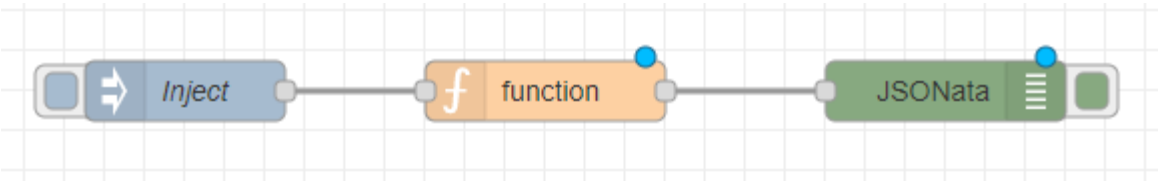
# Laboratory 2

## Different function node features

Let's make an application that is going to have two separate flows.

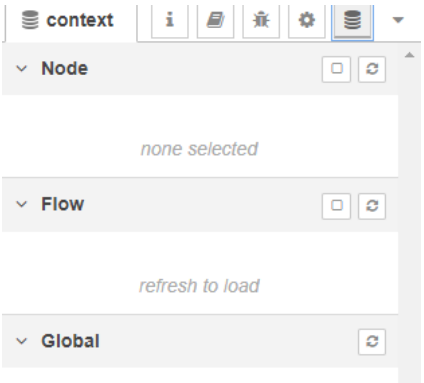
- One that is going to keep track of the number of time the flow has been triggered
- A second one that is going to print the count value every time the other one is triggered

Set up the flow so that you have an inject node, and a debug node that outputs the complete message object. Add a function node in between the two nodes, and wire them together.

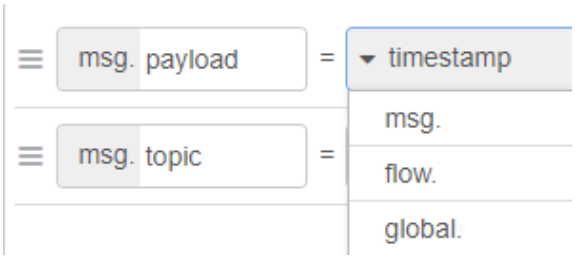



The initial setting up part is done, but now comes the question, how do you store data inside a node?

The answer is context variables. They store the data in the Context Data tab



There are three types of context variables: context, flow and global. We can use them directly in the inject node.




But what exactly are they?

- The context variable stores data within a node, no matter how many times you trigger the flow
- The flow variable stores data for the entire flow
- The global variable stores data for the entire workspace – they are never cleared, even when resetting the workflow

If we are going to build a counter, which variable type should we use?

If you said **context**, you were right, otherwise, my explanations weren't good enough.

Here is a small js code that increments and sets the counter:

```
// initialize to 0 if it doesn't exist
var count = context.get('count') || 0;
count += 1;
// store the value
context.set('count',count);
// set it as part of the outgoing object
msg.count = count;
return msg;
```

The variable count is the variable that we're going to use to store our data. So how do we use the context variables?

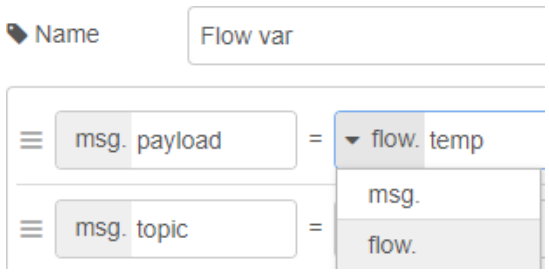
Using the get() and the set() access method. Every time we are going to **trigger** the flow, first the variable will be **initialized** to its old value, and then it is going to be **updated** and set to its new value.

When running it, we can see that the counter is going up every time we trigger the flow.

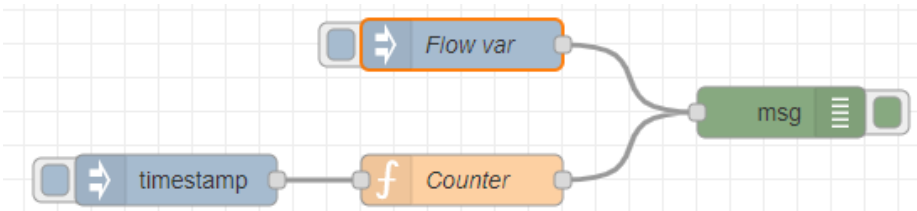

Now comes the question… how do we get it to another flow, which variable type should we use?

