**Documentation Fischertechnik Plant**

1. Information about the different devices

First the different devices are listed with all the information needed to access them:

* Lenovo Laptop W540

Username: showroom

Password: Cognitive4Factory2021 (has to be changed every 90 days)

* Own IBM Cloud Account

Username: IoT\_Demonstrator\_Munich@outlook.com

Password: Cognitive4Factory2021

* [Node RED](https://node-red-fischertechnik-munich.mybluemix.net/red/) instance in the cloud:

Username: showroom\_Fischertechnik

Password: Cognitive4Factory2021

* Raspberry Pi 3 (Used for plant communication, placed in the closet)

Username: pi

Password: Cognitive4Factory2020

IP: 9.236.65.196 (is not static and may change)

* Raspberry Pi 3 (Used filming the plant, placed on top of the plant)

Username: pi

Password: raspberry

IP: 9.236.65.232 (is not static and may change)

* Raspberry Pi 1 (Used for Visual Recognition, placed in the processing station)

Username: pi

Password: raspberry

IP: 9.236.65.192 (is not static and may change)

* Github: <https://github.ibm.com/GBSWatsonIoTPracticeMUC/int-connected-factory>
* Gmail account (Used to send an email over Node-RED)

Username: fischertechnik.plant.munich@gmail.com

Password: Cognitive4Facory2021

App-Passwort: emjhpcwtokpokiqd (used in Node-RED)

If the IP of the raspberrys have changed you can get the new ones with the tool ‘advanced IP scanner’. The raspberry pi 3 are listed under the producer ‘Raspberry pi foundation’. The raspberry pi 1 can be found under the producer ‘Edimax Technoloy Co’ (This is the manufacturer of the WiFi dongle).

All devices should be connected to the same WiFi network with the SSID name ‘IOTDEMOS’ and the pwd “?????????”

1. Presentation of the different hardware devices used:

This chapter should give a short overview of all the different components used in the setup. We start with the Fischertechnik modules:

* Warehouse 1
  + In this configuration provides the workpieces
  + A space-saving storage area for storing and retrieving goods
  + Uses pallets to transport the workpieces
  + Crane transports the pallets inside the warehouse
* Vacuum Gripper 1
  + The vacuum gripper can be used for handling Tasks
  + Can be moved in 3 directions
  + Uses a compressor to pick up workpieces
  + It connects the warehouse 1 to the processing station
* Processing Station
  + Consists of an oven or a painting station, depending on use-case
  + Possesses a saw or a polishing station depending on the use-case
  + Has a conveyor belt for delivering the workpieces to the sorting line

* Sorting Line
  + Is used for an automated separation of different colored workpieces
  + A light sensor detects the color of every workpiece
  + Pneumatic cylinders sort the workpieces in different slides
* Vacuum Gripper 2
  + Same as vacuum gripper 1 but connects the sorting line to the warehouse 2
* Warehouse 2
  + Same as warehouse 2 but serves as output storage

Every of those modules is equipped with one Fischertechnik controller (except the processing station, which has 2 of them. Then one of them acts as master and the other one as slave). The controllers have multiple input and output ports which are connected to the different actuators and sensors of the corresponding module. Further information about the TXT controllers can be found in the folder “Documents Fischertechnik”. Every module has to be programmed individually. The corresponding programs can be found in the folder “Fischertechnik\_plant\Code\Robo\_Pro\_Code”. Later more on this topic.

The next components of the setup are the Arduinos. Two different versions were used. First the [Arduino Uno](https://www.amazon.co.uk/Arduino-A000066-ARDUINO-UNO-REV3/dp/B008GRTSV6) and secondly the [Arduino Nano](https://www.amazon.co.uk/dp/B07143JN73/ref=twister_B0741D84CS?_encoding=UTF8&psc=1) . The reason different versions were used is because the Arduino Uno’s were available. However, the number was not sufficient and so for cost reasons only Nanos were newly ordered. Within the framework of this setup they offer the same functionalities. The Arduinos can be programmed with the [Arduino IDE](https://www.arduino.cc/en/main/software) . Different source codes (which can be uploaded to the Arduinos) are available depending on the application of the Arduino’s. A detailed description is given later.

Next to the Arduinos an [ESP 32](https://www.conrad.de/de/p/entwickler-platine-sbc-nodemcu-esp32-1656367.html?refresh=true) is in use. This one has the advantage of a built-in WiFi module. So, it can be directly connected to the local wireless network. However, this functionality s not used in this setup. The ESP32 is connected to a temperature and humidity sensor. This board can also be programmed with the Arduino IDE ([tutorial](https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/)).

The Arduinos and the ESP32 are all connected via USB to an [Raspberry Pi 3](https://www.raspberrypi.org/products/raspberry-pi-3-model-b/). The Arduinos have no built-in WiFi module, so they can’t communicate through the local network. That’s why the Raspberry is used. More on this topic later in the architecture section

Two other Raspberry Pi’s (one version1 and one version 3) were used along the one connected to the Arduinos. Both use a [Raspberry Pi camera](https://www.raspberrypi.org/products/camera-module-v2/). One of them (with the other Raspberry Pi 3) is used to film the whole plant. It is mounted on top of the model factory. So the plant operator can observe what’s going on, even if he is not on site. The second Pi is placed at the processing station. The camera is mounted above the conveyor belt, so that it can take a picture of every passing workpiece. This picture can then be sent to the cloud for a visual inspection of the according workpiece. Figure 1 shows the Raspberry Pi with the Camera mounted on top of the conveyor belt. More to this subject later.

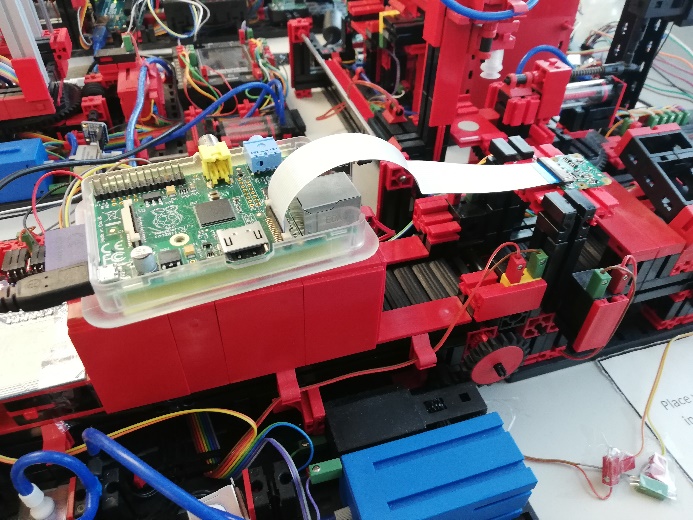


Figure : Visual Recognition

1. Presentation of the Cloud Components

After having presented all the hardware devices, now the resources deployed in the [IBM Cloud](https://cloud.ibm.com) are listed. A list of all cloud components and their current price is stored in “\Documents\List\_of\_Components”. Most of them are deployed in the practice cloud account under the organisation ‘IoT-FTF-Fischertechnik’. However I had not all the permissions to deploy all the ressources so that the Visual Recognition and the Cloudant Database are based in the “IoT\_Demonstrator\_Munich@outlook.com” account.

* Node Red Instance
  + Used to control the plant
  + Takes the role as a manufacturing execution system (MES)
  + Provides a dashboard
* IoT Platform
  + Connects the different devices to each other
  + Makes the data send by the plant available for the other services
  + Offers the possibility of sending commands to the plant
* Cloudant Database (not in the practice cloud)
  + Stores all the events send to the IoT platform
  + Can be used to query informations
* Visual Recognition
  + Is used for visual inspection of the workpieces
  + Model trained with pictures of workpieces with and without a scratch
  + Uses to model to classify new images
* Watson Assistant
  + Chatbot for asking informations and to control the plant
  + Integration in Slack
* Cloud functions
  + Used to send commands to the IoT platform
  + Can query information in the Cloudant database

1. Presentation of the different functionalities

In the following chapters the different parts of the plant are explained more in detail and their interaction is shown. Figure 2 displays the used IT architecture. The individual relationships are also explained in the following sections

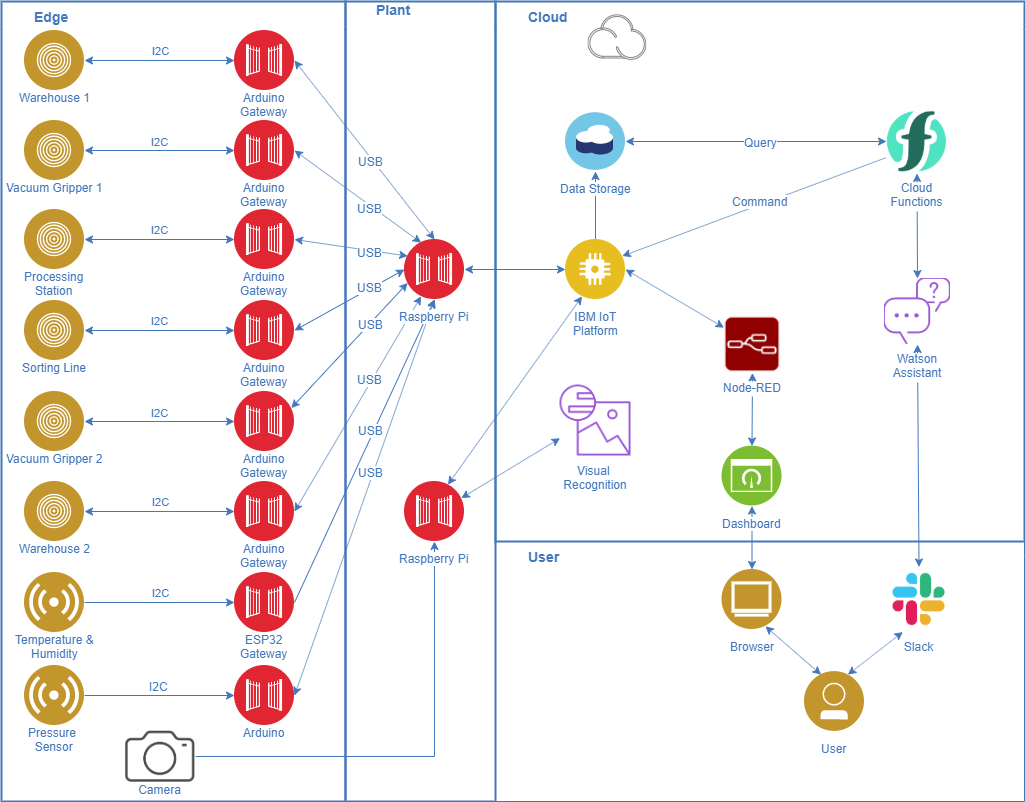


Figure : IT Architecture

* 1. Functionality of the Fischertechnik Modules/Controllers

The Fischertechnik modules consist of different actuators (e.g. motors and valves) and of multiple sensors (e.g. light barriers). All these components are connected to the controller (or to the two controllers in the case of the processing station). This one (as the name suggests) controls the connected module. For doing so an individual code/program is needed. This one can be created on the PC with the software “[ROBO Pro](https://www.fischertechnik.de/en/products/playing/robotics/93296-robo-pro-software)” before being loaded to the controller. The language used for programming the controller is similar to the language ‘sequential function chart’. It consists of several blocks which can be wired together. These compose then the process flow. An example is given in the Figure 3. It shows an excerpt of the flow which sends informations to the Arduino over the I2C interface. The flow starts at the first block named “Entry” and continues directly to the next block which is an intersection comparable to an if else query. If the input “Ready2Send” is greater than 0, we go on to the next step. When not, the query is repeated and so on.

