

## Implementing an Inventory Service API – GET method

**Difficulty Level:** Hard

### Objective

Implement an inventory database service for the smart factory scenario; develop a Web API for reading the current inventory for any given part or product (e.g., “retrieve the number of part A units available in the warehouse”).

### Achievements

The skills to be acquired at the end of this module:

- Using *flow variables* in Node-RED to store values persistently across different nodes
- Learning to implement a *web service* with an *Application Programming Interface (API)*
- Implementing a *GET* method for external software modules to read the inventory states

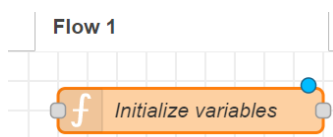
### 1. Initializing the inventory and stock values

In order to implement an “inventory service”, we first need a mechanism to keep track of the number of parts and products in the smart factory. For this purpose, we will use a built-in functionality of Node-RED to create a set of variables, called “**flow variables**”, allowing us to store and retrieve values across different nodes in the flow.

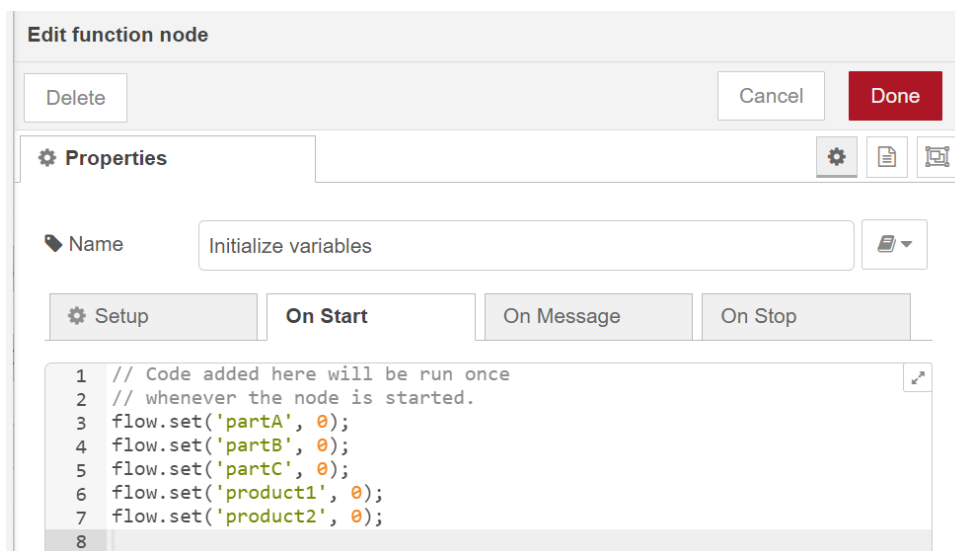
For the sake of simplicity, in this module we are not permanently storing the values in the file system or in an external database. Our application works fine with persistent inventory values as long as our Node-RED application is running. Every time the application redeployed/restarted, the inventory for the products and parts would reset to zero, but the rest of the functionality remains the same. For permanent storage of the inventory values in the file system, one can use the “*file*” nodes under “*storage*” group in Node-RED palette. There are also other ways to achieve such permanent storage, but we will leave this topic out of the scope of this module.

Let us start by creating and initializing the flow variables in a *function* node.

- Drag and drop a function node from Node-RED palette into the flow editor.
- Double click on the function node to edit its settings.
- Give this function node an intuitive name, such as “Initialize variables”



- On the “edit function node” window, note that there are multiple tabs: *Setup*, *On Start*, *On Message*, and *On Stop*.
- When you double click on a function node, Node-RED opens the “On Message” tab by default. This is because the code here is triggered whenever some data arrives to this function node, and this is the most common use case for function nodes.
- In our case, we would like to initialize some variables **only once** at the beginning of our application, rather than running the code at each message arrival. This is where the “On Start” tab of the function comes into play.
- So, switch to the “On Start” tab in the “Edit function node” window, and add the following code, as depicted in the figure below.



Let us understand what this code does:

- **flow.set('partA', 0)** on line 3 is telling Node-RED to set the value of a variable called “partA” to 0. If this variable does not exist, then NodeRED creates it.
- The “flow” object in this code indicates that we are creating the variable in the scope of our flow. That means, any node within the current flow (current tab in Node-RED) will have access to this variable, as we will see in the next sections.
- The same logic applies to the other lines of code: **Lines 3-5** create a separate variable **for each part** in our smart factory scenario to store their **inventory values**.
- Similarly, **lines 6-7** create a separate variable for **each product** in our smart factory scenario to store their **stock values**.