The **flow** variable is enough for this case. Edit the function node so that before you return the message you set save the count to a temp value in the flow. You can use: flow.set('temp',count);

Get another inject node, and wire it to the debug node. Set the payload to flow.temp, to access the variable previously stored. Using this, we are basically using a workaround for a getter method, if that makes sense.




After adding the new node, the current flow should look similar to this:

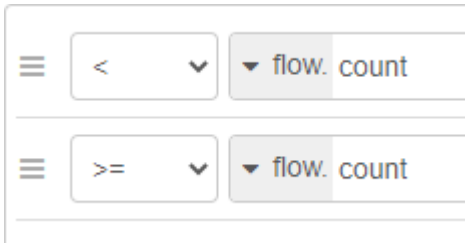



After deploying it, test the flow to see how it works.

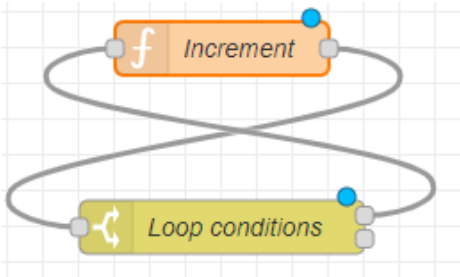
Let's move on to the next flow. Here we are going to create a loop using the function node, and the switch node.

Write the code so that the function node increments the value of the payload by 1(tip: use msg.payload). Also, the switch node should have two rules. To add rules, click the "+" in the bottom side of the rules window: 

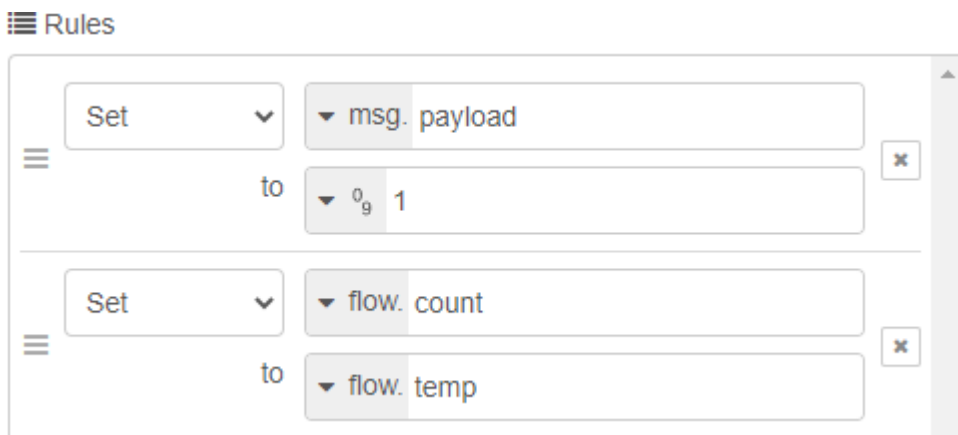

Make it so that one rule checks if the payload is less than the count value, and the other checks if it is greater or equal to it.



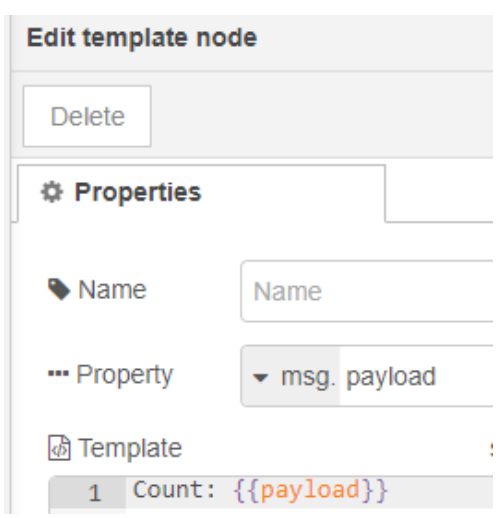
Set up the wiring in the following way:



