

Simulation of Parts Delivery and Product Sales

Difficulty Level: Easy

Objective

Implement a web-based user interface (UI) to simulate the delivery of parts and sale of products in a smart factory scenario, by utilizing the inventory/stock web service API from the last two modules.

Achievements

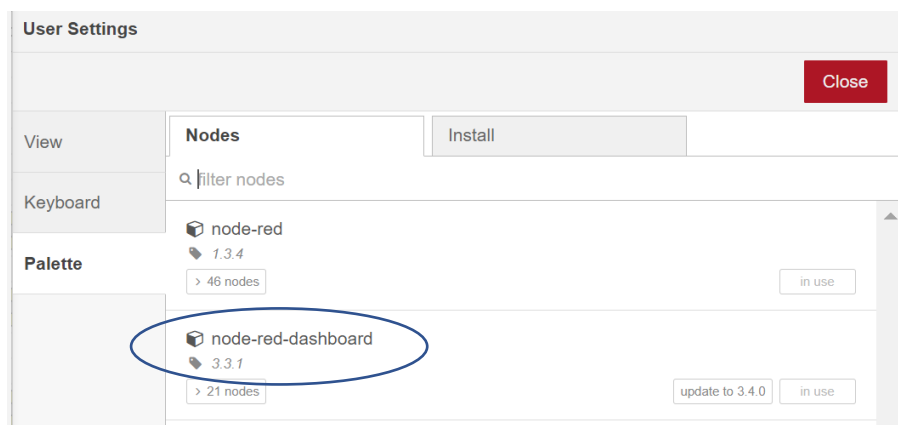
The skills to be acquired at the end of this module:

- Using forms in Node-RED to retrieve and process user input
- Implementing a client that uses the POST method of the web service API

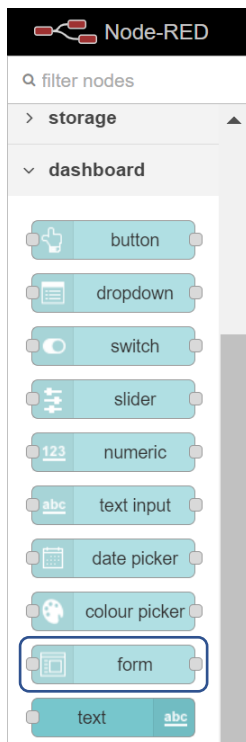
1. Retrieving User Input via Forms in Node-RED

In this module, we will utilize the Node-RED Dashboard **form** nodes to retrieve user inputs via the web UI and use those to construct API requests to the inventory/stock web service.

Before proceeding to the next steps, make sure that **node-red-dashboard** is installed in your palette manager. If not, refer to the previous IoT-factory exercise module “*Monitoring and Visualization of Sensor Data*” to install it in your Node-RED environment.

