropriate value
```

```
            // this.credentials.password is not set

            // this.credentials.has_password indicates if the property is pres
ent in the runtime

            ...

        }
```

Node appearance

There are three aspects of a node's appearance that can be customised; the icon, background colour and its label

Icon

The node's icon is specified by the icon property in its definition.

The value of the property can be either a string or a function.

If the value is a string, that is used as the icon.

If the value is a function, it will get evaluated when the node is first loaded, or after it has been edited. The function is expected to return the value to use as the icon.

The function will be called both for nodes in the workspace,
where this references a node instance, as well as for the node's entry in the palette. In this latter case, this will not refer to a particular node instance and the function must return a valid value.

```
...
icon: "file.png",
...
```

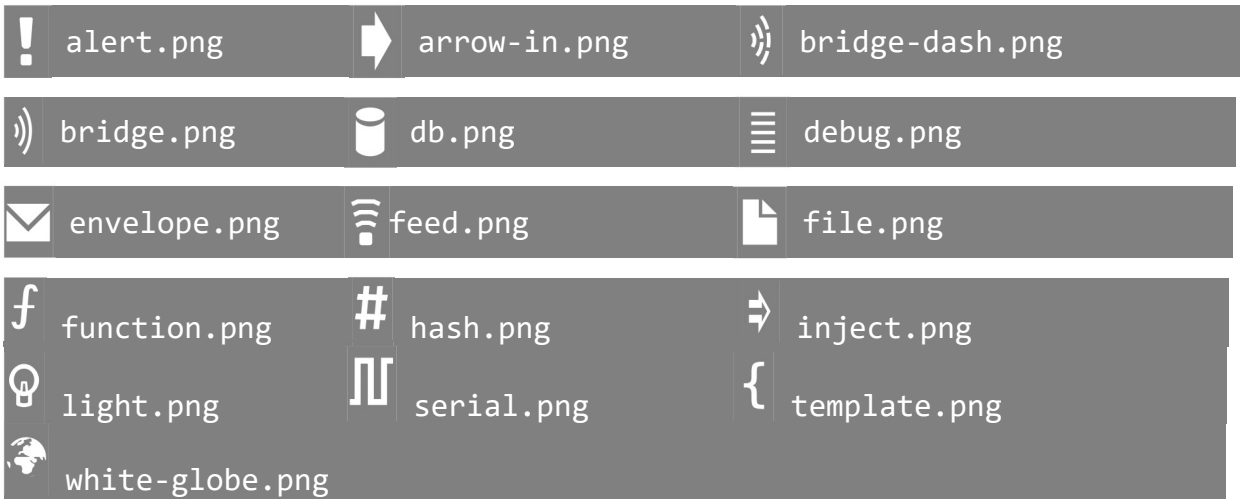
The icon can be either:

+ the name of stock icon provide by RED-Stack,

+ the name of a custom icon provided by the module,

+ a Font Awesome 4.7 icon

Stock icons



Custom icon

A node can procide its own icon in a direction called icons alongside its .js and .html files. These directories get added to the search path when the editor looks for a given icon filesname. Because of this the icon filename must be unique.

The icon should be white on a transparent background, 40 x 60 in size.

Font Awesome icon

Node-RED includes the full set of Font Awesome 4.7 icons.

To specify a FA icon, the property should take the form:

```
...
icon: "font-awesome/fa-automobile",
...
```

Background Colour

The node background colour is one of the main ways to quickly distinguish different node types. It is specified by the color property in the node definition.