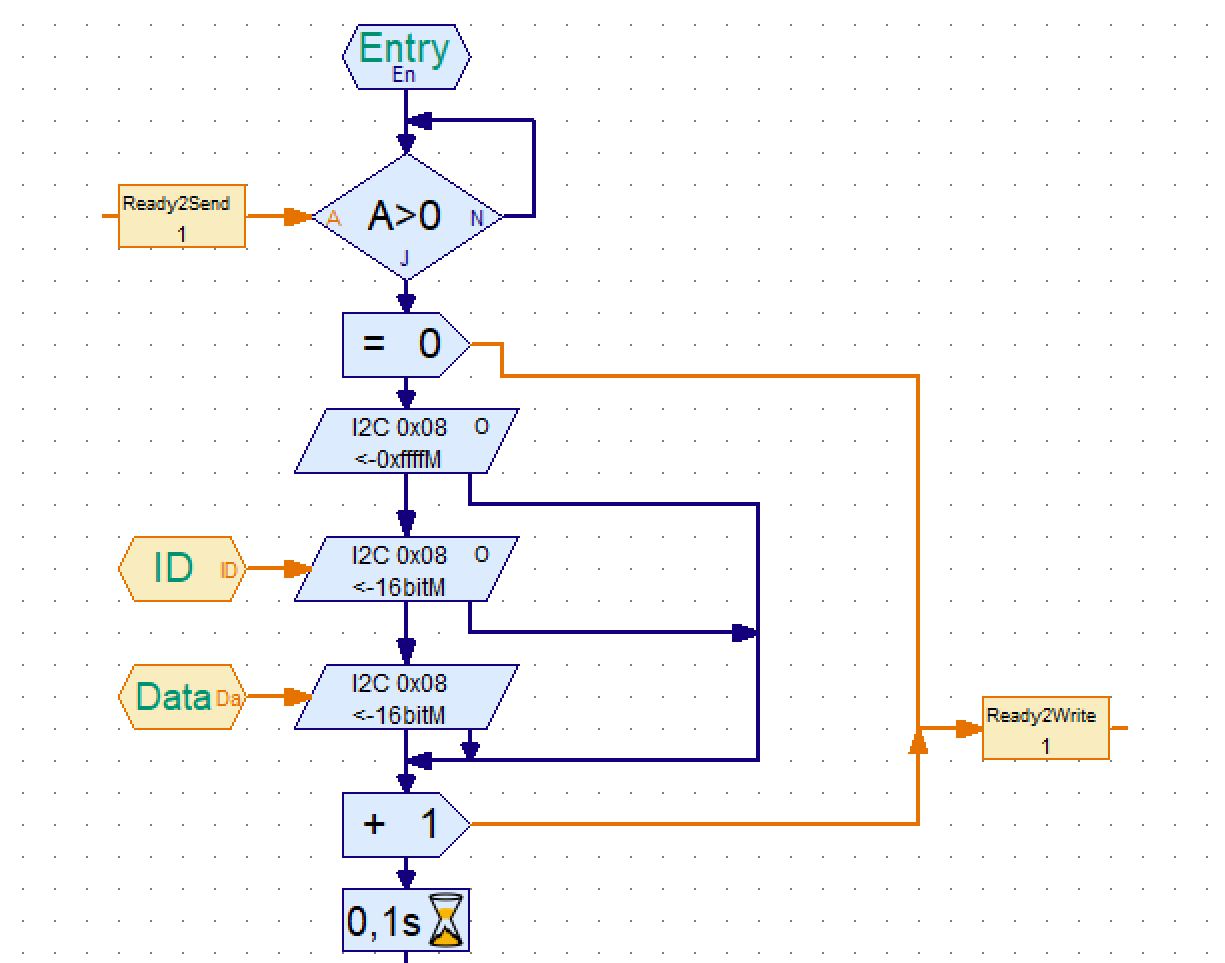


Figure : Robo Pro Example

In the actual field of application (learn children/young adults how to program) the controllers can only receive commands via the built-in touch display. For the rest they execute the program completely independent form the outside world. This means a given workflow is followed step after step. The controller then reacts to inputs from sensors and sets the actuators in a predefined way. For example, if a workpiece is detected by the light barrier the controller starts the conveyor.

One goal was now to be able to send commands not only via the display but via other input channels like a dashboard or a chatbot. Furthermore, the controller should send informations to a cloud. These informations could be current sensor events or just the current state of the controller. Therefore, a communication to the outside of the controller had to be established.

Due to the circumstances that the controllers were designed to do local plant controlling, they were not equipped with a regular interface for an easy data exchange with an external device or even a network. The only interface allowing a communication with the outside world is the I2C interface. It was designed to connect further sensors which are not delivered by Fischertechnik (e.g. a temperature sensor). Thus there is a possibility to build up a communication with an external device. However, there is still a problem. When connecting sensors to the controller, this one normally takes the role as master in the communication. If the controller wants to know a certain sensor value, it first sends a command to the connected sensor. This is only allowed to answer to a request. So the controller sends a request and waits then for an answer.

When using the I2C interface for a communication with another device, which is not a sensor, this causes some problems. First the controller decides when he sends an event and when he listens to a command. So it is not possible to send a command to the controller at any time. We must wait until the controller makes a request. Unfortunately there is no immediate solution for this problem, but we can work a way around it. More information to this topic in the section about the Arduinos. A second problem occurs because the controller is set as master, which can’t be changed. Thereby it is not possible to wire all the controllers together and connect them to one single external device which enables the connection to a network. This problem can be solved by connecting every controller to only one individual communication partner, which is set as slave. In this setup Arduinos were used to do this task. So for every controller one single Arduino must be used, which serves as gateway between the connected controller and the Raspberry Pi as shown in Figure 2.

For the [I2C bus](https://en.wikipedia.org/wiki/I%C2%B2C) 3 cables are necessary. One to set the ground level, one for a clock signal and the third one for sending the data. That’s it.

When we want to send an information, we use the “write block” (shown in Figure 3) in Robo Pro. 3 values are transmitted when sending from the controller to the Arduino. First a value (0xffff), which serves as identification for the Arduino that he is receiving a valid message, is send. This one is followed by an ID, which serves as identification for the origin module. Every module has his own ID, which is documented in the Excel files in the folder “\Documents\Module Events”. The third and last value is then the actual data we want to send. Since it is easier to send a number instead of a string, numbers were used to encode the events and commands. The Excel files which list the values for every event also can be found in the folder “\Documents\Module Events”. These events can be sensor information such as ‘the first light barrier has been activated’ but also informations about what step is currently executed by the RoboPro code (eg. “crane is moving to position 2”).

Just for information why numbers (ID and Data) were used to encode the events: It’s easier to send a number via the I2C interface than an entire string. That’s it. The encoded commands are afterwards decrypted by the Node-Red. Another advantage is the easier handling when we want to trigger a command after a certain event. With this procedure, we have just to check if the evet is equal to a number (e.g. event==32).

On the other hand, we want to send commands to the controller. To do so we can use the “read block”. This is only used in situations when the controller must take a decision which needs a command from the outside.

An example for this is shown in Figure 4 where an excerpt of the flow of the processing station is shown. When the program is started, it first executes a reference run. After this has been completed and the light barrier (Sensor I5) has been triggered, the process jumps into the block “Wait4Command”. This one implements the reading of a new command. If any new command is available at the Arduino this one is transmitted to the controller, stored in the variable “lastcommand” and the flow continues. If no command is currently available at the Arduino the controller keeps asking continuously. In the meanwhile the controller can’t send any informations (like sensor measurements) to the Arduino.

Once the process continues because a new command could be read, no further command can be registered until the process comes back to the “Wait4Command-Block”. So no interruption of the process is possible one it has started.

Since a command can only be considered when the process is currently at the “Wait4Command-Block”, a command send during the execution of the process has to be temporarily stored at the Arduino. For example, the process is in the block “ansaugen” and a new command arrives at the Arduino, this one has to store the command until the controller asks for it. After this, the command can be deleted. So always one command can be stored locally on the Arduino