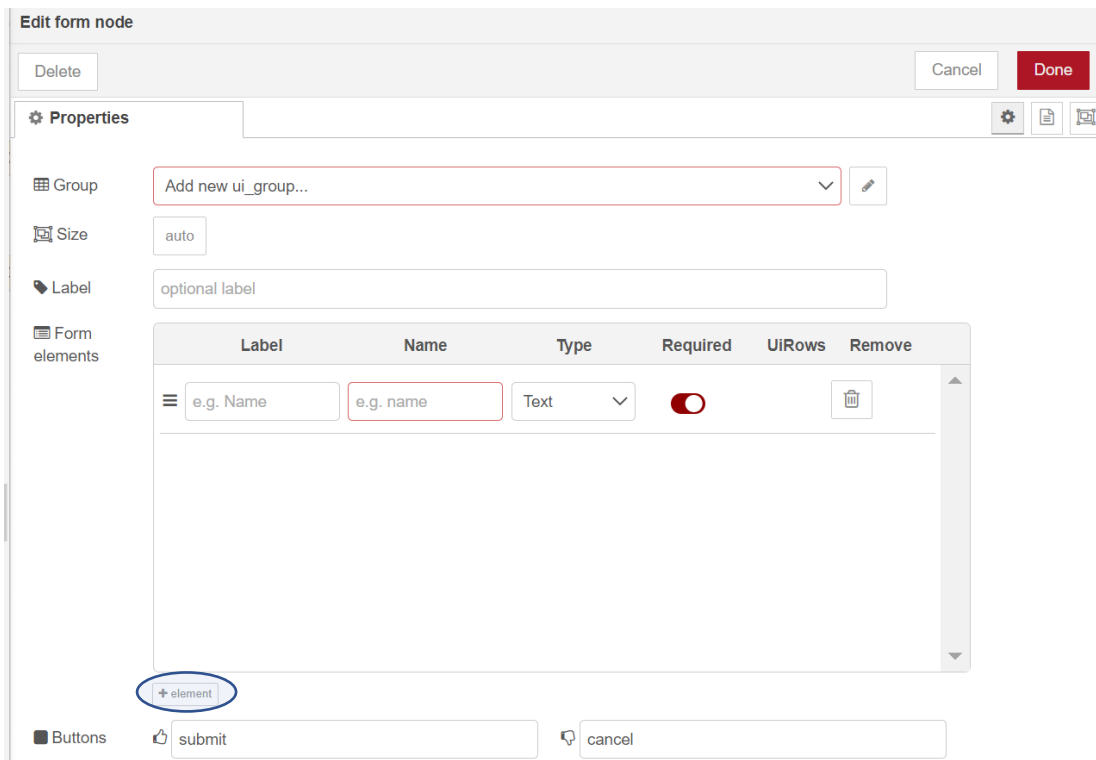


After node-red-dashboard is installed, as depicted above, you should be able to see the *dashboard* node group and the **form** node in your Node-RED palette:



The **form** node adds a web form element to user interface and helps us collect user inputs on submit-button click as an object in msg.payload.

Drag and drop a form node into your Node-RED flow and double click on it to open the settings page.



The image shows the 'Edit form node' settings page. It includes a 'Delete' button, 'Cancel', and 'Done' buttons. The 'Properties' section contains a 'Group' dropdown, 'Size' set to 'auto', and a 'Label' field. The 'Form elements' section contains a table with columns: Label, Name, Type, Required, UIRows, and Remove.

Label	Name	Type	Required	UIRows	Remove
e.g. Name	e.g. name	Text	<input checked="" type="checkbox"/>		

At the bottom, there is a '+ element' button and a 'Buttons' section with 'submit' and 'cancel' buttons.

As depicted above, we can add multiple input fields to the form node by clicking on the “+element” button and specify the label and data type for each. We can also customize the submit and cancel buttons of the form if needed.