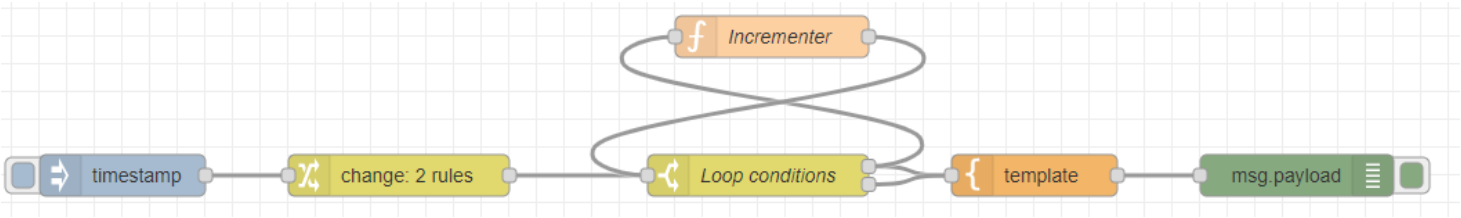

Let's make a way to modify the temp flow variable. Set up a change node that sets the payload to 1 and the flow.count to flow.temp. Make sure the "1" is a number and not a string.




To trigger the flow, add an inject node. In both flows, the inject node is only used to trigger the flow. Let's also add a debug node with a template node. We are going to use the template node to "beautify" the output.



The flow should look similar to this:




Run the program to test it out. (Increment the count value with the first flow, and then run the 2nd one)

# The link node

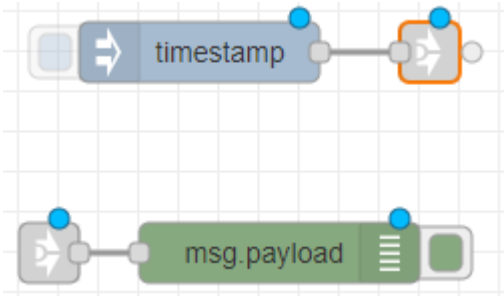
To properly see the usage of this node, let's make use of a global variable. For this, change the temp value from flow to global.

The link node is used to break flows without breaking the functionality. It also works between workspaces. There are two link nodes:

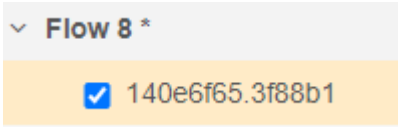
- Input (link in)
- Output(link out)


These two nodes will have a special connection between them that cannot be seen.

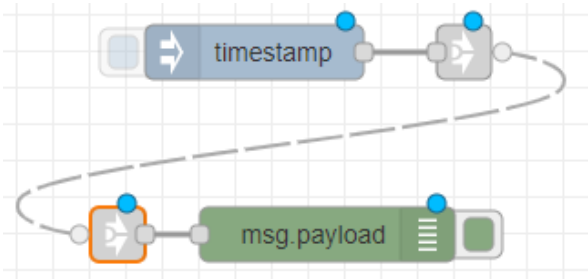
Let's try a simple example using these two nodes. Setup a simple input – output program, but instead of wiring the input and output together, use the link nodes.



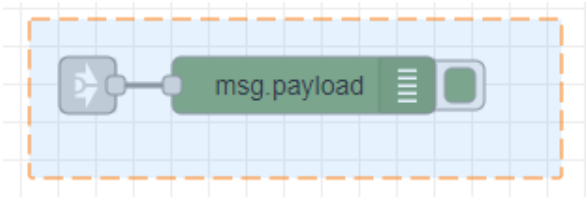
After that, you have to connect the two link nodes. To do this, open either of the two nodes, and select the other node as its connection:




After that, the two nodes should connect with a dashed line.




You can also have these two in separate flows. Select one of the two sides and move it to new flow.




If you click on the either of the link nodes, you can see that it has a connection to another flow:

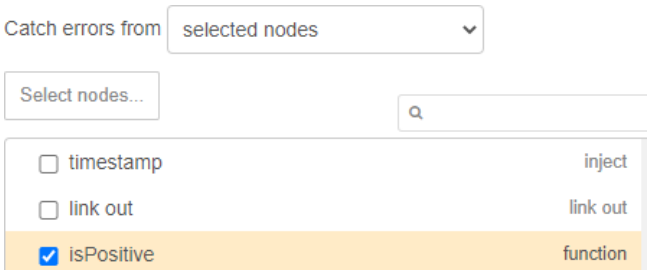

Running the program works fine in both cases. You can also use the comment node to better understand where the node is linked in case the flow becomes too complicated to understand.