Since we will not process any messages in this function node, we do not need to add any other code in the “On Message” part or connect this function node to any other functions in the flow. Simply click on “**Done**” to save the settings after you add the code above under the “On Start” tab.

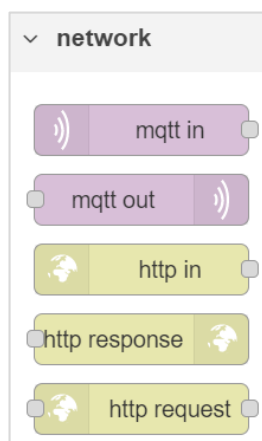
## 2. Implementing the Web Service and API for Product Stock

We will now create a **web service**, which allows other (possibly remote) software entities to interact with our application through an **Application Programming Interface (API)**.

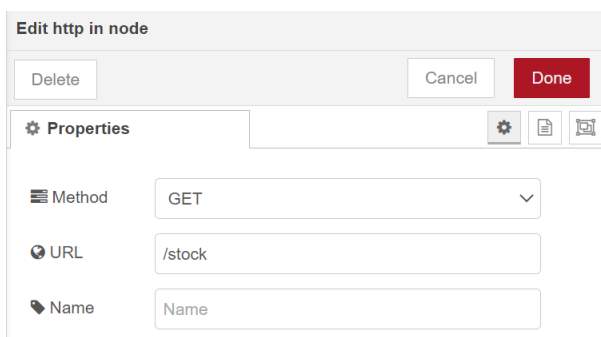
In this module, we will implement only the **GET** method part of this API, which allows another application to **read/retrieve** the inventory and stock values from our inventory tracking service. The next module will complement this API with a **POST** method in order to allow other software modules also to **change/update** the inventory and stock values.

In order to create a web service, our application should be able to handle **HTTP requests**.

- We can accomplish this easily by using the “*http in*” and “*http response*” nodes in the network group of Node-RED palette, as depicted below:



- Drag and drop an ***http in*** node to the flow editor.
- Edit the node settings as follows:

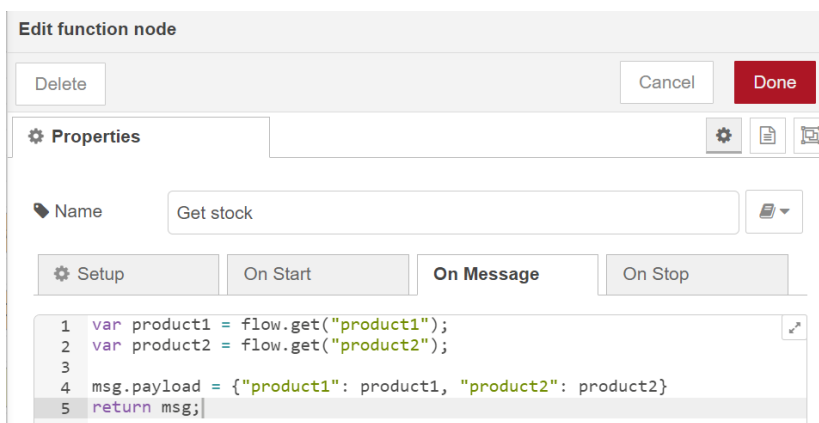


- With this, we are creating an “**API endpoint**” called “/stock” that uses the **GET** method.
- The URL field in the settings shown above is relative to the base URL, which is dictated by where our NodeRED application is running (by default localhost:1880). That means, by default our application will listen for **HTTP requests** at the URL “localhost:1880/stock”.

- In order to handle the incoming HTTP requests, connect a **function** node to the output of the *http in* node:



- Give an intuitive name to the function node (“Get stock”) and add the following code to its “On Message” section:



**Edit function node**

Delete Cancel Done

Properties

Name: Get stock

Setup On Start **On Message** On Stop

```

1 var product1 = flow.get("product1");
2 var product2 = flow.get("product2");
3
4 msg.payload = {"product1": product1, "product2": product2}
5 return msg;
  
```

- On lines 1-2 above, we are reading the flow variables we had created in the first step of this module and assigning them to the local (temporary) variables of the same name.
- On line 4, we are effectively creating the following **JSON formatted string** and assigning it to msg.payload;

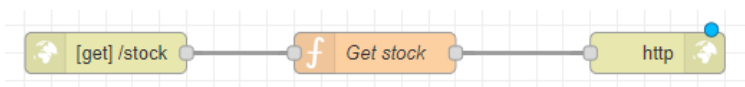
```

{
  "product1": <value of var product1>
  "product2": <value of var product2>
}
  
```

- On line 5, we are returning the msg object, which is sent out of the output connection of this function node.