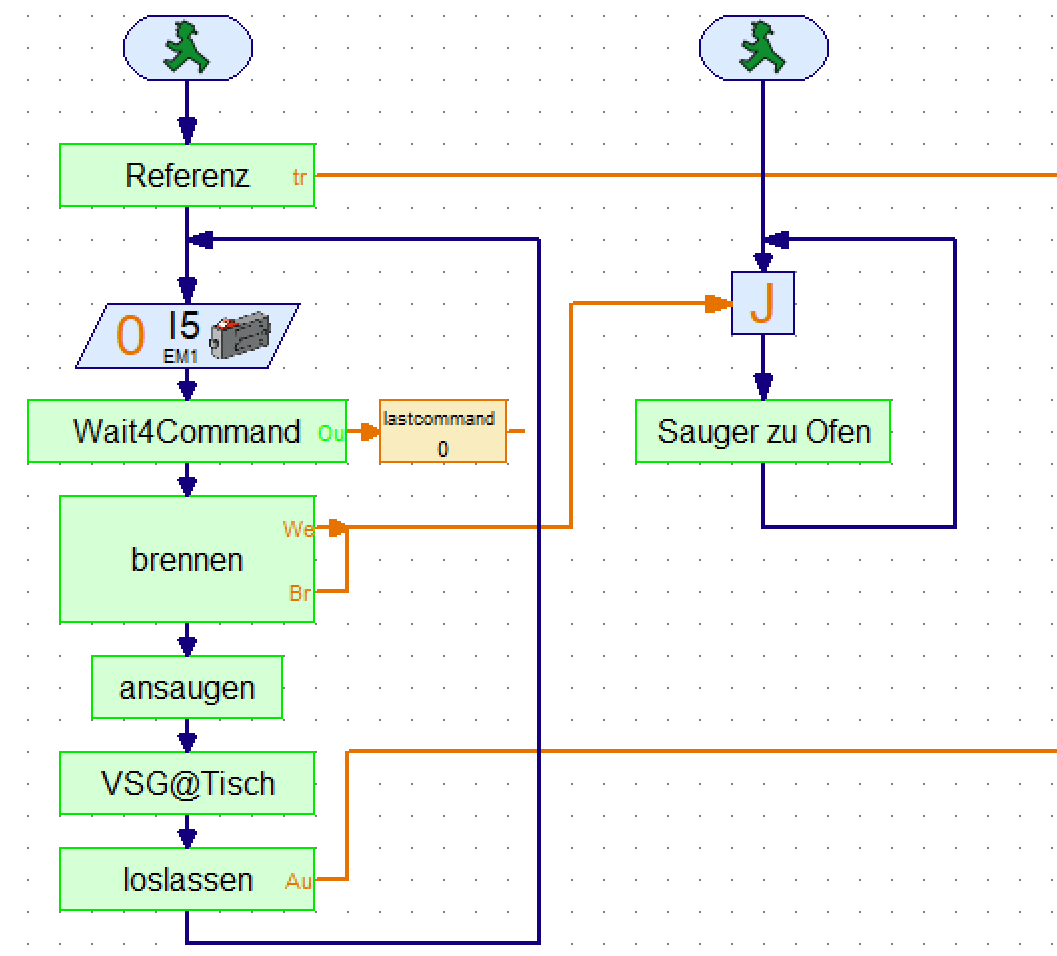


Figure : Code Exerpt of the Processing Station

The controller is responsible for the main controlling tasks. So he steers the motors, valves and so on. The controllers always follow a defined flow which they execute step after step. An example can be: ‘Turn motor 1 on’, ‘after 20 revolutions turn the motor 1 off again’ or ‘if light barrier 1 has been activated open the valve 3’. The only way we can interfere in this process is by the commands. We don’t ‘control’ any movements but simply give instructions like ‘drive to poition2’ or ‘do no processing to the workpiece’. Then the controller takes over and commands the different components. If this process has been finished, the controller waits/listens to the next command.

Since the I2C interface does not allow a parallel communication in both directions, the controller can’t send and read informations at the same time. Therefore two subprograms with the names “send2byte” and “wait4command” were created in RoboPro. These were the same for every controller program (if needed). They ensure, that the controller never starts to send a command when he is requesting a command from the Arduino and vice-versa.

1. Functionality of the Arduinos

The main task of the Arduinos is to take the role as a gateway (See Figure 2). On the one side the Fischertechnik controller is connected via I2C. On the other side a USB connection is used to establish a communication to a Raspberry Pi. (Note: We also could have taken one Raspberry Pi for every controller set is as a slave in the I2C connection. This way however we would have to buy 6 raspberry Pis, which would have been more expensive than 6 Arduinos and one Raspberry Pi). The Arduino has to forward the messages it receives. If a message is sent by the controller, the Arduino reads it and forwards it to the Raspberry Pi and vice-versa.

First let’s start by explaining how a message send by the controller is forwarded. The Arduino waits all the time for a signal of the controller (as already mentioned before the Arduinos are operated as slave devices). Once the connected controller sends the 3 values successively in one message, these messages are stored locally. As said before the message is composed of an identifier (0xffff), the ID and the actual data. If the Arduino has recorded the whole message it parses it and extracts the ID and the data value. The identifier is not needed anymore. After this step, the ID and the data value are brought into a new json format, containing a controller’s ID (which is always 0), a subadress (the module ID), a timestamp and the actual data. After this the Json is send over the serial (USB) connection to the Raspberry Pi.

When we want to send a command to the controller, this one is first sent from the Raspberry Pi to the Arduino over the USB connection. The message is in Json format. The controller however is not capable of handling this format. Therefore, the needed information (namely the command) is extracted an only this value should be forwarded to the controller via the I2C interface. As previously mentioned, the Arduino can only send to the controller if this one makes a request. So when the Raspberry Pi sends a command and the controller is not waiting for one, the value should be stored in the local memory of the Arduino. As soon as the controller makes its next request for a command, the value should be transferred and afterwards be deleted from the local storage.

That’s basically the role if the Arduino is only connected solely to a controller. Some modules however are equipped with an NFC reader (the warehouse 1, the processing station and the sorting line). In those cases the NFC reader is also connected to the Arduino via a [SPI interface](https://de.wikipedia.org/wiki/Serial_Peripheral_Interface). Then the Arduino has additionally the task of asking the NFC reader in regular intervals if this one has read the NFC ID of any new workpiece. If a new workpiece (and so a new NFC tag) has been detected, the NFC ID should be sent to the cloud (via the Raspberry Pi). This is done the same way as an event received by the controller. The value is stored in a JSON format and then send to the Raspberry Pi via the USB connection.

Another role of the Arduino is the connection of a [pressure sensor](https://www.adafruit.com/product/3965). The sensor also uses the I2C Interface. To avoid problems in the communication with a controller a separate Arduino is used to connect the pressure sensor (Remember the master-slave problematic). The procedure is like the ones before. The Arduino asks the pressure sensor for the current pressure value in regular intervals. When receiving the value, the Arduino inserts it into a Json format containing the subadress, value, timestamp and controller ID (Just as if a controller would have been connected to the Arduino) and sends it to the Raspberry Pi.

Due to the circumstance that the Arduinos have to take different tasks, different codes were created. These can be found in the folder ‘Code\Ardunio\_code’. There 3 codes can be found. One for the connection with only a controller (‘i2c-basic’), one for the connection with a module which possesses an NFC reader (‘i2c\_and\_NFC’). The third one (‘Pressure\_Sensor’) is then for the connection to the pressure sensor. They can be uploaded with the Arduino IDE to each Arduino. When uploading to one of the Arduino Nano you have to pay attention to selected bootloader. One of the Arduinos needs the old one where as the others need the new one. So if you get an error message during the upload, try to change the bootloader.

1. Functionality of the Raspberry Pi

As already mentioned before the Raspberry Pi is connected to all Arduinos via an USB connection. Because the Pi has only 4 USB ports, these are expanded with 2 USB Hubs to 10 ports.

The Raspberry Pi takes the role of another gateway (as shown in the architecture in Figure 2). It links the USB connection to the wireless network of the IBM centre. So, it is possible to send the informations received from the controllers to the IBM IoT platform and to return commands. Do so we use a [Node-RED](https://nodered.org/) instance. This one runs locally on the Raspberry Pi and is started automatically after the Raspberry Pi has booted (instructions concerning the setup of Node RED can be found [here](https://nodered.org/docs/getting-started/raspberrypi)). The Node-RED flow on the Raspberry Pi can be accessed with your pc/laptop when you are in the same network (IOTDEMOS) through the address ‘http://IP\_of\_the\_Pi:1880’

The messages send by the Arduinos are already in the correct format (JSON). So, they only have to be forwarded to cloud without being converted. Every module of the Fischertechnik plant is represented by one device in the IBM IoT platform. When taking a look at Figure 5 you can see the blue nodes on the right side. They represent the individual devices in the cloud. The informations send by every module have to be directed to the correct IoT node (the blue ones). Since all the modules respectively their Arduinos are connected over USB, we can use the serial nodes (here in brown). Every USB port is identified by a unique name, which must be stored in the serial node. However, we can’t assign a fix port to every Arduino. They can change after every restart of the Raspberry Pi/Node-RED instance. Therefore we have to identify the origin (which module has sent them) of every message an then direct it to the correct IoT node. This can easily be done, because every message contains the ID of the origin module. We simply have to check the ID of every new incoming message and forward it to the correct IoT node.

This proceeding is shown in Figure 5. The serial nodes on the left side collect all the messages received on all the USB ports an forward them to the single “json” node. This prepares the messages, so that these can be handled by the Node-RED functions. The following node (“Determine Device”) then checks the ID of every message and forwards it to the proper IoT node. The green nodes are debug nodes, which can be activated to see the incoming messages in the debug window. The yellow nodes with the names “store port” are explained in the following section.

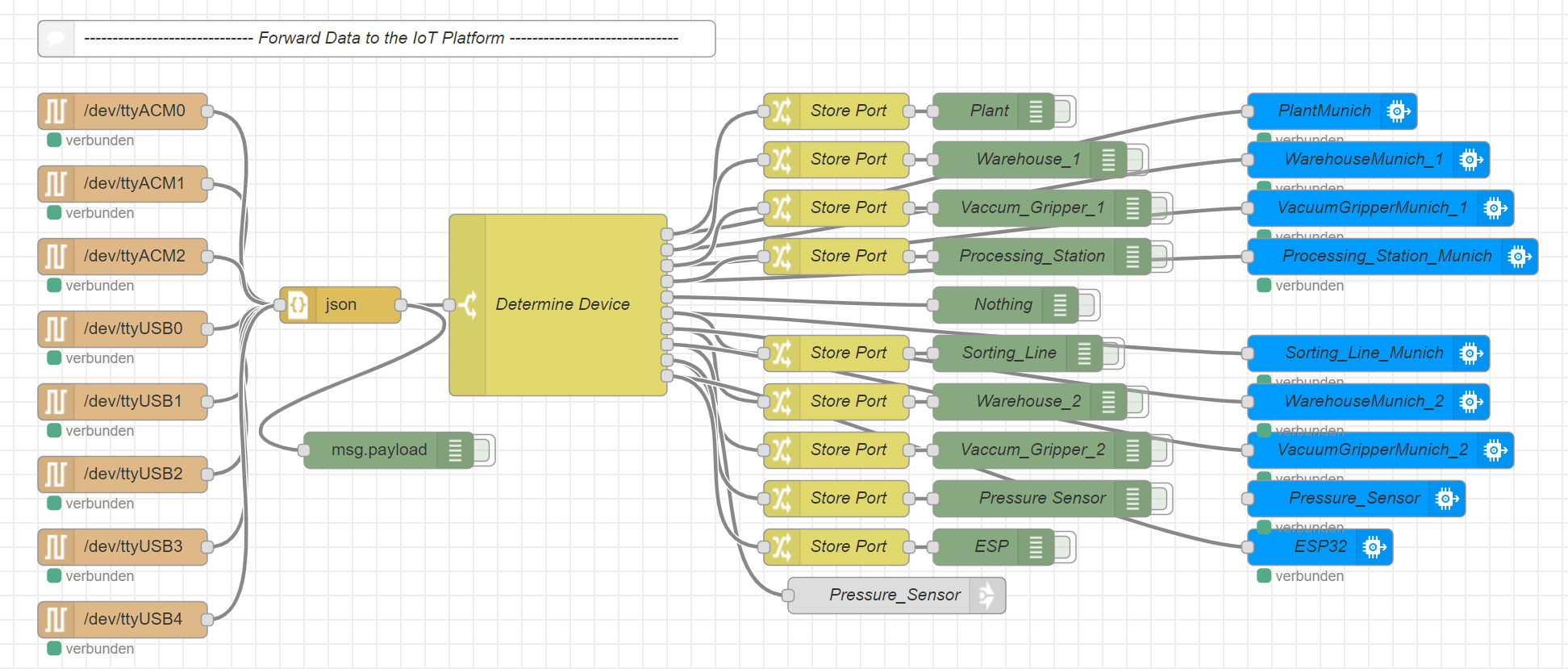


Figure 5: Node RED on Raspberry Pi 3

FYI: the values send by the pressure sensor are currently not forwarded to the IoT platform because that would consume all the available data volume (200 MB/month). Therefore the flow has no connection to the IBM IoT node ‘Pressure\_Sensor’. When you want to enable this feature, you have to make this connection and redeploy the flow. Just pay attention not to exceed the 200 MB.