Let's complicate this a bit. In the link below you have a node containing a function which checks if a number is positive, and only passes the payload if the function returns true, otherwise, an error is thrown.

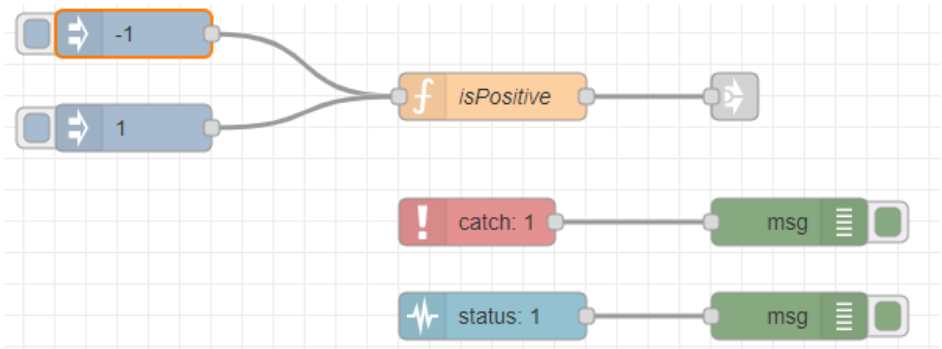
https://github.com/bmorariu/IIOTCA/blob/main/lab3_help.json

Insert the node between the inject node and the first link node:


Afterwards, drag a Catch and Status node into the flow as well, and connect both of them to debug nodes. Configure both of them to collect the information only from the isPositive function, and to output complete message objects.




Instead of one inject node, add at least one more, and change their payload to numbers, and set one to a positive number, and one to a negative one. Your flow should look similar to this:




You can also add a delay node if you want, before the output.

Extras:

Here you have some examples showing the functionalities of the change and template nodes:

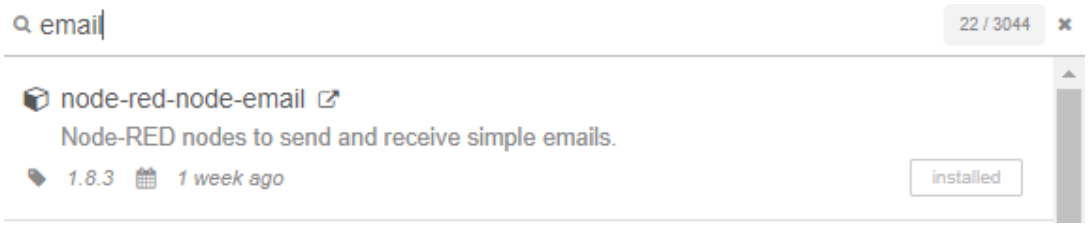
https://github.com/bmorariu/IIOTCA/blob/main/lab3.json

In mustache language, a section begins with # and ends with /. An inverted section begins with ^ and ends with /.

# The email node

With this node we can easily make a newsletter kind of application. We can use this node to email its payload to someone.

For this, the first thing we need to do is to install the email nodes.




The purpose of this shouldn't be to create a spam application, so let's make it only send the email every other day/on a certain date. For this, we are going to use a function node.

We can use the getDay() method to get the week day.

## The getDay() Method

The `getDay()` method returns the weekday of a date as a number (0-6):

### Example

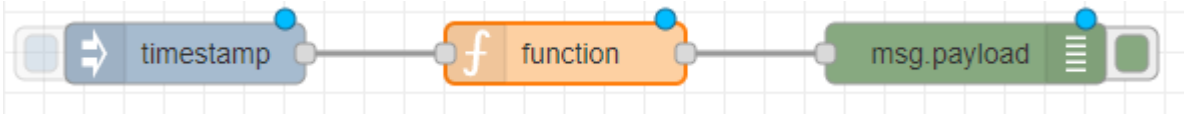
```
var d = new Date();
document.getElementById("demo").innerHTML = d.getDay();
```

Add the function node and make it return the day as an attribute of the message:

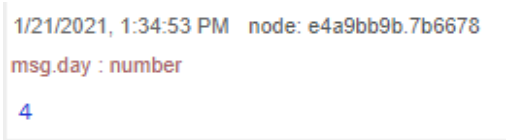
```
var d = new Date();
msg.day = d.getDay();
return msg;
```

If you want to use the date, use the getDate() method.

