HTTP Extension Tutorial

CyberPi is a versatile and compact microcontroller designed to inspire innovation in coding, electronics, and IoT (Internet of Things) applications. It's perfect for building Internet of Things (IoT) applications, where everyday objects can connect to the internet and work together.

To fully utilize the capabilities of CyberPi, you need to install specific extensions in mBlock 5. These extensions provide additional coding blocks tailored for CyberPi's advanced features.

a. Installing Extensions

This is step by step guide for installing the extension that needs to unlock the Internet of things feature in CyberPi with mBlock5:

1. Open your mBlock5, look for the add extension icon.

2. Search the extension with keyword "HTTP"

To see how the code works, please refer to the source code.

If the download Icon shows in the top-right the icon you may click the icon first to get the latest version of the extension.

b. Work with Internet of Things Blocks

We will learn how CyberPi and MIT App Inventor communicate, so that an IoT System can be created. This method involves the CyberPi being an Access Point for the MIT App to connect to. The CyberPi will act as a handler for the GET and POST request sent to it.

Since the CyberPi acts as a handler, the messages of the HTTP communication will be the MIT App sending requests with the CyberPi sending responses. The responses, more specifically the response for GET requests, will contain the data of the CyberPi. The POST requests will send data from the MIT App to the CyberPi and store it there locally.

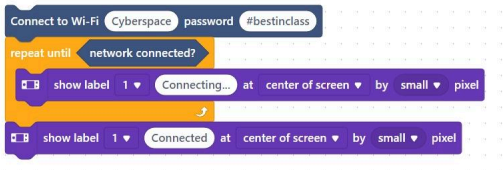
1. Connecting the CyberPi to a network

   To connect to the local WiFi Network, we can use the block in the extension which is,

   This functions similarly to the block in the IoT Extension, however this will allow us to further use the HTTP extension. Then, we need to wait for it to connect to the WiFi, which we can detect by using,

   Which returns the boolean value of *True* when the CyberPi is connected. The whole function can be combined as follows,

To see how the code works,



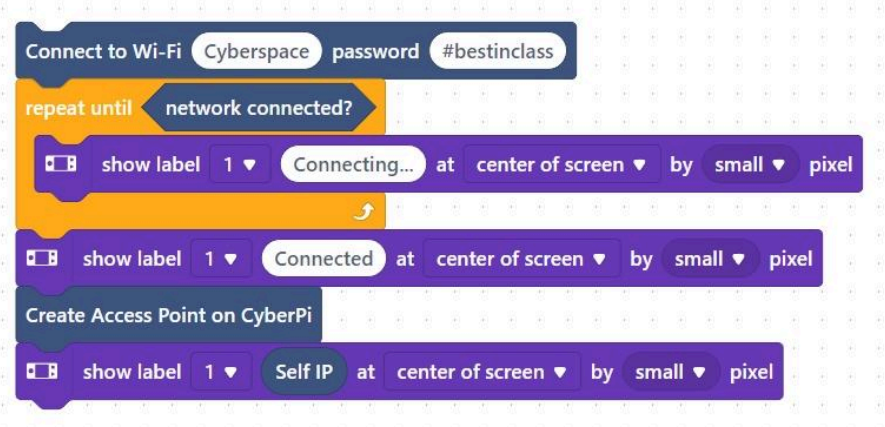
Note: The CyberPi can only connect to 2.4GHz WiFi Networks, and 5G is not detectable. Try to connect to secured WiFi networks instead of public unprotected networks.

2. Creating an Access Point on the CyberPi

   To allow local communication on the local network with the CyberPi, we need to configure it to be an Access Point. The extension has a block to create the AP, however it is dependent on the Network Connection blocks above.

   To connect the MIT App to the CyberPi, we need the local IP of the CyberPi. This can be retrieved by using the *Self IP* block of the extension which will return the value of the IP Address.

   The network setup for the CyberPi can be combined as follows,



To see how the code works, please refer to the source code.

3. Initializing for GET and POST handlers

   Before being able to use the GET and POST handlers, we first need to initialize the values that will be sent and received. There will be two initializers which are,

   - Sending Data

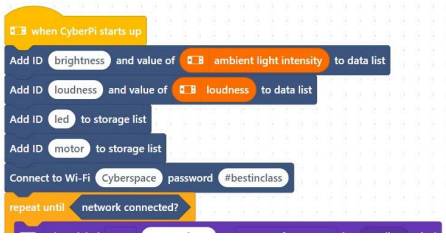
     This block is used to initialize the data for sending data later on in the GET request. The ID will is used as an identifier similar to variable names (The extension also translates it into an address for HTTP). The value can be a function, or it can also be a data type (bool, string, int, etc).