After explaining the process of forwarding messages from Arduino to the IoT platform, we now look on how to send commands to the controller (commands coming from the IoT platform). Again, the same problem occurs when using the serial nodes in Node-Red. The ports affiliation may change after a restart. So it is not possible to assign a fixed port for every module. To know which module is connected to which port we use the fact, that every module sends some informations when starting (eg. by doing a reference run). So we always receive some data from the modules/controllers before we want to send a command to them. When looking at the flow in Figure 5, we can see, that there are some nodes with the name “store port”. As the name suggests, they store the USB ports name of every module as soon as it sends a message. So if for example the vacuum gripper makes a reference run, it sends some message and then the port is linked to this module for this session. If we want now to send a command to the vacuum gripper, we know which port we have to address. This is shown in Figure 6. First the commands are separated according to their destination (in the nodes “commands for xxx”). This can be done because every command message contains the information of its destination. We then check which port is connected to the corresponding module (this information is stored as a global variable). Then the message is forwarded to the corresponding serial port and so to the desired Arduino.

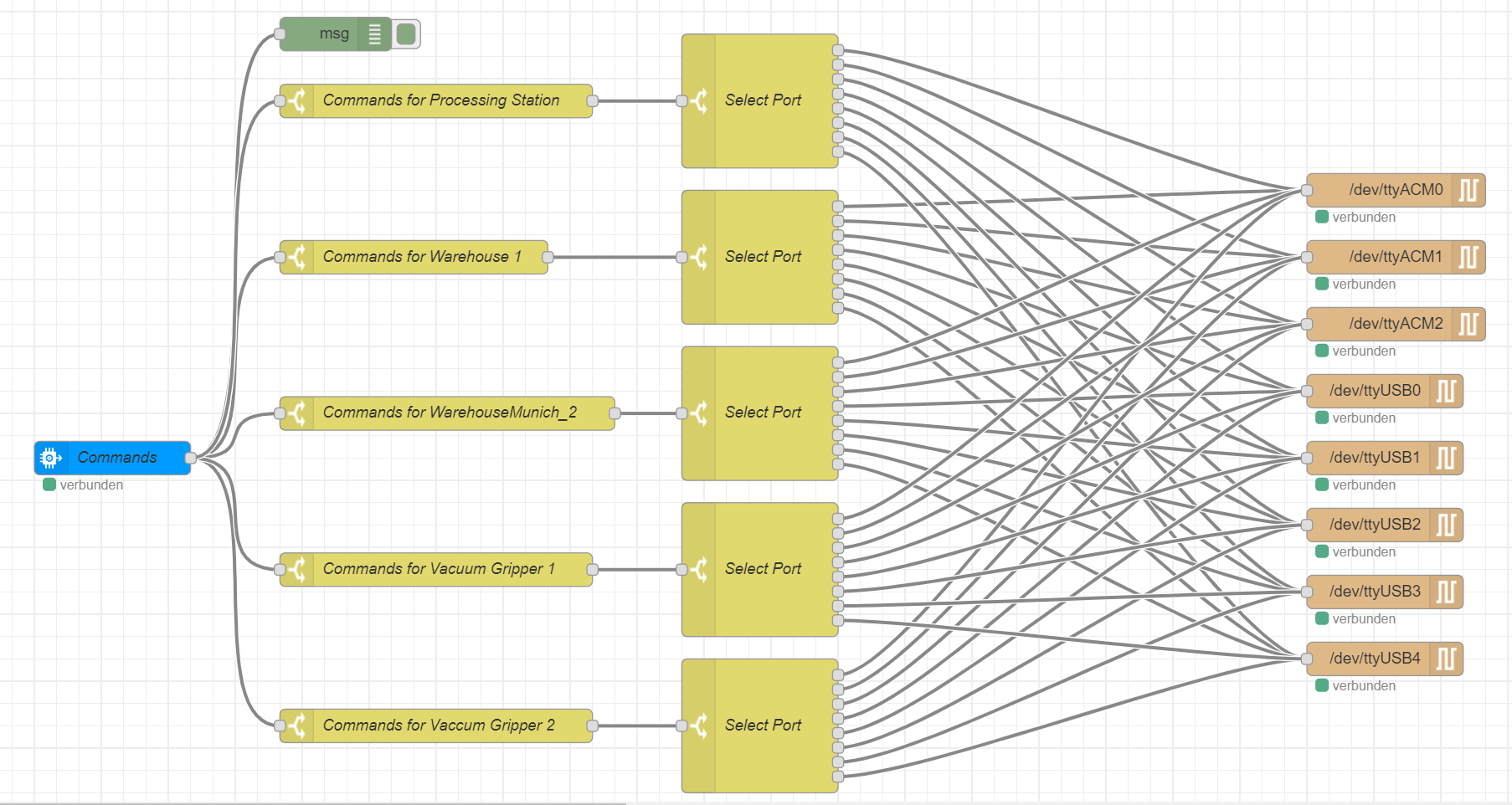


Figure : Node-RED Raspberry Pi 3

1. The IoT Platform

The next step in the journey of the data is the [IBM IoT Platform](https://www.ibm.com/internet-of-things/solutions/iot-platform/watson-iot-platform). As can be seen in the architecture overview in Figure 2, the IoT platform represents the central element in the interaction between the individual instances.

Every physically existing device is represented through a device in the IoT platform (Figure 7 shows an excerpt showing some devices). Every time a message is send to one of the devices in the IoT platform, this one is made accessible to all subscribing instances. Additionally, every message send to the IoT platform is stored in a database (more to this topic in the next chapter). The role of the IoT platform is to receive messages and commands and make them accessible to other participants.

One example for this communication is the data exchange between the Node RED instance hosted in the cloud (responsible for the dashboard and the overall controlling of the plant) and the Node-Red running locally on the Raspberry Pi. As explained in the previous section, the messages are forwarded via the Raspberry pi to the IoT Platform. This means they are now accessible for every subscriber like the cloud Node Red. This one uses again the IoT nodes (again the blue ones) to listen to the messages published by a single (or even all) devices. Once the message is received by the cloud Node Red, this one can react to it and perhaps send a command (again over the IoT platform) or simply display the event on the dashboard.



Figure : Excerpt of the IoT Platform



1. Cloudant

As previously mentioned, all messages/device events send to the IoT Platform are stored in a database. As shown in the architecture overview in Figure 2 we use the [IBM Cloudant Database](https://www.ibm.com/cloud/cloudant). Every day a new database is created, where all events of this one day are stored. This becomes important again when it comes to making a query of the database. Figure 8 shows a screenshot of the Cloudant dashboard. You can see for example the database “18” which stores the events recorded on the 18.10.2019 from the IoT platform with the organization ID ‘6feat5’.

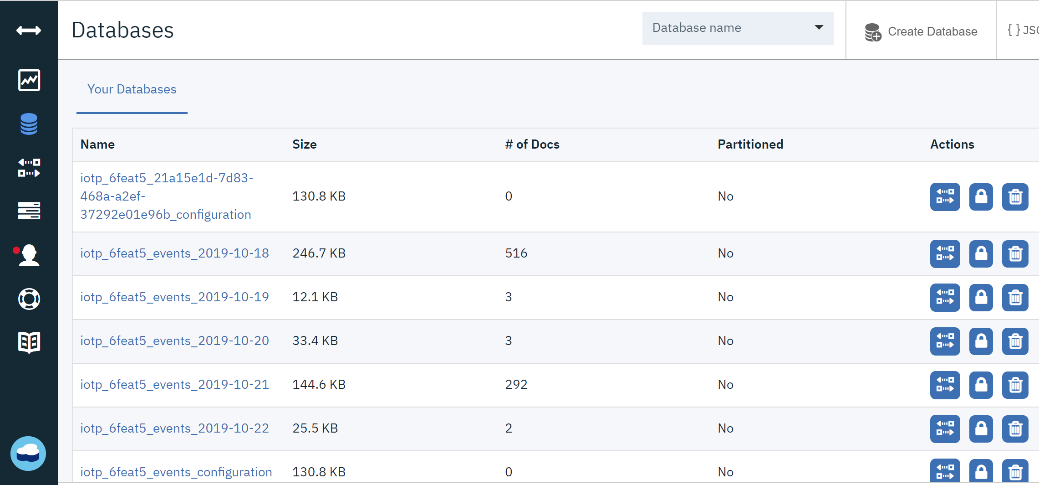


Figure : Excerpt of the Cloudant Dashboard

Every message is stored in the database as a json object containing amongst other things the origin of the message (the device), the time and so on. An example of a stored message is shown in Figure 9.

****

Figure : Cloudant Database Json Object

To connect the IoT Platform to the Cloudant database, you have to establish the connection via a python script (see github)

So that the storing of the data makes sense, we must be able to access it again afterwards. This for example can be done through a query of the entries. To do so we use cloud functions. But we will only discuss this topic in a later chapter.

**!!** Due to missing permissions, I was not able to create and use a Cloudant resource in the IBM cloud of the practice. I therefore used the ‘IoT\_Demonstrator\_Munich@outlook.com’ account.

1. Node Red in the Cloud

The Node-RED instance in the IBM cloud is one of the most important parts when it comes to control the plant and display current informations.