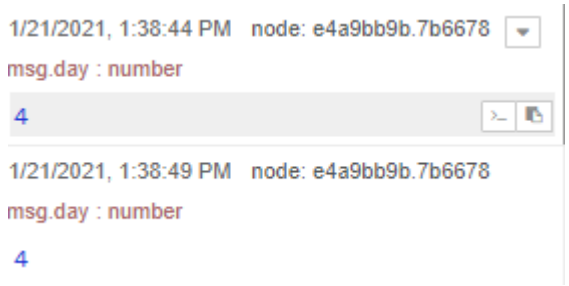
Set up the flow using an inject and a debug node, and run the program to check the output. (Set the output as msg.day)



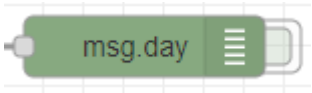
In my case, I got “4”, which corresponds to Thursday.



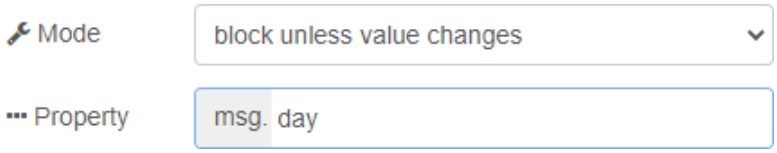

In an automatic system, we are going to need this to repeat itself, so let’s set the inject node to input the message every 5 seconds.




But as previously mentioned, we only want to input the message under certain conditions. For this, let’s learn a new node, the RBE(ReportByException) node. To stop the node from outputting messages to the debug console, click on the end of the debug node as you would on the inject node, and deploy the changes. Basically, the node will not output anything until we re-enable it.



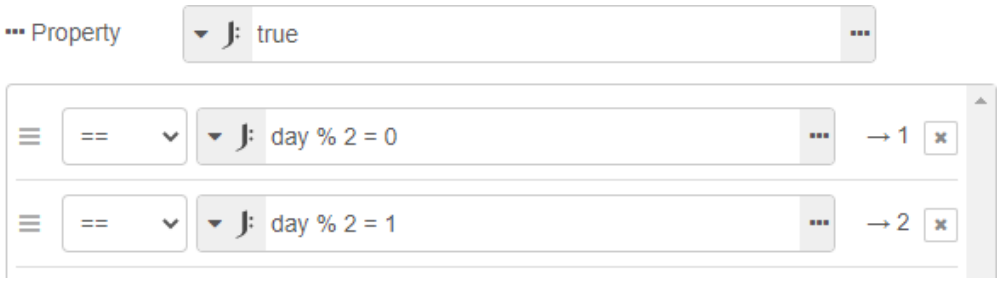

The RBE node can be used as an intermediate block which checks if the output changed. This can also be used in different automation features, like web crawling, where you don’t want to parse the site if there are no new news on it. In our case, we only want to output it when the information from the node changes:



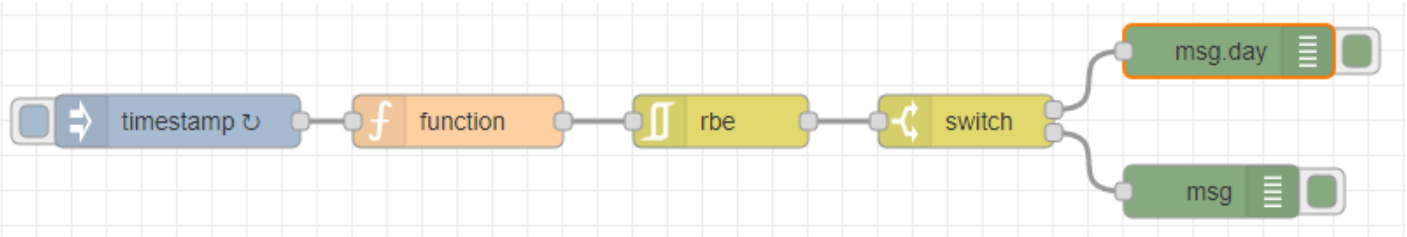

Enable the output and check the debug window again after running the flow. As you can see, only the initial message comes through, and the RBE node drops the rest.

As we said before, we only want to send this message every other day. Let's also output another message in the rest of the days. For this, we can use the switch node.