   - Storing Data

     This block is similar, in that it initializes the data to be stored from the POST request.

   These blocks are to be placed before the handlers, preferably at the start of the program.

   Example:



To see how the code works, please refer to the source code.

4. Handlers

   To use the HTTP handlers, we need to first open the HTTP query to be able to receive requests. This can be done by using the *Open HTTP Query* block.

   Then when the handlers are done, we need to close the query by using the *Close HTTP Query* block.

   The handlers that are in the extension are as follows,

   - GET handler for a specific address.

     This block will allow the handling of a request on a specific address. By using this block, multiple data can also be sent in a single response message. This allows better data communication as we do not need to send multiple messages for different data. The address also needs to be written with a backslash, for example we assign it with an address of sensors, then we write it as “/sensors”.

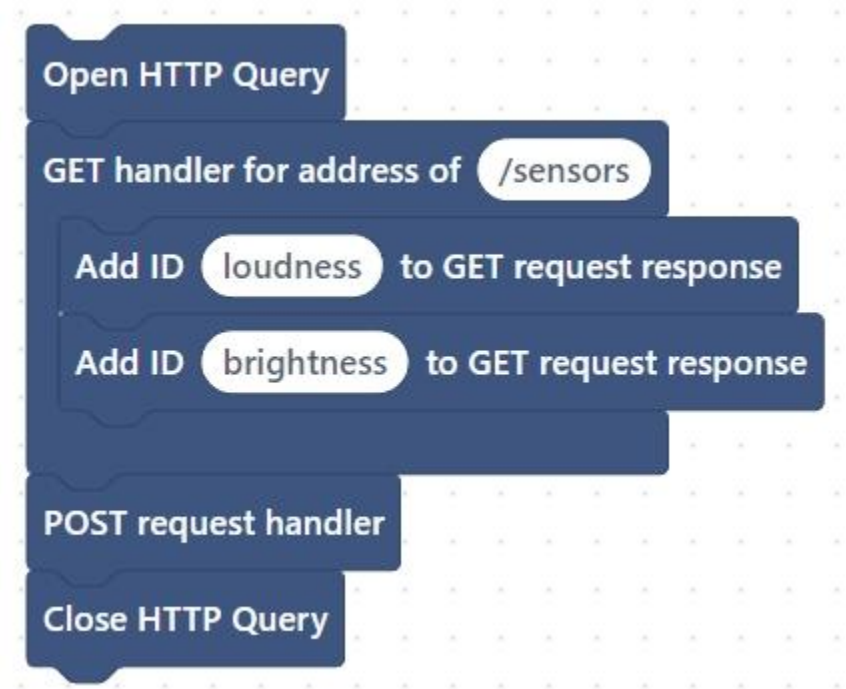
To add a specific data to the response message, we use the block below,

This will add the data with the ID that we initialized in the beginning, and needs to be inside the GET handler block.

- GET handler for all addresses (single data or value).
  There is also a GET handler block which scans each of the initialized data at the beginning. This makes it easier to retrieve data, however it can only do a single data at a time.

- POST handler
  The post handler can be simply added to the code, anywhere between after opening the query and closing it.

The combination of these handlers can should look like this,

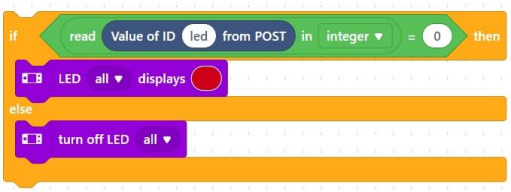


Note: The POST and GET don't need to be used together all the time. Either can be removed if it isn't necessary.

5. Data Processing

   For getting the data from the stored POST request, we can use the *Value of ID [] from POST,* which returns the variable stored.

   For example, if we want to control an LED through the POST messages, we can do so like this,



To see how the code works, please refer to the source code.

Example program of controlling LED and Motor, retrieve loudness and brightness.

To see how the code works, please refer to the source code.