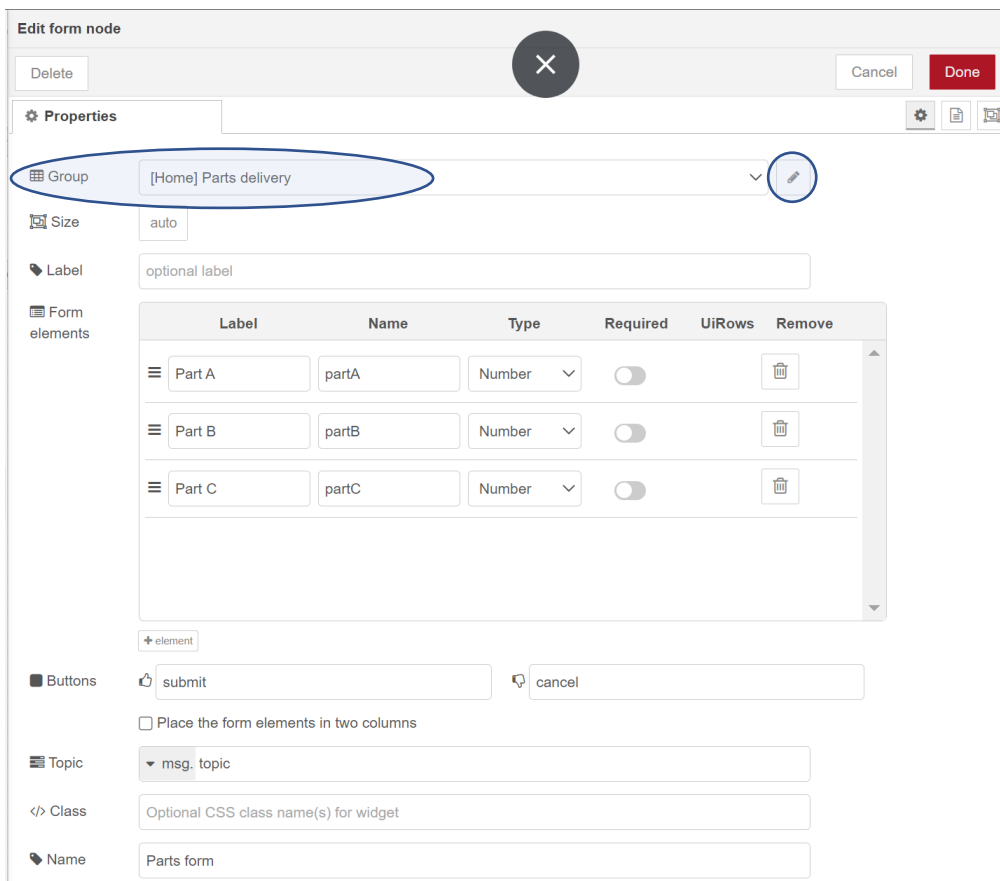
In the next two sections, we will see the details of how to configure the form node and what we get as a result in the web UI.

2. Simulated Delivery of Parts in the Smart Factory

For the simulated delivery of parts, we would like to have a form with three input fields, one for each part, as depicted below.

As with all dashboard / UI nodes, we also need to specify the **Group** for our *form* node. This serves for the organization of different UI elements / nodes on the web page.

We can create a new group or edit an existing group by clicking on the pen icon, as depicted below. (For details, refer to the previous IoT-factory exercise module “*Monitoring and Visualization of Sensor Data*”). Here, we see that our form is configured to be placed within a *group* called “Parts delivery” under the “Home” *tab/page* of the web UI.



Edit form node

Delete [X] Cancel Done

Properties

Group [Home] Parts delivery

Size auto

Label optional label

Form elements

Label	Name	Type	Required	UiRows	Remove
Part A	partA	Number	<input type="checkbox"/>		
Part B	partB	Number	<input type="checkbox"/>		
Part C	partC	Number	<input type="checkbox"/>		

+ element

Buttons submit cancel

☐ Place the form elements in two columns

Topic msg. topic

Class Optional CSS class name(s) for widget

Name Parts form

Once you deploy this flow, you can go to the “Dashboard” web UI by pointing your browser to <http://localhost:1880/> (assuming the default settings in Node-RED environment). You should then be able to see a web page with the following form with three inputs fields and a submit button (and an additional cancel button to clear the form fields).

Parts delivery

Part A

Part B

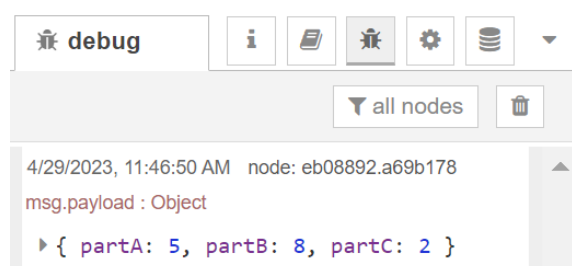
Part C

In order to observe what happens when the user fills in and submits this form, you can add a debug node to your flow and connect it to the output of the form node:



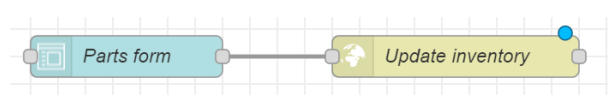
Now, switch to the web UI; enter some values for each part. E.g., 5 for Part A, 8 for Part B, and 2 for Part C. After clicking on submit, switch to the flow editor and see the debug window on the right side panel.