We can use this in multiple ways, by creating 7 different cases, and to check each (from 0 to 6) individually, or we can use expressions.




We could also use the "or" syntax, to check if the day is 0, 2, 4 or 6 respectively 1, 3 or 5. Add another debug node for the second condition. In the second one, let's output the complete msg object .

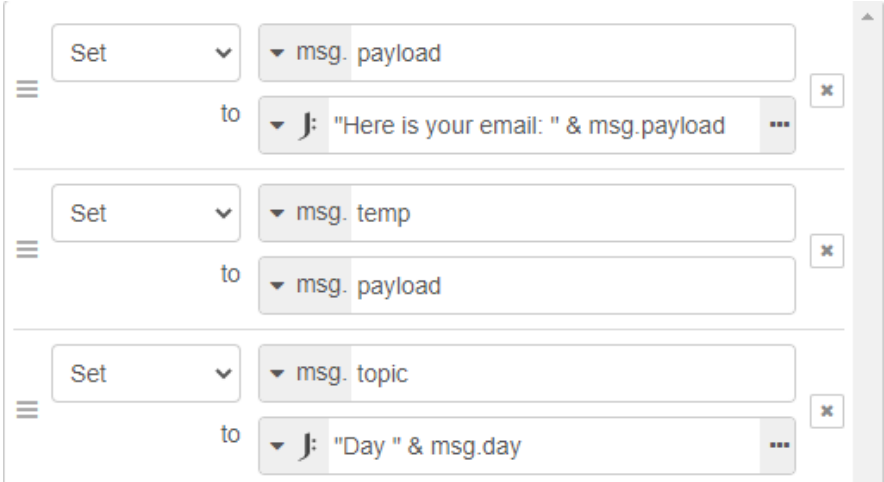



Switch the cases to check if both work (make the first case = 1, and second to 0).

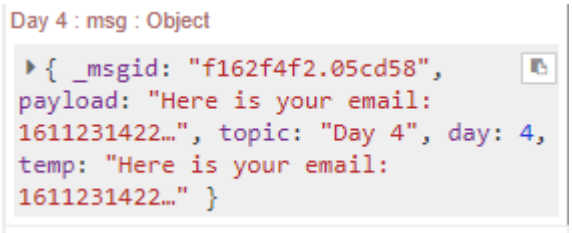
Before getting to the email part, let's use the change node to output a string instead of the initial payload. Replace the two debug nodes with change nodes for now.

In these change nodes, let's replace the initial payload, which is a timestamp, with a small message: "Here is your email:". When doing this, we can simply set the output to a string. If we want, we can also add the old payload to this example: "Here is your email:" & msg.payload . This way we have a more complex-ish example. But in this case, we will have to use an expression. We could also save the old payload to a temp value, for future usage.

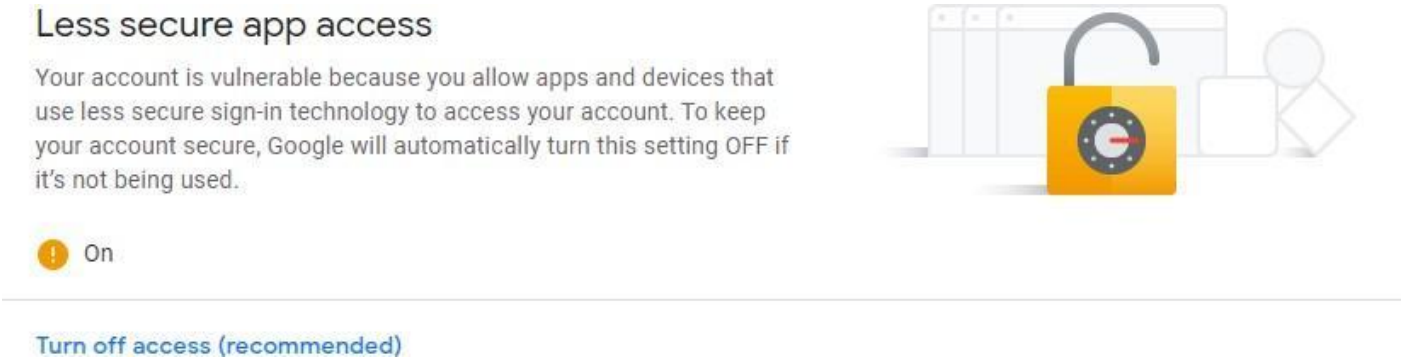
We are also going to need a subject for the email. For this, we can use the topic of the message object. We can also set this in the change node.