With the function node above, we have prepared the response (a JSON object with the product stock values) and now we have to send back this response to the requesting client via http.

For this, we can simply use the “*http response*” node in Node-RED and connect it to the output of our function node (note that we do not need to change any setting in the *http response* node):



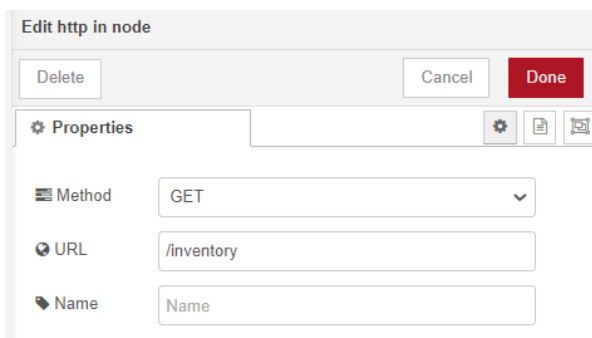
Once we deploy the flow in Node-RED, our API endpoint “/stock” should be ready and accepting incoming requests. We will test this in the next section.

### 3. Implementing the Web Service and API for Parts Inventory

We also need a similar GET method to read the inventory of each part in the warehouse. For this, we will implement another API endpoint called “/inventory” for our web service.

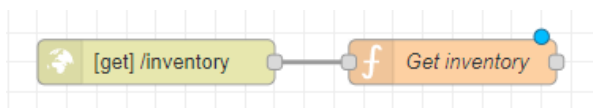
The process is quite similar to the instructions in the previous step.

- In order to create a new API endpoint, we will add a new **http in** node.
- Since this API endpoint is also for other entities to read/retrieve the value of our parts inventory, we use the **GET** method and set the URL for the API endpoint to “/inventory”:

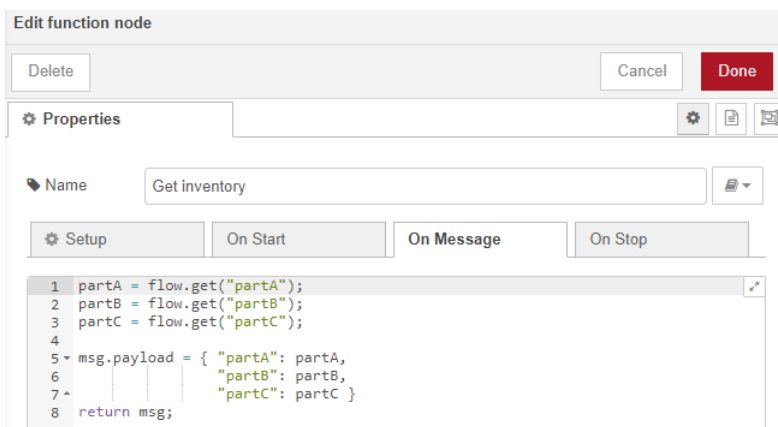


Dialog box titled "Edit http in node". It contains a "Delete" button, a "Cancel" button, and a "Done" button. Below these are three tabs: "Properties", "Setup", and "On Message". The "Properties" tab is active, showing a "Method" dropdown set to "GET", a "URL" text field containing "/inventory", and a "Name" text field containing "Name".