Then should see a JSON object being returned from the forms node, with the values of each part as expected:

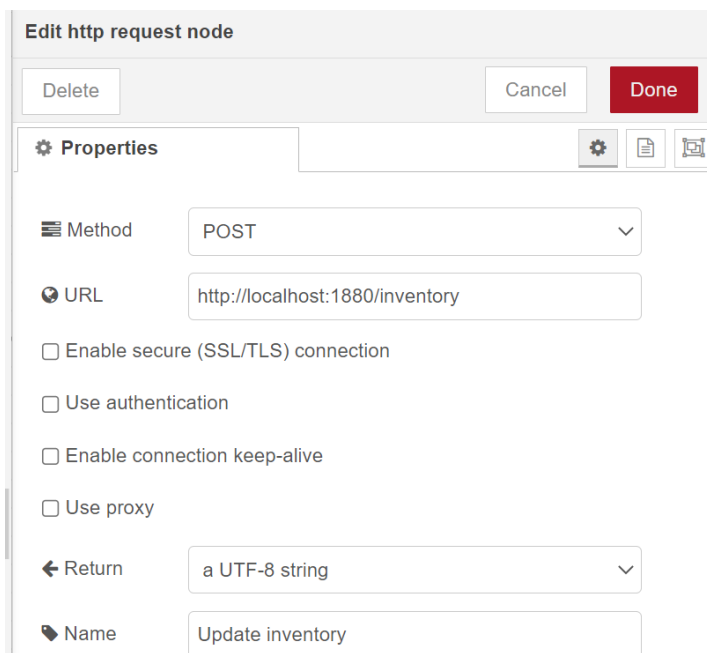


So, this makes it clear that we have actually a JSON object that we can directly use for sending an http PUT request to the inventory service API. Note that the json object us constructed using the “name” fields of the form elements, so it is important to adjust them properly when creating the form.

In order to send a request to the web service, you can simply connect an “**http request**” node to the form:



As depicted below, the *http request* node should be configured to use the *PUT method* (since we are trying to update some values on the server, rather than simply reading some data). The *URL* field should be set to the inventory API endpoint (<http://localhost:1880/inventory>), as created in the previous exercise modules on “Inventory Service API”.



Once we deploy this flow, the form on the web UI should indeed be serving as a way of simulating the desired number deliveries for any part.

In the final exercise module of the smart factory use case, we will develop a monitoring dashboard to directly test and observe the effect of our delivery simulation module.

You may also test the functionality by switching to the “Inventory Service API” module on the server side (which was created in the previous modules) and adding a debug node there to see the inventory values after each form submission via the current module.

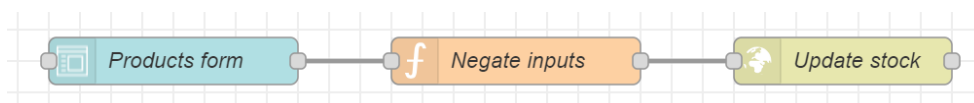
3. Simulated Sales of Products

Just like in the previous step for part deliveries, we will use a form with two input fields, one for each product, for the simulated sales of products.

Similarly, we will connect the output of this *form* node to an *http request* node to send the API request with POST method. However, in this case, we need an additional mechanism in between the two.

Note that the product stock values should be decreased whenever a product is sold; therefore, we should send negative values in the API request via the *http* node. On the other hand, the user enters positive values in the form fields. Therefore, we will add a *function* node after the *form* node, which will simply negate the form input values (i.e., multiply them by -1).