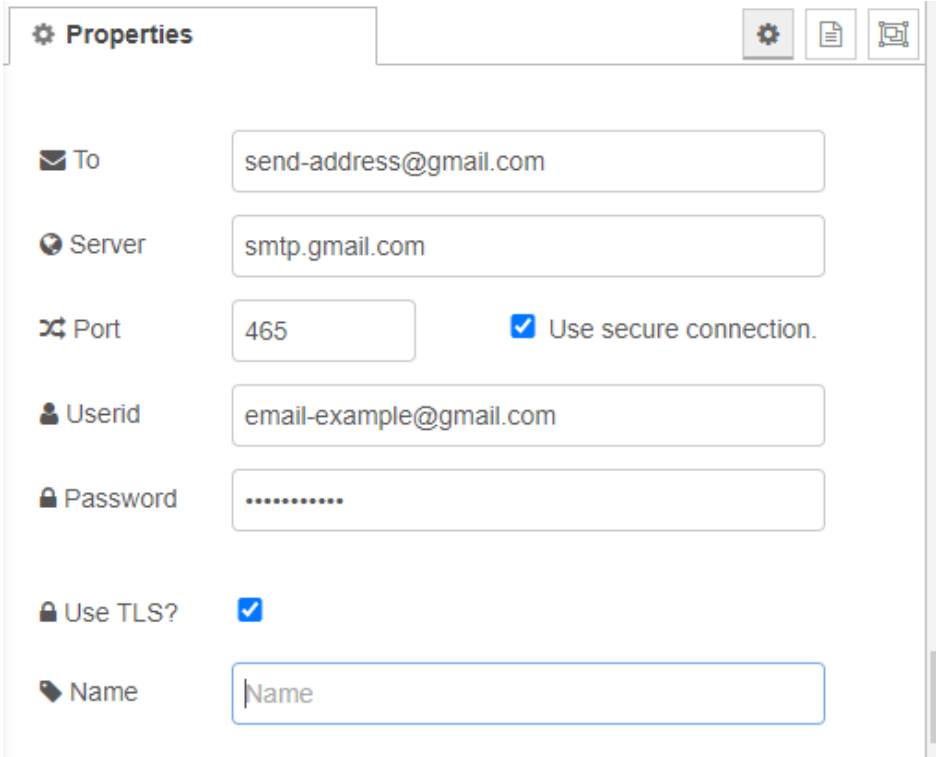

Set the other node in a similar way, but change things up a bit, to have different emails. Check the outputs, if you output the entire message object, it should look like this:



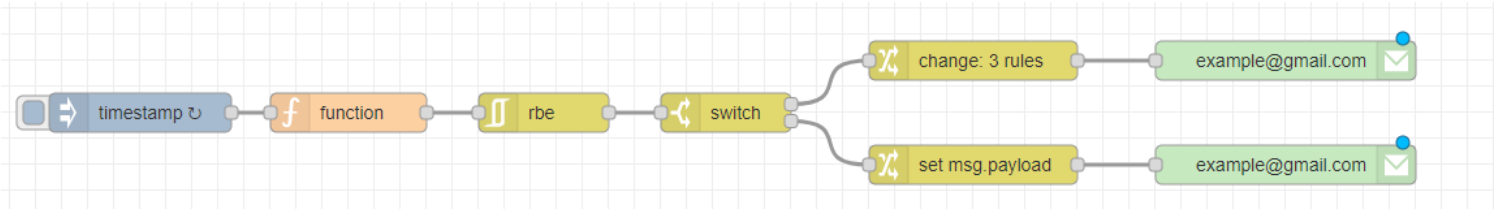

Now on to the last step, we need to add the email node. For this, you need to go to you Google account, in the security tab(if you are using gmail), and turn ON access from less secure apps**(SWAP THIS BACK AFTER THIS EXAMPLE!!):** https://myaccount.google.com/security



Then setup the email node:




At the end, this is how the flow looks like:




Run the program, and check your email:


REMEMBER TO TURN OFF LESS SECURE APP ACCESS

It’s over… It’s finally over! Here’s an apologetic doggo