```
...
color: "#a6bbcf",
...
```

- #3FADB5
- #87A980
- #A6BBCF
- #AAAA66
- #C0C0C0
- #C0DEED
- #C7E9C0
- #D7D7A0
- #D8BFD8
- #DAC4B4
- #DEB887
- #DEBD5C
- #E2D96E
- #E6E0F8
- #E7E7AE
- #E9967A
- #F3B567
- #FDD0A2
- #FDF0C2
- #FFAAAA
- #FFCC66
- #FFF0F0
- #FFFFFF

Labels

There are four label properties of a node; label, paletteLabel, outputLabel and inputLabel.

Node label

The label of a node in the workspace can either be a static piece of text, or it can be set dynamically on a per-node basis according to the nodes properties.

The value of the property can be either a string or a function.

If the value is a string, that is used as the label.

If the value is a function, it will get evaluated when the node is first loaded, or after it has been edited. The function is expected to return the value to use as the label.

As mentioned in a previous section, there is a convention for nodes to have a name property to help distinguish between them. The following example shows how the label can be set to pick up the value of this property or default to something sensible.

```
…
label: function() {
    return this.name||"lower-case";},
…
```

Note that it is not possible to use credential properties in the label function.

Palette label

By default, the node's type is used as its label within the palette.
The paletteLabel property can be used to override this.

As with label, this property can be either a string or a function. If it is a function, it is evaluated once when the node is added to the palette.

Label style

The css style of the label can also be set dynamically, using the labelStyle property. Currently, this property must identify the css class to apply. If not specified, it will use the default node_label class. The only other predefined class is node_label_italic.

The following example shows how labelStyle can be set to node_label_italic if the name property has been set:

```
...
labelStyle: function() {
    return this.name?"node_label_italic":"";
},
...
```

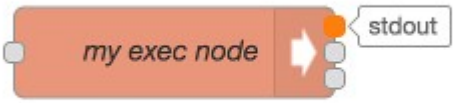
Alignment

By default, the icon and label are left-aligned in the node. For nodes that sit at the end of a flow, the convention is to right-align the content. This is done by setting the align property in the node definition to right:

```
...align: 'right',...
```

Port labels

Nodes can provide labels on their input and output ports that can be seen by hovering the mouse over the port.



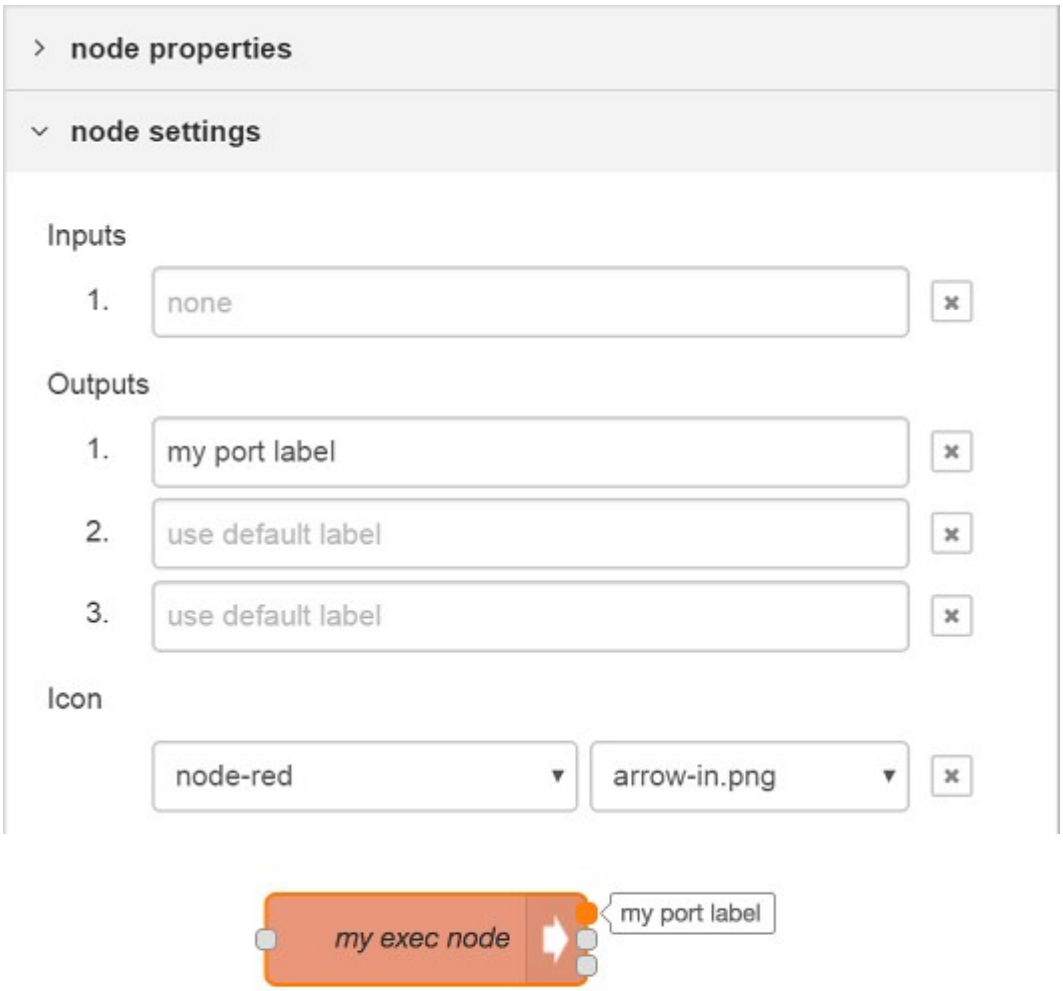
These can either be set statically by the node's html file