So, our flow should look like this at the end:



The setting for each node are shown next:

Note that we created a new UI group called “Product sales” to place this new **form** separately from the parts delivery form we created earlier:

Edit form node

Delete Cancel Done

Properties

Group: [Home] Product sales

Size: auto

Label: optional label

Form elements

Label	Name	Type	Required	UiRows	Remove
Product 1	product1	Number	<input type="checkbox"/>		
Product 2	product2	Number	<input type="checkbox"/>		

The **function** node “Negate inputs” would simply multiply the values of both product1 and product2 that are received via the form node, as depicted below. Note that the code statement “a *= b” is a shortcut for “a = a * b”.

Name: Negate inputs

Setup On Start **On Message** On Stop

```

1
2 msg.payload.product1 *= -1;
3 msg.payload.product2 *= -1;
4
5 return msg;
  
```

The **http request** node settings are also similar to the one we used for the parts delivery, but we use the API endpoint *stock* (<http://localhost:1880/stock>) instead of the *inventory* endpoint.

Edit http request node

Delete
Cancel
Done

Properties

Method
POST

URL
http://localhost:1880/stock

☐ Enable secure (SSL/TLS) connection

☐ Use authentication

☐ Enable connection keep-alive

☐ Use proxy

Return
a UTF-8 string

Name
Update stock

After the flow is deployed, the end result is what you would expect – a form with two input fields:

Product sales

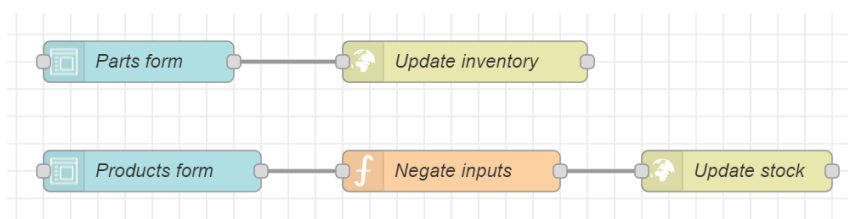
Product 1

Product 2

SUBMIT
CANCEL

As we have done in the previous step, you may add a debug node to see the output JSON object when the user fills in and submits the web form.

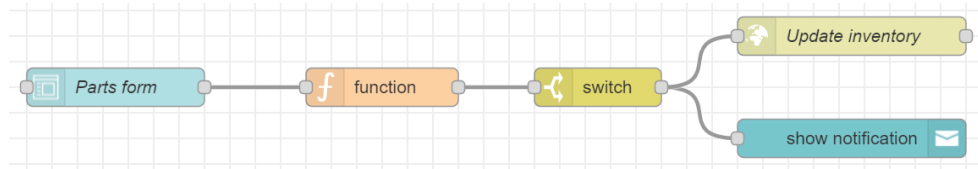
At this point, the simulated delivery of parts and sales of products is actually complete, for which the final Node-RED flow could look like the following. As you noticed, we have achieved this almost without writing any code, but only configuring the *flow*, *UI groups*, as well the *form* and *http request* nodes (except the two lines of code in the function node).



On the other hand, you can also test and see that this basic implementation omits many details, such as checking for negative values in the user input, or checking the availability of products in the stock before “sales”.

For example, if we have only 2 units of Product 1 in the stock and the user enters 5 units of Product 1 for sales, the stock value of Product 1 would be updated to -3, which is not realistic.

In order to handle such cases, we can also add some nodes in between the form and the http request nodes to do the necessary error checking. If the input is valid, the http request is triggered, otherwise, e.g., an error messages is printed on the web ui that the input is not valid or that there is not enough stock for the requested product.



For example, the end result for the simulated delivery of parts with error checking may look like the flow above. Go ahead and try on your own to make such extensions on your flow. Good luck!