Until now the different modules act completely independent. For example, the warehouse can deliver a workpiece but the vacuum gripper doesn’t know that a workpiece is available and so it doesn’t pick it up. So, the warehouse on its own has no possibility to send a workpiece to the processing station. That’s the reason a ‘supervisor’ controller is needed, which makes among other things this connection between different modules. The Node-Red has somehow the task of a [Manufacturing Execution System](https://en.wikipedia.org/wiki/Manufacturing_execution_system) (MES). It knows about the current state of all modules, keeps track of all workpieces and commands the modules.

Additionally the Node-Red should provide a dashboard (shown in Figure 10) which displays all the current informations about the plant and offers the possibility to send commands to it. This can simply be done over buttons. More of this later on.

The workplace of the Node-RED (which can be accessed over the IBM Cloud) is divided in different tabs/flows. To get a better overview every module has at least one flow (two for the warehouses and the processing station). This division can also be found on the dashboard. Here every module as well as the plant has one individual tab. You can access the dashboard via this [link](https://node-red-fischertechnik-munich-2.eu-gb.mybluemix.net/ui/#!/0?socketid=MdxCCvcUV07RR2aPAAAF).

The description of the cloud Node-Red instance is quite difficult because it takes so many different roles. Therefore I would recommend, that you take a look on your own on the workspace. All steps should be explained in detail using the ‘comment’ node.

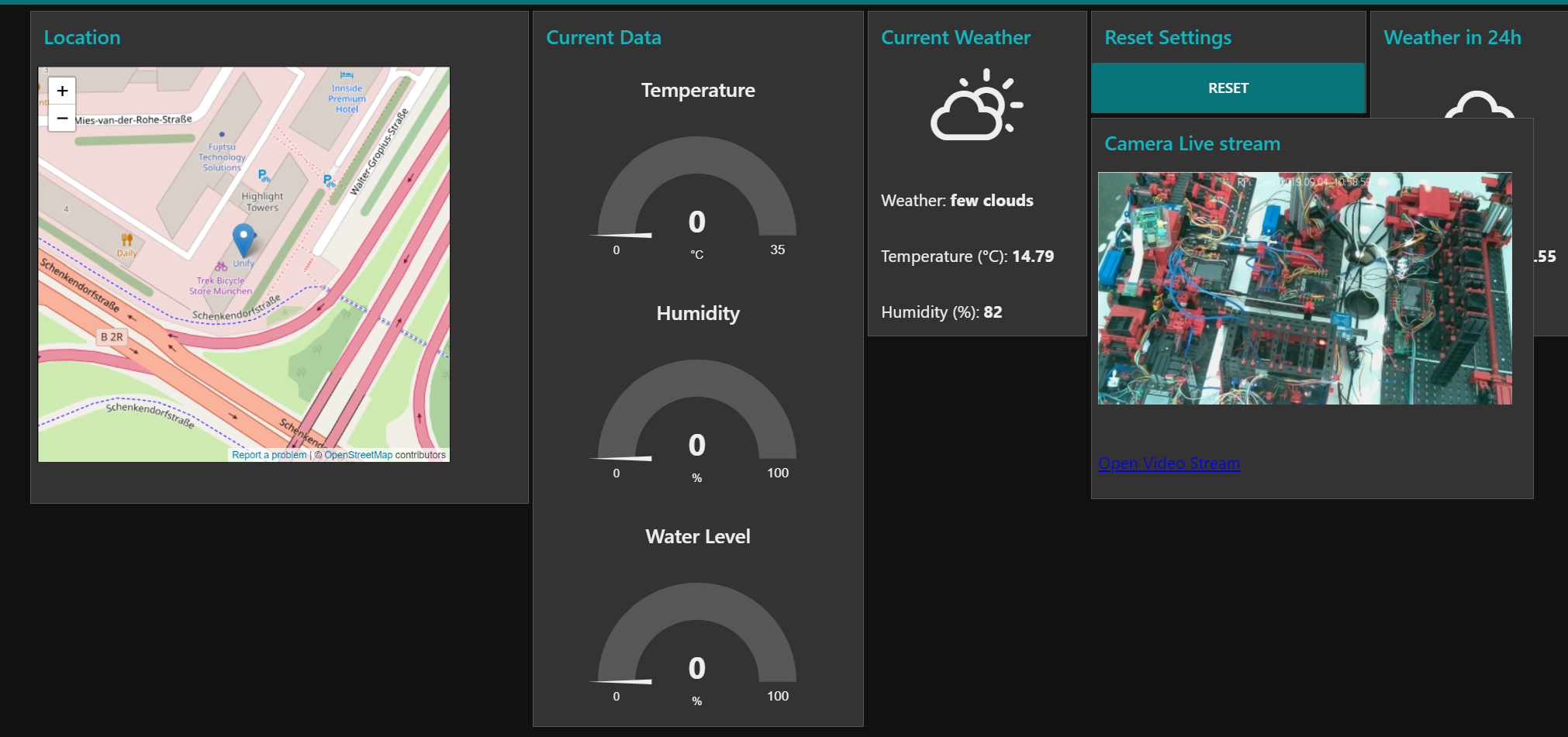
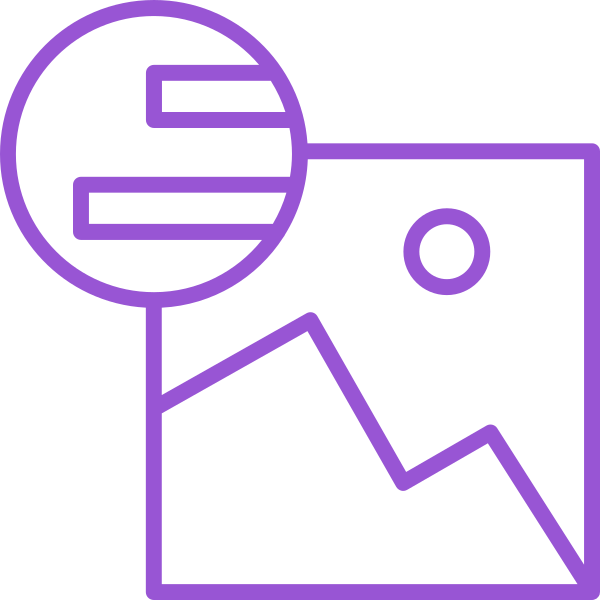


Figure : Node-RED dashboard



1. Visual Inspection

As already briefly mentioned before, we can make a visual inspection of the workpieces in the processing station. So we can check if the surface of the objects is fine or if we can detect any scratch (to simulate a scratch we can use a whiteboard marker as shown in Figure 11 and 12)

We want to make a visual inspection, because we want to represent a painting process in the processing station. In a first step a painting process is simulated in the oven and afterwards a polishing process should take place on the turning table. After these 2 processes a visual inspection should check if the surface of the objects fits the requirements or not.

To do so a camera should take a picture of the workpiece before leaving the processing station. Here we use a second Raspberry Pi with a Raspberry Pi camera which is mounted above the conveyer belt of the processing station. After the polishing process, the workpieces are pushed off the turning table onto the conveyor. This one brings the workpieces to the camera (the arrival of the workpieces can be detected with a light barrier). After reaching the camera, the conveyor belt stops so that the camera can take a picture.

As shown in the architecture overview in Figure 2 the Raspberry Pi sends the image to the Watson Visual Recognition service in the IBM cloud. This service hosts a model, which has been trained with some test images (these can be found in the folder ‘Code\Visual Recognition’). The trained model can classify an object in 7 different categories. Every colour can be classified once with and once without a scratch. The 7th category is an empty conveyor belt, which allows to detect a missing workpiece. The result of the classification is then sent back to the Raspberry Pi.

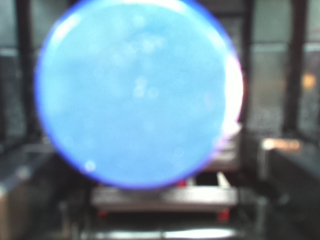
 

Figure :Workpiece without Scratch Figure : Workpiece with Scratch

In the following the process of taking an image, sending it to the Watson Visual Recognition and feeding back the result is explained more in detail.

The process of taking an image an sending it to the Visual Recognition is done with another Node-RED instance, now executed on the Raspberry Pi connected to the camera.

First, we must get the command that the Raspberry Pi should take a picture. Because the Node-RED on the Raspberry Pi doesn’t know when a workpiece is in the position to get photographed, we have to get this information from anywhere else. The cloud Node-Red takes this task. When the processing station reports, that a workpiece is ready below the camera to be photographed and that the processing station is waiting until the visual inspection has finished, the cloud Node-RED sends a command over the IoT platform to the Node-RED running on the Raspberry Pi. This flow is also shown in the architectural overview in Figure 2.

The Node-RED flow responsible for the visual Inspection is shown in Figure 13. The blue IoT node listens to commands send to the Visual Inspection. When receiving the command to start the visual inspection, the flow starts the camera node which then takes a picture and stores it locally on the Raspberry Pi. When this step is finished, the flow continues to a delay node. This one has to purpose to wait 2 seconds until the picture is stored completely in the local storage. When not waiting for some time it can happen, that the picture can’t be opened because the camera node is still in the storing process.

After waiting for the delay-node, we have a picture of the workpiece in the local storage of the Raspberry Pi. For a visual inspection we must send it to the Watson Visual Recognition. Therefore the image is loaded again in the node red flow via the “get file” node. We set the model to the one trained in advance with some sample images of workpieces. Afterwards the image as well as the information which model to choose are forwarded to the “Visual Inspection” node. This one sends the necessary information to the cloud and as soon as an answer is available, the flow continues. We get an answer in Json format. The “get result” node selects only the necessary information (the class with the highest score and the corresponding score). These two informations are then send back to the cloud Node-RED (over the IoT platform).

Another important node which must be explained is the “Get error” node. When an error occurs, this node gets active and reports the IoT platform, that something didn’t work well.

As soon as an answer is feed back to the Cloud Node-Red (via IoT platform), this one sends a command to the processing station, that this one can continue.

Some Nodes in the flow in Figure 13 have not been explained. Most of them are just used to display some informations on the local dashboard. This was done while setting up the flow to check if everything works out the way it should.