```
…
inputLabels: "parameter for input",outputLabels:
["stdout","stderr","rc"],
…
```

or generated by a function, that is passed an index to indicate the output pin (starting from 0).

```
...
outputLabels: function(index) {
    return "my port number "+index;}
...
```

In both cases they can be overwritten by the user using the node settings section of the configuration editor.



P/s: Labels are not generated dynamically, and cannot be set by msg properties.

Buttons

A node can have a button on its left or right hand edge, as seen with the core Inject and Debug nodes.

A key principle is the editor is not a dashboard for controlling your flows. So in general, nodes should not have buttons on them. The Inject and Debug nodes are special cases as the buttons play a role in the development of flows.

The button property in its definition is used to describe the behaviour of the button. It must provide, as a minimum, an onclick function that will be called when the button is clicked.

```
...button: {
    onclick: function() {
        // Called when the button is clicked
    }},...
```

The property can also define an enabled function to dynamically enable and disable the button based on the node's current configuration. Similarly, it can define a visible function to determine whether the button should be shown at all.

```
...button: {
    enabled: function() {
        // return whether or not the button is enabled, based on the
current
        // configuration of the node
        return !this.changed
    },
    visible: function() {
        // return whether or not the button is visible, based on the
current
         // configuration of the node
        return this.hasButton
    },
    onclick: function() { }},...
```

The button can also be configured as a toggle button - as seen with the Debug node. This is done by added a property called toggle that identifies a property in the node's defaults object that should be used to store a boolean value whose value is toggled whenever the button is pressed.

```
...
defaults: {
    ...
    buttonState: {value: true}
    ...},button: {
    toggle: "buttonState",
    onclick: function() { }}
...
```

Node status

Whilst running, a node is able to share status information with the editor UI. For example, the MQTT nodes can indicate if they are currently connected or not.

To set its current status, a node uses the `status` function. For example, the following two calls are used by the MQTT node to set the statuses seen in the image above:

```
this.status({fill:"red",shape:"ring",text:"disconnected"});

this.status({fill:"green",shape:"dot",text:"connected"});
```

Status object

A status object consists of three properties: `fill`, `shape` and `text`.

The first two define the appearance of the status icon and the third is an optional short piece of text (under <20 characters) to display alongside the icon.

The `shape` property can be: `ring` or `dot`.

The `fill` property can be: `red`, `green`, `yellow`, `blue` or `grey`

This allows for the following icons to be used:

If the status object is an empty object, `{}`, then the status entry is cleared from the node.

========== End of document ==========

References

[1] Because my program write based on Node-RED so you can learn more at : node RED documents.

[2] Node RED document : https://nodered.org/docs/.

[3] ZendVn page : https://zendvn.com/

[4] FreeCodeCamp page : https://www.freecodecamp.org/

[5] Github: https://github.com/node-red