- Again, we will now connect a **function** node to the output of the **http in** node, in order to handle the incoming HTTP requests and prepare the JSON object for the response:



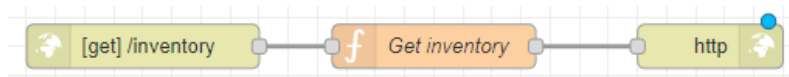
- We set the name of the function node to “Get inventory” and add the following code to its “On Message” section:



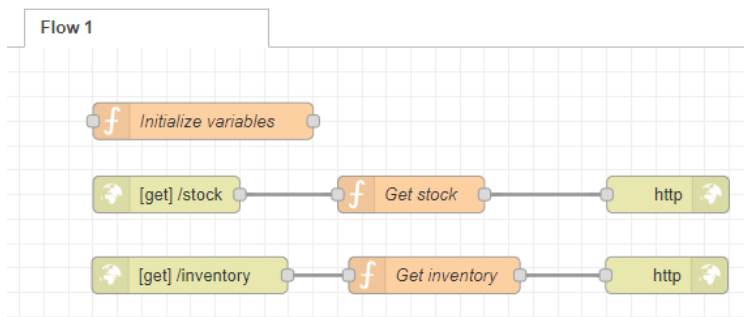
Dialog box titled "Edit function node". It contains a "Delete" button, a "Cancel" button, and a "Done" button. Below these are three tabs: "Properties", "Setup", and "On Message". The "Properties" tab is active, showing a "Name" text field containing "Get inventory". Below the tabs are four sub-tabs: "Setup", "On Start", "On Message", and "On Stop". The "On Message" sub-tab is active, showing a code editor with the following code:

```
1 partA = flow.get("partA");
2 partB = flow.get("partB");
3 partC = flow.get("partC");
4
5 msg.payload = { "partA": partA,
6               "partB": partB,
7               "partC": partC };
8 return msg;
```

- The function node above prepares the response object for the GET request, and we will again connect its output to an “**http response**” node to send back this response to the requesting client via http:



The final flow for the API should look like the following:



As a summary, we have

- A **function node** at the top that initializes our flow variable to keep track of the parts inventory and product stock (runs only once at the beginning of the application);
- An **API endpoint “/stock”** that responds to the HTTP GET messages for returning the product stock values;
- An **API endpoint “/inventory”** that responds to the HTTP GET messages for returning the part inventory values.

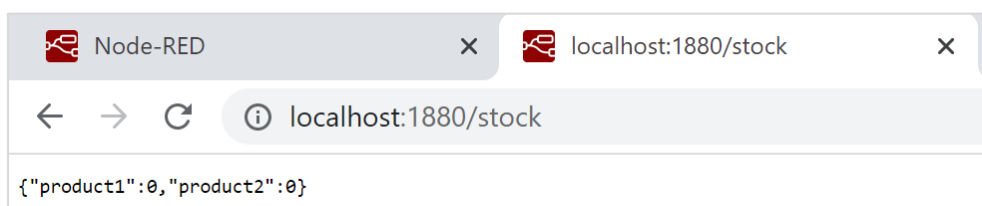
#### 4. Testing the API Endpoints on the Browser

Once you are done implementing the previous steps and deploy the flow, the API should be responding to the http requests. To test this, we can simply use a web browser to send a request to the API endpoint.

Again, assuming that your Node-RED instance is running on localhost:1880 (you can double check this from the address bar of your browser), the API endpoints we implemented should be accessible via the following addresses:

- <http://localhost:1880/stock>
- <http://localhost:1880/inventory>

Let us try the first to see the response:

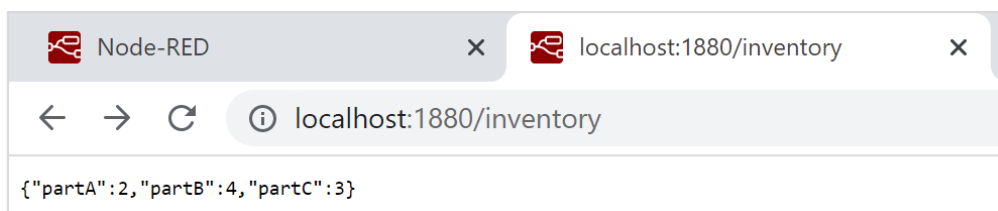


As depicted above, our web API sends back a response in the form of a **JSON object**, in which the values of product1 and product2 are both zero - as expected.

Before we test the second one, let us make a small change in our “*Initialize variables*” function node and set the inventory values of parts A, B, and C to some non-zero values:

```
flow.set('partA', 2);  
flow.set('partB', 4);  
flow.set('partC', 3);
```

Then, when we call our inventory API endpoint in our browser, we get the following response, which correctly returns the inventory values as we have set in our code.



Congratulations! You have accomplished the first part of a REST API for our inventory tracking web service. In the next module, we will complete this with the POST methods, so that other software modules can also use our API to update the values of the inventory, besides simply reading them.