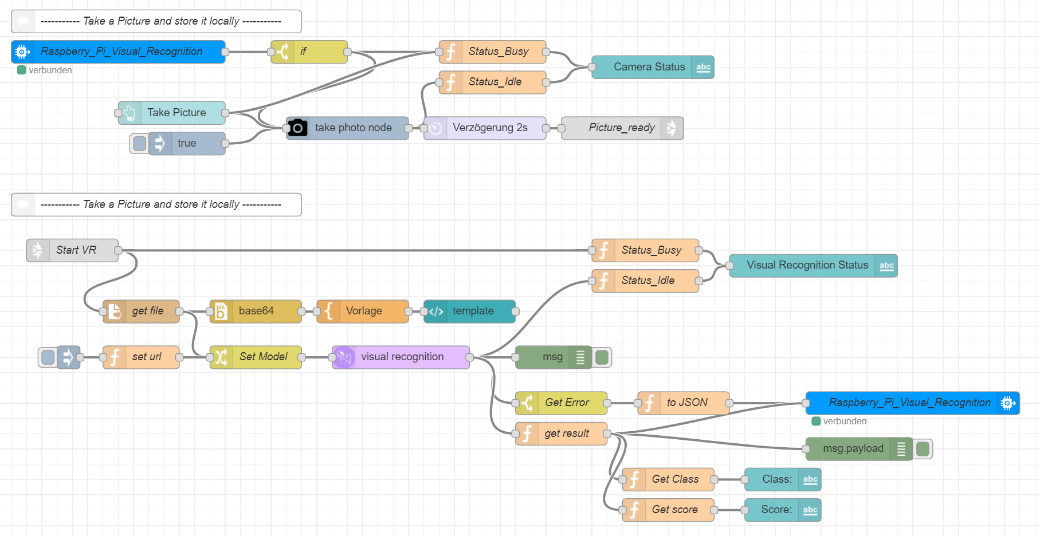


Figure : Node Red on the Raspberry Pi 1

Now just a short explanation how the model used for the classification was trained in Watson Visual Recognition. To train the model you first need some images which are sorted according to the needed categories (e.g. white with scratch, white without scratch, …). Then you can use another IBM cloud resource, the Watson studio to train a classifier. First you must set some categories before adding the associated images. When done, you can start the training process. And that’s is. You now have a trained model which can be used.

The images used for the current model is stored in the folder “Code\Visual Recognition\Images”.

**!!** Due to missing permissions, I was not able to create and use a ‘Visual Recognition’ resource in the IBM cloud of the practice. I therefore used the ‘IoT\_Demonstrator\_Munich@outlook.com’ account.



1. Watson Assistant

We used the [Watson assistant](https://www.ibm.com/cloud/watson-assistant/) to create a chatbot for the Fischertechnik plant. With this one, it should be possible to ask current information about the plant state and to control the plant by sending commands.

First, the organizational stuff. The best way to get started is to check out the Watson Assistant (for example, by completing a [tutorial](https://cloud.ibm.com/docs/services/assistant?topic=assistant-tutorial)) before reading this section. That’s the best way of understanding how the Watson Assistant works and what the individual parts explained below are doing.

To use the Watson Assistant we first need a ‘skill’. This one contains the whole dialog as well as all the parameters needed. It can be imported/exported in JSON format. A current version of the skill for the Fischertechnik plant is stored in [GitHub](https://github.ibm.com/GBSWatsonIoTPracticeMUC/int-connected-factory/tree/master/Source_Code/Watson%20Assistant). You can also find there an instruction on how to import the skill. Once you imported the skill, you can test the chatbot in the Watson Assistant. However you still doesn’t have an integration which enables you to interact with the chatbot via another channel. This can be done in several ways. We used the integration in Slack. So every person which is in the IBM workspace can use the chatbot without any difficulties. Just search for the name “Fischertechnik Chatbot” or scan the QR code below in Figure 13.



Figure : QR Code for Chatbot

Now that we know how to import the chatbot let’s take a close look on the capabilities of the chatbot itself. To start the chatbot can do some basic chitchat. You can ask him questions about his name, creator and so on. Even some jokes are implemented. These functionalities have nothing to do with the use for the Fischertechnik plant, but they are only there for the amusement of the user. When considering the ‘dialog tab’ in the Watson assistant this chitchat takes the first part of the dialog nodes.

Additionally, the user can ask the chatbot questions about the plant itself. The answers for this are written right in the dialog of the chatbot and do not change. The user can ask about the capabilities of the plant as a whole or for every module. For example you can ask the chatbot for a description of the warehouse. Then this would be his answer:

*The plant is occupied with two warehouses. One takes the role as input warehouse which delivers new workpieces to the process flow. Once travelled through the several processing modules the workpieces are then stored in the second and final warehouse.*

*Every warehouse has a capacity of 9 workpieces. These are stored in pallets and can be transported via a crane*

Furthermore, the chatbot can give an instruction on how to start the plant. Al these dialogs can be found in the dialog tab of the Watson assistant skill (an excerpt of this tab is shown in Figure 15).

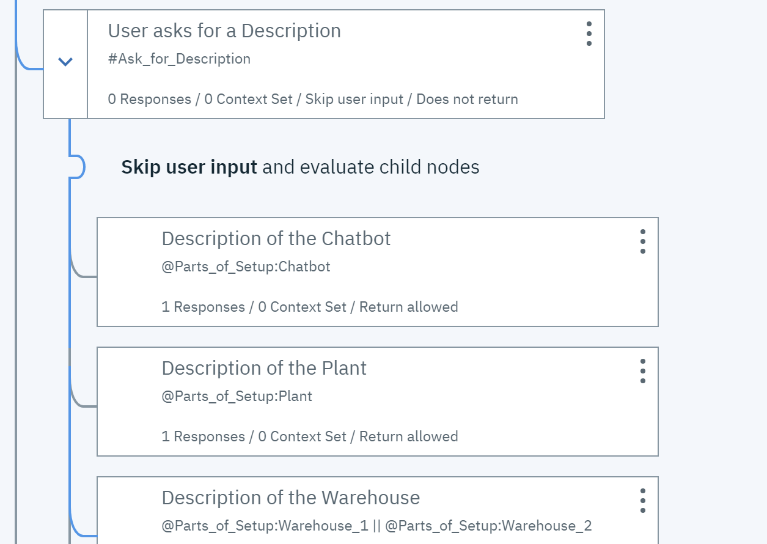


Figure : Excerpt of the Chatbot Dialog 1

Supplementary to these static dialogs, the user can also ask for current informations of the plant (See the architecture diagram in Figure 2). For example, the temperature&humidity sensor records informations in regular intervals, which are then stored in the Cloudant database (check the section about the IoT platform and the Cloudant for more informations). When the user now asks for the current temperature, the last stored temperature value should be given to the user. To do so we use cloud functions. These can be called by the Watson assistant by their URL and by to hand over the necessary credentials (see Figure 16 for a query).

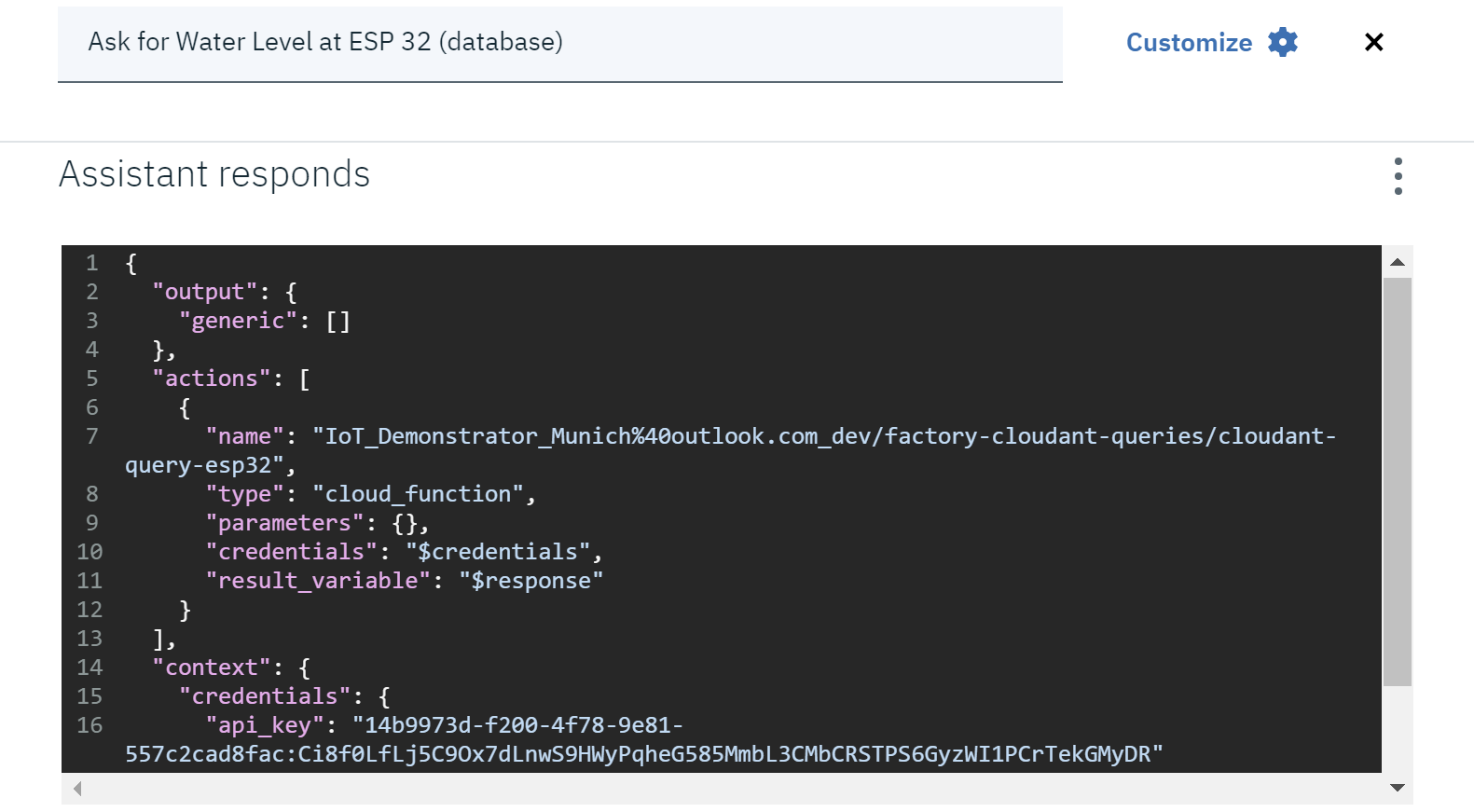


Figure : Excerpt of the Chatbot Dialog 2

When the cloud function is called, this one makes a query in the Cloudant database and hands back the requested value as a response. When no error occurs, the Watson assistant takes this value and integrates it into the answer.