HTTP Extension Documentation

Contributors : Geraldi Kolin
Last Updated : 10/03/2025

This is the documentation for the mBlock HTTP custom extension for PT. Joindo Eka Handal. Here, the extension code will be explained in detail. Before reading this documentation, please understand these key concepts,

- Python Basics
- HTTP Communication

Note that the source code and the Extension code differ for block programming interpolation.

1. Overview

   The HTTP Extension is a custom library for the mBlock Compiler. It is based on the source code mentioned in the previous documentation, but translated into a block based code for easier understanding and usage. Some basic info of the extension are,

   - ID : jehcorpiot
   - Name : CyberPi HTTP Handler
   - Support : CyberPi

   As mentioned, the blocks are an interpretation of the source code, which functions for connecting to the local WiFi network, creating a local Access Point, and HTTP GET POST handlers. See source code on how this functions!

   To edit the extension, the *.mext* file can be imported to mBlock's plugin compiler website. To use the extension, the *.mext* file can be inputted directly to the mBlock software. Additionally, the published version is able to be downloaded from the *Extension* page of the mBlock software.

2. Blocks

The blocks used in the extension are converted to a more tile-able code structure, which is why some bits of code are changed to suit block based better.

2.1. Connect to WiFi

To connect to the WiFi, this block is runned at the start of the code. It is recommended to use this block, instead of other blocks due to other blocks' dependencies on the variable of this block. The inputs are two strings, the SSID and Password of the WiFi network. It also imports the <network> library into the code.


```
global wlan
wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect({ssid}, {password})
```

2.2. Create AP

To create an Access Point to allow the HTTP communication, the following block is used,


```
try:
    addr = socket.getaddrinfo(str(wlan.ifconfig()[0]), 80)[0][-1]
    s = socket.socket()
    s.bind(addr)
    s.listen(5)
    cyberpi.display.show_label(str(wlan.ifconfig()[0]), 12, "center", index= 0)

except:
    pass
```

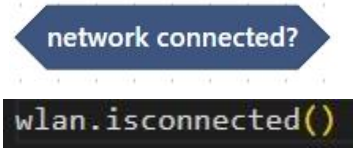
To see how the code works, please refer to the source code.

The block has dependencies on all the blocks of the extension, so it is mandatory to use this block. Moreover, the block is responsible on creating the common code for the object class of the extension as well as importing <network> and <socket> library,

```
class dataObject:
    def __init__(self, address, function):
        self.address = "/" + address
        self.object = function

    def getEncodedData(self):
        data = self.object()
        return data

class storeObject:
    def __init__(self, id):
        self.address = "/" + id
        self.storedValue = 0
        self.id = id #dictionary key when decoding/converting

    def storeValue(self, val):
        self.storedValue = val

    def getValue(self):
        return self.storedValue

        #alterntively can store and get
        #directly with "name_of_object".storedValue

data_list = []
store_data = []
```

2.3. Network Connected?

To check the status of the connection to the network, this block will return a boolean data,



To see how the code works, please refer to the source code.

2.4. Open Query

To receive the messages sent to the CyberPi, the socket needs to be opened. This is done by using the *Open HTTP Query* block.


```
cl, addr = s.accept()
web_query = cl.recv(1024).decode("utf-8")
```

2.5. Close Query

To close the incoming message socket, the *Close HTTP Query* block can be used.


```
cl.close()
```

2.6. Add ID to GET list

To initialize the class objects that are used for the GET handler, this block needs to be placed at the area of the starting code. It needs 1 input string for the address or ID, and 1 input function or value which is stored in the class.


```
data_list.append(dataObject({data_id}, lambda: {data_value}))
```

2.7. Add ID to POST list

To initialize the class objects that are used for the POST handler, this block needs to be placed at the area of the starting code. It needs 1 input string for the address or ID.


```
store_data.append(storeObject({data_id}))
```

To see how the code works, please refer to the source code.