Just a short example to clarify the procedure. When we enter the question “What is the current temperature”, the Watson assistant calls the cloud function “cloudant-query-esp32” (again see Figure 16) to make a query of the temperature sensor information. The function then hands back the value in the response “response.data”. This value is then integrated in the sentence “The temperature inside the factory is currently at $response.data.State.Temperature degrees.”

Sometimes it can happen, that an error occurs. Then the cloud function returns the answer “response.error” which contains the reason why the query couldn’t be realised. Be this an error when connecting to the database or an error while doing the query.

The next section 4.9 is more about the cloud functions and how they are used to query for informations in the Cloudant database.

Next to the possibility to ask the chatbot for informations, we should also be able to send commands. This can be done in a similar way to the request of current sensor/state information. The Watson Assistant calls a Cloud function, which on his side sends a command to the IoT platform (see the architecture diagram in Figure 2). This command is then sent to the Cloud Node-Red and from there if possible, to the plant.

We also explain this process briefly by means of an example. When the user enters the wish: “Can you please deliver a red workpiece”, the activated dialog node in the Watson Assistant calls the cloud function “factory-iotplatform-commands-delivery-red”. When executed, this one sends a command to the IoT Platform, which forwards it to all the subscribing instances. In our case one of these instances is the cloud Node-RED responsible for the plant controlling. Once the command reaches the Node-RED stores it and when possible sends it to the plant for excecution. Here too, errors can occur. If this is the case an error message is responded by the cloud function. When no error could be detected, the Watson assistant sends the sentence “Alright! A red piece is taken to the processing station :factory: As soon as it has arrived you can tell me to paint or polish it.” as answer to the user request.

1. Cloud Functions

The cloud functions can be found in the IBM cloud by searching for “function” in the search bar (see Figure 17). Afterwards switch to the ‘Actions’ tab in the left tab bar. Then you are directed to the page where all created cloud functions are listed.

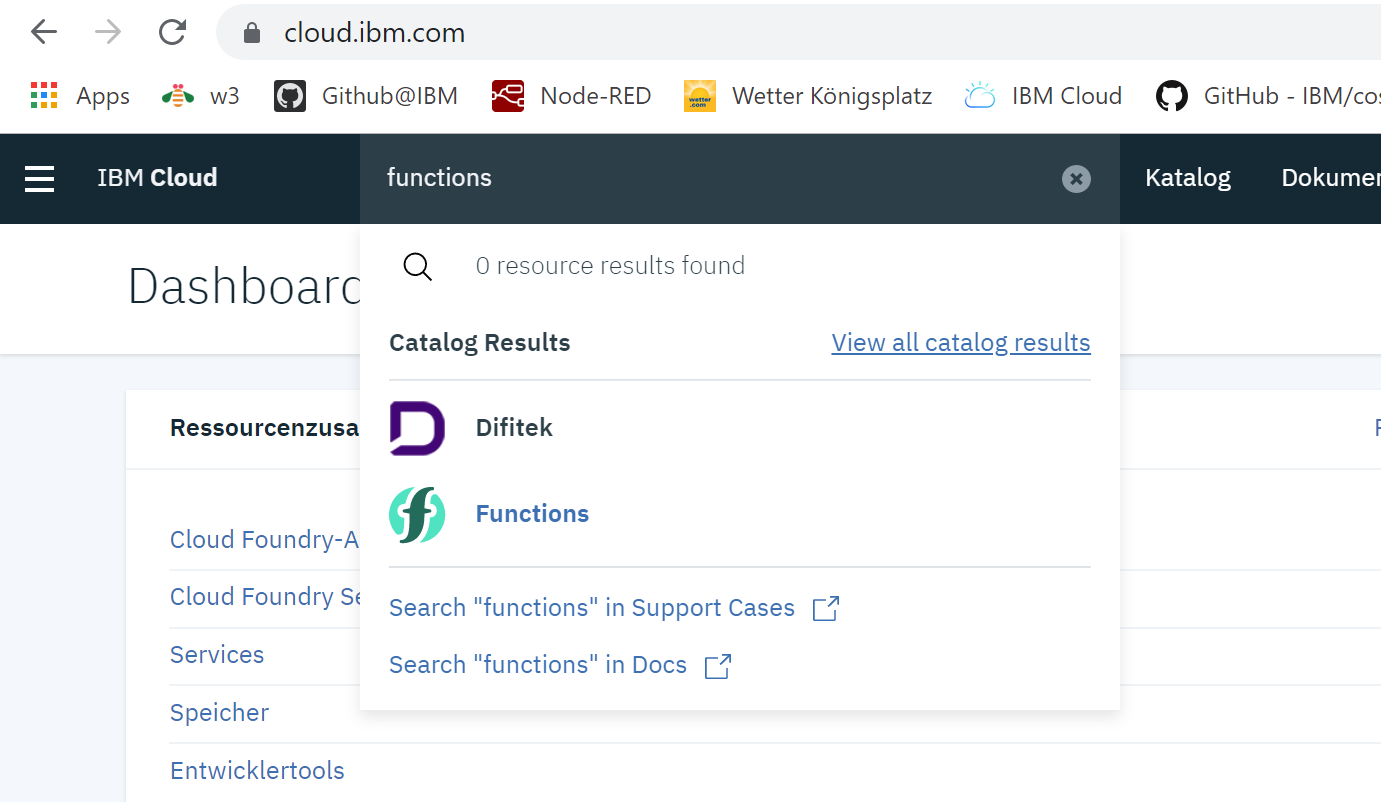


Figure : Search for Cloud functions

Cloud functions are used to query information in the Cloudant database or to send commands over the IoT Platform to the plant. Therefore, two different packages/categories were created. One with the name “factory-cloudant-queries” for queries in the Cloudant database (shown in Figure 19). The other one is called “factory-iotplatform-commands”, which contains the functions to send commands over the IoT platform to the plant (see Figure 20). The category ‘cloud-object-storage’ is not in use.

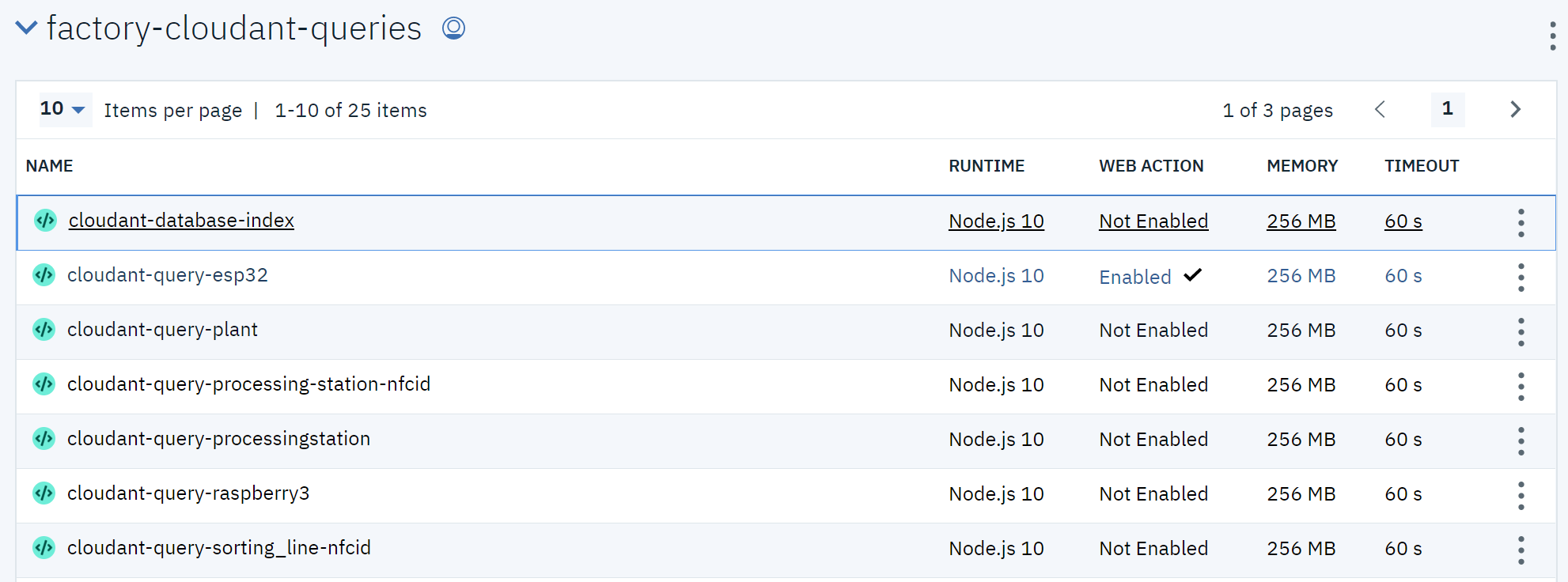


Figure 19: Cloud Functions: Query Database

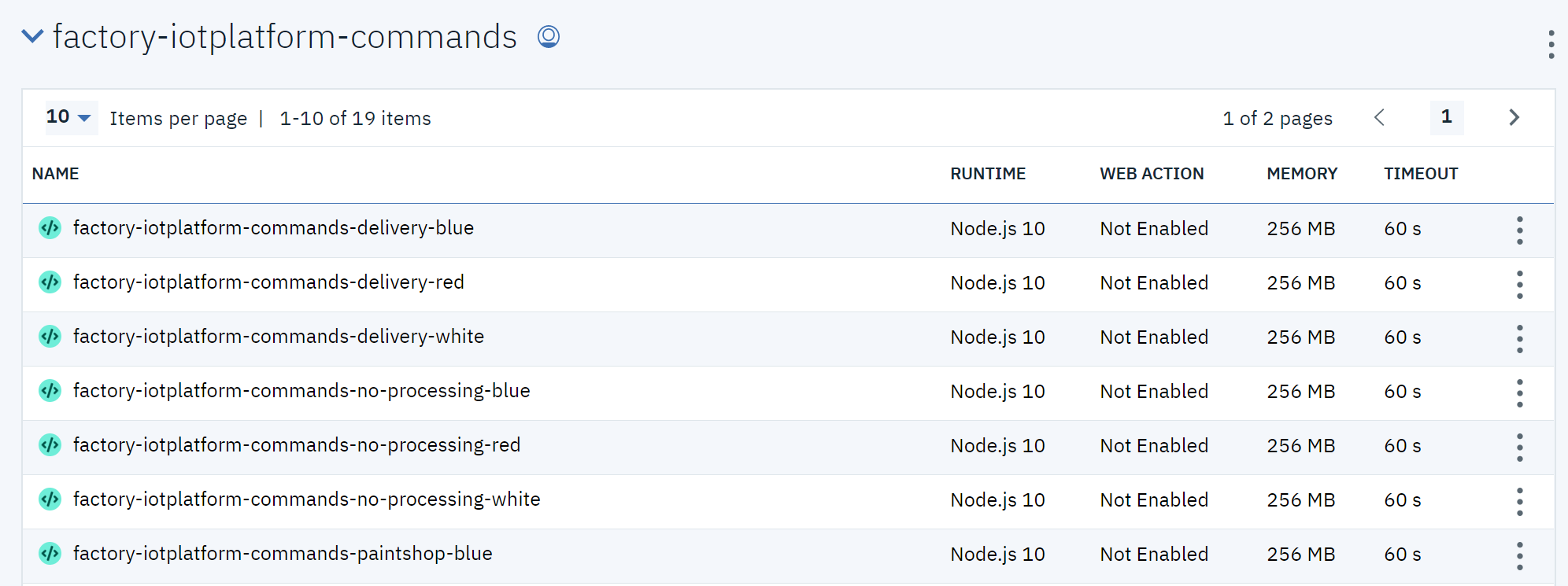


Figure 20: Cloud Functions: Send Commands

Here is the same again. It's best to look at each feature and try to track how it works. The individual functions are commented and contain a short self-explanation. The cloud functions are in most cases called by the Watson Assistant (chatbot).