2.8. GET handler for specific address

The GET handler will process the outgoing response message by adding the needed variables or data into the message. This block must be placed after the HTTP Query is opened. It needs a string input which is the target address (add "/" before the actual string e.g. /address). Since this block is conditional, subsequent blocks placed in it are put in between the *data_array* variable.


```
if ({data_address} in web_query):
    data_array = "\u007B"
    {$BRANCH1}
    data_array += '"' + "NULL" '":"' + "ARRAY_CLOSE" + '"' + "\u007D"
    data_response = (
    "HTTP/1.1 200 OK\r\n" +
    "Content-Type: application/json\r\n" +
    "Access-Control-Allow-Origin: *\r\n" +
    "Vary: Origin\r\n" +
    "Access-Control-Allow-Methods: GET, POST, OPTIONS\r\n"
    "Access-Control-Allow-Headers: Content-Type\r\n"
    "Server: CyberPi " + '''IP Address here''' "0"  + "\r\n" +
    "Connection: close\r\n\r\n" +
    str(data_array)
    )
    try:
        cl.send(data_response.encode("utf-8"))
    except:
        cyberpi.console.println("Sending Data Failed")
```

2.9. Add ID to GET response

This block is used with the GET handler above, by adding needed data into the data array. It needs 1 string input, which is the ID of the initialized variable in the GET list.


```
for i in data_list:
    if str(i.address) == '/' + str({data_id}):
        data_array += '"' + str(i.address) + '":"' + str(i.getEncodedData()) + '",'
```

To see how the code works, please refer to the source code.

2.10. GET handler for all address

This is an additional GET handler for a shorter code. It functions by scanning all the objects in the GET list, then matches the ID with the request address. If it matches, it will then send a response with the object's value.


```
for i in data_list:
    if (i.address in web_query):
        key = i.address
        value = i.getEncodedData()

        data_array = "\u007B"
        data_array += '"' + str(key) + '":"' + str(value) + '",'
        data_array += '"' + "NULL" '":"' + "ARRAY_CLOSE" + '"' + "\u007D"


        data_response = (
        "HTTP/1.1 200 OK\r\n" +
        "Content-Type: application/json\r\n" +
        "Access-Control-Allow-Origin: *\r\n" +
        "Vary: Origin\r\n" +
        "Access-Control-Allow-Methods: GET, POST, OPTIONS\r\n"
        "Access-Control-Allow-Headers: Content-Type\r\n"
        "Server: CyberPi " + '''IP Address here''' "0"  + "\r\n" +
        "Connection: close\r\n\r\n" +
        str(data_array)
        )
        try:
            cl.send(data_response.encode("utf-8"))
        except:
            cyberpi.console.println("Sending Data Failed")
```

To see how the code works, please refer to the source code.

2.11. POST handler

The POST functions to store the values that are received by the Cyberpi. It functions similarly to the previous handler, as it automatically inputs the data by matching the ID. It also defines a custom function that is used to process the HTTP message.


```
for x in store_data:
    if ("POST" in web_query) and (x.address in web_query):
        body = separate_post_data(web_query)
        convert_to_dictionnary = json.loads(body)
        endVal = convert_to_dictionnary[x.id]
        x.storeValue(endVal)

        data_response = (
        "HTTP/1.1 200 OK\r\n" +
        "Content-Type: application/json\r\n" +
        "Access-Control-Allow-Origin: *\r\n" +
        "Vary: Origin\r\n" +
        "Access-Control-Allow-Methods: GET, POST, OPTIONS\r\n"
        "Access-Control-Allow-Headers: Content-Type\r\n"
        "Server: CyberPi " + '''IP Address here''' "0"    + "\r\n" +
        "Connection: close\r\n\r\n"
        )
        cl.send(data_response.encode("utf-8"))
```

```
def separate_post_data(post_request_text):

    lines = post_request_text.splitlines()
    body_start_index = -1

    for i, line in enumerate(lines):
        if not line.strip():
            body_start_index = i + 1
            break

    if body_start_index != -1:
        headers = lines[:body_start_index - 1]
        body = "\n".join(lines[body_start_index:])
        return body
    else:
        return "Decode failed!"
```

To see how the code works, please refer to the source code.

2.12. Value of ID

To use the or retrieve the data that is stored by POST, this block can be used that returns the value of the object by matching the ID, it needs a string input which is the ID. It defines the function at the start.


```
valuePostData({data_id})
```

```
def valuePostData(dataId):
    for i in store_data:
        if i.id == dataId:
            return i.getValue()
    return "No ID"
```

2.13. Self IP

The *Self IP* block is an additional block used to retrieve the value of the local IP in the Cyberpi. Usually used for debugging or connecting the AP.


```
wlan.ifconfig()[0]
```

3. Bugs, Limitations, and Optimization
   - 3.1. Change the ID system to allow dropdown selection
   - 3.2. Determine if the for loop (additional) is necessary
   - 3.3.

To see how the code works, please refer to the source code.