To be able to query the database and to sort the entries according to their timestamp (in order to find the newest event), we must create an index. When you want more informations about this topic you can ask Arvid Ottenberg. Due to the fact, that every day a new database is created, the step of creating the index must be repeated every day. The database for a new day is always created one day in advance. So, the database for the 22.10.2019 has been created on the 21.10.2019. This allows us to create the index also one day in advance and thus ensure that we can make a query right at 00:00 of every day. The function used to create the index has the name ‘cloudant-database-index’. It is called by a trigger with the name ‘iotplatform\_cloudant\_index’ (see Figure 18). The trigger calls the function every day at 03:00 and sets the index for the next day.

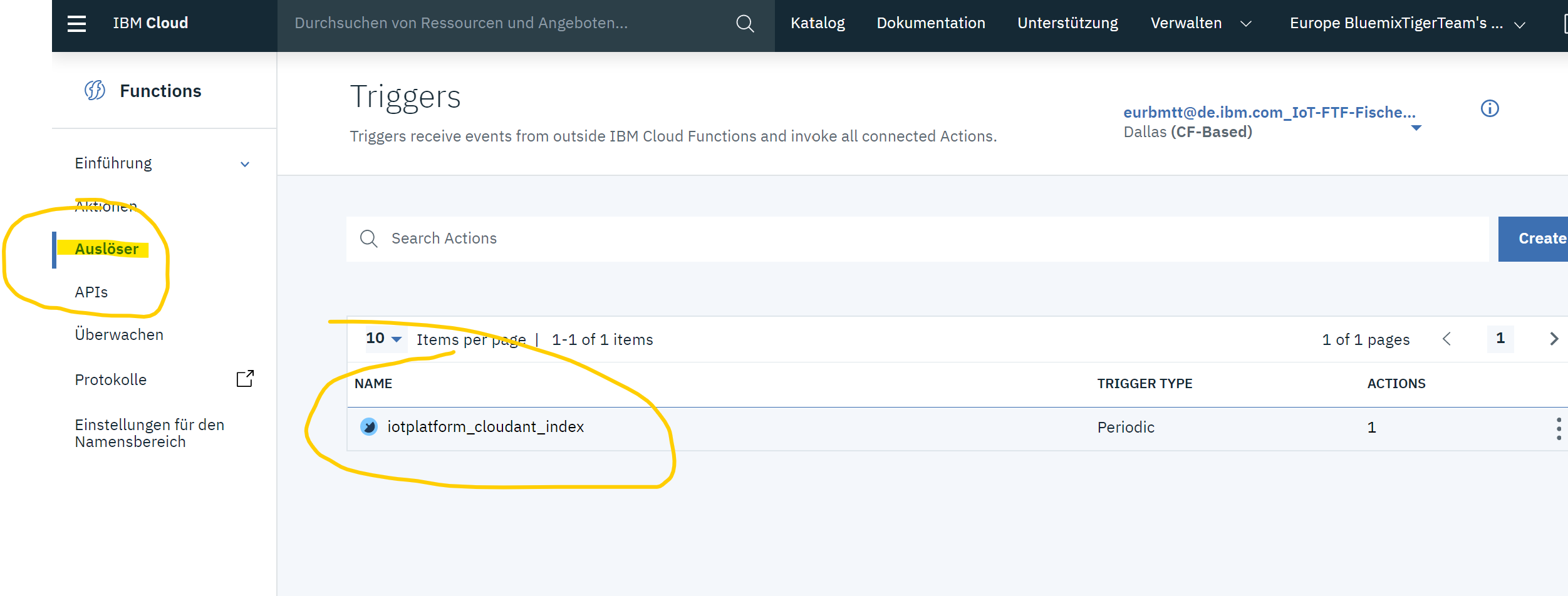


Figure : Cloud Fucntions Trigger

1. Cloud Object Storage

Next to the Cloudant database, we use an additional cloud storage for storing images. These images are then used among other things in the cloud Code Red dashboard. Due to the fact, that it is not possible to store an image in the cloud Node-RED instance you have to call it every time you restart the instance. The [IBM Cloud Object Storage](https://www.ibm.com/cloud/object-storage?cm_mmc=Search_Google-_-Hybrid+Cloud_Cloud+Platform+Digital-_-WW_IUK-_-ibm%20cloud%20object%20storage_e&cm_mmca1=000016GC&cm_mmca2=10007090&cm_mmca7=2826&cm_mmca8=aud-635498185098:kwd-320507222281&cm_mmca9=Cj0KCQjw_5rtBRDxARIsAJfxvYCrNZUqMQ7gxBKj573FnAtA9uXLBbvbFB0TFgGyaqqyWigvU3ilxN4aAlvQEALw_wcB&cm_mmca10=376202124712&cm_mmca11=e&gclid=Cj0KCQjw_5rtBRDxARIsAJfxvYCrNZUqMQ7gxBKj573FnAtA9uXLBbvbFB0TFgGyaqqyWigvU3ilxN4aAlvQEALw_wcB&gclsrc=aw.ds) offers the possibility to call an image via an URL. Figure 21 shows how the image for the warehouse is called after the flow has been deployed.

**!!** Due to missing permissions, I was not able to create and use a ‘Cloud Object Storage’ resource in the IBM cloud of the practice. I therefore used the ‘IoT\_Demonstrator\_Munich@outlook.com’ account.

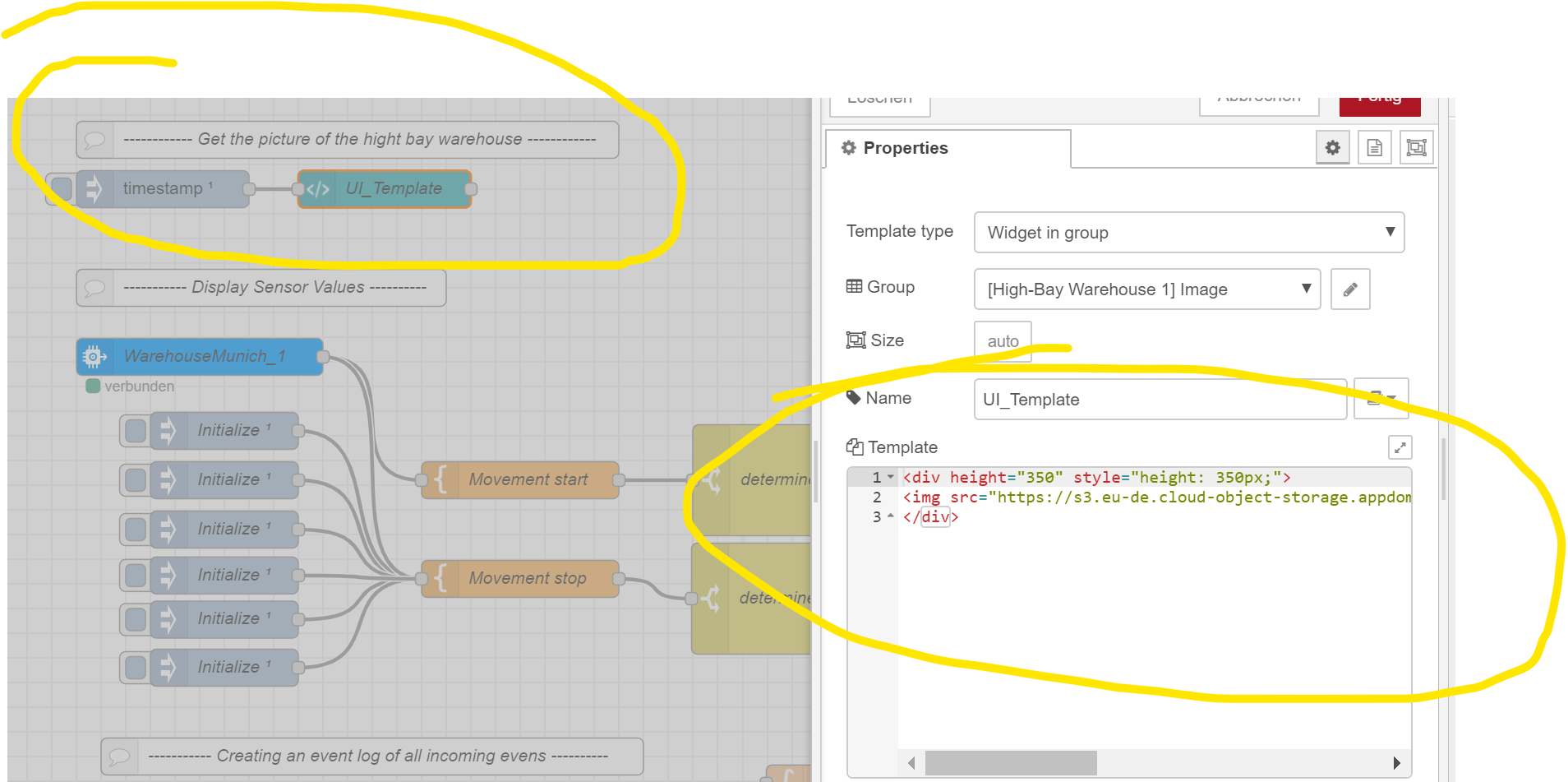


Figure : Node-RED uses Cloud Object Storage